LEARN PYTHON IN A DAY

THE ULTIMATE CRASH COURSE TO LEARNING THE BASICS OF PYTHON IN NO TIME

ACODEMY

By Acodemy

LEARN PYTHON IN A DAY

The Ultimate Crash Course to Learning the Basics of Python in No Time

Disclaimer

Table of Contents

Chapter 1: Introduction

Chapter objectives:

- You will learn what Python is.
- You will be able to understand why Python is a good programming language to learn and what Python is used for.
- You will learn how to install the latest version of Python
- You will write your first Python program.
- You will understand how to run Python code lines interactively and also how to create Python scripts.
- You will learn how to declare variables in Python and how to assign different data types to them.
- You will be able to perform basic math operations and get the expected output.

What is Python and why learn it?

Python is a high-level programming language that was conceived in the late 1980s and implemented in 1989. Python is widely used for a wide range of applications which are:

- Web and internet development
- Database access
- Creating desktop graphical interfaces
- Scientific and numeric
- Education
- Network programming
- Software and game development

Python is strongly labeled as an extremely readable language, and it often stands out of other programming language due to its distinct syntax which allows us programmers, and prospective programmers to write programs and scripts in very few lines of code. Its readability and efficiency has made Python a very popular language which has reflected to an increase in job demand.

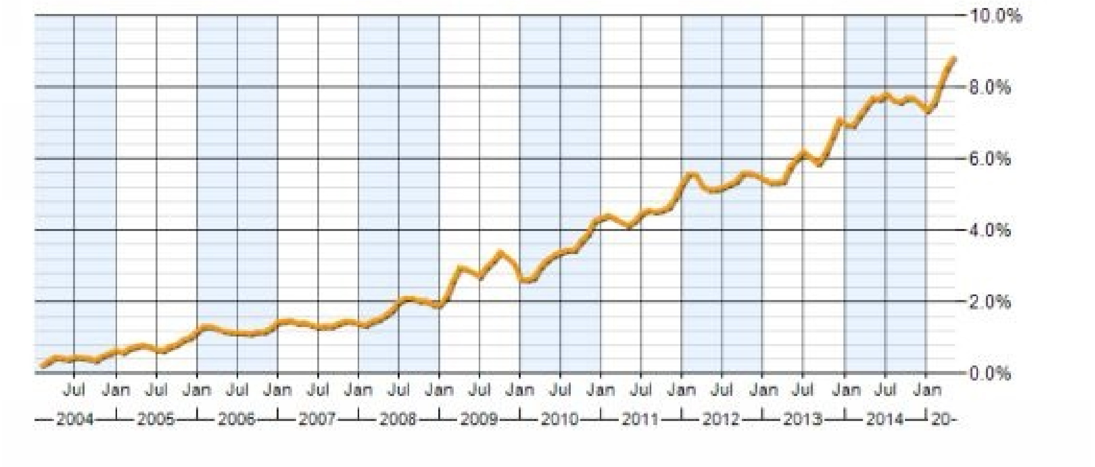


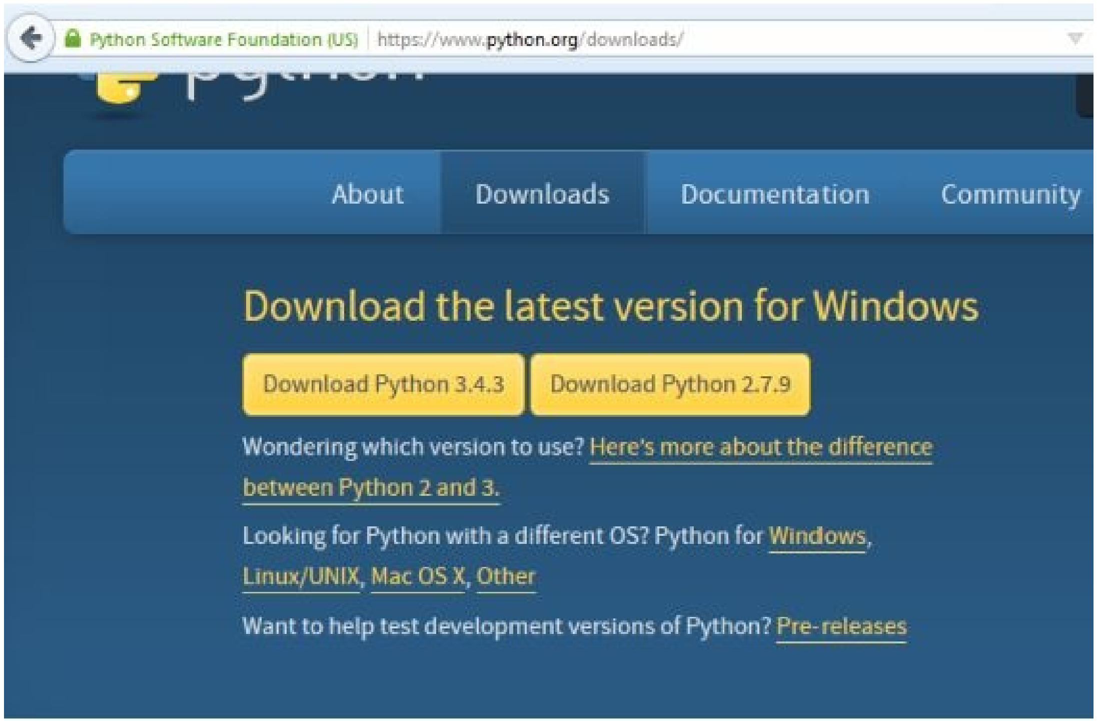
Figure 1: Trend for Python jobs (UK) Python is a pure object-oriented language meaning that the language is designed based on objects. In Python, everything is an object and every object has attributes and methods. Being an object-oriented programming language includes Python in the stack of the modern programming languages that consume most of the programming industry today.

Python is a cross-platform language designed to work the same in every computer operating system. Because Python has become such a popular programming language, it boasts a broad community that supports it. That has led to the development of numerous Python libraries which support different science and technology fields, from bioinformatics, to web scrapping, to earthquakes.

To conclude, we could say that the name Python was inspired from the Monty Python comedy show with the desire to make Python a language that is fun to use. And it was a very successful shot. Writing Python code is very playful. Python is by all means an ideal language for beginners.

Installation

Python is not only easy to learn, but it is also a breeze to install it. The official Python website https://www.python.org/ contains the latest Python installer available for download for various platforms including Windows, Linux and Mac OS X. The latest version of Python at the time this book was written was 3.4.3. You can download your Python installer from the downloading page of the official Python webpage: https://www.python.org/downloads/



Download the latest version for Windows

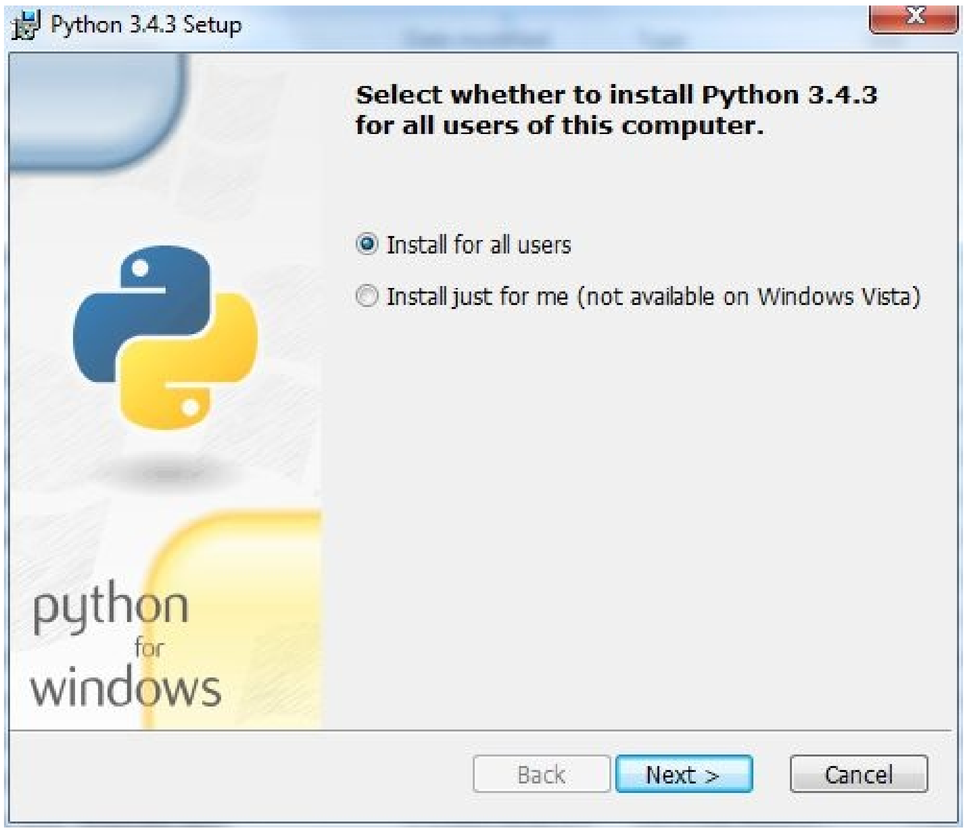
Wondering which version to use? Here's more about the difference between Python 2 and 3.

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Pre-releases

Figure 2: Downloading Python 3.4.3

Once you have downloaded Python 3.4.3, double click it and the installer will guide you through the installation steps.

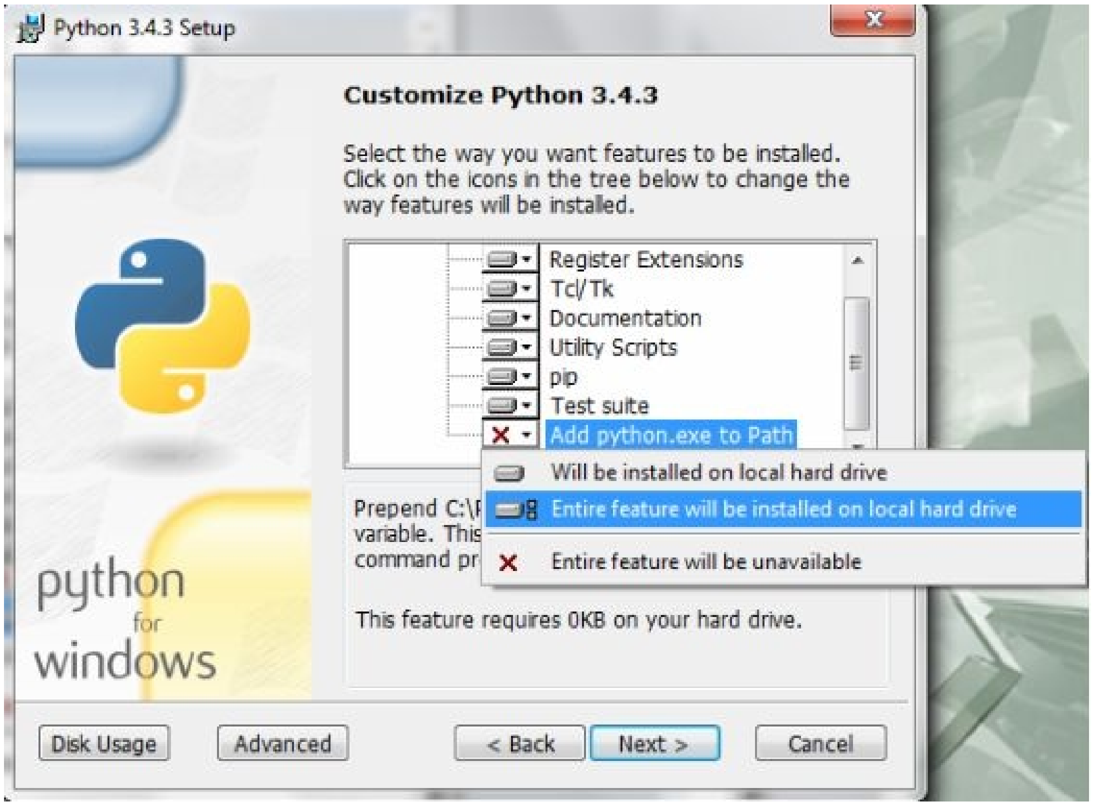

Python 3.4.3 Setup

Select whether to install Python 3.4.3 for all users of this computer.

[✓ Install for all users]

Install just for me (not available on Windows Vista)

Figure 3: Screenshot of the Python installation process A good practice is to enable the option to run Python from your command line. To be able to do that, you should select *Entire feature will be selected on local drive* in the Customize Python 3.4.3 window as showing in the following picture.


Python 3.4.3 Setup

Customize Python 3.4.3

Select the way you want features to be installed.
Click on the icons in the tree below to change the
way features will be installed.

Prepend C:\P
variable. This
command pr

This feature requires 0KB on your hard drive.

Figure 4: Enabling Python to be called from the command line Clicking *Next* should smoothly take you to the end of the installation. Once you have finished the installation, Python should be up and running and ready for your code:

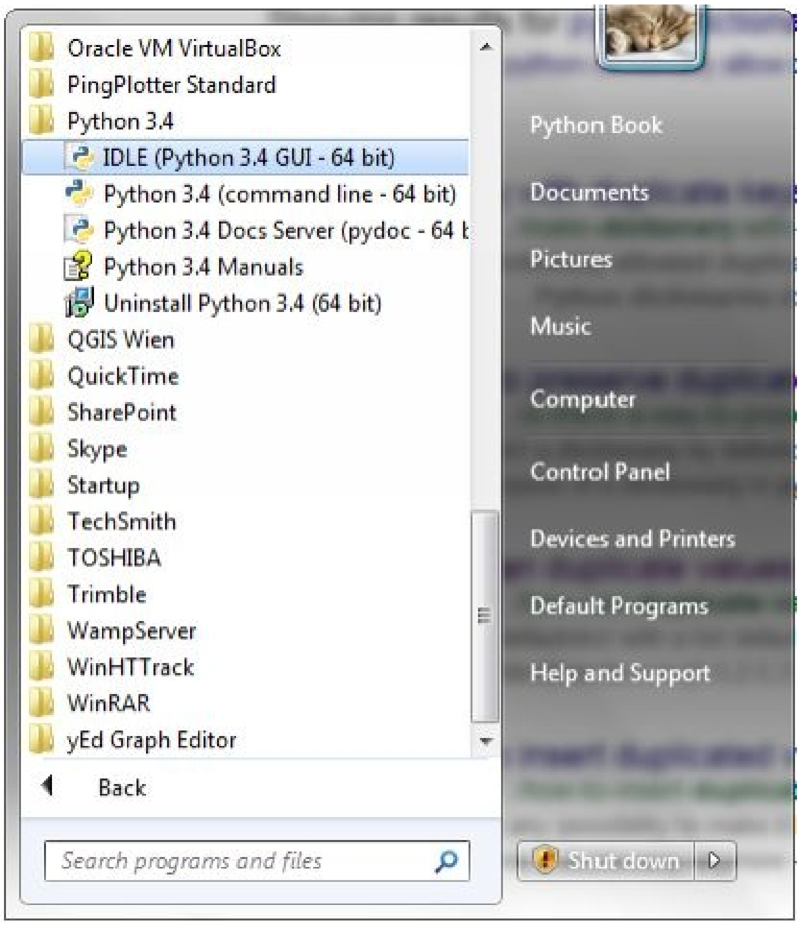
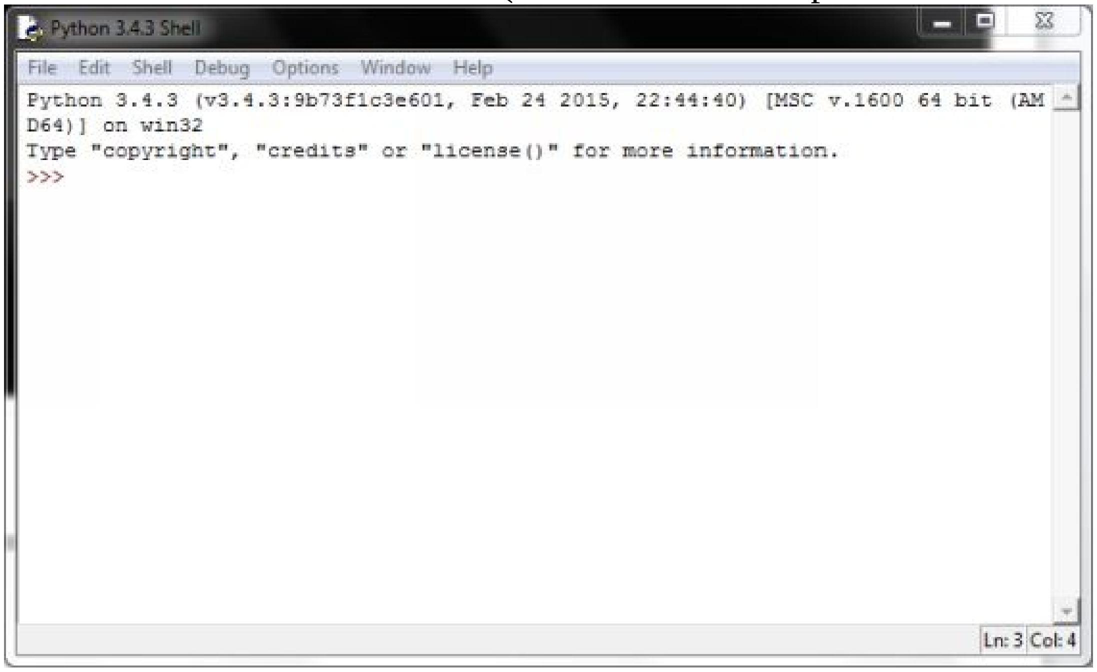


Figure 5: Python should be listed along with your other programs

Your First “Hello World” Program

At this point, we have installed Python in our computer and we are more than ready to start creating our first program. A good practice that will kick start your understanding on how to write a program in Python is to write a very simple and famous program called "Hello World!". What this program will do is simply printing out the "Hello World!" text on the screen.

