

# COMPUTER PROGRAMMING

JavaScript, Python, HTML, SQL, CSS

The step by step guide for beginners to intermediate  
Including some black hat hacking Tips



Bundle 5 Books In 1 The #1 Coding Book for 2019

By William Alvin Newton    and    Steven Webber

Computer Programming

JavaScript, Python, HTML, SQL, CSS

The step by step guide for beginners to intermediate

Including some black hat hacking Tips

Bundle 5 Books In 1 The #1 Coding Book for 2019

By William Alvin Newton

and Steven Webber

Copywrite 2019 Production date 2019

Narrated by Michael Waterson

THE ULTIMATE PYTHON PROGRAMMING GUIDE FOR BEGINNERS TO INTERMEDIATE:

A STEP BY STEP GUIDE TO

COMPUTER PROGRAMMING AND BLACK HAT TECHNIQUES THE LEARNERS

BIBLE FOR 2019

© Copyright 2019 by William Alvin Newton - All rights reserved.

This eBook is provided with the sole purpose of providing relevant information on a specific topic for which every reasonable effort has been made to ensure that it is both accurate and reasonable. Nevertheless, by purchasing this eBook you consent to the fact that the author, as well as the publisher, are in no way experts on the topics contained herein, regardless of any claims as such that may be made within. As such, any suggestions or recommendations that are made within are done so purely for entertainment value. It is recommended that you always consult a professional prior to undertaking any of the advice or techniques discussed within.

This is a legally binding declaration that is considered both valid and fair by both the Committee of Publishers Association and the American Bar Association and should be considered as legally binding within the United States.

The reproduction, transmission, and duplication of any of the content found herein, including any specific or extended information will be done as an illegal act regardless of the end form the information ultimately takes. This includes copied versions of the work both physical, digital and audio unless express consent of the Publisher is provided beforehand. Any additional rights reserved.

Furthermore, the information that can be found within the pages described forthwith shall be considered both accurate and truthful when it comes to the recounting of facts. As such, any use, correct or incorrect, of the provided information will render the Publisher free of responsibility as to the actions taken outside of their direct purview. Regardless, there are zero scenarios where the original author or the Publisher can be deemed liable in any fashion for any damages or hardships that may result from any of the information discussed herein.

Additionally, the information in the following pages is intended only for informational purposes and should thus be thought of as universal. As befitting its nature, it is presented without assurance regarding its prolonged validity or interim quality. Trademarks that are mentioned are done without written consent and can in no way be considered an endorsement from the trademark holder.

---

## Description

In The Ultimate Python Programming Guide for Beginners you will learn all the essential tools to become proficient in the python programming language. Learn how to install python in all major operating systems: Windows, Mac OS, and even Linux. You will be guided step by step from downloading the necessary files to making adjustments in the installation for your particular operating system. Learn the command line shell, and how to use it to run python in interactive and script modes.

Discover how the python interpreter functions, and learn how to use the interactive command line shell through practical examples you can try on your own. Learn datatypes and variables in depth, with example code and discussion of the generated output.

Numbers are covered in detail, including a discussion of the 4 number types in python: integer, float, complex, and boolean. Learn about Truthy and Falsy returns and how they relate to the boolean type. Practice with some of the many built-in python math functions, and discover the

difference between `format()` and `round()` functions.

Strings are one of the most important variables in any programming language. Learn in-depth how to explore, search, and even manipulate strings in python. Practice with python's built-in string methods.

Learn about python's control structures and how to use boolean logic to achieve your software requirements.

Deal with operators and develop an understanding of the strengths and differences of mathematical, relational and logical operators, as well as the importance of operator precedence and associativity.

Learn about strings and the many ways to search through and manipulate them.

Discover the power of inheritance and polymorphism.

Learn how to open, manipulate and read, and close files on your file system.

Learn about the philosophy and importance of code reuse, and how modules in python makes this simple.

Examine the difference between procedural and Object Oriented programming. Which is right for you may depend on what kind of code you are writing.

Practice control structures in python.

Study operators and learn about operator overloading.

An in-depth discussion of python sequences: lists, sets, tuples and dictionaries. Learn the strengths and weaknesses of each. Practice creating and manipulating python sequences.

Table of Contents

Introduction

Chapter 1: Installation

Chapter 2: Interpreters

Chapter 3: Data Types and Variables

Chapter 4: Numbers

Chapter 5: Operators

Chapter 6: Strings

Chapter 7: String Methods

Chapter 8: If Else Statements

Chapter 9: Loops

Chapter 10: Break and Continue

Chapter 11: Lists

Chapter 12: Functions

Chapter 13: Modules

Chapter 14: Objects and Classes

Chapter 15: Inheritance and Polymorphism

Chapter 16: Operator Overloading

Chapter 17: File Handling

Chapter 18: Exception Handling

Chapter 19: Tuples

Chapter 20: Sets

Chapter 21: Dictionary

Chapter 22: black hat hacking tips

Conclusion

Prologue:

For those listening to the audio book version it is recommended that you also purchase the paperback or kindle eBook version in order to view figures, illustrations and to see exactly how to type different codes and instructions in a precise manner. Also while narration this book I will, in order to simplify a frequently repeated phrase. I will say the computer symbols >>> ( as in less than greater than) only one time (>) unless otherwise indicated by being in the code itself. The first few chapters these (>) symbols will be repeated 3 time as they are written, to give you an idea of their place. After chapter 7 these will be repeated one as the chapters get more advanced.

Introduction

Congratulations on downloading THE ULTIMATE PYTHON PROGRAMMING GUIDE FOR BEGINNERS: A STEP BY STEP GUIDE and thank you for doing so.

You do not need to know how to code to use this book, but it does assume the reader is familiar with his or her operating system of choice, knows how to invoke a terminal or command window, and how to install and configure software. Beyond that, you should find everything you need to learn in detail how to become a python developer.

The following chapters will provide a step-by-step method to install, configure, and begin coding in the python programming language. Python is an excellent language for those new to programming, and an easy transition from other languages. It has a natural language feel and avoids many of the rigid blocking and syntax of other programming languages. Python was first developed in 1991 by Guido Van Rossum. It is called a high level language because it hides many of the details going on under the hood from the programmer. Those activities are dealt with by the python interpreter, which takes each line of text provided by the programmer and creates the machine language code required by your computer or other device to run the program.

Python is a strongly typed language, which means it does not automatically convert data types. Instead, the programmer must convert as necessary, meaning it is more difficult to make data type errors in python.

Python has a massive set of libraries, which are essentially tools to achieve goals written by other developers that you can download and attach to your program. You can view these libraries here: <https://pypi.python.org/pypi> . There are also many other user contributions to python, which you can find on the web. If you decide to become a python developer, one day you might contribute work you have done to the python community.

While this book does cover the core aspects of the python programming language, no book can teach you everything. Python, like all active programming languages, itself is like a living system. Each new release brings new functionality, and occasionally deprecates some older, less useful or less secure functionality. To really become a software developer, you must commit yourself to continually learning new tricks and new ways of writing elegant code, and to keep up to date on the latest release of any software you are using. That said, this book is the perfect first step on what could become a life-long journey and fulfilling career as a software developer.

There are lots of books on beginning python on the market, so thanks again for choosing this one! Every effort was made to ensure it is full of as much useful information as possible, please enjoy!

Chapter 1: Installation

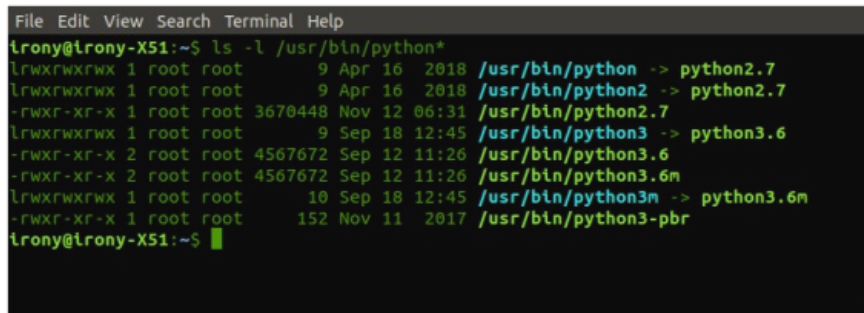
Before you can begin to code in python, you will need to install it to your computer. Python is not a large install, and it does not require a lot of system resources. Even the most humble computer should be able to run the python interpreter and small programs without issue. Keep in mind, however, that python is a real programming language, and some larger programs can use large amounts of resources.

Some operating systems come with python installed, some do not. Below you will find step-by-step instructions on how to install and configure python for your operating system.

This book uses a specific version of python, 3.6.6 – you will need to install this version or higher (ie: 3.6.7 or above) to ensure you will be running a version compatible with the code samples and examples in this book. Many operating systems, if they have python pre-installed, will have a version of python 2. While the developers who produce python make every effort to keep newer versions of python compatible with older versions, python 3 introduces changes that are not compatible with version 2.

