Learning Python by Building Games

A beginner's guide to Python programming and game development

Packt>

www.packt.com

Sachin Kafle

Learning Python by Building Games

A beginner's guide to Python programming and game development

Sachin Kafle

BIRMINGHAM - MUMBAI

Learning Python by Building Games

Commissioning Editor: Amarabha Banerjee
Acquisition Editor: Kajal Bhagure
Content Development Editor: Aamir Ahmed
Senior Editor: Hayden Edwards
Technical Editor: Jinesh Topiwala
Copy Editor: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Alishon Mendonsa

First published: October 2019

Production reference: 1111019

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78980-298-6

www.packt.com

Dedicated to mom and dad, for all your love and support.
And, to Sonu and Susaan, for the wonderful memories of growing up.

– Sachin Kafle

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Foreword

I have known and worked with Sachin Kafle for more than four years. Sachin is one of the most well-known individuals among Nepal-based cyber and Python experts. In this book, *Learning Python by Building Games*, Sachin takes you on a learning journey of core and advanced Python programming paradigms with the help of hands-on examples. For more than 15 years, Python has continued to evolve to meet the needs of developers around the world. For the majority of this time, Sachin has been a key team member in initiating projects by creating and reusing modular programs.

In his presentations and examples, Sachin shows you how easy it is to create a wide range of applications/games using different Python libraries, such as Pygame, Pymunk, and PyOpenGL. Sachin has also helped developers to create a game with a taste for AI.

With *Learning Python by Building Games*, you'll learn the best practices for writing high-quality, reliable, and maintainable code with Python, a general-purpose language. After you have completed Sachin's book, you'll understand how to create and deploy your own mobile/computer games and apps.

Beyond developing apps for desktops and smartphones, you'll learn how to use the Python programming paradigm to accomplish architecture based on AI and simulation.

In *Learning Python by Building Games*, Sachin encapsulates the knowledge gained through years as an academic specialist and Python developer, a Python cybersecurity analyst, and a passionate advocate. Through his words, step-by-step instructions, screenshots, source code snippets, examples, and links to additional sources of information, you will learn how to continuously enhance your skills and apps.

Become a proficient Python developer and build stunning cross-platform apps with Python.

Prof. Dr. Subarna Shakya

Chairman, Computer Engineering Subject committee, Ministry of Education, National Curriculum Development Center (Nepal)

Contributors

About the author

Sachin Kafle is a computer engineer from Tribhuvan University, Nepal, and a programming instructor currently living in Kathmandu. He is the founder of Bitfourstack Technologies, a software company that provides services including automation for real-time problems in businesses. One of his courses, named *Python Game Development*, is the best seller on many e-learning websites. His interests lie in software development and integration practices in the areas of computation and quantitative fields of trade. He has been utilizing his expertise in Python, C, Java, and C# by teaching since 2012. He has been a source of motivation to younger people, and even his peers, regardless of their educational background, who are embarking on their journey in programming.

I would like to acknowledge the amazing staff and editorial team at Packt Publishing: without their talent and dedication, this book would not be such a valuable asset. In particular, I would like to thank Aamir Ahmed and Mohammed Yusuf Imaratwale for having faith in this book from the beginning. Adapting Aamir's many insightful comments and suggestions really uplifted the quality of this book, and I am grateful for all the time and effort he put into this book.

I'd also like to thank the technical reviewer, Jose Angel Munoz, and the technical editor, Jinesh Topiwala, for their thorough attention to the programming aspect of this book. Their detailed labels and understanding of target audiences, along with their invaluable comments, greatly improved the clarity of this book.

Finally, a special thanks to all of my students for their support and zeal for having this book published. Your voracity toward learning game development using Python is what inspired me to write this book.

About the reviewer

Jose Angel Munoz is a system engineer and architect with multiple years of IT infrastructure and infrastructure-as-code development experience. He is an expert in a variety of technologies, has collaborated with different open source projects, including Ansible, Microsoft, Inspec by Chef, Pimoroni, and XLDeploy, and has published different articles in Linux specialised magazines. For Packt, he has reviewed two PowerShell-related books. You can find him on GitHub (`@imjoseangel`).

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

In September of 2018, I was teaching some of my students about game programming and automation using Python. Then, I realized that it was time to create a book that not only offers information on the rich content of game programming using Python but also shows how to make and deploy games that mimic real, world-famous games such as Flappy Bird and Angry Birds. I wanted to equip you with all the essentials and primitives of game programming to become a real-world Python game developer. This book is not your usual and traditional Python theoretical book; our approach will be as practical as possible. Each chapter will contain a single, yet powerful, real-world game example that will not only be interesting but will also edify you with programming paradigms, which will be your first step to becoming a proficient Python developer.

Python is one of the most widely used programming languages of 2018/19, according to a survey conducted by Stack Overflow and TIOBE, and its rate of popularity growth is not expected to decrease any time soon. If you observe what big tech companies use for handling their businesses, you can see that they depend highly upon Python because of its easy usage and rapid prototyping. Not only that, but you can also see that Python can be used to develop a variety of applications ranging from data science to high-end web applications, and as you proceed to learn the basics of Python, you will be ready to create almost anything you want.

There are many reasons to learn Python, and a big one is the Python community. Many of the world's greatest developers contribute incessantly to this Python community by adding new libraries/modules and functionalities. These libraries prove to be extremely helpful if you want to create something new and rapidly. As such, Python is focused on products rather than being bogged down in the routines and complexities of low-level programming, which makes it the most loved programming language of beginners.

In this book, we will start by introducing some important programming concepts, such as variables, numbers, Boolean logic, conditionals, and looping. After building a solid foundation of core programming concepts, we will hop into advanced sections such as data structures and functions. The pace of learning will be increased with the difficulty of the chapters. After finishing `Chapter 7`, *List Comprehension and Properties*, we will be fully equipped with all the basics to be applied while creating advanced things such as flappy bird emulators, angry bird emulators, and AI players. In each chapter, there will be a *game testing and possible modification* topic to compel you to think about how errors should be handled and how programs should be refined.

Requirements for this book

To get a good grasp of each of the topics written about in this book, I encourage you to follow along with the source code and examples. To write code properly, you will need to install Python on your machine. I have used Python's latest version (as of September 2019), version 3.7, but you can use any version newer than 3.5+. The thorough installation process of Python is covered in the first chapter for your machine, based on the OS (Linux, macOS, or Windows) you're using. You will also need an internet connection up and running to download GitHub code and Python third-party libraries. We will be installing different Python libraries, including PyGame, Pymunk, and PyOpenGL later in this book. For each of them, the installation process will be covered in the chapter concerned. While using such modules, our programs will tend to become lengthier, so we strongly encourage you to use a good Python text editor. I will be using the PyCharm IDE to create complex games using Python, and its installation is also covered in the first chapter. Apart from these software requirements, there are no specific requirements for this book.

Who this book is for

This book is for anyone who wants to learn Python. You can be a beginner or someone who has tried learning it previously, but a boring course or book set you off track, or someone who wants to brush up on their skills. This book will help you gain core knowledge and advance your skills in the most interesting way: by building games. It primarily focuses on GUI programming using the Python modules PyGame, PyOpenGL, and Pymunk. No programming skills are expected from learners as we will cover everything you need to know about Python in this book. We will study the `turtle` module by building three mini-games, and you will learn how to create your very own 2D games, even if you are a complete beginner. If you ever wanted to explore game development with Python's PyGame module, this book is for you.

What this book covers

`Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, covers the background of game development and the scope of Python in game development. We will set up Python on our local machine and install the appropriate editor. We will also become familiar with the project settings and the interface of the editor. We will see how to install modules in PyCharm. We will execute our first Python program in this chapter.

`Chapter 2`, *Learning the Fundamentals of Python*, takes us through the invigorating stuff of the Python ecosystem, giving us knowledge about the basic concepts of programming such as variables, numbers, and modules. This chapter will give us with knowledge of values, types, and type-casting techniques. We will make a simple tic-tac-toe game using concepts learned in this chapter. This will teach us how to track data in Python programs.

`Chapter 3`, *Flow Control – Building a Decision Maker for Your Game*, covers the concepts of Boolean logic, conditionals, and looping. This chapter will be life-changing for any learning developer. This chapter will provide mainly deal with how things can be automated with logic. We will also see looping patterns and debugging. Some practical examples will be covered in this section. We will refine our tic-tac-toe game by incorporating game logic and flow controls.

`Chapter 4`, *Data Structures and Functions*, covers lists, dictionaries, sets, and tuples. This chapter will help programmers to distinguish between, and choose among, different built-in storage solutions based on different situations. We will learn how to create each of these data structures and how to perform different operations, including adding, deleting, and traversing. We will make use of advanced data structures such as trees and queues in our tic-tac-toe game, which will make our game more rugged.

`Chapter 5`, *Learning About Curses by Building a Snake Game*, covers terminal-independent screen-painting and keyboard-handling facilities for text-based terminals; such terminals include VT100s, the Linux console, and the simulated terminals provided by various programs. We will make a snake game using curses events and screen painting. We will make simple snake game logic using curses properties.

`Chapter 6`, *Object-Oriented Programming*, deals with creating and using objects in your project. We will learn how to wrap data using properties and restrict data access using specifiers. We will also learn how to use the built-in methods of Python to execute overloading. This chapter will mainly deal with the terminologies of **object-oriented programming** (**OOP**), such as classes, encapsulation, inheritance, and polymorphism. We will use the OOP paradigm to make our snake game made with curses more robust and reusable.

`Chapter 7`, *List Comprehension and Properties,* targets making our code simpler and faster in execution. This chapter will teach us how to work with conditions and logic to implement more understandable single-line code. We will see list comprehension and properties in action with our snake game.

`Chapter 8`, *Turtle Class – Drawing on the Screen*, deals with the `turtle` module of Python. This chapter will give a detailed explanation of how to use Python's turtle to draw all over the screen with simple forward/backward commands. We will learn how to make basic objects with turtle and build some skeleton code with Python in this chapter.

`Chapter 9`, *Data Model Implementation*, covers base class implementation. The base class makes use of operator overloading using special built-in Python methods. We will make use of vectors to specify the positions of objects and we will manipulate them with some algebraic operations. Special functions such as `__add__()`, `__mul__()`, `__str__()`, and `__repr__()` will be used to overload operators.

`Chapter 10`, *Upgrading the Snake Game with Turtle*, shows us how to create our first 2D game with a Python script. We will make use of the `turtle` module to create animations on the screen. This will be a simple game, but we will learn how to use the methods of the `turtle` module to move a pen and draw all over our canvas. We will modify our snake game, made following simple OOP concepts, to one that contains simple animations made with turtle. In addition to the snake game, we will also see how to make games such as Pong and Flappy Bird with turtle.

`Chapter 11`, *Outdoing Turtle – Snake Game UI with PyGame*, covers the installation of PyGame on your machine, and we will also cover how to make the basic skeleton code of our game containing display initialization, game loops, states, events, and colors. We will modify our snake game, made with the `turtle` module, by using a sprite and a game controller library named PyGame.

`Chapter 12`, *Learning About Character Animation, Collision, and Movement*, covers game animation, game character movement (such as jumping and walking), random object generation, game loops, collision and hit pipes, scrolling backgrounds, and scoreboards.

`Chapter 13`, *Coding the Tetris Game with PyGame*, deals with basic PyGame graphics, multi-dimensional list processing, increasing game speed and difficulty, the menu for a game, the creation of a game grid, and shapes and valid space determination.

`Chapter 14`, *Getting to Know PyOpenGL*, covers the installation of PyOpenGL on your machine. We will see how to create an OpenGL window. We will make a simple rectangle to begin with, and then look at PyOpenGL and see how the `draw()` method of PyOpenGL works. We will also learn how to draw objects from vertices and edges, adding views for object and clipping parameters.

`Chapter 15`, *Getting to Know Pymunk by Building an Angry Birds Game*, covers Pythonic 2D physics simulation. We will create a space that contains the simulation and sets its gravity, create a body with mass and moment, set the position of the body, create a box shape and attach it to the body, and then add both the body and shape to the simulation. We will create a complete Angry Birds game clone with Pymunk, dealing with sprite sheets and 2D physics.

`Chapter 16`, *Learning Game AI – Building a Bot to Play*, shows how to create game AI. In this game (snake), both the computer and you play as a snake, and the computer snake tries to catch you. The opponent AI tries to determine and go to the destination point based on your location on the board.

To get the most out of this book

To make the most of the information presented in this book, you are encouraged to follow along with the examples. Prior knowledge of Python is not required, but experience of mathematical concepts such as arithmetic and logical operations is essential for understanding the code thoroughly. Python-based applications are not limited to any particular OS, so all that is required is a decent code editor and a browser. Throughout the book, we have used the PyCharm Community 2019.2 editor, which is an open source editor and is free to download.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learning-Python-by-building-games`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Visit the following link to check out videos of the code being run:

`http://bit.ly/2oE9mHV`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The screenshot shows the edited `python_ex_1.py` file."

A block of code is set as follows:

```
n = int(input("Enter any number"))
for i in range(1,100):
    if i == n:
        print(i)
        break
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def fun(b):
    print("message")
    a = 9 + b
    move_player(a)

fun(3)
```

Any command-line input or output is written as follows:

```
>>> cd Desktop
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In the installer, make sure you check the **Add Python to PATH** box."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Getting to Know Python - Setting Up Python and the Editor

Python is notorious in the data and analytics industry, but it is still a hidden artifact in the gaming industry. While making games using other gaming engines such as Unity and Godot, we tend to combine our design logic with core programming principles. But in the case of Python, it is mostly the analysis of problems and programming paradigms that coalesce together. A program flow or structure is a sequence that is dovetailed with its programming paradigms. A programming paradigm, as its name suggests, facilitates the programmer to write a solution to a problem in the most economical and efficient way possible. For instance, writing a program in two lines of code instead of ten lines is an outcome of using a programming paradigm. The purpose of program flow analysis or structural analysis is to uncover information about procedures that need to be invoked for various design patterns.

In this chapter, we will learn about the following topics:

- Introducing programming with Python
- Installing Python
- The building blocks of Python
- Installing the PyCharm IDE
- Programming code without *Hello World*

Technical requirements

The following is a list of the minimum hardware requirements you'll need for this book:

- A working PC with a minimum of 4GB RAM
- An external mouse adapter (if you are using a laptop)
- A minimum of 5GB of hard disk space to download an external IDE and Python packages

You will need the following software to get the most out of this book (we will download all of them in this chapter):

- Various open source Python packages like pygame, pymunk and pyopenGL
- The Pycharm IDE (community version), which you can find at `https://www.jetbrains.com/pycharm/`
- Various open source packages, such as `pygame` and `pycharm`
- The code for this chapter, which can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter01`

Check out the following video to see the code in action:

`http://bit.ly/2o2pVgA`

Introducing programming with Python

The old adage of programming states the following:

> *"Coding is basically the computer language that's used to develop apps, websites, and software. Without it, we'd have none of the major technology we've come to rely on such as Facebook, our smartphones, the browser we choose to view our favorite blogs on, or even the blogs themselves. It all runs on code."*

We couldn't agree more with this. Computer programming can be both a rewarding and tedious activity. Sometimes, we might be in a situation where we can't find the tweaks of the exception (unexpected behavior of the program) that we caught in the program and, later, we find that the error was because of wrong modules or bad practices. Writing programs is similar to *writing essays*; first, we have to learn about the patterns of an essay; then, we analyze the topics and write them; and finally, we check the grammar.

Similar to the process of writing an essay, when writing code, we have to analyze the patterns or grammar of the programming language, then we analyze the problems, and then we write a program. Finally, we check its grammar, which we normally do with alpha and beta testing.

This book will try to turn you into a person who can analyze a problem, build noble logic, and come up with an idea that will solve that problem. We won't make this journey monotonous; instead, we will learn about Python syntax by building games in each chapter. By the end of this book, you will be thinking like a programmer—maybe not a professional one, but at least you will have developed the skill to make your own programs using Python.

There are two crucial things you'll learn about in this book:

- Firstly, you will learn about the vocabulary and grammar of Python. I don't mean learning about Python theory or history. First, we have to learn about Python syntax; then, we will see how we can create statements and expressions with that syntax. This step includes collecting data and information and storing it in an appropriate data structure.
- Then, you will learn about the procedures that come with the idea of calling the appropriate methods. This process includes using the data that was collected in the first step to get the intended output. This second step is not specific to any programming language. This is going to teach us about various programming prototypes rather than just Python.

Learning any other programming languages after learning about Python is a lot easier. The only difference you will observe in other programming language is syntax complexities and program debugging tools. In this book, we will try to learn about as many programming paradigms as possible so that we can start a programming career.

Are you still unsure about Python?

Let's take a look at some of the products that have been made with Python:

- No list starts without mentioning Google. They use it in their web search system and page rank algorithm.
- Disney uses Python for its creative processes.
- BitTorrent and DropBox are written in Python.

- Mozilla Firefox uses it to explore content and is a major contributor to Python packages.
- NASA uses it for scientific purposes.

The list goes on and on!

Let's take a look at how code procedures work in simple terms.

Explaining code procedures

To explain how code procedures work in simple terms, let's take the example of making an omelet. You start by learning the basics from a recipe book. First, you gather some utensils and make sure they are clean and dry. After that, you beat the eggs, salt, and pepper until it's all blended. Then, you add butter to your non-stick pan, add your egg mixture, and cook it or even tilt the pan to check whether every part of the omelet is cooked or not.

In terms of programming, first, we talk about collecting our tools, such as the utensils and eggs, which relates to collecting data that will be manipulated by the instructions we write in our programs. After that, we talk about cooking the eggs, which is your methods. We normally manipulate data in methods to get output in a form that is meaningful to the user. Here, the output is an omelet.

Giving instructions to a program is the job of a programmer. But let's distinguish between a client and a programmer. If you are using a product where you give instructions to the computer to perform tasks for you, then you are a client, but if you design instructions that will complete tasks for a product you've created for everyone, this indicates that you are a programmer. It is only a matter of *for one* or *for everyone* to determine whether a user is a client or programmer.

Some of the instructions we will use in our Windows Command Prompt or Linux Terminal will be for opening the directory of our machine. There are two ways of performing this action. You can either do it using a GUI, or you can use the Terminal or command prompt. If you type in the `dir` command in the respective field, you are now telling the computer to display the directories in that location. The same thing can be done in any programming language. In Python, we have modules to do this for us. We have to import that module before we can use it. Python provides a lot of modules and libraries to perform such operations. In a procedural programming language such as C, which allows low-level interaction with memory, this makes it harder to code, but with Python, it is easier to use the standard library, which makes the code shorter and readable. David Beazley, the author of *How to Think Like a Computer Scientist Learning Python*, was once asked, *why Python?* He simply replied, *Python is simply a lot of fun and more productive*.

Conversing with Python

Python has been around for many years (nearly 29), and regardless of all of the upgrades it has had to go through, it's still standing as the easiest language for beginners to learn. The primary reason for this is that it can be correlated to the English vocabulary. Similar to how we make statements with English words and vocabulary, we can write statements and operations with Python syntax that commands can interpret, execute, and provide us with a result. We can make a sentence such as *go there* as a command to reflect the position of something with conditionals and flow controls. Learning the syntax of Python is pretty easy; the actual task is to use all of the resources provided by Python to build brand new logic to solve intricate problems. Just learning the basic syntax and writing a couple of programs is never enough; you have to practice enough so that you can come up with revolutionary ideas to solve real-world problems.

We have a lot of vocabulary in the English dictionary. Unlike the English dictionary, Python only contains a few words in its container, which we normally call reserved words. There are 33 of them in total. They are instructions that tell the Python interpreter to perform specific operations. Modifying them isn't possible—they can only be used to perform specific tasks. In addition, when we call a print statement and write some text in it, it is expected that it prints out that message. If you want to make a program that takes input from the user, calling the print statement is useless; the input statement has to be called to achieve that. The following table shows our 33 reserved words:

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

Each of the preceding words can be found in our English dictionary. In addition, if we search for the word `return` in the dictionary, it simply gives us the verb meaning of coming or going back to the original place. The same semantics are used in Python; when you use the return statement with functions, then you are pulling out something from the function. In the upcoming chapters, we will see all of these keywords in action.

Now that we have started to learn how to converse in Python by examining its keywords, we will install Python. Gear yourself up and open your machine for some fun.

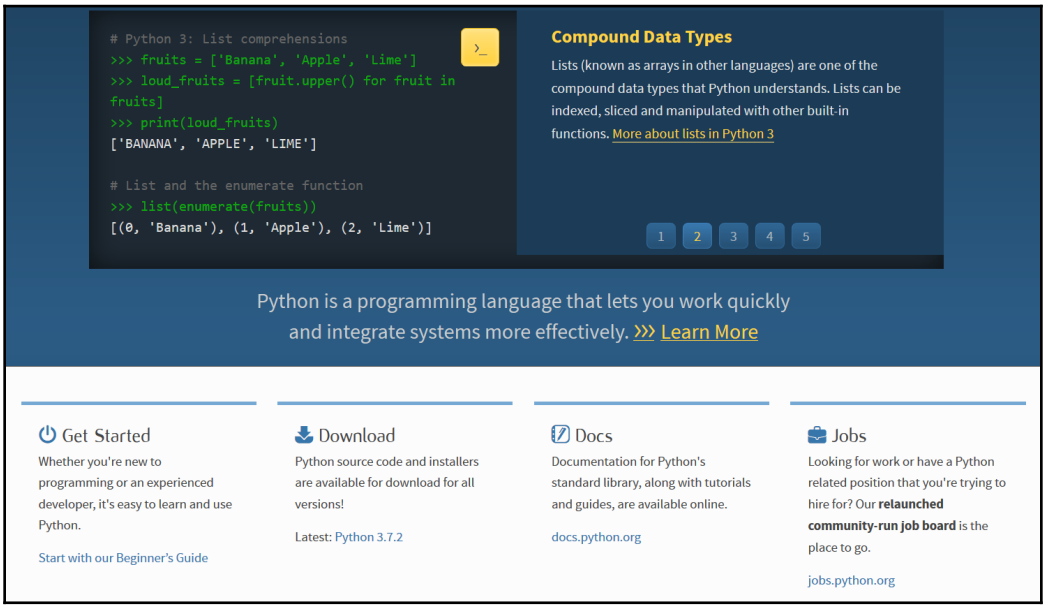
Installing Python

In this section, we will look at installing Python on Windows and macOS.

For the Windows platform

Python doesn't come pre-installed on Windows. We have to download it manually from its official website and then install it. Let's look at how to do this:

1. First of all, open your favorite browser and open the following URL: `https://www.Python.org/`.
2. You will be directed to the page that's shown in the following screenshot. Once you have been redirected to Python's official website, you will see three sections: **Download**, **Docs**, and **Jobs**. Click on the **Download** section at the bottom of the page:



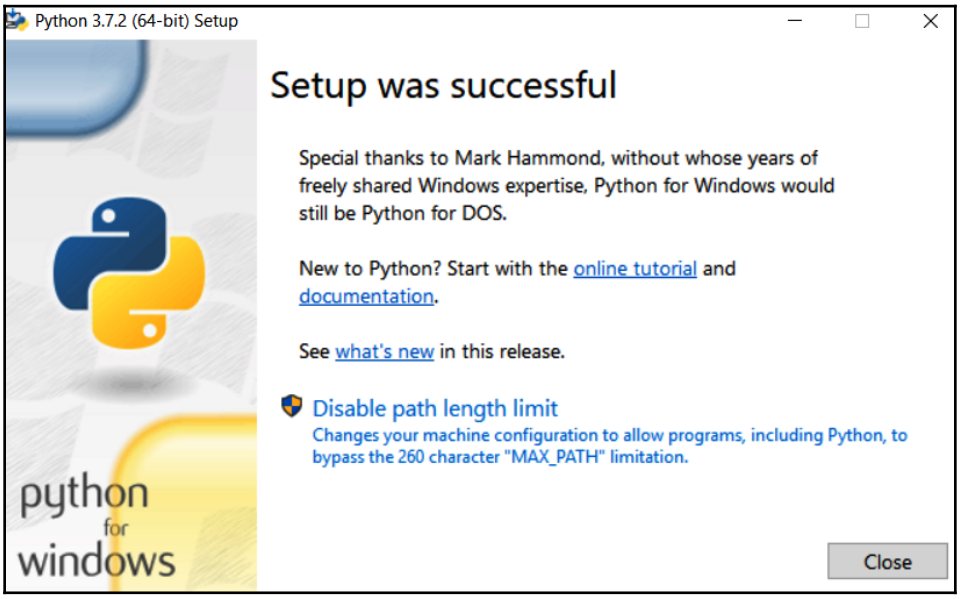
3. You will see a list of files, as shown in the following screenshot. Pick the file that's appropriate for your platform. We're looking at the installation for Windows in this section, so we will click on the Windows executable link. This is highlighted in the following screenshot:

Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		02a75015f7cd845e27b85192bb0ca4cb	22897802	SIG
XZ compressed source tarball	Source release		df6ec36011808205beda239c72f947cb	17042320	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	d8ff07973bc9c009de80c269fd7efcca	34405674	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	0fc95e9f6d6b4881f3b499da338a9a80	27766090	SIG
Windows help file	Windows		941b7d6279c0d4060a927a65dcab88c4	8092167	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	f81568590bef56e5997e63b434664d58	7025085	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	ff258093f0b3953c886192dec9f52763	26140976	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	8de2335249d84fe1eeb61ec25858bd82	1362888	SIG
Windows x86 embeddable zip file	Windows		26881045297dc1883a1d61baffeecaf0	6533256	SIG
Windows x86 executable installer	Windows		38156b62c0cbcb03bfddeb86e66c3a0f	25365744	SIG
Windows x86 web-based installer	Windows		1e6c626514b72e21008f8cd53f945f10	1324648	SIG

4. After clicking on that, you will get a file that needs to be downloaded. After opening that downloaded file, you will get the installer, as follows:

5. In the installer, make sure you check the **Add Python to PATH** box. This will put the Python library files in our environment variables so that we can execute our Python programs. Afterward, you will get a message about its successful installation:



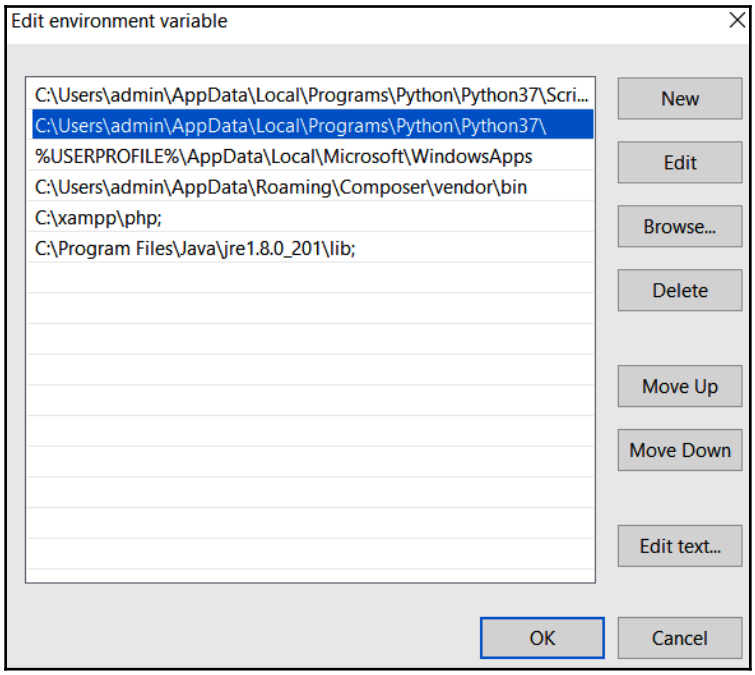
6. Press the Windows key + *R* to open **Run** and type `cmd` in the **Run** tab to open your Windows Command Prompt. Then, type `Python` in the command shell:

```
Microsoft Windows [Version 10.0.17763.316]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\admin>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you get the Python version that's displayed in the preceding screenshot, then Python has been successfully installed on your machine. Congratulations! Now, you can get your hands dirty by writing your first program with Python.

If you get an error saying **Python is not recognized as an internal or external command**, you have to explicitly add Python to the path environment variable. Follow these steps to do so:

1. Open the **Control Panel**, navigate to **System and Security**, and then go to **System** to view the basic information about your system.
2. Open your **Advanced system settings** and then **Environment Variables....**
3. In the **Variable** section, search for **Path**. Select the **Path** variable and press the **Edit...** tab.
4. Click **New** in the **Edit Environment Variable** tab.
5. Add this path so that it's pointing to your Python installation directory, that is, **C:\Users\admin\AppData\Local\Programs\Python\Python37**.
6. Click on the **OK** button to save these changes:

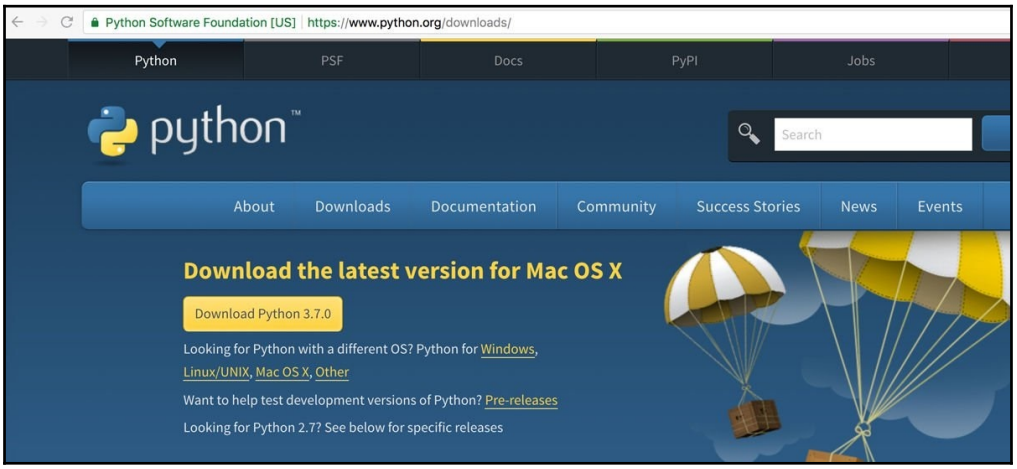


Now, we have successfully installed Python for Windows. If you are using a Mac, the next section will help you to access Python too.

For the Mac platform

Python comes pre-installed with Mac OS X. To check the version of Python you have installed, you should open your command line and type `Python --version`. If you get a version number of 3.5 or newer, you don't need to go through the installation process, but if you have version 2.7, you should follow these instructions to download the latest available version:

1. Open your browser and type in `https://www.Python.org/downloads/`. You will be sent to the following page:



2. Click on the **macOS 64-bit/32-bit installer**. You will be provided with a `.pkg` file. Download it. Then, navigate to that installed directory and click on that installer. You will see the following tab. Press **Continue** to initiate the installer:

Whenever you download Python, a bundle of packages will be installed on your computer. We can't use those packages directly, so we should call them individually for each independent task. To write programs, we need an environment where we can call Python so that it can complete tasks for us. In the next section, we will explore the user-friendly environment provided by Python where we can write our own programs and run them to view their output.

Now that you have installed Python version 3.7 on Mac OS X, you can open your Terminal and check the version of Python you have with the `python --version` command. You will see Python 2.7.10. The reason for this is that Mac OS X comes preinstalled with version 2.7+ of Python. To use the newer version of Python, you have to use the `python3` command. Type the following command into your Terminal and observe the result:

```
python3 --version
```

Now, to make sure Python uses the interpreter with the newer version that you just installed, you can use an aliasing technique that will replace the current working Python version with Python3. To perform aliasing, you have to follow these steps:

1. Open your Terminal and type in the `nano ~/.bash_profile` command to open a bash file using the nano editor.
2. Next, go to the end of the file (after import PATH) and type in the `alias python=python3` command. To save a nano file, press *Ctrl* + *X* and then *Y* to save.

Now, open your Terminal again and type in the same command that we used previously to check the Python version we have. It will be updated to the newer version of Python. From now on, in order to run any Python file from your Mac, you can use this Python command, followed by the signature of the file or filename.

Introducing the Python Shell and IDLE

The Python Shell is similar to Command Prompt for Windows and the Terminal for Linux and Mac OS X where you write commands that will be executed in the filesystem. The results of these commands are printed instantly within the shell. You can also get direct access to this shell using a Python command (> python) in any Terminal. The result will contain an exception and an error due to the improper execution of the code, as follows:

```
>>> imput("Enter something")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    imput()
NameError: name 'imput' is not defined


>>> I love Python
SyntaxError: invalid syntax
```

As you can see, we ran into an error and the Python IDE is explicitly telling us the name of error we ran into, which in this case is `NameError` (a type of syntax error). `SyntaxError` occurs due to an incorrect pattern of code. In the preceding code example, when you write the `I love Python` syntax, this implies nothing to the Python interpreter. You should write proper commands or define something properly if you want to rectify that problem. Writing `imput` instead of input is also a syntax error.

Logic errors or semantic errors occur even if your program syntax is correct. However, this doesn't solve your problem domain. They are dangerous as they are hard to track. The program is perfectly correct but does not solve any problem that it's intended to.

When you download the Python package on your machine, a **Python Integrated Development Environment** (**IDE**) called IDLE (Python's built-in IDE) is downloaded automatically onto your machine. You can type `IDLE` into the search bar to navigate to this environment. IDLE is a free open source program that provides two interfaces where you can write code. We can write scripts and Terminal commands in IDLE.

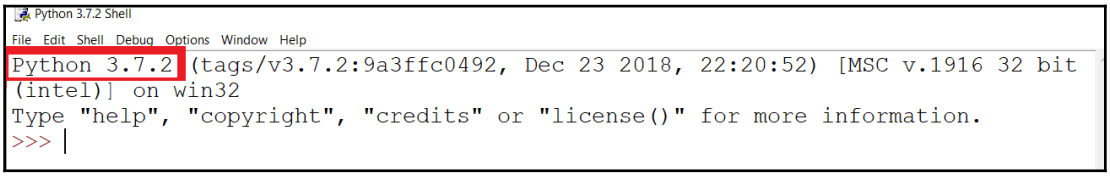
Now that we are familiar with what not to do in the Python Shell, let's talk about the particulars of the Python Shell—an environment where you can write your Python code.

Particulars of the Python Shell

As we mentioned previously, in this section, we are going to take a tour of the particulars of Python. This includes Python's built-in shell, Python's text editor (usually called Python script), and the Python documentation page.

Follow these steps to learn about the particulars of the Python Shell:

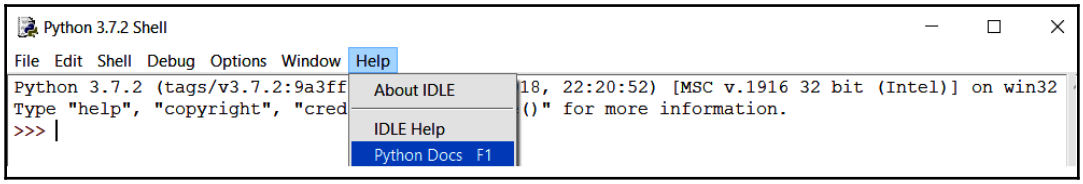
1. When you open Python Shell, you will see the following window. The first thing you will see in the shell is Python's current version number:



2. In the Python shell, there are three angular brackets placed next to each other, like this: >>>. You can start writing your code from there:

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 5 + 7
12
>>> print("Hello")
Hello
>>>
```

3. Press *F1* to open the Python documentation or go to the **Help** tab and click **Python Docs F1** (on a Windows machine). To get access the documentation online, please go to `https://docs.python.org/3/`:



I hope that you are now familiar with the Python Shell. We are going to write a lot of code in the shell, so make sure you get familiar with it by customizing or playing with this environment a bit longer. After you are done with it, you can proceed to the next section, where you are going to learn about what you need to know before you write your first Python program.

Building blocks of Python

There are some conventional patterns that we make use of while writing programs in Python. Python, being a high-level language, doesn't care about low-level routines, but has the capability to interact with them. Python is made up of six building blocks. Every program that is made with Python revolves around them. These building blocks are input, output, sequential execution, conditionals, recursion, and reuse. Let's go over them now:

- **Input**: Input is everywhere. If you make an application with Python, it mainly deals with formatting the input of the user in a way that would harvest meaningful results. There is an built-in `input()` method in Python so that we can get data input from the user.
- **Output**: After we have manipulated the data that was entered by a user, it's time for us to present it. In this layer, we make use of design tools and presentation tools to format meaningful output and send it to the user.
- **Sequential execution**: This preserves the sequence of execution of statements. In Python, we normally use indentation, which is spaces that denotes scopes. Any commands that are at zero-level indentation are executed first.
- **Conditionals**: These provide flow control to programs. Based on comparisons, we make logic that will make a flow of the code and will either execute or skip it.
- **Recursion**: This is anything that needs to be done until some condition is met. We normally call them **loops**.
- **Reuse**: Write code once, use it a million times. Reuse is a paradigm where we write a set of code, give it a reference, and use it whenever required. Functions and objects provide reusability.

Writing a program in Python Shell may be easy to debug for most programmers, but it can create overhead in the long run. If you want to save your code for future reference or you want to write multi-line statements, you will probably be overwhelmed with the deficit feature of the Python interpreter. To solve this problem, we have to create a script file. They are called scripts because they allow you to write multi-line statements in single files that you can run immediately. This comes in handy when we have multiple data storage and files to deal with. You can distinguish a Python file from other files by its extension, that is, `.py`. You should also save your Python script files with the `.py` extension.

To run your script file from a Terminal or Windows Command Prompt, you have to tell your Python interpreter to run that file by its filename, like so:

```
$ Python Python_file_name.py
```

In the preceding command, `$` is an operating system prompt. First, you have to call the Python interpreter with the `Python` command and tell it to execute the file name next to it.

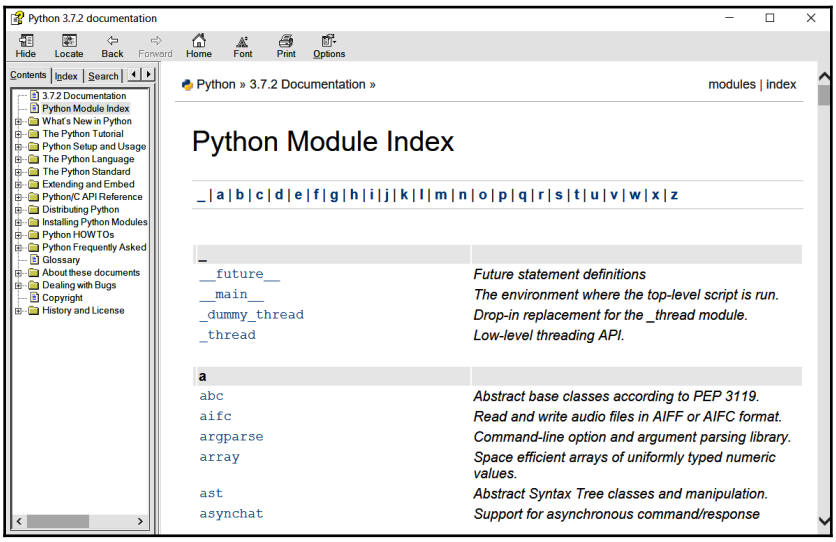
If you want to see the content of the `Python` file within the Terminal, use the following command:

```
$ cat Python_file_name.py
$ nano Python_file_name.py
```

To exit the Python Terminal, write the `exit()` command in the Terminal.

Now that we've learned how to open and exit the interface of the Python environment, we have to learn about its building blocks. Many beginners make a fallacious assumption that a program has only two building blocks: input and output. In the next section, we will see how to debunk this assumption by employing six building block of programming.

The toughest part of programming is learning the art of programming paradigms such as object-oriented programming, DRY principles, or the linear time complexity model. If you get a good grasp of these prototypes, learning any new programming language will be a piece of cake. Having that said, learning all of these paradigms with Python is a lot easier than Java or C# as, in Python, the code will be shorter and the syntax is English-friendly:



Before we write our first program, we will install one more IDLE for the upcoming chapters where we will be writing program-intricate games. In those types of games, the features that are provided by IDLE are not enough, and so we will see how to install PyCharm—an advance IDLE—in the next section.

Installing the PyCharm IDE

Earlier in this chapter, we discovered IDLE. We have already seen an environment where we can write code and get output straightaway. However, you can probably imagine what happens if we have lots of code to be executed at once, maybe 1,000 lines of code, one by one. We have to solve this problem by writing a script, which is a collection of Python code. This will be executed at once instead of being executed line by line in the shell of IDLE.

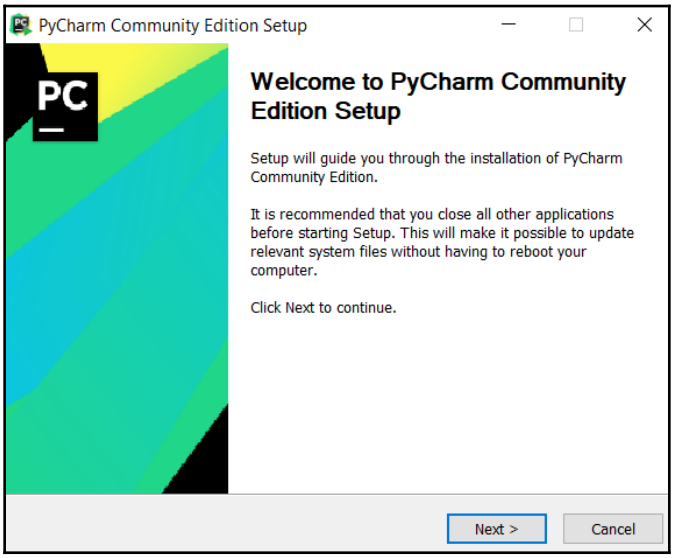
If you want to write a script, follow these steps:

1. Open the **Search** tab from your PC and type `IDLE`.
2. Click on the **File** tab.
3. Press on **New File**.
4. A new file will be generated. You can write multiple expressions, statements, and commands in that single file. The left-hand side of the following screenshot shows the Python script where you can write multi-line statements, while the right-hand side of the following screenshot shows the Python Shell, where you will execute your script and get instant results:

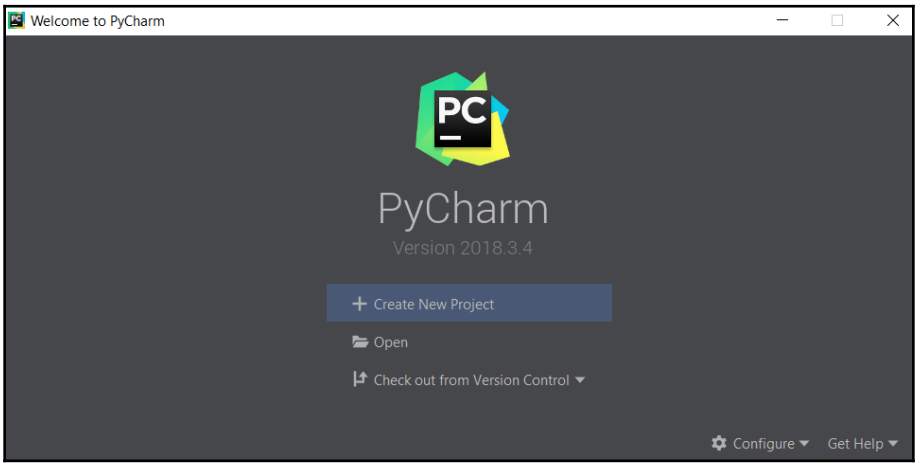
After you've finished writing your scripts, you have to save it before running it. To save your file, go to **File** and click on **Save**. Provide the appropriate filename for your script by placing the `.py` extension at the end of it, for example, `test.py`. Press *F5* to execute your script file.

We will build many games throughout this book where we will have to deal with images, physics, rendering, and the installation of Python packages. This IDE, that is, IDLE, is not capable of providing smart IDE features such as code completion, integration and plugins, and branching of packages. Hence, we have to upgrade to the best Python text-enriched IDE, that is, the PyCharm IDE. Let's get started:

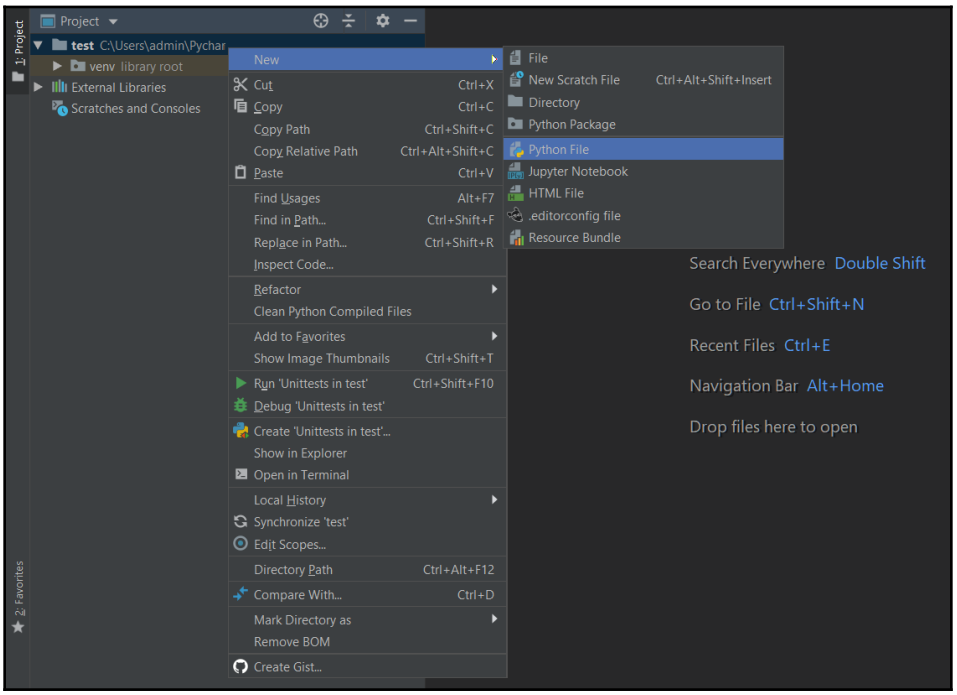
1. Visit `https://www.jetbrains.com/pycharm/` to download the PyCharm environment. The installation of PyCharm is as simple as the installation of any other program. After you've downloaded the installer from the website, click on that installer. You should see the following window:



2. Click on the **Next>** button and install it on the appropriate drive. After you've installed it, search for `PyCharm` in the search bar and open it. You should see the following window:



3. Now, click **+Create New Project** and give your project a name. To create a new Python file, left-click on your project name, click on **New**, and then click the **Python File** tab:



Now, we have everything that we need to master this book—I mean the tools, but obviously, we have to learn every possible paradigm of Python to master the concept of Python. Now that you are fully equipped with these tools, let's write our first effective Python program, *No Hello World.*

Programming code without Hello World

There is a tradition in the programming world to print *Hello World* as our first program. Let's break the mold and make our first program one that takes input from the user and prints it to the console. Follow these steps to execute your first program:

1. Open your IDLE and type in the following commands:

```
>>> print(input("Enter your Name: "))
```

2. Press *Enter* to execute the command. You will get a message saying **Enter your Name**:. Type in your name and hit *Enter*. You will see the output print the name that you just passed.

We made use of two commands here, also known as functions We will learn about them in the upcoming chapters. Let's go over these two functions now:

- `input()` is a built-in function of Python that will take input from the user. Spaces are also included as characters.
- `print()` is a built-in function of Python that will print whatever is passed inside the parentheses.

Now that we have started to code our first program with Python using Python's built-in IDLE, it's your turn to test the working of IDLE. Since we are going to be building lots of games using IDLE, make sure you get familiar with its interface. The core programming modules that we learned about in this chapter, such as Python keywords and the input-print function, are important as they help us to build programs that can take input from users and display it.

Summary

In this chapter, we took a tour of the basics of Python and learned how similar its vocabulary is to English. We installed the Python package on our machine and viewed the pre-installed IDE of Python, known as IDLE. We saw how scripts can be written on the Python IDE and how we can execute them. Then, we installed the feature-rich Python text editor known as PyCharm IDE on our machine. We wrote our first Python program, which is able to take input from the user and display it on the screen.

The skills that you have acquired in this chapter are fundamental for building the flow of a program. For instance, our program was able to take input/output data. Any game that's made in Python has to be interactive for the users or players, and this is done through the input and output interface. In this chapter, we looked at how to take input from a user and display it. As we continue with this book, we will explore various ways to build a program that handles user events such as taking input from the mouse, keyboard, and screen-taps.

The next chapter will be crucial as we will look at Python essentials such as values, types, variables, operators, and modules. We will also start to build a tic-tac-toe game.

2
Learning the Fundamentals of Python

Python doesn't need in-game development, design, and analysis are considered the steps that are done before programming. Designing and analysis require that we brainstorm for ideas, model the procedures, and format the input. All of these procedures have something to do with data. Data can be something as simple as a list of numbers or as complex as weather history. This data has its own types and structures. Data needs to have its own storage location so that we can reference it. Python provides an abstraction of data in the form of objects that facilitate us to create a nested data structure.

This chapter will give you a roller-coaster ride of the core programming paradigm within Python. We will begin by learning about the different data types that are available and ways to capture them in variables or storage units. We will learn about different mathematical operations (arithmetic and trigonometric) using the `math` module. By the end of this chapter, we will have made our first game, tic-tac-toe, by using the knowledge that we have learned in this chapter.

In this chapter, we are going to cover the following topics:

- Handling values and data
- Variables and keywords
- Operators and operands
- Writing comments in the code
- Requesting User Inputs
- String Operations
- Building your first game – tic-tac-toe
- Possible errors and warnings
- Game testing and possible modifications

Technical requirements

You will need the following requirements to get the full benefits of this chapter:

- You'll need the Python **IDLE**
- The code assets for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter02`

Check out the following video to see the code in action:

`http://bit.ly/2o6Kto2`

Handling values and data

Software is evaluated as good or bad based on its capability to handle data. Every program has its own database design and implementation. A database is a schema where data is stored in such a way that it can be retrieved fast and securely so that it can be manipulated. It is assumed that social networks such as Facebook and Twitter collect 1.7 billion people's data each day. Such huge amount of data, which is collected on a daily basis, should be addressed properly because we don't have enough memory to store and process it. Hence, Python provides flexible built-in methods to map, filter, and reduce these datasets so that they can be stored and fetched faster for processing.

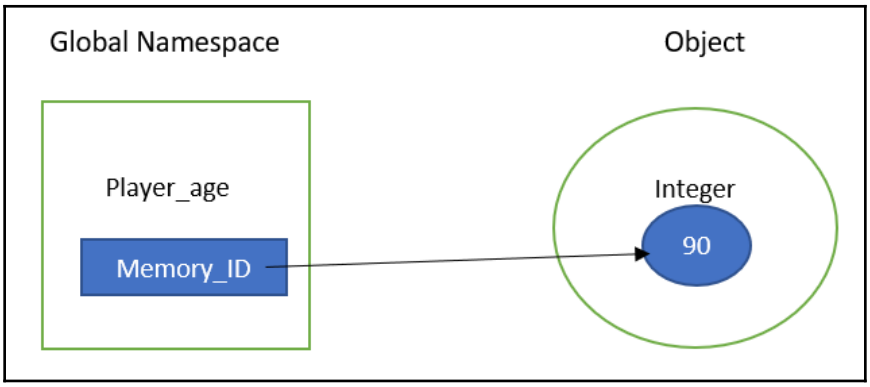
Python is lightning-fast when it comes to storing data as a schema. Its integration with big data platforms such as Hadoop, which inherits its compatibility, is the main reason we use Python in big datasets. Powerful packages such as NumPy, pandas, and scikit-learn provide data support for today's data and analytical needs.

A value is the representation of data for some attributes that are computed by programs. Here, attributes are the properties of any object. For example, when we talk about a person, we reference them by name, age, and height. These attributes have an *r*-value (the content of the attribute) and an *l*-value (memory location) attached to them. The *content of the attribute* refers to the value that is stored as the content of the variable, while the *memory location* refers to the physical place where the value is stored. For example, `name = "Python"` has a `name` variable as an attribute; its *r*-value is `Python` and its *l*-value is a unique ID that is assigned by the Python parser automatically as a memory location for the name attribute.

In Python, values are stored in the form of objects. Objects have four particulars: ID, namespace, type, and value. Let's look at a simple example to uncover the particulars of an object:

```
>>> player_age = 90
```

When the `player_age` variable is created, its instance is created, which we call an object. Whenever an object is created, it receives a unique memory storage location, that is, a unique ID number, and assigns a type dynamically, that is, an integer, because we are assigning 90 to it. After that, the player variable is added to the namespace so that we can retrieve its value, that is, 90. The following diagram attempts to simplify this explanation:



Whenever any assignment statement is executed, the parser creates an object, which gets a unique memory ID from where we can reference the value of the variable. Since Python is a dynamically typed language, it assigns a type of variable dynamically by analyzing the value that's been assigned to the variable. Eventually, it adds that variable to a global namespace so that whenever you want to fetch that variable, you can use the variable name. Here, `memory_ID` is the location that points to the value of an object. Some programming languages, such as C, call this a pointer.

Each of these values has a type associated with it. `1` is an integer, `a` is a character, and `Hello World` is a string. `Hello World` is a collection of characters and it is called a string since it defines a string of characters. In the previous chapter, we saw an example where we asked a user for input. Whenever a user types something as input, it is considered a string. These values define objects in programming. Let's use a parrot as an example. It will have a name as a string, its age as an integer, and its sex as male or female, denoted by *M* or *F*, which are characters. In Python, a character is also represented as a string. To verify that, we have the `type` method. The `type` method is represented as follows:

```
>>> type('a')
```

The output of the preceding command will be `<class 'str'>`, which implies that the character is also part of a string. To check the type of any value, we can use the same `type` method:

```
>>> type(1)
>>> type('Hello World')
```

The preceding commands will show class as `int` and `str`, respectively.

Now, let's talk about numbers. Numbers are of two types: whole numbers and decimal numbers. As we saw, a whole number is an integer, but a decimal number is a float. We call decimal numbers floating numbers in Python as they are represented in floating-point format, like so:

```
>> type(3.4)
```

The output of the preceding command is `<class 'float'>`.

You can print these values in the Terminal using the `print` method. The `print` method takes values within parenthesis and gives the result in the interpreter, like so:

```
>>> print(1)
```

If you put `1` within the `print` statement, it will print `1` as a number. However, when you put `1` within double quotes, it will print `1` as a string, like so:

```
>> print("1")
<class 'str'>
```

Any number, text, or special symbols, such as @, $, %, or * you put inside a single quote or double quote will eventually be a string. The following is an example of a string: `1`, `Hello`, `False`, `#$(#`.

When you put a comma between two values in a `print` statement, it puts a space between them, like so:

```
>>> print("abc","abc")
```

The preceding code will give you the output of `abc abc`, but they are no longer considered strings. This is the first semantic error we've seen in Python. We passed `abc` as a string, but the result is a non-type:

```
>>> type(print("abc","abc"))
<class 'NoneType'>
```

This is a perfect example of a semantic error. We got the output without any errors, but we didn't get the result we wanted.

You can also check integers. It's not possible to print integers with a comma between them. The Python interpreter converts commas into spaces between each value that's passed:

```
>>> print(0,000,000)
```

This command will give us a result of `0 0 0`. Each comma was converted into spaces and printed. If you check the type of the value that is returned from the function, it will also be `NoneType`.

Now that we know about value and types, let's get ourselves familiar with variables and keywords.

Variables and keywords

Programming is all about accepting and manipulating values that we've learned about. We make use of variables while accepting and manipulating those values so that we can reference them for future use. Variables are like boxes, where you put in different things and fetch them whenever they're required. A variable is created with a name and a value assigned to it.

We use the equal to sign (=) to make an assignment statement. Variables are created with assignment statements; for example:

```
>>> myAge = 24
>>> info = "I love Python"
>>> isHonest = True
```

Here, we created three variables with assignment statements. In the first command, we made the `myAge` variable and assigned an integer to it. You do not have to specify types of variables explicitly in Python since Python does it internally. This is what makes Python a dynamically typed language. In the second command, we made the `info` variable and assigned a string to it. Finally, we made the `isHonest` variable and assigned a Boolean to it.

Boolean types are logic types. They are either `True` or `False`. Creating a Boolean variable is the same as creating other variables, for example, `is_hungry = True`.

Variables are basic pieces of data storage. We can assign one value to a variable at a time. Whenever you assign another value to the same variable name, it will overwrite the original one. For example, here, we made the `info` variable a string, but if I replace it with another value, say, `integer`, this is valid:

```
>>> info = 23
```

If you make variables in Python, it will create separate memory references for each variable. So, any time you replace the same variable in another value, the value at that particular position will be retrieved and overwritten with the new one. Variable names are pointers to the value in the reserved memory location. You can't store multiple values in a variable. You have to use advanced data structures to do this. We will cover this in the upcoming chapters (CHAPTER 4: Data Structures and Functions: Refine Your Game with Taste of AI).

Assigning multiple variables to different variables can be done in a single line of code. We can assign them with a single assignment statement. The variable's name should be given at the left-hand side and they should have commas between them. You can create as many variables with distinct data types in a single line as you want with the following command:

```
>>> even, odd, num = 2, 3, 10
```

You can see the value of your variable by directly writing the variable's name in the Terminal:

```
>>> even
```

If you only write the name of a variable in the script, the value won't be printed. Instead, it will terminate. If you want to print something on the screen, you have to use the `print()` method. To print the value of any variable, type `print(variable_name)` in your shell or script, like so:

```
>>> print(even)
```

If you want to see the type of the value that's stored in the variable, you can call the `type()` method. To do this, pass the variable's name inside parentheses:

```
>>> type(even)
```

The preceding command will give us the output of `<class 'int'>`, which implies that integer values can be stored in variables.

We can also assign the same value to multiple variables in Python. In the preceding command, instead of assigning multiple values, we assigned a single value to it, like so:

```
>>> even, num = 10
```

In the preceding command, we assigned an integer value of 10 to two different variables, `even` and `num`.

Python doesn't need variable instantiation and declaration. Hence, there is no need for reserved memory space in Python. Python does this internally when we create variables with assignment statements.

Python has reserved 33 words as keywords for specific functionality. We cannot use them to name variables. Python checks the name of the variable with these keywords internally with its in-built script. Whenever it detects one of those words, it will throw a syntax error, as in the following example:

```
>>> and = 23
```

The preceding command isn't executed and cannot be used as a variable name because it is a keyword. Python uses it to do some logical operations. However, if you create a variable called `And` and assign a value to it, Python will create the `And` variable for you. For Python, `And` and `and` are not the same. It is a case-sensitive language.

To avoid any issues with your variable names, we can follow a few simple rules. We'll go over these rules in the following section.

Rules for naming variables

We normally choose meaningful variable names because, in the long run, there may be cases where we completely forget about code sequence and flow, and variables that don't have a proper name can create confusion. Although you can create variables with any name by following some rules, it is highly suggested to create variable names that make sense. Let's say you are making a game where you want to create a variable for the player's health; naming that variable `a` is not good practice. Instead, you should name it `player_Health` so that it's clear to you and those who may look at your code what the code in this variable does.

Usually, from a programming perspective, there are two ways of giving a variable a name effectively. Two of them are famously known as `CamelCase` and `PascalCase`. Observing the naming convention of the previously defined variable, `playerHealth`, the first character of the variable should be lowercase and all of the others should be in uppercase. Similarly, in the case of `PascalCase`, every first character of the variable should be in uppercase. Hence, using `PascalCase`, the previously defined variable can be written as `PlayerHealth`. You can use either of them to name your variable.

Your variable name can be any length. It can contain a combination of uppercase alphabetical letters (A-Z), lowercase letters (a-z), digits (0-9), and an underscore (_). An underscore is used in-between two words to distinguish two entities in a variable. For example, the `player_Health` variable is made up of two words. We use the underscore in-between them. Alternatively, you can also use `camelCase`, where you start your first word in lowercase and the first letter of the second word in uppercase, for example, `playerHealth`.

We can also use an underscore at the start of the variable's name. We use them in our code if it's being used as a library for others. We can use them in recursive statements, too, as in the example:

```
>>> _age = 34
```

There are some rules we need to follow while naming a variable, otherwise Python will declare it illegal and throw a syntax error. The following screenshot shows some illegal assignment statements:

```
>>> 45age = 45
SyntaxError: invalid syntax
>>> ms%g = "Hi"
SyntaxError: can't assign to operator
>>> and = 45
SyntaxError: invalid syntax
>>>
```

To remove the preceding errors, we have to follow a few rules. Some are mandatory while some are just good practices:

- We give the variable a name that makes sense. Naming the age variable `age` is more meaningful than naming it `a`.
- We cannot use special symbols (@, #, $, and %) while naming variables. For example, n@me is not a valid variable name.
- A variable name should not start with digits. **45** age is not a proper variable name and Python will throw an error.
- Declare constants with an uppercase name, for example, `>>> PI = 3.14`.
- It's good practice to use `camelCase` to create variables name, for example, `>>> myCountry = "USA"`.

We have now seen what variables and keywords are and some rules to follow when naming them. Now, let's move on and see what operators and operands are.

Operators and operands

Math and programming are two distinct fields that are closely related. The former deals with theory and provides formulated principles to solve any problem domain, while the latter deals with using those principles to solve a business domain. Programming is all about accepting data using models and manipulating it with the appropriate mathematical operations. Operators are used to perform those operations. We have arithmetic and logical operators in Python.

Operators are symbols that perform computations such as addition, multiplication, division, and so on. Symbols such as `+`, `-`, and `/` are used to perform those operations. The values that operators are applied to are called operands. Some examples of operators are shown in the following code:

```
>>> 3 + 4
>>> 14 - 5 - 9
>>> 2 * 4
```

In the preceding examples, the result of the first operation is 7, the result of the second operation is 0, and the result of the last operation is 8. You can add or subtract as many numbers as you like within the shell. Here, all of the numbers are operands, and symbols such as `+`, `-`, and `*` are operators.

Another important operator in Python is division (`/`). In Python 3.x, the division operation results in floating-point numbers, as in the example:

```
>>> 10 / 4
```

The preceding operation gives you a result of 2.5. This is the same result that we get using a calculator.

In Python 2.x, the interpreter would truncate the decimal part and give us a result of 2. If you want to get the same result in Python 3.x, you should use floor division (`//`); for example:

```
>>> 10 // 4
```

The preceding operation will give us a result of 2 instead of 2.5.

Let's go over what we've learned so far, that is, values, variables, and operators. Let's combine all of these into one statement. This is known as an expression:

```
>>> x = 10 + 2 * 5
>>> x
```

You can combine all of these to make any type of expression. The assignment operation is the simplest expression to use. We saw the assignment operation while creating variables.

When there are multiple operators being used in an expression, the order of these operations becomes important to solve the expression. We'll go over the order of operations in the following section.

Order of operations

Let's recall the basic math that we mostly learned in our school days. You may have heard of the BODMAS rule or the PEDMAS rule. Whenever more than one operator is used in our expression, an operation is performed with this rule of precedence. Operations in brackets/parenthesis, exponentiation, division, multiplication, addition, and subtraction are performed in this order:

- **Parentheses/brackets**: This symbol has the highest precedence, which means that operations within parentheses are completed first. With the use of parentheses in your expression, you are telling the interpreter to explicitly execute a certain expression forcefully. For example, in `(10 - 5) + 5 * 6`, the operation within the parentheses is done first, that is, *10 - 5*, and then multiplication is done.
- **Exponential/of**: Th exponential operation is done after operations within parentheses are completed. The output of `9**0+1` is not 9; instead, it is 1. Exponential is done first, and then addition is done.
- **Division**: Division operations are done after exponential if operations, including division if it's not inside parentheses. For example, `10 / 2 + 3 + 9 / 3` is 11 but not 5. If the expression was `10 / (2 +3) + 9 /3`, the output would be 5.
- **Multiplication**: It has also the same precedence as that of division. However, if the expression has both division and multiplication, operations are done sequentially from left to right. Scanning from left to right, if we get multiplication before division, it is done first. For example, the output of `3*4 / 3` is `4` but not 3.999.
- **Addition and subtraction**: These two operations also have the same level of precedence. Thus, we perform these operations according to what comes first while scanning from left to right. For example, in terms of `5 - 5 + 6`, we subtract first as it comes first and then add, which gives us 6.

If you are still confused about the BODMAS/PEDMAS rule, you can simply use parentheses to make sure you get the intended result. In the next section, we will learn about two important operators: `//` and `%`. The former is known as floor division, while the latter is known as the modulus operator.

Modulus operator

Earlier, we saw how to use floor division (`//`) and how it provides us with only the quotient of the division operation. But if you want the remainder of your division, use the modulus operator. The modulus operator yields the remainder of when the first operand was divided by the second operand. The symbol for the modulus operator is the percentage sign (`%`). The following screenshot shows two operations: the first one is a floor division, which will result in a quotient, while the next one is a modulus operation, which will result in the remainder of the division:

```
>>> quotient = 23 // 2
>>> print(quotient)
11
>>> remainder = 23 % 2
>>> print(remainder)
1
>>>
```

The modulus operator is very useful when we want to search number patterns and make programs that can divide numbers based on that pattern. For example, we can check the remainder of division between any number and **2** to find whether the number is even or odd:

```
>>> 5 % 2
```

Since the preceding operation gives the remainder as 1, 5 can be considered an odd number.

All of the preceding operations are pretty basic and don't need any hard work to calculate. However, we know that computers are known for processing complex tasks. Hence, in the next section, we will learn about the `math` module, which is capable of performing intricate mathematical operations such as calculating trigonometric and complex equations.

Using the math module

Math is not only limited to addition and multiplication. So far, we have learned about various arithmetic operations. We haven't at logical operators and comparisons yet as those will be covered in the next chapter. To incorporate many domains of mathematics, Python has given us one powerful library, called the `math` module. We call the file that contains the code a module. These libraries are also called in-built libraries because they come prepackaged whenever we install Python.

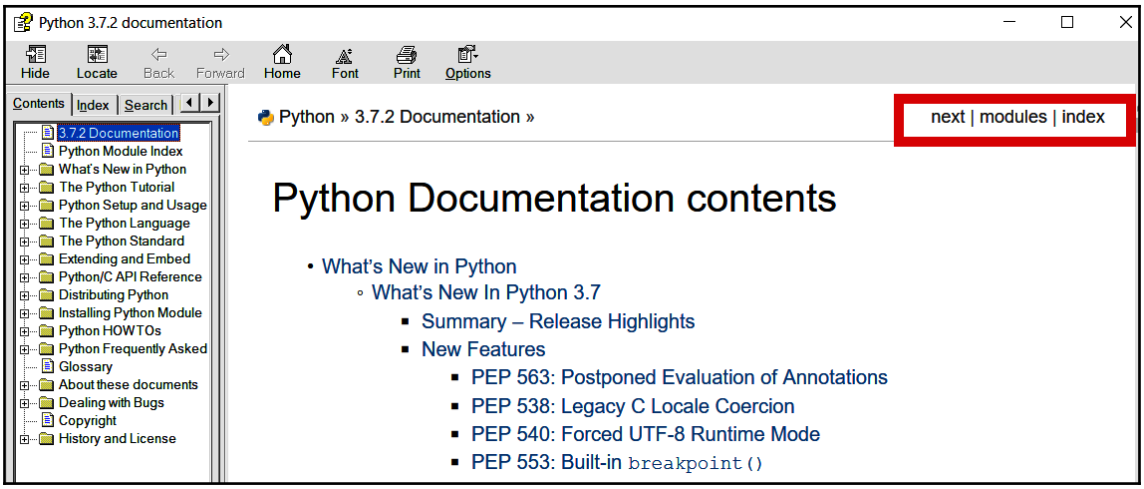
They are made by Python and we can call them whenever we want in our code without having to install it manually. If you want to use the code of any in-built library, you have to call it first. Calling them means importing them. To import and use that in-built library, we use the `import` keyword. As you may recall from the previous chapter, it's a reserved word that has a specific purpose in Python. Hence, the `import` keyword imports any library into your code. If you want to import the `math` module, for example, just write the following:

```
>>> import math
```

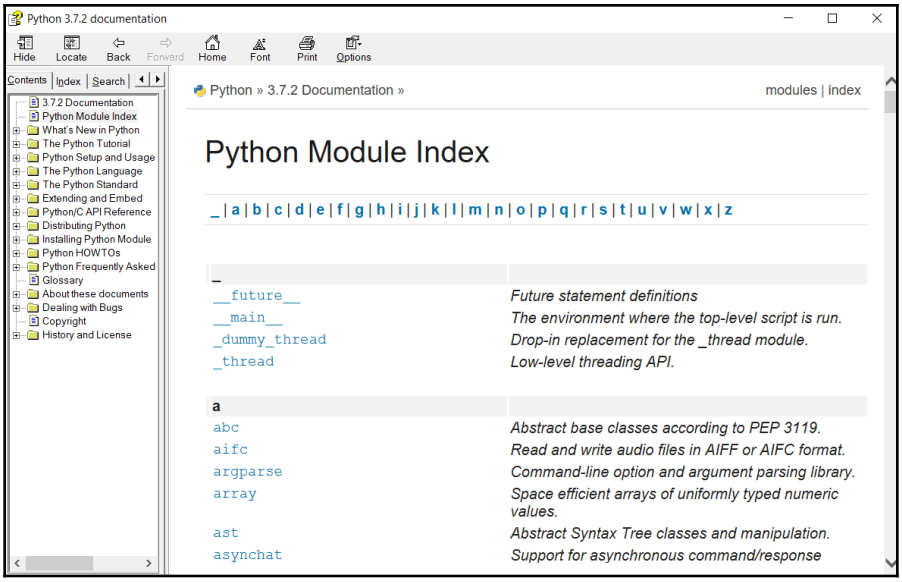
You will instantly see the next line with an empty shell, like this: `>>>`.

That just specifies that you are importing it. Import statements are not the same as print or input methods, which give us an instant response. We should call something from that module in order to see any response or result. The `math` module provides us with numerous operations. These can be accessed by following these steps:

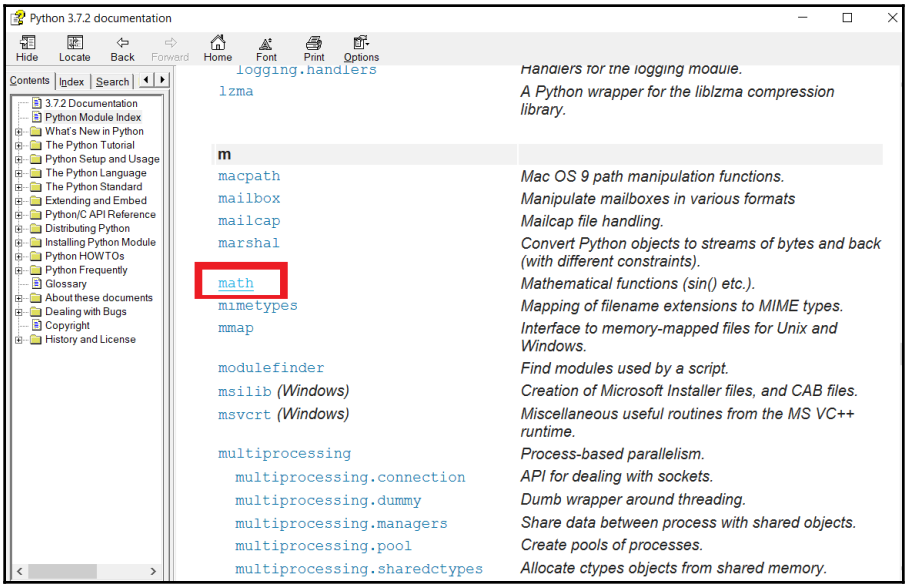
1. Open your IDLE and press *F1* to open the documentation. You will see the following window:



2. Now, click on **modules**. You will see a new window containing a list of modules:



3. Search for the **math** module from that tab or simply press *M* on your keyboard if you want to navigate through the list of modules that starts with the letter `m`:



There are so many methods to use! Don't get overwhelmed with the term methods; we have dedicated a section to object-oriented programming where we will learn how to create our own methods. Right now, just think of a method as operations we use to create expressions. The methods that are provided by the `math` module are also going to perform simple arithmetic operations and many other complex ones. If you want to get a square root, we don't have specific operators to do that, nor can we perform complex mathematical operations; instead, you have to use a math module. We'll look at square roots in the following example.

To get the square root of a number, we can use the `sqrt` method. Check out the documentation for the `sqrt` method to find out more about it and learn how to call it. It's super easy! First of all, we write `math`, then a period (`.`), which signifies that we want to use something from the math module and use the `sqrt` method:

```
>>> import math
>>> math.sqrt(49)
```

The square root of `49` is 7. Our interpreter prints 7.0 as `sqrt` performs a floating-point operation.

If you didn't import the `math` module and instead called `sqrt` directly, you will receive the following error:

```
>>> sqrt(49)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    sqrt(49)
NameError: name 'sqrt' is not defined
>>>
```

As you may recall when we discussed the `print()` function, we didn't call it using any module because it was an in-built function. However, this `sqrt()` function is not in-built. It is from an in-built library of Python. Although we don't have to install it like any other third-party modules, we have to import it before using any of the features provided by it. All of the modules that are provided by Python are in lowercase.

We can call a range of functions and constants from math modules. This allows us to do numerous operations that support complex mathematical computations. If you want to print the value of PI, you can do so with the `math` module, like so:

1. Firstly, we import it with `>>> import math`.
2. Then, we use `module_name` and provide a period (`.`) to specify we want to use that module and the type operations we want to perform, for example, `>>> math.pi`.

You can perform algebraic, logarithmic, trigonometric, hyperbolic, and a wide range of other operations with math functions. However, this module cannot perform math operations for complex numbers, for example, `z = a + ib`.

For those types of complex numbers, we have to import the `cmath` module. Importing and working with this module is also similar to that of the `math` module.

If you want to use the functions provided by the `math` module with calls to `print()` or `input()` without putting a dot, you can use the following command:

```
>>> from math import *
```

In the preceding command, `*` implies that you want to import everything. It is canonically saying *From the math module, import everything*. Now, if you want to call any functions from the math module, you can call it directly, similar to what we do with the input and print functions:

```
>>> factorial(4)
```

The preceding function will be executed perfectly and give us a result of 24.

You may be wondering why the concept of modules wasn't explained at the beginning of this book. It's simple! We just learned about operators, operations, and expressions, which means it's easy to relate to the math module. Every function we call from the `math` module contains operators, operands, and expressions, but its implementation is hidden from our eyes. For example, we simply use the `sqrt` function to perform a square root operation, but we don't know how the square root is done with expressions and logic. We will learn about this in the upcoming chapters when we cover flow controls and functions. Hence, modules provide us with a way to perform high-level operations without having to know how they work. However, if you want to make your own libraries and modules, then the upcoming chapters will help you.

If you want to learn more about modules and functions, you can simply use the `help` command. The Python `help` command will give you a complete list of documentation for built-in functions, modules, and keywords, as in the following example:

```
>>> help([object])
>>> help(input)
Help on built-in function input in module builtins:
```

```
input(prompt=None, /)
    Read a string from standard input. The trailing newline is stripped.

    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.

    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise
    EOFError.

    On *nix systems, readline is used if available.
```

That's enough talk about values and types. Now, let's look at how we can make our code more readable and reusable, that is, others should be able to read our code easily. We talked about the rules and conventions that should be followed while naming the variable, which also leads to readability. There are two ways of making code readable:

- Write notes within the program.
- The Pythonic way is to make a function.

We can add notes to the program via commenting, which will be covered in the next section.

Writing comments in code

Even if you are making a normal piece of software, it has to interact with data in one way or another. Eventually, your code will become lengthier and complicated and becomes hard to manage, read, and understand. Although we will eventually understand the code we've written, it will be harder in the long run. If you have 50,000 lines of code and want to debug the semantic and logic errors in it, it would be hard for you to search and index them. Hence, comments come in handy. Comments are a way of writing notes along with your code so that anyone who tries to read your code knows what that program is doing. Comments are not interpreted by Python, which means whenever the Python parser sees that the statement starts with a hash symbol (`#`), its execution will be skipped.

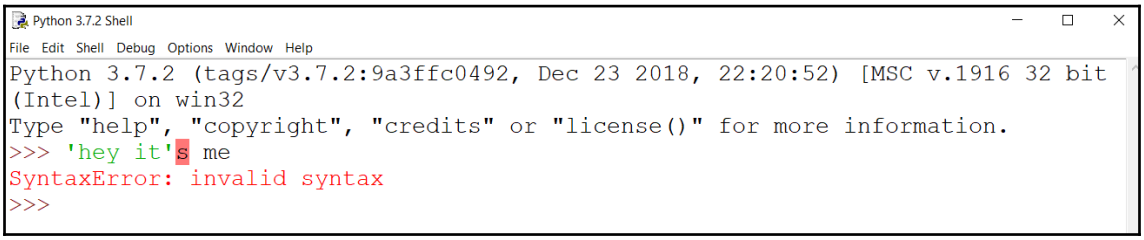
Python design patterns can be convoluted, which makes it difficult for any naive programmer to look at the code and understand what it is doing. Hence, we add simple notes about the program in our native language that explains why we are writing a particular piece of code. Comments that start with `#` are single-line comments. If you write something below the line containing hash, it won't be considered a comment. This is shown in the following code:

```
>>> # this is single line comment
>>> but this is not comment
```

In Python, there are no multi-line comments. People usually think that triple double quotes (`""" """`) are used for multi-line comments, but that's not true. Using hashes is the only way of commenting in Python. In Python 3.x, a string inside triple quotes is considered a regular string. You can use triple double quotes to remove the broken string. Strings are considered to be broken when the scope of the string is not totally enclosed, as in the example:

```
>>> 'Hey it's me'
```

The preceding string was created with a single quote. An apostrophe was used in the string, which creates confusion for the interpreter as it thinks `hey it` is a string and it ignores `s me`. This is a broken string. Not every piece of text you encounter will be in a string. If you run this code in IDLE, you will get the following syntax error:



To eradicate this error, you can use a triple quote. A triple quote will remove the broken string, even if a double quote or single quote appears on your string:

```
>>> """ Hey! it's me """
>>> """ He said, "How may I help you" """
```

Many people think that the preceding line of code represents a multi-line comment, and do something like this:

```
>>> """ this
          is
          multi
          line
          comment """
' this\n          is\n          multi\n          line\n          comment '
>>>
```

You can clearly see that, instead of ignoring to execute that command, it has reflected our command by creating a string for us. If we do not assign a value enclosed in triple double quotes to the variable, it is treated as a garbage collector and gives us a string. Many people confuse it as a multi-line comment because of its behavior as a docstring. Docstrings are strings that are placed at the top of functions, modules, or classes. For example, this is the function that performs the add operation:

```
def add:
```

Obviously, we haven't learned how to create functions yet, but you can get the idea that a triple-double quote is used to provide some information about functions, classes, and modules. Hence, some people think of it as a multi-line comment. You can tell it is not a multi-line comment because the notes inside the triple quote can be accessed with a special function of Python.

Since this is docstring, we can access it through `obj.__doc__`. Since it can be accessed by a method and it is not ignored by the interpreter, it cannot be considered a multi-line comment. Due to this, we can conclude that there are only single-line comments in Python. If we do want multi-line comments, it should be done using triple double quotes, but we have to make sure that we place them above the definition of the function, class, or module.

In the following code, `\n` represents a new line. This will cause a line break in the code. As we can see, the following code prints `hey` in the first line and `it's me` on the next line:

```
>>> print("hey \n it's me")
hey
it's me
```

From this, we can conclude the following about comments:

- Comments are redundant. They simply tell us what every line of code is doing:

  ```
  >>> print(customer_info)  # printing customer information
  ```

- Comments may contain useful information about the code – even some critical information that we cannot extract by looking at the code:

  ```
  >>> d = (400, 200) # d is for display of game console 400*200
  >>> TEMP = 23 # temperature is in Celsius
  ```

As we discussed in the previous chapter, we have to follow a convenient pattern while creating programs. Although this is not mandatory, it is always good practice. In the *Building blocks of Python* section, the first block was requesting user input, which will be our next topic of discussion.

Requesting user input

One of the building blocks of programming is to make the user input data with their keyboard. Any application, whether it be for management tools or games, all should take input from the user. In a user management application, we gather user information such as their name, address, and age, and insert it into a database. In games, we take user input from the keyboard to make movements. Based on the key that's pressed by the user, we can make our character perform some actions. For example, pressing the *Shift* key on the keyboard will make the character jump. Thus, every application has to be user-friendly, which means it has to make the user interact with the application.

Letting a user input something on their keyboard and storing it in a variable so that we can process it further when required is a common practice. Python has in-built functions to get input from users, which means you don't have to import or install anything to use this function. The `input()` function is used to take input from a user:

```
>>> input()
```

When you enter the preceding command, it will give you a place to write something. The interpreter holds its other execution until the user presses a button on their keyboard and presses *Enter*. On pressing the *Enter* key, the program resumes and gives us the text input of the user.

The following screenshot shows how the `input()` function works in Python:

```
>>> input()
I love Python
'I love Python'
>>>
```

In the preceding screenshot, we used the `input()` method and entered the string '**I love Python**'. The black text color is input from the user, and the interpreter instantly gave us some output, which was the same input string from user. You can store input text into variables so that we can perform computations on it:

```
>>> message = input()
```

Now, we have seen how to input data from the user. It is always good practice to provide a message or prompt to the user telling them what they need to enter in that field. A message or prompt should be given as a string within the parentheses of the `input` method, like so:

```
>>>user_name = input(" \n Enter your name? : \n")
Enter your name? :
```

```
John Doe #this is input from user
'John Doe' #printing content of user_name
```

In the preceding example, when the user inputs something and hits *Enter*, our program takes the input from the user and does the specified task. But if you want to make an application where you want to take data from the user continuously, we have to make use of loops. We will study loops in the upcoming chapters:

```
>>> while True:
        input("Enter user_names: \n")
```

The preceding statement takes input from the user continuously. It doesn't stop, even after pressing *Enter* or typing in the `return` keyword. In the preceding command, `while` is used for looping. `True` is a Boolean type that represents the truth value of the logic and Boolean algebra. Boolean types are either `True` or `False`. Hence, the `while True` statement implies that the code inside it should run infinitely, which asks the user to make input infinitely. The result of this is as follows:

```
>>> while True:
        input("Enter user_names: \n")


Enter user_names:
John
'John'
Enter user_names:
Sara
'Sara'
Enter user_names:
Racheal
'Racheal'
Enter user_names:
```

Anything you enter on your keyboard while calling `input()` method will be in string form, even you input it as integers, as in the example:

```
>>> a = input()
1 #store integer 1 to the variable a
```

If you check the type of the `a` variable by using the `type` method, that is, `>>> type(a)`, you will see some unexpected results. We input `1` from the user and stored it in a variable, `a`. When we check the type of value that's stored in the `a` variable, it won't be an integer. Instead, it will show `str class: <class 'str'>`, which means anything that you enter on your keyboard by calling the `input()` method will be of the string type. But sometimes, it may be the case that we want the integer that was input by the user to remain an integer. In such a case, we have to perform typecasting, which will be covered in the next section.

Typecasting or type conversion

There may be times where you want to use the input data of a user as an integer. We saw that the input data from a user will be a string, even if it is an integer, as in the example:

```
>>> age = input("Enter your age? \n")
>>> Enter your age?
29
>>> type(age)
<class 'str'>
>>> age
'29'
```

Age is represented in terms of numbers. However, in the preceding code, it's a string. Hence, we have to convert it into an integer so that information entered by the user will be meaningful for computation. This type of conversion is called typecasting. However, if you do some computation in this value without casting it to the appropriate type, your result will be undesirable. For example, if you want to change the value of age by adding 2 to 29, you cannot change it from 29 to 31. This is because strings do not support increments; instead, they support concatenation:

```
>>> age
'29'
>>> age + 2
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    age + 2
TypeError: can only concatenate str (not "int") to str
```

Hence, if you want to use the age that was entered as an integer, we have to use typecasting methods. These methods are also in-built functions of Python. Some of them are as follows:

- `int(arg1, base)`: This method converts any other data type into an integer. If you put a string inside the parentheses of the `int` function, it will convert it into an integer. `arg1` is the string to be converted and the base argument indicates the base of the data is a string:

```
>>> a = int("10101", 2)
>>> a
21 #conversion from string to integer

>>> b = int("255")
>>>b
255
```

- `float()`: This method converts any integer into a floating-point number, as in the example:

```
>>> float(3)
3.0 #this is floating point number
```

- `str()`: This method converts any other data types into a string, as in the example:

```
>>> str(255)
'255'
```

- `ord()`: This method converts a character type into integer and gives back its ASCII value, as in this example:

```
>>> ord('a')
97  #ASCII value of a is 97
```

Other functions such as `tuple()`, `list()`, `set()`, and `dict()` will be covered in the upcoming chapters.

Now that you are familiar with the first building block of Python, that is, inputting data from the user, let's see how we can format this data using different features provided by Python. In the next section, we will look at string operations that will, in turn, call different methods provided by Python to manipulate the input entered by the user.

String operations

Any data type, be it text, an integer, or a Boolean, written either in double quotes (`" "`) or single quotes (`' '`) is considered a string by Python. String values uncover the broad meaning of data. Data that's stored as strings can be easily accessed but cannot be changed. Hence, it is considered as immutable data types. Let's take a look at the following code:

```
>>> msg = "happy birthday"
>>> msg.upper() # upper() is inbuilt method of string class that converts
string to upper case
'HAPPY BIRTHDAY'
>>> msg
'happy birthday'
```

In the preceding code, we created the `msg` variable and stored a string in it. We used the built-in method of the `string` class to manipulate that string and when we printed the `msg` variable back, it was unchanged. This implies that strings are immutable data types. If you want to change the content of a string, you should completely overwrite it, as in this example:

```
>>> msg = msg.upper()
>>> msg
'HAPPY BIRTHDAY'
```

Strings do not support item assignment. If you want to add an item to the string, you have to make a completely new string. Hence, this feature of Python makes it immutable, as in this example:

```
>>> str1 = "John"
>>> str1[0] = "Hello"
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    str1[0] = "Hello"
TypeError: 'str' object does not support item assignment
```

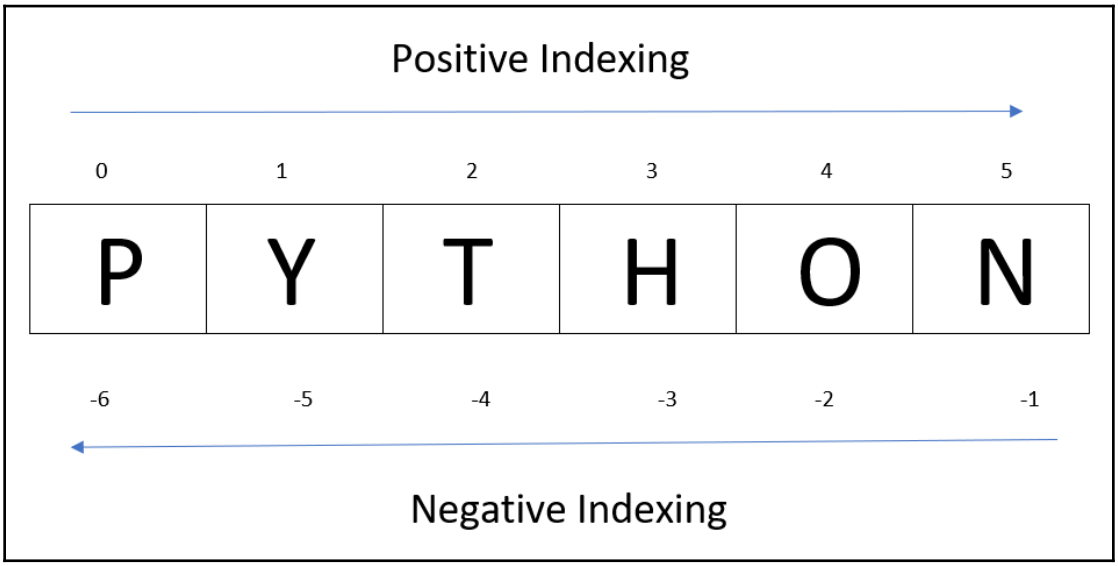
To use the built-in functions of a string, you have to call a method on the string. Let's look at a pattern we can use in the in-built method, that is, `"String".method_name()`:

```
>>> "Python".capitalize() #capitalize first letter of string
Python
>>> "xyz".join("pqr") #joins every letter of string "pqr" with xyz except
for first and last letter
'pxyzqxyzr'

#len function does not have to call like this, call simply len() with
string passed inside parenthesis
>>> len("Python") #prints length of string
6
```

You can access every element of a string by using square brackets. We should put the position inside the square brackets. These positions are called indexes in Python. The index of a string starts from 0 and increases by 1 from left to right:

```
>>> info = "Python"
>>> info[2]
t
>>> info[0]
P
```

You can observe the indexing pattern in the following diagram. Here, we have a `Python` string. The index of the string starts from 0. For each element right next to it that has an index, a unit is incremented to that of the previous element. This is called positive indexing:



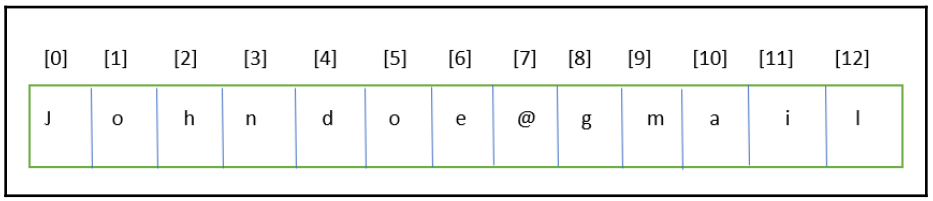
Strings also support negative indexing. If you want the last digits from a string, you can give a `-1` index, as follows:

```
>>> info = "Python"
>>> info[-1]
n
>>> info[-3]
h
```

Now, we have learned how to extract the particular elements of a string based on indexing. But if you want to extract more than one element from a string, you can use string slicing operation. The slicing operation is the same as a pizza slice, which represents we are taking out some parts of the string in a sequential order. String slicing can be done with the same square brackets that we used for extracting a single character from a string. The difference between these two operations is seen when we extend our square brackets with a colon and provide start, end (exclusive), and step indexes to it. Although the theory of string slicing may seem complicated, it is easy to program. Let's take a look at an example to clarify this:

```
>>> email = "johndoe@gmail.com"
```

Suppose we want to extract the name of a person from this email address. We have to track all of the indexes to do so:



Since we are slicing some parts of that string, we have to imagine it as a container where each character resides with its index so that referencing them would be easier. To achieve string slicing, follow these steps:

1. Use `name_of_string[start: stop`: We use the `[step]]` command for string slicing. Here, `start` is the starting index and `stop` is an exclusive position, which means if you put an index on it, the element of - 1 will be included but the element at the stop index will be excluded. Here, step is optional. We will talk about the step index position in an upcoming chapter (Chapter 3: Flow Controls: Build Decision Maker For Your Game)
2. Decide what needs to be extracted first. You cannot extract any part of a string randomly. It should be done sequentially. For example, you cannot extract `jo` and `mail` with a single command. We can extract `johndoe` because every element is in a sequential manner. Let's try to extract it from our code:

```
  >>> email = "johndoe@gmail.com"
  >>> email[0:7:] # 0 is starting position, 7 is stopping position and it
is not included
  'johndoe'
  >>> email[:7:] #empty starting position also means start from 0 index
  'johndoe'
```

In the preceding code, `email[0:7:]` or `email[:7:]` tells us that the first index, 0, is starting an index of a string, which means we want to print from `start`. Instead of 0, you can also put nothing, which represents the default state, and `start` will print from the start. The second index, 7, is the stopping position, but it is an exclusion position which means the interpreter will print until the `e` character but not `@` because `@` is at position 7. Finally, the third index position is for step. We put an empty space here to represent the value it should hold by default, which means we are printing without skipping any numbers. If you put step as `>>> email[0:7:2]`, you will get `jhde` as the output; it will skip one character between each of them.

We can also perform addition and multiplication operations with strings. Adding two strings together is called concatenation. We make use of operators such as + and * to perform string operations, as in this example:

```
>>> "-" * 50 #this will create 50 hyphen or dashes (-)
'--------------------------------------------------'

>>> "a" * 4
'aaaa'
```

However, you cannot multiply two string types. One must be a string and the other must be an integer if we wish to perform multiplication operations with strings:

```
>>> "a" * "b"
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    "a" * "b"
TypeError: can't multiply sequence by non-int of type 'str'
```

If you also want to add strings, both of the operands must be strings. Otherwise, it will throw a type error:

```
>>> str1 = "Happy"
>>> str2 = "Birthday"
>>> str3 = "John"
>>> str1 + str2 + str3
'HappyBirthdayJohn'

>>> str1 + 45 # YOU CANNOT ADD STRING AND INTEGER
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    str1 + 45
TypeError: can only concatenate str (not "int") to str
```

Now that we have learned about the fundamentals of string operations, such as assignment, concatenation, and assignment, we will learn about string formatting. This is an important concept if we need to change the format of the text based on the input.

String formatting

String formatting is where we build our string by replacing placeholders with the content of variables. We apply % (the modulus operator) to perform string formatting. If you want to specify a digit as a placeholder, `%d` is used. If it is string, `%s` is used as the placeholder. The result of string formatting is also a string. Let's look at a small example:

```
>>> key = "love"
>>> value = 13

#lets use string formatting technique
>>> print(" I %s number %d"%(key,value))
'I love number 13'
```

In the preceding code, the position of `%s` was replaced by the value of the key variable and the position of `%d` was replaced by the value of the `value` variable. Hence, `%d` and `%s` are placeholders. You cannot assign a string value in place of `%d` and cannot assign an integer value in `%d`.

The number of values that are passed must match the number of format sequences used in a string. Otherwise, it will throw a type error, like so:

```
>>> '%s %d %s'%("Hello",1)
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    '%s %d %s'%("Hello",1)
TypeError: not enough arguments for format string
```

You can also format your string using Python's built-in format function. It is relatively easier to format using this function. Instead of using placeholders or format sequences such as `%d` and `%s`, we can use curly braces `{}` as placeholders. We can also assign numbers inside curly braces so as to format with a particular value, like so:

```
>>> print(" I love {}".format("Python"))
'I love Python'

>>> print(" I love {0} and I hate {1}".format("Python", "Java"))
'I love Python and I hate Java'

>>> print(" I love {1} and I hate {0}".format("Python","Java"))
'I love Java and I hate Python'
```

Now that we are familiar with the core programming paradigms of Python, let's hop over to the next section, where we will learn to make our first game: **tic-tac-toe**.

Building your first game – tic-tac-toe

The Python language is a cross-platform language, which means we can make games for any device. However, here, we will focus more on the logic and its implementation rather than coding for a specific platform. Coding games with Python is simple compared to other languages as its syntax is shorter and it provides rich-content libraries that make production faster. With that being said, it isn't that easy if you don't make plans before coding. We have to break our game entity into parts so that each entity can be debugged easily. We will follow these general steps while making games from now on:

- Brainstorming and information gathering
- Choosing a proper code editor
- Programming model
- User interaction—user input/manipulation

So far, we have covered a variety of topics, including variables, operators, expressions, taking input from a user, and printing it to a user. Let's apply all of these techniques now to make our first game.

Brainstorming and information gathering

Before we start coding, let's think about the design and interface of our game. Pull out your pen and paper and start thinking about the interface of the game! Did we learn anything about the GUI so far? Obviously not! That means we have to make use of a simple interface for our first game. We will modify it later, after we learn about some advance concepts of Python. Tic-tac-toe is a game that takes input from a user and places either X or O based on the player's movement. Hence, our interface should be a placeholder for these symbols. We will make a simple interface containing _ for now. An underscore (_) will be our placeholder where we will put either X or O based on player selection:

```
_ | _ | _
_ | _ | _
_ | _ | _
```

The preceding code shows the simple layout of our game. It contains _ (underscores) as placeholders and | to separate the symbols:

```
_ | _ | O
_ | X | _
_ | _ | X
```

As you can see, whenever the player takes a step, we replace that underscore with a symbol corresponding to that user's decision. Now, we have a basic interface for our game.

Now that we have planned the interface, we need to work out how to track the position of the underscore and how to find out where to replace the underscores with the appropriate symbols. We can assign numbers to each of these underscores and tell the user to choose a number. Then, based on that number, we can assign its symbol to that location, like so:

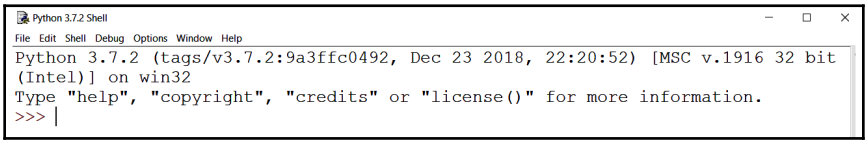
```
0 | 1 | 2
3 | 4 | 5
6 | 7 | 8
```

Now, we have gathered enough information to start our simple game. In complex real-world games, the brainstorming and information gathering process would take around 6 months. Now, let's look at choosing a code editor.

Choosing proper code editor

We have already installed Python On our machine, and we took a look at the pre-installed editor of Python, IDLE. We will use that editor for this project. Let's get started:

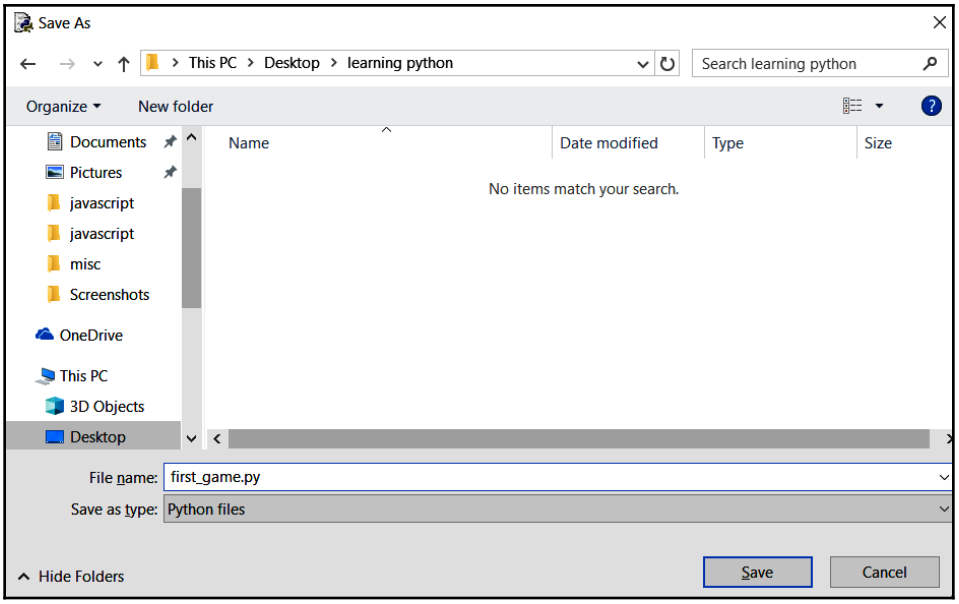
1. Search for IDLE in your search bar and open it. You will get the following Shell:



   This Terminal or Shell is normally used to interpret commands instantly within the Shell. This means that one command will execute at a time, but we have to write many lines of code to make our game. Hence, writing a game with this Shell is not possible. We have to create a file where we can write many lines of code and execute them all at once. Python provides us with scripts to facilitate this problem.

2. Click on **File** and then **New File**, or press *Ctrl* + *N*. A new script file will open where we can write multiple lines of code.

3. At the top of the window, we will see **Untitled**, which means we haven't saved our file yet. Let's save it first because we have to save it anyway afterward. Press *Ctrl* + *S* to save it. I have saved it as `first_game.py`.



Now that we have selected the proper IDE for development, let's start developing our model for the game.

Programming model or modelling

In programming, a model is a way to represent the flow of data in your program. In our game, it is about how to use data that's been obtained as user input. We uncovered some information in the *Brainstorming and information gathering* section, where we talked about positions and how each number was assigned to the position that represents player selection. The model does not contain presentation logic; instead, it will deal with data logic. The computer doesn't care about layouts or interfaces. The user, on the other hand, requires an interface in order to react. Thus, every program has a frontend and a backend. The frontend is everything that you see in the application, whether it be an aesthetic or visible part of the application. **User experience** (**UX**) designers work mostly on frontends in big projects. The backend doesn't care about designs—it only cares about the algorithms and security that are applied to the data layer for the transaction of data. Models are used as a way of communication between the frontend and the backend.

The computer does not care how the model presents data, but the user should get data out of the model in an informative and pretty way. Due to this, we made simple layouts that look as follows:

```
_ | _ | _
_ | _ | _
_ | _ | _
```

Let's start creating our model for the presentation layer:

```
#this code is written as scripts
game_board = ['_'] * 9 #this will create 9 underscores
print(game_board[0] + '|' + game_board[1] + '|' + game_board[2])
print(game_board[3] + '|' + game_board[4] + '|' + game_board[5])
print(game_board[6] + '|' + game_board[7] + '|' + game_board[8])
```

The preceding code represents the layout for our game. It is displayed to the user. Let's break it down line by line:

- `game_board = ['_'] * 9`: This statement creates 9 underscores, which is the placeholder for our game characters. It is stored in the `game_board` variable. As you may recall, a variable cannot store multiple values. If we perform multiple assignments to the same variable, the variable will store the latest value that was added to it. Hence, this board is not a simple type of variable. This is a `list` variable. We can store multiple pieces of data in a list. Let's print the value of the board:

```
>>> board = ['_'] * 9
>>> board
['_', '_', '_', '_', '_', '_', '_', '_', '_'] # 9 underscores is
  stored in board list
```

- `>>> print(game_board[0] + '|' + game_board[1] + '|' + game_board[2])`: The preceding command prints the first line of the layout. We have learned about the `print` statement earlier in this chapter. Anything inside parentheses (either a string or variable value) is printed as it is by the `print` statement . We passed `board[0]` to get the first element of the board, which is the first underscore (_) We print a separator (`|`) between each underscore. The output of the preceding statement is `_ | _ | _` .
- We have to print the preceding layouts two more times, which means we have to use two more `print` statements:

```
>>> print(game_board[3] + '|' + game_board[4] + '|' + game_board[5])
>>> print(game_board[6] + '|' + game_board[7] + '|' + game_board[8])
```

- The number that's inserted in the square brackets is the position that we normally call an `index` in programming. This refers to a certain position of the `list` variable. The list index always starts with zero indexes:

```
>>> board = [1,2,3,4,5,6]
>>> board[0] # this will give value 1 from "board" list
>>> board[5] # this will give value 6 from "board" list
```

- The following code shows the final layout for our tic-tac-toe game. Make sure you write the program as a script and press *F5* to run it:

```
game_board = ['_'] * 9
print(game_board[0] + '|' + game_board[1] + '|' + game_board[2])
print(game_board[3] + '|' + game_board[4] + '|' + game_board[5])
print(game_board[6] + '|' + game_board[7] + '|' + game_board[8])

#output
"""
    _ | _ | _
    _ | _ | _
    _ | _ | _

"""
```

- In the preceding code, we did two things: first, we printed underscore in every position of our layout, and then we assigned a number to each of those positions:

```
0th | 1st | 2nd
3rd | 4th | 5th
6th | 7th | 8th
```

Now that we've developed the programming model that represents the basic layout of our game, it's time to make an interaction between the programming model and player of the game. In the next section, we will learn how to take user input and manipulate it so that we can interact with the model of our game.

User interaction – user input and manipulation

We're making games for our users to play. Hence, we should make an interface so that we can make our application user-friendly. We did this in the previous section. Now, we have to take some input from the user and place it to the layout through the model. We know that a simple way to take the input from the user is by using the `input()` method. Let's use it now.

To do this, we will think of this problem: *what should we input from the user? Is it a symbol, like X/O, or is it positions?*

Taking input as a symbol is useless because after taking it, we should know where to place it. Hence, we can take the positions from the user and place the symbol into our code automatically:

```
#code from models

#.............................................................