To start writing the code of our program, let’s go ahead and open up the Python interactive console called IDLE (Interactive DeveLopment Environment):



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Once you open it, all you need to write for this first program is this single line of code: `print("Hello World!")`

Once you write the code, hit *Enter* on your keyboard, and you shall see the output just below the code you wrote: What happened in the background is that the built-in Python *print* function fetched the text that we passed in the brackets, and did what is programmed to do which is printing it out on the screen.

This is the interactive way of writing Python code. In the next section, you will learn another method to write code, and also understand the benefits of each of the methods.

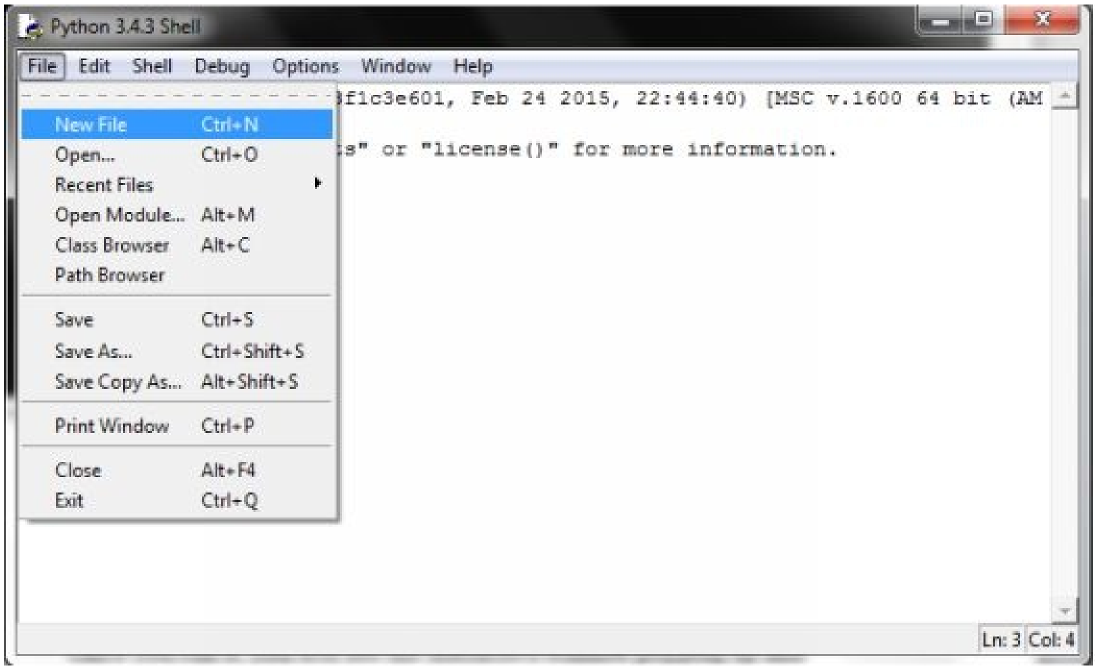
Running Code Interactively VS Running Scripts

The window where we wrote the code for our "Hello World!" program in the previous section is the shell of the IDLE (Interactive DeveLopment Environment) platform. Using the IDLE shell is a great way to learn Python and test things as you write them. You simply need to press *Enter* to execute a line of code. However, when you are ready to start writing a bit more complicated programs that are more demanding in terms of code amount that you write, you will have to write this code inside a more notepad-like environment. This allows you as a programmer to flawlessly write blocks of code and then execute the entire block at once.

Let's say we want to make our "Hello World!" program more "complicated" by adding one more line of code in it. If we would do it the interactive way through the IDLE shell, we would write the first line: `print("Hello World!")`

We would hit *Enter* after that, see the output, and then write the second line:
`print("End of the Program")`

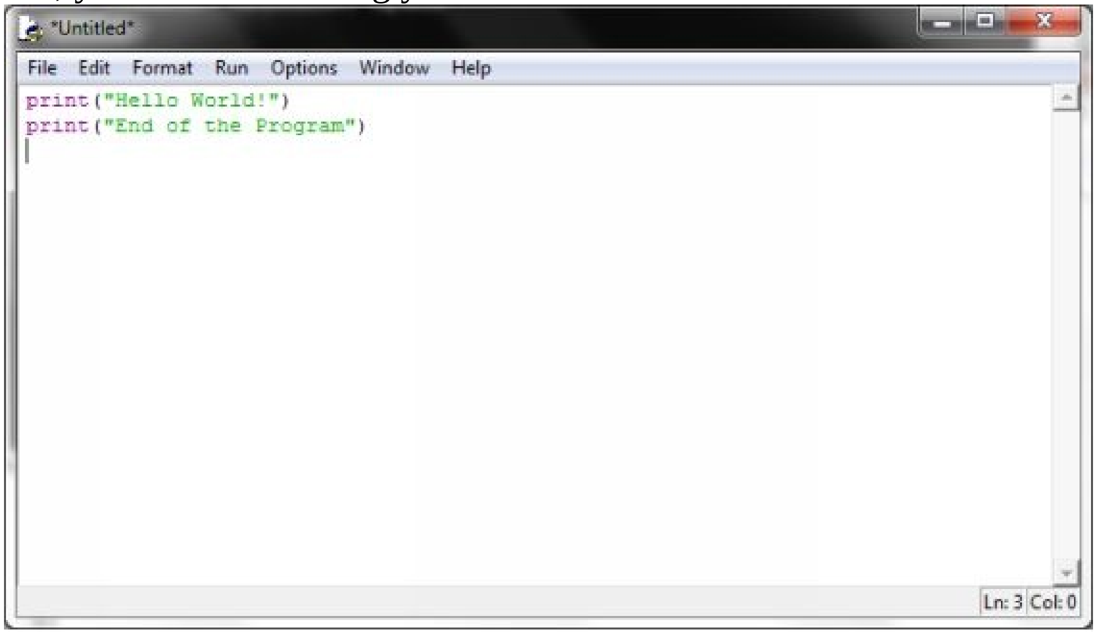
And we would press *Enter* again to execute. As you can see this resembles more to a testing scenario, rather than executing a unified program. If you instead, would like to write the entire code all at once, and then execute all the lines with a single button, you would have to use an editor, and IDLE does offer an editor too. This requires that you create a Python *.py* file where to write your code. You can create such a file from IDLE by going to *File* -> *New File*.



```
3f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
s" or "license()" for more information.
```

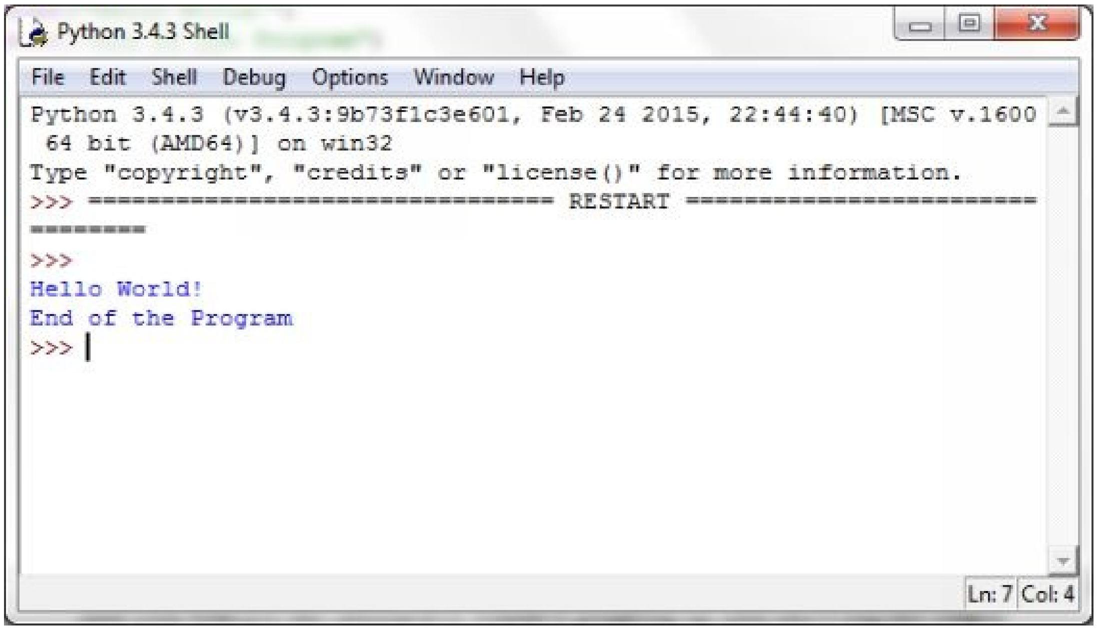
Figure 6: Creating a new Python file from IDLE

This will create a new file under a *.py* extension. Once you have created your file, you can start writing your code in the editor:



```
print("Hello World!")
print("End of the Program")
```

Figure 7: Writing code inside the IDLE editor In the editor, you can write as much code as you want and then execute by going to *Run* -> *Run Module*, or even easier by just pressing the F5 key. If you haven't saved the file yet, you will be asked to save the file first in your computer before Python executes it. Once you do that, the code will run and you will see the interactive window popping up and showing the output of your code which in this case should be something like this:



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600
 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ========================
========
>>> 
Hello World!
End of the Program
>>> |
```

Figure 8: Output displayed in IDLE after running the code from the editor As you can see, the code we wrote printed out two lines of code one after the other.

There are other editors out there that you can use to write your code. However, Python IDLE is a great editor to start. It is simple, light, flexible and it comes with the default installation of Python.

To conclude, we could say that the interactive environment and the editor are both very useful, with the former being a best option for testing while the latter is used whenever you want to write, store and execute blocks of Python code.

Variables and Datatypes

A variable in Python, just like in other programming languages is a storage location symbolized with a name given by the programmer. Variables are designed to contain a value which can change depending on the actions you write in your code.

The value that a variable can hold may be of various data types. To create a variable that holds a *float* datatype, simply type in a name for the variable, use the assignment operator, and then type in the value you want to assign to your variable:
```
version = 3.4
```

Tip: Whether you write version = 3.4 or version=3.4, Python doesn't care. Spaces are not of any importance, unless when used for indentation which is an aspect that you will learn later in the book.

If you want to check the value of your variable, simply pass in the variable name to the printing function:
```
print(version)
```

Tip: When we passed in text to the *print* function, we used quotes. When you pass in a variable or other object that is not text, you don’t use quotes.

The code output you should see is the value that variable *version* currently holds:
```
3.4
```

In case you want another value for the variable, you should reassign a new value:
```
version = "Latest"
```

Of course, if you try to print out the variable now, you shall see the new value as your code output.

Let's now focus on the various data types that you can use in Python. Quite frankly, we already introduced two different datatypes in this section. The first one was a *float*, and the second was a *string*.

A *float* which is short of floating point number is basically a decimal number. The number 3.4 in our example was a *float* datatype.

A *string* is a sequence of characters. Strings are normally used to express text. The word "Latest" in our example above was a string datatype.

These two belong to the basic datatypes available in Python. As you work with Python, you will learn and practice other complicated datatypes. However, for now, we will stick to the simple built-in datatypes. The other datatypes to be mentioned are integers, lists, tuples, sets and dictionaries.

Here is how an integer looks like: `version_year = 2015`

Integers are like whole numbers, but they include negative numbers as well. As you see, there are scenarios like this one above where you will need to use an integer.

In case you want to check the datatype of your variable, simply use the *type* built-in function which will print out the datatype of the value that the variable is holding: `type(version_year)`

And you should get this output: `<type 'int'>`

which indicates that the datatype of the variable *version_year* is an integer.

Strings, integers, and floats are some of the built-in datatypes. Other built-in datatypes that you have to know about are lists, tuples, sets and dictionaries. These are datatypes that are compound datatypes in the sense that they can contain sequences or collections of other datatypes. For example, let's start by defining a list: `python_list = [1989, 3.4, "Latest"]`

The word *python_list* is simply a variable name. What's after the assignment operator is what we refer to as a Python *list*. As you see, a list is made of a sequence of items separated by commas. Each item of the list has its own datatype. For example, the item *1989* is an *integer*, *3.4* is a *float*, and *"Latest"* is a *string*. All of them together enclosed in square brackets make a *list*. Lists are very important and they will follow you everywhere if you decide to make Python your programming weapon in your specific industry. Therefore, we will dedicate a separate section to lists later on in the book.

Let's now jump into *tuples*. A tuple just like a list is a sequence of items. The difference between tuples and lists is that lists are mutable, but tuples are not. Once you have defined a list, you can add, remove or modify its items. The same is not true for tuples. Creating a tuple follows the same concept as with lists, but here you need to use round brackets to let Python understand that you are creating a tuple: python_tuple = (1989, 3.4, "Latest")

Both lists and tuples are indexed sequences meaning that each item is associated to an index starting from zero. This allows an item to be accessed via its index.

Sets are another datatype that resembles lists and tuples. However, unlike lists and tuples, sets are unordered collections of items. That means no indexing is associated to the items of a set. The syntax of creating a set involves the use of curly brackets: python_set = {1989, 3.4, "Latest"}

A unique feature of sets is that they do not allow duplicate items. This can be very useful when you need to handle data that require collections of unique items. Practically, you might have to use a set when you already have a list that has duplicate items such as this one: python = {1989, 3.4, "Latest", "Latest"}

If you would want to have a list that is clean of duplicates, you could convert the list to a set using the *set* function: python = set(python)

Basically, what is happening in this line of code is that the old value of the variable *python*, is changed to the new value generated by the *set* function. Here, the old value is a list, and the new value is a set. We could now print out the new variable value: print(python)

That would give this output:

```
{3.4, 1989, 'Latest'}
```

As you see, this is a set object and there are no duplicate items here anymore. In case you want to retain the datatype of your original object, which is a list, you could convert the set back to a list using the *list* function and print it out to check what you've got: python = list(python)
print(python)

The output you should expect is a list: [3.4, 1989, 'Latest']

And this time we have a list without any duplicate item.

Dictionaries are yet another compound datatype. Just like sets, dictionaries are also unordered collections of items. Dictionaries have a very distinct feature that makes them special. Each item in a dictionary is associated to a custom key. That makes a dictionary a collection of pairs of keys and values. To illustrate it, let's look at the following example:

```
python_dict = {"year":1989, "version":3.4, "generation":"Latest"}
```

The dictionary keys here are *year, version,* and *generation,* and their corresponding values are *1989*, *3.4*, and *Latest*, respectively. Each dictionary value can be later accessed using its corresponding key, but this something that we will consume later on in the book.

Basic Operators

There are a few types of basic operators in Python which you need to know about. They are the *arithmetic operators, comparison operators,* and *assignment operators.*

Arithmetic Operators

Arithmetic operators are the same operators which you have been learning in elementary school and they are addition, subtraction, multiplication, division, modulus, exponent, and floor division. Using these operators in Python is very easy. Let's create two variables here, assign a value to each of them and them use the addition operator to add them up:

```
a = 3
b = 3.5
a + b
```

The output you will get is simply the sum of *a* and *b*:

```
6.5
```

You could even choose to store the result in a third variable, and then simply call that variable to get the output:

```
c = a + b
c
```

And Python will produce the same value again:

```
6.5
```

Because Python is a very straightforward programming language, it is often used as a calculator for daily routine tasks. All you need to do is open the Python IDLE and start typing your mathematical operations without the need to write variables. For example, by just typing in the numbers directly:

```
3 + 3.5
```

Once you press *Enter* you get the expected sum:

```
6.5
```

Similarly, you can use any other math operators. The following table gives a summary of available arithmetic operators that you can use in Python:

Operator symbol	Operator Name	Example	Output
+	Addition	5 + 2	7

-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2.5
%	Modulus	5 % 2	1
**	Exponent	5 ** 2	25
//	Floor Division	5 // 2	2

Comparison Operators

Comparison operators are used to compare two values. Here is a simple example of the *greater than* operator: 5 > 2

And the output you will get is: True

Comparison operators are often used in conditional blocks where a statement is evaluated and different actions are performed depending on whether that statement was evaluated to *True* or *False*.

The following summary gives a complete list of available comparison operators in Python:

Operator symbol	**Operator name**	**Example**	**Output**
==	Equal	5 == 2	False
!=	Not equal	5 != 2	True
>	Greater than	5 > 2	True
<	Less than	5 < 2	False
>=	Greater or equal to	5 >= 2	True

<=	Less than or equal to	5 <= 2	False

Assignment Operators

Lastly, let's focus on assignment operators. While there are a few assignment operators, only one of them is the most used one. And that is the assignment operator "=" which you already know now. The assignment is used to assign a value to a variable: `v = [1,2,3]`

Here we just assign a list to variable *v.*

Tip: The assignment operator = should not be confused with the equal operator ==. The former is used to assign a value to a variable, while the latter is used to check if two values are the same or not.

There are also derivatives to the assignment operator. One of them is the *add and* operator. Here is an example:

```
a = 2
a + = 3
```

And the output would be this:

```
5
```

The previous code is the same as this:

```
a = 2
a = a + 3
```

So, what the += operator is doing is adding up the left hand operand with the right hand operand and assigning the sum to the left hand operand.

The following table provides a full list of assignment operators that you can use in Python:

Operator symbol	**Operator Name**	**Example**	**Output**
=	Assignment	a = 5	5

+=	Addition	a = 5
		a += 2
			7

-=	Subtraction	a = 5	3

		a -= 2	
*=	Multiplication	a = 5 a *= 2	10
/=	Division	a = 5 a /= 2	2.5
%=	Modulus	a = 5 a %= 2	1
**=	Exponent	a = 5 a **= 2	25
//=	Floor Division	a = 5 a //= 2	2

Summary

In this chapter, you were introduced to the Python programming language with the focus on what the language is used for and why it is a good idea to learn it. You also learned how to install Python on your operating system.

Moreover, we presented the two methods of how to write Python code – the interactive shell method and the editor method. You were able to understand when to use each of the two methods to write your Python code.

Further, you wrote your first Python program, once in the Python IDLE shell, and once in the IDLE editor, and learned how easy it is to execute the code and get the output.

You also learned how to create a variable and how to assign different values of different datatypes.

Lastly, we discussed different basic operators that you can use in Python and you were provided with some full lists of Python operators along with examples for each operator.

Assignment

Exercise 1

Various examples of different datatypes are given in the left column of the following table. Write the corresponding datatype in the right column. The first row is an example.

Datatype example	**Datatype name**
1000	Integer
[0,11,22,33]	
{"Name":"John Smith","Age":95,"Profession":"Archer"}	
("Diameter","Radius","Perimeter")	
1000.0	
80.855	
"John Smith"	
{"Temperature","Wind speed",Wind direction"}	

Exercise 2

Look at the Python code on the left column and write down the output you would expect if those lines of code are executed. You can either respond directly, or you can try the code in Python if you are unsure of the output. Again, the first row is just an example.

Operation	**Output**
a = 3	4
print (a + 1)	
a = 3	
a == 3	
a = 3	
a += 10	
print(a)	
a = 3	
a += 3.3	
type(a)	
[1,23,33] == (1,23,33)	
10**2 + 10**2	

a = (1,2,3)
b = list(a)
print(b)	

Solution for Exercise 1

Datatype example	Datatype name
1000	Integer
[0,11,22,33]	List
{"Name":"John Smith","Age":95,"Profession":"Archer"}	Dictionary
("Diameter","Radius","Perimeter")	Tuple
1000.0	Float
80.855	Float
"John Smith"	String
{"Temperature","Wind speed",Wind direction"}	Set

Solution for Exercise 2

Operation	**Output**
a = 3 print (a + 1)	4
a = 3 a == 3	True

a = 3
a += 10
print(a)
	13

a = 3
a += 3.3
type(a)
	<class 'float'>

[1,23,33] == (1,23,33)	False
10**2 + 10**2	200

a = (1,2,3)
b = list(a)
print(b)
	[1,2,3]

Chapter 2: Sequences

Chapter objectives:

- You will be introduced to an extremely useful Python datatype category – sequences, which include lists, tuples and strings.
- You will learn how to extract a particular item from a sequence using indexing.
- You will also learn how to extract a portion from a sequence using index slicing.
- You will learn how to access list and tuple element via indexing.
- You will deepen your knowledge about lists by performing list operations using list methods.
- You will understand the usefulness of strings and how they can be used to handle and format text as you like

What are sequences?

You already learned now the datatypes that belong to the family of sequences. These were lists, tuples, and strings. Again, here is an example of a list: ls = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]

An example of a tuple:

tp = ("Mon","Tue","Wed","Thu","Fri","Sat","Sun")

And a string:

st = "Friday"

As you can see, a sequence is just an array of objects in the case of lists and tuples, or an array of characters in the case of strings. Sequences allow us to store multiple values in one single variable in an organized and easily manageable manner. We say "manageable" because there is a hidden structure rule under all these sequence datatypes. This rule is referred to as *indexing*.

Accessing sequence items through indexing

Let's suppose we want to know what the value of the third element of our list is. To access that element, we need to know its position in the list, and Python offers an index notation that start from zero and increases by one from left to right. Therefore, the hidden indexes of our list would look like this: ls = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]

0 1 2 3 4 5 6

So, if we need to extract the third item of the list, we would write this code: ls[2]

And that would output this string: 'Wed'

And of course you can store the extract in a new variable if you need to use that string for further operations: third_day = ls[2]

Tuples and strings work under the same index notation. Let's access the third items of both the *ts* tuple and the *st* string under a single printing function: print(tp[2] + st[2])

And that would output this:

Wedi

which was generated by the concatenation of the string *Wed* and the string *i*.

Sometimes you may want to extract a portion of a sequence, and not just one single element. That is referred to as splitting and Python has a specific syntax for that too.

Splitting sequences

Accessing a portion of a sequence is also very straightforward. The notation for splitting is this: ls[*start*:*stop*]

The *start* element here is the index of the first item that is to be included in the split, while *stop* is the index of the item where the portion stops, and this item is **not** included in the split. Let's illustrate it with an example, by first reminding you of our *ls* list: ls = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]

```
ls[2:4]
```

This would output a smaller list: ['Wed', 'Thu']

which is a the portion of the *ls* list including only the items with index 2 and 3.

The same works with tuples:

```
tp[2:4]
('Wed', 'Thu')
```

And so does it with strings:

```
st[2:4]
'nd'
```

Negative indexing

We could access the third item of our list: ls = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]

by just writing the index of that item. However, when you want to access an item that is near the end of the list, it can be a bit frustrating to count from left to right. Nothing to worry though – Python has a solution to this problem and that is *negative indexing*. Negative indexing is a second underlying index notation that specifies indexes starting from -1, and decreasing by one from right to left. Here is how negative indexing would look like: ls =

```
["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]
    -7 -6 -5 -4 -3 -2 -1
```

That means, if we want the last item, we'd better use the negative index of -1 than the positive 6: ls[-1]

And that would give us the last string of the list: 'Sun'

Negative indexing works exactly the same with tuples and strings too and it can be very useful with sequences that have many items stored in them.

Moreover, you can use negative indexing for splitting too. To extract the string *Fri* and *Sat*, you would have to write this code: l[-3:-1]

and that would generate this: ['Fri', 'Sat']

And in case you're wondering how to extract the last two items of the list, the answer is this: l[-2:]

which would output this:

['Sat', 'Sun']

So, if you don't pass any index after the colon, the rest of the sequence is included in the extract. This would work basically the same with positive indexing. Let's extract the first two items of our list: l[:2]

And we would get what we want: ['Mon', 'Tue']

This kind of slicing is often referred to as shorthand slicing.

Working with lists

Unlike tuples, lists are mutable. That means we can add, remove or modify items of a list. The way you modify an object in general, in Python is to call the object, use a dot notation, and then call the method that applies to that object, and then pass an argument to that method. Generally the syntax looks like this: object.method(argument)

Let's illustrate this by removing the last item from our ls list and print out the updated list: `ls = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]`

```
ls.remove("Sun")
print(ls)
```

So, to remove an item, we use the list *remove* method. And the output to be expected is this: `['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']`

If you changed your mind about Sunday, let's append the item again to the end of the list using the *append* method: `ls.append("Sun")`

```
print(ls)
```

This will update the list to this again: `['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']`

Sometimes, you might want to check what index a specific item is associated with. For that, we use the *index* method: `ls.index("Wed")`

And that would return:

```
2
```

which is the index of the *Wed* item.

And if you want to append more than one item to your list use the *extend* method: `ls.extend(["Mon","Tue"])`

And that would output this longer list: `['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Mon', 'Tue']`

In case you want to count the occurrence of a specific item of the list, simply use the *count* method: `ls.count("Tue")`

And that would output:

which indicates that there are *Tue* in the list.

And if you want to know the total number of items in a list, use the *len* function which stands for *length:* len(ls)

And that would produce this result: 8

That means the *ls* list has eight items.

And lastly, you can add any item at any position in the list by using the insert method which syntax is as follows: ls.insert(index,item)

And to illustrate it, let's add the string *Sun* at position -2: ls.insert(-2,"Sun") And that would give us this outpu: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'Mon', 'Tue']

These are some of the methods that you really need to know about.

Tip: For a full list of available list methods, simply type **dir(list)** in your Python shell and you will get a complete list of methods that you can apply to a list.

Working with strings

Strings are also sequences, but they differ from lists in the sense that the methods to manipulate strings are somewhat different. You cannot apply some of the list methods to a string, such as *append, remove, insert,* and *expand,* but you can apply the *count* method to count the occurrence of a specific character in the string.

Nevertheless, splitting strings would work the same as list. Let's take the string: file = "File.txt"

Splitting this string would work the same: ext = file[-4:]

which would output the last four characters of the string: '.txt'

A distinct aspect of working with strings is the feature of string formatting.

String formatting

String formatting in Python is all about formatting text the way you want it. Let's suppose a scenario where we a collection of names in a database, and we want to construct some email addresses of a *name@example.com* format out of these names. We would first have to get each name into a Python variable, so let's suppose we have done that and we have our first variable: name = "John"

Now, we need some sort of functionality to generate a full email address string by grabbing the value of variable *name.* And this functionality is provided by the *format* method. Let's see how we would work around this: "{0}@{1}.com".format(name,"example")

So, we have a string inside the double quotes, and we have two placeholders denoted by curly brackets. These two placeholders are waiting to be replaced by whatever is found inside the format method. To understand it better, let's have a look at the output: 'John@example.com'

As you see, the first placeholder was replaced by the value of the variable *name* and the second by the string *example.* String formatting is very important when you iterate through a lot of data and you want to generate a string with a predefined format such as the one above.

Summary

In this chapter you deepened your knowledge on sequences, particularly with lists, tuples, and strings. You learned how to access a particular item or a particular portion of a list, tuple or string. You were able to do such extracting operations using both positive and negative indexing.

Further, you learned how to perform various operations with lists. This involved several methods that can be applied to a list object using the dot notation. You were also able to understand some differences between lists and strings. Specifically, you learned how to do string formatting and how string formatting comes in handy when manipulating text.

Exercises

Exercise 1

Consider this Python list:

li = ["USA","Mexico","Canada"]

Keeping list *li* in mind, write the correct Python code in the right hand side of the table that would generate the output on the right.

Output	**Code**
['USA', 'Mexico', 'Canada', 'Greenland']	li.append("Greenland")
['USA', 'Mexico', 'Canada']	
['Greenland', 'USA', 'Mexico', 'Canada']	
['Greenland', 'USA']	
['Mexico', 'Canada']	
['USA', 'Mexico']	

Exercise 2

Consider this list:

li = ["USA","Mexico","Canada","Greenland"]

With a single line of code that applies to list *li*, try to produce the following result: 'Green'

Hint: Use indexing to access the last item of the list, and just after that, use indexing again to access the first five character of the extracted string.

Exercise 3

Create a new empty *.py* file, and type in the following code in the file: ls = [7, 31, 365]

"A week has {} days, the longest month has {}, while the longest Julian year has {}"

As you see, the second line of the above script is incomplete. Complete it so that when you run the script, it produces the following output: 'A week has 7 days, the longest month has 31, while the longest Julian year has 365'

Solution for Exercise 1

Output	**Code**
['USA', 'Mexico', 'Canada', 'Greenland']	li.append("Greenland")
['USA', 'Mexico', 'Canada']	li.remove("Greenland")
['Greenland', 'USA', 'Mexico', 'Canada']	li.insert(0,"Greenland")
['Greenland', 'USA']	li[:2]
['Mexico', 'Canada']	li[-2]
['USA', 'Mexico']	li[1:3]

Solution for Exercise 2

```
li = ["USA","Mexico","Canada","Greenland"]
li[-1][0:5]
```

Solution for Exercise 3

```
ls = [7, 31, 365]
"A week has {0} days, the longest month has {1}, while the longest Julian year has
{2}".format(ls[0],ls[1],ls[2])
```

Chapter 3: Collections

Chapter objectives:

- You will be introduced to another family of datatypes – collections.
- You will understand how collections differ from sequences and you will also know when to use collections and why they are useful.
- You will be introduced to *sets* and *dictionaries* which are the two datatypes belonging to the collection category.
- You will learn how to declare a set and perform operations over a *set* such as remove, modify and add items to a set
- You will practice the use of sets through a practical example that removes duplicates from an array of elements.
- You will understand how a dictionary is constructed; how it is useful for us, and how we work with dictionaries.

Sets

A *set* is an unordered collection of unique items. Unlike lists, tuples, and strings sets cannot contain duplicate items. The other crucial difference between the set datatype and sequence datatypes such as lists and tuples is that the items of a set are unordered. You were able to access a list item using indexing, but you can't do the same with sets. That's why sets are referred to as unordered collections. To create a set, simply pick a variable name and use curly brackets to enclose the items of the set: `IDs = {"10A","10B","10C","10D"}`

To prove what we just mentioned above about the unordered nature of the items of a set, we could simply print out the set using the *print* function: `print (IDs)`

And that will output this:

```
{'10B', '10A', '10D', '10C'}
```

As you see, the original order of the items has not been maintained. Python does not want to use its resources for keeping the items of the set ordered because keeping the order of the items is not the purpose of a set. The real purpose of a set is to take care of having a collection of unique items. Let's illustrate this by creating another set like the one above, but with one more intentionally duplicate item: `IDs = {"10A","10B","10C","10D","10A"}`

Python will not throw any error during the declaration of a set with duplicate items, but if you check your set, you will see that any duplicate item has automatically dropped out of the set: `print (IDs)`

And this is what to expect:

```
{'10A', '10B', '10D', '10C'}
```

And that is the set we declared, but without duplicates.

If you haven't yet wrapped your mind over the usefulness of sets in a practical situation, that's perfectly normal. Let's make an illustration of the most common situation when a set datatype comes in handy. The scenario involves the use of probably the most used Python datatype – lists.

Let's suppose you have a big array of IDs records stored in a list: identities = ["10A","10B","10C","10D","10E","10A","10C"]

For some reason, you want to make sure that your list is free of duplicate items and the Pythonic way to achieve that is to convert the list object into a set object, and that is something that can be easily done via the *set* function: **Tip:** Don't confuse a *set* datatype with a *set* function. The *set* function is designed to receive an input which could be a list or a tuple, and generate an output which would be a *set* datatype.