Installing on Linux:

Most versions of Linux come with python reinstalled. You can check by running the following command: `ls -l /usr/bin/python*`



```
File Edit View Search Terminal Help
irony@irony-X51:~$ ls -l /usr/bin/python*
lrwxrwxrwx 1 root root 9 Apr 16 2018 /usr/bin/python -> python2.7
lrwxrwxrwx 1 root root 9 Apr 16 2018 /usr/bin/python2 -> python2.7
-rwxr-xr-x 1 root root 3670448 Nov 12 06:31 /usr/bin/python2.7
lrwxrwxrwx 1 root root 9 Sep 18 12:45 /usr/bin/python3 -> python3.6
-rwxr-xr-x 2 root root 4567672 Sep 12 11:26 /usr/bin/python3.6
-rwxr-xr-x 2 root root 4567672 Sep 12 11:26 /usr/bin/python3.6m
lrwxrwxrwx 1 root root 10 Sep 18 12:45 /usr/bin/python3m -> python3.6m
-rwxr-xr-x 1 root root 152 Nov 11 2017 /usr/bin/python3-pbr
irony@irony-X51:~$
```

Here we see python 2.7 and python 3.6 are installed, and that python 2.7 is default. If you run python in the command line you will start up the default 2.7 python command line. You can type `exit()` to quit the program. To run python 3.6, simply type `python3` into the terminal.

To see the actual version of python installed on your Linux system, type `python3 -V` into the terminal.

Version 3.7.1 is the latest stable release as of the writing of this book, but anything 3.6.6 or better will be fine.

If python is not pre-installed on your version of Linux, try these steps:

```
$ sudo apt-get update
```

```
$ sudo apt-get install python3.6
```

If you're using the latest LTS release of Ubuntu you can use the deadsnakes PPA to install Python 3.6:

```
$ sudo apt-get install software-properties-common
```

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

```
$ sudo apt-get update
```

```
$ sudo apt-get install python3.6
```

If you are not using Ubuntu, use your distribution's package manager. On Fedora, for example, you would use `dnf`:

```
$ sudo dnf install python3
```

If you are having difficulty figuring out how to install python3 on your particular version of Linux, find a community on the web for your distribution and get help there. Linux communities are open and friendly, and you should have help in no time.

Installing on Windows:

Python offers a Windows binary installer for python. This means just it operates just like installing any other Windows program.

To get the installer switch to a web browser and navigate to this URL:

<https://www.python.org/downloads/>

You should see a button recommended the latest stable release for you to download. Click it to download the installer.

---

## Download the latest version for Windows

Download Python 3.7.1

Looking for Python with a different OS? Python for [Windows](#),  
[Linux/UNIX](#), [Mac OS X](#), [Other](#)

Want to help test development versions of Python? [Pre-releases](#)

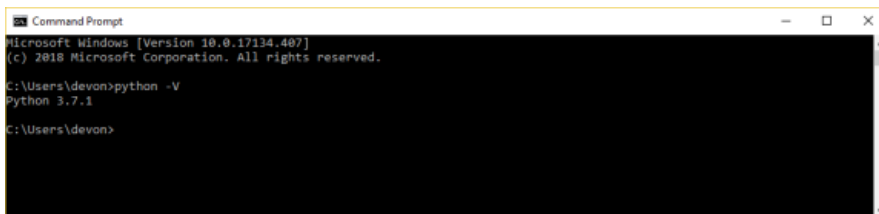
Looking for Python 2.7? See below for specific releases

Now find and run the binary (they usually land in your Downloads folder). The only customization you will need to do during the installation is to ensure you add python to the Windows path (see the red circle in the image below) – this will allow you to run python from anywhere in your filesystem.

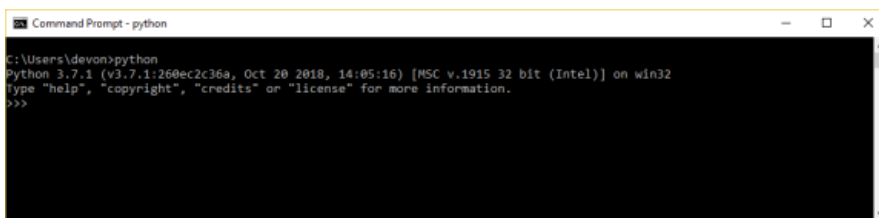


Now open a command terminal and type `python -V`. If you are not sure how to open a command shell in Windows, use the search feature and type “cmd” – the command terminal should appear in the search results.

When you’ve launched your command terminal and typed the text above, you should see Python 3.7.1 (or whatever is the latest stable release you downloaded).



And if you type python you will get the interactive python shell (we’ll discuss this later). To exit the shell, type `exit()` and then the [return] key.



Installing on Mac OS:

Python offers a Mac OS binary installer for python. To get the installer start a web browser and navigate to this URL:

<https://www.python.org/downloads/>

You should see a button recommended the latest stable release for you to download. This will install just like installing any software on your Mac. Click the button to download the installer.



When the download is complete, find the binary on your computer's file system and double-click to install (they usually end up in your Downloads folder). Follow the onscreen instructions, accept the license, and enter your system password if it is required.

On Mac OS no other configuration is required. Python 3 should now be installed and available on your system.

Test your installation by opening a Terminal window and typing: `python3 -V`. You should see Python 3.7.1 (or whatever is the latest stable release you downloaded).

```
File Edit View Search Terminal Help
r1Mac:~ irony$ python3 -V
Python 3.7.1
r1Mac:~ irony$
```

Now run the interactive python shell by typing `python3`. When you are done, type `exit()` to leave the shell.

```
File Edit View Search Terminal Help
r1Mac:~ irony$ python3
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 03:13:28)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What to take away from this chapter:

python 3 can be installed on Windows, Mac OS, or Linux.

Python usually comes pre-installed on Linux.

Both Windows and Mac OS have install binaries available for download

Only Windows requires a path setting on installation.

Test your python version by running `python3 -V` ( `python -V` on Windows)

run the interactive shell by opening a command terminal and typing `python3` ( `python -V` on Windows) – exit the shell by typing `exit()` and the [return] key.

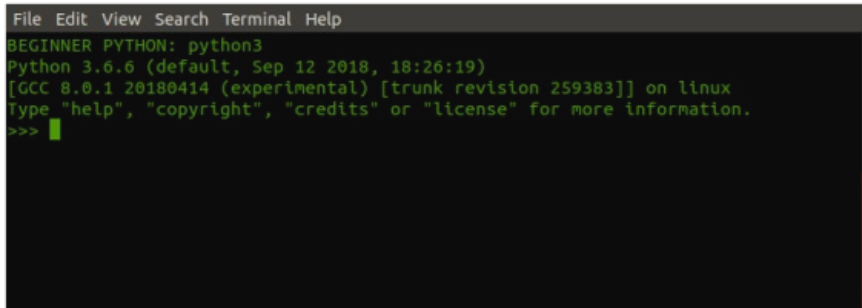
## Chapter 2: Interpreters

This book will be using Python 3.6.6 for all examples. As long as you are running 3.6.6 or higher you will be fine. Refer to chapter 1 on how to install python to your operating system.

The installation of python in chapter 1 installs a python interpreter on your computer. This is a piece of code that translates whatever you type into machine code instructions that can be handled by your computer. This interpreter operates in two different modes: interactive and script. In the first case, interactive, the interpreter receives your input when you press the [return] key, and will respond to what you have written immediately. It then waits for you next input (hence interactive). The script interpreter receives a series of commands you have put in a standard text file, and runs them all at once.

### Python Interactive Interpreter:

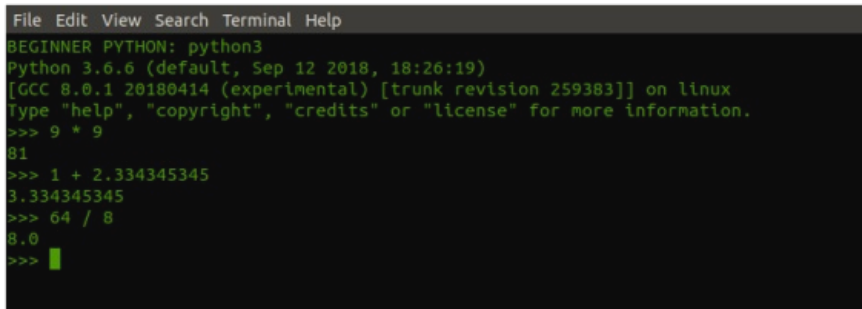
If you ran the commands in examples provided in chapter 1, you would have invoked the python interactive interpreter. Open a Terminal or Command window and type python3 ( python on Windows). Notice the prompt string: >>>

A screenshot of a terminal window with a dark background. The title bar at the top shows 'File Edit View Search Terminal Help'. The terminal text is as follows:

```
BEGINNER PYTHON: python3
Python 3.6.6 (default, Sep 12 2018, 18:26:19)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

The prompt is there to let you know the interactive shell is ready for your input. When you type something and click the [return] key, what you have typed will be interpreted and the results will be printed immediately (if there is any result to be printed). The interpretive shell is useful for testing short snippets of code.

