A STEP-BY-STEP GUIDE TO CODING IN PYTHON FAST

LEARN PYTHON BY CODING VIDEO GAMES

(BEGINNER)

PATRICK FELICIA

Learn Python by Coding Video Games (Beginner)

Patrick Felicia

Published by Patrick Felicia, 2022.

LEARN PYTHON BY CODING VIDEO GAMES (BEGINNER)

First edition. September 16, 2022.

ISBN: 979-8215851357

Written by Patrick Felicia.

TABLE OF CONTENTS

Learn Python

By Coding Video Games

(Beginner)

First Edition

A step-by-step guide to learning Python fast.

Patrick Felicia

Learn Python by Coding Video Games (Beginner)

First published: September 2022

Published by Patrick Felicia

Credits

Author: Patrick Felicia

About the Author

Patrick Felicia is a lecturer and researcher at Waterford Institute of Technology, where he teaches and supervises undergraduate and postgraduate students. He obtained his MSc in Multimedia Technology in 2003 and PhD in Computer Science in 2009 from University College Cork, Ireland. He has published several books and articles on the use of video games for educational purposes, including the Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches (published by IGI), and Digital Games in Schools: A Handbook for Teachers, published by European Schoolnet. Patrick is also the Editor-in-chief of the International Journal of Game-Based Learning (IJGBL), and the Conference Director of the Irish Symposium on Game-Based Learning, a popular conference on games and learning organized throughout Ireland.

Support and Resources for this Book

To complete the activities presented in this book you need to download the startup pack on the companion website; it consists of free resources that you will need to complete your projects. To download these resources, please use the following link:

- **http://learntocreategames.com/book_downloads/python_games/book**

This book is dedicated to Mathis

[ ]

Table of Contents

Preface

This book will show you how you can very quickly code in Python and create games.

Python is a powerful programming language used in a wide range of industries and that you can use even if you have (or you are teaching with) computers with very low technical specifications.

This book series entitled **Learn Python by Coding Video Games** allows you to play around with Python’s core features, and essentially those that will make it possible to create interesting 2D games rapidly. After reading this book series, you should find it easier to code in Python and to create simple yet entertaining video games.

This book series assumes no prior knowledge on the part of the reader, and it will get you started on Python so that you quickly master all the wonderful features that this programming language provides by going through an easy learning curve.

By completing each chapter, and by following step-by-step instructions, you will progressively improve your skills, become more proficient in Python, and create several games.

In addition to becoming proficient with Python, you will also create games that include many of the common techniques found in video games such as: level design, object creation, textures, collision detection, lights, weapon creation, character animations, particles, artificial intelligence, and menus.

You will learn how to create customized menus and simple user interfaces using both Python and Pygame and animate and give artificial intelligence to Non-Player Characters (NPCs) that will be able to follow the player character using pathfinding.

Finally, you will also get to export your game at the different stages of the books, so that you can share it with friends and obtain some feedback as well.

Content Covered by this Book

Chapter 1, Introduction to Programming in Python, provides an introduction to Python and to core principles that will help you to get started. It explains key programming concepts such as variables, variable types, or functions.

Chapter 2, Creating your First Script, helps you to code your first script. It explains common coding mistakes and errors in Python, and how to avoid them easily. It also goes through some common error messages for beginners and explains what they mean and how they can be avoided easily.

Chapter 3, Creating a Number Guessing Game, gets you to add interaction to your games and to create a simple number guessing game. You will learn to create a scoring system, to use loops and conditional statements, and to detect and process the keys pressed by the user.

Chapter 4, Creating a Word Guessing Game, gets you to create a word guessing game where a random word from a large file will need to be guessed by the player. You will learn to use loops and global variables, to read data from a text file, and to display messages in the Command Prompt.

Chapter 5, Using Graphics and the Pygame Library, introduces the use of graphics with Python and Pygame so that you can draw objects onscreen, and detect mouse interaction from the user.

Chapter 6, Creating a Coin Collection Game, explains how you can create an adventure game where the player has to collect coins in order to move on to the next levels. Along the way, you also learn how to detect collisions, to move an animated character on screen, to create a splash screen as well as a timer.

Chapter 7 provides answers to Frequently Asked Questions (FAQs) related to the topics covered in this book (e.g., scripting, audio, interaction, AI, or user interface). It also provides links to additional exclusive video tutorials that can help you with some of your questions.

Chapter 8 summarizes the topics covered in the book and provides you with more information on the next steps.

What you Need to Use this Book

To complete the project presented in this book, you only need to install Python and Pygame (this will be explained in the next sections).

In terms of computer skills, all knowledge introduced in this book will assume no prior programming experience from the reader. So for now, you only need to be able to perform common computer tasks such as downloading files, opening and saving files, be comfortable with dragging and dropping items, and typing.

Who this book is for

If you can answer **yes** to all these questions, then this book is for you:

1. Are you a total beginner in Python?
2. Would you like to become proficient in the core functionalities offered by Python and Pygame?
3. Would you like to teach students or help your child to understand how to create games, using programming?
4. Would you like to start creating entertaining 2D games?
5. Although you may have had some prior exposure to Python or Pygame, would you like to delve more into these topics and understand the core functionalities in more detail?

Who this book is not for

If you can answer yes to all these questions, then this book is **not** for you:

1. Can you already code with Python to implement simple behaviors such as score, collision detection, or to update the user interface?
2. Can you already easily code a 2D game with Pygame and Python?
3. Are you looking for a reference book on Python?
4. Are you an experienced (or at least advanced) Python user?

If you can answer yes to all four questions, you may instead look for the next books in the series. To see the content and topics covered by these books, you can check the official website (**www.learntocreategames.com/books**).

How you will learn from this book

Because all students learn differently and have different expectations of a course, this book is designed to ensure that all readers find a learning structure that suits them. Therefore, it includes the following:

- A list of the learning objectives at the start of each chapter so that readers have a snapshot of the skills that will be covered.
- Each section includes an overview of the activities covered.
- Many of the activities are step-by-step, and learners are also allowed to engage in deeper learning and problem-solving skills through the challenges offered at the end of each chapter.
- Each chapter ends up with a quiz and challenges through which you can put your skills (and knowledge acquired) into practice, and see how much you know. Challenges consist in coding, debugging, or creating new features based on the knowledge that you have acquired in the chapter.
- The book focuses on the core skills that you need. Some sections also go into more detail; however, once concepts have been explained, links are provided to additional resources, where necessary.
- The code is introduced progressively and is explained in detail.

Format of Each Chapter and Writing Conventions

Throughout this book, and to make reading and learning easier, text formatting and icons will be used to highlight parts of the information provided and to make it more readable.

The full solution for the project presented in this book is available for download on the official website (**http://learntocreategames.com/books**). So if you need to skip a section, you can do so; you can also download the solution for the previous chapter that you have skipped.

Special Notes

Each chapter includes resource sections so that you can further your understanding and mastery of Python; these include:

- A quiz for each chapter: these quizzes usually include 10 questions that test your knowledge of the topics covered throughout the chapter. The solutions are provided on the companion website.
- A checklist: it consists of between 5 and 10 key concepts and skills that you need to be comfortable with before progressing to the next chapter.
- Challenges: each chapter includes a challenge section where you are asked to combine your skills to solve a particular problem.

The author’s notes appear as described below:

The author’s suggestions appear in this box.

Code appears as described below:

```
score = 100
player_name = “Sam”
```

Checklists that include the important points covered in the chapter appear as described below:

- Item1 for the checklist
- Item2 for the checklist
- Item3 for the checklist

How Can You Learn Best from this Book?

- **Talk to your friends about what you are doing.**

We often think that we understand a topic until we have to explain it to friends and answer their questions. By explaining your different projects, what you just learned will become clearer to you.

- **Do the exercises.**

All chapters include exercises that will help you to learn by doing. In other words, by completing these exercises, you will be able to better understand the topic and gain practical skills (i.e., rather than just reading).

- **Don’t be afraid of making mistakes.**

I usually tell my students that making mistakes is part of the learning process; the more mistakes you make and the more opportunities you have for learning. At the start, you may find the errors disconcerting, or that Python does not work as expected until you understand what went wrong.

- **Compile and run your games early.**

It is always great to compile and run your first game.

- **Learn in chunks.**

It may be disconcerting to go through five or six chapters straight, as it may lower your motivation. Instead, give yourself enough time to learn, go at your own pace, and learn in small units (e.g., between 15 and 20 minutes per day). This will do at least two things for you: it will give your brain the time to “digest” the information that you have just learned, so that you can start fresh the following day. It will also make sure that you don’t “burn out” and that you keep your motivation levels high.

Feedback

While I have done everything possible to produce a book of high quality and value, I always appreciate feedback from readers so that the book can be improved accordingly. If you would like to give feedback, you can email me at **learntocreategames@gmail.com**.

Downloading the Solutions for the Book

You can download the solutions for this book after creating a free online account at **http://learntocreategames.com/books/**. Once you have registered, a link to the files will be sent to you automatically.

To download the solutions for this book (e.g., code) you need to download the startup pack on the companion website; it consists of free resources that you will need to complete your projects and code solutions. To download these resources, please open the following link:

- **http://learntocreategames.com/book_downloads/python_games/book**

Improving the Book

Although great care was taken in checking the content of this book, I am human, and some errors could remain in the book. As a result, it would be great if you could let me know of any issue or error you may have come across in this book, so that it can be solved, and the book updated accordingly. To report an error, you can email me (**learntocreategames@gmail.com**) with the following information:

- Name of the book.
- The page where the error was detected.
- Describe the error and also what you think the correction should be.

Once your email is received, the error will be checked, and, in the case of a valid error, it will be corrected, and the book page will be updated to reflect the changes accordingly.

Supporting the Author

A lot of work has gone into this book and it is the fruit of long hours of preparation, brainstorming, and finally writing. As a result, I would ask that you do not distribute any illegal copies of this book.

This means that if a friend wants a copy of this book, s/he will have to buy it through the official channels (i.e., through your favourite e-store, or the book's official website: **http://www.learntocreategames.com/books**).

If some of your friends are interested in the book, you can refer them to the book's official website (**http://www.learntocreategames.com/books**) where they can either buy the book, enter a monthly draw to be in for a chance of receiving a free copy of the book, or to be notified of future promotional offers.

[ ]

CHAPTER 1: INTRODUCTION TO PROGRAMMING IN PYTHON

In this section we will discover Python programming principles and concepts, so that you can start programming in the next chapter. If you have already coded using Python (or a similar language), you can skip this chapter.

After completing this chapter, you will be able to:

- Become familiar with and understand the concepts of variables, methods, and scope.
- Understand key best practices for coding, especially in Python.
- Understand how to use conditional statements and decision-making structures.
- Understand the concept of loops.
- Understand Object-Oriented Programming (OOP) concepts when coding in Python.

Introduction

When coding in Python, you are communicating with the system and asking it to perform actions. To communicate with the system, you are using a language or a set of words bound by a syntax that the computer and you know. This language consists of keywords, key phrases, and a syntax that ensures that your instructions are understood properly. In computer science, this sentence needs to be accurate, precise, unambiguous, and with correct syntax. In other words, it needs to be **exact**. The syntax is a set of rules that are followed when writing code in Python. In addition to its syntax, Python programming also uses classes; so your scripts will be saved as classes.

In the next section, we will learn how to use this syntax. If you have already coded in Python or other object-oriented programming languages, some of the information provided in the rest of this chapter may look familiar and this prior exposure to programming will definitely help you.

When scripting in Python, you will be using a combination of the following:

- Classes.
- Objects.
- Statements.
- Comments.
- Variables.
- Constants.
- Operators.
- Assignments.
- Data types.
- Functions.

- Methods.
- Decision-making structures.
- Loops.
- Inheritance (more advanced).
- Events.
- Comparisons.
- Type conversions.
- Reserved words.
- Messages to the Command Prompts.
- Declarations.
- Calls to methods.

The list may look a bit intimidating but, not to worry, we will explore these in the next sections, and you will get to know and use them smoothly using hands-on examples.

Statements

When you write a piece of Python code, you need to ask the system to execute your instructions (e.g., to print information onscreen) using statements. A statement is literally an order or something you ask the system to do. For example, in the next line of code, the statement will tell Python to print a message in the **Command Prompt**:

```
print (“Hello World”)
```

When writing statements, there are a few rules that you need to know:

- Order of statements: each statement is executed in the order it appears in the script. For example, in the next example, the code will print **hello**, then **world**; this is because the associated statements are in that sequence.

```
print (“hello”)
print (“world”)
```

- Use one statement per line, as much as possible. Otherwise, use a semi-colon to separate these statements.
- For example, the next line of code has incorrect syntax.

```
print(“hello”) print (“world”)
```

- Multiple spaces are ignored for statements; however, it is good practice to add spaces around the operators such as **+**, **-**, **/**, or **%** for clarity. For example, in the next example, we say that **a** is equal to **b**. There is a space both before and after the operator **=**.

```
a = b;
```

- Statements to be executed together (e.g., based on the same condition) can be grouped using what is usually referred to as **code blocks**. In Python code blocks are implemented using indentation. So, in other words, if you needed to group several statements, we would indent all of them at the same level, as follows:

```
if (x > 100):
print (“hello stranger!”)
print (“today, we will learn about scripting”)
```

In the previous statements, if x is greater than 100 then two **print** statements will be executed.

Note that we use a colon after the condition to start a block of instructions that will be executed if the condition is verified.

As we have seen earlier, a statement usually employs or starts with a keyword (i.e., a word that the computer understands). All these keywords have a specific purpose and the most common ones (at this stage) are used for:

- Printing a message in the **Console Prompt**: the keyword is **print**.
- Declaring a function: the keyword is **def**.

In Python the keywords **method** and **function** have a different meaning. While functions are a set of statements that accomplish a task when called, a method refers to a function declared within a class, as we will see later.

- Marking a block of instructions to be executed based on a condition: the keywords are **if**...**else**.
- Exiting a function: the keyword is **return**.

Note that in Python you can end your statements using a semi-colon; this is however optional in Python; this being said, it can be useful if you would like to write multiple statements on one line, each separated by a semicolon, as illustrated in the next code snippet:

```
print ("Hello"); print ("World")
```

Note that python code is usually saved in a file with a .py extension, before being compiled and interpretated; python code can also be used in the command line directly, as we will see later in this book.

Comments

In Python, you can use comments to explain the code and to make it more readable. This becomes important as the size of your code increases; and it is also important if you work in a team, so that the other team members can understand your code and make amendments in the right places, if and when it is needed.

When some code is commented it is not executed. For comments:

- A # is added at the start of a line or after a statement, so that this line (or part thereof) is commented, as illustrated in the next code snippet.
- You can also comment multiple lines by using three consecutive quotes, as illustrated in the next code snippet.

```
#the next line prints Hello in the Command Prompt
print (“Hello”)
#the next line declares the variable name
Name = 0 #sets the value of the variable name to 0
name = “Hello”;#sets the value of the variable name to “Hello”
‘’’
Several lines
Are commented at
a time
‘’’
```

In addition to providing explanations about your code, you can also use comments to prevent part of your code from being executed. This is very useful when you would like to debug your code and find where the errors or bugs might be, using a very simple method. By commenting part of your code, and by using a process of elimination, you can usually find the issue quickly. For example, you can comment all the code and run the script, then comment half the code, and run the script. If it works, it means that the error is within the code that has been commented, and if it does not work, it means that the error is in the code that has not been commented. In the first case (if the code works), we could then just comment half of the portion of the code that has already been commented. So, by successively commenting more specific areas of our code, we can get to discover what part of the code includes the bug. This process is often called **dichotomy** as we successively divide a code section into two. It is usually effective to debug your code because the number of iterations (dividing part of the code in two) is more predictable and also potentially less time-consuming. For example, for 100 lines of codes, we can successively narrow down the issue to 50, 25, 12, 6, and 3 lines and therefore use 5 to 6 iterations in this case, rather than going through the whole 100 lines of code.

Variables

A variable is a container. It includes a value that may change over time. When using variables, we usually need to: (1) declare the variable, (2) assign a value to this variable, and (3) possibly combine this variable with other variables using operators. This is illustrated in the next code snippet.

Note that in Python, the type of the variable does not have to be specified.

```
my_age = 20 #we declare the variable
my_age = int(20) #we declare the variable as an integer; this is called casting
my_age = my_age + 1 #we add 1 to the variable my_age
```

In the previous example, we have declared a variable **my_age**, its type is **int** (integer), we set it to **20** and we then add **1** to it.

In Python, thanks to casting, you can specify the type of a variable; this being said, the type of a variable can change throughout the programme, as illustrated in the next code snippet.

```
my_age = int (12)
my_age = “Not sure”
```

Note that in the previous code we have assigned the value **my_age + 1** to **my_age**; the **=** operator is an assignment operator; in other words, it is there to assign a value to a variable and is not to be understood in a strict algebraic sense (i.e., that the values of the variables on both sides of the **=** sign are equal).

When using variables, there are a few things that we need to determine including their name, their type, and their scope:

- **Name of a variable:** A variable is usually given a unique name so that it can be identified uniquely. The name of a variable is usually referred to as an identifier; it can contain letters, digits, a minus, or an underscore, and it usually begins with a letter. Identifiers cannot be keywords. For example, the keyword **if** cannot be a variable name.

- **Type of variable:** variables can hold several types of data including numbers (e.g., integers, doubles, or floats), text (i.e., strings or characters), Boolean values (e.g., True or False), or arrays as illustrated in the next code snippet.

```
my_age = str (“Patrick”) "#the text is declared using double quotes
current_year = int (2015) #the year needs no decimals and is declared as an integer
width = float (100.45) #width is declared as a float (i.e., with decimals)
```

- **Variable declaration:** unlike in other languages, a variable does not need to be declared before it can be used in Python. This being said, each variable needs to be assigned a value before it can be used.
- **Scope of a variable:** a variable can be accessed (i.e., referred to) in specific contexts that depend on where in the script the variable was initially declared. We will look at this concept later.
- **Accessibility level:** as we will see later, a Python programme consists of classes; for each of these classes, the methods and variables within can be accessed depending on **accessibility** levels. We will look at this principle later on (there is no need for any confusion at this stage :-)).

Common variable types include:

- **String**: same as text.
- **Int**: integer (1, 2, 3, etc.).
- **Boolean**: True or False.
- **Float**: with a decimal value (e.g., 1.2, 3.4, etc.).

- **Arrays**: a group of variables. If this is unclear, not to worry, this concept will be explained further in this chapter.

Arrays and lists

Sometimes arrays can be used to make your code leaner, by applying features and similar behaviors to a wide range of data.

As we will see in this section, arrays can help to declare fewer variables (for variables storing the same type of information) and to also access them more easily.

To use arrays in Python, you will need to use a specific library called NumPy, which makes it possible to create and manipulate arrays as you would in other programming languages.

This being said, Python natively provides other structures that can be used as arrays, including lists.

A list will make it possible to store multiple items in one variable.

When creating lists, you can create single-dimensional lists and multidimensional lists.

Let’s look at the simplest form of lists: single-dimensional lists. For this concept, we can take the analogy of a group of 10 people who all have a name. If we wanted to store this information using a String variable, we would need to declare (and set) ten different variables.

```
name1 = “”
name2 = “”
name3 = “”
```

While this code is perfectly fine, it would be great to store these in only one variable. For this purpose, we could use a list. A list is comparable to a list of elements that we access using an index. This index usually starts at 0 for the first element in the list.

So let’s see how we could do this with a list; first, we could declare the list as follows:

```
names = []
```

You will probably notice the syntax **name_of_the_list** = **[]**. The syntax **name_of_the_list = []** means that we declare a **list**.

We can then store information in this list as described in the next code snippet.

```
names.append("Paul")
names.append("Mary")
names.append("Pat")
```

In the previous code, we store the name **Paul** as the first element in the list; we store the second element as **Mary**, as well as the last element, **Pat**.

Note that for a list of size **n**, **the index of the first element is 0** and **the index of the last element is n-1**. So for an array of size 10, the index for the first element is 0, and the index of the last element is 9.

If you were to use lists of integers or floats, or any other type of data, the process would be similar.

Now, one of the interesting things you can do with lists is that you can initialize your list in one line, saving you from the headaches of writing 10 lines of code if you have 10 variables, as illustrated in the next example.

```
names = [“Paul”,”Mary”,”John”,”Mark”, “Eva”,”Pat”,”Sinead”,”Elma”,”Flaithri”,
“Eleanor”]
```

This is very handy, as you will see in the next chapters, and this should definitely save you a lot of time.

Now that we have looked into single-dimensional lists, let’s look at multidimensional lists, which can also be very useful when storing information. This type of lists (i.e., multidimensional lists) can be compared to a building with several floors, and on each floor, several apartments. So let’s say that we would like to store the number of tenants for each apartment in a building; we would, in this case, create variables that would store this number for each of these apartments.

The first solution would be to create variables that store the number of tenants for each of these apartments with a variable that refers to the floor, and the number of the apartment. For example, **ap0_1** could be for the first apartment on the ground floor, **ap0_2**, would then be for the second apartment on the ground floor, **ap1_1**, would then be for the first apartment on the first floor, and **ap1_2**, would then be for the second apartment on the first floor. So in term of coding, we could have the following:

```
ap0_1 = 0
ap0_2 = 0
...
```

Using lists instead we could do the following:

```
level1 = [0,0,0,0,0]
level2 = [1,1,1,0,0]
level3 = [1,1,1,1,1]
building = [level1, level2, level3]
building[0][1] = 10
building[0][2] = 1
print (building[0][1])
```

In the previous code:

- We declare our lists.
- We add values to our lists.
- The last line of code prints (in the **Command Prompt**) the value of the first element of the array.

One of the other interesting things with lists that, using a loop, you can write a single line of code to access all the elements of this list, and hence, write more efficient code.

Lists include built-in functions that make it possible, amongst other things to:

- Add an element to the list: the function is called **append**.
- Remove an element from a list: the function is called **remove**.
- Sort a list: the function is called **sort**.
- Copy a list: the function is called **copy**.
- Join two lists: the function is called **join**.

As you can see, there are several built-in functions available to work with lists.

In addition to lists, it is possible to store multiple items in one variable using similar structures called sets, and that include **Tuples** and **Dictionaries**.

Dictionaries

Lists are very useful, and dictionaries, which are special types of lists, take this concept a step further. With dictionaries, you can define a dataset with different records, and each record is accessible through a key instead of an index; for example, let’s consider a class of students, each with a first name, a last name, and a student number. To represent and manage this data, we could create code similar to the following to define a class (i.e. a structure) for a student:

```
class Student:
def __init__(self, first_name, last_name):
self.f_name = first_name
self.l_name = last_name
```

- We could then create code that uses this class as follows:

```
student1 = Student("John", "McCarthy")
student2 = Student ("Mary", "Black")
student3 = Student ("Peter", "Sweeney")
student_dictionary = {"ST101":student1, "ST102":student2, "ST103":student3}
print("Student 101's last name is " + str(student_dictionary["ST101"].l_name)) student1
= Student("John", "McCarthy")
student2 = Student ("Mary", "Black")
student3 = Student ("Peter", "Sweeney")
student_dictionary = {"ST101":student1, "ST102":student2, "ST103":student3}
print("Student 101's last name is " + str(student_dictionary["ST101"].l_name))
```

In the previous code:

- We declare a dictionary of **Students**.

- When declaring the dictionary: the first parameter, which is a **string**, is used as an **index** or a **key**; this index will be the **student id**.
- The second parameter will be an object of type **Student**.
- So effectively we create a link between the **key** and the **Student** object.
- We then add students to our dictionary.
- When using the **Add** method, the first parameter is the **key** (or the **student id** in our case: **ST101, ST102** or **ST124** here), and the second parameter is the student object. This student object is created by calling the constructor of the class **Student** and by passing relevant parameters to the constructor, such as the student's first name and last name.
- Finally, we print the **first name** of a specific student based on its **student id**.

As for lists, **Dictionaries** have several built-in functions that make it easier to manipulate them, including:

- **pop**: to remove a new item from the dictionary.
- **copy**: to copy a dictionary.
- **update**: to update an item within the dictionary.
- **remove**: to remove an item from the dictionary.

Constants

In many programming languages, constants are used to assign a value that will not change over time; while this practice can be useful, constants are not used in Python; this being said, you can indicate that a variable’s value should remain constant by using upper-case letters in its declaration. The following shows you the benefits of constants and how they can be used in Python.

So far we have looked at variables and how you can store and access them seamlessly. The assumption then was that a value may change over time, and that this value would be stored in a variable. However, there may be times when you know that a value will remain constant. For example, you may want to define labels that refer to values that should not change over time, and in this case, you could use constants. Let’s look at the following example: let’s say that the player may have three choices in the game (e.g., referred to as 0, 1, and 2) and that you don’t really want to remember these values, or that you would like to find a way that makes it easier to refer to them. Let’s look at the following code:

```
if (user_choice == 0): print ("you have decided to restart")
if (user_choice == 1): print ("you have decided to stop the game")
if (user_choice == 2): print ("you have decided to pause the game")
```

In the previous code:

- The variable **user_choice** is an integer and is set to **2**.
- We then check its value and print a message accordingly.

Now, you may or may not remember that 0 corresponds to restarting the game; the same applies to the other two values. So instead, we could use variables with a constant value to make it easier to remember (and use) these values. Let’s look at the equivalent code with the use of variables with a constant value.

```
CHOICE_RESTART = 0
CHOICE_STOP = 1
CHOICE_PAUSE = 2
...
...
...
if (user_choice == CHOICE_RESTART): print ("you have decided to restart")
if (user_choice == CHOICE_STOP): print ("you have decided to stop the game")
if (user_choice == CHOICE_PAUSE): print ("you have decided to pause the game")
```

In the previous code:

- We declare three variables that we will not modify over the lifecycle of the game.
- These variables are then used to check the choice made by the user.