```
identities = set (identities)
```

What is happening here is that we are updating the value of the *identities* variable by changing it from the existing list to a new set that is being generated by the *set* function. If we now print out the updated variable: print (identities)

we get a set of unique items: {'10A', '10B', '10C', '10D', '10E'}

So, that is how you get rid of duplicates from a list. You may sometimes want to get rid of the duplicate, but at the same time you may want to retain the datatype of your original object. To retain the list datatype, you simply need to convert the generated set back to a list using the *list* function: identities = list(identities)
print (identities)

And that would output a list: ['10A', '10B', '10C', '10D', '10E']

This example concludes this section and it should have given you quite a good understanding over the use of sets.

Dictionaries

Dictionaries are another type of object that become very useful in many programming situations. You already know that lists, tuples and sequences are indexed by predefined numbered indexes. Dictionaries are indexed by values that are explicitly given by us. Here is an example of a dictionary: `table= {"10A":"John Smith", "10B":"Jack Smith", "10C":"George Smith"}`

10A, *10B* and *10C* are referred to as dictionary *keys*, while the names *John Smith*, *Jack Smith,* and *George Smith* are referred to as dictionary *values*. So, a dictionary is a set of pairs of keys and values. If we make an analogy with sequences, the dictionary keys would be compare to sequence indexes, while the dictionary values would be the sequence items.

Accessing a dictionary value follows the same syntax as accessing an item of a list , tuple, or a string. The difference here is that we refer to the key instead of referring to the index: `table ["10B"]`

And that would return the value associated to the key *10B:* `'Jack Smith'`

This information should be enough for you to understand how a dictionary is structured and how you can extract information from it. We will cover more about dictionaries in the next sections.

Working with Sets

Let's get back to sets now for a more detailed tour on how you can work with them. We already covered a common scenario of using sets in the first section of this chapter. However, there may be other less common situations where you would need to work with sets. Let's suppose we have two sets now, *id1*, and *id2*:

```
id1={"10A","10B","10C"}
id2={"10C","10D","10E"}
```

If you wanted to find out which of the items occurs in both the sets, simply apply the *&* operator to the sets: `id1 & id2`

And that would return the item of the intersection: `{'10C'}`

If you instead would need to know what letters are in *id1* that are not in *id2*, you could simply perform a subtraction between the sets: `id1 – id2`

And that would return this output: `{'10B', '10A'}`

You could even perform a set union: `id1 | id2`

And that would return a set containing the items that are in either *id1* or *id2*:
`{'10C', '10B', '10A', '10D', '10E'}`

Sets support mutations, just like lists do. You can remove, modify or add new items to a set. To add a new item to a set, use the *add* method: `id1.add("10X")`

To remove an item, use the *remove* method: `id1.remove("10X")`

These are the main methods that will help you start off using sets during your programming activities. For a full list of methods that you can always apply the *dir* method to the *set* object: `dir (set)`

Running this line in the Python IDE will print out a list of methods that you can apply to a set object.

Working with Dictionaries

We already talked about how to create a dictionary, how a dictionary is constructed, and how you can access dictionary values. In this section we will be consuming more dictionary operations such as adding a new key-value pair to a dictionary, removing an existing pair. Let's take this dictionary as a sample: table= {"10A":"John Smith", "10B":"Jack Smith", "10C":"George Smith"}

If you were to add another pair here, you can simply do this: table["10D"]="Gunnar Smith"

And that will add the *10D* key and its associated *Gunnar Smith* value to the dictionary *table*: {'10A': 'John Smith', '10B': 'Jack Smith', '10C': 'George Smith', '10D': 'Gunnar Smith'}

If you changed your mind about Gunnar, simply go ahead and expel it from your dictionary using the *del* keyword: del table["10D"]

And that will remove the entire pair: {'10A': 'John Smith', '10B': 'Jack Smith', '10C': 'George Smith'}

Sometimes, you might need to extract only the keys or the only the values of a dictionary. Python provides the *keys*, and the *values* methods to achieve just that. To extract the keys: table.keys()

And that will produce an output similar to this: dict_keys(['10A', '10B', '10C'])

In case you need the values:

```
table.values()
```

And that will output this:

```
dict_values(['John Smith', 'Jack Smith', 'George Smith'])
```

Python dictionaries are very useful for lookup programs. A simple use of a Python dictionary could be the case of creating a language vocabulary program where the user would enter a word in the program and expect to get the meaning associated to that word. You could get the vocabulary data from your data source which could be a database for example, and then use a dictionary to store the data. Then the program would wait for the user to input a word and get the associated meaning in the dictionary for that key word. So, the words would be stored as dictionary keys and the word meanings as dictionary values.

Likewise, there exist more and more examples, but you will understand them better as you learn other concepts such as conditionals and looping throughout the book.

Summary

In this chapter you were introduced to two other datatypes that are categorized as collections in Python. These were sets and dictionaries. You learned that sets are designed to hold a collection of unordered unique items and you were able to understand when sets would be proper for you to use. Moreover, you learned the main processes involved with sets which included various intersecting and union operations.

In this chapter, you also learned how to create a dictionary. You also learned the essence of the dictionary structure and how you can extract information from such a data structure. You were introduced to main operations you can apply to a dictionary, such as adding or removing dictionary values and even extracting all the keys and all the values out of a dictionary. Moreover, we explained a use case of dictionaries so that you could have a better idea of how datatypes become useful when building real-life programs.

Assignment

Exercise 1

Consider this dictionary:

```
company = departments = {"Human Resources":3, "Sales":5, "R&D":4}
```

Write a script that will output these two lists: ['Sales', 'R&D', 'Human Resources']
[5, 4, 3]

Hint: Consider using the dictionary *keys* and *values* method and the *list* function.

Exercise 2

Look at the short scripts in the left column of the table, and write the expected output on the right.

Code	**Output**

s = {3,4,5,6,7,5}
l = list(s)
l.count(5)	

d = {"Mon":20, "Tue":25, "Thu":30}
l = list(d.values())
d[""Wed"] = 40
sum = l[0]+l[1]+l[2]+l[3]	

a = [0,1,2,3,4,5,6,7,8,9,10]
b = [0,5,10,15,20,25]
a = set(a)
b = set(b)
list(a & b)	

d = {0:10,1:20,2:20,3:30} d[0]+d[3]	

Solution for Exercise 1

```
departments = {"Human Resources":3, "Sales":5, "R&D":4}
keys = list(departments.keys())
values = list(departments.values()) print(keys)
print(values)
```

Solution for Exercise 2

Code	Output

s = {3,4,5,6,7,5}
l = list(s)
l.count(5)
	1

d = {"Mon":20, "Tue":25, "Thu":30}
l = list(d.values())
d[""Wed"] = 40
sum = l[0]+l[1]+l[2]+l[3]
	An error will be produced because list l contains only three items, not four.

a = (0,1,2,3,4,5,6,7,8,9,10)
b = (0,5,10,15,20,25)
a = set(a)
b = set(b)
list(a & b)
	[0, 10, 5]

d = {0:10,1:20,2:20,3:30} d[0]+d[3]	40

Chapter 4: Conditionals

Chapter objectives:

- You will learn how to write programs using the power of conditional statements.
- You will learn how to make your program executes an action depending on whether a condition is true or not via an *if* statement.
- You will further advance your knowledge about conditionals by integrating an *else* statement in your script which will help your script execute an alternative actions.
- You will be introduced to a dynamic aspect of programming that has to do with the user interaction by learning how to receive and handle user input
- You will learn how to write conditional blocks with multiple conditions using *elif* statements.
- You will also learn a shortcut to *if-else* statements that is *inline if*.

If Statements

Now that you have learned the main datatypes and the available methods and operations that can be applied to them, you are more than ready to start writing some more complex code which will help you enter the world of programming and give you the proper understanding of how powerful programming is and what can you do with it. Conditionals are a crucial point of this transition from the abstracts to the real world and you will be using them over and over again during your programming activities.

Conditionals are blocks of statements that perform certain actions depending on whether a specified condition evaluates to true or false. There are a few constructs that are used to build up a conditional statement. One of them is the *if* construct. Let's go ahead and look at an example to get a better idea:

```
if 2>1:
    print ("Sure, 2 is greater than 1")
```

What the above conditional block is doing is checking whether number two is greater than one, and if that is true, it prints out a string. Two is greater than one here, so the string *Sure, 2 is greater than 1* will be printed out:

```
Sure, 2 is greater than 1
```

In case we had this conditional block:

```
if 1>2:
    print ("Sure, 2 is greater than 1")
```

Nothing would happen after executing this block because one is not greater than two.

Let's look closer now at the syntax of an *if* statement block. As you see, the block is quite readable and that's a distinct characteristic of Python. Python minimizes the use of brackets, or semi-colons for the sake of readability. All you need to write is the *if* construct and the conditional statement after that without putting any brackets. The colon after the statement denotes that the conditional line of the block has been written, and you are ready now to write the action to be performed in the next line.

Notice here that we have indented the print statement with four white spaces. Indentation is a Python syntax feature used to denote that the indented block belongs to the unindented line above it. The *print* function here belong to the *if* statement and it will be executed depending on that statement. Look at this other example:

```
if 2>1:
    print (2-1)
    print ("Yes, 2 is greater than 1")
```

Here we checked whether two is greater than one, and executed two lines of code. Both the lines were indented meaning that they were both depended on the unindented line above them, so they will be both printed out:

```
1
Yes, 2 is greater than 1
```

So, what happens if we don't indent a line under the *if* block? Let's try it out:

```
if
1>2:
    print (2-1)
    print ("Yes, 2 is greater than 1") print ("This line will be always printed regardless of the condition")
```

Here we have a condition that is evaluated to false meaning that the **indented** lines below the condition will not be executed. However, any line that is **unindented** will not belong to the condition and thus it will be printed out regardless of what happens inside the conditional block. Here is the output of the block:

```
This line will be always printed regardless of the condition
```

Tip: When writing code in the interactive IDE, the console will automatically indent the line for you when it detects that you are writing a code block. If you are writing longer blocks such as the one above, it is advisable to write it in the editor as a standalone script as that will give you full control over the indentation.

An issue that should be mentioned is the amount of indentation that should be applied before a line of code. In the examples above we used four white spaces, but that number is not obligatory as far it is consistent through all the indented lines. The strict rule is to to use at least one white space for indentation. However, a good practice is to use four spaces as that makes the code more readable and aesthetic.

If-else Statements

In the case of the *if* statement, the action under *if* was executed when the condition was evaluated to true. If the condition was evaluated to false, nothing would happen. An *if-else* statement is a sophistication of the *if* statement in that it provides a ground for executing an alternative action in case the condition evaluates to false. Here is an example: age = 13

```
if age >= 17:
    print ("Qualifies for a driver's license") else:
    print ("Under the legal age")
```

What we are doing here is we are first assigning the value of 20 to variable *age*, and then we write a conditional block starting with the *if* construct and passing the condition of whether the given variable is greater or equal to 18. If it is, the string in the next line is printed out, otherwise the string under the else line will be printed.

Because number 13 is not greater or equal to 17, we will get this output: Under the legal age

This should give you a good understanding of the *if-else* block and its difference from the *if* block. The next section will introduce you to a dynamic aspect of Python and programming and that has to do with the user interaction. More specifically, we will be incorporating the *if-else* statement with the input that the user types in to the program that we will make.

Receiving user input

The purpose of a program is often to get some input from the user and produce an output. As programmers, we are responsible of handling this user input, writing the algorithm that calculates what the program is intended to do, and then return the output to the user. We will be doing such a program in this section by incorporating the *if-else* block.

We will ask the user about their age, check whether their age qualifies for applying for a driver's license and print out an answer for the user. To create such a program, we need to write a script, and not just type in lines of code in the interactive IDE.

The way we receive user input is actually quite simple. All you need to do is use the *input* function. Here is how we use the *input* method combined with an *if-else* block:

```
age = input("Enter your age: ") if age >= 17:
    print ("Qualifies for a driver's license") else:
    print ("Under the legal age")
```

What the input method does is prompting the user to enter a value, and then it stores that value into a variable so that we can use it for whatever we want. And here we are using it in a comparison statement that serves as the condition for the *if-else* block.

Warning: The *input* function reads the given input as a string datatype even when the user enters a number. Therefore, in our example above, Python will try to compare the given string with the number 18 and it will fail doing so because a string cannot be compared to a number. However, there is a solution to that.

As the warning above suggests, trying to execute the code above will throw an error similar to this:

```
Traceback (most recent call last): TypeError: unorderable types: str() >= int()
```

What we need to do is simply convert the string into a *float* datatype which Python treats it as a real number that can handle mathematical operations. So, let's update our code to fix that error:

```
age = float(input("Enter your age: ")) if age >= 17:
    print ("Qualifies for a driver's license") else:
    print ("Under the legal age")
```

As you see, all we needed to add was a *float* function which gets an input and converts it to a *float* datatype. In this case the input for the *float* function is the string that the user will input when the program is executed. Running this program will produce first output this:

```
Enter your age:
```

Once the user types in a number, another output will be generated. If for instance we pass 20, we will get: `Qualifies for a driver's license`

That was a good introduction on how to get user input and use it in your script.

Elif Statements

You already know now that an *if* statement can handle one condition, while an *if-else* statement can handle two. But how do we go about if we had to handle more than two conditions?

Let's take as an example our driver's license program. We know that in some states you are allowed to drive if you are at least 14, and in some others you should be at least 17. This functionality can be easily applied through *elif* statements:

```
age = float(input("Enter your age: ")) if age < 14:
    print ("You're under 14 - too young for a license") elif age >= 14 and age < 17: print("Subject of your
state of residence") elif age >= 17 and age < 18: print ("Qualifies for a driver's license in US") else:
    print ("Qualifies for a driver's license in every country")
```

What this script is doing is prompting the user to enter their age. Then it goes to every condition one by one starting with the *if* statement which prints out a string if the entered number is less than 14. Then the first *elif* statement checks whether the entered number falls between 14 and 17, excluding 17, and executes the printing function below.

Tip: The *and* operator is used to evaluate if two expressions are true at the same time. In our example, if age is greater or equal to 14, **and** at the same time less than 17, the entire line evaluates to true.

The second checks whether the number falls between 17 and 18, excluding 18. The last line is an *else* statement that will execute the line below it in case none of the conditions above evaluates to true. In this example, the else statement will be executed if the user inputs a number that is not less than 18, or in other words, equal or greater than 18.

You are free to use as many *elif* statements as you need in your scripts, and you are not obliged to use an *else* statement at the end of the conditional block. In our example above we could have used another *elif* statement instead of *else* that would check if the given number was greater or equal than 18, but using *else* makes the code more elegant.

Inline if

Sometimes, you might want to quickly perform brief conditionals in Python that assign a value to a variable depending on a condition. Python offers a very condensed syntax for such operations, and that syntax is referred to as *inline if*. Here is an example:

```
age = 21
license = "Yes" if age > 18 else "No"
print (license)
```

And the output of this would be:

```
Yes
```

You could read the second line of the code like this: The *license* variable will be assigned *Yes* if *age* is greater than 18, otherwise, it will be assigned *No.* In our case, *license* was assigned to *Yes,* because *age* was greater than 21. While *inline if* is not an urgent syntax to learn, it can be useful later on when you write short functions that require brief conditionals within.

Summary

In this chapter you were able to write much more advanced programs than before by using the power of conditional statements. Specifically, you learned how to use an *if* statement for evaluating a condition and executing an action depending on whether the evaluation returned to true or false.

Furthermore, you learned how to perform an alternative action when the given condition evaluated to false using the *if-else* construct.

Moreover, in this chapter you were introduced to a crucial aspect of programming that is user interaction. You learned how to receive user input using the *input* function, and how to use that input in your script.

You were also able to write conditional blocks that contained multiple conditions using the *elif* construct. Lastly, you also learned how to write *inline if* conditional expressions which are very practical when applying brief conditions while writing short functions.

Assignment

Exercise 1

Consider this variable:

```
a = "3"
```

Write a program that checks whether the variable is an integer datatype, and convert it to an integer if it is not. Then print out the value of the variable.

Hint: In the conditional line of your code, consider comparing the type of variable *a* against the string 'int'.

Exercise 2

Write a program that prompts the user to enter their email address. Then check whether this is a *gmail, hotmail, yahoo,* or other user.

If the user is a *gmail* user, print out the message *This is a gmail user.* Follow the same concept for the other email types of email accounts. If the user is neither *gmail, hotmail,* nor *yahoo,* print out *This is another user.*

Hint: To check whether a string is part of another string, use this syntax: "apple" in "apple juice"

Solution for Exercise 1

```
a = "3"
if type(a) != 'int':
    a = int(a)
print(a)
```

Solution for Exercise 2

```
email = input("Enter your email address: ") if "gmail" in email:
    print("This is a gmail user") elif "hotmail" in email: print("This is a hotmail user") elif "yahoo" in email:
    print("This is a yahoo user") else:
    print("This is another user")
```

Chapter 5: Loops

Chapter objectives:

- You will be introduced to the concept of looping in Python and its importance in programming.
- You will learn how to repeatedly execute an action for a predefined number of times using the *for-loop*.
- You will again learn how to repeatedly execute an action as far as a given condition remains true and you will do that using the *while-loop*.
- You will learn a quick way of generating lists using a single line expression via *list comprehensions*.

What are loops?

A *loop* in programming is a block of code that is able to execute the same action multiple times. The process of executing the same action many times is called *looping*. Looping is encountered in almost every programming language because that’s the way you to tell your program to do something over and over again without you having to type that “something” in multiple lines . Looping in Python can be implemented via two different looping techniques. These two techniques are the *for-loop* and the *while-loop*.

The *for-loop* is designed to repeat the execution of a piece of code for a specified number of times. On the other hand, the *while-loop* is again used to repeat the execution of a piece of code until a specified condition is met. If these theoretical descriptions of looping still don’t make much sense, then let’s carry on to the next section where we will be going through each of the two loop techniques and use them in some examples so that you get the your head around them.

For-loop

Given the knowledge you have gained so far by reading this book, if I ask you to write a script that would print each item of the list in a separate line, you would probably write something similar to this:

```
ls = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]
print (ls[0])
print (ls[1])
print (ls[2])
print (ls[3])
print (ls[4])
print (ls[5])
print (ls[6])
```

This would be a correct solution because it would print out each item in a separate line, but a very immature one because it lacks efficiency, and efficiency is a crucial aspect of programming. The best solution to the problem would be this:

```
for item in ls:
    print (item)
```

This is what we refer to as a *for-loop.* A for loop always starts with the *for* keyword followed by the loop variable which can have any name that you want; then comes the keyword *in* and the object that is to be iterated. The next line after the declaration of the loop is always indented. A good practice is to indent four white spaces.

In fewer words, if we could translate the loop into human language, we could say: *for each item in the list, print out the item.* And Python will do just that, it will access each item of the list and it will perform the action written in the second line. That’s how a *for-loop* works.

Tip: Most of the time you will be iterating through lists, just as we did in the previous example. You can also iterate through other objects such as tuples, strings, sets, and dictionaries, but you cannot iterate through basic datatypes such as integers. Iterating through an integer would throw an error letting you know that an integer is not iterable.

There can be times when you would want to iterate through a list of progressing numbers, but you could find it cumbersome to type a long list of items. Anyway, Python offers a technique to generate such a list through the *range* function. The range function creates a list-like range object given a start, stop, and a step. For example:

```
for item in range(10,20,2):
    print (item)
```

This loop would produce this result:

```
10
12
14
16
18
```

And those are the numbers contained in the range that we created and printed out from our *for-loop.* As you can see, the range generated an array of numbers starting at 10, increasing by two, and stopping at 20, excluding 20.

If you want a default step of one, you don’t have to explicitly declare the step:

```
for item in range(10,15):
    print (item)
```

And here is the output:

```
10
11
12
13
14
```

Again, we got an array of numbers starting at 10, increasing by one, and stopping at 15.

Notice that a range function generates a range object, which even though behaves almost like a list, it is still not a list object. To convert a range object to a list object, use the list function:

```
lst = list (range (10,15) )
```

Printing the *lst* list will produce this output:

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

You can perform any action that you want inside the loop. You can even nest an *if* statement:

```
ls = ["1.txt","2.txt","3.csv","4.csv","5.csv"]
for item in ls:
    if "csv" in item: print(item)
```

What we are doing here is we are iterating through each string of the list *ls*, and then checking whether the current string contains the part *csv*. If it does, the entire string is printed out. As you see here we have two levels of indentation, one under the *for-loop*, and another under the *if* statement. That means, the *if* statement depends on the for loop, thus it has four spaces, while the *print* function depends on both the *if* statement, but also on the *for-loop*, thus it has eight spaces.

This should be enough to help you wrap your mind around a *for-loop* and understand how it works. In the next section, you will be introduced to the other loop that is used in Python – the *while-loop.*

While-loop

The *while-loop* is another looping technique which even though it is not used as much as the *for-loop*, it is still a powerful looping technique that comes in handy in certain situations.

A *while-loop* is used to execute an action until a given condition is met. To illustrate this, let’s have a look at the following example:

```
start = 0
while start < 4:
    print (start, " is less than 4") start = start + 1
```

We are starting here with the *start* variable which has an initial value of zero. Our intention is to print out a string while the *start* variable value is still less than four. Notice that the variable will always be less than four because the we have included a counter in the last line which increments the value of our variable by one each time an iteration occurs.

The output of the code would be this:

```
(0, ' is less than 4')
(1, ' is less than 4')
(2, ' is less than 4')
(3, ' is less than 4')
```

So, in each iteration of the loop, Python is checking whether the variable *start* continuous to be less than four, and prints out the value of *start* and the string *is less than 4* and just after that it adds one to the value *start* and it does everything again until *start* reaches a value that is higher than four.

Let’s now look at a real world example that uses a while loop to check whether the user is entering the correct password or not. As we are mentioning the word “user”, that means we will be using the *input* function to receive the user password and then use that password string in our *while-loop*. Here is the entire script:

```
passwd = input("Enter the password: ") while passwd != "book":
    print ("Wrong password: ") passwd = input("Enter your password: ") print ("You have successfully
logged in")
```

Immediately here we are asking the user to enter their password, and then we store the entered password in the *passwd* variable. Then we check the value of the *passwd* variable against the string *book* which is the real password. While the value of *passwd* is not equal to the value *book*, we print out the string *Wrong password* and prompt the user to enter their password again. The user enters the password again, and the loop repeats the same actions. You should be able to understand now that the *while* loop will run as long as the user keeps entering the wrong password, and it will stop only when the user enters the string *book*. As soon as this happens, the loop ends and the next line outside the loop is going to be executed. In this case this line is the printing function that will print out the string *You have successfully logged in*. That is quite a classic use case of the *while-loop* that should have given you a very good understanding of the loop structure and its usefulness.

List Comprehensions

List comprehensions are a short and efficient way of creating lists in Python. They are often referred to as a quick substitution of *for-loops* as far as list production is concerned. Let’s take an example where we suppose we have a list of strings: `strings = ["10", "20", "30"]`

And we want to convert this list of string to a list of integers maintaining the values of the list *strings*. For that, we would need to create an empty list: `integers = []`

And then we would iterate through each item of *strings*, convert it from *str* to *int* and append the converted value to *integers*:

```
for i in strings:
    integers.append(int(i))
```

These two lines of code would populate our *integers* list with integer datatypes. So if you print out *integers*, you will get this output: `[10, 20, 30]`

That’s one good way to convert the a list of strings to a list of integers. However, if you’re looking for a more elegant way, you could choose to use a list comprehension, just like below: `integers = [float(i) for i in strings]`

This would produce the same result: `[10, 20, 30]`

Similarly, you can use list comprehension when you want to generate a brand new list quickly. For example: `[a**2 for a in range(10)]`

This code would produce this result: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

And that is a list which items are the squares of the items contained in the range starting at one and ending at nine.

That is all about *list comprehensions*. With some practice you will be able to embed the syntax of list comprehensions in your mind and make use of it every time you need to quickly generate a list of items.

Summary

In this chapter you were able to understand what loops are and how they work. You learned that a loop can be used to repeatedly execute an action multiple times, and you learned how to do this using both a *for-loop* and a *while-loop*.

You learned that a *for-loop* is a loop that can execute an action or a series of actions for a predefined number of times. You were able to understand that iterating through lists is a very common task that can be implemented via the *for-loop*.

You were also introduced to the *while-loop* and you were able to understand that such a loop is very useful when you want to repeatedly execute an action while a condition is true. You were able to practice this by creating a program that checked whether the password entered by the user was the correct one or not.

Furthermore, you learned a shortcut to the *for-loop* that is used to quickly create lists via a single line of code.

Assignment

Exercise 1

Consider the following list:

```
ls = ["Mon","Tues","Wednes","Thurs","Fri","Satur","Sun"]
```

Write a script that uses the *ls* list to generate this output: Monday

```
Tuesday
Wednesday
Thursday
Friday
```

Notice that the Saturday and Sunday have not been printed out.

Hint: Consider iterating to a split list, instead of the entire list, and remember that we can concatenate strings using the *add* operator.

Exercise 2

Let's suppose the 1st of January for year *x* is Monday. Count how many Mondays are there in a year, given that year *x* has 365 days.

Hint: Consider using a *range* and a *len* function. Look up chapter 2 in case you need to refresh your knowledge about the *len* function.

Exercise 3

Consider this list of names:

```
names = ["johnsmith","jacksmith","georgesmith"]
```

Write a program that generates this list: ['johnsmith@gmail.com', 'jacksmith@gmail.com', 'georgesmith@gmail.com']

Solution for Exercise 1

```
ls = ["Mon","Tues","Wednes","Thurs","Fri","Satur","Sun"]
for item in ls[:5]:
    print (item+"day")
```

Solution for Exercise 2

```
x = range(1, 365+1, 7)
x = list (x)
len(x)
```

Solution for Exercise 3

```
names = ["johnsmith","jacksmith","georgesmith"]
emails = [name + "@gmail.com" for name in names]
```

Chapter 6: Custom Functions

Chapter objectives:

- You will know how to code more efficiently by learning how to define and execute custom functions.
- You learn how to define the input parameters of a function, how to write the function algorithm, and how to return the calculated output.
- You will make use of the output generated by custom functions and use it in your code for further operations.
- You will learn the different types of parameters and understand the usefulness of having default parameters in a function.

What are custom functions?

We have worked with quite a few functions so far throughout this book, such as *print, int, list, set, len, input* to mention a few. These are all built-in functions meaning that they have been built by Python authors and are readily available for programmers to use. However, the authors cannot define any possible function that one may need. Therefore, there exists a very powerful feature in almost every programming language and that is *custom functions.* As you may have already gotten the hint, a *custom function* is a function that is defined by the programmer. In general, a function is a block of code that is designed to receive an input, and calculate and return an output. *Custom functions* are usually referred to as just *functions* from the programming community, and that's how we are going to call them in this book from now on.

Custom functions, or functions are extremely useful and their usefulness stands behind the fact that you can define a function once, but you can call it later in your program as many times as you want by just referring to the name that you chose for your function when you defined it. In the next sections, you will learn how to both define a function and how to call it for execution.

Defining and calling a function

Defining a function requires its own syntax. Let’s look at an example of how to define a simple function that calculates the area of a square:

```
def area (x):
    a = x**2
    return a
```

As you can see, the definition of a function starts with the keyword *def* followed by a custom given name for the function, and the input parameter in brackets, ended by a colon. Once you have written this very first line, then you are free to write the algorithm that specifies what the function does. Note that the lines under the *def* line need all to be indented.

In our case here we are simply calculating the area given the length of the square side *x*, and we are storing the area value in variable *a*. The third line where we specify what the function will return. In our case we want to return the area, so we pass in the *a* variable to the *return* keyword.

That’s all about defining a function. What you also need to know is that there are two ways where to define the function. One way would be to define it in the interactive IDE, but if you want to store that function for later use, you need to write it in a file through the editor. We recommend you use the second method. Once you write and execute the function from your editor, the function definition is stored in the memory of the interactive IDE, and now you can just go ahead and call an instance of the function for execution. To do that, just type in the interactive IDE this code:

```
area (5)
```

And you will promptly see this output:

```
25
```

As you see, calling a function simply requires writing the function name, and passing an input value. Similarly, you can pass any value that you want to the function, and you can even use the returned value for further calculations. Let’s say we want the area of four similar squares:

```
area(5) * 4
```

And you would get:

```
100
```

You can even repeatedly execute the function in a loop:

```
for side in [5,10,100]:
    print(area(side))
```

What’s happening here is that the *print* function will print out the output generated by the functions for the three input values of the list. Here is what the loop would generate:

```
25
100
10000
```

As you see, there were three outputs printed out for each input item of the list.

Functions with multiple parameters

In the previous section we looked at an example of a function that was designed to calculate the area of a square. But, how about calculating the area of a rectangle instead? This would require entering two parameters, one for the length of the long side, and one for the length of the short side. Good news because it is quite easy to do just that: def rectangle (x,y):

```
    area = x * y
    return area
```

As you can see, to input more than one parameter, you just need to use a comma as a separator. Then you can use those parameters in the function as you need. Calling a function is just as easy: `rectangle (10, 5)`

That would output the expected area of the rectangle: `50`

Tip: The variables that are passed in when defining a function are referred to as *parameters*. However, the values passed when calling the function are referred to as *arguments*.

Let’s go ahead now and look at a real life example. We are going to calculate the fixed monthly payment *P* required to fully amortize a loan *L* over a term of specified *m* number of months at a specified annual interest rate *i*. The formula to calculate the fixed monthly payment is this: $P = L[i(1 + i/12)^m]/[(1 + i/12)^m - 1]$

Now, we need to define a function and implement this formula in Python code inside the function. Here it is: def mortgage(L,r,m):

```
    P = (L * (i / 12 *(1+ i / 12)**m)) / ((1 + i / 12)**m - 1) return P
```

As you see, this is a three-parameter function. The math syntax for the formula of the function looks rather complicated, but you should be able to understand every part of it if you carefully compare it with the source formula we wrote before defining the function.

The benefit here is that once you store your function in a Python file, you don’t have to type this formula any time you need to calculate the monthly payment required for loan amortization.

Let’s now call an instance of our function, by supposing we have taken a loan of 100,000 dollars at an annual interest rate of 5%, and we want to amortize it over a term of 120 months. Here is how we get the payment we would have to pay per month:

```
mortgage (100000, 0.05, 120)
```

And that would output this:

```
1060.6551523907553
```

That means you would need to pay around 1060 dollars per month for your loan.

Functions are not limited to mathematical formulas only. You can create any function that you want. Let’s create a function that gets a name, surname, email domain, and an extension and generates an email address. If you can recall from the previous chapters, we were able to generate email addresses using string formatting. We will do the same here, but everything will be written inside a function. Here is the complete function:

```
def email_generator (name, surname, email, extension): address = "{0}{1}@{2}{3}".format (name, surname, email, extension) return address
```

As you see, we have assigned four parameters to our function, and therefore we need to pass four arguments when we call it:

```
email_generator("John","Smith","gmail",".com")
```

And that would output this string:

```
'JohnSmith@gmail.com'
```

Storing the function somewhere will allow you to reuse it as many time as you will need it.

Functions with default parameters

It might happen that you will need to call certain functions over and over again in your programs. For functions with many parameters, this would require you to type in many arguments which is something that soon may become quite tiresome. In such scenario, it may often happens that you may have to enter the same arguments over and over again for the same function.

For example, you might want to generate email addresses for a wide range of names and surnames, but you only want to generate email addresses that have *gmail.com* as their base. Python has a technique that allows you to define a function with default parameters, so that you don't have to explicitly declare them when you call an instance of the function. Here is the function we defined in the previous chapter, but with default parameters:

```
def email_generator (name, surname, email="gmail", extension=".com"):
    address = "{0}{1}@{2}{3}".format(name,surname,email,extension) return address
```

See now how we go about calling this function:

```
email_generator("John","Smith")
```

And the output should this again:

```
'JohnSmith@gmail.com'
```

As you see, we didn't have to pass arguments for the *email*, and *extension* parameters, because we were fine keeping the default ones. If we don't want the default arguments, we can just go ahead and explicitly declare new ones:

```
email_generator("John","Smith","yahoo",".it")
```

And that would produce an email address having *yahoo* as a base:

```
'JohnSmith@yahoo.it'
```

To conclude it, we could say that if you foresee that a certain argument is going to be used extensively in your program, it can be a good idea to set it as a default parameter when you define a function because that will save a lot of time later on when calling instances of that function.

Tip: Non-default arguments are often referred to as *required arguments* because calling a function without specifying them will not work.

Summary

In this chapter you were introduced to custom functions. You were able to understand the difference between built-in and custom functions in Python. You learned how to define a custom function and store it in a Python file for later use. We were able to create several functions through this chapter and you learned how to create functions with more than one parameter. Furthermore, we practiced the use of functions by incorporating their output to perform further actions such as looping through a list and executing the function repeatedly. We practiced the use of functions by creating a function that calculated the monthly payment required amortizing a loan, and we also created a function that could generate email addresses out of a given set of strings. Lastly, you learned how to create functions with default parameters and understand its improved efficiency compared to having a long list of required parameters.

Assignment

Exercise 1

Create a function that calculates and returns the area of a triangle.

Hint: The formula that calculates the area of the triangle is: $a = \frac{h \cdot b}{2}$

Exercise 2

Create a script that accepts a string entered from the user, and passes the string to a function. The function itself checks the number of characters that the string has and returns the number back to the user.

Exercise 3

Consider this function we created in this chapter: def email_generator (name, surname, email="gmail", extension=".com"): address = "{0}{1}@{2}{3}".format(name,surname,email,extension) return address

Using this function, generate the following list of email addresses: ['JohnSmith@gmail.com', 'MarkPond@gmail.com', 'GeorgeLondon@gmail.com']

Hint: To iterate over two lists at the same time, make use of the *zip* function. Here is an example: [i+j for i,j in zip ([1,2],[100,1000])]

Solution for Exercise 1

```
def triangle (h, b):
    a = (h*b)/2
    return a
```

Solution for Exercise 2

```
a = raw_input ("Enter some text: ") def count (a):
    return len(a)
print(count(a))
```

Solution for Exercise 3

```
[email_generator(nm,sr) for nm,sr in zip(["John","Mark","George"],["Smith","Pond","London"])]
```

Chapter 7: Classes

Chapter objectives:

- You will learn what classes are, and when you should use them in the programs that you will build.
- You will learn how to define a class.
- You will learn how to call an instance of a class.
- You will learn how to access methods and attributes from a class that you have defined.
- You will learn how to make use of the class inheritance technique that allows for better organization of class and code in general.

What are classes?

If you have reached this point of the book, that means you now know about variables, functions, parameters, and attributes. These are all crucial elements that build up the essence of a program. Now it is time to introduce you to the concept of a *class*. A class is a technique that groups all the crucial elements we mentioned above together to offer a more organized and object-oriented way to create programs. The purpose of a class is to create some sort of object template that will produce object instances having similar *attributes* and *methods*. Therefore, we can think of classes as cookie cutters that we can mold at the moment that we define a class. Once we define a class, we can produce as many object instances as we want out of that class. As you already got the hint, a class is basically constructed of *methods* and *attributes*. In the next section, we'll look at what methods and attributes are, and of course you will learn how to define a class.

Defining a class

When we wanted to create a custom function, we first had to define it using the proper syntax, and then we could use it by calling an instance of the function. Working with classes is basically the same. Here is a simplified architecture of a Python class: class Something:

```
    def __init__(self, attribute1, attribute2, attribute3): self.attribute1 = attribute1
    self.attribute2 = attribute2
    self.attribute2 = attribute3
    def method1(self): "Perform some action with attribute1"
    "Maybe perform some other action with attribute2 also"
    def method2(self): "Here come some action with self.attribute3 as well"
```

This is how a class would roughly look like. However, you might have a hard time understanding it for now. Classes are truly useful when building relatively large scripts that have a lot of code to absorb, and this code needs a way to stay organized. The way to achieve that is by using classes to group similar things together and that is often considered advanced programming. That said, it is still a good idea to learn how to use classes now so that you are aware when to choose a class for organizing your code.

We mentioned *attributes* and *methods* earlier. Methods are just functions that are defined inside a class. Attributes are parameters that are passed to the class methods. In the class we defined earlier, *__init__*, *method1*, and *method2* are the methods of the class. *Self*, *attribute1*, *attribute2*, and *attribute3* are the attributes. The method *__init__* and the attribute *self* are special elements of a class, but we will cover them later. Now, it is time to actually write a real class that you can execute in your Python editor. Let's suppose we are running a service for managing properties, and we want to create some sort of form that generates property information. It would be a good idea to create a class that would generate instances of property information in an organized way. Here is the complete class for that: class Property:

```
    def __init__(self, address, prop_type): self.address = address self.prop_type = prop_type
print("Generates information about properties") def getaddress(self): print("Address: %s" % self.address)
def gettype(self): print("Property type: %s" % self.prop_type)
```

Let's start explaining what each of the lines does. The very first line is where we start defining our class. The *class* keyword is what denotes that a class definition is about to take place. The name *Property* is the class name chosen by us. Then, we use indentation, just like we have done with functions, loops, and conditionals. As you see, we are using the *def* keyword which defines a function. That's technically a function, but when we write a function inside a class, we refer to it as a class *method.* So, __*init*__ is the first method of our *Property* class, and it is a special method. The __*init*__ method is automatically executed when you call an instance of the class; the other methods are not. Let's quickly call an instance of the class:

```
Property("Lighted Street", "Hotel")
```

Note: The class variables are written in the __*init*__ method as you see it. Our class has three variables including *self.* However, here we passed only four attributes. This is because you don't pass an attribute for *self* beceuase *self* is an internal class variable. More details about *self* will be given later.

And let's look at the output you should expect:

```
Generates information about properties
<__main__.Property object at 0x00000000033848D0>
```

As you see, when we are calling an instance of the class, the printing function inside the __*init*__ method was executed. The other methods such as *getaddress* and *gettype* methods were not executed. The __*init*__ method is referred to as a *constructor* in other object-oriented programming languages.

You're probably wondering what these three lines of the __*init*__ method are about:

```
self.address = address
self.prop_type = prop_type
```

self is a variable that represents the instance of the object itself. On the left side of the expression, the *self.address* is referring to the *address* member variable which belongs to the class instance that the method is being called for. On the other side, the *address* refers to the parameter that was passed into the method. In fewer words, the member variables of the class are being assigned the values of the parameters. You should always assign your class member variables to the parameters in the__*init*__ method. Beside these assignments, you can also include any other code which you would want to be executed every time you call an instance of the class. In our case, we wanted to print out some general information, so we incorporated the *print* function inside the __*init*__ method. Variables *address* and *prop_type* passed to __*init*__ are referred to as class *attributes*. If we want to execute any method of the class, we should again call the class, and the method we want to execute using a dot notation as follows:

```
Property("Lighted Street", "Hostel").getaddress()
```

This is a normal scenario of how you would call an instance of a class. And the output of that would be: Generates information about properties Address: Lighted Street

Notice that, the code inside __*init*__ was again executed, but besides that, the code inside the *getaddress method* was executed as well. So, what we did here is that we executed an instance of the class, and we got an object instance generated given the attributes that we passed in when calling the class instance.

Similarly, you can call the other method as well in case you want to produce a piece of information related to property type: Property("Lighted Street", "Hostel").gettype()

And as you might have guessed, the output would be this: Generates information about properties Property type: Hostel

The code inside __*init*__ was executed again, and this time we had the method *gettype* executed, but not *getaddress.*

If you want to access only the methods of your class with the aim to execute only the code that is inside that particular method by excluding the code under __*init*__ you can leverage a technique that uses a variable to store the class instance to: prop = Property("Lighted Street", "Hostel")

And from there you can access the methods of your class by just pointing to the variable: prop.getaddress()

This would execute the code inside *getaddress* only: Address: Lighted Street

Methods are not the only thing you can access from a class; you can access attributes too. Just like this: prop.address

And that will print out the value of the given attribute: 'Lighted Street'

This could be useful if you want to perform further operations with the attributes of your class.

As you see, a class is like a structure that defines how an object should be laid out. For example, a class pretends that a property has an address and a type, but the class definition doesn't fill the methods or the attributes with content. That part is implemented when we call an instance of the class with the desired class methods as we just did.

To conclude it, you might be able to see now that classes give way to a high level of organization and grouping of the functions and variables allowing you to have better control over your code.

Inheritance

In the previous section we created a class for properties which generated instances of property information. Sometimes, one class may not be enough because you might ran across objects that do not share the same attributes. For example, all properties are expected to have at least an address, and a certain type. However, some properties such as hotels might also have a specified number of stars. We could simply go ahead and add a *star* attribute to our class, but then we might later need to use the class for generating instances of apartments and apartments don't have stars. For this reason we might need two different classes, one class that produces instances of apartment objects, and another one producing instances of house objects. The good news is we don't have to write each class from the beginning. Instead, we can just inherit the features of the base class that we already have, and then add other features that are specific to the derived class being created. This mechanism of creating a class that inherits features from another class is called *inheritance*. The class that the features are being inherited from is usually referred to as the *base class,* while the new class that inherits the attributes and the methods is referred to as the *derived class*. Let's now create a class that will generate a hotel object by inheriting the attributes of the *Property* class we defined in the previous section:

```
class Hotel(Property):

    def __init__(self,address,stars): super().__init__(address, "Hotel") self.stars = stars def getstars(self):
print("Hotel stars: %s" % self.stars)
```

Creating a derived class follows almost the same structure with some minor differences. To tell Python that you are defining a derived class, you need to pass the base class inside the brackets in the first line.

Again, we need to define an *__init__* method where we write everything that we want to be executed when an instance is called. Here we are passing two attributes for our derived class, *address*, and *stars.* We are not passing the property type because the type will always be *hotel* and we can specify that default value via the *super* function that you see below.

Further down, the *super* function will execute the *__init__* method of the base class. Here we need to specify the attributes of the base class again. These are the *address,* and the *type* of the property. Because the type of our *hotel* class will be always "Hotel", we can just enter the *Hotel* string here, so we don't have to explicitly type it any time we produce a class instance. Next, we do the usually assignment operation between the *self.stars* member variable and the *stars* attribute.

Once, you write the __*init*__method, you can then write whatever method that is specific only to the derived class that you are creating. In our case, we wrote the *getstars* method which will generate some piece of information regarding the number of stars that the hotel instance will have.

Now, we can just go ahead and assign an instance of the class to a variable: h = Hotel ("Valley Street", 5)

And from there, we can access features from both the base class and the derived class as well. Let's try to get the info about the address: h.getaddress()

And that would produce this:

```
Address: Valley Street
```

As you see, even though we didn't explicitly define a *getaddress* method for our *Hotel* class, we still were able to access it because of the inheritance that we embedded in our derived *Hotel* class. Similarly, you can access methods that were indeed defined in the *Hotel* class: h.getstars()

And that would execute the code that is inside *getstars* and produce the following output: Hotel stars: 5

As you see, inheritance is an extra feature of classes that allows you to keep your code organized. With practice, you will be able to master the syntax of defining and inheriting classes, and you will also be able to spot the right occasion where you should go ahead and use classes in your code.

Summary

In this chapter you learned what classes are and how to create them. You learned that classes are a way that will help you code in a more organized and efficient way. You were able to define a class with attributes and methods and then call instances of the class or its features. Further, we extended the knowledge about classes by practicing inheritance which is a concept that helps keeping classes more organized. Throughout the chapter, you were introduced to an example where a *Property* class was created, and you should now be able to create your own classes, and call instances of them.

Assignment

Exercise 1

Create a *Phone* class designed to create mobile phone objects. Define two attributes for the class, *price*, and *brand*. In addition to the __*init*__ method, define two other methods, a *getprice*, and a *getbrand* method. The *getprice* and the *getbrand* method will display information about the phone price and brand respectively.

Hint: You should define the class from what you remember from the class definition syntax, but you can have a quick look at the *Property* class if you get stack.

Exercise 2

Create a Smartphone class designed to create smartphone objects. The class should have three attributes, *price*, *brand* and *screensize*. It should also have an __*init*__ method, a *getprice*, a *getbrand*, and a *getscreensize* method with each method displaying relevant information about *price*, *brand*, and *screensize* respectively.

Hint: You can make use of inheritance for defining your *Smartphone* class.

Solution for Exercise 1

```
class Phone:
    def __init__(self, price, brand): self.price = price self.brand = brand def getprice(self): print("Price: %s"
% self.price) def getbrand(self): print("Mobile brand: %s" % self.brand)
```

Solution for Exercise 2

```
class Smartphone(Phone):
    def __init__(self,price,screensize, ): super().__init__(price, "Beta") self.screensize = screensize def
getscreensize(self): print("Mobile screen size: %s" % self.screensize)
```

Chapter 8: Modules and Packages

Chapter objectives:

- You will learn how to work with modules and packages.
- You will learn that to keep the language light and flexible, there exist a wide range of external modules that can be easily installed to suit different programming needs.
- You will be able to tell the difference between modules and packages and know how to use them in Python.
- You will also learn how to get help information about Python classes, functions and keywords from within Python.
- You will learn how to create your own module and use it in Python.

What are modules?

Python is quite small as a software and that's an intentional design of Python to keep the language simple and light. However, Python has a wide range of external modules that you can import into your scripts and use their functionalities. A module is simply a Python file with a *.py* extension containing classes, functions and statements. When you start a Python session through IDLE for example, external modules are not loaded by default. You need to import them to make use of their features.

What are packages?

Packages are folders containing several modules or sub-packages inside. The reason packages exist is to keep large amounts of code organized. Just like modules, packages can also be imported into an interactive Python session or a Python script. Some commonly used packages are installed by default when you install Python, but some others need to be installed separately. The same goes for modules. We will consume everything you need to know about packages and modules in the next sections.

Working with modules and packages

As we already mentioned, modules are simply Python files that can be imported into an interactive Python session or editor. In this section we will cover the *datetime* module as an example of working with modules.

Note: All the techniques that we will use with the *datetime* module work the same as with any other module or package.

datetime is a module that contains classes and functions for working with dates and times. To access its functionalities you need to import it to Python like this:

```
import datetime
```

Once you have imported it, you can access the module features using a dot notation. Let's try to call the *now* function inside the *datetime* class of the *datetime* module: datetime.datetime.now()

And that would output this:

```
datetime.datetime(2015, 6, 2, 14, 47, 29, 843894)
```

And that’s an array containing the current year, month, day, hour, minutes, seconds, and milliseconds.

Working with the *datetime* module is quite cool, but let's don't forget we are talking about modules in general here. We mentioned that a module is just a Python file, so, let's try to find out where the *datetime* module is located in our computer. To get the path, simply type in the module name in the interactive session: datetime

And that will output something like this: <module 'datetime' from 'C:\\Python34\\lib\\datetime.py'>

As you see, the output contains the path where the module file is located. The path can be slightly different in your computer, but you get the idea. If you browse through that path and open the datetime.py file using the editor, the code you will see is the entire code that makes up the *datetime* module. If dive deeply into the code, you will find a *datetime* class. That's the class we called from our module using *datetime.datetime.*

Note: The fact that the class *datetime* has the same name with its module is just a coincidence.

If you look even deeper inside the *datetime* class, you will find the *now* function which we called in using *datetime.datetime.now()*.

Tip: Normally, you will not need to open the source code of your modules or packages, and you should be careful if you do because if you change or delete something by mistake, it might make cause the module not to work. We are accessing the module source code in this book so that you to understand how things work.

Accessing packages and their functionalities follows exactly the same procedure. You first import your package, and then use the dot notation to access its features. Here is an example of accessing the *tkinter* package, a package that implements graphical interfaces for your Python programs:

```
import tkinter
```

And from there, you can access the package features via the dot notation:

```
tkinter.Tk()
```

And that would produce an empty graphical window.

Sometimes, you might need to repeatedly use a feature of a module or package. In such cases, it would be more practical to import only that feature from your module or package. To import a single feature, do this:

```
from datetime import datetime
```

This would import only the *datetime* class from the *datetime* module. This keeps your program lighter because not every feature of the module is loaded to the script, but just the one you need. It also reduces the code amount, because instead of doing this:

```
datetime.datetime.now()
```

You can just do this now:

```
datetime.now()
```

And that would give you the same output.

Installing external modules and packages

The *datetime* module and the *tkinter* package we used as examples in the previous section came shipped with the Python installation by default because they are considered commonly used packages by the community. Other more specialized modules or packages need to be installed manually into your system. The good news is, there is a way to install modules and packages very easily. You just need to open your computer command prompt or terminal and type in this code: pip install modulename

For example, *numpy* is an external package offering 2D array objects in Pyhton. To install numpy you should type this code: pip install numpy

That will download numpy from the online repositories and it will install it in your computer. *pip* itself is a package that is installed by default with Python and it is package that implements easy installation of external Python packages, just as we're doing here.

Once the installation is completed, you can just go ahead and import the package as we did in the previous section: import numpy

Last thing worth to mention is that if you want to remove an imported module or package from a session, you can use the del keyword followed by the module or package name. For example: del numpy

You can always import the module or the package back if you want.

And that's all you need to know about installing external modules and packages.

Code introspection

Code introspection is a set of techniques used to inspect packages, modules, classes, functions and keywords. Code introspection in Python is implemented through several functions that return various information about a feature. In this section we will be trying out some of the introspection techniques. An introspection "task" could be to find some general information for a feature. For example, if you want to display some help info about the *datetime* module, you could use the *help* function:

```
import datetime
help(datetime)
```

That would print out a description of the module, and it would list the module features and their details. If you simply want a list of features contained inside the module, you can use the *dir* function:

```
dir(datetime)
```

You can perform the same operations over classes and functions as well. For instance:

```
help(datetime.datetime)
```

And even deeper than that:

```
help(datetime.datetime.now)
```

Another useful code introspection feature is the *type* function:

```
type(datetime)
```

A *type* function will output the type of the object that passed to the function. The line above would output this:

```
<class 'module'>
```

And that says that *datetime* is a module.

As you see, there are quite a few functions that allow you to introspect the code you are about to use.

Creating your own module

You now know that a module is simply a Python file having classes, functions and statements inside, and you also know that you can import a module or its features into a Python script or an interactive session.

As you get better with Python and start developing relatively long scripts that you want to reuse inside other scripts, it might be a good idea to import your script or its features as a module in another script.

Let's take an example to illustrate the idea. Consider the class we created in the previous chapter: `class Property:`

```
    def __init__(self, address, prop_type): self.address = address self.prop_type = prop_type
print("Generates information about properties") def getaddress(self): print("Address: %s" % self.address)
def gettype(self): print("Property type: %s" % self.prop_type)
```

To use it as a module, first of all, we need to store it inside a Python *.py* script, and store it among the other modules in the Python installation directory which could be something similar to *C:\Python34\Lib.*

Tip: To store some code in a *.py* script, you can either create an empty file in the modules location, paste the code inside the file, and name the file under a *.py* extension, or you can create a new file through the Python interactive IDLE and save it in the modules location.

Once you have saved the script under a name such as property.py, you can open a new interactive session, and import the new module as follows: `import property`

As you see, we used the name of the file to refer to the model. Once you have imported it, you can use all its functionalities by referring to the model name. Here is an example: `property.Property("An address", "Apartment").getaddress()`

And that would produce this output:

```
Generates information about properties Address: An
address
```

And of course, you can also import only the feature that you want from the module. Let's import the *Property* class only: `from property import Property`

That would import only the *Property* class and its components from the module.

As you see, Python is highly flexible so that it can suit various custom needs.

Summary

In this chapter you learned about modules and packages. You were able to understand the use of modules and packages in your programs. You were also explained the difference between a module and a package. However, you saw that using packages and modules in your scripts follows the same importing and accessing rules.

Furthermore, you learned how to install external Python modules and packages that didn’t come automatically installed with Python. You were also introduced to code introspection by learning how to get relevant information about Python classes, functions, and keywords.

Lastly, you were able to create your own Python module and use its features inside Python.

Assignment

Exercise 1

1. Which of the following statements about modules is not true?
    1. A module can be imported into a script or an interactive session just like a package does.
    2. A module is a *.py* file containing Python packages.
    3. A module is a *.py* file containing classes, functions and statements.
    4. A module provides a good technique for keeping scripts organized.
2. Which of the following statements about packages is not true?
    1. Packages are imported just like modules.
    2. Packages can be collection of modules.
    3. Some packages do not come with the Python standard installation.
    4. Packages are a must for Python to work.
3. What is not true about the following code?

