EVAN LANE

PYTHON PROGRAMMING

A STEP BY STEP GUIDE

FOR BEGINNER'S TO CODE WITH PYTHON

amazon BEST SELLING BOOK

PYTHON MADE EASY

Python Programming

A Step by Step Beginner's Guide to Coding with Python in 7 Days or Less!

Evan Lane

Printed in the USA

Contents

Introduction

When it comes to coding, there are many options that you are able to choose from. But there are none that are as easy to use and learn for beginners as Python. The Python coding language is really easy to learn how to work with and it can provide you with a lot of power and versatility for the types of code that you will be able to work with.

Inside this guidebook, you will find all the steps that you need to get started with Python and how to go from beginner to proficient in no time at all. We will start with some of the basics of the Python language and how to get it installed onto your own computer system so that you can start working on code without having to mess around.

After the Python interpreter is all installed and ready to go on your computer, it'll be time to start learning how to code. We will talk Python syntax, how to work with the conditional statements, how to handle exceptions, how creating loops can save you a lot of time when working on control flow, and so much more.

Python is a user-friendly language, one that will teach you and help you as you go along. The examples in this book are for you to try out on your own Python program so work through them as you read the book. They are designed to give you a good idea of how the Python concepts work and what they mean but are by no means exhaustive examples.

Chapter 1

Python

There are many things that you can consider when getting started on your first project in coding. You need to figure out what kind of operating system you would like to use to write your code. You need to have a good idea of the types of code you would like to write as well as the end result so that you can concentrate on the code that you need to make that program that you want. And you need to pick out the type of programming language that you would like to work with.

Most beginners like to go with the Python coding language because it is easy to use, can be used across all of the platforms, and has a lot of versatility in the kinds of code that you are able to work with. Python is easily available, for free, and it will help to simplify the process of finding a solution or creating a program because you won't have to do a lot of nonsense you would otherwise.

In addition to being easy to read and write, one of the biggest advantages of using the Python language is that you are able to run it on any computer system that you want. You don't need to go out and get a new operating system, but if you are still looking for some options, consider going with Linux because Python is designed to work the best with this one, and it is already installed on Linux.

When Python was first developed during the Nineties, a team of volunteers ran the project. This is an open source project though which makes it easier for other programmers to get ahold of the Python base code and make adjustments as needed. This is why there are several versions of Python available. As issues arise with one of the versions or new developments occur, programmers are able to get ahold of the code and make updates and changes so you always have the very best and you can choose the version of Python that you would like to use.

While Python is considered a language for beginners, there are actually quite a few uses for it. Many of the websites that you find online will use Python code and sites like Google and YouTube were even created with some Python code. These are just a few of the many websites online that have been used with Python code from the government to business sites and so much more.

There are many benefits to choosing to go with the Python programming language compared to one of the others. Beginners like that this is an easy one to work with, a language that they can pick up on really quickly – and since it is based on the English language, they aren't going to be stuck trying to figure out what some of the words mean.

Many people choose to use the Python programming language as their beginning language, and you may become so familiar with how this language works that there will be no want to learn any of the others. Python is really simple and even beginners are able to catch on to the syntax when working on some of the code.

The Python project code is also a free and open source. You will not have to spend any money to get Python downloaded unless you choose to pick out an IDLE that will cost some money to get more features, but there are free versions of most IDLEs as well IDLE stands for Integrated Development and Learning Environment and it is integrated into the default Python package as well as some of the Linux distributions that use Python. In terms of Python being open source, this basically means that anyone can get the code and make improvements and changes, which is why there are so many great versions of Python on the market. You will be able to pick out which version of Python you would like to work with based on what you would like the program to do for you. Most people tend to use Python 2 or Python 3, depending on what is required of but it is worth bearing in mind that Python 3 is NOT an update to Python 2, rather it is a separate program and is not truly backward compatible. Therefore, trying to use Python 3 with a Python 2-based program will likely result in errors.

As a beginner, you will also appreciate that there is a really large community of support for Python. Python is one of the most popular coding languages, which means that there are people all over the world who choose to use this language for their needs. Because of this, you can easily go online and find answers to your questions, tutorials, and other information to help make your coding so much easier.

This programming language is able to combine with some of the other coding languages if you would like to do this. Python does not have the power to do everything, but when it combines with other programs, this issue can be fixed. It is possible to add together the Python programming language with some other coding languages, such as C++ and JavaScript, in order to get some of the other stuff that you would like done finished.

While there are many things that you are going to love about Python, it is important to understand that it is a beginner language, so some of the more complex things that you would like to do with this language may not be possible. But as a beginner, you probably won't be able to get to these options right at the start anyway and you can at least use Python as a stepping stone to learning one of the other languages. There are still many other things that you are able to do with Python so you should be able to create some of the programs that you want.

Overall, for a beginner, Python is one of the best coding languages to work with. It has a simple

syntax that you will catch on to quickly and it isn't hard to put this to work for you.

Working with Python

Once you decide that it is time to work with the Python language, it is time to get the interpreter and download it to your computer. There are a few different versions that you are able to work with and each is going to have different features based on what you would like the program to do for you. Go and visit the website www.python.org/downloads so that you can see which versions are there and then pick the one that you want to download. Remember that if you are using Linux, Python will already be downloaded for you.

In addition to downloading the actual interpreter for coding, you should take the time to pick out and set up a development environment for Python. This is basically the environment that you will write your codes on and work within and without it, it is hard to get your code to work properly. There is an IDLE that is available with the Python program and you should be able to download these at the same time. Many prefer text editors to the Python IDLE. Notepad or a similar product on Mac or Linux will work just fine as the text editor, but you may wish to get another such as Atom or Sublime Text.

Once you get these few things set up, you will be ready to go through and write out some of the code that you want. Python is pretty easy to use and we will take some time to work on code for the various features of programs that you want to write, but you do need to take the time to get all the software downloaded before getting started.

How to Install Python

Python is built into Mac OS and to Linux so only those on Windows need to install it.

1. If Python is not already on your Windows PC, open your browser and head over to the Python download page.
2. Decide which version of Python you wish to download and click on the download link. For most of this book, we will be using Python 2 so click the latest version
3. Now click the Download link on the page
4. At the bottom of that page, you will see a link for the Windows x86 MSI Installer – click on it to download it
5. Once the download has completed, locate it in your downloads and double-click the file
6. Click on Run
7. If you are the sole user of your PC, leave the option for Install for all Users selected. If others use your PC and you don't want Python on all their accounts, choose Install Just for Me and then click on Next
8. Choose where you want the download to go and click on Next
9. In the window, find where it says Add Python.exe to Path and click on the red x.
10. Click on the option for Will be Installed on Local Hard Drive
11. Click on Next
12. The installation will open a Command Prompt window and will install a package management tool called Pip. This is what will let you install the other Python packages that you want through PyPi – the Python Package Index
13. Once the process completes, click Finish

Add Python to the System Path Variable

Whether you need to do this or not will depend on the Python version you are running. If you chose Python 3, you can ignore this simply because this is incorporated into the new updates. If you opted for Python 2, you will have to follow these steps:

1. Open the Start menu
2. Type in the word Environment and then click on Edit the System Environment Variables
3. A window called System Properties will appear, click Environment Variables
4. At the bottom of the window is a section for System Variables – click New to make a new Python variable
5. Type in a name for the path and then enter the following code. Call it PythonPath for now: C:\Python27\;C:\Python27\Scripts;
6. Click on OK>OK>OK and the click the red x; this will save the changes and you will come out of the window

That sets Python up on your Windows PC.

Chapter 2

Python Fundamentals

There is no one way to code. Some codes are basic and will just need a line or two of code to tell Python what to do. Others can take up blocks of code to tell the interpreter how to behave. It doesn't necessarily matter what kind or length of code you'll be working with, there are going to be parts (Python fundamentals) that are similar with all of them. Learning these fundamentals is the key to learning to code your own programs. When you see the word "interpreter", we are referring to Python itself. When you download Python, you download everything that you need, including the text editor where you write your code and Python will automatically interpret everything you do.

The Keywords in Python

Like many programming languages, Python has a set of keywords that are strictly reserved for the compiler. They can't be used as identifiers. Using them as identifiers will result in errors.

Here's a list of keywords in the Python language.

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

As you read this book you'll come across these keywords. Keywords such as class, elif, for, while, if, raise and more.

Naming Identifiers

In a Python program, there are quite a few identifiers that you can work with. You will find that they have many different names and can go by things like variables, entities, classes, and functions. When you work on naming an identifier, you can use the same information and the same rules for all of them - this makes it a bit easier to remember what you should do. There are guidelines that help with naming different identifiers.

You have a lot of options when naming identifiers. You can use lowercase and uppercase letters in combination. You can also use the underscore symbol and numbers. Any combination of these are fine, just make sure that you don't start an identifier name with a number and that there aren't spaces in between the words if you use more than one to name the identifier. Examples include:

- *myClass*
- *var_1*
- *print_this_to_screen*

CamelCase is used by most programmers (depends on programmer) when naming identifiers. It's easier on the eyes and has been in practice for years. CamelCase is just a convention for naming identifiers. An example would be if you name a variable to hold a value for the *number of cars in a parking lot* for a program that requires this value. With CamelCase, you would capitalize the first letter of each word: *NumberOfCarsInLot* or *numberOfCarsInLot*.

Other programmers prefer the underscore symbol to separate words. This really boils down to your style of coding.

Other guidelines include picking an easy name to remember. This is helpful when you've been working on a program for a while or when you return to a program after a few days off. If you name it something that you won't remember later on, this could cause some issues.