In the next example, we use variables that we will not modify over the lifecycle of the game to calculate a tax rate; this is a good practice as the same value will be used across the programme with no or little room for errors when it comes to using the exact same tax rate across the code.

```
VAT_RATE = 0.21;
...
price_before_vat= 23.0
price_after_vat = price_before_vat * VAT_RATE;
```

In the previous code:

- We declare a variable for the VAT rate.
- We declare a variable for the item’s price before the VAT.
- We calculate the amount of tax.

Note that variables with that we will not modify over the lifecycle of the game are usually declared at the beginning of your script in Python.

It is a very good coding practice to use variables that we will not modify over the lifecycle of the game. Using this type of variables makes your code more readable; it saves work when you need to change a value in your code, and it also decreases possible occurrences of errors (e.g., for calculations). Also, note that it is common practice to use uppercase for variables that you will not modify over the lifecycle of the game.

Operators

Once we have declared and assigned values to a variable, we can use operators to modify or combine variables. There are different types of operators including: arithmetic operators, assignment operators, comparison operators, and logical operators.

Arithmetic operators are used to perform arithmetic operations including additions, subtractions, multiplications, or divisions. Common arithmetic operators include **+**, **-**, *****, **/**, or **%** (modulo).

Note that the % (modulo) operator will divide the first number by the second and return the remainder.

```
number1 = 1 #the variable number1 is declared
number2 = 1 # the variable number2 is declared
sum = number1 + number2 # adding two numbers and store them in sum
sub = number1 - number2 # subtracting two numbers and store them in sub
```

Assignment operators can be used to assign a value to a variable and include **=**, **+=**, **-=**, ***=**, **/=** or %=.

```
number1 = 1;
number2 = 1;
number1 += 1; #same as number1 = number1 + 1;
number1 -= 1; #same as number1 = number1 - 1;
number1 *= 1; #same as number1 = number1 * 1;
number1 /= 1; #same as number1 = number1 / 1;
number1 %= 1; #same as number1 = number1 % 1;
```

Note that the **+** operator, when used with strings, will concatenate strings (i.e., add them one after the other to create a new string).

When you need to concatenate a number and a string, you usually need to convert the number to a string first; for example:

```
name = "Cars"
my_number = 3
message = "I have " + str(my_number) + " " + name
print(message)
```

Comparison operators are often used for conditions to compare two values; comparison operators include **==, !=**, **>**, **<**, **>=** and **>=**.

```
number1 = 1; number2 = 3;
if (number1 == number2):print("number1 equals number2")
if (number1 != number2):print("number1 and number2 have different values")
if (number1 > number2): print("number1 is greater than number2")
if (number1 >= number2): print("number1 is greater than or equal to number2")
if (number1 < number2): print("number1 is less than number2")
if (number1 <= number2): print("number1 is less than or equal to number2")
```

Conditional statements

Statements can be performed based on a condition, and in this case, they are called **conditional statements**. The syntax is usually as follows:

```
if (condition): statement
```

This means **if the condition is verified (or true) then (and only then) the statement is executed**. When we assess a condition, we test whether a declaration is true. For example, by typing **if (a == b)**, we mean **"if it is true that a equals b"**. Similarly, if we type **if (a >= b)** we mean **"if it is true that a is greater than or equal to b"**

As we will see later on, we can also combine conditions. For example, we can decide to perform a statement if two (or more) conditions are true. For example, by typing **if (a == b and c == 2)** we mean **"if a equals b and c equals 2"**. In this case, using the operator **and** means **AND**, and that both conditions will need to be true. We could compare this to deciding on whether we will go sailing tomorrow. For example, "**if the weather is sunny and the wind speed is less than 5km/h then I will go sailing**". We could translate this statement as follows.

```
if (weather_is_sunny  ==  true  and wind_speed < 5): I_go_sailing  =  true
```

When creating conditions, as for most natural languages, we can use the operator **OR** noted **or.** Taking the previous example, we could translate the following sentence "**if the weather is too hot or if the wind is faster than 5km/h then I will not go sailing** " as follows.

```
if (weather_is_too_hot  ==  true or wind_speed > 5): I_go_sailing  =
false;
```

Another example could be as follows.

```
if (my_name == "Patrick"): print("Hello Patrick")
```

```
else: print ("Hello Stranger")
```

When we deal with combining true or false statements, we are effectively applying what is called **Boolean logic**. Boolean logic deals with Boolean variables that have two values 1 and 0 (or true and false). By evaluating conditions, we are effectively processing Boolean numbers and applying Boolean logic. While you don’t need to know about Boolean logic in-depth, some operators for Boolean logic are important, including the **not** operator. It means **NOT** or the opposite. This means that if a variable is true, its opposite will be false, and vice versa. For example, if we consider the variable **weather_is_good = true**, the value of **not weather_is_good** will be **false** (its opposite). So the condition **if (weather_is_good == False)** could be also written **if (not weather_is_good)** which would literally translate as “if the weather is **NOT** good”.

Match statements

From Python version 3.10, it is possible to use match statements. You can check your Python version by typing **python –version** in the Command Prompt; if the version is older than 3.10, then please download and install the most recent version of Python.

If you have understood the concept of conditional statements, then this section should be pretty much straightforward. Match statements are a variation on the if/else statements that we have seen earlier. The idea behind match statements is that, depending on the value of a specific variable, we will switch to a particular portion of the code and perform one or several actions. The variable considered for the match structure can be of different types including **integer** or **String**. Let’s look at a simple example:

```
choice = int(1);
match choice:
case 1:
print ("you chose 1")
case 2:
print ("you chose 2")
case 3:
print ("you chose 3")
case _:
print ("Default option")
print ("We have exited the match structure")
```

In the previous code:

- We declare the variable **choice**, as an **integer** and initialize it to **1**.
- We then create a **match** structure whereby, depending on the value of

the variable **choice**, the programme will switch to the relevant section (i.e., the portion of code starting with **1:**, **2:**, etc.). Note that in our code, we look for the values 1, 2, or 3. However, if the variable **choice** does not equal 1, 2, or 3, the program will branch to the section starting with **case _**. This is because this section is executed if any of the other possible choices (i.e., 1, 2, or 3) have not been fulfilled (or selected).

Note that in Python, contrary to other languages, the system will exit the match structure once one of the options (or the default one) has been executed. So the **break** statement that is usually found in other languages to specify to leave the switch structure after executing the commands included in the branch (or the current choice) is no longer necessary.

So let's consider the previous example and see how this would work. In our case, the variable **choice** is set to **1**, so we will enter the **match** structure, and then look for the section that deals with a value of **1** for the variable **choice**. This will be the section that starts with **case 1:**; then the command **print ("you chose 1")** will be executed, we then exit the match structure (implicit break); finally the command **print ("We have exited the match structure")** will be executed.

Match statements are very useful to structure your code and when dealing with mutually exclusive choices (i.e., when only one of the choices can be processed) based on an integer value, especially in the case of menus. Besides, match structures make for cleaner and easily understandable code.

Loops

There are times when you have to perform repetitive tasks as a programmer; many times, these can be fast-forwarded using loops. Loops are structures that will perform the same actions repetitively based on a condition. So, the process is usually as follows:

- Start the loop.
- Perform actions.
- Check for a condition.
- Exit the loop if the condition is fulfilled or keep looping.

Sometimes the condition is performed at the start of the loop, some other times it is performed at the end of the loop.

Let’s take the following example that is using a **while** loop.

```
x = 0;
while (x < 10):
print("x"+str(x))
x += 1
```

In the previous code:

- We set the value of the variable **x**.
- We then create a loop that starts with the keyword **while**.
- We set the condition to remain in this loop (i.e., **x < 10**).
- Within the loop, we increase the value of the variable **x** by **1** and print its value.

So effectively:

- The first time we go through the loop: the variable **x** is increased to **1**; we reach the end of the loop; we go back to the start of the loop and check if **x** is < 10; this is true in this case (**x** = 1).
- The second time we go through the loop: **x** is increased to **2**; we reach the end of the loop; we go back to the start of the loop and check if **x** is <10; this is true in this case (**x** = **2**).
- ...
- The 10th time we go through the loop: **x** is increased to 10; we reach the end of the loop; we go back to the start and check if **counter** is < 10; this is now false in this case (**counter** = 10). As a result, we then exit the loop.

So, as you can see, using a loop, we have managed to increment the value of the variable **x** iteratively, from 0 to 10, but using less code than would be needed otherwise.

Another variation of the code could be as follows:

```
for x in range(0,10):
print ("x"+str(x))
```

In the previous code:

- We declare a loop in a slightly different way: we say that we will use a variable called **x** that will go from 0 to 9 (we exclude the upper boundary which is **10**).
- This variable **x** will be incremented by 1 every time we go through the loop.
- We remain in the loop as long as the variable **x** is less than 10.
- The test for the condition, in this case, is performed at the start of the loop.

Loops are very useful to perform repetitive actions for a finite number of objects, or to perform what is usually referred to as recursive actions. For example, you could use loops to create (i.e., instantiate) 100 objects at different locations (this will save you some code :-)), or to go through an array of 100 (or more) elements.

Classes

When coding in Python, you will be creating code that includes your own classes or uses built-in classes. So what is a class?

As we have seen earlier, Python supports Object-Oriented Programming (OOP). More specifically, Python is what is called a multi-paradigm language as it supports a wide range of programming approaches including object-oriented programming and functional programming.

In OOP, a programme consists of a collection of objects that interact amongst themselves. Each object has one or more attributes, and it is possible to perform actions on these objects using what are called **methods**. Also, objects that share the same properties are said to belong to the same **class**. For example, we could take the analogy of a bike. There are bikes of all shapes and colors; however, they share common features. For example, they all have a specific number of wheels (e.g., one, two or three) or a speed; they can have a color, and actions can be performed on these bikes (e.g., accelerate, turn right, turn left, etc.). So in object-oriented programming, the class would be **Bike**, speed or color would be referred to as member variables, and accelerate (i.e., an action) would be referred to as a member method. So if we were to define a common type, we could define a class called **Bike** and for this class define several member variables and attributes that would make it possible to define and perform actions on the objects of type **Bike**.

This is, obviously, a simplified explanation of classes and objects, but it should give you a clearer idea of the concept of object-oriented programming, if you are new to it.

Defining a class

So now that we have a clearer idea of what a class is, let’s see how we could define a class. So let’s look at the following example.

```
class Bike:
def __init__(self):
self.speed = 0
self.color = "blue"
self.name = ""
def accelerate(self):
self.speed += 1
def turn_right(self):
print("Turning Right")
def calculate_distance(self):
print("Calculating Distance")
```

In the code above, we have defined a class, called **Bike**, that includes three member variables (**speed, color** and **name**) as well as two member methods (**accelerate, turn_right,** and **calculate_distance**). It also include a function called **__init__** which is called a constructor. Let’s look at the script a little closer; you may notice a few things:

- The name of the class is preceded by the keyword **class**.
- Three variables called **speed**, **color** and **name** are defined; they are called member variables because they are declared before any method and are therefore accessible throughout the class.
- Three functions are declared: **accelerate**, **turn_right** and **calculate_distance**.

If you have worked with other programming languages such as C#, you may be used to defining the access modifiers for both the member methods and variables; in Python, these are set to **public** by default.

Accessing class members and variables

Once a class has been defined, it’s great to be able to access its member variables and methods. In Python (as for other object-oriented programming languages), this can be done using the **dot notation**.

The dot notation refers to **object-oriented programming**. Using dots, you can access properties and methods related to a specific object.

Once a class has been defined, objects based on this class can be created. For example, if we were to create a new **Bike** object, based on the code that we have seen previously, the following code could be used.

```
my_bike = Bike()
```

In the previous code, we instantiate a new object called **my_bike** from the class **Bike**. So, this code will effectively create an object based on the “template” **Bike**. You may notice the syntax:

```
variable_name = data_type()
```

By default, this new object will include all the member variables and methods defined earlier. So it will have a color and a speed, and we should also be able to access its **accelerate** and **turn_right** methods. So how can this be done? Let’s look at the next code snippet that shows how we can access these.

```
my_bike = Bike()

my_bike.accelerate()

my_bike.color = "Blue"
```

In the previous code:

- The new bike **my_bike** is created.
- The speed is then increased after calling the **accelerate** method. This

function can be called using the dot notation because it is a member function.

- We also set the color for the bike.
- Note that to call an object’s method we use the dot notation.

Constructors

As we have seen in the previous sections, when a new object is created, it will, by default, include all the member variables and methods. To create this object, we can use the name of the class, followed by an opening and closing round bracket, as per the next example.

```
my_bike = Bike();
my_bike.accelerate();
```

In fact, it is possible to change some of the properties of the new object created when it is initialized. For example, instead of setting the speed and the color of the object as we have done in the previous code, it would be great to be able to set these automatically and pass the parameter accordingly when the object is created. Well, this can be done with what is called a **constructor**.

A constructor literally helps to construct your new object based on parameters (also referred to as arguments) and instructions. So, for example, let’s say that we would like the color of our bike to be specified when it is created; we could modify the **Bike** class, by adding the following method:

```
def _init__ (self, new_color):
self.color = new_color;
```

This constructor takes a **String** as a parameter (**self** refers to the class); so after modifying this constructor (as per the code above), we could then create a new bike object as follows:

```
my_bike = Bike.new(“Blue”);
```

This being said, we could also modify the constructor so that if we call the constructor without parameters, a default value is assigned for the color, as follows:

```
func _init (new_color = ”Blue”):
color = new_color;
```

In the previous code, we specify that by default, if no parameters are entered, the color will be blue; however, if a parameter has been entered, then it will be used as the new color.

Furthermore, we could create a constructor that includes several parameters, all with default values as follows:

```
def __init__(self, new_name ="My Bike", new_color = "Blue", new_speed = 0):
self.color = new_color
self.name = new_name
self.speed = new_speed
```

In the previous code the define a constructor with the following features:

- It takes three parameters for the **name**, **color**, and **speed** of the bike.
- All parameters have a default value: “**A New Bike**”, “**Blue**”, and **0** respectively.
- If parameters were entered they are then used to initialize the member variables **name**, **color** and **speed**.

We would then call this constructor as follows:

```
my_bike = Bike("Fast Bike", "Red", 100)
my_bike2 = Bike()
```

In the previous code:

- We create two different bikes.
- For the first bike, we call the constructor by passing parameters for the name, color and speed or the bike

- We also create a second bike but with no parameters this time; this means that this bike will have the default values assigned to its member variables **name**, **color** and **speed** (i.e., “**My Bike**”, “**Blue**”, and **0**).

Inheritance

I hope everything is clear so far, as we are going to look at the concept of inheritance, which is very important in object-oriented programming.

The main idea behind inheritance is that objects can inherit their properties from other objects (their parent). As they inherit these properties, they can remain identical or evolve and overwrite some of these inherited properties. This is very interesting because it makes it possible to minimize your code by creating a parent class with general properties for all objects sharing similar features, and then, if need be, overwrite and customize some of these properties for the children.

Let’s take the example of vehicles: vehicles would generally have the following properties:

- Number of wheels.
- Speed.
- Number of passengers.
- Color.
- Capacity to accelerate.
- Capacity to stop.

So we could create the following class for example:

```
class Vehicle:
def __init__(self):
self.nb_wheels = 0
self.speed = 0.0
self.nb_passengers = 0
self.color = ""
```

```
def  accelerate():
self.speed+=1
```

These features could apply for example to cars, bikes, motorbikes, or trucks. However, all these vehicles also differ; some of them may or may not have an engine or a steering wheel. So we could create a subclass called **MotorizedVehicles**, based on **Vehicles**, but with specificities linked to the fact that they are motorized. These added attributes could be:

- Engine size.
- Petrol type.
- Petrol levels.
- Ability to fill up the tank.

The following example illustrates how this class could be created.

```
class MotoredVehicle (Vehicle):
def __init__(self):
super().__init__()
self.nb_wheels = 4
self.engine_size = 1
self.petrol_type = "Diesel"
self.petrol_levels = 100
def fill_up_tank():
petrol_levels += 10
```

- At the beginning of the file we specify that the current class inherits from the class **Vehicle**. So it will, by default, avail of all the methods and variables already included in the class **Vehicle**.
- We then define the name of the class (i.e., **MotoredVehicle**).

- We call the constructor from the parent using the keyword **super**.
- We update the value of the member variable **nb_wheels**.
- We have created new member variables that will be related to instances of the **MotoredVehicle** class (i.e., **engine_size**, **petrol_type** or **petrol_levels**).
- We have created a new member method for this class, called **fill_up_tank**.
- In the previous example, you may notice that the methods and variables that were defined for the class **Vehicle** do not appear here; this is because they are implicitly added to this new class, since it inherits from the class **Vehicle**.

When using inheritance, the parent is usually referred to as the **base class**, while the child is referred to as the **inherited class**.

Now, while the child inherits "behaviors" from its parents, these can always be modified or, put simply, overwritten. In Python, a method with the same name as a method defined in the parent will override the latter.

This is illustrated in the following code snippets.

```
class Vehicle:
def __init__(self):
self.nb_wheels = 0
self.speed = 0.0
self.nb_passengers = 0
self.color = ""
def  accelerate(self):
self.speed+=1
```

In the previous code snippet, we have defined the class called **Vehicle**; as you may have noticed, it includes member variables **nb_wheels**, **speed**, **nb_passengers**, and color. It also includes a member method called **accelerate** that increases the **speed** by one when it is called.

The next snippet illustrates the definition of a class called **MotoredVehicle**. As you will see it includes additional member variables, as well as a function called **accelerate**; however, contrary to the **accelerate** function defined for the parent class (**Vehicle**), this function increases the speed by **10** (and not 1).

```
class MotoredVehicle (Vehicle):
def __init__(self):
super().__init__()
self.nb_wheels = 4
self.engine_size = 1
self.petrol_type = "Diesel"
self.petrol_levels = 100
def  accelerate(self):
self.speed += 10
def fill_up_tank(self):
petrol_levels += 10
```

Finally, in the next code snippet we create two types of objects: an instance of the class **Vehicle** (**v1**) and an instance of the class **MotoredVehicle** (**v2**).

```
v1 = Vehicle.new();
v1.accelerate();
print ("V1 Speed: " + str(v1.speed));
v2 = MotoredVehicle.new();
```

```
v2.accelerate();
print ("V2 Speed: " + str(v2.speed));
```

As you may have noticed:

- **v1** is created as an instance of the class **Vehicle**.
- The function **accelerate** is called; because this object is an instance of the class **Vehicle**, the function **accelerate** that is a member of the class **Vehicle** will be called; as a result, the speed should be increased by 1.
- **v2** is created as an instance of the class **MotoredVehicle**.
- The function **accelerate** is called; because the object v2 is an instance of the class **MotoredVehicle**, and because this class has its own function **accelerate**, the function **accelerate** that is a member of the class **MotoredVehicle** will be called; as a result, the speed should be increased by **10**.

If we were to run the last snippet, the following would be displayed in the **Command Prompt**:

```
V1 Speed: 1
V2 Speed: 10
```

Functions and Methods

While functions include a series of statements to be executed; methods are functions defined as part of a class. This section will focus on functions; however, most of the information will also be applicable to methods.

Functions can be compared to a friend or a colleague to whom you gently ask to perform a task, based on specific instructions, and to return the information to you then. For example, you could ask your friend the following: "**Can you please tell me when I will be celebrating my 20th birthday given that I was born in 2000**". So you give your friend (who is good at Math :-)) the information (date of birth) and s/he will calculate the year of your 20th birthday and give this information back to you. So in other words, your friend will be given an input (i.e., the date of birth) and return an output (i.e., the year of your 20th birthday). Methods work exactly this way: they are given information (and sometimes not), perform an action, and then (sometimes, if it is needed) return information.

In programming terms, a function is a block of instructions that performs a set of actions. A method is executed when invoked (or put more simply **called**), or when an event occurs (e.g., the player has clicked on a button or the player collides with an object; we will see more about events in the next section). As for member variables, functions and methods are declared and they can also be called.

Functions are very useful because once the code for a method has been created, it can be called several times without the need to rewrite the same code over and over again. Also, because functions can take parameters, a function can process these parameters and produce or return information accordingly; in other words, they can perform different actions and produce different information based on the input. As a result, functions can do one or all of the following:

- Take parameters and process them.
- Perform an action.
- Return a result.

A function has a syntax and can be declared in at least two ways:

```
def name_of_the_function():
Perform actions here...
```

Please note that any statement part for a specific method needs to be indented.

In the previous code, the method does not take any input, neither does it return an output. It just performs actions.

OR

```
def name_of_the_function(parameter1):
Perform actions here...
```

In the previous code snippet, the method takes one parameter and then performs actions.

Let’s look at the following method for instance.

```
def calculate_sum(a,b):
return (a+b)
```

In the previous code:

- The function is declared and is called **calculate_sum**.
- The function takes two parameters.

- The function returns the sum of the two parameters which are referred to as **a** and **b** within this function.

A function can then be called using the **()** operator, as follows:

```
name_of_the_function1 ();
name_of_the_function2 (parameter1);
var test = name_of_the_function3 (parameter2);
```

In the previous code, a function is called with no parameter (line 1), or with a parameter (line 2). In the third example (line 3), a variable called **test** will be set with the value returned by the function **name_of_the_function3**.

Note that when a method is declared (i.e., a function within a class), it needs to include the parameter **self** which refers to the class itself.

So for example while a function could be declared as

```
def calculate_sum(a,b):
return (a+b)
```

It would be defined with a class as follows:

```
def calculate_sum(self, a,b):
return (a+b)
```

Default parameters and return types for functions

Now that you know a bit more about functions, we will see how the definition of functions can be refined to include a return type, a type for the parameters, or default parameters.

Let's start with the parameters; using Python, you can specify a type for the parameters passed to the function, as illustrated in the next code snippet.

```
def calculate_sum(a:int,b:int):
return (a+b)
```

In the previous code:

- We declare a function called **calculate_sum**.
- This function takes two parameters.
- Each parameter (referred to as **a** and **b**) should be integers. This means that passing parameters that are not integers (e.g., float or string) will result in an error.
- The function then returns the sum of the two integers passed as parameters.

Now that we have seen how to specify the type of the parameters passed to a function, let's see how we can specify default values; in Python, you can specify default values for each of the parameters that should be passed to a function, as illustrated in the next code snippet.

```
def calculate_sum(a:int=0,b:int=0):
return (a+b)
```

In the previous code:

- We declare a function called **calculate_sum**.

- This function takes two parameters, each of them are integers.
- The default value for the first parameter is **0**.
- The default value for the second parameter is **0**.
- As a result, if no parameters are specified when calling this function, it will return **0** (i.e., 0 + 0)

Let’s illustrate this with the following code:

```
sum1 = calculate_sum(1,2);
print("Sum1: " + str(sum1))
sum2 = calculate_sum()
print ("Default Sum: " + str(sum2))
```

In the previous code:

- We create a variable called **sum1**; we then call the function **calculate_sum**, passing the values **1** and **2**, and what is returned by the function (**1 + 2**) is then stored in the variable **sum1**.
- We then print the value of the variable **sum1**.
- We create a variable called **sum2**; we then call the function **calculate_sum**, passing no parameters (i.e., empty round brackets), and what is returned by the function is then stored in the variable **sum2**. In that case, because no parameters were passed to the function, the default value for the parameters will be **0**; as a result, the function will return **0** and this value will be stored in the variable called **sum2**.
- We then print the value of the variable **sum2**.

If we were to run the previous code snippet, the **Command Prompt** should display the following:

```
Sum1: 3
```

```
Default Sum: 0
```

Last but not least, let’s look at the return type for a function. As we have seen previously, functions can return values, and it is also possible to specify the return type for a function; this is especially useful to ensure that the correct type of data is returned by the function or that this function doesn’t return any value if it is not meant to.

Let’s look at the following code snippet to see how this can be done:

```
def calculate_sum(a:int=0,b:int=0) -> int:
return (a+b)
```

In the previous code, we declare the function **calculate_sum**, as we have done before; however, we add “**-> int**” just before the colons to indicate that this function should return an integer.

Scope of variables

Whenever you create a variable in Python, you will need to be aware of its scope so that you can use it accordingly.

The scope of a variable refers to where you can use this variable in a script. In Python, we usually make the difference between **member variables**, **global variables** and **local variables**.

When you create a class definition along with member variables, these variables will be seen by any function within your class.

Member variables can be used anywhere in your class (or outside, if they are public). These variables need to be declared with the prefix **self** and the dot notation within the class ; they can then be used anywhere in the class as illustrated in the next code snippet.

```
class Bike:
def __init__(self):
self.color = “red”
self.name = “The Bike”
self.speed = 10
```

In the previous code we declare the member variables **color**, **name** and **speed** as a member variable and access them from the method accelerate.

Local variables are declared within a function and are to be used only within this function, hence the term local, because they can only be used locally, as illustrated in the next code snippet.

```
def function1():
a = "Hello"
def function2():
b = "World"
```

In the previous code, **a** is a local variable to the function **function1**, and can only be used within this function; **b** is a local variable to the function **function2**, and can only be used within this function.

This being said, sometimes a variable can also be declared outside of any function, and it is as a result called **global** as it can then be accessed from anywhere in the program. Let’s look at the next code snippet:

```
the_name = "Pat"
def function1():
a = "Hello"
print (a+ " "+str(the_name))
function1()
```

In the previous code:

- The variable **the_name** is declared outside of any function, and it is therefore global and accessible throughout the script.
- The function **function1** uses the variable **the_name** to create a string
- The message is displayed.