For fun and to get us started, you can do simple calculations using the shell, just like a calculator. Try typing some common math into your interactive terminal:

A screenshot of a terminal window with a dark background. The title bar at the top shows 'File Edit View Search Terminal Help'. The terminal text is as follows:

```
BEGINNER PYTHON: python3
Python 3.6.6 (default, Sep 12 2018, 18:26:19)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 9 * 9
81
>>> 1 + 2.334345345
3.334345345
>>> 64 / 8
8.0
>>> █
```

Or you can do more traditional programming, like having python print out some text by typing print("hello world") and then the [return] key:

```
print("hello world")
```

hello world

Those lines we've entered above into the interpreter are called statements. Things like print("hello world") or 9 \* 9 are all statements, which means they are simple instructions sent to the interpreter for execution. We'll discuss more about statements later on in this book.

You will have noticed above there are two different forms of displaying output. For the rest of the book, for simple command line input and output, the drop-shadow box will be used for examples, allowing you to copy and paste them into your own shell for testing. Images will be used to demonstrate interpreter output from longer scripts, or to demonstrate system specific output like error messages and the like.

### Python Script Interpreter:

So the examples above were fun, but they are not stored anywhere. They are like a single use straw that you throw away after you've finished your drink. If you want another drink tomorrow, you have to get another straw. Programmers frown on this wasted code the way we should all frown on single use plastics. The name of the game in software development is "code re-use". It is better to re-use code than to write it again. This saves



both time and money, because programmers can re-use not only their own code, but the code from other programmers (with permission of course), code re-use can vastly decrease development time. This re-use allows for large pieces of software to be written fairly quickly, as often the core code is already there and all that needs to be built is the interface between the code and the user, the User Interface, or UI.

This is why all programmers use script interpreters. You put all your code in a simple text file, then feed the entire thing to the interpreter all at once. This way, you can run your code over and over just by sending it to the interpreter again and again.

In order to write code in this manner, all you need is a text editor. On Linux you have many options, from the terminal to a plethora of simple text editors. On Windows you can use Notepad. And on Mac OS you can use the Text Editor App (so long as you make sure you click on “Format” in the menu bar and select “Make Plain Text” to make sure TextEdit saves your file in plain text).

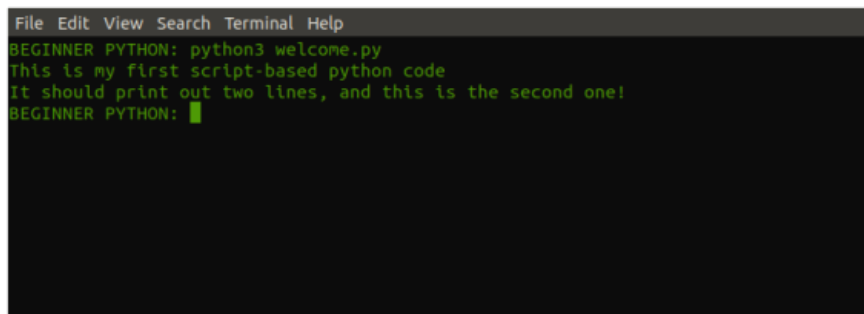
So create a new folder on your computer, somewhere you will remember, and call it “mypython”. We will save all our example text files in this folder. Create a folder inside called “chapter02” – here we will store the files for this chapter.

Now open your text editor and put in the following:

```
print("This is my first script-based python code")

print("It should print out two lines, and this is the second one!")
```

When you have saved your file as “welcome.py”, open your terminal, navigate to the same directory (mypython/chapter02) and run the following command: `python3 welcome.py` (remember it is `python` welcome.py on Windows).

A screenshot of a terminal window with a dark background and green text. The terminal shows the command `BEGINNER PYTHON: python3 welcome.py` being entered. The output of the script is displayed on the next two lines: `This is my first script-based python code` and `It should print out two lines, and this is the second one!`. The prompt `BEGINNER PYTHON:` is visible on the third line, followed by a cursor. The terminal window has a menu bar at the top with options: File, Edit, View, Search, Terminal, Help.

Congratulations, you have now written your very first python script!

A Quick Chat On IDE Options:

Most software developers work inside an IDE, which stands for Integrated Development Environment. This is a tool or collection of tools that organizes the code writing workflow. In short, IDE tools speed up the development process by gathering together in one program useful tools to facilitate and speed up software code writing, organization, and testing. However, given this is a book on learning python, we will not use an IDE. Although they are very useful for working in large projects, they have their own complexity and learning curve, and you’re here to learn python.

You should take some time to look into an IDE if you intend to become a software developer. At the very least, you might want to acquire a text editor dedicated to writing software. A very popular free editor that is under heavy development and with lots of features (including a python extension) is Visual Studio Code. It is available for Windows, Mac OS, and Linux, so if you have to work in more than one operating system you can to use the same code editor. This is the editor the author uses, after having tried many others.

What to take away from this chapter:

You can invoke the python shell and quickly run small bits of code.

By saving your code in a text file, you can run large pieces of code through the python interpreter.

Invoke python with `python3` on macOS and Linux, and with `python` on Windows.

Integrated Development Environments (IDEs) are useful tools for speeding up software code writing and testing.

Files for the script interpreter should be saved with a “py” extension, as in `file.py` – this is an extension of convenience, so developers understand what kind of code is inside the file simply by looking at the name.

## Chapter 3: Data Types and Variables

A data type just means a particular kind of information. What particular kind of information is determines what you can do with. For example, an

integer is a data type in many programming languages. An integer is whole numbers (no fractions or decimals), so 1, 99, -3 are all integers. Knowing you have an integer means knowing what you can do with it. In this case, add, subtract, multiply, divide, etc.

Another data type is a string. This is a series of characters, like a simple sentence: "the brown dog". If you know you have a string, you know you can concatenate it (splice together) with another string.

But you cannot add an integer and a string:

For example, if we add the literals `10 + 20`, we get the result: 30

If we add the literals "this is a" and " string", we get the result "this is a string"

But if you attempt to add different types, you'll get the results below:

```
print(10 + 2)
```

12

```
print("hello" + " world")
```

hello world

```
print(10 + "hello world")
```

Traceback (most recent call last):

File "", line 1, in

TypeError: unsupported operand type(s) for +: 'int' and 'str'

The first output makes sense, `10+2 = 12`. The second line also makes sense, concatenating "hello" and " world" give the text string "hello world". The third output line demonstrates what happens if we attempt to combine two different data types, in this case an integer data type (10) with a string data type ("hello world"): the result is a `TypeError`, with an explanation of what the interpreter thinks has gone wrong, in this case adding (+) an integer (int) and a string (str).

Python has many built in functions, including one for detecting data types, `type()`.

```
type(99)
```

```
type("hello world")
```

```
type(99.93)
```

```
type("99.93")
```

Here we see a new data type, float, which is numbers that can include decimals. If you look after the float type, you'll see what looks like a repeat of the float value, but here the `type()` function declares it a string. This is because the number is enclosed in quotation marks. This is how we declare a string, by putting it in quotes (these quotation marks can be single or double). If you do not use quotes, python will attempt to identify what you type as a different data type (an int or float, for example), or throw an error.

Variables:

A variable is a kind of box where we can keep things like strings or integers, or other data types. You create a variable in python like this:

```
variable-name = expression
```

The variable name is something you come up with. It is good programming practice to give variable names something meaningful to do with what they are being used for. For example, if you were creating a variable to store a number of oranges, a good variable name might be "oranges".

An expression can be a value, variable, operator, or a combination of them. The equals sign (=) is called an assignment operator.

So, if I want a variable called "wheels" I create it by typing:

```
wheels = 3
```

When the python interpreter reaches this declaration, it does a couple things. First, it takes the literal value of 3 and stores it in memory. Then it creates the variable 'wheels' and points the variable to this value in memory. It's vital to understand the value and the variable name assigned to it

are not the same thing. One only refers to the other. So you can do things like this:

```
wheels = 3
print(wheels)
```

3

```
wheels = "square"
print(wheels)
```

square

Here you can see the value of wheels has been changed by “pointing” it to a different value in memory, in this case from the integer 3 to the string “square”.

Python is different from many other programming languages because you do not need to declare variables before using them. To create a variable you simply type it out, use the equals sign, and then type the value you want it to have.

Variable Rules:

Variable names can only contain characters (a-z and A-Z), underscores ( \_ ), and numbers ( 0-9 ), and they can only begin with a character or underscore. They cannot begin with a number. When in doubt, open up an interactive terminal and try a variable name out. The interpreter will complain with a `NameError` if it doesn't like it.

Here are a few examples:

```
my_var = 6
>
my_var
```

6

```
>
_myvar = "two"
>
_myvar
```

'two'

```
>
my_var7 = False
>
my_var7
```

False

```
>
3vars = 3
```

File "", line 1

3vars = 3

^

SyntaxError: invalid syntax

- -

There are several words, called keywords, in python. They are reserved words and you cannot use them as variable names. If you try, the interpreter will throw an error.

Here is a list of python keywords. But keep in mind this can change over time with different version of the language:

False

None

continue

class

try

is

assert

return

for

or

lambda

from

global

while

True

finally

def

not

and

del

with

nonlocal

as

if

elif

import

yield

else

raise

break

pass

except

in

Keep in mind that the python language is case sensitive. This means welcome, WELCOME, welcomE, and Welcome are considered different variable names.