Outside of these simple guidelines, you should be able to find plenty of names for your identifiers but, as stated before, you must not use reserved keywords for your identifiers. For example, you could not do this:

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
```

If you did, the result would be this:

SyntaxError: invalid syntax

You also cannot use special characters, such as:

- !
- @
- #
- $
- %
- And so on.

If you do something like this:

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
```

You would get this:

SyntaxError: invalid syntax

Control Flow in Python

One of the most important things in Python is control flow. This is the order the program will be executed in and this is regulated using function calls, loops, and conditional statements and, if they are not in the right order in your code, the program simply cannot execute. For the purposes of this section, we are going to concentrate on statements, which are nothing more than strings. A string is text that you want to be displayed or you want exported out of your Python program. Python knows that a piece of text is meant to be a string because you use either double (“”) or single (‘’) quote marks around the text. **IMPORTANT** – consistency is key – if you start the string with, let’s say, double quotes, you must end it with double quotes. Never mix your singles and doubles up because Python won’t know what is happening. Let’s look at some examples of common Python conditional statements:

If Statements

The syntax for the if statement is:

```
if condition:
    # do something
elif condition:
    # do something else
else:
    # do something else again
```

There isn’t a statement that will end the if statement and there is also a colon (:) at the end of each control flow statement. Python is reliant on the correct use of colons and indentation so that it can tell if it is in a specific code block or not.

For example:

```
if a == 1:
    print "a is 1, and changes to 2"
    a = 2
print "finished"
```

In this example, the initial print statement will only be executed if a is 1. The same goes for the a = 2 statement. However, the print “finished” statement will be executed, no matter what, when Python comes out of the if statement.

The conditions in an if statement may be anything so long as a Boolean value is returned.

For example, these are valid conditions:

- *a == 1*
- *b !*
- *c < 5=*

They are valid because each will return a Boolean value of True or False, dependent on whether the statement is true or false. You can use standard comparisons, such as:

- *equal - ==*
- *not equal - !*
- *less than or equal to - <=*
- *greater than or equal to - >=*

You can also use logical operators, like:

- and
- or
- not

Parentheses () are used as a way of isolating sections of the condition and this is to make sure that Python knows which order to execute the comparisons in.

Take this example:

```
if (a == 1 and b <= 3) or c > 3:
    # do something
```

Python now knows which order the comparisons must be run in.

For Loops

You will hear about loops whenever you hear about Python and the most common one is the for loop. The basic syntax is:

```
for the value in iterable:
    # do some things
```

The iterable can be whichever of the Python objects that can be iterated over and that includes tuples,

lists, strings, and dictionaries. Try inputting this into the editor:

```
In [1]: for x in [3, 1.2, 'a']:
   ...:     print x
   ...:
3
1.2
'a'
```

Because the colon was placed at the end of line 1, Python will automatically know that the next line is to be indented so you don't have to do that yourself. When you have typed the print statement in, remember to press the enter key twice so that Python knows you have completed the code.

One of the most common of all the for loop types is where a value goes in between two integers with a set of a specific size. We do this by using the range function. If a single value is provided, we get a list that ranges from 0 to the value of minus1:

A common type of for loop is one where the value should go between two integers with a specific set size. To do this, we can use the range function. If given a single value, it will give a list ranging from 0 to the value minus 1:

```
In [2]: range(10)
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If two values are given, these are the starting value and one added to the ending value:

```
In [3]: range(3, 12)
Out[3]: [3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Lastly, if a third number is provided, this will be the step size:

```
In [4]: range(2, 20, 2)
Out[4]: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The range function may be used in a for loop as the iterable.

While Loops

Like other computer languages, Python also has a while loop, which is pretty much like the for loop, with the exception that the iterations are defined by conditions and not iterators:

```
while condition:
```

```
    # do something
```

Have a look at this example:

```
In [1]: a = 0


In [2]: while a < 10:
   ...:     print a
   ...:     a += 1
   ...:
0
1
2
3
4
5
6
7
8
9
```

In this, the loop will be executed until a equals or goes above 10.

More on loops in further chapter.

Comments

Sometimes when you are writing out your code, you will need to include some comments. These are like little notes inside of the code that you, and the other programmers who look at the code, will be able to read through to understand what is going on in the different parts of the code. These will include a # symbol in front of the comment so that the compiler knows to just skip over that part and go on to the next part of the code.

You are able to add in as many of these comments as you would like to help explain the code that you are writing and to help it make sense. As long as you include the # sign in front of the comment that you are leaving, the compiler will leave it alone and move on to the next block of code.

Variables

Variables are another common thing used in Python code and are nothing more than storage locations. In fact, they are portions of the memory that are reserved for the storage of values and how they are stored is dependent on the data type of the variable. By assigning certain data types, you can store integers, characters, and decimals.

Variables can be named pretty much anything, but the name must begin with an underscore or a letter and they are case sensitive. To assign a value to a variable, we use the = sign.

In this example, the operand that is on the left of the operator (=) is the variable while the right operand is the value that is assigned to it:

```
#!/usr/bin/python

counter = 100          # integer assignment
miles   = 1000.0       # floating point
name    = "Brian"      # string

print counter
print miles
print name
```

Let's break this down:

- 100 – the value assigned to the variable called counter
- 1000.0 – the value assigned to the variable called miles
- Brian – the value assigned to the variable called name

This will be the ouput:

```
100
1000.0
Brian
```

Operators

Operators are pretty simple parts of your code, but you should still know how they work. There are a few you can use. For example, the arithmetic functions are great for helping you to add, divide, subtract, and multiply different parts of code together. There are assignment operators that will assign a specific value to your variable so that the compiler knows how to treat this. There are also comparison operators that will allow you to look at a few different pieces of code and then determine if they are similar or not and how the computer should react based on that information.

Let's look at the different types of operators.

The main types of operator in Python are:

- Arithmetic operators
- Boolean operators
- Relational operators

An operator can have one or two operands – the input argument of the operator. If the operator will only go with one operand, it is called a unary operator; two operands and it is called a binary operator. Plus and minus signs are both unary sign operators and addition and subtraction operators, depending on the situation:

```
>>> 2
2
>>> +2
2
>>>
```

The plus may be used to indicate a positive number, although this is not often used for this, while the minus will change the sign of a particular value:

```
>>> a = 1
>>> -a
-1
>>> -(-a)
1
```

Addition and multiplication operators are both binary operators, which means two operands are used:

```
>>> 3 * 3
9
```

```
>>> 3 + 3
6
```

The Assignment Operator

The assignment operator is = and this is what we use to assign values to variables. In math, = has another meaning altogether. When we use equations, the = operator is used as an equality operator = the left-hand side of the equation will be equal to the right:

```
>>> x = 1
>>> x
1
```

In this example, we are assigning the x variable a number:

```
>>> x = x + 1
>>> x
2
```

While this may not make any sense whatsoever in math, it does make sense in programming. What we have done is added 1 to the variable, making the right side equal to 2 and assigning 2 to x.

You can also assign one value to several variables:

```
>>> a = b = c = 4
>>> print a, b, c
4 4 4
```

Note this syntax error – this is because values cannot be assigned to literals.

```
>>> 3 = y
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Arithmetic Operators

These are the arithmetic operators in Python:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Floor division (//)

- Modulo (%)
- Power (**)

And these are the arithmetic operators in use in Python:

```
#!/usr/bin/python

# arithmetic.py

a = 10
b = 11
c = 12

add = a + b + c
sub = c - a
mult = a * b
div = c / 3

power = a ** 2

print add, sub, mult, div
print power
```

The division operator is a bit of a surprise because, usually, this will perform an integer division. That means the result is an integer but, if you wanted a result that was a bit more exact, you would use a floating point as an operand:

```
#!/usr/bin/python
# division.py

print 9 / 3
print 9 / 4
print 9 / 4.0
print 9 // 4.0
print 9 % 4
```

The floor division operator is used to find the floor of any value that is returned by true division:

print 9 % 4

The modulo operator will find the remainder when one number is divided by another. So, 9 % 4 equals 1 because 4 will go twice into 9 and leave 1 over.

Boolean Operators

There are three Boolean operators in Python – and, not, and or. These are used to perform logical operations and are normally used with the reserved keywords, if and while:

#!/usr/bin/python

 # andop.py

print True and True
print True and False
print False and True
print False and False

This is using the and operator which will evaluate true provided both of the operands are true.

The or operator will evaluate true only if one of the operands is true:

#!/usr/bin/python

 # orop.py

print True or True
print True or False
print False or True
print False or False

We use the negation operator to make True into False and False into True

#!/usr/bin/python

 # negation.py

print not False
print not True
print not (4 < 3)

Relational Operators

We use relational operators when we want to compare values and these will always give a Boolean value as a result. These are the relational operators:

- Strictly less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)
- Equal to (==)
- Not equal to (!= or <>)
- Object identity (is)
- Negated object identity (is not)

This example shows them in use:

```
>>> 3 < 4
True
>>> 4 == 3
False
>>> 4 >= 3
True
```

Relational operators are not just used for numbers, but can also be used for other things, even though, they may not always be terribly meaningful:

```
>>> "four" == "four"
True
>>> "a" > 4
True
>>> 'a' < 'b'
True
```

The basics of Python are found in all codes, ranging from basic to the more complicated code. Practice these basics so that the more complicated parts make sense later on.

Chapter 3

File Management

After understanding the basics of Python, it's time to learn how to work with files within Python. When you are working on code, you are basically creating something new, and you want to make sure that Python is storing this data in a way that will help you to gain access to it again later on. When it is saved inside of Python, you also want to be able to have that information display inside of the program when it is time.