Note that while you can use a global variale inside a function; it is necessary to delare it inside the function with the keyword global if you intend to modify this global variable inside the function, as illustrated in the next code snippet:

```
the_name = "Pat"
def function1():
global the_name
the_name = the_name + str (“Murphy”)
a = "Hello"
```

```
print (a+ " "+str(the_name))
function1()
```

One more thing: lets' imagine that the locale function uses a variable for which the name is the same as one of the global variables; in that case, by default, the value of the local variable will be used; this being said, if you'd prefer to use the value of the global variable in that case, you need to specify it using the keyword global, as illustrated in the next code snippet.

```
the_name = "Pat"
def function1():
global the_name
a = "Hello"
print (a+ " "+str(the_name))
function1()
print ("Value of the global variable: "+str(the_name))
```

In the previous code:

- We define a global variable called **the_name**.
- We declare a function **function1**.
- Within the function we declare a global variable **the_name**.
- We then print then a new message.
- After calling the function **function1**, we also print the new value of the global variable **the_name**.

The previous code would display the following message and it shows that we have not modified the value of the global variable **the_name** within the function.

```
Value of the global variable: Pat
```

However, if we consider the following code:

```
the_name = "Pat"
def function1():
a = "Hello"
global the_name
the_name = "Patrick"
print (a+ " "+str(the_name))
function1()
print ("Value of the global variable: "+str(the_name))
```

In the previous code:

- We define a global variable called **the_name**.
- We declare a function **function1**.
- Within the function we use the keyword global to specify that the variable **the_name** refers to the global variable **the_name**.
- We then print the a new message.

After calling the function **function1**, we also print the new value of the global variable **the_name**.

The previous code would display the following message and it shows that we have modified the value of the global variable **the_name** within the function by using the **global** keyword.

```
Value of the global variable: Patrick
```

Events or Signals

Throughout this book and in Python, you will employ events that can be compared to a notification that you may receive on your phone. For example, when an event occurs, such as the alarm going off (event), we can either get up (action) or decide to go back to sleep. When you receive a message (event), you can decide to read it (action), and then reply to the sender (another action).

In computer terms, events are quite similar, although the events that we will be dealing with will be slightly different. So, in a game, we could be waiting for the user to press a key (event) and then move the player character accordingly (action), or wait until the user clicks on a button on screen (event) to load a new scene (action).

In this book, we will be using both Python and Pygame, a library that is dedicated to create video games with Python. With Pygame, we will be able to catch and handle events related to the user interface such as mouse clicks or keyboard strokes.

For every one of these events, we will then be able to decide what should be done in each case; for example, when the player presses the left mouse button on the screen, we may detect the position of the mouse and draw a specific shape at that position.

We could also detect weather the user has pressed one of the keyboard arrow keys and move the corresponding character onscreen accordingly (i.e., up, down, left or right).

As you can see, we can deal with a wide range of events and signals in our game, and we will get to do that later. In this book, we will essentially be dealing with the following events:

- Moving the mouse.
- Pressing a mouse button.
- Pressing a key on the keyboard.

Workflow to create a script

There are many ways to create and use scripts in Python, but generally the process is as follows:

- Create a new file with the text editor of your choice.
- Save this file with the **py** extension.
- Compile this file using the **Command Prompt**.

Coding convention

When you are coding, there are usually naming conventions based on the language that you are using. These often provide better clarity for your code and depend on the language that you will be using. In **Python**, you will usually need to do the following:

- Name **classes** using **Pascal Casing** (i.e., a script): In Pascal casing the first letter of each word included in a name is capitalized; for example: **MyClass**.
- Name **variables** using **Snake Casing**: In Snake casing the words included in a name are in lower-case and are separated by an underscore; for example: **my_variable**.
- Name **functions** using **Snake Casing**: In Snake casing the words included in a name are using lower-case and are separated by an underscore: for example, **my_function**.
- When you create functions or blocks of instructions, the code within needs to be indented.
- Use, as much as possible, one statement per line.

Once you feel comfortable with Python and if you want to know more about the official naming scheme for Python, you may look at Python official naming guidelines:

https://peps.python.org/pep-0008/

A few things to remember when you create a script (checklist)

As you create your first scripts in the next chapter, there will be, without a doubt, errors and possibly hair pulling :-). You see, when you start coding, you will, as for any new activity, make small mistakes, learn what they are, improve your coding, and ultimately get better at writing your scripts. As I have seen students learning to code, there are some common errors that are usually made. These don't make you a bad programmer; on the contrary, it is part of the learning process.

We all learn by trial and error, and making mistakes is part of the learning process.

So, as you create your first script, set any fear aside, try to experiment, be curious, and get to learn the language. It is like learning a new foreign language: when someone from a foreign country understands your first sentences, you feel so empowered! So, it will be the same with Python, and to ease the learning process, I have included a few tips and things to keep in mind when writing your scripts, so that you progress even faster. You don't need to know all of these by now (I will refer to these later on, in the next chapter), but just be aware of it and also use this list if any error occurs (this list is also available as a pdf file in the resource pack, so that you can print it and keep it close by). So, watch out for these :-).

- Indent your code properly for functions or blocks of instructions.

- All variables are written consistently (e.g., spelling and case). The name of each variable is case-sensitive; this means that if you declare a variable

my_variable but then refer to it as **my_Variable** later on in the code, this may trigger an error, as the variable **my_Variable** and **my_variable**, because they have a different case (upper- or lower-case **V**), are seen as two different variables.

- All variables are set to a value before being used.

- The type of the argument passed to a function is the type that is required by this function.

- The type of the argument returned by a function is the type that is required to be returned by this function.

- Built-in functions are spelled with the proper case.

- Use **Snake casing** for variables and functions and or **Pascal casing** for file and class names.

- When calling a function, the exact name of this function (i.e., case-sensitive) is used.

- When referring to a variable, it is done with regards to the access type of the variable (e.g., member or local).

- Local variables are declared and can be used within the same function.

- Global variables are declared outside functions and can be used anywhere within the file.

Common Errors

The following is a list of common errors in Python and their likely causes:

```
SyntaxError: '(' was never closed
```

- Meaning: you need to add a closing bracket.

```
TypeError: can only concatenate str (not "int") to str
```

- Meaning: When you concatenate a string and an int, the latter needs to be converted to a string (using str) .

```
NameError: name 'score' is not defined
```

- Meaning: You have used a variable without setting its value first.

```
NameError: name 'true' is not defined. Did you mean: 'True'?
```

- Meaning: Boolean values need to be spelt True or False (upper-case T or F) .

```
SyntaxError: expected ':'
```

- Meaning: you need to add a colon (possibly after the definition of a function or before a block of instructions (e.g., conditional statement).

```
NameError: name 'my_func2' is not defined
```

- You are calling a function that has not been defined before or the spelling needs to be corrected.

Level roundup

Summary

In this chapter, we have become familiar with Python and different programming concepts. We also investigated object-oriented programming. In the next chapter, we will harness these skills to be able to create (and execute) our first script.

Quiz

It is now time to test your knowledge.

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following statement will print the text **Hello World** in the **Command Prompt**.

```
print(“Hello World”);
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The value of the variable **c** in the following statement will be **3**.

```
a =1
b = 1
c = a + b;
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The value of the variable **full_name**, in the following code snippet, will be **JohnPaul**.

```
f_name = “John”;
l_name = “Paul”;
full_name = f_name + l_name;
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following code snippet will print **I will not go sailing**.

```
wind_is_strong = True;
if (wind_is_strong): print (“I will not go sailing”);
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

 The following code snippet will print **I will not go sailing**.

```
weather_is_sunny = True
wind_is_strong = False
i_will_go_sailing = False
if (weather_is_sunny and not wind_is_strong ): print ("I will go sailing")
if (not weather_is_sunny or wind_is_strong ): print ("I will not go sailing")
```

1. Spot three coding mistakes in the following snippet.

```
test = 0
test2 == 0
test3 = 0;
test 3 = test1 + test2;
```

1. Consider the method described in the next code snippet, and select the correct way to call it (i.e., A, B, or C):

a) **display_A_message()**

b) **displayAMessage()**

c) **display_a_message()**

```
def display_a_message ():
print (“Hello”)
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The value of the variable **counter** in the following code snippet will be **3** after the code has been executed.

```
counter = 0;
counter = counter + 1;
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following code will print the message **Hello**

```
def print_message():
print (“Hello”);
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

A local variable can be used from any part of a script.

Answers to the Quiz.

1. TRUE.
2. FALSE.
3. TRUE.
4. TRUE
5. FALSE.
6. Spot three coding mistakes in the following snippet.

```
test = 0
    test2 == 0 #should be test2 = 0
    test3 = 0;
    test 3 = test1 + test2;#no space between test and 3; test1 was not declared.
```

1. **C.**
2. FALSE (should be 1).
3. TRUE.
4. FALSE.

Checklist

If you can do the following, then you are ready to proceed to the next chapter:

- Understand the concept of classes.
- Know how to call a method.
- List and understand at least three types of variables in Python.
- Answer at least 7 out of 10 of the questions correctly in the quiz.

CHAPTER 2: CREATING YOUR FIRST SCRIPT

In this section, we will start to create Python code. Some of the objectives of this section will be to:

- Introduce Python.
- Explain some basic scripting concepts.
- Explain how to display information from the code to the **Command Prompt**.

After completing this chapter, you will be able to:

- Understand basic concepts in Python.
- Understand the best coding practices.
- Code your first script in Python.
- Create classes, methods and variables.
- Instantiate objects based on your own classes.
- Use built-in methods.
- Use conditional statements.

You can skip this chapter if you are already familiar with Python or if you have already created and used Python scripts.

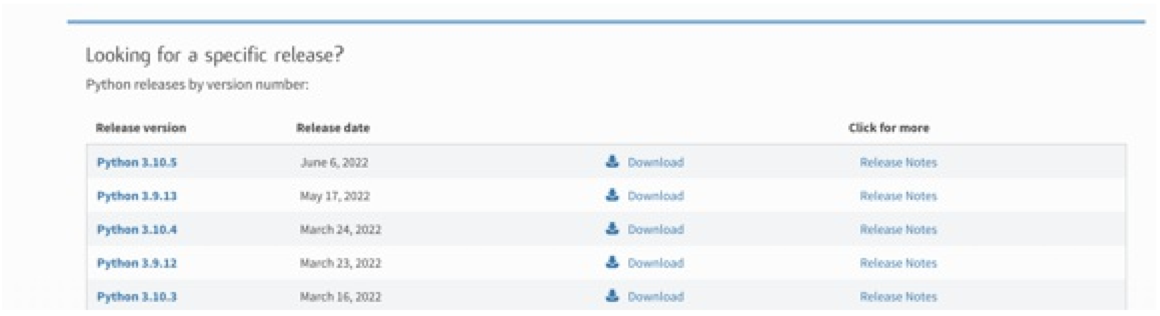
installing Python and an IDE

Before you can compile your first code in Python, you will need to do the following:

- Download the latest version of Python.
- Install Python.
- Install a text editor of your choice or an Integrated Development Environment (IDE).

To download and install Python please do the following:

- Open the following page: https://www.python.org/downloads/.
- Scroll down to the section entitled "**Looking for a Specific Release**".



Looking for a specific release?

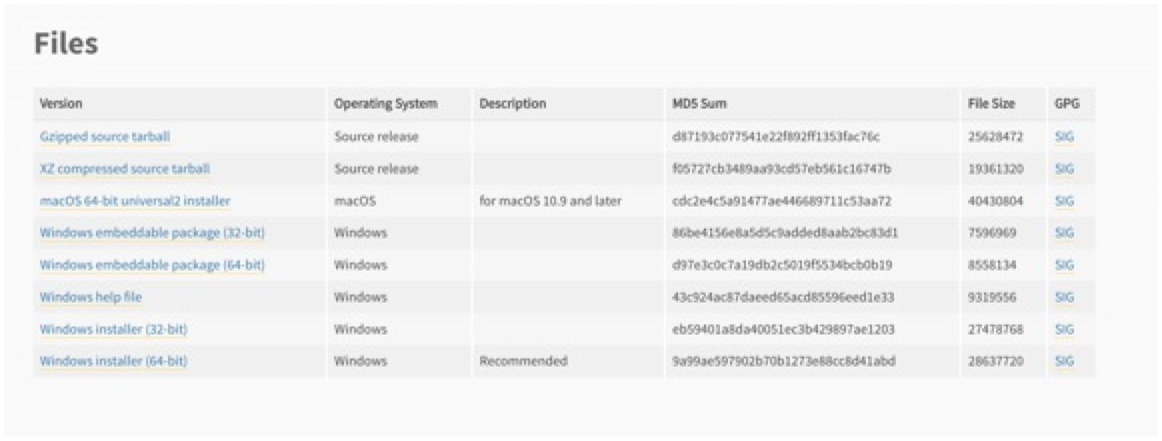
Python releases by version number:

Release version	Release date		Click for more
Python 3.10.5	June 6, 2022	Download	Release Notes
Python 3.9.13	May 17, 2022	Download	Release Notes
Python 3.10.4	March 24, 2022	Download	Release Notes
Python 3.9.12	March 23, 2022	Download	Release Notes
Python 3.10.3	March 16, 2022	Download	Release Notes

- Click on the latest release (e.g., Python 3.10.5), this should open a new window that lists the release available for your operating system. This book uses Python 3.10.

Bear in mind that code compiled in one version of Python might not work using a different version as Python is constantly evolving and features present in one version might or might not be available in the next version.

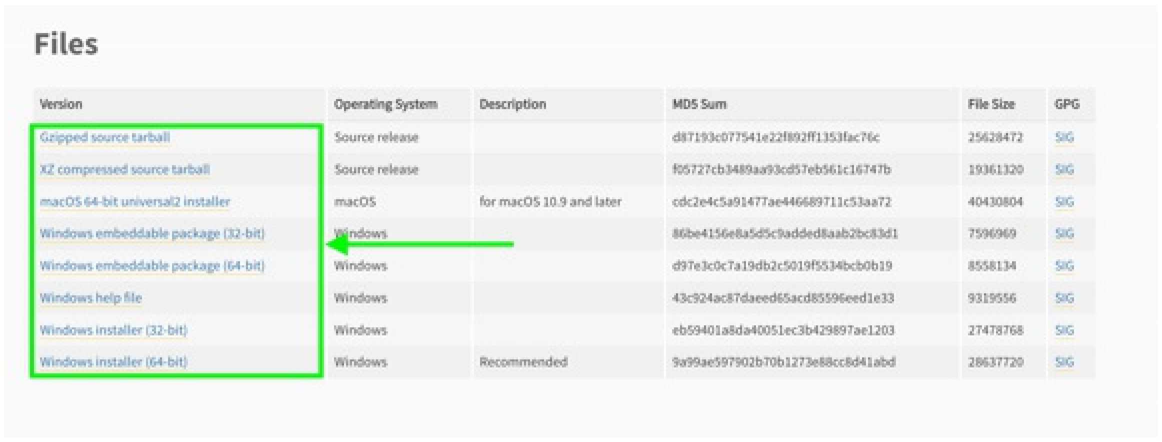
- Scroll down to the section called "**Files**".



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		d87193c077541e22f892ff1353fac76c	25628472	SIG
XZ compressed source tarball	Source release		f05727cb3489aa93cd57eb561c16747b	19361320	SIG
macOS 64-bit universal2 installer	macOS	for macOS 10.9 and later	cdc2e4c5a91477ae446689711c53aa72	40430804	SIG
Windows embeddable package (32-bit)	Windows		86be4156e8a5d5c9added8aab2bc83d1	7596969	SIG
Windows embeddable package (64-bit)	Windows		d97e3c0c7a19db2c5019f5534bcb0b19	8558134	SIG
Windows help file	Windows		43c924ac87daeed65acd85596eed1e33	9319556	SIG
Windows installer (32-bit)	Windows		eb59401a8da40051ec3b429897ae1203	27478768	SIG
Windows installer (64-bit)	Windows	Recommended	9a99ae597902b70b1273e88cc8d41abd	28637720	SIG

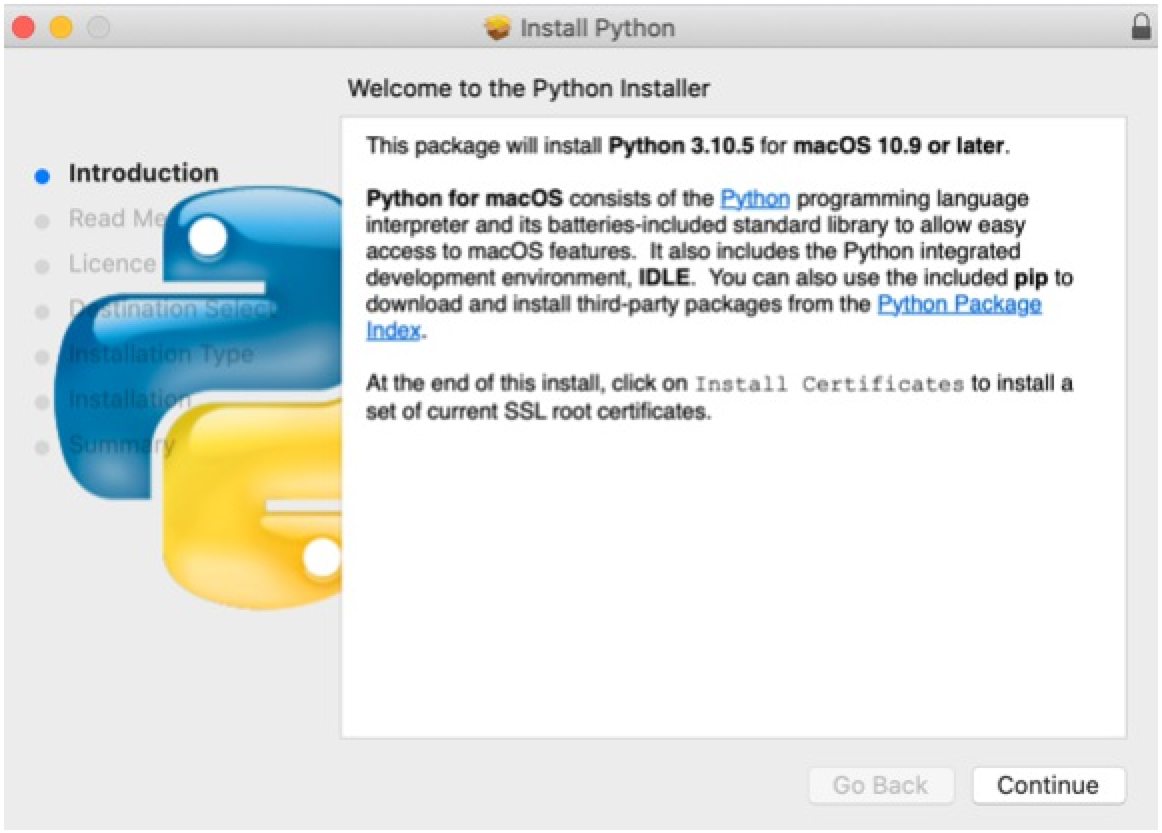
- Click on the relevant link, based on your operating system to download the installer.



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		d87193c077541e22f892ff1353fac76c	25628472	SIG
XZ compressed source tarball	Source release		f05727cb3489aa93cd57eb561c16747b	19361320	SIG
macOS 64-bit universal2 installer	macOS	for macOS 10.9 and later	cdc2e4c5a91477ae446689711c53aa72	40430804	SIG
Windows embeddable package (32-bit)	Windows		86be4156e8a5d5c9added8aab2bc83d1	7596969	SIG
Windows embeddable package (64-bit)	Windows		d97e3c0c7a19db2c5019f5534bcb0b19	8558134	SIG
Windows help file	Windows		43c924ac87daeed65acd85596eed1e33	9319556	SIG
Windows installer (32-bit)	Windows		eb59401a8da40051ec3b429897ae1203	27478768	SIG
Windows installer (64-bit)	Windows	Recommended	9a99ae597902b70b1273e88cc8d41abd	28637720	SIG

- Once the download is complete please launch the installer and follow the instructions as per the next figure.



This package will install **Python 3.10.5** for **macOS 10.9 or later**.

Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, **IDLE**. You can also use the included **pip** to download and install third-party packages from the Python Package Index.

At the end of this install, click on `Install Certificates` to install a set of current SSL root certificates.

Once the install is complete, you can check that Python has been installed by doing the following:

- Open the **Command Prompt** (or terminal if you are using a Mac).
- Type the text **python -v** followed by carriage return.
- The system should then print the current version.

Note that it is possible to install and compile with different versions of Python. For more information on installing Python, please see the official pages here:

https://docs.python.org/3/installing/index.html

Finally, you will also need a text editor; there are many of them available for free, with several including interesting options such as syntax highlighting, autocomplete, and debugging

Popular IDEs and text editors for Python include: IDLE (the default IDE provided after installing Python), PyCharm, Visual Studio, or Sublime Text 3.

While you can install and use an IDE of your choice, it might probably be better to start with a simple IDE for now. So in this book, we will be using **Sublime Text 3**, a simple, yet efficient IDE that you will be able to use to create your first Python scripts.

Please do the following:

- Open the following page: https://www.sublimetext.com/3
- This should offer you an installer based on your operating system (e.g., Mac OS or Windows).
- Download the installer and launch the installation.
- Once the installation is complete you can launch **Sublime Text**.

Your First Script

In this section, we will get to create your first Python script:

First let’s create a variable of type **integer** called **number** as a member variable.

- Please open **Sublime Text** if it is not open yet.
- Select **File | New File**.
- This will create a new file.
- Save this file: Select **File | Save As**, select a location of your choice for the script and save it as **my_first_script.py**.
- This will create a blank file.
- Then type the following code at the beginning of the file.

```
number = 12
```

- This code declares the variable called number and sets its value to **12**; this variable is declared outside of any function and can therefore be accessed from anywhere within the script.
- Then type the following code after the previous statement (you can replace the word **Patrick** with your own name if you wish):

```
my_name = “Patrick”
```

- As you type this line, make sure that the name of the variable is spelled properly with proper case (i.e., lower-case letters).

If you happen to copy/paste this code, please ensure that you are using straight double quotes otherwise you may get an error.

- Then type the following code after the previous statement to display a message in the **Command Prompt**.

```
print ("Hello " + my_name + " , your number is "+ str(number))
```

- This should print the message **“Hello Patrick, your number is 12”** in the **Command Prompt** after the code has been compiled and executed. You may notice the quotes around the word **Patrick**, this means that the text **Hello** will be displayed and we will add the value of the variable **my_name** to it. So these two strings will be concatenated (i.e., grouped) to form a dynamic sentence (i.e., a sentence for which the content will vary) for which the content will depend on the value of the variables **my_name** and **number**. You may also note that we use the code **str(number)**, and this is to convert the number to a **String** so that it can be concatenated to the rest of the sentence.

So at this stage, your code should look as follows (and if it doesn’t, you can use the next code snippet as a template):

```
number = 12
my_name = "Patrick"
print ("Hello " + my_name + " , your number is "+ str(number))
```

- At this stage, we can save our script (**File | Save**).

We now need to compile this script:

- Please open the **Command Prompt** (or the **Terminal** if you are using a Mac computer).
- Type the command Python followed by a space.

```
python
```

- Locate the file that you have just created and drag and drop it atop the **Command Prompt**.

```
python /Users/patrick/my_first_script.py
```

- That should add the full path to this file just after the command that you have typed.
- You can now press return

This should compile and run the script, and the following message should be displayed in the Command Prompt

```
Hello Patrick , your number is 12
```

This is it! We have created our first script using some variables of type **integer** and **String**. The full script should look as described in the next code snippet.

```
number = 12
my_name = "Patrick"
print ("Hello " + my_name + " , your number is "+ str(number))
```

Creating your first function

So what is a function? A function is usually employed to perform a task outside the main body of the game. I usually compare functions to a friend or a colleague to whom you gently ask to perform a task for you. In many cases you will call them, and they will agree to perform the task. Sometimes they will need additional information to perform the task (e.g., a number to be able to call someone on your behalf); some other times, they will call you back to give you the information that they found, but in other cases, this may not be necessary, and they will perform the task without contacting you afterwards.

So there are essentially three types of functions:

- Functions that just perform actions with no parameters.
- Functions that perform actions with parameters.
- Functions that perform actions (with or without a parameter) and return a result.

Declaring a function

In Python a function declaration usually includes the parameters passed to this function. You can also specify the type of the parameters, and the type of data returned by the function, but this is optional.

The syntax to declare a method is as follows:

- The keyword **def**.
- The name of the function.
- Opening round brackets.
- The type of the parameter and their names.
- Closing round brackets.

- Colons.
- Any action (i.e., statement) performed by this function will be added after but indented from the function definition.

In the next sections, we will see examples of how functions can be declared.

Functions that don’t return or take any parameter

In this case, the function is called with no parameters; it will then perform an action. This is the simplest form of function. The syntax sequence is as follows: the keyword **def**, followed by the **name of the function**, followed by **opening and closing round brackets**, followed by **colons**. Any action (i.e., statement) performed by this function will be added after the line after the colons and indented, as illustrated in the next code snippet.