#code for user input

while True:
     pos = input(" Enter any position you want from (0-8): \n")
     pos = int(pos)
     game_board[pos] = 'X'
     print(game_board[0] + '|' + game_board[1] + '|' + game_board[2])
     print(game_board[3] + '|' + game_board[4] + '|' + game_board[5])
     print(game_board[6] + '|' + game_board[7] + '|' + game_board[8])
```

Let's break this down part by part:

- `while True`: This will run an infinite amount of times. We saw this happen in the *Requesting user input* section. Therefore, we will take the input data from the user an infinite amount of times, which means our game loop has no termination.
- `pos = input(" Enter any position you want from (0-8): \n")`: This statement will take input from the user as a position from 0 to 8 and store it in the `pos` variable.
- The data that's stored in the `pos` variable will be a string, but the position should be an integer. Due to this, we have to typecast it as an integer using the `int` method. Then, we store the integer in the `pos` variable as `x = int(x)`.
- `game_board[pos] = 'X'`: This statement assigns `X` to the position that's selected by the user. The `pos` variable contains a position from 0 to 8 that was selected by the user in the previous command. Now, we are assigning `X` to that position in place of an underscore, like so:

```
0th | 1st | 2nd
3rd | 4th | 5th
6th | 7th | 8th
```

If the user enters, `4` then we will put `X` in `4th` position, as follows:

```
0th | 1st | 2nd
3rd |  X  | 5th
6th | 7th | 8th
```

- After we assign a player symbol to the specified position, we have to print the board again with those three print statements. It should be kept inside the loop because we have to print the board every time the user enters a new position from the keyboard.

Now that we have finished making models for rendering layouts and user input, we can run the game and observe the output. The game you are going to see won't be appealing because it doesn't have a proper layout and it won't have as many features that our tic-tac-toe game should have. We will try to make the game as playable as possible while learning more Python in the upcoming chapter. For now, we will take a look at the possible errors and warnings that may be encountered in our game.

Possible errors and warnings

We've only covered the basic fundamentals of Python so far, so you won't have found many semantic errors until now. However, you are likely to be accustomed to the syntax error. First and foremost, an error can be caused while naming a variable. If you do not follow the rules or conventions for naming a variable, you are likely to get the following errors:

```
>>> my name = "John Doe"
SyntaxError: invalid syntax
```

The preceding name is invalid because you cannot provide spaces while creating variable names. You can put an underscore between them to specify that it consists of two words. `my_name` is a valid name for a variable.

If you spell your variable name incorrectly, you are going to get an error instantly. Suppose you created a variable called `Msg` and used it as `msg`. An error will be returned, stating that this is the wrong definition. Python is case-sensitive, which means that `True` and `true` are different in Python. If you name a variable `True`, it will be illegal because it is one of the keywords of Python.

However, you can call a variable `true`:

```
>>> True = 45
SyntaxError: can't assign to keyword
>>> true = 45
>>> true
45
```

The same rule goes for naming modules. In this chapter, we looked at how to import the `math` module and use its methods. However, if you spell the module's name wrong, it will cause many problems in the long run. You won't see an instant error on IDLE; you have to compile your script to see it. Thus, debugging is a lot harder with IDLE. Because of this, make sure you spell all of your modules and their methods correctly.

`>>> import math` will successfully import the `math` module into your project, but if you use the wrong module name, you will get the following error:

```
>>> import Math
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    import Math
ModuleNotFoundError: No module named 'Math'
```

There's also another type of error that's more dangerous than a syntax error; these are called semantic errors. A semantic error occurs when we didn't get the intended results. They won't be detected by the interpreter, and so they are hard to debug. We can get semantic errors due to executing expressions incorrectly. If we didn't care enough for the rule of precedence, we will end up making wrong statements for the program.

The output of the `1 + 3**2` expression is 10, not 16. However, we can force our interpreter to make this statement print 16 by enclosing the statement with parenthesis. `(1 + 3) **2` will give us 16.

Now that you've learned how to rectify errors that are encountered in your program, let's learn about the possible ways to modify our very first tic-tac-toe game.

Game testing and possible modifications

There are several ways to find errors in your game. First of all, you can reach out to your friends and make them play your game. The suggestions that you gather the first time you test your game is known as alpha testing and it is an essential part of any game development life cycle. After collecting enough information through interviewing, you can start modifying your game.

The things we have learned so far will not be enough to make our game more appealing. We will learn about several topics in upcoming chapters and modify our tic-tac-toe game accordingly.

The game that we made in this section is bland and does not galvanize our user to play, but we have learned so many things by making it. We looked at the basic process of creating games using the concepts of models and views. View refers to layouts where we render data that helps us to interact with the user through the interface, while model refers to the way we communicate data between our program and user. We haven't covered advanced Python language paradigms yet and so we have limited power. This means that the game in this chapter is simple. However, we will make changes to this game after we cover conditionals, looping, and functions.

The following are some possible modifications we could make to our game:

- Let's analyze our code and see its limitations. We told the user to explicitly enter a number from 0 to 8 to specify the movements of the user. What if the user didn't input a number and input a string? Our program will terminate the loop and crash with an exception. Hence, the first modification we would make is to restrict the user to entering only numbers. If they enter anything else, we can print a user-friendly message instead of crashing the program. This concept is called *exception handling* and will be covered in the next chapter.
- Currently, this game only works with one player, but tic-tac-toe is a multi-player game. Hence, we have to learn about conditionals and flow controls that will help us to achieve transition between players. We will do this in the next chapter.
- When a user captures an entire row, column, or diagonal, then they should be considered the winner of the game and the game should complete its execution. However, we haven't created any logic to make a player a winner. What we've learned so far is not enough, but after we complete the next chapter, we will be able to make drastic changes to our game.

By looking at the modifications we can make to our game, we can see that we have bigger things to come in the next chapter. Although the knowledge that we acquired in this chapter was enough to create a programming model and allow us to interact with a single player, this was not enough for us to interact with multiple players, which requires a good understanding of looping and conditional structures. We'll cover these concepts in the next chapter.

Summary

In this chapter, we covered the two basic building blocks of Python: inputting and providing formatted output. We looked at Python's built-in data types in this chapter and started by learning about the different data values and their types, such as integer, string, float, Boolean, and none. We took a tour of the Python ecosystem by learning about variables, numbers, and the `math` module. We saw how to use the `math` module and got a good grasp of topics such as the rules and conventions that need to be followed while creating variables and using modules. These topics are essential if you want to start your programming career with Python. These topics not only make for a strong foundation in Python but also teach you what good and bad practices in programming need to be followed and removed, even if you are a proficient Python programmer. Coding is not only writing code—it's about presenting information in a readable and usable way. Hence, we saw how we can use comments in programming to make our code more readable and reusable for other programmers.

Making the user input data and then using it in our program is the only way to make an application user-friendly. Hence, we learned how to make the user input data and store it in the structure so that accessing it will be easier for further manipulation. We eventually looked at the unusual working behavior of the `input()` method, which converts our integer or Boolean input data into strings. Due to this, we learned about typecasting methods and we saw how easy it was to perform data conversion with Python's built-in methods.

A string is the most fundamental and primitive data type and stores text. We dedicated an entire section to the creation and manipulation of strings. We learned how to access elements of strings. We also learned that string assignment isn't possible, and so we concluded that strings are immutable. We learned about the basic methods of the string class such as capitalize, join, upper, lower, and len. We looked at two formatting techniques for strings, that is, `%s` and `%d`, which are used as placeholders and format the method. You can use either of them, although it's better to have knowledge of each before you do so.

Then, we built our first game. We saw the building games is not only about coding. We need to go through a variety of processes, such as brainstorming, modeling, and user interaction. We learned how model and view work together. Then, we made a simple game and had the chance to revise everything we'd learned so far. Finally, we suggested some modifications that we could make to that tic-tac-toe game. Every modification will be covered as we progress through this book. In the next chapter, we will learn about flow control and how to build a decision-maker for our game.

3
Flow Control - Building a Decision Maker For Your Game

One of the greatest blessings of Python is automation. When we talk about automation, there is no staggering logic; it's all about the power of conditionals and branching. They control sequencing when it comes to the execution of programs. Any program at its rudimentary stage is made with a simulation. Whenever we deploy such programs in a real-world environment, we are overwhelmed by various noises and unexpected behaviors. To preclude such behavior, conditionals play a major role. Flow controls decide how to execute a specific part of a program based on the Boolean logic that's present. We covered topics such as statements and operators in the previous chapter, both of which are useful when it comes to creating Boolean logic. Such statements are used to perform arithmetic computation. In this chapter, we will see how to manipulate such statements, which will result in true or false Boolean logic.

Mid-way through this chapter, we will learn about looping, an important technique that will make us competent enough to make code shorter and more powerful. This chapter will be a package that's complete with core programming, conditionals, and recursive programming. We will refine the tic-tac-toe game we made in the previous chapter by incorporating Boolean logic and flow controls.

The following topics will be covered in this chapter:

- Boolean logic and logical operators
- Conditionals
- Iteration
- `for` and `while` loops
- Making a game controller for our tic-tac-toe game

Technical requirements

You will need the following requirements to be able to complete this chapter:

- Python script and IDLE
- The code assets for this chapter, which can be found at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter03`

Check out the following video to see the code in action:

`http://bit.ly/2pvpBas`

Understanding Boolean logic and logical operators

There won't be a day that goes by without us stating that a Boolean type is either `True` or `False`. We use these keywords to make logic that determines whether we are going to execute a certain portion of code. Let's talk about the bool type in terms of a real-life scenario. If we are hungry, we eat something. If we are tired, we rest. Let's convert these scenarios into the appropriate Boolean statements:

- `is_hungry = True: eat something || is_hungry = False: don't eat`
- `is_tired = True: take rest || is_tired = False: do your work`

You perform these quotidian tasks based on the Boolean logic at hand. Now, let's relate this to programming; you can use two sets of code based on Boolean data types:

- `(True): Do something || (False): Do something`

We use Boolean expressions to make such types of logic. We look at how to create expressions in the previous chapter. Combining a variable and an operator will give us a simple form of expression, as in this example:

```
>>> y
>>> x = y + 6 * 7
```

Boolean expressions, however, are a bit different. Instead of giving a result as an integer, they provide an outcome of either `True` or `False`. The simplest form of a Boolean expression can be made with a double equal to operator (==). Don't confuse it with a single equal sign (=). This is used for assignment, while a double equal sign (==) is used to check whether they are equal, as in the example:

```
>>> 5 == 5
True
>>> "Python" == "Java"
False
```

If you compare the data of two different types, the result is always `False`:

```
>>> "5" == 5 # String(5) not equal to int(5)
```

You can always type-caste it to make your logic `True`:

```
>>> int("5") == 5
True
>>> "5" == str(5)
True
```

To check the type of any Boolean variable, you can use the `type()` method and get the output of `<class 'bool'>`, which implies that `True` or `False` are values of the bool type:

```
>>> logic = False
>>> type(logic)
'<class 'bool'>'
```

Boolean logic can also be used with comparison operators. We will learn how to create statements using **comparison operators** in the next section.

Comparison operators

Any expression that results in either `True` or `False` is a Boolean expression. These Boolean expressions cannot be made without comparison and logical operators. We've already looked at the basic comparison operator (==); however, there are six more we need to learn about (`<`, `>`, `<=`, `>=`, `!=`, and `is`). Let's take a look at them in action:

- **5 < 10**: 5 is less than 10, which results in `True`.
- **5 > 10**: 5 is greater than 10, which results in `False`.
- **10 <= 5**: 10 is less than or equal to 5, which results in `False`. 10 is neither less than nor equal to 5.

- **10 >= 5**: 10 is greater than or equal to 5, which results in `True`. 10 is greater than 5.
- **10 != 10**: 10 is not equal to 10, which results in `False`. 10 is equal to 10.
- **5 is 5**: 5 is the same as 5, so this results in `True`. However, `5` is 5, and so this results in `False`.

You can store the preceding numbers in different variables and try the same Boolean expression on the IDLE to get the following results:

```
>>> v1 = 5
>>> v2 = 10
>>> v1 < v2
True
>>> v1 > v2
False
>>> v2 <= v1
False
>>> v2 >= v1
True
>>> v2 != v2
False
>>> v1 is v2
False
>>> v1 is v1
True
```

In order to make logic that's applicable to the real world, we need operators that can combine different comparison operations at once and provide results instantly. These types of operators are called logical operators. In the next section, we will learn about the different types of logical operators and the ways we can use them.

Logical operators

Operators are widely categorized as arithmetic operators, comparison operators, and logical operators. We've already covered the arithmetic and comparison operators; now, it's high time to cover logical operators.

You can relate logical operators with a logic gate (`and`, `or`, and `not`), which is the basic building block of any digital circuit. They have two inputs, but with certain circuit computations, we only get one output. Circuit processing is done by `and`, `or`, and `not` gates. Similar to the digital circuits of a logic gate, logical operators can have many conditions passed with it, but the output will eventually be either `True` or `False`. Here, `conditions` refer to our Boolean expression, which we make using comparison operators. The working principles of these three elementary logical operators are as follows:

- `and`: Two conditions are attached with a single `and` operator, that is, `condition_one` and `condition_two`. The entire condition will be `True` when each of these conditions is also `True`. If either of the conditions that are attached to the `and` operator is `False`, the result will be `False`. Let's take a look at an example:

```
>>> condition_one = 5 > 2 #True
>>> condition_two = 6 < 10 #True
>>> condition_one and condition_two
True
>>> condition_two = 6 > 10
>>> condition_one and condition_two
False
```

The truth table for the `and` operator, which sets out the functional value as either `True` or `False` based on a combination of Boolean or logical expressions, is as follows:

Condition one	Condition Two	Result
True	True	True
True	False	False
False	True	False
False	False	False

- `or`: The same as the `and` operator—two conditions are attached with a single `or` operator. You can add more `or` operators if you want to add more conditions. In the case of the `or` operator, if both of the conditions that are attached to it are `False`, the result will be `False`; otherwise, it will be `True`. Let's look at an example:

```
>>> 4 < 10 or 5 == 5
True
>>> 4 <= 10 or 100 < 50
True
>>> 10 <= 4 or 100 < 50
False
```

The truth table for the `or` operator, which sets out the functional value to either `True` or `False` based on a combination of Boolean or logical expressions, is as follows:

Condition one	Condition Two	Result
True	True	True
True	False	True
False	True	True
False	False	False

- `not`: This operator inverses the type of the logic. It changes `False` to `True` and vice versa. Hence, it is called known as a logical inverter. It only takes one condition with it, as follows:

```
>>> not (5 < 4) # condition 5 < 4 is False
True
>>> not True
False
```

The truth table for the `not` operator, which sets out the functional value as either `True` or `False` based on a combination of Boolean or logical expressions, is as follows:

Condition	Result
True	False
False	True

You can also represent `True` and `False` with 1 and 0 in Python. Hence, we can conclude that any non-zero integers can be used alone to make a condition with logical operators, as in the example:

```
>>> 1 and 1
1
>>> 1 and 0
0
>>> 1 or 0
1
>>> 49 or True
49
```

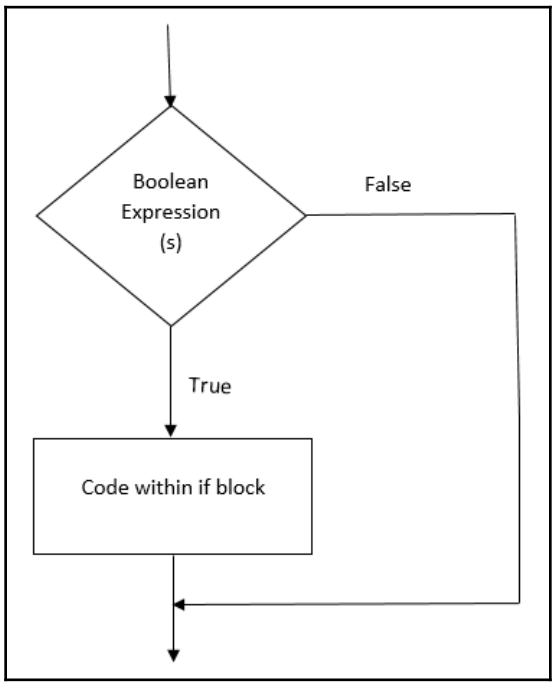
Learning about the different types of operators was quite fun, but now we are going to hop over to the section, where you will learn how to use these conditions (made by comparison and logical operators) to make several decisions. **Conditionals** are highly practical in any real-world scenario. I'm excited to learn about them—are you?

Conditionals

So far, we've learned about making conditions with comparison and logical operators. Now, we'll talk about how we can evaluate these conditions. Conditionals are tools that come in handy when we want to compute the result of those conditions and control the flow of the program accordingly. As we already know, the results of these conditions are going to be either `True` or `False`. So, based on the type of bool we use, the conditionals are going to execute some part of the code. We use `if` statements in Python to perform conditionals. After writing the `if` keyword, we put conditions next to it. The condition can be singular or a combination of many with logical operators. We end an `if` statement with a colon; subsequent statements are indented property. Take a look at the following example:

```
#filename: conditionals.py

if (True):
  #Do something
```

The following figure represents a boolean logic for implementing conditional statements:



Take note of the following while using Python:

- **Colon (`:`)**: If you want to declare scope in Python, inside where you can write more than one statement, you need to use a colon (`:`) to specify it. Most of the programming language uses curly braces (`{ }`) for this, but Python is strange when it comes to defining scope and the extent of block statements for features such as functions, `if` statements, classes, and loops. However, once you get familiar with using this, you will find it amusing and be able to distinguish code written in Python from any other language within a second.
- **Indentation** (**whitespaces**): After we define the scope with a colon, we can enter its scope. Any subsequent statements that are written within its scope should start with uniform white spaces, which we call indents. You can press the *Tab* key to give each statement a uniform indentation. Most of the errors that beginners make are due to improper indentation. If you don't provide the proper indentation, you will get the following warning from the Python interpreter:

```
>>> if (True):
print("Hello world")
SyntaxError: expected an indented block
>>>
```

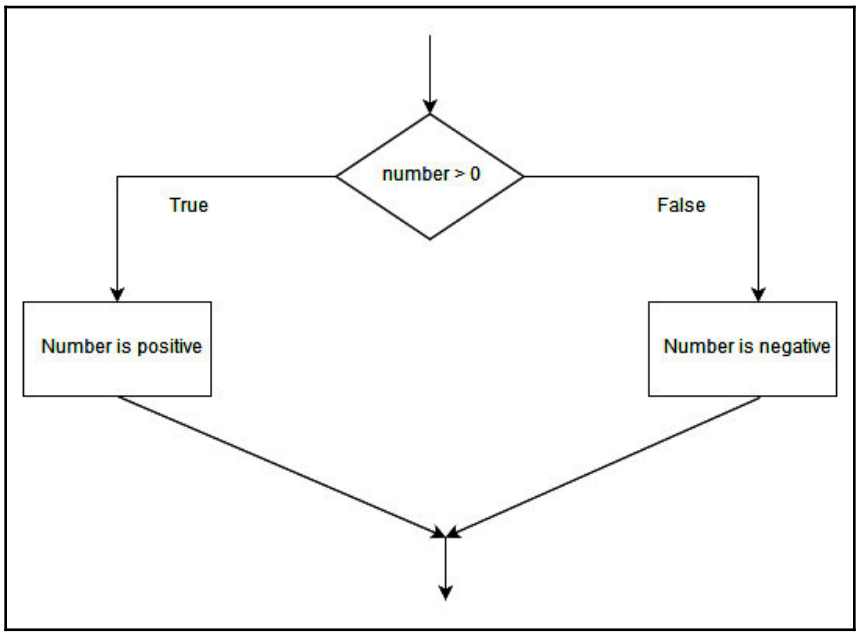
If statements evaluate logical statements. Whenever that statement is true, its indented statement will be executed; otherwise, it will be skipped. You can also add the `pass` keyword to tell the interpreter not to execute anything inside the indented block, as in this example:

```
>>> if ( 4 == 4):
          pass
>>> #does not print anything
```

As we already know, Boolean statements will either will result in `True` or `False`. Indented code inside an `if` statement will be executed if the condition is `True`, but if the condition is `False`, the indented code inside the `else` part will be executed. Let's take a look at an example:

```
>>> number = 1
>>> if number > 0:
        print("Number is positive")
    else:
        print("Number is negative")
Number is positive
>>>
```

Following figure represents the flowchart for implementing program to check whether number is positive or negative using conditional statements:



You can see that we have created two branches of conditions for the `True` or `False` logic. Based on the result of the Boolean logic, flow control is transferred to either side of a program. Hence, conditionals are also called *branching*.

Although our code is able to execute the code with two branches, there is a little gap in our code. If the number variable contains zero, it is neither positive or negative. Therefore, we have to add one more condition to this conditional. Whenever we need more than two branches for the computation of logic, we can make chained conditionals. We can add as many conditions as we like with a chained sequence. To perform chained conditionals with any other programming language, we use the `else if` command. Python improvises by making different commands with `elif`. Let's take a look at an example:

```
>>> number = input("Enter any number: ")
>>> number = int(number) #converting string to integer
>>> if number > 0:
        print("Number is Positive")
    elif number == 0:
        print("Number is Zero")
    else:
        print("Number is Negative")
```

```
Enter any number: 0
Number is Zero
>>>
```

We can put any number of conditionals within one conditional statement. We call these nested conditionals. Let's take a look at an example:

```
>>> number = 10
>>> if number > 0:
        if number % 2 == 0:
                print("Number is positive and even")
        else:
                print("Number is positive and odd")
Number is positive and even
```

In the preceding example, the outer conditional contains two sub-branch conditions where, in the first branch, we check for an even number. The next default condition is checked for an odd number. We use a simple single statement to make a condition in this example, but conditions in nested conditionals can be made complex with logical operators, as in this example:

```
>>> number = 4
>>> if number > 0:
        if number % 2 == 0 and number < 10:
                print("Number {} is small even & positive
number".format(number))
Number 4 is small even & positive number
```

Now that you know how to make decisions with several conditional statements, we will take a look at a highly practical topic known as **iteration**. This allows us to execute a sequence of instructions. This is repeated until a certain condition is reached.

Iteration

Let's say you want to write a program where you have to print your name 100 times. What we have learned so far dictates that the easiest way to do this is to use the print statement 100 times. But what if you want to print your name 10,000 times? Writing a print statement for 2/3 pages continuously is not good programming. We have to use loops in such a case. Loops will help us to iterate over datasets until a condition is met. In each iteration, a part of the code is executed and we have to update the iterating variable each time. The following is an example of iterating over a variable:

```
>>> i = 0
>>> i = i + 1
```

We update the iterating variable with an increment and decrement unit. Here, we update the value of `i` by adding `1` to it. This is known as incrementing. You can also subtract 1 from it, which is known as decrementing. Each time we execute code inside indented loops, we update the iteration using either increment or decrement statements.

Similarly, there is a comparatively easier and faster way of implementing increment and decrement statements. You can use the following statements to perform multiple operations:

- `+=` adds a number to the variable and changes the variables in its process.
- `-=` subtracts the variable with a value and sets the new value to its resulting variable.
- `*=` multiplies the variable by a value and changes the outcome of the variable.
- `/=` divides the variable with the value and places the result on the resulting variable.

Let's look at an example to see its effect:

```
>>> value = 4
>>> value += 5
>>> print(value)
9
```

The effectiveness of the increment and decrement operators can be seen with looping, where we repeat a set of operations multiple times. Let's take a look at looping in action with for and while loops. We will begin by learning about the `for` loop.

Th for loop

Whenever you want to loop within a dataset, let's say, within a range of numbers, within a certain file, or within some definite word sets, we use a `for` loop. It is also referred to as a definite loop. Until and unless there is certain item left in your bucket of items, it will iterate. The `for` loop is terminated at the end of the bucket. Here, bucket is a metaphor for a list of items, such as a list of numbers, words, or sequences, as in this example:

```
>>> for i in range(10):
         print(i, " John Doe") #range(10) gives [0,1,2,3,4,5,6,7,8,9]
0  John Doe
1  John Doe
2  John Doe
3  John Doe
4  John Doe
5  John Doe
```

```
6  John Doe
7  John Doe
8  John Doe
9  John Doe
```

In the preceding code, the `range()` method is used to create a list of numbers. `range(10)` provides a list of numbers from `0` to `9`. It is stored as `[0,1,2,3,4,5,6,7,8,9]`.

In the first iteration, the `i` value becomes `0`, it executes code within the block of the `for` loop, and changes the value of `i` to the next element of that list automatically, as follows:

```
>>> for i in [6,7,8]:
        print(i)
6
7
8
```

You can also loop within data that contains words or text. The iterating variable will contain a value as a word each time we loop within that list, as in this example:

```
>>> for name in ['Tom','Harry','Ricky','Matt']:
            print(name)
Tom
Harry
Ricky
Matt
```

In the preceding example, the iterating variable is a `name` variable and, every time it iterates through that list, it fetches its value and stores it in `name`. Hence, we can only use the `name` variable inside the body of our `for` loop. No other variable can be used except the iterating variable inside the `for` loop. This is shown in the following code:

```
>>> person_names = ['Tom','Harry','Ricky','Matt']
>>> for name in person_names:
        print(person_names)
Traceback (most recent call last):
   File "<pyshell#26>", line 1, in <module>
     for name in person_name:
NameError: name 'person_name' is not defined
```

In the preceding example, `person_names` is a type of variable where we can store an array of items. This variable is called a list. We will cover lists in the next chapter. Here, the iterating variable is `name`, which is declared with a `for` loop. However, inside the body of a for loop, we didn't use the `name` variable. Instead, we used `person_names`, which gave us `NameError`. Hence, iterating variables can be only used inside the body of a `for` loop.

The next type of loop we will cover is the `while` loop, which will perform operations similar to the `for` loop but with some tweaks. The `while` loop is known to be used in scenarios where we don't care about the terminating point of loops.

While loop

Another form of iteration in Python can be performed using a `while` loop. Let's recall the features of a `for` loop: it's used to iterate over a finite sequence of elements, either as a list of numbers, words, or files. There has to be a termination point if you want to use a `for` loop. We also don't care about the terminating condition while using `for` loops. It's terminated when it reaches to end of the items or sequences. Now, what if we want to terminate the loop based on custom conditions? The `while` loop is the most appropriate loop in such cases. We can make a custom condition where we can terminate recursion with the help of a `while` loop.

Both the `while` and `for` loops are going to perform incessant looping. At each iteration, they are going to execute body of loop. The main difference between the `for` and `while` loop is that the `while` loop has to declare the update statement and terminating condition with its declaration. First of all, we have to make an iterating variable, and then we have to make a terminating condition in order to specify a stopping point for the loop. At each iteration, we have to update the iterating variable, like so:

```
>>> i = 0
>>> while (i < 10):
        print("John Doe")
        i = i + 1
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
```

In the preceding example, we created an iterating variable, `i`, and assigned a value of 0 to it. After that, we made use of a `while` loop. To use this loop, we used the `while` keyword and followed that by its terminating condition. We tell the interpreter we want to run this loop until `i` is less than 10. If `i` is equal to or greater than 10, we want to terminate this loop.

After that, we put a colon to specify the scope for our loop. Then, we added a simple print statement to it, which will be executed each time this loop runs. Finally, we added an *i = i + 1* statement to specify the updating condition. This is going to change the value of `i` into the new one with an increment of one. This is important so that we don't end up using an infinite loop. If you remove your updating condition, the loop is going to run an infinite amount of times and the Python Terminal isn't going to be interactive to a user's response. One way of creating an infinite loop is by using a condition that has no endpoint,as in this example:

```
>>> while True:
         print("Infinite loop")
```

The preceding loop is an infinite loop as there is no endpoint or termination point attached to the `while` keyword. If we are able to change the `True` keyword to `False`, only this loop is going to terminate:

```
>>> condition = True
>>> while condition:
        print("This will run only one time")
        condition = False
This will run only one time
```

In the next section, we'll learn about the looping pattern so that we can find out about how loop works under the hood.

Loop pattern

There may be trade-offs between the `for` and `while` loops, but both work well when we want to loop around a known list of elements or the content of files. We can arrange or sort the elements out of the list or file using these loops. A `for` loop cannot be made to loop an infinite amount of times, but a `while` loop can do so using a condition that is never going to be. The main purpose of looping is to get items from particular files or lists so that we can process them further. We can sort these items based on smallest and largest or important and superfluous while scanning datasets.

The construct of the loop pattern contains the following three pinpoints:

- Making an iterating variable. There can be one or more. They are used to make the conditions that represent the loop's terminating point.

- Some computation is done inside the body of the loop so that we can manipulate the data items that are fetched with the loop one by one. We can also change the value of the iterating variable inside the loop's body, which we normally do in the case of a `while` loop.
- Look for the possible base condition so that the loop can be terminated. Otherwise, it will result in an infinite loop. We have to observe the resulting variable after the loop ends.

If you want to demonstrate the construct and working paradigm of loop patterns, it's always a good idea to use loops with a list of items. In the following example, we are going to make a program where we will take a list of numbers and check for the smallest number in the list.

We can do this in two ways. Python makes programming easy for both types of people: the naive or the professional. They have various ways of implementing the same logic, but the most common is to use Python's built-in methods, such as `min()` and `max()`. These get the smallest and largest number from the list of numbers in Python, respectively:

```
>>> numbers = [113,115,55,66,65,90]
>>> min(numbers)
55
>>>max(numbers)
115
```

The second way to write program is by making our own logic. We should instantly make a decision to use loops as there are many items in this list, which means we have to do some comparison repeatedly. Hence, it's always better to use looping if you want to perform a task repeatedly. Now, we need to decide on what to use: a `for` or a `while` loop. It's better to use a `for` loop here because for loops work on finite lists. Each time we iterate over an iterating variable, it will contain an element from the list so that we can compare them with the previous element repeatedly. At first, we won't have anything to be the smallest number. Hence, we have to make a variable that will contain a `None` value. This means we won't have any value. After the first iteration, we will assign its value to the first element of the list. Let's see how it works:

```
>>> smallest_number = None
>>> for item in [113,115,55,66,65,90]:
            if smallest_number is None or item < smallest_number:
                    smallest_number = item
>>> print("Smallest:", smallest_number)
Smallest: 55
```

Let's break down the preceding code into the following segments:

- In the first statement, `smallest_number = None` is assigning `None` to that comparing variable. We assigned `None` instead of any other number so that we don't miss any numbers while comparing.
- In the second statement, we made `item` an iterating variable, which is going to read a list of numbers. At each iteration, it is going to store elements from that list. In the first iteration, the value of the item is 113. In the second iteration, the value of the item is 115; at the third iteration, the value is 55; and so on.
- We are now inside the body of our `for` loop, where we have to build a comparison statement. First, we need to check whether the smallest number is `None` to make sure we are starting from the base. After that, we are going to check whether the current item from the list is smaller than `smallest_number`. The first iteration's second condition is False, but the first condition, that is, `smallest_number`, is `None`, that is, True, which means we are going inside the body of the conditionals. We will assign `smallest_value` to the first item of the list, that is, 113.
- In the second iteration, the item is 115. We are going inside the `for` loop and checking whether 115 is smaller than the value of `smallest_numbe`, which is 113. This is `False`, and so it doesn't go inside the conditional's body; instead, it jumps to the third iteration.
- In the third iteration, the item is 55. We are going to check for the condition, which is going to check whether the value of the item, that is, 55, is less than that of `smallest_number`, which is 113. The condition (55 < 113) is `True`, and so it changes the value of the `smallest_number` variable to 55. The `for` loop is going to iterate until the last number of that list. It is going to use the same comparison operation at each iteration to give us the smallest value, that is, 55.

With just a change in the comparison operator, we can make a program that will print the largest number from the list. Instead of using the `item < smallest_number` statement, we can use the `item > largest_number` statement to get the largest number as follows:

```
>>> largest_number = None
>>> for item in [113,115,55,66,65,90]:
            if largest_number is None or item > largest_number:
                    largest_number = item
>>> print("Largest: ",largest_number)
Largest: 115
```

In the next section, we'll look at how to use two different statements, **break and continue**, in order to change or skim the sequence of iteration.

The break and continue statements

While writing programs, sometimes, you want to skip the execution of statements or stop the iteration forcefully. These operations are handled by the continue and break statements. They can be powerful in multiple use cases where you want to make a program sort the elements of a list or to break the loop when an `if` condition is met. The `continue` statement is used to skip the execution of the program. We use these statements inside the body of loops. We can sort elements out of lists using these statements. We can't use both of these statements in a single loop, even if we use both of them together since the break is going to stop the loop, which will make the continue statement useless. We can use these statements with conditionals. Whenever a condition is met, we are going to either break or skip the iteration. Let's make a program that can sort the elements of a list:

```
>>> items = [1,5,7,8,"Free","spam",False,89,90,11,"Python"]
>>> refined_items = []
>>> for item in items:
        if type(item) != int:
                continue
        else:
                 refined_items.append(item)
>>> print(refined_items)
[1,5,7,8,89,90,11]
```

In the preceding code, we are refining the elements of a list by keeping the integer numbers in the output list. Other data values, such as strings and Booleans are removed. First, we looped an entire list and at each iteration, the element is stored in the item variable. We check the type of data that are stored in the item variable with the `type()` method. If the type of the value that's stored in the item is not an integer, we are using the continue statement to infer that we won't do anything if it is not an integer. If the type of item is a Boolean or string, we are skipping that iteration with the continue statement. However, if the type of the value that's stored in an item variable is an integer, we are going to execute the statement that's inside the else part of the code. We are going to take that integer item and add it to the new output list, which is called `refined_items`. After each element is checked, we print the refined list, which is the ultimate collection of numbers.

If you use a `break` statement instead, things will be the same until element 8. But instead of printing other elements from `[89,90,11]`, our output will be limited to `[1,5,7,8]`. This is because the `break` statement is going to stop iteration after appending element 8 to the list. Hence, we can conclude that whenever the Python interpreter triggers a break statement, the loop is going to be terminated:

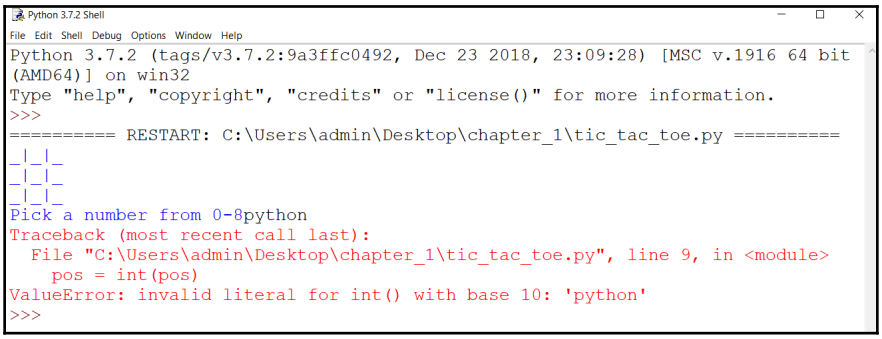
```
>>> items = [1,5,7,8,"Free","spam",False,89,90,11,"Python"]
>>> refined_items = []
>>> for item in items:
```

```
        if type(item) != int:
                break
        else:
                refined_items.append(item)
>>> print(refined_items)
[1,5,7,8]
```

We know that, while deploying programs in a real-world environment, such programs will be accustomed to a different scenario that our code won't be able to handle. In such a case, our program will terminate, which will have a negative impact on the user or player of the game. Hence, we have to code in such a way that our code can be applied to any scenario, even when it encounters unexpected errors or exceptions. This type of powerful technique in programming is known as **exception handling** and is what we will cover in the next section.

Handling exceptions using try and except

In the preceding chapter, we created a simple tic-tac-toe game. We talked about making some modifications at the end of that chapter. One of the modifications was suggested due to the deficiency of the code, which was unable to handle the input of a user other than an integer. What if our user enters a string as input to the position variable of our game? The following exception will be thrown:



In the preceding screenshot, we can see that our code was unable to handle a string as input by the user. Our code is going to perform well if—and only if—we enter an integer. If the user mistakenly enters any other data values, the program will crash. The aim of this topic is to handle this type of error. We have two types of error: syntax errors and exception errors. The following code shows examples of both:

```
>>> print("Hey! it's me")))
SyntaxError: invalid syntax
```

Whenever you type in the wrong statement, it is going to throw an error message, that is, a syntax error. Here, we used two more parentheses than normal to enclose the print statement, which is incorrect. Due to this, the Python parser throws a syntax error. Remove those extra two parentheses to eradicate the syntax error:

```
>>> a = 34
>>> a / 0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    a / 0
ZeroDivisionError: division by zero
```

Now, the Python parser has thrown an exception error. This type of error occurs even if your Python syntax is correct. This can be either a mathematical or logical error. In mathematics, you cannot divide any number by zero. This results in an infinite, which has not been defined by Python. Hence, we ran into an exception. There are different types of exceptions. If you want to know the name of the exception you ran into, check the last statement of your code after received the exception. In our error message, the last statement says `ZeroDivisionError`. Hence, we ran into a `ZeroDivisionError` exception. If you run into any of these exceptions, then it's likely that your code has crashed. Hence, our game tic-tac-toe has also crashed because it was unable to handle input data other than integers.

Now, our main aim is to make our code reliable so that even if our code runs into an exception, instead of crashing, it gives the user a friendly message. In the preceding example, instead of terminating the program, we can send the user a message saying `You cannot divide any number by zero`. This process is called exception handling. These are done within `try` and `except` blocks in Python.

If you are unsure about the code regarding whether it gave you an error, you should always use `try` and `except` blocks. Your main code, which is likely to run into exceptions, should be kept inside a `try` block. Then, if it runs into an exception, the Python parser should execute the code that's inside the `except` block. The following example should make this clearer to you:

```
>>> a = 34
 #INSIDE TRY BLOCK: put code that can give you error or exception
>>> try:
       print(a/0) #this will give you exception
    except:
       print("You cannot divide any number by Zero. It is Illegal!")
#message to user

You cannot divide any number by Zero. It is Illegal!
```

The preceding code shows how easy it is to handle these kinds of errors. You put your main code inside a `try` block and if it ran into an exception, the main code won't be executed. Instead, the code inside of the except block will be executed, which in this case is a user-friendly message. You can also use `pass` so that you terminate the program without providing the user with a message.

With the except keyword, you can also pass the name of an exception explicitly. However, make sure you know the proper exception name you are going to run into. In this case, we know we are going to run into `ZeroDivisionError`, and so we can write the exception name with an except block, like so:

```
>>> a = 34
 #INSIDE TRY BLOCK: put code that can give you error or exception
>>> try:
        print(a/0) #this will give you exception
    except ZeroDivisionError:
        pass
```

Now, let's see how we can refine our tic-tac-toe game with everything we have learned about so far. We will use conditionals, looping, and exception handling to modify the code that we wrote in the previous chapter.

Making a game controller for our tic-tac-toe game

In the preceding chapter, we build a simple tic-tac-toe game. Since we have learned about conditionals and looping in this chapter we are now able enough to make some advancement on our game. We will make the following changes to our game:

- We have to make the game multiplayer, which means we have to make modifications to the program so that two players can play our game. We can make conditions that can toggle who's playing when.
- When we talked about exception handling, we saw that our game was unable to handle string data that was inputted by a user. We can use `try` and `except` blocks to handle that exception.
- We were unable to determine the winner of our game with the code that we wrote in the previous chapter. Now that we have learned about `if` conditionals, we can come up with some logic to check whether the player is the winner.

We will start our game development process by brainstorming in order to gather some critical information about the game.

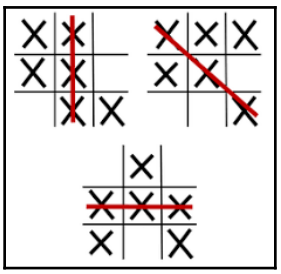
Brainstorming and information gathering

One of the important features our code lacks is readability. In the game code from the preceding chapter, we didn't have a proper way to track the positions of the game board. The first thing we can do with our code is make a choices list, which will contain all of the choices that can be made by the player. Here, the choices are for the tic-tac-toe board position. In the tic-tac-toe game, the player can choose between 0 and 8 numbers, which are placeholders if they're not occupied by another player.

The second thing we have to add in our code is a way we can toggle whose turn it is. Since we have only two players, we can make player 1 play first and make the first move of the game. Hence, making logic would be easier. We will make one Boolean variable and change its value from True to False so that we can make a condition to change the player's turn, as follows:

- `playerOne = True` will make sure it's player 1's turn.
- `playerOne = False` will allow player 2 to make a move on our game board.

We have to go through the rules of the tic-tac-toe game to make any player a winner based on the positions they occupy on the game board. If a player, either X or O, occupies entire rows, columns, or diagonals of the tic-tac-toe board, that player will be considered the winner. This is depicted in the following screenshot:



Modifying the model

The program we wrote in the previous chapter was only able to make one player play the game. Since tic-tac-toe is a multiplayer game, modifications should be made so that multiple players can play this game. For that, we have to do two things:

- Track empty places on the board.
- Make a condition to toggle the player's turn.

For both of those modifications, we have to make one variable that can track every empty and occupied position on the game board:

```
#code is written in same previous file

choices = []
for pos in range(0,9):
      choices.append(str(pos+1))
```

If you use the `>>> print(choices)` variable, this will result in a list of values: `['1', '2', '3', '4', '5', '6', '7', '8', '9']`. They are positions for our board game.

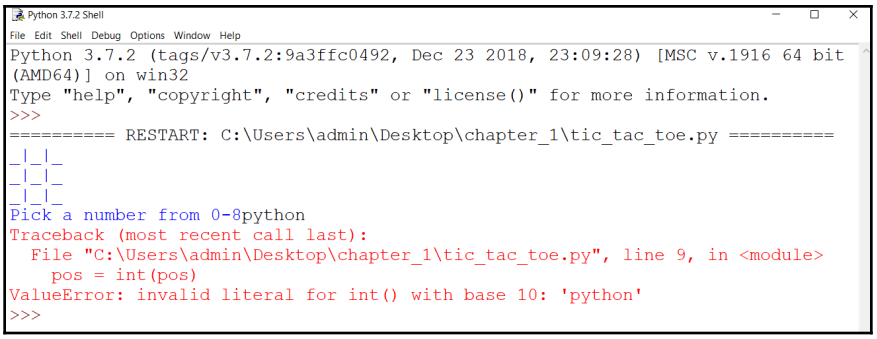
Now, if you want to print the board's layout, you cannot use code from the previous chapter. Instead of using the `game_board` variable, we are going to use the `choices` variable. This works in the same way it did in the previous example. We are going to add a single line of dashes in-between each row:

```
#board layout
print('\n')
print('|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|')
print('----------')
print('|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|')
print('----------')
print('|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|')

#output
'''
|1|2|3|
----------
|4|5|6|
----------
|7|8|9|

'''
```

The main problem in our game may arise when the user inputs something other than a number. For example, if the player enters a string, the game will be terminated with an exception—we don't want that situation to happen. As you may recall, we use **exception handling** to avoid such a scenario. We will add this to our game in the next section.

Handling the exceptions of the game

Let's run the game that we've made so far and input a string into the input field instead of an integer value. You will get the following exception:



We don't want our program to crash whenever the user makes a mistake while interacting with our game. Instead, we can send them a user-friendly message saying `This is not valid, press integer only`. Let's handle this type of exception by using a `try` and `catch` block, as follows:

```
while True:
    print('\n')
    print('|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|')
    print('----------')
    print('|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|')
    print('----------')
    print('|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|')
    #above code is to print board layouts

    try:
        choice = int(input("> ").strip())
    except:
        print("Please enter only valid fields from board (0-8)")
        continue
```

When we run our program this time, we will get the following output:

```
|1|2|3|
----------
|4|5|6|
----------
|7|8|9|
> python
Please enter only valid fields from board (0-8)


|1|2|3|
----------
|4|5|6|
----------
|7|8|9|
>
```

Here, we entered a Python string value into the input field. Instead of it crashing the program, we got a message saying `Please enter only valid fields from board`. This is a convenient way of handling exceptions using `try` and `except` blocks. We used the same main loop from the previous chapter, which was going to loop an infinite amount of times. Inside the body of the `try` block, we kept the code that might throw an exception. `strip()` is a string method that is going to remove white spaces from the user's input. We have to type-cast user input using the `int` method so that the input data, which will be in the form of a string, is converted into an integer. If we get an exception, we are going to execute the code that's inside the body of the `except` block. The `continue` keyword is going to make the main loop run again from the beginning if we run into an exception.

The main feature that must be added to our tic-tac-toe game is multiplayer so that two players can play the same game turn by turn. This toggling feature will be added in the next section.

Toggling the player's turn

Writing programs to make two players play the game is easy with Python—all you need to do is create one Boolean variable that will denote who the current player is. Then, based on the two values of the Boolean, either `True` or `False`, we can change who's playing the game. However, if you want to add more than two players, this idea isn't going to work. We are going to use the following Booleans:

- `Is_Current_One = True`: The current player is player 1 or X.
- `Is_Current_One = False`: The current player is player 2 or O.

This is shown in the following code:

```
#creating Boolean variable
Is_Current_One = True #default player is player X

#first move is done by player X
while True:
    #put code of board layouts here
    if Is_Current_One:
        print("Player X")
    else:
        print("Player O")
    #put try and except block here
    #-------------------------------------------
    #code to put either X or O on position selected by user
    if Is_Current_One:
        choices[choice-1] = 'X'
    else:
        choices[choice-1] = 'O'
    #code to toggle between True and False
    Is_Current_One = not Is_Current_One
```

Let's break the preceding code into segments so that we can understand it better:

- We have our main loop, which is going to run an infinite amount of times until it triggers a `break` statement. We've already learned that the `break` keyword was going to terminate our loop. In the body of the main loop, we print whether it's player X's or O's turn to make the player aware of their turn.
- We have created a Boolean variable called `Is_Current_One`, which was assigned a value of `True`. This means that the first player to make a move will be player `X`. If we make this variable False, then the default player to make the first move will be player `O`.
- Inside the main loop, we created a condition to check whether player `X` or player `O` has placed either `X` or `O` on the board layout accordingly. The `choices[]` variable reflects the board's position. `choice` is the user's input, which we subtract by 1 because our choices variable is a list type. We know that list indexes start from index 0, but we have entered user input from 1 to 9. Hence, we subtract the `choice` input variable with 1 to accommodate this list variable.
- The `>>> Is_Current_One = not Is_Current_One` statement is going to toggle between the players. As we mentioned previously, if `Is_Current_One` is `True`, the player is `X`, Now, we've also made a condition so that we can change True to False in the next iteration so that player `O` can make the next move.

Let's see what we are up to now by running our script file. You will see the following result printed in the shell:

```
|1|2|3|
----------
|4|5|6|
----------
|7|8|9|
Player X
> 8

|1|2|3|
----------
|4|5|6|
----------
|7|X|9|
Player O
> 5

|1|2|3|
----------
|4|O|6|
----------
|7|X|9|
Player X
>
```

Now, we have created our game, which can take some input from a user and place it on the tic-tac-toe board. We've created some logic to change whose turn it is. We were also able to handle exceptions that might appear in our game using `try` and `catch` blocks.

We have been progressing at a rapid pace, but our game is still incomplete. We haven't made any logic to make a player the winner if they occupy either a row, a column, or three diagonal cells. We will do this in the next section.

Making a player the winner

Tic-tac-toe is an easy game to make, but the main purpose of building this game is to cover almost every core programming paradigm of Python, such as variables, numbers, models, built-in methods, looping, branching, and exception handling. Now, our game is good enough to be played by two players, but a multiplayer game can only have one winner at the end of it. Therefore, we have to make brand new logic that will reward the player if they win. We need to cover three use cases, as follows:

- If an entire row of the tic-tac-toe board is occupied by a player, that player will be the winner.
- If an entire column of the board is occupied by a player, that player will be the winner.
- If an entire diagonal of the board is occupied a player, that player will be the winner.

Let's print our game board layouts, along with their positions, so that we can track all of the positions of the board while making the preceding conditions:

```
| 1 | 2 | 3 |
 ----------
| 4 | 5 | 6 |
 ----------
| 7 | 8 | 9 |
```

Since we have to loop through all of these positions from 1 to 9, we need to use a `for` loop. Since we have a finite list of numbers, a `for` loop is easy to use. We have to make two conditions to check whether a player occupies an entire row or column. After dealing with row and column, we will examine diagonal conditionals with isolation:

- **For row**: If any user occupies [1,2,3], [4,5,6], [7,8,9], that particular player will be considered the winner.
- **For column**: If any user occupies [1,4,7] , [2,5,8], [3,6,9], that particular player will be considered the winner.

However, the position inside the `choice` variable ranges from 0 to 8, that is, ['0','1','2','3','4','5','6','7','8'], and so index 0 references the first position of the board, an index of 1 indicates the second position of the board, and so on.

We have been using a while `True` statement for our main loop. Let's modify that so that our code will run until we a player is the winner. We will run our main loop until `won= False`. If we get a winner of the game, we will change the value of the `won` variable to `True` so that the main loop will end:

```
won = False #at first we don't have any winner
while not won:
    #code from previous topics
    #logic to make any player winner:
    for pos_x in range(0,3):
        pos_y = pos_x * 3

        #for row condition:
        if (choices[pos_y] == choices[(pos_y + 1)]) and (choices[pos_y]
           == choices[(pos_y + 2)]):
            #code to change won to True
            won = True #main loop will break
        #column condition:
        if (choices[pos_x] == choices[(pos_x + 3)]) and (choices[pos_x]
          == choices[(pos_x + 6)]):
            won = True #main loop will break
```

In the preceding code, we made two conditions to check whether the player is the winner. We made the `won` variable to track whether any player has won. If any player occupies an entire row or column, we are making the `won` variable's value True and our loop will break, which means we will end our game. However, we haven't given the user a message about being the winner. Let's write some code that will tell the user they're the winner after we check for the row and column condition:

```
while not won:
    #code from previous topic
    for pos_x in range(0,3):
        pos_y = pos_x * 3

    #add condition for row and column here

#print who is winner
print("Player " + str(int(Is_Current_One + 1)) + " won, Congratulations!")
```

The statement we've written with the `print` method may create confusion because of the `str(int(Is_Current_One + 1))` command. Here, `Is_Current_One` is either `True` or `False`. However, it also corresponds to 1 or 0, where 1 is for `True` and 0 is for `False`. If player `X` is the winner, then it's player 1 who won, but the turn will have changed over to player `O`, that is, player 2.

Hence, we have to add this to 1 so that the current player is determined the winner, rather than the player who goes next. Since this is a two-player game, this makes sense. Let's run our code to examine the result:

```
|X|2|3|
----------
|X|O|O|
----------
|7|8|9|
Player X
> 7
Player 1 won, Congratulations!
>>>
```

We haven't finished yet—we also have to add one more condition to check whether the diagonals are also occupied by a player. Let's add that condition now:

```
| 1 | 2 | 3 |
 ----------
| 4 | 5 | 6 |
 ----------
| 7 | 8 | 9 |
```

If any player occupy positions [1,5,9] or [3,5,7], they will be regarded as the winner. However, our `choices` variable is a list that contains all of the positions. Its index starts from 0, which means if you want to locate the position for player 1, you should pass this as `choices[0]`, like so:

```
while not won:
    #code from previous topic
    for pos_x in range(0,3):
        pos_y = pos_x * 3
    #add condition for row and column here

    #diagonal condition here:
    if ((choices[0] == choices[4] and choices[0] == choices[8]) or
        (choices[2] == choices[4] and choices[4] == choices[6])):
```

```
            won = True


    #print who is winner
    print("Player " + str(int(Is_Current_One + 1)) + " won, Congratulations!")
```

Now, let's run our game one more time to check whether this condition works properly:

```
|O|2|3|
----------
|X|O|6|
----------
|X|X|9|
Player O
> 9
Player 2 won, Congratulations!
>>>
```

Finally, we have completed our game! We were able to include many features in our game, such as exception handling, multiplayer mode, and logic to make a player a winner. However, we still have to refine this game by adding user-defined functions so that we can print our board layout. This will follow the DRY principle and will be covered in the next chapter.

Summary

This chapter has given us a roller-coaster ride of all of the core topics of programming paradigms with Python. We have covered flow controls and how to achieve them using branches and loops. We learned how to make conditions and fed them to the conditionals. Then, based on those conditions, we were able to make a switch between the execution of statements. We saw how to automate things using Python looping and branching. We fed multiple possible conditions with the `if` keyword and, based on the result of the Boolean expressions, the flow of the program was controlled. We also learned about the different types of looping and saw how to use them to iterate a list of items or objects. Then, we saw how to handle exceptions using try and except block.

Finally, we made our tic-tac-toe game more playable than ever before by incorporating the different paradigms we learned about in this chapter. We added `try` and `except` blocks so that any exception will be caught and handled properly. We also added features such as multiplayer mode and logic to make a player the winner. This makes the game highly interactive. Finally, we made a game controller using conditionals and looping. However, we won't stop here; more modifications will be made in the upcoming chapters.

The next chapter will be life-changing for us. Up until now, we have only been using the built-in functions of Python, such as `min()`, `max()`, and `input()`. In the next chapter, we will see how to make our own functions and to use them so that we can make our game more readable and reusable. We will cover data structures such as list, set and, dictionary so that we know how to manage and store more complex datasets. Don't be overwhelmed by all of these statements, though. You have come this far, and are now on the brink of becoming a proficient Python programmer. Before moving on to the next chapter, make sure you are comfortable with all of the topics that we have learned about so far.

4
Data Structures and Functions

In this chapter, we are going to traverse through the concept of data structures and functions, two primary building blocks of Python. Normal variables are a good way to store singular units of data of any type, but for arrays of data, we should always use data structures. Python has a raft of data structures available that you can use to represent and manipulate your datasets, or even to combine them in order to make your own data structures. We have already seen built-in data types, such as integers, Booleans, floating numbers, strings, and characters. They are called built-in because they come as a dovetail with Python. Now, we are going to explore built-in data structures, such as lists, dictionaries, tuples, and sets. Combinations of these built-in data types result in data structures that are implemented independently. For example, if we put different integers in one place, they are arrays of numbers. Python call them lists, which are widely used data structures.

In order to become proficient programmers, first we have to learn about core programming paradigms, such as variables, numbers, modules, and built-in functions, before then diving into the data structures and algorithms. This book is not any different. We have already covered the basics of Python; now, it's time to delve into the data structures and the method that is used to access and manipulate data. In the previous chapter, we modified our game with conditionals and looping.Now, let's extend our knowledge of Python to include the broad concept of data structures and functions so that we can refine our game decide the most fa
further.

The following topics will be covered in this chapter:

- Why do we need data structures?
- The four structural pillars of Python—lists, dictionaries, sets, and tuples
- Functions
- Adding AI to a tic-tac-toe game
- Game testing and possible modifications

Technical requirements

The following are the requirements that you will need to understand this chapter properly:

- The Python IDLE
- The code assets for this chapter can be found at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter04`

Check out the following video to see the code in action:

`http://bit.ly/2oNoxOL`

Why do we need data structures?

As a programmer or computer scientist, we always search for ways to optimize our code. Optimization is a way of refining code in order to improve code efficiency and quality. Data structures are a shrewd way of organizing data in a computer, therefore making it easier to retrieve and access data, which results in code optimization.

So far, we have learned how to use conditionals to make conditions, and how to make flow controls with normal variables. However, real-world data is not limited to one unit. We may collect profuse amounts of data, which would have the highest level of intricacies. It may contain thousands of integers, hundreds of Booleans, or a combination of these. Thus, storing them into a single, normal variable with an assignment statement is not possible. Take a look at the following example:

```
>>> a = 8 9
SyntaxError: invalid syntax
>>> a = "Hey" "Hello"
>>> a
'HeyHello'
>>>
```

In the preceding code, we tried to assign two values to a single variable. It produced a syntax error. We even tried to put two string values into a single variable, `a`, but instead it performed a concatenation, and assigned it as a single value. Thus, storing multiple values in a normal variable is not possible. However, we can easily convert this normal variable into a data structure, as shown in the following snippet of code:

```
>>> a = 8 , 9
>>> a
(8,9)
>>> type(a)
<class 'tuple'>
```

We have converted the normal `a` variable into a tuple, which is a built-in data structure of Python. We will cover this in detail in the upcoming sections.

This variable was only able to store a single unit of data, but if we carry out multiple assignments, the preceding value would be overwritten. However, if you want to preserve all the data in one placeholder, data structures are a way of doing this.

As a programmer, our primary responsibility is to perform some sort of manipulation on input datasets. Input can be anything, such as emails or passwords, or maybe a request to enter into a system or location for Google Maps, where we can use the data to perform some sort of computation using algorithms. In addition, the haversine algorithm (refer to the following URL to learn more about this algorithm: `https://rosettacode.org/wiki/Haversine_formula`) gives you the exact distance between your location and your destination. Thus, input data can have a wide range, but the main task is to manipulate it. Our systems and processors are not powerful enough to handle the manipulation of several terabytes of data all at once. Thus, choosing a proper data structure is a major optimization that can be carried out by the programmer. If we are able to store such inputs into faster data structures in any organized form, we can perform even complex tasks with ease. Data structures are just places or storage that provide structure to such complex data, but processes such as fetch and manipulation are performed using algorithms.

Still in doubt? Let's make things clear by taking an example of a library in order to understand data structures and algorithms. First of all, imagine a scenario where we don't have proper management in a library. Books are not properly placed in their relevant sections. Now, searching for a book in a particular section is useless, because it won't be there. The best-case scenario is that you may find your book within a few minutes, but the worst-case scenario is that you may have to search the entire library to find a book about history, for example. However, if the library is properly organized and managed, you will be able to go directly to the relevant section where history books are stored, and search for your book in that section only. Here, the library represents the data structure and the book is the data that you are searching for. Whenever you need data, you go to the data structure, and if it is properly managed, you will be easily be able to retrieve it. The steps that define how you are going to search for the books are called algorithms.

Enough of the theory—let's get our hands dirty by coding and learning about the four pillars of data structures of Python—**lists**, **dictionaries**, **sets**, and **tuples.**

The four structural pillars of Python – lists, dictionaries, sets, and tuples

In `Chapter 2`, *Learning the Fundamentals of Python*, we learned about strings, and we called them immutable data types because they do not allow assignment operation. This is shown in the following code:

```
>>> name = "Python"
>>> name[0] = 'hey'
TypeError: 'str' object does not support item assignment
```

However, data structures must be flexible, which means that we should be able to store and extract elements of data from any position. Thus, most of the built-in data structures of Python are mutable, which means that they can be changed and manipulated with proper indexing. The proper categories for the four data structures are as follows:

- **List and tuple**: Mutable data structures
- **Dictionary**: Mapped data structure
- **Set**: Mutable and unordered data structure

Each category exists because of its uniqueness, and you will see how easy it is to distinguish them as superior or inferior to one another in the upcoming sections. However, remember that they are all superior at some point; it's up to us to choose a data structure that is appropriate to the situation. For example, we say that dictionaries are the king of data structures, but we may come across a situation where tuples may be the faster way to store data, which is usually the case when we make programs in Python using databases such as SQLite and MySQL. Now, let's take a look at each of these built-in data structures of Python, starting with a basic mutable data structure, which is known as a list.

Lists

Just like a string is a sequence of characters, a list is a sequence of values. Values can be a combination of any type. The values in a list are called items of that list. Lists are mutable and ordered data structures, and elements of lists can be extracted using indexes. Like strings, we can extract multiple elements from a list using the slicing technique. Lists are notorious for storing homogeneous data types, but they also support heterogeneous data types. We are not confined to creating lists using only a single method; there are multiple ways of doing this. Let's look at some of basic ways of creating lists in Python:

```
>>> first_list = []
>>> type(first_list)
<class 'list'>
```

The simplest way of creating lists is using square brackets—`[]`. You can add multiple elements inside these brackets, and there are multiple ways of doing this:

- Firstly, we can add elements to the list at the moment of its declaration, as shown in the following example:

```
>>> numbers = [1,2,3,4,5,6,7,8,9]
```

- You can also add elements to lists using the built-in methods within Python. For example, the `append` method can be used to insert elements into the list. Elements are added to the last position of the list, as follows:

```
>>> numbers.append(10)
>>> print(numbers)
[1,2,3,4,5,6,7,8,9,10]
```

We can also make a list containing multiple types of values, as shown in the following example:

```
>>> [3,7,9,"odd",True]
[3,7,9,"odd",True]
```

Here, we made a list containing numbers, strings, and Booleans. Thus, we stored heterogeneous data types in a single list. We can also add multiple lists within a single list, and these are known as nested lists. As the term suggests, one list is nested within another list, as shown in the following example:

```
>>> [1,2,3,[4,5,6],7,["hey","Python"]]
```

In the previous example, we created a single list containing six elements. We have integers and two whole lists (`[4,5,6]` and `["hey","Python"]`) within that primary list. Thus, these types of list are called nested lists.

Whenever you assign those lists to variables, the variable type eventually becomes the list type. Now, the type of variable has been changed from a built-in data type such as `int`, `str`, or `bool`, to a built-in data structure, which is the list.

Another way of creating lists is by using the built-in Python method—the `list()` method—but it is redundant in the preceding process because we have to pass the whole list as an argument of this list method. This is known as the type-casting method. If you want to convert any other data structure into a list, we use the `list()` method, as shown in the following example:

```
>>> list([1,2,3,4,5])
[1,2,3,4,5]
```

Inside the `list()` method, we have to pass an argument in the form of the list that contains the elements, which are enclosed using square brackets. You must have guessed by this point that every built-in data structure that is available in Python must have one built-in method in order to create its data structures. We use the `dict()` method to create dictionaries, the `set()` method to create sets, and `tuple()` to create tuples in the same way that the `list()` method creates lists.

Since we have uncovered the different ways of creating a simple, yet powerful, data structure that goes by the name of `list` in this section, let's see how we can access and manipulate its stored data.

Accessing list elements

If you recall the way that we access the elements of a string, you can also replicate this process in the case of lists. We use square brackets in lists in order to indicate the position inside it so that we can extract and interact with particular elements. We call this the index, and it is added inside this `[]` bracket symbol. It is same when creating new lists. Indexes of lists start from 0 and increase in unit digits, while also traversing from left to right. Like strings, lists also support negative indexing:

```
>>> winner_names = ["Chandler","Joey","Monica","Racheal","Ross"]
>>> winner_names[0] #0 is first index
'Chandler'
>>> winner_names[-1] #-1 is last element
'Ross'
```

When we tried to assign elements to the string, it was not valid. Unlike strings, lists provide reassigned items to the list. Thus, we can say that lists are mutable, which means that they are changeable and modifiable. This feature makes the list the simplest and most flexible data structure of all. We can assign elements using the append method, which we saw in the preceding section, but this method only allows us to add elements to the end of the list. If you want to add elements to any particular position, you can explicitly tell the Python parser to do this through indexing and assignment statements.

For example, if you want to add `loves` in-between two elements of the list, you can do the following:

```
>>> msg = ["Joey","Monica","Racheal"]
>>> msg[1] = "loves"
>>> msg
['Joey','loves','Racheal']
```

Thus, we can see that the element at position one, `Monica`, has been replaced by `loves`, which shows that we can change the order of the elements and reassign any other element to the list.

While dealing with data structures, it's always good practice to observe them hypothetically. We can consider them as a mapping process, where every element on the list is mapped to certain indexes. Indexes are positions, and whenever we backtrack the list through the indexes, we are able to access elements of these indexes. Even if you have a nested list, that is, one or multiple lists inside the single list, they will be also mapped to an index, as shown in the following example:

```
>>> web_dev = [["Django","Flask"],["Laravel","Symfony"],"Nodejs","GOLang"]
>>> web_dev[0]
['Django','Flask']
>>> web_dev[1]
['Laravel','Symfony']
```

We know that square brackets are used to access elements of a list, but if we want to access elements of a nested list, we have to add another square bracket in order to specify the level of indexes that are needed to access these elements, as shown in the following example:

```
>>> web_dev[0][0]
'Django'
>>> web_dev[1][1]
'Symfony'
```

We can check whether the element is in the list or not by using the `in` keyword. The syntax that's used in the statement's results give a Boolean value that is either `True` or `False`:

```
>>> names = ["John","Jack","Cody"]
>>> "Cody" in names
True
>>> "Harry" in names
False
```

Accessing the elements of the list is easier, but sometimes if you make mistakes when counting proper indexes, it may give an unintended result. Thus, you must count the elements of the list from the index, 0. If you put indexes inside the square brackets that do not map to any value, you will run into an error, known as `IndexError`, as shown in the following example:

```
>>> odd = [1,3,5,7,9]
>>> odd[20]
IndexError: list index out of range
```

The `IndexError` message pretty much explains why we ran into this error. The index of the list named odd stops at 4. However, we passed 20, which is the position that has no mapping to the values, or simply, we don't have any elements in this position. Thus, while working with lists, we have to track every position of the inserted values so that we don't run into any exceptions. However, we have a solution to prevent such conditions—just recall the exception handling mate! That's what you need to call in order to handle these exceptions so that our code runs properly instead of crashing.

Since we have learned how to access these elements using the indexing technique, let's dive into how to traverse the entire list, which is part of accessing the entire list. First and foremost, the thing that you must be aware of is looping. Since we are dealing with a list that has multiple pieces of data items stored in it—which means accessing multiple data, multiple times—we just need to recall the method that we would usually use if we want to do things repeatedly. There is nothing better than looping that suits this condition perfectly. Thus, the `for` loop is the most appropriate method if you want to read the entire elements of the list; for example:

```
>>> for number in [1,2,3,4]:
         print(number)
1
2
3
4
```

We can also update and refine our list within a `for` loop. The following example is among one of the most important examples that we have learned about so far; make sure you grasp every teeny-tiny piece of information from it:

```
>>> even_num, odd_num = [], []
>>> for i in range(0,10):
        if i % 2 == 0:
            even_num.append(i)
        else:
            odd_num.append(i)
```

```
>>> print(even_num)
[0,2,4,6,8]
>>> print(odd_num)
[1,3,5,7,9]
```

As always, let's break up the preceding code into segments. First of all, we declared two empty lists, which will be our output lists of even and odd numbers. Then, we used looping to access the elements of the list. The statement range (0, 10) will yield a list that contains numbers from 0 to 9. Here, 10 is the exclusion position. Thus, we have looped an entire list of elements one by one. If you have any difficulty in understanding the concept of recursion programming, recall the *Looping through dictionaries* section. After taking every element of the list at each iteration, we enter the body of the loop and check for the condition that will determine whether the elements are even or not. If it is even, we append it, which means that we insert that element into the `even_num` list, and we do a similar thing in the case of odd numbers.

Wow, do you realize what you just did? You have used a simple yet powerful data structure and carried out a linear search. Although we have many more topics to cover, this is the best thing that we have done so far. Now, gear yourself up to learn more about list operations and methods.

List operations and methods

Can you recall the type-casting method of Python from the preceding chapter? It is surely the best way to convert one data type into another. We have looked at strings, its slicing techniques, and methods. However, we came to realize that it was immutable. This restriction is so robust that we can't change any of the elements of that string. However, now we have come to the most flexible data structure, which goes by the name of `list`. So, why not convert the string into a list so that we can make it mutable too. Let's use the following example to clarify this:

```
>>> name = "python"
>>> type(name)
<class 'str'>
>>> name = list(name) #list() method converts any data type to list
>>> type(name)
<class 'list'>
>>> name[0] = 'c'
>>> name
['c', 'p', 'y', 't', 'h', 'o', 'n']
```

Now, we can manipulate the preceding list however we like; maybe using built-in methods. However, most of the operation, apart from the assignment, is quite similar to that of strings. We learned so much in the string section, such as slicing, addition, and multiplication operations, and even some of the string methods. Strings and list manipulation are quite similar-they even start with same index, 0. That said, the built-in methods that are provided by Python for strings and lists are not quite as similar, and why would they be? They are different types of data or structures.

You can do arithmetic operations with a list, such as addition and multiplication. Remember, though, that addition can only be done between two lists, whereas multiplication must be done between a list and any integer number, as shown in the following examples:

```
>>> even = [0,2,4,6,8]
>>> odd = [1,3,5,7,9]
>>> number = even + odd
>>> number
[0,1,2,3,4,5,6,7,8,9]

>>> ["john"] * 3
['john','john','john']
```

In the first example, we performed concatenation between the lists using the addition operator. In the second example, we multiplied the list by three, and the effect of the multiplication can be observed within the content of that list. In our case, `john` has been multiplied by three to create three `john` values.

The built-in methods that are provided by Python are used to manipulate the values of the list. They act upon the list by creating an object of it. Let's not confuse ourselves by talking about objects here; we have a dedicated chapter for that.

There are a lot of built-in methods available that can manipulate list structures, but we are going to cover the most important ones here. I find them useful because most developers use only a few of these when carrying out big projects. However, if you want to discover more, it's always good practice to take a tour of the documentation page.

We have already seen how we can use the `append` method to insert elements into the list. This method adds elements to the end of the list. But if you want to insert more than one element into the list, we can use the `extend` method, as shown in the following example:

```
>>> list_1 = [1,2,3]
>>> list_1.append(4)
>>> list_1
[1,2,3,4]
>>> list_2 = [5,6,7]
```

```
>>> list_1.extend(list_2)
>>> list_1
[1,2,3,4,5,6,7]
```

In the preceding code, the `extend` method takes the list as an argument, and appends all the elements of the list that are being called upon. When we print `list_2`, we will see that the list will remain unchanged.

In the same way that there is a method to add elements to the list, we also have a method that can delete elements from the list. Actually, there are two methods that can be used to delete elements. One works by passing an argument as an index, while the other works by passing an argument directly as an element that needs to be deleted. When we use the `pop` method, we have to pass the index of the element that needs to be deleted from the list; but when we use the `remove` method, we have to pass the element to it in order to specify that this particular element needs to be deleted. Take a look at the following snippet of code for an example:

```
>>> fruits = ["Apple","Banana","Orange","Mango"]
>>> fruits.pop(1)
"Banana"
>>> fruits
["Apple","Orange","Mango"]
>>> fruits = ["Apple","Banana","Orange","Mango"]
>>> fruits.remove('Orange')
>>> fruits
["Apple","Banana","Mango"]
```

There is another way of deleting elements in a list, which is by using the simple `del` keyword. Warning: if you write `>>> del fruits`, the entire list will be deleted. Make sure you explicitly specify the elements that need to be deleted. Specific elements can be fetched in a similar way to how we access elements using square brackets, as shown in the following example:

```
>>> fruits = ["Apple","Banana","Orange","Mango"]
>>> del fruits[-1]
>>> fruits
["Apple","Banana","Orange"]
```

There are a bunch of built-in functions that are available in Python that can perform arithmetic and logical operations on a list. Using these functions inevitably makes code cleaner and readable, and we can perform numerous tasks within a single line. Some of the important built-in functions for a Python list are as follows:

```
>>> prime = [2,3,5,7,11,13,17]
>>> sum(prime)
```

```
58
>>> min(prime)
2
>>> max(prime)
17
>>> len(prime)
7
```

Here, the `sum` function will give us a result of addition between the elements of the list. This method works only on the integers and floating values. Next, the min and max functions give the minimum and maximum values of the list, respectively. Another important function is `len()`, which will give us the length of the list. This `len` function works on any of the objects. In addition, we can use it with strings in order to find the number of characters in the list.

Sometimes, you may only want to extract particular portions or slices of the list, for example, only the first four items stored in a list that contains 1,000 items. In such cases, you have to use the slicing technique, which will be covered in the next section.

Slicing the list

Before learning the technique of slicing a list, let's recall how we sliced parts of a string. We used the square brackets operator to specify the start and endpoint for the slicing. It is quite similar in the case of lists, as shown in the following example:

```
>>> book = "Python Games"
#lets extract Games
>>> book[7:]
'Games'
```

Slicing a list can be done by adding a start index and a stop index within square brackets. In the preceding example, the stop index element is excluded from the resulting slices. Let's make a simple example that can slice parts of the element of our list:

```
>>> info = ["I","Love","Python","Java","NodeJS","C"]
>>> info[:3:]
["I","Love","Python"]
```

The second colon given in the `info[:3:]` statement is optional. The first semicolon separates two blocks as the start and end positions, but the second colon would be unnecessary if you don't want to add `step`. To learn more about `[start:stop:step]`, check out the *String slicing technique* section in `Chapter 2`, *Learning the Fundamentals of Python*. Take the following code as an example:

```
>>> info[:3] #same result as previous
["I", "Love", "Python"]
```

In the preceding code, `>>> info[:3:]`, we have added a colon (`:`) separator within square brackets to specify the indexes of the list. The space before the first colon is the starting index for slicing; here, we passed the empty index, which means it is the default, and it will start to slice from the beginning of that list. We passed index three to the next placeholder after the first colon in order to specify the end index for the slicing procedure. Here, the element at index three is `Java`, but it is in the exclusion position, which means it will slice from the beginning of list until the element at index two. The last placeholder after the second colon specifies the steps that need to be included in the slicing. Its value is empty, which means it's the default;, thus, we get a result without skipping any of the elements in-between those indexes. It works in the same way as the string slicing technique.

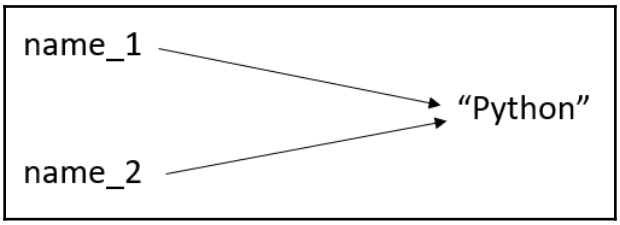
Now, let's learn about the needs of lists by examining the pitfalls of string objects. We will see how a list is considered superior and more prevalent than a string in the next section**.**

String and list objects

So far, we have covered multiple topics about lists; we saw how to create one for ourselves, and we saw how to add, delete, and manipulate elements of lists using built-in methods. Now, let's talk about another important concept of string and list-objects. Whenever we create any string, an object is created and stored in a particular memory reference. For any string that's created in a program, the Python parser creates one object for them, as shown in the following example:

```
>>> name_1 = "Python"
>>> name_2 = "Python"
>>> name_1 is name_2
True
```

In the preceding examples, both `name_1` and `name_2` point to same object. Thus, we can say that they are equivalent and identical. Two variables were created with the same `Python` string. These two assignment operations do not create two objects; instead, a single object is created and mapped into a global namespace. We can see that both of these variables, which have the same content, create a single object:



But in the case of a list, even if the contents is the same, they create two distinct objects, as shown in the following example:

```
>>> list_1 = ['a',1,2]
>>> list_2 = ['a',1,2]
>>> list_1 is list_2
False
```

You can clearly see that we get a result of `False` in the preceding code, which means that these two list are two different objects. They are not similar, although their contents are similar. Thus, whenever we create list variables, we term them as a list object, and its content is the value of that object.

Finally, we have covered our elemental and powerful `list` data structure in this section. Although we have not discovered the power of `list` yet, we have been using it from `Chapter 2`, *Learning the Fundamentals of Python*. Do you remember that we used `list` to represent the positions of the tic-tac-toe board game? Thus, we can conclude that, even when we have more robust and complex data structures such as dictionaries, trees, and queues, lists are considered the **Queen** of data structures because of their usefulness in holding complex data types within simple structures. Now, let's learn about **dictionaries**, which are considered the **King** of the data structures.

Dictionaries

Any discovery of a new data structure occurs because of the deficits in the preceding ones. Let's recall the demerits of the list. We have stored elements in the list structure that follow some order, and we must use indexes to retrieve those values. However those indexes are imaginary. Whenever you want to work with the list, you will have the overhead of remembering the order of that sequence, otherwise you will run into an `IndexError` exception.

Now, let's learn about more of the sturdy data structure that is available in Python. Dictionary, as the term implies, involves working the data structure in a way that is quite similar to our Oxford Dictionary. In our real-world dictionary, we have key and value pairs. Key refers to the word that you want to search for in the dictionary, while value refers to the meaning of that word. Similar to the Oxford Dictionary, we have key and value pairs in our dictionary data structure, and we call them elements or items, collectively. In the case of lists, we also have key and value pairs. Key was imaginary, which was the index, and the value was the element of that list, as shown in the following example:

```
>>> my_list = ["python","java"]
```

Here, the `python` string is the value and index zero is its key. In the case of lists, keys are only integers. In the case of dictionaries, keys can be of any type. We need to explicitly specify keys within the dictionary structure. Between each key and value pair, we need to put a single colon (`:`). Let's create one dictionary to make things clear:

```
>>> my_dict = {}
>>> type(my_dict)
<class 'dict'>
```

We used square brackets, `[]`, to create lists. But now, we will use curly braces, `{}`, to create dictionaries. We have to add items to the dictionary using the `key:value` pair. Let's create a simple dictionary, which will contain the names of people as keys, and their ages as values:

```
>>> info = {"Monica" : 32, "Joey" : 29, "Ross" : 55 }
>>> info
{'Monica': 32, 'Ross': 55, 'Joey': 29}
```

You can imagine a dictionary as a mapper between a set of indexes and a set of values. Here, indexes can be of any type, unlike integers for lists. In our `info` dictionary, we made keys as sets of strings and values as integers.

Now, let's observe the `info` dictionary that was printed in the preceding code. We can clearly see that the output sequence is not printed in the same order to that of the input. The element positions have been exchanged. In this case, where there are fewer elements, this might not be an issue. However, if we create a dictionary with 1,000 of items in it, you will clearly observe that the order of the output dictionary won't be the same to that of the input. In our example, we add the `Ross` key at the end of the dictionary, but while printing the same dictionary, we got `Ross: 55` added in the second position. So, you might be wondering, will it make any difference while accessing the elements of that dictionary? Not at all! dictionaries are arranged without an order, unlike that of a list. To access the elements of the dictionary, we have to use keys as an identifier. Accessing elements of the dictionary is quite similar to that of a list, but instead of putting indexes inside the square brackets, we put keys into it. For example, if you want to fetch the age of `Monica`, we use the following code:

```
>>> info["Monica"]
32
>>> info["Joey"]
29
>>> info["Chandler"]
KeyError: 'Chandler'
```

Instead of `IndexError`, we will get `KeyError`, which specifies that there is no such element inside the dictionary that has a key named `Chandler`. Thus, accessing the list can be an overhead because we have to track every possible index of that list. It won't be a problem for lists that are smaller in length, but imagine a list containing 10,000 or more elements. To overcome this expense, it's better to use dictionaries, since they are easier to access and the chances of running into exceptions is also meager. That being said, dictionaries are also not perfect data structures, and we will see the reason why most people prefer lists over dictionaries in the upcoming sections.

There is also another way of creating a dictionary, which is using the `dict()` method. Let's see how it is used:

```
>>> info = dict()
>>> info
{}
```

We have created an empty dictionary using the built-in `dict()` method. Now, let's see how we can add elements to that dictionary:

```
>>> info["Python"] = 1990
>>> info["C"] = 1973
>>> info["Java"] = 1970
>>> info
['Python': 1990, 'C': 1973, 'Java': 1970]
```

Since we have seen how to create our own dictionary using two methods, let's see how we can fetch every element of that dictionary. Since our data structure may contain many values, we must use loops to iterate over it. We'll look at how, w can loop through dictionaries in the next section.

Looping through dictionaries

Since dictionaries contain a finite number of keys and values, we can use a `for` loop to iterate over it. A `for` loop will traverse through the keys of the dictionary. The value of a particular key can be extracted by using square brackets, `[]`, and passing keys inside it. Let's see how this works:

```
>>> info = {'Python': 1990, 'C': 1973, 'Java': 1970}
>>> for key in info:
        print(key,info[key])

Python 1990
C 1973
Java 1970
```

In the preceding code, `info[key]` is going to extract the value of that key. The `for` loop will traverse through the keys of the dictionary, and iterating the `key` variable will store the key of the dictionary in each iteration. However, if we want to extract the key and value within the for loop, we will get `ValueError`. Let's see what I mean by this:

```
>>> for key,value in info:
        print(key,value)
ValueError: too many values to unpack (expected 2)
```

We get the preceding error because dictionaries are not iterables. However, we can convert it into another data structure, such as a tuple or lists so that we can fetch keys and values directly within the definition of the `for` loop. We will make this dictionary iterable by converting it into a tuple, which will be covered in the upcoming section about tuples.

Python provides a bunch of built-in methods in order to manipulate dictionaries according to your needs. For example, if you want to delete an item or insert an item to the dictionary, you don't have to make your custom logic to implement it; instead, Python has built-in functions for this. We will cover some of the most important dictionary methods in the next section.

Dictionary methods

Adding elements to the dictionary is easier, and we have already seen a couple of examples of this. Now, let's see how we can remove an element from the dictionary using the `pop()` method. For the argument that's given as a key to `pop()`, this method removes and returns an element from that dictionary. Let's look at a simple example:

```
>>> info = {'Python': 1990, 'C': 1973, 'Java': 1970}
>>> info.pop('C')
1973
>>> info
{'Python':1990, 'Java': 1970}
```

If you want to retrieve a particular value of the key, we can use the `get` method:

```
>>> info.get('Python')
1990
```

We can call the `values` method into the dictionary, which will return an object view that will represent all the values of the dictionary. Similar to `values()`, we can use the `keys()` method to print dictionary objects, which will represent all the keys of the dictionary:

```
>>> info.values()
dict_values([1990, 1970])
>>> info.keys()
dict_keys(['Python', 'Java'])
```

We can also use the `len()` method, which will return the number of items that are stored in the dictionary, as shown in the following example:

```
>>> len(info)
2
```

If you want to print a shallow copy of your dictionary, the `copy()` method can be used, as shown in the following example:

```
>>> old = { "Zero" : 0 , "One" : 1}
>>> new = old.copy()
>>> new
{'Zero': 0, 'One': 1}
```

Now,we have looked at some examples that have given us the knowledge to create our own dictionary and showed us how to access them using various dictionary methods. Now, let's explore tuples—another immutable data structure.

Tuples

Tuples are quite similar to lists in terms of processing, but they are immutable, unlike lists, which are mutable or changeable. We can store sequence of values within the tuple in a fashion that is similar to a list. Like we used `[]` to create a list, and `{}` to create a dictionary, we use `use()` to create tuples. The values that are stored in the tuple can be of any type, and each of these values are mapped by indexes in the same way as a list. The index of the first element of a tuple is zero, and it starts to increment with one, while at the same time, traversing from left to right. One of the merits of tuples is that they are iterables. Thus, we can convert non-iterable data structures, such as dictionaries, into tuple, so that we can extract key and value pairs within the loop declaration.

Let's create a simple tuple:

```
>>> numbers = (1,2,3,4,5)
>>> type(numbers)
<class 'tuple'>
```

We can also use the built-in method in Python to create tuples. We can create empty tuples using the `tuple()` method:

```
>>> numbers = tuple()
>>> numbers
()
>>> numbers = tuple('abcde')
>>> numbers
('a','b','c','d','e')
```

If you want to create a tuple with a single element in it, you have to add a comma after adding this element, otherwise Python treats it as a built-in data type, such as an integer or a string, as shown in the following code:

```
>>> odd = (1,)
>>> type(odd)
<class 'tuple'>
>>> even = (2)
>>> type(even)
<class 'int'>
```

Another way to create tuples is to add a comma between each item:

```
>>> numbers = 1,2,3,4,5,6,7
>>> type(numbers)
<class 'tuple'>
```

Most of the operations that we perform for lists also work in the case of tuples. In order to access the elements of a tuple, we use the square bracket operator and pass the index to it, as shown in the following example:

```
>>> numbers[0]
1
>>> numbers[-1]
7
```

Slice operations can be also performed for tuples in the same way as for lists. This operation will result in a range of values that can be extracted from the tuple. Have a look at the following example:

```
>>> numbers[3:]
(4,5,6,7)
>>> numbers[::2]
(1,3,5,7)
```

Tuples do not support item assignment, which makes it an immutable data structure, as shown in the following example:

```
>>> names = ("Jack","Cody","Hannah")
>>> names[0] = "Perry"
TypeError: 'tuple' object does not support item assignment
```

Now that you have learned about dictionaries and tuples, let's see how we can convert them from one to another. Because all the available data structures are not perfect, they have some pitfalls; therefore, the following section will be one of the most important sections that we have covered so far. This is where we will perform conversions between dictionaries and tuples.

Tuples and dictionaries

Dictionaries are not perfect iterables, which means we cannot use a `for` loop to extract keys and values directly from them. We can only extract keys from a dictionary, but if you want to extract the `key:value` pair, we have to convert it into another iterable data structure. Let's look at an example and observe the result, which shows the conversion from a dictionary into a list:

```
>>> person_address = {"Carl": "London", "Montana": "Edinburgh"}
>>> list(person_address)
["Carl","Montana"]
```

The direct conversion of a dictionary into a list does not preserve the values of the dictionary. It gives an object that contains only the keys of the dictionary. This information is useless due to a lack of values. Let's try to convert it into the tuple and see the result:

```
>>> tuple(person_address)
("Carl","Montana")
```

Instead of using the `tuple()` method to convert a dictionary into a tuple, there is another effective way. We can perform the same task using the `items()` method. This is used to return the dictionary object that contains the list of where the keys and values are stored in the nested tuples, as shown in the following example:

```
>>> person_address.items()
dict_items([('Carl', 'London'), ('Montana', 'Edinburgh')])
```

Now, we can iterate in this object using a `for` loop and fetch the keys and values in the same step of its declaration, as shown in the following example:

```
>>> for key,value in person_address.items():
        print(key,value)
Carl London
Montana Edinburgh
```

We have covered three powerful data structures up until now—lists, dictionaries, and tuples. Next is **sets**; an unordered structure that is considered iterable and mutable, but does not store duplicate elements.

Sets

Let's make things simple by comparing this data structure with the well-known concept of mathematics, which is a set. In mathematics, sets are considered a collection of distinct entities, which are normally considered objects. The numbers 1, 2, and 3 are objects independently, but when they are combined, they form a single set, which has a size of 3. They are no different in Python. A set in Python is a collection of objects, which are neither ordered nor indexed.

Python sets can be created using two different methods:

- The first one is similar to the way in which dictionaries are created; instead of key and value pairs, we will pass objects in their own right, as shown in the following example:

```
>>> num = {1,2,3,4,5}
>>> type(num)
<class 'set'>
```

- Another way is by using a Python built-in method, that is, `set()`, where you have to pass your sequence of objects in the form of list, as shown in the following example:

```
>>> set(['a','b','c'])
{'c','a','b'}
```

In the preceding code, we can see that the elements inside the curly braces are unordered. The order of objects, which we passed while creating the sets, is not preserved. They also do not support duplicate items in the sets. If there is repetition of the same elements multiple times within the set, only one element will be kept, and all the others will be removed from the structure, as shown in the following example:

```
>>> {"laptop","mobile","mouse","laptop","mobile"}
{'mouse', 'laptop', 'mobile'}
```

Sets are also non-indexed, unlike lists and tuples. If you want to access the elements of sets, you cannot use the indexing technique, as it will throw a `TypeError`:

```
>>> names = {"Ariana","Smith","David"}
>>> names[0]
TypeError: 'set' object is not subscriptable
```

Since sets are iterables, we can only access them through loops. The appropriate loop will be the for loop because we do not have to worry about the terminating point while using it:

```
>>> names = {"Ariana","Smith","David"}
>>> for name in names:
        print(name)

Ariana
Smith
David
```

Now that we have seen how to create and access sets of our own, let's delve into the basic methods of sets that are available so that we can manipulate their structure.

Set methods

Sets are mutable, but once they have been created, you cannot change their items; rather, you can add or delete items from that set. It is quite similar to a list, but it is ordered. Now, let's start this topic with the most commonly used methods of Python sets:

- We can add single and multiple items to the list, and there are two ways of doing this. The `add()` method will insert only a single item into the set at any time. On the other hand, the `update()` method will add multiple items to the set at the same time. The addition of the elements will be unordered, and they may be inserted at any position:

```
>>> favorite = {"Java","C","C#"}
>>> favorite.add("Python")
>>> favorite
{'Java','C#','Python','C'}
>>> #for update method
>>> favorite.update(["Python","JavaScript","R"])
>>> favorite
{'Python','Java','R','C#','C','JavaScript'}
```

- There are many ways of removing elements of sets. Methods such as `remove()`, `discard()`, and `pop()` can be used. If the item that you want to remove from the set does not exist, `remove()` will throw an exception, which goes by the name of `KeyError`, but in the case of the `discard()` method, our code won't run into any errors, as shown in the following example:

```
>>> favorite.remove('C')
>>> favorite
{'Python','R',"JavaScript','Java','C#'}
```

```
>>> favorite.remove("NodeJS")
KeyError: 'NodeJS'
>>> favorite.discard("NodeJS")
>>> #no error
```

- We can also use the `pop()` method to remove elements from a set. `pop()` will remove only the last element from the set. However, we don't know which element will be the last in the set, since it is unordered and non-indexed. Thus, using `pop()` will be dangerous, as we won't know about the removal of specific items. `pop()` will return an item that is removed from the set, as shown in the following example:

```
>>> favorite.pop()
'R'
```

- If you want to delete each and every element from the set, two methods can be used, but the results of these operations are slightly different. The `del` keyword can be used, along with the name of the set, in order to remove an entire element of a set along with the set, structure. On the other hand, the `clear()` method is used to empty the set, but its structure won't be completely removed:

```
>>> favorite.clear()
>>> favorite
set()
>>> del favorite
>>> favorite
NameError: name 'favorite' is not defined
```

- We can also perform operations such as union, an intersection between sets, just like we do in mathematics. The union operation returns a set that contains all the elements from the original set, and all the items from the specified set. The set removes duplicate items. If any item is present in more than one set, it will be added only once in the resulting set. You can perform union between multiple sets by separating each of them with a comma:

```
>>> set_1 = {1,2,3}
>>> set_2 = {3,4,5}
>>> set_1.union(set_2)
{1,2,3,4,5}
>>> set_3 = {4,5,6,7}
>>> set_1.union(set_2,set_3)
{1,2,3,4,5,6,7}
```

- We have the `intersection()` method, which will result in a set of items that is common between multiple sets, as shown in the following example:

```
>>> set_1 = {'a','b','c'}
>>> set_2 = {'b','c','d'}
>>> set_1.intersection(set_2)
{'b','c'}
```

In the previous sections, we covered the fundamentals of Python. We have established a strong foundation of core programming up to this point, but we are not capable of building an advanced game yet.

In the upcoming sections, we will delve into the most important concept, not only for Python, but for programming in general, that is, **functions**. After that section, you will possess procedural programming power, which will be helpful when we cover every advanced game that we will build from that point onward.

Functions

First of all, let's recall all the topics that we have learned so far and observe procedural programming functions and why they are needed in the first place. We learned how to create multiple lines of statements using variables, numbers, modules, conditionals, and looping. However, we didn't stop there; we covered all the fundamental data structures of Python, such as lists, dictionaries, tuples, and sets. This programming paradigm will result in an abundance in lines of code, and sometimes we may need to call the same code again and again. Have a look at the following example:

```
>>> 3 + 5
8
>>> 6 + 7
13
```

In the preceding code, we are adding two digits. Every time we perform an addition, we need to write two digits, followed by addition operators. Instead of doing the same task for many addition operations, why not make a single statement, which can perform addition, and put that statement into the scope where we can call it multiple times? This scope represents functions. We can invoke the execution of this statement multiple times by calling these functions. Let's make a function that can add any two numbers:

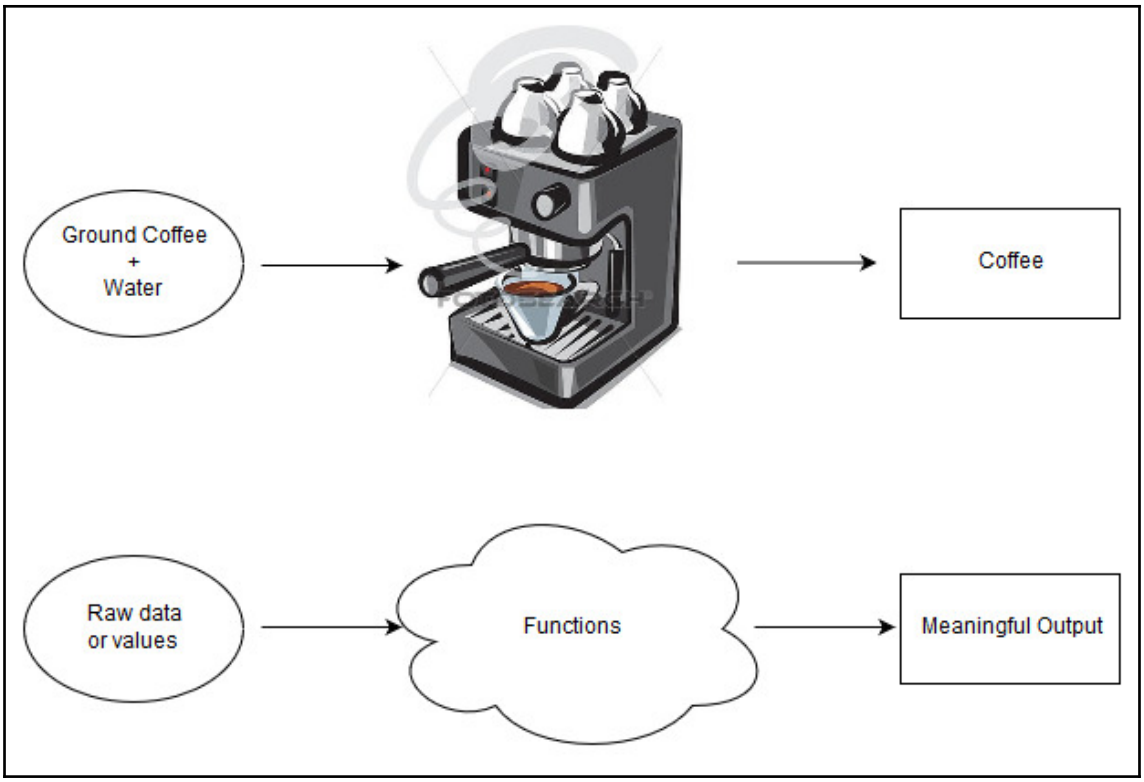
```
>>> def add(a,b):
        print(a + b)
```

In the preceding code, we defined the function with `add`. The `def` keyword, along with a name, is used to specify the Python parser in order to create functions. Inside the scope of the function, we can add multiple statements. Now, instead of manually adding two digits every time, we can call this `add` function to perform addition between any digits. So, this part of code is usable for operations that can add any two digits. The first task is to declare the function, which is what we just did; the next task is to call that function. You won't execute any operation that is inside that function until you call that function. You have to use the same function name in order to invoke that function. Now, if you want to perform an `add` operation, you need to call it with same the signature, `add`, and pass two values as an argument to it. If you pass a number, it will be passed as an argument to that function call:

```
>>> add(4,5)
9
>>> add(10,11)
21
```

In the preceding result, each digit that is inside parentheses is passed to the function argument: a and b. At the first operation, `add(4,5)`, 4 is passed as a value to variable a, and 5 is passed as a value to variable b.

Let's compare these functions with the following coffee machine. We feed raw materials such as coffee beans, sugar, and water to the coffee machine, which will process those materials, and provide us with a coffee. In the same way as a coffee machine, functions also take raw arguments containing values as input. These arguments will be used for processing, which is done inside the function, and gives us meaningful results. Sometimes, functions do not return anything; we call these `void`:



We have look at couple of examples where we called the functions by name, but their declaration was carried out internally by Python. For instance, take the example of the `print()` method. We used this function to print any messages to the user on the Terminal, but we didn't define it using the `def` keyword; we simply called it because it's a built-in function. Thus, if you are using any functions such as `print()`, `input()`, or `type()`, you are calling that function by passing an argument inside its parentheses. You can see the implementation of `print()`, or any other built-in method of Python, by taking a tour of the official Python documentation. While calling `input()` or `print()`, we pass a string as an argument inside its parenthesis. Let's look at an example of a function call:

```
>>> type('a')
<class 'str'>
```

In the preceding code, we made a call to the function with `type`. The arguments are passed inside the parenthesis of the function. We can pass as many arguments as we like as an expression inside the parenthesis, but we have to make sure that we pass only the required positional arguments. In the function declaration, if we made a function with two parameters, while calling, we should pass the same amount of arguments. Otherwise, it will throw us an error, as shown in the following example:

```
>>> def add(a,b):
        print(a+b)
>>> add(3)
TypeError: add() missing 1 required positional argument: 'b'

>>> add(3,4,5)
TypeError: add() takes 2 positional arguments but 3 were given
```

Thus, we can conclude that functions take an argument, execute some statements based on that argument, and return a result. In our `add(a,b)` function, we printed the result inside the function, but instead of printing it inside the scope of the function, we used the `return` keyword in order to return a `result` from the function:

```
>>> def add(a,b):
        c = a + b
        return c

>>> result = add(3,5)
>>> result
8
```

Thus, we have two types of functions. One prints the results inside the scope of a function, rather than returning results from it which are normally void. Although Python has nothing nomenclature for void functions, other programming languages call these void functions, which means they return nothing. Another type will yield a return value of the function. Those return values should be captured when a function is called, just like in the code: `result = add(3,5)`. The value of `result` is the return value of the function.

You may encounter a situation where a function has to return multiple value. We can use the tuple structure to return multiple values from the function. Let's take a look at the following simple example:

```
>>> def result(a,b):
        print("Before Swapping: ")
        print(a,b)
        print("After Swapping: ")
        return b,a
>>> result(4,5)
Before Swapping:
```

```
4 5
After Swapping:
(5, 4)
```

We'll learn about the concept of *default arguments* in the next section. Learning about this will help us build more flexible functions and so this is an important topic.

Default arguments

During the function call, we usually pass a value as a positional argument to the respective parameters. However, if we make a mistake by passing one less or one more than is required, our program will run into an exception. Thus, it is always good practice to specify some arguments as default:

```
>>> def msg(str1,str2):
        print("I love {} and hate {}".format(str1,str2))

>>> msg("Python")
TypeError: msg() missing 1 required positional argument: 'str2'
```

Now, let's look at the power of default arguments. Before using them, you should remember that default arguments must be placed at the end of the argument order. The syntax for creating a default argument is `argument_name = value`. In the preceding example, if you want to make `str1` the default argument, it should be placed after `str2`, otherwise you will get a syntax error from the Python interpreter, as shown in the following example:

```
>>> def msg(str1 = "Python",str2):
        print("I love {} and hate {}".format(str1,str2))

SyntaxError: non-default argument follows default argument
```

As the error message clarifies, we cannot specify a default argument to the left positional one. They should be followed by non-default arguments, as shown in the following example:

```
>>> def msg(str1,str2 = "Java"):
        print("I love {} and hate {}".format(str1,str2))

>>> msg("Python")
I love Python and hate Java
```

In the preceding example, have a look at the part where we called the function with only one argument in it. Now, that argument is a positional argument. As it is in position one, it will be passed to the first parameter of the function. Thus, the `Python` value will be passed to the `str1` parameter. After the `Python` value, we passed nothing. Instead of running into `TypeError`, we were able to get a proper result. This is the power of default arguments. However, if you pass another value to that default argument at the time of the function call, the default argument value will be overwritten with a new one:

```
>>> msg("Python","C")
I love Python and hate C
```

Up until now, we were able to call the function with a few positional arguments, such as a and b. But what if we have to make a function that can add 200 numbers? Calling a function such as `add(a,b,c,d,..)`, in which each variable represents one number, is not possible. We will have a shortage of variables, too, because for 200 numbers, we have to maintain 200 variables. So, the most efficient way would be to pack all of those arguments into one, and pass it as a single argument to the function. Then, the function will unpack that variable and perform the relevant operations. We can use the list data structure as a variable to store those multiple values. We'll look at how to pack and unpack normal and keyword arguments in the next section.

Packing and unpacking arguments

Let's take a simple example that will help us understand why we need this method of packing and unpacking in the first place. In this example, we are going to add numbers:

```
>>> def add(a,b):
        result = a + b
        return result

>>> print(add(4,5))
9
```

Our code works fine for fewer numbers, maybe up to 10 values. Little modification should be done with a small increase in numbers, but that's fine. However, what if we have 100 numbers? Tracking each of these numbers into variables is not possible and not effective. Our code would also look unprofessional. Now, here comes the crazy feature of Python that goes by the name of packing the arguments. Here, we are taking about arguments, that is, normal arguments such as list and tuple. We can make a list that contains multiple numbers. Let's see how we can make a function that can add multiple numbers using the case of packing the arguments:

```
>>> def add(*args):
        result = 0
        for item in arg:
                  result = result + item
        print(result)

>>> add(1,2,3,4,9,4,2,5,5,8)
43
```

Let's observe the code that we have written here. The `*arg` convention is used for packing the arguments. Here, `args` refers to arguments, which is the default naming convention for arguments in Python, but you can name it anything as long as you follow the rules and conventions of the variable naming pattern. A single asterisk (`*`) is essential, which shows that we are packing into a single argument. We are packing every item into `args`; therefore, `args` will be built as a list. We know that the lists are iterable, which allows us to loop within it using for loop. Now, while calling the function, we do not have to worry about any positional arguments or even parameters that contain values. Every piece of data that is passed during the function call will be packed into the list using this method. Now, we are not restricted to using parameters that assign values to specified positional arguments. We can perform these packing argument techniques for every data type, or even for structures.

Unpacking arguments also works in a similar way to that of packing. We use a single asterisk abreast of the argument to specify that we are using the unpacking technique. Here, the argument must be a list, a string, or another structure that represents collections of items. Have a look at the following example:

```
>>> print(*"Python")
P y t h o n
```

Since the argument is passed as a string (`Python`), we unpacked it so that every element is printed separately, with some spaces. You can also unpack elements of a list structure as follows:

```
>>> numbers = [1,2,3,4]
>>> print(*numbers)
1 2 3 4
```

So, we can pack and unpack normal arguments using a single asterisk, but in order to pack and unpack keyword arguments, we have to use a double asterisk. The syntax that is used for packing and unpacking a keyword argument is `**kwargs`. Just remember to use a single asterisk for normal arguments, and a double asterisk for keyword arguments. `args` represent arguments and `kwargs` is the naming convention for keyword arguments. We'll see some examples of packing and unpacking keyword arguments in the next section.

Packing and unpacking keyword arguments

Keyword arguments refer to dictionaries. Dictionaries cannot be packed and unpacked in a similar way to normal arguments such as lists or tuples. Dictionaries contain key and value pairs; thus, they cannot be packed and unpacked in the normal way. To distinguish them from normal arguments, we use a double asterisk. `**kwargs` is used to pack all the elements of a dictionary into a single argument. However, we know that dictionaries are not iterable, or in other words, we cannot loop inside dictionaries and fetch key and value pairs directly. In order to retrieve key and value pairs, we need to convert `kwargs` into a tuple using the `items()` method. We have seen its implementation in the preceding section. Let's look at a simple example of how to implement packing keyword-arguments:

```
#code is written as script
pack_keyword_args.py

def about(name,age,like):
    info = "I am {}. I am {} years old and I like {}.
".format(name,age,like)
    return info

dictionary = {"name": "Ross", "age": 55, "like": "Python"}
print(about(**dictionary))

>>>
I am Ross. I am 55 years old and I like Python
```

In the preceding example, we did two things: we made a dictionary that will be packed into a single argument using `**dictionary`, and passed each value to the positional arguments of function. In the dictionary definition, the keys of the dictionary must be the same as the parameters that are used while making the function, that is, `name`, `age`, and `like`. Even single typos will result in `TypeError`.

Now, it's time to cover unpacking keyword arguments. The syntax will be similar, which contains a double asterisk and is followed by the dictionary name, or `kwargs`. Since we are unpacking, we have to add `**kwargs` as a parameter of function, because unpacking has to be done inside the function. Let's look at the a simple example to clarify this:

```
#unpacking_key_args.py
def about(**kwargs):
    for key, value in kwargs.items():
          print("{} is {}".format(key,value))

about(Python = "Easy", Java = "Hard")

>>> #output
```

```
Python is Easy
Java is Hard
```

While calling the `about` function, we passed a value to the argument, like we normally pass in the case of a normal function. For example, `Python` is the argument and it has a value of string. Now, this value is passed to the parameter of the `about` function. However, there is no parameter with the name of `Python` or `Java` within the function parenthesis. Instead, there is `**kwargs`, which is going to convert these `argument_name = value` formats into the dictionary. This is a form of packing the argument. Now, while inside the function, we have to unpack it. At this time, we know that `kwargs` is a dictionary, which is not iterable. We cannot fetch its `key:value` pair without converting it into a tuple or a list. One easy way to convert a dictionary into a tuple is by using the `items()` method. Now, after converting a dictionary into a tuple object using the `items()` method, `kwargs` looks like this:

```
>>> kwargs.items()
dict_items([('Python', 'Easy'), ('Java', 'Hard')])
```

Now, we are looping around these items of the tuple object, and each object contains a key and a value separated by a comma. Thus, for each iteration, we get key and value pairs, and we print it by formatting it properly.

Now, we possess the knowledge that will not only help us create our own functions, but also modify them according to our needs. If you want to make your program more reusable and sturdy, methods such as packing and unpacking arguments must be used. After this broad concept of functional programming, it's time to explore three important functions in Python: the anonymous, recursive, and built-in functions. Let's take a look at each of them one by one. We will begin with the *Anonymous function*.

Anonymous function

As the name suggests, these functions do not have any name or signature. In the same way that we used the name of the `add(a,b)` function to carry out an addition operation between two numbers, this `add` signature is invalid in the case of the anonymous function. If you recall the way we created a normal function using the `def` keyword, in the case of an anonymous function, we use the `lambda` keyword. Thus, anonymous functions are also called lambda functions. We need to remember two things while creating any function: arguments and expressions. Arguments are the independent and specific input to the function, whereas expressions are embedded inside the body of the function. In the case of the `lambda` function, we can pass any number of arguments, but only one expression. This implies that only one operation can be done with the `lambda` function.

Let's make a simple `lambda` function in order to grasp this information easily:

```
>>> square = lambda x: x**2
>>> square(8)
64
```

In this example, `square` is the container for the result. Since the `lambda` function does not contain a unique signature or name, we should pass an argument as a value using this container, that is, `square`. Here, the syntax to use for the `lambda` function is as follows:

```
lambda arguments: expression
```

Notice the `arguments` and `expression` names; we cannot add multiple statements inside the `lambda` function. If we try to execute multiple statements inside the `lambda` function, we will run into the following error:

```
>>> result = lambda x, y: x//y, x%y
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    result = lambda x, y: x//y, x%y
NameError: name 'x' is not defined
```

We passed `x,y`, that is, multiple arguments, which is completely valid, but two expressions, `x//y` and `x%y`, are not executed by `lambda`. We will use these `lambda` functions for creating games in the upcoming chapters. Since we have many things to cover in this chapter, and we are running out of space, I would like to end this topic right here; however, I highly urge you to practice these types of functions a little more. You can always use the Python documentation to help.

Let's look at another type of function: **recursion**—a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step, has a termination condition, and when the terminating condition is met, the program also terminates.

Recursive functions

In this section, we are going to uncover another programming paradigm, known as recursive programming. Recursion is a way of programming where a function will call itself multiple times until a particular condition is met. Inside the body of the function, we will call the same function itself, which makes it a recursion. It is somewhat similar to nested conditionals, where we have another scope of `if..else` inside single `if` conditionals.

Recursion should have a base or a terminating condition in order to specify the stopping criterion for the program. Without a base condition, our recursive program is not going to operate viably. If the base condition is not met at the point of program execution, the recursive program will result in an infinite loop. Let's jump to a simple programming example to observe the working principle of recursion:

```
>>> def factorial(number):
        if number == 1:
            return 1
        else:
            return number*factorial(number-1)

>>> factorial(4)
24
```

Let's explore the preceding example to uncover interesting facts about recursive programming. Printing a factorial of any number is a simple example that we can refer to while learning about recursive programming. In the preceding program, we have a base or terminating condition: when the number is one, we return one. This is not a random statement; rather, it is a mathematical pattern for finding a factorial number. Have a look at the following example:

```
To find factorial of 5= 5! = 5*4*3*2*1! = 5*4*3*2*1 = 120
```

For any number, the process of finding a factorial ends after we encounter one. Thus, it is a base condition, and whenever our program triggers it, we can terminate our program. Inside the scope of the `else` part of the program, we are calling the factorial function again, but with a different argument. You can observe the example where we found a factorial of five; each time we go to the next step, we are decreasing that number by one and multiplying it with the current number, which represents this statement:

`>>> number*factorial(number-1)`. This condition is called a recursive case, which leads to recursion.

So, there are two ways of making logic with Python: with fundamental logic using loops and conditionals, or with recursion. Sometimes, it will be hard to get a solution using brand new logic, and in such situations we give recursion a try. Although recursion code looks simpler and cleaner, it is an expensive call in comparison to other code, because it takes a lot of time and space during computation. Now, let's talk about a faster and cheaper way to execute an operation using built-in functions. We have covered so many built-in functions already, such as `max()`, `min()`, and `len()`. Thus, this section will be rather easier to follow.

Built-in functions

Python comes with multiple built-in functions that are available for us to use directly in our programs. We shouldn't have to import them, or make any extra effort to execute them. For example, we have `print()`. We have unknowingly used so many built-in functions before, but they are also a type of function. The only difference is that they are made by Python creators. They are fast, and more importantly, using them makes our code simpler and cleaner. Just think like this: adding two numbers using our own custom method may take a minimum of three lines of code, but using a built-in function, we can do it in a single line, using the `sum()` function.

You can check each and every built-in function by taking a tour of the Python official documentation. Secondly, we can also get information containing a list of built-in functions within our Python shell. You can type the following `>>> dir(__builtins__)` command in order to get a list containing 68 built-in functions. We have already seen a few of the most important among them., for example, the `type()` method and the type-casting techniques. They all are achieved using built-in functions.

I won't be covering every built-in function in this section, as that is not the actual motive of this book; instead, we will be going directly to the next topic, which will be an interesting one since we are going to modify our tic-tac-toe game using the functions and data structures that we have learned about so far. However, I highly encourage you to take a prudent step forward by learning about a few built-in functions on your own. They may not be important just yet, but they will surely come in handy at some point during your career.

Now that we have learned about data structures and functions, we will use them to modify the previously built tic-tac-toe game by adding intelligence to it. We will cover this in the next section.

Adding intelligence into our game

We have made multiple modifications throughout this chapter, such as adding conditionals and looping to enhance code structure and processing. However, this not isn't perfect yet. In this section, we are going to modify our tic-tac-toe game using the functions and data structures that we have learned about in this chapter.

Since the function is going to make our code smaller in length by eliminating the repetition of code, and also debugging it, it will be also easier to make changes in the code at a later stage; you can simply redirect to a specific function instead of traversing the entire program. So, these two features will be helpful for us while printing the game board into the terminal. If you recall the code that we wrote in the previous chapter, code for printing board layouts was used repeatedly. Now, we can create a function that will have all the code we need inside it so that we can print the layout of the board, and we can call it any time and anywhere within the code.

The next implementation on our code will be subsuming intelligence for our tic-tac-toe game. Until this moment, if you run your tic-tac-toe game, you will find that it can be played by two players. However, both players should be using the same computer, and they should play it by toggling their turn. Now, we are going to add computer intelligence that can play our game as one player. We are literally making a game where the player can play against the computer.

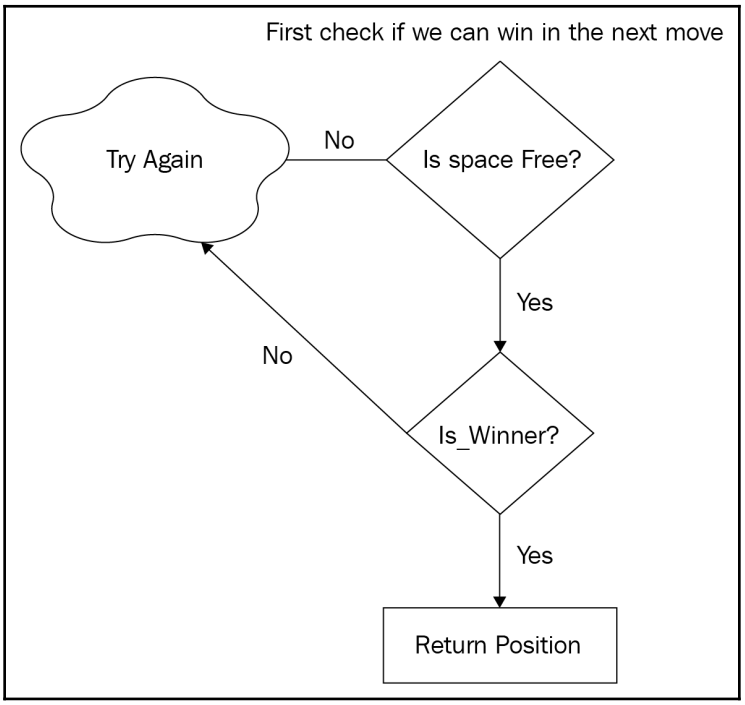
As usual, we will start by brainstorming the essentials of the game, and we will gather critical information about the game layout and models.

Brainstorming and information gathering

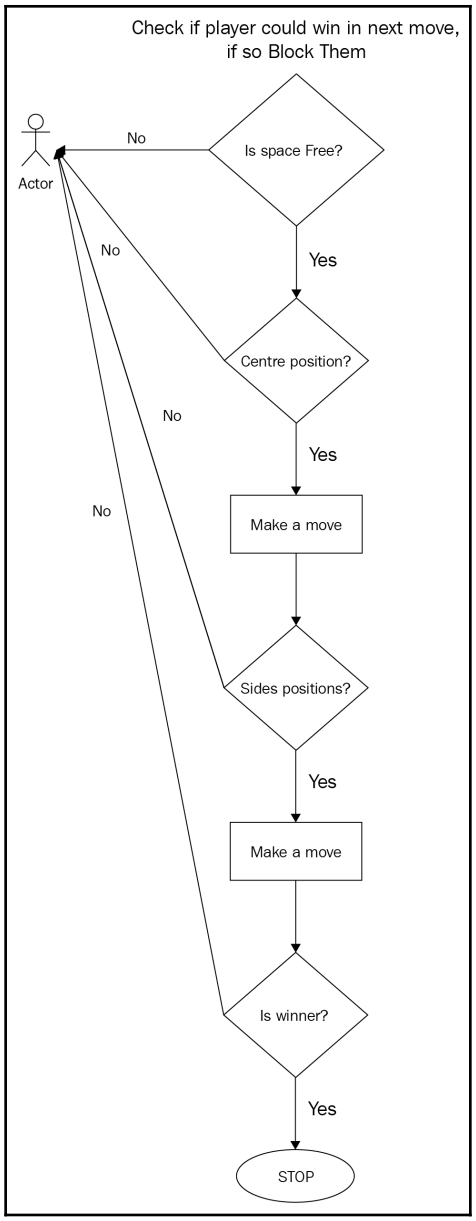
The term *artificial intelligence* is very notorious in the tech world, but if you inspect the depth of it, it is a bunch of modules and conditionals that determine the flow of agents. Here, agents can be anything that make decisions, such as machines, humans, and robots. All these agents perform actions that can produce the most desirable results. In our game of tic-tac-toe, the agent is a computer player, and it should take actions that can beat our player in the game. We have a dedicated chapter to learning about AI and its rational agents, which will be covered after we finish learning basic game programming. However, in this section, we are going to create a simple AI that can decide on the most favorable move in order to beat a human player, or even end the game in a tie most of the time.

We are going to take on the approach of procedural programming in order to add intelligence to the system. Don't get overwhelmed with the term procedural programming—it is just a way of making and using functions to achieve a goal. One thing you must remember is that every function should perform only one task. For example, we can make the `print_board()` function, which will just print the layout of the game every time we call it. This `print_board()` function is not going to take input from a user, or make any player a winner. Thus, the existence of functions should be preserved by performing only one modular task. We can also make the `is_winner()` function, which will check whether any player is the winner.

The following diagram shows how a simple algorithm can be made for our game. Here, we can see how we can check for the positions on the tic-tac-toe board so that the computer's next move will produce the best result; something closer to winning the game, or in the worst case, making the game be a draw instead of the computer losing:



The following diagram show the procedures that we need to complete in order to implement the second part of the algorithm, where we will track every occupied position of the human player and check whether they could win with their next move. If they can win, we will block those positions. We will also occupy the center and side positions so that no human player can win the game easily:



Now, we have formed the basic algorithm so that we can start writing the code that can implement basic intelligence in our game. We will use this knowledge in the next section, *Implementation of models for intelligence,* in order to address the model for intelligence.

Implementation of models for intelligence

First of all, let's refine our code using functions; let's create a function named `printBoard()`.This function will contain lines of code that will print the board layout of our tic-tac-toe game:

```
#tic_tac_toe_AI.py

def printBoard(board):
    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')
    print('---------------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('---------------')
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')
```

The previous code will print board's layout; if you want to execute the statements that are inside the function you have to call it. Here, we have to call it using the `board` argument, which is the list containing all the positions of the board, that is, ten empty places, `[' ']*10`. Let's call this function and observe the result:

```
>>> board = [' ']*10
>>> #calling the function:
>>> printBoard(board)

      |     |
      |     |
      |     |
-----------------
      |     |
      |     |
      |     |
-----------------
      |     |
      |     |
      |     |
```

Now, it's time to make a function that can check whether any player is the winner or not. We are not making any brand new logic here; instead, we are putting all the statements that we made in the preceding chapters inside the scope of the function. Now, each time any user makes a move in the board, we can call this function to check whether that particular player is the winner or not. Thus, functions can remove repetition or duplication of code. Let's use the `isWinner()` method to check whether any user satisfies the condition to become the winner:

```
#tic_tac_toe_AI.py
#after printBoard(board) function

def isWinner(board, current_player):
    return ((board[7] == current_player and board[8] == current_player and
board[9] == current_player) or
    (board[4] == current_player and board[5] == current_player and board[6]
== current_player) or
    (board[1] == current_player and board[2] == current_player and board[3]
== current_player) or
    (board[7] == current_player and board[4] == current_player and board[1]
== current_player) or
    (board[8] == current_player and board[5] == current_player and board[2]
== current_player) or
    (board[9] == current_player and board[6] == current_player and board[3]
== current_player) or
    (board[7] == current_player and board[5] == current_player and board[3]
== current_player) or
    (board[9] == current_player and board[5] == current_player and board[1]
== current_player))
```

In the preceding code, the parameter to the `isWinner` function is `board`, which contains the positions of the board layout and the player's letter; either `X` or `O`. We are reusing the same code that we wrote in the previous chapter, with small modifications. This method is going to return a Bool type of either `True` or `False`, and we will call it every time the player makes a new move in the game. We are checking entire rows, columns, and diagonal positions of the board layout using this method, and if any user occupies it, this will return `True` or `False`.

In the game of tic-tac-toe, we get to move the player in the form of the position and we assign the player's character's either `X` or `O`, to it. We have seen its implementation in the previous chapter. Here, we are going to make a separate function that will assign a value to the position. In the following code, the board represents the layout of the game containing the positions; `current_player` is either `X` or `O` and `move` is the input from the user:

```
def makeMove(board, current_player, move):
    board[move] = current_player
```

Now, it's time to make the computer play our game. Let's recall the algorithm that we made in the preceding section. We are going to perform multiple checks: whether the computer can win in the next move or not, and whether a human player could win in the next move. If so, we will block the winning position. We cannot perform these operations in the real board layout game, because we don't want our board layout to be populated. Thus, we are going to make a copy of the board layout so that we can perform these checking operations in the new clone board layout. Let's make a function to copy the original board layout:

```
def boardCopy(board):
     cloneBoard = []
     for pos in board:
         cloneBoard.append(pos)
     return cloneBoard
```

After we clone the original board, we have to check whether there are free spaces available for the computer to make a move. Let's make a function to check the available free spaces inside the board layout:

```
def isSpaceAvailable(board, move):
     return board[move] == ' '
```

`isSpaceAvailable` returns a Bool type: either `True` or `False`. It will return `True` if the move is available on the passed board layout. If the position is already occupied by any player, it will return `False`.

Now, it's time to get into the main part of our topic: making the computer play our game. Let's create a function named `makeComputerMove()` and pass the board argument and a `computerPlayer` character to it. Here, the board represents our board layout containing all the positions, and `computerPlayer` is a character, either `X` or `O`:

```
#tic_tac_toe_AI.py

 def makeComputerMove(board, computerPlayer):
     #part 1
     for pos in range(1,10):
         #pos is for position of board layout
         clone = boardCopy(board)
         if isSpaceAvailable(clone, pos):
             makeMove(clone, computerPlayer, pos)
             if isWinner(clone, computerPlayer):
                return pos
```

In the preceding code, `#part1`, we checked if the computer can win in the next move or not. At first, we looped into the entire position of the board layout and made a clone of the board using the `boardCopy` function. After that, we passed every position from 1 to 10 to check for space availability. We checked if that move was going to make the computer player the winner or not by using the `isWinner` function, and returned that specific move as the position if that was the case. This part of the code makes our computer player intelligent enough to decide the next move, based on its favorable prediction.

In the process of adding intelligence to our computer player, the next step is to keep track of the human player's movements. On doing so, we can make a smart move on the board so that the player won't win the game easily. In addition, if the human player has occupied two positions on the row of the board, we can make our move to block the third position. Let's write `#part2` of our `makeComputerMove()` function. In order to check whether the human player is going to win, we have to play the game as a human, but virtually. We can do this without affecting the original board because we can play as a human within the copy of the board. Now, to check whether the human player is going to win, we have to get a player letter, that is, either `X` or `O`. We can make a condition to check whether the human player is `X` or `O`. After getting that letter, we can play as a human virtually on the copy of board game, but remember that we are coding for the computer player:

```
def makeComputerMove(board, computerPlayer):
     if computerPlayer == 'X':
         humanPlayer = 'O'
     else:
         humanPlayer = 'X'

     #add part1 code here
     #now check if human player will win on next move or not in part2:
     #part2
     for pos in range(1,10):
         clone = boardCopy(board)
         if isSpaceAvailable(clone, pos):
             makeMove(clone, humanPlayer, pos)
             if isWinner(clone, humanPlayer):
                return pos
```

The code that we have just written is going to add a smart move for the computer player. We are making the computer play the tic-tac-toe game as a human, but virtually. We are checking if the human will win on the next move or not. If it they are, we will return that move so that the computer will place its letter in that position to block the human from winning the game.

During the brainstorming and information gathering processes, we made a flowchart to track the activities that will embed intelligence into our computer player. We executed two of them: checking the best move to win, and blocking the next best move of the human player. We can also make the computer player smarter by making an initial move, which the human player would normally do. For example, when we play tic-tac-toe, we start by taking the center position, because we think it is the best position to start with. So, why not make the computer to do that too? Let's write some code where the computer will check the availability of the center position on the board and reserve that position accordingly:

```
def makeComputerMove(board, computerPlayer):
     #add part1
     #add part2
     #Occupy center position if it is available
     #part3
     if isSpaceAvailable(board, 5):
         return 5
```

We can make this computer player even smarter by checking the availability of the corner positions, too. The corner positions on the board are `[1,3,7,9]`. Since there are four corners on our board, we maintained the list to track them. Now, let's create a new `getRandomMove()` function, which will take the board and moves as arguments. The moves argument will be in the form of a list, such as the corner positions:

```
#tic_tac_toe_AI.py
 import random
 def getRandomMove(board, moves):
     availableMoves = []
     for move in moves:
         if isSpaceAvailable(board, move):
             availableMoves.append(move)
     if availableMoves.__len__() != 0:
         return random.choice(availableMoves)
     else:
         return None
```

A lot of things are going on in the preceding code, so let's break things down to make it simpler. First of all, this method is going to take moves which will be in the form of a list, that is, `[1,2,3,4,5]`; among them, we have to choose only one element using this function. However, the elements of this list are not only numbers; they are moves or positions of the board layout. Thus, we have to check for the availability of spaces for each move of that list. If there are available spaces, we append that move to a new list called `availableMoves`. After the filtering is done, we perform conditionals to check whether there are any available moves or not.

The `>>> availableMoves.__len__() != 0` expression is the same as `len(availableMoves)`, which is going to return the length of the list. We call these implementations (`__len__()`) data models, and we have an upcoming dedicated chapter that will cover them. If the length of `availableMoves` is zero, we are going to return `None`. But if it is not zero, we will execute an expression. Let's break this expression down into fragments:

- `import random`: If you recall the topics of `Chapter 2`, *Learning the Fundamentals of Python*, where we imported the math modules to perform mathematical computations such as square root and factorials, we imported math modules using the `import math` command. Now, we are importing a random module, which means that we can use methods that are defined inside it using this module. The syntax for calling a method from a random module is `random.method_name()`.
- `random.choice()`: The choice method is going to pick up one random element from the list of elements that it has been called upon with. For example, the execution of the following command will give one random value from the range of values that's been passed into it:

```
>>> import random
>>> random.choice([1, 2, 4, 5, 6])
5
>>> random.choice([1, 2, 4, 5, 6])
2
```

3. We passed `availableMoves` into it so that the `choice` method would pick any one of the moves randomly. This is essential for our gameplay because sometimes the computer must make decisions randomly.

Now, let's call this `getRandomMove` function within the `makeComputerMove` function. If you take a look through the code of the `makeComputerMove` function, we have added a statement that will help the computer occupy the center position. What about corner positions? They are also an important position of the tic-tac-toe game. If we occupy the center and corner positions of the board, our computer will have a high chance of winning the game. Thus, we have to enhance our code, which will make the computer player capture the corner positions. Since the corner positions are `[1, 3, 7, 9]` we have to pass this as a list argument to the `getRandomMove` function, which we have just created:

```
#tic_tac_toe_AI.py
 def makeComputerMove(board, computerPlayer):
     #add part1
     #add part2
     #add part3
```

```
        #code to occupy corner positions
        move = getRandomMove(board, [1, 3, 7, 9])
        if move is not None:
            return move

        #moves for remaining places ==> [2, 4, 6, 8]
        return getRandomMove(board, [2, 4, 6, 8])
```

In the preceding code, we added code that will get random moves on any of the corner positions. We have covered the player moves for the center `position [5]`, and for corner positions `[1,3,7,9]`; now, we are left with the side positions, `[2,4,6,8]`. We made a call to the `getRandomMove` function, which will choose any one random move from the passed list.

We have learned so many things in the preceding sections such as loops, conditionals, and many more. In the next section, we are going to write some code that will use them to control program flow. We will call it, the **main function**.

Controlling program flow with main function

We have written a bunch of functions such as `makeComputerMove`, `isWinner`, and many more, but they have not been called anywhere. We know that the function is not going to execute code inside it until we call it. Thus, we will make new function, that will take care of the flow of the program. We normally name it the main function, but you can literally name it anything you like. The code that we have wrote in the previous chapters, such as for the main game loop or toggling player turn, will be embedded inside this main function. The only function that needs to be called explicitly is this main function. Let's create one now:

```
#tic_tac_toe_AI.py
 def main():
     while True:
         board = [' '] * 10
         player, computer = 'X', 'O'
         turn = "human"
         print("The " + turn + " will start the game")
         isGameRunning = True
         while isGameRunning:
             if turn == "human":
                 printBoard(board)
                 move = ' '
                 while move not in '1 2 3 4 5 6 7 8 9'.split() or not
                  isSpaceAvailable(board, int(move)):
                     print('What is your next move? (choose between 1-9)')
```

```
                    move = int(input())
                makeMove(board, player, move)
                if isWinner(board, player):
                    printBoard(board)
                    print("You won the game!")
                    isGameRunning = False
            else:
                #computer turn
```

We have written the preceding code multiple times before, such as when printing the board, toggling the players, and creating a winner. The difference is, here, we are using functions. We have one task related to one function, such as `isWinner`, which checks if a player is the winner or not, and instead of writing an entire piece of code to check the winner, we write it once and use it in our main function. You can see that we have written some code take input from a user as a move value to the board game. We can make a function to take an input from a user. Let's make it now, and make the main function cleaner and more readable:

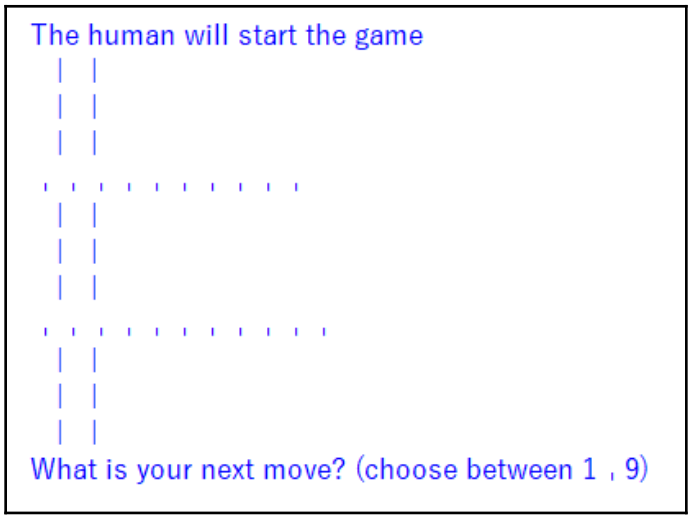
```
def makePlayerMove(board):
    move = ' '
    while move not in '1 2 3 4 5 6 7 8 9'.split() or not
     isSpaceAvailable(board, int(move)):
        print('What is your next move? (choose between 1-9)')
        move = int(input().strip())
        return move
```

Now, let's add this newly created function to the main function. We will also complete the `else` part of the code, which will make the computer play our game:

```
def main():
    while True:
        board = [' '] * 10
        player, computer = 'X', 'O'
        turn = 'human'
        print("The " + turn + " will start the game")
        isGameRunning = True
        while isGameRunning:
            if turn == 'human':
                printBoard(board)
                move = makePlayerMove(board)
                makeMove(board, player, move)
                if isWinner(board, player):
                    printBoard(board)
                    print("You won the game!")
                    isGameRunning = False
                else:
                    printBoard(board)
```

```
                    turn = 'computer'
                else:
                    move = makeComputerMove(board, computer)
                    makeMove(board, computer, move)
                    if isWinner(board, computer):
                        printBoard(board)
                        print('You loose!')
                        isGameRunning = False
                    else:
                        turn = 'human'
main() #calling main function
```

Now, let's run our game and play against our custom-made AI agent. The following illustration show the output of our game, and shows the new tic-tac-toe board layout. This is made with the function call to `printBoard`:



The following illustration depicts the gameplay where the human player is playing against the computer AI. You can see that the human is defeated by the computer player:

```
What is your next move? (choose between 1 , 9)
8
  |   |
  | X |
  |   |
 ' ' ' ' ' ' ' ' ' ' '
  |   |
  | X |
  |   |
 ' ' ' ' ' ' ' ' ' ' '
  |   |
  | O | O
  |   |
What is your next move? (choose between 1 , 9)
9
  |   |
  | X | X
  |   |
 ' ' ' ' ' ' ' ' ' ' '
  |   |
  | X |
  |   |
 ' ' ' ' ' ' ' ' ' ' '
  |   |
 O | O | O
  |   |
You loose!
The human will start the game
```

Now, we have made a layout that is appealing enough to entice any player to play the game. However, there are a few modifications that can be done, which will be covered in the next section.

Game testing and possible modifications

The game that we have made in this chapter is playable enough against a computer. Using AI in the game is all about addressing all the probable situations that a game can face while interacting with the environment. In our tic-tac-toe game, we don't have many moves compared to Chess or Go, and so making an AI agent is easier. We were able to compete with the human by making an AI that was able to make two smart moves, such as checking for the next best move to win or blocking the human from winning through simulation. If you are wondering what simulation is, you will have to recall the algorithm that we have just implemented in order to check whether the human player is going to win in the next move. In addition, the computer player was acting as a human player on the clone board and played virtually, like a human. This is called simulation, where we made the computer imitate the real-world process or behavior of the system.

After predicting the best move to make through simulation, our program was returning the best possible next move for our computer player. Let's extrapolate this technique further. What we just did in our game was make an AI that can make a simulated environment to predict the next best move. The same technique is applied to a whole range of AI applications, for instance, a self-driving car; we make a simulated environment within the computer where the car is an agent, and will make decisions either to turn left or right based on obstacles. Tic-tac-toe is simple while interacting with the environment because of its lesser number of moves or situations, but programming a self-driving a car simulation requires us to acknowledge a whole range of situations that may arise while driving car on the road. Thus, we can conclude that AI is all about creating a program where the agent must consider all the situations it may face while interacting with the environment and respond to each of those situations.

Our competitor is smart enough to make this game harder for the player to win, but humans also possess the ultimate power to make the computer player hamstrung. Human players won't let a computer have an easy win. Thus, most of the time, our game will end in a tie. However, if you run the game, you'll see we haven't addressed those cases. Now, whenever our game is a tie, instead of stopping the game, our game will incessantly ask for input from the user. Instead, we have to give the user a message saying, try again, and facilitate the user to play our game again. In order to check for a tie condition, we have to check whether the board is full. When the board positions are fully occupied and nobody is the winner, we have a tie condition. We can make another function to check whether the board is full:

```
def isBoardOccupied(board):
    for pos in range(1,10):
        if isSpaceAvailable(board,pos):
            return False
    return True
```

The preceding `isBoardOccupied()` function is going to return a Bool type, either `True` or `False`, based on the check, which will determine if the board is full or not. If the board is full, it will return `True` and if it is not, it will return `False`. We are using the `isSpaceAvailable()` function that we created in the preceding section, which will check whether there are any empty spaces on the board. Now, let's refine our code with this new function:

```
def main():
     while True:
         # add the code here from part1
         while isGameRunning:
             if turn == 'human':
                 move = makePlayerMove(board)
                 makeMove(board, player, move)
                 if isWinner(board, player):
                     printBoard(board)
                     print("You won the game!")
                     isGameRunning = False
                 else:
                     if isBoardOccupied(board):
                        print("Game is a tie")
                        break
                     else:
                        turn = 'computer'
             else:
                 move = makeComputerMove(board, computer)
                 makeMove(board, computer, move)
                 if isWinner(board, computer):
                     printBoard(board)
                     print('You loose!')
                     isGameRunning = False
                 else:
                     if isBoardOccupied(board):
                        print("Game is tie")
                        break
                     else:
                        turn = 'human'
 main() #calling main function
```

Summary

This chapter was concise and terse, containing an abundance of information, ranging from data structures to functions. These topics are the building blocks of any complex program, and so on, we will use them in every game that we will cover from the next chapter onward. We started this chapter by learning the necessity of data structures, and we delved into the fundamental data structures of Python such as lists, dictionaries, tuples, and sets. We covered the ways in which we can create those data structures and manipulate them.

We learned about ways to create user-defined functions, call them, and document them. We also saw that functions are like machines, where you can feed raw data in, and get output back as meaningful information. We saw the ways of inputting data to the functions using positional and default arguments. Then, we looked at saw the ways of modifying our functions by packing and unpacking of normal and keyword arguments in order to achieve the best performance from them.

We also modified our game further using functions and data structures, and we made simple algorithms that can address different possible situations of gameplay. We made our computer player smart enough to beat our human player. Then, we also made a simulation environment where an agent can test and train itself in order to predict the next best possible move. Although our game was simple to make, it has given us a broad range of ideas about the processes that need to be undertaken, such as brainstorming, designing, coding essentials, and analysis, before we actually start writing modular code.

Finally, we covered procedural programming, which refers to using functions to build programs. In the next chapter, we will cover procedural programming in terms of curses. We will create programs using terminal-independent, screen painter, and text-based terminals. We will make a snake game using a curses event and a screen painter and then use curses properties in order to make logic for playing the snake game.

Are you excited to hop into the next chapter? It will take you through an adventurous tour of game programming with the curses module, and we will learn how to handle user events and game consoles with it. Before that, I highly suggest that you refer to the official Python documentation and take a tour of the Python built-in data structures and modules; and practice with them without any additional help. The knowledge that we have gained so far will be used throughout the chapters of this book, so it's high time that revise the topics that we have learned about so far.

5
Learning About Curses by Building a Snake Game

Whenever any developer writes games or applications, it is likely that they need to reuse some parts of the code repeatedly. For example, when we want a player to move within the game console, they use the left and right arrow keys multiple times. Thus, we need the code that can handle such events and process them. Writing the same code multiple times to handle the same action does not support **don't repeat yourself** (**DRY**) principles, so we need to use functions that can be called multiple times to perform the same action time and again.

To facilitate this, these functions are bundled into containers known as modules. As you may recall from the preceding chapter, we have used modules in most of our programs. For example, by using the `random` module functions, we were able to get a random number between the specific range; math modules, on the other hand, allowed us to perform different mathematical computations. In this chapter, we are going to cover another module, known as curses. It will provide us with an interface where we can handle the curses library, which contains functions that directly interact with the Python terminal. This means we can make a simple Terminal-based game.

The following topics will be covered in this chapter:

- Understanding curses
- Starting the curses application
- User input with curses
- Making a Snake game with curses
- Game testing

Technical requirements

You will require the following to get the full benefit of this chapter:

- **Python IDLE** (**integrated development kit**)
- The code assets for this book, which can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter05`

Check out the following video to see the code in action:

`http://bit.ly/2oG1CVO`

Understanding curses

Curses is a Terminal controller library that allows us to write text-based applications. The word Terminal is independent to any platform, and so curses can be used on any operating system. With curses, developers will be able to write applications directly without interacting with the Terminals. The curses library is the medium that sends the commands in the form of control characters while determining the operating system or Terminal it should be executed on.

In Python, we have two libraries called windows-curses and unicurses. Both of these libraries provide the functions that can set up the desired look for the output Terminal screen. They are updated using control sequences. In short, developers will design the appearance of the output window screen and call the functions to make curses do its work. Thus, in curses-based applications, we won't get output that's a user-friendly as we expect it to be because we will be only able to write text-based applications with the curses library. Thus, any game you write with curses will run in the Terminal, that is, the command prompt of Windows or the Terminal of Linux.

Python's *curses* library will allow us to write text-based user interfaces and control the screen with user inputs. The library that we are using in this chapter will help us control the screen movements and handle user events or inputs. The programs that will be built from curses will not have features that will resemble modern GUI applications or Android applications, which have widgets such as text view, label, slider, graphs, and templates. Instead, it will provide simple widgets and tools such as the **command-line interface** (**CLI**), most of which are found in text-only applications.

The curses module of Python is an adaptation of the curses of the C-programming language. The only difference is using Python; everything can be done without us having deep knowledge of low-level routines. We can call the interfaces to invoke functions that will, in turn, call the curses to handle user operations.

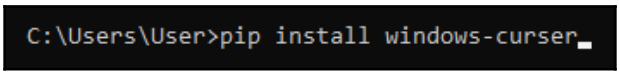
While dealing with curses, the window screen is considered a character matrix. Each window interface is set up by the programmers, and includes height, width, and border. After setting such coordinates, the programmer will invoke Python curses to update that screen. Working with widgets such as the text view, button, and label is also done in the same manner; that is, we will initialize the coordinates where it should be placed in the window and call curses to update it accordingly. To handle user input from curses, we have to import it. We can easily import actions such as RIGHT, LEFT, UP, and DOWN and handle their behavior according to the program's needs. In most games, these events are going to provide movement to the game characters. The game we are going to cover at the end of this chapter will be the Snake game, and the snake itself will be our main character. This means that actions such as LEFT, RIGHT, UP, and DOWN will move the snake to a new position. The Windows version of Python does not have an inbuilt curses module and is not available with the same name. However, there's are two compatible modules available that do the same thing. These are called unicurses and windows-curses. We are going to use the latter in this chapter.

Let's start learning about curses by making a simple application. We will make a simple `hello` program that's printed in the curses Terminal.

Starting the curses application

We are going to build the application using a module that does not come pre-packaged with Python. Therefore, we have to install that package manually on our machine. With the installation of Python, a package management system should have been installed automatically on your machine known as pip. This management tool is used to install and organize the libraries that are written with Python. Thus, if you want to use any third-party libraries or dependencies on your program, you have to install them using the pip tool. The method to install any package is simple. You simply have to write the `pip install` command, followed by the name of the library you wish to install. The name of the library is case-sensitive, and so no mistakes should be made while writing the name of the library. If you want to check the code that is written inside the library, simply search for the documentation of that library. You will get information about the library, as well as the functions that are available to be used inside your programs.

We are going to use the windows-curses library to write text-based programs, and so we have to install that package using the `pip` command. The pip command should be executed in the command prompt if your machine is Windows, and in the Terminal of your machine if you're using Mac OS X or Linux. The following screenshot shows how we need to execute the `pip` command:



Similarly, to install curses in your Linux machine, you can open your Terminal and run the following command:

```
$ sudo apt-get install libncurses5-dev libncursesw5-dev
```

Now, we will be able to write programs using the curses module. At this point, the curses module that we have installed will be available in the same way that other built-in modules such as math or random and available. Similar to built-in modules, we can simply import the curses module and start invoking the functions that are defined within it. The following steps explain the roadmap for creating any curses applications:

1. Let's start by importing curses and seeing if it was installed properly or not. The command we use to import any module is `import` followed by the module's name. Our module name is curses. Thus, the command would be as follows:

```
>>> import curses
>>> #no any error
```

2. We can conclude that it is imported successfully as the Python parser hasn't thrown an error. Now, we can use this module to write programs. Let's write a simple program to observe the working procedure of the curses module:

```
#program is written as Scripts
# curser_starter.py

import curses
import time
window_screen = curses.initscr()
window_screen.clear()

time.sleep(10)
```

We cannot run any curses applications directly from Python IDLE. To run it, you have to navigate to the folder where you have stored the Python file and double-click on that file to open it. You will get a blank screen with a cursor at the top, which will remain there for 10 seconds. After 10 seconds, the same blank window screen will pop out from the screen. That screen will yield the text-based application that can be written with curses.

Let's look at our preceding code and uncover the interesting functions of curses:

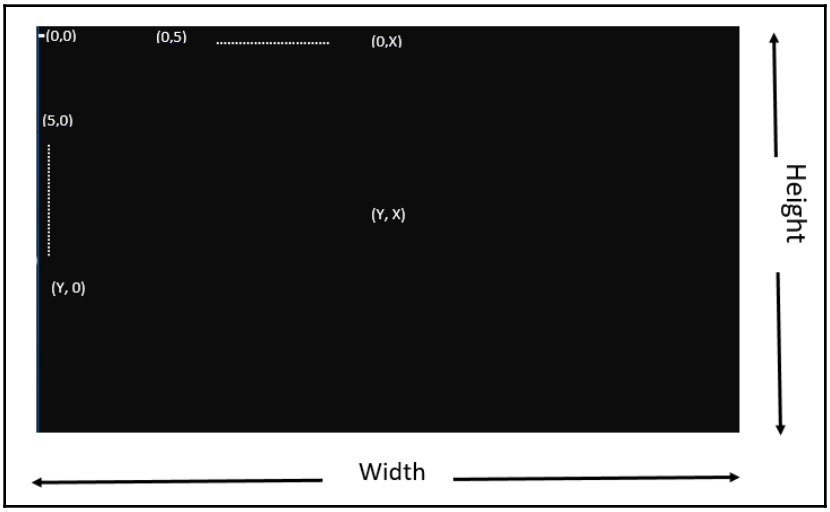
- First of all, as always, we imported the module that we wanted to use in our program. We imported two modules here: curses and time. The curses module has different functions available for us which can be used to write text-based applications, while the time module has different functions available that can be used to update our output screen behavior. In this program, we called the `sleep` method of the `time` module, which will hold the output of the screen for the amount of time that was passed within its parenthesis (10 seconds, in our case). After 10 seconds, our output screen will disappear.
- Before writing any code with curses, it should be initialized. The invocation of the `initscr()` function is going to initialize curses. Thus, for any curses application, we should initialize curses at the first line of the code. This initialization code is going to return us a window object that represents the output screen of our program. Here, this initialization is captured by the window object named `window_screen`, which represents the screen of our output Terminal. Thus, any function call to the curses API should be done with `window_screen`. The first invocation is done with the `clear()` function.

We successfully created a game screen and held it with a method call. However, the current screen is not modifiable enough. We, as programmers, may want to customize the screen by specifying the height and width explicitly. Fortunately, Python's curses module provides another method to accomplish this, that is, the `newwin` method. We will learn about it in the next section.

New screen and window objects

The window object that's returned from the invocation of the `initscr()` function represents the entire screen of the output window. This window object also supports different methods that can display text to the window, alter it, take events from the user and update positions, and so on. The only demerit with this `initscr()` function is that we cannot a pass custom height or width of the screen to it. It only represents the default whole screen of the output Terminal.

Sometimes, we might want the screen of the game to be customized so that it has a height of 20 and a width of 60, for example. Here, height and width can be considered as column and row, where each unit represents a line within that matrix. Since we have passed a width of 60, there will be 60 horizontal lines. The same goes for our height of 20; there will be 20 vertical lines. You can represent them as pixels too. To create a new screen, which is probably what we are going to do while making a curses application since the `initscr()` function isn't going to do this for us, we have to call the new function. This function is going to divide the bigger window screen into a new one based on the coordinates specified. The name of this function is `newwin()`, which literally means new window, and it takes four arguments, that is, **Height**, **Width**, **Y**, and **X.** The order these are passed in is **Y**, **X** which is unusual compared to other libraries. The **Y** value is for the column's position while the **X** value is for the row's position. Take a look at the following diagram, which explains the values of **Y** and **X**:



Thus, by increasing the value of **Y**, we are going downward, which is the same as the column in a matrix. Similarly, by increasing the value of **X**, we are toward the right-hand side of the screen, which is the same as the row in a matrix. As we can see, curses stores a window screen in the form of the character matrix. We can use those coordinates to represent the position of the game's display, as well as the game characters. For example, if you want to make your player move in the position of (5,0), as shown in the preceding diagram, you would call the `move(5,0)` function to achieve that. Remember the order in which the argument is passed. The value of **Y** is followed by **X**, which may confuse you if you have a background in game programming from any other library.

As an example, we are going to create a program where we will make a new screen inside a big screen using the `newwin()` function. The four arguments inside this function are `height`, `width`, `y`, and `x`. Remember this order as we have to pass it in a similar fashion:

```
height = 20
width = 60
y = 0
x= 0

screen = curses.newwin(height, width, y, x)
```

Now, it's time to write a simple program that can add some text to our curses application:

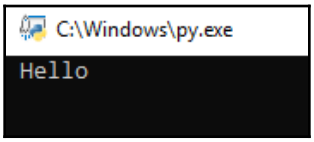
```
# text_app.py
import curses
import time

screen = curses.initscr()
curses.noecho()
curses.cbreak()
screen.keypad(True)
screen.addstr(0,0, "Hello")
screen.refresh()
time.sleep(10)
curses.endwin()
```

Let's observe the preceding code line by line and learn about each of the methods we used, as follows:

- First of all, we imported the two important modules: curses and time. After that, we initialized the window object with the `initscr()` function.
  The `noecho()` function will turn off the automatic echoing process in our application. This is essential because while the user is playing the game, we don't want them to show us what they pressed; instead, we want them to perform an action based on that event. The next function call is `cbreak()`. This mode will help our program react instantly to the user's input. For example, in the case of the `input()` method of Python, until and unless we are going to press *Enter* on our keyboard, this method won't perform any action. However, in the case of the `cbreak()` function, it is going to help the application react to any input keys instantly without the need for pressing *Enter*. This is important because we have to make a game where the user will get a response without any delay. For example, if the user presses the DOWN key, the character of the game must move in downward instantly. This is different to a buffered input function, which is going to take all the input and store it in a buffer that's going to react only if the user presses *Enter*.

- The next function call is the `keypad()` function. We have enabled keypad mode by passing True as an argument. Whenever we press any key in the Terminal, it returns data in the form of a multibyte escape sequence. For example, `Esc` sends `\x1b`. That is 1 byte. `Page Up` sends `\x1b[H`. That is 3 bytes. To handle such data that is returned by the Terminal, curses uses a special value that can be imported manually. For example, to handle the DOWN key being pressed on the keyboard, we can import it as `curses.KEY_DOWN`. This is done by enabling keypad mode.
- After that, we called the `addstr()` function. This function will add a string to the output screen in the position specified during its call. We passed three arguments to it. Remember that the first two arguments are in the order y, x. The last argument that's passed is a string that needs to be added to the position of (y,x). We have passed a value of (0,0), which means the string will be added to the uppermost left corner of the output window. The next method that we called was `refresh()`, which is going to update the character matrix of the window object *screen*. If you take a look at the code carefully, you will see that whenever we are adding or refreshing the content of the screen, we are doing it using a window curses object, which was initialized using the `initscr()` function. However, the behavior of the Terminal has been altered by the curses module. For example, to change the default echoing behavior of the Terminal, we made a direct call to the `noecho()` function from the curses module instead of from the window cursor object.

Now, let's run our code to observe the result. Make sure you run your application from your Terminal or command prompt with `filename.py`:



You can change the position from (0,0) to any other value, for example, (5,5), to observe the windows and padding format.

Finally, we have made our first program with curses. Now, it's time to explore another feature of curses, which is based on the ability to handle user input.

User input with curses

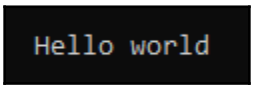
In any game, user input is one of the most crucial pieces of information that needs to be handled properly. We cannot make any delay while handling these types of actions. In the case of curses, we have two ways to take input from the user. These two methods are as follows:

- `getch()`: If you have any programming background of languages such as C or C++, this should not be new to you. The `getch()` function, just like in C, is used to make a listener that will listen to the user key continuously. It returns an integer from 0 to 255 which represents the ASCII code of the key that was pressed. For example, the ASCII code for `a` is `097`. Values that are greater than 255 are special keys, for example, the *Page Up* and navigation keys, that is, UP, DOWN, LEFT, and RIGHT. We can compare the values of such keys with constants stored in curses; for example, `curses.UP`, `curses.DOWN`, `curses.LEFT`, and `curses.RIGHT`.
- `getkey()`: `getch` and `getkey` do the same thing, but the `getkey` function converts the returned integer into a string. Normal keys such as a-z or A-Z will be returned as a 1 character string that can be compared with the `ord()` function. However, special keys or functional keys will be returned as a longer string containing a key and representing the type of action, such as `KEY_UP`.

Let's write a program that can handle keyboard events:

```
#program3.py
import curses as c

screen = c.initscr()
win = c.newwin(20, 60, 0, 0)

c.noecho()
c.cbreak()
screen.keypad(True)
while True:
  char = screen.getch() #takes input
  if char == ord('q'):
      break
  if char == ord('p'):
      win.addstr(5,10, "Hello World")
      win.refresh()
screen.endwin()
```

We discussed this code when we talked about using a True loop. Make sure you revise the preceding topics if you are confused about any of these commands. One strange thing you might observe in this code is that we have imported curses and gave it an alias of c. This is the process of renaming your module. Now, instead of using `curses.method_name` at every method call, we can simply call it using `c.method_name()`, which certainly removes the overhead of writing the same module name every time. Inside the loop, we used the `getch()` function to take input from the user. After that, the character is retrieved in the `char` variable and we compare it with the returned value of the `ord` function. Remember that the `getch` function was going to return a value in Unicode? The same is done by the `ord` function. It takes an argument as a character and returns the Unicode value of that character. We use conditionals to make a condition. So, if the user presses *q* on their keyboard, we will end the program, and if the user presses *p* on their keyboard, we will print `Hello World` to the output window at the position of (y,x). Let's run our Python file, `C:\User\Desktop> python program3.py`, and take a look at the output:



Press *q* on your keyboard to terminate the loop and close the application.

Note that *q* is not the same as *Q* because the ASCII code for these characters isn't the same.

Our code is running perfectly but is becoming lengthier, even though the application is so simple. We are calling so many methods already, such as `noecho()`, `cbreak()`, `keypad()`, and `endwin()`. To remove the overhead of calling so many functions, we can use the wrapper function from the curses module. All of these functions, including the initialization of curses objects, is done automatically by the wrapper function. Just remember that the wrapper function as a call to the bundle that wraps all of these methods inside of it.

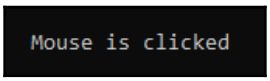
Similarly, we can also handle mouse events using the curses module. Let's make a program using the wrapper function and handle the events of the mouse buttons in the same program:

```
#mouse_events.py

import curses as c
def main(screen):
  c.curs_set(0) #hides the cursor
```

```
    c.mousemask(1)
    inp = screen.getch()
    if inp == c.KEY_MOUSE:
        screen.addstr(17,40, "Mouse is clicked")
        screen.refresh()
    screen.getch()

c.wrapper(main)
```

Let's take a look at the preceding code in detail:

- We will start from the last line, where we called the wrapper function with some callable object as an argument. We've already learned about the objectives of `wrapper()`; it eliminates multiple function calls such as `initscr()`, `noecho()`, and so on. Thus, debugging is easier using the wrapper function. Not only that, but this function also handles exceptions internally by using try and catch blocks. Whenever you run into an unknown exception that you might not have caught, you can always trust the wrapper function to do so. This will identify the bugs of your program and provide you with an exception message without crashing the application. The argument to the wrapper function will be a callable object, which, in this case, is the main function. This main function has a *screen* argument, which is the curses window object. We didn't initialize the curses object anywhere in the program with the `initscr()` function because this was done internally by the wrapper function.
- Inside the scope of the main function, we made a call to two methods: `curs_set(0)`, which is going to hide the cursor in the output screen, and `mousemask(1)`, which is going to accept the mouse events. Here, mouse events will be special symbols or functional characters that will be different from normal alphabetical characters. Thus, curses has made constants to address those functional characters. This is the same for the UP keyboard key; we have the `KEY_UP` constant; in the case of mouse events, we have the `KEY_MOUSE` constant. These should be invoked from the curses module, for example, `curses.KEY_MOUSE`. After we get such mouse events, we are going to print `Mouse is clicked` to the output Terminal. The `getch()` method is going to input any events that can be either mouse-related or keyboard buttons. Let's run our program to achieve the following output:



Now that we have gained enough knowledge to make games using curses, let's proceed to the next section, which will give us an idea about how gaming logic is made under the hood. We will be making a simple Snake game.

Making a snake game with curses

We already know that the process of writing games is not as easy as it seems. We have to follow many procedures to make a game playable because, while exposing the game to the environment, we can be overwhelmed by many unwanted and unexpected exceptions. Thus, following the proper order of execution is always essential, even if it may take more time than usual. In this section, we are going to create a Sake game using curses. We will be modifying it into a more appealing game in the upcoming chapters. A good game does not always mean a good user interface because the interface provides value to the user but not the programmer. We have to make a habit of writing proper code along with making good interfaces, which requires us to follow each of the steps we will go over in this section. We are going to use the curses module to make the initial Snake game. Then, in the next chapter, we will modify it using object-oriented programming.

Before we code, we have to gather information about the model and the interface of the game. While modeling, we have to extract critical information, such as *how to render the game characters into the screen*, *how to make an event listener*, and *how to make logic that will allow the game characters to move*. We will cover all of this in the next section.

Brainstorming and information gathering

As we have been doing up until now, the first step is to brainstorm and gather critical information about the game layouts and game models. In the Snake game, we have two characters: the snake (the player) and its food. Whenever the snake eats food, its length should be increased. That's the basic idea, anyway. Now, let's revise the resources that are available to us. Obviously, the resources that are provided by Python are more abundant, but we haven't learned how to make graphical characters and use them in our game yet. All we've learned to do is make games with text-based Terminals. We can use characters such as A-Z to specify game objects. For example, we can make the snake `XXXXXXX`, which is a combination of Xs. The food can be represented by `O`. Let's see what this would look like in our game console:

We also have to decide on the screen for the game. The `initscr()` method is going to create the entire screen as a curses object. We don't want that; instead, we want to make a game screen that can be customized by the height, width, and y, x positions. As you may recall, we can divide the screen into a new one, using the `newwin()` method.

The most important thing to remember is to track the coordinates because we have to make a boundary for our gameplay. We can make certain rules that specify the boundary position of the game character and, if they touch that boundary, we can terminate our game.

We have to make logic for two things:

- Whenever the snake eats the food, we have to generate new food in a new position.
- Whenever the snake eats the food, we have to increase the speed of the snake to make the game more difficult. We should also track collisions between the snake's head and its body.

In terms of the former point, we can use the random module, which provides a random coordinate position of (y, x) that we can assign food to. For the latter point, we have to use a curses method called timeout. We have to pass the value of the delay as an argument to that function. According to Python's official documentation, the timeout function sets blocking or non-blocking read behavior for the window. If *delay* is negative, a blocking read is used (which will wait indefinitely for the input). If *delay* is zero, then a non-blocking read is used, and -1 will be returned by `getch()` if no input is waiting. If *delay* is positive, then `getch()` will block for *delay* milliseconds and return -1 if there is still no input at the end of that time. Thus, we can change the speed of the game based on the delay when it is zero or positive.

So, in terms of the `curses.timeout(delay)` command, if you make delay negative, your snake will move at a rapid pace. However, we need to remember that we have some constraints here; the speed of the snake should be increased, along with the length of the snake. First of all, what is a snake? How it is made in our game? We learned about lists in the previous chapter. Let's use that to make a snake. We have already seen the structure of our snake, which is a bunch of X characters. But at the beginning of the game, we should only provide a small length for the snake, maybe a length of 3, that is, `XXX`. We will store each of these X's in the list, which represents coordinates such as [[4,10], [4,9], [4,8]]. Here, each of these lists represents one X, that is, at the position of [4,10], we will have one X and another X at 4,9. Remember that these should be y, x positions and that they should be next to each other because they represent the body of the snake.

Let's say our delay is 100 which would be constant. Thus, our command to represent speed will be `curses.timeout(100)`, which will be the snake's constant speed throughout the game. However, we can change the speed of the game by incrementing the length of the snake. For now, let's proceed to the next section, where we will make a boundary for our game.

Inception

In this section, we will start writing the code for our game. We will use the curses module to do this. First, we will initialize the screen for the game and make some game characters. Take a look at the following code:

```
#snake_game.py
import curses as c

c.initscr()
win = c.newwin(20,60,0,0)
win.keypad(1)
c.noecho()
c.curs_set(0)
win.border(0)
win.nodelay(1)

snake = [[4,10], [4,9], [4,8]]
food = [10,20]

win.addch(food[0],food[1], 'O')
```

There's nothing new in the preceding code. You can also eliminate all of the function calls using the `wrapper()` function. We can see that we have two list variables, `snake` and `food`, which contain the coordinates that represent their positions in our game console. We also made a call to the `addch` function. It will work in a similar fashion to the `addstr` function. We passed the position of the food and added the `O` character to that position.

Making computer games require two steps: the first step is to make a visual that must be naturally attractive, while the second step is to make the player interact with the game. To make games interactive, we have to handle the events provided by the player. This is what we will do in the next section.

Handling user key events

We've started to build the basic layout of our game. Now, let's write some code that can handle user keyboard events. Snake is a simple game. We can make it work by handling only four keys of the keyboard, that is, UP, DOWN, LEFT, and RIGHT. We can use `getch()` to get user input. But remember, these are not alphabetical characters, they are functional characters. Thus, we have to import constants such as `KEY_UP`, `KEY_DOWN`, `KEY_LEFT`, and `KEY_RIGHT` to fetch those ASCII values. Let's start writing the code that will handle user events:

```
from curses import KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT
#CODE FROM PREVIOUS TOPIC

key = KEY_RIGHT #default key

#ASCII value of ESC is 27
while key != 27:
  win.border(0)
  win.timeout(100) #speed for snake
  default_key = key
  event = win.getch()
  key = key if event == -1 else event
  if key not in [KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN, 27]:
     key = default_key
```

The code that we have written may seem complicated, but all of these things have been covered already. Let's take a look at what we've done:

- In the first statement, we made the default key `KEY_RIGHT`. This is important because we don't want to make the snake move if the user hasn't pressed a key. So, our snake character will move right automatically when the game is started.
- After that, we made a game loop. This loop is going to be executed until we press *Esc* since the ASCII value for *Esc* is 27. Inside the loop, we made a call to the timeout method, which will represent the speed of our snake character. In the next line, we get the event of the user using the `getch()` method. Remember that, if you press any key event, its value is going to be -1. Thus, we can compare it and put the key that was pressed by the user into the key variable. However, the key can be anything, such as an alphabetical character or a special symbol such as [!,@,#,$], and so we have to filter them with the appropriate keys, for example, LEFT, RIGHT, UP, and DOWN. If the key that's pressed by the user is not among these, we are going to make the key have a default value of `KEY_RIGHT`.

Now, we can communicate our program with input devices such as a keyboard or joystick. It's time to move on to the next section, where we will create our first logic to update the **head position** of the snake characters when the user presses the LEFT, RIGHT, UP, and DOWN keys.

Game logic – updating the head position of the snake

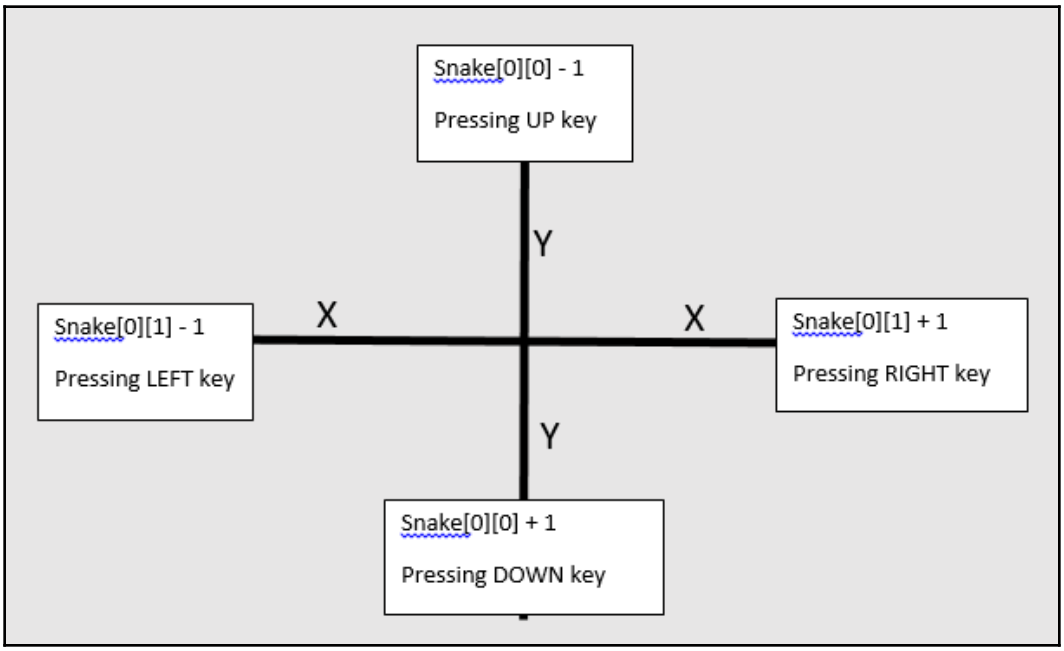
In the previous section, we were able to handle user events using constants provided by curses. Just like movement, the head of the snake can also be changed. We have to make brand new logic that will update the position of the snake's head. Our snake is a composition of coordinates that are stored in the list. The first element of that nested list is the position of the snake's head. Thus, we only need to update the first element of the list. Let's see how we are going to do this:

```
while key != 27:
  #code from preceding topic
  snake.insert(0, [snake[0][0] + (key == KEY_DOWN and 1) +
  (key == KEY_UP and -1), snake[0][1] + (key == KEY_LEFT and -1) +
  (key == KEY_RIGHT and 1)])
```

This may seem like it's a little difficult to understand, so let me make this clear.

The `snake` variable is a list. Thus, we can use the `insert()` method to manipulate that list element. The `insert()` method will take two arguments: one will be the index and the other will be the element to be inserted. In the preceding code, index is 0, which means we want to insert an element in the first element of the list, which represents the head of the snake. The next argument is the element, which needs to be added to index 0. We can see a comma (`,`) in-between two statements: `snake[0][0] + (key == KEY_DOWN and 1) + (key == KEY_UP and -1)` and `snake[0][1] + (key == KEY_LEFT and -1) + (key == KEY_RIGHT and 1)`. The first statement represents the y coordinate of the snake's head, while the second statements represents the x coordinate of the snake's head. In the y part of the snake's head, which can be represented as a column, we can have two movements: either DOWN or UP. While going down, we have to add 1 to the current head position y element and while going up, we have to decrease 1 in the current y position. For the x part of the snake's head, we have the LEFT and RIGHT movements.

On pressing the LEFT key, we are going to decrease the coordinate of x with 1 and on pressing the RIGHT key, we are going to add 1 to x. Still confused? Taking a look at the following diagram should make things clearer for you:



Remember that this update has to be done in the order of (y,x). For every **UP** and **DOWN** key that's pressed, a decrement or increment of 1 is done in the coordinate of y, which is snake[0][0] for the head. For x, this is snake[0][1], which is the same increment and decrement that we used previously, but for when the user presses the **RIGHT** and **LEFT** keys.

Now that we have made some logic to update the position of the snake, we need to make the snake eat the food. The logic we are going to cover is simple: when the snake's head position is the same as the food's position, we can say that the snake ate the food. Let's go over this now.

Game logic – when the snakes eats the food

Let's make the next bit of logic for our game. In this section, we are going to make the snake eat the food. This is quite simple to implement. Whenever the snake's head touches the food, we will assume that the snake has eaten the food. Thus, the snake's head coordinates and the food's coordinates will be the same. We also have to make some logic that will generate food in the next location as soon as the snake eats the current piece of food. The location for the next piece of food should be random. We can use the `random` module to create such an arbitrary location. Let's start writing the code:

```
from random import randint
```

This is a new way of importing any function from the module. While calling this function, we don't have to write something like `random.randint()`. Instead, we can call it directly in our program. The arguments inside the `randint()` method must be the range of values. For example, `randint(2,8)` returns a number between 2 to 8, like so:

```
while key != 27:
#add the following code after updating head position
  if snake[0] == food:
      food = []
      while food == []:
        food = [randint(1,18), randint(1,58)]
        if food in snake:
            food = []
      win.addch(food[0], food[1], 'O')
  else:
      last = snake.pop()
      win.addch(last[0], last[1], ' ')
  win.addch(snake[0][0], snake[0][1], 'X')

c.endwin()
```

Inside the if part of the code, we have added the logic that will put the food in a new position. Remember that, at the beginning of the game, we initialized the new window height to be 20 and the width to be 60. Thus, we can only generate food within this boundary. In the else part of the code, we pop out the last element if the user is unable to eat the food. In the second to last line, we added the snake's head position with the `'X'` character.

Let's run our game and see what it looks like so far:

Now, our game is playable enough. We learned about so many things while making this game, such as how to make game logic while working with the methods and coordinates of the game console. Now, let's proceed to the next section, where we will learn how to test and modify our game.

Game testing and modification

To uncover the deficit of any program, its always a good idea to run and test it. Just like our previous games, we can also make modifications to the Snake game. The following points explain some of the modifications we could make to our game:

- When you run the game, the first thing you will notice is that our game does not have logic to decide whether the snake has collided with itself or not. If it does collide with another part of its body, we have to stop the game. Let's add that logic inside the while loop:

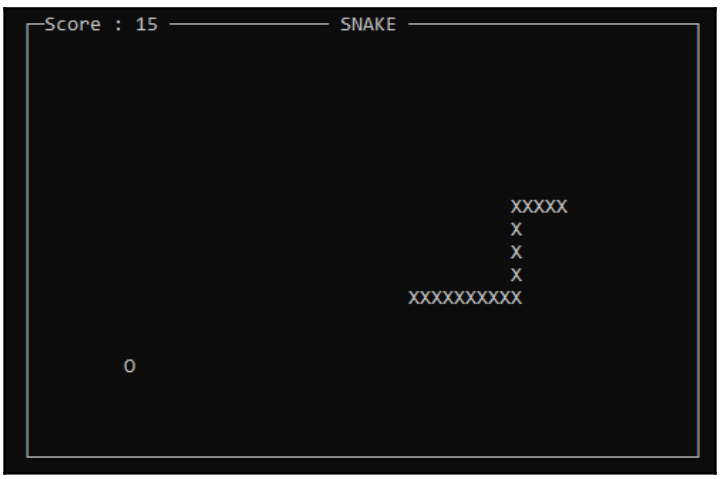
```
if snake[0] in snake[1:]:
    break
```

- In the preceding code, snake[0] represents the head of the snake while snake[1:] represents the body of the snake. Thus, the preceding condition means that the head coordinates are inside the body of the snake, which means a collision occurred. In this case, we use a `break` statement to get out of the loop and terminate the game.

- Let's say we want to add the score of the player. Adding a score is simple; the number of food that has been eaten by the snake is equivalent to the player's score. We can initialize the value of the score as 0 to start with:

```
score = 0
while key != 27:
  # CODE TO ADD SCORE IN THE SCREEN
  win.border(0)
  win.addstr(0, 2, 'Score : ' + str(score) + ' ')
  win.addstr(0, 27, ' SNAKE ')
  if snake[0] == food:
      food = []
      #AFTER EATING EVERY FOOD SCORE = FOOD
      score += 1
      while food == []:
        food = [randint(1,18), randint(1,58)]
        if food in snake: food = []
      win.addch(food[0], food[1], 'O')
  else:
      end = snake.pop()
      win.addch(last[0], last[1], '')
  win.addch(snake[0][0], snake[0][1], 'X')
c.endwin()
```

In the preceding code, we have added some statements with the `addstr` method that will provide the score of the player at the specified position. Now, let's run our game:



After running the game, you can see that we are able to play within the interface of curses. However, you will encounter an exception as soon as your snake hits the boundary line, and your game will be terminated automatically. We will learn how to handle boundary collisions in the upcoming chapters in detail (to be specific, `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*), but, for now, let's learn about the easiest method we can use to handle and get rid of triggering an exception. First of all, observe the dimension of the boundary screen and take note of the actual height and width at which the boundary resides. Consider looking at the `win` variable to get an idea about the size of the boundary screen. Now, looking at the height of 20, we might assume that whenever the snake touches the top boundary, that is, he head position of the snake at 0, the snake's head must enter through the own boundary, which has a y coordinate of 19. Remember that, in the upper and lower boundaries, only the y-coordinate changes. The code for this will be as follows:

```
if snake[0][0] == 0:
    snake[0][0] = 18 #regenerate snake from lower boundary line
if snake[0][0] == 19:
    snake[0][0] = 1 #regenerate snake from upper boundary line
```

Similarly, we have to address the case where the snake hits either the right or left boundary. Since the height remains the same for either case, we are only interested in the width (x-position). Since the width of the screen that's declared by the win variable is 60, we can expect the snake hitting a boundary at around 0 (for the right) and 59 (for the left) to cause the snake to be regenerated accordingly. You have to add the following line of code to handle collisions that occur at the left and right boundaries:

```
if snake[0][1] == 0:
    snake[0][1] = 58 #regenerate from left
if snake[0][1] == 59:
    snake[0][1] = 1 #regenerate from right
```

Finally, we have completed the snake game. It is appealing enough to make any user play this game. We also learned how to create programs with our own brand new logic. This was the first simple module we have used to make text-based games. Even though it's playable, we haven't added any graphics to it, and so it looks quite bland. We will make it more scintillating by learning about a new Python paradigm named Object Oriented Programming. We have successfully made some modifications to our game. Now, it's time to learn about the most important concept of Python: object-oriented programming.

Summary

In this chapter, we started to uncover the world of game programming with curses. Obviously, it wasn't the perfect game as it had no amazing animations or a fantastic interface. We barely touched on these topics since curses provides applications that are text-based and run on a plain Terminal. Even the game characters, such as snake and food for the Snake game were made out of alphabetical letters. Even though we didn't put extra effort in to make the game more appealing, we have learned about how to make the game logic. Two of the pieces of logic that we made in the Snake game were important: the first was the interaction of the coordinates of the game console with the player's positions, and the second was making the characters collide. The coordinate system's order that's supported by curses was strange. In most libraries, such as pygame and pyopengl, we have a coordinate system represented in the order (x,y), but in curses, it's (y,x). The collision between two characters is confirmed if they are in the same coordinate point (y,x). To do this, we have to check the collision between the snake's head and its body. This logic might sound simple, but it will come in handy in the long run. For example, in upcoming games such as Flappy Bird or Angry Birds, we are going to check the collision between the characters with the same logic.

The code that we've written for the Snake game is meticulous and thorough because the game was written with procedural programming in mind. In the next chapter, we will learn about the most important concept of Python, object-oriented programming, and modify our code accordingly, which will make our code more readable and reusable.

6
Object-Oriented Programming

Programming is not only about writing programs—it's just as important to understand them so that we can fix the bugs and errors in them, if there are any. Thus, we say that programmers are born to read and understand code. However, as programs become more and more complicated, it becomes more difficult to write programs that maintain readability. We have written both aesthetic and messy code in this book. We made a tic-tac-toe game with sequential programming that had less readability. We can consider these programs as inelegant because we will have a hard time reading and understanding their code and sequential flow. After writing those programs, we modified them using functions, which upgraded our messy code so that it was more elegant. However, if you are working on programs containing thousands of lines of code, it's hard to write a program within the same file and understand each and every behavior of the functions you're using. Thus, discovering and fixing the bugs of programs written in a procedural manner is also difficult. Due to this, we need a way in which we can easily break multi-line programs into smaller modules or parts so that discovering and fixing these bugs is easier. There are many ways of achieving this, but the most efficient and popular way is using the **object-oriented programming** (**OOP**) approach.

As it turns out, we have been using objects since the beginning of this book, but haven't understood how precisely they are made and used. This chapter will help you to learn about the terminology and concepts of object-oriented programming through some simple examples. We will also modify our Snake game code that we wrote using functions in the preceding chapters in line with the OOP approach at the end of this chapter.

The following topics will be covered in this chapter:

- Overview of OOP
- Python classes
- Encapsulation
- Inheritance
- Polymorphism
- Snake game implementation using OOP
- Possible errors and modifications

Technical requirements

You will require the following in order to get the most out of this chapter:

- Python version 3.5 or newer
- Python IDLE (Python's inbuilt IDE)
- A text editor
- A web browser

The files for this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter06`

Check out the following video to see the code in action:

`http://bit.ly/2oKD6D2`

Overview of OOP

Everything in Python is an object. We have been eloquently stating this remark from the beginning of this book and we have been proving this statement in every chapter. Everything is an object. Objects can be a collection of elements, properties, or functions. Data structures, variables, numbers, and functions are objects. OOP is a programming paradigm that provides an elegant way of structuring programs with the help of objects. The behavior and properties of the objects are bundled together into templates, which we call a class. That behavior and their properties can be called from the different objects of that class. Don't get confused by the terms behavior and properties. They are just different names for methods and variables, respectively. Functions that are defined inside some classes are referred to as methods.

We will dig into the concepts of classes and methods later in this chapter, but for now, let's learn more about objects before actually making a template for them.

We have been using objects unknowingly from the beginning of this book. We have used methods from different classes before, such as the `randint()` method. This method was used by importing a module named `random`. This method is also a built-in Python class. A class is a template where we can write functions of objects. For example, a man can be represented as an object. A man has different characteristics, such as a `name`, `age`, and `hair_color`, which are unique properties. However, the actions that the man performs, such as eating, walking, and sleeping, are behaviors or methods. We can make as many objects as we like from these templates. But for now, let's visualize two objects:

```
Object 1: Stephen : name = "Stephen Hawking", age= 56, hair_color= brown,
eating, walking, sleeping

Object 2: Albert: name = "Albert Einstein", age = 77, hair_color= black,
eating, walking, sleeping
```

In the preceding two objects, `name`, `age`, and `hair_color` are unique. All of the objects will have unique properties, but the behavior or method that they perform is the same as eating, walking, and sleeping. Thus, we can conclude that the data model that interacts with the input and output is a property since it will be fed into the methods. Based on the unique properties of each object, the methods of the classes will produce different results.

Thus, we can say that OOP is the approach of modeling real-world entities as objects that have unique data associated with them and can execute certain functions. Functions that are defined inside classes are called methods, and so we just have to switch from functions to methods. Note, however, that the way methods work is similar to that of functions. Just like functions are called with the names or signs of it, methods also need to be called with their names. However, this call should be initiated with objects. Let's look at a simple example to clarify this:

```
>>> [1,2,3].pop()
3
```

We looked at these methods in the preceding chapters. But if you take a good look at this code, you will find that we are making a method call from the object. We used the `pop` method and called it on the list-objects. This is a simple prototype of object-oriented programming. One of the advantages of OOP is that it hides the inner complexity of the method call. As you may recall, we called the `randint` method with the random module. We didn't even look at the content of the random library. Thus, we obviate the working complexities of the library. This feature of OOP will allow us to focus only on the important parts of the program rather than the internal working of the methods.

The two main entities of object-oriented programming are objects and classes. We can remember a class by emulating it with templates where methods and attributes are mapped. Methods are a synonym for functions, while attributes are the properties that distinguish each object from another. Let's get a good grasp of this terminology by making a simple class and object.

Python classes

As we discussed in the previous section, objects inherit all the code that's written inside the classes. Thus, we can use methods and attributes that are mapped inside the body of class. A class is a template from which instances can be created. Take a look at the following:

In the preceding, the `Bike` class can be considered a template from which objects can be instantiated. In the `Bike` class, there are attributes that uniquely represent objects that are created from this class. Each object that is created will have different properties, such as name, color, and price, but they will invoke the same methods. This method should be invoked with the instance of the class. Let's see how classes can be created in Python:

```
>>> class Bike:
        pass
```

We create a class in Python with a class keyword, followed by the name of the class. Usually, the first letter of the class name is written in uppercase; here we have written `Bike`, with a capital B. Now globally, we have created the `Bike` class. Instead of methods and attributes, we have written a pass inside the body of the class. Now, let's make the object of this class:

```
>>> suzuki = Bike()
>>> type(suzuki)
<class '__main__.Bike'>
```

In the preceding code, we made an instance named `Suzuki` from the `Bike` class. The instantiation expression looks similar to the function call. Now, if you check the type of the `Suzuki` object, it is a type of the `Bike` class. Thus, the type of any object will be the class type because objects are an instance of classes.

It's now time to add a couple of methods to this `Bike` class. This is similar to the declaration of the function. The `def` keyword, followed by the names of the methods, is the best way to declare the methods of a class. Let's take a look at the following code:

```
#class_ex_1.py
class Bike:
    def ride_Left(self):
        print("Bike is turning to left")

    def ride_Right(self):
        print("Bike is turning to right")

    def Brake(self):
        print("Breaking......")

suzuki = Bike()
suzuki.ride_Left()
suzuki.Brake()

>>>
Bike is turning to left
Breaking......
```

We have added three methods to the `Bike` class. The parameter that we used while declaring these methods was the `self` variable. This `self` variable or keyword is also an instance of the class. You can compare this `self` variable with the pointer, which is pointing to the current object. At each instantiation, the `self` variable represents the pointer object that is pointing to the current class. We will clarify the usage and importance of the `self` keyword shortly, but before that, take a look at the preceding code, where we created a `Suzuki` object and called the methods of the class with it.

The preceding code is similar to the code where we called the `randint` method from the random module. This is because we are using methods of the random library.

When any classes are defined, only the representation for the object is defined with it, which eventually reduces memory loss. In the preceding example, we made a prototype with the name `Bike`. Different instances can be made out of it, las follows:

```
>>> honda = Bike() #first instance
>>> honda.ride_Right()
 Bike is turning to right

 >>> bmw = Bike() #second instance
 >>> bmw.Brake()
 Breaking......
```

Now that we've looked at how to create objects and use the methods that are defined inside the class, we will add attributes to the class. The attributes or properties define the unique features of each object. Let's add some attributes, such as `name`, `color`, and `price`, to our class:

```
class Bike:
     name = ''
     color= ' '
     price = 0
     def info(self, name, color, price):
         self.name = name
         self.color = color
         self.price = price
         print("{}: {} and {}".format(self.name,self.color,self.price))

 >>> suzuki = Bike()
 >>> suzuki.info("Suzuki", "Black", 100000)
 Suzuki: Black and 100000
```

There's a lot of jargon in the preceding code. Under the hood, this program is about the creation of classes and objects. We have added three attributes: `name`, `color`, and `price`. To use those properties of the class, we have to reference them with the `self` keyword. The `name`, `color`, and `price` arguments are passed into the `info` function and are assigned to the corresponding `name`, `color`, and `price` properties of the `Bike` class. The `self.name, self.color, self.price = name,color,price` statement is going to initialize the class variables. This process is called initialization. We can also do initialization using a constructor, like so:

```
class Bike:
    def __init__(self,name,color,price):
        self.name = name
```

```
            self.color = color
            self.price = price

        def info(self):
            print("{}: {} and {}".format(self.name,self.color,self.price))

>>> honda = Bike("Honda", "Blue", 30000)
>>> honda.info()
Honda: Blue and 30000
```

In Python, the special `init` method is going to simulate the constructor. A constructor is a method or function that's used to initialize the attributes of the class. The definition of the constructor is executed when we make instances of the class. Depending on the `init` definition, we can provide any number of arguments while creating the objects of the class. The first method of the class should be a constructor, and it must initialize the members of the class. The basic format of the class should have an attribute declaration at the beginning and the methods after it.

Now that we've created our own class and declared some methods of it let's explore some essential features of the object-oriented paradigm. We will start with **encapsulation**, which is used to embed the access permissions of the methods and variables that are declared within the classes.

Encapsulation

Encapsulation is a way of binding data with the code into a single unit known as a capsule. This way, it provides security so that no unwanted modifications can be made to the code. The code that's written with the object-oriented paradigm will have critical data in the form of attributes. Thus, we have to prevent that data from being corrupted or becoming vulnerable. This is known as data hiding, and is the prime feature of encapsulation. To prevent data from being modified accidentally, encapsulation plays a vital role. We can make members of the class private members in order to embrace encapsulation. Private members, either methods or properties, can be made using double underscores at the beginning of their signature. In the following example, `__updateTech` is a private method:

```
class Bike:
    def __init__(self):
        self.__updateTech()
    def Ride(self):
        print("Riding...")
    def __updateTech(self):
        print("Updating your Bike..")
```

```
>>> honda = Bike()
Updating your Bike..
>>> honda.Ride()
Riding...
>>> honda.__updateTech()
AttributeError: 'Bike' object has no attribute '__updateTech'
```

In the preceding example, we were unable to invoke the `updateTech` method from the object of the class. This is due to encapsulation. We made this method private using a double underscore at the beginning of it. But sometimes we may need to modify the value of these attributes or behaviors. We can modify this using getters and setters. These methods will get the value and set the value for the attributes of the class. Thus, we can conclude that encapsulation is a feature of OOP that will prevent us from modifying and accessing data accidentally, but not intentionally. The private members of the class are not actually hidden; instead, they are just made unique from other members so that the Python parser will interpret them uniquely. The `updateTech` method is made unique and private using a double underscore (__) at the beginning of its name. The attributes of the class can also be made private using the same technique. Let's take a look at this now:

```
class Bike:
    __name = " "
    __color = " "
    def __init__(self,name,color):
        self.__name = name
        self.__color = color
    def info(self):
        print("{} is of {} color".format(self.__name,self.__color))

>>> honda = Bike("Honda", "Black")
>>> honda.info()
Honda is of Black color
```

We can clearly see that the `name` and `color` attributes are private as they begin with double underscores. Now, let's try to modify those values using an object:

```
>>> honda.__color = "Blue"
>>> honda.info()
Honda is of Black color
```

We tried to modify the color attribute of the `Bike` class, but nothing happened. This shows us that encapsulation will prevent accidental changes from being made. But what if we need to change it intentionally? This can be done with getters and setters. Take a look at the following example to find out more about getters and setters:

```
class Bike:
    __name = " "
```

```
    __color = " "
    def __init__(self,name,color):
        self.__name = name
        self.__color = color
    def setNewColor(self, color):
        self.__color = color
    def info(self):
        print("{} is of {} color".format(self.__name,self.__color))

>>> honda = Bike("Honda", "Blue")
>>> honda.info()
Honda is of Blue color
>>> honda.setNewColor("Orange")
>>> honda.info()
Honda is of Orange color
```

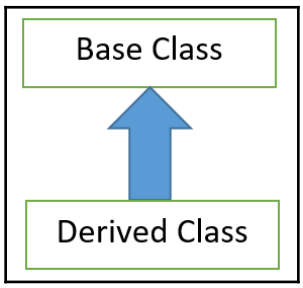
In the preceding program, we defined a `Bike` class with some private members, such as `name` and `color`. We used the `init` constructor to initialize the values of the attributes while creating the instance of the class. We tried to modify its value. However, we couldn't change its value because the Python parser treats these attributes as private. Thus, we used the `setNewColor` setter to set a new value for that private member. By providing these getters and setter methods, we can make a class either read-only or write-only, which prevents accidental data modification and intentional theft.

Now, it's time to take a look at another important feature of the object-oriented paradigm known as inheritance. Inheritance helps us write classes that will inherit each and every member from its parent class and also allows us to modify them.

Inheritance

Inheritance is the most important and well-known feature of the object-oriented programming paradigm. Do you remember the reusability feature of functions? Inheritance also provides reusability but with a lot of code. To use inheritance, we must have an existing class with some code inside it. This must be inherited by a new class. Such an existing class is called a **Parent** or **Base** class. We can create a new class as a `Child` class, which will acquire and access all the properties and methods of the parent class, so that we don't have to write the code from scratch. We can also modify the definitions and specifications of the methods that are inherited by the child class.

In the following illustration, we can see that the **Child** class, or the **Derived** class, is pointing to the **Base** or **Parent** class, which implies that there is a single inheritance:



In Python, it is easy to use inheritance. A `Child` class can inherit from a `Parent` class by mentioning the name of the `Parent` class within the brackets after the `Child` class. The following code shows how we can implement single inheritance:

```
class Child_class(Parent_class):
    <child-class-members>
```

A single class can also inherit multiple classes. We can achieve this by writing all of those classes' names within the brackets:

```
class Child_class(Base_class1, Base_class2, Base_class3 .....):
    <child-class-members>
```

Let's write a simple example so that we can understand inheritance a little more. In the following example, `Bike` will be the `Parent` class and `Suzuki` will be the `Child` class:

```
class Bike:
    def __init__(self):
        print("Bike is starting..")
    def Ride(self):
        print("Riding...")

class Suzuki(Bike):
    def __init__(self,name,color):
        self.name = name
        self.color = color
    def info(self):
        print("You are riding {0} and it's color is
          {1}".format(self.name,self.color))

#Save above code in python file and Run it

>>> suzuki = Suzuki("Suzuki", "Blue")
```

```
>>> suzuki.Ride()
Riding...
>>> suzuki.info()
You are riding Suzuki and it's color is Blue
```

Let's have a look at the preceding code and be amazed by inheritance. First, we created a `Base` class with two methods in it. After that, we created another class, that is, the child or derived class, called `Suzuki`. It is a child class because it has inherited the members of its parent `Bike` class with the `class Suzuki(Bike)` syntax. We added a couple of methods to the child class too. After creating these two classes, we created an object of the child class. We know that, when an object is created, the method that is going to be invoked automatically is a constructor, or `init`. Thus, we passed a value that was demanded by the constructor while creating the object of that class. After that, we made a call to the `Ride` method from the object of the `Suzuki` class. You can check the `Ride` method inside the body of the `Suzuki` class. It isn't there—instead, it's inside the suite of the `Bike` class. Due to inheritance, we were able to call the methods of the `Base` class as if they were inside the `Child` class. We can also use every property that's defined inside the `Base` class in the `Child` class.

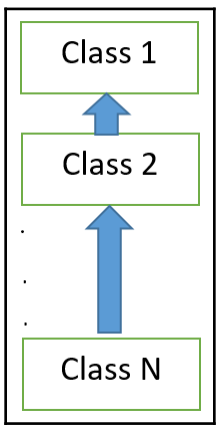
However, not all features are inherited inside the child class. When we create instances of the child class, the `init` method of the child class was called, but not those of the `Parent`. However, there is a way to call that constructor: by using the `super` method. This is shown in the following code:

```
class Bike:
    def __init__(self):
        print("Bike is starting..")
    def Ride(self):
        print("Riding...")

class Suzuki(Bike):
    def __init__(self,name,color):
        self.name = name
        self.color = color
        super().__init__()

>>> suzuki = Suzuki("Suzuki", "Blue")
Bike is starting..
```

The `super()` method refers to the superclass or `Parent` class. Thus, after the instantiation of the superclass, we made a call to the `init` method of that superclass.

It is similar to `Bike().__init__()`, but in this case **Bike is starting..** will be printed twice because the `Bike()` statement is going to create an object of the `Bike` class. This means that the `init` method will be called automatically. The second call is made with the object of the `Bike` class.

In Python, multi-level inheritance is available. This is a chained sequence that's created when any child class inherits from another child class. There are no limits regarding how a multi-level inheritance chain can be created. The following diagram depicts multiple classes inheriting features from their parent class:



The following code shows the features of multi-level inheritance. We have made three classes, with each one inheriting the features of the preceding one:

```
class Mobile:
    def __init__(self):
        print("Mobile features: Camera, Phone, Applications")
class Samsung(Mobile):
    def __init__(self):
        print("Samsung Company")
        super().__init__()
class Samsung_Prime(Samsung):
    def __init__(self):
        print("Samsung latest Mobile")
        super().__init__()

>>> mobile = Samsung_Prime()
Samsung latest Mobile
Samsung Company
Mobile features: Camera, Phone, Applications
```

Now that we've looked at inheritance, it's time to have a look at another feature, known as polymorphism. In a literal sense, **polymorphism** is the ability to accommodate different forms. Thus, this feature is going to help us use the same code in a different form so that multiple tasks can be carried out with it. Let's take a look.

Polymorphism

In the object-oriented paradigm, polymorphism allows us to define methods in the `Child` class with the same signature that's defined in the `Parent` class. As we know, inheritance allows us to use every method of the `Parent` class as if it were inside the `Child` class with the help of child class objects. However, we may encounter a situation where we have to modify the specification of the method that is defined inside the parent class so that it is executed independently of the `Parent` class. This technique is called method overriding. As the name suggests, we are overriding the existed method of the `Base` class with the new specification inside the `Child` class. Using method overriding, we can call both of the methods independently. If you have overridden a method of the parent class in the child class, then any version of that method (either the new one of the child or the old one of the parent) will be called based on the type of object it is being used to call. For example, if you want to call the new version of the method, you should call it with the `Child` class object. Speaking of the Parent class method, we have to use a `Parent` class object to call it. Thus, we can visualize that the two sets of methods have been developed but with the same name and signature, which signifies basic polymorphism. In programming, polymorphism is where the same function or method is used in different forms or types.

We can start thinking about examples of polymorphism from what we have learned so far. Do you remember the `len()` function? This is a built-in Python function and takes an object as a parameter. Here, an object can be anything; it can be a string, list, tuple, and so on. Even if it has the same name, it is not limited to performing a single task—it can be used in different forms, as shown in the following code:

```
>>> len(1,2,3) #works with tuples
3
>>> len([1,2,3]) #works with lists
3
>>> len("abc") #works with strings
3
```

Let's look at an example to demonstrate polymorphism with inheritance. We will write a program that will create three classes; one will be a `Base` class while the other two will be `Child` classes. The two `Child` classes will inherit each and every member of the `Parent` class, but each of them will have one method implemented independently. This will be the application of method overriding. Let's look at an example of polymorphism using the concept of polymorphism with inheritance:

```
class Bird:
    def about(self):
        print("Species: Bird")
    def Dance(self):
        print("Not all but some birds can dance")

class Peacock(Bird):
    def Dance(self):
        print("Peacock can dance")
class Sparrow(Bird):
    def Dance(self):
        print("Sparrow can't dance")

>>> peacock = Peacock()
>>> peacock.Dance()
Peacock can dance
>>> sparrow = Sparrow()
>>> sparrow.Dance()
Sparrow can't dance
>>> sparrow.about() #inheritance
Species: Bird
```

The first thing you see is that the `Dance` method is common among all three classes. But in each of these classes, we have different specifications for the `Dance` method. This feature is particularly useful because, in some cases, we may want to customize the method that is inherited from the `Parent` class, which may not have any significance in the `Child` class. In such cases, we redefine this method with the same signature that's inside the `Child` class. This technique of reimplementing a method is known as method overriding, and the different methods it creates using this process enable polymorphism.

Now that we've learned about the important concepts of object-oriented programming and their prime features, such as encapsulation, inheritance, and polymorphism, it's time to use this knowledge to modify the Snake game that we made using curses in the previous chapter. Since we can't use these object-oriented principles to make the code from the previous chapter less messy and abstruse, we will make our code more reusable and readable. We will start modifying our game with OOP in the next section.

Snake game implementation

We've explored various features of object-oriented programming in this chapter, including inheritance, polymorphism, data hiding, and encapsulation. One feature that we didn't cover, known as method overloading, will be covered in `Chapter 9`, *Data Model Implementation*. We have learned enough about OOP to make our code more readable and reusable. Let's start this section by following the conventional pattern, that is, brainstorming and information gathering.

Brainstorming and information gathering

As we have already discussed, object-oriented programming is not related to game interface programming; instead, it is a paradigm that makes code sturdier as well as more lucid. Thus, our interface will be similar to that of programs that are made by the curses module—text-based terminals. However, we will use the object-oriented paradigm to refine our code, and we will focus on the object rather than the actions and logic. We know that OOP is a data-driven methodology. Thus, our program must accommodate the game screen and user events data.

The main aims of using the object-oriented principle in our game are as follows:

- To divide programs into smaller parts, called objects. This will make programs more readable and allow us to track bugs and errors easily.
- To be able to communicate between objects through functions.
- Data is secure as it cannot be used by outer functions. This is called encapsulation.
- We will put more emphasis on the data rather than the methods or procedures.
- Making modifications to the program, such as adding properties and methods, can be done easily.

Now, let's start brainstorming and gather some information about the game model. Obviously, we have to use the same code from the previous chapter for the game layout and its characters, that is, `Snake` and `Food`. Thus, we have to take two classes for each of them. The `Snake` and `Food` classes will have methods defined in them that will control game layouts and user events.

We have to use curses events such as `KEY_DOWN`, `KEY_UP`, `KEY_LEFT`, and `KEY_RIGHT` to handle the movement of the snake character. Let's visualize the essential classes and methods:

1. First, we have to import curses to initialize the game screen and handle user key movements.
2. Then, we have to import the random module as we have to generate food in a random position once the snake has eaten it.
3. After that, we initialize the constants, such as screen height, width, default snake length, and timeout.
4. Then, we declare eh `Snake` class with a constructor, which will initialize the default position of the snake, window, head position, and the body of the snake.
5. Inside the `Snake` class, we will add a couple of methods, as follows:
    - `eat_food` will check whether the snake has eaten the food. If it has, the length of the snake will increase.
    - `collision` will check whether the snake has collided with itself.
    - `update` will be invoked every time the user makes a move and changes the position of the `Snake` character.
6. Finally, we declare the `Food` class and define the render and reset methods to generate and delete the food from a random position.

Now, let's start writing the program by declaring the constants and importing the essential modules. This is no different from the previous chapter—we will use curses to initialize the game screen and handle user events. We will use the random module to generate a random position on the game console so that we can generate new food at that position.

Declaring constants and initializing the screen

Similar to the preceding chapter, we are going to import the curses module so that we can initialize the game screen and customize it by specifying the height and width. We have to declare the default snake length and its position as constants. The following code will be familiar to you, except for the `name == "__main__"` pattern:

```
import curses
from curses import KEY_RIGHT, KEY_LEFT, KEY_DOWN, KEY_UP
from random import randint

WIDTH = 35
HEIGHT = 20
```

```
MAX_X = WIDTH - 2
MAX_Y = HEIGHT - 2
SNAKE_LENGTH = 5
SNAKE_X = SNAKE_LENGTH + 1
SNAKE_Y = 3
TIMEOUT = 100

if __name__ == '__main__':
    curses.initscr()
    curses.beep()
    curses.beep()
    window = curses.newwin(HEIGHT, WIDTH, 0, 0)
    window.timeout(TIMEOUT)
    window.keypad(1)
    curses.noecho()
    curses.curs_set(0)
    window.border(0)
```

In the preceding code, we have declared a bunch of constants to specify the height, width, default snake length, and timeout. We are familiar with all of these terms, except for the `__name__ == "__main__"` pattern. Let's talk about it in detail:

By looking at this pattern, we can conclude that the assignment of the `"main"` string is done in the name variable. Just like `__init__()` was a special method, `__name__` is a special variable. Whenever we execute our script file, the Python interpreter will execute the code that is written at the zero indentation level. But in Python, there is no `main()` function like there is in C/C++, which is invoked automatically. Thus, the Python interpreter will set the special `__name__` variable with the `__main__` string. Whenever the Python script is executed as a main program, the interpreter sets the special variable with the string. But when the file is being imported from another module, the value of the name variable will be set to that module name. Thus, we can conclude that the name variable will determine the current working module. We can evaluate how this pattern works as follows:

- **When the current source code file is the main program**: When we run the current source file as the main program, that is, `C:/> python example.py`, the interpreter will assign the`"__main__"` string to the special name variable known as `name == "__main__"`.

- **When another program is importing your module**: Suppose any other program is the main program and it is importing our module. The `>>> import example` statement will import the example module into the main program. Now, the Python interpreter will refine the name of the script file by removing the `.py` extension and setting that module name to the name variable, that is, `name == "example"`. Due to this, the code that is written in the example module will be available for the main program. After the special variable has been set up, the Python interpreter will execute the statements line by line.

Thus, the `__name__ == "__main__"` pattern can be used to execute the code that's written inside it if the source file is executed directly, and is not imported. We can conclude that the code that's written inside this pattern is the code that will be executed. Functions, classes, and the code inside them that isn't defined aren't going to run until they are called from the zero indentation level. This is due to the lack of a `main()` function in Python, which is automatically invoked in low-level programming languages.

In this case, the top-level code starts with an `if` block that's followed by the pattern's **name**, which evaluates the current working module. If the current program is `main`, we are going to execute the code that's written inside the `if` block, which initializes the game screen and creates a new window for the game by using curses.

Now that we have started writing a program that's initialized the game screen and declared some constants, it's time to create some classes. We have two characters in the game: `Snake` and `Food`. We will begin by creating two classes for now and modify them according to our needs. Let's start by creating the `Snake` class.

Creating the snake class

After creating the screen for our game, our next focus will be on rendering the game character in our screen. We will start off by creating the `Snake` class. We know that classes will have different members, that is, attributes and methods. As we mentioned in the previous chapter, while creating the `Snake` character, we have to track the *x* and *y* positions of the snake in the game window. To track the body position of the snake, we have to extract the *x* and *y* coordinates of the snake. We should use alphabetical characters to make up the body of the snake as curses only supports text-based Terminals. Let's start creating the `Body` class, which will provide us with the position of the snake and provide the character for the body of the snake:

```
class Body(object):
    def __init__(self, x, y, char='#'):
        self.x = x
```

```
            self.y = y
            self.char = char

        def coor(self):
            return self.x, self.y
```

In the preceding program, `#` is used to make up the body structure of the snake. We have defined two members inside the `Body` class: the constructor and the `coor` method. The `coor` method is used to extract the current coordinates of the snake body.

Now, let's create a class for the game characters. We will start with the `Snake` class. We should maintain a listed data structure so that we can store the body position of the snake. Initializing these properties should be done using a constructor. Let's start writing the constructor for the `Snake` class:

```
class Snake:
    REV_DIR_MAP = {
        KEY_UP: KEY_DOWN, KEY_DOWN: KEY_UP,
        KEY_LEFT: KEY_RIGHT, KEY_RIGHT: KEY_LEFT,
    }

    def __init__(self, x, y, window):
        self.body_list= []
        self.timeout = TIMEOUT
        for i in range(SNAKE_LENGTH, 0, -1):
            self.body_list.append(Body(x - i, y))
        self.body_list.append(Body(x, y, '0'))
        self.window = window
        self.direction = KEY_RIGHT
        self.last_head_coor = (x, y)
        self.direction_map = {
            KEY_UP: self.move_up,
            KEY_DOWN: self.move_down,
            KEY_LEFT: self.move_left,
            KEY_RIGHT: self.move_right
        }
```

Inside the `Snake` class, we made a dictionary. Each of the keys and values represents a reverse direction. If you are confused about how the direction on the screen is represented, go back to the previous chapter. The positions of the characters are represented in the coordinates. We declared the constructor, which allows us to initialize the properties of the classes. We made `body_list` to hold the snake body; a window object that represents the game screen for the snake game; the default direction of the snake, which is the RIGHT direction; and a direction map, which accommodates the movement of the character with curses constants such as `KEY_UP`, `KEY_DOWN`, `KEY_LEFT`, and `KEY_RIGHT`.

For every direction map, we make a call to the `move_up`, `move_down`, `move_left`, and `move_right` functions. We will create these methods shortly.

The following lines of code are declared inside the `Snake` class and will add the coordinates of the snake body to `body_list`. The `Body(x-i,y)` statement is the instance of `Body` class that will specify the coordinates of the snake's body. In the constructor of the `Body` class, `#` is used to specify the layout of the snake's body:

```
for i in range(SNAKE_LENGTH, 0, -1):
            self.body_list.append(Body(x - i, y))
```

Let's take a look at the preceding code and explore it. This code is going to extend the characteristics of the `Snake` class:

1. First, we have to begin by adding some new members inside the `Snake` class. We start by adding a simple method that will extend the body of the snake:

```
def add_body(self, body_list):
        self.body_list.extend(body_list)
```

2. Now, we have to create another method that will render game objects onto the screen. One of the important steps of this program is to render the snake's body onto the game screen. Since we have to represent the snake with`#`, we can use curses for this and use the `addstr` method. In the following render method, we looped the entire `body_list` of the snake and added `'#'` to each instance:

```
def render(self):
            for body in self.body_list:
                self.window.addstr(body.y, body.x, body.char)
```

3. Now, let's create the object of the `Snake` class. We can create it inside the `name == '__main__'` pattern:

```
if __name__ == '__main__':
 #code from preceding topic
 snake = Snake(SNAKE_X, SNAKE_Y, window)
 while True:
 window.clear()
 window.border(0)
 snake.render()
```

In the preceding program, we created a snake object. Since the constructor of the `Snake` class will be automatically invoked while creating an object of it, we passed in the `SNAKE_X` and `SNAKE_Y` arguments, which provide the default position of the snake and window, The window object screen is created by the `newwin` method from curses. Inside the while loop, we used the snake object to invoke the render method, which will add a snake in the game screen.

Although we have successfully rendered the snake into the game console, our game isn't ready to test yet because the program is unable to address certain actions, for example, whenever the user presses the LEFT, RIGHT, UP, and DOWN keys on the keyboard to move the `Snake` character. We know that the curses module provides us with a method so that we can get input from the user, and we can handle it accordingly.

Handling user events

As we saw in the previous chapter, it is really easy to take input from the user and handle it using the curses module. In this section, we are going to add those methods inside the `Snake` class because methods related to the user's actions are related to the movement of the `Snake` character. Let's add a couple of methods inside the `Snake` class:

```
def change_direction(self, direction):
        if direction != Snake.REV_DIR_MAP[self.direction]:
            self.direction = direction
```

The preceding method is going to change the direction of the snake. Here, we have initialized the `REV_DIR_MAP` directory, which contains the key and value that represent their opposite directions. Thus, we pass the current direction to this method to change it based on the event that's pressed by the user. The direction argument is inputted from the user.

Now, it's time to extract the head and coordinates for the head of snake. We know that the head position of the snake changes while the snake moves. Even when crossing the boundary of the snake, we must make the snake appear from the other side. Thus, the snake's head position will change according to the user's movements. We need to create a method that can accommodate these changes. We can use the property decorator for this, which will treat changing the head properties of the `Snake` class as a method. This works like a getter. Don't be overwhelmed by these terms, as we will cover these in a later chapter (List Comprehension and Properties). This being said, let's take a look at the following example. This example will help you understand the `@property` decorator:

```
class Person:
    def __init__(self,first,last):
```

```
        self.first = first
        self.last = last
        self.email = '{0}.{1}@gmail.com'.format(self.first, self.last)

per1 = Person('Ross', 'Geller')
print(per1.first)
print(per1.last)
print(per1.email)

#output
Ross
Geller
Ross.Geller@gmail.com
```

Now, let's change the value of the `first` attribute and print all those values:

```
per1.first = "Rachel"
print(per1.first)
print(per1.email)

#output
Rachel
Ross.Geller@gmail.com
```

You can clearly see that the change has not been reflected in the email. The name for the email has been preserved from the previous value of `Ross`. Thus, in order to make the program accommodate changes spontaneously, we need to make the attributes property decorators. Let's make the email a property and observe the result:

```
class Person:
    def __init__(self,first,last):
        self.first = first
        self.last = last

    @property
    def email(self):
        return '{0}.{1}@gmail.com'.format(self.first,self.last)
```

The following code is executed in the Python shell:

```
>>> per1 = Person('Ross', 'Geller')
>>> per1.first = "Racheal"
>>> per1.email()
Racheal.Geller@gmail.com
```

The change we have made to the attribute has been reflected spontaneously in the attribute of the class with the help of the decorator property. We will learn about this in detail in the next chapter. This was just a quick introduction.

We only covered it because it's an essential part of making the head attribute of the snake a property decorator:

```
    @property
     def head(self):
         return self.body_list[-1]

     @property
     def coor(self):
         return self.head.x, self.head.y
```

The `head` method is going to extract the last element of the list, which indicates the head of the snake. The `coor` method is going to return a tuple containing the (*x*,*y*) coordinates, which represent the head of the snake.

Let's add one more function that will update the direction of the snake:

```
 def update(self):
        last_body = self.body_list.pop(0)
        last_body.x = self.body_list[-1].x
        last_body.y = self.body_list[-1].y
        self.body_list.insert(-1, last_body)
        self.last_head_coor = (self.head.x, self.head.y)
        self.direction_map[self.direction]()
```

The preceding `update` method is going to pop out the last part of the body and insert it with the head position before updating the new head position.

Now, let's handle the user events using the curses module:

```
if __name__ == '__main__':
    #code from preceding topic
    #snake is object of Snake class
    while True:
        event = window.getch()
         if event == 27:
            break

        if event in [KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT]:
            snake.change_direction(event)

          if event == 32:
            key = -1
            while key != 32:
                key = window.getch()

        snake.update()
```

We learned about the working mechanism in the preceding code in the previous chapter, so you shouldn't have any problems grasping it. Now, let's make the snake move in a certain direction. Previously, in the `Snake` class, we added the `direction_map` attribute, which held the dictionary mapping to different functions, such as `move_up`, `move_down`, `move_left`, and `move_right`. These functions will change the position of the snake based on the user's action:

```
#These functions are added inside the Snake class
 def move_up(self):
        self.head.y -= 1
        if self.head.y < 1:
            self.head.y = MAX_Y

    def move_down(self):
        self.head.y += 1
        if self.head.y > MAX_Y:
            self.head.y = 1

    def move_left(self):
        self.head.x -= 1
        if self.head.x < 1:
            self.head.x = MAX_X

    def move_right(self):
        self.head.x += 1
        if self.head.x > MAX_X:
            self.head.x = 1
```

We made this logic in the previous chapter and will make the snake move either up, down, left, or right. We can imagine the screen as a matrix containing rows and columns. With the up action, the snake will move in the Y-axis, and so the y position should be decreased; similarly, with the down action, the snake will move to down the Y-axis, and so we need to increment the y coordinate. For the LEFT and RIGHT movements of the snake, we will have to decrement and increment the X-axis, respectively.

Now, that we have handled user events, this concludes the `Snake` class. It's time to handle the collision, if there is one. We also have to add another character to the game, that is, `Food`, which will be made by creating a new class.

Handling collisions elp of decorator property.

No noble logic will be created in this section. We have to check whether the head of the snake has collided with the body part of the snake. This should be done by checking the coordinates of the head (y,x) against any of the coordinates of the snake's body. Thus, let's make a new method, `@property`, which will check for the collision:

```
    @property
    def collided(self):
        return any([body.coor == self.head.coor
                    for body in self.body_list[:-1]])
```

In the preceding example, any function will return `True` if any item in the iterable is `True`; otherwise, it will return `False`. The statement inside the `any` function is a list comprehension statement that checks whether the coordinates for the head of the snake are the same as the coordinates for any part of the body of the snake.

Now, let's invoke this method with the `snake` object in our main loop:

```
if __name__ == "__main__":
     while True:
        #code from preceding topics
        #snake is Snake class object
        if snake.collided:
            break
```

Adding the food class

The next character we need to add to our game is `Food`. As we have already said, we have to make a different class for each character because they should have different behaviors and attributes. Let's create another class for the `Food` character. We will call this the `Food` class:

```
class Food:
    def __init__(self, window, char='&'):
        self.x = randint(1, MAX_X)
        self.y = randint(1, MAX_Y)
        self.char = char
        self.window = window

    def render(self):
        self.window.addstr(self.y, self.x, self.char)

    def reset(self):
```

```
            self.x = randint(1, MAX_X)
            self.y = randint(1, MAX_Y)
```

If you read the *Python classes* section in this chapter carefully, this section should not create any confusion for you. To create a class in Python, we use the `class` keyword, followed by the class name. However, we have to use parentheses to show inheritance. If you left the parentheses empty, they will throw an error. Thus, we have added an object inside the parentheses, which is optional. You can simply remove the parentheses and they will work perfectly. We used the `randint` method from the random module to create food in a random position. The render method is going to add the `X` character to the specified (y,x) position.

Now, let's create the object of the `Food` class and render the food on the screen by invoking the `render` method:

```
if __name__ == '__main__':
    food = Food(window, '*')
    while True:
        food.render()
```

As you may recall, the logic that we've created to make the snake eat the food is the same logic that we used for the snake head coordinate colliding with the food coordinate. Before we actually make that logic, we will make another method for the `Snake` class that will add the logic for the aftermath of eating the food:

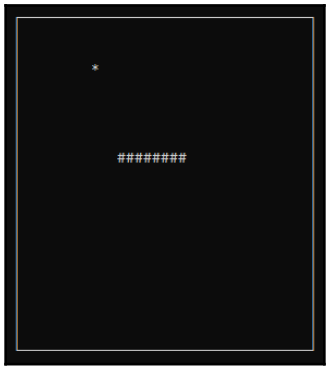
```
def eat_food(self, food):
    food.reset()
    body = Body(self.last_head_coor[0], self.last_head_coor[1])
    self.body_list.insert(-1, body)
```

The preceding logic is going to be called after the snake eats the food. After eating the food, we will reset it, which means the food will be generated in the next random position. We will then make an increment in the body position by adding the last coordinate of the food to the body of the snake.

Now, let's add some logic that will make sure we invoke this method. As we have already discussed, the logic will be simple: whenever the head of the snake collides with the position of the food, we will invoke the `eat_food` method:

```
if __name__ == '__main__':
#snake is object of Snake class
#food is object of Food class
    while True:
        if snake.head.x == food.x and snake.head.y == food.y:
            snake.eat_food(food)

curses.endwin()
```

Let's run our game and observe the output:



Finally, we have modified the game with the object-oriented paradigm. You might feel that working with classes and objects is more complicated and lengthy, but with more practice, you will become more comfortable with it. That being said, OOP has provided more readability and reusability features in our program. As an example, if you find a bug in the `Snake` character, you can simply track it down by overlooking the unnecessary code for the food. Now, let's hop over to the next section, where we will test the game and make the necessary modifications to it.

Game testing and possible modification

The curses application cannot be run directly from the Python script by pressing *F5*. Thus, we have to run it externally from the command prompt with the `filename.py` command.

Now, let's add the score to our game:

1. First of all, we have to initialize the score value as 0 at the `Snake` class. We will also add a `score` method in the `Snake` class:

```
class Snake:
    self.score = 0
    @property
    def score(self):
        return 'Score : {0}'.format(self.score)
```

2. Now, we have to increase this score every time the snake eats the food. The method that will be called after the snake eats food is the `eat_food` method. Thus, we will increase the score inside this method:

```
def eat_food(self, food):
    food.reset()
    body = Body(self.last_head_coor[0], self.last_head_coor[1])
    self.body_list.insert(-1, body)
    self.score += 1
```

3. Now, let's render the score with the `addstr` method of the curses window object:

```
while True:
    window.addstr(0, 5, snake.score)
```

4. The preceding statement will call the `score` method from the snake object and add the score at the (0,5) position. Remember that, in curses, the first position is y and the second is x.

Let's run our game one more time:

Summary

In this chapter, we learned about one of the most important paradigms in programming—object-oriented programming. We covered all the concepts of classes and objects to make it easier for you to read and write your own code. We also explored how to define the members of a class and access them. We got familiar with the features of the object-oriented approach by implementing hands-on examples. We also learned about inheritance, encapsulation, polymorphism, and method overriding. These features will be used in upcoming chapters too, so make sure you have a good grasp of each of these topics.

In the next chapter, we will learn about list comprehension and properties. The aim of the next chapter is to find a way to optimize code to make the program shorter and faster in terms of its execution. We will look at how to work with conditions and logic in order to implement one-line code that will be more readable and easier to debug. We will also use that concept to modify our Snake game.

7
List Comprehension and Properties

Necessity is the Mother of Invention is a popular English proverb which means that any pioneer ideas that have been invented so far or will be invented are because of their need. For instance, the giant video hosting platform YouTube became popular not only because of its business model but also because of the time it was introduced. Many creative artists such as video editors, singers, dancers, and gamers wanted the platform to be recognized globally without any initial investments, and audiences wanted a platform where they could learn and be entertained free of charge. Thus, the need is the driving force for any new invention. However, this doesn't mean that every revolutionary idea that was created at the right time succeeded. Some of them have failed miserably because they didn't address the limitations that were posed by the technology. Our quixotic imagination is fettered by these technologies, and, although we have been progressing, we are not there yet.

Thus, in order to make any revolutionary ideas successful, we have to know our limitations. Our primary limitations are memory space and processing power. Taking care of these limitations, this chapter will teach us to write an elegant program that will save memory storage and running time to some extent. We will learn about the comprehension and generation that are provided by Python. They will make the program run faster while maintaining its readability.

The following topics will be covered in this chapter:

- Overview of code complexities
- For loop versus list comprehension
- Decorators
- Python property
- Refining the snake game with LC and property

Technical requirements

You will need the following requirements to complete this chapter:

- Python version 3.5 or newer
- Python IDLE (Python's built-in IDE)
- A text editor
- A web browser

The files for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter07`

Check out the following video to see the code in action:

`http://bit.ly/2pzX8Au`

Overview of code complexities

So far, we have been learning about the basics of Python, such as functions, data structures, and object-oriented programming. Now, we are able to create our own logic and even program some games too. As we continue to add features for these games, we are expected to have millions of lines of code. Those huge **lines of code** (**LOC**) will be hard to understand, interpret, and process. For example, in some cases, we may have to make a trade-off between code maintainability and optimization. Let's suppose you maintain the code for a shopping website and one day you got millions of hits on your website, which is beyond the processing speed of your server. Now, you have to accommodate a situation in which you must either serve the page without a delay and give the customers no proper recommendations about products or serve the page with little delay and give proper recommendations.

On the other hand, we may want to achieve some amount of code optimization. If any program takes seconds to execute, then after optimization, we may want to run it within a millisecond. Now, we may think this time is negligible, which it is on the first run. However, when we have to run the same program a thousand times, we may cut off some seconds and this could be potentially useful for any real-time applications.

In this chapter, we will focus on the ways we can modify our code to improve its quality and efficiency. Any original program can be said to have been optimized if we managed to make its code shorter, reduce memory consumption and increase its execution speed and the interaction of fewer input/output instructions. The basic canonization followed by optimization is that the outcome of the optimization must have the same output and consequences as that of the non-optimized one.

However, these requirements may be insignificant whenever the achieved optimized program has favorable results over the non-optimized one in terms of time and space complexities. For example, in the rocket-launch activities, we may want the real-time data of the surrounding area while trading for the accuracy of the data. Thus, optimization is important in such cases even though it might affect the system's output in one way or another.

Before we learn about optimization, we will look at the necessity for it. In order to check the room for optimization, we have to analyze the code first, and the prime way to analyze it is by using complexity analysis. Algorithm complexity analysis is a tool that will explain the behavior of the program as the size of the program increases. The size increases when input to the program increases. Thus, we have to check the program against the mathematical `f(n)` function, where *n* represents the input to the programs. Now, you may be wondering whether running this algorithm may cause a difference in time units, depending on the different computers that are used by companies such as NASA or Apple Inc. as they will have higher processing power than our simple computer. Therefore, it might be an injustice to judge the algorithm that is running on our PC. If you're ever faced with such ambiguity, just pat on yourself on the back as you are thinking like a programmer. To test whether the algorithm is independent of processing speed, disk power, and powerful software, scientists have developed something called symptotic analysis. This analysis will check the algorithm against the size of the input and without recording the time it took to execute it. We call this **time complexity**, and it allows us to check how the algorithm runs with respect to the size of the input data. To observe the time complexity of the algorithm, the best and well-known notation we should use is Big-Oh notation. This notation will help us analyze the worst-case scenario of the algorithm and help us optimize it. Let's analyze the following complexities using some simple examples:

- `O(1)`: This notation is used to define the algorithms that are independent of input size. Increasing or decreasing any sets of data from the input might not affect the execution speed of the algorithm:

```
arr = [1,2,3,4,5]
for i in arr:
    print(arr[0])
```

The preceding program is going to print the first element of the array, no matter what data is in it. Thus, it has a time complexity of `O(1)`. This is considered a best-case scenario and is hard to achieve in a real-life scenario.

- `O(n)`: This notation describes the algorithm that will have a linear increase in running time as the size of the input data, `(n)`, increases. For example, in the following program, the worst-case scenario may lead us to iterate over the whole list. Thus, performance depends on the size of the input:

```
n = int(input("Enter any number"))
for i in range(1,100):
    if i == n:
        print(i)
        break
```

- $O(n^2)$: This notation specifies the performance of algorithms, which is proportional to the square size of the input data. It is highly common in nested loops.

There are a few more notations, such as $O(2^N)$ and `O(log N)`, but we don't need to go any further as we have already learned enough so that we can differentiate between good and bad code.

Now that we have gained enough information about optimization and the way we can analyze algorithms, it's time to look at some examples to clarify the differences between non-optimized and optimized code. Before diving into the algorithmic analysis of the following code, we will learn how to analyze the complexities of the programs. Since this book is not going to teach advanced algorithmic concepts, we will take a look at the basic ideas to evaluate performance and optimization. This will provide you with a tool that will help you write programs that are shorter, readable, and don't waste memory resources. Thus, this practice will make us able to make proper decisions while differentiating between the algorithms in terms of their efficient use of resources, which means time and memory, depending on the scenario. Let's get started by taking a look at the following code:

```
for i in range(1, 10):
    for j in range(i):
        print(i, end='')
    print()

#output
1
22
333
4444
```

```
55555
666666
7777777
88888888
999999999
```

In the preceding code, we used two nested `for` loops to get the desired output. In the case of the first for loop, it takes all the elements of the range one by one and for each iteration, we make a second for loop. For the second loop, we will have a range of the same elements with the same number of counts. For example, for element 2, we will have [2,2] for the second j loop, thus printing the same number multiple times. If you followed the preceding chapters properly, this code shouldn't be hard to understand. Now, let's observe the fun part. We know that the first iloop is going to iterate into the whole range of datasets, which will lead to the time complexity of `O(n)`. The same goes for the j-loop. Thus, the total time complexity will be `O(n) * O(n)`, which will result in $O(n^2)$. This represents an expensive algorithm. We have to try and convert the programs with nested loops into single loops, as follows:

```
for i in range(1, 10):
        print (str(i) * i)

#output
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

The preceding program contains a single for loop, and so it will loop the entire datasets once, which will result in only `O(n)` and not $O(n^2)$.

You may be wondering why these things are so important and why we covered them in this chapter. The answer is simple. Although in some applications written by Python, that is, Android applications or websites, saving some milliseconds would be unnecessary. But, in a large application that's handling gigantic amounts of data, this time measurement can be increased. For example, let's think about an application calling a function to predict whether the news is fake or not. Let's say the non-optimized code would take a few seconds to make a prediction and that optimization would take some milliseconds. Here, the quantity would seem small but imagine we are calling the same function 1 million times. Now, calculate the time that would be saved as a whole: 277.5 hours.

That's cumbersome, isn't it? Python provides two constructs to facilitate faster and efficient processing of these huge data collections: comprehension and generators. There are three types of comprehensions, that is, list, dict, and set. First, we will delve into learning about list comprehension. Then, we will explore the other two (dict and set) by relating to them.

For loop versus list comprehension

Since we've been coding our program with loops since `Chapter 3`, *Flow Control – Building a Decision Maker For Your Game*, we are quite familiar with looping patterns, especially for loops. They are going to iterate through some items and, at each iteration, the iterating variable is going to perform some manipulation. The power of for loops can be alleviated by combining it with the appropriate data structure, like so:

```
new_list = []
for i in range(10):
    new_list.append(i)
print(new_list)

#output
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

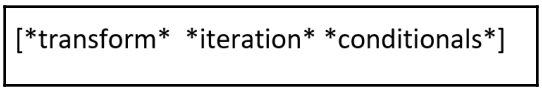
Python has another easier way of doing the same thing, and is known as list comprehension. The output of list comprehension will always be a list, which will be the result of the evaluation of the expression in the context of the for loop. This is followed by if conditionals. The code that emulates the for loop with expressions and conditionals by using list comprehension will be single-line code. Thus, code that's written using list comprehension is shorter and easily maintainable. To understand how list comprehension works, we have to be familiar with its pattern. We'll learn about the list comprehension pattern in the next section.

List comprehension pattern

In this section, we will use list comprehension to modify the preceding code that was written by a for loop. The result of list comprehension is a list. The pattern inside the square bracket is an expression followed by a loop, as follows:

```
new_list = [i for i in range(10)]
print(new_list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In the preceding code, the left-hand side object, that is, `new_list` represents the output list that stores the result of list comprehension. On the right-hand side expression, the statement enclosed by square brackets will result in list comprehension. First, we pass the expression to be performed, then the loop and conditionals (if any). The following illustration represents the pattern for list comprehension:



Let's take a look at a simple example to explain the preceding pattern:

```
even_power  = [i * i          for i in range(5)         if i % 2 == 0]
print(even_power)
[0, 4, 16]
```

The first statement inside the square brackets represents an expression. There can only be a single expression while we use list comprehension, unlike the body of a `for` loop. After the expression, we apply spaces and provide iteration. We can add nested loops too. After adding the iterations, we have to specify the conditionals, if there are any. List comprehension is widely used to concatenate the elements of two lists and create a new one, like so:

```
numbers = [1,2,3,4,5]
alphabets = ['a','b','c','d','e']

new_list = [[n,a] for n in numbers for a in alphabets]
print(new_list)

[[1, 'a'], [1, 'b'], [1, 'c'], [1, 'd'], [1, 'e'], [2, 'a'], [2, 'b'], [2,
'c'], [2, 'd'], [2, 'e'], [3, 'a'], [3, 'b'], [3, 'c'], [3, 'd'], [3, 'e'],
[4, 'a'], [4, 'b'], [4, 'c'], [4, 'd'], [4, 'e'], [5, 'a'], [5, 'b'], [5,
'c'], [5, 'd'], [5, 'e']]
```

The preceding code was able to create a complex list of lists. The comprehensions are not only limited to lists; there's also dict and set comprehensions. As for the list, we used square brackets to perform comprehension. For set and dict comprehension, we need to use curly braces `{}`. Note, however, that the patterns will be similar for all of these comprehensions. Let's take a look at an example:

```
dict_comp = {x:chr(65+x) for x in range(1, 6)}
print(dict_comp)
{1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
```

The preceding code represents the usage of dict comprehension. The pattern is similar to list comprehension except we use curly braces to perform comprehension. The result of dict comprehension will be a dictionary. Similarly, in the case of set comprehension, the result of comprehension will be set. This is shown in the following code:

```
set_comp = {x ** 2 for x in range(5) if x % 2 == 0}
type(set_comp)
print(set_comp)

#output
<class 'set'>
{0, 16, 4}
```

Before wrapping up this section, we have to cover two powerful built-in functions of Python that manipulate the data of collections faster than ever. If you ever got a chance to learn about big data, you should have heard these two words: zip and map. Python has provided these two functions in order to work with a high amount of data with minimal load and faster computation. Let's look at a simple example to understand the concepts of zip and map. Let's say we have two lists containing limited integers. Now, you have to write a program to create a new list that will store the smallest number from each of them. A comparison will be made between the elements that have the same indexes:

```
Input: a = [2,3,4,5,6,7] and b = [0,3,2,1,3,4]
Output: [0, 3, 2, 1, 3, 4]
```

The most simple and conventional approach is shown in the following code:

```
a = [2,3,4,5,6,7]
b = [0,3,2,1,3,4]
result = []
length = len(a)
for i in range(length):
    result.append(min(a[i],b[i]))
print(result)
#output
[0, 3, 2, 1, 3, 4]
```

Now, let's learn about the other way we can perform the preceding computation. This is done with a single line of code which is made by using the `zip` and `map` functions. The `zip` function is a simple Python built-in function that takes two objects of equal length and merges them together. If you pass two lists of equal length to the `zip` function, it will merge it into a single one so that computation can be performed within a single object. This is shown in the following code:

```
>>> numbers = [1,2,3]
>>> letters = ['a','b','c']
```

```
>>> list(zip(numbers,letters))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

We know that the comparison between the numbers should be made since they have the same indexes. Thus, we can combine the original array of numbers with the `zip` function so that we can store tuples of numbers in the single list, like so:

```
>>> list(zip(a,b))
[(2, 0), (3, 3), (4, 2), (5, 1), (6, 3), (7, 4)]
```

Map function

The prime task of programming is to perform a computation. The operation that's done on elements can be done independent of one another; that is, we can perform a comparison on lists a and b separately, like we did in the preceding code, or simply merge them so that comparison can be done faster. The `zip` method is able to merge two objects that are the same length into a new iterable. Now, the major task is to create a comparison operation and use it on each element of the iterable, which is done by using the `map` function. The `map` function takes some function and applies it to each of the elements of the iterable.

According to Python's official documentation, map can be described as follows:

Map applies a function to every item of iterable and returns a list of the results. If additional iterable arguments are passed, the function must take that many arguments and is applied to the items from all the iterables in parallel. If one iterable is shorter than another, it is assumed to be extended with None items. If the function is None, the identity function is assumed; if there are multiple arguments, `map()` *returns a list consisting of tuples containing the corresponding items from all the iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list.*

The argument that's passed when we call map function is a function that's followed by iterables. Normally, we use an anonymous or lambda function, such as `some_function`, which takes some positional arguments and returns them as a tuple. This is shown in the following code:

```
map(some_function, some_iterables)
```

Let's create a simple example to illustrate the use of the `map` function:

```
>>> map(lambda x: x*2, (1,2,3,4))
<map object at 0x057E9AF0>
```

The preceding code was not fruitful as the `map` function doesn't return any iterables or objects. Instead, it prints the string representing the map object. To achieve a desirable result, we have to wrap the `map` method with a list constructor, like so:

```
>>> list(map(lambda x: x*2, (1,2,3,4)))
[2, 4, 6, 8]
```

Now, we will use the concepts of the `map` and `zip` functions to find the list of minimum elements from the two lists. The following code is pretty simple; we started by defining two arrays. After that, we used the `map` function, which will take the `lambda` function containing the comparison operation and `zip` method and merge the two arrays into the list of tuples. Each pair of tuples made by the `zip` method are passed to `lambda` functions for comparison:

```
>>> a = [2,3,4,5,6,7]
>>> b = [0,3,2,1,3,4]
>>> list(map(lambda pair: min(pair), zip(a,b)))
[0, 3, 2, 1, 3, 4]
```

With the power of `map` and `zip`, we can do anything, similar to list comprehension. With the preceding program complete with list comprehension, the `map` function, and a for loop, we can see the following runtime performance:

```
For Loop: 4.56s
List comprehension: 2.345s
Map: 2..11s
```

Thus, these three features of Python primarily enable the manipulation of collections faster than anything. But in terms of code maintainability and readability, list comprehension tops the list as it provides us with a way to customize the inner workings of programs effectively. Now, it's time to learn about another feature of Python, known as decorators. These allow us to modify the functionality of an existing object without modifying its current structure.

Decorators

A decorator is a design pattern that adds new functionality to an existing object without deferring its original structure. We must be comfortable with the fact that everything in Python is an object – even functions. The different names that are used for defining these objects are just their identifiers. Let's run the following code:

```
def fun1(info):
    print(info)
```

```
fun1("Good Morning")
fun2 = fun1
fun2("Good Morning")
```

When we run the preceding code, the `fun1` and `fun2` functions print the same output of `"Good Morning"` as both refer to the same objects (functions). Thus, functions are just objects with attributes. Let's get back to decorators. In a basic sense, a decorator is a construct where a part of the program tries to change the behavior of another part of the program at compile time. In the case of functions, the decorator takes a function, adds unique functionality to it, and eventually returns it, as follows:

```
def decorate_it(func):
    def inner():
        print("Decorated")
        func()
    return inner

def non_Decorated():
    print("Not-Decorated")
```

Now, let's try to run the preceding code from the Python shell:

```
>>> non_Decorated()
Not-Decorated

#now try to decorate the above function
>>> decorate = decorate_it(non_Decorated)
>>> decorate()
Decorated
Not-Decorated
```

In the preceding example, `decorate_it()` is a decorator that takes a non-decorated function as an argument. The `decorate = decorate_it(non_Decorated)` statement is an assignment, where the `Non_Decorated` function was passed to the decorator and it returned the function called decorate. Thus, we can conclude that decorators are callables that return a callable. In the preceding example, we can see that the `decorate_it()` decorator added some functionality to the `non_Decorated` or ordinary function. When decorators started getting famous, the introduced design pattern was based on decorating the function first and returning to the name of the second callable, just like we did in this example. However, programmers found this job to be redundant. Thus, they developed another syntax that simplified the preceding construct: using the`@` symbol.

To decorate an ordinary function, we use the `@` symbol, along with the decorator's name, and place it at the top of the non-decorated function, like so:

```
@decorate_it
def non_Decorated():
    print("Not-Decorated")
```

The preceding code is auxiliary to the following code, which we wrote earlier:

```
def non_Decorated():
    print("Not-Decorated")

decorate = decorate_it(non_Decorated)
```

Let's look at another example. We want to make a decorator that acts like an exception handler that throws error messages whenever unusual activity is encountered by the programs. The preceding decorator was simple since it wasn't concerned about the argument that was passed to the inner function. Now, we are going to make a program that will multiply any two numbers but also handle the error if any other data is passed, such as a string or complex numbers:

```
def multiply(a,b):
    print(a*b)

>>> multiply(2,5)
10
>>> multiply('c', 'f')
TypeError: can't multiply sequence by non-int of type 'str'
```

Now, we will try to make a decorator that will check whether we got an exception, like in the preceding code, and handle it automatically:

```
def smart_multiply(func):
    def inner(a,b):
        if (a.isdigit() and b.isdigit()):
            a = int(a)
            b = int(b)
            print("multiplying",a," with ",b)
            return func(a,b)
        else:
            print("Whoops!! Not valid multiplication")
            return
    return inner

@smart_multiply
def multiply(a,b):
    print(a*b)
```

```
a = input("value of a: ")
b = input("value of b: ")
multiply(a,b)
```

As soon as you run the previous code, you will be asked for entries in the Python Shell. You have to enter two entities for `a` and `b`, and then the code does the rest:

```
value of a: 4
value of b: 5
multiplying 4 with 5
20
```

Let's run the preceding code one more time. This time, we will input the values of `a` and `b` as strings:

```
value of a: t
value of b: y
Whoops!! Not valid multiplication
```

As you can see, the `inner` function of the decorator has the same number of arguments as those that were passed in by the non-decorated function. Thus, generalization can be done with `inner(*args, **kwargs)`, where `args` is the tuple of positional arguments and `kwargs` represents the dictionary of keyword arguments. Now, we can make decorators that will work with any number of arguments, as follows:

```
def universal(func):
    def inner(*args, **kwargs):
        print("It works for any function")
        return func(*args,**kwargs)
    return inner
```

Thus, at compile time, decorators modify the operations of the original function, methods, or even classes without altering the code of the objects being decorated. This ultimately leads to the use of **don't repeat yourself** (**DRY**) technique. In the next section, we are going to learn about the `@property` decorator – a built-in decorator of Python for implementing the `property()` function. As you may recall from the previous chapter, this construct of `@property` has already been used and it was defined as a Pythonic way of implementing getters and setters. Now, we will learn about it in detail.

Python property

To understand the usage of the property in the first place, we have to recall one of the principles of the object-oriented paradigm: data encapsulation. This bundles the data with the methods as a single capsule. The methods that are going to get and set the attributes of the classes are getters and setters. This principle of OOP infers that the attributes of the class must be made private so that accident modification or theft is prevented. Let's look at a simple example to begin:

```
class Speed:
    def __init__(self, speed = 0):
        self.speed = speed

    def change_to_mile(self):
        return (self.speed*0.6213,"miles")
```

In the preceding code, we made a class called `Speed` that stores the speed of the vehicle in kilometers. It has `members` as a method that converts kilometers into miles. Now, we can make the objects of the `Speed` class and manipulate the members of this class as we like. We will use the Python Shell for this, like so:

```
>>> car = Speed()
>>> car.speed = 45
>>> car.speed
45
>>> car.change_to_mile()
(27.958499999999997, ' miles')
```

Whenever assignment is done to the attributes of the class, the Python interpreter maintains the dictionary where the attributes and their values are maintained as key and value. In the case of the `Speed` class, we can retrieve any attributes of the object, that is, `speed`, with `__dict__ attributes`:

```
>>> car.__dict__
{'speed': 45}
```

Thus, whenever we execute the `car.speed` operation, the Python interpreter makes a search in the preceding dictionary and fetches the value as `car.__dict__['speed']`.

Now, let's assume that the preceding code became popular worldwide in the field of traffic control. One day, traffic police argued that there should be constraints in terms of the speed of a vehicle so that law can be enforced. Now, we have to modify the code in such a way that, if any driver drives too fast, the program provides them with a warning message. We can do this using getters and setters. Inside the `setter` method, we can explicitly check the maximum speed of the vehicle using conditionals. This can be done as follows:

```
class Speed:
    def __init__(self, speed = 0):
        self.set_speed(speed)

    def change_to_mile(self):
        return (self.get_speed*0.6213," miles")
#new updates are made as follows using getter and setter
    def get_speed(self):
        return self._speed
    def set_speed(self, km):
        if km > 50:
            raise ValueError("You are liable to speeding ticket")
        self._speed = km
```

In the preceding code, two major modifications were done and we are familiar with them. They are the `getter: get_speed` method and `setter: set_speed` method. Another change that was made in the code is the signature of the attribute. The speed attribute begins with a single underscore, which makes it private (data encapsulation). Try the following code in the Python Shell:

```
>>> car = Speed(30)
>>> car.get_speed()
30
>>> car.set_speed(38)
>>> car.get_speed()
38
>>> car.set_speed(70)
ValueError: You are liable to speeding ticket
```

The update to the original program was successfully reflected with new ranges of restriction. The driver is not allowed to drive their vehicle at a speed of more than 50km/hr.

Now, let's run the preceding code and observe the overhead that might be caused by the new updates. We can simply compare the code that was written with the getter and the setter with the code that was written without them. A major headache will arise when you try to accommodate the original code with the new changes as you have to modify your code from calling the attributes of the `car.speed` object to calling the attributes of `car.get_speed()`. The constructor must be changed to `car.set_speed` (speed). We might find it easier to make changes in this program, but imagine if the program had 10,000+ lines of code. It would be a hard time for any programmer to update and synchronize it with the new code. Now, here comes the property decorator in action. The following code solves this problem for us:

```
class Speed:
    def __init__(self, speed = 0):
        self.speed = speed
```

```
    def change_to_mile(self):
        return (self.speed*0.6213," miles")

    @property
    def speed(self):
        return self._speed
    @speed.setter
    def speed(self,km):
        if km > 50:
            raise ValueError("You are liable to speeding ticket")
        self._speed = km
```

Since we are familiar with decorators, the preceding construct should be familiar to us. Now, let's run our code in the Python Shell:

```
>>> car = Speed(40)
>>> car.speed
40
```

Using the property construct, we modified our original class and provided some constraints too. But this time, we removed the changes we made, such as `get_speed` and `set_speed`, that were added by the getter and the setter. Thus, the traffic control system can use this new code without making any changes to the original code, which leads to backward compatibility.

We also have another way of implementing the preceding code, which is by using the `property()` function. The following code is equivalent to the preceding code being written with the `@` property construct:

```
class Speed:
    def __init__(self, speed = 0):
        self.speed = speed

    def change_to_mile(self):
        return (self.speed*0.6213," miles")
    def get_speed(self):
        return self._speed
    def set_speed(self, km):
        if km > 50:
            raise ValueError("You are liable to speeding ticket")
        self._speed = km
    #using property
    speed = property(get_speed,set_speed)
```

The last line of the preceding code makes an object of the speed property. Remember that the property must be made out of those attributes, which are likely to be changed. We added some code that creates the object of property and inside parenthesis, we passed the getter and setter method. Now, any program that uses the value of speed will invoke the `get_speed` method automatically and any program that assigns the value of speed will invoke the `set_speed` method without having to look up `dictionary(obj.__dict__)`, which is managed by class.

Now, let's use our knowledge of list comprehension and property that we learned about in this chapter to modify our snake game.

Refining the snake game with LC and property

This section will be kept as concise as possible because there is nothing new to cover. Now that we have learned about list comprehension and property in detail, we should be able to cover this topic quickly, as we discussed in the summary of the previous chapter. Just as a recap: list comprehension is a technique that is used to create a new list of elements from other iterables. A list comprehension statement consists of square brackets containing transformation that must be made for each element, along with a for loop. This is followed by some conditions. On the other hand, the `@property` or `property()` constructs are the Pythonic way of implementing getters and setters.

Let's go over some of the refinements we can make to our snake game:

1. First, we can make a function that will check the collision of the snake with the boundary or with itself. For example, if the coordinate (x,y) for the head of the snake is the same as the coordinate for its body, we have a collision. This logic can be made with list comprehension: `[body.coor == self.head.coor for body in self.body_list[:-1]]`. The following expression is going to store a Boolean that's either True or False in the result list. The `body.coor == self.head.coor` comparison is going to be made for every position representing the body of the snake. The following lines of code represent a function that returns either True or False based on the check for collision:

```
def collided(self):
    return any([body.coor == self.head.coor
                    for body in self.body_list[:-1]])
```

2. Secondly, we can decorate the preceding method with the `@property` construct. Since we've covered it in detail, this should not create any confusion for us. If there is, let me enlighten you. The main use of `@property` is to support backward compatibility. We can modify the specifications of classes and implement the constraints without actually modifying the code of the previous versions that are distributed to the clients. Similarly, we can decorate a score function with `@property` since we need to update its time value. Thus, in order to continually access the score method as an attribute, we can add the property we decorated previously, like so:

```
@property
def score(self):
    return 'Score : {0}'.format(self.score)
```

The preceding implementation of property and list comprehension is both an easy and efficient way of making code more readable and maintainable. We are going to find these types of constructs more often while programming with Python at an enterprise level.

Summary

This chapter has uncovered the advanced concepts of comprehension and generation, followed by some examples and its applications in the real-world. We saw the usage of comprehension and some of the built-in functions of Python such as map and zip, which over-shadowed the performance of for loops. Although these concepts of comprehension and mapping may seem overrated, we usually find it helpful if we have gigantic lines of code where performance matters rather than code readability. We also explored decorators in this chapter, which added some extra functionality to the existing code without affecting its original substance. Then, we learned about the concepts of the property decorator, which is a Pythonic way of implementing getters and setters while maintaining backward code compatibility.

From the next chapter onward, our main goal will likely be lean toward game programming. We have successfully learned about the essentials of Python in order to become proficient game programmers. Now, we will learn about the graphical user interface and ways of making it using modules provided by Python, such as turtle and pygame. But before we hop over to the next chapter, make sure you are playing with the code we've written so far properly. It is a very important thing for any programmer to be able to read the code by breaking it line by line. If you already have enough confidence in your skills, proceed to the next chapter, where we will look at the turtle module, which is a basic way of drawing shapes into the game screen.

8
Turtle Class - Drawing on the Screen

Not so long ago, programmers, especially game programmers, would face many intricacies while building programs. No wonder! Back then, there wasn't enough assistance from the internet portal, including no stack overflow, but more than that, there were no universal tools that programmers could use; they had to create one first and use it in the programs. The tools they created would handle some game specifics (specific drivers for sound and graphics). Programmers had to create games using assembly language due to the meager resources available, which would be trade-offs for processing power, display, sound, and control routines. Even the worst scenario would be encountered at the time of debugging. They would need complex and expensive machines in order to replicate their programs, and they would also have logging and debugging extensions. The main goal of this chapter will be to make us familiar with two-dimensional (2D) space drawing using turtle, along with the event handling method of turtle, and to create simple 2D idle animations.

At the time of writing, we have made gargantuan progress in the gaming industry. We have created tools that allow us to use any programming language in order to make games, such as Python and C (low-CPU-demanding games). All of the low-level routines are hidden by higher-level software due to the communication of device drivers. The high-level languages such as Python are abstract; they provide less access to the lower-level functions. We can group multiple things together as classes that can inherit characteristics from another class, which removes the duplication of code. Python provides modules such as turtle and Pygame, which contain a bunch of methods for designing game characters and handling user events. In this chapter, we will learn about the turtle module. Each of the things that will be built from this chapter onward will use techniques from the preceding chapters—with the addition of a few notable characteristics.

The following topics will be covered in this chapter:

- Overview of turtle
- Technical requirements
- Introduction to turtle commands
- Turtle events
- Drawing shapes with turtle

Technical requirements

This section takes you through the basic Python graphical programming module and its working. Therefore, you are expected to have the following resources:

- Python 3.5 or later; refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*
- Python IDLE
- A text editor
- A keyboard
- A mouse (a laptop's touchpad won't work)

The files for this chapter can be found here: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter08`

Check out the following video to see the code in action:

`http://bit.ly/2pAmrCs`

Understanding the turtle module

Just like the different components of a computer are equally important in order to provide a better computing experience, we also need the different components of a computer to work together in order to provide a better gaming experience. The video card of the computer is responsible for computing the visual images of the screen and then modularizing the image signal before sending it to the monitor. The input devices such as the mouse, keyboard, and joysticks are required to handle user events according to the programs. The audio card is required to process the audio signals and then send them to output devices such as the speaker. At the early age of game programming, programmers needed to read the technical manual for each of these devices separately and code each of them in isolation. This meant making communication between them would take a single year, even for simple games.

However, with advancements in technology—and drivers in particular—programmers obviated the headache of making communication between these devices and the operating system manually.

Although we developed a simple program known as drivers, which acts as a common interface to communicate with different devices, different hardware and version incompatibilities made programmers' lives harder when they were developing games that could be played across multiple platforms. Luckily, we have Python, a language that has the adept capability to make programs that can be platform independent. Turtle is the Python module that provides the drawing board that can be used to create pictures and packets. It is believed that the turtle module is the sister of another popular programming language from the 90s—*Logo*—which had an imaginary icon of a turtle, and a pen, which was used to draw over the screen. Python's standard library, *turtle*, is similar to the Logo programming language. In order to use the turtle module, we have to import it. Importing it is easier as it comes packed as a standard Python library and it does not need to be installed manually. The following steps explain how to make any turtle application:

1. Import the turtle module with the `import` command. If you ignore this step, there won't be any interface to control turtle.
2. Create a turtle to control. This step is used to instantiate turtle in order to create a new turtle controller, for example, `game = turtle.Turtle()`.
3. After creating a control, we use the new turtle to draw and carry out multiple tasks in the drawing screen by calling the methods of the turtle module.
4. We need to call one important method explicitly, which holds the game screen, that is, `turtle.done()`. This method will pause the program. You need to close the window manually in order to close the application.

In a turtle package, when we run the program that is made by calling the methods of the turtle module, a new window will appear with a pen, along with the shapes that are drawn by the turtle commands. Let's learn about a few important turtle commands.

Introduction to turtle commands

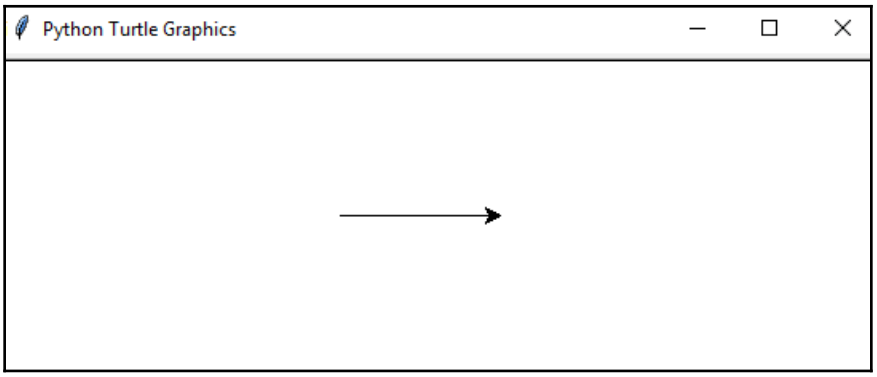
The turtle module comes with multiple commands in the form of methods that can be used independently. There are methods to make the pen move forward and backward, and some to create shapes. Take a look at the following table to find out about the most important turtle commands. You can read about them in detail on their official Python documentation pages:

Method	**Parameter**	**Description**
`Turtle()`	None	Creates and returns a new turtle object.
`forward()`	Distance	Moves the turtle forward by the specified amount.
`backward()`	Distance	Moves the turtle backward by the specified amount.
`right()`	Angle	Turns the turtle clockwise.
`left()`	Angle	Turns the turtle counter-clockwise.
`penup()`	None	Picks up the turtle's pen.
`pendown()`	None	Puts down the turtle's pen.
`up()`	None	Picks up the turtle's pen.
`down()`	None	Puts down the turtle's pen.
`color()`	Color name	Changes the color of the turtle's pen.
`fillcolor()`	Color name	Changes the color that the turtle will use to fill a polygon.
`heading()`	None	Returns the current heading.
`position()`	None	Returns the current position.
`goto()`	*x*, *y* (positions)	Move the turtle to position *x*, *y*.
`begin_fill()`	None	Remembers the starting point for a filled polygon.
`end_fill()`	None	Closes the polygon and fills it with the current fill color.
`dot()`	None	Leaves the dot in the current position.
`stamp()`	None	Leaves an impression of a turtle shape at the current location.
`shape()`	Shape name	Should be *arrow*, *classic*, *turtle*, or *circle*.

In the preceding table, we can guess the result of calling those methods by observing the literal meaning of the method's name. For example, the `forward(amount)` method is going to move the pen forward with the amount specified as the argument. All of these methods are used to plot different shapes into the drawing canvas of turtle. Observe the first `>>> Turtle()` method. This will return the object of turtle, which must be used in order to invoke these methods. As an example, we are going to make a program that will draw a line onto the screen. The following is the code for this example:

```
import turtle

pacman = turtle.Turtle()
pacman.forward(100)

turtle.done()
```

We can observe the following output by running the preceding code:



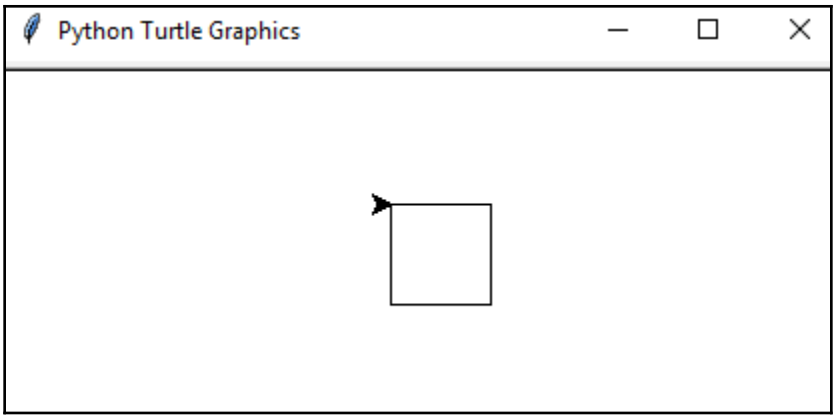
Along with the Python shell, the new screen, like the preceding one, should pop out, and this represents the turtle drawing board. Initially, the pen attached to the imaginary turtle will reside at the center of the drawing board. Any method call from the turtle object must manipulate the movement of the pen. The preceding code can be explained in the following steps:

1. First, we have to import turtle, which is a first step that will make sure all the commands that reside inside the turtle class will be available for us to use.
2. The second step is to create a turtle controller, and we refer to it as pacman.
3. Then, we make a movement of 100 pixels from the point that pacman is facing. Initially, the *pacman* turtle controller was facing toward the right; thus, the pen moved 100 pixels to the right from the center, creating the straight line.
4. Finally, `turtle.done()` is going to pause the turtle drawing board screen so that we can observe the output clearly. In order to close the turtle screen, we have to manually close the Python shell or the turtle graphics screen.

We've just learned how to create a straight line, but the lines look boring and do not add any aesthetics to the program. It's time to learn how to use another method, which is going to turn the pen in another direction. For example, we may want to change the direction of the pen from where it was originally facing to another direction:

```
import turtle

pacman = turtle.Turtle()

pacman.forward(50)
pacman.right(90)

pacman.forward(50)
pacman.right(90)

pacman.forward(50)
pacman.right(90)

pacman.forward(50)
pacman.right(90)

turtle.done()
```

We are already familiar with the `forward` method, and alongside it, we have now introduced the `right()` method. If you have a look at the previous table of methods, you will see that the `right` method and angle have been passed as arguments. Thus, this method is going to perform some rotation, accompanied by the angle that was passed along with it. Since we passed 90 degrees to it, this method is going create a 90-degree clockwise rotation. If you want to rotate the pen anticlockwise, we have to call the left method and specify the angle of rotation. In the preceding program, we rotated it by 90 degrees. The geometrical shape that has all angles of 90 degrees is either the square or the rectangle. However, we know that the `forward` method will result in a straight line, which is the same as the sides of geometrical shapes. The sides that are created by the `forward` method will be equal in length, which is 50, and this is passed as an argument with the `forward` method. With all this evidence, we can surely expect the square shape to be drawn in the turtle board. Let's run the preceding code to observe the output. As expected, the square shape is drawn:



Have a closer look at the preceding code; did you see some repetition of code? Obviously, the invocation of the `forward` and `left` methods is done multiple times, which ultimately disrespects the DRY principles. This epiphany does not come without practicing the paradigm of Python. Thus, we can say that practice is what differentiates good and bad programmers. Now, recall what we need in order to eliminate the redundancy of the code; we should use either loops or functions. We will use a loop here:

```
import turtle

pacman = turtle.Turtle()
for i in range(4):
    pacman.forward(50)
    pacman.right(90)

turtle.done()
```

I guess we won't have any problems in reading and understanding this code. As we mentioned in `Chapter 3`, *Flow Control - Building a Decision Maker For Your Game*, we can create an iteration level with a range of functions. Since we need to run these methods four times, we have created four iterations using the range function. Anything that needs to be repeated is indented by four blocks inside the scope of the for loop.

One thing to notice in this example is that we have multiple methods for handling the movements of the pen in the drawing screen. The two turtle commands that we have learned so far are `forward(amount)`, which moves the turtle forward in the direction it is facing with some amount, and `right(degree)`, which makes the turtle turn clockwise by a specified degree. Notice here that the `right` and `left` commands are not going to write anything on the screen; instead, they are used for rotation only.

Following the patterns of everything that we have learned so far, we can predict that the `backward` method is going to move the pen in the backward direction from the original direction that it was facing by a specified amount. I recommend that you try modifying the preceding code a little bit—by refactoring the forward method using `backward`, and by refactoring `right` using `left`—and observe the result accordingly. I would like to take the time to conclude this topic here, without covering other functions, because we will go through each of them while making games in the upcoming chapters. We will make multiple games, such as the Snake game, the Pong game, and Flappy Bird using the turtle module. Now, we will explore the ways we can connect input devices, such as a mouse and a keyboard, to our game so that players can interact with the turtle environment.

Exploring turtle events

As we mentioned in the previous chapters, handling the events of a user is one of the prime building blocks for creating any game. The event represents the action that needs to be performed at any time during the game. Have you ever wondered how the events are handled by programs at the low levels? When a user executes any event using the keyboard or mouse, that request is stored in a queue-like structure. The queue structure is important because the order of handling these events must be on a first come, first served basis. Then, according to the behavior of the user actions, events are handled by the program. These two tasks of rendering and action handling are performed independently by the programs. For example, in a counter strike game, the user can shoot from their gun, even when enemies are not around them. Here, the event is the user pressing a key to fire the gun and the rendering task is spawning the enemies around the player. These two tasks are not executed independently unless we write programs to make them. In this section, we are going to learn how to take a user action as an input, and handle it accordingly. Handling the user actions means serving the actions that are stored in the queue structure.

Most of the events are based on the use of a mouse or a keyboard, but some events must be predicted automatically by the program and handled accordingly, such as the `ontimer(fun, time)` method. This method takes two arguments: function and time in milliseconds. This method sets a timer that calls the `fun` function after `time` in milliseconds. Let's make a simple program to understand this:

```
import turtle
star = turtle.Turtle()

exit = False
def main():
    if not exit:
        for i in range(100):
```

```
 star.forward(i)
 star.right(144)
main()
turtle.mainloop()
```

The last line of code (`turtle.mainloop()`) simply performs the same operations that are carried out while looping. Until, and unless, the user exits the window screen explicitly, the call to the `main` function will not be terminated. Its importance can be observed when the program has a while loop, which is used to listen to the incoming connection, but we don't want the computer to be constantly focused on the one case:

```
def draw_objects():
    #statements
    draw_objects() #may be you want to call it within the time interval
                    of 100ms

draw_objects()
turtle.mainloop()
```

The previous code works in exactly the same way as a while loop, but now the Python parser is not dedicated to performing only one task constantly. Instead, for every 100 milliseconds, `draw_objects()` tasks will be performed, and for the remaining 99.99 milliseconds, the Python parser is free to carry out any other tasks.

Interestingly, the preceding code represents the proper outcome of any turtle program. Although calling a different function would make a different character on the screen, the main aim of using turtle is to render the game character onto the screen. Let's break down the preceding code into the following points:

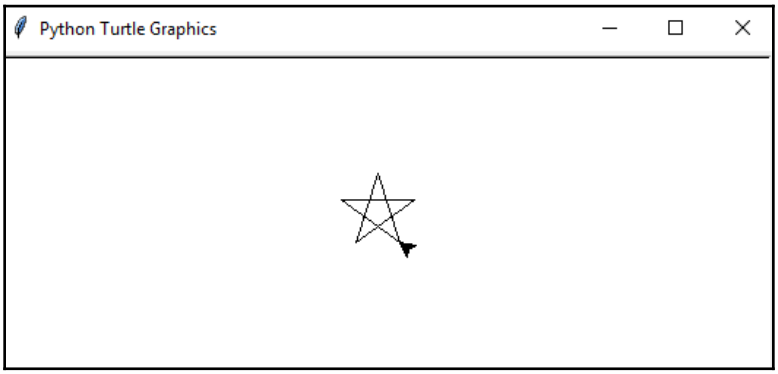
- The first couple of steps represent importing turtle and creating a turtle controller, which will allow us to call all the `turtle` methods through it.
- We have created a `main` function, and inside it, we have some code to create a star pattern. The iteration is 100 times, which means we will have 100 stars printed onto the output screen, but remember, they will be closely spaced.

The best way to render the characters properly in the screen is by using the `ontimer` method. Let's modify the same program with the `ontimer` method. Let's see how we can use it in the program:

```
import turtle
star = turtle.Turtle()

exit = False
def main():
    if not exit:
```

```
            star.forward(50)
            star.right(144)
        turtle.ontimer(main,500)
main()
```

Unlike before, the preceding program is not going to print multiple stars; instead, it prints a single one. However, the `ontimer` method removes the overhead of calling the for loop since it sets the timer to call the same function again and again. In this program, we passed the `main` function and 500 as arguments, which means that the `main` function should be called in every `500` milliseconds. Running the preceding program will yield the following output:



It's time to learn how to handle keyboard and mouse events. As always, there are methods that have been defined to handle keyboard events and methods that have been defined to handle mouse events. But, before handling user events, turtle must launch a listener, which continuously remains awake to listen to any events. Such a listener controller is created using the `listen` method, that is, `>>> turtle.listen()`. The following table depicts the methods that are used to handle keyboard events:

Method Name	**Parameters**	**Description**
`turtle.onkeypress(function, key = None)`	**Function**: A function with no arguments or `None`. **Key**: A key in the form of strings or symbols, for example, `q` or `space`.	It is used to bind the function to any key events that are pressed on a keyboard. If no key is specified, any key will work.

`turtle.onkeyrelease(function, key)`	**Function**: A function with no arguments or `None`. **Key**: A key in the form of string, `a`, or symbols, `enter`.	It is used to bind the function to key-release events that are performed by key actions. If the function is `None`, the binding of events is removed.

Let's make a simple program in order to grasp the idea of using these methods of handling keyboard actions:

```
import turtle
star = turtle.Turtle()
def main():
    for i in range(30):
        star.forward(100)
 star.right(144)
turtle.onkeypress(main,"space")
turtle.listen()
turtle.mainloop()
```

Let's run the program and observe the output. After pressing *F5*, you will observe two screens, one of which will have the turtle graphics board and pen at the center of it. Now, press the *Spacebar* key on the keyboard. As soon as you press it, it starts to draw a star onto the screen.

Inside the `main` function, we have added some code that will make a star. However, as you can see, the `main` function has not been called explicitly, as we normally do while calling functions; instead, it is called using the `onkeypress` method. This method binds the key to the function, and whenever the key is pressed, the function is called automatically. If you remove the last line from the preceding code, the listener controller is not going to work. The `listen` method is used to make a controller for listening incessantly to these types of actions.

In a similar fashion, we can call the `onkeyrelease` method. Replace `onkeypress` with `onkeyrelease` in the preceding code and observe the output. The output is going to be the same. The `onkeyrelease` method is used to bind the function to be called with the key-release event of the key.

Similarly, the ways of handling mouse events are not too different—they are also handled by method calls. The following table depicts the methods that can be used to handle mouse events:

Method	Parameter	Description
`onclick(function, button = 1, add = None)`	**Function**: A function is called with two arguments (*x*, *y*), which represent the coordinates of the clicked position by mouse or pointer. **Button**: It represents the mouse button, `default = 1`, which means the left mouse button. **Add**: It is used to add multiple bindings. If `True` is passed, a new binding will be added, otherwise it will stick to the current one.	Binds functions to mouse-click events. If the user clicks on any position of the turtle canvas, the coordinates of the clicked position will be used to call the function.
`onrelease(function, button = 1, add = None)`	**Function**: A function is called with two arguments (*x*, *y*), which represent the coordinates of the clicked position on the drawing board of turtle. **Button**: `Default = 1` means that the left mouse button is used. It is used to add a number for mouse-button. **Add**: According to its value of `True` or `False`, it decides whether to add a new binding or not.	Binds functions to mouse-button release event.
`ondrag(function, button = 1, add = None)`	**Function**: A function with two arguments, which represent the coordinates of the clicked point into the game screen. **Button**: Adds a number to indicate the mouse button listener.	Binds functions to mouse move events on the current turtle controller. If the function is `None`, the current binding will be removed.

Let's make a simple program to grasp the idea of using the preceding methods for handling mouse events:

```
import turtle
pacman = turtle.Turtle()
def move(x,y):
    pacman.forward(180)
    print(x,y)

turtle.onclick(move) #calling move method
#turtle.onclick(None) #to remove event-binding
```

You can see that the `onclick` method was called with only the move function, which in turn calls the move method with the *x* and *y* coordinates representing the clicked point onto the canvas. Running the preceding program does not draw any lines on the screen until you click on the drawing canvas. When you click on any point of the screen, you will see its coordinate printed in the Python shell, and a straight line will appear on the canvas. We will cover the remaining `turtle` methods in the upcoming chapter, along with how to make some mini games. Before that, we will try to make some shapes using the `turtle` module and the Python design patterns that we have learned about so far.

Drawing shapes with turtle

The process of making shapes may seem like a boring and tedious task for a human being, but it's not for computers. Imagine making a hexagonal shape with exact geometrical measurements while taking care of angles and sides. The process itself overwhelms most of us. On the other hand, computers are considered to work sedulously; we can throw as many tasks as we like at it, and it will perform them gracefully.

As we have mentioned previously, two critical pieces of information while drawing any shape are the angle and length of each side. We can make variables to store them so that we can refer to them in the program whenever they are needed. For any shape, there will be a different number of sides. For example, a triangle has three sides, while a hexagon has six sides. We need to specify the number of sides explicitly in the program. In this section, we are going to make two shapes, a hexagon and a star shape, with some added colors. The main aim of this section is to help you understand how the programming paradigm is used, along with a particular module, in order to make appealing games.

The following list of steps depicts the roadmap that is needed in order to create two shapes, one by one. The first shape that we will create is a hexagon: a shape that has six sides, with a custom length. After that, we will make a star pattern again, but this time, we will add color properties to it:

- **Hexagon**: We will create this shape by defining specific variables, such as the number of sides, interior angle, and length of sides. After that, we will use the for loop to create six iterations because we have to call the line rendering method six times (since a hexagon has six sides). We will use the `forward` method to draw a straight line and the `right` method to turn the turtle clockwise by a specific angle:

```
import turtle
hexagon = turtle.Turtle()
num_of_sides = 6
length_of_sides = 70
angle = 360.0 / num_of_sides
for i in range(num_of_sides):
    hexagon.forward(length_of_sides)
    hexagon.right(angle)
turtle.done()
```

- You can see how convenient it is to draw the shapes onto the canvas using the `turtle` module. We are already familiar with these methods and the usage of loops in order to remove the repetition of multiple lines of code; thus, it won't be hard to grasp the code that we have written over here.
- **Star**: Making a star shape with Turtle is easier than using any other module. We have already made it using two methods of `turtle`, that is `forward` and `left`. But in this section, we are going to color the star shape using the `color` method provided by the `turtle` module. We will start by defining the color palette, that is, different color names, and we will make a method call of `begin_fill` and `begin_end`, which will add the color to the shapes. The following table shows three methods that can be used for coloring shapes in turtle:

Method	Parameter	Description
`color(*args)`	Args represents the color's name. The current color is used for drawing lines using the `forward` or `backward` methods. The color name can be given as `single value: color("blue"), double value: color("black","green")`, or `rgb` float values.	Used to change the color of the turtle pen.

`begin_fill()`	None	This method will remember the starting point for filled polygons.
`end_fill()`	None	It will close the shape drawn in the turtle canvas and fill it with the current fill color.

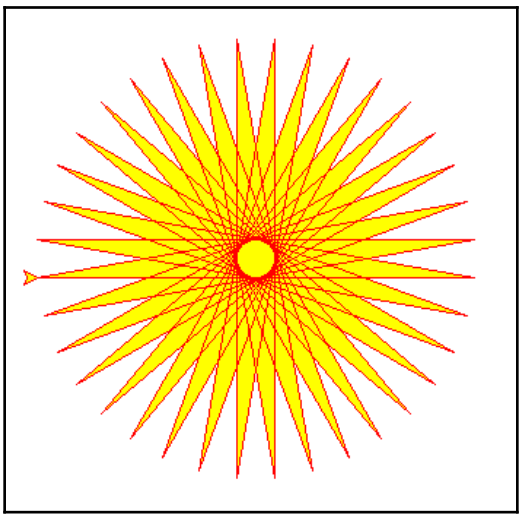
As an example, we will write a program that will use these methods to color the star pattern. We will use the color combination of red and yellow to make the star more attractive. We have been using the `import turtle` command to make `turtle` methods available for the program to use. Instead of doing it this way, we can import everything from turtle with the `from turtle import *` command. Now, instead of calling the `turtle` method with `>>> turtle.forward(100)`, we can call it directly, that is, `forward(100)`. Let's write a program to create such a star pattern:

```
from turtle import *

color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

I just love the way that turtle works with Python. Being able to bind every function to the programming paradigm of Python makes the `turtle` module effective to use. In the preceding code, we might not have any confusion with the first line of code, which simply imports everything from the `turtle` module—every attribute and member. We made a color palette of red and yellow using the `color` method. Inside the main loop, we will encounter two methods, which we have been using from the beginning of this chapter. In addition, we have added a conditional to indicate the stopping point for the turtle pen. The `abs()` method is used to return the absolute value of the number, that is, `>>> abs(-4)`, which yields 4. Inside the `abs()` function, we called the `pos()` method of the `turtle` module, which will return the position of turtle as a two-element list. We checked for the current position, and if it's less than 1, for example, 0, then it must represent the center position, because (0,0) represents the center position. If we encounter the center position after an iteration, that means we can terminate the program because at this point, we must have already drawn a star. If we proceed further, the turtle pen will draw another star on the same position, therefore overriding the old one.

Thus, in order to prevent this continuous iteration, we have added a conditionals line: `if abs(pos()) < 1`.

Executing the preceding programs yields the following output. One thing you must remember here is that, from the color palette, at the beginning, we used a red pen to draw the star, and after finishing it, we used yellow to fill the inner part of the star shape:



Now that you know everything about the ways of using `turtle` methods for creating shapes and coloring them, we will wrap this chapter up here. We will be using the concepts we've learned in this chapter, such as creating patterns and handling user events, in the upcoming chapters by making simple mini games such as Snake, Pong, and Flappy Bird.

Summary

The Python *turtle* module is a powerful platform for building 2D mini games. It contains a variety of methods in order to facilitate the design process of game characters. We have written a bunch of programs in this chapter, and also handled user events. We started this chapter by introducing the key features of the `turtle` module, and built a universal prototype for any game that can be made with the Python `turtle` module. This chapter taught us about animating a 2D canvas using the `turtle` module. Along with animating game characters, we learned how to create interfaces that communicate between the game interface and the user controller by handling user events.

Following the completion of this chapter, you will be well equipped to create simple 2D games using the `turtle` module. You will also be able to handle the user actions that are provided by the mouse and the keyboard, which allows us to make user-interactive games. Now that you have learned how to create simple animations using the 2D Turtle canvas, you can create any geometrical shape; try a few more before hopping into the next chapter.

We didn't cover any games in this chapter, because in order to create games with the `turtle` module, we need to explore vectors first—creating vectors, storing vectors, finding the magnitude of vectors, vector additions, negations, diagonal movements, and many more. We will cover all of these concepts in the next chapter.

The topic of vectors is undoubtedly the most essential topic for any game developer's toolkit. Vectors are mathematical terms that represent the magnitude and direction of our game character that appears on the screen. Magnitude represents the modulus of the current coordinates of a point in which the character resides, while direction represents a course that the game character moves on. Now would be the perfect time for you to play around with the `turtle` module and grasp the idea of handling user events and building appealing shapes and characters.

9
Data Model Implementation

Games are a medium that try to emulate, or at least simulate, real-world environments through the use of interplay, where players use motions and movements in order to control the game characters. As we know, there are a variety of ways in which players can interact with the game, mostly with input devices such as a keyboard, a mouse, or a joystick. In order to translate these input signals into meaningful information, we need to address the signals with corresponding actions. In most games, we use keyboard keys to make movements for the game character, but internally, the signals are handled by mathematical objects called vectors. This is extremely important for any game, regardless of how the graphics appear, as it causes players to create actions and address them with appropriate reactions.

In this chapter, we will be introduced to 2D vectors—ways of manipulating the positions of game characters. The change in the coordinates of vectors (x, y) represents the movement that's specified by the game player. This chapter will be life-changing for any programming beginner as this will teach us how to use mathematical concepts such as addition, subtraction, multiplication, rotation, and reflection with a programming paradigm, which we know as data model implementation. The end goal of this chapter is to make you familiar with the concept of operator overloading using Python, the usage of Python built-in methods in order to manipulate vectored positions, and the implementation of data models or magic functions.

The following topics will be covered in this chapter:

- Overview of operator overloading
- Technical requirements
- Dealing with 2D vectors
- Data model for vectored motion

Technical requirements

This chapter will give us a roller coaster ride of Python's simple, yet powerful, concept of operator overloading. Therefore, you are expected to be equipped with the following tools:

- Python 3.5 or newer
- Python IDLE (Python's inbuilt IDE)
- A text editor
- A web browser

The files for this chapter can be found here: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter09`

Check out the following video to see the code in action:

`http://bit.ly/2psS6pd`

Understanding operator overloading

This is a new concept, and may be ambiguous to naive programmers, but it is obligatory to have this knowledge. In the programming nomenclature, everything that is defined with a programming language has a specific usage. For example, we cannot use the `sum()` method to find the difference between elements. We can extend the meaning of any operation beyond its normal usage or predefined operational usage. Take a simple example of an addition (+) operator; this operator can be used to add simple integers, concatenate two independent strings, and even merge the two lists. This is possible because the addition operator is overloaded in different classes, that is, it has different implementations defined in the string and integer classes. This is the power of operator overloading.

Another factor that must be kept in mind is that the same function or built-in operator depicts different behaviors for the objects of several classes, as shown in the following example:

```
>>> 6 + 6
12
>>> "Python" + " is " + "best"
'Python is best'
>>> [1,2,3] + [4,5]
[1,2,3,4,5]
```

Several methods support operator overloading; these are known as data models, or sometimes, magic methods. They are called so because these special methods extend the functionality of methods, which in turn adds magic to our classes. These data models should not be invoked by us; rather it happens internally from the classes. For example, when we perform an addition operation with the `+` operator, the Python parser internally invokes the `__add__()` method. Different built-in classes of Python such as `str`, `int`, `list`, and many more, have different internally defined magic functions. We can print the list of magic functions that are dedicated to a particular class using the `dir` function. For example, the following list indicates several methods and attributes that are defined in the `str` class:

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

As shown in the preceding list of methods and attributes of the `str` class, we can observe several methods that start and end with a double underscore. For example, the `__add__()` method is used to concatenate two strings using the `+` operator:

```
>>> first_name = "Ross"
>>> first_name.__add__(" Geller")
'Ross Geller'
```

In the preceding example, we can see that the `__add__()` function works in a similar way as `+`. These data models are meant to be used to extend the predefined meaning with overloaded behavior. Based on mathematical canonization, we normally use operators such as `+`, `-`, `/`, and `*` with numeric objects. However, with the overloading technique, we use the same operator for multiple objects, such as strings and lists. We can conclude that the addition operator (`+`) is overloaded. Similarly, different data models are defined by Python for different operators, that is, `__sub__()` for the `-` operator, `__mul__()` for the `*` operator, and `__div__()` for the `/` operator.

Now that we have learned how to use data models in the most basic form of an add function, we will implement some examples of custom-made classes.

Using data models in custom classes

Now that we know how to implement the `__add__()` magic function with various data types such as integer and string, let's observe how it can be used in custom-made (user-defined) Python classes. We will consider the following example to illustrate the usage of data models in our custom class:

```
class Base:
    def __init__(self, first):
        self.first = first

    def add(self, other):
        print(self.first + other)
```

We will make the objects of the preceding class with the following code. This code is executed in the Python shell:

```
>>> obj1 = Base(1)
>>> obj2 = Base(2)
>>> obj1.add(obj2)
TypeError: unsupported operand type(s) for +: 'int' and 'Base'
```

As expected, we get an error saying **unsupported operands for different types**, which implies that the + operator is not functional for adding the objects of custom classes. As mentioned previously, to solve such problems, we can use operator overloading. We can explicitly define such special methods inside our class in order to make objects compatible with built-in methods and operators. For example, in the case of the addition operation, we have to define the `__add__()` method explicitly inside the class, which looks something like this:

```
class Base:
    def __init__(self, first):
        self.first = first

    def __add__(self, other): #operator '+' is overloaded
 print(self.first + other.first)
```

Let's check whether this works by making different objects of the `Base` class:

```
>>> obj1 = Base(1)
>>> obj2 = Base(2)
>>> obj1.__add__(obj2)
```

```
3
#for strings as add method is defined internally inside str class
>>> obj3 = Base("Hello ")
>>> obj4 = Base("World")
>>> obj3.__add__(obj4)
'Hello World'
```

Thus, the magic function, or the `__add__()` data model, is overridden, which successfully performs the addition operation between two integers and two strings. We can also check this for other data objects such as lists and tuples. Now, we can clearly predict the pattern; if we want to overload any mathematical operator and implement it differently in our custom-made classes, we have to define data models in our classes. I hope you get the idea! Now, we can predict the `__mul__()` pattern so that we can perform multiplication between different objects, `__sub__()` to perform subtraction, and so on.

Let's observe another powerful, yet less frequently used, magic method of Python, before actually learning about the importance of using these magic functions. Let's talk about the `__new__()` data model. You can easily observe the working of these methods; just remove the underscore and parentheses that surround the method name and you will come up with the `new` keyword. If you have a programming background from any high-level language such as Java and C#, you will already understand my point. For those who are new to the concept of the `new` keyword, this operator is used to create instances of the classes. For example, in Python, we have `object = class_name()`, while in Java, we have `object = new class_name()`.

Thus, the `__new__()` magic method is the first method to be called while creating objects of classes—even before the `__init__()` constructor is called—and it is called implicitly. The `__new__()` method is responsible for creating new objects, and it returns the object that's initialized using the constructor's `__init__()` method. Do you remember, in object-oriented chapter, we referred to the `__init__()` method as a special method, which is, in fact, a magic method. Let's consider the following example to learn about the `__new__()` magic method:

```
class Base:
    def __new__(cls):
        print("This is __new__() magic method")
        obj = object.__new__(cls)
        return obj
    def __init__(self):
        print("This is __init__() magic method")
        self.info = "I love Python"
```

The following code is executed inside the Python shell. We are creating an object of the `Base` class, and observing that the `new` method is called before the `init` method:

```
>>> obj = Base()
This is __new__() magic method
This is __init__() magic method
```

Note that in the preceding code, we passed `cls` as an argument while defining the `new` magic method and the `self` variable as an argument while defining the `init` constructor. The distinction between these two variables—`cls` and `self`—is defined in PEP 8, which defines the style guide for Python code. This coding style is not mandatory, but according to PEP 8, we should always do the following:

- Always use `self` for the first argument to instance methods.
- Always use `cls` for the first argument to class methods.

I think that we are now capable enough to predict the working internals of any built-in function. Let's take the example of the `len()` method. If there is any built-in `fun()` function in Python, it corresponds to `__fun__()`. The Python parser makes an internal call as `object.__fun__()`, where the object is the instance of a class. Considering this analogy, for the `len()` function, the Python parser interprets its call as `object.__len__()`, and it returns the length of the object. We have seen how it works internally; however, since the main topic we want to cover is how to override it, let's define this magic method inside our custom-made classes (in a similar way to the preceding example, where we used the `add` magic function to add objects of a class). In the case of `__len__()`, consider the following example:

```
>>> info = "I love Python"
>>> len(info)
13
>>> info.__len__()
13
```

Therefore, when we define such magic methods or data models in our own class, we override the behavior of the functions that are originally defined by Python; thus, we are now no longer calling the original method. When you override the original method with your new one, we refer to this as method overriding. Up to this point, we have been learning about data models and ways of using them in our own classes. Now, let's learn about why they are essential in game programming. We will do this by exploring vectors in the next section.

Dealing with two-dimensional vectors

Before actually exploring vectors, let's start with the basic overview of motion and how characters are moved in a straight line. To move any object or image, we have to make a slight change to the frames by a fixed amount. The movement must be fixed for each frame in order to make it symmetrical. To make an object move in a horizontal direction, we carry out an addition of a fixed amount to the *x* position, and to make it move in a vertical direction, we add the same amount to the y position. Thus, motion in 2D games can be represented as (*x*, *y*). Let's consider the following example to illustrate the usage of these coordinates on order to draw any shape into the game environment:

```
def line(a, b, x, y):
    "Draw line from `(a, b)` to `(x, y)`."
    import turtle
    turtle.up()
    turtle.goto(a, b)
    turtle.down()
    turtle.goto(x, y)
```

We are using the `turtle` module, which we used in the previous chapter to draw a line using the `(a, b)` and `(x, y)` positions. The `goto()` method is used to move the pen to the passed positions. These coordinates—`(x, y)` or `(a, b)`—clearly show the importance of knowing the positions in order to create game characters (we use line as a metaphor for any game character).

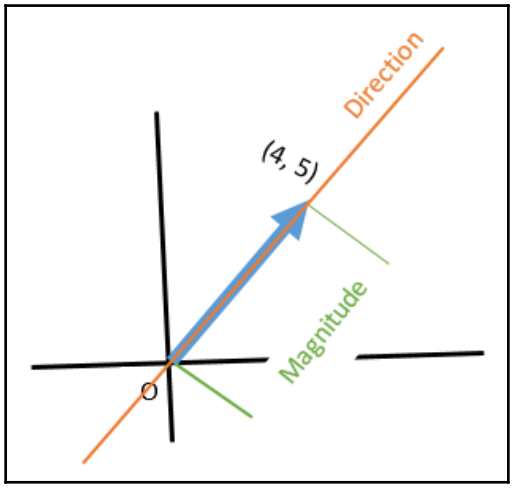
We can deem that the usage of a straight line motion is pretty useful, but looking at it from a different perspective, a game that only supports vertical or horizontal motions may seem dull and unexciting. For example, in the Pacman game, where a player would move either in a vertical or horizontal direction, this may be appropriate, but in the case of a car-racing game, where users can move in any direction, this motion doesn't work properly. We must be able to move in any direction by adjusting the positions of *x* and *y* for each frame. We will use the same two positions, *x* and *y*, to generate both straight and diagonal motions: a rate that indicates speed for the *x* and *y* positions. The form that represents `(x, y)` is known as a vector, but more importantly, vectors signify direction, unlike scalar. We will explore vectors in more detail in the following subsection.

Exploring vectors

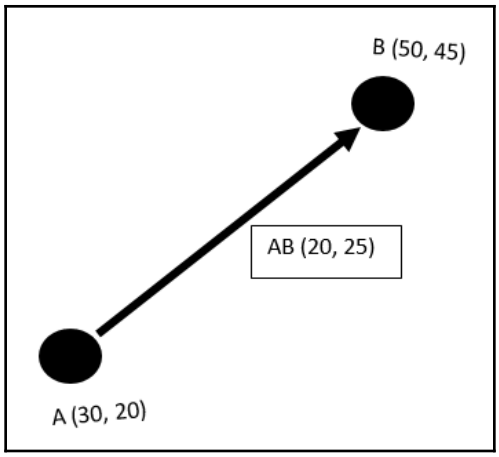
As the mathematical adage says:

> *" Vector refers to any quantity that has magnitude as well as direction, especially for determining the position of one point in space relative to another."*

We couldn't agree more. This concept is taken from mathematics, and is the most well-known topic for any game programmer, naive to suave. Vectors are the proper representation of any position of an object, with the critical information of the direction attached to it. Vectors have similar representations as a straight line motion in the form of x and y coordinates (2D), but they are not restricted to only providing information about magnitude; they have a specific purpose. For example, vector (4, 5) represents the next position, where 4 is added to the x coordinate of the current position and 5 is added to the y coordinate of the current position; something like this—(0 + 4, 0 + 5)—where (0, 0) is the origin or center position. Let's examine vectors figuratively with the following examples:



In the preceding diagram, vector (4,5) has magnitude and direction. The green line indicates magnitude and the orange line indicates direction. Thus, a vector is incomplete without the information of its previous direction. Let's look at another simple example to clarify this further:



The preceding diagram says it all. The vector AB is the subtraction of the *x* and *y* positions from the target with the initial position. Suppose a Pacman is at position (30, 20), and he has to reach the target, that is, (50, 45). Vector AB is the critical information which indicates that Pacman has to move 20 units more in the *x* direction, and 25 more in the *y* direction.

It is well-known that Python does not have a built-in *vector* data structure. If you think there is, perform a quick internet search on it; you will get the basic idea. However, we didn't cover vectors as built-in data structures in the preceding chapters. Although we don't have vectors as built-in data types, we can make one for ourselves. As we know, vectors constitute two different positions (*x*, *y*), and our main aim is to use other built-in data structures to make them. For example, we can use lists to make vectors, but indicating each point with indexes such as `[0]` and `[1]` adds unwanted overheads. The same goes for using tuples. Probably the best way of creating vectors would be by making our own vector class. In doing so, we can reference points as *x* and *y* instead of indexes. Furthermore, the best exploits can be made by using data models with vectors. We can use `__add__()`, `__mul__()`, and many more magic functions inside the vector class, which will introduce motion to the game characters. As an example, we will create a simple vector class and make use of the `__str__()` method, along with a constructor, which will provide a proper representation of positions with vectors:

```
class Vector(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "(%s, %s)"%(self.x, self.y)
```

In the preceding program, we created a `Vector` class and defined two members in it: one being the constructor, and the other being the magic method. Now, when we create any object of this class, such as `> pos = Vector(10,40)`, the `init()` method will perform initialization so that we can reference each component of the vector as `>>> pos.x` and `>>> pos.y`. The `__str__()` method is the magic method that is used as the overriding method and it has a custom definition in our `Vector` class, which is used as a representation of the components of vector in the form of the *x* and *y* positions. Let's see how it works by running the following code and creating a `Vector` class object:

```
>>> pos = Vector(10, 40)
>>> pos.__str__()
'(10, 40)'
```

Apart from the `__str__()` method, we have a bunch of magic functions that are applicable for manipulating vectors. We can use `__add__()` to perform addition between vectors, `__sub__()` to perform subtraction, `__neg__()` to perform negation, and so on. We will learn about these data models and ways of using them to modify vectors in the next section.

Modeling for vectored motion

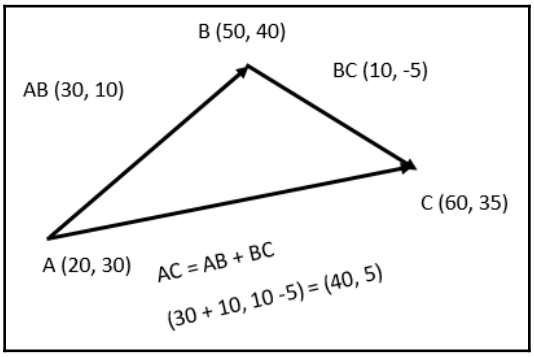
As we know, vectors are the quantity that constitute both magnitude and direction. These two pieces of information can be extremely critical when determining the next position for game characters, based on a user's action. For example, a game character, Steve (a Minecraft character), can use vectors to determine the units he has to travel further using magnitude (AB) and direction (→AB) in order to track his goal. Although we can change both of these sources of information one by one, we are primarily concerned with magnitude because magnitude is responsible for providing motion in 2D games. In this section, we will uncover the techniques that will teach us how to add and subtract vectors, and even perform multiplication and division. These types of operations will be added as logic in the game, along with user events, so that whenever the user presses any keys on the keyboard, it is addressed by a particular event. The techniques that can be used while performing this operation mathematically are as follows:

- Performing operations (subtracting/adding) on vectors with known components
- Performing operations by finding components, or simply using head/tail methods

Let's learn about the use of these techniques by hopping over to the next section, where we will perform vectored operations using magic functions or data models.

Vector addition

Similar to numeric addition, we can overload the + operator using the `__add__()` data model, which will add two different vectors and combine its effect in order to produce a new single vector. Using this method, we can make diagonal motions with the game characters. We need two vectors to perform addition; the first one will be the current position of the game character, and next one will be a predefined fixed amount for each component of the vector that needs to be added when the user presses any key on the keyboard. The following diagram illustrates the addition of vectors in detail:



Never perform addition operation of vectors with the + operator when you have a vector that's represented by tuples or lists. [1,2] + [3,4] does not add individual digits like this: [4, 6]. Instead, it will concatenate two lists into one, like so: [1,2,3,4].

The following code uses the `__iadd__()` magic function to add two vectors. The `iadd` and `add` methods work in a similar way, but the main difference between them is that `__iadd__()` stores its result into the memory location, unlike `__add__()`. You can use either of these to write the following code:

```
def __iadd__(self, other):
    if isinstance(other, Vector):
        self.x += other.x
        self.y += other.y
    else:
        self.x += other
        self.y += other
    return "(%s, %s)"%(self.x, self.y)
```

Make sure that the preceding code is included inside the previously made `Vector` class. The `__iadd__()` method takes the argument, *other*, which represents a second vector that needs to be added to the vector that it is called upon. Inside the magic function, we have made conditionals to check whether the passed *other* vector is a type of `Vector` class. If it is, we are adding matching components of the first vector with the second vector, which is `first.x` to `second.x`, and `first.y` to `second.y`, where first and second are vectors. Let's make the instances of the `Vector` class and check the output of the vector addition:

```
>>> a1 = Vector(10,20)
>>> a2 = Vector(30,40)
>>> a1.__iadd__(a2)
'(40, 60)'
```

Now that we have successfully used the magic method to implement vector addition, it's time to learn a few more of them in order to implement vector subtraction and vector negation.