Whenever you want to store programs, you will create a file, but there are times when you know you're going to want to reuse this block of code over again within a program. This chapter is going to take some time to address different files that you can work with Python.

When you create a new file in Python, you must always save it, similar to the way that you save a file in Word or in Excel. If you don't save it, all your hard work will be lost. When you write a new file in IDLE, open a new window by going to File>New File. TO save the file, go to File>Save As. It is best practice to save your files with the .py extension, just as you would save a Word file as .doc. Some of the things that we are going to explore in this chapter include:

- Creating a new file
- Moving a file
- Editing a file with some more code inside
- Closing a file.

Working with Files in Python

There is quite a bit to learn about working with files in Python but, as a beginner, this is absolutely the first place to start. If you don't know how to create a file, write to a file, open, save and move files, then your Python experience is not going to be an easy one. So, let's start with creating and writing text to files and the most basic program that all beginners start with – Hello, World!

To create a new file, you first need to input some text, so try this:

print('hello world')

Python is an object-oriented language, which means that the code is constructed around objects. These objects contain data and several methods that are needed to access the data and alter it. Once an object has been created, it will then be able to interact with any of the other objects in the code.

In the basic example above, we see just one object type, a string that says, 'hello world'. This string is nothing more than a character sequence that is encased in quote marks.

Strings can be written in three ways:

message1 = 'hello world'

message2 = "hello world"

message3 = """hello

hello

hello world"""

As I mentioned to you earlier, you cannot mix single and double quotes in the same string. For example, this is wrong:

message1 = "hello world'

message2 = 'hello world"

message3 = 'I don't like pickles'

Look at how many single quotes are in message 3. If that is going to work, the apostrophe in the word "don't" must be escaped so it isn't counted as a single quote:

message3 = 'I can\'t eat pickles'

Or you can rewrite it using double quotes to enclose the string:

message3 = "I don't like pickles"

In the final example above, we see triple quotes – *hello world""""*

These indicate that a string goes over more than a single line.

The print command will print an object in textual form and, when you combine it with a string, you get a statement. We use print in this way when we want to create information that must be acted on straight away. Sometimes though, you will want to create information that is to be saved, sent to another place or used by another program for more processing. In situations like this, the information needs to be sent to a file on your hard drive instead of to the "command output" pane. Input this program in the text editor and then save it with the name file-output.py.

 # file-output.py

f = open('helloworld.txt','w')

f.write('hello world')

f.close()

Where you see a hash (#) at the start of a line, this is a comment and the interpreter will ignore it. Recall, comments are used to make notes for yourself or for others who might be reading your code.

In this particular program, f is the file object and the file methods are open, close, and write. What that means is, those methods each do something to that object, in this case, we have defined it as a .txt file. You will come to know this as a method – a piece of code that performs a specific action.

f is the name we have given to a variable – you can name it whatever you want, provided you use the naming rules – lower and uppercase letters and underscores but no special characters and definitely no reserved keywords. If you attempted to name your variable "print", for example, your program would fail because print is a reserved keyword. Don't forget, variable names are also case sensitive and that means the following three would all be different variables:

- FOOBAR
- Foobar
- foobar

Back to the file; when you run this, the "open" method will inform your computer that a new file needs to be created. The file is to be called helloword.txt, it will be a text file and it is to be saved in the same folder that you saved file-output.py. We use the parameter "w" to indicate that we are going

to use Python to write new content to the file.

NOTE – because the parameter and the file name are both enclosed in single quotes, they have both been stored as strings. If you omit these quote marks, your program will fail.

Your program writes a message to the file on the next line, a string of characters that reads "hello world" and then closes that file.

In your editor, execute the program. You won't see anything in the Command Output pane but you will see a message that will read something along these lines if you use Mac or Linux:

`/usr/bin/python file-output.py` returned 0.

Or like this in Windows:

'C:\Python27\Python.exe file-output.py' returned 0.

What this message is telling you is that your program was successful in executing. Open the file by selecting File>Open>File and selecting your file, you should see the message:

Hello World!

Text files do not contain a whole lot of formatting information and, as such, they are small and can easily be exchanged between platforms – Windows to Mac, or to Linux, or the other way around – and they can be read by those using different text editors to the one the file was written in.

Reading from Text Files

Python also contains a number of methods that allow you to retrieve information from a file. Input the following program into your editor and save it with the name file-input.py. When you execute it by clicking on Run, the text file you just created will open, the message will be read from it and the message will be printed to the command output pane:

```
# file-input.py

f = open('helloworld.txt','r')

message = f.read()

print(message)

f.close()
```

Here, we have used the parameter "r" to indicate that we want to open and read from a file.

Parameters allow you to choose from all the different options offered by the method. Let's say that you train your dog to bark for a treat – once for beef, twice for chicken. The flavor of that snack is the parameter in our code. Each method differs in what parameters will be accepted.

Read is a file method. The file contents, in this case, one line of text, will be copied to "message", the name we have given to the string, and print will send this to the command output pane.

Appending to a Text File That Already Exists

You can also open a file that has already been created and add in some more. Be aware – if you open any file and then use the "w", or write method, the contents of the file will be over-written so take care what you are doing. This isn't a problem when you create new files, or when you want the contents over-written but it can cause huge problems when you want to compile large data sets into one single file or create event logs. Instead, we use the append method, or "a".

Input this program into the editor and then save it with the name file-append.py. Now, when this program is run, the helloworld.txt file you created will open and a second "hello world" will be appended to the file. '\n' indicates a new line:

```
# file-append.py

f = open('helloworld.txt','a')

f.write('\n' + 'hello world')

f.close()
```

When this has been run, open the text file called helloworld.txt. What do you see? Close it, run append.py another two or three times and then open the helloworld.txt file again. You should see that the hello world message is repeated as many times as you ran the file.

Moving Files

In Python, moving a file is actually renaming it and it is incredibly easy. This is down to a useful module named shutil, a module that contains a function called move. That function does exactly what it says on the tin – it will move a file or a directory from one place to another. Have a look at this simple example:

```
    import shutil

def move(src, dest):

  shutil.move(src, dest)
```

See? Dead simple. The move function will take the source directory r file and move it to the new directory or file:

shutil.copy vs os.rename

If the directory or file is located on the current local file system, shutil.move will use os.rename to move it. Otherwise, shutil.copy2 is used to copy the directory or file to the new locations and will then delete the source.

So, why do we use shutil.move instead of using os.rename? The answer to that is above – shutil.move looks after any cases where files are not located on current local file systems and copies directories to new destinations. If there are any exceptions thrown up by os.rename, shutil.move will handle them correctly so you don't need to spend time worrying about them.

shutil.move will throw up its own exceptions, shutil.Error and this happens when the destination directory or file is already in existence or if you are trying to copy the source directory or file into or onto itself.

That is how simple it is to move a file. The only thing you need to note here is that, if the file is on the current file system, the call to move function will be instant, whereas if you are moving the file to another destination or drive, it will take a little longer.

Working with Binary Files

What you know as a file is not quite what a file is in Python. For example, in Word, a file is any item that is created, edited or manipulated by the user, such as images, executables, text documents, and so on. These folders tend to be organized into folders so they are easy to find again.

In Python, files come under just two categories – text and binary – the difference between this is very important.

Text files are sequences of lines and each line will contain a sequence of characters – this is known as syntax or code as you already know by now. Each of the lines of code ends with the EOL character or End of Line. There are several of these but the ones used most often are the comma or a newline, which tell the interpreter that a new line is starting.

A binary file, on the other hand, is a file that isn't a text file. Binary files can't be processed by just any application; it must be an application that knows the structure of the file and, more importantly, understands it. In layman's terms, binary files can only be processed by those applications that know how to read and to interpret binary.

Writing to a Binary File

It is very simple to write to a binary file in Python and one way is to open the file in binary mode and then write the data into the file as strings of hexadecimal characters.

Have a look at this example:

output_file = open("myfile.bin","wb")

output_file.write(b"\x0a\x1b\x2c")

output_file.write(b"\x3d\x4e\x5f")

output_file.close()

To read the contents of the binary file generated, in Linux we use the hex.dump command:

hexdump -C myfile.bin

The -C option also tells hex.dump to show the file contents in hexadecimal form and as an ASCII string.

The output of that command would be:

00000000 0a 1b 2c 3d 4e 5f |..,=N_|

00000006

The dot that you see on the right-hand side is representative of a byte that hexdump cannot interpret as an ASCII character or a byte that contains the ASCII code for the dot character.

There is one problem with this; this is not an easy way to go when we want several objects written into a binary file. For example, what if we wanted to add strings, integer values or a list into the file? How could we possibly read these values when we wanted to? The long way around would be to write metadata that specified the structure of each addition and this is incredibly advanced stuff – not something you need to be worrying about at this stage. Luckily, there is an effortless way because Python has one module that will do all this work for us. It's a module called pickle and it lets us convert objects to bitstreams, which we can then store in files and use to construct the original object later. Pickle cannot do this for all data types but it can do it for most of what you will use in Python.

I expect this all sounds a little on the complicated side so let's look at a few examples which will show you just how easy pickle makes things.

To write an object straight to a binary file (for write, we use the word dump), input this command:

```
import pickle

output_file = open("myfile.bin", "wb")

myint = 42
mystring = "Hello, world!"
mylist = ["spoon", "fork", "knife"]
mydict = { "name": "Simon", "job": "Doctor" }

pickle.dump(myint, output_file)
pickle.dump(mystring, output_file)
pickle.dump(mylist, output_file)
pickle.dump(mydict, output_file)
output_file.close()
```

What you see generated in the binary file will be different depending on whether you are using Python 2 or Python 3 and the reason for that is because the way pickle works has changed as time has gone by. The recommendation is to use Python 3 when you use pickle so there are no compatibility issues.