What to take away from this chapter:

Data types describe what kind of information you are dealing with.

Data types cannot be added/concatenated to different data types.

Two data type are integer and string.

type() is the built-in function in python for detecting data types.

Variables are placeholders that point to values in memory

Variable names can only begin with a character or underscore “\_”.

## Chapter 4: Numbers

Python has 4 number types:

integer

float

complex

boolean

We've already discussed integer and float data types: integers are all whole numbers (no fractions), while floats are all numbers including fractions.

Complex numbers deserves a book all its own. Complex numbers are used in math to make calculations involving oscillations. This applies to Quantum Mechanics, Electrical Engineering, and many other fields. If you are curious, you can look into complex numbers and how they function in python as a research project. We will not be covering complex numbers in this book.

Boolean values are either True or False. Determining the condition of an expression as either True or False will be something you'll be doing a lot in programming. Boolean logic is essentially what software programs consist of, along with looping and control structures. Here is a simple example of the boolean datatype:

```
a = 11
b = 79
print(a > b)
```

False

```
print(a < b)
```

True

On the first two lines we assign values to the variables a and b (a is 1, while b is 2). On the third line we print the result of claiming a is greater than b. The interpreter responds False, because a is in fact smaller than b. When we print the result of this claim, the interpreter responds True, as expected.

The Boolean data type leads to what are called Truthy and Falsy values. That is, there is a series of values that are equivalent to True, and others equivalent to False.

Falsy values are:

None

False

Zero (the integer value of 0)

An empty Sequence (discussed later on)

An empty Dictionary (discussed later on)

Every other value is considered Truthy. Here are some examples:

```
|| | bool("hello")
```

True

```
|| | bool("0")
```

True

```
|| | bool(8)
```

True

```
|| | bool(0)
```

False

```
|| | bool({})
```

False

```
|| | bool([])
```

False

```
|| | bool(())
```

False

Once again, pay attention to `bool("0")` and `bool(0)` – can you tell why the first is true and the second is false? That's right, because the first 0 is enclosed in quotation marks ("0") it is a string, and strings are Truthy. The other example is the number 0 (zero), and 0 is Falsy.

There are many built-in mathematical functions in python. Some of the more common ones are:

`abs()`

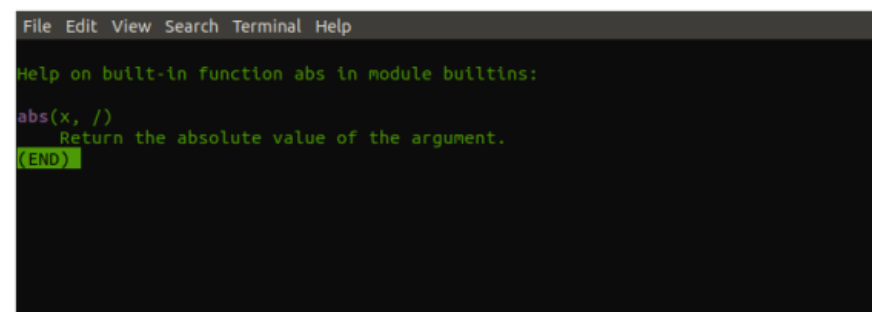
`pow()`

`round()`

`min()`

`max()`

To understand what a math function does, you can simply ask the interpreter. Type `help(abs)`



```
File Edit View Search Terminal Help
Help on built-in function abs in module builtins:
abs(x, /)
    Return the absolute value of the argument.
(END)
```

The `help()` function can be used to explore any built-in python function.

Python's math module allows you access to any more common mathematical functions and constants. You include a module in python by using the import statement:

```
File Edit View Search Terminal Help
>>> import math
>>> math.pi
3.141592653589793
>>> math.floor(3.9)
3
>>> █
```

You can view a complete list of python's math functions at <https://docs.python.org/3.4/library/math.html> .

#### Number Formatting:

Let's say you want to know how much you are spending on groceries each day. You know you spent \$339.14 last month, and that there were 30 days in the month. In python you could find this value by typing `339.14/30`, but you would receive an answer like this: `11.304666666666666` – this answer, while it is very accurate, isn't very user-friendly. After all, we can't split pennies. You might then chose to employ the built-in `round()` function to round the result to 2 decimal places: `round( 339.14/30,2)` . Your answer would now be: `11.3` – closer, but it's still not in a dollar and cents format money format.

To achieve this goal we can use another built-in function called `format()`, like this:

```
groceries = 339.14/30
print("Last month cost for groceries per day: $", format(groceries, "0.2f"))
```

Last month cost for groceries per day: \$ 11.30

`Format()` works by allowing you to specify how many digits to display in total (decimal point included), and how many digits to display after that decimal. The 'f' option tells the interpreter to display the value as a float. Thus it is *totalwidth.precisionof\_decimals*f.

If you specify a *total\_width* smaller than the output, the interpreter will expand the width automatically to the minimum required to display all the necessary values. Try inputting `print(format(11.7924, "2.3f"))` into your interpreter. You will see `11.792` as the output. Even though you requested a total width of 2 characters, the interpreter automatically expanded that to 6 (that's 1 – 1 - . - 7 – 9 – 6) characters. At the same time, you requested a precision of decimals to 3 digits, so the interpreter truncated the `.7942` to `.794`. Try changing the `.7942` to `.7948` and it will return `.795` – `format()` rounds to the nearest value, 0-4 down and 5-9 up.

If you specify a width larger than the total number of characters returned, the interpreter will add blank spaces to the left of the value until it reaches the specified width.

Numbers are returned right-aligned, but you can specify left and right align alignment with the < and > characters.

Here are some examples:

```
format(11.78345, "20.3f")
```

```
' 11.783'
```

```
format(11.78345, "<20.3f")
```

```
'11.783 '
```

```
format(11.78345, ">20.3f")
```

```
' 11.783'
```

Scientific notation is just as easy. You replace the "f" with "E" or "e":

```
format(11.78345, "20.3e")
```

```
' 1.178e+01'
```

```
format(11.78345, "0.3E")
```

```
'1.178E+01'
```

```
format(11.78345, "0.2e")
```

```
'1.18e+01'
```

Using commas to add commas to large numbers, or % notation to generate percentages:

```
format(9238987345.7219, "5,.2f")
```

```
'9,238,987,345.72'
```

```
format(9238987345.7219, ",.2f")
```

```
'9,238,987,345.72'
```

```
format(9238987345.7219, ",f")
```

```
'9,238,987,345.721901'
```

```
format(0.72325, "%")
```

```
'72.325000%'
```

```
format(0.72325, "20.2%")
```

```
' 72.32%'
```

```
format(1.1, ".2%")
```

```
'110.00%'
```

You can also use `format()` with integers. In this case you can translate integers to decimal, octal, binary, or hexadecimal notation using the codes `d`, `o`, `b`, `x`. There is no precision for integers, only a width value:

```
format(65, "5d")
```

```
' 65'
```

```
format(65, "b")
```

```
'1000001'
```

```
format(65, "o")
```

```
'101'
```

```
format(65, "7x")
```

```
' 41'
```

What to take away from this chapter:

Python has 4 number types: integer, float, complex and boolean

All integers are whole numbers.

A float is any number with a decimal

Booleans are either `True` or `False`.

5 conditions are `Falsy`, everything else is `Truthy`.

Python has many built-in math functions, as well as a math module that can be imported for use in scripts.



The python interactive shell has a built-in help() function.

The difference between round() and format() functions.

## Chapter 5: Operators

Operators in python are symbols the perform operations on operands. A simple example:

```
5+3
```

8

Above, the + symbol is the operator, while the 5 and 3 are the operands. The 8 is the result of the operator function. There are several kinds of operators, which we will go through below.

### Mathematical Operators:

Mathematical operators perform mathematical operations on operands. The mathematical operators in python are:

### Relational (Comparison) Operators:

Relational operators compare the values of operands. When relational operators are used with operands it is called a relational expression. Relational operators return a bool() data type result, True or False. The relational operators in python are

### Logical Operators:

Python's logical operators examine the combination two more boolean expressions (those that return a True or False result):

### Operator Precedence & Associativity:

Below is a table of python operator precedence and associativity. We will discuss many of the items below later in the book, so don't be overly concerned if you realize you don't know what some of them are. Also listed below in this chapter are some operators we won't be covering, but which you can study on your own when you have developed a feel for programming in python.

Operator Precedence determines the order that operators are evaluated. In the chart above, the items at the top are evaluated before the items below them. Take a simple example:

```
5 + 10 * 2
```

If we move left to right, the answer would be 30. But in fact the answer is 25, because multiplication is evaluated before addition. Refer to the chart above and see that multiplication is above addition. Here is another example:

```
5 * (10 + 2)
```

Evaluated from left to right would return the answer 52, but we see that parentheses in the table above (...) are above multiplication, so in fact the 10 + 2 is evaluated first, returning the answer 60.