Vector subtraction

Just like the addition of vectors implies the forward motion for game characters, vector subtraction suggests the opposite direction from where it is currently facing. We can use either `__sub__()` or `__isub__()` to implement vector subtraction. We normally prefer `isub` because it stores a result before returning it and it can be perfectly used in order to clone the vector objects so that we can perform different manipulation in the duplicate objects, without harming original one. Vector subtraction is quite similar to addition; instead of adding each components of a vector, we are simply going to subtract them. This motion is useful in games such as Pacman, where users have to reverse their direction spontaneously without disturbing the gameplay. Let's write the following code inside the `Vector` class in order to perform vector subtraction:

```
def __isub__(self, other):
    if isinstance(other, Vector):
            self.x -= other.x
            self.y -= other.y
        else:
            self.x -= other
            self.y -= other
        return "(%s, %s)"%(self.x, self.y)
```

Let's run the preceding code in the Python shell in order to observe the result of vector subtraction:

```
>>> a1 = Vector(10,20)
>>> a2 = Vector(30,40)
>>> a1.__isub__(a2)
'(-20, -20)'
```

Vector multiplication and division

Operations such as multiplication and division will make vectors larger and smaller, respectively. The change of motion due to multiplication can be linear when a vector is multiplied by any scalar number. For example, when we multiply any vector by two, its magnitude will be twice than before, but the direction will remain unchanged. Similarly, when we multiply the same vector with a negative number, let's say, -2, its direction will be opposite to the direction it was originally facing. Multiplication operations are normally used for scaling vectors. We can multiply and divide two vectors as follows:

```
def __imul__(self, other):
        if isinstance(other, Vector):
            self.x *= other.x
            self.y *= other.y
        else:
            self.x *= other
            self.y *= other
        return "(%s, %s)"%(self.x, self.y)

def __itruediv__(self, other):
        if isinstance(other, Vector):
            self.x /= other.x
            self.y /= other.y
        else:
            self.x /= other
            self.y /= other
        return "(%s, %s)"%(self.x, self.y)
```

Similar to vector multiplication and division, we can perform a scaling process using scalar quantity. We will pass a number, instead of the second vector, as a parameter to the magic methods. It can be done as follows:

```
def __mul__(self, scalar):
    return (self.x * scalar, self.y * scalar)
def __div__(self, scalar):
    return (self.x / scalar, self.y / scalar)
```

Vector negation and equality

Since we have covered the most important operations of vectors, such as addition, multiplication, and subtraction, we will now learn the easy, yet important, vector manipulation technique, which is known as vector negation and equality. Vector negation is important when a player wants to reach out to the preceding state from the current one (since AB = -BA), which implies that negating any vector creates another vector of the same magnitude but in the opposite direction. In order to negate a vector, we can simply add the − negative operator to each component of the vector. As an example, we can consider the following lines of code:

```
def __neg__(self):
    return (-self.x, -self.y)
```

We can check whether two vectors are equal by checking each of the components of the vector. For example, `first.x` should be compared with `second.x`, and `first.y` should be compared with `second.y`. For example, the following method will return `True` if two vectors are equal:

```
def __eq__(self, other):
        """v.__eq__(w) -> v == w
 >>> v = vector(1, 2)
 >>> w = vector(1, 2)
 >>> v == w
 True
 """

 if isinstance(other, vector):
            return self.x == other.x and self.y == other.y
        return NotImplemented
```

According to the Python official documentation:

("`NotImplemented` signals to the runtime that it should ask someone else to satisfy the operation. In the expression `a == b`*, if* `a.__eq__(b)` *returns* `NotImplemented`*, then Python tries* `b.__eq__(a)`*. If b knows enough to return* `True` *or* `False`*, then the expression can succeed. If it doesn't, then the runtime will fall back to the built-in behavior (which is based on the identity of == and !=)").*

Summary

We have covered a wide range of topics in this chapter, starting from data models to the creation and manipulation of vectors. Vectors are undoubtedly the most essential topic for any game developer; they help to create motion for the game characters and sprites so that the game will be more user interactive. We have learned about different operations, such as addition, subtraction, division, negation, and many more. We also manipulated our vector components using these operations and magic methods. Magic methods are a part of method overriding, which should have been covered in `Chapter 6`, *Object-Oriented Programming*. However, I reserved it until this chapter because it makes more sense to learn about it while exploring vectors.

As the mathematical logic concerning vectors is a primary building block for the expedition of character movements in the game, you have learned how to implement operator overloading using magic functions. The vector manipulation skills we've learned in this chapter are important because they specify the position of an object and help us perform manipulation with some algebraic operations.

This chapter has introduced us to two-dimensional vectors—a mathematical concept that makes the motion of game characters possible in a game. To implement this, we had to use the concepts of data overloading using magic functions. To overload any operator— that is, change the implementation of any operator such as `+` or `-`—we extend the usage of such operators from primitive data types to complex data structures. The main goal of this chapter was to introduce you to the ways you can accomplish mathematical concepts such as 2D vectored operations using the Python programming paradigm.

In the next chapter, we will take a roller coaster ride of game programming using the turtle module by applying our knowledge from this chapter. We will make multiple games such as Snake, Pong, and Flappy Bird. Now, it's time for you to start experimenting with vectors; try to mix them together and develop various kinds of movements for the vector.

10
Upgrading the Snake Game with Turtle

Most computer gamers regard games as exciting and appealing due to their appearance. To some extent, this is true. Computer games must be visually attractive so that the player feels like they are physically participating in them. Most game developers and game designers spend a profuse amount of time developing game graphics and animations so as to provide a better experience to the player.

This chapter will teach you how to build the basic layout of games from scratch using the Python `turtle` module. As we know, the `turtle` module allows us to make games with a two-dimensional (2D) motion; thus, we will only be making 2D games such as flappy bird, pong, and snake in this chapter. The concept that we will be covering in this chapter is extremely important in order to bind movements with user actions for the game character.

By the end of this chapter, you will have learned how to implement data models by creating 2D animations and games. Consequently, you will learn how to deal with the different components of game logic, such as defining collisions, boundaries, projections, and screen tap events. By learning about such aspects of game programming, you will be able to learn how to define and design game components using the `turtle` module.

The following topics will be covered in this chapter:

- Overview of computer pixels
- Simple animation using the Turtle module
- Upgrading the snake game using Turtle
- The pong game
- The flappy bird game
- Game testing and possible modifications

Technical requirements

You are expected to have the following resources:

- Python 3.5 or newer
- Python IDLE (Python's inbuilt IDE)
- A text editor
- A web browser

The files for this chapter can be found here: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter10`

Check out the following video to see the code in action:

`http://bit.ly/2oJLeTY`

Exploring computer pixels

When you observe the computer screen closely, you might find small dots forming rows and columns. From a certain distance, the matrix of dots represents images, which we normally see when we look at the screen. These dots are called pixels. Since computer games should be made to be pleasantly visual in nature, we have to work with these pixels in order to create and customize the game screen, and even use them to make a player move in the game, which will be shown on the screen. Whenever a player presses any key on the keyboard, changes in movement must be reflected in the pixels of the screen. For example, when a player presses the **RIGHT** key, a specific character must move a number of units in pixels to the right on the screen in order to represent motion. We discussed vectored motion in the previous chapter, which is able to override the methods of some classes in order to implement motion. We will use the technique of vectors to make pixel movement for the game characters. Let's observe the following outline, which we are going to adapt for making any games using vectors and the turtle module:

1. Make a `Vector` class, which will have methods such as `__add__()`, `__mul__()`, and `__div__()`, which will perform arithmetic operations on our vector points.
2. Use the `Vector` class to instantiate a player on the game screen, with its aiming target or movements.
3. Make a game boundary using the `turtle` module.

4. Draw game characters using the `turtle` module.
5. Operations such as rotate, forward, and move should be used from the `Vector` class in order to make a game character move.
6. Handle user events using the main loop.

We will learn about pixel representation by making a simple **Mario** pixel art. The following code shows the representation of pixels in the multi-dimensional list, which is a list of lists. We have stored each pixel on a single line using the multi-dimensional list:

```
>>> grid = [[1,0,1,0,1,0],[0,1,0,1,0,1],[1,0,1,0,1,0]]
```

The preceding grid consists of three lines that represent the pixel positions. Similar to the list element extract method, the `>>> grid[1][4]` statement returns a positional value of '0' from the second list (that is, [0,1,0,1,0,1]) of the grid. (Refer to `Chapter 4`, *Data Structures and Functions*, to learn more about list operations.) Thus, we can access any cell within the grid.

The following code should be written inside the Python script. By creating a `mario.py` file, we will use it to create Mario pixel art:

1. Start by importing turtle—`import turtle`—the only module we are going to use.
2. Instantiate the turtle module using the `>>> Pen = turtle.Turtle()` command.
3. Specify two properties for the pen using speed and color attributes:

```
Pen.speed(0)
    Pen.color("#0000000")   #or Pen.color(0, 0, 0)
```

4. We must make a `new` function, named `box`, which will draw a box by drawing the square shape using turtle methods. This box size represents the dimension for the pixel art:

```
def box(Dimension): #box method creates rectangular box
        Pen.begin_fill()
    # 0 deg.
        Pen.forward(Dimension)
        Pen.left(90)
    # 90 deg.
        Pen.forward(Dimension)
        Pen.left(90)
    # 180 deg.
        Pen.forward(Dimension)
        Pen.left(90)
    # 270 deg.
```

```
        Pen.forward(Dimension)
        Pen.end_fill()
        Pen.setheading(0)
```

5. We must position the pen to start painting from the top-left position of the screen. These commands should be defined outside of the `box()` function:

```
Pen.penup()
Pen.forward(-100)
Pen.setheading(90)
Pen.forward(100)
Pen.setheading(0)
```

6. Define the box size, which represents the dimension of the pixel art that we are going to draw:

```
boxSize = 10
```

7. In the second phase, you have to declare pixels in the form of multi-dimensional lists, which represents the position of each pixel. The following `grid_of_pixels` variable represents the grid of lines that represent the positions of the pixels. The following line of code must be added outside the `box` function definition. (Refer to `https://github.com/PacktPublishing/Learning-Python-by-building-games` to locate the game file, that is, `mario.py`.):

Remember that a combination of pixels in a single form represents a straight line.

```
grid_of_pixels = [[1,1,1,1,2,2,2,2,2,2,2,2,1,1,1,1]]
grid_of_pixels.append([1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,1])
grid_of_pixels.append([1,1,1,0,0,0,3,3,3,3,3,0,3,1,1,1])
grid_of_pixels.append([1,1,0,3,0,3,3,3,3,3,3,0,3,3,3,1])
grid_of_pixels.append([1,1,0,3,0,0,3,3,3,3,3,3,0,3,3,3])
grid_of_pixels.append([1,1,0,0,3,3,3,3,3,3,3,0,0,0,0,1])
grid_of_pixels.append([1,1,1,1,3,3,3,3,3,3,3,3,3,3,1,1])
grid_of_pixels.append([1,1,1,0,0,2,0,0,0,0,2,0,1,1,1,1])
grid_of_pixels.append([1,1,0,0,0,2,0,0,0,0,2,0,0,0,1,1])
grid_of_pixels.append([0,0,0,0,0,2,2,2,2,2,2,0,0,0,0,0])
grid_of_pixels.append([3,3,3,0,2,3,2,2,2,2,3,2,0,3,3,3])
grid_of_pixels.append([3,3,3,3,2,2,2,2,2,2,2,2,3,3,3,3])
grid_of_pixels.append([3,3,3,2,2,2,2,1,1,2,2,2,2,3,3,3])
grid_of_pixels.append([1,1,1,2,2,2,1,1,1,1,2,2,2,1,1,1])
grid_of_pixels.append([1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1])
grid_of_pixels.append([0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0])
```

8. Define the color palette for the pixel art using colors. We will use color code to define the colors for the art, as shown in the following code. The hexadecimal color code (HEX) represents the color combination of red, green, and blue (#RRGGBB). Refer to `https://htmlcolorcodes.com/` in order to analyze the different codes for different colors:

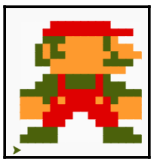
```
palette = ["#4B610B" , "#FAFAFA" , "#DF0101" , "#FE9A2E"]
```

9. Next, we should start drawing the pixel art using a grid of pixels and the color palette that we defined in *steps 7* and *step 8*. We have to use the `box` class that we made previously using the `box()` function to make the pixel art. The pixel art consists of rows and columns; thus, we have to declare two loops for drawing art. The following code calls different functions from the turtle module, such as `forward()`, `penup()`, and `pendown()`. We studied them in the previous chapter; they will make use of the pen to draw, based on the list of lists that were defined by the grid of pixels:

```
for i in range (0,len(grid_of_pixels)):
        for j in range (0,len(grid_of_pixels[i])):
            Pen.color(palette[grid_of_pixels[i][j]])
            box(boxSize)
            Pen.penup()
            Pen.forward(boxSize)
            Pen.pendown()
        Pen.setheading(270)
        Pen.penup()
        Pen.forward(boxSize)
        Pen.setheading(180)
        Pen.forward(boxSize*len(grid_of_pixels[i]))
        Pen.setheading(0)
        Pen.pendown()
```

Let's digest the previous code snippet. It contains a `for` loop, which loops from an initial value of 0 to the length of the grid of pixels that represent positions in the canvas. Each pixel represents one position, where we must draw using the pen; thus, we loop on each of those pixels, one at a time. Inside the 2D `for` loop, we fetch the color from the palette and call the `box` method, which creates a rectangular box where our Mario art should be rendered. We draw inside this box with the turtle pen by using the `forward()` function. We do the same operation in the rows of pixels, as indicated by the i^{th} loop.

Once we have finished combining the preceding code, that is, we have carried out the `box` method, initialization, and two main `for` loops, we are ready to run the code and observe the following Mario pixel art. After running our code, the pen from the `turtle` module will start drawing, and eventually it will give us the following art:



Since we are familiar with the concept of pixels and vectored motion, it's time to make games using 2D graphics. We will be using the `turtle` module, along with data models, in order to create game characters and make them move. We will start this adventure by making a simple animation in the next section.

Understanding simple animation using the Turtle module

By now, we are probably familiar with the different methods of the `turtle` module. This means we won't have any problem creating characters for the game. Similarly, motions for game characters are provided using vectored movements. Operations such as vectored addition and subtraction provide linear movement in a straight line through the rotation of objects (refer to `Chapter 9`, *Data Model Implementation*, for more information). The `move` operation that is defined in the following code snippet will provide random movements to game characters. The `move` method will take another vector as a catalyst and will perform mathematical operations in order to update the current position, while also considering the direction of the game character:

```
>>> v = (1,2) #vector coordinates
>>> v.move(3,4) # vector addition is done (1,2) + (3,4)
>>> v
(4,6)
```

The `rotate` method will rotate the vector counter-clockwise by a specific angle (in-place). The following sample represents the `rotate` method call:

```
>>> v = vector(1, 2)
>>> v.rotate(90)
>>> v == vector(-2, 1)
True
```

We have to define the preceding two methods inside the `Vector` class. Follow this procedure to implement the `Vector` class:

1. You have to start by defining the `Vector` class with the class keyword. We will define slots as class attributes, which will contain three attributes. The slots represent an attribute, which contains three pieces of critical information: *x*, *y*, and hash. The values *x* and *y* are the current position of the game character, while hash is used to locate the data record. For instance, if the `Vector` class is instantiated with *x* and *y* coordinates, then the hash attribute will be activated. Otherwise, it remains deactivated.
2. The coordinates of the vectored elements, that is, (5,6), are represented by *x* and *y*, where *x = 5* and *y = 6*, and the hash variable represents whether the slot is empty. The hash variable is used to locate the data records and to check whether the `Vector` class is instantiated. If the slot attribute already contains *x* and *y*, this hash attribute will restrain from further assignment to the slots. We will also define the `PRECISION` attribute (user-defined), which will round the coordinates of *x* and *y* to a certain level. In order to make things clear, several examples have been added inside the code, and you can observe this inside three-line comments:

```
#following class will create vector
#representing current position of game character
class vector(collections.Sequence):
    """Two-dimensional vector.
    Vectors can be modified in-place.
    >>> v = vector(0, 1)
    >>> v.move(1)
    >>> v
    vector(1, 2)
    >>> v.rotate(90)
    >>> v
    vector(-2.0, 1.0)
    """
    PRECISION = 6 #value 6 represents level of rounding
    #for example: 4.53434343 => 4.534343
    __slots__ = ('_x', '_y', '_hash')
```

3. Next, we need to define the first member of the class. We know that the first member of the class is the `__init__()` method. We will define it in order to initialize the class attributes, which are *x* and *y*. We have rounded the values of *x* and *y* to a certain level of precision, as indicated by the `PRECISION` attribute. `round()` is a built-in function of Python. The following line of code contains a constructor, where we initialize the vector coordinates (*x*, *y*) using the `round` method:

```
def __init__(self, x, y):
        """Initialize vector with coordinates: x, y.
        >>> v = vector(1, 2)
        >>> v.x
        1
        >>> v.y
        2
        """
        self._hash = None
        self._x = round(x, self.PRECISION)
        self._y = round(y, self.PRECISION)
```

4. You might have observed that you have made *x* and *y* attributes as private attributes, as they begin with a single underscore (`_x`, `_y`). Thus, direct initialization cannot be done in these types of attributes, which leads to **data encapsulation**, which we covered back in the object-oriented paradigms topic. Now, in order to fetch and set the values of these attributes, you have to use the `getter` and `setter` methods. These two methods will be a property of the `Vector` class. The following code represents how to implement `getter` and `setter` for our `Vector` class:

```
@property
    def x(self):
        """X-axis component of vector.
        >>> v = vector(1, 2)
        >>> v.x
        1
        >>> v.x = 3
        >>> v.x
        3

        """
        return self._x
    @x.setter
    def x(self, value):
        if self._hash is not None:
            raise ValueError('cannot set x after hashing')
```

```
        self._x = round(value, self.PRECISION)

    @property
    def y(self):
        """Y-axis component of vector.
        >>> v = vector(1, 2)
        >>> v.y
        2
        >>> v.y = 5
        >>> v.y
        5
        """
        return self._y

    @y.setter
    def y(self, value):
        if self._hash is not None:
            raise ValueError('cannot set y after hashing')
        self._y = round(value, self.PRECISION)
```

5. Along with the `getter` and `setter` methodologies, you may have observed `_hash`, which represents if the slot is already allocated or not. In order to check whether the slot is already appropriated, we have to implement a data model, that is, `__hash__()`.

Just a quick review: data models, or magic functions, allow us to change the implementation of a method that is provided by one of its ancestors.

Now, we will define the `hash` method on our `Vector` class and implement it differently:

```
def __hash__(self):
        """v.__hash__() -> hash(v)
        >>> v = vector(1, 2)
        >>> h = hash(v)
        >>> v.x = 2
        Traceback (most recent call last):
            ...
        ValueError: cannot set x after hashing
        """
        if self._hash is None:
            pair = (self.x, self.y)
            self._hash = hash(pair)
        return self._hash
```

6. Finally, you have to implement two main methods in the `Vector` class: `move()` and `rotate()`. We will start with the `move` method. The `move` method will move the vector by other (in-place). Here, other is the argument that is passed to the `move` method. For example, `(1, 2).move(2, 3)` will result in (3, 5). Remember: movement is done by any of the vectored arithmetic operations, that is, add, multiply, divide, and so on. We will use the `__add__()` magic function (refer to `Chapter 9`, *Data Model Implementation*) in order to create movement for the vector. Before that, we have to make a copy method that will return the copy of the vector. The `copy()` method is essential because we don't want the operations to harm our original vector; instead, we will perform arithmetic operations on the copy of the original vector:

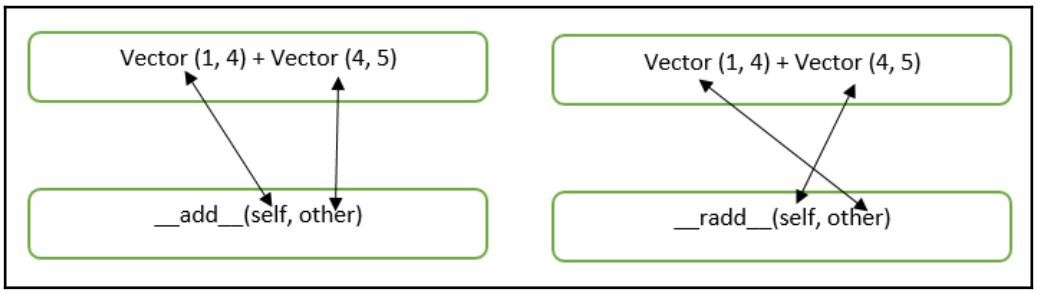
```
def copy(self):
        """Return copy of vector.
        >>> v = vector(1, 2)
        >>> w = v.copy()
        >>> v is w
        False
        """
        type_self = type(self)
        return type_self(self.x, self.y)
```

7. You have to implement the `iadd` magic function before implementing the `add` function. We use the `__iadd__` method to implement the extended `add` operator assignment. We can implementing the `__iadd__()` magic function inside the `Vector` class as follows. We saw its implementation in the previous chapter (`Chapter 9`, *Data Model Implementation)*:

```
def __iadd__(self, other):
        """v.__iadd__(w) -> v += w
        >>> v = vector(1, 2)
        >>> w = vector(3, 4)
        >>> v += w
        >>> v
        vector(4, 6)
        >>> v += 1
        >>> v
        vector(5, 7)
        """
        if self._hash is not None:
            raise ValueError('cannot add vector after hashing')
        elif isinstance(other, vector):
            self.x += other.x
            self.y += other.y
        else:
```

```
            self.x += other
            self.y += other
        return self
```

8. Now, you have to make a new method, `__add__`, which will call the preceding `__iadd__()` method on the copy of the original vector. The last statement, `__radd__ = __add__`, has significant meaning. Let's observe the following diagrammatic relationship between `radd` and add. It works like this: Python tries to evaluate the expression, *Vector(1,4) + Vector(4,5)*. First, it calls `int.__add__((1,4), (4,5))`, which raises an exception. After this, it will try to invoke `Vector.__radd__((1,4), (4,5))`:



It's easy to recognize that the implementation of `__radd__` is analogous to `add`: (refer to the example code defined inside the comments in the `__add__()` method):

```
def __add__(self, other):
        """v.__add__(w) -> v + w
        >>> v = vector(1, 2)
        >>> w = vector(3, 4)
        >>> v + w
        vector(4, 6)
        >>> v + 1
        vector(2, 3)
        >>> 2.0 + v
        vector(3.0, 4.0)

        """
        copy = self.copy()
        return copy.__iadd__(other)
    __radd__ = __add__
```

9. Finally, we are ready to make the first movement sequence for our animation. We will start by defining the `move` method in our class. The `move()` method will take a single argument as a vector and add it to the current vector that represents the current position of the game character. The `move` method will implement a straight line addition. The following code represents the definition of the `move` method:

```
def move(self, other):
        """Move vector by other (in-place).
        >>> v = vector(1, 2)
        >>> w = vector(3, 4)
        >>> v.move(w)
        >>> v
        vector(4, 6)
        >>> v.move(3)
        >>> v
        vector(7, 9)
        """
        self.__iadd__(other)
```

10. Next, we need to create the `rotate()` method. This method is quite tricky to create, as it will rotate the vector counter-clockwise by a specified angle (in-place). This method will use trigonometric operations such as the sine and cosine of the angle; thus, we have to import a math module first: `import math`.
11. The following code depicts the way of defining the rotate method; inside it, we have added comments to make this operation clear to you. At first, we have converted the angle into a radian with the: `angle*π/ 180.0` command/formula. After that, we fetched *x* and *y* coordinates of the vector class and performed the `x = x*cosθ - y*sinθ` and `y = y*cosθ + x*sinθ` operations:

```
import math
def rotate(self, angle):
        """Rotate vector counter-clockwise by angle (in-place).
        >>> v = vector(1, 2)
        >>> v.rotate(90)
        >>> v == vector(-2, 1)
        True

        """
        if self._hash is not None:
            raise ValueError('cannot rotate vector after hashing')
        radians = angle * math.pi / 180.0
        cosine = math.cos(radians)
        sine = math.sin(radians)
        x = self.x
```

```
        y = self.y
        self.x = x * cosine - y * sine
        self.y = y * cosine + x * sine
```

The mathematical formula, $x = x*cos\theta - y*sin\theta$, is significant in vectored motion. This formula is used to provide rotational movements to the game characters. $x*cos\theta$ represents the base x-axis movements, while $y*sin\theta$ represents the vertical y-axis movements. Thus, this formula facilitates the rotation of a point in a 2D plane with an angle of θ.

Finally, we have completed two methods: `move()` and `rotate()`. These two methods are completely unique, but they both represent vectored motion. The `move()` method has implemented the `__iadd_()` magic function, while the `rotate()` method has its own custom trigonometric implementation. The combination of these two methods can form complete movement for game characters on the canvas or game screen. To construct any type of 2D game, we have to implement similar kinds of movements. Now, we will make a simple animation of an ant game in order to begin our tour of our gaming adventure.

The following steps depict the procedure of making any animation for 2D games:

1. Firstly, you have to import the necessary module. Since we have to give random vector coordinates to the previously made `move()` method, we can predict that we will need a random module.
2. After that, we need another module—a `turtle` module—which will allow us to call methods such as ontimer and setup. We also need the methods of a vector class, that is, `move()` and `rotate()`.
3. We have to import it if that class is maintained in any other module or file. Create two files: `base.py` for vector movements and `animation.py` for animation. Then, import the following statements:

```
from random import *
from turtle import *
from base import vector
```

4. The first two statements are going to import everything from the random and `turtle` modules. The third statement is going to import the vector class from the base file or module.
5. Next, we need to define the initial position for the game character, along with its aim. It should be initialized as an instance of the vector class:

```
ant = vector(0, 0) #ant is character
aim = vector(2, 0) #aim is next position
```

6. Now, you have to define the wrap method. This method takes *x* and *y* positions as an argument that is referred to as `value` and returns it. In the upcoming games, such as flappy bird and Pong, we will extend this function and make it wrap the value around certain boundary points:

```
def wrap(value):
    return value
```

7. The main controlling unit of the game is the `draw()` function, which calls a method to make game character move. It also draws a screen for the game. From the `Vector` class, we are going to call the `move` and `rotate` methods. From the turtle module, we are going to call the `goto`, `dot`, and `ontimer` methods. The `goto` method will move the turtle pen to a specified position on the game screen, the `dot` method will create a small dot of a specified length when called, and the `ontimer(function, t)` method will install a timer, which calls that function after `t` milliseconds:

```
def draw():
    "Move ant and draw screen."
    ant.move(aim)
    ant.x = wrap(ant.x)
    ant.y = wrap(ant.y)
    aim.move(random() - 0.5)
    aim.rotate(random() * 10 - 5)
    clear()
    goto(ant.x, ant.y)
    dot(10)
    if running:
        ontimer(draw, 100)
```

8. In the preceding code, the `running` variable was not declared. We will do it now, outside the definition of the `draw()` method. We will also set up the game screen using the following code:

```
setup(420, 420, 370, 0)
hideturtle()
tracer(False)
up()
running = True
draw()
done()
```

Finally, we have completed a simple 2D animation. It consists of a simple dot of a length of 10 pixels, but more importantly, it has motion attached to it, which is the result of implementing magic functions inside the `Vector` class. The next section will teach us how to use the magic functions that we implemented in this section in order to make a more robust game, which is the Snake game. We will make a Snake game using the turtle module and magic functions.

Upgrading the snake game using Turtle

As it turns out, we have been building the snake game in the previous chapters of this book: in `Chapter 5`, *Learning About Curses by Building a Snake Game*, using the curses module; in `Chapter 6`, *Object-Oriented Programming*; and in `Chapter 7`, *List Comprehension and Properties,* by refining it using properties and list comprehension. We started with the curses module (`Chapter 5`, *Learning About Curses by Building a Snake Game*), and modified it using an object-0riented paradigm. The curses module was able to provide a character-based Terminal game screen, which eventually made the game character look awful. Although, we learned how to build logic using **OOP** and curses, along with making the Snake game, it should be noted that games are primarily concerned with visuals: how a player sees and interacts with the characters. Thus, our primary concern is to make games visually appealing. In this section, we will try to upgrade the Snake game using both the turtle module and vectored movements. Since there is only one possible movement in the case of the Snake game, which is a straight-line movement by pressing the **LEFT, RIGHT, UP,** or **DOWN** key, we don't have to define anything new inside the vector class of the base file. The `move()` method, which we made previously, is enough to provide the movements for the snake game.

Let's start coding the Snake game using the turtle module and `Vector` class, by following these steps:

1. As usual, start by importing the necessary modules, as shown in the following code. It is not compulsory for you to import everything first; we can do it along with coding other stuff too, but it's good practice to import everything at once so that we don't forget anything afterward:

```
from turtle import *
from random import randrange
from base import vector
```

2. Now, let's brainstorm a little bit. We can't use sprites or images yet. We will learn about these in the upcoming chapters, after we get started with Pygame. For now, we have to make a shape that represents a 2D snake, which is our main character. You have to open the `base.py` file, where we created a `Vector` class and defined a `Square` method. Note that the `Square` method is declared outside the `Vector` class. The following code is a simple implementation of the turtle methods that will create square shapes using the turtle pen:

```
def square(x, y, size, name):
    """Draw square at `(x, y)` with side length `size` and fill color
     `name`.
    The square is oriented so the bottom left corner is at (x, y).
    """
    import turtle
    turtle.up()
    turtle.goto(x, y)
    turtle.down()
    turtle.color(name)
    turtle.begin_fill()
    for count in range(4):
        turtle.forward(size)
        turtle.left(90)

    turtle.end_fill()
```

3. Next, import this newly made method inside the Snake game module. Now, we can call the square method inside our Snake game's Python file:

```
from base import square
```

4. After importing everything, we will declare variables such as food, snake, and aim. The food represents the vector coordinates, which is an instance of the `Vector` class, for example, vector(0,0). The snake represents the initial vectored position for the snake character, that is, (vector(10,0)), while the body of the snake must be a list of representations for the vector, that is, (vector(10,0), vector(10,1), and vector(10,2)) for a snake of length 3. The `aim` vector represents the unit that must be added or subtracted to the current snake vector, based on the user's keyboard actions:

```
food = vector(0, 0)
snake = [vector(10, 0)]
aim = vector(0, -10)
```

5. Inside the `snake-Python` file (the main file), after importing everything and declaring its attributes, we will start by defining the boundary for the Snake game, as follows:

```
def inside(head):
    "Return True if head inside boundaries."
    return -200 < head.x < 190 and -200 < head.y < 190
```

6. You should also define another important method of the Snake game, which is known as `move()`, since this will take care of the movement of the Snake character on the game screen, as follows:

```
def move():
    "Move snake forward one segment."
    head = snake[-1].copy()
    head.move(aim)

    if not inside(head) or head in snake:
        square(head.x, head.y, 9, 'red')
        update()
        return
    snake.append(head)

    if head == food:
        print('Snake:', len(snake))
        food.x = randrange(-15, 15) * 10
        food.y = randrange(-15, 15) * 10
    else:
        snake.pop(0)
    clear()
    for body in snake:
        square(body.x, body.y, 9, 'black')

    square(food.x, food.y, 9, 'green')
    update()
    ontimer(move, 100)
```

7. Let's start by understanding the code line by line:

    - At the beginning of the `move` method, we fetched `snakehead` and performed a copy operation, which is defined inside the `Vector` class, and we made a snake move one segment ahead automatically because we want the snake to move automatically as soon as the user starts playing the game.
    - After that, the `if not inside(head) or head in snake` statement is used to check for any collisions. If there are any, we will return by rendering the `Red` color to the snake.

- In the next line of the statement, `head == food`, we checked whether the snake was able to eat food or not. As soon as the player eats the food, we will make food appear in another random position, as well as print the score in the Python console.
- In the `for body in snake: ..` statement, we looped into the entire body of the snake and rendered the `black` color to it.
- The `square` method, which is defined inside the `Vector` class, is called to create food for the game.
- At the last statement of the code, the `ontimer()` method was called, which takes the `move()` function, and it will install a timer that will call in the `move` method every 100 milliseconds.

8. After defining the `move()` method, you have to set up the game screen and handle the turtle screen. The parameters that are passed with the `setup` method are the `width`, `height`, `setx`, and `sety` positions:

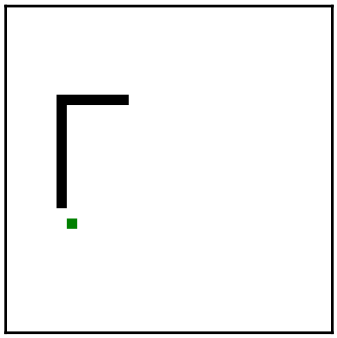
```
setup(420, 420, 370, 0)
hideturtle()
tracer(False)
```

9. The last part of our game is to handle the user events. We have to make the user play the game; thus, we must call the appropriate functions whenever a keyboard input is received from the user. As Snake is simple game, consisting of only a few movements, we will address it in the next section. As soon as the user presses any key, we have to handle it by changing the snake's direction. Thus, we have to make one quick method for handling the user's actions. The following `change()` method is going to change the snake's direction, based on the user events. Here, we've used the `listen` interface provided by the turtle module, which will listen for any incoming user events or keyboard inputs. `onkey()` takes the function, which will call the change method based on the user events. For example, when `Up` is pressed, we will make changes in the *y* coordinate by increasing the current `y` value by 10 units:

```
def change(x, y):
    "Change snake direction."
    aim.x = x
    aim.y = y

listen()
onkey(lambda: change(10, 0), 'Right')
onkey(lambda: change(-10, 0), 'Left')
onkey(lambda: change(0, 10), 'Up')
onkey(lambda: change(0, -10), 'Down')
```

```
move()
done()
```

It's time to run our game, but before that, remember to keep both files (the file containing the `vector` and the `square` class, and the file containing the Snake game) in the same directory. The output of the game looks something like this:



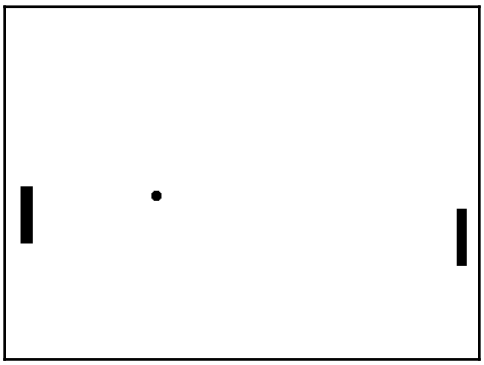
Along with the turtle graphics, we can look at the score printed right next to it within the Python terminal:

```
Snake: 2
Snake: 3
Snake: 4
Snake: 5
Snake: 6
Snake: 7
Snake: 8
Snake: 9
Snake: 10
Snake: 11
Snake: 12
Snake: 13
Snake: 14
Snake: 15
Snake: 16
Snake: 17
```

Now that we have covered the Snake game by making use of several methods provided by the Python module and the OOP paradigm, we can reuse these things over and over again in the upcoming games. The `Vector` class that was defined in the `base.py` file can be revisited time and again for many 2D games. Thus, the reutilization of code is one of the prime merits that is provided by OOP. We will make several games, such as Pong and flappy bird, in the upcoming sections using only the `Vector` class. In the next section, we are going to build the Pong game from scratch.

Exploring the Pong game

Now that we have covered the Snake game (although it's cliche, it's perfect to grasp the knowledge of 2D game programming) now, it's time to make another interesting game. The game we are going to cover in this section is the Pong game. If you have played it before, you might find it easier to grasp the concept that we will cover in this section. For those who haven't played it before, don't worry! We will cover everything in this section, which will help you make your very own Pong game and play it or even share it with your friends. The following diagram is the pictorial representation of the Pong game:



The preceding diagram depicts the playground for the Pong game, where two players are two rectangles. They can move up and down, but not left to right. The **dot** in the center is the ball, which has to be hit by either player. We have to address two types of motion for the game characters in this game:

- For the ball, which can move in any position, but if the player on either side fails to receive the ball, they lose, and the opposing player wins.
- For the player, they should only move either up or down: four keyboard key actions should be handled for two players.

Apart from the motion, it is even trickier to specify the boundary for the game. The horizontal line, which can move up and down, is the position from where the ball must be hit and reflected in the other direction, but if the ball hits either the left or right vertical boundary, the game should be halted and the player who missed the ball will lose. Now, let's brainstorm so that we know about the essentials before actually starting to code:

- Create a random function, which may return a random value but within the same range that is determined by the screen's height and width. The value that is returned from this function might be useful to make it aim, which is a random movement for the ball in the game.

- Create a method that will draw two rectangles on the screen, which are, in fact, our players of the game.
- The third function should be declared, which will draw the game and move the Pong ball across the screen. We can use the `move()` method, which is defined inside the previously made `Vector` class, which will move the vector by other (in-place).

Now that we are done with the logistics, we can start to code. Follow these steps in order to make your own Pong game:

1. Start by importing the necessary modules, that is, random, turtle, and our custom-made module, named `base`, which has a bunch of methods for vectored motion:

```
from random import choice, random
from turtle import *
from base import vector
```

2. The following code represents the definition for the `value()` method, and three assignments of variables. The `value()` method will randomly generate values between (-5, -3) and (3, 5). The three assignment statements are understandable by their names:

    - The first statement represents the initial position of the ball.
    - The second statement is the further aim of the ball.
    - The third statement is the `state` variable, which tracks the status of the two players:

```
def value():
    "Randomly generate value between (-5, -3) or (3, 5)."
    return (3 + random() * 2) * choice([1, -1])
ball = vector(0, 0)
aim = vector(value(), value())
state = {1: 0, 2: 0}
```

3. The next function is an interesting one; this will render the rectangular shape onto the game screen. We can use the turtle module and its method to render any shape, as follows:

```
def rectangle(x, y, width, height):
    "Draw rectangle at (x, y) with given width and height."
    up()
    goto(x, y)
    down()
```

```
    begin_fill()
    for count in range(2):
        forward(width)
        left(90)
        forward(height)
        left(90)
    end_fill()
```

4. After we make the function to draw a rectangle, we need to make a new method that can call the methods that were defined in the preceding steps. Along with this, the new method should also move the Pong ball flawlessly onto the game screen:

```
def draw():
    "Draw game and move pong ball."
    clear()
    rectangle(-200, state[1], 10, 50)
    rectangle(190, state[2], 10, 50)
    ball.move(aim)
    x = ball.x
    y = ball.y

    up()
    goto(x, y)
    dot(10)
    update()
```

5. Now, it's time to address the main riddle of the game: what happens when the ball hits the horizontal and vertical boundaries, or when it hits the player's rectangular bat ? We can use the `setup` method to create the game screen with a custom height and width. The following code should be added within the `draw()` function:

```
#when ball hits upper or lower boundary
#Total height is 420 (-200 down and 200 up)
    if y < -200 or y > 200:
        aim.y = -aim.y
#when ball is near left boundary
    if x < -185:
        low = state[1]
        high = state[1] + 50
        #when player1 hits ball
        if low <= y <= high:
            aim.x = -aim.x
        else:
            return
#when ball is near right boundary
```

```
    if x > 185:
        low = state[2]
        high = state[2] + 50
        #when player2 hits ball
        if low <= y <= high:
            aim.x = -aim.x
        else:
            return
    ontimer(draw, 50)
```

6. Now that we've addressed the movement for the game characters, we have to make the game screen and find a way to handle user events. The following code will set up the game screen, which is called in from the turtle module:

```
setup(420, 420, 370, 0)
hideturtle()
tracer(False)
```

7. After we make a game screen, we have to listen to and handle the user's key events by making a custom function. We will make the `move()` function, which will move the player's position by a certain number of units that are passed while calling this function. This move function will take care of the up and down movements of the rectangular bat:

```
def move(player, change):
    "Move player position by change."
    state[player] += change
```

8. Finally, we will use the `listen` interface that is provided by the turtle method to handle incoming key events. Since there are four possible movements, that is, up and down for each player, we will reserve four keyboard keys [*W*, *S*, *I*, and *K*], which will have the listener attached internally by turtle, as shown in the code that follows:

```
listen()
onkey(lambda: move(1, 20), 'w')
onkey(lambda: move(1, -20), 's')
onkey(lambda: move(2, 20), 'i')
onkey(lambda: move(2, -20), 'k')
draw()
done()
```

The previous steps are quite simple to understand, but let's grasp the concepts we defined in *step 4* and *step 5* more eloquently. In *step 4*, the first two lines of code after the `clear()` method will create a rectangular geometrical shape of a specified height and width. `state[1]` represents the first player, while `state[2]` represents the second player. The `ball.move(aim)` statement is a call to the `move` method that is declared inside the vector class.

This method call will perform the addition between the specified vectors, which results in a straight line of motion. The `dot(10)` statement will create a ball of a width of 10 units.

Similarly, in *step 5*, we used the `>>> setup(420, 420, 370, 0)` statement to create a screen that has a width of 420px and a height of 420px. There must be a change in direction when the ball hits the upper and lower boundaries by some amount, and the amount is exactly the negative of the current *y* (*-y* reverses the direction). However, when the ball hits either the left or right boundary, the game must terminate. After we check for the upper and lower boundaries, we make a comparison for the *x* coordinate and check for low and high states. If the ball is under these values, it must have collided with the bat, otherwise we return the `from` function. Make sure you add this code inside the previously defined `draw()` function.

When you run your Pong game file, you will observe two screens; one screen will have a turtle graphics screen consisting of two players ready to play your very own Pong game. The output will be similar to the diagram we saw previously when brainstorming the Pong game. Now that you know a lot about the ways of handling keyboard actions, and making a call to the custom functions with the turtle `ontimer` function, let's make something new, which will have a controller. It will listen for screen tap actions and provide responses to them. We need this in games such as flappy bird, where the user taps on the screen and changes the position of the bird.

Understanding the flappy bird game

Whenever we talk about games having a screen-tap action or onscreen click action, flappy bird comes to mind. If you haven't played it before, make sure you check it out at `https://flappybird.io/` in order to get familiar with it. Although the interface that you see in this website won't be the same as the flappy bird game we are going to make in this section, don't worry—we will emulate its interface after we learn about Python's GUI module, known as *Pygame*. But for now, we will make a simple 2D flappy bird game using the Python turtle module and vectored motion. We have been using the `onkey` method to handle keyboard actions, and in the preceding section, we used the `onkey` method to embed a listener to the specific keyboard keys.

However, there are also games that can be played using mouse actions—by clicking onto the game screen. In this section, we are going to follow these steps in order to create Flappy, a game inspired by flappy bird:

1. First of all, you should define a boundary for the gameplay. You can make a function that takes an argument as a vector point and checks if it is inside the boundary or not and accordingly returns `True` or `False`.

2. You have to make a rendering function that will draw game characters onto the screen. As we know, turtle is unable to handle many images or sprites in the GUI; therefore, your game character will resemble geometrical shapes. You can represent your bird character by making any shape. If possible, try to make it small.
3. After making a render function, you have to create a function that will be able to update the objects' positions. This function should be able to handle the `tap` action.

We can use the predefined `Vector` blueprint throughout the coding of the flappy bird game. The previous roadmap clearly implies that we can make a simple flappy bird game by defining three functions. Let's define each of these functions, one by one:

1. Firstly, you have to set up the screen. This screen represents the output game console where you will play our flappy bird game. You can create a game screen using the turtle module by using `setup()`. Let's create a screen which has a width of 420 pixels and a height of 420 pixels:

```
from turtle import *
setup(420, 420, 370, 0)
```

2. You should define a function that will check whether the user has tapped or touched inside the boundary. This function should be a Boolean and should return `True` if the tapped-point is inside the boundary; otherwise, it should return `False`:

```
def inside(point):
    "Return True if point on screen."
    return -200 < point.x < 200 and -200 < point.y < 200
```

3. I have already recommended that you take a tour of the flappy bird game if you haven't played it before. While playing it, you will observe that the goal of the game is to protect the *bird* character from the obstacles. In a real-world game, we have obstacles in the form of vertical pipes. Since we don't have enough resources to use while coding with the turtle module, we won't be able to use such sprites or interfaces in this section. As I have already told you, we are going to make cool interfaces by ourselves while learning about Pygame, but for now, instead of the GUI, we will be focusing highly on the game logic. Thus, we will give some random shapes to the game character; small round shapes for the bird character and big round shapes for the obstacles. The bird will be instantiated from the vector class, which represents its initial position. The ball (obstacles) must be made as a list because we want obstacles to be in the path of the bird:

```
bird = vector(0, 0)
balls = []
```

4. Now that you are familiar with the game characters, you can render them by creating some functions. In the function, we have passed `alive` as a variable, which will be a Boolean, and this will check whether the player is dead or not. If the bird is alive, we jump to that position using `goto()` and render a dot with a green color to it. If the bird is dead, we render the dot with a red color. The for loop in the following code will render a number of obstacles:

```
def draw(alive):
    "Draw screen objects."
    clear()
    goto(bird.x, bird.y)
    if alive:
        dot(10, 'green')
    else:
        dot(10, 'red')
    for ball in balls:
        goto(ball.x, ball.y)
        dot(20, 'black')
    update()
```

5. As we discussed in the previous blueprint, next up is the main controller of the game. This function must perform multiple tasks, but all of them will be related to updating the objects' position. It will be hard for the users who haven't played flappy bird before to understand the following code; that is why I had encouraged you to take a tour of original flappy bird game. If you inspect the movement of the bird in the game, it is restricted to moving in only the *y*-axis, that is, either up or down. Similarly for the obstacles, they must move from right to left, the same as vertical pipes in the real-world game. The following `move()` function consists of the initial motion for the bird. Initially, we want it to fall by 5 units, and decrease it accordingly. For the part of bird as obstacles, we want it to move from right to left by 3 units:

```
from random import *
from base import vector #for vectored motion
def move():
    "Update object positions."
    bird.y -= 5
    for ball in balls:
        ball.x -= 3
```

6. You have to explicitly create numbers of obstacles inside the `move` function. Since obstacles should spawn randomly, we can use a random module to create it:

```
 if randrange(10) == 0:
    y = randrange(-199, 199)
    ball = vector(199, y)
    balls.append(ball)    #append each obstacles to list
```

7. Next, we need to check whether the player is able to prevent the bird from touching the obstacles. The method to check this is simple. If the ball, or obstacle, is out of the left vertical boundary, we can remove it from the list of balls. Initially, we made the `inside` function to check whether any point is within the boundary; now, we can use it to check whether the obstacle is within the boundary. It should look something like this:

```
while len(balls) > 0 and not inside(balls[0]):
    balls.pop(0)
```

8. Notice that we have added a condition for the obstacles; now, it's time to add a condition to check whether the bird is alive. If the bird falls down and touches the lower boundary, the program should be terminated:

```
if not inside(bird):
    draw(False)
    return
```

9. Now, we will add another condition—one that will check whether the obstacle has collided with the bird. There are several ways of doing this, but for now, we will do this by checking the position of the ball and the obstacle. Firstly, you have to check the size of the obstacle and the bird: the obstacle or ball has a size of 20 pixels, and the bird has a size of 10 pixels (defined at point number 4); thus, we can assume that they have collided when the distance between them is 0. Thus, the `>>> if abs(ball - bird) < 15` expression will check whether the distance between them is less than 15 (considering the width of the ball and the bird):

```
for ball in balls:
    if abs(ball - bird) < 15:
        draw(False)
        return
draw(True)
ontimer(move, 50) #calls move function at every 50ms
```

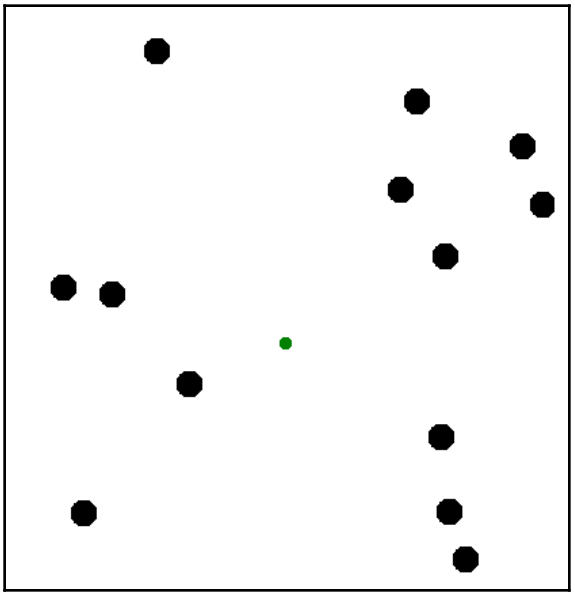
10. Now that we are done updating the object's position, we need to handle user events—this is what should be implemented when the player taps the game screen. When the user taps the screen, we want the bird to rise up by a certain number of pixels. The argument that is passed to the tap function (*x,y*) is the coordinates of the clicked point on the game screen:

```
def tap(x, y):
    "Move bird up in response to screen tap."
    up = vector(0, 30)
    bird.move(up)
```

11. Finally, it's time to add a listener using the turtle module. We will use the `onscreenclick()` function, which will take any user-defined function as an argument (in our case, it is the `tap()` function), which will be called with the coordinates of the clicked point (*x, y*) on the canvas. We have used the tap function to call this listener:

```
hideturtle()
up()
tracer(False)
onscreenclick(tap)
move()
done()
```

This seems like a lot of work, right? It is indeed. We have covered so many things in this section: ways to define boundaries, rendering game objects, updating object positions, and handling tap events or mouse events. I feel that we have already studied a lot about 2D game architecture using the turtle module. Although the games that are made by using the turtle module are not very appealing, the logic we learned about by building these games will be used repeatedly in the upcoming chapters. In these types of games, we don't care about the interface too much, but we will run our game into the Python shell and observe how it looks. The outcome of the preceding program will be something like this:



Error message: No module named 'base'. This is because you haven't added your `Base` module (the Python file that contains the `Vector` class, which we made in the *Simple animation using Turtle module* section) and the Python game file to the same directory. Make sure you create a new directory and store the two files together, or grab the code from the following GitHub link: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter10`.

There is little place for modifying the games that are made out of Turtle. However, I strongly suggest that you to go through it, test the game, and uncover the possible modifications by yourself. If you get any, try to implement them. In the next section, we will cover how to test the game properly and apply modifications so that these games will become more sturdy than before.

Game testing and possible modifications

The fallacious misconception that many people believe is that, in order to become a proficient game tester, you should be a gamer. This may be true to some extent, but mostly, game testers don't care about the frontend design of the game. They primarily focus on the backend part, which deals with *data* communicating between game servers and client computers. I will take you through the game testing and modification process for our Pong game, while covering the following points:

1. **Enhancing game characters**: The following code represents the new model for the game characters. We implement it solely using the turtle module. The *Paddle* is the rectangular box, which represents the player of the pong game. There are two of them, namely paddle A and paddle B:

```
import turtle
# Paddle A
paddle_a = turtle.Turtle()
paddle_a.speed(0)
paddle_a.shape('square')
paddle_a.color('white')
paddle_a.penup()
paddle_a.goto(-350, 0)
paddle_a.shapesize(5, 1)

# Paddle B
paddle_b = turtle.Turtle()
paddle_b.speed(0)
paddle_b.shape('square')
paddle_b.color('white')
paddle_b.penup()
paddle_b.goto(350, 0)
paddle_b.shapesize(5, 1)
```

2. **Adding the main character in the game** (**a ball**): Similar to the creation of the A and B paddles, we will use the turtle module along with commands such as `speed()`, `shape()`, and `color()` to create a ball character and add such functionalities to it:

```
# Ball
ball = turtle.Turtle()
ball.speed(0)
ball.shape('circle')
ball.color('white')
ball.penup()
ball.dx = 0.15
ball.dy = 0.15
```

3. **Adding a score interface to the game**: We will use the turtle pen to draw an interface for the points scored by each player. The following code consists of a method call from the turtle module, that is, the `write()` method, which writes text. It puts the string representation of *arg* in the specified position:

```
# Pen
pen = turtle.Turtle()
pen.speed(0)
pen.color('white')
pen.penup()
pen.goto(0, 260)
pen.write("Player A: 0  Player B: 0", align='center',
  font=('Courier', 24, 'bold'))
pen.hideturtle()
# Score
score_a = 0
score_b = 0
```

4. **Keyboard binding with proper actions**: In the following code, we have bound the keyboard with proper functions. Each keyboard key, when pressed, will call the specified function by using `onkeypress`; this is known as **event handling**. Confused with methods such as `paddle_a_up` and `paddle_b_up`? Be sure to revise *The Pong game* section:

```
def paddle_a_up():
    y = paddle_a.ycor()
    y += 20
    paddle_a.sety(y)

def paddle_b_up():
    y = paddle_b.ycor()
    y += 20
    paddle_b.sety(y)
def paddle_a_down():
    y = paddle_a.ycor()
    y += -20
    paddle_a.sety(y)

def paddle_b_down():
    y = paddle_b.ycor()
    y += -20
    paddle_b.sety(y)

# Keyboard binding
wn.listen()
wn.onkeypress(paddle_a_up, 'w')
wn.onkeypress(paddle_a_down, 's')
```

```
wn.onkeypress(paddle_b_up, 'Up')
wn.onkeypress(paddle_b_down, 'Down')
```

5. **Turtle screen and main game loop**: The following couple of method calls represent the setup for the turtle screen: the screen size and title for the game. The `bgcolor()` method will render the background of the turtle canvas with a specified color. Here, the background of the screen will be a black color:

```
wn = turtle.Screen()
wn.title('Pong')
wn.bgcolor('black')
wn.setup(width=800, height=600)
wn.tracer(0)
```

The main game loop looks a bit trickier, but if you take a look, you will see that we have already learned about this concept. The main loop starts by setting the ball in motion. The values of `dx` and `dy` are constant units for its movement. For the part of **#border checking**, we start by checking the condition if the ball hits the upper or lower wall. If so, we reverse its direction so that the ball comes back into the gameplay. For **#2: For RIGHT boundary**, we check if the ball hit the right-hand side vertical boundary, and if so, we write the score to another player and we end the game. The same goes for the LEFT boundary:

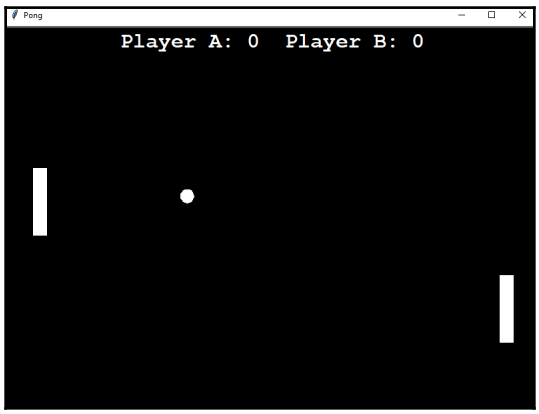
```
while True:
    wn.update()

    # Moving Ball
    ball.setx(ball.xcor() + ball.dx)
    ball.sety(ball.ycor() + ball.dy)

    # Border checking
    #1: For upper and lower boundary
    if ball.ycor() > 290 or ball.ycor() < -290:
        ball.dy *= -1
    #2: for RIGHT boundary
    if ball.xcor() > 390:
        ball.goto(0, 0)
        ball.dx *= -1
        score_a += 1
        pen.clear()
        pen.write("Player A: {}  Player B: {}".format(score_a, score_b),
          align='center', font=('Courier', 24, 'bold'))

    #3: For LEFT boundary
    if ball.xcor() < -390:
        ball.goto(0, 0)
        ball.dx *= -1
        score_b += 1
```

```
        pen.clear()
        pen.write("Player A: {}  Player B: {}".format(score_a, score_b),
          align='center', font=('Courier', 24, 'bold'))
```

Now, we have to address the condition where the ball hits the paddle of the player. The following two conditions represent the collision between the paddle and the ball: the former one is for paddle B and the latter for paddle A. Since paddle B is at the right-hand side of the screen, we check whether the ball's coordinate is the same as the paddle's coordinate, plus its width. If so, we reverse the ball's direction using the `ball.dx *= -1` command. The `setx` method will change the first coordinate of the ball to **340**, and leaves the *y* coordinate unchanged. The logic here is similar to the logic that we used while making the Snake game, when the snake's head collided with the food:

```
# Paddle and ball collisions
    if (ball.xcor() > 340 and ball.xcor() < 350) and (ball.ycor()
        < paddle_b.ycor() + 60 and ball.ycor() > paddle_b.ycor() -60):
        ball.setx(340)
        ball.dx *= -1

    if (ball.xcor() < -340 and ball.xcor() > -350) and (ball.ycor()
        < paddle_a.ycor() + 60 and ball.ycor() > paddle_a.ycor() -60):
        ball.setx(-340)
        ball.dx *= -1
```

The benefit of implementing such a rigorous modification is to not only enhance the game characters, but also to control the inconsistent frame rate—the rate at which consecutive images, **frames**, appear on our display screen. We will learn about this in detail in the upcoming chapter about *Pygame*, where we will customize the turtle-based Snake game by using our own sprites. Before summarizing this chapter, let's run the customized Pong game and observe the result, as follows:



Summary

In this chapter, we explored the world of 2D turtle graphics, along with vectored motions.

I tried to make this chapter as comprehensive as I can, especially when dealing with vectored motion. We have created two separate files; one for the `Vector` class and another for the game file itself. The `Vector` class provided a way to represent the 2D coordinates in the *x* and *y* positions. We performed multiple operations, such as *move* and *rotation* using data models—overriding its actual behavior in our custom-made `Vector` class. We briefly observed a way of working with computer pixels by creating Mario pixel art. We made a grid of pixels (list of lists) to represent the positions of the pixels, and eventually used turtle methods to render the pixel art. After that, we made a simple animation using the turtle module by defining a separate `Vector` class which represents the position of the game characters. We used the turtle module and our custom-made `Vector` class throughout the game. Although I feel that you are ready to begin your career as a 2D game programmer, as we say, *Practice makes Perfect*, and you need to experiment with it a lot before you get comfortable with it.

This chapter was a breakthrough for all of us who want to become game programmers. We learned the basics of building games with Python using the turtle module, and we learned how to handle different user events such as a mouse and a keyboard. Finally, we also learned how to create different game characters using the turtle module. As you continue to work through this book, you will find out how extremely important these concepts of turtle are, so make sure that you revise them before moving on.

In the next chapter, we will learn about the Pygame module—the most important platform for building interactive games with Python. From the next chapter onward, we will delve into topics about where you can load images or sprites and making your own game animations. You will also find out how easy it is to build games with Python in comparison to C or C++.

11
Outdo Turtle - Snake Game UI with Pygame

Python game development, in one way or another, is related to the `pygame` module. We have learned about a variety of topics and techniques regarding Python so far because we have to know about them before we progress to the `pygame` module. And all of these concepts will be used as apportioned techniques while build a game using Pygame. We can now start to use object-oriented principles, vectored movements for event handling, rotation techniques to rotate the images or sprites used in th e game, and even use things that we learned about in the turtle module. In the turtle module, we learned how to create objects (refer to `Chapter 6`, *Object-Oriented Programming*), which can be used to debug different features at the rudimentary stages of the game that we may build using Pygame. Thus, whatever we have learned so far will be used, along with the additional features of the Pygame module, which can help us make more appealing games.

In this chapter, we are going to cover multiple things, starting with learning the basics of Pygame—the installation, building blocks, and different features. After that, we are going to learn about different objects of Pygame. They are the modules that can be used for several functionalities, such as drawing shapes into the screen, handling mouse and keyboard events, loading images into the Pygame projects, and many more. At the end of this chapter, we will try to make our snake game visually attractive by adding multiple features, such as a custom-made snake image, apples as food, and a menu screen for the game. Finally, we will convert our snake game into executable files so that you can distribute your game with your friends and family and get responses from them. The following topics will be covered in this chapter:

- Pygame basics
- Pygame objects
- Initializing display and handling events
- Object rendering—making the snake game
- Menu for the game

- Converting into an executable
- Game testing and possible modifications

Technical requirements

You will need the following requirements to complete this chapter:

- Python—3.5 or higher
- PyCharm IDE—refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, for the download procedure

The files for this chapter can be found at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter11`.

Check out the following video to see the code in action:

`http://bit.ly/2o2GngQ`

Understanding pygame

Writing games with the `pygame` module requires pygame to be installed on your machine. You can download it manually from the official Pygame library by visiting the website (`www.pygame.org`.), or install it by using the Terminal with the `pip install pygame` command.

The pygame module is free to download from the aforementioned website, so we can download it by following a similar process to what we do for any other Python module. However, we can remove the headache of downloading pygame manually by using a visually more attractive and effective alternative IDE, **PyCharm**, which we downloaded back in `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*. We are became familiar with the techniques that are used to download and install third-party packages within PyCharm in that chapter.

Once you have downloaded the pygame package into PyCharm, give it some time to load. Now, we can test it by writing the following code. The following two lines of code check if the `pygame` module is downloaded or not, and if it is, it will print its version:

```
import pygame
print(pygame.version.ver) #this command will check pygame version installed
print(pygame.version.vernum) #alternate command
```

If pygame is successfully installed onto your machine, you will observe the following output. The version may vary, but at the time of writing this book, it is version 1.9.6 (the latest version of 2019). The contents of this book works for any version of `pygame` because of its backward compatibility. Make sure that you have a pygame version that is newer than 1.9+:

```
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
1.9.6
```

Pygame is a utopia for many Python game developers; it contains a surfeit amount of modules, ranging from making an interface to handling user events. All of these modules that are defined within pygame can be used independently, according to our needs. Most importantly, you can also make games using pygame, which may or may not be platform-specific. Invoking the modules of pygame is similar to invoking the methods of the class. You can always access these classes using the pygame namespace, followed by the class that you want to use. For instance, `pygame.key` will read the key that is pressed on the keyboard. Thus, the `key` class is responsible for handling keyboard actions. Similarly, the `pygame.mouse` module is used to manage mouse events. These, and many other modules of pygame, can be called independently from one another, which makes our code more manageable and readable. You can search for the list of modules that are available in the pygame module from its official documentation page, but almost 80 percent of the games require only four to six modules. If you want to learn more about them, it's always good to explore its official documentation page. Among them, we mostly use two classes in each and every game, that is, the display module, in order to access and manipulate the game display; and the mouse and key or joystick module, in order to handle input events for the game. I won't say that the others are less important, but these modules are the building blocks of games. The following table has been extracted from the Python pygame official documentation; it gives us a succinct idea about the `pygame` modules and their usages:

Module name	**Description**
`pygame.draw`	Draws shapes, lines, and points.
`pygame.event`	Deals with external events.
`pygame.font`	Deals with system fonts.
`pygame.image`	Loads the image into the project.
`pygame.joystick`	Deals with joystick movements/events.
`pygame.key`	Reads key presses from the keyboard.
`pygame.mixer`	Mixing, loading, and playing sound.
`pygame.mouse`	Reads mouse events.
`pygame.movie`	Plays/runs movie files.

`pygame.music`	Plays streaming audio files.
`pygame`	Bundled as high-level pygame functions/methods.
`pygame.rect`	Deals with rectangular areas and can create a box structure.

There are also a few more, such as surface, time, and transform. We will explore each of them in this and the upcoming chapters. All of the preceding modules are platform-independent, which means that they can be evoked, regardless of the operating system that is used by the machines. But there are some OS-specific errors, and errors due to hardware incompatibilities or improper device drivers. If any module is not compatible with any machine, the Python parser returns it as `None`, which means we can check beforehand to make sure that the game works properly. The following line of code will check whether any specified module (`pygame.module_name`) is present, and if not, it will return a custom message in the print statement, which in this case is, *No such module! Try other one*:

```
if pygame.overlay is None:
    print("No such module! Try other one")
    print("https://www.pygame.org/contribute.html")
    exit()
```

To completely grasp the concept of `pygame`, we have to make a habit of observing the code that is written by other pygame developers. In doing so, you will learn the pattern for building games with `pygame`. If, like me, you only check documentation if you are at an impasse, then we can make a simple program to help us understand the concept of `pygame` and the ways that we can call its different modules. We are going to write a simple code to illustrate this:

```
import pygame as p #abbreviating pygame with p

p.init()
screen = p.display.set_mode((400, 350)) #size of screen
finish = False

while not finish:
    for each_event in p.event.get():
        if each_event.type == p.QUIT:
            finish = True
    p.draw.rect(screen, (0, 128, 0), p.Rect(35, 35, 65, 65))
    p.display.flip()
```

Before discussing the preceding code, let's run it and observe the output. You will get a geometrical shape—a green rectangular box that will be rendered inside the screen of a certain height and width. Now, it's time to make a quick mental note of the building blocks of the `pygame` module. To make things simpler, I have listed them in the following points:

- `import pygame`: The import statement that we have been with familiar from the beginning of this book. This time, we are importing the pygame framework into our Python file.
- `pygame.init()`: This method is going to initialize a bundle of modules/classes that are embedded inside pygame. This means that we can call pygame's other modules with its namespace.
- `pygame.display.set_mode((width, height))`: The size passed as a tuple (width, height) is the desired screen size. This size represents our games console. The returned object will be a window screen, or surface, on which we will perform different graphical computations.
- `pygame.event.get()`: This statement is going to handle the event queue. The queue, as we discussed in previous chapters, is going to store different events of the user. If this statement is not called explicitly, the game will be hindered by overwhelming Windows messages, and eventually it will become unresponsive.
- `pygame.draw.rect()`: We will be able to draw into the screen using the draw module. Different shapes can be drawn with this module. More on this will be covered in the next section—*Pygame objects*. Taking a screen object, color, and position as arguments, the `rect()` method draws a rectangle. The first argument represents the screen object, which is the returned object of the display class; the second is the color code that is passed as a tuple in the form of RGB (red, green, blue) code; and the third is the dimensions of a rectangle. In order to manipulate and store rectangular areas, pygame uses `Rect` objects. `Rect()` can be created by combining four different values—height, width, left, and top.
- `pygame.QUIT`: This event is invoked whenever you explicitly close the pygame screen, which is done by pressing the `close(X)` button at the top-most right corner of the games console.
- `pygame.display.flip()`: This is same as the `update()` function, which makes any new updates on the screen visible. While making or blitting shapes or characters, this method must be invoked at the end of the game in order to ensure that all the objects are rendered properly. This will swap the pygame buffer, as pygame is a double-buffered framework.

The aforementioned code renders the green rectangular shape when executed. As we mentioned previously, the `rect()` method is responsible for creating the rectangular area, and the color code (0, 128, 0) represents the green color.

Don't get overwhelmed with this jargon; you will learn about it in detail in the upcoming chapters. While you read this chapter, make sure that you make a habit of making a logical connection between the code: something like a blueprint that maps the game from one position to another, that is, the display screen, to rendering characters, to handling events.

If you get into a situation where you are unable to close the pygame Terminal, it's surely because you haven't handled the event queue properly. In such cases, you can always stop Python from the Terminal by pressing *Ctrl + C*.

Before hopping over to the next section, I want to discuss the rather simple but abyss working of commands—pygame initialization—which is done by the `pygame.init()` statement. This is just a single line of command, but it carries out more tasks than we can imagine. As the name suggests, it is the initialization of pygame. Thus, it must initialize each of the sub-modules of the `pygame` package, that is, `display`, `rect`, `key`, and so on. Not only that, but it is also going to load all the essential drivers and queries of the hardware components in order to communicate.

If you want to load any submodules quicker, you can explicitly initialize the specific ones, and avoid all the unnecessary ones. For example, `pygame.music.init()` will only initialize the music sub-module from the bucket of submodules that is maintained by Pygame. For most of the games that we are going to cover in this book, the `pygame` module requires more than three submodules. Thus, we can use the universal `pygame.init()` method to perform initialization. After making the preceding call, we will be well-equipped to use all of the specified submodules of the `pygame` module.

After the process of initialization, it's good practice to start creating a display screen. The dimension of the display screen depends on the demand of the game. Sometimes, you may have to provide full-screen resolution to the games in order to make it fully interactive and appealing. The manipulation of the screen size can be done via the pygame surface object. The method call of `set_mode` on the display class returns the object, which represents the entire window screen. You can also set the caption to the display screen if you want; the caption will be added to the top navigation bar, which is along with the close button. The following code represents a way of adding captions or game names to the game screen:

```
pygame.display.set_caption("My First Game")
```

Now, let's talk about the argument that is passed with the `set_mode` method. The first—and an most important–argument is the dimension of the screen's surface. The size should be passed as a tuple, that is, in terms of width and height, and it is mandatory. The others are optional (in the previous program, we didn't even bother using them); they are called flags. We need them because the information related to width and height sometimes won't be enough to make an appropriate display.

We may want a **fullscreen** or **resizable** display, and in such cases, flags can be better suited for display creation. Speaking of flags, it is a feature that can be turned on and off, based on the situation, and sometimes working with it may be time-saving, relatively speaking. Let's observe some of the flags in the following table, though we are not going to use them any time soon, but covering them here obviates the unnecessary introduction in the upcoming sections:

Flag	**Purpose**
`FULLSCREEN`	Creation of a display that covers the entire screen. Windowed screen recommended for debugging.
`DOUBLEBUF`	Used in the creation of a *double-buffered* display. It is highly recommended for `HWSURFACE` or `OPENGL`, which simulates a 3D display.
`HWSURFACE`	Used in creating a hardware-accelerated display, that is, it uses video card memory instead of main memory (must be combined with the `FULLSCREEN` flag).
`RESIZABLE`	Creates a resizable display.
`NOFRAME`	Display without frame or border, along with no title bar.
`OPENGL`	Creates an OpenGL renderable display.

You can use the bitwise OR operator to combine multiple flags together, which facilitates a better experience in terms of the screen surface. In order to create a double-buffered OpenGL rendered display, you can set an optional flag parameter to `DOUBLEBUF|OPENGL`; here, (`|`) is the bitwise `OR` operator. Even if pygame is not able to render the perfect display that we asked for, which may be due to the lack of the appropriate graphics card, pygame will make a decision for us in terms of choosing a display that is compatible with our hardware.

One of the most important aspects of game development is handling a user event, and it is normally done within the game loop. Inside the main game loop, we usually have another loop to handle user events—an event loop. An event is a course of messages that inform pygame of what to expect outside the periphery of the code. Events may vary from the user pressing the key events, to any information transferred through a third-party library, for example, the internet.

The events that are created as a chunk are stored in the queue, and remain there until we explicitly address them. While there are different functions in event modules from pygame that provide a way to capture the events, `get()` is the most reliable, and is also easy to use. After getting the gamut of actions, we can address them using the pygame event handler—using functions such as `pump` or `get`. Remember that if you are only addressing specific actions, the event queue may infuse with other superficial events that you might not be interested in. Thus, the handling of events must be done explicitly using event attributes, similar to what we did in the preceding example using the `QUIT` event attribute. You may also get full access to the event object's attributes through the `eventType.__dict__` attribute. We will learn about them thoroughly in the upcoming *Event handling* section.

Before learning how to upgrade our own previously made *snake* game using pygame, we have to learn about a few important concepts of pygame—*Pygame objects*, *Drawing into the screen*, and *Handling User Events*. We will learn about these concepts in detail, one by one. We will start with *Pygame objects*, where we will learn about surface objects, creating surfaces, and rectangular objects. We will also learn how to draw shapes using pygame.

Pygame objects

The `pygame` module, which is made by internally using classes, makes code readable and reusable by allowing us to create objects and use their properties. As we mentioned earlier, there are several classes that are defined in the `pygame` module that can be called independently to perform independent tasks. For instance, the `draw` class can be used to draw different shapes such as rectangles, polygons, circles, and many more; the `event` class can call functions such as get or pump in order to handle user events. These invocations can be done using objects, by creating them first for each action. In this section, you are going to explore the concepts that will help you learn about accessing surface objects, rectangular objects, and drawing to the screen.

The most basic way to create a blank surface of customized dimensions is by calling a `Surface` constructor from the pygame namespace. While creating objects of the `Surface` class, tuples containing width and height information must be passed. The following line of code creates a blank surface of 200 by 200 pixels:

```
screen_surface = pygame.Surface((200,200))
```

We can specify a few more optional parameters that can ultimately affect the screen visuals. You can set the flag parameter to one or more of the following parameters:

- `HWSURFACE`: Creates a hardware surface. This is not very important in the context of games because it is done internally by pygame.
- `SRCALPHA`: It uses *alpha information* for the conversion of the background, which refers to a process that makes the background of the screen transparent. It creates a surface with alpha conversion. The alpha information will make a part of your surface transparent. If you are using this as an optional flag, you have to specify one more mandatory parameters, including depth, and assign its value to 32, which is standard for alpha information.

Furthermore, if you want to create a surface that contains an image as a background, you can call up the `image` class from the `pygame` module. The image class contains the `load` method, which can be called with the argument of the background image filename that needs to be rendered. The filename that is passed should be the full name, with its original extension:

```
background_surface = pygame.image.load(image_file_name.extension).convert()
```

The load function that is called from the `image` class reads an image file from your machine and then returns the surface containing an image. Here, the screen dimension will be determined by the image size. The `convert()` member function of the `Surface` object will convert the specified image into the format that is supported by your display screen.

Now, let's learn how to create multiple surfaces inside a single one, which is normally called a subsurface.

Subsurfaces

As the name suggests, subsurfaces are a list of nested surfaces inside the single main surface. The main surface can be referenced as the parent surface. The parent surface can be created with any of the aforementioned methods using the `Surface` constructor, `set_mode`, or image. When you draw onto the subsurface, it is also going to draw onto the parent one, as subsurfaces are part of the parent, too. Creating a subsurface is easy; you just need to call the `subsurface` method from the `Surface` object, and the argument that is passed should indicate the position of the `parent` class to be covered. Normally, the coordinate that is passed should create a small rectangle inside the parent screen. The following code shows how a subsurface can be created:

```
screen = Pygame.load("image.png")
screen.subsurface((0,0),(20,20))
screen.subsurface((20,0),(20,20))
```

You can store these subsurfaces into data structures such as a dictionary so that you can reference them easily. You can observe the position that is passed inside the subsurface method—they are heretic from the others. The point (0, 0) always means that a subsurface starts from the top-left corner of the parent screen.

There are several methods available with subsurfaces, all of which you can pry from its official documentation. One of the most useful methods is `get_parent()`, which returns the parent surface of the subsurface. It will return `None` if the `get_parent` method is not called with any subsurface.

Now, we will learn about the next method regarding surface objects that you will frequently use while making any game with pygame, which is `blit`, which stands for **bit block transfer**.

Blitting your objects

While the term *blitting* may not have been defined in the Oxford dictionary, it has greater significance while making games with pygame. Often referred to as bit-boundary block transfer, or Block Information Transfer, `blit` is a way of copying the image from one surface to another, usually by cropping or shifting. Let's asume you have `Surfaceb`(your screen), and you would like to draw a shape, let's say, a rectangle onto the screen. So, what you have to do is draw a rectangle and then transfer a rectangular block of the buffer to the screen buffer. This process is called *blitting*. When we cover games using pygame, you will find it being used for drawing backgrounds, fonts, characters, and everything that you can imagine.

In order to `blit` the surface, you can call the `blit` method from the resulting surface object, which is often the display object. You have to pass your source surface, such as characters, animations, and images, along with the coordinate to `blit` in as arguments. The invocation of the `blit` method is rather simple, compared to what it sounds like, theoretically. The following line of code shows how to `blit` the background image in the specified position (0,0), which is the top-most corner of the screen:

```
screen.blit(image_file_name.png, (0,0))
```

Let's say you have a collection of images that needs to be rendered, based on different frame rates. We could also do this using the `blit` method. We can change the value of the frame number and `blit` in a different area of the resulting screen to make an animation of the images. This is normally done in the case of static images. For example, we are going to create a clone of the flappy bird game, using Pygame, in the next chapter.

In that game, we have to `blit` the pipes and the bird (characters for the flappy game) on the different position, out of the static image, which we normally call sprites. These sprites are nothing but images that can be used directly from the internet, or we can make one for ourselves, according to our needs. The following code shows a simple way to `blit` images, based on the different frame rates:

```
screen.blit(list_of_images, (400, 300), (frame_number*10, 0, 100, 100))
```

In the case of the flappy bird game, a list of images contains the images of the bird in two positions: flying and falling. Based on the user events, we will render each of them using the `blit` method.

Before jumping into the next section, let's learn about the maybe paltry, but must-know topic, of *frame rates*. It is the term that is often used as the benchmark for measuring game performance. The frame rate in a video game infers the resultant simulation movements, or motions of how many times the images that you observe in the screen, are refreshed or fetched. The frame rate is a measurement that is done in **frames per second** or **FPS** (do not be confuse this with the term **first person shooter**).

There are many factors that go into determining a game's frame rate, but contemporary game players want anything but lag, or a sluggish game. Therefore, the higher rate is always better. Low frame rates may develop a hapless situation at an inopportune time. An example may be in games where users are able to jump or are chopped from a certain height; low FPS causes a lag in the system, and often makes the screen *Frozen*, which makes the user unable to interact with the game. Many modern games, for example, first-person shooter games such as Pubg and Fortnite, are developed with the intention of approaching a frame rate of around 60 FPS. But in a simple game developed by Pygame, anywhere from 15 to 30 FPS is considered acceptable. Some critics argue that anything below 30 FPS will create choppy animation and unrealistic motion, but as we know, pygame allows us to create mostly mini games. Therefore, anything between 15 to 30 FPS will be sufficient for us.

Let's hop into the next section, in which we will learn how to draw different shapes using the `pygame` draw module.

Drawing with the pygame draw module

One of the most used modules is `draw`, which has plenty of methods declared, and can be used to draw shapes onto the game screen. The aim of using this module is to draw lines, circles, and polygons—in fact, any geometrical shape. You might be wondering about the importance of using it—it has a broad range of uses. We might have to create shapes so as to perform cropping, or to blit the sprites or images onto the screen. Sometimes, you may want to use these shapes as characters for your game; games such as Tetris, which is one of the most popular games, is a perfect example. Even though you might not find it very useful while creating the games, and you would use sprites instead, it may be helpful while testing your game animation. You don't have to go anywhere to understand the importance of these shapes in game development; you can observe the games that we have created so far. Up until now, in the snake game, we have been using simple rectangular shapes to denote the snake's body and head. Although it might not be very appealing, at the incipient stage of games, we can always make games using such shapes.

Creating such shapes using pygame is easier than with any other module. We can call up the draw module, along with the function name. The function name will be the name of the shape that you want to draw. For example, for a circle, we would use `pygame.draw.circle()`, and for a rectangle, we would use: `pygame.draw.rect()`. The first two parameters for the functions in `pygame.draw` are the surface on which you want to draw, followed by the color with which you want to draw it. The first parameter for drawing the function is the `Surface` object, which represents the screen in which you want to draw. The next parameter represents the position of the screen on which you want to draw your shapes.

These three arguments are mandatory for each of the geometrical shapes, but the last one depends on the shapes. The last argument of the method represents the mathematical quantity that is used while drawing such shapes, such as the radius or diameter of a circle. Normally, the third argument that is passed should represent the coordinate position in the form of *x* and *y* coordinates, where the point (0, 0) represents the top-most left area of the screen. The following table lists the number of methods that are available inside the draw module, which can be used to draw any geometric shape:

Function	Description
`rect`	Draws a rectangle
`polygon`	Draws a regular polygon (geometrical shape that has three or more enclosed sides)
`line`	Draws a line
`lines`	Draws several lines

`circle`	Draws a circle
`ellipse`	Draws an ellipse

As an example, let's use the `circle` method and observe the `pygame` draw module in action. We need to know the value of the radius in order to draw a circle. The radius is the distance from the center of the circle to the edge of the circle, which is the arc of the circle. The arguments that should be passed while calling the circle functions are screen, which represents the surface object; the color of the circle; the position where the circle should be drawn; and finally, the radius of the circle. Since we used the random module to generate random values of the radius for the circle, instead of giving specific values, the following code creates multiple circles, with random widths in a random position, and blatantly with a random color. If you type specific values for each parameter, a shape will be drawn:

```
import pygame as game
from pygame.locals import *
from random import *
import sys

game.init()
display_screen = game.display.set_mode((650, 470), 0, 32)
while True:
    for eachEvent in game.event.get():
        if eachEvent.type == QUIT:
            sys.exit()
    circle_generate_color = (randint(0,255), randint(0,255),
                             randint(0,255))
    circle_position_arbitary = (randint(0,649), randint(0,469))
    circle_radius_arbitary = randint(1,230)
    game.draw.circle(display_screen, circle_generate_color,
    circle_position_arbitary, circle_radius_arbitary)
    game.display.update()
```

The code, which will be written from this chapter onward, is in the PyCharm Community IDE, which was downloaded in `Chapter 1`, *Getting to Know Python - Setting Up Python and the Editor*. Make sure that `pygame` is installed on the interpreter main directory so that `pygame` can be used universally on any newly created Python file.

One of the important features that can be noted while using the PyCharm IDE is that it can give us information about all the modules that come with the installation of the `pygame` module. To determine which functions reside in the `draw` module, select the `circle` or `draw` keyboard from your code and press *Ctrl + B* on the keyboard, which will, in turn, redirect you to the declaration file of the `draw` module.

While talking about the code, it is simple to understand. The main three lines of code are highlighted so that you can directly observe the importance of them. Mostly, the third line, which calls the `circle` method, is declared within the `draw` module, which takes the arguments, screen object, color, position, and radius in order to draw a circle. The output of the preceding program will print the circle with a random radius and a random color incessantly, until and unless the user closes the screen manually, which is due to the event handler, and is done by the `pygame.event.get` method.

Similarly, you can draw polygons of many shapes and sizes, which may range from a three-sided triangle to a 9999-sided polygon. Just like we have used the `pygame.draw.circle` function to create a circle, we can use `pygame.draw.polygon` to draw any kind of polygon. A call to the polygon function takes the argument in the form of a list of points, and will draw a polygon shape using these points. We can draw different geometrical shapes using a specific appellation in a similar fashion.

In the next section, we are going to learn about the different ways of initializing the display screen and handling keyboard and mouse events using the `pygame` module.

Initializing the display and handling events

Primarily, the game developer will be focusing on how to make the game more interactive by making players feel like they are engaged. The two things that must be tied as dovetail in such cases is a visually attractive display and handling the events of the player. We don't want our player to be overwhelmed with a deplorable display screen and a game that lags in movement. In this section, we are going to address the two primary things that the developer must take into account while making games: different ways of initializing the display by accommodating the available optional parameters and handling the user action events, such as when a keyboard key or a mouse button is pressed. The type of the display that you want to create depends on the type of game that you are planning to develop.

One thing that you have to remember while making games with the `pygame` module is that adding more actions to the game will affect the smoothness of the game, which means that if you add multiple features into the game, the more the game will lag in interactivity. Thus, we will primarily focus on making mini games with the `pygame` module. There are more advanced Python modules on the market that can be used for making high-feature games, and we will explore them in the upcoming chapters. For now, we will see how to initialize the display, which is done by selecting a lower resolution, because we don't want our game to lag in any way.

Any games that will be made from now on will have a fixed and low resolution, but you can experiment on your own by making the user choose their own customized display. The following code is a simple way of creating a pygame window, and we have also seen this in the previously written code:

```
displayScreen = pygame.display.set_mode((640, 480), 0, 32) #standard size
```

The first parameter of `set_mode()` will be the dimension of the screen. The value in the tuple (640, 480) represents the height and width of the screen. This dimension value will create a small window box, which is compatible with most desktop screens. We might, however, encounter a situation where a game must have a `FULLSCREEN`, instead of a small screen. In such cases, we can use an optional parameter, giving the value of `FULLSCREEN`. The code that displays the fullscreen looks something like this:

```
displayScreen = pygame.display.set_mode((640, 480), FULLSCREEN, 32)
```

We might, however, observe the performance difference between using fullscreen mode versus a customized display. While opening the game in fullscreen mode will run faster, as it doesn't interact with other background desktop screens, the other screen, with a customized display, may become incorporated with other running display screens on your machine. Apart from that, debugging games with a small display screen is easier than games with a fullscreen because you should address alternative ways of closing the game in fullscreen mode as the close button will not be visible. To check the different resolutions of display that are supported by your PC, you can call the `list_modes()` method, which will return tuples containing a list of the resolutions, which appear like this:

```
>>> import pygame as p
>>> p.init()
>>> print(p.display.list_modes())
[(1366, 768), (1360, 768), (1280, 768), (1280, 720), (1280, 600), (1024,
768), (800, 600), (640, 480), (640, 400), (512, 384), (400, 300), (320,
240), (320, 200)]
```

Sometimes, you may feel that there is slight decrease in the quality of the image that is displayed in your screen. This is primarily due to a graphics card with fewer features, which doesn't provide the color of the image that you have requested . This is compensated for by `pygame`, who converts the image into one that is appropriate for your device.

In some games, you might want the user to decide on choosing the size of the display screen. The trade-offs are concerned with whether a player chooses either high-quality visuals or making the game run smoothly. Our main goal will be to handle the event, which can toggle a screen between a resizable screen and a fullscreen. The following code illustrates a switch between a windowed screen to a fullscreen, and vice versa. When the user presses *F* on the keyboard, it will toggle between screens.

As you run the program, the toggling process between the windowed screen and the fullscreen is not spontaneous. This is because `pygame` takes some time to check the features of the graphics card, and handles the quality of images itself if the card is not capable enough:

```
import pygame as p #abbreviating pygame module as p
from pygame.locals import *
import sys
p.init()
displayScreen = p.display.set_mode((640, 480), 0, 32)

displayFullscreen = False
while True:
    for Each_event in p.event.get():
        if Each_event.type == QUIT:
            sys.exit()
        if Each_event.type == KEYDOWN:
            if Each_event.key == K_f:
                    displayFullscreen = not displayFullscreen
                    if displayFullscreen:
                        displayScreen = p.display.set_mode((640, 480),
                                        FULLSCREEN, 32)
                    else:
                        displayScreen = p.display.set_mode((640, 480), 0,
                                        32)

    p.display.update()
```

Let's learn about the display toggling process, line by line:

1. You must begin by importing the `pygame` module. The second import statement is going to import constants that are used by Pygame. However, its contents are automatically placed in the `pygame` module namespace, and we can use `pygame.locals` to include only the `pygame` constants. Examples of constants include: KEYDOWN, Keyboard `k_constants`, and so on.
2. You will set the default display mode at the start of the game. This display will be a default display, whenever you run your program for first time; the current customized display will be rendered. We have a passed display screen of (640, 480) by default.
3. To toggle the display screen, you have to make a Boolean variable, `Fullscreen`, which will be either `True` or `False`, and based on that, we will set the mode for the screen.

4. Inside the main loop, you must handle the event for keyboard key actions. Whenever the user presses *F* on the keyboard, we will change the value of the Boolean variable, and if the value of the `FULLSCREEN` variable is `True`, we have to change the display to fullscreen. The extra flag, `FULLSCREEN`, is added as a second argument to the `add_mode()` function with a depth of 32.
5. In the else part, if the value of fullscreen is `False`, you have to display the screen in the windowed version. The same key, *F*, is used to toggle the screen between the windowed and the fullscreen.

Now that we have learned how to modify the windowed visuals using the different available flags, let's hop into the next section, where we will discuss accepting user input and controlling the game, which is often referred as *handling user events*.

Handling user events

On conventional PC games, we normally see the player playing games using just the keyboard. Even today, most games fully rely on keyboard actions. With the advancement of the game industry, we can accept user input from several input devices, such as the mouse and joysticks. Often, the mouse is used to handle the action, which gives a panoramic view of the game visuals. If you have ever played counter strike, or any first person shooter game, the mouse allows the player to rotate the view in several angles, whereas keyboard actions handle the player movements, such as moving left, right, jumping, and so on. The keyboard is normally used to trigger actions such as firing and dodging, because its action is pretty much like a switch. A switch only has two possibilities: on or off; keyboard keys are also either pressed, or not, which generalizes the technique for handling the keyboard actions. In typical 19th-century games, we used to spawn the game enemy by checking the actions of the keyboard. When a user presses a keyboard key relentlessly, we used to generate the enemy in a greater quantity.

The combination of the two input devices, that is, the mouse and the keyboard, works perfectly for these games because the mouse is capable of handling directional movements and does it in a smooth manner. For instance, when you play a first-person shooter game, you rotate the player using the keyboard and the mouse. Whenever any enemy is behind you, you normally use the mouse to rotate quickly to that position, rather than using the keyboard to rotate.

In order to detect and listen to (capture) all of the keyboard keys, you have to use the `pygame.key` module. This module is capable of detecting whether any key is pressed or not, and even supports directional movements. This module is also capable of handling any keyboard actions. Basically, there are two ways of handling key presses in pygame:

- By handling KEYDOWN events, which are triggered when a key is pressed on the keyboard.
- By handling KEYUP events, which are triggered or issued when a key on the keyboard is released.

While these event handlers are a great way to check for the key presses, handling the keyboard input for movement is not appropriate with them. We need to know beforehand if the keyboard key is being pressed or not in order to draw the next frame. Thus, using the `pygame.key` module directly will give us the power to handle keyboard keys effectively. The keys of the keyboard (a-z, 0-9, and F1-F12) have key constants which are predefined by pygame. These key constants can be referred to as keycode, which is used to identify them uniquely. Keycode always starts with `K_`. For every possible key, the keycode looks something like (`K_a` to `K_z`), (`K_0` to `K_9`), and contains other constants such as `K_SPACE`, `K_LEFT`, and `K_RETURN`. Some keyboard keys cannot be handled by pygame due to hardware incompatibility. This anomaly is discussed in "Keyboards are Evil," by several developers online. You might want to refer to them to understand this in more detail.

The most basic way of handling any keyboard action is by using the `pygame.key get_pressed` function. This method is quite powerful as it assigns Boolean values to all the keyboard constants; either `True` or `False`. We can check this by using `if` conditionals: is the value of the keyboard constant `True` or `False`? If it is `True`, it is obvious that a key is being pressed. The `get_pressed` method call returns a dictionary of the key constants, where the key of the dictionary is the key constants of the keyboard and a value of the dictionary is boolean, `dictionary_name[K_a] = True`. Let's say you are making a program that will use an *up* movement as a jump button. You would have to write the following code:

```
import pygame as p
any_key_pressed = p.key.get_pressed()
if any_key_pressed[K_UP]:
    #UP key has been pressed
    jump()
```

Let's learn about the `pygame.key` module in more detail. Each of the following functions are going to handle keyboard keys, but with different approaches:

- `pygame.key.get_pressed()`: As we saw in the preceding code, this method returns a dictionary containing Boolean values for each key of the keyboard. You have to check the value of the keys to determine if it has been pressed or not. In other words, if any value for the keyboard key is set to `True`, the key for that index is said to be pressed.
- `pygame.key.name()`: As the name suggests, this method call will return the name of the pressed key. For example, if I get a `KEY_UP` event for a key that has a value of 115, you can use `key.name` to print out the name of the key, which in this case is a string, *s*.
- `pygame.key.get_mods()`: This will determine which modifier key has been pressed. Modifier keys are normal keys combined with *Shift*, *Alt*, and *Ctrl*. In order to check if any modifier key is being pressed or not, you have to call the `get_mods` method first, followed by `K_MOD`. The method call and constants are separated by a bitwise AND operator, for example, `event.key == pygame.K_RIGHT` and `pygame.key.get_mods() & pygame`. The `KMOD_LSHIFT` method can be used to check for the LEFT *Shift* key.
- `pygame.key.set_mods()`: You can also automatically set the modifier key temporarily to observe the effect of the modifier key being pressed. To set multiple modifier keys, we normally combine them using the bitwise OR operator (|). For instance, `pygame.key.set_mods(KMOD_SHIFT | KMOD_LSHIFT)` will set the SHIFT and LEFT *Shift* modifier keys.
- `pygame.key.get_focused()`: To grab every pressed key from the keyboard, the display must focus on the keyboard actions. This method call will return a Boolean value by checking if the display is receiving keyboard input from the system or not. In the case of games where there may be a customized screen, and the game screen is not focused because you may be using other application; this will return `False`, which means that the display is not active or focused to listen to keyboard actions. But in the case of a fullscreen display mode, you will be fully focused on the single screen, and in such cases, this method will always return `True`.

There are couple more pygame key functions, such as `get_repeat` and `set_repeat`, they are useful in cases where you want the repeated action to occur when you continously hold down any key on the keyboard. For instance, open your notepad and press the *s* key, continuously. You will see that the character `s` will be printed several times. This feature can be embedded using the `pygame.key set_repeat` function. This function will take two arguments: delay and interval in milliseconds.

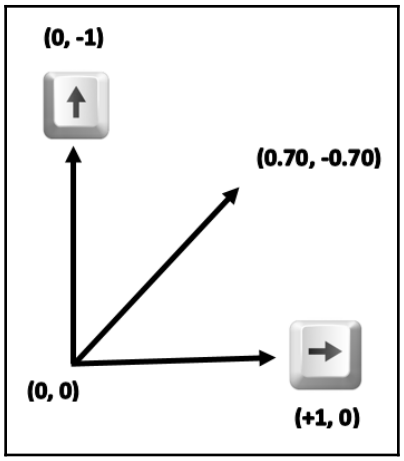
The first delay value is for the initial delay before a key repeats, while the next interval value is for the delay between repeated keys. You can disable these key-repeating features by using the `calling set_repeat` method with no parameter. By default, when pygame is initialized, the key-repeat feature is disabled. Thus, you need not have to disable it manually. Go to the following website for the pygame official documentation in order to learn more about pygame's key functions: `https://www.pygame.org/docs/ref/key.html`.

You can set a movement for a sprite/image/object of the game screen with the keyboard by assigning a key of Up, Down, Left, or Right. Up until now, we have been doing this using different modules such as Python turtle and curses. However, we were unable to handle the movement of static sprites or images. We were only handling up, down, left, right, and key events for geometrical objects, but now pygame allows us to use more intricate graphics and handle them accordingly.

We can allocate any keyboard key to perform directional movements, but following conventional methods, we can appropriate cursor keys or arrow keys as they are perfectly placed on the keyboard, which allows the player to play easily. But in some complex multiplayer games, such as first-person shooter games, the *A*, *W*, *S*, and *D* keys are allocated for directional movements. Now, you might be wondering what you have to do in order to make any arrow key behave in such a way that it can be used for directional movements. Just recall the power of vectors: the mathematical concept that is useful for game development, irrespective of whatever language or module you use. The technique for moving any geometrical shapes and images is the same; we need to create a vector that points in the direction that we might want to head in. Representing the position of a game character is quite simple: you can represent it in 2D using the (x, y) position, and in 3D using (x, y, z) position. The directional vector, however, is the unit quantity that must be added to the current vectored position in order to change to the next frame. For instance, by pressing the down key on the keyboard, we have to move downward with no change in the x position, but with a unit increment in the y coordinates. The following table explains the directional movement for four directions:

Position	**Directional vector**
Up	(0, -1)
Down	(0, 1)
Left	(-1, 0)
Right	(1, 0)

We may also want the player to allow diagonal movements, as shown in the following illustration:



The preceding illustration represents a vectored motion for the up and right keyboard keys. Suppose, at the beginning of the game, the player is at position (0, 0), which means they are at the center. Now, when the user presses the up (arrow key) keyboard key, there will be addition of (0, 0) with the up directional vectors (0, -1), and the resulting vector will be the player's new position. The diagonal movement (the combination of two keys, in this case, up and right), will give an addition of (0.707, -0.707) to the current vectored position of the player. We can use this technique of vectored motion in order to provide the directional movement to any game objects, either sprites/static images or geometrical shapes. The following code represents the vectored movement using pygame event handling techniques:

```
import pygame as p
import sys
while True:
    for anyEvent in p.event.get():
        if anyEvent.type == QUIT:
            sys.exit()
        any_keys_pressed = p.key.get_pressed()
        movement_keys = Vector2(0, 0) #Vector2 imported from gameobjects
        #movement keys are diectional (arrow) keys
        if any_keys_pressed[K_LEFT]:
            movement_keys.x = -1
        elif any_keys_pressed[K_RIGHT]:
            movement_keys.x = +1
        if any_keys_pressed[K_UP]:
            movement_keys.y = -1
        elif any_keys_pressed[K_DOWN]:
            movement_keys.y = +1
        movement_keys.normalize() #creates list comprehension
                                  [refer chapter 7]
```

Although it is worth knowing how to make things move in eight directions (four basic directions and four diagonal movements), using all eight of these won't make the game smoother. Hypothetically, it is a little artificial to make things go in eight directions. However, games nowadays allow game players to observe a view with a 360-degree facility. Thus, in order to make games with such features, instead of using eight keyboard actions, we can make rotational movements with the keys. To calculate the resultant vector from the rotation, we must calculate the sine and cosine of the angle using math modules. The sine of the angle is responsible for the movement in the *x*-component, while cosine is responsible for the movement in the *y*-component. Both of these functions take angles in radians; if the rotation angle is in degrees, you have to convert it into radians using (`degree*pi/180`):

```
resultant_x = sin(angle_of_rotational_sprite*pi/180.0)
#sin(theta) represents base rotation about x-axix
resultant_y = cos(angle_of_rotational_sprite*pi/180.0)
#cos(theta) represents height rotation about y-axis
new_heading_movement = Vector2(resultant_x, resultant_y)
new_heading_movement *= movement_direction
```

Now, let's learn about implementing mouse control and observe how it can be used in game development.