We can load or retrieve the original objects from the file called myfile.bin, in the exact same order they were dumped into it:

```
import pickle

input_file = open("myfile.bin", "rb")

myint = pickle.load(input_file)
mystring = pickle.load(input_file)
mylist = pickle.load(input_file)
mydict = pickle.load(input_file)

print("myint = %s" % myint)
print("mystring = %s" % mystring)
print("mylist = %s" % mylist)
print("mydict = %s" % mydict)
```

input_file.close()

The program output shows you that the original objects have been retrieved correctly from within the binary file:

myint = 42
mystring = Hello, world!
mylist = ['spoon', 'fork', 'knife']
mydict = {'job': 'Doctor', 'name': 'Simon'}

That completes this overview of working with text and binary files.

Chapter 4

Expressions and How They Function in Python

Python programming can be a great experience. You can create a program that is completely unique. One of the things that'll make your life easier is the Python library. When you use this library, you are going to be working with mostly regular expressions, which will be responsible for executing any of the tasks that you would like without having to deal with glitches and sometimes they will be able to handle the searches that you may want to do. As you progress with Python, there will be times when regular expressions can be used to filter out text and to check if some strings of text match up the way you want.

Regular expressions are sequences of characters, that are used to match patterns of text or to find sets of strings or other strings. This is done through a special syntax that is held in a pattern and fully supported by a Python module called 're'. The basic syntax for a regular expression is:

```
match = re.search(pat, str)
```

The method called re.search() will take a string and a regular expression pattern and will look in the string for the specific pattern. If the pattern is found, search() will return an object called Match; if there is no match, None will be returned. Searches are usually followed by an if-statement, used to test the success of the search. Have a look at the example below – we are searching for a pattern of 'word:' immediately followed by another word of three letters:

```
str = 'an example word:dog!!'
match = re.search(r'word:\w\w\w', str)
# the If-statement after the search() will test if it succeeded
 if match:
  print 'found', match.group() ## 'found word:dog'
 else:
  print 'did not find'
```

The code that reads match=re.search(pat, str) will store the result of the search in a variable called "match". The if-statement will then test that match. If the match is true, the search will have been successful and match.group() will show the matching text, in this case, 'word:dog'. If there is no match, None is returned and there will be no matching text.

Note – the string begins with the letter 'r' which denotes that this is a raw string. It is recommended that raw strings are used with regular expressions, rather than a regular string because, that way, Python will not interpret any of the backslashes or special characters that may be in the string, thus passing through them to a regular expression.

In more basic terms, where you see a pattern such as "\n\w", for example, it will not be interpreted and can, in fact, be written r "\n\w" rather than \\n\\w, as other languages write it – much easier and better to read, don't you think?

Basic Patterns

Just to demonstrate how powerful the regular expression is, they don't just work with fixed characters; they can also specify patterns. These are the most basic patterns, each of which matches a single character (char):

- **a, X, 9, <** -- all ordinary characters will be an exact match of themselves.
- . (a period) – will match up to any single character with the exception of a newline '\n'
- \w -- (lowercase w) -- matches "word" characters: a-z, A-Z, 0-9, _ (digits, letters, underscores). Only single characters are matched, not entire words.
- \W (upper case W) – matches characters that are non-word.
- \b – this is the boundary between non-word and word characters.
- \s -- (lowercase s) – matches a single whitespace character - newline, space, tab, return, form (\n, \t, \r, \f).
- \S (upper case S) – matches characters that are non-whitespace.
- \t, \n, \r -- tab, newline, return.
- \d -- decimal digits [0-9].
- ^ = start, $ = end – matches the beginning or the end of the string.

Basic Examples

Like everything in Python, regular expressions follow a basic set of rules:

- The search will work through the string, from beginning to end, and will only stop when the first match is found
- The whole pattern has to be matched but not the whole string
- If match = re.search(pat, str) succeeds, the match will not be None. Match.group() indicates the matching text.

Have a look at the following example, which explains these rules:

Search for the pattern 'aaa' in string 'caaat'.
All the pattern must match, but it can show up anywhere.
On success, match.group() is the matched text.
match = re.search(r'aaa', 'caaat') => found, match.group() == "aaa"
match = re.search(r'ats', 'caaat') => not found, match == None

. = any char but \n
match = re.search(r'..g', 'caaat') => found, match.group() == "iig"

\d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g') => found, match.group() == "123"
match = re.search(r'\w\w\w', '@@abcd!!') => found, match.group() == "abc"

Now, whenever you want to make a regular expression and get it to work with the program you are working on, make sure that you are opening the Python library and import the right one – type in *import re*. A great place to get this done is when you are getting things organized - at the beginning of your program. When you get a new program to open, you can go through and import the expressions that you will need right out of the library. This helps keep the code looking nice (organized and professional) and gets the expressions set up right away.

It also helps to keep the code organized so that others who read through your program can see these expressions right from the beginning.

As you look through the Python library, you will notice that there are quite a few expressions. It may take a bit of time for you to understand which ones will help your program. Often, these expressions are used along with statements so that the statements execute properly. Take a bit of time to look through some of the different expressions that are found in the Python library – get familiar with a few that may potentially help your program(s).

Expressions will be able to help you do a wide variety of actions inside of your code. Let's look at how some of these can work so you can implement them into your own Python programs.

Using the Methods of Regular Expressions

Methods. We saw some of these in the above examples and we are now going to look at them in more detail to understand how they work and what they do. There are several methods associated with regular expressions, often called queries and this is because a method is used to ask the expression to do something or look for something that you specified. There are several methods that you can work with and it is up to you to decide which one, or ones, are right for the action you want to happen.

Some of these methods include:

- re.match()
- re.search()
- re.findall()

Let's look at each one in more detail, with working examples:

re.match(pattern, string):

This method will find matches that happen at the beginning of a string. For example, if you had a string "AV Analytics AV" and you used match() to find the pattern AV, it would find the match. However, if you used it to look for the pattern "Analytics", it would not match because that pattern is not at the beginning of the string. See for yourself; input this example into your text editor now:

```
import re

result = re.match(r'AV', 'AV Analytics Vahnya AV')

print result
```

Output:

```
<_sre.SRE_Match object at 0x0000000009BE4370>
```

This shows that the pattern has, indeed, been found. If we wanted to print the string that matched, we would use the method group, helpful for returning the matching string. Don't forget to put 'r' at the beginning of the pattern string to indicate a raw string:

```
result = re.match(r'AV', 'AV Analytics Vahnya AV')

print result.group(0)
```

Output:

AV

Now do the same thing but this time, look for the pattern Analytics in the same string. Because the string does not begin with Analytics, there should not be a match. Try it and see:

result = re.match(r'Analytics', 'AV Analytics Vahnya AV')

print result

Output:

None

We also use the methods start() and end() to determine the beginning and the end position of any matching text:

result = re.match(r'AV', 'AV Analytics Vahnya AV')

print result.start()

print result.end()

Output:

0

2

Here, you can see the beginning and the end position of the pattern AV in the specified string; this can help an awful lot when you are manipulating the string.

re.search(pattern, string):

This method is similar to the one above but we are not restricted to finding only those matches at the start of a string. Try using this method to search for the pattern Analytics. You should get a positive result this time:

result = re.search(r'Analytics', 'AV Analytics Vahnya AV')

print result.group(0)

Output:

Analytics

The search() method can find the specified pattern anywhere in the string but only the first match will be returned.

re.findall (pattern, string):

Sometimes, you might want to find all the matches. This method is not restricted in any way so, if we were to use it to look for the pattern AV in the string, it will return all instances of it, in this case, two:

result = re.findall(r'AV', 'AV Analytics Vahnya AV')

print result

Output:

['AV', 'AV']

A Word on Functions

Python functions are used as a way of dividing our code into blocks and this lets us put our code into some kind of order, to make it easier to read. It also means that we can reuse that code and save a good deal of time.

How to Write a Python Function

The blocks of code that we talked about are written like this:

block_hehttps://www.abc.org/en/Functionsad:

1st block line

2nd block line

...

And the output would be

File "<stdin>", line 1

block_head:

^

SyntaxError: invalid syntax

We use the keyword, def, to define a Python function, then the name of the function, which is also the name of the block. For example:

def my_function():

print("Hello From This Function!")

Functions can receive arguments, which is a variable that is passed to the function. For example:

def my_function_with_args(username, greeting):

print("Hello, %s , From this Function!, I wish you %s"%(username, greeting))

Functions can also return values, by using the keyword called return, as such:

def sum_two_numbers(a, b):

return a + b

How to Call a Function in Python?

To call a function, all you do is write the name of the function followed by open and close parentheses – (). Inside those parentheses go the arguments. For example, let's have a go at calling the functions we wrote above:

```
# Define the 3 functions
def this_function():
    print("Hello From This Function!")

def this_function_with_args(username, greeting):
    print("Hello, %s , From This Function!, I wish you %s"%(username, greeting))

def sum_two_numbers(a, b):
    return a + b

# print(a nice greeting)
this_function()

#prints - "Hello, Billy Boy, From this Function!, I wish you a fantastic year!"
my_function_with_args("Billy Boy", "a fantastic year!")

# after this line x holds the value 3!
x = sum_two_numbers(1,2)
```

and the output is

```
Hello From This Function!
Hello, Billy Boy, From This Function!, I wish you a fantastic year!
```

Chapter 5

How to Work with Conditional Statements

There are times when you'll be working on a program where you want it to make its own decisions based on the conditions that you set in the code. You can choose how the code is going to react based on the input that the user places inputs, which can make the code much more interactive than what we have with some of the basic codes.