```
from base import element
```

1. It imports only the *element* feature from the *base* and no other element more.
2. It waives the use of *base.element* in the code that will come after that line.
3. It works with modules, but not with packages.
4. *element* belongs to *base.*

Exercise 2

Create a module that contains the following code and name the module *assignment.py.*

```
intro = "Enjoy using our module!"
def area (x):
    a = x**2
    return a
def email_generator (name, surname, email, extension): address = "{0}{1}@{2}{3}".format (name,
surname, email, extension) return address
```

Once you have created the module and placed it along the other Python modules, perform the actions on the left column of the table and write the code that you used to perform those actions in the right column.

Action	**Code**
Import the module into the interactive session.	
Display "Enjoy using our module!"	
Output the area of a square with a side length of 6 units using the *area* function.	
Remove the assignment module from the session.	
Import only the *email_generator* function to the session.	
Import the *area* function following the same technique.	

Solution for Exercise 1

Correct answers:

1: b

2: d

3: c

Solution for Exercise 2

Action	**Code**
Import the module into the interactive session.	import assignemnt
Display "Enjoy using our module!"	print(assignment.intro)
Output the area of a square with a side length of 6 units using the *area* function.	assignment.area(6)
Remove the assignment module from the session.	del assignment
Import only the *email_generator* function to the session.	from assignment import email_generator

Import the *area* function following the same technique.	from assignment import email_generator

Chapter 9: File Handling

Chapter objectives:

- You will learn how to open an existing text file in Python and read its content.
- You will also learn how to write new content to an existing file.
- You will be able to create new text files and type text in them from within Python.
- You will also be able to open existing binary files and create new ones essentially using the same techniques as with text files.
- You will learn a very robust way of handling files in Python using the *with* keyword.
- You will also be introduced to the *os* module which is a built-in module used to interact with directories and perform actions such as creating new directories, changing the current working directory, listing directory content, and deleting folders and files as well.

Opening a file with Python

Opening files using computer programs is a regular routine of every computer user. For example, to read a text file, you need to open it with a certain reader such as a notepad program. Likewise, you can open files in Python too. However, the technique to open files in Python does not involve a simple double clicking of the file – you need to write a few lines of code to open a file in Python.

Files can be either text or binary. We will be handling both these types files in this chapter. There are also two different opening modes available in Python, a *read* mode and a *write* mode. When you open a file in *read* mode, you cannot write anything to that file. All you can do with the file is read its content. This works well when you just want to get information from the file. If instead you are planning to modify your files with Python, you have to specify a *write* mode when opening them. In the following sub-sections, you will see how to open text and binary files. Opening a file in *read* mode is often simply referred to as reading a file, and opening it in *write* mode, is referred to as writing a file. We will use these two abbreviations in this chapter.

Before looking at examples on how to open the different file types with different modes, let's first have a look at the syntax on how to open a file in general. Here it is the general syntax: `file = open (filepath, mode)`

This is it. In the first line, we are creating a variable called *file*. That's where we store our opened file object. And inside the *open* method we specify the file path string, and the opening mode string. Next, if you need to access the content of the file, you would use a *read* method like this: `content = file.read()`

This would store the text content of the *file* object in the *content* variable. Once you have grabbed the content, you can do whatever you like with it. For example, you can simply print the content out: `print(content)`

And once you have finished working on your file, you need to close the file, just as you do with any other program, so that you don't cause any file conflict. To close a file with Python, you would use the *close* method.

```
file.close()
```

That would denote the end of reading a file and it would free the file from any connection with the Python program. In the next sub-sections we will explains the different scenarios.

Reading a text file

Let's suppose we have a text file named *sample.txt* in our *C:/examples* directory in our computer and the text file has a line of text inside saying *This is a sample*. Here is a code that would open the text file in *read* mode:

```
file = open("C:/examples/sample.txt", 'r')
```

As you see, we passed in the file path, and the read mode denoted by an *r.* At this point, a file object has been opened in Python. Let's now read its content and store it in a variable:

```
content = file.read()
```

And let's print the content out:

```
print(content)
```

And you would expect this output:

```
This is a sample.
```

When finished with the file, you need to close the file as follows:

```
file.close()
```

That's how you read the content of a text file. However, if there were more than one line in the file you might have to take another workaround. Let's suppose the file had two lines:

```
This is a sample.
This is some content.
```

You could use this code to read such a file with the *read* method:

```
file = open("C:/examples/sample.txt", 'r') content = file.read()
print(content)
```

You would get this output:

```
'This is a sample.\nThis is some content.'
```

Note: The *\n* is a special Python character denoting a break line.

The problem here is that as you see, all the lines were read as a single string. A better solution would be to store each line as a list item so you can manage the lines better. This is very easy to do. Simply apply this code: **Tip**: Don't forget to close the file before reading it again because that may cause problems.

```
file = open("C:/examples/sample.txt", 'r') content = file.readlines()
print(content)
```

This would print out a list:

```
['This is a sample.\n', 'This is some content.']
```

As you see in the code, instead of *read,* we used *readlines* which produces a list of text lines. Now we can treat and print those lines one by one using a simple loop:

```
for line in content:
    print(line)
```

That would output this:

```
This is a sample.
This is some content.
```

This is all you need to know to get started with reading text files. Next, we will see how we can write content inside text files.

Writing inside a text file

In this section we will focus on how to write some content in a text file. The syntax is generally the same as in the case of reading content, but here we need to specify a *write* mode in the *open* function and also apply a *write* method to the file object. Here is an example that will add a third line to our *sample.txt* file:

```
file = open("C:/examples/sample.txt", 'a') file.write("This is some more content\n") file.close()
```

<u>Note</u>: Remember that once you have written something in the file, you need to close the file to see the content inside it.

As you see in the last block of code, we passed in the file path and an *append* writing mode denoted as *a.* Next, we used the *write* method to append a line to the text file. The *\n* special character creates a break line just after the line we are writing. The reason we create a break line is that if we write a second line later, the line will be written below the previous line, and not next to it.

If you check the text file now, you will notice three lines in it:

```
This is a sample.
This is some content.
This is some more content.
```

There is one more writing mode that is worth mentioning and that’s the *w* mode. The *w* mode is used when you want to overwrite the existing content of the text file. Here is an example:

```
file = open("C:/examples/sample.txt", 'w')
file.write("This overwrites everything\n")
file.close()
```

If you execute this code, you will see that whatever text you had in your file, the text is now gone. All you will see is the line you just wrote:

```
This overwrites everything
```

This is how you can write content in text files. Combining these functions with looping techniques makes the file handling a powerful tool.

Creating a new text file

Creating a new text file involves the same syntax as in opening an existing file in *write* mode. Let's try to create a file called *new.txt* and write a line of text inside it:

```
file = open("C:/examples/new.txt", 'w')
file.write("New text here.\n")
file.close()
```

As you see, the code is exactly the same as in the case where the *new.txt* file would be an existing file. So, Python checks whether there already exists a file in the computer or not. If it exists, it opens that file. If it doesn’t, it creates such a new file.

Reading, writing and creating new binary files

A binary file is any file that is not a text file. Reading a binary file follows the same syntax as in reading text file with the difference in the opening mode. Here is an example on how to read a binary file:

```
file = open("C:/examples/sample.jpg", 'rb')
content = file.read()
```

This will store the binary content of the *sample.jpg* file in the *content* variable. You can then use that content to perform further actions. For example, you can write the content in a new file in a new location:

```
outfile = open("C:/output/out.jpg", 'wb')
outfile.write(content)
file2.close()
```

If you open the *out.jpg* file with a photo viewer you will see that the files have the same content as the *sample.jpg* file. As you see, creating such a copy is a use case of writing binary files in Python. As you build Python programs, you will run into more complex scenarios where you may interact with different types of files and use Python as an intermediator to handle files.

Handling files easier with "with"

In the previous section, we covered all the techniques that you need to know for handling text and binary files. These techniques are a must for beginners to understand how file handling works in Python. However, they are not a must to use in practice because Python offers a more robust way of handling files through the *with* technique. For example, you learned that in order to create a new file, you had to write these lines of code:

```
file = open("C:/examples/new.txt", 'w')
file.write("New text here.\n")
file.close()
```

You can do exactly the same thing by wrapping the functions inside a *with* method block. Here is the code:

```
with open("C:/examples/new.txt", 'w') as file:
file.write("New text here.\n")
```

This will create a new file for you and write some text in it.

The way we would read these two lines of code in human language would be this: with the opened *new.txt* file as *file*, perform the following action which is writing the *New text here* line.

If you didn't noticed it, here we didn't use the *close* method because the *with* block will close the file for automatically once the operations have been finished. Notice that we are using indentation here just as we have done with other kinds of blocks such as *for*, *while*, and *if* blocks.

The *with* method is often preferred over the classic method because it is considered shorter and more solid and we suggest you use it whenever you want to read and write existing text or binary files, or create new ones.

Working with directories

It can happen that you might have to deal with directories while coding in Python and there are quite a lot of Python features out there to help you handle directories. In this section we will cover the most important ones.

Changing the current working directory and creating new ones

You may have noticed that when we accessed or created a new file, we passed a full file path as the file object: with open("C:/examples/new.txt", 'w') as file: file.write("New text here.\n")

Sometimes, you might need to just pass the file name only. However, if you just pass the file name to your *open* function as follows: with open("new.txt", 'w') as file: file.write("New text here.\n")

That would create a file *new.txt* in the current working directory which is often the folder where the Python installation is located. You probably don't want the file to be created in that directory, and for this reason there exists a way to change the current working directory.

The method that changes the current directory and other similar methods that interact with your operating system are accessed through the *os* module. Here is how we would handle our case of creating a new text file: import os

os.chdir("C:/examples")

with open("new.txt", 'w') as file: file.write("New text here.\n")

This would create the *new.txt* file in the *examples* folder. As you see, once we imported the module, we called the *chdir* method which changes the current working directory to *C:/examples*. Once we have done that, we can just pass a file name to the *open* function.

Be aware that the *examples* folder must exist in your system. Otherwise, an error will be thrown. So, you should either create a folder manually or have Python do that for you. If you're headed for the second option, then what you need is the *makedirs* method of the *os* module. Here is the updated code that includes the *makedirs* method: import os

os.makedirs("C:/examples") os.chdir("C:/examples")

with open("new.txt", 'w') as file: file.write("New text here.\n") As you see, you need to create the directory first if it doesn't exist, and then change the current directory to that. If not sure which the current directory is, you can always make use of the *getcwd* method as follows: os.getcwd()

That would return something like this depending on what your current directory is: 'C:/examples'

Notice that the object that you pass or that you get from the *os* methods are often strings. You can check that out yourself if you like using the type method: type(os.getcwd())

And that would return this:
<class 'str'>

This tells you that the *getcwd* method generates a string. Therefore, you are able to manipulate these strings using string formatting techniques which we covered in the string formatting section of the second chapter.

However, there are also methods such as *listdir* which would output othera list datatype. Specifically, the *listdir* method would generate a list of file and folder names contained in the current working directory. Here is how you would use it: os.listdir()

Simply enough, this would output something like this: ['new.txt', sample.txt', 'subfolder']

As you see, we didn't pass anything to *listdir.* You can also list the content of every other directory path in your computer without having to change the current working directory, but you would have to pass the directory path in that case: os.listdir("C:/other")

This would output a list of file and folder names contained inside the *other* folder.

Deleting files and directories

In case you need to delete an empty directory, you can use the *rmdir* method as follows: os.rmdir("C:/examples")

Note: If the directory you are trying to delete contains any files inside, the deletion will fail.

To delete a directory that contains files inside, you need to use a method called *rmtree* which is a property of the *shutil* module. Python 3.4 has *shutil* already installed, so you don't need to do any extra installation work. Here is how you would use *rmtree* after you have imported *shutil*: `shutil.rmtree("C:/examples")`

That would delete the folder and all its content.

Note: If a file contained in the folder you are trying to delete is already opened in Python or other program, the deletion of the directory will fail.

Finally, to delete a single file, you can use the *remove* method of the *os* module as follows: `os.remove("C:/examples/sample.txt")`

Again, if the file is already opened in another program, Python will throw an error.

That was some compact information about how to work with directory from within Python. Now that you know how to do code introspection, you can go ahead explore other features that are available in the *os* or the *shutil* modules.

Summary

In this chapter you learned how to handle files with Python. We looked at how to handle both text and binary files. Specifically, you learned how to read a text file in Python and use the file content for preforming other operations within Python. We also looked at how to open a text file in writing mode and write some text inside it. You were able to understand the difference between writing and appending some text in a text file. Moreover, you also learned how to create a new text file in a certain directory, and we also wrote some content inside the newly created file as well.

Further, we looked at how to handle binary files. You learned that handling binary files follows the same concept as with text files.

On top of the classic methods to handle files, you also learned a compact method of handling files using the *with* keyword. You were able to understand the efficiency in using *with* by practicing a few file handling examples.

You also learned how to interact with directories in Python using the *os* module. You now know how to change the current working directory, create a new directory, and list directory content. Lastly, you also learned how to delete existing directories from within Python.

Assignment

Exercise 1

Write a Python script that will create a new text file in your computer. Then, using looping, write the following lines of text inside the text file: 1145

```
1139
1132
No data
1141
```

Hint: You could build a list with the required values and iterate through it inside the *with* block. Don't forget that you should pass strings datatypes to the *write* method.

Exercise 2

In this exercise, you have to do the reverse of exercise 1 that is reading the existing text file you created in exercise 1, and store the lines of text in a list.

So, you will start from the text file that contains these lines: 1145

```
1139
1132
No data
1141
```

At, the end you should create a list that looks like this: ['1145', '1139', '1132', 'No data', '1141']

Hint: If you choose to use *readlines,* you should care to end up with items without the *\n* part. One way to do this is that instead of passing *item* to the *append* method, you can pass *item.strip("\n")*. The *strip* method removes a part from a string.

Solution for Exercise 1

```
with open("C:/examples/data.txt", 'w') as file: for item in ["1145","1139","1132","No data","1141"]:
file.write(item+"\n")
```

Solution for Exercise 2

```
data = []
with open("C:/examples/data.txt", 'r') as file: for item in file.readlines(): print(item)
    data.append(item.strip("\n"))
```

Chapter 10: Other Advanced Functionalities

Chapter objectives:

- You will be introduced to more specific functionalities that are available in Python.
- You will learn how to make your code more readable by commenting it.
- You will know more about errors and will also learn how to handle exceptions.
- You will learn how to create and connect to databases; create tables, insert, update and delete rows, and query database data from tables.
- You will how to find patterns in text using regular expression operations.
- You will also be strengthening your general Python skills by implementing examples while learning these advanced concepts.

About this chapter

If have reached this point of the book, that means you now know how to handle Python and make quite a few programs using the core operations of the language such as making math calculations, manipulating strings and lists, creating and using custom functions, working with classes and modules, and building conditionals and loops. In this chapter you will learn a few more functionalities that maybe are not that vital to build basic non-specialized programs, but very important as they will strengthen your foundations in Python.

Some of the functionalities that you will learn in this chapter will help you write more efficient code, allow you to better handle your errors, and also make you aware of what can you do more with Python.

Commenting you code

Often scripts look intimidating, especially the ones that don't contain any comments. Commenting your code means writing general human language inside your scripts with the aim to provide information and explanations to the programming code. However, you can't just go ahead and write lines without following some predefined syntax rules. The good news is that inserting comments in your code in Python is very easy. There are actually two ways of doing that. You can either add single or multiple line comments. We will explain them both in the following example. Spend a few moments and try to understand this piece of code: id = input("Enter your ID card number: ") def texting(id):

```
    with open("C:/examples/ids.txt", 'a') as file: file.write(id+"\n") texting(id)
```

Even though this code is not complicated, even an experienced program may spend some time trying to understand what it does. Now, we will add comments to the code and ask you to go through the code again and try to understand it better: """This program prompts the user to enter their ID number. Once the user enter their ID number, the number is stored under a new line in a text file.

```
Each entered ID will be stored in a new line."""