Mouse control

Having mouse control, along with keyboard control, comes in handy if you want to make games more interactive. Sometimes, handling eight directional keys is not enough, and in such cases, you also have to handle mouse events. For example, in games such as flappy bird, users have to essentially be able to play with a mouse, and although it uses screen taps in mobile games, on a PC, you have to be able to provide mouse actions. Drawing a mouse cursor into the display screen is quite simple; you need to get the coordinates of the mouse from `MOUSEMOTION` events. Similar to the keyboard `get_pressed` function, you can call up the `pygame.mouse.get_pos()` function in order to obtain the position of the mouse. Mouse movements are extremely helpful in the game—if you want to make the game characters rotate, or make a screen tap game, or even if you want to look up and down the game screen.

In order to understand the ways of handling mouse events, let's look at a simple example:

```
import pygame as game #now instead of using pygame, you can use game

game.init()
windowScreen = game.display.set_mode((300, 300))
done = False
```

```
# Draw Rect as place where mouse pointer can be clicked
RectangularPlace = game.draw.rect(windowScreen, (255, 0, 0),(150, 150, 150,
150))
game.display.update()
# Main Loop
while not done:
    # Mouse position and button clicking.
    position = game.mouse.get_pos()
    leftPressed, rightPressed, centerPressed = game.mouse.get_pressed()
    #checking if left mouse button is collided with rect place or not
    if RectangularPlace.collidepoint(position) and leftPressed:
        print("You have clicked on a rectangle")
    # Quit pygame.
    for anyEvent in game.event.get():
        if anyEvent.type == game.QUIT:
            done = True
```

I have highlighted some important parts of the code. The focus is primarily on those parts that help us understand the implementation of mouse events. Let's look at the code, line by line:

1. First of all, you have to define an object—an area that will have the mouse event listener set to capture it. In this case, you have to declare the area as a rectangle using the `pygame.draw.rect` method call.
2. Inside the main loop, you have to get the position of the mouse, which will represent the current cursor coordinates using the `pygame.mouse.get_pos()` function.
3. Then, you have to call the `get_pressed()` method from the `pygame.mouse` module. A list of Boolean values will be returned. A Boolean `True` value for LEFT, RIGHT, or CENTER means that, at a particular instance, a specific mouse button is pressed, and the remaining two are not. Here, we captured three Boolean values for three mouse buttons.
4. Now, to check if the user has pressed in the rectangle or not, you have to call the `collidepoint` method and pass a position value to it. The position represents the current cursor position. `pressed1` is going to be `True` if the mouse is clicked at the current position.
5. When both of these statements are `True`, you can perform any action accordingly. Remember that this program is not going to print a message, even if you clicked in the window screen, which is not part of the rectangle.

Similar to the `pygame.key` module, let's learn about the `pygame.mouse` module in detail. This module contains eight functions:

- `pygame.mouse.get_rel()`: It will return the relative mouse movement as a tuple, with the *x* and *y* relative movement.
- `pygame.mouse.get_pressed()`: It will return three Boolean values, which represent the mouse buttons, and if any one is `True`, the corresponding button is assumed as pressed.
- `pygame.mouse.set_cursor()`: It will set the standard cursor image. This is rarely needed since better results can be achieved by blitting an image to the mouse coordinate.
- `pygame.mouse.get_cursor()`: Two different tasks are performed: firstly, it sets the cursor standard image, and secondly, it fetches the deterministic data regarding the system cursor.
- `pygame.mouse.set_visible()`: It changes the visibility of the standard mouse cursor. If `False`, the cursor will be invisible.
- `pygame.mouse.get_pos()`: It returns a tuple containing the *x* and *y* values of the position in the canvas where the mouse is clicked.
- `pygame.mouse.set_pos()`: It will set the mouse position. It takes an argument in the form of a tuple containing the coordinates of *x* and *y* in the canvas.
- `pygame.mouse.get_focused()`: This Boolean function result is based on the condition of whether the window screen is getting input from mouse or not. It is similar to the `key.get_focused` function. When pygame is running in the current window screen, the window will get the input from the mouse, but only if the pygame window is selected and is running at the front of the display. If another program is running in the background and is selected, then the pygame window won't get an input from the mouse, and the output of this method call will be `False`.

You might have played games where you fly an airplane or destroy tanks where the mouse is used as an aiming device and the keyboard is used for movement and firing actions. These games are highly interactive. Therefore, you have to try to make a game that can combine both of these events as much as possible. These two types of events are very useful and are important for any game development I suggest that you take the time to experiment with these events. If possible, try to make your own game using only geometrical objects. Now, we are going to learn how to make a game using pygame, and our own sprites.

This game will be a modified version of the snake game that was made by the turtle module in the previous chapter. All the concepts will be same, but instead of dull and bland-looking game characters, we will make visually appealing characters, and we will handle events using pygame.

Object rendering

Computers store images in the form of grids of colors. Mostly, RGB (red, green, and blue) are enough to provide information for pixels. But apart from RGB values, there is another component of an image that is useful when dealing with pygame game development, which is alpha information (usually known as attribute components). The alpha information represents image transparency. This extra bit of information is quite useful; what usually happens in the case of pygame is that we normally draw or place one image on top of another with the alpha property activated. By doing this, we can see part of the background through it. We normally use third-party software such as GIMP in order to make an images' background transparent.

Apart from knowing how to make an images' background transparent, we have to know how to import them into our project so that we can use them. Importing any static images or sprites into the Python project is easy, and pygame makes it even easier. We have an image module, which provides a load method to import images. While calling the load method, you have to pass an image with the full filename, including the extensions. The following code represents a way of importing images into Python projects:

```
gameBackground = pygame.image.load(image_filename_for_background).convert()
Image_Cursor =
pygame.image.load(image_filename_mouseCursor).convert_alpha()
```

The image that you want to import into the game project should be in the same directory where the game project resides. For example, if the Python file is saved in the snake directory, the image that is loaded by the Python file should also be saved inside the snake directory.

In the image module, the load function will load a file from your hard drive and return a newly generated surface that contains the image that you want to load. The first call to `pygame.image.load` will read the image file, and then an immediate call to the `convert` method takes place, which will convert the image into the same format as our display. Due to the conversion of the image and the display screen being in the same depth level, drawing into the screen is relatively faster.

The second statement is to load the mouse cursor. Sometimes, you might want to load a custom-made mouse cursor into game, and a second line of code is the way to do this. In the case of loading `mouse_cursor`, `convert_alpha` is used instead of the convert function. This is because the image of the mouse cursor contains special information regarding transparency, which is termed as *alpha information*, and makes part of the image invisible to detect. By disabling the alpha information, our mouse cursor would be besieged by rectangular or square shapes, and thus would make the cursor look unprepossessing. Essentially, the alpha information is used to denote images that will have a transparent background.

Now that we have learned how to import images into the Python projects, let's learn how to rotate these images. This is an extremely useful technique because, while building games, we may have to rotate images by a certain degree in order to make the game appealing. For instance, let's say that we are making a snake game, and we are using an image for the head of the snake. Now, when the user presses *up* key on the keyboard, the head of the snake should rotate, and must move smoothly upward. This is done by the `pygame.transform` module. The `Rotate` method can be called from the transform module in order to facilitate rotation. The rotate method takes the image surface, which is loaded from the `image.load()` function and specifies the degrees by which the rotation must be done. Usually, the operation of transformation would resize, or move part of the pixel, in order to make the surface look compatible with the display screen:

```
pygame.transform.rotate(img, 270) #rotation of image by 270 degree
```

Before we begin to develop our own visually appealing snake game, you have to learn about the Pygame `time` module. Follow this link to learn more about it: `https://www.pygame.org/docs/ref/time.html#pygame.time.Clock`. The `Pygame.time` module is used for monitoring time. The time-clock also provides several functions to help control a game's frame rate. The term frame rate is the rate or frequency at which consecutive images appear on a display screen. Whenever you call the `Clock()` constructor of the time module, it will create an object, which can be used to track time. There are a variety of functions that are defined internally by Pygame developers inside the Pygame time module. However, we are only going to use the `tick` method, which will update the clock.

`Pygame.time.Clock.tick()` should be called once per frame. Between two successive calls of the function, the `tick()` method tracks the time between each call in milliseconds. By calling `Clock.tick(60)` once per frame, programs are limited to running within the boundary of 60 FPS, and cannot exceed it, even if the processing power is higher. Thus, it can be used to limit the runtime speed of the game. This is important in the case of games that are developed by Pygame because we want to run the game smoothly, instead of compensating with CPU resources. The value of frames per second (frame rate) can be anywhere from 15 to 40 in the games that are developed by Pygame.

Now, we have enough information to make our very own game using Pygame, which will have sprites and smooth movements for game characters. We will start by initializing the display in the next section. We are going to update our snake game using the Pygame module.

Initializing the display

Initializing the display is pretty basic; you can always start by importing the essential modules and providing specific dimensions of the display to the `set_mode()` method in order to create a windowed screen. Apart from that, we are going to declare a main loop. Refer to the following code to observe the declaration of the main loop:

```
import pygame as game
from sys import exit
game.init()

DisplayScreen = game.display.set_mode((850,650))
game.display.set_caption('The Snake Game') #game title

game.display.update()

gameOver = False

while not gameOver:
    for anyEvent in game.event.get():
        print(event)
        exit()

game.quit()
quit()
```

After the initialization, you can run your program to check if everything works. If you get an error saying `No pygame module`, make sure that you follow the aforementioned steps for installing Pygame on your PyCharm IDE. Now, we will learn how to work with colors.

Working with colors

The basic principle that works with computer color is *color addition*, which is a technique that will add the three primary colors in order to create a new one. The three primary colors are red, green, and blue, and often referred to as the RGB value. Whenever Pygame requires any color to be added into a game, you have to pass it in the tuple of three integers, one for each of the components referring to either red, green, or blue.

The order in which you pass the integer value to the tuple matters, with a small change being made in integer resulting in different colors. The value of each of the components of color must range from 0 to 255, where 255 represents a color having absolute intensity, and 0 represents that color having no intensity at all. For example, (255, 0, 0) represents a red color. The following table indicates the color codes for different colors:

Color name	Hex code #RRGGBB	Decimal code (R,G,B)
Black	#000000	(0,0,0)
White	#FFFFFF	(255,255,255)
Red	#FF0000	(255,0,0)
Lime	#00FF00	(0,255,0)
Blue	#0000FF	(0,0,255)
Yellow	#FFFF00	(255,255,0)
Cyan/Aqua	#00FFFF	(0,255,255)
Magenta/Fuchsia	#FF00FF	(255,0,255)

Now, let's add some color to our snake game project:

```
white = (255,255,255)
color_black = (0,0,0)
green = (0,255,0)
color_red = (255,0,0)

while not gameOver:
    #1 EVENT GET
    DisplayScreen.fill(white) #BACKGROUND WHITE
    game.display.update()
```

Now, in the next section, we will learn how to create game objects using the `pygame` module.

Making game objects

In order to begin the creation of game objects, we won't use snake sprites or images directly. Instead, we will start by using a small rectangular box, and later we will replace it with a snake image. This needs to be done in most game because we have to test multiple things at the beginning of game development, such as frame rate, collisions, rotations, and so on. After we deal with all of these, it is easy to add images to the pygame project. Thus, in this section, we will make game objects that resemble the rectangular box. We will make the head and body of the snake, which will be a small rectangular box. We will initially make one box for the head of the snake and another for the food, and then add color to it:

```
while not gameOver:
    DisplayScreen.fill(white) #background of game
    game.draw.rect(DisplayScreen, color_black, [450,300,10,10]) #1. snake
    #two ways of defining rect objects
    DisplayScreen.fill(color_red, rect=[200,200,50,50]) #2. food
```

We will now add movement to the `game` objects. We have been talking about these a lot in the previous chapters, such as while handling directional movements using vectors::

```
change_x = 300
change_y = 300
while not gameOver:
    for anyEvent in game.event.get():
        if anyEvent.type == game.QUIT:
            gameOver = True
        if anyEvent.type == game.KEYDOWN:
            if anyEvent.key == game.K_LEFT:
                change_x -= 10
            if anyEvent.key == game.K_RIGHT:
                change_x += 10

    DisplayScreen.fill(white)
    game.draw.rect(DisplayScreen, black, [change_x,change_y,10,10])
    game.display.update()
```

In the preceding code, `change_x` and `change_y` denote the initial position for the snake. Whenever start playing our game, the default position for the snake will be (`change_x`, `change_y`). By pressing either the left or the right key, we change its position.

When you run the game at this moment, you might observe that your game will move only one step, and will eventually stop when you press, and then immediately release, the keyboard key. This anomaly can be corrected by handling multiple movements. In this case, we will create `lead_x_change`, this will change according to the main `change_x` variable. Remember that, we are not handling key events for up and down; thus, `lead_y_change` is not needed:

```
lead_x_change = 0

while not gameOver:
    for anyEvent in game.event.get():
        if anyEent.type == game.QUIT:
            gameOver = True
        if anyEvent.type == game.KEYDOWN:
            if anyEvent.key == game.K_LEFT:
                lead_x_change = -10
            if anyEvent.key == game.K_RIGHT:
                lead_x_change = 10
```

```
    change_x += lead_x_change
    DisplayScreen.fill(white)
    game.draw.rect(DisplayScreen, black, [change_x,change_y,10,10])
    game.display.update()
```

Since, in the new line of code, we added extra information, `lead_x_change`, it will be called as a change in the *x* coordinates, and every time the user hits the left and right keyboard keys, the snake will move automatically. The highlighted part of the code (`change_x += lead_x_change`) is responsible for giving the snake continuous movement, even if the user doesn't press any keys (the rule of the snake game).

Now, when you press one key, you might see another unusual behavior in the game. In my case, I ran my game, and as soon as I started to press the left key, the snake began to move quickly, and continuously, from left to right. This is due to leniency in the frame rate; we now have to explicitly indicate the frame rate for the game so that it limits the runtime speed of the game. We will cover this in the next section.

Using the frame rate concept

This topic is not foreign to us; I have tried my best to introduce this topic as early as I could. We learned about the concept of frame rate while discussing the clock module, too. In this section, we will look at the concept of frame rate in action. Up until now, we have made a game that can run, but that has no restraint in its movements. It is continuously moving in one direction or another, with high speed, and we certainly don't want that. What we really want is to make the snake move continuously, but within a certain frame rate. We will use `pygame.time.Clock` to create an object, that will track the time on our game. We will use the `tick` function to update the clock. The tick method should be called once per frame. By calling `Clock.tick(15)` once per frame, the game will never run at more than 15 FPS:

```
clock = game.time.Clock()
while not gameOver:
    #event handling
    #code from preceding topic
    clock.tick(30) #FPS
```

It is important to understand that FPS is not the same as the speed of a sprite in the game. Developers make games in such a way that they can be played on both high- and low-end devices. You would see that the game is a little sluggish and jerky in a low-featured machine, but sprites or characters in both devices will move at an average speed. We are not denying that machines that use time-based motion games with slow frame rates will have a less appealing visual experience, but it won't slow down the speed of the actions.

Thus, to make a game that is visually appealing, and even compatible in pervasive devices, it is usually good practice to offer a frame rate of between 20 to 40 FPS.

In the upcoming sections, we will handle the remaining directional movements. Handling these movements is no different; they can be handled by vectored motion.

Handling directional movements

We have already handled movements for a change in the *x*-axis. Now, let's add some code that will handle movements in the *y*-axis. To make continuous movements of the snake, we have to make `lead_y_change`, which represents the directional quantity that is added continuously to the current position, even if the user doesn't press any keyboard keys:

```
lead_y_change = 0
while not gameOver:
        if anyEvent.type == game.KEYDOWN:
            if anyEvent.key == game.K_LEFT:
                lead_x_change = -10
                lead_y_change = 0
            elif anyEvent.key == game.K_RIGHT:
                lead_x_change = 10
                lead_y_change = 0
            elif anyEvent.key == game.K_UP:
                lead_y_change = -10
                lead_x_change = 0
            elif anyEvent.key == game.K_DOWN:
                lead_y_change = 10
                lead_x_change = 0

    change_x += lead_x_change
    change_y += lead_y_change
```

Now that we have handled every possible movement for the snake, let's define the boundary for the snake game. The values of `change_x` and `change_y` represent the current position of the head. If the head hits the boundary, the game will be terminated:

```
while not gameOver:
    if change_x >= 800 or change_x < 0 or change_y >= 600 or change_y < 0:
            gameOver = True
```

Now, we will learn about another concept of programming, that will make our code look cleaner. Until now, we have been using numerical values for many components, such as the height, width, FPS, and so on. But what happens if you have to change one of these values? There will be a lot of overheads in searching the code and debugging it again. Now, instead of using those numerical value directly, we can create constant variables, in which we store the values and retrieve them whenever they are needed. This process is called the *removal of hardcoding*. Let's create a variable for each of these numeric values with an appropriate name. The code should look like something like this:

```
#variable initialization step
import pygame as game

game.init()

color_white = (255,255,255)
color_black = (0,0,0)
color_red = (255,0,0)

#display size
display_width = 800
display_height = 600

DisplayScreen = game.display.set_mode((display_width,display_height))
game.display.set_caption('') #game title

gameOver = False

change_x = display_width/2
change_y = display_height/2

lead_x_change = 0
lead_y_change = 0

objectClock = game.time.Clock()

pixel_size = 10 #box size
FPS = 30 #frame rate
```

After removing the hardcoding from the variable initialization steps, we will move onto the main game loop. The following code represents the main game loop (add it after the initialization step):

```
#main loop
while not gameOver:
    for anyEvent in game.event.get():
        if anyEvent.type == game.QUIT:
            gameOver = True
```

```
        if anyEvent.type == game.KEYDOWN:
            if anyEvent.key == game.K_LEFT:
                lead_x_change = -pixel_size
                lead_y_change = 0
            elif anyEvent.key == game.K_RIGHT:
                lead_x_change = pixel_size
                lead_y_change = 0
            elif anyEvent.key == game.K_UP:
                lead_y_change = -pixel_size
                lead_x_change = 0
            elif anyEvent.key == game.K_DOWN:
                lead_y_change = pixel_size
                lead_x_change = 0

    #step 3: adding logic which will check if snake hit boundary or not
```

Now that we have added ways to handle user events inside the main loop, let's refractor the code that represents logic, such as what happens when the snake hits boundary of game, or when the snake changes its speed. The following code should be added inside the main loop after handling the user events:

```
if change_x >= display_width or change_x < 0 or change_y >= display_height
               or change_y < 0:
       gameOver = True

   change_x += lead_x_change
   change_y += lead_y_change
   DisplayScreen.fill(color_white)
   game.draw.rect(DisplayScreen, color_black,
     [change_x,change_y,pixel_size,pixel_size])
   game.display.update()

   objectClock.tick(FPS)
```

All of the preceding code has been described briefly already and what we actually did in the preceding three blocks of code is refract the variable to some meaningful names so as to remove hardcoding; for instance, adding a variable name to display the width, adding a variable name to the color code, and so on.

In the following section, we are going to add a food character to the screen, and create some logic to check if the snake has eaten the apple or not.

Adding food to the game

Adding a character to the screen is pretty simple. First of all, create a position for the character, and eventually, `blit` the character to that position. In the case of the snake game, the food must be rendered in the arbitrary position. Therefore, we will use a random module to create the random position. I have created a new function, `gameLoop()`, which will use the code from the preceding section. I have used `apple` as the food. Later, I will add an apple image to it. The following line of code defines the main loop for the game:

```
def MainLoopForGame():
    global arrow_key #to track which arrow key user pressed

    gameOver = False
    gameFinish = False
    #initial change_x and change_y represent center of screen
    #initial position for snake
    change_x = display_width/2
    change_y = display_height/2

    lead_x_change = 0
    lead_y_change = 0
```

After defining some initials for the game display and the characters, let's add some logic to add the apples (food) for the snake game (this should be inside the `MainLoopForGame` function):

```
    XpositionApple = round(random.randrange(0, display_width-pixel_size))
    YpositionApple = round(random.randrange(0, display_height-pixel_size))
```

The two lines of code will create random positions for *x* and *y*. Make sure that you import the random module.

Next up, we need to define the main game loop inside the `MainLoopForGame` function. The code that is added inside the main loop will handle multiple things, such as handling user events, drawing game characters, and so on. Let's start by getting the user events from the following code:

```
 while not gameOver:

        while gameFinish == True:
            DisplayScreen.fill(color_white)
            game.display.update()

            #game is object of pygame
            for anyEvent in game.event.get():
                if anyEvent.type == pygame.KEYDOWN:
```

```
                    if anyEvent.key == pygame.K_q:
                        gameOver = True
                        gameFinish = False
                    if anyEvent.key == pygame.K_c:
                        MainLoopForGame()
```

The preceding code will be easy to grasp, as we did this earlier in this chapter. We start by filling background screen of the game with a white color, and then we get the event using the event class of the `pygame` module. We check if the user entered the `q` key, and if they did, then we quit the game. Similarly, now that we have an event from the user, let's handle the events that make the movements for snake game—the arrow keys such as the left and right keys. The following code must be added after getting the user events:

```
#event to make movement for snake based on arrow keys
       for anyEvent in game.event.get():
           if anyEvent.type == game.QUIT:
               gameOver = True
           if anyEvent.type == game.KEYDOWN:
               if anyEvent.key == game.K_LEFT:
                   arrow_key = 'left'
                   lead_x_change = -pixel_size
                   lead_y_change = 0
               elif anyEvent.key == game.K_RIGHT:
                   arrow_key = 'right'
                   lead_x_change = pixel_size
                   lead_y_change = 0
               elif anyEvent.key == game.K_UP:
                   arrow_key = 'up'
                   lead_y_change = -pixel_size
                   lead_x_change = 0
               elif anyEvent.key == game.K_DOWN:
                   arrow_key = 'down'
                   lead_y_change = pixel_size
                   lead_x_change = 0
```

The preceding code was already written, so make sure you follow the sequence of the program. Refer to the code asset that is provided at the `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter11`. Let's add the remaining code inside the main loop, which handles the logic to render the snake's food. The following code should be added after handling the user events:

```
        if change_x >= display_width or change_x < 0 or change_y >=
                       display_height or change_y < 0:
           gameFinish = True
       change_x += lead_x_change
       change_y += lead_y_change
       DisplayScreen.fill(color_white)
```

```
            Width_Apple = 30
            game.draw.rect(DisplayScreen, color_red, [XpositionApple,
                YpositionApple, Width_Apple, Width_Apple])
            game.draw.rect(DisplayScreen, color_black,
                [change_x,change_y,pixel_size, pixel_size])
            game.display.update()

            objectClock.tick(FPS)
        game.quit()
        quit()

MainLoopForGame()
```

In the highlighted part of the code, we will draw a rectangle that is red, and render it in the position that is defined by the random modules of the height and width of `pixel_size= 10`.

Now that we have added food for the snake, let's make a function which that make the body of the snake. Up until now, we have only been working with the head of the snake; now, it's time to make a function that will increase the snake's body by unit blocks. Remember, this function is only going to be called if the snake eats the food:

```
def drawSnake(pixel_size, snakeArray):
    for eachSegment in snakeArray:
        game.draw.rect(DisplayScreen, color_green
[eachSegment[0],eachSegment[1],pixel_size, pixel_size])
```

Inside the main game loop, we have to declare multiple things. To begin with, we will declare `snakeArray`, which will contain the body of the snake. The snake length's will be one at the beginning of the game. We will increase it whenever the snake eats the food:

```
def MainLoopForGame():
 snakeArray = []
 snakeLength = 1

    while not gameOver:
        head_of_Snake = []
 #at the beginning, snake will have only head
 head_of_Snake.append(change_x)
 head_of_Snake.append(change_y)

        snakeArray.append(head_of_Snake)

        if len(snakeArray) > snakeLength:
            del snakeArray[0] #deleting overflow of elements

        for eachPart in snakeArray[:-1]:
```

```
            if eachPart == head_of_Snake:
                gameFinish = True #when snake collides with own body
        drawSnake(pixel_size, snakeArray)
        game.display.update()
```

The name of the variable tells you everything that you need to know. We have done this many times previously, that is, making lists for the snake's head and checking if it collides with the snake's body. The snake method call takes `pixel_size`, which is the snake dimension, and the snake's list, which contains a list of positions that relate to the snake's body. The snake will be `blit`, according to these lists, by drawing statements that are defined inside the `snake` function.

Next, we need to define the logic to make the snake eat the food. This logic has been repeatedly used, and it is no different in the case of pygame. Whenever the snake's head position is the same as the food position, we will increase the length of the snake by one and generate food in a new, random position. Make sure that you add the following code inside the main game loop, after updating the display:

```
#condition where snake rect is at the top of apple rect
if change_x > XpositionApple and change_x < XpositionApple + Width_Apple or
change_x + pixel_size > XpositionApple and change_x + pixel_size <
XpositionApple + Width_Apple:

      if change_y > YpositionApple and change_y < YpositionApple +
        Width_Apple:
              #generate apple to new position
              XpositionApple = round(random.randrange(0,
                               display_width-pixel_size))
              YpositionApple = round(random.randrange(0,
                               display_height-pixel_size))
              snakeLength += 1

      elif change_y + pixel_size > YpositionApple and change_y + pixel_size
            < YpositionApple + Width_Apple:

              XpositionApple = round(random.randrange(0, display_width-
                               pixel_size))
              YpositionApple = round(random.randrange(0, display_height-
                               pixel_size))
              snakeLength += 1
```

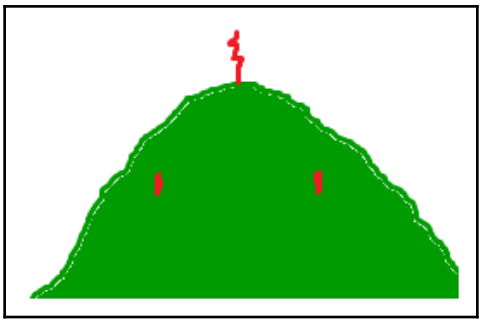
Since we are able to add some logic that will check if the snake has eaten the food or not, and respond accordingly, it's time to add a sprite or image to the characters. As we mentioned earlier, instead of using dull rectangular shapes, we are going to add our own snake head. Let's start creating one.

Adding snake sprites

Finally, we can start making our game more appealing—we are going to make the snake's head. We don't need any extra knowledge to create images for game characters. You can also download images from the internet and use them instead. However, here, I will show you how to create one for yourself, and how to use it in our snake game.

Follow these steps, line by line:

1. Open any *paint* application, or search paint in the search tab, and open the application.
2. Press *Ctrl* + *W* to resize and skew the picture that you have selected, or simply use the **resize** button on the upper menu bar. It will open a new resize window. Resizing can be done by percentage and pixels. Use percentage resize and maintain an aspect ratio of 20 by 20, that is, horizontal: 20 and vertical: 20.
3. After that, you will get a draw screen. Choose the color of the snake head that you want to make. While making the game, we created a snake body that was green; therefore, I will also choose green for the snake's head. I will use a pen and draw something like the following image. You can take your time and create an even better one if you wish. After completing it, save the file:



4. Now, you have to make the background of the image transparent. You can use several online tools too, but I am going to use GIMP software, which we have talked about before. You have to download it from its official website. It is open source, and freely available to use. Go to the website and download GIMP: `https://www.gimp.org/downloads/`.
5. Open your previously made snake head with the GIMP software. Go to the **Layer** tab from the upper-most menu, select **Transparency**, and click on **Add alpha channel**. This will add a channel, which can be used to make the background of our image transparent.

6. Click on the Color tab from the menu screen. A drop-down menu will appear. Click on **Color to Alpha** to make the background transparent. Export that file in the same directory as where your Python file is stored.

Now that we have a sprite of the snake head, let's use it and render it using the `blit` command in the Python file. As you know, before using any image, you have to import it. Since I have saved the snake head image in the same directory where the Python file is saved, I can use the `pygame.image.load` command:

```
image = game.image.load('snakehead.png')
```

Inside the body of the `drawSnake` method, you have to blit the image; something like this:

```
DisplayScreen.blit(image, (snakeArray[-1][0], snakeArray[-1][1]))
```

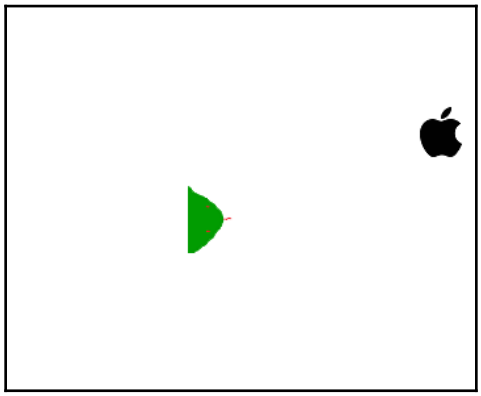
Now, when you run the game, you will observe one strange thing. As we press any one arrow key, the head won't rotate accordingly. It will remain in its default position. Thus, in order to make the sprite rotate, based on the directional movements, we have to use the `transform.rotate` function. Observe the snake method, as it has a way to `blit` images without rotation. Now, we will add couple of lines of code that will make the sprites rotate:

```
def drawSnake(pixel_size, snakeArray):

 if arrow_key == "right":
 head_of_Snake = game.transform.rotate(image, 270) #making rotation of 270

 if arrow_key== "left":
 head_of_Snake = game.transform.rotate(image, 90)

 if arrow_key== "up":
 head_of_Snake = image #default

 if arrow_key== "down":
 head_of_Snake = game.transform.rotate(image, 180)

 DisplayScreen.blit(head_of_Snake, (snakeArray[-1][0], snakeArray[-1][1]))
 for eachSegment in snakeArray[:-1]:
 game.draw.rect(DisplayScreen, color_green,[eachSegment[0],eachSegment[1],
 pixel_size, pixel_size])
```

Now, instead of using a rectangular box for the apple, let me download a sample of an apple from internet, in the form of a PNG (transparent background), and `blit` that, too:

```
appleimg = game.image.load('apple.png')
#add apple.png file in same directory of python file
while not gameOver:
    #code must be added before checking if user eats apple or not
    DisplayScreen.blit(appleimg, (XpositionApple, YpositionApple))
```

Let's run the game and observe the output. Although the snake head looks bigger, we can always resize it:



In the next section, we will learn how to add a menu to our game. The menu is a screen that is seen whenever you open a game, and it is generally a welcome screen.

Adding a menu to the game

Adding an introductory screen to any game requires us to have the knowledge of working with fonts using the `pygame` module. pygame provides a feature so that we can use different types of fonts, including a feature to change the size of them. The `pygame.font` module is used to add fonts to games. Fonts are used to add text to the game screen. Since the intro or welcome screen requires a player to show a screen containing fonts, we have to use this module. The `SysFont` method is called to add a font to the screen. The `SysFont` method takes two arguments: the first is the name of the font, and the second one is size of the font. The following line of code initializes three different sizes of the same font:

```
font_small = game.font.SysFont("comicsansms", 25)
font_medium = game.font.SysFont("comicsansms", 50)
font_large = game.font.SysFont("comicsansms", 80)
```

We will use the `text_object` function first in order to create a surface for the small, medium, and large fonts. The text object function will create a rectangular surface using the text. The text that is passed to this method is added to the box-shaped object and is returned from it, as follows:

```
def objects_text(sample_text, sample_color, sample_size):
 if sample_size == "small":
 surface_for_text = font_small.render(sample_text, True, sample_color)
 elif sample_size == "medium":
 surface_for_text= font_medium.render(sample_text, True, sample_color)
 elif sample_size == "large":
 surface_for_text = font_large.render(sample_text, True, sample_color)

 return surface_for_text, surface_for_text.get_rect()
```

Let's create a new function in the Python file, which will add a message to the screen using the aforementioned fonts:

```
def display_ScreenMessage(message, font_color, yDisplace=0,
font_size="small"):
 textSurface, textRectShape = objects_text(message, font_color, font_size)
 textRectShape.center = (display_width/ 2), (display_height/ 2) + yDisplace
 DisplaySurface.blit(textSurface, textRectShape)
```

The message to the `screen` method will create a rectangular surface to `blit` the text that is passed as a `msg` to it. The default font size is small, and the text is aligned at the center of the rectangular surface. Now, let's create a game intro method for our game:

```
def intro_for_game(): #function for adding game intro
 intro_screen = True

 while intro_screen:

 for eachEvent in game.event.get():
 if eachEvent.type == game.QUIT:
 game.quit()
 quit()

 if eachEvent.type == game.KEYDOWN:
 if eachEvent.key == game.K_c:
 intro_screen = False
 if eachEvent.key == game.K_q:
 game.quit()
 quit()

 DisplayScreen.fill(color_white)
 display_ScreenMessage("Welcome to Snake",
```

```
    color_green,
    -99,
    "large")

    display_ScreenMessage("Made by Python Programmers",
    color_black,
    50)

    display_ScreenMessage("Press C to play or Q to quit.",
    color_red,
    180)

    game.display.update()
    objectClock.tick(12)
```

This game `intro` method is called before the game `loop` method call. For example, look at the following code:

```
intro_for_game()
MainLoopForGame()
```

Finally, the output of the welcome menu should look like this:

Finally, our game is ready to be distributed. You might see that our game is a Python file with the extension of `.py`, and it cannot be executed in a machine that doesn't have Python installed. Thus, in the next section, we will learn how to convert a Python file into executables so that we can distribute our game globally on Windows machines.

Converting into executables

If you have got to the point of making your own game with pygame, it's obvious that you would like to share it with your friends and family. In the world of the internet, sharing a file is pretty easy, but problems arise when a user on the other side doesn't have Python preinstalled. It is not possible for everybody to install Python for the sole purpose of testing your game. A better idea is to make executables that can be executed on many of these machines. We will learn how to convert into `.exe` in this section—other versions (Linux and Mac) will be covered in the upcoming chapters.

The conversion of Python files into executables is easier if you use the modules that are provided by Python. There are a couple of them—`them`—`py2exe` and `cx_Freeze`. We will use the first one in this section.

Using py2exe

To convert Python files into executables, we can use another Python module, which is named `py2exe`. The `py2exe` module is not preinstalled in pygame—it's not a standard library—but it can be downloaded by using the following command:

```
pip install py2exe
OR
py -3.7 -m pip install py2exe
```

After downloading the `py2exe` module, navigate to the folder that contains your Python file. Open a Command Prompt or Terminal in that position and run code.. It will package your Python file into an `.exe` file, or into executables. The following command will search for and copy all the files that are used by the script to a folder called `dist`. Inside `dist` will be a `snake.exe` file; this file will be the output simulation of the Python code, which can be executed without Python being installed on the machine. For example, your friend might not have installed Python on their machine, but he or she can still run this file. In order to distribute the games to any other Windows machine, you can simply send the content of the `dist` folder or `snake.exe` file. Just run the following command:

```
python snake.py py2exe #conversion command
```

This will create your game with the name, *snake* and an extension of `.exe`. You can distribute these files across Windows platforms and get a response from them. Congratulations! You have finally made it. Now, let's learn about game testing using pygame.

Game testing and possible modifications

Sometimes, there may be a case of memory shortage in your machine. If you run out of memory and you try to load more images into the game, even with pygame's best efforts, this process will be aborted. `pygame.image.load` must be accompanied by some memory in order to perform tasks properly. In the case of memory shortages, you can predict that you are surely going to trigger some kind of exception. And even if there is enough memory, if you try to load an image that is not in your hard drive, or say, you made a typo when writing the name of the file, you are likely to get an exception. Therefore, it's better to handle them beforehand so that we won't have the trouble of debugging them afterward.

Secondly, let's check what happens when we provide unusual dimensions of the screen to the `set_mode` method. Recall that `set_mode`, is a method that we use to create a `Surface` object. For instance, let's say we forget to add two values to `set_mode` and we carried on adding only one. We are going to trigger an error in such cases, too:

```
screen = pygame.display.set_mode((640))
TypeError: 2 argument expected
```

Let's say that, instead of forgetting to add proper dimensions for the height and width, what happens if we add a height value of 0? This problem does not create any exception in the case of PyCharm IDE. Instead, the program will run infinitely, causing your machine to break down. However, these programs will normally throw an exception of `pygame.error: cannot set 0 sized display`. Now that you know the areas where `pygame` could go wrong, you can catch those exceptions and handle them accordingly:

```
try:
    display = pygame.display.set_mode((640,0))
except pygame.error:
    print("Not possible to create display")
    exit()
```

So, it's better to choose your display screen sensibly in order to remove any unwanted exceptions. But, more likely, you can get the exception of the `pygame` error if you try to load an image that is not in your hard drive. Thus, it's good practice to handle the exceptions so that the sprites or images for the game are loaded properly .

Summary

In this chapter, we looked at the `pygame` module and discovered the reasons for using it in game development. Most of the games that we are covering from the next chapter onwards will be somehow based on the `pygame` module. Thus, make sure that you make one simple game using pygame by yourself, before moving on.

We began by learning about how to use pygame objects to make games. We learned various things, including handling user key events that involve input devices such as the mouse and the keyboard; we made a sprite animation; we learned about color properties; and we handled different diagonal and directional movements using vectored motion. We have created our own sprites using a simple paint application, and added alpha properties using the GIMP application. We tried to make a game more interactive by incorporating an interactive game screens, that is, the menu screen. Finally, we learned how to convert Python files into executables using `py2exe` modules.

The main goal of this chapter was to make you familiar with the usage of sprites so that you can build 2D games. You have also learned how to handle user events and different movements, including diagonal movements. You also learned how to create custom sprites and images using external software, and also the ways of using them in the game. Not only that, but you were made familiar with the concepts of color and `rect` objects, and learned how to use them to make games more user-interactive, by deploying menu and score screens.

In the next chapter, we are going to use the concepts that we have learned in this chapter to make our own flappy bird clone. In addition to whatever we have learned in this chapter, we will learn about game animation, character animation, collision principles, random object generation, adding scores, and many more concepts.

12
Learning About Character Animation, Collision, and Movement

Animation is an art. This raises questions about how we can create a virtual world that imitates the physical behavior of a person or objects by adding a texture or skin to each character or by maintaining an impeccable graphical user interface. While creating animation, we do not require knowledge of how controllers or physical devices work, yet animation is a medium between the physical devices and the characters of the games. Animation instructs players by guiding them with proper shading and movements in a pictorial view, and thus it is an art. We, as programmers, are accountable for where and why game characters move in certain directions, while animators are accountable for how they look and move.

In the Python `pygame` module, we can create animation and collision using sprites—a two-dimensional image that is part of the larger graphical scene. Maybe we can make one for ourselves or download one from the internet. After loading such sprites with pygame, we are going to learn about two fundamental blocks for building games: handling user events and building animation logic. Animation logic is a simple yet powerful logic that makes sprites or images move in a particular direction that is governed by user events.

By the end of this chapter, you will be familiar with the concepts of the game controller and ways of using it to create animations for game characters. Along with this, you will also learn about collision principles and ways of dealing with them using the pygame masking method. Not only that, but you will also learn about ways to handle movements for game characters, such as jumping, tapping, and scrolling while making games such as flappy bird.

In this chapter, we are going to cover the following topics:

- Overview of game animation
- Scrolling background and character animation
- Random object generation
- Detecting collisions
- Scoring and end screen
- Game testing

Technical requirements

You will need the following list of requirements to be able to complete this chapter:

- Pygame editor (IDLE) version 3.5 or higher.
- Pycharm IDE (refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, for the installation procedure).
- The code assets and sprites for the Flappy Bird game are available in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter12`

Check out the following video to see the code in action:

`http://bit.ly/2oKQQxC`

Understanding game animation

Like just about everything you see in computer games, animation mimics the real world or tries to create one in which players can feel that they are interacting with it. Drawing a game with two-dimensional sprites is fairly simple, as we saw in the previous chapter while making the snake character for our Snake game. Even with the 2D characters, we can create three-dimensional movements with proper shading and motion. Animating single objects is easier with the `pygame` module; we saw a bit of this in action in the previous chapter when we created a simple animation for the Snake game. In this section, we are going to animate a number of objects using the `pygame` module. We will make a simple program that will create a snowfall animation. To begin, we will use some shapes to fill in the snowflakes (in this program, we are using a circular geometrical shape, but you can choose any shape) and then create some animation logic that will make the snowflake move within a milieu.

Before we write the code, make sure you brainstorm a little bit. Since we coded some advanced logic in the previous chapter too, this section might be easier for you, but make sure you learn about what we do here too as it is extremely useful for the next section, in which we will start making a clone of the Flappy Bird game.

As we know, the snowflakes animation requires a location (*x*, *y*) to render snow on it. This location can be chosen arbitrarily, and so you can use a random module to choose such locations. The following code shows how any shape can be drawn in a random location using the `pygame` module. Since a `for` loop is used for iteration, we will be using it to create a range for an iteration of at 50 calls at the most (the value of `eachSnow` ranging from 0 to 49). Recall the previous chapter, where you learned how to use pygame's `draw` module to draw any shape into the screen. Considering this, let's take a look at the following code:

```
#creates snow
for eachSnow in range(50):
     x_pos = random.randrange(0, 500)
     y_pos = random.randrange(0, 500)
     pygame.draw.circle(displayScreen, (255,255,255) , [x_pos, y_pos], 2)
#size:2
```

Imagine that we made animation using the preceding code which will, in turn, draw circular snowflakes. After running this, you will observe something odd in the output. You may have guessed this already, but let me shed some light on this. The preceding code makes a circle—in some random position—and the previously made circle vanishes as soon as the new circle is created. Instead of that, we want our code to generate numbers of snow and must make sure that the previously made circle is on the right-hand position instead of vanishing. Did you discover that the preceding code was kind of buggy? Now that you know what causes that error, take your time and think about how to solve this error. One ubiquitous idea that might occur to you is solving this using a data structure. I prefer to use lists. Let's make some modifications to the preceding code:

```
for eachSnow in range(50):
     x_pos = random.randrange(0, 500)
     y_pos = random.randrange(0, 500)
     snowArray.append([x_pos, y_pos])
```

Now, in the `snowArray` list, we have added the position of the randomly created snow at position *x* and *y*. For multiple `x_pos` and `y_pos` values of snow, a nested list will be formed. For instance, a list might look something like `[[20,40],[40,30],[30,33]]` for three randomly made circular pieces of snow.

For each of piece of snow made by using the preceding `for` loop, you have to render it using another loop. Getting the length of the `snow_list` variable might be helpful as this will give us an idea about how much snow should be drawn. For the number of positions indicated by `snow_list`, we can use the `pygame.draw` module to draw any shape, as follows:

```
for eachSnow in range(len(snowArray)):
 # Draw the snow flake
     pygame.draw.circle(displayScreen, (255,255,255) , snowArray[i], 2)
```

Can you see how easy it is to make drawings with the `pygame` module? Even though it is not alienating stuff for you, this concept will come handy in a little while. Next, we will look at how to make the snow fall downward. Follow these steps to create a downward movement for the circular snow:

1. To begin, you have to make the snow move downward with unit pixels. You only have to make changes to the `y_pos` coordinates of the `snowArray` elements, like so:

```
color_WHITE = (255, 255, 255)
for eachSnow in range(len(snowArray)):
 # Draw the snow flake
 pygame.draw.circle(displayScreen, color_WHITE, snow_Array[i], 2)
 # moving snow one step or pixel below
 snowArray[i][1] += 1
```

2. Secondly, you have to make sure that, whenever snow falls out of sight, it is created continuously. In *Step 1*, we have created a downfall for the circular snow. At some point, it is going to strike with a lower horizontal boundary. If it hits this, you have to reset it to render it from the top. By adding the following code, the circular snow will be rendered at the top of the screen using a random library:

```
if snowArray[i][1] > 500:
# Reset it just above the top
y_pos = random.randrange(-50, -10)
snowArray[i][1] = y_pos
# Give it a new x position
x_pos = random.randrange(0, 500)
snowArray[i][0] = y_pos
```

The full code for this animation is as follows (code that's written with comments is self-explanatory):

1. To begin, the preceding code that we wrote needs to be redefined and refactored so that the code looks good. Let's start by initializing it:

```
import pygame as p
import random as r
# Initialize the pygame
p.init()
color_code_black = [0, 0, 0]
color_code_white = [255, 255, 255]
# Set the height and width of the screen
DISPLAY = [500, 500]
WINDOW = p.display.set_mode(DISPLAY)
# Create an empty list to store position of snow
snowArray = []
```

2. Now, add your `for` loop, right below the initialization:

```
# Loop 50 times and add a snow flake in a random x,y position
for eachSnow in range(50):
    x_pos = r.randrange(0, 500)
    y_pos = r.randrange(0, 500)
    snowArray.append([x_pos, y_pos])
    objectClock = game.time.Clock()
```

3. Similarly, we will end the logic by creating the main loop, which *loops* until the user clicks the Close button explicitly:

```
# Loop until the user clicks the close button.
finish = False
while not finish:
     for anyEvent in p.event.get(): # User did something
         if anyEvent.type == p.QUIT: # If user clicked close
             finish = True # Flag that we are done so we
                      exit this loop
 # Set the screen background
         WINDOW.fill(BLACK)
 # Process each snow flake in the list
         for eachSnow in range(len(snowArray)):
 # Draw the snow flake
             p.draw.circle(WINDOW, color_code_white, snowArray[i], 2)
 # One step down for snow [falling of snow]
             snowArray[i][1] += 1
```

4. Finally, check if snow is within the boundary or not:

```
# checking if snow is out of boundary or not
if snowArray[i][1] > 500:
# reset if it from top
y_pos = r.randrange(-40, -10)
snowArray[i][1] = y_pos
# New random x_position
x_pos = r.randrange(0, 500)
snowArray[i][0] = x_pos
```

5. Finally, update the screen with whatever has been drawn:

```
# Update screen with what you've drawn.
    game.display.update()
    objectClock.tick(20)
#if you remove following line of code, IDLE will hang at exit
game.quit()
```

The preceding code consists of many fragments of code: the initialization of game variables, followed by creating game models. In *Step 3*, we created some simple logic that governs the animation of the game. We built two models of code in *Step 3*, which make our game interactive for the user (handling user events) and make a game object (circular snowfall) that it renders with a `for` loop. Although we are going to create more intricate animations in upcoming chapters, this is a good animation program to start with. You can clearly see that, under the hood, the creation of animations requires the use of looping, conditionals, and game objects. We use Python programming paradigms such as if-else statements, looping, arithmetic, and vectored manipulation to create game-object animations.

Apart from animating geometrical shapes, you can even animate sprites or images. To do this, you have to make your own sprites or download some from the internet. In the next section, we are going to animate sprites using the `pygame` module.

Animating sprites

Animating sprites is no different from animating geometrical shapes, but they are considered complex because you have to write extra bits of code to `blit` such images using animation logic. This animation logic, however, won't be the same for every image you load; it differs from game to game. Thus, you must analyze what type of animation is suitable for your sprites beforehand so that you can code it accordingly. In this section, we aren't going to create any custom images; instead, we will be downloading some (thanks to the internet!). We are going to embed animation logic into those sprites so that our program will facilitate adequate shading and movement.

Just to give you a flavor of how easy it is to animate static images or sprites, we are going to create a simple program that will load about 15 images of a character (moving left and right). We will `blit` (render) them whenever the user presses the LEFT or RIGHT key on their keyboard. Perform the following steps to learn how to create an animated sprite program:

1. To begin, you should start by creating a base template for the `pygame` program. You must import some important modules, create a surface for the animation console, and declare the *idle* friendly `quit()` function:

```
import pygame
pygame.init()
win = pygame.display.set_mode((500,480))
pygame.quit()
```

2. Secondly, you must load all the sprites and images listed in the *images* directory. This directory contains several sprites. You must download it and save it in the directory where your Python file is stored (the sprites/images file can be found on GitHub at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter12`):

```
#walk_Right contains images in which character is turning towards
   Right direction
walkRight = [pygame.image.load('Right1.png'),
 pygame.image.load('Right2.png'), pygame.image.load('Right3.png'),
 pygame.image.load('Right4.png'), pygame.image.load('Right5.png'),
 pygame.image.load('Right6.png'), pygame.image.load('Right7.png'),
 pygame.image.load('Right8.png'), pygame.image.load('Right9.png')]

#walk_left contains images in which character is turning towards
   left direction
walkLeft = [pygame.image.load('Left1.png'),
 pygame.image.load('Left2.png'), pygame.image.load('Left3.png'),
 pygame.image.load('Left4.png'), pygame.image.load('Left5.png'),
 pygame.image.load('Left6.png'), pygame.image.load('Left7.png'),
 pygame.image.load('Left8.png'), pygame.image.load('Left9.png')]
#Background and stand still images
background = pygame.image.load('bg.jpg')
char = pygame.image.load('standing.png')
```

3. Next, we need to declare some essential variables, such as the initial position of the character and its velocity, that is, the distance traveled per unit keystroke by the game sprites. In the following code, I have declared the velocity as five units, which suggests that the game character will move a fixed 5 pixels from the current position:

```
x = 50
y = 400
width = 40
height = 60
vel = 5
clock = pygame.time.Clock()
```

4. You have to declare a few more variables in order to track the movement of sprites based on what the user is pressing on the keyboard. If the LEFT arrow key is pressed, the `left` variable will be `True`, while if the RIGHT arrow key is pressed, the `right` variable will be `False`. The `walkCount` variable will track the number of times the key is pressed:

```
left = False
right = False
walkCount = 0
```

Here, we have completed the basic layout for any pygame program—importing appropriate modules, declaring variables to track movements, loading sprites, and so on. The two remaining parts of the program are the most important ones, so make sure you understand them. We will start by creating a main loop, as usual. This main loop will handle user events, that is, what to do when a user presses the LEFT or RIGHT key. Secondly, you have to create some animation logic, which will determine what image to `blit` at what point of time based on user events.

We will start by handling user events. Follow these steps to do so:

1. To begin, you must declare a main loop, which much be an infinite loop. We will provide **FPS** for the game using the `tick` method. As you may recall, this method should be called once per frame. It will compute how many milliseconds have passed since the previous call:

```
finish = False
while not finish: clock.tick(27)
```

2. Secondly, start by handling critical user events. In simple sprite animations, you can start by handling two basic movements: LEFT and RIGHT. In upcoming sections, we will make games by handling the JUMPING/TAPPING action. This code should be written inside a while loop:

```
while not finish:
     clock.tick(27)
     for anyEvent in pygame.event.get():
        if anyEvent.type == pygame.QUIT:
            finish = True
     keys = pygame.key.get_pressed()
    #checking key pressed and if character is at x(boundary) or not?
     if keys[pygame.K_LEFT] and x > vel:
        x -= vel #going left by 5pixels
        left = True
        right = False

    #checking RIGHT key press and is character coincides with
       RIGHT boundary.
    # value (500 - vel - width) is maximum width of screen,
       thus x should be less
     elif keys[pygame.K_RIGHT] and x < 500 - vel - width:
        x += vel #going right by 5pixels
        left = False
        right = True
     else:
        #not pressing any keys
        left = False
        right = False
        walkCount = 0
    Animation_Logic()
```

Observe the last line of the preceding code—the call to the `Animation_Logic()` function is complete. However, this method hasn't been declared yet. This method is a central block for any game that's made out of sprites or images. Code written inside the animation logic will perform two different tasks:

- Blit or render images from the list of images defined while loading the sprites. In our case, these are `walkRight`, `walkLeft`, `bg`, and `char`.
- Redraw the game window based on the logic, which will check which image to select from the pool of images. Note that `walkLeft` contains nine different images. This logic will make a selection from these images.

Now that we have handled user events, let's learn how to make animation logic for our previously loaded sprites.

Animation logic

Sprites are static images that contain characters and have a transparent background. Extra alpha information for these sprites is essential because, in 2D games, we want the user to only see the characters and not their background. Imagine a game that has a character blit with a bland background. It would leave the players with a bad impression of the game. For instance, the following sprites are Mario characters. Let's say you are making a Mario game and you crop a character from the following sprites and forget to remove its blue background. The character, along with its blue background, will be rendered in the game, making the game awful. Thus, we have to manually remove (if any) the character background using online tools or offline tools such as GIMP. An example of a sprite sheet is as follows:

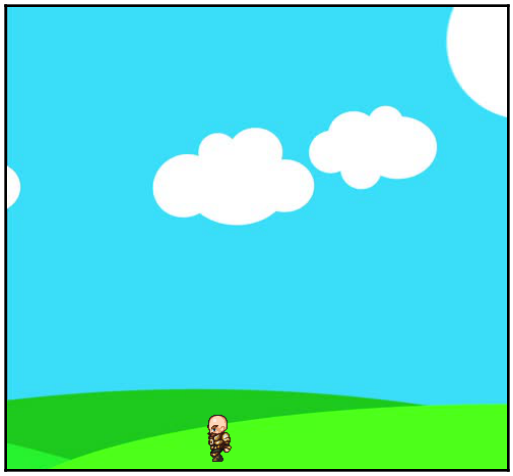
Now, let's continue with our sprite animation. Up until now, we have declared a template for handling events using `pygame`; now, let's write our animation logic. As we previously affirmed, *Animation logic is simple logic that will make a selection between the images and blit it accordingly.* Let's make that logic now:

```
def Animation_Logic():
    global walkCount
    win.blit(background, (0,0))
    #check_1
    if walkCount + 1 >= 27:
        walkCount = 0
    if left:
        win.blit(walkLeft[walkCount//3], (x,y))
        walkCount += 1
    elif right:
        win.blit(walkRight[walkCount//3], (x,y))
        walkCount += 1
    else:
        win.blit(char, (x, y))
        walkCount = 0
    pygame.display.update()
```

The first thing you will see is the `global` variable. The `walkCount` variable was initially declared inside the main loop and counts the number of times the user has pressed any keys. However, if you remove the `global walkCount` statement, you won't be able to change the value of `walkCount` inside the `Animation_Logic` function. If you only want to access or print the value of `walkCount` inside the function, you don't need to define it as global. However, if you want to manipulate its value inside a function, you must declare it as a global variable. The `blit` command is going to take two arguments: one is the sprite that needs to be rendered while the other is the position at which the sprite must be rendered onto the screen. In the preceding code, the code that's written after `#check_1` is to qualify the character whenever it reaches extreme positions. It is a check for which we have to render a *char* image, which is a still image of a character.

Rendering the sprites begins with our checking whether the left movement is active or not. If `True`, `blit` the images at the (*x*, *y*) position. The value of (*x*, *y*) is manipulated by the event handler. Whenever the user presses the LEFT arrow key, the *x* value will be decreased by five units from its previous value and the image will be rendered to it. Since this animation allows the character to move only in a horizontal direction on either the positive *X*-axis or negative *X*-axis, there is no change in the y-coordinates. Similarly, for the right movement, we are going to render images from the pool of `walkRight` at the position specified by (*x*, *y*). In the else part of the code, we `blit` a char image, which is an idle image of the character with no movements. Thus, `walkCount` is equal to zero. After we `blit` everything, we have to update it to reflect the changes. We do this by calling the `display.update` method.

Let's run the animation and observe the output:



In the console, if you press the LEFT arrow key, the character will begin moving to the left, and if you press the RIGHT arrow key the character will move to the right. Since there is no change in y coordinates and we are not handling any events in the main loop to facilitate vertical movements, the character is restricted to moving in a horizontal direction. I strongly urge you to experiment with these sprites and try handling vertical movements by changing y-coordinates. Although I have provided you with a list of resources containing a list of images, if you want to use any other sprites for your game, you can go over to the following site and download any of the sprites from there: `https://www.spriters-resource.com/`. This website is a paradise for any pygame developer, so make sure you visit it and download any game sprites you want so that you can experiment with this (Mario would be better to experiment with).

From the next section onward, we will start making a clone of the Flappy Bird game. We will learn about techniques such as a scrolling background and character animation, random object generation, collision, and scoring.

Scrolling background and character animation

Now that you know enough about pygame sprites and animation, you are capable of making a game that contains intricate sprite animations that contain multiple objects. In this section, we are going to learn about scrolling backgrounds and character animation by making a Flappy Bird game. This game contains multiple objects, with Bird being the main character for the game and a pipe pair for the obstacles in the game. If you haven't played this game before, give it a go by visiting its official website: `https://flappybird.io/`.

Speaking of the game, it isn't that hard to make, but by taking care of multiple aspects of game programming it can be an arduous task for beginners. Having said that, we aren't going to make any sprites ourselves—they are freely available on the internet. This makes our task even easier. Since the designs of the game characters are open source, we can directly focus on the coding part of the game. But if you want to design your game characters from scratch, start making them using any simple Paint application. For this Flappy Bird game, I am going to use sprites that are freely available.

I have added resources in the GitHub links. If you open the images folder and then open the background image file, you will see that it contains background images of a specific height and width. But in the Flappy Bird game, you can observe that background images are rendered continuously. Thus, using pygame, we can make a scrolling background so that we can `blit` the background image continuously. Thus, instead of using thousands of copies of the same images for the background, we can use one image and `blit` it continuously.

Let's start by making a character animation, along with a scrolling background. The following steps show us how to use object-oriented programming to make a class for each game character:

1. To begin, you must start declaring modules such as math, os (for loading images with a specified filename), random, collections, and pygame. You must also declare some variables representing the frames-per-second setting, animation speed, and the game console's height and width:

```
import math
import os
from random import randint
from collections import deque
import pygame
from pygame.locals import *

Frame_Rate = 60 #FPS
ANIMATION_SPEED = 0.18  # pixels per millisecond
WINDOW_WIDTH = 284 * 2  # Background image sprite size: 284x512 px;
                        #our screen is twice so to rendered twice: *2
WINDOW_HEIGHT = 512
```

2. Now, let's load all the images from the image folder into the Python project. I will also make two more methods that will perform the conversion between frames to milliseconds and vice versa.
3. Let's see how the `loading_Images` function works by using the following code:

```
def loading_Images():
 """Function to load images"""
 def loading_Image(image_name):

"""Return the sprites of pygame by create unique filename so that
     we can reference them"""
        new_filename = os.path.join('.', 'images', image_name)
        image = pygame.image.load(new_filename) #loading with pygame
                                                module
        image.convert()
```

```
        return image
    return {'game_background': loading_Image('background.png'),
'endPipe': loading_Image('endPipe.png'),
'bodyPipe': loading_Image('bodyPipe.png'),
# GIF format file/images are not supported by Pygame
'WingUp': loading_Image('bird-wingup.png'),
'WingDown': loading_Image('bird-wingdown.png')}
```

In the preceding program, we defined the `loading_Image` function, which loads/extracts all the images from a certain directory and returns them as a dictionary containing name as key and image as value. Let's analyze how the keys and values will be stored in such a dictionary via the following arguments:

- `background.png`: The background image for the flappy bird game.
- `img:bird-wingup.png`: This image of the flappy bird has one wing pointing upward and is rendered when the screen is tapped in the game.
- `img:bird-wingdown.png`: This part of the image is used when the flappy bird has free fall, that is, when a user is not tapping the screen. This image has the flappy bird's wing pointing downward.
- `img:bodyPipe.png`: This contains the discrete body parts that can be used to create a single pipe. For instance, in the Flappy Bird game, there should be two discrete slices of the pipe rendered from the top and bottom, leaving a gap between them.
- `img:endPipe.png`: This part of the image is the base of the pipe pair. There are two types of such images: the small pipe-end for the small pipe pair and the big pipe-end image for the larger pipe pair.

Similarly, we have a nested `loading_Image` function that creates a filename for each sprite that's being loaded. It loads images from `/images/ folder`. After loading each image successively, they are called with the `convert()` method to speed up the blitting (rendering) process. The argument that's passed to the `loading_Image` function is the filename of the image. `image_name` is the filename that's given (along with its extension; `.png` is preferred) to load it via the `os.path.join` method, along with the `convert()` method to speed up the blitting (rendering) process.

After loading the images, we need to make two functions that perform conversions of frame rates (please go to `Chapter 10`, *Upgrading the Snake Game with Turtle* to find out more about frame rates). These sets of functions primarily perform conversion from frames to milliseconds at the specified frame rates and vice versa. This conversion of frames to milliseconds is important because we have to use milliseconds for the movement of the `Bird` character, that is, the number of milliseconds left to climb, where a complete climb lasts `Bird.CLIMB_DURATION` milliseconds. Use this if you want the bird to make a (small) climb at the very beginning of the game. Let's make such two sets of functions (an exhaustive description of the code is also available on GitHub at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter12`) in the following code:

```
def frames_to_msec(frames, fps=FPS):
    """Convert frames to milliseconds at the specified framerate.

    Arguments:
    frames: How many frames to convert to milliseconds.
    fps: The framerate to use for conversion.  Default: FPS.
    """
    return 1000.0 * frames / fps


def msec_to_frames(milliseconds, fps=FPS):
    """Convert milliseconds to frames at the specified framerate.

    Arguments:
    milliseconds: How many milliseconds to convert to frames.
    fps: The framerate to use for conversion.  Default: FPS.
    """
    return fps * milliseconds / 1000.0
```

Now, declare a class for the bird character. Recall `Chapter 6`, *Object-Oriented Programming* where we learned that each entity should be represented by a single class. In the Flappy Bird game, the entity or model representing the `PipePair` (obstacles) is different from another entity, let's say, Bird. Thus, we have to make a new class to represent another entity. This class will represent the bird that will be controlled by the player. Since the bird is the "hero" of our game, any movements that are defined for the Bird character are only allowed by the user who's playing the game. The player can make the bird climb (ascend quickly) by tapping the screen; otherwise, it will sink (descend slowly). The bird must pass through the space in-between the pipe-pair, and for every pipe that's passed, one point will be rewarded. Similarly, if the bird crashes into a pipe, the game ends.

Now, we can start coding our main character. Do you remember how to do this? This is one of the most important characteristics of any good game programmer—they brainstorm too much and write small but optimized code. So, let's brainstorm and predict how we want to build the bird character beforehand so that we can code flawlessly afterward. The following are some essentials attributes and constants that must be defined as Bird class members:

- **Attributes of the class:** `x` is the bird's *X* coordinates, `y` is the bird's *Y* coordinates, and `msec_to_climb` represents the number of milliseconds left to climb, where a complete climb lasts `Bird.CLIMB_DURATION` milliseconds.
- **Constants**:
  - `WIDTH`: The width, in pixels, of the bird's image.
  - `HEIGHT`: The height, in pixels, of the bird's image.
  - `SINK_SPEED`: The speed at which, in pixels per millisecond, the bird descends in one second while not climbing.
  - `CLIMB_SPEED`: The speed at which, in pixels per millisecond, the bird ascends in one second while climbing, on average. See the `Bird.update` doc-string for more information.
  - `CLIMB_DURATION`: The number of milliseconds it takes the bird to execute a complete climb.

Now that we have enough information about the Bird character in our game, we can start writing the code for it. The following line of code represents the Bird class, which has members defined as class attributes and constants:

```
class Bird(pygame.sprite.Sprite):

    WIDTH = HEIGHT = 50
    SINK_SPEED = 0.18
    CLIMB_SPEED = 0.3
    CLIMB_DURATION = 333.3

    def __init__(self, x, y, msec_to_climb, images):
        """Initialize a new Bird instance."""

        super(Bird, self).__init__()
        self.x, self.y = x, y
        self.msec_to_climb = msec_to_climb
        self._img_wingup, self._img_wingdown = images
        self._mask_wingup = pygame.mask.from_surface(self._img_wingup)
        self._mask_wingdown = pygame.mask.from_surface(self._img_wingdown)
```

Let's talk about the constructor, or initializer, that's defined inside the Bird class. It contains many arguments that might overwhelm you, but they are rather easy to grasp. In the constructor, we normally define the attributes of the class, in this case, variables such as the x and y coordinates that represent the bird's position, as well as other arguments. Let's go over these now:

- `x`: The bird's initial *X* coordinates.
- `y`: The bird's initial *Y* coordinates.
- `msec_to_climb`: The number of milliseconds left to climb, where a complete climb lasts `Bird.CLIMB_DURATION` milliseconds. Use this if you want the bird to make a (small) climb at the very beginning of the game.
- `images`: A tuple containing the images used by this bird. It must contain the following images, in the following order:
    - Bird wing when flying up
    - Bird wing when falling down

Finally, three important properties should be declared. These properties are `image`, `mask`, and `rect`. Imagine properties are what the bird is essentially doing in the game. It can fly up and down, which is defined inside the image property. However, the other two properties of the bird class are quite different. The `rect` property will get the bird's position, height, and width as a `Pygame.Rect` (in the form of a rectangle). Remember that `pygame` can track every game character with the `rect` property, and something like an invisible rectangle will be drawn around the sprites. The mask property gets a bit-mask that can be used in collision detection with obstacles:

```
@property
def image(self):
    "Gets a surface containing this bird image"
    if pygame.time.get_ticks() % 500 >= 250:
        return self._img_wingup
    else:
        return self._img_wingdown

@property
def mask(self):
    """Get a bitmask for use in collision detection.

    The bitmask excludes all pixels in self.image with a
    transparency greater than 127."""
    if pygame.time.get_ticks() % 500 >= 250:
        return self._mask_wingup
    else:
        return self._mask_wingdown
```

```
    @property
    def rect(self):
        """Get the bird's position, width, and height, as a pygame.Rect."""
        return Rect(self.x, self.y, Bird.WIDTH, Bird.HEIGHT)
```

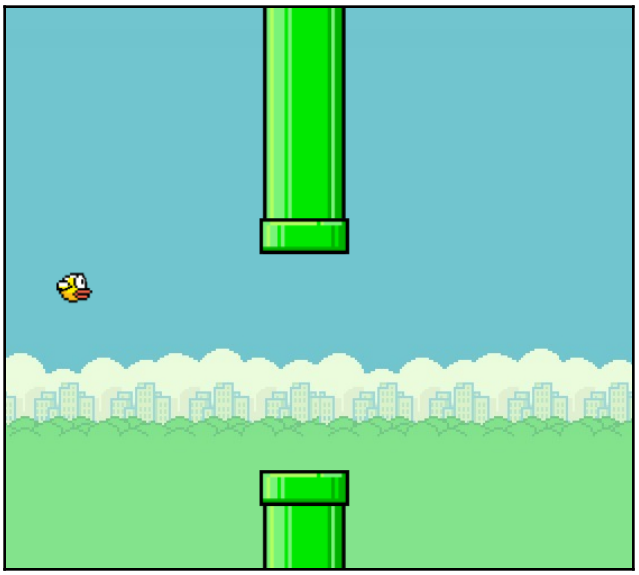
Since we are already familiar with the concepts of the `rect` and `mask` properties, I won't bother repeating myself here, so let's learn about the image property in detail. The image property gets the surface that points to the current image of a bird. This will decide whether to return an image where the bird's visible wing is pointing upward or where it is pointing downward based on `pygame.time.get_ticks()`. This will animate the Flappy Bird, even though pygame doesn't support *animated GIFs.*

The time has come to wrap up the Bird class, but before that, you have to declare one more method, which will update the bird's position. Ensure that you read the description that I've added inside the triple quote as a comment:

```
    def update(self, delta_frames=1):
        """Update the bird's position.

        One complete climb lasts CLIMB_DURATION milliseconds, during which
        the bird ascends with an average speed of CLIMB_SPEED px/ms.
        This Bird's msec_to_climb attribute will automatically be
        decreased accordingly if it was > 0 when this method was called.

        Arguments:
        delta_frames: The number of frames elapsed since this method was
            last called.
        """
        if self.msec_to_climb > 0:
            frac_climb_done = 1 - self.msec_to_climb/Bird.CLIMB_DURATION
            #logic for climb movement
            self.y -= (Bird.CLIMB_SPEED * frames_to_msec(delta_frames) *
                       (1 - math.cos(frac_climb_done * math.pi)))
            self.msec_to_climb -= frames_to_msec(delta_frames)
        else:
            self.y += Bird.SINK_SPEED * frames_to_msec(delta_frames)
```

The mathematical `cosine(angle)` function is used to make a smooth climb for the bird. Cosine is an even function, which means it is an even climb and a fall movement is given to the bird: when the bird is in the middle of the screen, a high jump can be performed, but when the bird is near the top/bottom boundary, only a slight jump can be made (this is a basic principle for the Flappy Bird's movement).

Let's run the game to check how the bird has been rendered. However, we haven't created any logic to enable the player yo play the game (we will do this soon). For now, let's run our game and observe what the interface looks like:



In the light of the preceding code, you must be able to make a complete `Bird` class that has properties for masking, updating, and getting the position, that is, the height and width, using `rect`. The bird character in our Flappy Bird game is only associated with motion—moving either up or down, vertically. The next character in our game is Pipes (obstacles for the bird), which is quite complex to deal with. We have to `blit` pipe pairs randomly and continuously. Let's see how we can do this.

Understanding random object generation

We've already covered the `Bird` character's animation in the previous sections. It consists of a list of properties and attributes that deal with the vertical motion of the bird. Since the `Bird` class is restricted to performing movements for the bird character only, we can't add any other character attributes to it. For instance, if you want to add attributes for obstacles (pipes) in the game, they can't be added to the `Bird` class. You have to create another class to define the next object. This concept is called encapsulation (we learned about this back in `Chapter 6`, *Object-Oriented Programming*) in which code and data are wrapped together within a single unit so that no other entity can harm it.

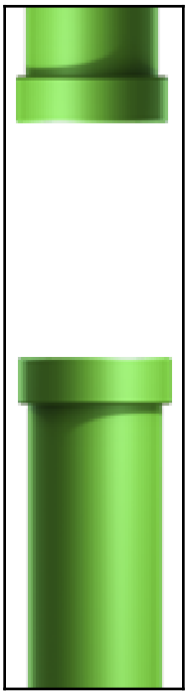
Let's make a new class to spawn obstacles for the game. You must start by defining a class, along with some constants. I have added comments along with the code so that you can understand the primary use of this class:

```
class PipePair(pygame.sprite.Sprite):
    """class that provides obstacles in the way of the bird in the form of
pipe-pair."""

    WIDTH = 80
    HEIGHT_PIECE = 32
    ADD_INTERVAL = 3000
```

Before we actually write this `PipePair` class, let me give you some pithy information about this class so that you can grasp each of the following concepts. We will use different attributes and constants, as follows:

- `PipePair` **class**: A pipe pair (a combination of two pipes) is inserted to form two pipes, and only a small gap is provided between them so that the Flappy Bird can pass through them. Whenever the bird touches or collides with any of pipe-pairs, the game will be over.
- **Attributes:** `x` is the *X*-position for `pipePair`. This value is a float to make movement smoother. There is no *Y*-position for `pipePair` as it doesn't change in the *y*-direction; it always remains 0.
    - `image`: This is the surface provided by the `pygame` module and is used to `blit` the `pipePair`.
    - `mask`: There is a bitmask that excludes all the pixels in `self.image` with a transparency greater than 127. This can be used for collision detection.
    - `top_pieces`: A combination of top-pipes along with an end-piece, which is the base for the top pieces of the pipe (this is a one-pair consisting of the top-pieces of the pipe).
    - `bottom_pieces`: A combination of down-pipes (tunnel pointing upward) with an end-piece, which is the base for the bottom pipes.
- **Constants**:
    - `WIDTH`: The width, in pixels, of a pipe piece. Because a pipe is only one-piece wide, this is also the width of a `PipePair` image.
    - `PIECE_HEIGHT`: The height, in pixels, of a pipe piece.
    - `ADD_INTERVAL`: The interval, in milliseconds, between adding new pipes.

As we already know, the first thing that we need to do for any class is the initialization of a class or constructor. This method will initialize the new random pipe pair. The following screenshot shows how the pipe pair should be rendered. There are two parts of the pipe, that is, the top and the bottom, and a small space is inserted between them:



Let's make an initializer for the `PipePair` class that will `blit` the bottom and top parts of the pipe, as well as mask it. Let's learn about the arguments that need to be initialized in this constructor:

- `end_image_pipe`: Image representing the base of the pipe (end-piece)
- `body_image_pipe`: Image representing the vertical piece of the pipe (one-slice of pipe)

The pipe pair only has an x-attribute and the y-attribute is 0. Therefore, the value of the `x` attribute is assigned as `WIN_WIDTH`, that is, `float(WIN_WIDTH - 1)`.

The following steps represent the code that needs to be added to the constructor to create a random pipe pair in the game's interface:

1. Let's initialize a new random pipe pair for `PipePair`:

```
def __init__(self, end_image_pipe, body_image_pipe):
    """Initialises a new random PipePair.
    """
    self.x = float(WINDOW_WIDTH - 1)
    self.score_counted = False
    self.image = pygame.Surface((PipePair.WIDTH, WINDOW_HEIGHT),
                 SRCALPHA)
    self.image.convert() # speeds up blitting
    self.image.fill((0, 0, 0, 0))
  #Logic 1: **create pipe-pieces**--- Explanation is provided after
              the code
     total_pipe_body_pieces = int((WINDOW_HEIGHT - # fill window from
                                                    top to bottom
     3 * Bird.HEIGHT - # make room for bird to fit through
     3 * PipePair.HEIGHT_PIECE) / # 2 end pieces + 1 body piece
     PipePair.HEIGHT_PIECE # to get number of pipe pieces
     )
     self.bottom_pipe_pieces = randint(1, total_pipe_body_pieces)
     self.top_pipe_pieces = total_pipe_body_pieces -
                            self.bottom_pieces
```

2. Next, we need to define two types of pipe pair—the bottom pipe and the top pipe. The code that adds the pipe pair blits the pipe image and only cares about the *y*-position for the pipe pair. No horizontal coordinates are required for pipe pairs (they should be rendered vertically):

```
# bottom pipe
   for i in range(1, self.bottom_pipe_pieces + 1):
       piece_pos = (0, WIN_HEIGHT - i*PipePair.PIECE_HEIGHT)
       self.image.blit(body_image_pipe, piece_pos)
   end_y_bottom_pipe = WIN_HEIGHT - self.bottom_height_px
   bottom_end_piece_pos = (0, end_y_bottom_pipe -
                           PipePair.PIECE_HEIGHT)
   self.image.blit(end_image_pipe, bottom_end_piece_pos)

   # top pipe
   for i in range(self.top_pipe_pieces):
       self.image.blit(body_image_pipe, (0, i *
            PipePair.PIECE_HEIGHT))
   end_y_top_pipe = self.top_height_px
   self.image.blit(end_image_pipe, (0, end_y_top_pipe))
```

```
        # external end pieces are further added to make compensation
        self.top_pipe_pieces += 1
        self.bottom_pipe_pieces += 1

        # for collision detection
        self.mask = pygame.mask.from_surface(self.image)
```

Although the comments that were provided alongside the code are helpful when it comes to understanding the code, we need to learn about the logic in a more pithy way. The `total_pipe_body_piece` variable stores the height for the number of pipe pieces that can be added in one frame. For example, it infers the number of bottom pipes and top pipes that can be inserted into the current instance. We typecast it to the integer since pipe pairs will be always integers. The `bottom_pipe_piece` class attribute represents the height of the bottom pipe. It may range anywhere from 1 to the maximum width supported by `total_pipe_piece`. Similarly, the height of the top pipe piece depends on the total pipe piece. For example, if the total height of the canvas is 10 and the height of the bottom pipe is 1, then by leaving a gap between the two pipe pairs (let's say, 3), the remaining height should be that of the top pipe (that is, its height is 10 - (3+1) = 6) which means that, except from the gap between the pipe pair, no other gap must be provided.

Everything that is written in the preceding code is self-explanatory. Although the code is simple, I want you to focus on the last line of code, which we used to detect a collision. This process of detection is significant because, in the Flappy Bird game, we have to check if the bird is colliding with the pipe pair or not. This is usually done by adding a mask using the `pygame.mask` module.

Now, it's time to a add few properties to the `PipePair` class. We will add four properties: `visible`, `rect`, `height_topPipe_px`, and `height_bottomPipe_px`. The `rect` property works similarly to the `Bird` class' `rect` call—it returns the rectangle that contains the `PipePair`. The `visible` property of the class checks if the pipe pair is visible in the screen or not. The two other properties return the top and bottom pipe's height in pixels. The following is the code for the preceding four properties of the `PipePair` class:

```
    @property
    def height_topPipe_px(self):
     """returns the height of the top pipe, measurement is done in pixels"""
     return (self.top_pipe_pieces * PipePair.HEIGHT_PIECE)

    @property
    def height_bottomPipe_px(self):
     """returns the height of the bottom pipe, measurement is done in pixels"""
     return (self.bottom_pipe_pieces * PipePair.HEIGHT_PIECE)

    @property
```

```
    def visible(self):
        """Get whether this PipePair on screen, visible to the player."""
        return -PipePair.WIDTH < self.x < WINDOW_WIDTH

    @property
    def rect(self):
        """Get the Rect which contains this PipePair."""
        return Rect(self.x, 0, PipePair.WIDTH, PipePair.HEIGHT_PIECE)
```

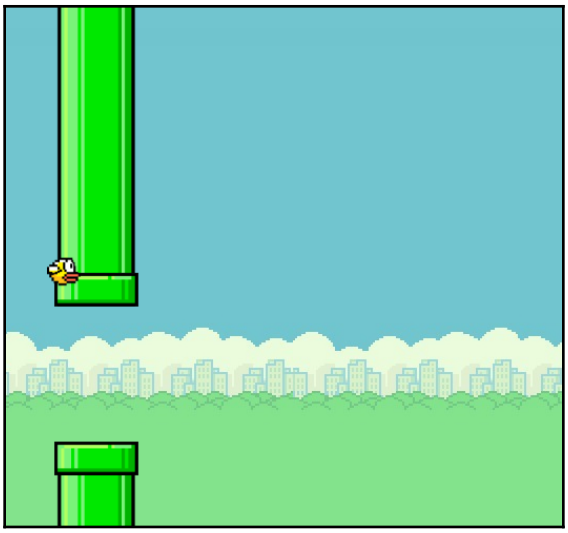
Now, it's time to add two more methods to the `PipePair` class before wrapping it. The first method, `collides_with`, is going to check whether the bird collides with a pipe in the pipe pair or not:

```
    def collides_with(self, bird):
        """check whether bird collides with any pipe in the pipe-pair. The
           collide-mask deploy a method which returns a list of sprites--in
           this case images of bird--which collides or intersect with
           another sprites (pipe-pair)

        Arguments:
        bird: The Bird which should be tested for collision with this
            PipePair.
        """
        return pygame.sprite.collide_mask(self, bird)
```

The second method, `update`, will update the pipe pair's positions:

```
    def update(self, delta_frames=1):
        """Update the PipePair's position.

        Arguments:
        delta_frames: The number of frames elapsed since this method was
            last called.
        """
        self.x -= ANIMATION_SPEED * frames_to_msec(delta_frames)
```

Now that we know how every method works, let's see the code in action. You won't understand any flaws in your game until you run it. Take the time to run your game and observe the output:



Okay, so the game is appealing enough to play. The tapping events are working perfectly, and the background image is rendered along with the bird images and the physics for climbing and sinking actions. However, one strange thing you might have observed (if not, take a look at the preceding screenshot), is that, after colliding with the pipe pair, our bird was able to move forward. This is a big flaw in our game, and we don't want it. Instead, we want to close the game when this happens. Thus, to overcome such an error, we have to use concepts of collision (a technique that handles the event when multiple game objects collide with each other).

Now that we have completed the two game character classes, that is, `Bird` and `PipePair`, let's move toward making the physical part of the game: initializing the display and handling collisions.

Detecting collision

The process of *handling collisions* is done by figuring out what actions must be performed when two independent objects touch each other. In the preceding section, we added a mask for each object to check whether two objects collide or not. The `pygame` module makes checking the process of collisions extremely easy; we can simply use `sprite.collide_mask` to check if two objects are touching or not. However, the argument that this method takes is the masking object. In the previous section, we added the `collides_with` method to check if the bird collides with one of the pipes in the pipe pairs or not. Now, let's use that method to check for collision.

Along with detecting collisions, we will make a physical layout/template for the game. I am not emphasizing the basic pygame layout in this section because it should be self-explanatory for you since we have been doing this for a long time now. The following steps depict the layout for making a model that detects game characters collisions (`Bird` with `pipePairs`):

1. Start by defining the main function, which will be externally called afterward:

```
def main():
    """Only function that will be externally called, this
      is main function

    Instead of importing externally, if we call this function from
      if name == __main__(), this main module will be executed.
    """

    pygame.init()

    display_surface = pygame.display.set_mode((WIN_WIDTH,
        WIN_HEIGHT)) #display for screen
    objectClock = pygame.time.Clock()
    images = loading_Images()
```

2. Let's create some logic that will make the bird appear at the center of the screen. If you have played the Flappy Bird game, you will know that the bird is placed at the center of the canvas and that it can move either vertically upward or downward:

```
 #at any moment of game, bird can only change its y position,
   so x is constant
    #lets put bird at center
    Objectbird = Bird(50, int(WIN_HEIGHT/2 - Bird.HEIGHT/2), 2,
                (images['WingUp'], images['WingDown']))

    pipes = deque()
#deque is similar to list which is preferred otherwise
   if we need faster operations like
#append and pop
    frame_clock = 0  # this counter is only incremented
      if the game isn't paused
```

3. Now, we have to add pipe pair images to the `pipes` variable since a pipe is formed by concatenating `pipe-body` with `pipe-end`. This concatenation is done inside the `PipePair` class, so that, after creating the instances, we can append the pipe pair to the pipes list:

```
done = paused = False
  while not done:
      clock.tick(FPS)
      # Handle this 'manually'.
        If we used pygame.time.set_timer(),
      # pipe addition would be messed up when paused.
      if not (paused or frame_clock %
        msec_to_frames(PipePair.ADD_INTERVAL)):
          pipe_pair = PipePair(images['endPipe'],
            images['bodyPipe'])
          pipes.append(pipe_pair)
```

4. Now, handle the user's actions. Since the Flappy Bird game is a tapped game, we will handle mouse events (refer to the *Mouse control* section we covered in `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*):

```
#handling events
    #Since Flappy Bird is Tapped game
    #we will handle mouse events
    for anyEvent in pygame.event.get():
        #EXIT GAME IF QUIT IS PRESSED
        if anyEvent.type == QUIT or (anyEvent.type == KEYUP and
          anyEvent.key == K_ESCAPE):
            done = True
            break
       elif anyEvent.type == KEYUP and anyEvent.key in
        (K_PAUSE, K_p): paused = not paused
       elif anyEvent.type == MOUSEBUTTONUP or
          (anyEvent.type == KEYUP and anyEvent.key in
          (K_UP, K_RETURN, K_SPACE)): bird.msec_to_climb =
          Bird.CLIMB_DURATION
     if paused:
        continue #not doing anything [halt position]
```

5. Finally, here's what you've been waiting for: how to build a collision interface with the help of Python's `pygame` module. The highlighted part of the following code will be discussed in detail after we've completed the rest of these steps:

```
# check for collisions
      pipe_collision = any(eachPipe.collides_with(bird)
        for eachPipe in pipes)
      if pipe_collision or 0 >= bird.y or
```

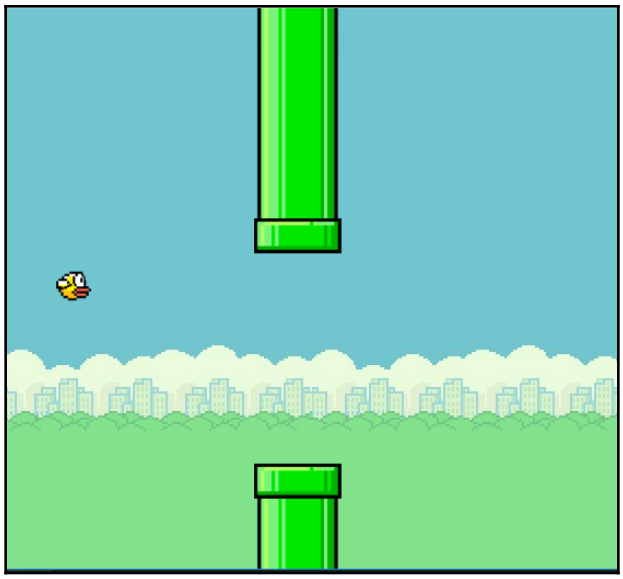
```
        bird.y >= WIN_HEIGHT - Bird.HEIGHT:
          done = True

      #blit background
      for position_x_coord in (0, WIN_WIDTH / 2):
          display_surface.blit(images['game_background'],
            (position_x_coord, 0))
      #pipes that are out of visible, remove them
      while pipes and not pipes[0].visible:
          pipes.popleft()

      for p in pipes:
          p.update()
          display_surface.blit(p.image, p.rect)

      bird.update()
      display_surface.blit(bird.image, bird.rect)
```

6. Finally, end the program with some superfluous steps, such as rendering the game with an update function, giving an extraneous message to the user, and so on:

```
        pygame.display.flip()
        frame_clock += 1
    print('Game Over!')
    pygame.quit()
#----------uptill here add it to main function----------
if __name__ == '__main__':
  #indicates two things:
  #In case other program import this file, then value of
     __name__ will be flappybird
  #if we run this program by double clicking filename
     (flappybird.py), main will be called
    main()     #calling main function
```

The highlighted parts in the preceding code are important, so ensure that you understand them. Here, the `any()` function returns a Boolean by checking whether the bird collides with the pipe pair or not. Based on that check, if it is `True`, we exit the game. We will also check whether the bird is touching the lowest horizontal or upper horizontal boundary or not and exit from the game if it is.

Let's run the game and observe the output:



The game is playable enough, so let's add one more feature to the game that tells the player how well they're scoring.

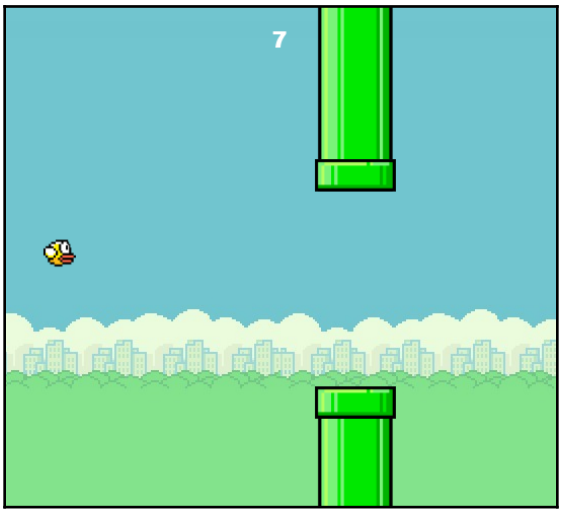
Scoring and end screen

Adding a score to the Flappy Bird game is quite simple. The player's score will be the number of pipes or obstacles a player has passed through. If the player passes through 20 pipes—their score will be 20. Let's add a score screen to the game:

```
score = 0
scoreFont = pygame.font.SysFont(None, 30, bold=True) #Score default font:
WHITE

while not done:
    #after check for collision
    # procedure for displaying and updating scores of player
     for eachPipe in pipes:
         if eachPipe.x + PipePair.WIDTH < bird.x and not
           eachPipe.score_counted:
            #when bird crosses each pipe
             score += 1
             eachPipe.score_counted = True

     Surface_Score = scoreFont.render(str(score),
```

```
            True, (255, 255, 255)) #surface
        x_score_dim = WIN_WIDTH/2 - score_surface.get_width()/2
        #to render score, no y-position
        display_surface.blit(Surface_Score, (x_score_dim,
            PipePair.HEIGHT_PIECE)) #rendering

        pygame.display.flip() #update
        frame_clock += 1
print('Game over! Score: %i' % score)
pygame.quit()
```

Now, the game looks more appealing:



In the next section, we'll look at how we can test everything out and even try to apply some modifications.

Game testing

Although there are fewer areas where Flappy Bird can be modified, you can always test the game by modifying some game character attributes in order to change the difficulty of the game. In the previous section, we ran our game and saw that there was a huge space between a pipe pair. This will make the game extremely easy for many users to play, and so we need to increase the difficulty by narrowing down the space between the two pipe pairs. For instance, inside the `Bird` class, we have declared four attributes. Change them to different values to observe the effect:

```
WIDTH = HEIGHT = 30 #change it to make space between pipe pairs
                     smaller/bigger
SINK_SPEED = 0.18 #speed at which bird falls
CLIMB_SPEED = 0.3 #when user taps on screen, it is climb speed
                  #make it smaller to make game harder
CLIMB_DURATION = 333.3
```

You can also vary the values of game attributes to give your game a unique look. Some of the different game attributes that are used in Flappy Bird are *frames per second* and *animation speed*. You can alter these values to implement the necessary changes. Although you can change the value of the animation speed, a value of 60 for FPS is adequate for the Flappy Bird game.

Instead of debugging and searching for possible modifications manually, you can simply run your program in debug mode to test it faster. Assuming that you have coded the Flappy Bird game in Pycharm's IDE (I recommend this), you can run your program in debug mode by pressing *Shift + F9* or simply clicking on the **Run** tab and running it in debug mode from there. After you run it, try to play the game and try to make it fit every possible situation that the user might encounter. Any errors will be located in the Terminal of the program, from which you can jump to a location in the program that has multiple errors.

Summary

In this chapter, we explored the concepts of sprite animations and collision more deeply. We looked at how to make a simple animation for geometrical shapes and create complex sprite animations, and learned which works best in certain scenarios. We combined pygame's event handling method with animation logic, which renders the images based on the current game state. Essentially, animation logic maintains a queue in which user events will be stored. Fetching one action at a time renders the image to a position.

Game prototypes that are made by using pygame have three central blocks: loading ant sprites (the original sprites or those downloaded from the internet), handling user events, and animation logic, which governs the movement of game characters. Sometimes, instead of having independent sprite images, you might have sprite sheets—sheets containing images of characters. You can crop them by using online tools or even pygame's `rect` method. After getting the proper images or sprites for the game, we handled user events and created animation logic to make the game sprites move. We also looked at pygame's masking properties, which can be used to detect collisions between objects.

After completing this chapter, you now understand game controllers and animation, have learned about the collision principle (including pygame's masking property), have learned about sprite animations (creating a running animation of a character), and have learned about adding an interactive score screen to make the game more user-friendly.

The wide range of areas where you can apply the knowledge you have gained in this chapter is *pure gold* to most Python pygame developers. Handling sprites is important for almost all pygame-based games. Although simple yet powerful concepts, character animation, collision, and movements are three primary aspects of Python games that make them appealing and interactive. Now, experiment by creating a simple **role-playing game** (**RPG**) game such as Junction Jam (if you haven't heard of it, Google it) and try to embed the concepts of collision and sprite movement in it.

In the next chapter, we are going to learn about pygame's primitive graphics programming by creating game grids and shapes. We will learn about multi-dimensional list processing and valid space determination by coding a Tetris game.

13
Coding the Tetris Game with Pygame

Think out of the box, an old adage that for the game developer might sound cliche, but is still very applicable. Most of the games that have revolutionized the gaming industry contain some unique elements and represent the taste of general audiences. But this worldwide assumption overestimates by discarding approaches that might be common among most game developers. After all, mathematical paradigms, object-rendering tools, and software remain the same. Thus, in this chapter, we are going to explore some of the advanced mathematical transformations and paradigms that every game programmer must know.

In this chapter, we will learn how to create one of the most played and downloaded games of the century that's very recognizable among 90s kids—*Tetris*. We will learn how to create it from scratch by building shapes that have been formatted from multi-dimensional lists. We will learn how to draw primitives and game grids, which will help us to locate the game objects. We will also learn how to implement rotational transformations of geometrical shapes and figures. Although this concept might sound simple, the application of these concepts ranges from different 2D to 3D **role-playing games** (**RPGs**).

By the end of this chapter, you will be familiar with different concepts such as creating grid (virtual and physical) structures to locate game objects based on the position and color code. Then, you will learn about multi-dimensional list processing by using list comprehension. Furthermore, readers will also learn about the different shifting transformations and collision-checking principles. In the previous chapter, we implemented collision checks with the help of masking using pygame. However, in this chapter, we will do it in a programmer's way—it may be little complicated but it contains a profuse amount of knowledge.

In this chapter, we are going to cover the following topics:

- Understanding Tetris essentials
- Creating a grid and random shapes
- Setting up the windows and game loops
- Converting the shape format
- Modifying the game loop
- Clearing the rows
- Game testing

Technical requirements

You will need the following requirements in order to complete this chapter:

- Pygame editor (IDLE)—version 3.5+ is recommended.
- PyCharm IDE—refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, for the installation procedure.
- The Code assets for the Tetris game can be found on GitHub at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter13`

Check out the following video to see the code in action:

`http://bit.ly/2oDbq2J`

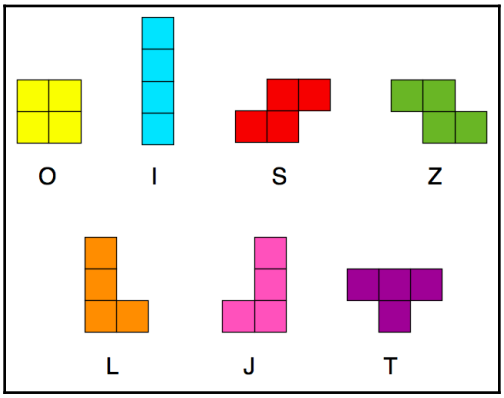
Understanding Tetris essentials

Incorporating pygame sprites and images into our Python game is a straightforward process. It requires a built-in Python module—*os*—that will load files from your machine. In the previous chapter, while building the Flappy Bird game, we learned how to make rotations, translations, and collisions of the sprites, and dealt with them one by one. Such transformations are not merely applied to images, but also to different geometrical figures and shapes. Tetris is a game that comes to everyone's mind when we talk about using such transformation operations—where a player is allowed to change the shape and size of the geometrical shapes through periodic motion. This periodic motion will create a realistic rotational transformation of the geometrical shapes, in both anticlockwise and clockwise directions. For those who are not familiar with Tetris, check out `https://www.freetetris.org/game.php` and observe the grid and the environment of the gameplay.

By observing the environment of the gameplay, you will notice three primary things:

- **Geometrical shapes, such as L, T, S, I, and square**: These geometrical shapes will be presented in the form of alphabetical characters, and to distinguish between them, each shape will have different colors.
- **Grid**: This will be the place where the geometrical shapes can move. This will be the game canvas, where geometrical shapes will fall from the top to the bottom. The player cannot control this grid, but they can control the shapes.
- **Rotate the shapes**: As shapes/blocks will be falling downwards, players can use the arrow keys from the keyboard in order to alter the structure of the shapes (remember that only rotation transformation is allowed).

The following diagram shows the shapes that we will be using for our game:



If you've played the game in the aforementioned link, you will have seen that the preceding shapes move within the grid (canvas) of the game. The respective letters represent each the geometrical shape they resemble. Players can only use the arrow key to rotate such shapes. For instance, when shape **I** is falling to the grid, players can switch between a vertical **I** and a horizontal **I**. But in the case of the square shapes, we do not have to define any rotations since the square (due to its equal sides) looks exactly the same after a rotation.

Now that you are familiar with the game characters for our Tetris game (geometrical shapes), let's brainstorm further in order to extract some critical information about the game. Let's talk about the essentials of Tetris. Since Tetris requires the creation of different geometrical shapes, it is undoubtedly true that we will require the `pygame` module. The `pygame` module can be used to create grids, borders, and game characters. Do you remember the `draw` module (from `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*) of pygame? Obviously, you cannot make good games without using the `pygame draw` module. Similarly, to handle user action events such as keyboard actions, we need pygame.

The blueprint of functions represents the top-level view of Tetris that can be built by the Python `pygame` module:

- `build_Grid()`: This function will draw the grid into the game canvas. The grid is the place where we can render geometrical shapes with different colors.
- `create_Grid()`: This function will create different horizontal lines into the grid so that we can track each shape for rotational transformation.
- `rotating_shapes`: This technique will rotate the geometrical shapes within the same origin. This means that rotation will not alter the dimension (length and height) of an object.

Now that we've completed the brainstorming process, let's dive into the fundamental concepts of Tetris. The environment of Tetris is simple, yet powerful. We have to draw grids into it so that we can track each (*x*,*y*) position of the different shapes. Similarly, for tracking each geometrical shape, we need to create a dictionary, which will store the **position** of an object as *key* and the **color** of an object as *value*.

Let's start by writing the template code for our game:

```
import pygame
import random

#declare GLOBALS
width = 800
height = 700

#since each shape needs equal width and height as of square
game_width = 300 #each block will have 30 width
game_height = 600 #each block will have 30 height
shape_size = 30

#check top left position for rendering shapes afterwards

top_left_x, top_left_y = (width - game_width) // 2, height - game_height
```

Now that we have finished declaring the global for our game, which mostly takes care of the width and height of the screen, we can start defining the shapes format for the game objects. In the next section, we will define a nested list, which we can use to define the multiple structures of the game objects (mostly for geometrical shapes).