In this chapter, we are going to look at several of the different conditional statements that you can use like the **if** statement, the **if else** statement, and the **elif** statement.

If Statements

The if statement is the most basic conditional statement. The if statement relies on a true or false decision (a binary decision). You will get to set conditions that you would like to be considered true inside of your code, and then your user will be able to input their binary answer.

If the answer that the user places in is true, based on the conditions set, a predetermined message (display output) will show up on the screen so that the user can continue. However, if your program determines that the input from your user is false based on the conditions, the program will just end and won't let the user go any further into the program. You can also have a predetermined message for a false condition.

Here's an example of the if statement:

```
age = int(input("Enter your age:"))

if (age <=18):

    print("You are not eligible for voting, try next election!")

print("Program ends")
```

What is this code trying to tell us?

If your user is on the site or program, and they try to put in an age that is under 18, they are going to meet the conditions that you set. The program will display "You are not eligible for voting, try next election!" if the condition age <= 18 is met. In words, if the user inputs an age equal to, or less than, 18 then the condition has been met.

What happens if the user places an answer that is above 18 into the system?

If the user of your program puts in an answer that is considered false, such as one that is 18 or above, the compiler is not going to know what to do since this is a simple if statement with one condition. The compiler will keep the screen blank if you haven't set up an output in case the answer is false.

No one wants to use a program that just goes to a blank screen when they put in their age. There needs to be something, another message, or another way to proceed through the program, to handle this issue.

This is where the **if else** statement comes in. The if else statement will allow us to accept inputs greater than 18.

If Else Statements

With the if else statement, you will be able to set up a statement that will occur if the user input ends up being true, but you can also set up a separate statement if the user input is found to be false. With the example above, we would be able to use the if else statement to add in a condition if the user puts in an age of 30, or an age higher than 18. This ensures that the user is going to get an answer regardless of how they enter their input.

Let's look at how the if else statement would look in Python.

```
age = int(input("Enter your age:"))

if (age <=18)

    print("You are not eligible for voting, try next election!")

else

    print("Congratulations! You are eligible to vote. Check out your local polling station to find out more information!")

print("Program ends")
```

Now, if the user places their age as 18 or younger, you will get the first statement, the statement "You are not eligible for voting, try next election!" will display for the user. But, it also works the other way this time too. If your user says that they are older than 18 and puts in any number that is 18 or above, they are going to get the second condition to be true and the second message you wrote out "Congratulations! You are eligible to vote. Check out your local polling station to find out more information!" will display.

There are many times when you'll utilize these powerful condition statements in Python. You can expect to use them any time that a user is able to place in more than one answer.

Testing Multiple Conditions

If you just wanted to test a single condition, the if-else statement is just fine but what if you wanted to test out several conditions? This is where we use something called the Elif statement. This is short for Else-If and the basic syntax is:

```
if condition1 = True:
    execute code1
```

```
elif condition2 = True:
    execute code2
else:
    execute code3
```

To put that in plain language, if condition1 turns out to be true then code 1 should be executed. Else, if condition2 turns out to be true then code2 is executed. If neither condition is true, then code3 is to be executed.
You can use as many elif statements as you want in a Python program; provided you close with an else statement, then you can test as many conditions as you want using as many elif statements as it takes.

Let's try and make this simpler with an example:

```
x = 5
if x == 5:
    print "Great, X is exactly five!"
elif x > 5:
    print "X is more than five!"
else:
    print "X is less than five!"
```

If you were to change x to a value of 6, the output would change too. However, if you reduced it to 4, the third code block that is under the else statement would come into play. Try that in your editor and then ask yourself, just what is happening here?

Let's look at that:

- First, Python will check to see if the value given to x is equal to 5 exactly, as we put in the first statement.
- If it is exactly equal, the code in the first if statement is executed and the Python exits out of the program – mission complete.
- If x is not equal to the value of 5 then Python will move on to the next elif statement. Now, it will check the value of x to see if it is more than 5
- If x is greater than 5, the second block in the elif statement is executed and Python comes out of the program.
- However, if neither of these conditions can be met, i.e. x isn't equal to or more than 5, Python will show the output from the third statement.

Does that make more sense?

The if else statement is a condition statement heavily used in Python. Creating a robust program, one that can handle multiple types of inputs, will help your program be more professional and user-friendly.

How to Work with Elif Statements

What if you want to display a list of choices on the screen and let the user pick from one of those?

The if statement and the if else statement can't help you there.

Enter elif statements.

The elif statement can handle this in a straightforward manner. You can write out as many of these statements as you would like, as long as they are written and placed into the code properly. Sometimes you will write out the code so that it only has two options for the elif statement, but often you'll want more options too.

Let's look at how to work with the elif statement so we understand what is going on and can see for ourselves how this is different from the other two conditional statements:

```
Print("Let's enjoy a Pizza! Ok, let's go inside Pizzahut!")

print("Waiter, Please select Pizza of your choice from the menu")

pizzachoice = int(input("Please enter your choice of Pizza:"))

if pizzachoice == 1:

    print('I want to enjoy a pizza napoletana')

elif pizzachoice == 2:

    print('I want to enjoy a pizza rustica')

elif pizzachoice == 3:

    print('I want to enjoy a pizza capricciosa')

else:

    print("Sorry, I do not want any of the listed pizza's, please bring a Coca Cola for me.")
```

With this condition statement, your user will use the program and see what choices the elif statement is allowing them to have (the code is behind the scenes of course). They can then read through the choices and pick out the number option they would like the most.

For example, if they want to eat a pizza rustica, they would need to choose number 2 to make this

happen. This example just had three pizza options, plus a break that caught all the other answers if the user decides that they don't like the pizza options provided. You can add as many of options as you would like, such as ten pizza choices if you wanted.

Python conditional statements will add a lot of versatility to your programs. Rather than just picking one statement that can come up on the screen, you can set up your program so that it can interact with the user while they use the program. The if statements are a great way to get some practice with conditional statements, but the if else and the elif statements are not much more difficult to put into action and they too can add a great deal power to your coding. As a beginner, you'll find that using these conditional statements can make a lot of changes to the programs you'll be working on and can simplify what you have to write out while still getting your program to interact with others.

Chapter 6

The Importance of Classes and Objects

Python, as you now know, is an OOP, or an object-oriented program language. Because of this, there is a Python construct called a class and classes are what we use to structure the program in a specific way. When we use classes, we bring consistency to the program, allowing it to be used in a better way, making the code cleaner and easier to read.

In this chapter, we are going to look at classes and objects and, to do that, we need to go back over modules. Get your editor ready to type in the examples as we go through.

Modules = Dictionaries

Well, not exactly. A dictionary is created and then used as a way of mapping one thing to another thing. Let's say you had a dictionary with a key called "oranges"; now let's say you wanted to get that dictionary; this is what you would do:

```
mystuff = {orange: "I AM ORANGES!"}

print mystuff[orange]
```

For now, keep "get x from y" in your head. Now we are going to look at modules. We already went over some in this book so you should have a good idea of what they are:

- First and foremost, a module is a file that has some variables or functions in it
- That module, or file, is imported
- The variables or functions that are in the module can be accessed by using the dot (.) operator

Ok, so let's assume that we have a module that we have called mystuff.py. We put a function in it that we call oranges. Here is that module:

```
# this will go in mystuff.py

def orange():

    print "I AM ORANGES!"
```

Now that we have the code, we can use this module with the import statement and then we can access the function called orange:

import mystuff

mystuff.orange()

We could also add in a variable named mandarin:

def orange():

print "I AM ORANGES!"

 # this is only a variable
Mandarin = "The juiciest fruit about"

We access that in the same way:

import mystuff

mystuff.orange()

print mystuff.mandarin

Now go back to the dictionary and you should begin to see how this is like using the dictionary but with different syntax. Let's compare them:

mystuff['orange'] # get orange from dict

mystuff.orange() # get orange from the module

mystuff.mandarin # the same thing, it's only a variable

What this means is that we have a pattern in Python:

- Take the key is equal to a value-style of container
- We get something out of this by using the name of the key

With the dictionary, the key will be a string and the syntax will be [key] while, with the module, the key is the identifier and the syntax will be .key. Aside from that, they are pretty much the same.

Classes are Similar to Modules

Think of a module as a dictionary that is specialized. You can store code in it that you can get to, or access, using the dot (.) operator. Python also contains something else that does something similar – it's called a class. A class is a way in which we take a group of functions and put them in a container. This is so that you can use the dot operator to access them.

If you were to create a class, similar to the module called mystuff, you would do something along the lines of this:

```
class MyStuff(object):

    def __init__(self):
        self.mandarin = "even juicier than oranges"

    def orange(self):

        print "I AM A CLASS ORANGE!"
```

Ok so, compared to the modules, that probably looks a little complicated and there is quite a lot going on here. But, to be fair, you should be able to see that this is a little like a mini module, with MyStuff containing an orange() function. What may well be confusing you is the function called _init_() and the way we used self.mandarin to set an instance variable called mandarin.

So, why do we use a class instead of a module? Simple – the MyStuff class can be used to make many classes, as many as you want, and none of them will interfere with any of the others. With the module, on the other hand, you can only import one for the whole program.

Before you can truly understand this, we need to look at objects, at what they are and how we work with MyStuff class in the same way as we do mystuff.py module.