Associativity refers to the direction operators are evaluated if they are on the same line in the chart. Consider this expression:

```
3+8/4*2
```

We know / and \* are evaluated before +, but what order are / and \* evaluated. In the chart above, the associativity is listed as left to right, so we calculate 8/4 and then multiply that result by 2. Finally add 3, and get the answer 7.0 (/ is float division and will always return a float result: try 3+8//4\*2 in your python shell to get a different result – the return will be an integer, as // returns integers from division – as long as no floats are involved in the equation).

### Less Common Operators:

On your own you can examine less common operators in python:

Bitwise

Identity

What to take away from this chapter:

Operators are symbols that perform operations on operands.

Mathematical operators.

Relational (Comparison) Operators

Logical Operators

Operator Precedence & Associativity

Less Common Operators

## Chapter 6: Strings

Strings in python are simply lists of characters like those on your keyboard. With the exception of some control characters to make new lines and tabs (discussed below), that's it. They can contain letters, numbers, special characters, just about any character on your keyboard. You create a string by enclosing it in quotation marks:

```
mystring = "this is a string with numbers 1234567890"
```

Strings can be empty:

```
mystring = ""
```

You can return the length (total count of characters) in strings with the built-in `len()` function:

```
len("string")
```

5

You can even alternate single and double quotes if you want to have quotes in your string:

```
mystring = "This is the 'best' string"
>
mystring = 'This is the "best" string'
```

Escape sequences can be used to include characters in your string that aren't part of your keyboard character set. For example, you might want to split a string into two parts by inserting a new line in the middle:

```
print("This is line one\nand this is line two")
```

This is line one

and this is line two

The `"\n"` character literally means "new line" to python. When the interpreter encounters it in a print function, it will break the string into two lines.

Here is a table of escape characters in python:

Using these escape sequences you can wrap a string in double quotation marks but still use a double quote in the string itself (same for single quotes):

```
mystring = "This is the \"best\" string"
print(mystring)
```

This is the "best" string

As we discussed earlier, you can concatenate (splice strings together) using the plus operator (`+`):

```
str1 = "This is the beginning,"
str2 = " this is the middle,"
str3 = " and this is the end"
print(str1 + str2 + str3)
```

- - -

This is the beginning, this is the middle, and this is the end

You can use the asterisk character to repeat a string:

The form is `string * n`, where `n` must be an integer or the interpreter will throw an error.

```
str = "knock " * 3
print(str)
```

knock knock knock

```
>
str = "knock " * 2.5
```

Traceback (most recent call last):

File "", line 1, in

TypeError: can't multiply sequence by non-int of type 'float'

Membership Operators will let you test for sub-strings inside a string:

```
str = "hello world"
"hell" in str
```

True

```
"wd" in str
```

False

```
"apple" not in str
```

True

```
"world" not in str
```

False

Strings in python are stored in sequence in the order the characters appear, and those characters can be accessed by reference to their position in the string. A string index starts at 0 and increases by one, moving left to right:

```
happy = h[0] a[1] p[2] p[3] y[4]
```

A reverse index is also available. Unlike the regular index, however, the reverse index starts at -1 at the last letter of the string and increments by one from right to left:

```
happy = h[-5] a[-4] p[-3] p[-2] y[-1]
```

Using these two methods you can reference any character in a string in two ways:

```
str = "hello world"
str
```

'hello world'

```
str[0]
```

'h'

```
str[5]
```

..

```
| | | str[11]
```

Traceback (most recent call last):

File "", line 1, in

IndexError: string index out of range

```
| | | str[10]
```

```
'd'
```

```
| | | str[-11]
```

```
'h'
```

```
| | | str[-1]
```

```
'd'
```

```
| | | >
```

As you see above with `str[11]`, if you attempt to reference a character index that is outside the length of the string, the interpreter will throw an `IndexError`. This is true for regular and reverse indexes.

You can always calculate the last character of a string by finding the length and subtracting one:

```
| | | str = "hello world"
```

```
| | | str
```

```
'hello world'
```

```
| | | >
```

```
| | | len(str)-1
```

```
10
```

```
| | | str[len(str)-1]
```

```
'd'
```

We can also access parts of strings by separating the start and end points with a colon, `stringname[startindex:end_index]`. This is known as a slice.

```
| | | str = "hello world"
```

```
| | | str
```

```
'hello world'
```

```
| | | >
```

```
| | | str[2:3]
```

```
'l'
```

```
| | | str[2:4]
```

```
'll'
```

```
| | | str[1:4]
```

```
'ell'
```

```
| | | str[0:4]
```

```
'hell'
```

If our `end_index` value is greater than the length (number of characters) in the string, the interpreter will simply return the string to the end:

```
str[2:1000]
```

```
'llo world'
```

If you omit the *startindex* or *endindex*, the interpreter simply starts at the beginning, or continues to the end:

```
str[:8]
```

```
'hello wo'
```

```
str[3:]
```

```
'lo world'
```

As with the regular index, you can also use the negative index values to slice your string:

```
str[2:-2]
```

```
'llo wor'
```

Everything in Python is an Object

Like Javascript, everything in python is an object. We use the `type()` function to determine the kind of object we are dealing with.

Of course, it would help to know what an object is, so let's take a preliminary look at them. We will delve into objects more deeply later on in the book.

In order to understand what it means to call something an object, we need to understand what a class is. Put simply, a class is a template which defines the data and methods contained within it. Above we referred to the function `type()`. Methods are exactly the same as functions, except that they occur within a class. So why not call them the same thing? You can if you really want to, but using `method` identifies a particular function as being within a class, so it is a useful distinction to maintain.

```
class new_class
```

```
data
```

```
methods
```

What declaring a class does is actually create a new data type, a type of that class. At this point, nothing has been done, no memory has been allocated. Memory will only be used when we create an object based on this class. Only a template has been created from which you can instantiate and actual object of that class.

So now, when we see this:

```
mynumber = 101  
type(mynumber)
```

We understand we have created an object called `mynumber`, of class "int", and assigned the value of 101 to that object.

Built-in classes have lots of useful methods, which you can call on your variables to transform their values. To do this, use the dot notation. Remember when we defined `str = "hello world"` above?

```
str
```

```
'hello world'
```

```
str.upper()
```

```
'HELLO WORLD'
```

```
str.lower()
```

```
'hello world'
```

```
- - -
```

```
str1 = str.upper()
```

```
str1
```

```
'HELLO WORLD'
```

```
str
```

```
'hello world'
```

The string object class has many methods, including `upper()` and `lower()`, which transform the string to all upper or all lower case values. They do not actually alter the value of the `str` variable, since at the end you can see `str` is the same as it was originally defined. Instead these methods return the changed value that you can chose to assign to a new variable and use in your program.

ASCII Character Codes:

Each character on your keyboard has a corresponding code your computer uses to store in its memory. You can view a list of these codes by doing an internet search for ASCII codes. There are 128 characters in the standard ASCII set.

You can use the `ord()` and `chr()` built in functions in python to return ASCII codes, and the characters they represent:

```
ord('a')
```

```
97
```

```
chr(97)
```

```
'a'
```

Controlling `print()` String Output:

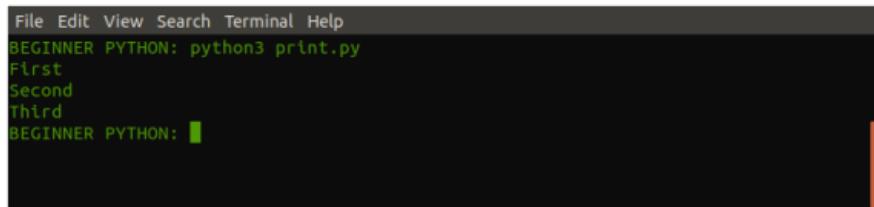
The `print()` function can be modified to alter how it outputs text. Open your text editor and, in a file called `mypython/chapter06/print.py` put in the following text:

```
print("First")
```

```
print("Second")
```

```
print("Third")
```

Now run `python3 print.py` and you'll see the following output:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 print.py
First
Second
Third
BEGINNER PYTHON: █
```

With no input, the `print()` function introduces a new line control code (`\n`) at the string's end. But what if you wanted those three print statements to be on the same line? You could do this:

```
print("First", end="-")
```

```
print("Second", end="-")
```

```
print("Third")
```

And the result:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 print.py
First-Second-Third
BEGINNER PYTHON: █
```

The end value can even be empty:

```
print("First", end="")

print("Second", end="")

print("Third")
```

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 print.py
FirstSecondThlrd
BEGINNER PYTHON: █
```

You can also run the print() function with more than one string at a time:

```
print("one", "two", "three")
```

one two three

And modify the separator between those strings:

```
print("one", "two", "three", sep="-")
```

one-two-three

String Comparison:

Like we did with numbers , we can compare strings using relational operators. But in the case of strings, the comparison is a little more complicated. Take a look at this:

```
"apple" > "application"
```

False

In this case, the interpreter is saying that apple is not greater than application. But what exactly is being compared? It is not the length of the word, len(), as we can see here:

```
"apple" > "appendicitis"
```

True

Here the interpreter says “apple” is greater than “appendicitis”, and clearly apple is not longer than appendicitis.

What the interpreter is doing is actually going character by character and comparing them. Like this:

Does the a of “apple” = the a of “appendicitis”?

If yes, go on to the next character:

p = p? Yes.

p = p? Yes.

l = e? No.

At this point, the interpreter compares the ASCII values of each character:

```
ord("I")
```

108

```
ord("e")
```

101

Is 108 (ASCII for “I”) greater than 101 (ASCII for “e”)? Yes. So the statement “apple” > “appendicitis” is true.

The equals relation operator (==) functions as we would expect. Remember that strings are case sensitive in python:

```
"tree" == "tree"
```

True

```
"trEe" == "tree"
```

False

```
"valve" >= "volume"
```

False

```
"" <= "bitwise"
```

True

String Immutability:

Strings in python are immutable. What this means is once created a string cannot be altered in any way. We can test this by using the id() function, which returns the memory location of the content of a variable:

```
str = "string"
```

```
id(str)
```

139845984714512

```
str = "not string"
```

```
id(str)
```

139845925719024

As you can see, the memory address for the value of the variable str changes when we alter the value of str. This demonstrates that you cannot alter a string once its been created. You can only create a new string object and assign it the variable name.

An implication of the fact strings are immutable is the fact you cannot use their index to change them. Remember we used the [] syntax to return the value of strings by referencing an index of their length. Even though the syntax str[index] can be treated like a variable, we cannot use it to alter a string:

```
str
```

```
'not string'
```

```
str[6]
```

```
'r'
```

```
str[6] = 'q'
```

Traceback (most recent call last):

File "", line 1, in

TypeError: 'str' object does not support item assignment



Once a string is created, it cannot be modified. It is immutable.

## Strings and the format() Function

Just like integer and float values, we can use the format() function to alter strings:

```
format("python rules", "20s")
```

```
'python rules '
```

```
format("python rules", "<20s")
```

```
'python rules '
```

```
format("python rules", ">20s")
```

```
' python rules'
```

```
format("python rules", "2s")
```

```
'python rules'
```

The format built-in function allows you to pad strings beyond their length when you print them out, and to right or left justify as required.

Remember, if you select a length that is less than the actual length of the string, python will automatically expand this number until the entire string is displayed.

What to take away from this chapter:

Strings are simply sequences of characters.

Everything in Python is an Object

ASCII Character Codes describe all the keys on your keyboard, as well as non-printing characters like tabs, in a numeric code computers use to place them in memory

the built-in print() function can have its automatic line breaks removed or even replaced with different characters

strings are compared letter by letter using ASCII character codes

string are immutable

format() can be used to pad and left/right justify strings

## Chapter 7: String Methods

Like other programming languages, python comes with several very useful built-in string manipulation methods. In this chapter we will look at:

String testing

Searching for sub-strings inside strings

String formatting

String conversion.

Remember when we discussed classes before, and identified that when functions exist inside classes, they are called methods? The distinction becomes more important now, because methods are called on the object using a dot notation, rather than the standard function() call:

```
object.method(arguments...)
```

Below we will examine a few string methods. A comprehensive list of string methods is available here:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

Let's spend a little more time examining some of the more common string methods:

```
str.isalnum()
```

Returns True if all characters in str are letters(a-z, A-Z) or numbers(0-9). False otherwise.

```
a = "hello world"
a.isalnum()
```

False

```
"987".isalnum()
```

True

```
"xyz".isalnum()
```

True

```
"xyz987".isalnum()
```

True

str.isalpha()

Returns True if all characters in str are letters (see above). False otherwise.

```
"hello".isalpha()
```

True

```
"hello world".isalpha()
```

False

```
"1234".isalpha()
```

False

```
"XYZ".isalpha()
```

True

str.islower()

Returns true if all characters in str a lower case, and str contains at least one character. False otherwise.

```
"Hello World".islower()
```

False

```
"hello world".islower()
```

True

```
"1234 world".islower()
```

True

```
"".islower()
```

False

```
" ".islower()
```

False

str.isupper()

Returns True if all the characters in str are upper case, and str contains at least one character. False otherwise.

```
"HELLO WORLD".isupper()
```

- - -

True

```
"hELLO WORLD".isupper()
```

False

```
"123".isupper()
```

False

```
"ALL123".isupper()
```

True

```
".isupper()
```

False

```
"A ".isupper()
```

True

There are many other methods to test strings. Take some time to go read through them by using the link above. The python documentation is an excellent resource for you to find built-in methods you can use before building out your own methods to achieve your goals.

Searching and Replacing in Strings:

Many times you will need to find parts of strings and modify them based on what you find. Python provides a suite of methods to accomplish this:

```
str.endswith(suffix, start, end)
```

Returns True if str ends with suffix , False otherwise. We have not covered tuples yet, but suffix can be a tuple, which is essentially a list of items separated by commas. You may specify a beginning location and end location for using the values in the tuple to test against str.

```
str.startswith(prefix, start, end)
```

Returns True if str begins with prefix. The search can also be a tuple, with an option to select the start and end points of the tuple.

```
str.find(part, start, end)
```

Returns the lowest index of part (first occurrence) found in str. Start and end points can be used to limit the search for part to a specific portion of str. If no occurrence is found, returns -1.

```
str.rfind(part, start, end)
```

Returns the highest index of part (last occurrence) found in str. Start and end points can be used to limit the search for part to a specific portion of str. If no occurrence is found, returns -1.

```
str.count(cnt, start, end)
```

Returns the number of cnt found in str, or 0 if there are none. Start and end can be used to limit the search location in str.

```
str.replace(original, replacement, count)
```

Returns the string with every example of original replaced by replacement . Optional count allows you to set the number of changes to make before stopping.

Examine the code below, where examples of all the methods described above are shown.

```
"hello world".endswith("rld")
```

True

```
tup = ("al", "el", "le")  
"apple".endswith(tup)
```

True

```
"""apple".endswith(tup,0,1)
```

False

```
"""hello world".startswith("H")
```

False

```
"""apple".startswith(tup)
```

False

```
str = "this is a string"
str.find("a")
```

8

```
str.find("a",9,99)
```

-1

```
str.find("a",9,15)
```

-1

```
str.find("a",9,14)
```

-1

```
str.find("x")
```

-1

```
str.find("a",8,14)
```

8

```
str
```

'this is a string'

```
str.rfind("t")
```

11

```
str.rfind("t",0,4)
```

0

```
str
```

'this is a string'

```
str.count("is")
```

2

```
str.count("is",7,16)
```

• • •

0

```
| | | >  
| | | str
```

'this is a string'

```
| | | str.replace("a", "not a")
```

'this is not a string'

```
| | | str.replace("s", "q", 2)
```

'thiq iq a string'

Converting Strings:

The following are some methods that can be used to modify strings. Remember these methods do not actually alter the string in any way, but instead return a new string with the changes.

`str.lower()`

Returns str in lower case.

`str.upper()`

Returns str in upper case.

`str.capitalize()`

Returns str and capitalizes the first letter of the string.

`str.title()`

Returns str with the first letter of each word capitalized.

`str.swapcase()`

Returns str with the upper case characters now lower case, and the lower case letters now upper case.

`str.strip()`

Returns str with any spaces or white space removed from the beginning and end

`str.strip(chars)`

Returns str with any spaces or white space removed from the beginning and end, as well as any letters in chars removed from the beginning or end of the string.

Once again examine the example changes made below using these methods:

```
| | | str.capitalize()
```

'This is a string'

```
| | | str.title()
```

'This Is A String'

```
| | | str.swapcase()
```

'THIS IS A STRING'

```
| | | >  
| | | new_str = " this has spaces before and after "
```

```
new_str
```

```
' this has spaces before and after '
```

```
new_str.strip()
```

```
'this has spaces before and after'
```

```
>
str = "...# inside #—"
str.strip(".-")
```

```
'# inside #'
```

You can even get a little fancy:

```
str
```

```
'this is a string'
```

```
str.title().swapcase()
```

```
'tHIS iS a sTRING'
```

String Formatting Methods:

The following methods can be used to modify how strings are displayed.

```
str.center(width, fill)
```

Returns str padded to width , with the content of str centered in the middle. If optional fill is provided, that character will be used to pad the returned string. Otherwise, a standard ASCII space is used. If width is smaller than len(str) str is returned unmodified.

```
str.ljust(width, fill)
```

Returns str padded to width , with the content of str positioned to the left. If optional fill is provided, that character will be used to pad the returned string. Otherwise, a standard ASCII space is used. If width is smaller than len(str) str is returned unmodified.

```
str.rjust(width, fill)
```

Returns str padded to width , with the content of str positioned to the right. If optional fill is provided, that character will be used to pad the returned string. Otherwise, a standard ASCII space is used. If width is smaller than len(str) str is returned unmodified.