```
def my_first_function():
print(“Hello World”)
```

The indentation that you use for the code within a function (i.e., the number of spaces), or within any block of instructions is up to you; however, it has to be consistent within the function (or block of instructions). Using four spaces is a common practice.

When called, this function will print the message **“Hello World”** to the **Command Prompt**.

At this stage we have just defined the function **my_first_function**; in other words, we have specified what the function should do when it has been called. So once the function has been defined, we can call it using the syntax: **name_of_the_function()** for example, to call **my_first_function** we could write the following statement at the end of the **script**:

```
my_first_function()
```

So that this message stands out in the **Command Prompt**, we can comment all other **print** statements inside the script so that the code of your script looks like this (the changes are highlighted in bold):

```
number = 12
my_name = "Patrick"
#print ("Hello " + my_name + " , your number is "+ str(number))
def my_first_function():
print("Hello World")
my_first_function()
```

Note that the function needs to be declared before it is called. So the location of the function in the script (i.e., at the end or the start) does matter,

Now that you have written your first function, please do the following:

- Check that your code is written properly (i.e., error-free).
- Save your code (**CTRL + *S***).
- Compile and execute your code using the **Command Prompt**.

```
python /Users/patrick/my_first_script.py
```

- You should see the following message in the **Command Prompt**:

```
Hello World
```

Defining a function that takes parameters

So far, we looked at functions that would not take or return any parameters. For now, we will create a function that still doesn’t return any data, but that takes one or several parameters to perform calculations.

So to borrow the previous example, you call someone, give them some information, and ask them to perform an action based on your instructions.

To illustrate this concept, let’s create a new method that will display a message based on a parameter passed as an argument.

- Please type the following code just after the first function.

```
def my_second_function(name):
print("Hello, your name is " + name)
```

- In the previous code, we have created a function called **my_second_function**. It takes a parameter called **name**. So when we call this function and include a variable within the brackets, this variable will be referred to as **name** within this method.
- Let me illustrate with the following code.

```
my_second_function(“Patrick”);
```

If we were to type the previous code at the end of the script , the function **my_second_function** would set the variable **name**, that is used in the function **my_second_function**, with the string **Patrick**, and then display the message “**Hello, your name is Patrick”**. The variable **name** is a local variable to the function **my_second_function**.

If you have not already done so, please add the following code at the end of the script. You can replace the word **Patrick** with your own name.

```
my_second_function(“Patrick”);
```

- Save and compile your code.
- You should see, amongst other messages, the message **“Hello, your name is Patrick”** in the **Command prompt**.
- You could now change the call to this method and pass your own name as a parameter and see the result as you play the scene.

Note that we could have created a function that takes many other parameters. For example, we could have created a function that takes the first and last name as parameters, as follows.

```
def my_third_function (f_name, l_name):
print("Hello, your name is "+f_name+ " " + l_name)
```

Defining a method that takes parameters and returns information

So far, we know how to declare a function that takes parameters; however, we have not seen yet how we could define a function that also returns information.

This type of function will, in addition to possibly taking parameters and processing this information, return information to where it was called.

In the following example, we will create a function that does all three: it will be called; it will then take the **year of birth** as a parameter, and then calculate and return the corresponding **age** (based on the current year).

- Please add the following code at the end of the script:

```
def calculate_age(YOB):
age = 2021-YOB
return (age)
```

In the previous code:

- The function called **calculate_age** is declared.
- The function called **calculate_age** takes a parameter called **YOB** (short for Year Of Birth).
- The function **calculate_age** then subtracts **YOB** from the current year and returns the result.

Please add the following code at the end of the script.

```
my_age = calculate_age(1998)
print ("Your age is " + str(my_age))
```

In the previous code:

- The function **calculate_age** is called once; it returns the calculated age, and this value is returned and saved in the variable called **my_age**.
- This variable **my_age** is then printed in the **Command Prompt**.

Now that you have written the code for your new function, please do the following:

- Save your code.
- Compile and run the code.
- The **Command Prompt** should display, amongst other things, the message **“Your age is 23”**.

As you can see, there are different types of functions that you can create, depending on your needs. They may or may not take parameters, and they may or may not return values.

Note that we could have specified the type of the data to be returned by the function by modifying its definition as follows (new code in bold):

```
def calculate_age(YOB)->int:
age = 2021-YOB
return (age)
```

- We could have specified the type of the argument passed to the function:

```
def calculate_age(YOB:int)->int:
```

```
age = 2021-YOB
return (age)
```

- We could also have specified a default value for the parameter YOB as follows (new code in bold):

```
def calculate_age(YOB:int = 2000)->int:
age = 2021-YOB
return (age)
```

Creating your own class

To finish this chapter, it would be great to see how you could create and use your own class, and we will simply create and use a class for a virtual bike.

- Please add the following code at the end of the file **my_first_script.py**.

```
class Bike:
```

- In the previous code, we declare the name of the class.

We will now define a constructor for our class, along with member variables, to define the feature of each new bike created.

- Please add the following code within the class, after the previous code (new code in bold):

```
class Bike:
def __init__(self, new_name:str = "A New Bike", new_color:str =
"Blue",new_speed:int = 0):
self.name = new_name
self.color = new_color
self.speed = new_speed
```

- In the previous code, we create a constructor that takes three parameters which are used to initialize the class's member variables **color**, **speed** and **name**. Default values are also defined for these parameters (i.e., "**A New Bike**", "**Blue**" and **0**).

Next, we will define a few member methods for the class **Bike**; so please add the following methods to the class **Bike**:

```
def display_name(self):
print("Name: "+self.name)
def display_color(self):
print("Color is: " + self.color)
def accelerate(self):
self.speed += 1
print("New speed: "+str(self.speed))
```

- In the previous code, we define three functions named **display_name**, **display_color** and **accelerate**.
- The first function displays the name of the bike.
- The second function displays the color of the bike.
- The third function increases the bike’s speed by one and displays it.

We can now add some code to instantiate the class that we have just defined:

- Please add the following code at the end of the file:

```
my_bike = Bike();
my_bike.display_color()
my_bike.display_name()
my_bike.accelerate()
```

In the previous code:

- We declare a variable called **my_bike**.
- You may notice that we use the constructor without passing any

parameter, so, as a result the default values for this bike’s member variables will be used: A **blue** color, a speed of **0**, and the name “**A New Bike**”.

- We then call the member functions **display_color**, **display_name**, and **accelerate**.

We can now test our code: please save and compile your code, and you should see the following messages in the **Command Prompt**.

```
Color is: Blue
Name: A New Bike
New speed: 1
```

Now that we know that the code works, we can add more code to test the constructors.

- Add this code to the at the end of the script.

```
my_bike2:Bike = Bike("My Bike","Red",10)
my_bike2.display_color()
my_bike2.display_name()
my_bike2.accelerate()
```

In the previous code:

- We create a new **Bike**.
- This time we pass parameters to the constructor so that our new bike’s member variables **name**, **color** and **speed** are respectively “**My Bike**”, “**Red**”, and **10**.
- We then display the value of the variables **color** and **name**.
- Finally, we call the method **accelerate** which increases the speed by **1**.

We can now test our code: please save and compile your code, and you should see the following messages in the **Command Prompt**.

```
Color is: Red
Name: My Bike
New speed: 11
```

Using Modules

The last topic we will cover in this chapter is modules; modules are a very handy way to store and re-use code for your Python programmes.

So let’s say that you have a main programme for your games, that includes several classes that you will be using for your game; however, for clarity, you’d prefer to write the core of the game in the main script while keeping the class definitions somewhere else. In that case modules would be very handy.

Let’s illustrate with an example.

- Please create a new file (**File | New**) and save it as **my_modules.py** in the same folder as the script that you have previously created and called **my_first_script.py**.
- Once this is done, please add the following code to the script **my_modules.py**.

```
class Wallet:
def __init__(self):
self.amount = 100
self.currency = "Euro"
def add_money(self, new_amount:int):
self.amout += new_amount
def display(self):
print("Amount in Wallet: "
+ str (self.amount)
+ " "
+ str(self.currency))
```

In the previous code:

- We define a class called **Wallet**.
- The constructor initializes the member variables **amount** and **currency**.
- We also define two member methods called **add** and **display** that respectively add money to the wallet and display its content.

Once this is done, we can use the module that we have defined as well as the class defined within:

- Please safe the file **my_modules.py** (**File | Save**).
- Switch to the file **my_first_script.py**.
- Add this code at the beginning of the script.

```
import my_modules
```

- Add this code at the end of the script

```
wallet = my_modules.Wallet()
wallet.display()
```

In the previous code, we create a new instance of the class **Wallet**, that is defined in the module called **my_modules**, and we then call the method **add** from the instance of the class **Wallet**.

Best practices

To ensure that your code is easy to understand and that it does not generate countless headaches when trying to modify it, there are a few good practices that you can start applying as you begin with coding; these should save you some time along the line.

Indentation

- Make sure you indent your code properly and consistently, especially within structures and content blocks such as classes, methods, loops and conditional statements.
- When defining a function with several parameters, you can break down the line into several lines as follows.

```
def __init__(
    self,
    new_name:str="A New Bike",
    new_color:str="Blue",
    new_speed:int=0):
```

Maximum line length

It is usually good practice to keep the length of your lines to a maximum of around 79 characters, as much as possible.

Line breaks

As mentioned earlier, it is good practice to break a single line into several lines. In the case of formula using operators, it is usually good practice to break the line before the operator, as illustrated in the following code snippet.

```
    my_result = (data1_from_the_db
    + data2_from_the_db
    + data3_from_the_db
    + data4_from_the_db
    + data5_from_the_db)
```

Variable naming

- Use meaningful names that you can understand, especially after leaving your code for two weeks.

```
my_name:String = “Patrick” #GOOD
    b = “Patrick” #NOT SO GOOD
```

- Use Pascal and **Snake** casing consistently.

```
    test_if_the_name_is_correct #GOOD
    testifthenameiscorrect #NOT SO GOOD
```

Methods

- Check that all opening brackets have a corresponding closing bracket.
- Indent your code when it is part of a function or a block of instructions.
- Comment your code as much as possible to explain how it works.

Level roundup

Summary

In this chapter, we became familiar with different programming concepts. We also looked into classes, constructors, and member variables. Finally, we created our first class and experimented with creating instances of classes and displaying their properties. In the next chapter, we will harness these skills to create our very first games with Python.

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available on the next page.

1. Each class has a default constructor.
2. A constructor can include several parameters.
3. A member variable can be accessed from anywhere in your class.
4. When a new instance of an object is created, the corresponding constructor is called.
5. A Python file can be compiled and run from the **Command Prompt**.
6. A simple text editor can be se to create and edit **Python** scripts.
7. The keyword **class** makes it possible to define the name of a class.
8. In **Snake casing** the first character of each word that makes up the name of a variable is capitalized except for the first word.
9. In **Pascal casing** the first character of each word that makes up the name of a variable is capitalized.
10. Functions are declared using the keyword **def**.

Answers to the Quiz

1. TRUE.
2. TRUE
3. TRUE.
4. TRUE.
5. TRUE.
6. TRUE.
7. TRUE.
8. FALSE
9. TRUE
10. TRUE.

Checklist

If you can do the following, then you are ready to go to the next chapter:

- Create a new Python script.
- Compile a script.
- Create a class.
- Create member variables.
- Create member methods.
- Call a constructor.
- Know how to comment your code in a script.
- Answer at least 7 out of 10 of the questions correctly in the quiz.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let’s put these to the test.

- Modify your code to create two additional member methods.
- Create instances from a class and call member methods.

CHAPTER 3: CREATING A NUMBER GUESSING GAME

In this section we will discover how scripting can be used to add interaction and create a simple number guessing game to the game and to provide more control over the game mechanics.

In this game, the user will need to guess a random number; every time s/he enters an answer, the system will provide a clue as to whether the number entered is greater or lesser than the number to guess. The number of attempts will be counted, and a congratulation message will be displayed when the user has finally found the correct number.

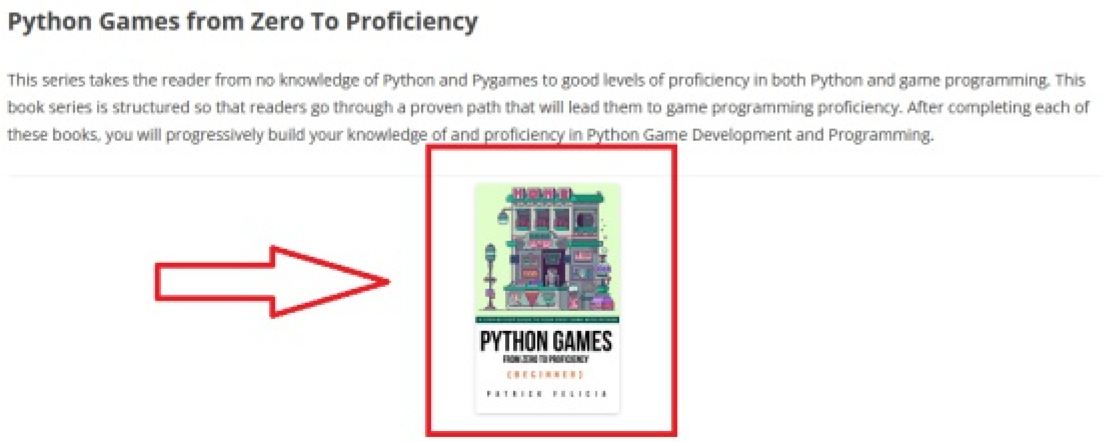
After completing this chapter, you will be able to:

- Use loops
- Generate random numbers.
- Wait for and process user's key inputs.
- Use conditional statements to check the user's answers.
- Use a simple scoring system.

Resources necessary for this chapter

To complete the activities presented in this book you need to download the startup pack on the companion website; it consists of free resources that you will need to complete your projects. To download these resources, please do the following:

- Open the page **http://www.learntocreategames.com/books**.
- Click on your book (**Python From Zero to Proficiency (Beginner)**)
- On the new page, please click the link that says "**Please Click Here to Download Your Resource Pack**"


Python Games from Zero To Proficiency

This series takes the reader from no knowledge of Python and Pygames to good levels of proficiency in both Python and game programming. This book series is structured so that readers go through a proven path that will lead them to game programming proficiency. After completing each of these books, you will progressively build your knowledge of and proficiency in Python Game Development and Programming.

Creating a simple script to guess a number

Let’s get started with our game:

- Please switch to your text editor (e.g., **Sublime**).
- Create a new file **(File | New)** and save it as **guess_numbers.py** (**File | Save As...**).
- Please enter the following code in the script:

```
import random
number_to_guess = 0;
nb_attempts = 1
min_value = 0
max_value = 100
```

In the prevous code:

- We import a library called random, a random library that can be used to generate random numbers
- We then declare and initialize four variable called **number_to_guess**, **nb_attempts**, **min_value** and **max_value**. Because these variables are declared outside of any function, they are global variables in the sense that they can be accessed from anywhere within this script.
- **number_of_guess** will be used to store the value of the number to guess.
- **nb_attempts** will be used to track the number of attempts by the user (this information will be displayed at the end of the game)
- **min_value** and **max_value** will be used to define the range for the random number; this will be defined by the user.

So at this stage we have defined the key variables for our game; what we need to do now is to define a function that will initialize the game (e.g., to define the range of the random number, as well as the number to guess) along with a function that will be used to require, store and evaluate the successive answers from the user.

Please add the following code to the script (at the end of the script):

```
def init_game():
global number_to_guess
global min_value
global max_value
global nb_attempts
```

In the previous code:

- We declare a function called **init_game** that will be used to initialize the game.
- We use the keyword global to specify that the variables that we are declaring refer to the global variables defined earlier; in other word any change to these variables within the function will be mirrored to the global variables. Or in other words, using these variables within the function is the same as using the global variables.

Please add the following code to the script:

```
min_value = input("Please enter the minimum value: ")
max_value = input("Please enter the maximum value: ")
min_value = int(min_value)
max_value = int(max_value)
```

In the previous code:

- We ask the user to enter the lower range for the random variable and we store this value in the variable **min_value**. At this stage, the value entered by the user is a string.
- We ask the user to enter the upper range for the random variable and we store this value in the variable **max_value**. At this stage, the value entered by the user is a string.
- We then convert both the variables **min_value** and **max_value** to an **integer** format.

Finally, we can define the number to guess by adding the following code to the function:

```
number_to_guess = random.randint(min_value,max_value)
nb_attemps = 0
```

In the previous code we generate a random number based on the lower and upper boundaries defined by the user and we set the number of attempts to 0.

At this stage, the full function **init_game** should look like the following code snippet:

```
def init_game():
global number_to_guess
global min_value
global max_value
global nb_attempts
min_value = input("Please enter the minimum value: ")
max_value = input("Please enter the maximum value: ")
min_value = int(min_value)
max_value = int(max_value)
```

```
number_to_guess = random.randint(min_value,max_value)
nb_attemps = 0
```

So at this stage, we have managed to complete the function **init_game** that basically defines the number to guess.

The next stage is to create a function that will ask and store the answers from the user; this function will do the following:

- Ask for a new number from the user.
- Save this number.
- Increase the number of attempts.
- Display a message that specifies whether the number entered is lower or greater than the number to guess.
- If the number was found, then a winning message will be displayed, and the game will restart with a new number to guess.

So, with tis being aid, lets code this function:

Please add the following code at the end of the script:

```
def ask_for_new_number():
global number_to_guess
global nb_attempts
global min_value
global max_value
```

In the previous code, we define the function called **ask_for_new_number**, and we then define (or refer to) global variables as we have done in the previous function.

Please add the following code to the function:

```
new_number = input("Please enter your number ["
+str(min_value)
+"-"
+str(max_value)
+"]: ")
new_number = int(new_number)
nb_attempts +=1
```

In the previous code:

- We ask the user to enter a number that is comprised within the boundaries defined earlier.
- We convert the user entry to an integer.
- We then increase the number of attempts.

So at this stage, the user has entered a number, and we need to evaluate (and provide feedback on) whether this number is lesser or greater than the number to guess.

Please add the following code to the function **ask_for_new_number**:

```
if (number_to_guess > new_number):
print("> My Number is greater than " + str(new_number))
elif (number_to_guess < new_number):
print("> My Number is lesser than " + str(new_number))
```

In the previous code:

- We check whether the number entered by the user is lesser or greater than the number to guess.
- If the number to guess is greater than the number entered by the user, we display the message (“My number is greater than ..”).
- If the number to guess is lesser than the number entered by the user, we

display the message (“My number is lesser than ..”).

Once we have displayed the corresponding message, we just need to check whether the user has found the number to guess.

Please add the following code to the function **ask_for_new_number**:

```
else:
print ("\n *** Well done; you have guessed my number; it took "
+ str(nb_attempts)
+ " attempts")
print ("\n\n *** New Game***")
init_game()
game_loop()
```

In the previous code:

- We are in the case where the number is neither lesser nor greater than the number to guess, that is, when it is equal to the number to guess.
- In that case, we display a congratulation message that includes the number of attempts
- We then display the message “**New Game**”
- We initialize the game so that a new random number to guess can be generated
- Finally, we call a function called **game_loop** that we yet must define; this function will basically constantly loop to ask the user for a new entry until the number to guess has been found.

So please add the following code just before the function **init_game**:

```
def game_loop():
while (True):
```

```
ask_for_new_number()
```

In the previous code:

- We declare and define a function called **game_loop**.
- Within the function, we declare a loop that will loop indefinitely (the condition is always **True**).
- Within this loop, we call the function **ask_for_new_number**.

That's it.

- You can save and compile your code
- As you do, the following prompts will be displayed:

```
Please enter the minimum value: 1
Please enter the maximum value: 10
Please enter your number [1-10]: 5
> My Number is greater than 5
Please enter your number [1-10]: 6
```

- And if you find the right number, the following messages will be displayed:

```
*** Well done; you have guessed my number; it took 3 attempts


*** New Game***
```

Level roundup

In this chapter, we have learned about creating a script using Python. We also became more comfortable with functions, variables, and their properties. We managed to create and use scripts to detect the user input, and we combined functions, loops, and conditional statements. So yes, we have made some considerable progress, and we have by now looked at several programming structures as well as common errors that you may come across on your coding journey.

Checklist

You can consider moving to the next chapter if you can do the following:

- Declare variables.
- Use both local and global variables.
- Create functions.
- Call a function.
- Use loops and conditional statements.
- Detect and save the user’s input.
- Display messages in the **Command Prompt**.

Quiz

It’s now time to check your knowledge with a quiz. Please answer the following questions. The solutions are on the next page. Good luck!

Please specify whether the following statements are **TRUE** or **FALSE**

1. The keyword **import** makes it possible to import and use a module.

2. The following code will loop indefinitely.

```
while (True):
ask_for_new_number()
```

1. The following code will save the input from the user in the variable **min_value**:

```
min_value = input("Please enter the minimum value: ")
```

1. The following code will create a random number between 1 and 100.

```
number_to_guess = random.randint(1,100)
```

1. The following code will display **True**:

```
if (1 > 0):
print("True”)
```

1. The following code will display **True**:

```
if (0 > 1):
print("True”)
```

1. The following code will display “**My Name is Patrick**”

```
my_name = “Patrick”
def ask_for_new_number():
global my_name
print(“My Name is ” + str (my_name))
```

1. The following code will display **FALSE:**

```
If (3 > 1):
print("False”))
```

1. A global variable can be used anywhere within a script

2. A local variable declared in a function can only be used in that function.

1.

Answers to the quiz

1. **TRUE**.
2. **TRUE**.
3. **TRUE**.
4. **TRUE**
5. **TRUE**.
6. **FALSE**.
7. **FALSE**.
8. **TRUE**.
9. **TRUE**.
10. **TRUE**.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let’s put these to the test.

- Modify the messages displayed to the user to say whether its “cold” or “hot” based on the difference between the number to guess and the numbered entered. For example, if the difference is 1 then the message “hot” will be displayed, otherwise, the system will say “cold.”
- Modify the function **ask_for_new_number** so that the system keeps asking for a new number if the number is outside the range defined earlier.

Challenge 2

For this challenge you could modify the code so that.

- The system asks for the make of the player at the start
- There is a limited number of attempts.
- If the player used more than the authorized number of attempts, a **Game Over** message is displayed and the game restarts.
- All messages are personalized by using the name of the user.

CHAPTER 4: CREATING A WORD GUESSING GAME

In this section we will discover how to create a word guessing game. Some of the objectives of this section will be to:

- Use loops and conditional statements.
- Use the **not** keyword.
- Use global variables
- Detect and record the user’s input.
- Create and use arrays.
- Randomly select an element of an array.
- Read text files and extract data from it.
- Detect if a letter is part of a word.
- Replace letters in a string

After completing this chapter, you will be able to:

- Display messages in the Console Prompt.
- Check if a letter is part of a specific string.
- Read a file to extract words within randomly.
- Create arrays to store and select words randomly.

-

In this section, we will create a game with the following gameplay:

- The player is offered to guess a random word.
- The player must guess this word within a specific number of attempts.
- All letters of the words are hidden.
- The player enters a letter.
- The game checks whether the letters are included in the word.
- If this is the case, the letter is displayed.
- The game carries on until the player finds all the letters within the given number of attempts or when the player has reached the maximum number of attempts.
- The game then starts over.

Initializing the game

In this chapter we will start by initializing the game and its variable so that the word to guess is chosen either from an array or from a file.

- Please open Sublime, if it’s not already launched.
- Create a new File: **File | New**.
- Save this file as **guess_word.py**.
- Add this code at the beginning of the script.

```
import numpy as np
import random
```

In the previous code we import two libraries: **numpy** is used to declare and use arrays in Python, while random is used (as we have done before) to generate random numbers.

- Please add the following code, just after the code that you have just entered.

```
word_to_guess = "";
string_to_display=""
length_of_word_to_guess = 0;
letters_to_guess = ""
letters_guessed=""
new_letter = ''
stay_in_loop = 1
nb_attempts = 1
max_nb_attemps = 5
```

In the previous code:

- We declare several variables.
- **word_to_guess**: will store the word to be guessed by the player.
- **string_to_display**: will be used to display the word to be guessed; it will first consist of question marked and these will progressively replace by the letters that the player has managed to guess.
- **length_of_word_to_guess**: will store the length of the word to be guessed.
- **letters_to_guess**: will store all the letters part of the word to be guessed.
- **letters_guessed**: will store whether a specific letter in the word to be guessed was guessed.
- **new_letter**: will store the letter that was just entered by the player.
- **stay_in_loop**: will be used to ensure that we stay in the main game loop.
- **nb_attemps**: will store the number of guessing attempts from the player.
- **max_nb_attempts**: will store the maximum of authorized attempts from the player before s/he loses.

Once we have initialized the key variables for our game, it is time to generate a random word to be guessed by the player.

- Please add the following code:

```
def init_game():
global word_to_guess
global string_to_display
global length_of_word_to_guess
global letters_to_guess
global letters_guessed
global new_letter
```

In the previous code we declare the function **init_game** and we then use the **global** keyword to make a reference to global variables that we will be using win the function.

- Please add the following code within that function:

```
words_to_guess = ["car","elephant","autocar"]
random_number = random.randint(0,len(words_to_guess)-1)
word_to_guess = words_to_guess[random_number]
```

In the previous code:

- We initialize the list **words_to_guess**; it now includes three words: **car**, **elephant** and **autocar**.
- We create a random number that will be between 0 and the size of the list -1, that is **2** (since we have 3 elements in the list).

Remember that to access an element in a list, we can use an index that will start at 0, and end at the size of the list -1; so in our case, the random number will be used to point to a specific element within this list.

- Finally, we select a word within this list, at the index determined by the random number that we have just generated, and this number is stored in the variable called **word_to_guess**.

Now that we have selected a random word to be guessed, we just need to initialize a few variables that will make it possible to know which letter has been guessed, and to also display them accordingly onscreen.

- Please add the following code to the function:

```
length_of_word_to_guess = len(word_to_guess)
word_to_guess = word_to_guess.upper ()
letters_to_guess = np.array(length_of_word_to_guess)
letters_guessed = np.array(length_of_word_to_guess)
letters_guessed = np.zeros(length_of_word_to_guess)
letters_to_guess = [char for char in word_to_guess]
string_to_display = ""
for i in range (length_of_word_to_guess):
string_to_display+="?"
```

In the previous code:

- We determine the length of the word to be guessed.
- We convert all the letters of the word to be guessed to upper case; this will make it easier to check whether the letter typed by the player (which will also be changed to upper-case) matches any of these.
- We then use the nuppy library (referred as **np**) to create and initialize two arrays: **letters_to_guess** and **letters_guessed**; the latter is filled with **0s**.
- The array **letters_to_guess** is composed of characters that are converted from the string variable **word_to_guess**; in other words, we go through the entire string, and include each of the letters into an array of characters called **letters_to_guess**.
- Finally initialize the variable **string_to_display**, and we loop so that every letter in the word to be guessed is replaced by a question marked when displayed to the player.

That’s it; so we have effectively created a function called **init_game** that defined the word to be guessed along with the string to be displayed to the user (i.e., initially question marks), and variables that will help us to know which letters have been guessed.

The next phase will be to: display the word to be, process the player’s answer.

.

Displaying the word to guess and processing the users' answer

At this stage, we have determined the word to guess and initialized the sting displayed to the player (i.e., initially question marks).

So, in this section, we will start to write code that displays the word to be guessed and also ask for and processes the player's answer.

- Please open the guess_word.py and add the following function to the script

```
def display_letters():
print(string_to_display)
```

In the previous code we declare a function called **display_letters** that prints the content of the string variable **string_to_display**, which will initially made of question marks, util the payer guesses one of the letters from the word to guess.

Now that the function has been created, we will be focusing on processing the players input when they enter a key and try to guess one of the letters that are part of the word to guess.

The next step will eb to create function that will gather the user's input, check if it the key pressed is one of the letters from the word to guess, and update the string displayed to the user (i.e., reveal the letters that were guessed correctly for the word to guess)

- Please add the following code to the script:

```
def check_keyboard():
```

```
global word_to_guess
global string_to_display
global length_of_word_to_guess
global letters_to_guess
global letters_guessed
global new_letter
global nb_attempts
global max_nb_attemps
```

In the previous code, we create the function called **check_keyboard**, and at the beginning of that function, we use the keyword **global** to refer to global variables that will be used in that function.

- Please add the following code to the function.