#prompting the user to enter their ID
id = input("Enter your ID number: ")
#Building a function that opens a text file and stores the IDs in there def texting(id):
    with open("C:/examples/ids.txt", 'a') as file: file.write(id+"\n")
#Calling the function and passing the user input to it texting(id)
```

As you see, we added some explanations to the code. The first part that is enclosed in triple quotes is a multiline comment. A multiline comment is used when you need to write a relatively long comment that is expected to consume more than one line. Usually, multiline comments are written at the beginning of the script to explain what the script does.

Next, we have a single line comment. Single line comments start with a hashtag (#) symbol and they are used to explain small parts of your code. If you want to add another line of comment, you should write it in another line and have it start with a hashtag.

The idea here is that Python will ignore the text that you write after the hashtag symbol. The same works for the code written inside the triple quotes. It is highly recommended that you use commenting regularly in your code because that will help understand the code better for both you and other who may be involved in your scripting as well.

Errors and Exceptions

As any other programmer, you will come across numerous errors as you write and test your code. These errors can be of different types. The most common ones are syntax errors. A syntax error happens when you haven't followed the syntax rules of the language. When you hit run, before executing the code, Python checks for syntax errors first. For example, if you write this code:

```
x = 3:
```

Python will detect there's an unexpected column at the end of the line and it will prompt you about a syntax error.

However, even when the syntax of your code is correct, you may still get an error. Errors during execution are called *exceptions*. Let's have a look at an example. The code is syntactically correct, but there's an exception in it:

```
a = 2
b = 0
c = a/b
```

This code will try to divide two by zero which mathematically doesn't make sense. Therefore, so you will get an exception as follows when executing this code:

```
Traceback (most recent call last): File "C:\examples\Exceptions.py", line 3, in <module> c = a/b
ZeroDivisionError: division by zero
```

This tells us that an exception was found in line 3. Every kind of exception has a name, and the name for this one is *ZeroDivisionError*. There's also a short explanation of the exception stated here as *division by zero*. So, Python is trying to tell us that the source of the problem and it does quite a good job in doing so. However, the error message we got maybe not very friendly for the user we are targeting for the program we build. This is why the idea of exception handling exists.

Handling exceptions

Using exception handling techniques, we can anticipate an exception and alter its display message. The specific technique to achieve this is the *try* and *else* clause. Let's incorporate the clause to our code:

```
a = 2
b = 0
try:
    c = a/b
except ZeroDivisionError:
    print("Numbers cannot be divided by zero. Please change the value of b.
```

")

As you see, we enclosed the expression where we anticipate a possible error inside a *try* block. First, Python will execute the first two lines of variable declaration, and then executes the *try* block. If *b* was not zero, the *c* variable will be assigned the calculated value and no error will be thrown. If there is a ZeroDivisionError exception, then the code under *except* will be executed. So, in this case, we would get this output: Numbers cannot be divided by zero. Please change the value of b.

You can also leave *except* without explicitly declaring what error to expect. This would be as follows: a = 2

```
b = 0
try:
    c = a/b
except:
    print("Numbers cannot be divided by zero. Please change the value of b.
")
```

This would not handle only division errors, but every kind of error. Leaving except without an explicit error is not recommendable because it generalizes the errors that may be thrown and hides the type of error you are getting, making it difficult to fix your code. If you expect to have more than one error in an expression, you can apply more than one *except* clause as follows: a = 2

```
b = 0
try:
    c = a/d
except ZeroDivisionError:
    print("Numbers cannot be divided by zero. Please change the value of b") except NameError:
    print("The formula seem to be wrong")
```

Running this code would output this: The formula seem to be wrong

That is because we are getting a *NameError* in the formula because we are trying to divide the *a* variable with the *d* variable which has not been defined anywhere, thus the *NameError*.

The list of exceptions is quite long, but only a few of them will be occuring often while you make programs. For a full list of exceptions you can refer to the Python official online documentation.

Database handling

Interacting with a database is one of the most outstanding aspects of Python. You already know that a database is an organized collection of data. The most common scenario on how Python interacts with a database is in a web server – web client scenario. In such a scenario, the database resides in the server, and Python makes requests to the database such as sending SQL queries, receiving the input, and generating some output for the web client.

To be able to use Python for interacting with a database, you need a special database library. We will use the *sqlite3* library here. *sqlite3* has been automatically installed when you installed Python, so you don't need to install it again.

In this section we will be connecting to a database, create a database table, inserting, updating and deleting table rows, and retrieving table rows. So, let's start by establishing a connection to a database:

```
import sqlite3
db = sqlite3.connect('C:/examples/mydb.db')
```

All we did here is we imported the *sqlite3* library, and connected to the *mydb.db* database. If the *mydb.db* database does not exist in your system, the code will create a new one under named *mydb.db,* and it will establish a connection to that database.

Next, we will create a table structure in our existing database. The following code does just that:

```
db.execute('create table song ( title text, author text, rating int)')
db.commit()
```

As you see, we are applying the *execute* method to the database object. The *execute* method gets SQL code as output. SQL is the language used to manage data held in databases. Even though you generally need to know some SQL to be able to interact with databases via Python, you will still be able to understand it as it is not a difficult language.

So, the expression *'create table song (title text, author text, rating int)'* is the SQL code we are sending to the database. What it does is create a table named *song*, and this table will have three fields, *title, author,* and *rating*. Each of the fields will later hold data of a predefined type. In this case the *title,* and the *author* fields will be text, while the *rating* filed will be an integer. The *commit* method then saves the changes made to the database.

Notice that if you try to create a table that already exists, you will get an error. To avoid that, you can add another line before creating the table as follows:

```
db.execute('drop table if exists song')
```

This line will delete the table if it already exists.

So, at this point, all we have is a database and a table with three empty fields. To insert some rows to your table, you need to call the *execute* method again and pass the appropriate SQL code that inserts rows to a database table. This would be as follows:

```
db.execute('insert into song (title, author, rating) values ("The map of reality", "The New Smiths",9)')
db.commit()
```

As you see, the SQL keyword to insert a new row is *insert* followed by *into* and the table name. Then you need to specify the fields you want to insert row values to, and then the actual values corresponding to those field names.

If you change your mind about the values you inserted, you can always updtate them using the SQL *update* keyword as follows:

```
db.execute('update song set rating = 10 where title = "The map of reality"') db.commit()
```

This will update the song table and set the rating to 10 for the row where the title value is *The map of reality*. You can update any other record using the same syntax.

At this point, we have a database with a table containing three fields and one row with values for each field. You probably want to see the values you have in your table now. Well, one way to do that is to open the *mydb.db* file via a database program if you have one. The other way is to use Python code to retrieve the data and display them in Python. We will do just that now. Remember that you need to have an established connection to the database before executing this code:

```
data = db.execute('select * from song')
```

This will create a *cursor* object and store it in the *data* variable. A *cursor* object contains tuples of data it receives from the database table. In this case we selected all the rows from the *song* table. We can now iterate through the cursor and print out the rows that it has fetched like so:

```
for row in data:
    print(row)
```

This loop would output this:

```
('The map of reality', 'The New Smiths', 10)
```

And that's a tuple object that we grabbed out of the *cursor* object. The benefit of retrieving data from the database is that you can perform various actions to the database data with Python and use them in your programs.

Lastly, we will see how we can delete rows from a table using Python. As you might expect, this also employs very few code: db.execute('delete from song where rating = 10') db.comit()

That would delete the entire row where the *rating* value is equal to 10.

Great! This should give you a good understanding of how Python can be used to interact with a database. You now know how to perform the most common database operations from within Python.

Regular expressions

Regular expressions are operations used to match text patterns. Regular expressions are commonly used when building text processor programs. For example, if you search for a word in a text document, some sort of regular expressions will be running in the background to search and match your text pattern. Therefore, if you are planning to process text using Python, chances are that you will have to learn regular expressions for matching text patterns. The regular expression operations are brought about by the *re* Python module. The module has been automatically installed when you installed Python, so you can just import it to your script or interactive session.

Let's now go ahead and look at an example. We are going to store some text inside a string, and then see if we can find a specific pattern inside the text. Here is the code:

```
import re
text = "Here is some text that contains something."
pattern= "that"
re.search(pattern, text)
```

What we did here is we imported the *re* module first, and then we created some text, which is a string datatype and stored it in the *text* variable. We also stored our string pattern in variable *pattern.*

The last line is what contains the regular expression operation. In this case we are looking at the *search* method which will scan through the text looking for a match to the given pattern and it will return a *match* object if the match was found, or *None* if there is no match. Because the pattern *that* is contained in the text, we will be returned a *match* object as follows:

```
<_sre.SRE_Match object; span=(18, 22), match='that'>
```

In the returned *match* object you can see the span of the pattern which is from index 18 to index 22. The object also contains the pattern that was being searched. If you need, you can extract the span of the pattern, by applying the *span* method to the *match* object as follows:

```
re.search(pattern,text).span()
```

This will return the tuple containing the span indexes:

```
(18, 22)
```

You can also extract the start index only:

```
re.search(pattern,text).span()
```

And that will return this:

```
18
```

Similarly, you can also get the end index: `re.search(pattern,text).end()`

And you will get this:

```
22
```

The *search* method seems to be working smoothly, but you were probably expecting some more friendly output. When you search for a pattern in a text processing program, you don't get an ugly *match* object as a return. So, let's try to improve the output for the user by returning two different messages depending on whether the match is found or not. Here is where the other fundamental programming concepts come in handy. To implement this, we need to employ an *if* conditional statement in the code as follows:

```
import re
text = "Here is some text that contains something."
pattern = "that"
if re.search(pattern, text):
    print("Yes, %s was found in the text" % pattern) else:
    print("No, %s was not found in the text" % pattern)
```

As you see we are passing the *search* method to the *if* statement. If the pattern is found in the text, a *match* object is generated and the conditional occurs. Therefore, the first *print* function will be executed. If the pattern is not found in the text, a *None* object is generated and the condition is not satisfied, thus the code under *else* is executed. In this case, the pattern occurs in the text, so we will get this output:

```
Yes, that was found in the text
```

If you like to improve the output even more, you could add an *input* function asking the user to enter the pattern, and you can also print out the span of the pattern in the text.

Matching more than one pattern

In our previous example, we had only one pattern occurring in the text. If we had two same patterns, the search method would fail to find them both:

```
text = "Here is some text that contains something that occurs twice."
pattern= "that"
re.search(pattern, text)
```

This would return this:

```
<_sre.SRE_Match object; span=(18, 22), match='that'>
```

As you see, that would capture only the pattern that occurs first. To capture all patterns, we need to use the *findall* method as follows:

```
text = "Here is some text that contains something that occurs twice."
pattern= "that"
re.findall(pattern, text)
```

That would return a list of both the patterns:

```
['that', 'that']
```

A use of *findall* would be to get the count of the occurring patterns as follows:

```
len(re.findall(pattern,text))
```

And that would return this:

```
2
```

So, you know how many patterns where found.

If you still want to get *match* object as return, instead of a list, you would have to use the *finditer* method.*finditer* will return an iterator object which contains the set of *match* objects found during the search. Here is how we would implement it:

```
import re
text = "Here is some text that contains something that occurs twice."
pattern= "that"
matches = re.finditer(pattern, text) for i in matches:
    print(i.span())
```

As you see, we stored the iterator object to the match variable, and then we iterated through the *iterator* and accessed the spans of the found matches and printed them out.

These were some of the main methods that we could mention in this book as far as regular expressions are concerned. More regular expressions exist out there that suits more complex matching patterns, but you will learn them as you advance with Python and the need calls you to work on building text processing scripts.

Summary

In this chapter, you learned some more specific concepts that not only introduced you to new techniques, but also enforced your existing fundamental Python skills. You learned how to make your code more readable by adding comments to your scripts.

You learned about errors and exceptions and you should be now able to tell the difference between the both. You also learned how to handle exceptions and make them more friendly so that the external users understands them.

Further, we explained how you can handle databases via Python. Specifically, we created a database file, and connected to the database. You also learned how to create database tables and table fields. Moreover, we worked with table rows, by inserting new ones, updating, and deleting them. Finally, you learned how to query data from the database table and use the queried data in Python.

Lastly, we covered regular expressions and you learned various ways of how to search for a pattern in a text.

Assignment

Exercise 1

In this exercise you will be able to practice exception handling, databases, and commenting. The exercise requires that you write a script that does the following:

1. Creates and connect to a database
2. Creates a database table called computer with three fields: brand, disk, and screen where brand is text type, while disk and screen are integers.
3. Inserts three rows of data into the table

In addition, you should write the code using the *try – except* clause to handle the *sqlite3* exception referred to as *sqlite3.OperationalError.* Also, enrich your script by adding a header multiline comment and single line comments where you see it appropriate to explain what your lines of code do.

Exercise 2

The following script searches for a pattern and displays a message notifying whether the pattern was found or not.

```
import re
text = "Here is some text that contains something."
pattern = "that"
if re.search(pattern, text):
    print("Yes, %s was found in the text" % pattern) else:
    print("No, %s was not found in the text" % pattern)
```

The message printed out in this case would be: Yes, that was not found in the text

Try to improve that message by displaying something like this instead: Yes, 'that' was found in the text spanning from index 18 to 22

Hint: Make use of string formatting, and the *start* and *end* methods of the *match* object.

Solution for Exercise 1

```
"""This script creates a database table having three fields, and populates the table with one row."""

import sqlite3
#Here we try to execute the SQL commands try:
    #connect to database db = sqlite3.connect('C:/examples/mydb.db') #delete table if it exists
```

```
db.execute('drop table if exists computers') #create table
    db.execute('create table computers ( brand text, disk int, screen int)') #insert rows
    db.execute('insert into computers (brand, disk, screen) values ("Cosmos", 512, 22)') db.execute('insert
into computers (brand, disk, screen) values ("Beta", 1024, 18)') db.execute('insert into computers (brand,
disk, screen) values ("Europa", 512, 19)') #save changes
    db.commit()
except sqlite3.OperationalError: #printing a message if an SQL error occurs print("You have something
wrong in the SQL declaration")
```

Solution for Exercise 2

```
import re
text = "Here is some text that contains something."
pattern = "that"
match = re.search(pattern, text)
if match:
    print("Yes, '%s' was found in the text spanning from index %s to %s" %
(pattern,match.start(),match.end())) else:
    print("No, %s was not found in the text" % pattern)
```

Chapter 11: Conclusion

This chapter concludes this book on Python programming for beginners. We had to start from the very beginning by explaining what Python is and what you can do with it. Now that you have read the book, you probably have more clear ideas on how you can use Python to suit the needs on your specific field. Whatever your interest is, the chapters of this book should have been a great source to get you started with Python because the concepts and the techniques you have learned here can be applied in every kind of application field. This could be web development, database access, desktop applications, scientific computations, network programming or game development. Without doubt you will be using variables, functions, conditionals, loops, and modules to mention a few, as the building blocks of your programs because whatever the purpose of your program is, you will always need these basic elements to build it up.

One advanced aspect that this book taught you was *classes*. Classes are often deemed as advanced not because their relatively complex syntax, but because it is difficult for a beginner to imagine scenarios where you would use a class as the main building block for your programs. Classes are mostly used when writing long programs, so they provide a way to have the most basic elements such as variables and functions more organized. You will mostly be able to solve your Python problems by simply using functions and simple statements, but whenever you're ready to build a larger program, classes are there for you to use. As you build programs, you will better understand how all the elements work together to make flawless scripts of code. That said it is generally a better idea to design your programs with classes in mind.

When talking about specialized functionalities, we have to look further away and call external modules and packages to use in our scripts. In this book we mentioned a few specialized modules such as *tkinter* which is used to produce graphical interfaces for your programs. We also looked at *sqlite3* which is used whenever you need to interact with a database be it local or on a remote server. You learned about regular expressions which are the tool to use when dealing with text processing. Using online resources, you will be able to discover on your own the modules that have the necessary functionalities to implement the special features of your programs. However, we could mention some of the modules that are used for each of the Python application fields.

If you're fond of web development, you would need one of the frameworks such as *Django*, *Pyramid*, *Flask* or *Bottle*. With Django being the most popular, all these frameworks will provide you a good ground where you don't have to start from scratch for building your web applications.

If you will be dealing with network interfaces and protocols, you might have to work with the *socket* library which provides a low-level networking interface. *Twisted* is also another event-driven network programming framework that you can use in Python for network engineering.

Python can also be used to build desktop applications. The *tkinter* library is one option you can use to build graphical interfaces with Python. Other libraries that are also used by the community are *wxWidgets* and *pyqt*. They all do the works, but one can build things with more clear code than the other, or it can offer a nicer look of the graphical interfaces they produce.

Python is also popular for game development and you can make modern high quality games using the specialized Python libraries. The *PyGame* package is a good library that will generally satisfy most of your needs. Pyglet, PyWeek and Panda3D are also other options to be considered.

Lastly, Python is great for scientific and numeric applications. We could mention the *SciPy* library here which is a collection of packages used for mathematics, science and engineering. *Pandas* is also a library used for data analysis and modelling. Python is also used to visualize data through its *matplotlib* library.

In almost any case you will be able to install these libraries by just executing the *pip* command in your computer command line just as you learned in chapter 8 of this book. There are some cases however, where you will need to download the packages from their websites and install them manually. Once you have installed them in on or the other way, packages and modules can be used following the same concepts that you learned in the book, and can suit almost any programming need that you have.

This is all about to get you started with Python. Python is truly a highly efficient and fun language. Good luck with any of the application fields you have chosen to handle with Python.

This book has found you because you have the ultimate potential.

It may be easy to think and feel that you are limited but the truth is you are more than what you have assumed you are. We have been there. We have been in such a situation: when giving up or settling with what is comfortable feels like the best choice. Luckily, the heart which is the dwelling place for passion has told us otherwise.

It was in 2014 when our team was created. Our compass was this – the dream of coming up with books that can spread knowledge and education about programming. The goal was to reach as many people across the world. For them to learn how to program and in the process, find solutions, perform mathematical calculations, show graphics and images, process and store data and much more. Our whole journey to make such dream come true has been very pivotal in our individual lives. We believe that a dream shared becomes a reality.

We want you to be part of this journey, of this wonderful reality. We want to make learning programming easy and fun for you. In addition, we want to open your eyes to the truth that programming can be a start-off point for more beautiful things in your life.

Programming may have this usual stereotype of being too geeky and too stressful. We would like to tell you that nowadays, we enjoy this lifestyle: surf-program-read-write-eat. How amazing is that? If you enjoy this kind of life, we assure you that nothing is impossible and that like us, you can also make programming a stepping stone to unlock your potential to solve problems, maximize solutions, and enjoy the life that you truly deserve.

This book has found you because you are at the brink of everything fantastic!

Thanks for reading!