```
str
```

```
'my string'
```

```
str.center(3)
```

```
'my string'
```

```
str.center(30)
```

```
' my string '
```

```
str.center(30,"#")
```

```
'#####my string#####'
```

```
>
str.ljust(20)
```

```
'my string '
```

```
str.ljust(20,"@")
```

```

'''
'''
'my string@@@@@@@@@@@@'

''' str.rstrip(20)

' my string'

''' str.rstrip(20,"@")

'@@@@@@@@@@@@@my string'

```

What to take away from this chapter:

methods are called on the object using a dot notation, rather than the standard function() call

use the many built-in python string methods to search and replace in strings

use the many built-in python string methods to convert strings

use the many built-in python string methods to format strings

## Chapter 8: If-else Statements

So far our experience with writing python statements is to either do one at a time in the shell, or put a series of statements in a text file and run them one after the other. In the real world, this is not how programming works. In real programming, the software makes decisions of what steps to take next based on its current operating state. We use Control Statements to give our software the ability to make decisions and produce different outcomes based on different inputs. Control Statements not only control the decision-making flow of programs, but allow us to repeat an action until a desired state is reached, as well as the ability to end these loops when a particular state is encountered.

We can break Control Statements down into 3 main groups:

Decision Control Statements

Loop Conditions

End-Loop Conditions

In this chapter we will look at #1, which involves if-else control statements.

The if Statement:

Just like in natural language, the if statement examines the current state and decides what to do based on that state:

If I'm late for the bus, I'll take a taxi.

Or, in a more python syntax:

if condition:

Our bus example:

if (late for the bus):

And a more python, programmatic approach:

if *currenttime* > *busdeparture\_time*:

taxi

The first line of the statement above is an if clause (if condition:), where "condition:" is a boolean expression that returns a True or False. If the condition returns True, the statement(s) beneath will be executed.

Take a look at how the if statement is structured. In many programming languages, curly braces {} are used to contain blocks of statements. In php, for example, an if statement would look like this:

```
if($currenttime > $busdeparture_time) {
```

```
taxi  
  
}
```

It's useful to be familiar with this type of formatting, as very few software developers work in only one language, and using braces around code blocks is common in many programming languages.

In contrast to many other languages, python uses indentation to identify blocks of statements:

if condition:

<1st statement>

<2nd statement>

<3rdstatement>

In the code above, if the if condition is True, statements 1-3 will be executed. When they are done, statement x will get executed. And if this if condition returns False, statements 1-3 will be ignored and statement x will be executed.

Indentation distance is important. The standard convention for python programming is four spaces. You should make sure you are using spacing of 4 characters. If you are using a plain text editor or IDE, there should be an option to set the number of spaces in a tab. Most likely this will already be set to 4. If your tab spacing is set to 4 spaces, then that will make it easy to ensure you are using the correct indentation distance.

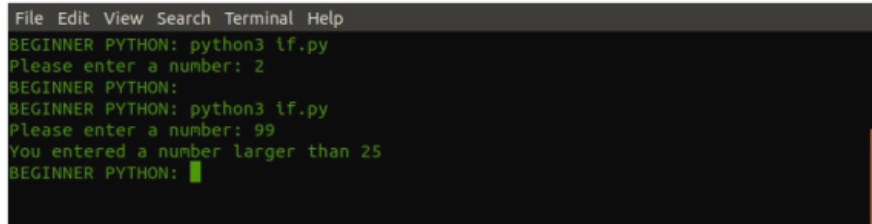
So let's try a practical example. Open your text editor and in a file called if.py put in the following:

```
my_number = int(input("Please enter a number: "))
```

```
if my_number > 25:
```

```
    print("You entered a number larger than 25")
```

Now run python3 if.py and test it out:



```
File Edit View Search Terminal Help  
BEGINNER PYTHON: python3 if.py  
Please enter a number: 2  
BEGINNER PYTHON:  
BEGINNER PYTHON: python3 if.py  
Please enter a number: 99  
You entered a number larger than 25  
BEGINNER PYTHON: █
```

In the example output above, when the program asked for a number, 2 was entered. If we look at the code, 2 is not greater than 25, so the condition returned False and the print statement was not executed. When the program was run a second time and 99 was entered for the number, the print statement was executed and "You entered a number larger than 25" was printed out.

Let's try this example, ifstatementexample.py:

```
number = int(input("Enter a number: "))
```

```
if number > 25:
```

```
    print("execute statement 1")
```

```
    print("execute statement 2")
```

```
    print("execute statement 3")
```

```
print("this statement executes even if the condition is not met")
```

```
print("program ends")
```

And the output:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 if_statement_example.py
Enter a number: 100
execute statement 1
execute statement 2
execute statement 3
this statement executes even if the condition is not met
program ends
BEGINNER PYTHON:
BEGINNER PYTHON: python3 if_statement_example.py
Enter a number: 2
this statement executes even if the condition is not met
program ends
BEGINNER PYTHON: █
```

As before, when the condition is True (100 entered, which is greater than 25), the if statement's condition block is executed. But in the second run, when the number entered is less than 25, the conditional statements are not run, although both of the last statements, the ones outside the if block's indented statements, always run.

You can create if statements in the shell as well. The interpreter will react when you enter an if statement:

```
num = 10
if num > 1:
```

...

You'll notice that as soon as you enter an if statement and hit return, the interpreter extends the line with an ellipse. To continue with your if statement, simply include four spaces and continue. But take note, the shell does not include the tab for you. If you put in another line without indenting the interpreter will throw an error:

```
num = 10
if num > 1:
```

```
... print("number is greater than 1")
```

File "", line 2

```
print("number is greater than 1")
```

^

IndentationError: expected an indented block

```
>
```

Here the interpreter is pointing out it expected an indent because it recognizes you have begun an if statement. Simply indent and continue:

```
num = 10
if num > 1:
```

```
... print("number is greater than 1")
```

...

To complete the if statement, simply hit return a second time. The if condition will be evaluated and any output will be sent to the screen:

```
num = 10
if num > 1:
```

```
... print("number is greater than 1")
```

...

number is greater than 1

█ █

So far so good. But up until now the script we have written does nothing if the condition returns False. In most cases, you will want your scripts to do one thing if the condition is True, and something different if the condition is False.

The if-else Statement:

To continue with our real-language example:

If I'm late for the bus,

I'll take a taxi,

or else, I'll take an Uber

The syntax of an if-else statement looks like this:

if condition:

else:

In this code, if the condition is True, statements 1-3 are executed. But if the condition is False, then statements a-c are executed.

Let's try an example, a converter that converts miles to kilometers, convert.py:

```
miles = int(input("Enter miles: "))
```

```
if miles >= 0:
```

```
kms = miles*1.60934
```

```
print(miles, "miles = ", kms, "kms")
```

```
else:
```

```
print("You must enter a positive number")
```

And when we launch this script it outputs the following:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 convert.py
Enter miles: 25
25 miles = 40.2335 kms
BEGINNER PYTHON:
BEGINNER PYTHON: python3 convert.py
Enter miles: -10
You must enter a positive number
BEGINNER PYTHON: █
```

Now our program only accepts positive numbers of miles to convert to kms. If a negative number is entered, the else condition asks for a positive number. You will notice that if you enter a non-integer for the miles, the script crashes:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 convert.py
Enter miles: apple
Traceback (most recent call last):
  File "convert.py", line 1, in <module>
    miles = int(input("Enter miles: "))
ValueError: invalid literal for int() with base 10: 'apple'
BEGINNER PYTHON: python3 convert.py
Enter miles: 2.2
Traceback (most recent call last):
  File "convert.py", line 1, in <module>
    miles = int(input("Enter miles: "))
ValueError: invalid literal for int() with base 10: '2.2'
BEGINNER PYTHON: █
```

As we continue with if-else statements, you should think of a way to use them to test for this condition and respond to the user appropriately if a non integer is entered.

Make sure in your if statements that you maintain the correct indentation of your if and else statements. If the indentation changes between them, the script will crash:

```
n = 10
if n > 5:
... print("greater than 5")
... else:
File "", line 3
else:
^
```

SyntaxError: invalid syntax

```
>
```

For an exercise, try writing a short script that asks for a user's password, and if it is "x23f" have the script respond "correct", otherwise have it respond "incorrect".

Nested if and if-else statements:

It is possible to make more complex decisions by nesting if and if-else statements inside one another. Open your text editor and in a file called grad.py put in the following code:

```
gpa = int(input("Enter your Undergrad % average: "))
gre = int(input("Enter your GRE score: "))
if gpa > 75:
```

## this is the outer if statement block

```
if gre > 175:
```

## this is the inner if statement block

```
print("You are eligible for grad school")
else:
print("Sorry, you are not eligible for grad school")
```

Above an if statement has been written inside another if statement. We call the inner if statement "nested", because we "nest" a statement inside another statement.

How it works:

After receiving both a gpa and gre values, the script then evaluates the first if statement condition. If gpa > 75, the control is then passed on to the nested if statement, where it's condition is evaluated. If gre > 175 the condition evaluates to True, and the code block for this if statement is run, printing out the user is eligible. If the nested if condition evaluates to False, the nested code block is not run and there is no output.

If the first condition, gpa > 75 evaluates to false, the control is not passed to the nested if condition and the script outputs "Sorry, you are not eligible for grad school".