```
new_letter = str(raw_input("Attempt " +str(nb_attempts) +">"))
new_letter = new_letter.upper()
nb_attempts+=1
```

In the previous code:

- We record the number entered by the player; we use the built-in function **raw_input** as we will be dealing with characters and strings.
- We convert the letter entered to upper-case; this is because we have already converted the letters of the word to be guessed to upper-case, this, making the detection easier (comparing two letters of different case would be unfruitful, even if they are the same letters).
- We then increase the value of the variable **nb_attemps** by 1.

Next, we need to loop through the letters of the word to be guessed and detect whether the letter entered by the player is one of these letters.

- Please add the following code to the function:

```
for i in range (length_of_word_to_guess):
if (ord(new_letter) >=65 and ord(new_letter) <= 90):
if (not letters_guessed [i]):
```

In the previous code:

- We create a loop that will go from the first to the last letter of the word to be guessed.
- Within that loo, we check if the Unicode code of the letter entered by the player; since we are looking for an upper case letter and that the Unicode codes for the upper-case letters **A** and **Z** are respectively **65** and **90**, we then just need to make sure that the Unicode code of the letter entered by the player is comprised between **65** and **90**.
- Finally, we check that this letter, if it is part of the word to be guessed, has not already been guessed by the player.

Once this is done, we can start to update the string to be displayed to the player, along with other variables that track the progress of the player.

- Please add the following code to the function (new code in bold):

```
if (not letters_guessed [i]):
if (letters_to_guess [i] == new_letter):
letters_guessed [i] = 1
string_to_display = replace_char_at_index(string_to_display,new_letter,i)
```

else:

In the previous code:

- We check whether the letter entered by the player has already been guessed.
- We also check whether the letter entered by the plyer is part of the word to be guessed.
- In that case, we update the list l**etters_guessed** to specify that this letter has now been guessed.
- We also replace the corresponding question mark in the string displayed to the user by the actual letter; this is done by calling a function called **replace_char_at_index**, that we yet must define.

Once we have checked that the player has entered a correct letter, we also need to check what happens when an incorrect letter has been entered; in that case, depending on whether the maximum number of attempts has been reached the player will either be prompted to enter another letter, or the game will be over.

- Please add the following code to the function **check_keyboard** (new code in bold):

```
else:
if (nb_attempts >= max_nb_attemps):
print("Sorry Max Nb Attempts Reached")
nb_attempts = 0
init_game()
game_loop()
```

In the previous code we check whether the player has reached the maximum number of attempts; we also reset the value of the variable **nb_attempts** and successively call the function **init_game** and **game_loop**; we will create the latter in the next section.

So now that we have created the function that processes the entries from the player, we just need to create two additional functions: **game_loop** and **replace_char_at_index**.

- First, please add the following function to the script:

```
def game_loop():
while (stay_in_loop):
display_letters()
check_keyboard2()
```

In the previous code, we declare a function called **game_loop** where we loop indefinitely (since **stay_in_loop** is **True**) and where we successively display the letters guessed by the player and check the user input.

Next, lets create the function **replace_char_at_index**; if you remember well, this function is used to replace the question marks with the letters guessed by the player for the string displayed to the player after each attempt.

- Please add the following function to the script:

```
def replace_char_at_index(string_to_work_with, new_character, index):
string_to_be_returned = ""
string_to_be_returned =
string_to_work_with[:index]+new_character+string_to_work_with[index+1:]
return (string_to_be_returned)
```

In the previous code:

- We define the function **replace_char_at_index**.
- This function takes three parameters: **string_to_work_with**, **new_character** and **index**.
- **string_to_work_with[:index]** is part of the string **string_to_work_with**, including all its letters, up to the index.
- **string_to_work_with[index+1:]** is part of the string **string_to_work_with**, including all its letters, from the index onwards
- So effectively, the code above splits the original string into two parts: the letters before and after the index.
- We then concatenate the string that was before the index, with the new character and the string after the index.

Last but not least, we just need to add the following lines to the script:

```
init_game()
game_loop()
```

In the previous code we call both the functions **init_game** and **game_loop**.

You can now compile your code and you should see the following

```
???????
Attempt 1>
```

As you go through successive attempts, you should see that the corresponding question marks in the word to be guessed are replaced by the letters that you have guessed.

```
Attempt 1>e
???????
```

```
Attempt 2>C
????C??
Attempt 3>A
A???CA?
Attempt 4>
```

gathering random words from a text file

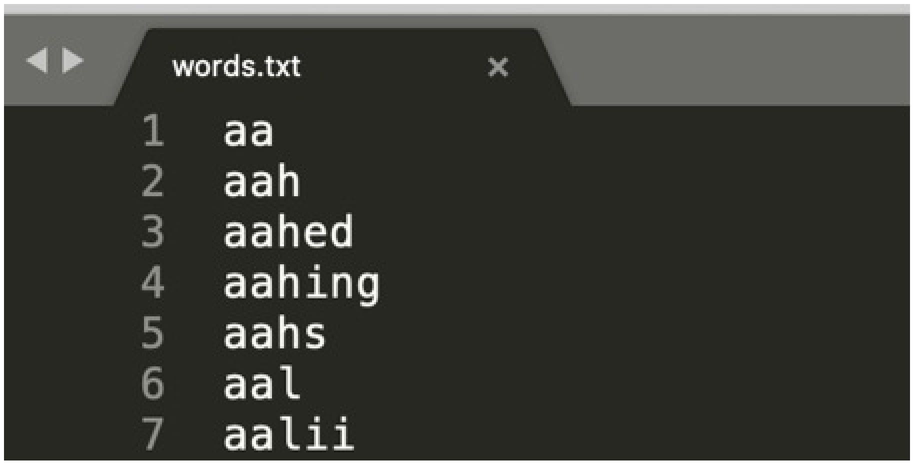
So at this stage, we have managed to create a game whereby the player has to guess a word that is part of a list that we have defined; this being said, this also means that you will need to enter every single word manually in the list if you wanted to expand the number of games to be guessed; so what if it would be possible to have a file with all the words available in a dictionary and to be able to choose one of them at random, without having to enter each of them manually.

Well this is what we will do in this section; In the next section we will do the following:

- Import a text file that includes all (or most) of the words present in the English dictionary.
- Pick a word at random within this file.
- Ask the player to guess that game.

So let’s get started:

- Please identify the file called words.txt in the resource pack and copy it to the folder where you have saved the Python file **guess_word.py**.
- You can even open this file briefly, if you wish, to see its content, and you will see, a list of over 79000 words organized alphabetically from **aa** to **zyzzyvas**.



```
aa
aah
aahed
aahing
aahs
aal
aalii
```

Once this is done, it is time to write the code to import and use this file:

- Please add the following code to the file **guess_word.py**.

```
def set_word_from_file():
global word_to_guess
global string_to_display
global length_of_word_to_guess
global letters_to_guess
global letters_guessed
global new_letter
```

In the previous code we define a new function called **set_word_from_file** and we then refer to global variables that will be used in this function by using the keyword **global**.

Now that we have defined the function and the global variables that will be used within, we can start to read the word file and extract words from it.

- Please add the following code to the function **set_word_from_file**:

```
with open('words.txt') as f:
lines = f.readlines()
```

```
random_number = random.randint(0,len(lines)-1)
word_to_guess = lines[random_number]
word_to_guess = word_to_guess.replace("\n","")
word_to_guess = word_to_guess.replace("\r","")
```

In the previous code:

- We open the file called wors.txt (that we assume is in the same folder as the current script).
- We then read every line and store them in a list called lines.
- Since each line stores a single word, each item in this list includes a word.
- Once this is done, we create a random number that will be used to select a word at random within this file.
- Using this random number, we store the random word to be guessed in the variable **word_to_guess**.
- Finally, because each line includes the characters "**\n**" and "**\r**" at the end each line, we make sure that these are removed using the built-in function replace.

Now that we have created this function, we just need to replace the previous code by a call to this function:

- Please create the following function:

```
def set_word_from_array():
global word_to_guess
global string_to_display
global length_of_word_to_guess
global letters_to_guess
global letters_guessed
global new_letter
words_to_guess = ["car","elephant","autocar"]
#print(words_to_guess[1])
random_number = random.randint(0,len(words_to_guess)-1)
word_to_guess = words_to_guess[random_number]
```

The previous code is essentially a copy paste of the code that we had already created to generate an array of words to be guessed; the only difference is that now this code is part of a function, so it can be called, if need be.

- Please locate he function **init_game**.
- Replace the following code:

```
words_to_guess = ["car","elephant","autocar"]
print(words_to_guess[1])
random_number = random.randint(0,len(words_to_guess)-1)
word_to_guess = words_to_guess[random_number]
```

- with this code:

```
set_word_from_file()
```

You can now save and compile your code and check that a new word has been used from the list.

Level roundup

In this chapter, we have further improved our skills to learn about how to read data from arrays and files. We became more comfortable with the creation of functions, manipulating strings (e.g., replacing characters), and manipulating lists to keep track of the users’ progress. We managed to create a game that displays messages, checks users’ inputs, modifies strings, and uses loops and conditional statements to check the uses. So, again, we have made considerable progress since the last chapter. Well done!

Checklist

You can consider moving to the next stage if you can do the following:

- Create a list.
- Read a text file.
- Replace characters in a string.
- Ask for and save the user’s input.
- Use loops and conditional statements.

Quiz

Please specify whether the following statements are **TRUE** or **FALSE**.

1. A function is declared with the keyword **def**.
2. A variable declared outside of any function in a script is global.
3. A global variable can be referred to in a function using the keyword global.
4. A loop can be defined using the keyword **while**.
5. The built-in function **random.randint** can be used to generate a random integer.
6. The built-in function **len** can be used to obtain the length of a string.
7. The built-in function **input** can be used to gather the input from the user.
8. Given that the file **words.txt** was saved in the same folder as the current script, the following code will read the file and store the content it in the variable lines.

```
with open('words.txt') as f:
lines = f.readlines()
```

1. The following code will replace the character “t” with the character “r”.

```
word_to_guess = word_to_guess.replace("t","r")
```

1. The keyword import can be used to import modules

Answers to the quiz

1. TRUE
2. TRUE
3. TRUE
4. FALSE
5. TRUE
6. TRUE
7. TRUE
8. TRUE
9. TRUE
10. TRUE

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let’s put these to the test.

- Create a new text file of your choice.
- Save this file.
- Use this file to generate random words for the game.

CHAPTER 5: USING GRAPHICS AND THE PYGAME LIBRARY

In this section, we will start to introduce the use of graphics with Python by drawing objects onscreen and detecting mouse interaction from the user.

After completing this chapter, you will be able to:

- Draw shapes onscreen.
- Modify the color and size of the different lines used in the graphics.
- Detect the position of the muse.
- Detect when the user has moved the mouse or clicked on a specific part of the screen.

Installing Pygame

Pygame is a Python library that make it easier to create games as it includes modules that make it possible to implement any features that are commonly found in games, including: using animated graphics, detecting user inputs (e.g., keyboard, joystick or mouse), managing sprites and 2D Graphics or collision detection.

To install **Pygame**, please follow the instructions included on the official page:

https://www.pygame.org/wiki/GettingStarted

Drawing basic shapes onscreen

Once Pygame has been installed, it is time to use this library for our new game.

So, in this section, we will start to get more familiar with the Pygame module and to draw some basic shapes onscreen.

- Please launch **Sublime**.
- Create a new file and save it as **graphics.py**.
- Add the following code to the file:

```
import pygame
pygame.display.init()
scr = pygame.display.set_mode((600,500))
pygame.display.set_caption('Pygame Window')
```

In the previous code:

- We import the Pygame library that we have just installed
- We then initialize the display module so that we can start to draw on it.
- We create a display surface in which we can draw and specify its width and height.
- Finally we set a caption for the graphics window that we have just defined.

So at this stage, weave initialized both the display module and surface, and we are ready to draw an object on screen.

So, to start with, we will draw a rectangle; for this purpose, we will define a color to be used for the drawing, along with the coordinates of the topic left corner of the rectangle, along with its width and height.

- Please add the following code to the script:

```
color = (0,0,255)
start_x = 10
start_y = 10
rect_width = 100
rect_height = 100
```

In the previous code:

- We specify the color to be used for the drawing; for this purpose, we use the **RGB** code; each of the three parameters within the round bracket ranges from 0 to 255 and specifies the amount of Red, Green and Blue, respectively, that the final color should contain.
- We then specify the coordinates of the top left corner of the rectangle.
- Finally, we specify the width and height of the rectangle.

Now that we have specified the features of this rectangle, we can draw it on the surface that we have defined earlier.

- Please add the following code to the script:

```
pygame.draw.rect(scr, color, pygame.Rect(start_x,start_y, rect_width, rect_height))
pygame.display.flip()
```

In the previous code:

- We use the built-in function **pygame.draw.rect** to draw a rectangle.
- The first parameter, **src**, is the display surface that we have defined

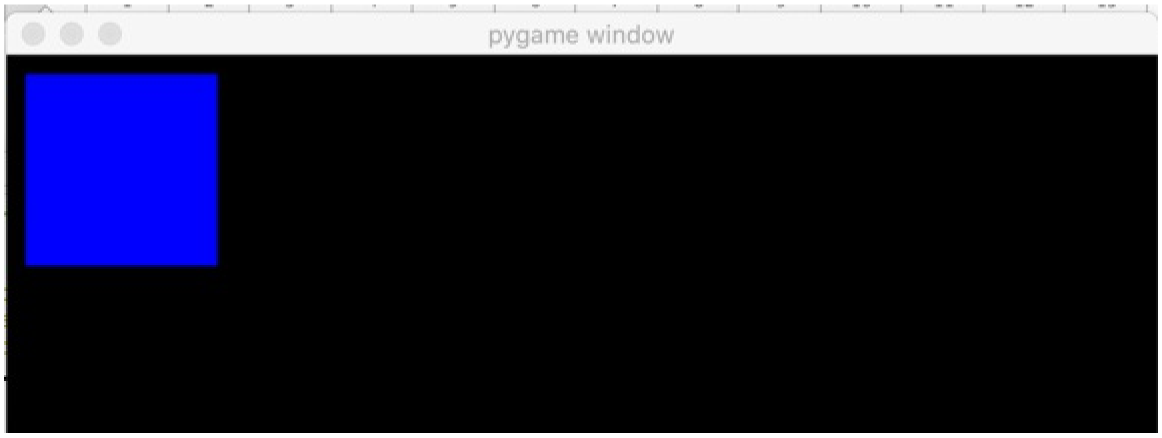
earlier.

- The second parameter, **color**, is the color that we have defined earlier.
- The last parameter uses the built-in function **pygame.Rect** to draw a rectangle based on the variables that we have defined earlier (i.e., the coordinates of the top left corner, width and height)
- The built-in function flip updates the drawing surface so that the changes that we have made (i.e., drawing the rectangle) are applied and visible to the user.

Finally, so that the rectangle is displayed indefinitely, you can add the next lines, that effectively loop indefinitely.

```
while (True):
pass
```

That’s it, you can compile your code; and after you do, the following screen should be displayed.



Once you want to quit the display, just switch to the Command Prompt and press **CTRL + C** simultaneously.

Next, we will draw a circle using the exact same principle, but this time with a different color.

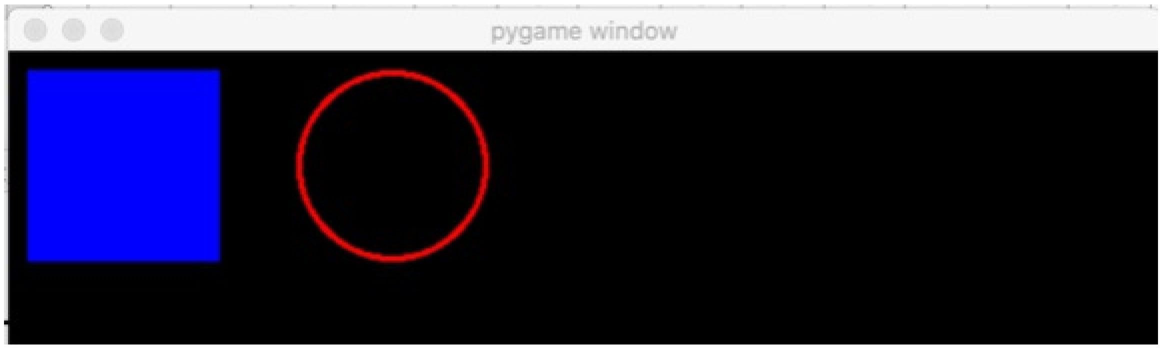
- Please add the following code to the script (new code in bold):

```
pygame.draw.rect(scr, color, pygame.Rect(start_x,start_y, rect_width, rect_height))
color = (255,0,0)
circle_center = (200, 60)
circle_radius = 50
pygame.draw.circle(scr, color, circle_center, circle_radius, 3)
pygame.display.flip()
```

In the previous code:

- We specify a new color for our circle: **red**.
- We also specify the coordinates of center of the circle along with its radius.
- Finally, we draw the circle using the built-in function **pygame.draw.circle**.
- The first parameter, **src**, refers to the display surface that we have defined earlier.
- The second parameter, **color**, refers to the color that we have just defined.
- We also specify, in addition to the center and radius of the circle, the thickness of the line to be used when drawing the circle (i.e., 3 pixels).

That's it, you can now compile your file and the display window should look like the next figure.



The last thing we will do is to draw a cross; this will eb useful as we will be drawing crosses in our next game (Tic Tac Toe).

The cross will be made of two lines each connecting the opposite corners of an imaginary rectangle: a line from the top left corner to the bottom right corner of the rectangle, and another line from the bottom left corner to top right corner of the rectangle.

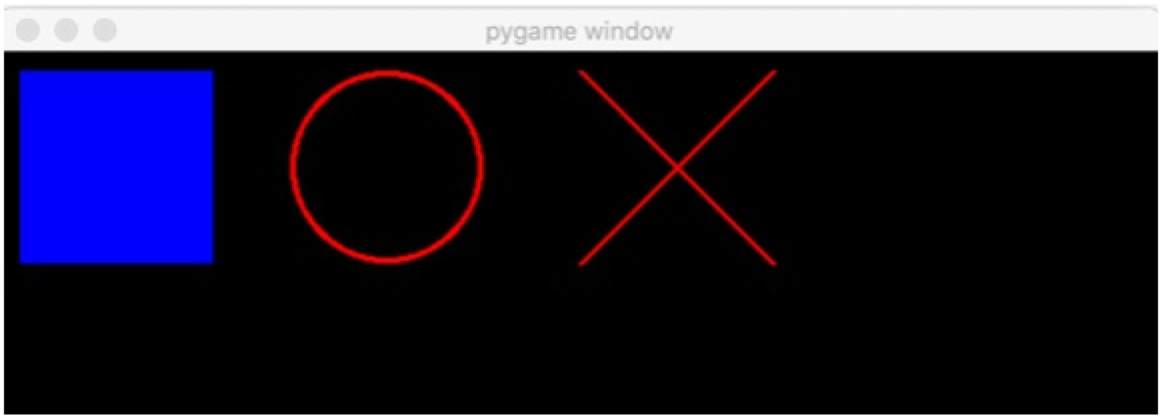
- Please add the following code:

```
cross_start_x = 300
cross_start_y = 10
pygame.draw.line(scr, color,(cross_start_x, cross_start_y), (cross_start_x + 100,
cross_start_y + 100),3)
pygame.draw.line(scr, color,(cross_start_x, cross_start_y+100), (cross_start_x + 100,
cross_start_y ),3)
pygame.display.flip()
```

In the previous code:

- We define the coordinates of the top left corner of the imaginary rectangle with the variables **cross_start_x** and **cross_start_y**.
- We then draw the first line from the top-left corner to the bottom-right corner of the rectangle.

- We then draw the second line from the bottom left corner to top right corner of the rectangle
- That's it, you can now save your code and compile it. As you do, you should notice a new cross added to the display screen, as illustrated by the next window.

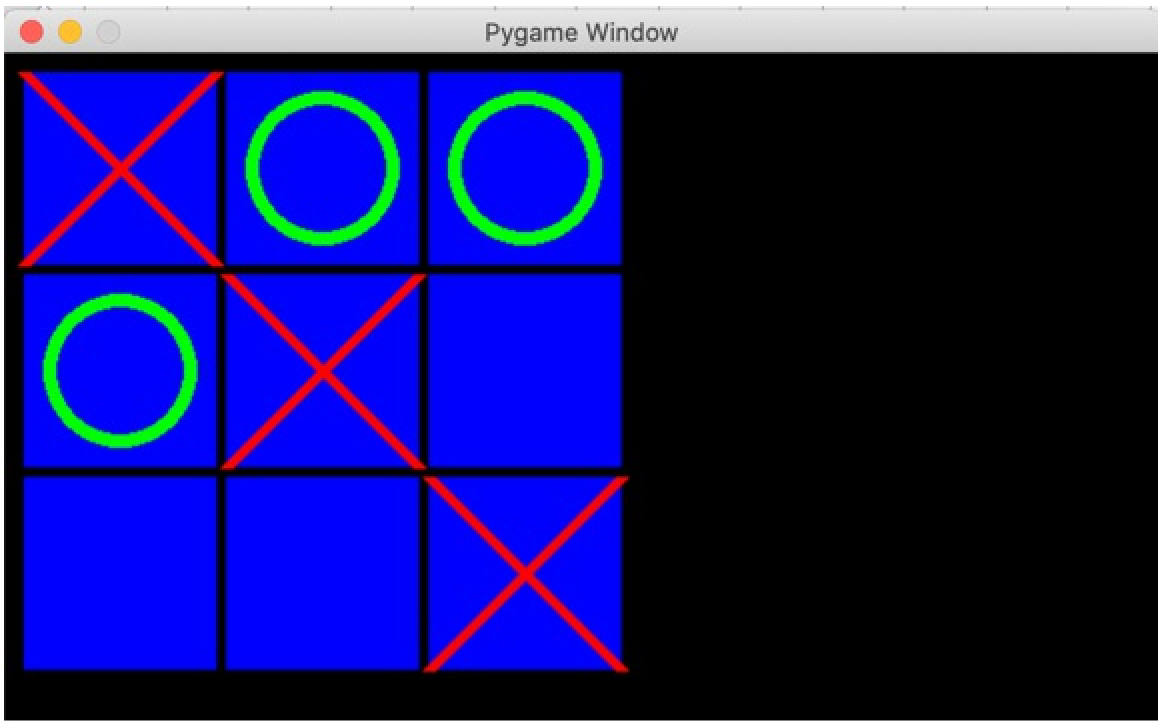


So at this stage, we have become familiar with drawing basic shapes on screen, and it would be great to compile these skills to create a simple Tic Tac Toe game, and this what we are going to complete in the next section.

Creating a TicTacToe game

In this section, we will build on the skills that we have acquired so far to create a Tic Tac Toe game.

- The game is relatively popular and simple:
- The game screen consists of a 3x3 grid.
- Each player can, in turn, click on one of the cells to add a circle or a cross.
- Once one of the players has managed to align three consecutive circles or crosses s/he has won.



So for this game, we will need to:

- Draw the 3x3 grid consisting of 9 blue rectangles.
- Detect where the player has clicked and draw a ross accordingly.
- Implement a simple AI that will play against the player and add circles

in places.

So first, let’s create a new file for that game, and start to work on the grid.

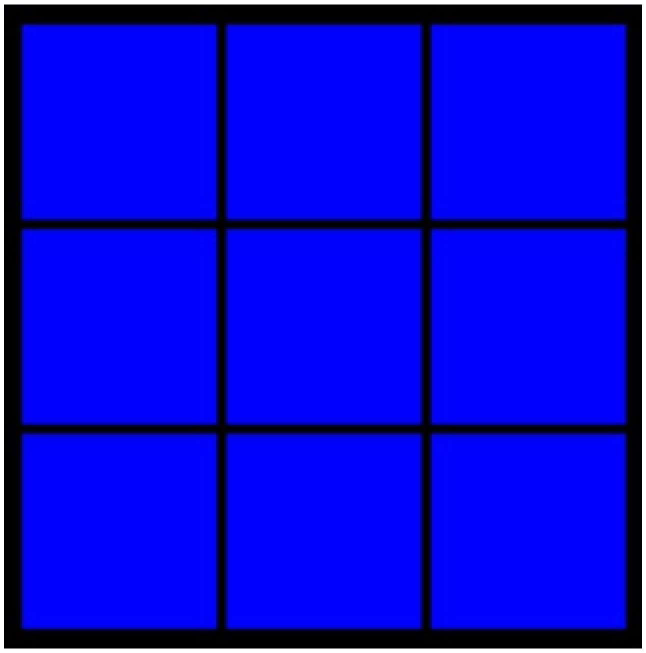
- Please create a new file in Sublime (**File | New**).
- Save this file **tictactoe.py**.
- Add the following code to the file.