Creating the shapes format

The upcoming information is tricky. We are going to declare the shapes format (all the essential geometrical shapes) for Tetris. Let's look at a simple example, as follows:

```
#Example for creating shapes I
I = [['..0..',
      '..0..',
      '..0..',
      '..0..',
      '.....'],
     ['.....',
      '0000.',
      '.....',
      '.....',
      '.....']] #each 0 indicates block for shapes
```

Observe the shapes format from the preceding code. It is a nested list, and we require it because `I` supports one rotation, which will change the vertical `I` into a horizontal `I`. Observe the first element of the preceding list; it contains a period (`.`), along with an identifier (`0`), to indicate null and block placement. In the place of the dot or period, we won't have anything, and so it will remain empty. But in the place of `0`, we will store the block. To do this, remove the dot from the preceding code, and observe only element `0`. You will see vertical `I` in the zeroth index and horizontal `I` in the first index. In the case of square shapes, we don't need an extra *rotation*, and so we will end up declaring only one element inside the list for the square shape. It will be something like this:

```
#for square shapes
square = [['.....',
      '.....',
      '.00..',
      '.00..',
      '.....']]
```

Now that we know how to create a format for the geometrical shapes, let's create the starter piece of code for different shapes:

```
#following is for shape I
""" first element of list represents original structure,
    Second element represents rotational shape of objects """


I = [['..0..',
      '..0..',
      '..0..',
      '..0..',
```

```
        '.....'],
      ['.....',
       '0000.',
       '.....',
       '.....',
       '.....']]
#for square shape
O = [['.....',
       '.....',
       '.00..',
       '.00..',
       '.....']]

#for shape J
J = [['.....',
       '.0...',
       '.000.',
       '.....',
       '.....'],
      ['.....',
       '..00.',
       '..0..',
       '..0..',
       '.....'],
      ['.....',
       '.....',
       '.000.',
       '...0.',
       '.....'],
      ['.....',
       '..0..',
       '..0..',
       '.00..',
       '.....']]
```

Similarly, let's define the shape format for another few geometrical shapes, like we did previously:

```
#for shape L
L = [['.....',
       '...0.',
       '.000.',
       '.....',
       '.....'],
      ['.....',
       '..0..',
       '..0..',
       '..00.',
```

```
        '.....'],
       ['.....',
        '.....',
        '.000.',
        '.0...',
        '.....'],
       ['.....',
        '.00..',
        '..0..',
        '..0..',
        '.....']]
#for shape T
T = [['.....',
        '..0..',
        '.000.',
        '.....',
        '.....'],
       ['.....',
        '..0..',
        '..00.',
        '..0..',
        '.....'],
       ['.....',
        '.....',
        '.000.',
        '..0..',
        '.....'],
       ['.....',
        '..0..',
        '.00..',
        '..0..',
        '.....']]
```

Now that we have successfully defined the characters for our game, let's make a data structure to hold these objects, along with their color. Let's write the following code to implement this:

```
game_objects = [I, O, J, L, T] #you can create as many as you want
objects_color = [(255, 255, 0), (255, 0, 0), (0, 0 , 255), (255, 255, 0),
(128, 165, 0)]
```

Since we have completed the basic starter file, that is, we have understood and created our game objects, in the next section, we will start creating a grid for our game, as well as render the game objects onto the screen.

Creating a grid and random shapes

Now that we have defined the format of the shapes, it is time to give actual characteristics to them. The way that we provide characteristics to the shapes is by defining dimensions and color. Previously, we defined the dimension of the block as being 30 in size, which is not arbitrary; the dimension of the shapes must be equal in height and width. Every geometrical shape that we are going to draw in this chapter will resemble at least square shapes. Confused? Look at the code where we defined the shape format, including period (`.`) and character (`0`). If you observe each element of the list closely, you will see the format of the square, with equal numbers of dots arranged in rows and columns.

As we mentioned in the *Understanding Tetris essentials* section, the grid is the place or environment where our game characters will reside. The player control, or action, will be activated only within the grid area. Let's talk about how the grid can be used in our game. The grid is the division of the screen in the form of vertical and horizontal lines, which will make up each row and column. Let's make one for ourselves and observe the result:

```
#observe that this is not defined inside any class
def build_Grid(occupied = {}):
    shapes_grid = [[(0, 0, 0) for _ in range(10)] for _ in range(20)]
    for row in range(len(shapes_grid)):
        for column in range(len(shapes_grid[row])):
            if (column, row) in occupied:
                piece = occupied[(column, row)]
                shapes_grid[row][column] = piece
    return shapes_grid
```

The preceding code is complex, but it is an essential building block for most of the games that are made out of pygame. The preceding code will return a grid, which is obviously the environment for our Tetris game, but it can also be used for multiple purposes, such as building tic-tac-toe with little modification, or Pac-Man, and so on. The argument to the `build_Grid()` function is a single argument—the *occupied* dictionary. This dictionary will be passed to this function from the place where this function is called. Mainly, this function will be called inside the main function, which will initiate the process of creating a grid for the game.

The occupied dictionary that is passed to `build_Grid` will contain a key and a value (as it is a dictionary). The key will represent the position where each block or shapes resides. The value will contain the color code of each shape that is represented by the key. For example, in your print dictionary, you will see something like `{position: color_code}`.

The next line of the operation should be a gotcha moment for you. If not, you are missing something! This can be found in `Chapter 7`, *List Comprehension and Properties*. With the help of one line of code, we defined an arrangement of rows and columns (multi-dimensional list). It will provide us with a range of values that can be used to create a grid of lines. Of course, lines will be drawn later in the main function, with the help of the `pygame draw` module. We will create a list of 10 rows and a list of 20 columns. Now, let's talk about the last couple of lines of code (the highlighted part). These lines of code will loop through each occupied position and add that to the grid by modifying it.

After defining the environment for our game, the next thing we need to do is define the shapes for our game. Remember that each shape will have attributes like these:

- **Row and column position**: The grid-specific position will be specified as a certain row and column of shapes or geometrical pieces.
- **Shape name**: The identifier for a shape, which indicates which shapes to render. We will add alphabetical characters for each shape, for example, character S for shape S.
- **Color**: The color of each shape.
- **Rotation**: The angle of rotation for each shape.

Now that we are aware of the available attributes for each shape, let's define the class for shape and attach each attribute to it. Write the following code in order to create `Shape` class:

```
class Shape:
    no_of_rows = 20 #for y dimension
    no_of_columns = 10 #for x dimension
    #constructor
    def __init__(self, column, row, shape):
        self.x = column
        self.y = row
        self.shape = shape
        #class attributes
        self.color = objects_color[game_objects.index(shape)]
#get color based on character indicated by shape name or shape variable
        self.rotation = 0
```

The `objects_color and game_objects` variable was defined previously, and is two different lists that contain alphabetical characters in one list. The color code for each of them in the other list.

At this moment, if you run your game, you won't see anything except for an empty black screen, which is because our grid background was rendered with the color code of black. We know that, if we want anything to draw, it can be done with the help of the Python `pygame` module. Furthermore, we are drawing shapes from the top to the bottom of the grid, and so we have to generate shapes randomly. Since we have five shapes, that is, I, O, J, L, and T, we need to render them randomly, one by one. Let's make a function to implement in the following code snippet. Remember, we already imported a random module at the beginning:

```
def generate_shapes():
     global game_objects, objects_color
     return Shape(4, 0, random.choice(game_objects)) #creating instance
```

The preceding backend logic is vital for any game that has something to do with geometrical shapes and pieces. The scope of this knowledge is much broader than you will have expected. Many RPG games, including Minecraft, have the player interact with different geometrical shapes. Thus, creating a grid is vital so that we can reference the position and color of each piece. Now that we have created some general logic that will create pieces of different shapes and color, we need a tool that can render such shapes into the grid, which is normally done by either OpenGL or pygame (PyOpenGL will be covered in the upcoming `Chapter 14`, *Getting to Know PyOpenGL*). However, the superior tool will be pygame, in the case of Python. Thus, we will make the Tetris game shapes and characters with the help of the `pygame` module.

In the next section, we will create some logic that will set up a game window for the grid structure. We will also try to run our game and observe its environment.

Setting up the window and game loop

The next big thing in our game, after setting up the game objects, is to render the grid. Don't get confused by thinking that we have already created the grid, after we defined the `build_Grid()` method. Although it is a valid point, the grid that we built is virtual up to this point. If you simply call the `build_Grid` method, you won't see anything but a black screen, which is the background of the grid. Here, we are going to provide a structure to this grid. Using each position, specified by row and column, we are going to create a straight line using the `pygame` module.

Let's make a simple function to draw a window for our game (the main window) in which the grid will reside:

```
def create_Grid(screen_surface, grid_scene):
     screen_surface.fill(0, 0, 0) #black background
```

```
        for i in range(len(grid_scene)):
        for j in range(len(grid_scene[i])):

    #draw main rectangle which represents window
        pygame.draw.rect(screen_surface, grid_scene[i][j], (top_left_x +
          j* 30, top_left_y + i * 30, 30, 30), 0)
    #above code will draw a rectangle at the middle of surface screen
       build_Grid(screen_surface, 20 , 10) #creating grid positions
       pygame.draw.rect(screen_surface, (255, 0, 0), (top_left_x, top_left_y,
         game_width, game_height), 5)
       pygame.display.update()
```

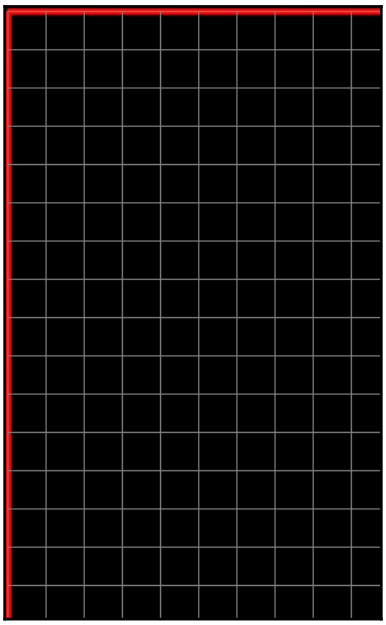
The preceding line of code will create the physical structure of the grid, which will have different rows and columns. After looping through the entire grid scene or positions of the grid, we will enter the grid scope in order to draw a rectangle and a grid border with the previously highlighted part of the code.

Similarly, let's provide a physical structure to this grid by defining borders for it. Each row and column will be distinguished by creating lines within it. Since we can draw lines with the pygame `draw` module, we will use it to write the following function:

```
"""function that will create borders in each row and column positions """

def show_grid(screen_Surface, grid):
    """ --- following two variables will show from where to
     draw lines---- """
    side_x = top_left_x
    side_y = top_left_y
    for eachRow in range(grid):
        pygame.draw.line(screen_Surface, (128,128,128), (side_x, side_y+
        eachRow*30), (side_x + game_width, side_y + eachRow * 30))
         # drawing horizontal lines (30)
        for eachCol in range(grid[eachRow]):
            pygame.draw.line(screen_Surface, (128,128,128), (side_x +
            eachCol * 30, side_y), (side_x + eachCol * 30, side_y +
               game_height))
            # drawing vertical group of lines
```

The preceding function has one main loop, which loops into several rows, as determined by the `build_Grid` method. After going into each row of the grid structure, it will use the `pygame draw` module to draw lines with a color code of (128, 128, 128), starting from (`side_x`, `side_y`) and then pointing to the next coordinate (`side_x + game_width, side_y + eachRow *30`). The starting point (`side_x`, `side_y`) is the left-most corner of the grid, while the next coordinate value of (`side_x + game_width, side_y + eachRow *30`) represents the coordinate of the right-most corner of grid. Thus, we will a draw line from the left-most corner of the grid to the right-most corner.

After you explicitly call the previous function, you will see the following output:



After setting up the aforementioned grid or environment, we will hop into the fun stuff, which is creating the main function. The main function will have a different bunch of stuff in it, mostly for calling up and setting the grid, and handling user events or actions, such as what happens when the user presses quit or presses an arrow key on the keyboard. Let's define it with the following code:

```
def main():
 occupied = {} #this refers to the shapes occupied into the screen
 grid = build_Grid(occupied)

 done = False
 current_shape = generate_shapes() #random shapes chosen from lists.
 next_shape = generate_shapes()
 clock = pygame.time.Clock()
 time_of_fall = 0 #for automatic fall of shapes

 while not done:
 for eachEvent in pygame.event.get():
 if eachEvent.type == pygame.QUIT:
 done = True
 exit()
```

Since we have started defining the main function, which is the director of our game, let's define what things it must do, as follows:

- Call multiple functions, such as `build_Grid()` and `create_Grid()`, which will set up the environment for games
- Define a method that will perform rotations for a shape that represents characters
- Define some logic that will add fall time constraints to the game, that is, the speed at which objects fall
- Change a shape in, after one shape fall to the ground
- Create some logic to check the occupied position of the shapes

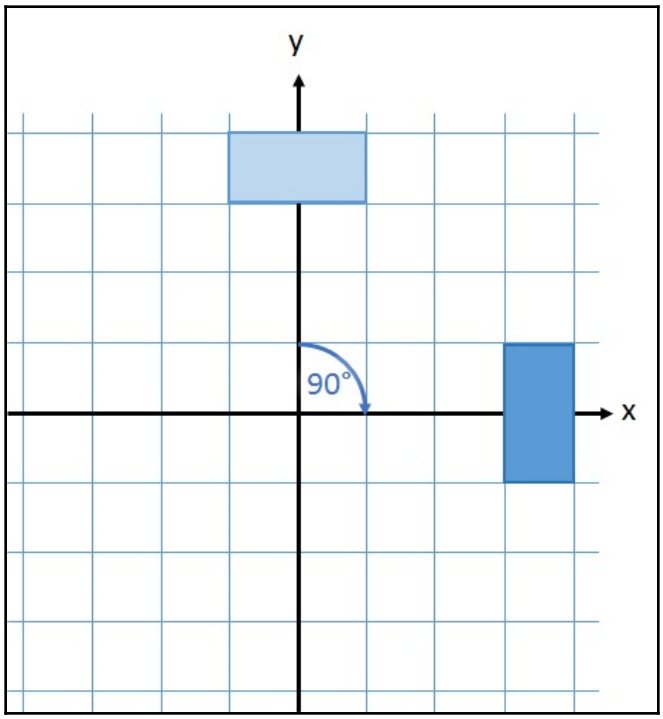
The aforementioned processes are the main function capabilities, and we should address them. We will address the first two in this section, but the remaining two will be covered in the upcoming sections. So, the first operation of the main function is to call some essential functions that will create the grid for the game. If you look at the aforementioned line of code, you will see that we have already called the `build_Grid` method, which is responsible for creating the virtual positions for rows and columns of a grid-like structure. Now, the remaining task is to only call the `create_Grid()` method, which will give a proper physical structure to this virtual grid, using the `pygame draw` module. We have already defined both of these functions.

In the next section, we'll learn about one of the important mathematical paradigms of transformation, which is known as rotation, and will add the feature of rotating game objects to our Tetris game.

Understanding rotations

Before we continue to code and modify the main function, let's get into the mathematical stuff. Games are nothing if they are not related to a mathematical paradigm. Movement, motions, shapes, characters, and controls are all handled by mathematical expressions. In this section, we are going to cover another important concept of math: transformations. Although transformations is a nebulous concepts in math, we will try our best to learn this concept. Specifically, there are different types of transformations: rotation, translation, reflection, and enlargement. In most games, we will need only two types of transformation: rotation and enlargement. In this chapter, we will implement rotational transformations using Tetris, and then we will implement the enlargement transformation (while building an Angry Birds game in `Chapter 16`, *Learning Game AI – Building a Bot to Play*).

The term *rotation* is a mathematical concept which states that *When an object is rotated, it means that it is turned either clockwise or anticlockwise with a certain amount of specified degree*. Consider the following example:



In the preceding example, we have a rectangular shape, which represents the alphabetical `I` character of our Tetris game. Now, imagine that the player presses the *up* arrow key on the keyboard. In such an event, the rectangular shape of `I` must be rotated with an angle of 90 degrees and placed as the horizontal `I` character, as shown in the preceding diagram. Thus, these rotations are done to change the shape of the figure, but not the dimensions. Horizontal `I` and vertical `I` have the same dimensions (height and width). Now that you know a little bit about rotations, you can go back to the code where we defined the shape format for each character (I, O, J, L, and T) and observe the multi-dimensional list. In the case of `I`, you could observe that it has two elements. The first element of the list is the original shape of the game object, `I`, and the second element of the list is a distorted shape after a rotation of about 90 degrees. Observe the same for the `O` character, which is square. The square will remain the same, even after a rotation by any degree. Thus, in the case of the square shape, we have only one element in the list.

Although we've learned this trivia about rotations, and how they are attached with the each shape format, the question still remains that: when can we render each shape, and when should the operation of rotations be carried out? The answer is simple. While a player presses any arrow key on the keyboard, we are going to perform rotations. But where is the code that implies that the user is pressing a keyboard key? Obviously, it is done inside the event handling process! In the main function, we started to capture the event, and we handled the actions for the `QUIT` key. Now, let's perform the rotations for any arrow key with the following code:

The code should be added inside the event handling steps, right after handling the `QUIT` key. Make sure that you provide a proper indentation for the code. The code will be available at `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter13`.

```
if anyEvent.type == pygame.KEYDOWN:
        if anyEvent.key == pygame.K_LEFT:
            current_shape.x -= 1  #go left with shape

        elif anyEvent.key == pygame.K_RIGHT:
            current_shape.x += 1 #go right with shape
        elif anyEvent.key == pygame.K_UP:
            # rotate shape with angle of rotation
             (rotation variable)
            current_shape.rotation = current_shape.rotation + 1 %
             len(current_shape.game_objects)
        if anyEvent.key == pygame.K_DOWN:
            # moving current shape down into the grid
            current_shape.y += 1
```

If you want to learn more about how the rotation of objects works under the hood, make sure that you check out the following URL: `https://mathsdoctor.co.uk`.

In order to set up the window canvas or game screen, we can simply call the `pygame set_mode` method and render the window of the grid accordingly. The following line of the method call should be added within the main function, right after you have set up the user handling events:

```
create_Grid(screen_surface) #screen surface will be initialized with
                             pygame below
```

Now that we have created a grid for the screen, let's set up the main screen and call up the main function:

```
screen_surface = pygame.display.set_mode((width, height))
main() #calling only
```

We have covered almost all of the important things, including rendering the display, rotating objects, creating grids, and rendering borders for grids; but one question still remains: how do we render the shapes into the grid? Obviously, our computer is not smart enough to understand the multi-dimensional list that we created earlier to define the shapes format. Still confused? Check the multi-dimensional list that we created for each character, such as I, O, J, L, and T—our computer won't understand such a list. Thus, we have to convert these list values or attributes into the dimensions that will be recognized by our computer for further processing. The dimensional value that our computer will understand refers to the positional value. Since we have established the grid already, we can use rows and columns from the grid structure to give positional value to the computer. Thus, let's make a function to implement it.

Converting the shape format

Our computer doesn't have the capability to understand the obscure content of data structures, such as the content that is stored inside the multi-dimensional list. For example, take a look at the following code:

```
#for square shapes
square = [['.....',
      '.....',
      '.00..',
      '.00..',
      '.....']]
```

In the previous square-shaped pattern, we have coupled a list of periods (`.`) with `0`. The computer won't recognize what 0 means, and what the period refers to. We only have the knowledge that the period is in a position that is an empty place, which means its position can be ignored, and the position where `0` resides is the position for the block. Thus, we need to write a program to tell the computer to extract the position from the grid where only `0` resides for each of the pieces. We will implement it by defining the following function:

```
def define_shape_position(shape_piece):
    positions = []
    list_of_shapes = shape_piece.game_objects[shape_piece.rotation %
                     len(shape_piece.shape)]
```

```
        for i, line in enumerate(list_of_shapes):
            row = list(line)
            for j, column in enumerate(row):
                if column == '0':
                    positions.append((shape_piece.x + j, shape_piece.y + i))

        for p, block_pos in enumerate(positions):
            positions[p] = (block_pos[0] - 2, block_pos[1] - 4)

        return positions
```

Let's look at the previous code in detail:

1. To begin with, this function returns the position of the block of the objects. Thus, we start by creating a block dictionary.
2. Secondly, we store several lists of shapes, as defined by a multi-dimensional list of characters, which are defined by `game_objects` (I, O, J, L, and T) with rotations.
3. Now, the important part: what are the positions that must be returned by this function? These positions are the position of `0`, placed in the grid.
4. Observe the multi-dimensional list again. You will see a bunch of dots (`.`) and `0` placed as elements. We only want the position where `0` resides, and not where the period or dot resides.
5. After we check each column for `0` with the `if column == \'0\'` command, we only store such positions into the positions dictionary, and return it from the function.

When operations such as rotation and movements are done, it is often the case that the user might trigger some invalid movements, such as rotating the objects outside of the grid. Thus, we have to check such invalid movements and prevent them from happening. We will create the `check_Moves()` function to implement this. The argument to this function will be the shape and grid position; shape is essential to check if a specific rotation is allowed or not within the position that is indicated by the grid argument. If the current position specified by the grid in which shape resides is already occupied, then we will get rid of such moves. There are different ways to implement it, but the quickest and easiest way is to check the color of the grid background. If the color of the particular position in the grid is anything other than black, it means that the position is occupied. Thus, you can make an articulated reference from this logic as to why we made the background color of the grid black. By doing so, we can check if the objects are already in the grid or not. If any new object comes down to the grid, we should not pass it through the object that is already present in the grid.

Now, let's make a function to check if the position is occupied or not:

```
def check_Moves(shape, grid):
    """ checking if the background color of particular position is
        black or not, if it is, that means position is not occupied """

    valid_pos = [[(j, i) for j in range(10) if grid[i][j] == (0,0,0)]
                 for i in range(20)]
    """ valid_pos contains color code in i variable and
        position in j variable--we have to filter to get only
        j variable """
    valid_pos = [j for p in valid_pos for j in p]
           """ list comprehension --same as writing
                    for p in valid_pos:
                        for j in p:
                            p
                            """
    """ Now get only the position from such shapes using
        define_shape_position function """
    shape_pos = define_shape_position(shape)

    """check if pos is valid or not """
    for eachPos in shape_pos:
        if eachPos not in valid_pos:
            if eachPos[1] > -1: #eachPos[1] represents y value of shapes
              and if it hits boundary
                return False #not valid move
    return True
```

Up until now, we were building the backend logic for our game, which refers to rendering the grid, manipulating the grid, changing grid positions, implementing logic that determines what happens when two objects collide, and so on. Even though we have done so much already, when you run your game, you will still see only the formation of the grid, and nothing more. This is because our main loop is the director of our game—it will sequentially order the other functions, but inside the main loop, we have nothing except the code that handles the user events. Thus, in the next section, we will the modify main loop for the game and observe the output.

Modifying the game loop

As we mentioned previously, our main game loop is accountable for performing many tasks, including handling user events, handing the grid, checking possible moves, and so on. We have been making functions that will check such actions, movements, and environments, but we have not called them once, which we will do in this section. If you observe the main game loop from a high-level perspective, it will contain four primary architectural building blocks:

- Creating the grid and handling movements of the game objects. For instance, what should be the speed of the objects that will fall down into the grid?
- Handling user events. We have already done this, when we checked the events and rotated the objects accordingly. But the preceding code didn't accommodate the `check_Moves()` function, which will check if the moves are valid or not. Thus, we will modify the preceding code accordingly.
- Adding color to the game objects (unique color). For instance, the color of `S` should be different to `I`.
- Adding the logic that will check what happens when the object hits the ground of the grid.

We will implement each of the aforementioned steps one by one. Let's start by adding speed to the object. Speed refers to the free-falling speed of the objects in the grid structure. The following code should be added inside the main function:

```
global grid

occupied = {} # (x pos, y pos) : (128, 0, 128)
grid = build_Grid(occupied)
change_shape = False
done = False
current_shape = generate_shapes()
next_shape = generate_shapes()
clock = pygame.time.Clock()
timeforFall = 0

while not done:
speedforFall = 0.25

grid = build_Grid(occupied)
timeforFall += clock.get_rawtime()
clock.tick()

# code for making shape fall freely down the grid
if timeforFall/1000 >= speedforFall:
```

```
timeForFall = 0
current_shape.y += 1 #moving downward
#moving freely downward for invalid moves
if not (check_Moves(current_shape, grid)) and current_shape.y > 0:
current_shape.y -= 1
change_shape = True
```

Suppose that the player tries to make an invalid move. Even in that case, the game objects (shapes) must fall freely downwards. Such an operation is done in the last three lines of the previous code. Other than that, the code is self-explanatory; we have defined the speed for the object to fall into the grid and used a clock module to implement the time constraints.

To implement the next logic, this is relatively easier. We have already discussed handling user events in Tetris while considering details such as rotating objects and performing simple left-to-right movements. However, in those lines of code, we didn't check if the moves that the user tried to make were valid or not. We have to check this first in order to make sure that users are prevented from making any invalid moves. To implement this, we are going to call the `check_Moves()` method, which we created previously. The following code will handle user events:

```
if anyEvent.type == pygame.KEYDOWN:
                if anyEvent.key == pygame.K_LEFT:
                    current_shape.x -= 1
                    if not check_Moves(current_shape, grid):
                        current_shape.x += 1  # not valid move thus
                           free falling shape

                elif anyEvent.key == pygame.K_RIGHT:
                    current_shape.x += 1
                    if not check_Moves(current_shape, grid):
                        current_shape.x -= 1
      """ ROTATING OBJECTS """
                elif anyEvent.key == pygame.K_UP:
                    current_shape.rotation = current_shape.rotation + 1 %
                        len(current_shape.shape)
                    if not check_Moves(current_shape, grid):
                        current_shape.rotation = current_shape.rotation - 1
                           % len(current_shape.shape)

"""Moving faster while user presses down action key """
                if anyEvent.key == pygame.K_DOWN:
                    current_shape.y += 1
                    if not check_Moves(current_shape, grid):
                        current_shape.y -= 1
```

Firstly, focus on the code that is highlighted. The first highlighted part of the code refers to whether the move is valid into the grid, which is checked by the `check_Moves()` function. We are allowing the current shapes to move to the right corner, which is toward the positive *x*-axis. Similarly, regarding the up key, it is responsible for checking if rotation of the object is allowed or not (only the up key will rotate the objects; the *left* and *right* keys will move the objects from left to right, and vice versa). In the case of rotation, we are rotating it through pixel transformations, which is done by selecting one of the positions indicated by the multi-dimensional list. For example, in the case of shape I, we have two elements in the list: one original shape and another rotational shape. Thus, to use another rotational shape, we will check if the move is valid or not, and if it is, we will render the new shape.

The third piece of code that should be added into the main function will deal with the technique that will add the color to the shapes in the grid for drawing. The following line of code will add the color to each of the objects that is inside the scope of the game:

```
position_of_shape = define_shape_position(current_shape)
""" define_shape_function was created to return position of blocks of
    an object """

   # adding color to each objects in to the grid.
   for pos in range(len(position_of_shape)):
       x, y = position_of_shape[pos]
       """ when shapes is outside the main grid, we don't care """
       if y > -1: # But if we are inside the screen or grid,
          we add color
           grid[y][x] = current_shape.color #adding color to the grid
```

Finally, the last piece of logic that must be added to the main function will address the situation of when an object hits the ground. Let's add the following code into the main function in order to implement it:

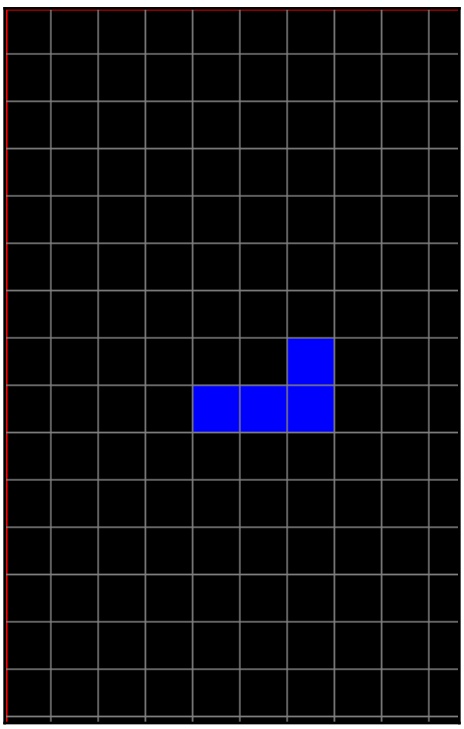
```
if change_shape:
        for eachPos in position_of_shape:
            pos = (eachPos[0], eachPos[1])
            occupied[pos] = current_shape.color
        current_shape = next_shape
        next_shape = generate_shapes()
        change_shape = False
```

In the preceding code, we are checking whether the objects are falling freely or not by checking the contents of the Boolean variable, `change_shape`. Then, we are checking the current position of the shapes and creating (*x*, *y*), which will represent the occupied position. We then add such a position to the dictionary named occupied. You must remember that the value of this dictionary is the color code of the same object. After assigning the current object to the grid scope, we will generate a new shape with the help of the `generate_shapes ()` method.

Finally, let's end our main function by calling the `create_Grid()` function with the argument of the grid and surface objects that were initialized by the pygame `set_mode()` method in the following code (we initialized the pygame `surface` object previously):

```
create_Grid(screen_surface, grid)
```

Let's run our game and observe the output for it:



Now, you can clearly see that we are able to make a Tetris game where the users are able to transform the objects and play accordingly. But wait! We are missing one important piece of logic in our game. How do we incentivize our player to play this game? If the game was all about making rotations of objects, and filling up grids with the objects, it would not have been the historical game that it is (the game that revolutionized the 90s gaming industry). Yes! There is some logic that must be added into the game, and when this logic is called, we will observe that whenever the row positions are *occupied* by the blocks, we have to clear such rows and shift the row one step down, which will leave us with fewer rows than before. We will implement this in the next section.

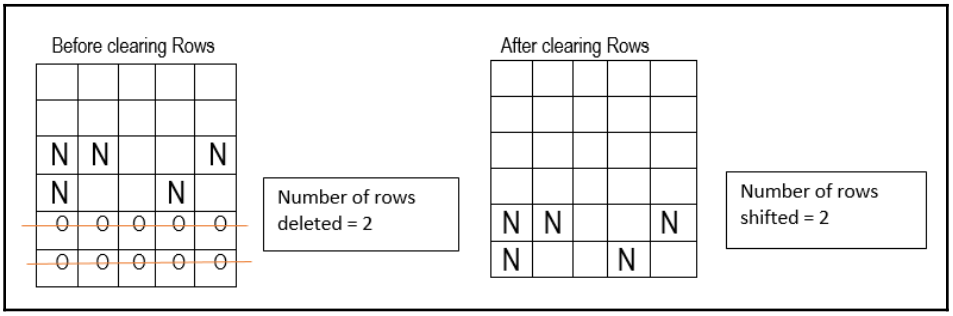
Clearing the rows

As we mentioned previously, in this section, we will check if every position, of all the rows, is entirely occupied or not. If they are occupied, we will delete such rows from the grid, and this will create a shift in each row by one step down into the grid. This logic is simple to implement. We will check whether or not the entire row is occupied and delete such rows accordingly. Do you remember the case of creating the `check_Moves()` function? If this function checked the background color of each row, and if in each row there is no black background color, it means that such a row is occupied. But even if we have one position empty, this means that the background color of such a position will be black, and will be considered as not occupied. Thus, we can use a similar type of technique in the case of clearing the rows: if, in any row, the background color of any position is black, it means that the position is not occupied, and such rows cannot be cleared.

Let's make a function to implement the logic of clearing the rows:

```
def delete_Row(grid, occupied):
    # check if the row is occupied or not
    black_background_color = (0, 0, 0)
    number_of_rows_deleted = 0
    for i in range(len(grid)-1,-1,-1):
        eachRow = grid[i]
        if black_background_color not in eachRow:
            number_of_rows_deleted += 1
            index_of_deleted_rows = i
            for j in range(len(eachRow)):
                try:
                    del occupied[(j, i)]
                except:
                    continue
```

Let's digest the preceding code. It is quite a complex piece of logic, so make sure that you learn everything about it; these concepts are not only suitable for game creation but are also asked many times in a technical interview. The question lies in how to shift the values of the data structure by creating logic, and not by using the Python built-in function. I wanted to teach you this in this way instead of using any built-in method because knowing this might be helpful in any technical field of programming. Now, let's observe the code. It starts with creating a `number_of_rows_deleted` variable, which indicates the number of rows that have been deleted from the grid. The information regarding the number of deleted rows is important because after deleting that number of rows, we need to shift the rows which reside above the deleted row(s) by an equal number down the grid. For example, look at the following diagram:



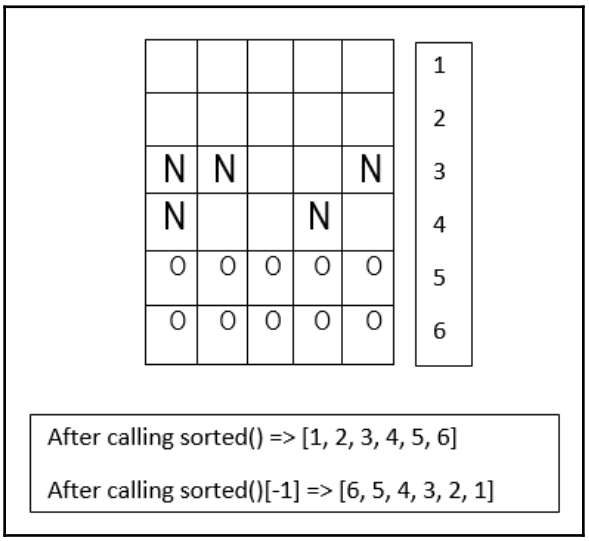
Similarly, now that we know what to delete with the `if black_background_color not in eachRow` expression, we can determine whether or not each row of the grid has empty places or not. If there are empty places, this means that the rows are not occupied, and if yes, then a black background color, that is, (0, 0, 0), won't be within any row. If we didn't find a black background color, then we can be sure that the rows are occupied, and we can delete them by checking further conditions. In the highlighted part of the code, you can observe that we are taking only the j^{th} element, which is only a column. This is because, while deleting row, the value of `I` remains the same, but the j^{th} column value differs. Thus, we loop on an entire column within a single row and use the `del` command to delete the occupied position.

From the preceding line of code, we were able to delete entire rows if any rows were occupied, but we didn't address what should happen after we delete it, and this is the tricky part. After we delete every occupied row, not only the blocks will be deleted—the entire grid containing rows will be deleted. Thus, in place of deleted block, we won't have empty rows; instead, whole rows containing the grid will be deleted. Thus, to make sure that we do not decrease the count of the actual grid, we need to add another row from the top to compensate for it. Let's write some code to implement this:

```
#code should be added within delete_Row function outside for loop
if number_of_rows_deleted > 0:       #if there is at least one rows deleted
        for position in sorted(list(occupied), position=lambda x:
          x[1])[::-1]:
            x, y = position
            if y < index_of_deleted_rows:
                """ shifting operation """
                newPos = (x, y + number_of_rows_deleted)
                occupied[newPos] = occupied.pop(position)

return number_of_rows_deleted
```

Okay! Let's digest it. This is quite complex but extremely powerful information. The preceding code will implement shifting the block of rows from the top, down into the grid. Firstly, the shift is required only if we have deleted any row; if yes, we enter into the logic to perform shifting. First of all, let's only observe the code that involves the lambda function, that is, `list(occupied), position=lambda x: x[1]`. The code will create a list of all the positions of the grid and then use the lambda function to take the *y*-part of the position only. Remember, taking the *x* position of the block is superfluous—for each row, the value of *x* remains constant, but the *y* values differs. Thus, we will take the value of the *y*-position and then sort it with the `sorted(x)` function. The sorted function will sort the position based on the value of the *y*-coordinates.

Firstly, the sorting will be done based on the lower value of *y* to the upper value of *y*. For example, look at the following diagram:



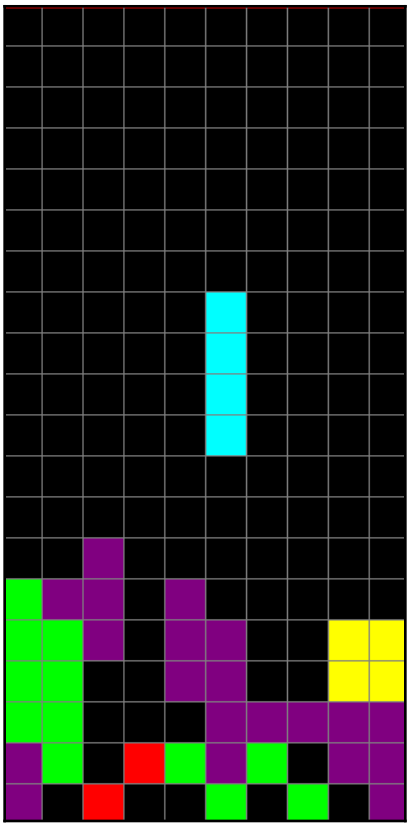
Calling the sorted method and then reversing the list (refer to `Chapter 4`, *Data Structures and Functions*, to learn more about how to reverse lists) is important because sometimes the bottom part of the grid won't be occupied, and only the upper layer will. In such cases, we don't want the shifting operation to cause any harm to the bottom rows, which are not occupied.

Similarly, after taking track of the position of each row, we will check if there are any rows above the deleted row with the `if y < index_of_deleted_rows` expression. Again, in this case, the value of *x* is irrelevant because it will be the same within the single row; after we check if there is any row above the row that is deleted, we perform the shifting operation. The operation of shifting is quite simple; we will try to assign the new position for each of the rows that reside just above the deleted row. We can create a new position by increasing the value of *y* with the number of deleted rows. For instance, if there are two rows being deleted, we need to add two to the value of *y* so that the block just above the deleted rows, and the subsequent ones, will shift two rows down. After we shift the rows down into the grid, we have to pop the blocks out from the previous position.

Now that we have defined a function that will clear the entire row if it is occupied, let's call it from the main function to observe its effect:

```
def main():
    ...
    while not done:
        ...
        if change_shape:
            ...
            change_shape = False
            delete_Row(grid, occupied)
```

Finally, with this protracted and tedious day of coding, we have a very productive result. When you run your module where the main function is declared, you will see the following output:



The game looks appealing, and I have tested everything in the code. The code looks thorough and exhaustive, with no loop holes. Similarly, you can play it and share it with your friends and uncover the possible modifications that can be done with this game. This is an advanced game and it adequately raises its bar, when it is coded with Python from scratch. We have learned so many things while building this game. We learned how to define the shapes format (we have done even more complicated stuff before, such as the transformation of sprites, and handling the collision of sprites), but this chapter was challenging on different aspects. For example, we had to take care of things such as invalid moves, possible collisions, shifting, and so on. We implemented some logic that determined if an object is placed in a certain position or not by comparing two distinct color objects: the **background color** of the grid or surface against the **game-object color**.

We are not done yet; we will try to implement some more logic in the next section. We will see what other modifications we can make to our game. We will try to build some logic that will increase the difficulty level of our game as we proceed.

Game testing

Several modifications can be made to our game, but the most important ones will be to add a welcome screen, an increased difficulty level, and a score screen. Let's start with the welcome screen, since it is easy to implement. We can use the `pygame` module to create a window, and a text surface to provide a message to the user. The following code shows how to create a main screen for our Tetris game:

```
def Welcome_Screen(surface):
    done = False
    while not done:
        surface.fill((128,0,128))
        font = pygame.font.SysFont("comicsans", size, bold=True)
        label = font.render('Press ANY Key To Play Tetris!!', 1, (255, 255,
                255))

        surface.blit(label, (top_left_x + game_width /2 -
         (label.get_width()/2), top_left_y + game_height/2 -
          label.get_height()/2))


        pygame.display.update()
        for eachEvent in pygame.event.get():
            if eachEvent.type == pygame.QUIT:
                done = True
            if event.type == pygame.KEYDOWN:
                main(surface) #calling main when user enters Enter key

    pygame.display.quit()
```

After you run the game, you will see the following output, in which the welcome screen will be rendered. After pressing any key, you will be redirected to the Tetris game:

Similarly, let's add some logic that will increase the difficulty of the game. There are two ways of implementing this logic. Firstly, you can create a timer, and if a player plays more than the range of the associated timer, we can decrease the fall speed so that shapes will fall faster than before (increase speed):

```
timeforLevel = 0

while not done:
    speedforFall = 0.27 - timeforLevel
    ...
    if timeforLevel / 10000 > 0.5:
        timeforLevel = 0
        if timeforLevel > 0.15:
            timeforLevel += 0.05
    ...

"""  ------------------------------------------------------
        speedforFall = 0.24 will make object to fall faster comparative
                       to speedforFall = 0.30

    ------------------------------------------------------ """
```

Similarly, we can implement another piece of logic to increase the difficulty of the game. This method is better one than the preceding one. In this method, we will use *score* to increase the difficulty of the game. The following code represents a blueprint of how to implement the score of the player in order to increase the level of the game:

```
def increaseSpeed(score):
    game_level = int(score*speedForFall)
    speedforFall = 0.28 - (game_level)
    return speedforFall
```

In the preceding code, we implemented the relationship between the score and the speed of objects. Let's suppose a player's score is higher. This means that the user has been playing a less difficult level, and thus, such a high score value will be multiplied with the higher speed of fall value, resulting in an increase in `speedforFall`, which will be then subtracted from the speed of the objects, which will create a faster fall motion. In contrast, a player playing on a higher level will have a lower score, which will be multiplied with a lower value of the speed of objects, resulting in a lower number, which will be then subtracted from the `speedforFall` variable. This will result in less change in the speed for the player who is playing the harder level. But let's say a player is a pro and has scored higher in a harder level. In this case, the speed of the fall of an object is increased accordingly.

We have finally completed a fully functional Tetris game. We have learned several advanced concepts of game programming using Python in this chapter. In the process of its creation, we revised a few of the concepts that we learned about previously while discovering the fundamental concepts of Python, such as manipulating multi-dimensional lists, list comprehensions, object-oriented paradigms, and mathematical transformations. Along with revising those concepts, we uncovered several novel concepts such as implementing rotations, implementing shifting operations, creating a shape format from scratch, creating a grid (virtual and physical) structure, and populating objects within the grids.

Summary

In this chapter, we have explored the *Pythonic* way of implementing multi-dimensional list processing. We have created a multi-dimensional list to store the format for different geometrical shapes, and manipulated it using mathematical transformations.

We have used the simple example of Tetris to demonstrate the usage of several data structures in the game, along with its manipulation. We have implemented a dictionary to store key as a position and value as the color code of those objects. Building such a dictionary is a life-saver for games such as Tetris. While making logic to check the collisions and shifting operations, we used the dictionary to observe whether the background color of any object is the same as the background of any position. Although Tetris is only one case study, the techniques that are used in this game are also used in many real-world games, including Minecraft, and in almost every RPG game.

The operations involving mathematical transformation were vital for us. We used rotational principles in this chapter in order to change the structure of the objects without changing their dimensions. The knowledge that you will have grasped from this chapter is enormous. Concepts such as the manipulation of a multi-dimensional list can be extended to data applications, and is termed as a 2D Numpy array, which is used in creating different analogies, such as street analogy, the multiple travelers problem, and so on. Although it is considered that the dictionary is the king of the data structures, processing a multi-dimensional list is not too far behind as it is combined with the simplicity of list comprehensions. Along with the implementation of such complex data structures, we learned how to implement mathematical transformations, that is, the rotational movement of game objects. This feature is extremely useful in any 3D game as it will provide the user with a 360-dimensional view of the scene. Similarly, we have learned how to create a grid structure.

A grid structure is used to track the positions of the objects. In complex games such as WorldCraft, it is a mandatory task of any game developer to track the objects and resources for the game, and in such cases, the grid works perfectly. Invisible grids can be implemented as a dictionary, or as any complex collection.

The main goal of this chapter was to familiarize you with 2D game graphics, that is, drawing primitives and game grids. Similarly, you learned about another way of detecting collisions between game objects (in the Flappy Bird game, we used the pygame masking technique to detect collisions). In this chapter, we implemented a universal and traditional way of implementing collision checks: by checking the background color attributes with the object color attributes. Similarly, we learned how to create different objects (that differed in structure) by using rotations. This technique can be used to spawn multiple enemies into games. Instead of designing multiple different objects for each character (which can be time-consuming and costly), we used transformations to change the structures of objects.

The next chapter is about Python OpenGL, which is often termed PyOpenGL. We will see how we can create different geometrical structures using openGL, and observe how to use PyOpenGL and pygame together. We will primarily focus on different mathematical paradigms. We will see how attributes such as vertices and edges are used to create different complex mathematical shapes. Furthermore, we will see how we can implement ZOOM IN and ZOOM OUT features in the game using PyOpenGL.

14
Getting to Know PyOpenGL

Geometrical shapes and figures play a vital role in game development. We tend to neglect their importance when it comes to the development of advanced graphics technologies. However, many popular games still use these shapes and figures to render game characters. Mathematical concepts such as transformations, vectored movements, and ZOOM IN and ZOOM OUT capabilities add weight when it comes to the manipulation of game objects. Python has several modules to support such manipulations. In this chapter, we are going to learn about one such powerful Python feature—the PyOpenGL module.

While exploring PyOpenGL, we will learn how to create complex geometrical shapes using primitives (that is, vertices and edges). We will start by installing Python PyOpenGL and start drawing with it. We will make several objects, such as triangles and cubes, with it. We won't be using pygame to create such shapes; instead, we will use pure mathematical concepts for defining rectangular coordinate points for vertices and edges. We will also explore different PyOpenGL methods such as clipping and perspective. We will cover each of them to gain knowledge about how PyOpenGL can be used to create appealing game characters.

By the end of this chapter, you will be familiar with the traditional and mathematical ways of creating primitives. This way of creating shapes provides programmers and designers with the ability to manipulate their game objects and characters. You will also learn how to implement ZOOM-IN and ZOOM-OUT capabilities in the game, as well as how to use color properties by drawing geometric primitives.

The following topics will be covered in this chapter:

- Understanding PyOpenGL
- Making objects with PyOpenGL
- Understanding PyOpenGL methods
- Understanding color properties

Technical requirements

You will need the following list of requirements to complete this chapter:

- Pygame editor (IDLE) version 3.5+ is recommended.
- You will need the Pycharm IDE (refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, for the installation procedure).
- The code assets for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter14`

Check out the following video to see the code in action:

`http://bit.ly/2oJMfLM`

Understanding PyOpenGL

In the past, graphical programs containing three-dimensional scenes that had been processed with 3D-accelerated hardware was something every game programmer wanted. Even though this is normal by today's standards, the hardware is not the same as it was years ago. Most of the game's graphics had to be rendered with the software that resided in the low-processing devices. Hence, apart from creating such scenes, rendering would also take quite a bit of time and would ultimately make the game slow. The advent of gaming interfaces, also known as graphics cards, created a revolution in the gaming industry; programmers were now only bound to making interfaces, animation, and autonomous gaming logic rather than concerning themselves with processing power. Hence, games that have been created post-90s have richer gameplay and a touch of artificial intelligence (multiplayer games).

It is well-known that graphics cards can handle three-dimensional capabilities such as rendering and optimizing scenes. However, to use such features, we need a programming interface that communicates between our project and such interfaces. The **Application Programming Interface** (**API**) we are going to use in this chapter is OpenGL. OpenGL is a cross-platform (the program runs on any machine) API that is generally used to render 2D and 3D graphics. The API is analogous to libraries that are used to facilitate interaction with a graphics processing unit, and it accelerates the graphics rendering method by using hardware-accelerated rendering. It comes pre-installed on most machines as part of a graphics driver, though you can check its version by using the *GL view utility*. Before we start writing programs so that we can draw geometrical shapes and figures using PyOpenGL, we need to install it on our machine.

Installing PyOpenGL

Even if OpenGL is already present on your system, you need to install the PyOpenGL module separately so that the required OpenGL drivers and Python frameworks can communicate with each other. The Pycharm IDE provides a service that can locate Python interpreters and install PyOpenGL, which removes the overhead of installing it manually. Follow these steps to install PyOpenGL in the Pycharm IDE:

1. Click on **File** from the top navigation bar and then **Settings**. Then, hover over the left-hand side navigation window and select the **project:interpreter** option.
2. Select the current project Python interpreter, that is, Python 3.8+ (followed by your project name), and press the **to add (+)** button from the menu screen next to the **Interpreter** drop-down menu.
3. Search for PyOpenGL in the search bar and press the **Install package** button.

Alternatively, if you want to install PyOpenGL externally, you can download it as a Python egg file.

A *Python egg* is a logical structure embodying the release of a specific version of a Python project, comprising its code, resources, and metadata. There are multiple formats that can be used to physically encode a Python egg, and others can be developed. However, a key principle of Python eggs is that they should be discoverable and importable. That is, it should be possible for a Python application to easily and efficiently find out what eggs are present on a system and ensure that the desired eggs' contents are importable.

These types of files are bundled together to create Python modules that can be downloaded from the **Python Enterprise Application Kit** (**PEAK**) with the help of an easy install procedure. To download a Python egg file, you have to download the Python `easy_install` module. Go to `http://peak.telecommunity.com/DevCenter/EasyInstall` and then download and run the `ez_setup.py` file. After successfully installing easy install, run the following command in your command shell/Terminal to install PyOpenGL:

```
easy_install PyOpenGL
```

Easy install is not only used for installing PyOpenGL—you can download or upgrade a large range of Python modules with its help. For example, the `easy_install` SQLObject is used to install SQL PyPi packages.

As usual, when we need to use packages, we need to import them into our project. In this case, you can make a demo project (`demo.py`) to start testing the OpenGL project. So that we can use features such as code maintainability and debugging, we will make a PyOpenGL project using the Pycharm IDE rather than using Python's built-in IDE. Open any new project and follow these steps to check whether PyOpenGL is running or not:

1. Start by importing every class of PyOpenGL with the following command:

```
from OpenGL.GL import *
```

2. Now, import the required OpenGL functions using the following command:

```
from OpenGL.GLU import *
```

3. Next, you should import `pygame` into your project:

```
from pygame.locals import *
```

4. Initialize the display for your project with the `pygame` command:

```
import pygame
from pygame.locals import *
window_screen = pygame.display.set_mode((640, 480),
  HWSURFACE|OPENGL|DOUBLEBUF)
```

5. Run your project and analyze the result. If a new screen appears, you can continue making projects. However, if the prompt says **PyOpenGL is not installed**, make sure to follow the preceding installation procedure.

The preceding four lines are easy to follow. Let's discuss them one by one. The first step was quite simple—it tells the interpreter to import PyOpenGL along with its multiple classes, which can be used for different functions. Importing in such a way reduces the effort of importing each class of PyOpenGL one by one. The first import is mandatory as this line imports different OpenGL functions that begin with the `gl` keyword. For example, we can use a command such as `glVertex3fv()`, which can be used to draw different 3D shapes (we'll cover this later).

The next line of the import statement, that is, `from OpenGL.GLU import *`, is used so that we can use commands that start with `glu`, for example, `gluPerspective()`. These types of commands are useful in making changes to the view of the display screen, along with the objects it rendered. For example, we can make conversions such as cropping and clipping using such `glu` commands.

Similar to the PyOpenGL GL library, GLU is a Python library that is used to explore the relationships within or between related datasets. They are mostly used to make changes on the display screen while affecting the shapes and dimensions of the rendered objects. To learn more about the internals of GLU, check out its official documentation page: `http://pyopengl.sourceforge.net/pydoc/OpenGL.GLU.html`.

The next line simply imports `pygame` into our project. While the surface that was created using OpenGL is 3D, it needs the `pygame` module to render it. Before using any commands from the `gl` or `glu` modules, we need to call the `pygame` module to create a display using the `set_mode()` function (feel the power of the `pygame` module). The display that's created by the `pygame` module will be 3D rather than 2D while using the `set_mode` function with the OpenGL library. After this, we are telling the Python interpreter to create an OpenGL surface and return it as a `window_screen` object. The tuple (`height`, `width`) that's passed inside the `set_mode` function represents the surface size.

In the final step, I want you to focus on the optional parameters, which are as follows:

- `HWSURFACE`: It creates the surface in the hardware. It is primarily used for creating an accelerated 3D display screen, but it is only used in FULL SCREEN.
- `OPENGL`: It makes a suggestion to pygame regarding the creation of an OpenGL rendered surface.
- `DOUBLEBUF`: It stands for double buffering, and is recommended for `HWSURFACE` and `OPENGL` by pygame. It reduces the flickering (the phenomena of burning and shining colors in the screen unsteadily).

There are a few more optional parameters, as follows:

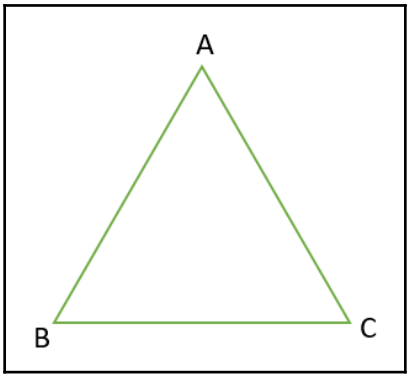
- `FULLSCREEN`: This will make the display of the screen rendered to a fullscreen view.
- `RESIZABLE`: This allows us to resize the window screen.
- `NOFRAME`: This will make the window screen borderless, controlless, and so on. For more information regarding pygame optional parameters, please go to `https://www.pygame.org/docs/ref/display.html#pygame.display.set_mode`.

Now that we have started installing PyOpenGL on our machine and set a window for screen objects, we can start drawing objects and primitives.

Making objects with PyOpenGL

OpenGL is primarily known for drawing different geometrical shapes or primitives, all of which can be used in the creation of scenes for a 3D canvas. We can make multiple-sided shapes (polygons), such as a triangle, quadrilateral, or hexagon. Several pieces of information, such as vertex and edges, should be given to the primitives so that PyOpenGL can render them accordingly. Since the information that's related to the vertex and edges is different for each shape, we have different functions to create different primitives. This is different compared to pygame's 2D function (`pygame.draw`), which was used to create multiple shapes using the same single function. For example, a triangle has three vertices and three sides, whereas a quadrilateral has four vertices.

If you have a mathematical background, knowledge of vertices and edges will be a piece of cake for you. But for those of you who are not, the vertices of any geometrical shapes are the corners or points in which two or more lines meet. For example, a triangle has three vertices. In the following illustration, **A**, **B**, and **C** are vertices of the triangle ABC. Similarly, edges are the line segments on the boundary joining one vertex to another. In the following triangle, AB, BC, and AC are edges of the triangle ABC:



To draw such geometrical shapes with PyOpenGL, we need to start by invoking some basic OpenGL primitives, which are listed as follows:

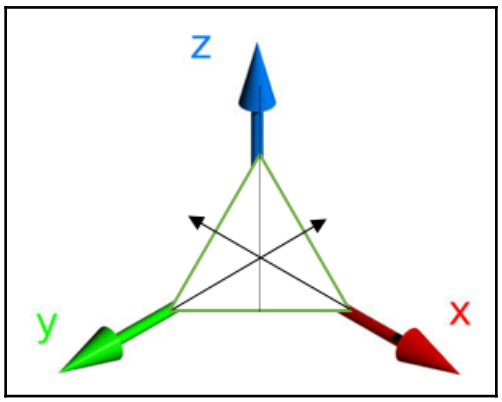
1. First of all, call the `glBegin()` function with any of the primitives you want to draw. For example, `glBegin(GL_TRIANGLES)` should be invoked to inform the interpreter about the triangular shapes we are going to draw.
2. The next piece of information regarding the vertices (A, B, C) is critical for drawing shapes. We send information regarding the vertices using the `glVertex()` function.

3. Apart from information about vertices and edges, you can provide additional information, such as the color of the shapes, using the `glColor()` function.
4. After providing enough essential information, you can invoke the `glEnd()` method to inform OpenGL that enough information has been provided. Then, it can start drawing the specified shapes, as indicated by the constants that are provided by the `glBegin` method.

The following code is the pseudocode for drawing triangular shapes using PyOpenGL (reference the preceding illustration to understand the operation of PyOpenGL functions):

```
#Draw a geometry for the scene
def Draw():
 #translation (moving) about 6 unit into the screen and 1.5 unit to left
     glTranslatef(-1.5,0.0,-6.0)
     glBegin(GL_TRIANGLES) #GL_TRIANGLE is constant for TRIANGLES
     glVertex3f( 0.0, 1.0, 0.0) #first vertex
     glVertex3f(-1.0, -1.0, 0.0) #second vertex
     glVertex3f( 1.0, -1.0, 0.0) #third vertex
     glEnd()
```

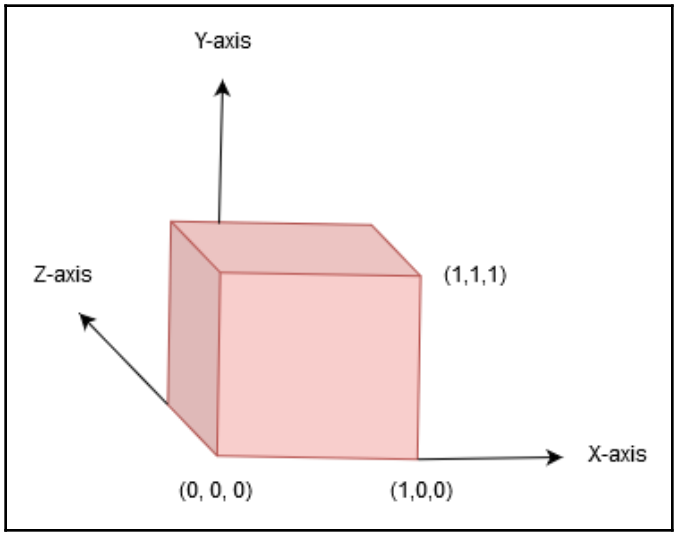
The following illustration shows the normal of the triangle. A normal is a mathematical term that means a unit vector (has a magnitude of 1 and has a direction—please refer to `Chapter 10`, *Upgrading the Snake Game with Turtle*, to find out more about vectors). This piece of information (normal) is essential because it tells PyOpenGL where each vertex resides. For example, `glVertex3f(0, 1, 0)` will put a vertex on the *y*-axis. Therefore, (*x*, *y*, *z*) represents the magnitude in the *x*-axis, *y*-axis, and *z*-axis, like so:



Now that we know how to create basic triangular primitives, let's take a look at the following table to understand the other different types of primitives that can be drawn using PyOpenGL:

Constants keywords	**Shapes**
`GL_POINTS`	Draws dots or points to the screen
`GL_LINES`	Draws lines (individual ones)
`GL_TRIANGLES`	Draws triangles
`GL_QUADS`	Draws quadrilaterals (four-sided polygons)
`GL_POLYGON`	Draws polygons (any edges or vertices)

We are now capable of drawing any primitives using primitive constants, provided that we have information about their vertices. Let's create the following quadrilateral:



The following is the pseudocode for drawing the preceding cube primitive:

```
glBegin(GL_QUADS)
glColor(0.0, 1.0, 0.0) # vertex at y-axis
glVertex(1.0, 1.0, 0.0) # Top left
glVertex(1.0, 1.0, 0.0) # Top right
```

```
    glVertex(1.0, 1.0, 0.0) # Bottom right
    glVertex(1.0, 1.0, 0.0) # Bottom left
    glEnd()
```

In the preceding line of code, we started by defining the `GL_QUADS` constants to inform PyOpenGL about the name of the primitives we are drawing. Then, we added the color attributes with the `glColor` method. Similarly, we defined the four primary vertices of the cube using the `glVertex` method. The coordinates that were passed as an argument to the `glVertex` method represent the *x*, *y*, and *z*-axes in the plane.

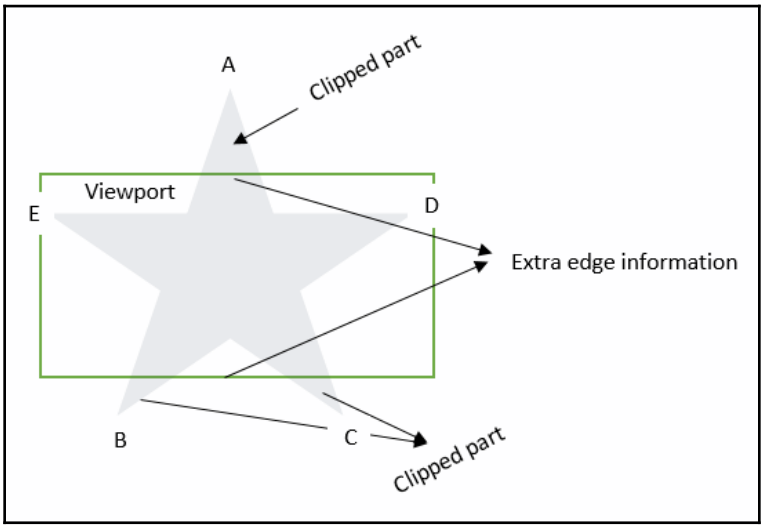
Now that we are able to draw different geometrical shapes using PyOpenGL, let's learn about the different rendering functions/premiers of PyOpenGL so that we can make other complex structures.

Understanding PyOpenGL methods

It is well-known that a computer screen has a two-dimensional view (its height and width). In order to display the three-dimensional scene created by OpenGL, the scene must go through several matrix transformations, which are commonly known as projections. This allows the 3D scene to be rendered in a 2D view. Among the various transformation methods, two are commonly used for projections (clipping and normalization). These matrix transformations are applied to the 3D coordinate system and reduced to a 2D coordinate system. The `GL_PROJECTION` matrix is frequently used for performing the transformation associated with projections. The mathematical deduction of projection transformation is another story and we are never going to use those, but understanding how it works is important for any game programmer. Let's go over how `GL_PROJECTION` works:

- **Clipping**: This transforms the coordinates of the vertex of the scene to the clip coordinates of the scene. Clipping is a process that resizes the length of the scene so that some parts are clipped from `viewport` (a window display).
- **Normalization**: This process is known as **Normalized Device Coordinates** (**NDC**), which transforms the clip coordinates into device coordinates by dividing by the `w` components of the clipping coordinates. For instance, the clip coordinates x_c, y_c, and z_c are tested by comparing with w_c. Vertices that does not lie in the range of $-w_c$ to $+w_c$ are discarded. Here the subscript *c* represents the clipping coordinate system.

Hence, it is easier to infer that the process of matrix transformation, including `GL_PROJECTION`, includes two steps: clipping, which is immediately followed by normalization to device coordinates. The following diagram illustrates how clipping is done:



We can clearly observe that the process of clipping (sometimes called culling) is only performed in the clipping coordinates, which are defined by the size of the 2D viewport. To find out which clip coordinates have been discarded, we need to look at an example. Let's assume that *x*, *y*, and *z* are clipping coordinates and that their values are compared with the coordinates of *w* (*x*, *y*), which decides whether any vertex (or part of shapes) remains in the screen or discarded. If any of the coordinates lie below the value of $-w_c$ and above the value of $+w_c$, that vertex is discarded. In the preceding diagram, vertex A lies above $+w_c$ while vertices B and C lie below $-w_c$, and so both vertices are discarded. Moreover, vertices D and E lie within the value of ($-w_c$, $+w_c$), and so they remain in the view. The value of w_c is determined by the width of the viewport. Hence, the projection matrix of OpenGL (`GL_PROJECTION`) takes the 3D coordinates and performs projection, which converts it into 2D coordinates that can be rendered into the 2D computer display screen. Although some information might be lost, it is considered one of the most effective methods of rendering 3D scenes into a 2D screen.

We are not done yet, though—after projection is performed, we have to convert the 3D scene into 2D, which requires the use of another OpenGL matrix transformation known as `GL_MODELVIEW`. The step of this transformation is, however, quite different. Firstly, matrix transformation is done, which multiplies the coordinate system by *view distance*.

To convert them into 2D components, z-components is provided for each of them. To understand the model-view matrix, we have to understand the two matrices that are a part of its composition: the model matrix and the view matrix. The model matrix performs several transformations such as rotation, scaling, and translations in the model world, whereas the view matrix adjusts the scene that's relative to the camera's position. The view matrix takes care of what the object looks like to the player who is watching the scene, something like the first player character screen/viewpoint.

Now that we are aware of the transformation matrices of OpenGL, let's make a simple program (`resize.py`) that can resize the display accordingly:

1. Start by importing OpenGL:

```
from OpenGL.GL import *
from OpenGL.GLU import *
```

2. Make a simple function, `change_View()`, that takes the size of the display screen, as follows:

```
def change_View():
    pass
```

3. The code that's stated in from *Step 3* to *Step 6* should be added inside the `change_View()` function. Add a function call to `ViewPort`, which takes the initial values and size of the display as follows:

```
glViewport(0, 0 , WIDTH, HEIGHT)
```

4. Now, it's time to add a projection matrix. To add `GL_PROJECTION`, we have to call the `glMatrixMode()` method, which checks the mode of matrices being called, as follows:

```
glMatrixMode(GL_PROJECTION) #first step to apply projection matrix
```

5. Immediately after applying the projection matrix, two important methods should be invoked, that is, `glLoadIdentity()` and `gluPerspective()`, which set the "touchstone" for the projection matrix:

```
aspect_ratio = float(width/height)
glLoadIdentity()
gluPerspective(40., aspect_ratio, 1., 800.)
```

6. After setting up the projection matrix, the next step is to set the model-view matrix. The model view matrix mode can be activated by calling the `GL_MODELVIEW` transformation matrix with the `glMatrixMode()` method:

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
```

The preceding six steps show us how we can resize the display screen in which a 3D scene is displayed in a 2D display screen. *Step 1* and *Step 2* are focused on importing openGL. In *Step 3*, we called the `glViewport()` method and passed an argument that ranges from (`0`, `0`) to (`width`, `height`), which informs OpenGL that we want to use the entire screen to display the scene. The next step calls the `glMatrixMode()` method, which tells OpenGL that every matrix transformation will apply the projection matrix in every successive function call.

Step 5 calls two new methods which, as the `glLoadIdentity()` signature states, are used to make the projection matrix identity, which means that all of the coordinates of the projection matrix should be changed to `1`. Eventually, we call another method, `gluPerspective()`, which sets the categorical/standard projection matrix. You may have noticed that the `gluPerspective()` method starts with `glu` and not `gl`; hence, this function is called from the GLU library. Four float arguments are passed with the `gluPerspective` method, that is, the field perspective of the camera viewpoint, the aspect ratio, and two clipping plane points (near and farther). Hence, clipping is done via the `gluPerspective` function. To observe how clipping is done, refer to the example of star geometrical shape that we discussed at the beginning of this topic.

Now, it's time to put what we've learned to the test by making a program that interacts with PyOpenGL structures. We will also define another attribute that will make objects more appealing. This is known as *color properties*. We will define a cube, along with mathematical information regarding vertices and edges.

Understanding color properties

In real-world scenarios, there are profuse amounts of colors associated with objects, but computer devices are not intelligent or capable enough to distinguish and capture all of them. Hence, to accommodate every possible color in digital form is nearly impossible. Due to this, scientists have provided us with a way to represent different colors: the *RGB* pattern. This is a combination of three major color components: red, green, and blue. Combining these components, we can create almost every color possible. The value of each component ranges from 0 to 255; changes to each component's code results in a new color.

The color properties that are used in OpenGL are quite similar to the real-world color reflection property. The color of the object that we observe is not actually its color; rather, it is the color that's reflected by the object. The object can have some properties of a wavelength, in which the object can absorb a certain color and might reflect a different one. For example, trees absorb sunlight except for green. We perceive and assume that it is green, but actually objects have no color. This concept of light reflection is fairly applied in OpenGL—we usually define a light source that might have a definite color code. Furthermore, we will also define the object's color code and then multiply it with the light source. The resultant color code or light is the result of reflection from the object, which is considered the color of the object.

In the case of OpenGL, color is given in the form of a tuple containing four components in which three are red, green, and blue. The fourth component represents alpha information, which indicates the level of transparency of the object. Instead of providing values of 0 to 255 for RGB components, we provide a value ranging from 0 to 1 in the case of OpenGL. For example, yellow is a combination of red and green, and so its alpha information is (1, 1, 0). Refer to `https://community.khronos.org/t/color-tables/22518` to find out more about OpenGL's color code.

The following functions/features are available in OpenGL's color properties:

- `glClearColor()`: This function sets a clear color, which means that it fills the color on a part of the area that hasn't been drawn. The value of the color code can be given as a tuple containing a value ranging from 0 to 1. For example, `glClearColor(1.0, 1.0, 1.0, 0.0)` represents filling with white.
- `glShadeModel()`: This function enables the lightening features of OpenGL. Usually, the argument that's passed to `glShadeModel` is `GL_FLAT`, which is used for shading the faces or edges of shapes such as cubes and pyramids. If you want to shade curved objects rather than faceted ones, you can use `GL_SMOOTH`.
- `glEnable()`: This is not actually a method related to color properties, but it is used to enable them. For instance, `glEnable(GL_COLOR_MATERIAL)` will enable *materials*, which allows us to interact with the surface and light source. Furthermore, by adjusting the settings, the properties of the materials are mostly used to make any object lighter and sharper.

Now that we are familiar with the concepts of color properties and ways of creating color attributes, let's make a simple program that will draw a cube using the color properties of PyOpenGL.

Brainstorming grids

Before we start to code, it's always good practice to brainstorm a little bit and acquire the necessary information so that we can create a program. Since we are going to create a program that will render a cube—a surface that has eight vertices, 12 edges, and six faces—we need to define such information explicitly. We can define each of these attributes as nested tuples—tuples inside a single tuple.

Taking one vertex as a reference, we can simultaneously get the positions of other vertices. Let's assume that a cube has one vertex at (`1`, `-1`, `-1`). Now, assuming that all of the edges of a cube have a length of 1 unit, we can get the coordinates of the vertices. The following code shows a list of the vertices of the cube:

```
cube_Vertices = (
    (1, -1, -1),
    (1, 1, -1),
    (-1, 1, -1),
    (-1, -1, -1),
    (1, -1, 1),
    (1, 1, 1),
    (-1, -1, 1),
    (-1, 1, 1),
    )
```

Similarly, there are 12 edges (edges are the lines that are drawn from one vertex to another). Since there are eight vertices (0 to 7), let's write some code that defines the 12 edges using eight vertices. The identifiers that are passed as tuples in the following code represent the edges or sides that are drawn from one vertex to another. For example, tuple (`0`, `1`) indicates the edge that was drawn from vertex 0 to vertex 1:

```
cube_Edges = (
    (0,1),
    (0,3),
    (0,4),
    (2,1),
    (2,3),
    (2,7),
    (6,3),
    (6,4),
    (6,7),
    (5,1),
    (5,4),
    (5,7),
    )
```

Finally, the last piece of information that must be provided is about surfaces. A cube has six faces, each of which contains four vertices and four edges. We can provide this information like so:

```
cube_Surfaces = (
 (0,1,2,3),
 (3,2,7,6),
 (6,7,5,4),
 (4,5,1,0),
 (1,5,7,2),
 (4,0,3,6)
 )
```

Note that the order in which the vertices, edges, and surface are provided matters. For example, in the `cube_Surfaces` data structure, if you swapped the second item of the tuple with the first one, the shape of the cube will deteriorate. This is because each piece of information is linked with vertex information, that is, surface (`0`, `1`, `2`, `3`) contains the first, second, third, and fourth vertices.

Now that we've finished brainstorming and gathered some useful information about the shape we are going to draw, it's time to start rendering the cube using PyOpenGL and its library, which is often referred to as the *GLU library*.

Understanding the GLU library

Now that we've collected information about the edges, sides, and vertices of our shape, we can start coding the model. We have already studied how we can draw shapes with OpenGL using methods such as `glBegin()` and `glVertex3fv()`. Let's use them and create a function that can draw a cube structure:

1. Start by importing OpenGL and the GLU library. Right after importing the library, add the information we acquired regarding the vertices, edges, and surfaces that we defined while brainstorming to the same file:

```
from OpenGL.GL import *
from OpenGL.GLU import *
```

2. Next, define the function and fetch the surfaces and vertex. This process is quite simple; we will start by drawing the surfaces for the cube. We should use the `GL_QUADS` property to draw four-sided surfaces (confused? Refer to the *Making objects with OpenGL* section of this chapter for more information):

```
def renderCube():
    glBegin(GL_QUADS)
    for eachSurface in cube_Surfaces:
        for eachVertex in eachSurface:
            glColor3fv((1, 1, 0)) #yellow color code
            glVertex3fv(cube_Surfaces[eachVertex])
    glEnd()
```

3. Finally, inside the `renderCube()` method, write some code that can draw a line segment. The `GL_LINES` parameter is used to draw a line segment:

```
glBegin(GL_LINES)
  for eachEdge in cube_Edges:
      for eachVertex in eachEdge:
          glVertex3fv(cube_Vertices[eachVertex])
  glEnd()
```

This three-line procedure is enough to create even complex geometrical shapes. Now, you can perform multiple operations on these cubes. For example, you can perform operations such as the rotation of objects by using mouse trackpads. As we know, handling such user events requires a `pygame` module. Hence, let's define a function that will take care of event handling, along with some of the characteristics of PyOpenGL. Begin your code with the `import pygame` statement and add the following code:

```
def ActionHandler():
    pygame.init()
    screen = (800, 500)
    pygame.display.set_mode(screen, DOUBLEBUF|OPENGL) #OPENGL is essential
    #1: ADD A CLIPPING TRANSFORMATION
    gluPerspective(85.0, (screen[0]/screen[1]), 0.1, 50)
    # 80.0 -> field view of camera
    #screen[0]/screen[1] -> aspect ration (width/height)
    #0.1 -> near clipping plane
    #50 -> far clipping plane
    glRotatef(18, 2, 0, 0) #start point
```

The preceding code snippet is quite simple to understand since we have been doing this from the beginning of this chapter. Here, we've used the `pygame` module, which sets the game screen with an OpenGL scene or interface. We have added a transformation matrix, which performs clipping using the `gluPerspective()` function. Finally, we added the initial position of the cube before actual rotation (where we might be at the beginning).

Now that we have addressed the basic primers of OpenGL, let's use pygame's event handling method to manipulate the structure of the cube, like so:

```
while True:

        for anyEvent in pygame.event.get():
            if anyEvent.type == pygame.QUIT:
                pygame.quit()
                quit()

            if anyEvent.type == pygame.MOUSEBUTTONDOWN:
                print(anyEvent)
                print(anyEvent.button) #printing mouse event
                #mouse button 4 and 5 are at the left side of the mouse
                #mouse button 4 is used as forward and backward navigation
                if anyEvent.button == 4:
                    glTranslatef(0.0,0.0,1.0) #produces translation
                      of (x, y, z)
                elif anyEvent.button == 5:
                    glTranslatef(0.0,0.0,-1.0)
```

After handling the events that are based on mouse button navigation, let's use some of the methods provided by PyOpenGL to render the cube. We will use methods such as `glRotatef()`, which will perform matrix transformation. Write the following code just after where we handled the events:

```
        glRotatef(1, 3, 1, 1)
#The glRotatef is used to perform matrix transformation which performs a
rotation
#of counterclockwise with an angle of degree about origin through the point
#provided as (x, y, z).
        #--------------------------------------------------------------------
        #indicates the buffer that needs to be cleared
        #GL_COLOR_BUFFER_BIT: enabled for color drawing
        #GL_DEPTH_BUFFER_BIT: depth buffer which needs to be cleared
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
        #render cube
        renderCube()
        pygame.display.flip()
        pygame.time.wait(12)
```

```
#call main function only externally
ActionHandler()
```

The highlighted part of the preceding code denotes a resize transformation, which ultimately leads to the ZOOM-UP and ZOOM-DOWN features being used. Now, you can run the program and observe the cube being rendered at the center of the pygame screen, in yellow. Try using an external mouse and using the navigation buttons (buttons 4 and 5) to zoom in and zoom out. You can also observe how clipping is used in the project: whenever we make a cube so big that it exceeds the clipping plane, some parts of the cube are removed from the viewport.

In this way, we can combine two powerful Python gaming modules, that is, *pygame* and *PyOpenGL*, to make 3D scenes and interfaces. We have only skimmed the ways of creating some shapes and how to transform them. Now, it's up to you to discover more about PyOpenGL and try to make a game that's much more user-friendly and attractive by providing rich textures and content.

Summary

We have covered many interesting topics in this chapter, mostly regarding surfaces and geometrical shapes. Although we used the term *matrix* in this chapter, we didn't bother performing matrix computation using a mathematical approach because Python has everything built-in to perform such operations. Still, we should remember the old adage, *game programmers don't need to have a PhD in mathematics*, since knowing a basic level of math is enough if we want to make games. Here, we only learned about translation, scaling, and rotation, which are enough if we want to make a 3D scene. We didn't get bogged down by learning about the concepts of translations or scaling using the mathematical approach—instead, we learned about using the programming approach.

We started off by learning how to set up the OpenGL display screen by using pygame's `setting` method. Since OpenGL is a vast and profound field of study, covering everything in a single chapter was impossible. Hence, we only covered how to load/store three-dimensional models and how to apply them to the OpenGL rendering surface by applying clipping, rotate, and resize transformations. We also studied color properties and used them with PyOpenGL and pygame. The main goal of this chapter was to make it easier for you to understand how to create 3D shapes using OpenGL 3D scenes while providing critical geometrical information such as vertices, edges, and surfaces. You will now be able to work with OpenGL to create 3D shapes, figures, and visuals. You now also know how to distinguish the color property of OpenGL from any other coloring patterns.

In the next chapter, we will learn about another important module, which goes by the name of *Pymunk*. This is a very powerful physics library that adds physics capabilities for game characters. We will learn about the different terms that are used when we need to talk about real-world environments, such as velocity and acceleration, which are used to handle collisions and the movement of game characters. While learning about these concepts, we will also be making an Angry Bird game, which we will deploy across various platforms.

15
Getting to Know Pymunk by Building an Angry Birds Game

Python, being a standalone language for half a decade in data science and machine learning, was not popular enough in the game development industry until open source packages such as pymunk evolved. These open source packages provided game developers with an easy interface for mimicking real-world environments through simulation, which allowed them to create single or multiple body objects that linked the player's input to physical impulses. This evolution brought the usage of a continuous physics model into Python game development, where some objects were allowed to rest for efficiency purposes and were only brought into the light with the principle of collision. With this model, we can handle multiple object collisions properly and efficiently.

By the end of this chapter, you will have learned about the fundamentals of Pythonic 2D physics library so that you know how to use classes and submodules to build complex games such as Angry Birds, which simulates the real-world environment by considering physical properties such as mass, motion, inertia, elasticity, and moment. You will also learn how to create 2D rigid bodies and link them to the player's input in order to simulate physical impulses. This will result in movement in the rigid bodies within the simulated environment (space). You will also learn how to use time interval steps (dt) by updating physical attributes that facilitate movement for rigid bodies within that space.

Up until now, you have been checking collisions between two game entities (in `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, you checked collisions between the snake and the boundary wall, while in `Chapter 12`, *Learning About Character Animation, Collision, and Movement*, you checked collisions between the bird and a vertical pipe), but this chapter will be more edifying in the sense that you will be checking collisions between three game objects one by one and performing actions by creating a collision handler.

The following topics will be covered in this chapter:

- Understanding pymunk
- Creating a character controller
- Creating a polygon class
- Exploring Pythonic physics simulation
- Implementing sling-action
- Addressing collisions
- Creating levels
- Handling user events
- Possible modifications

Technical requirements

You must have the following requirements to be able to complete this chapter:

- The Pygame editor (IDLE) version 3.5 or higher.
- The PyCharm IDE (refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, for the installation procedure).
- The `pymunk` module (an open source library that's available at `http://www.pymunk.org/en/latest/`).
- The code for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter15`
- An external link to the sprite sheets for angry birds: `https://www.spriters-resource.com/mobile/angrybirds/sheet/59982/`.

Check out the following video to see the code in action:

`http://bit.ly/2oG246k`

Understanding pymunk

In a real-world environment, objects move in ubiquitous directions arbitrarily. Thus, to mimic such movements, games must address the different physical behaviors of objects. For example, when we throw an object in the air, due to the presence of gravity, the object will hit the ground at some point. Similarly, we must also address the reduction in the velocity of objects every time an object bounces back from the surface. For example, if we were to take a ball and throw it in the air, after some time, it must hit the ground with the original velocity, V_o, and after one hit to the surface, it will bounce off the surface and then ascend with velocity V_f. Therefore, it is obvious that $V_o > V_f$. Implementing this kind of behavior for objects in a game environment will leave players with a good impression of the game.

Physics, being a branch of natural science, tries to simulate real-world actions through simulation and mathematical deduction. Different terminology is defined in physics, such as mass, inertia, impulse, elasticity, friction, and so on. These terminologies define the characteristics of objects when they're exposed to different environments. Without getting bogged down in the intricacies of physics, let's get down to business. The real question is, why do we need physics in games? The answer to this question is simple: similar to real-world objects, games also have objects/characters. These characters are governed by the players of the game. Most players love to play a game that simulates a real-world phenomenon.

Some physical terms you must understand before using the `pymunk` module are as follows:

- **Mass**: Literally, mass refers to the weight of any object. While considering its physical definition, the mass of an object is a measure of the amount of matter in the object.
- **Force**: A force is a push or a pull upon an object resulting from the object's *interaction* with another object.
- **Gravity**: The force that causes, for example, an apple to fall toward the ground. Gravity is the force that attracts two bodies to each other.
- **Elasticity**: A property of deformed objects where they get reshaped and go back to their original form. For example, a spring and a rubber band will go back to their original shape, even if force is applied to distort them.
- **Moment**: A moment of force is a property that causes an object to rotate around a specific point or axis.

If you have not played Angry Birds before, make sure to check out this link: `http://freeangrybirdsgame.org/play/angry_birds_online.html`. While playing the game, observe the number of characters, structures, and catapult actions.

It would be boring if both of the characters in our Angry Birds game (Bird and Pig) had horizontal movements. For instance, when the player shoots an Angry Bird from the catapult or material slingshot, what if it doesn't follow projectile motion (45-degree motion) and just goes in a horizontal direction (90-degree motion)? This violates one of the laws of physics which states that *Earth pulls down on you*. Maybe we could argue that's why this is a big deal. Violating such laws would make games asinine and absurd, which might hamper the reputation of the game. In order to simulate such real-world physics in games, the Python community has developed a 2D physics library. We can employ different characteristics for game objects such as mass, inertia, impulse, and friction using this library.

First of all, I recommend that you check out the official documentation of pymunk at `http://www.pymunk.org/en/latest/pymunk.html`. Since the packages and modules of `pymunk` are frequently updated, you will see a huge amount of resources on their official documentation page. Just don't get overwhelmed by how many there are—we will need only a few of them to make a game that uses the pymunk 2D physics library.

Now that you have gone through the documentation, I assume that you might have seen several submodules and classes. We will need some of them, all of which we will discuss. We will start with pymunk, which is the most popular and widely used submodule. It's named `vec2d`. To observe the working of `vec2d`, you have to brush up on your basics, which we learned about in `Chapter 9`, *Data Model Implementation*. To recap, we used different data models to implement vector manipulation (we used `__add__()` to add vectors, `__str__()` to format vectors, and so on). We've already learned about vector manipulation, but in a Pythonic way; now, let's learn about it in a modular way. The Python developer community has created a submodule for `vec2d`; that is, the `Vec2d` class, in order to perform any vector-related manipulation.

Before looking at an example of the `Vec2d` class, let's set up our PyCharm project first. Open the PyCharm editor and create a new project. I will call it *Angry Bird*. After providing a name for the project, press the **Create** button to create a project. When PyCharm is ready with your project, create a new Python file called `test.py`. Before writing any code, we have to install the `pymunk` module in the current project.

Follow these steps to do so (to get a detailed description of how to install any third-party library in PyCharm, refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*):

1. Click on **File | Settings**. The **Settings** window will open.
2. On the left-hand side tab, click on the **Project: Angry Bird** tab. It will list all the modules that have been installed in the Python interpreter.
3. To add a new module, click on the (+) button next to the **Packages** tab.
4. Search for `pymunk` and install the module (make sure your internet connection is up and running).

Now that the `pymunk` module has been successfully installed, let's get back to the `Vec2d` class. As we mentioned previously, this class can be used to perform vector manipulation. It is an alternative to using data models for vector manipulation. Let's look at a simple example of creating a vector using the `Vec2d` class:

```
from pymunk.vec2d import Vec2d
print(Vec2d(2, 7))

#add two vectors
print(Vec2d(2, 7) + Vec2d((3, 4)))

#results
Vec2d(2, 7)
Vec2d(5, 11)
```

Apart from performing mathematical computations, `Vec2d` can also perform different high-level functional computations. For instance, if you want to find the distance between two vectors, we can call the `get_distance()` function, as follows:

```
print(Vec2d(3,4).get_distance(Vec2d(9,0)))
7.211102550927978
```

The preceding function calculates the distance between two vectored points using the formula √(x2 – x1)^2 + (y2 – y1)^2, where (x1, y1) and (x2, y2) are two vectored coordinates. To learn more about the distance formula, please go to `https://www.purplemath.com/modules/distform.htm`.

Now that we have explored `Vec2d`, we will learn about `pymunk` classes. There are more than 10 of them but we will only learn about the important ones. You can explore them by going to their official documentation pages. Let's learn about them one by one.

Exploring pymunk's built-in classes

To begin, we will start with the `Space` class. This class refers to the placeholders where all your game characters will reside. The movement of the game characters will also be defined within this space. The properties of rigid objects (with physical properties such as mass, friction, elasticity, and inertia) will change in this space as we begin to progress in the game. For example, an object in a different space will have different velocity and acceleration. In the case of the Angry Birds game, the velocity of an Angry Bird will differ from when the player initially slung it from the catapult to it then colliding with the structures in the game (beams and columns, which we will cover in a minute).

There are many methods defined inside the `pymunk` modules, so we will start with the most important one: `add_collision_handler(collision_type_a, collision_type_b)`. Recall from `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, that you made a Snake game and added the collision handler by yourself, adding some logic that implies that *When the position of two objects is the same, they are said to have collided*. This method is a way of doing the same thing in an easier way, which is just by calling the `pymunk` built-in function. This collision handler that's made by `pymunk` will take two arguments: `type_a` and `type_b`. You must remember that these two types are integers. We will use them to define two objects explicitly. For example, in the Angry Birds game, there will be three main characters: Bird, Wood, and Pig (to download the required assets, check the GitHub link mentioned in the *Technical requirements* section). Since we have three characters, we have to add a collision handler for each of them, like so:

- **When Bird and Pig collide**: We will call `add_collision_handler(0, 1)`, where `0` indicates the integer type of the Bird character and 1 represents the integer type of the Pig game character.
- **When Bird and Wood collide**: We will call `add_collision_handler(0, 2)`, where `2` indicates the integer type of the Wood game character. (Remember that, throughout the game, 0 must represent the Bird character and must not be used for any other character).
- **When Pig and Wood collide**: We will call `add_collision_handler(1, 2)`.

By doing this, we will get to feel the power of the collision handler defined inside the `Space` class. This function checks whether two objects collide and returns `CollisionHander` for collisions between objects represented by `type_a` and `type_b`.

Now that we have learned about handling collisions in pymunk, we will learn about two of the most important and most used classes of the `pymunk` module: `Body` and `Shape`. Firstly, we will start by learning about the pymunk `Body` class and its properties. Then, we will explore the pymunk `Shape` class, where we will learn how to add different physical properties such as elasticity, mass, and moment to geometrical figures.

Exploring the pymunk Body class

When making complex games such as Angry Birds, we have to define multiple game characters, such as the Bird, Pig, and Wood structures. The following illustration provides a visual of these game characters:

All of these are images (in the sense of Pygame, they are sprites). They can't be used directly until and unless we convert them into rigid bodies. The way that Pygame defines physical measurements (mass, motion, friction, and impulse) means that it will convert these sprites into rigid bodies. Here comes the power of the `Body` class: the `Body` class takes any shape (circular, polygon, sprites, and so on) and injects properties such as mass, moment, force, and many more, like so:

```
import pymunk
space = pymunk.Space() #creating Space instance
body = pymunk.Body() #creating Body instance
object = pymunk.Circle(body, 4)
object.density = 2
#print body measurements
print("Mass : {:.0f} and Moment: {:.0f}".format(body.mass, body.moment))

space.add(body, object)
print("Mass: {:.0f} and Moment: {:.0f}",format(body.mass, body.moment))
```

The result of the preceding code is as follows:

```
Mass : 0 and Moment: 0
Mass: 101 and Moment: 804
```

In the preceding code, we started by defining `space`. As we mentioned previously, `Space` is a class that represents the placeholder for the objects. Take a close look at the `space.add(body, object)` statement: we have used the `add()` method to add the object to the `space`. Similarly, we made an instance of the `Body` class. The `Body` class does not necessarily mean the objects or game characters; rather, it is a virtual place where we can add game characters. The `object = pymunk.Circle(body, 4)` statement will create a circular object with a radius of `4` units and will add it to the scope of `Body`. After creating the circular objects, we added density (the intensive property of the object: mass per unit volume occupied by an object; please refer to the following link to learn more about density: `https://www.nuclear-power.net/nuclear-engineering/thermodynamics/thermodynamic-properties/what-is-density-physics/`).

After adding the `density` property to the objects, we printed the two bodies: the first one when the body was not added into the space and another circular body (along with `density`) added to the space. We printed both bodies. As expected, the first bodies were not into space and we didn't define any properties for that body, and so its mass and moment were displayed as zero. Similarly, after the body was added to `space`, their mass and moment changed to 101 and 804 standard units, respectively.

Now, let's learn about another important class of the `pymunk` module, which goes by the name of `Shape`.

Exploring the pymunk Shape class

There are three different class categories that come under the `Shape` class: `Circle`, `Poly`, and `Segment`. However, learning about the `Shape` class itself is enough for us to understand these categories. Let's learn about a few important physical properties (all in lowercase) that we can call upon the shapes from the following points:

- `copy()`: Performs the deep copy of the current shape.
- `density`: The density of shapes. An extremely important property that calculates the mass and moment of inertia of a body from which shapes are attached. We looked at an example of this in the *Exploring* the *pymunk Body class* section.
- `elasticity`: Defines the elasticity of a shape. This property is used to define the bouncing nature of shapes. If the elasticity value is 0, that shape cannot bounce. For a perfect bounce, the value of elasticity should be 1.

- `friction`: Defines the `friction` coefficient for a shape. A `friction` value of `0` defines a frictionless surface while `1` defines a perfectly fine (no rough) surface.
- `mass`: Defines the weight for a shape. When `mass` is higher, the object cannot bounce and move freely.
- `moment`: Calculates the moment for a shape.

To observe the application of the preceding properties, we don't create instances of the `Shape` class. Instead, we use the `Circle`, `Poly` and `Segment` classes.

The `Circle` class (which we used in the previous section) can be instantiated like so:

```
pymunk.Circle(body, radius_of_circular_shape)
```

Properties such as density, elasticity, friction, mass, and moment can be also defined in the case of circular objects. We will see an example of this while making the Angry Birds game.

Similarly, we can create a polygon shape using the `Poly` class. The following syntax represents the creation of instances using the `Poly` class:

```
pymunk.Poly(body, vertices, transform = None, radius = 0)
```

In the preceding line of code, `body` is the instance of the `Body` class which represents the virtual space for the shape. The `vertices` argument defines the vertices for the convex hull of the polygon. A convex hull is calculated by the `Poly` class using vertices automatically. The remaining two arguments, *transform* and *radius*, are optional. `transform` is an object of the `Transform` class (refer to `http://www.pymunk.org/en/latest/pymunk.html#pymunk.Poly` to find out more about `transform`), which applies the transform to each vertex of the polygon, while the `radius` argument sets the radius of the created poly shape.

You may be wondering what the application of the `Poly` class will be while making the Angry Birds game. In this game, we have two main characters, as well as the wood structures, consisting of beam and column, which are made using the `Poly` class. More on this will be discussed when we start making the Angry Birds game.

Finally, we have another useful class, known as the `Segment` class. Let's explore how its instance is created:

```
pymunk.Segment(body, point1, point2, radius)
```

As its name suggests, the `Segment` class is responsible for defining a line segment shape between two points: `point1` and `point2`. It is an important class since it defines the surface for the game. The `radius` argument defines the thickness of the line segment drawn from `point1` to `point2`. Several aforementioned properties, such as `mass`, `density`, `elasticity`, and `friction` can also be added to this shape. Mostly, friction is used to define the roughness of the surface while creating the surface of the game. Even in the Angry Birds game, we can create a game surface using the `Segment` class and associate the body with some level of friction (0—1), which defines the level of fineness and roughness of the surface. The value of 0 represents 100 percent fine, while 1 represents totally rough.

Now that we are fully equipped with all the classes and properties associated with the `pymunk` module, we can start coding our Angry Birds game.

Creating a character controller

If you haven't played Angry Birds yet, I highly encourage you to do so. Search for Angry Birds online and play it for a few minutes. While playing the game, observe the main characters (bird and pig), their actions, and their interaction with wooden structures. The wooden structures are made of different beam and column structures where a different number of wooden structures are dovetailed, one after the other.

After you've taken a look at the original game, you can start coding your own Angry Birds game. We made the **Angry Bird** project previously, in PyCharm, while installing the `pymunk` module. We will use the same project folder to create this game. Create a new Python file and name it `characters.py`.

In this **Angry Bird** project, we are not going to write whole pieces of code within a single file. While coding complex games such as Angry Birds, it is important for us to create different modules for different tasks. Doing so, we can easily find bugs while testing our game. In this Angry Birds game, we will create four Python files: `characters.py`, `polygon.py`, `main.py`, and `level.py`.

The first file, which we just created, will contain the main game characters: Bird and Pig. The wooden beam and column structures will be created in the next file; that is, `polygon.py`. But for now, let's concentrate on the `characters.py` file.

The `characters.py` file will contain two classes: one for `Bird` and another for `Pig`. Then, we will define several attributes that govern the movement, that is, the physical property, for each of these classes. The following code represents the content of the `characters.py` file:

```
import pymunk as p #aliasing pymunk as p
from pymunk import Vec2d #for vector manipulation
```

After importing the necessary modules, let's define the class for the `Bird` character (Angry Bird movement is handled by the player who is playing the game):

```
class RoundBird():
    def __init__(self, distance, angle, x_pos, y_pos, space):
        weight = 5
        r = 12 #radius
        value_of_inertia = p.moment_for_circle(weight, 0, r, (0, 0))
        obj_body = p.Body(weight, value_of_inertia)
        obj_body.position = x_pos, y_pos
        power_value = distance * 53
        impulse = power_value * Vec2d(1, 0)
        angle = -angle
        obj_body.apply_impulse_at_local_point(impulse.rotated(angle))
        obj_shape = p.Circle(obj_body, r, (0, 0))
        obj_shape.elasticity = 0.95 #bouncing angry bird
        obj_shape.friction = 1 #for roughness
        obj_shape.collision_type = 0 #for checking collisions later
        space.add(obj_body, obj_shape)
        #class RoundBird attribute ----
        self.body = obj_body
        self.shape = obj_shape
```

In the preceding line of code, we defined all the physical and positional attributes for the Angry Birds character. We start by defining the constructor. The arguments for the constructor are as follows:

- `distance` between the two body positions, usually calculated by the distance formula (`https://www.purplemath.com/modules/distform.htm`) and passed to the `Bird` class.
- `angle` in degrees to perform the movement of the `Bird` character.
- `x_pos`, `y_pos` represents the position of `Bird`.
- `space` represents the `space` object where `Bird` is rendered.

Inside the constructor, we have added multiple physical attributes to the `Bird` character. For example, `elasticity= 0.95` represents the bouncing capability (standard), `friction = 1` (level of roughness of surface), power = work done (distance) * time (53). The mass (weight) of the Bird is 20, and the `birdLife` class attribute represents the quantity that reduces whenever the Bird character collides with the ground or other characters (Pig or the wooden structures).

The values of friction, elasticity, and work done are not random (I didn't use them arbitrarily). They are defined on the official documentation page. Refer to the following URL to explore the chart: `http://www.pymunk.org/en/latest/pymunk.html#pymunk.Shape`.

The two important methods of the `Bird` class (highlighted in the preceding code) are the built-in functions defined by the `pymunk` module. The first method, `moment_for_circle()`, calculates the moment of inertia (the resistance of any physical object to any change in its velocity) for the hollow circle. The argument that's passed to the function is the *mass* of the object; that is, the *inner radius* and the *outer radius*. Observe the inner radius, which is passed as `0`, which means the Angry Bird (the main character of the game is a circular solid circle). If the inner radius is `0`, it means this is a solid circular object. The outer radius defines the circular dimension of the Angry Bird. Similarly, observe the `collision_type = 0` attribute. This statement will add the integer type to the Bird game character. When checking collisions between two objects with `add_collision_handler(type_a, type_b)`, we use this collision type value to indicate that the `0` value for the character is `Bird`. For the `Bird` character, we have a collision type equal to `0`. The `Pig` class will have its collision type defined as `1`.

Similarly, the next method, `apply_impulse_at_local_point(impulse, point = (0, 0))`, will apply a local impulse to the body. This, in turn, will represent how much the momentum of the angry bird will change when force is provided. Refer to `https://study.com/academy/lesson/impulse-definition-equation-calculation-examples.html` to learn more about impulse and momentum.

Next, we need to define the class for the `Pig` character. The following code should be written just after the `Bird` class:

```
class RoundPig():
    def __init__(self, x_pos, y_pos, space):
        self.life = 20 #life will be decreased after
          collision of pig with bird
        weight = 5
        r = 14 #radius
        value_of_inertia = p.moment_for_circle(weight, 0, r, (0, 0))
        obj_body = p.Body(weight, value_of_inertia)
```