Objects Are Similar to Import

If we can look at a class as being similar to a mini module, there must be some concept that is similar for import but for classes. There is and that concept is named “instantiate”. This is just a fancy smart way of saying “create”. When a class is instantiated, we get an object.

To create or instantiate a class, we call the class as if it were a function, like this:

thing = MyStuff()

thing.orange()

print thing.mandarin

Line 1 is the operation to instantiate, as you can see, similar to the way we call a function. However, Python is working behind the scenes to come up with a coordinated sequence of events. Using the code above for MyStuff, here’s what Python does:

- It looks for MyStuff() and it can see that it is a class that you defined
- It then creates an object, an empty one, suing the functions that you specified, using def, in the class
- Then it will look to see if there is a _init_function
- If you included the _init_ function, it will call it to initialize the new and empty object
- In the _init_function, there is an extra variable called self – this is the empty object made by Python. You can set variables n this variable in the same way that you do a dictionary, module or any other object
- In the case of the example in this chapter, self.mandarin has been set to song lyrics and has been initialized.
- Python will now take this new object and assign it to the variable called thing so that you can get to work with it

Classes are like a set of blueprints, a definition used to create mini modules. To make or create a mini module and import it, all at the same time, we instantiate it, which just means we are creating an object from a class. The result is a mini module called an object which is them assigned to a variable.

Getting a Thing from a Thing

Now we have three different ways to get a thing from a thing;

```
# dict style
mystuff['oranges']

# module style
mystuff.oranges()
print mystuff.mandarin

# class style
thing = MyStuff()
thing.oranges()
print thing.mandarin
```

By now you should be starting to see how similar the three key=value container types are so, rather than trying to anticipate what questions you may have, we're going to look at another example, which should bring it all together for you. Type the following code into your editor:

```
class Song(object):

    def __init__(self, lyrics):
        self.lyrics = lyrics

    def sing_me_this_song(self):
        for line in self.lyrics:
            print line

happy_bday = Song(["Happy birthday to you",
                   "I hope your dreams all come true",
                   "and when you blow out the candles"])

bulls_on_parade = Song(["think about me too",
                        "At least I hope you do!"])
```

happy_bday.sing_me_this_song()

dogs_on_parade.sing_me_this_song()

You should see this as the output:

$ python ex40.py

Happy birthday to you

I hope your dreams all come true

And when you blow out the candles

Think about me too

At least I hope you do!

That, in a nutshell, is classes and objects in their most basic form.

Chapter 7

Exception Handling

In this chapter, we are going to look at the way in which Python handles errors in your code with exceptions. First, what is an exception? It is nothing more than an error which occurs when the program is being executed. When the error happens, Python will generate something called an exception. This can be handled and this means your program will not crash. Make sense? We use exceptions because they are a convenient and easy way for handling special conditions and errors in your program. If you think your program contains code that may produce errors, you would use exception handling.

Raising an Exception

To raise an exception on your program, you use the “raise exception statement”. For example:

>>> raise NameError()

When you raise an exception, you stop the current execution of the code and return the exception backward until it can be handled.

Exception Errors

These are the most common exception errors used in Python:

- **IOError –** raised when the file can't be opened
- **ImportError –** raised when Python can't locate the specified module
- **ValueError –** raised when built-in functions or operations receive arguments that are of the correct type but the wrong value
- **KeyboardInterrupt –** raised when the interrupt key on the keyboard is hit by the user – usually the Delete key or CTRL+C
- **EOFError –** raised when a built-in function – either raw_input() or (input() – comes to an EOF – End Of File condition without having read any data first

Examples of Exception Errors

Now that we know what these errors mean, we can look at a few examples:

```
except IOError:
    print('An error occurred when attempting to read the file.')

except ValueError:
    print('Non-numeric data that is found in the file.')

except ImportError:
    print "NO module was found"

except EOFError:
    print(Where did the EOF come from?')

except KeyboardInterrupt:
    print('You stopped the operation.')

except:
    print('An error occurred.')
```

Setting Up Exception Handling Blocks

In order to use exception handling, first you must have a “catch-all except” clause and we use the reserved keywords, “except” and “try” to catch the exceptions.

try-except [exception-name] blocks

For exception names, see the section above.

The code that is included in the try clause is executed, one statement at a time, in order. If an exception happens, the remainder of the block will be ignored and Python will execute the except clause.

```
try:
   a few statements here
except:
   exception handling
```

Let’s have a look at an example of this:

```
try:
   print 1/0

except ZeroDivisionError:
   print "Don’t be daft, you can't divide by zero."
```

In the example above, your Python program is going to show an error since you are attempting to divide up a number by zero (mathematically impossible). In Python, this isn’t something that’s allowed. Now, you can imagine how messy the default output will be for the end user. The end user will have no idea what’s happening if the program is left as is. The good news is that you can show the user what error is occurring. A user-friendly message is often preferred.

Most of these errors are going to be simple, such as accidently typing in a 0 instead of a 10. In such a scenario, you can add a new message so that your user has a clue and can make their changes to move along with the program.

Here is an example of how to change up the message so that your error message is easier on the user:

```
x = 10
```

```
y = 0

result = 0

try:

    result = x/y

    print(result)

#the following is an exception used from the library:

except ZeroDivisionError:

    print("You are trying to divide by zero – an impossible task")
```

With this example, you'll see that we are raising the same exception as above, but there is one simple change. The program is still going to end up with the result that you have an error, but rather than sending over all that messy default output display your user is going to get a much simpler message of "You are trying to divide by zero – an impossible task". It is simple, sweet, and to the point so that your user doesn't have to guess what is going on.

So how does it work?

Error handling is carried out through exceptions, each of which is caught in a try block and then handled in the except block. If Python comes against an error, the execution of the try block code is stopped and passed to the except block.

As well as using an except block after a try block, we can also use another block called finally. The code in this block is executed, no matter whether an exception is raised or not.

Let's look at another example. This time, we are going to write some code that shows us what will happen if we don't use error handling in our program. The program is going to ask a user to input any number from 1 to 10 and will then print that number:

```
number = int(raw_input("Enter a number from 1 - 10"))
print "you input number", number
```

Provided the user does enter a number, the program will work fine but what if no number is entered? What if the user put something else instead, like a string?

```
Enter a number from 1 - 10
```

hello

As you can see from your editor, an error is thrown up when a string is entered (the string is “hello”) You should see something like this:

Traceback (most recent call last):

File "enter_number.py", line 1, in

number = int(raw_input("Enter a number from 1 - 10 "))

ValueError: invalid literal for int() with base 10: 'hello'

As we know from earlier, ValueError is a type of exception. Let’s look at how to use exception handling to make the previous program work right:

```
import sys

print "Let's fix the last code using exception handling"

try:
   number = int(raw_input("Enter a number from 1 - 10
"))

except ValueError:
   print "Err.. numbers only"
   sys.exit()
```

Now if the program is run, and the user inputs a string instead of the number, you would get a different output.

Now let’s fix the last code again using exception handling:

Enter a number from 1 - 10
hello
Err.. numbers only

Try ... except ... else clause

When you use an else clause inside a try, except statement, it has to follow all of the except clauses. This makes it useful for a piece of code that is to be executed if the try clause doesn’t raise any

exceptions.

```
try:
    data = something_that_could_go_wrong

except IOError:
    handle_the_exception_error

else:
    doing_some_different_exception_handling
```

Any exception in the else clause cannot be handled by any preceding except clause.

You must also make sure that the else clause runs before the finally block runs.

Try ... finally clause

This is an optional clause only intended for use in defining cleanup actions that have to be executed no matter what the circumstance:

```
try:
   raise KeyboardInterrupt
finally:
   print 'Goodbye, world!'
...
Goodbye, world!
```

KeyboardInterrupt

The finally clause must always be executed before you leave the try statement, regardless of whether any exception has happened or not.

Remember; if an exception type is not specified on the except line, all exceptions will be caught and this is not a good idea. If this happens, your program won't take any notice of any unexpected errors, as they will all be lumped in with the exceptions that the except block can handle.

Defining Personal Exceptions

Earlier, we discussed an exception that's inside of the Python library. Your program is not going to be able to have the user divide by zero because this is just not allowed and we learned how to change up the message so that it worked nicely with your code.

Depending on what you are writing code for, there are times when you may want to create some of your own exceptions.

For example, you may be writing a code where you want the user *not* to be able to input certain numbers. You can create an exception for that. If you only want to allow the user to have a few guesses, it is possible to make an exception for that as well.

There aren't many limits on creating exceptions. If there is something that you don't want the user to be able to do while they are using the program, you would want to raise the exception.

To define personal or custom exceptions, you need to create a new class and this must be derived from the Exception class, indirectly or directly. To be fair, most o the exceptions already built into Python are derived from this class as well:

```
>>> class CustomError(Exception):
...     pass
...

>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError

>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

So, what we did here was create a custom exception named CustomError. This has been derived from Exception class. We can raise this exception in the same way as we do any other – with the raise statement and, optionally, an error message.

Custom exceptions can do everything that a standard class can do but it is best practice to make them

simple. Most will declare customized superclass and derive subclass exception classes from this superclass. This will look clearer after this example:

Here we are going to show you how to use a custom exception to raise and to catch errors in a program. The program is going to ask a user to input a number until the stored number is guessed. To give them some help in working it out, a hint is given as to whether their guess is lower than or greater than the number stored.

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass
class ValueTooSmallError(Error):
   """Raised when input value is too low"""
   pass

class ValueTooLargeError(Error):
   """Raised when input value is too high"""
   pass

# our main program
# user will guess a number until it is right

# you must guess this number
number = 10

while True:
   try:
      i_num = int(input("Enter number: "))
      if i_num < number:
         raise ValueTooLowError
      elif i_num > number:
         raise ValueTooHighError
      break
   except ValueTooLowError:
```

```
    print("This value is too low, try again!")
    print()
  except ValueTooHighError:
    print("This value is too high, try again!")
    print()

print("Well Done! You got it right.")
```

This is what the program would look like when it is run:

Enter a number: 12
This value is too high, try again!