```
import pygame
import pygame.mouse
pygame.init()
scr = pygame.display.set_mode((600,500))
pygame.display.set_caption('Pygame Window')
```

In the previous code:

- We import two libraries: the Pygame library and a more specific library that is used to handle and process mouse clicks.
- We then initialise the Pygame module.
- Finally, we set the size of the display window to **600 x 500**.
- We then set the cation for the window that will be used to display the game.

In the next section, we will start to create the interface for our game, including the 9 blue squares on which the player will need to click to draw either a circle or a cross as illustrated on the next figure:



- Please add the following code to the file **tictactoe.py**:

```
color = (0,0,255)

start_x = 10

start_y = 10

rect_width = 100

rect_height = 100

margin_x = 5

margin_y = 5
```

In the previous code:

- We set the color of the pen to be used to draw our interface to blue.
- We define the coordinated of the top left corner of the top left square.
- We define the width and height of each square.
- We also define the vertical and horizontal margin between each square.

Now that we have defined the color, the size and the margin for each of the 9 squares for our interface, it is time to draw them.

Please add the following code to the file **tictactoe.py**:

```
for i in range (3):
for j in range (3):
pygame.draw.rect(
scr,
color,
pygame.Rect(start_x+(rect_width+margin_x)*i,
start_y+(rect_height+margin_y)*j,
rect_width, rect_height))
pygame.display.flip()
```

In the previous code:

- We create two nested loops, both counting from 0 to 2 (i.e., 3-1) to draw the 9 squares (3 lines and 3 columns).
- We use the built-in function **pygame.draw.rect** to draw each square (i.e., rectangle).
- For each of them, we define: a screen (i.e., **scr**), a color, along with a rectangle shape. Each of these rectangles is defined by the coordinates of its top left corner, along with its width and its height.
- Finally, we refresh the display using the built-in function **display.flip**.

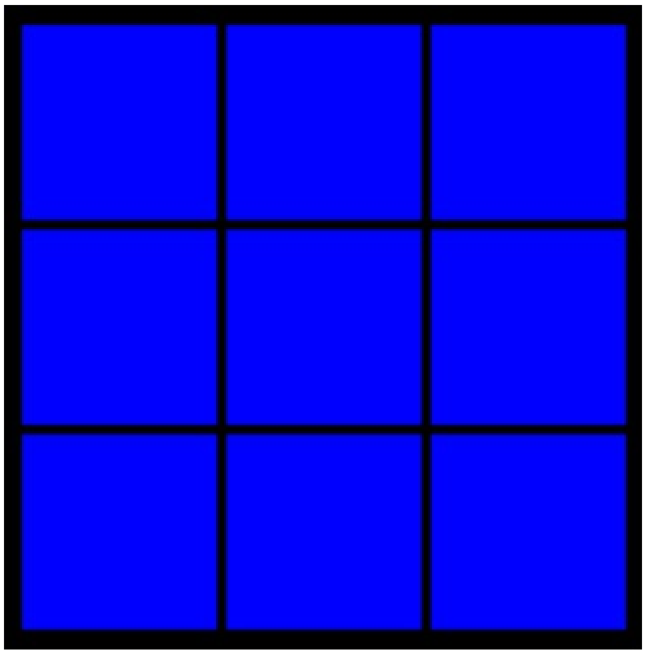
Once we have displayed these 9 squares, we just need to ensure that they remain displayed unless the user decides to quit the programme.

- Please add the following code to the script **tictactoe.py**:

```
done = False
while not done:
for event in pygame.event.get():
if event.type == pygame.QUIT:
done = True
pass
```

In the previous code we create a loop for the game, and we loop until the player decides to quit the game.

- You can now save and compile your code, and the following window should appear.



You can now quit the game.

At this stage the primary interface has been created, and we now need to detect the players mouse clicks to draw a circle or a cross in the corresponding square.

For this purpose, we will create a function that will do the following:

- Detect when the user has clicked the left mouse button
- Record the position of the mouse when the clicks occurred.
- Detect whether the user has already clicked in that cell
- If this is not the case, then draw a cross in this cell.

So let’s start:

- Please add the following code to the script **tictactoe.py** (new code in bold):

```
import pygame
import pygame.mouse
from pygame.locals import *
```

In the previous code, we import only the constants used by **Pygame**.

- Please add the following code (new code in bold):

```
margin_x = 5
margin_y = 5
cell_used = [[0,0,0],[0,0,0],[0,0,0]]
```

In the previous code, we create a two-dimensional list (mirroring a 3x3 grid) called **cell_used**, that stores information on whether a cell has already been used (i.e., selected by the user); initially, all the items in that list are set to **0**.

- Please add the following function to the script, after the code that you have just entered.

```
def detect_square_clicked(x,y):
```

```
global start_x
global start_y
global rect_width
global rect_height
global margin_x
global margin_y
global cell_used
print("Clicked at x="+str(x)+"__y="+str(y));
```

In the previous code:

- We create a function called **detect_square_clicked**. This function will be called whenever a user clicks on the screen. It takes two parameters which will correspond to the coordinates of the mouse where the user has clicked on the screen.
- We then make a reference to the global variables that we have defined earlier, as we will use them inside this function.
- Finally, we display the value of the parameters passed to this function.

So that we can test this function, please add the following code to the script (new code in bold):

```
while not done:
for event in pygame.event.get():
if event.type == pygame.QUIT:
done = True
if event.type == MOUSEBUTTONDOWN:
if event.button == 1:
position  = pygame.mouse.get_pos()
detect_square_clicked(position[0],position[1])
```

In the previous code:

- We check whether the mouse was clicked.
- If this is the case, we check that it was the left button.
- And if the **left-mouse-btton** (**LMB**) was pressed, we obtain the position of the mouse using the function **mouse.get_pos**, and pass both coordinates of the mouse to the function **detect_square** that we have defined earlier.

You can now save and compile your code; as the screen is displayed, you can click on one of the squares and check on the Command Prompt that the corresponding position has been displayed with a message similar to the following one:

```
Clicked at x=240__y=69
```

If that works, it means that the programme works properly and that you have managed to capture and display the position of the mouse where the user clicked.

You can now close the programme.

So, at this stage, we just need to display a cross where the mouse has been clicked.

- Please add the following code to the function **detect_square_clicked**:

```
print("Clicked at x="+str(x)+"__y="+str(y));
new_x_pos = ((x - start_x)/(rect_width + margin_x))
new_y_pos = ((y - start_y)/(rect_height + margin_y))
new_x_pos = int(new_x_pos)
new_y_pos = int(new_y_pos)
```

In the previous code:

- We define two variables **new_x_pos** and **new_y_pos** that will determine in which cell the player has clicked. So both the variables **new_x_pos** and **new_y_pos** can be **0**, **1**, or **2**.
- These two variables are calculated based on the position of the mouse (i.e., **x** and **y**), the width and height of each square, along with the vertical and horizontal margins.
- Regardless of the position of the mouse, we need to know in which cell the player clicked, and this is the reason why both variables **new_x_pos** and **new_y_pos** are floored by casting them to the nearest integer.

Now that we have defined the top-left corner of the square where the cross should be drawn, we just need to draw that cross, after checking that the user has not already checked that cell.

- Please add the following code to the function **detect_square_clicked** (new code in bold):

```
new_x_pos = int(new_x_pos)
new_y_pos = int(new_y_pos)
if (cell_used [new_y_pos][new_x_pos] == 0):
cell_used [new_y_pos][new_x_pos] = 1
color = (255,0,0)
cross_top_left_x = start_x+(rect_width+margin_x)*new_x_pos;
cross_top_left_y = start_y+(rect_height+margin_y)*new_y_pos
```

In the previous code:

- We checked whether the player has already clicked on that cell.

- If that is not the case, then we draw the cross.
- We set the color (i.e., red)
- We then define the coordinates of the top-left corner of the cell were the cross should be drawn.

Now that we have defined where the cross should be drawn, we can actually draw it using two lines; please add the following code to the function **detect_square_clicked**:

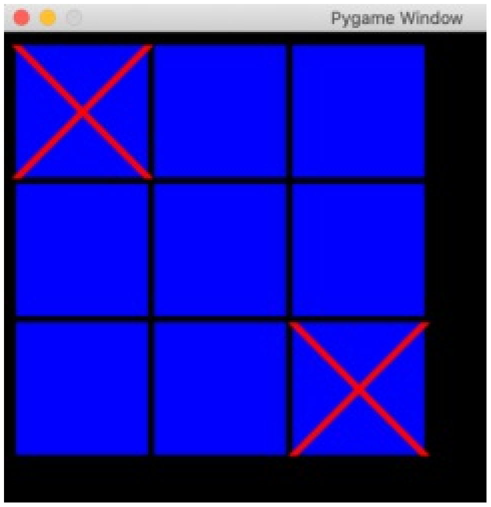
```
cross_top_left_y = start_y+(rect_height+margin_y)*new_y_pos
pygame.draw.line(
scr,
color,
(cross_top_left_x ,cross_top_left_y),
(cross_top_left_x + rect_width, cross_top_left_y +rect_height),
7)
pygame.draw.line(
scr,
color,
(cross_top_left_x ,cross_top_left_y + rect_height),
(cross_top_left_x + rect_width, cross_top_left_y),
7)
pygame.display.flip()
```

In the previous code:

- We draw the first line for the cross from the top-left corner to the bottom-right corner.
- We draw the second line for the cross from the bottom-left corner to the top-right corner

- We then update the display.

You can now save and compile your code, as the programme is displayed, you can click on one of the squares and a cross should be drawn in that cell, as per the next figure.



implementing Artificial intelligence

In this section, we will add the artificial intelligence that will make it possible to simulate a second player against the current one.

So the player will play first, the computer will then evaluate the game and play accordingly.

In this version, the computer will play very simply by filling any cell that has not been filled yet by the player, regardless of whether the player already crossed two consecutive cells.

- Please add the following code to the script **tictactoe.py**:

```
def computer_plays_simple():
global start_x
global start_y
global rect_width
global rect_height
global margin_x
global margin_y
global cell_used
is_looping = True
```

In the previous code:

- We define a function called **computer_plays_simple**.
- In this function we refer to global variables that have already been defined and that will be used within the function.
- We also define a variable called **is_looping** that will be used to determine whether the computer should exit a specific loop.

Now that these variables have been defined, we need to go through the 9 cells in the grid, and check which one is actually empty.

- Please add the following code (new code in bold):

```
is_looping = True
for j in range (3):
for i in range (3):
if (cell_used[j][i] == 0):
cell_used[j][i] = -1
```

In the previous code:

- We create two nested loops to go through each row (3 in total) and columns (3 in total) for our grid.
- We then check if any of the cells in the grid is empty using the list **cell_used**.
- And if the cell is empty, we then mark it as full accordingly. We use -1 to specify that the cell is now full but with the computer’s circle.

Please add the following code:

```
if (cell_used[j][i] == 0):
cell_used[j][i] = -1
cell_top_left_x = start_x+(rect_width+margin_x)*i
cell_top_left_y = start_y+(rect_height+margin_y)*j
color = (0,255,0)
circle_center = (cell_top_left_x + rect_width/2,cell_top_left_y + rect_height/2)
circle_radius = (rect_width)*.8/2
```

In the previous code:

- We define the coordinates of the top-left corner of the current cell that was found to be empty.
- We set the pen color to green.
- We then define the coordinates of the center of the circle to be drawn, along with its radius.

Now that the location of the circle has been defined, it is time to draw it onscreen.

- Please add the following code (new code in bold):

```
circle_radius = (rect_width)*.8/2
pygame.draw.circle(scr, color, circle_center, circle_radius, 7)
pygame.display.flip()
is_looping = False
break
if (is_looping == False):
break
```

In the previous code:

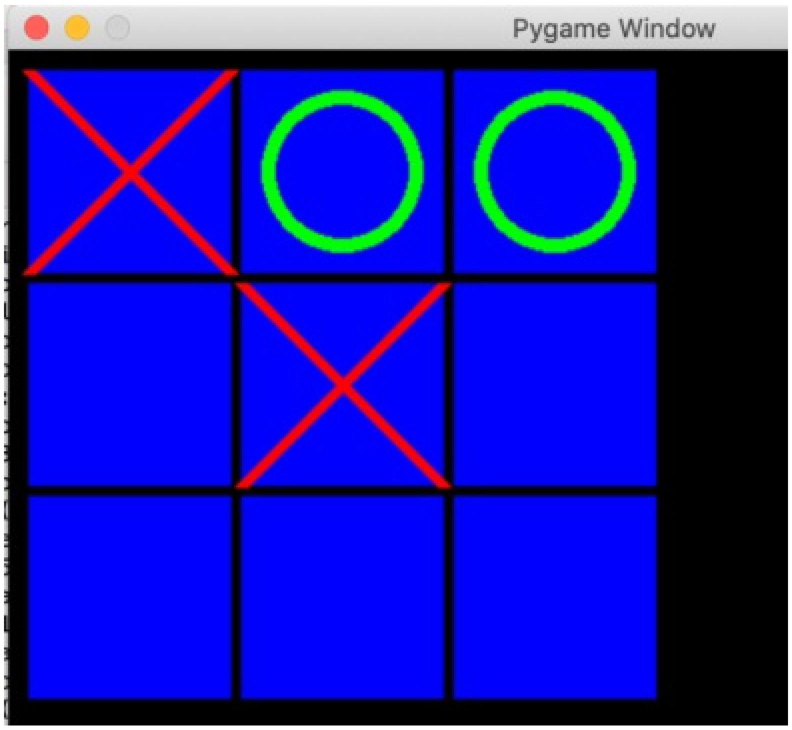
- We draw the circle based on the coordinates and radius that we have calculated previously.
- We refresh the display using the function **display.flip**.
- Since the circle has been drawn, we set the variable **is_looping** to **false** since we have indeed found (and filled) an empty cell and we exit both loops.

The last thing we need to be able to test this code is to call the function after the player has made its move; so please ass the following code to the script tictactoe.py (new code in bold):

```
if event.type == MOUSEBUTTONDOWN:
if event.button == 1:
position  = pygame.mouse.get_pos()
detect_square_clicked(position[0],position[1])
computer_plays_simple()
```

In the previous code, once the layer has made its move, it’s the computer’s turn to play, by calling the function **computer_plays_simple**.

You can now save and compile your code. As the programme launches, you will see that after you play, the computer will pick an empty cell and fill it with a green circle.



So at this stage the game works pretty well, and we just need to detect when the player (or the computer) has won.

To do so, we will check whether 3 consecutive cross or circles have been added on the same line, row, or diagonal.

- Please add this code to the script **tictactoe.py**.

```
def check_if_a_player_won():
global cell_used
found_a_line = False
```

In the previous code, we define a new function called **check_if_a_player_has_won** and we refer to the global variable called **cell_used**, which stores information about the grid cell.

- Please add the following code to the function:

```
line1_sum = cell_used[0][0] + cell_used[0][1] + cell_used[0][2]
line2_sum = cell_used[1][0] + cell_used[1][1] + cell_used[1][2]
line3_sum = cell_used[2][0] + cell_used[2][1] + cell_used[2][2]
col1_sum = cell_used[0][0] + cell_used[1][0] + cell_used[2][0]
col2_sum = cell_used[0][0] + cell_used[1][0] + cell_used[2][0]
col3_sum = cell_used[0][2] + cell_used[1][2] + cell_used[2][2]
diagno1_sum = cell_used[0][0] + cell_used[1][1] + cell_used[2][2]
diagno2_sum = cell_used[2][0] + cell_used[1][1] + cell_used[0][2]
```

In the previous code we calculate the sum of each row, column, and diagonal, bearing in mind that each cell that is empty, with a cross or a circle is marked as **0**, **1**, or **-1**, respectively. This will be useful to know whether three consecutive crosses or circles from a line.

- Please add the following code to the function:

```
line1_prod = cell_used[0][0] * cell_used[0][1] * cell_used[0][2]
line2_prod = cell_used[1][0] * cell_used[1][1] * cell_used[1][2]
line3_prod = cell_used[2][0] * cell_used[2][1] * cell_used[2][2]
```

```
col1_prod = cell_used[0][0] * cell_used[1][0] * cell_used[2][0]
col2_prod = cell_used[0][0] * cell_used[1][0] * cell_used[2][0]
col3_prod = cell_used[0][2] * cell_used[1][2] * cell_used[2][2]
diagno1_prod = cell_used[0][0] * cell_used[1][1] * cell_used[2][2]
diagno2_prod = cell_used[2][0] * cell_used[1][1] * cell_used[0][2]
```

In the previous code we calculate the product of each row, column, and diagonal, bearing in mind that each cell that is empty, with a cross or a circle is marked as **0**, **1**, or **-1**, respectively. This will be useful to know whether the line that has been detected is made of crosses or circles.

- Please add the following code to the function:

```
if (abs(line1_sum) == 3
or abs(line2_sum) == 3
or abs(line3_sum) == 3
or abs(col1_sum) == 3
or abs(col2_sum) == 3
or abs(col3_sum) == 3
or abs(diagno1_sum) == 3
or abs(diagno2_sum) == 3):
```

In the previous code we check if any of the cell content from any of the rows, columns or diagonals adds up to 3 or -3; since each cell occupied with a circle is marked as **-1**, and each cell occupied with a cross is marked as **1**, a line of crosses will add up to **3** and a line of circles will add up to **-3**.

- Please add the following code (new code in bold):

```
or abs(diagno2_sum) == 3):
```

```
if (line1_prod > 0
or line2_prod > 0
or line3_prod > 0
or col1_prod > 0
or col2_prod > 0
or col3_prod > 0
or diagno1_prod > 0
or diagno2_prod > 0):
```

In the previous code we check if the product of the content of the cells of any of the rows, columns or diagonals is positive; since each cell occupied with a circle is marked as **-1**, and each cell occupied with a cross is marked as **1**, the product of the content of cells in a line of crosses will be positive (**+1**) and the product of the content of cells in a line of circles will be negative (**-1**).

- Please add the following code (new code in bold):

```
or diagno2_prod > 0):
print ("GAME OVER >> PLAYER WINS")
else:
print ("GAME OVER >> COMPUTER WINS")
```

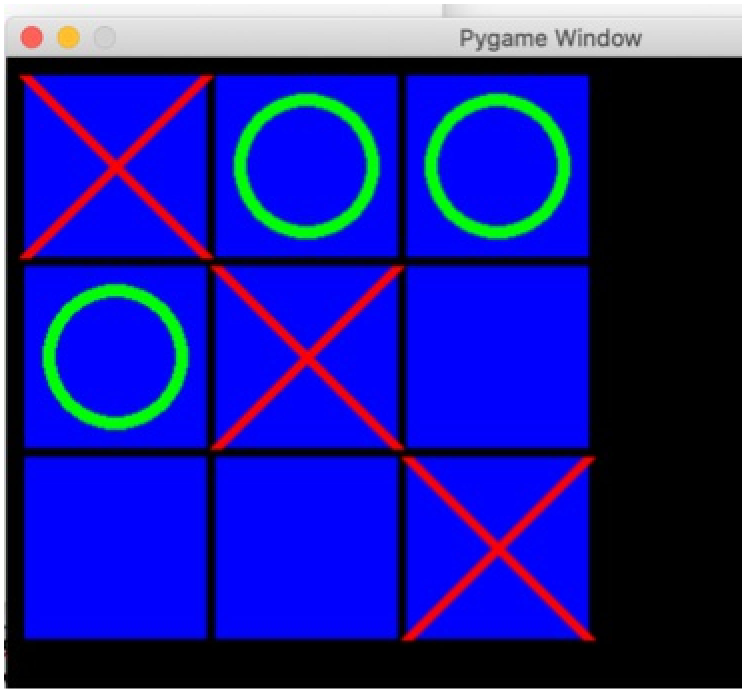
In the previous code we display who has won based on the previous calculations.

Finally, we just need to call this function every time a player has made a move; so please add the following code to the script (new code in bold):

```
detect_square_clicked(position[0],position[1])
check_if_a_player_won()
computer_plays_simple()
check_if_a_player_won()
```

In the previous code, we call this function every time a player has completed a move.

You can now save and compile your code, and check that whenever you have managed to align three crosses that the corresponding message is displayed in the **Command Prompt**, as per the next figure.



As per the previous figure, if you have aligned three crosses, the **Command Prompt** should display the following message:

```
GAME OVER >> PLAYER WINS
```

Level roundup

In this chapter, we have learned how to use graphics and draw basic shapes onscreen with the Pygame. We became more comfortable with functions, loops, and lists and we finally created a simple tictactoe game with a relatively simple artificial intelligence so that the player can play against the computer.

Checklist

You can consider moving to the next stage if you can do the following:

- Detect mouse clicks
- Import the Pygame library.
- Draw circles and lines.
- Change the color of the pen.
- Use loops and conditional statements

Quiz

Now, let’s check your knowledge! Please answer the following questions. The answers are on the next page.

Please specify whether the following statement are **TRUE** or **FALSE**

1. The Pygame library needs to be installed before it can be used
2. The Pygame library needs to be imported before it can be used
3. It is possible to set the size of the display area with Pygame.
4. It is possible to specify the caption of the display window with Pygame.
5. The color to be used to draw shapes in Pygame is specified using the RGB code.
6. The function **pygame.draw.rect** can be used to draw a rectangle.
7. Pygame.QUIT is the event called when the user quits the Pygame window.
8. MOUSEBUTTONDOWN is the event called when the user clicks on the mouse
9. The function **pygame.draw.line** can be used to draw a line.
10. The function **pygame.draw.circle** can be used to draw a circle.

1.

Solutions to the Quiz

1. **TRUE**.
2. **TRUE.**
3. **TRUE**.
4. **TRUE**.
5. **TRUE.**
6. **TRUE.**
7. **TRUE.**
8. **TRUE.**
9. **TRUE**
10. **TRUE.**

1.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, you could use these to improve the flow of your game. So for this challenge, you will be creating an instruction screen and a game-over screen.

- Change the shapes drawn on screen for both the player and the computer (e.g., triangles).
- Change the color of these chapes.
- Change the thickness of the pen used to draw these shapes.

Challenge 2

Here, you will modify the artificial intelligence of the computer:

- Check whether the player has two consecutive cells filled.
- In that case, fil the third cell so that the player cannot win at the next move.

CHAPTER 6: CREATING A COIN COLLECTION GAME

In this section, we will create a simple coin collection game where the layer needs to collect several coins in each level, before accessing the door to the next level.

After completing this chapter, you will be able to:

- Move an animated character with the arrow keys.
- Detect collision between the player and the walls.
- Create moving enemies with moderate intelligence.
- Create buttons that the player can click.
- Create a level with sprites.
- Implement a scoring system.
- Display text onscreen.
- Implement a timer.
- Implement a splash-screen.

Introduction

So, as mentioned earlier, we are going to create a coin collection game. The principle is quite common and involves the following:

- The player will be presented with a splash-screen and given the option to start by pressing on a button.
- The player will have three lives in total.
- A timer will be starting from the first level.
- Each level will include walls, items to collect, enemies, and a door to access the next level.
- The player will be able to navigate this maze using the arrow keys.
- The player loses a life after colliding with an enemy or when the time is up.
- The player needs to collect at least four coins in each level to be able to open the door.
- The player wins after completing the three levels.

Moving the player around

In this section, we will get to move the main character based on the keyboard arrows; for the time being, the player will be symbolized by a red square that we will replace, later, by an animated character.

- Please create a new file and save it as **adventure.py**.
- Please add the following code to the script **adventure.py**.