```
        #creates virtual space to render shape
          obj_body.position = x_pos, y_pos
          #add circle to obj body
          obj_shape = p.Circle(obj_body, r, (0, 0))
          obj_shape.elasticity = 0.95
          obj_shape.friction = 1
          obj_shape.collision_type = 1
          space.add(obj_body, obj_shape)
          self.body = obj_body
          self.shape = obj_shape
```

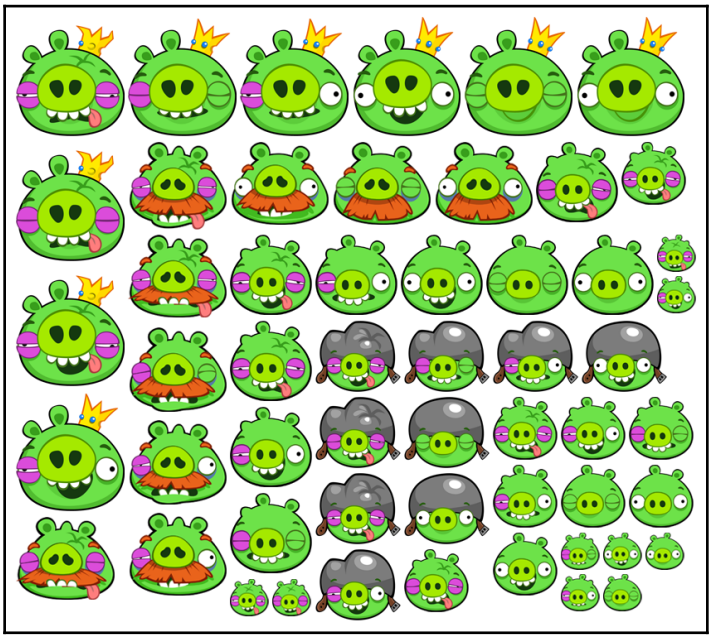
The preceding code is similar to the `Bird` class. Like before, we defined the same level of elasticity and friction as the `Pig` character. We added the inertia and mass effects to the object. For the `Pig` character, the `collision_type` is added as `1`, which means that while checking the collision between Pig and Bird, we can simply call `add_collision_handler(0, 1)`, where `0` represents Bird and `1` represents Pig.

Now that we have created two main classes for the Angry Birds game, that is, `RoundBird` and `RoundPig`, inside the `characters.py` file, we will create another game character, that is, the wooden structures (beams and columns).

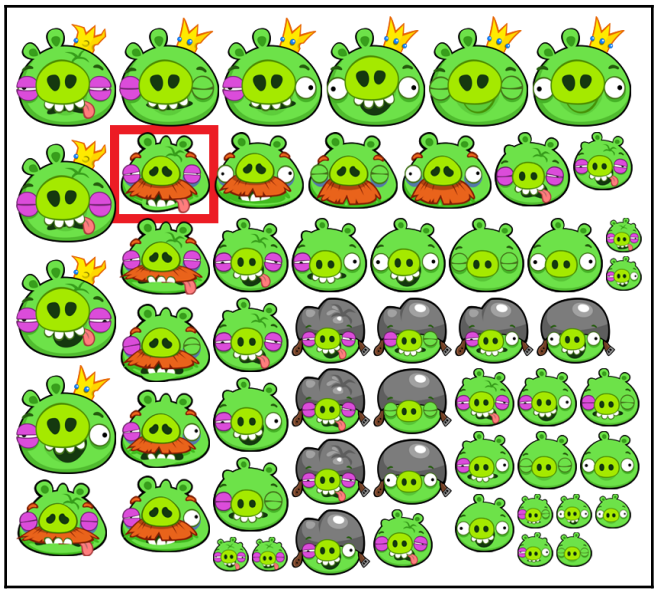
Creating the Polygon class

For each of the game entities, we have created separate classes, that is, Bird and Pig. Since our final game entity is a wooden structure (that the player shoots at with the slingshot), we will make a different Python file and create a class for this entity. But before that, let's go through one of the important concepts regarding sprite sheets.

Images that are used in Python game development are usually called sprites, and they are the static images on which some manipulation (vectored movement) is done based on the user's actions (such as moving the snake when clicking the arrow keys on the keyboard). In the preceding chapters (`Chapter 12`, *Learning About Character Animation, Collision, and Movement*, and `Chapter 13`, *Coding the Tetris Game with Pygame*), we used sprites (single images), but not sprite sheets (sheets containing multiple static images). The following is an example of a sprite sheet, and is specific to our Angry Birds game:



These image files generally don't contain a single image of a game character. As you can see, they usually contain a large number of distinct game characters. But most of the time, we will only require a single image from the entire sprite sheet. Thus, the question is, how can we extract a single image from such sprite sheets? We do so using the `Rect` class of the `Pygame` module. Do you remember the `Rect` class (`Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*) from the Pygame module? This class creates a rectangular object based on the left, top, width, and height dimensions. To extract an image from the aforementioned sprite sheet, we will draw a rectangle around one of the sprites, as follows:



This mapping is done with the help of the `Rect` class. The `Rect` class will create a rectangle with the dimensions of four points on the screen (left, top, width, and height). In this way, by altering any four dimensions of the `Rect` object, we can extract the part or sub-surface of sprite sheets.

Now, let's see this in action by creating a wooden structure. To begin, download the sprite assets from the following GitHub link: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter15/res`. You will see various images, along with code assets. There will be two folders inside the `res` folder: one for photos and another for sound. You have to copy the entire folder and paste it into the **Angry Bird** project folder in the PyCharm editor.

After you've imported the resources, I suggest that you open the `wood.png` file. This file contains different wooden structures. When creating a polygon out of these wooden structures, we have to crop one of the images using the `Rect` class.

In the same **Angry Bird** project, create another Python file with the name `polygon.py`. We will start by importing the necessary modules:

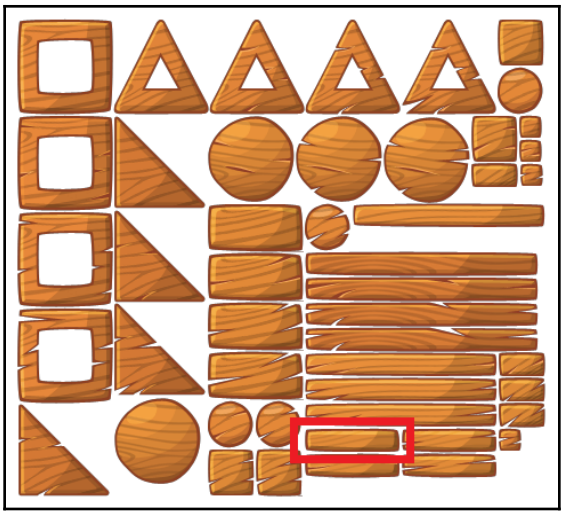
```
import pymunk as pym
from pymunk import Vec2d
import Pygame as pg
import math
```

Now, let's create the `Polygon` class:

```
class Polygon():
    def __init__(self, position, length, height, space, mass=5.0):
        value_moment = 1000
        body_obj = pym.Body(mass, value_moment)
        body_obj.position = Vec2d(position)
        shape_obj = pym.Poly.create_box(body_obj, (length, height))
        shape_obj.color = (0, 0, 255)
        shape_obj.friction = 0.5
        shape_obj.collision_type = 2 #adding to check collision later
        space.add(body_obj, shape_obj)
        self.body = body_obj
        self.shape = shape_obj
        wood_photo =
          pg.image.load("../res/photos/wood.png").convert_alpha()
        wood2_photo =
          pg.image.load("../res/photos/wood2.png").convert_alpha()
        rect_wood = pg.Rect(251, 357, 86, 22)
        self.beam_image = wood_photo.subsurface(rect_wood).copy()
        rect_wood2 = pg.Rect(16, 252, 22, 84)
        self.column_image = wood2_photo.subsurface(rect_wood2).copy()
```

The attributes we've defined for the `Polygon` class are quite similar to what we did for the `Bird` and `Pig` classes: we initialized friction and added the `collision_type` so as to reference the Polygon shape with an integer of `2`. The constructor takes an argument, that is, `position`, to tell us about the position of the polygon to render, the `length` and `height` of the polygon, the `space` object where the polygon will be rendered, and the `mass` for the polygon shape.

The only novel thing in the preceding code is the highlighted part of the code. We have loaded the `wood.png` and `wood2.png` images into the Python project using Pygame's `load` method. The `convert_alpha()` method acts as an optimizer and will create a new image surface that is suitable for quick blitting. The `Rect` class takes four dimensions to create a rectangular surface (refer to `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*). The dimensional values that are provided aren't given randomly and represent the values that cover the subsurface of the sprite sheets that we need to extract. For example, the `self.beam_image = wood.subsurface(rect).copy()` command will extract the horizontal beam image (the piece of wood enclosed by a red rectangle) from the `wood.png` file, as follows;



Now that we have extracted horizontal and vertical wooden images (a beam and a column, respectively), we can start drawing a polygon containing them. However, there's a problem. Since, although we have been using Pygame and `pymunk` together, their coordinate systems are not alike: `pymunk` uses a coordinate system with the origin at the bottom left, while Pygame, as you probably already know, uses a coordinate system with the origin at the upper left. Thus, we will make a function that will convert the `pymunk` coordinate system into a compatible Pygame coordinate system:

```
def convert_to_pygame(self, pos):
    """Function that will transform pymunk coordinates to
         Pygame coordinates"""
    return int(pos.x), int(-pos.y+610)
```

The preceding function is important because the game surface will be made out of the `Pygame` module. Thus, we must track the position where the beam and column must be rendered. Now, let's start drawing the `polygon` into the surface:

```
def draw_poly(self, element, screen):
    """Draw beams and columns"""
    polygon = self.shape

    if element == 'beams':
        pos = polygon.body.position
        pos = Vec2d(self.convert_to_pygame(pos))
        angle_degrees = math.degrees(polygon.body.angle)
        rotated_beam = pg.transform.rotate(self.beam_image,
                                           angle_degrees)
        offset = Vec2d(rotated_beam.get_size()) / 2.
```

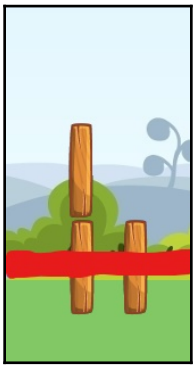
```
            pos = pos - offset
            final_pos = pos
            screen.blit(rotated_beam, (final_pos.x, final_pos.y))
```

The preceding function will be used to place a beam on the screen in which an object is passed as an argument to it. The first argument to the function is *element*, which tells the function which polygon to draw: is it a beam or a column? We will add some logic to the draw column in the following code, but for now, let's observe what we have written so far. The code starts by getting the *shape* object. Then, we check whether the element is `beam`. If it is `beam`, then we get the position of the image and convert it into the `Vec2d` coordinate position. The highlighted part of the code (getting the angle to rotate the beam image) will ensure that the image of the beam is within the red rectangular (virtual) area, as follows:

Just remove that highlighted line from the preceding code and observe the result. You will see that the beam won't be perfectly aligned because of the offset of the `Vec2d` coordinate system. Similarly, let's add some code so that we can draw the column to the screen:

```
if element == 'columns':
    pos = polygon.body.position
    pos = Vec2d(self.convert_to_pygame(pos))
    angle_degrees = math.degrees(polygon.body.angle) + 180
    rotated_column = pg.transform.rotate(self.column_image,
                                         angle_degrees)
    offset = Vec2d(rotated_column.get_size()) / 2.
    pos = pos - offset
    final_pos = pos
    screen.blit(rotated_column, (final_pos.x, final_pos.y))
```

In the preceding code, the first couple of lines will convert the `pymunk` coordinates into Pygame. Since column should be rendered in the Pygame surface, this conversion is necessary. Similarly, after getting the position coordinates, we take an angle of coordinate and make sure to add 180 or 0 to it so that it remains an original image with no rotation. After getting that image, we transform it and create a new image as a `rotated_column` image. Remember that if the rotating angle is not a multiple of 90, the image will be distorted. In the preceding line of code, if `offset` is not removed from the rotated image, the image will move down the surface, as shown in the following screenshot:



In the preceding screenshot, the red line represents the surface. Thus, if you do not remove the offset from the column's body position, the column will be displayed below the surface.

Now that we have finished the `Polygon` class, which will render either a beam or a column whenever the `draw_poly()` function is called from the main class, it's time to make our main class, which is the director of all the classes. This class will be responsible for creating instances of all the classes and for calling the methods that are defined inside different classes to render game objects into the Pygame game surface.

Exploring Pythonic physics simulation

To begin, let's start by revising what we have done so far. We started by defining two main game entities: `Bird` and `Pig`. All of the major physics properties, such as mass, inertia, and friction, were defined for each of these characters so as to simulate real-world physics. After creating the two major game characters, we made another Python file so that we could create the `Polygon` class. This class was created to render the wooden structures in the game with the help of beam and column. Now, we are going to create another Python file with the name `main.py`. This will be the main controller of the game.

Use the following code to declare the base physics in the `main.py` file. We will start by importing some essential modules:

```
import os
import sys
import math
import time
import Pygame
import pymunk
from characters import RoundBird #our characters.py file have Bird class
```

After importing the necessary modules, we need to crop some subsurfaces from the sprites that we added previously. Obviously, we don't want everything from the sprite sheets, and so we will be extracting only parts of them to create game characters. However, since our main character, our Angry Bird, only has a single image and is not present in the sprite sheets, we don't need to crop the image for the Angry Bird and the slingshot. However, for the `Pig` character, we have to create a `Rect` object because the `Pig` images are bundled together in a sprite sheet. Thus, we will have to load the images by using the following code:

```
Pygame.init()
screen = Pygame.display.set_mode((1200, 650))
redbird = Pygame.image.load(
 "../res/photos/red-bird3.png").convert_alpha()
background_image = Pygame.image.load(
 "../res/photos/background3.png").convert_alpha()
sling_image = Pygame.image.load(
 "../res/photos/sling-3.png").convert_alpha()
full_sprite = Pygame.image.load(
 "../res/photos/full-sprite.png").convert_alpha()
rect_screen = Pygame.Rect(181, 1050, 50, 50)
cropped_image = full_sprite.subsurface(rect_screen).copy()
pig_image = Pygame.transform.scale(cropped_image, (30, 30))
#(30, 30) resulting height and width of pig
```

In the preceding code, we started by defining a game screen using the Pygame module. After that, we loaded all the images that exist as a single image but not the sprite sheets, such as `red-bird3.png`, `background3.png` and `sling-3.png`. As we mentioned previously, the image of the pig is part of a bundle of images in `full-sprite.png`. Since we need only one image of the pig, we will perform a similar process the one we carried out while extracting beam and column. We will create a `Rect` object with the exact dimensions of the pig's shape and then use it to extract a pig image from the sprite sheets. Then, we will crop that image and store it as a cropped object, which will eventually be transformed so that is has a height and width of `30`, `30`, respectively.

Now that we have extracted the necessary images for the game objects, let's get down to business by declaring the physical variables and positional variables for each of these objects:

```
running = True
#base physics code
space_obj = pymunk.Space()
space_obj.gravity = (0.0, -700.0)
```

As we know, the Angry Birds game is played by using a mouse to stretch the catapult in a slinging motion. Thus, we have to declare some variables that will take care of these sling actions:

```
mouse_distance = 0 #distance after stretch
rope_length = 90

angle = 0
mouse_x_pos = 0
mouse_y_pos = 0

mouse_pressed = False
time_of_release = 0

initial_x_sling, initial_y_sling = 135, 450 #sling position at rest (not
stretched)
next_x_sling, next_y_sling = 160, 450
```

In the preceding code, we have defined different variables so that we can track the position of the mouse before and after the sling action. The `sling_action()` function, which we will declare afterward, will manipulate these values. For now, let's create a list that will track the number of pigs, birds, beams, and columns that are displayed in the space:

```
total_pig = []
total_birds = []
beams = []
columns = []
#color code
WHITE = (255, 255, 255)
RED = (255, 0, 0)
BLACK = (0, 0, 0)
BLUE = (0, 0, 255)
```

Now that we have defined all the necessary variables for the Angry Birds game (we will add more variables later if needed), it's time to create a surface for the screen. This surface is not a background surface; instead, it's some ground in which all the structures will reside. The Angry Bird will also bounce off this surface, and so we have to add some physical properties to this ground, as follows:

```
# Static floor
static_floor_body = pymunk.Body(body_type=pymunk.Body.STATIC)
static_lines_first = [pymunk.Segment(static_floor_body, (0.0, 060.0),
(1200.0, 060.0), 0.0)]
static_lines_second = [pymunk.Segment(static_floor_body, (1200.0, 060.0),
(1200.0, 800.0), 0.0)]

#lets add elasticity and friction to surface
for eachLine in static_lines_first:
    eachLine.elasticity = 0.95
    eachLine.friction = 1
    eachLine.collision_type = 3
for eachLine in static_lines_second:
    eachLine.elasticity = 0.95
    eachLine.friction = 1
    eachLine.collision_type = 3
space_obj.add(static_lines_first)
```

The preceding line of code will create some static ground. While instantiating the static body, we can explicitly set `body-type` as `STATIC` by adding the `pymunk.Body.STATIC` constant. After defining the static body, we have to use the `Segment` class to create a line segment between one point and another (recall the `Segment` class from the *Exploring the pymunk Space class* section). For each line segment, we have added `elasticity` to support the bouncing property, `friction` to indicate the roughness, and `collision_type` to check whether other game objects have collided with the ground surface or not, which will be checked later, in the *Checking collision* section. After creating these static surfaces, we will add them to the `Space` object, which will render them onto the screen.

After defining the static surface, we need to define the sling action, that is, what happens when the player stretches the rope of the catapult. We will implement this in the next section.

Implementing the sling action

In this section, we are going to implement the sling action. The player is going to interact with the game character through the sling action. But before implementing the sling action, we have to take care of a few things: how far can the player stretch the rope of the catapult? What is the angle of impulse (the track of motion after the player releases the rope)? What is the distance between the mouse action point and the rope's current stretch point? All of these things must be addressed by declaring functions. First of all, we need to convert pymunk coordinates into Pygame coordinates so that we can align the game objects with the screen properly (the reason for this conversion was discussed in the *Creating the Polygon class* section).

The following function will convert the `pymunk` coordinates to Pygame coordinates:

```
def convert_to_pygame(pos):
    """ function that performs conversion of pymunk coordinates to
        Pygame coordinates"""
    return int(pos.x), int(-pos.y+600)
```

Although pymunk's *x*-coordinates are the same as Pygame's *x*-coordinates, due to pymunk's origin being at the bottom left, we have to change it to the upper left. Similarly, let's define another function, that is, `vector`, which will convert the passed point into a vector. The following code represents the implementation of the `vector` function:

```
def vector(a, b):
    #return vector from points
    p = b[0] - a[0]
    q = b[1] - a[1]
    return (p, q)
```

Refer to `Chapter 9`, *Data Model Implementation*, to find out more about how vectors are created using positional vectors. Here, the arguments *a* and *b* represent the points that are converted into vectors from the reference point. Now that we have created a vector, let's define a function that will return the distance between two points:

```
def distance(x0, y0, x1, y1):
    """function to calculate the distance between two points"""
    dx = x1 - x0
    dy = y1 - y0
    dist = ((dx ** 2) + (dy ** 2)) ** 0.5
    return dist
```

The preceding code will calculate the distance formula between two points, that is, `(x0, y0)` and `(x1, y1)`, using the `sqrt((x1 - x0) + (y0 - yo))` distance formula, where `sqrt` represents `square root (math.sqrt(4) = 2)`. The ** operator represents the power. For example, `dx ** 2` is equivalent to $(dx)^2$.

Now that we've calculated the distance, we need to learn how to calculate unit vectors. A unit vector is a vector that has a magnitude of 1. We don't really care about the magnitude, but the significance of a unit vector is that it tells us about the direction of the vector. Once we have a unit vector, we can amplify it by any factor to achieve the new vector in that particular direction. While creating the sling action, having knowledge about the unit vector is important as this will give us information about the direction the catapult is stretched in. To find the unit vector in the same direction as a vector, we have to divide it by its magnitude. Using mathematical deduction, let's build a function and create a unit vector:

```
def unit_vector(v):
    """ returns the unit vector of a point v = (a, b) """
    mag = ((v[0]**2)+(v[1]**2))**0.5
    if mag == 0:
        mag = 0.000000000000001
    unit_p = v[0] / mag #formula to calculate unit vector:
vector[i]/magnitude
    unit_q = v[1] / mag
    return (unit_p, unit_q)
```

In the preceding code, the value of h is determined by the `sqrt(a^2 + b^2)` magnitude formula. To find the unit vector, each component of the vector (`v[0]`, `v[1]`) is divided by the magnitude (`mag`).

Now that we have declared different functions to define the position, magnitude, and direction for the sling action, we can begin to define the method that performs the sling action. The following picture represents the catapult, which has two ends but no rope attached to it:

Here, our main task will be to add bird (the main character) to this catapult and to define the position for it. Let's start by defining some globals inside `sling_action`:

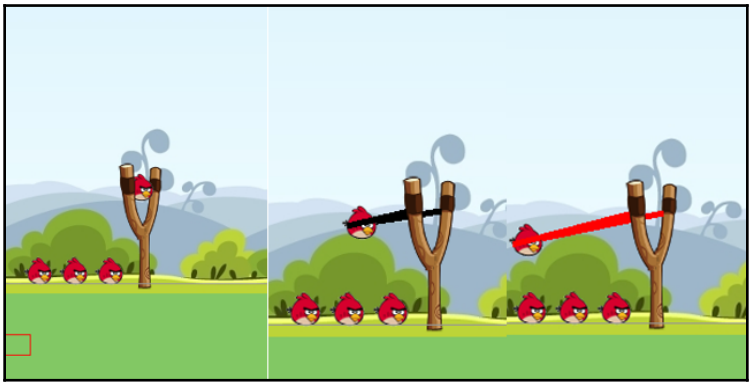
```
def sling_action():
    """will Set up sling action according to player input events"""
    global mouse_distance
    global rope_length
    global angle
    global mouse_x_pos
    global mouse_y_pos
```

In the preceding line of code, we have declared some globals. However, these attributes were initialized with some initial values at the beginning of the *Exploring pythonic physics simulation* section. This means we will have to do some manipulation to update the values of these variables. The `mouse_distance` variable will contain the distance value from the point where the catapult is at rest to the point where the player stretches the rope of the catapult. Similarly, `rope_length` represents the length of the rope when it's been stretched by the player. The angle represents the angle of impulse, which is calculated as a slope angle. The slope for the rope of the catapult represents how steep the rope is when it's stretched by the player. `mouse-x-pos` and `mouse-y-pos` represent the current position of the mouse when the rope of the catapult is stretched.

Now, we have to address three things in this `sling_action` function:

1. Add the Angry Bird to the rope of the sling (as shown in the following screenshot).
2. Make the bird stay on the rope, even when the rope of the sling is stretched.
3. Address a situation where the rope of the sling is fully stretched.

To understand what these events are, take a look at the following picture:



Now, let's address all of the preceding actions in the `sling_action` function:

```
#add code inside sling_action function
""" Fixing bird to the sling rope (Addressing picture 1)"""
vec = vector((initial_x_sling, initial_y_sling), (mouse_x_pos,
mouse_y_pos))
unit_vec = unit_vector(vec)
uv_1 = unit_vec[0]
uv_2 = unit_vec[1]

mouse_distance = distance(initial_x_sling, initial_y_sling, mouse_x_pos,
mouse_y_pos)
#mouse_distance is a distance between sling initials point to the point at
which currrent bird is

fix_pos = (uv_1*rope_length+initial_x_sling,
uv_2*rope_length+initial_y_sling)
highest_length = 102 #when stretched
```

The preceding code will create a view for the Angry Birds character in the sling action. First of all, the `v` vector is created by the two coordinate points (`sling_original`, `mouse_current`) for example, ((2, 3), (4, 5)), where (2, 3) represents the sling at the static position or the center point of the sling, while (4, 5) represents the position when the mouse action is activated by the player. We will create a unit vector from this vector to understand the direction of stretch made by the player. Then, we will calculate `mouse_distance`, which will be calculated by calling the previously defined `distance()` function. This distance represents the distance from the static sling's center to the current mouse position. The (`mouse_x_pos`, `mouse_y_pos`) value represents the final position for the bird after the rope has been stretched. The `uv_1` and `uv_2` unit vectors will ensure that the bird will remain on the rope, which is indicated by the mouse's position. For example, if the mouse pointer is pointed upward, the rope and the bird will stretch in an upward direction.

Similarly, let's address the second scenario, that is, making an Angry Bird remain on the rope, even when the rope has been fully stretched. We will implement it in the following code:

```
#to make bird stay within rope
x_redbird = mouse_x_pos - 20
y_redbird = mouse_y_pos - 20
if mouse_distance > rope_length:
    pux, puy = fix_pos
    pux -= 20
    puy -= 20
    first_pos = pux, puy
    screen.blit(redbird, first_pos)
    second_pos = (uv_1*highest_length+initial_x_sling,
```

```
uv_2*highest_length+initial_y_sling) #current position ==> second_pos

    Pygame.draw.line(screen, (255, 0, 0), (next_x_sling, next_y_sling),
        second_pos, 5)
    #front side catapult rope
    screen.blit(redbird, first_pos)
    Pygame.draw.line(screen, (255, 0, 0), (initial_x_sling,
       initial_y_sling), second_pos, 5)
    #ANOTHER SIDE of catapult
```

A lot is going on in the preceding code, but the actions are easier and more mathematical. You must try to understand the logic rather than trying to understand the syntax. Let's delve into the code and uncover the reason behind each line of code. We start by decrementing the mouse position by 20 units to make sure that, while stretching, the bird remains at the edge of the rope. Try changing this value to 40 and observe the effect. Next, we checked whether `mouse_distance` is greater than `rope_length` to make sure that the distance of the stretch is within the limit. We don't want the mouse distance to be greater than the maximum rope length. In this situation, we will take the mouse distance and decrease it until and unless it comes under the maximum length of rope.

After that, we will `blit` the redbird (Angry Birds image) at the end of the rope. Similarly, we have to `blit` the rope too. In the preceding picture, observe the rope pull where the rope has turned red. This red color will be created if we `blit` the rope from the center of the static sling to the maximum possible rope length. Observe the bold part of the code; we have drawn a line that represents the rope with the color code (255, 0, 0), that is, red. There are two statements for this: one on each side. Thus, we have implemented the condition where the user will stretch the rope to its maximum defined length.

Now, we have to address the third and last scenario, that is, what happens when the player stretches the rope to its maximum length? In the preceding line of code we checked `if mouse_distance > rope_length`, and thus if the player stretches to less than `rope_length`, it should be addressed in the `else` part of the code, as follows:

```
else:
    #when rope is not fully stretched
    mouse_distance += 10
    third_pos = (uv_1*mouse_distance+initial_x_sling,
      uv_2*mouse_distance+initial_y_sling)
    Pygame.draw.line(screen, (0, 0, 0), (next_x_sling, next_y_sling),
      third_pos, 5)
    screen.blit(redbird, (x_redbird, y_redbird))
    Pygame.draw.line(screen, (0, 0, 0), (initial_x_sling,
       initial_y_sling), third_pos, 5)
```

Similar to the previous code, we make the distance no less than 10, which means that when the user even slightly stretches the rope, its `mouse_distance` will be equal to or more than 10. Then, we create `third_pos` to define the position in which to render the rope and the Angry Bird. `uv_1` and `uv_2` are unit vectors that indicate the direction of stretch. After getting the position, we `blit` the Angry Bird and then draw a line to indicate the rope. This will be in black and will be done on the front and the back.

Now that we have defined the scenario for all our cases, let's add a line of code to calculate the angle of impulse. This angle will be made whenever there is stretch in the rope. The tan (angle of impulse) is equal to the slope of the stretched rope. The slope is defined as the rise by run or (`dy`/`dx`), where `dy` is the change in `y` and `dx` is the change in `x`. Thus, the angle of impulse can be calculated as $\tan^{-1}$`(dy / dx)`. To learn more about the origins and application of this formula, check out `https://www.intmath.com/plane-analytic-geometry/1b-gradient-slope-line.php`.

Let's use this formula to calculate the angle of impulse, as follows:

```
#this is angle of impulse (angle at which bird is projected)
change_in_y = mouse_y_pos - initial_y_sling
change_in_x = mouse_x_pos - initial_x_sling
if change_in_x == 0:
    #if no change in x, we make fall within the area of sling
    dx = 0.00000000000001
angle = math.atan((float(change_in_y))/change_in_x) #tan^-1(dy / dx)
```

The preceding angle of impulse will be necessary to determine the path of the Angry Bird after the sling action is performed.

Finally, we have completed the sling action. Now, let's hop over to the next section, where we will address a collision between two game objects.

Addressing collisions

To recap, answer the following question: When do we know when two game objects have collided? Do you have your answer? Whenever two objects are in the same location within the coordinate system, they are said to have collided. However, in the case of pymunk, we don't have to check whether a collision has occurred or not. Rather, a single method call will check this for us. For example, a call to `space.add_collision_handler(0, 1)` will add a collision handler to check whether there has been a collision between the Bird and Pig characters. Here, the `0` integer represents the `collision_type` that's defined inside the `Bird` class. The `collision_type` that's defined for the `Pig` class is `1`. Thus, these `collision_type` must be unique so that each game entity can identify them uniquely.

Although we have an easier method for adding a handler to check collisions, the program still asks for the details; that is, what happens when two game objects collide? What actions must be performed? This is resolved by using `post_solve`. We will explicitly tell the collision handler that if there is a collision between *X* and *Y*, then a specific method should be called; for example, `space.add_collision_handler(0, 1).post_solve = perform_some_action`.

Let's define each of these actions whenever there is a collision between game objects. We will start by defining an action that must be performed whenever there is a collision between Bird and Pig. Let's write a function to do this:

```
def post_solve_bird_pig(arbiter, space_obj, _):
    """Action to perform after collision between bird and pig"""

    object1, object2 = arbiter.shapes #Arbiter class obj
    bird_body = object1.body
    pig_body = object2.body
    bird_position = convert_to_pygame(bird_body.position)
    pig_position = convert_to_pygame(pig_body.position)
    radius = 30
    Pygame.draw.circle(screen, (255, 0, 0), bird_position, radius, 4)
      #screen => Pygame surface
    Pygame.draw.circle(screen, RED, pig_position, radius, 4)
    #removal of pig
    pigs_to_remove = []
    for pig in total_pig:
        if pig_body == pig.body:
            pig.life -= 20 #decrease life
            pigs_to_remove.append(pig)

    for eachPig in pigs_to_remove:
        space_obj.remove(eachPig.shape, eachPig.shape.body)
        total_pig.remove(eachPig)
```

In the preceding code, the method takes an object of the Arbiter class: `arbiter`. The `arbiter` object will encapsulate all the colliding objects/shapes and even store all of the colliding object's positions. Since the game objects are drawn into the Pygame screen, we need to know their exact location in terms of the Pygame coordinate system. Thus, a conversion is made from pymunk coordinates into Pygame coordinates. Similarly, the process that we defined for the `post_solve` function is to address the action that must be performed instantaneously after the collision between Pig and Bird. The action will reduce the health of pig and then eventually remove it from the space. The `space.remove()` statement will remove the game objects from the screen.

Similarly, let's define another action that must be performed after the collision between Bird and the wooden structure. Similar to the preceding code, after a collision, the wooden beams and columns must be removed from the space or screen. The following function will address such actions:

```
def post_solve_bird_wood(arbiter, space_obj, _):
    """Action to perform after collision between bird and wood structure"""
    #removing polygon
    removed_poly = []
    if arbiter.total_impulse.length > 1100:
        object1, object2 = arbiter.shapes
        for Each_column in columns:
            if object2 == Each_column.shape:
                removed_poly.append(Each_column)
        for Each_beam in beams:
            if object2 == Each_beam.shape:
                removed_poly.append(Each_beam)
        for Each_poly in removed_poly:
            if Each_poly in columns:
                columns.remove(Each_poly)
            if Each_poly in beams:
                beams.remove(Each_poly)
        space_obj.remove(object2, object2.body)
        #you can also remove bird if you want
```

Similar to before, the `arbiter` object will hold information about the colliding shapes and positions. Here, the `total_impulse` attribute will return the impulse that was applied to resolve the collision. To find out more about the `Arbiter` class, go to `http://www.pymunk.org/en/latest/pymunk.html`. Now, after getting the collision's impact, we will check whether `arbiter` has a shape of either `beam` or `column` since the arbiter object will contain the list of the collided object. After looping through the `beam` and `column` stored inside the `arbiter` object, we will remove it from the space.

Finally, we will address the last collision—an action that must be implemented when `Pig` has collided with the wooden structure. Let's add a method to implement it:

```
def post_solve_pig_wood(arbiter, space_obj, _):
    """Action to perform after collision between pig and wood"""
    removed_pigs = []
    if arbiter.total_impulse.length > 700:
        pig_shape, wood_shape = arbiter.shapes
        for pig in total_pig:
            if pig_shape == pig.shape:
                pig.life -= 20

                if pig.life <= 0: #when life is 0
```

```
                removed_pigs.append(pig)
    for Each_pig in removed_pigs:
        space_obj.remove(Each_pig.shape, Each_pig.shape.body)
        total_pig.remove(Each_pig)
```

Similar to the previous two methods, this function will also check the content of the `arbiter` object, which is responsible for encapsulating all the information regarding the shape of collided objects and the position at which the collision happened. Using the content of the `Arbiter` class object, we have checked the length of after impact and then either removed or decreased the life unit of the `Pig` character.

The next step is to add a collision handler. Since we have declared all the `post_solve` actions that must be performed after a collision between two objects, let's add it to the collision handler using `post_solve`, as follows:

```
# bird and pigs
space.add_collision_handler(0, 1).post_solve=post_solve_bird_pig
# bird and wood
space.add_collision_handler(0, 2).post_solve=post_solve_bird_wood
# pig and wood
space.add_collision_handler(1, 2).post_solve=post_solve_pig_wood
```

After adding the collision handler, all we need to do is add an event handler that handles the events of the player who's playing the game. But before that, it is easier to work on the levels. What I really mean by level is to create a structure using beams and columns. Although we extracted the beams and columns from the sprite sheets, we never created a structure out of them. Let's create some wooden structures using beams and columns.

Creating levels

Not only have we created three major game entities, but we have also made a collider handler and `sling_action` function. But we aren't done yet. We have to add wooden structures to the space with the help of `beam` and `column` game objects. `beam` is a horizontal wooden rectangular structure while `column` is a vertical wooden rectangular structure. In this section, we'll create another class and define a level for the game by defining different wooden structures. You will have to create a new Python file and name it `level.py`. In that file, start writing the following code to define the wooden structures:

```
from characters import RoundPig #HAVE TO ADD PIG IN STRUCTURE
from polygon import Polygon #POLYGON
```

After importing the essential modules, we can start creating a `Level` class:

```
class Level():
    #each level will be construct by beam, column, pig
    #will create wooden structure
    def __init__(self, pigs_no, columns_no, beams_no, obj_space):
        self.pigs = pigs_no #pig number
        self.columns = columns_no
        self.beams = beams_no
        self.space = obj_space
        self.number = 0 #to create build number
        self.total_number_of_birds = 4 #total number of initial bird
```

In the aforementioned code, we have created a `Level` class with a constructor that takes `pigs`, `columns`, `beams`, and `space` as arguments. These arguments must not be foreign to you. All of these represent the objects of different classes. Similarly, we initialized the class variable using a constructor. The use of the `number` attribute will be discussed in a minute. It won't make sense describing its usage until and unless we use it. There is another attribute with a `total_number_of_birds` signature, which represents the number of Angry Birds that must be available for the player to project with the catapult. Now, let's build the first level for the game:

```
def build_0(self):

    pig_no_1 = RoundPig(980, 100, self.space)
    pig_no_2 = RoundPig(985, 182, self.space)
    self.pigs.append(pig_no_1)
    self.pigs.append(pig_no_2)
    pos = (950, 80)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (1010, 80)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (980, 150)
    self.beams.append(Polygon(pos, 85, 20, self.space))
    pos = (950, 200)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (1010, 200)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (980, 240)
    self.beams.append(Polygon(pos, 85, 20, self.space))
    self.total_number_of_birds = 4
```

In the preceding code, we have arranged `beam` and `column` in a window fashion (one layer on top of another). We also added two pigs inside the structure. To create such beams and columns, we have to create instances of the `Polygon` class (which we made in the *Creating the Polygon class* section). Although the code that's written inside the function seems lengthy, no novel logic is created here. We have just instantiated different beams and columns and provided a position to render. The value of `pos` is a tuple that represents the position in the space where the polygon should be placed.

Now, let's create another method inside the same `level.py` file and call this level `0`. Remember that this is the method of the `Level` class:

```
def load_level(self):
    try:
        level_name = "build_"+str(self.number)
        getattr(self, level_name)()
    except AttributeError:
        self.number = 0
        level_name = "build_"+str(self.number)
        getattr(self, level_name)()
```

Finally, here is the application of the number attribute that we initialized while creating the constructor of the class. This `load_level()` method will perform string concatenation to build the function name that represents `level_levelNumber`. For example, the highlighted part of the preceding code will yield `build_name = "build_0"` `[initially number = 0] and getattr(self, "build_0")()`, which is equivalent to `build_0()`.

`get_attr(object, p)` is equivalent to `object.p`. This method is important if you feel that there might be an Attribute Error exception. For example, `get_attr(object, p, 10)` will return 10 if there is an exception. Thus, this method can be used to provide a default value. Attribute Error occurs when an attribute with the given name doesn't exist in the object.

Since this `load_level()` method should be called explicitly from a file, we will do this in the `main.py` file. Open your `main.py` file and then continue with the code from where we left off. Write the following code to call the recently made `load_level()` method:

```
#write it in main.py file
from level import Level
level = Level(total_pig, columns, beams, space)
level.number = 0
level.load_level()
```

In the preceding line of code, we import the `Level` class from the `level` module. We create an instance of the `Level` class by passing a list of `pig`, `columns`, `beams`, and `space`. Similarly, we assign an initial value of `number` equal to `0`, which means that the beginning of the `build_0` method should be called by the `load_level()` method. You can increment the value of `number` by adding more difficult levels.

Now that we've loaded the level into our `main.py` file, it's high time to handle user action events. We will handle mouse events using Pygame in the next section.

Handling user events

In this section, we are going to handle user events. This won't be new to you. Ever since `Chapter 5`, *Learning About Curses by Building a Snake Game*, we have been handling user action events in various cases. While building the snake game, we handled keyboard events, and for Flappy Bird, we handled mouse tapping events. While handling those events, we found that the easiest and most universal way of doing this was by using the `pygame` module; it was just one line of code where we had to listen for the incoming actions and handle them accordingly.

But in the case of Angry Birds, handling mouse actions is a little bit tricky. Problems arise when we take the mouse action beyond the scope of space and try to perform a sling action. This must not be allowed, and so we have to check whether or not the mouse action should be associated with the sling action (the previously created function that pulls the rope of the catapult). Thus, let's learn how to handle the input events of the user by writing the following code:

```
while running:
    # handle Input events
    for eachEvent in Pygame.event.get():
        if eachEvent.type == Pygame.QUIT:
            running = False
        elif eachEvent.type == Pygame.KEYDOWN and event.key ==
          Pygame.K_ESCAPE:
            running = False
```

Now that we've checked for `QUIT` action events, we can get to mouse event handling (when the user uses the mouse to project the Angry Bird from the catapult):

```
if (Pygame.mouse.get_pressed()[0] and mouse_x_pos > 100 and
        mouse_x_pos < 250 and mouse_y_pos > 370 and mouse_y_pos < 550):
    mouse_pressed = True
if (event.type == Pygame.MOUSEBUTTONUP and
        event.button == 1 and mouse_pressed):
```

```
    # Release new bird
    mouse_pressed = False
    if level.number_of_birds > 0:
        level.number_of_birds -= 1
        time_of_release = time.time()*1000
        x_initial = 154
        y_initial = 156
```

In the preceding code, we start off by checking whether the mouse action is within scope. We check whether the mouse click is within the scope of space of `(mouse_x_pos > 100 and mouse_x_pos < 250 and mouse_y_pos > 370 and mouse_y_pos < 550)`. If it is, we will assign a Boolean of `True` to the `mouse_pressed` variable.

Next, we will perform the action to release the bird from the catapult or sling. After releasing each bird, we check whether any other birds are left or not. If there is, we decrease the number of birds by one and assign the value of *x-initial, y-initial* = 154, 156, respectively. These values are the center coordinates of the sling when the sling is at rest. Now, when the sling is stretched, there will be a new value, which we will call `mouse-x-pos`, `mouse-y-pos`. Remember that we don't have to calculate the distance from (`mouse_x_pos`, `mouse_y_pos`) to (`x-initial`, `y-initial`) because we did this while creating the `sling_action` function. Thus, we will use the `mouse_distance` we calculated there to perform the bird release action:

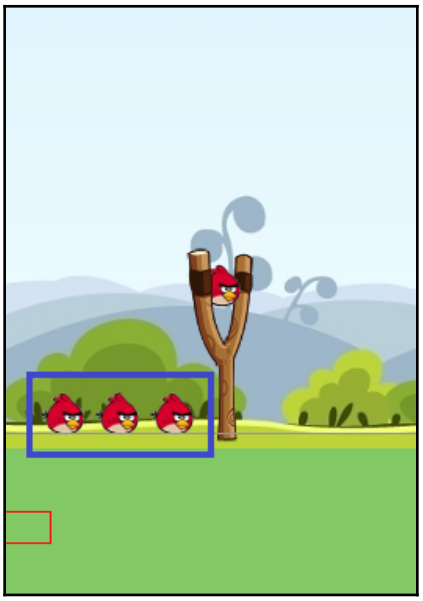
```
#add code after x-initial and y-initial declaration
if mouse_distance > rope_length:
    mouse_distance = rope_length
if mouse_x_pos < initial_x_sling+5:
    bird = RoundBird(mouse_distance, angle, x_initial, y_initial,
           space_obj)
    total_birds.append(bird)
else:
    bird = RoundBird(-mouse_distance, angle, x_initial, y_initial,
           space_obj)
    total_birds.append(bird)
if level.number_of_birds == 0:
    game_finish_time = time.time()
```

In the preceding code, we are adding the current `Bird` object that's attached to the rope to the birds list. This list will provide us with information about the current bird distance from the center of the catapult, the angle of impulse, and the `space` object. Now that we have handled the input actions of the player, let's `blit` every object into the space with the following code:

```
mouse_x_pos, mouse_y_pos = Pygame.mouse.get_pos()
# Blit the background image
```

```
screen.fill((130, 200, 100))
screen.blit(background_image, (0, -50))

# Blitting the first part of sling image
rect = Pygame.Rect(50, 0, 70, 220)
screen.blit(sling_image, (138, 420), rect)

# Blit the remaining number of angry bird
if level.total_number_of_birds > 0:
    for i in range(level.total_number_of_birds-1):
        x = 100 - (i*35)
        screen.blit(redbird, (x, 508))
```

In the preceding code, we got the current mouse position (the position in the space for the mouse action). Then, we drew the background with the background image that we loaded previously. Similarly, we `blit` the sling image into the screen. Now, we have to `blit` the Angry Birds that are waiting in line to be placed in the sling, as shown in the following screenshot:



Since `total_number_of_birds` is an attribute that's defined inside the `Level` class, we have to use it by creating an instance of it. Until and unless the number of birds is greater than 0, we create a list representing the number of birds. In the `for` loop code, we have to decrease the number of birds by 1 because one bird will be in the sling. After getting the actual number of remaining birds, we have to get the position to render these birds into the space. Although the *y*-position (height) is constant, that is, 508 units, the *x*-position is calculated by providing a space between each of them by `i*35` units, where `i` represents the iterables that were created by the `for` loop. For example, for bird number 2, the position in the space will be (2*35, 508).

Now, we will call the sling action. When the mouse is pressed within the scope and the bird possesses some angle of impulse in the space, we have to call the `sling_action` method using the following code:

```
# Draw sling action checking user input
if mouse_pressed and level.total_number_of_birds > 0:
    sling_action()
else: #blit bird when there is no stretch of sling
    if time.time()*1000 - time_of_release > 300 and
      level.number_of_birds > 0:
        screen.blit(redbird, (130, 426))
```

If we have `mouse_pressed` and the number of birds is greater than 0, we perform the sling action; otherwise, we just `blit` the bird in the position (`130`, `426`). In the `else` part of the code, we do not perform a sling action. The way to determine whether the sling action must be performed or not is by observing whether the mouse has been pressed or not (released) and the `time_of_release` after the release. If the current time has a significant difference, we do not perform the sling action. If there are differences of a significant amount, that means the bird hasn't been released. In order to release the bird, the current time must be equal to `time_of_release`. This is the case when we `blit` redbird in the sling just before release.

After performing `sling_action`, we can track the number of birds and pigs that must be removed from the scope with the following code:

```
removed_bird_after_sling = []
removed_pigs_after_sling = []
# Draw total_birds
for bird in total_birds:
    if bird.shape.body.position.y < 0:
        removed_bird_after_sling.append(bird)
    pos = convert_to_pygame(bird.shape.body.position)
    x_pos, y_pos = pos
    x_pos -= 22 #Pygame compatible
```

```
    y_pos -= 20
    screen.blit(redbird, (x_pos, y_pos)) #blit bird
    Pygame.draw.circle(screen, BLUE,
                       pos, int(bird.shape.radius), 2) #creates blue circle
                                                        at the edge of bird
```

In the highlighted part of the code, we check whether the bird hits the ground. If it does, that means we have to append the bird to the `removed_bird_after_sling` list. Similarly, we get the Pygame coordinates for the bird character and `blit` it in the (`x_pos`, `y_pos`) position. A blue circle is made around the bird after the impact.

Similarly, we have to remove birds and pigs after impact. Write the following code to implement this:

```
# Remove total_birds and total_pig
for bird in removed_bird_after_sling:
    space_obj.remove(bird.shape, bird.shape.body)
    total_birds.remove(bird)
for pig in removed_pigs_after_sling:
    space_obj.remove(pig.shape, pig.shape.body)
    total_pig.remove(pig)
```

Similarly, let's draw pigs into the space:

```
# Draw total_pig
for Each_pig in total_pig:

    pig = Each_pig.shape
    if pig.body.position.y < 0: #when pig hits ground or fall to the ground
        removed_pigs_after_sling.append(pig)

    pos = convert_to_pygame(pig.body.position) #pos is a tuple
    x_pos, y_pos = pos

    angle_degrees = math.degrees(pig.body.angle)
    pig_rotated_img = Pygame.transform.rotate(pig_image, angle_degrees)
    #small random rotation within wooden frame
    width,height = pig_rotated_img.get_size()
    x_pos -= width*0.5
    y_pos -= height*0.5
    screen.blit(pig_rotated_img, (x_pos, y_pos))
    Pygame.draw.circle(screen, BLUE, pos, int(pig.radius), 2)
```

After the pig hits the ground, we have to add it to the `removed_pigs_after_sling` list. We get the position of the body using Pygame coordinates. Similarly, we perform a transformation on the `pig` objects. The rotation transformation is within 0.5 units. This auto transformation will make the pig move smoothly in the space without remaining static. If you change the value of rotation to more than 2 units, the pig's position will be drastically deteriorated.

Two primary game entities have already been rendered into the space; that is, pig and bird. Now, it's time to add some other game entities to the game screen; that is, beam and column. We previously made a `beam` and `column` list to track the number of beams and columns. Let's use it to render structures in the game:

```
# Draw columns and Beams
#beam and column are object of Poly class
for column in columns:
    column.draw_poly('columns', screen)
for beam in beams:
    beam.draw_poly('beams', screen)
```

Now, it's time to update the physics: how fast the bird should travel after the sling action, and how many updates per frame are to be established for the stability of the game. First of all, let's define the time step's length:

```
time_step_change = 1.0/50.0/2.
```

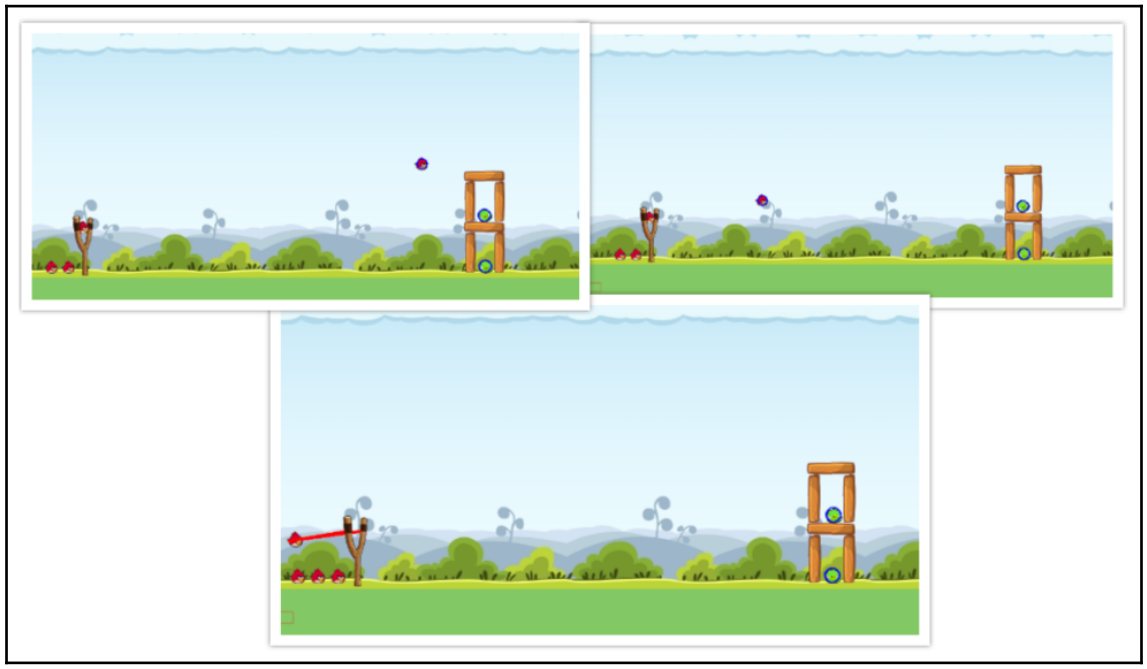
In the previously defined time interval (`dt` or time step interval), observe that we have moved the simulation of space forward 50 times with a `dt` of 2 units. If you increase the value of `dt` from 2 to 4 or more, the simulation will be slower. According to pymunk's official documentation: *Performing more steps by using smaller* `dt` *creates a stable simulation*. Here, the value 50 represents the steps defined and a `dt` of 2 creates a movement of a total of 100 units forward into the space. The forward simulation in the space represents the speed at which the Angry Bird is projected toward the wooden structure.

Now, using this time interval, let's add these steps to the simulation:

```
#time_step_change = 1.0/50.0/2.
for x in range(2):
    space_obj.step(time_step_change) # This causes two updates for frame

# Blitting second part of the sling
rect_for_sling = Pygame.Rect(0, 0, 60, 200)
screen.blit(sling_image, (120, 420), rect_for_sling)

Pygame.display.flip() #updating the game objects
clock.tick(50)
```

The `step` method, which is called using the `space` object, will update the space for the given time step interval (`dt` or time step interval). Refer to `http://www.pymunk.org/en/latest/_modules/pymunk/space.html` to find out more about the `step` method.

Finally, let's run our game. Click on the **Run** tab and then click on the `main.py` file. The following is the result of running the Angry Birds game:



Finally, our game is complete. You can test the different physical attributes we defined for the game entities by changing their values and observing their results. If I were you, I would probably change the step size value of `dt` and check how it affects the simulation of objects. Obviously, changing the value of `dt` from lower to higher would make the speed of objects slower after `sling_action` is triggered. For example, changing the value of step size (`dt = 4`), you would experience the Angry Bird going slower than before. This is due to an increase in the simulation forward movement by extra units.

Although our game is perfectly fine to play and test with, there are a few tweaks that can be implemented to make our game even more appealing. For example, we can add sound effects to the game and add more levels. We'll go over this in the next section.

Possible modifications

When testing our game, it might be the case that there is not much space for further modifications. However, I came up with an important one: adding `soundFx` to the game. To provide an active experience to the user while they're communicating with the virtual world, sound effects play an important role. With this consideration, Python's `Pygame` module provides an interface so that we can add a soundtrack to the game.

First of all, to add sound effects to the game, we need to load music into the game. Check out this book's resource folder on GitHub: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter15/res`. Then, check out the `sounds` folder, which will contain music files that can be added for the game project. I will use the `angry-birds.ogg` file (you can use any file you like—you can even download one off the internet).

The following code will load the music file into your Python project. Make sure the code is written inside the `main.py` file:

```
def load_music():
    """Function that will load the music"""
    song_name = '../res/sounds/angry-birds.ogg'
    Pygame.mixer.music.load(song_name)
    Pygame.mixer.music.play(-1)
```

In the preceding function definition, we started by defining the path for the music file and stored it as a string in the `song_name` variable. Now, to load the playback file, we can use the `mixer.music` class, which has a predefined `load()` method that will load the song into the Python project. To play the music that we have just loaded, we will call the `play()` method. The play method takes two arguments: loop and start. Both of these arguments are optional. The loop value will be `-1`, which means the loaded music must be played continuously. If you want to play music continuously, for example, six times, you can call the `play` method with a *loop = 5* argument on it. For example, `play(5)` will make the music play 6 times, continuously.

Now, let's call the aforementioned function within the same `main.py` file. You can call it as follows:

```
load_music()
```

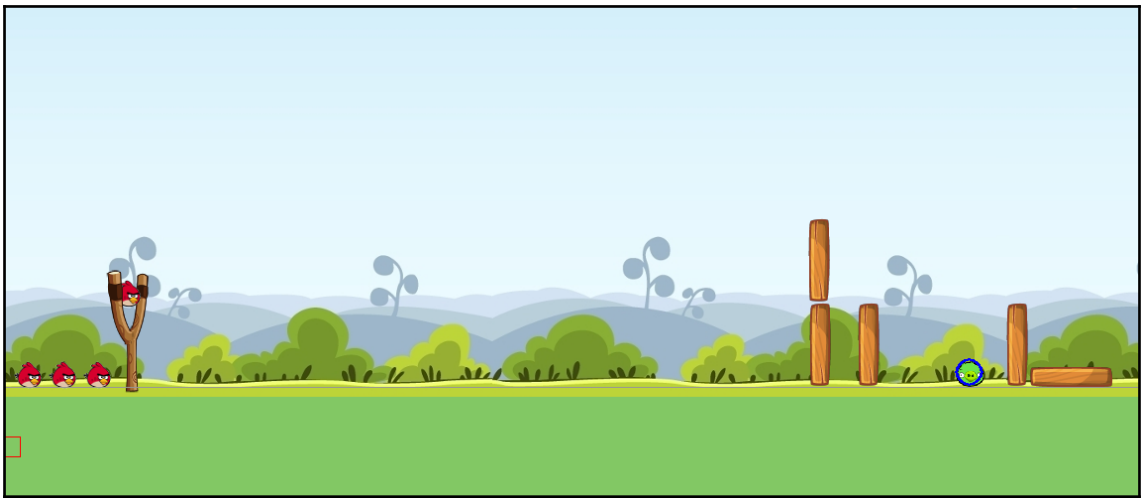
That's it if we want to load music into our Python game. Now, you can play your game and enjoy the soundtrack.

The next modification we can make is adding different levels. Go back to the Python project and open the `level.py` file. It will contain the `Level` class, along with a single function called `build_0`. You can add as many levels as you want. In this section, we will add another level for the game and call it `build_1`. The following function should be written inside the `Level` class of the `level.py` file:

```
def build_1(self):
    """Function that will render level 1"""
    obj_pig = RoundPig(1000, 100, self.space)
    self.pigs.append(obj_pig)
    pos = (900, 80)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (850, 80)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (850, 150)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (1050, 150)
    self.columns.append(Polygon(pos, 20, 85, self.space))
    pos = (1105, 210)
    self.beams.append(Polygon(pos, 85, 20, self.space))
    self.total_number_of_birds = 4 #reduce the number to
       make game more competitive
```

In the preceding code, we have defined a function that will create a wooden structure. Observe the code closely – we have created instances of the `Pig` and `Polygon` classes. The pig character was created at the position (1000, 10) in the space. Similarly, the three columns were created one after another and aligned vertically. The `pos` local variable denotes the position in the space where these game entities must be rendered. To create any random structure with the help of these game entities, you can test the different values for the `pos` variables. However, make sure that the position you defined is within the space and at the left-hand side corner of the space. For example, giving a position of (50 , 150) would render any game entities closer to the catapult and would not make the game competitive. Therefore, while building such structures, make sure that the entities are drawn far away from the catapult.

Now, when you run the program with the second level, you will see the following output:



You can add as many levels as you want. All you need is a bit of creativity for making game levels—forming beam and column structures that would be hard to break by the player. If you want to add further modifications, you could add a score to the game. You could assign some values to the game entities (pig, beam, and column) and whenever the bird collides with those game entities, you could add that value to the player's score. We implemented similar logic in `Chapter 12`, *Learning About Character Animation, Collision, and Movement*.

Finally, our game is playable, and you can test the sound effects and physical attributes of each game entity. You can test how the elasticity property has provided real-world simulation for the surfaces of the game. You can also test the simulation speed of the space. To learn more about the simulation step and step time interval, check out the online resources that are available at `http://www.pymunk.org/en/latest/_modules/pymunk/space.html`.

I had a lot of fun writing this chapter, along with building this game. I hope it was the same for you too. In the next chapter, we will learn about other important skills that every Python game developer must possess—adding an artificial character to the game. This character will play and compete with the human player in the same game. To be precise, we will be creating a human-like player in the game and adding intelligence to it, similar to what we humans have. The next chapter will be an interesting and edifying one. Let's get to it!

Summary

In this chapter, we explored how we can create Pythonic 2D physics simulation spaces by adding real-world physical attributes to game characters and the environment. We began by learning about the fundamentals of various `pymunk` modules, such as vec2d, sub-modules, different classes, and attributes that will build 2D rigid bodies. These bodies have the ability to simulate real-world object characteristics such as mass, inertia, motion, and elasticity. Using these characteristics, we were able to provide unique features to each of the game entities; that is, bird, pig, beam, and column.

The main aim of this chapter was to make you understand how to use the `pymunk` module effectively to create complex games such as Angry Birds. Games such as Angry Birds are considered intricate—not because it contains a variety of entities, but because they have to simulate real-world physical attributes. Since `pymunk` contains different classes to address such an environment, we used it to create a game environment, surfaces, and game entities such as Angry Birds, pigs, and polygons. In this chapter, you also learned how to handle collisions and movements between more than two game characters. So far, we've learned how to create a handler to address collisions between two game objects (between the snake and the boundary and between the flappy bird and the vertical pipes), but this chapter helped you understand how easily a collision handler can be created to address collisions between multiple game entities.

The next chapter will be a fun and challenging one. We will be learning about how to create **non-player characters** (**NPCs**)—an artificial player who is smart enough to compete with human players. We will create these NPCs by defining the moves and actions that human players would perform in the same instances. For example, when a human player sees a wall in front of them, they will make a move to omit a collision. A similar strategy will also be fed into the artificial player so that they can make smart moves and be able to compete with human players effectively.

16
Learning Game AI - Building a Bot to Play

–A game developer aims to create a game that is challenging and fun. Despite many attempts, many programmers have failed to do this. The main reason for the failure of games, is that human players love to be challenged by an artificial player in the gameplay. The result of the creation of such artificial players is generally referred to as a **non-player character** (**NPC**), or an artificial player. While the creation of such a player is fun (only for the programmer), it doesn't add any value to the game until and unless we inject some intelligence into that artificial player. The process of creating such NPCs and making them interact with human players with some degree of awareness and intelligence (closely comparable to human intelligence) is called **artificial intelligence** (**AI**).

In this chapter, we will create an *intelligent system*, which will be able to compete with a human player. The system will be smart enough to make moves similar to the moves of the human player. The system will be able to check collisions on its own, check the different possible moves, and make the one that is the most beneficial. Which move is beneficial will be highly dependent upon the target. The target of the artificial player will be defined explicitly by the programmer, and will be based on that target—a computer player will be able to make a smart move. For example, in the Snake AI game, the target of the computer player is to make a move that will lead them closer to the snake food, and in **first-person shooter** (**FPS**) games, the target of an artificial player is to approach the human player and to start to fire at the human player.

By the end of this chapter, you will have learned how to create an artificial system by defining machine states—ways to define what an artificial player will do in any instance. Similarly, we will take the example of Snake AI in order to illustrate how intelligence can be added to a computer player. We will create different entities for the game characters: the player, the computer, and the frog (snake food), and explore the power of object-oriented and modular programming. In this chapter, you will mostly find stuff that we have already covered and learn how to use it efficiently in order to make productive games.

We will cover the following topics in this chapter:

- Understanding AI
- Starting Snake AI
- Adding a computer player
- Adding intelligence to a computer player
- Building the game and frog entities
- Building the surface renderer and handler
- Possible modifications

Technical requirements

The following list of requirements must be acquired in order to work through this chapter effectively:

- The Pygame editor (IDLE)—version 3.5+ is recommended
- The PyCharm IDE (refer to `Chapter 1`, *Getting to Know Python – Setting Up Python and the Editor*, for the installation procedure)
- Assets (snake and frog `.png` files)—available at the GitHub link: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter16`

Check out the following video to see the code in action:

`http://bit.ly/2n79HSP`

Understanding AI

With the advent of numerous algorithms and models, today's game developers make use of them in order to create artificial characters, and then make them compete with human players. Playing a game passively, and competing with oneself, is not fun in real-world games anymore, thus, programmers intentionally set several difficulties and states, so that games are more challenging and fun. Among the several methods that programmers use, one of the best, and most popular, is making a computer compete with human beings. Sounds fun and complicated? The question that begs is how is it possible to create such algorithms, which will be able to compete with intelligent human beings. The answer is simple. We, as programmers, will define several smart moves, which will allow the computer to perform in a similar way to how we humans would respond to such situations.

While playing games, human beings are smart enough to protect their game characters from obstacles and defeat. Thus, in this chapter, our main aim is to provide such skills for NPCs. We will use the previously made Snake game (`Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*), refine it a little bit, and add a computer player to it, which will have some degree of awareness about where the food (things that the snake eats) is, and where obstacles are. Speaking literally, we are going to define different moves for our computer character, so that it will have a life of its own.

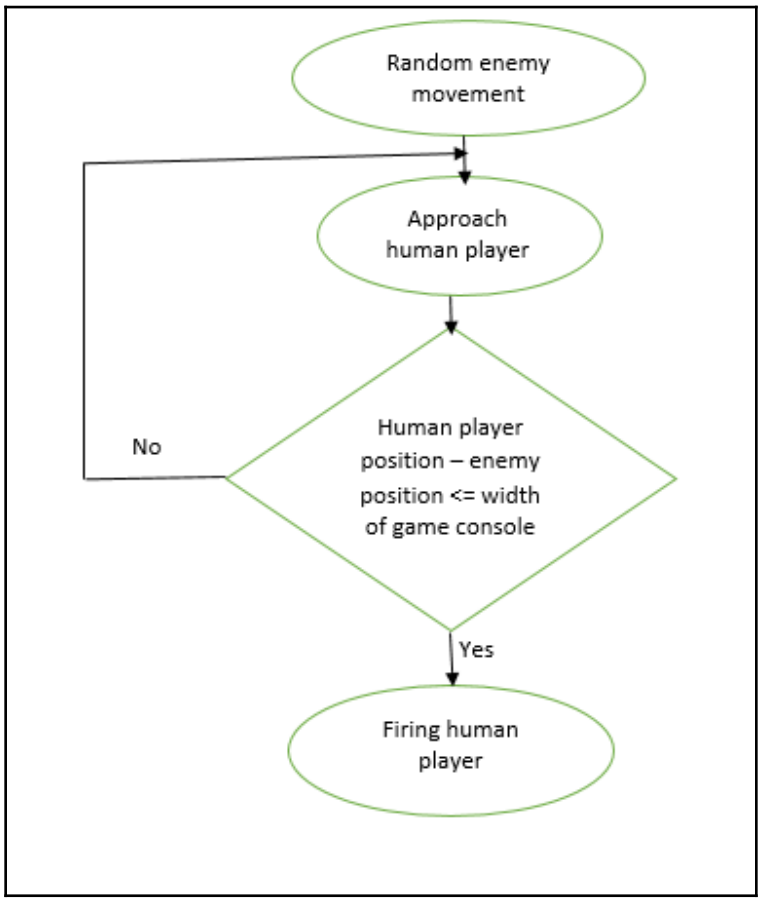
First of all, recall `Chapter 4`, *Data Structures and Functions*. In that chapter, we created a simple tic-tac-toe game, and embedded a simple *intelligent* algorithm in it. In that tic-tac-toe game, we were able to make a human player compete with the computer. We started by defining the models, handled the user events, and then finally added different moves in order for the computer to play on its own. We also tested the game, and the computer was able to beat the player in some instances. Thus, the basic concepts of AI were already learned by us back in `Chapter 4`, *Data Structures and Functions*. Nevertheless, in this chapter, we are going to dig deeper down into the world of AI, and uncover other cool things about *intelligent algorithms*, which can be added into our previously made Snake game.

To know how an AI algorithm works, we have to have a fair amount of knowledge of state machine charts. A state machine chart (originating generally from the *theory of computation*) defines what an NPC must do in different instances. We will learn about state machine charts, or animation charts, in the next topic.

Implementing states

The number of states are different for each game, and is highly dependent upon the complexity of the game. For instance, in a game such as an FPS, the NPCs, or enemies, must have different states: seeking the human player randomly, spawning a number of enemies randomly within player positions, shooting the human player, and many more. The relation between each of these states is defined by the state machine diagram. This diagram (not necessarily pictorial) represents the change from one state to another. For example, at what point should the enemy fire at the human player? At what distance should a random number of enemies be spawned?

The following diagram represents different states, and when such states must be changed from one to another:



While observing the preceding diagram, you might not find it foreign. We have done similar things before, in the case of adding an intelligent computer player to the tic-tac-toe game. In the figure, we start with random enemy movement, since we don't want each and every enemy to be rendered in the same place. Similarly, after enemies are rendered, they are allowed to approach the human player. There is no restriction on the movement of enemies. Thus, a simple conditional check between the position of the enemy and the human player can be implemented, in order to perform vectored movements (`Chapter 10`, *Upgrading the Snake Game with Turtle*) for the enemies. Similarly, after each position change, the position of the enemies is checked against the human player's position, and if they are near to each other, then the enemies can begin to fire toward the human player.

Between every state, there is a check on the steps, which make sure that the computer player is intelligent enough to compete with the human player. We can observe the following pseudo code, which represents the preceding machine states:

```
#pseudocode for random movement
state.player_movement():
    if state.hits_boundary:
        state.change_movement()
```

In the preceding pseudo code, each state defines the code, which must be executed in order to perform check operations such as `player_movement`, `hits_boundary`, and `change_movements`. Furthermore, in the case of approaching the human player, the pseudo code looks something like the following:

```
#pseudocode for check if human player and computer are near
if state.player == "explore":
    if human(x, y) == computer(x, y):
        state.fire_player()
    else:
        state.player_movement()
```

The preceding pseudo code is not the actual code, but it provides us with a blueprint about what we can expect AI to do for us. In the next topic, we will see how we can use our knowledge of pseudo code, and state machines, to create different entities for implementing AI in our snake game.

Starting snake AI

As discussed in the case of FPS, similar machine states can be used in the case of snake AI. The two important states that need to be considered for our computer player in the Snake AI game are as follows:

- What movements are valid for the computer player?
- What are the crucial stages at which changes from one state to another occur?

With regard to the preceding points, the first one indicates that whenever the computer player approaches the boundary line or wall, the movements of the computer player must be changed (making sure it remains within the boundary line), so that the computer player can compete with the human player. Secondly, we have to define a target for the computer snake player. In the case of FPS, as stated before, the main target for the computer enemy, is to find a human player and perform a *shooting* operation, but, in snake AI, the computer player has to approach the food in the game. The real competition in snake AI, between, the human and the computer player, is who can eat the food faster.

Now that we are aware of the actions that must be defined for the NPC (computer player), we can define the entities for the game. Similar to what we did in `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, our Snake AI has three major entities, and they are listed as follows:

- **Class** `Player`: It represents the human player, and all actions are related to the human—event handling, rendering, and movements.
- **Class** `Computer`: It represents the computer player (a form of AI). It performs actions such as updating the position and updating the target.
- **Class** `Frog`: It represents the food for the game. The aim of the competition between the human and the computer is approach to the frog as fast as possible.

Besides these three major game entities, there are two remaining game entities that define the peripheral tasks, and they are as follows:

- **Class** `Collision`: It represents the class that will have a method in order to check whether any entity (the player or the computer) has collided with the boundary, or not.
- **Class** `App`: It represents the class that will render the display screen and check whether any entity has eaten the frog or not.

Now, with the help of these entity blueprints, we can start to code. We will start by adding a `Player` class, along with the method that can render the player and handle its movement. Open your PyCharm editor, create a new project folder with a new Python file in it, and add the following code to it:

```
from pygame.locals import *
from random import randint
import pygame
import time
from operator import *
```

In the preceding code, every module will be familiar to you, except `operator`. When writing programs (especially when checking for collisions between the game entity and the boundary wall), it is extremely helpful to use mathematical functions in order to perform operations, rather than using mathematical operators directly. For instance, if you want to check `if value >= 2`, we can easily do the same operations by using the functions that are defined inside the `operator` module. In this case, we can call the `ge` method, which represents *greater than equal to*: `if ge(value, 2)`. Similar to the `ge` method, we can call different methods such as these:

- `gt(a, b)`: to check whether a > b—returns `True` if a > b; otherwise, `False`
- `lt(a, b)`: to check whether a < b—returns `True` if a < b; otherwise, `False`
- `le(a, b)`: to check whether a <= b—returns `True` if a <= b; otherwise, `False`
- `eq(a, b)`: to check whether a == b—returns `True` if a == b; otherwise, `False`

Now that you have imported the necessary modules, let's get to the fun stuff, by creating the `Player` class:

```
class Player:
    x = [0] #x-position
    y = [0] #y-position
    size = 44 #step size must be same for Player, Computer, Food
    direction = 0 #to track which direction snake is moving
    length = 3 #initial length of snake

    MaxMoveAllow = 2
    updateMove = 0

    def __init__(self, length):
        self.length = length
        for i in range(0, 1800):
            self.x.append(-100)
            self.y.append(-100)

        # at first rendering no collision
        self.x[0] = 1 * 44
        self.x[0] = 2 * 44
```

In the preceding code, we started defining class attributes: (`x`, `y`) represents the initial snake position, `size` represents the step size of the snake block, `direction` (value ranges from 0 to 4) represents the current direction in which the snake is moving, and `length` is the original length of the snake. The value of the attribute named `direction` will range from 0 to 3, where 0 represents that the snake is moving *right*, 1 represents that the snake is moving *left*, and similarly, 2 and 3 are for the *up* and *down* directions, respectively.

The next two class attributes are `MaxMoveAllow` and `update`. These two attributes will be used in the function named `updateMove` (shown in the following code), and they make sure that the player is not allowed to make a movement of the snake more than twice. It may well be case that the player might enter more than two arrow keys at once, but if all the effects or arrow keys are reflected at once, the snake will move incongruously. To omit this, we have defined the `maxMoveAllowed` variable, in order to ensure that, at most, two arrow key presses are handled at once.

Similarly, we have defined the constructor inside the class, which performs the initialization of the class attributes. It is not limited to that—after rendering the snake player in a random position (done by the `for` loop), we have written a statement that ensures that there are no collisions at the beginning of the game (the highlighted part). The code implies that the position between each block of the snake and the other blocks must be three units apart. If you change the value of `self.x[0] = 2*44 to self.x[0] = 1 *44`, then a collision will happen between the snake head and its. Thus, to ensure that there is no collision at the beginning (before the players start to play), we have to provide a specific positional gap between the blocks.

Now, let's use the `MaxMoveAllow` and `updateMove` attributes to create the `update` function:

```
def update(self):

    self.updateMove = self.updateMove + 1
    if gt(self.updateMove, self.MaxAllowedMove):

        # update previous to new position
        for i in range(self.length - 1, 0, -1):
            self.x[i] = self.x[i - 1]
            self.y[i] = self.y[i - 1]

        # updating the position of snake by size of block (44)
        if self.direction == 0:
            self.x[0] = self.x[0] + self.size
        if self.direction == 1:
            self.x[0] = self.x[0] - self.size
        if self.direction == 2:
            self.y[0] = self.y[0] - self.size
        if self.direction == 3:
            self.y[0] = self.y[0] + self.size

        self.updateMove = 0
```

The preceding code will not be foreign to you. You have seen such logic many times before (in `Chapter 6`, *Object-Oriented Programming*, and in `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, while handling the snake position). To recapitulate, the preceding line of code changes the current position of the human player to a new one, based on which arrow key is pressed. You can see in the code that we have not handled any arrow keys (we will do this in the `App` class afterward), but we have created an attribute named `direction`, which can track which key has been pressed. If `direction` is equal to `0`, it means that the right arrow key has been pressed, thus, we increase the *x*-position with the block size.

Similarly, if `direction` is `1`, we change the *x* positional value, by decrementing it with a block size of `44`, which means that the snake will move toward the negative *x*-axis. (This information is not new; a detailed discussion can be found in `Chapter 9`, *Data Model Implementation*.)

Now, in order to make sure that each `direction` attribute is associated with a value ranging from 0 to 3; we will create functions for each of them, as follows:

```
def moveRight(self):
    self.direction = 0

def moveLeft(self):
    self.direction = 1

def moveUp(self):
    self.direction = 2

def moveDown(self):
    self.direction = 3

def draw(self, surface, image):
 for item in range(0, self.length):
 surface.blit(image, (self.x[item], self.y[item]))
```

Observing the preceding code, you might have noticed the importance of the `direction` attribute. Each movement has an associated value that can be used when handling user events with the `pygame` module (we will discuss this later in the chapter). But, for now, just have a look at the `draw` function, which takes the arguments of `surface` and `image` of the snake (human player), and blits them accordingly. You might have a question such as: instead of using the `direction` attribute to handle user events, why don't we use a traditional approach (which we have been doing since `Chapter 8`, *Turtle Class – Drawing on the Screen*)? The question is valid, and obviously you can do it in that way, too, but there are major drawbacks to implementing such code in the case of Snake AI. Since Snake AI has two main players or game entities (the human being and the computer), each of them must have movements that are independent of one another. Thus, using traditional approaches for handling events differently for each entity would be both tedious and lengthy. A better option would be to use one attribute to track which key has been pressed, and handle it uniquely for each player, which we are going to do, in this case, using the `direction` attribute.

Now that we are done with the main human player, we will reach out to the computer player. We will start writing code for the `Computers` class, which will handle the moves that the computer makes, in the next topic.

Adding a computer player

Finally, we are in the main part of our chapter—the meaty part—it is easier to add the computer snake character into the game. As with the appearance, the movement handling technique of the computer must resemble the human player. We can reuse the code that was written inside the `Player` class. The only the thing that must differ from the `Player` class is the *target*. In the case of the human player, the target is not defined, since the target of movement is implemented by the player's mind. For example, the human player can play the game effectively by controlling its snake movement in the direction of the snake food. If the snake food is on the left, then there is no way that the human player will press the right arrow key and move the snake in the opposite direction. But, the computer is not smart enough to think of the best way to win the game on its own. Thus, we have to explicitly specify the target for the computer player. This technique of specifying the target for an individual player/system will result in an intelligent system, and its application ranges widely—from games to robotics.

For now, let's replicate the code that was written inside the `Player` class and add it to the new class, which is named `Computer`. The following code represents the creation of the `Computer` class, along with its constructor:

```
class Computer:
    x = [0]
    y = [0]
    size = 44 #size of each block of snake
    direction = 0
    length = 3

    MaxAllowedMove = 2
    updateMove = 0

    def __init__(self, length):
        self.length = length
        for item in range(0, 1800):
            self.x.append(-100)
            self.y.append(-100)

        # making sure no collision with player
        self.x[0] = 1 * 44
        self.y[0] = 4 * 44
```

Similar to the `Player` class, it has four attributes, with `direction` specified with an initial value of `0`, which means that before the computer actually starts to play, the snake will be automatically moving in the right (positive *x*-axis) direction. Furthermore, everything that is initialized within the constructor is similar to the `Player` class, except the highlighted part of the code. The last line of the code has `y[0]`, which started from `4 *44`. Recalling the same part of code in the case of the human player, it was `2*44`, which represents the column position. Writing this code, we are implying that there must not be a collision between the human player snake and the computer player snake at the beginning of the game. But, the value of `x[0]` is the same, because we want each of the snakes to start within the same row, but not in the same column. By doing this, we omit their collision, and each player's snake will be rendered properly.

Similarly, we have to add the `update` method, which will reflect the changes in the *x*, *y* position of the computer snake, based on the `direction` attribute. Th following code represents the `update` method, which will make sure that the snake computer is limited to using a combination of only two arrow key movements at one time:

```
def update(self):

    self.updateMove = self.updateMove + 1
    if gt(self.updateMove, self.MaxAllowedMove):

        # Previous position changes one by one
        for i in range(self.length - 1, 0, -1):
            self.x[i] = self.x[i - 1]
            self.y[i] = self.y[i - 1]

        # head position change
        if self.direction == 0:
            self.x[0] = self.x[0] + self.size
        if self.direction == 1:
            self.x[0] = self.x[0] - self.size
        if self.direction == 2:
            self.y[0] = self.y[0] - self.size
        if self.direction == 3:
            self.y[0] = self.y[0] + self.size

        self.updateMove = 0
```

The preceding code is similar to the `Player` class, so I won't bother explaining it. You can refer to the `update` function of the `Player` class to see how this method works. Similar to the `Player` class, we have to add four methods that will change value of the `direction` variable accordingly:

```
def moveRight(self):
    self.direction = 0

def moveLeft(self):
    self.direction = 1

def moveUp(self):
    self.direction = 2

def moveDown(self):
    self.direction = 3
```

The code that is written will be able to update the *direction* of the computer player, but it is not enough to make a smart move. Let's say, if the snake food is on the right-hand side, the code that has been written up till now won't be able to track the position of the food, and thus, the computer snake might go to the opposite place. Thus, we have to explicitly specify that the computer player will move in such a direction, which is close to the position of the snake food. We will cover this in the next topic.

Adding intelligence to a computer player

Up till now, two game entities have been defined, and both of them handle the players' movements. Unlike the `Player` class, another game entity (the computer player) is not going to decide its next move on its own. Thus, we have to explicitly enjoin the computer player to make a move that would take the snake closer to the snake food. By doing this, there will be immense competition between the computer player and the human player. This looks quite complex to implement; however, the idea still remains the same, as discussed earlier, along with the machine state diagram.

Going through the machine state diagram, the AI player must accommodate two things:

- Check the position of the snake food, and make a move in order to get closer to it.
- Check the current position of the snake, and make sure it that doesn't hit the boundary wall.

The first step will be implemented as follows:

```
def target(self, food_x, food_y):
    if gt(self.x[0] ,  food_x):
        self.moveLeft()

    if lt(self.x[0] , food_x):
        self.moveRight()

    if self.x[0] == food_x:
        if lt(self.y[0] , food_y):
            self.moveDown()

        if gt(self.y[0] , food_y):
            self.moveUp()

def draw(self, surface, image):
     for item in range(0, self.length):
         surface.blit(image, (self.x[item], self.y[item]))
```

In the preceding line of code, we called different previously made methods, such as `moveLeft()`, `moveRight()`, and so on. These methods will cause the snake to move as indicated by the `direction` attribute value. The `target()` method takes two arguments: `food_x` and `food_y`, which compositely refer to the position of the snake food. The operators, `gt` and `lt`, are used to perform comparison operations with the snake *x*-head and *y*-head positions. For instance, if the snake food is on the negative *x*-axis, then a comparison is made between the *x*-position of the snake and the *x*-position of the food (`gt(self.x[0], food_x)`). It is obvious that `food_x` is on the negative *x*-axis, which means that the snake *x*-position is greater, thus, `moveLeft()` is called. As the signature of the method suggests, we are going to make a turn, and move the computer player snake toward the negative *x*-axis. Similar comparisons are done for each (*x*, *y*) position of the food, and each time a different method is called, so that we can lead the computer player toward the snake food.

Now that we have added the simple computer player, which is able to pass through multiple obstacles, let's add the `Frog` and `Collision` classes in the next topic. The `Frog` class is responsible for rendering the frog (the snake food) on the screen at random positions, and `Collision` will check whether there is a collision between the snakes, and/or between a snake and the boundary wall.

Building the game and frog entities

As previously mentioned, we are going to add two more classes into our code in this topic. Each of these classes serve different purposes in our Snake AI. The `Game` entity will check whether there is any sort of collision, by checking the argument that is passed to their member methods. In the case of the `Game` entity, we will define a simple, yet powerful method, named `checkCollision()`, which will return a Boolean of either `True` or `False`, based on the collision.

The following code represents the `Game` class and its member method:

```
class Game:
    def checkCollision(self, x1, y1, x2, y2, blockSize):
        if ge(x1 , x2) and le(x1 , x2 + blockSize):
            if ge(y1 , y2) and le(y1, y2 + blockSize):
                return True
        return False
```

The call to the `checkCollision()` method will be done inside the main class (which will be defined in a moment). But, the important thing that you will notice is that the argument that is passed (the *x* and *y* values), will be the current position of the snake, from which this method will be called. Let's say you make an instance of the `Game` class, and pass the (`x1`, `y1`, `x2`, and `y2`) positional values of the human player. In doing so, you are calling the `checkCollision` method for the human players. The conditional statements check whether the positional value of a snake is the same as the boundary wall, or not. If yes, it will return `True`; otherwise, it will return `False`.

The next important game entity is `Frog`. This class renders the image of `Frog` in a random position, after each time it gets eaten by any player (the human or the computer). The following code represents the declaration of the `Frog` class:

```
class Frog:
    x = 0
    y = 0
    size = 44

    def __init__(self, x, y):
        self.x = x * self.size
        self.y = y * self.size

    def draw(self, surface, image):
        surface.blit(image, (self.x, self.y))
```

In the preceding code, we have defined the *x*-position, the *y*-position, and the `draw` method in order to render the frog image. The call to this method will be made by creating `Frog` from the main class.

In the next topic, we will wrap up our program by creating and implementing one last entity: the main `App` entity. This will be the central director of our game.

Building the surface renderer and handler

To begin, let's recap what we have done so far. We started to write the code by defining two major game entities: `Player` and `Computer`. Both of these entities were quite similar in terms of actions and rendering methods, except an extra `target()` method was introduced within the `Computer` class, in order to make sure that the computer player is smart enough to compete with the human player. Similarly, we declared two more entities: `Game` and `Frog`. These two classes provide the backend facility for the Snake AI, such as adding collision logic, and checking the position for the snake food to be rendered in. We have created multiple methods within these different entities, but we have never made instances/objects out of them. Such instances can be created from the main single class, which we are going to implement now. I am going to call this class the `App` class.

Look at the following snippet in order to write the code for the `App` class:

```
class App:
    Width = 800 #window dimension
    Height = 600
    player = 0 #to track either human or computer
    Frog = 0 #food

    def __init__(self):
        self._running = True
        self.surface = None
        self._image_surf = None
        self._Frog_surf = None
        self.game = Game()
        self.player = Player(5) #instance of Player with length 5 (5
        blocks)
        self.Frog = Frog(8, 5) #instance of Frog with x and y position
        self.computer = Computer(5) #instance of Computer player with
        length 5
```

The preceding code defines some attributes, such as `Height` and `Width`, for the games console. Similarly, it has a constructor, which initializes the different class attributes, along with creating the `Player`, `Frog`, and `Computer` instances.

Next up, is to load the image from the computer and add it to the Python project (refer to `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, to learn more about the `load` method). The assets of the game, such as the snake body and food, are available at this GitHub link: `https://github.com/PacktPublishing/Learning-Python-by-building-games/tree/master/Chapter16`. But, you can also create your own, and experiment with it. I have taught you how to create a transparent sprite using GIMP and a simple paint application before, in `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*. Try to recap those concepts, and it try on your own. For now, I am going to load two images into the Python project.

It is better to use a .png file for sprites, and don't create a filename with a numeric value in it. For example, a filename for the snake body that is named `snake12.png` is not valid. The filename should be given without numeric values. Similarly, make sure that you add those `.png` files within the Python project folder. Revisit `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, to check how an image is loaded with PyCharm into the Python project.

The following code will load two image files into the Python project:

```
def loader(self):
 pygame.init()
 self.surface = pygame.display.set_mode((self.Width, self.Height),
 pygame.HWSURFACE)

 self._running = True
 self._image_surf = pygame.image.load("snake.png").convert()
 self._Frog_surf = pygame.image.load("frog-main.png").convert()
```

In the preceding line of code, we created a `surface` object using the `pygame.display` module. Then, we loaded two images—`snake.png` and `frog-main.png`—into the Python project. The `convert()` method will change the pixel formatting of the rendered object, so that it works perfectly on any surface.

Similarly, if a game has events, and it interacts with the user, then the `on_event` method must be implemented:

```
def on_event(self, event):
    if event.type == QUIT:
        self._running = False
def on_cleanup(self):
    pygame.quit()
```

Finally, let's define the `main` function:

```
def main(self):
    self.computer.target(self.Frog.x, self.Frog.y)
    self.player.update()
    self.computer.update()
```

In the preceding function, we called the `target` method to make sure that the computer player is able to use the capabilities that have been defined inside it. As discussed before, the `target()` method takes the *x*, *y* coordinates of the food and the computer makes a decision to move closer to the food. Similarly, the `update` method of both the `Player` and `Computer` classes is called.

Now let's define the `renderer()` method. This method will draw the snakes and the food onto the game surface. This is done using the `pygame` and `draw` modules:

```
def renderer(self):
    self.surface.fill((0, 0, 0))
    self.player.draw(self.surface, self._image_surf)
    self.Frog.draw(self.surface, self._Frog_surf)
    self.computer.draw(self.surface, self._image_surf)
    pygame.display.flip()
```

If you feel that you do not understand the workings of the `renderer()` method, go to `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*. In summary, this method will draw different objects (`image_surf` and `Frog_surf`) onto the game screen.

Finally, let's create a `handler` method. This method will handle the user events. Different methods, such as `moveUp()`, `moveDown()`, `moveLeft()`, and `moveRight()` will be called, based upon the arrow keys that are pressed by the user. These four methods are created within both the `Player` and `Computer` entities. The following code defines the `handler` method:

```
def handler(self):
    if self.loader() == False:
        self._running = False

    while (self._running):
        keys = pygame.key.get_pressed()

        if (keys[K_RIGHT]):
            self.player.moveRight()

        if (keys[K_LEFT]):
            self.player.moveLeft()
```

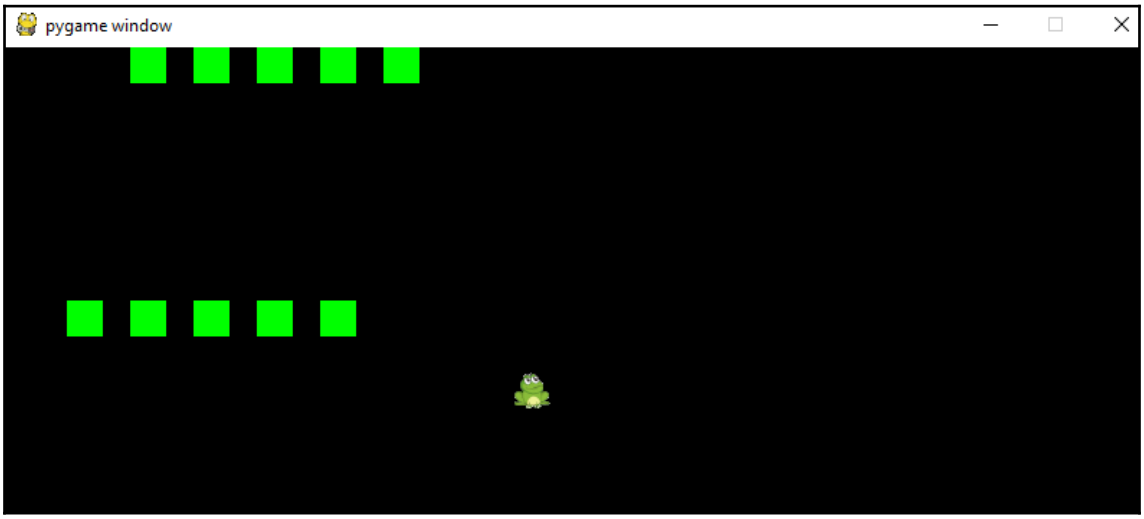
```
            if (keys[K_UP]):
                self.player.moveUp()

            if (keys[K_DOWN]):
                self.player.moveDown()

            self.main()
            self.renderer()

            time.sleep(50.0 / 1000.0);
```

The preceding `handler` method has been created so many times before (we saw both advanced and easy methods), and this one is the easiest one. We used the `pygame` module to listen to incoming key events and handled them accordingly, by calling different methods. For example, when the user pressed the down arrow key, the `moveDown()` method was called. The last `sleep` method will embed the timer, so that there is a difference between two successive key events.

Finally, let's call this `handler` method:

```
if __name__ == "__main__":
    main = App()
    main.handler()
```

Let's run our game and observe the output:



As expected, there are several things that must be added to this game, including: what happens when the human player and the computer player eat the food, and what happens when the snake collides with itself? If you have followed the book throughout, properly, this should be a piece of cake for you. We have added this same logic multiple times (in `Chapter 7`, *List Comprehension and Properties*; `Chapter 10`, *Upgrading the Snake Game with Turtle*; and `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*). But apart from that logic, focus on the two alike snakes: one must be moving with human player actions, and the other independently. The computer snake was aware of the collision with the boundary wall and the position of the food. As soon as you run your game, the computer player will react instantaneously, and will try to make a smart move, before the human does. This is the application of AI in the real-world gaming industry. Although you might think that the Snake AI example is simpler, in the real world, AI is also all about the machine acting independently, regardless of how complex the algorithm is.

But, there are several tweaks that must be made within the game, which will be covered in the next topic—*Possible modifications*.

Game testing and possible modifications

First of all, I suggest that you look back and observe the part where we defined the `Game` class. We defined the `checkCollision()` method inside it. This method can be used for multiple purposes: firstly, to check whether a player collides with the snake food; and secondly, to check whether a player collides with the boundary wall, or not. You must have a *gotcha* moment at this time. `Chapter 7`, *List Comprehension and Properties*, to `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*, was all about using this technique to implement the collision principle, which states that, *If the (x, y) position of food objects is same with the (x, y) coordinates of any player, there is said to be a collision.*

Let's add code that will check whether any player has collided with the food or not:

```
# Does human player snake eats Frog
for i in range(0, self.player.length):
    if self.game.checkCollision(self.Frog.x, self.Frog.y,
    self.player.x[i], self.player.y[i], 44):
        #after each player eats frog; next frog should be spawn in next
        position
        self.Frog.x = randint(2, 9) * 44
        self.Frog.y = randint(2, 9) * 44
        self.player.length = self.player.length + 1

# Does computer player eats Frog
for i in range(0, self.player.length):
```

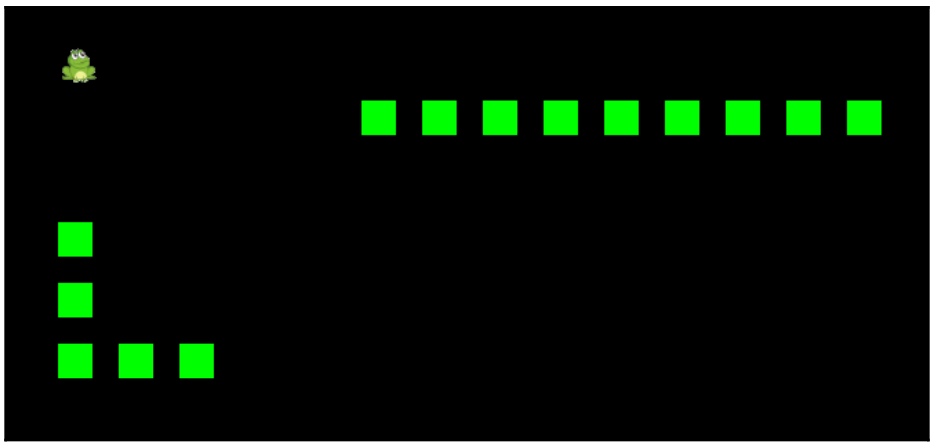
```
        if self.game.checkCollision(self.Frog.x, self.Frog.y,
            self.computer.x[i], self.computer.y[i], 44):
            self.Frog.x = randint(2, 9) * 44
            self.Frog.y = randint(2, 9) * 44
```

Similarly, let's use the same function to check whether the human player's snake has hit the boundary wall or not. You might think that you need to check this in the case of the computer player too, but that is useless, because the `target` method that was defined in the `Computer` class will not let this happen. In other words, the computer player will never hit the boundary wall, thus, checking whether a collision happened or not is useless. But, in the case of the human player, we will check it using the following code:

```
    # To check if the human player snake collides with its own body
    for i in range(2, self.player.length):
        if self.game.checkCollision(self.player.x[0], self.player.y[0],
       self.player.x[i], self.player.y[i], 40):
            print("You lose!")
            exit(0)

    pass
```

We will end this topic right here, but you can make this game even more appealing by adding a *game over* screen, which we learned how to create using `pygame` back in `Chapter 11`, *Outdo Turtle – Snake Game UI with Pygame*. Instead of the last `pass` statement, you can create a surface and render a font with a label in it, in order to create such a game over screen.

But, before wrapping up this chapter, let's look at the final output of our game:



Another thing that you might notice in the game is that the computer player's snake *length* is constant, even if it eats the food. I did this intentionally, so that my game screen wouldn't be polluted too much. But, if you want to increase the computer player's snake length (every time the snake eats the food), you can add a statement after the computer player snake eats the frog:

```
self.computer.length = self.computer.length + 1
```

Finally, we have come to the end of this chapter. We have learned different things, as well as revising old ones. The concepts that are associated with AI are vast; we have just attempted to touch the surface. You can find other implications of AI in the game using Python by heading to this URL: `https://www.pygame.org/tags/ai`.

Summary

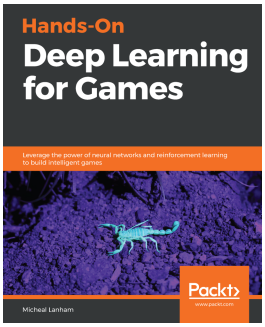
In this chapter, we explored the basic way of implementing AI in our game. Nonetheless, the workings of AI depend heavily on rewarding the intelligent system for its each and every move. We used a machine state diagram to define the possible states for our computer player, and used it to perform different actions for each entity. We employed different programming paradigms in this single chapter; in fact, it was a recap of everything that we have learned so far, in addition to employing smart algorithms for NPCs.

For each defined entity, we made a class, and employed an object-oriented paradigm such as the encapsulation and model, based on properties and methods. Furthermore, we defined different classes such as `Frog` and `Game` in order to implement the logic for collisions. The reason for making separate classes for implementing single logic is because these methods should be called by each game entity (`Player` and `Computer`) independently. You could infer it as multi-inheritance. The main aim of this book was to make the reader understand how a gaming bot can be created with Python. Furthermore, to some extent, the aim was to revise all the programming paradigms that we have learned throughout the book, in a single chapter.

As the old adage says: *Known is a drop. Unknown is an Ocean*. I hope you are still yearning to learn more about Python. I suggest you brush up on your basic programming skills and experiment more often, which will surely lead you to your dream job of becoming a game developer. The gaming industry is huge, and having knowledge of Python will make a difference. Python is a beautiful language, thus, you will be incentivized to learn it more deeply, and this book will be the first of many steps that you will take in order to become an expert in Python.

Other Books You May Enjoy

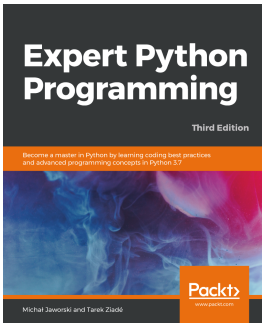
If you enjoyed this book, you may be interested in these other books by Packt:



Hands-On Deep Learning for Games
Micheal Lanham

ISBN: 9781788994071

- Learn the foundations of neural networks and deep learning.
- Use advanced neural network architectures in applications to create music, textures, self driving cars and chatbots.
- Understand the basics of reinforcement and DRL and how to apply it to solve a variety of problems.
- Working with Unity ML-Agents toolkit and how to install, setup and run the kit.
- Understand core concepts of DRL and the differences between discrete and continuous action environments.
- Use several advanced forms of learning in various scenarios from developing agents to testing games.



Expert Python Programming - Third Edition
Tarek Ziadé, Michał Jaworski

ISBN: 9781789808896

- Explore modern ways of setting up repeatable and consistent development environments
- Package Python code effectively for community and production use
- Learn modern syntax elements of Python programming such as f-strings, enums, and lambda functions
- Demystify metaprogramming in Python with metaclasses
- Write concurrent code in Python
- Extend Python with code written in different languages
- Integrate Python with code written in different languages

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W