Enter a number: 0

This value is too low, try again!

Enter a number: 8

This value is too low, try again!

Enter a number: 10

Well Done! You got it right.

What we have done here is defined a superclass called Error. We then derived two more classes, subclasses, from the superclass. We called them ValueTooLowError and ValueTooHighError.

Exceptions are a harder concept to learn, but they are useful in helping you to get things done. Instead of allowing the code to bring up a standard message that may not make much sense to a non-coder, you can change up the message to explain exactly what is wrong with the program. Or you can create some of your own exceptions to help move your code along as well. Take the time to try out a few of the examples above in your compiler to get a little practice and see how they work.

Chapter 8

Creating an Inheritance

Every OOP language supports inheritance, even Python but what is inheritance? The short answer is that it is the transfer of one or more characteristics from one class to another class that is derived from the original class. Make sense? Let's make that a little simpler.

Every class in Python can inherit some or all of its characteristics, including behavior methods and attributes, from another class. This class is called the superclass and the class that inherits from the superclass is called a subclass. You will come across other names for them – the superclass is often called the base or parent class while the subclass is often called the child or heir class. Classes exist in a hierarchical relationship, similar to what we come against in the real world.

Think of animals; dogs, cats, horse, sheep, cow, etc. We could create a class called Animal and then give it some methods – "woof", "meow", "neigh", "baa" and "moo" for example. We could then create subclasses called "dog", "cat", "horse", "sheep" and "cow". These subclasses would inherit the methods from the superclass.

To take that a step further, if you create an object using a subclass, that object will have all the methods that are in the superclass and the subclass, as will any variables for the superclass and subclass. The basic syntax for inheritance is:

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

The BaseClass is the superclass or parent class. The DerivedClass is the subclass or child class. DerivedClass will inherit the methods and variables, etc, from BaseClass and add new methods, etc. The result of this is code that can easily be reused anywhere.

Let's take a look at a working example of inheritance in Python:

First, we define the parent or superclass and call it "user":

class User:

```
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
        print "Name  = " + self.name

paul = User("paul")
paul.printName()
```

What we have here is a single instance, called paul, and this will output its own name.

Next, we create a class named Programmer:

```
class Programmer(User):

    def __init__(self, name):
        self.name = name
    def doPython(self):
        print "Programming Python"
```

Ok, so this looks very much the same as a normal class but we have given User in the parameters by using def_init_(self, name). What this means is, we can access all functionality that is in the class called User in the class called Programmer – Programmer has inherited everything from User.

The following is a complete inheritance example:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
        print "Name  = " + self.name

class Programmer(User):
```

```
    def __init__(self, name):
        self.name = name

    def doPython(self):
        print "Programming Python"

paul = User("paul")
paul.printName()

sally = Programmer("Sally")
sally.printName()
sally.doPython()
The output:
Name  = Paul
Name  = Sally
Programming Python
```

Because Paul is an instance of the class called User, it only has access to the method called printName. Sally, on the other hand, is an instance of the class called Programmer. This class inherits from User and, as such, can have access to the methods in User and Programmer.

Overriding Methods

Overriding refers to the ability of a subclass to change how a method inherited from a superclass is implemented. This is an incredibly important part of Python because it is this that shows the true power of inheritance. Overriding a method allows a class to "copy" another class, cutting out the risk of duplicate code while enhancing and/or customizing a part of that code to suit the purpose. Let me show you an example:

Method overriding is the definition of a method in the subclass that has the same name as a method in the superclass and, by defining the method in an object, the object can satisfy the method call without needing to call the method in the superclass:

```
class Parent(object):
    def __init__(self):
        self.value = 4

    def get_value(self):
        return self.value

class Child(Parent):
    def get_value(self):
        return self.value + 3
```

The subclass objects will now behave in a different way:

```
>>> c = Child()
>>> c.get_value()
7
```

And now we can see the difference in the class:

```
>>> Parent.__dict__
dict_proxy({'__module__': '__main__',
            'get_value': <function get_value at 0xb69a656c>,
            '__dict__': <attribute '__dict__' of 'Parent' objects>,
            '__weakref__': <attribute '__weakref__' of 'Parent' objects>,
            '__doc__': None,
            '__init__': <function __init__ at 0xb69a6534>})
```

>>>

>>> Child.__dict__

dict_proxy({'__module__': '__main__',

'get_value': <function get_value at 0xb69a65a4>,

'__doc__': None})

The subclass, or child class, now has a method called get_value(), the same as the parent or superclass but with a different implementation – look at the id of both functions and you will see they are different.

To sum up, method overriding is a way of letting a subclass, provide an alternative implementation of a specific method that has already been defined in the superclass. This subclass implementation will override the superclass implementation because the new method has the same name, signature or parameters and return type of the superclass method.

Method Overloading

Python allows you to do something quite clever – define a method in a way that there is more than one way to call on it. This is called overloading and, given a function or a method, we can specify how many parameters there are. For example, depending on the definition of a function, we may call it with zero parameters, one, two, or even more.

Let's look at an example of method overloading:

First, a class is created. It has one method in it, called sayHi(). We set the first parameter of sayHi() to None, giving us the option of calling it in two ways – with a parameter or without a parameter. Next, we create an object, based on sayHi() and then call the method on it using zero parameters and one parameter:

```
#!/usr/bin/env python

class Human:

    def sayHi(self, name=None):

        if name is not None:
            print 'Hi ' + name
        else:
            print 'Hi '

# Create instance
obj = Human()

# Call the method
obj.sayHi()

# Call method with a parameter
obj.sayHi('Donald')
```

Output:

```
Hi
Hi Donald
```

To make method overloading a little clearer, we now have two ways to call sayHi() method:

```
obj.sayHi()
obj.sayHi('Donald')
```

We have a method that can now be called using fewer arguments than the definition of the method allows. You are not limited to just two either; use as many variables as you want.

Multiple Inheritances

Another great feature with Python is creating multiple inheritances at a time. You can make as many of them as you want, taking the features that you want from the parent class and bringing them down to the child class in as many steps as you would need for your code.

Multiple inheritance is a feature where a class is going to have two or more parent classes. If you do this properly, you will be able to use as many parent classes as you would like within the new child class. What basically happens is that you are creating a new class, ClassC, and it is created from the features of ClassB. ClassB was designed from ClassA, meaning that ClassC is going to have some features from both ClassB and ClassA to make it work. You'll be able to make adjustments at each level, but you will get some of the features of the parent class into the child one.

Inheritance is one of the most important features of any Object-Oriented Programming language, simply because it is always better to reuse what functionality you already have, instead of attempting to create the same functionality repeatedly. That makes for nothing more than messy and unreadable code. By reusing what we have, we can save time and ensure that our program is reliable. One of the main advantages of inheritance is that module, methods, classes, etc, that all have similar interfaces can share code between them, cutting down on the complexity of your program.

As a final overview, the benefits of inheritance are:

- Subclasses or child classes, provide special behaviors based on elements that are provided by the parent or superclass.
- By using inheritance, the code can be reused multiple times, saving time, cleaning up the code and making it more readable for everyone, as well as being less prone to errors because of too much unnecessary code.

Chapter 9

How to Create a Loops

Earlier we spent time covering conditional statements and why you would want to use them. Conditional statements are great for adding some interaction with your program and the user and there are many times you'll use them. But, there may times when you may need to get away from the conditional statements and *automate* the process for your program.

For example, you could decide to write a program that will list all the numbers from 1 to 100. With the methods that we have talked about thus far, you would have to write out a separate block of text for each number from 1 to 100. This doesn't sound like much fun for anyone, even a beginner, but luckily there is another way to accomplish this with just a few lines of code. Creating loops inside with Python will get all of this into a single block of code.

Loops tell the compiler that it should keep on reading the exact block of code repeatedly until your conditions are met. If you want to write out the numbers from 1 to 100, you need to make sure that the compiler knows when to stop

Breaks are important, otherwise, your loop will continue endlessly. For the example above, we would tell the loop to stop when a number was greater than 100. If we forget to put in a break, the loop will be infinite and will not be able to stop. The breakpoint can be anything that you would like, but make sure that it is there, in all loops that you use in Python.

You have the standard loop choices with Python. The type you use will depend on what you want your program to do. The two loop types that we will look at in this chapter include the while loop and the for loop.

The While Loop

While loops are used to execute a statement repeatedly, so long as a specified condition remains true. The syntax of the while loop is:

while expression:
  statement(s)

statement(s) may be a block of statements or it may be a single statement. The condition can be any expression you choose and the value of True will be any value that is non-zero. The loop will continue to iterate or repeat, for as long as the condition remains true.

When it is no longer true, i.e. a zero value is found, the loop will stop and the control of the program will then go to the next line of code after the loop.

There is one key point of the while loop – it may never run. If the condition is tested and returns false, the loop will be skipped over and the statement that comes directly after the loop is executed. Let's look at an example:

```
#!/usr/bin/python

count = 0

while (count < 9):

  print 'The count is:', count

  count = count + 1

print "Goodbye!"
```

When we run this code, we get:

```
The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6
```

The count is: 7

The count is: 8

Goodbye!

The block in the code, consisting of the increment and print statements, will be repeatedly executed until count is no longer less than 9 and, with each repeat, the index count value in increased by 1

Code courtesy of TutorialsPoint.

The For Loop

With the for loop, the user won't be responsible for inputting and as a result determining when the loop should stop. Rather, in the for loop, Python will see that you've set up an iteration in a certain order and this information will be shown on the screen when the user gets to this point. The input is not necessary and the code will keep displaying the information until it reaches the end.

An example of how the for loop would work:

```
# Measure strings

# The function "len()" measures the length of a string

words = ['apple', 'mango', 'banana', 'orange']

for w in words:

print(w, len(w))
```

When you run this code, you'll see there are four fruits that display and they will display in the same order that they were written. If you would like to see them display in a different order, you need to adjust the code in the order you want the fruits to be displayed.

In some cases, the while loop and the for loop are going to work the same. If you want to create a list of numbers 1 to 100, it'll work to get this done and the user won't be able to tell the difference.

Before you choose a loop for your program, decide how many times you would like the loop to go through. If you want the loop to happen at least one time in the program, the while loop is probably best suited. But, if the amount of times doesn't matter, then you can go with the for loop.

Nested Loops

In Python, we can use one loop inside of another loop and these are called nested loops. The syntax used for these is:

```
for iterating_var in sequence:
  for iterating_var in sequence:
    statements(s)
  statements(s)
```

If we wanted to use a nested while loop the syntax would be:

```
while expression:
  while expression:
    statement(s)
  statement(s)
```

With loop nesting you can use any type of loop inside any type of loop. For example, you could place a while loop inside a for loop and vice versa. Let's look at an example.

In this program, we are going to use a nested for loop to find all the prime numbers between 2 and 100:

```
#!/usr/bin/python

i = 2
while(i < 100):
  j = 2
  while(j <= (i/j)):
    if not(i%j): break
    j = j + 1
  if (j > i/j) : print i, " is prime"
  i = i + 1

print "Goodbye!"
```

When we run this code, this is what we get:

```
2 is prime
```

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

Goodbye!

Code Courtesy of TutorialsPoint.

Loops are a great way to make sure that your code stays clean and tidy while you still get the iterations that you want. No one wants to write out a code that has 100 blocks of code just to create a big table. When you can do this in just a few lines of code, it's much easier and makes the code look a nicer as well. As a beginner, you'll appreciate the amount of time and effort this is going to save you.

Conclusion

I hope this book was able to help you to understand Python and how to use it.

Python is one of the easiest of all the computer programming languages to learn and this is because it will very quickly teach you to start thinking like a computer programmer. It is incredibly readable and you will not need to waste time trying to remember strange and mysterious code like you get with many of the other computer languages. Instead, you can get right into learning the paradigms and concepts of Python programming and, once you have learned these, you can go on to other languages, more powerful ones and be able to understand the code.

That said, I don't want you to think that Python is like a child's toy because it isn't. It is an incredibly powerful language – if it wasn't, the likes of Nasa wouldn't be using it! Remember too, many of the biggest websites on the net today are also built on Python. However, it is easy to learn once you get your brain in gear and your programming mind in place. There are no steep learning curves in Python; all you need is a bit of ambition and plenty of attention span and you will soon be coding like a pro.

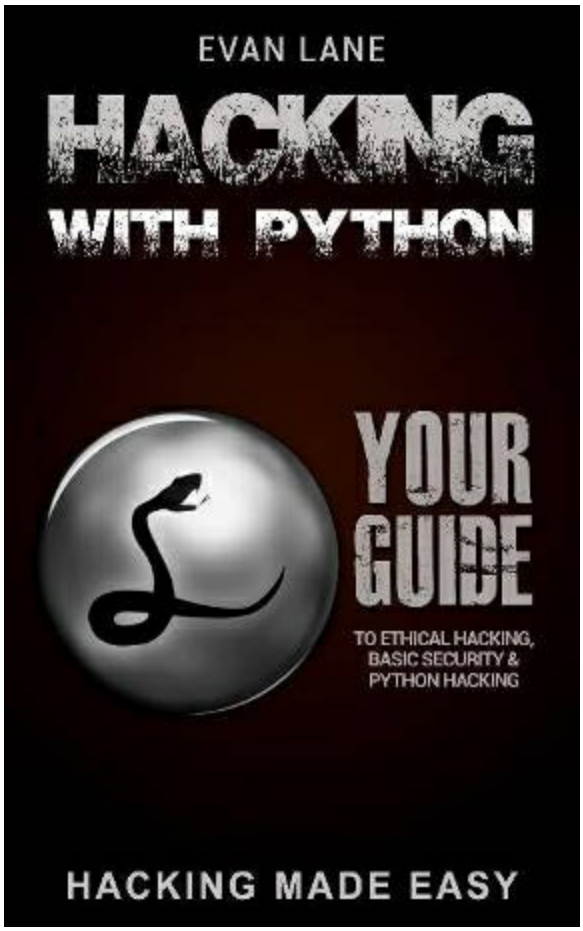
One of the truly great things about Python is the amount of documentation available, especially on the official Python website. If you really can't find what you want, head over to one of the many Python forums on the internet, filled with a community of friendly folk, ready to help and put you on the right path to success.

The next step is to get programming, it's as simple as that. You can read all the books you want but unless you open your Python interpreter and start coding, you won't learn much. There is nothing like practical work to hammer a concept home in your brain and the internet is full of Python tutorials, Python courses and interactive learning platforms so get out there and find them; fire up the interpreter and start inputting your code today.

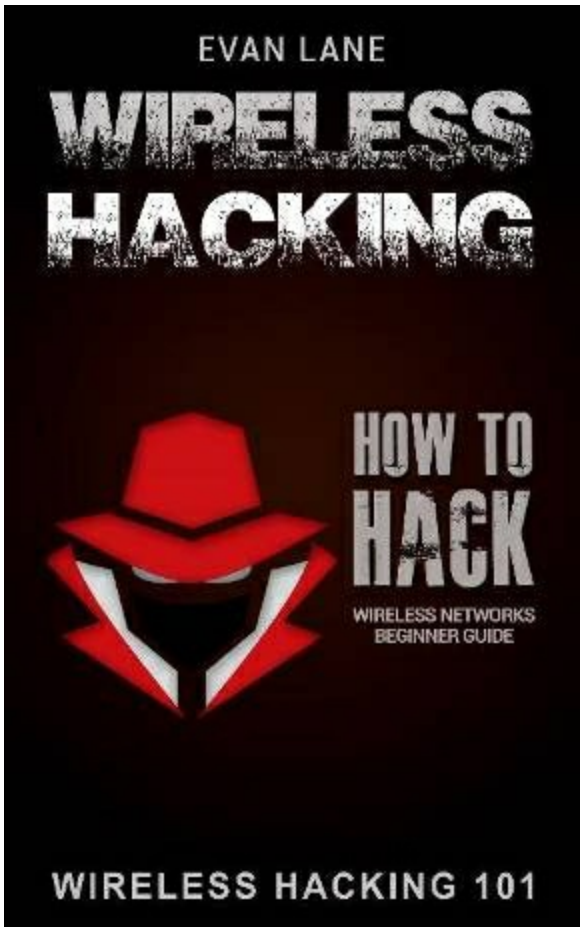
Other Books by Author

Click here to check out Evan Lane's Amazon Author Page for more great books!

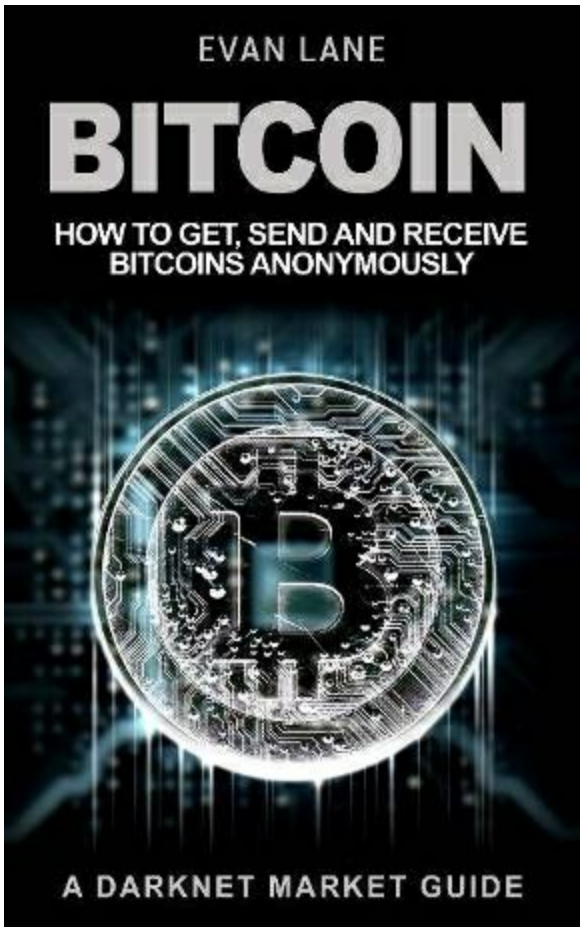
Hacking with Python: Beginner's Guide to Ethical Hacking, Basic Security, Penetration Testing, and Python Hacking



Wireless Hacking: How to Hack Wireless Networks



Bitcoin: How to Get, Send and Receive Bitcoins Anonymously



TOR: Access the Darknet, Stay Anonymous Online and Escape NSA Spying