```
import pygame
screen_width, screen_height=900,600
main_window=pygame.display.set_mode((screen_width,screen_height))
pygame.display.set_caption("Coin Collector")
```

In the previous code:

- We import the **pygame** module.
- We set the size of the screen that will be used to display our game.
- We set the caption for that window.

Please add the following code:

```
fps_rate=60
player_velocity=4
bg_img=pygame.image.load("assets/bg_image.jpg")
bg_img=pygame.transform.scale(bg_img,(screen_width,screen_height))
```

In the previous code:

- We define a variable called **fps_rate** that will be used to set the refresh rate for our window.
- We define a variable called **player_velocity** that will be used to

determine the moving speed of the player character.

- We also load an image called **bg_image.jpg** located in the sub-folder called **assets**.
- This image is then stretched to match the size of the screen.

Now that we have defined and set global variables that will be used to display the game, it is time to create specific functions that will deal with the display and movement of the character player.

- Please add the following function to the script:

```
def display_main_window(player_rect):
main_window.blit(bg_img,(0,0))
pygame.draw.rect(main_window,"Red",player_rect)
pygame.display.update()
```

In the previous code:

- We define a function called **main_window** that will be in charge of displaying the different elements of the game (e.g., the player, the NPCs or the walls).
- We display the background image.
- We display the red square that symbolized the player character for the time being.
- We then refresh the display.

Next, we will need to detect the keys pressed by the player.

- Please add the following function to the script:

```
def controls(keys_pressed,player_rect):
if keys_pressed[pygame.K_LEFT] and player_rect.x>10 :
player_rect.x -=player_velocity
if keys_pressed[pygame.K_RIGHT] and player_rect.x<screen_width-10 :
player_rect.x +=player_velocity
if keys_pressed[pygame.K_UP] and player_rect.y>10 :
player_rect.y -=player_velocity
if keys_pressed[pygame.K_DOWN] and player_rect.y<(screen_height -10 - 25) :
player_rect.y +=player_velocity
```

In the previous code:

- We define a function called controls that takes two parameters consisting of the key pressed and the rectangle used to symbolize the player character.
- We then use conditional statements to determine the key pressed and to move the player character consequently.
- If the left arrow is pressed and the player character is more than 10 pixels from the left side of the screen, we move the player character to the left.
- If the right arrow is pressed and the player character is less than 10 pixels from the right side of the screen, we move the player character to the right (we also account for the rectangle’s with which is **25**).
- If the up arrow is pressed and the player character is less than 10 pixels from the top side of the screen, we move the player character upwards.
- If the down arrow is pressed and the player character is less than 10 pixels from the bottom of the screen, we move the player character down.

So now that we have defined functions that detect the keys pressed by the player and that display the player character, we can create the main loop of the game where we will call both of these functions to be able to move and display the character based on the keys pressed.

- Please add the following function:

```
def main():
player_rect=pygame.Rect(50,300,25,25)
clock= pygame.time.Clock()
Time=120
```

In the previous code:

- We define a function called main, that will be the main loop of our game
- We define a variable called **player_rect** that will be the rectangle used to represent the player character.
- We define a variable called clock that will be used to set the refresh rate for the game.

Now that these variables have been defined, we can start to create the main loop.

- Please add the following code (new code in bold):

clock=pygame.time.Clock()

while True:

clock.tick(fps_rate)

for event in pygame.event.get():

if event.type==pygame.QUIT:

```
pygame.quit()
key = pygame.key.get_pressed()
display_main_window(player_rect)
controls(key,player_rect)
```

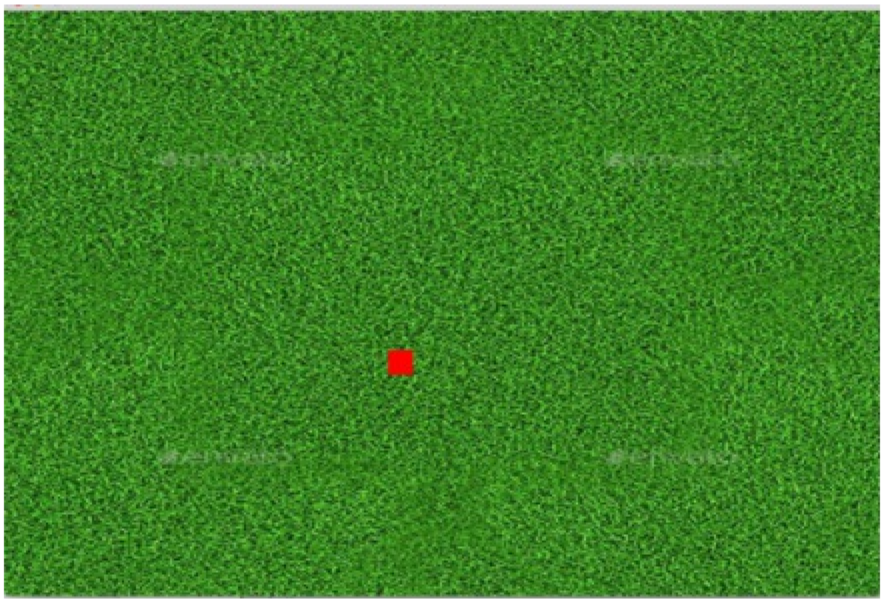
In the previous code:

- We create an infinite loop.
- Within this loop we then set the refresh rate using the function called tick.
- We then capture all events and detect whether the Pygame window is closed to un-initialize all Pygame modules.
- We then detect whether a key has been pressed by the player.
- We call the function **display_main_window** passing the rectangle for the player character as a parameter.
- We also call the function **controls** and pass two variables as parameters: **key** which is the key that has been pressed by the player and **player_rec** which is the rectangle that represents the player character.

Finally, please add the following code at the end of the script:

```
main()
```

That's it. You can now compile and run the script. As you do you should see a red square on the screen that you should be able to move by pressing the arrow keys. You may check that the player cannot go beyond the borders of the screen, as per the next figure.



Collecting coins

So at this stage you can move the red square that symbolizes the player character, and you can also ensure it remains within the screen.

The next step will be to make it possible to collect an item, that we will symbolize by a square. This will involve the following features:

- Defining rectangles that symbolize the objects to be collected.
- Drawing these rectangles on screen.
- Detect collisions between the player character and these rectangles.

Please add the following code at the beginning of the script (new code in bold):

bg_img=pygame.transform.scale(bg_img,(screen_width,screen_height))

coin1 = pygame.Rect(200,200,20,20)

coin2 = pygame.Rect(200,250,20,20)

coin3 = pygame.Rect(250,200,20,20)

coin4 = pygame.Rect(250,250,20,20)

In the previous code, we define four rectangles called **coin1**, **coin2**, **coin3** and **coin4**, along with their position and size (**20** by **20**).

- Please add the following code to the function **display_main_window** (new code in bold):

pygame.draw.rect(main_window,"Red",player_rect)

pygame.draw.rect(main_window,"Yellow",coin1)

pygame.draw.rect(main_window,"Yellow",coin2)

pygame.draw.rect(main_window,"Yellow",coin3)

pygame.draw.rect(main_window,"Yellow",coin4)

In the previous code we draw the four squares that symbolize the items to be collected by the player using a yellow color.

Finally, we need to create a function that will detect the collision between the player and these items to be collected.

- Please add the following function to the script:

```
def detect_collision(player_rect):
global coin1,coin2,coin3,coin4
if player_rect.colliderect(coin1):
coin1.x=screen_width + 10
elif player_rect.colliderect(coin2):
coin2.x=screen_width + 10
elif player_rect.colliderect(coin3):
coin3.x=screen_width + 10
elif player_rect.colliderect(coin4):
coin4.x=screen_width + 10
```

In the previous code:

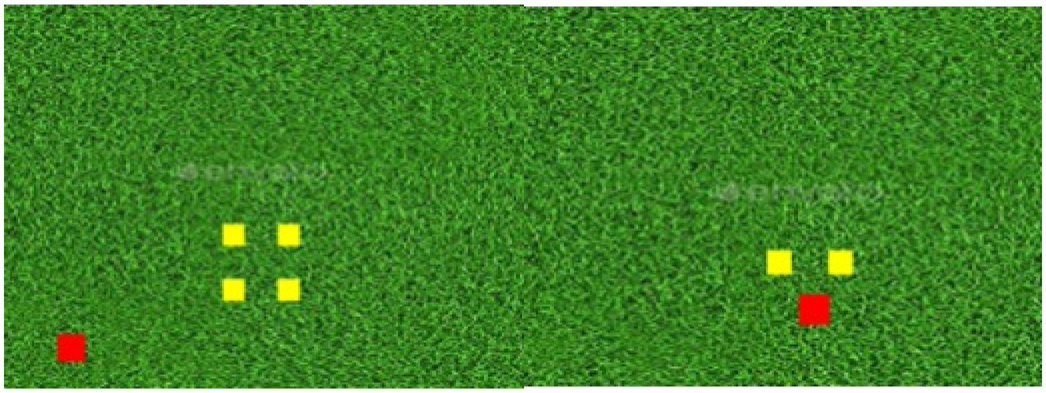
- We define a function called **detect_collision** that takes one parameter that will be referred to as **player_rect** within this function.
- We make a reference to the global variables **coin1**, **coin2**, **coin3**, and **coin4** that will be used within this function.
- If the player collides with any of these items, the coin (i.e., yellow square) is moved outside the screen.

Finally, we just need to call this function from the main loop; so please add the following code to the function main (new code in bold):

controls(key,player_rect)

detect_collision(player_rect)

You can now compile and run your code, and you as you move the player character and collide with the items, these should disappear from the screen.



Tracking and displaying the score

In the previous section, we have managed to collect items; now we will start to create and display a score that will track the number of items collected.

- Please add the following code at the beginning of the script (new code in bold)

```
import pygame
pygame.font.init()
```

In the previous code, we initialize the fonts

- Please add the following code before the first function (new code in bold):

```
coin4 = pygame.Rect(250,250,20,20)
score = 0
coin_font = pygame.font.SysFont('arial',50)
```

In the previous code:

- We declare a variable called score that will be used to track the number of coins collected.
- We declare a (font) variable called **coin_font** that will be used for the text displayed for the score
- This font will be based on the font "**Arial**" available by default on your computer, and the size used will be **50** pixels.

Next, we will ensure that the score is increased every time we collide with an item to collect.

- Please modify the function **detect_collision** as follows (new code in bold):

```
def detect_collision(player_rect):
global coin1,coin2,coin3,coin4, score
if player_rect.colliderect(coin1):
coin1.x=screen_width + 10
score += 1
elif player_rect.colliderect(coin2):
coin2.x=screen_width + 10
score += 1
elif player_rect.colliderect(coin3):
coin3.x=screen_width + 10
score += 1
elif player_rect.colliderect(coin4):
coin4.x=screen_width + 10
score += 1
```

In the previous code we make a reference to the global variable score, and we increase the code by one at each collision between the player and a coin.

Finally, we just need to display the score onscreen.

- Please add the following code (new code in bold) to the function **display_main_window**:

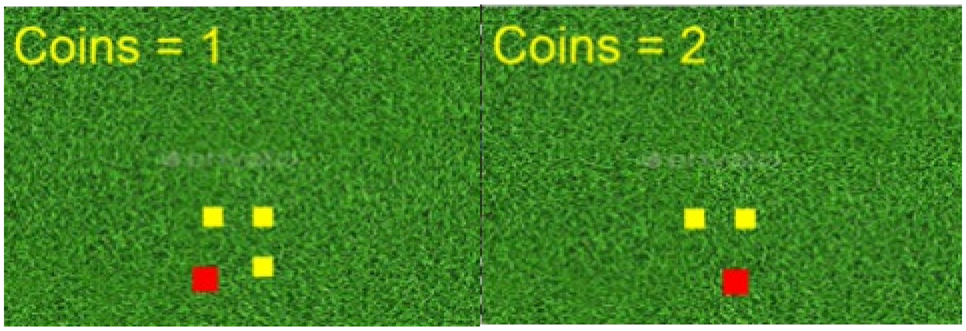
```
score_text = coin_font.render("Score = "+str(score),1,"Yellow")
```

```
main_window.blit(score_text,(10,10))
pygame.display.update()
```

In the previous code:

- We create a variable called **score_text**.
- **score_text** is a surface used to display the message "**Score**" followed by the number of coins collected. The text will be displayed in **yellow**.
- This surface is then displayed at the position **(10, 10)**.

You can now save and compile your code. As you run the programme, and collect the coins, you should see that a message displayed in the top-left corner indicates the number of coins collected, as per the next figures.



Creating the second level

Now that we can track the score, we can start to modify our code so that the player can move on to the next level if 4 coins have been collected.

- Please add the following code before the first function (new code in bold):

```
coin_font=pygame.font.SysFont('arial',50)
level  = 1
```

In the previous code, we create a variable called **level** that will keep track of the current level achieved for the player.

- Please add this code to the function **detect_collision** (new code in bold).

```
def detect_collision(player_rect):
global coin1,coin2,coin3,coin4, score, level
```

In the previous code we make a reference to the global variable **level** as we will be using it within the function.

- Please add this code to the function **detect_collision**.

```
if (score == 4):
score +=2
change_level()
```

In the previous code, we check whether the player has collected four items; if tats the case, we add a bonus of 2 to the score and call the function **change_level** which will be in charge of drawing the layout for the new level.

- Please add the following function to the script:

```
def change_level():
global coin1, coin2, coin3, coin4, level
level += 1
if (level == 2):
coin1.x, coin1.y = (20,90)
coin2.x, coin2.y = (300,90)
coin3.x, coin3.y = (10,500)
coin4.x, coin4.y = (300,400)
main()
```

In the previous code:

- We define the function called **change_level**.
- We make a reference to five global variables.
- We increase the level by 1.
- If the player has reached the second level, we then set the new positions of the items to be collected.

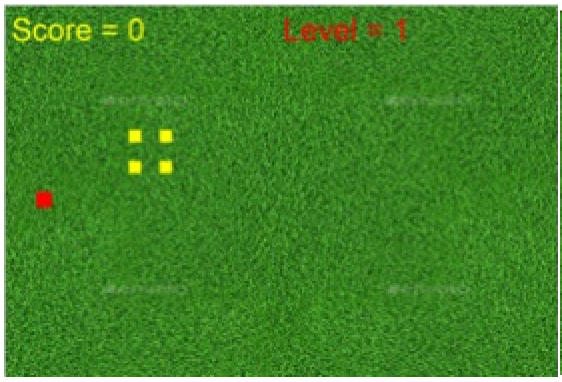
Finally, we just need to display the level onscreen.

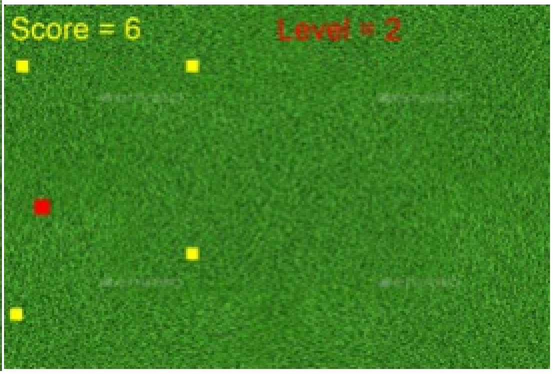
- Please add the following code to the function **display_main_window** (new code in bold):

```
level_text = coin_font.render("Level = " + str(level),1,"Red")
main_window.blit(level_text,(screen_width/2 ,10))
pygame.display.update()
```

In the previous code, we display the text that indicates the current level on screen.

You can now save and compile your script. As the program runs, you should see that the current level is displayed onscreen and that the layout changes as soon as you collect four coins.





Adding walls

In this section, we start to add walls that the player can collide with; these will consist of black squares, for the time being.

The idea here will be do:

- Create squares that will symbolize the walls.
- Detect collisions between the player and the walls.
- Move the player in the opposite direction in that case (i.e., step back).

Please add the following code to the start of the script before any function (ne code in bold):

```
    coin4=pygame.Rect(250,250,20,20)
    player_direction_x, player_direction_y = (0,0)



wall1 = pygame.Rect(100,100,100,100)
    wall2 = pygame.Rect(300,100,100,100)
    wall3 = pygame.Rect(500,100,100,100)
    wall4 = pygame.Rect(700,100,100,100)
    wall5 = pygame.Rect(100,300,100,100)
    wall6 = pygame.Rect(300,300,100,100)
    wall7 = pygame.Rect(500,300,100,100)
    wall8 = pygame.Rect(700,300,100,100)
```

In the previous code:

- We create a variable called **player_diretion** that will store the direction of the player along the x and y axis.

- We also create a set of 8 walls labelled **wall1** to **wall8**.
- The position and size of each wall is defined also.

Now that the rectangles for each wall have been defined, it is time to draw them on screen.

Please add the following code to the function **display_main_window** (new code in bold):

pygame.draw.rect(main_window,"Yellow",coin4)

pygame.draw.rect(main_window,"Black",wall1)

pygame.draw.rect(main_window,"Black",wall2)

pygame.draw.rect(main_window,"Black",wall3)

pygame.draw.rect(main_window,"Black",wall4)

pygame.draw.rect(main_window,"Black",wall5)

pygame.draw.rect(main_window,"Black",wall6)

pygame.draw.rect(main_window,"Black",wall7)

pygame.draw.rect(main_window,"Black",wall8)

In the previous code, we draw each of the 8 walls with a black ink.

Finally, we just need to detect collision between the player and these walls; for this purpose, we will modify the function **detect_collision**.

- Please add the following code at the start of the function **detect_collision** (new code in bold):

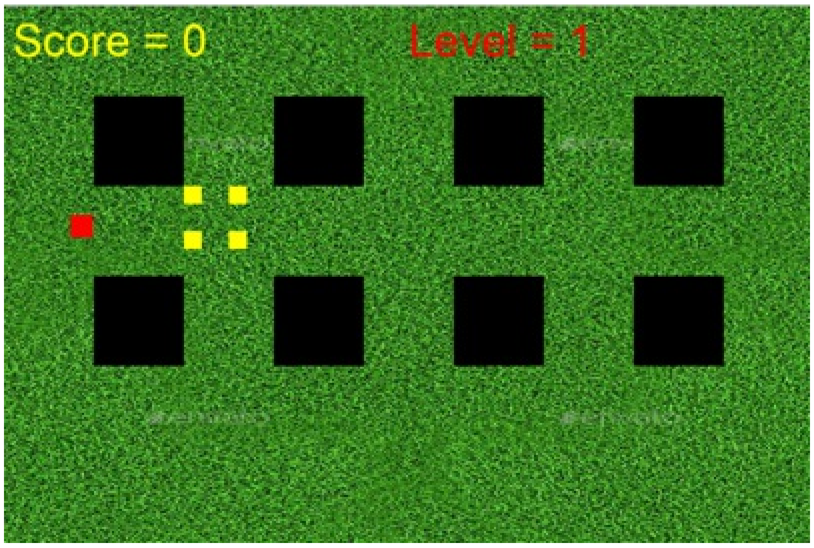
global coin1, coin2, coin3, coin4, wall1, wall2, wall3, wall4, wall5, wall6, wall7, wall8, level

- Add the following code to the function **detect_colision** (new code in bold):

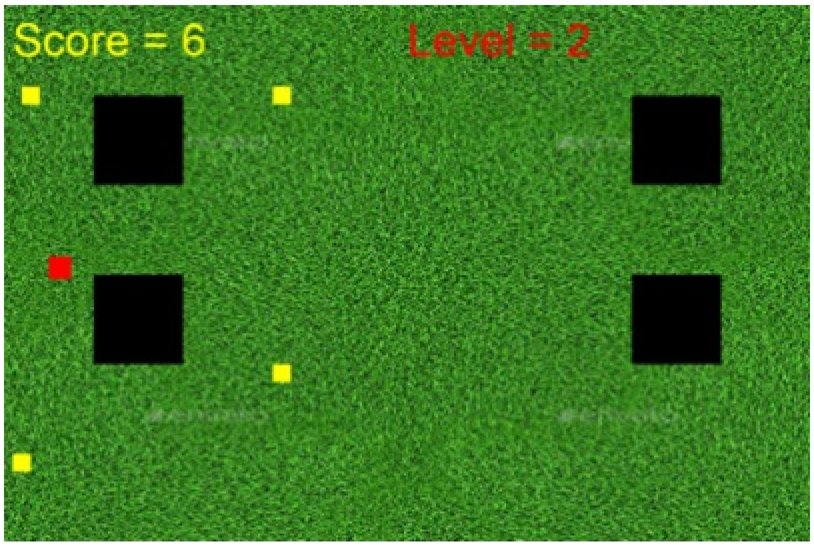
```
elif player_rect.colliderect(coin4):
coin4.x=screen_width + 10
score += 1
elif (
player_rect.colliderect(wall1) or
player_rect.colliderect(wall2) or
player_rect.colliderect(wall3) or
player_rect.colliderect(wall4) or
player_rect.colliderect(wall5) or
player_rect.colliderect(wall6) or
player_rect.colliderect(wall7) or
player_rect.colliderect(wall8)):
player_rect.y -= player_direction_y * player_velocity
player_rect.x -= player_direction_x * player_velocity
if (score == 4):
```

In the previous code, we check whether the player collides with one of the eight walls defined earlier, and if this is the case, we then move the player back using its current direction and its velocity.

You can now save and compile your code; as you run the program, you should be able to move the player character and check that it collides with the wall.



As you collect all the coins in that level, the second level should have a different layout similar to the following figure:



Adding a timer

In this section, we will add a timer that will count down from 1 minute; if the player has not completed the current level within 1 minute, then that level will restart.

- Please add the following code before the first function (new code in bold):

```
level  = 1
timer = 0
timer_message = ""
```

In the previous code, we declare two global variables **timer** and **timer_message** that will be used to both implement and display the timer:

- Please modify the definition of the function **change_level** as follows:

```
def change_level(new_level):
global coin1, coin2, coin3, coin4, wall1, wall2, wall3, wall4, wall5, wall6, wall7, wall8, level
level = new_level
if (level == 1):
coin1=pygame.Rect(200,200,20,20)
coin2=pygame.Rect(200,250,20,20)
coin3=pygame.Rect(250,200,20,20)
coin4=pygame.Rect(250,250,20,20)
wall1=pygame.Rect(100,100,100,100)
wall2=pygame.Rect(300,100,100,100)
wall3=pygame.Rect(500,100,100,100)
```

```
wall4=pygame.Rect(700,100,100,100)
wall5=pygame.Rect(100,300,100,100)
wall6=pygame.Rect(300,300,100,100)
wall7=pygame.Rect(500,300,100,100)
wall8=pygame.Rect(700,300,100,100)
elif (level == 2):
coin1.x, coin1.y = (20,90)
```

In the previous code:

- We modify the definition of the function called **change_level** so that it can take one parameter; this is because we will need this function to be able to reload the current level (and not always display the next level) when the timer is up.
- We set the global variable level with the parameter passed to that function.
- We then specify where the walls should be displayed for the first level.
- We modify the conditional statement for **level2** to **elif** instead of **if**.

Now that the function **change_level** has been modified, we can start to implement our timer and to also modify previous code that using a call to this function without parameter.

- Please modify the function **detect_collision** as follows (new code in bold):

```
if (score == 4):
score +=2
change_level(level+1)
```

In the previous code, we call the function **change_level**; however, we pass the value **level + 1** so that the new level is displayed instead.

- Please modify the function **main** as follows (new code in bold):

```
def main():
global timer, timer_message
```

- Add this modification as well.

```
while True:
clock.tick(fps_rate)
timer += clock.tick(fps_rate)/1000
minutes = int (timer/60)
seconds = int (timer%60)
timer_message = str(minutes) + ":" + str(seconds)
if (timer >= 120):
change_level(level)
timer = 0
```

In the previous code:

We add one to the variable timer every seconds.

- We then define the variables minutes and seconds.
- The variable minutes is obtained by dividing the number of seconds by 60, and then converting the result to an integer.
- The variable seconds is obtained by using the modulo operators (which provides the reminder of the division), and then by converting the result to an integer.
- We then set the variable **timer_message**, which is the time that will be

displayed on screen.

- Finally, we check whether we have reached two minutes (i.e., **120** seconds); in that case, we call the function **change_level** passing the current level as a parameter so that the current level is reloaded.
- We also reset the timer variable to 0.

The next and last part is to display the timer.

- Please add the following code at the end of the function **display_main_window** (new code in bold), before its last line:

```
time_text = coin_font.render("Time = "+timer_message,1,"Green")
main_window.blit(time_text,(screen_width/2 ,screen_height - 50))
pygame.display.update()
```

In the previous code we create a new surface on which the text will be displayed, using a green ink, and we place that text at the bottom of the screen.

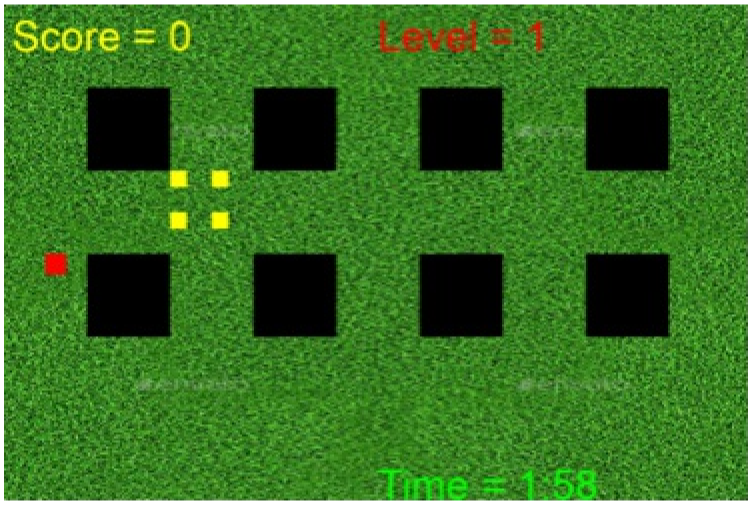
That’s it. To be able to test your code without having to wait for too long before the timer is up, you can modify this line from ...

```
timer = 0
```

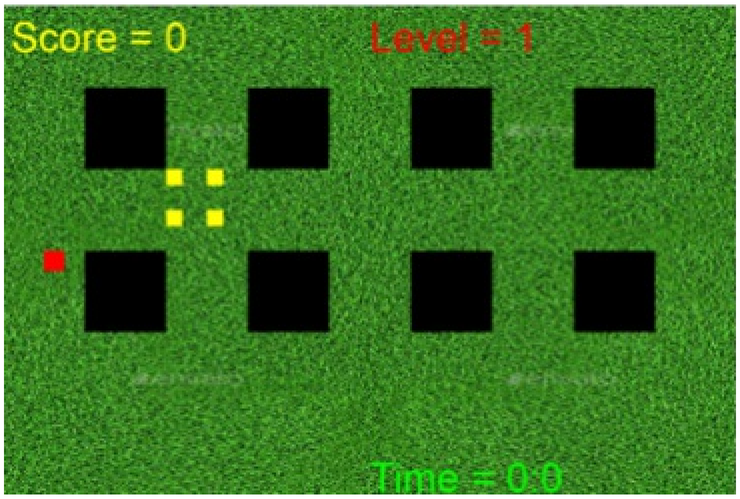
to ...

```
timer = 115
```

You can now save and compile your code. As you run the programme, you should see the timer displayed at the bottom of the screen, as per the next figure.



And after 5 seconds, the level should reload with a reset timer.



Adding a splash screen and end screen

At this stage we have a fully working game; however, it would be great to add a splash screen and an end screen as well, and this is exactly what we are going to do in this section.

So in this section, we will create a **splashscreen** that will display the instructions to the player; after reading the **instructions**, the player will need to press the **S** key to be able to start the first level.

To create this feature, we will successively:

- Display the instructions message.
- Wait for the player to press the **S** key.
- Open the first level once the key has been pressed.

First, we will declare some variables that will be used to display the slash-screen:

- Please add the following code, before any function at the beginning of the scrips (new code in bold):

```
timer_message = ""
menu_img = pygame.image.load("assets/menu.jpg")
menu_img = pygame.transform.scale(menu_img,(screen_width,screen_height))
info_font=pygame.font.SysFont('arial',30)
```

In the previous code:

- We create an image variable **menu_img** that points to the image called menu.jpg stored in the folder called assets.

- We scale the image so that it fits the width and height of the screen.
- We create a new font called **info_font** that will be used to display the information onscreen.

Next, we will display the splash-screen, using a combination of an image for the background and text for the instructions.

- Please, add the following function to the script:

```
def menu():
while True:
for event in pygame.event.get():
if event.type==pygame.QUIT:
pygame.quit()
```

In the previous code:

- We create a function called menu.
- Within this function, we loop indefinitely.
- Within the loop, we check whether the player quit the game window; in that case, we quit Pygame.

- Please add the following code just after the code that you have typed (new code in bold):

```
pygame.quit()
main_window.blit(menu_img,(0,0))
key = pygame.key.get_pressed()
```

```
    instructions = info_font.render("Collect all coins in each level to access the next level",1,"RED")
    main_window.blit(instructions,(10,250))
    instructions_text = info_font.render("PRESS S TO START THE GAME",1,"Yellow")
    main_window.blit(instructions_text,(170,300))
    pygame.display.update()
```

In the previous code:

- We display the background image.
- We detect the keys pressed.
- We create a surface that will contain the instructions for the player, and we ensure that these are displayed in red.
- We display the instructions.
- We update the screen.

Once this is done, we just need to check that the player has pressed the **S** key. So please add the following code just after the one that you have just typed.

```
    if key[pygame.K_s]:
    main()
```

In the previous code we check the **S** key has been pressed; in that case we call the function **main** to start the first level.

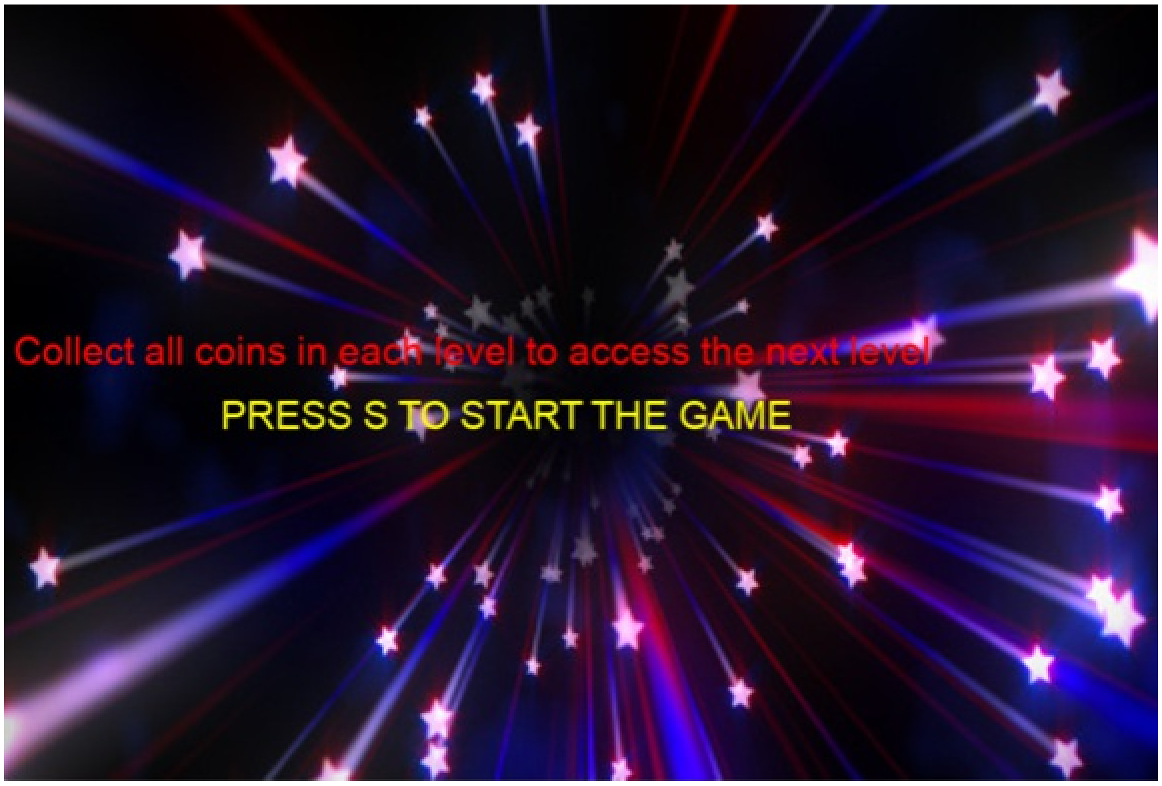
Finally, we need to ensure the menu is displayed first (i.e., not the first level); please replace this code at the end of the script:

```
    main()
```

...with this code ...

```
    menu()
```

You can now compile and save your code; you should see the splash-screen as illustrated in the next figure, and as you press the **S** key, the first level should start.


Collect all coins in each level to access the next level
PRESS S TO START THE GAME

Now that we have added the splash-screen, we are going to use the same principle to display the end screen. This screen will be displayed once the player has completed both levels:

- Once the player has completed both levels, a congratulation message will be displayed.
- Then the player will be offered to restart the game if s/he presses the **R** key.
- Once this key is pressed, the splash-screen will be displayed.

So, first we will display a congratulations message to the player and then wait for a few seconds.

- Please add the following function to the script:

```
def victory():
global score, level
main_window.fill('White')
info_text = info_font.render("YOU WON! ",1,"RED")
main_window.blit(info_text,(340,270))
pygame.display.update()
time.sleep(2)
```

In the previous code:

- We define a function called **victory**.
- In that function we refer to two global variables **score** and **level**.
- We fill the main window with a white color.
- We then display the message “**YOU WON**” in red.
- Finally, we pause the game for two seconds.

Now that the congratulations message has been displayed, we need to let the player know that s/he needs to press the **R** key to restart the game.

- Please add the following code just after the one that you have typed (new code in bold):

```
time.sleep(2)
while True:
for event in pygame.event.get():
if event.type==pygame.QUIT:
pygame.quit()
main_window.fill('White')
```

```
info_text = info_font.render("PRESS R TO RESTART THE GAME ",1,"RED")
main_window.blit(info_text,(100,270))
pygame.display.update()
```

In the previous code:

- We check whether the player quits the game window, and we then quit Pygame.
- We then fill the main window with white.
- We display the message “**PRESS R TO RESTART THE GAME**" in red.
- And we update the screen.

Next, we just need to detect whether the player has pressed the **R** key. So, please add the following code to the function (new code in bold):

```
Key = pygame.key.get_pressed()
if key[pygame.K_r]:
level = 1
score = 0
change_level(1)
menu()
```

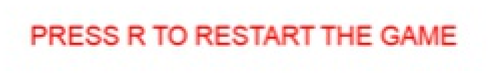
In the previous code, we check whether the **R** key has been pressed; in that case, we set the variables **score** and **level**, and we display the first level.

Last but not least, we just need to call this function when the player has completed both levels.

- Please add the following code at the end of the function **change_level** (new code in bold):

```
main()
elif (level == 3):
victory()
```

You can now save and compile your code; as you go through and complete the two levels, a congratulation message will be displayed, followed by instructions on how to restart the game, as illustrated in the next figure:



PRESS R TO RESTART THE GAME

Adding Animations

At this stage, we have managed to create the game from start to end; however, it would be great to use an animated character instead of a square; so in this section, we will use static images of a character in different positions to create an animation for our main player character.

This will consist in:

- Loading the images that make-up the animation.
- Play the animation as the character is moving or changing direction.
- Rotate them depending on the player character's direction.

First, please make sure that you have copied the folder called **assets** from the **resource** folder to the folder where you have stored your Python scripts as it contains the animations that we will need for our main character.

- Next, please add the following code before the first function in the script adventure.py (new code in bold):

```
info_font=pygame.font.SysFont('arial',30)
animation_frame = 0
player_angle = 0
p_1 = pygame.image.load("assets/p_1.png")
p_1 = pygame.transform.rotate(pygame.transform.scale(p_1,(25,25)),player_angle)
p_2 = pygame.image.load("assets/p_2.png")
p_2 = pygame.transform.rotate(pygame.transform.scale(p_2,(25,25)),player_angle)
p_3 = pygame.image.load("assets/p_3.png")
p_3 = pygame.transform.rotate(pygame.transform.scale(p_3,(25,25)),player_angle)
p_4 = pygame.image.load("assets/p_4.png")
```

```
p_4 = pygame.transform.rotate(pygame.transform.scale(p_4,(25,25)),player_angle)
p_5 = p_1
p_5_r = pygame.transform.rotate(pygame.transform.scale(p_5,(25,25)),player_angle)
```

In the previous code:

- We declare the variables **animation_frame** and **player_angle** that will be used to set the current frame in the animation along with the rotation for the animated image; this is because we will be using the same animation for all directions, and simply rotate this animation based on the direction of the player.
- We then declare the variables that will correspond to each of the images that make up the animation: **p_1**, **p_2**, **p_3**, and **p_4**. Each of these variables (image) is loaded from the **assets** folder and scaled so that its 25-pixel high and wide. Since the value of the variable **player_angle** is **0**, no rotation is applied yet to these images.
- **p_5r** is a temporary image that will store the current image in the animation, and this image will possibly be rotated based on the direction of the player. It is set with the first image in the sequenced animation (i.e., p_1) and rotated using the variable **player_angle**.

So now that we have loaded the images that are needed for the animation, we will create a function that creates the illusion of an animation.

- Please create the following function:

```
def animation():
global p_1,p_2,p_3,p_4,p_5_r,p_5,animation_frame
p_5_r = pygame.transform.rotate(p_5,player_angle)
```

```
animation_frame += 1
if animation_frame == 5:
p_5 = p_1
elif animation_frame == 10:
p_5 = p_2
elif animation_frame == 15:
p_5 = p_3
elif animation_frame == 20:
p_5 = p_4
animation_frame = 0
```

In the previous code:

- We refer to the global variables used for the animation.
- We set the value of the variable **p_5_r**.
- We increase the value of the variable **animation_frame** by 1.
- We then set the value of the variable **p_5** based on the animation frame.

Next, we need to call this function from the main loop; so please add the following code at the end of the function **main** (new code in bold):

```
controls(key,player_rect)
detect_collision(player_rect)
animation()
```

In the previous code, we call the function **animation**; since the function **main** is called from the game loop, we will effectively call the function animation indefinitely, causing the value of the variable **animation_frame** to increase, and the animation to happen.

Now that the call to the function **animation** has been processed, the last thing we need to do is to display the frame for the animation, and to also comment the previous code that was displaying a red square.

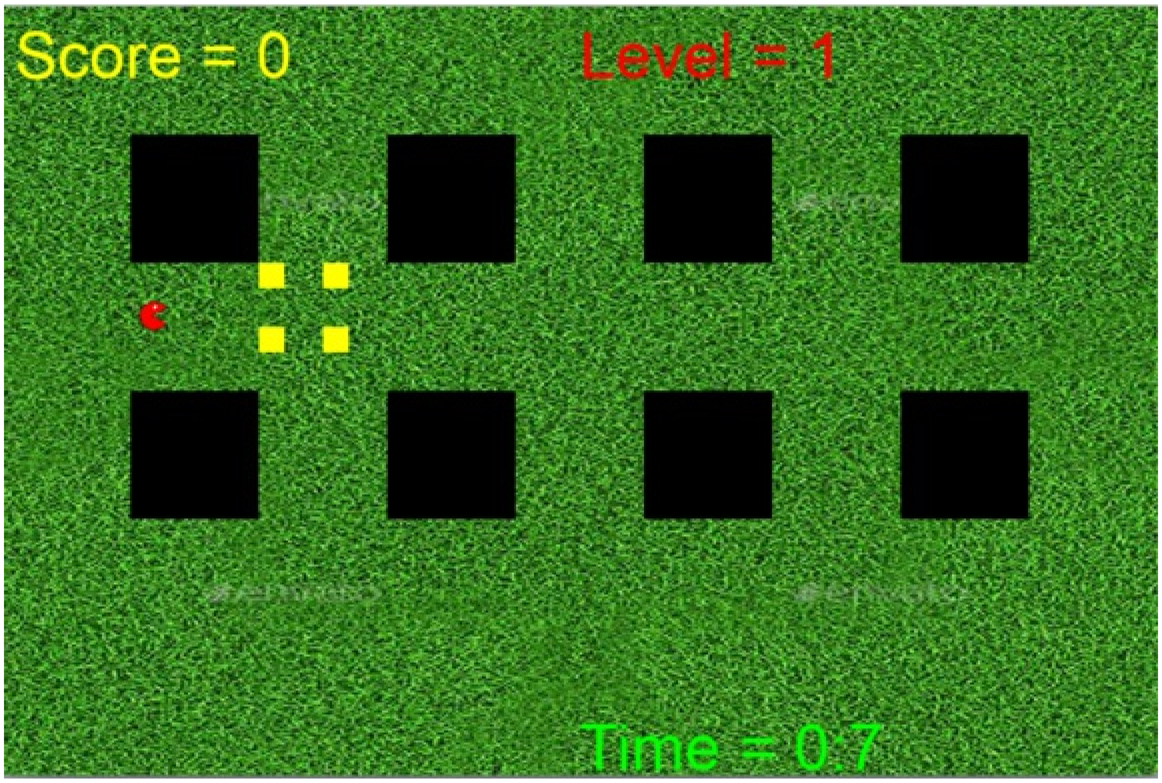
- Please add (or modify) the following code in the function **display_window** (new code in bold):

```
main_window.blit(bg_img,(0,0))
#pygame.draw.rect(main_window,"Red",player_rect)
if (player_angle == 180): p_5_r = pygame.transform.flip(p_5_r, 0,-1)
main_window.blit(p_5_r,(player_rect.x,player_rect.y))
```

In the previous code:

- We comment the code used to display a red square.
- We flip the image vertically if the rotation angle is 180 so that the character’s eyes are still on top.
- Finally, we display the current animation frame on screen.

You can now save and compile your code; as you run the programme, you should see that the player character is now animated, and that the animation is rotated as you move the character right, up or down.



Next, we could also change the sprites for the coins to be collected.

- Please add the following code at the beginning of the script before any function (new cold in bold):

p_5_r=pygame.transform.rotate(pygame.transform.scale(p_5,(25,25)),player_angle)

coin_img=pygame.image.load("assets/coin.png")

coin_img=pygame.transform.scale(coin_img,(25,25))

In the previous code we load the image for the coins, and scale it so that its 25 by 25.

- Please modify the function **display_window** as follows (new code in bold):

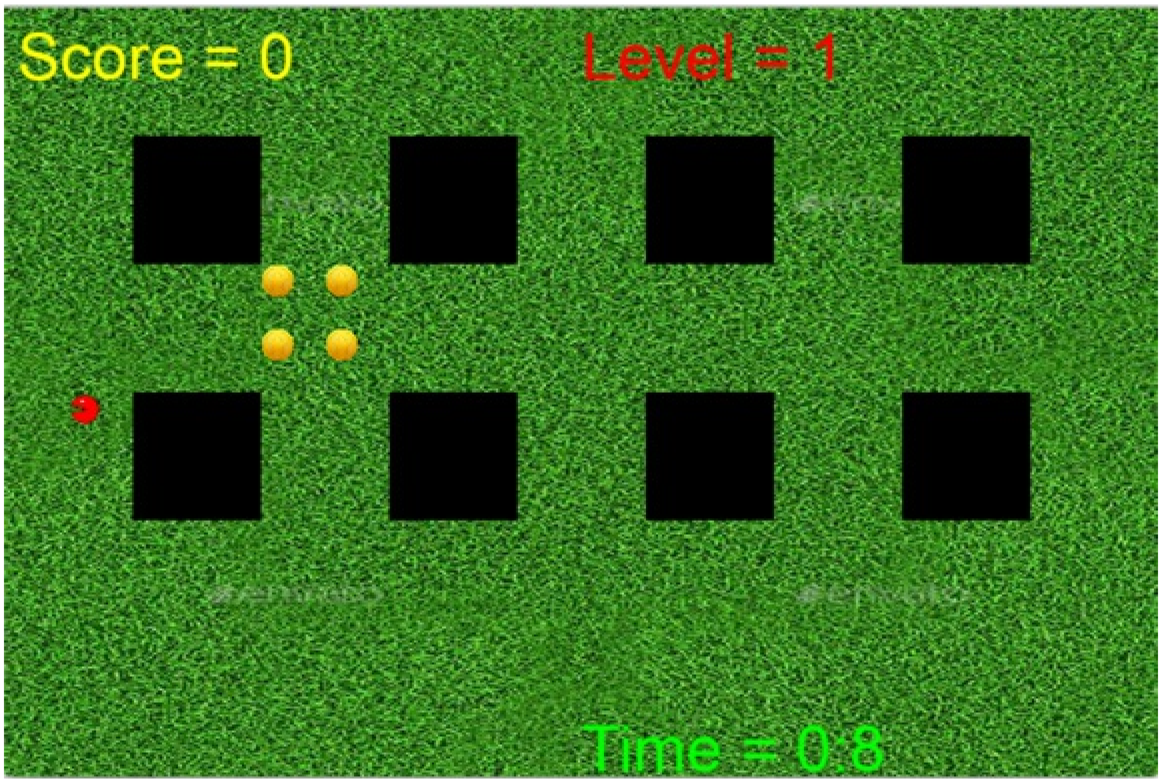
#pygame.draw.rect(main_window,"Yellow",coin1)

main_window.blit(coin_img,(coin1.x,coin1.y))

```
#pygame.draw.rect(main_window,"Yellow",coin2)
main_window.blit(coin_img,(coin2.x,coin2.y))
#pygame.draw.rect(main_window,"Yellow",coin3)
main_window.blit(coin_img,(coin3.x,coin3.y))
#pygame.draw.rect(main_window,"Yellow",coin4)
main_window.blit(coin_img,(coin4.x,coin4.y))
```

In the previous code, we basically stop displaying yellow squares and we replace them with the image of the coin that we have defined earlier.

You can now save and compile your code; as you run the programme, you should see that now all the previous yellow squares have been replaced by the actual image of a coin, as illustrated in the next figure:



Adding a sound effect

In this section, we will just add a sound effect to the game when we collect a coin.

- Please add this code at the start of the script **adventure.py** (new code in bold).

```
pygame.font.init()
pygame.mixer.init()
```

- Please add the following code before any function (new code in bold):

```
coin_img=pygame.transform.scale(coin_img,(25,25))
beeping_snd = pygame.mixer.Sound('assets/beep.wav')
```

In the previous code, we load the beeping sound from the **assets** folder so that it can be played later.

- Please add (or modify) the following code in the function **detect_collision** (new code in bold):

```
if player_rect.colliderect(coin1):
coin1.x=screen_width + 10
score += 1
pygame.mixer.Channel(1).play(beeping_snd)
elif player_rect.colliderect(coin2):
coin2.x=screen_width + 10
score += 1
pygame.mixer.Channel(1).play(beeping_snd)
elif player_rect.colliderect(coin3):
```

```
coin3.x=screen_width + 10
score += 1
pygame.mixer.Channel(1).play(beeping_snd)
elif player_rect.colliderect(coin4):
coin4.x=screen_width + 10
score += 1
pygame.mixer.Channel(1).play(beeping_snd)
```

In the previous code, we lay the beeping sound that we have loaded earlier, every time a coin is collected.

You can now save and compile your code and check that a beeping sound is played every time a coin is collected.

Level roundup

In this chapter, we have managed to create coin-collection game where the player collects coins, moves to new levels, and can eventually win; along the way, we have also looked into displaying images on screen, processing the player’s inputs, and detecting collision.

Checklist

You can consider moving to the next stage if you can do the following:

- Detect the keys pressed by the player.
- Import the Pygame library.
- Display images on screen.
- Change the color of the pen.
- Use loops and conditional statements.

Quiz

Now, let’s check your knowledge! Please answer the following questions. The answers are on the next page.

Please specify whether the following statement are **TRUE** or **FALSE**

1. It is possible to display images on screen with Pygame.
2. It is possible to use and display a GIF image using Pygame.
3. It is possible to set the size of the display area with Pygame.
4. It is possible to specify the caption of the display window with Pygame.
5. The constant pygame.K_LEFT is used for the left arrow key.
6. The function pygame.draw.circle can be used to draw a circle.
7. The function pygame.draw.rectangle can be used to draw a circle.
8. The function pygame.drawfont.SysFon can be used to set a font and its size.
9. The function **render** can be used to create a surface for some text to be displayed on screen
10. This code defines a rectangle for which the top-left corner is located at (100,100)

```
pygame.Rect(100,100,100,100)
```

1.

Solutions to the Quiz

1. **TRUE**.
2. **FALSE.**
3. **TRUE**.
4. **TRUE**.
5. **TRUE.**
6. **TRUE.**
7. **FALSE.**
8. **TRUE.**
9. **TRUE**
10. **TRUE.**

1.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, you could use these to improve the flow of your game. So for this challenge, you will be improving the look and feel of the game.

- Use different font and font sizes to display the score, the time, and the on-screen information
- Import and use wall images for the walls.
- Add tree object symbolized by a tree image and for which the collision is managed through a rectangle.

CHAPTER 7: FREQUENTLY ASKED QUESTIONS

This chapter provides answers to the most frequently asked questions about the features that we have covered in this book.

Scripts

How do I create a script?

Open a new text file in a text editor of your choice and save it with the extension **.py**.

How can I check that my script has no errors?

You can compile your script using the Command Prompt.

What is the dot notation for?

The dot notation refers to **Object-Oriented Programming**. Using dots, you can access properties and functions (or methods) related to a particular object.

Interaction with assets

How do I detect collisions?

To detect collisions for a sprite you need to create a rectangle of the same size of the sprite and then detect collision between this rectangle and the player character using the function **colliderect**.

How can I create a scoring system?

For a simple scoring system, you can create an integer and increase its value by one every time the player has collected an item.

Using a graphical user interface

How do I create a text to be displayed onscreen?

To display text on screen, you will need to create a font, then a surface based on this font that includes the text to display, and then display this surface onscreen.

How do I update a text to be displayed onscreen?

You will need to update the surface used to display the text

How do I detect the players’ choice (when asked to select an option)

You just need to create an infinite loop within which you detect the key that has been pressed, as illustrated in the next code snippet.

```
Key = pygame.key.get_pressed()
if key[pygame.K_r]:
level=1
```

Audio

How do I play a sound?

- Initialize the audio mixer
- Load the sound to be played
- Play the sound

Example:

```
pygame.mixer.init()
...
beeping_snd = pygame.mixer.Sound('assets/beep.wav')
pygame.mixer.Channel(1).play(beeping_snd)
```

Detecting user inputs

How can I detect keystrokes?

You can detect keystrokes by using the function **pygame.key.get_pressed()**. For example, the following code detects when the key **E** is pressed.

```
key=pygame.key.get_pressed()
if key[pygame.K_e]:
```

How can I detect a click on a button?

To detect clicks on a button, you can do the following:

- Detect events.
- Detect whether an corresponds to a button pressed on the mouse.
- Detect which button was pressed.

```
for event in pygame.event.get():
if event.type == MOUSEBUTTONDOWN:
if event.button == 1:
```

THANK YOU

I would like to thank you for completing this book; I trust that you are now comfortable with Python and that you can create interactive 2D game environments. This book is the first in a series of four books on Python and Pygame, so it may be time to move on to the next book for the intermediate level where you will learn more advanced features, including Artificial Intelligence, 2D character animation, and finite-state machines.

The book is currently in the making, and you should be able to access it soon from the official page: **http://www.learntocreategames.com/books/**. If you have subscribed to my mailing list, you should be able to receive a notification.

Please also leave an honest review, this would mean the world to me and it would also help other people to assess whether this book could help them.

So that the book can be constantly improved, I would really appreciate your feedback and hear what you have to say. So, please leave me a helpful review on your e-store letting me know what you thought of the book and also send me an email (**learntocreategames@gmail.com**) with any suggestions you may have. I read and reply to every email. Thanks so much!

[ ]