When we run this scrip, we get this output:

BEGINNER PYTHON: python3 grad.py

Enter your Undergrad % average: 70

Enter your GRE score: 110

Sorry, you are not eligible for grad school

BEGINNER PYTHON:

BEGINNER PYTHON: python3 grad.py

Enter your Undergrad % average: 90

Enter your GRE score: 110

BEGINNER PYTHON:

BEGINNER PYTHON: python3 grad.py

Enter your Undergrad % average: 90

Enter your GRE score: 180

You are eligible for grad school

BEGINNER PYTHON:

Can you spot the flaw? When the gpa meets our minimum, but gre does not, there is no output at all. The nested if statement does not have an else. We can correct this:

```
gpa = int(input("Enter your Undergrad % average: "))
```

```
gre = int(input("Enter your GRE score: "))
```

```
if gpa > 75:
```

## this is the outer if statement block

```
if gre > 175:
```

## this is the inner if statement block

```
print("You are eligible for grad school")
```

```
else:
```

```
print("Your GRE is too low. Maybe retake the exam?")
```

```
else:
```

```
print("Sorry, you are not eligible for grad school")
```

This version works identically to our previous example. The only change is we added an else clause to deal with the lack of response when gpa is good but gre is not. By handling the False condition of the nested if statement, we make sure the user is alerted about why he or she is not eligible to go to grad school.

When we re-run the script again provides feedback for each condition, including if the gpa is good enough but the gre is not:

BEGINNER PYTHON: python3 grad.py

Enter your Undergrad % average: 90

Enter your GRE score: 150

Your GRE is too low. Maybe retake the exam?

## BEGINNER PYTHON:

Once again, when nesting if and if-else statements, pay attention to your indentation. If you do not keep them inline you will end up with the interpreter complaining about a syntax error.

Nesting if-else statement inside else clause:

Lets try an example of some if statements nested inside else clauses. Open your text editor and in a file called `favorite_number.py` put in the following code:

```
fav_num = int(input("Enter your favorite number between 1 and 100: "))

if fav_num > 100:

    print("Your number is too large")

else:

    if fav_num >= 75:

        print("You like large numbers")

    else:

        if fav_num >= 25:

            print("You like numbers in the middle")

        else:

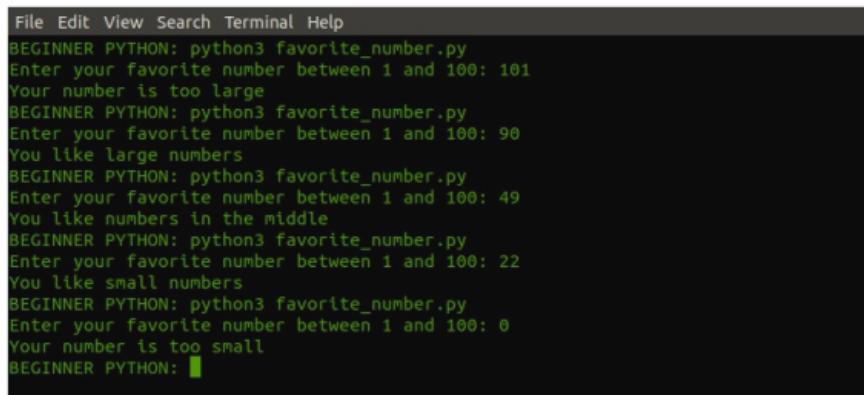
            if fav_num > 0:

                print("You like small numbers")

            else:

                print("Your number is too small")
```

When we run this program several times and enter in 101, 90, 49, 22, 0 (examples to hit each of our if-else conditions), we see the following:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 favorite_number.py
Enter your favorite number between 1 and 100: 101
Your number is too large
BEGINNER PYTHON: python3 favorite_number.py
Enter your favorite number between 1 and 100: 90
You like large numbers
BEGINNER PYTHON: python3 favorite_number.py
Enter your favorite number between 1 and 100: 49
You like numbers in the middle
BEGINNER PYTHON: python3 favorite_number.py
Enter your favorite number between 1 and 100: 22
You like small numbers
BEGINNER PYTHON: python3 favorite_number.py
Enter your favorite number between 1 and 100: 0
Your number is too small
BEGINNER PYTHON: █
```

How this works:

When you enter a number and hit return, that value is stored to the `favnum` variable and the first if condition is run. If the condition is True and `favnum > 100`, the script outputs “Your number is too large”. If the condition evaluates to False the control continues to the next line where the next if condition is evaluated. If it is True and `favnum >= 75` the script outputs “You like large numbers”. If the next condition also evaluates to False, the control continues to the next line where the next if condition is evaluated. If it is True and `favnum >= 25` the script outputs “You like numbers in the middle”. If the condition evaluates to False, control continues to the next line and the next condition is evaluated. If this condition is True and `favnum >= 1` the script outputs “You like small numbers”. If this condition evaluates to False, control continues to the last nested statement. If the condition evaluates True and `favnum` is  $< 1$  then the program complains “Your number is too small”. At this point the script exits the outer if statement and returns control of the script to the next line after the nested statements.

Nesting if-else statements does allow us to build fairly complex decision-making systems, but they are not easy to read, and, more importantly, not easy to write. Code writing complexity is something to be avoided. Simplicity is key, because large, complicated programs can become impossible to maintain or add to if they are difficult to read. We can make these kind of nested statements much more simple by using else-elif-else statements.

The else-elif-else statement:

Take a look at this example code:

if condition#1:

## if block

statement 1

statement 2

statement n

elif condition#2:

## elif block #1

statement 1

statement 2

statement n

elif condition#3:

## elif block #2

statement 1

statement 2

statement n

... #continue adding as many elif statements as required

else

statement 1

statement 2

statement n

continue program

Functionally, the above code works the same as nested if-else statements. When the script runs condition#1 is evaluated. If it evaluates to True, then the statements in that block are run and the code jumps out of the if statement and continues running the lines beneath (continue program). If condition#1 evaluates to False, control is passed to the elif condition#2. If condition#2 evaluates to True then the code block beneath # elif block #1 is run. If condition#2 evaluates to False then control is passed on to elif condition#3. This next condition is then evaluated, and if #3 evaluates to True the block beneath # elif block #2 is run. Otherwise, if the condition#3 evaluates to False, then control passes to the final else block and those statements are run.

Let's rewrite *favoritenumber*, *favoritenumber2.py*:

```
fav_number = int(input("Enter your favorite number between 1 and 100: "))
```

```
if fav_number > 100:
```

```

print("Your number is too large")

elif fav_number >= 75:

print("You like large numbers")

elif fav_number >= 25:

print("You like numbers in the middle")

elif fav_number > 0:

print("You like small numbers")

else:

print("Your number is too small")

print("End program")

```

Just looking at the code difference, it is much easier to read and understand what the script will do when it is run. And, like *favoritenumber.py*, *when we run favoritenumber2.py*:

```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 favorite_number2.py
Enter your favorite number between 1 and 100: 101
Your number is too large
End program
BEGINNER PYTHON: python3 favorite_number2.py
Enter your favorite number between 1 and 100: 90
You like large numbers
End program
BEGINNER PYTHON: python3 favorite_number2.py
Enter your favorite number between 1 and 100: 47
You like numbers in the middle
End program
BEGINNER PYTHON: python3 favorite_number2.py
Enter your favorite number between 1 and 100: 22
You like small numbers
End program
BEGINNER PYTHON: python3 favorite_number2.py
Enter your favorite number between 1 and 100: 0
Your number is too small
End program
BEGINNER PYTHON:

```

By using the same inputs we see the exact same results. In this second version we added a notice outside the if block to announce the program has reached the end. But other than this simple change, the if-elif-else code is behaving exactly the same.

What to take away from this chapter:

We can break Control Statements down into 3 main groups: Decision Control Statements, Loop Conditions, and End-Loop Conditions

the if statement examines the current state and decides what to do based on that state.

if-else statements expand on the if statement, and execute the if code block if the if condition evaluates to True, otherwise, if False, the else code block is executed.

If and if-else statements can be nested inside one another

If-elif-else examines condition 1, condition 2,...condition n, running the appropriate code block if a True condition is found. If no True condition is found, the else code block is executed. Often the if-elif-else statement can replace difficult to write and read nested if and if-else statements

## Chapter 9: Loops

Loops allow us to run a set of statements over and over until a particular condition is reached.

Say for example we wanted to output "Today is Thursday" 100 times. One option would be simply open a text file and put in print("Today is Thursday") 100 times:

```
print("Today is Thursday") #1st line
```

```
print("Today is Thursday") #2nd line
```

```
print("Today is Thursday") #3rd line
```

```
print("Today is Thursday") #4th line
```

## endless typing later...

```
print("Today is Thursday") #100th line
```

This would do the trick, and when run this script would output “Today is Thursday” 100 times. But what happens tomorrow? We would need the script to output “Today is Friday”. Now, many text editors have a search and replace function, so it would be trivial to search for “Thursday” and replace with “Friday”. But what if you were then told you had to make it output the line 1000 times? Well, copy the 100 lines you have and past them 9 more times. Now it will out put 1000 times “Today is Friday”. Except the next day is Saturday, so back to the script you would have to go. Another search and replace (“Saturday” for “Friday”). Now you’re asked to output this line 651 times. So scroll down to the 652 line and delete from there to the end. Finally, you’re asked to change each line to two lines: “Today is Saturday” on the first, and “Tomorrow is Sunday” on the next, each of these double lines running for 625 times. Oh, and when that’s done, it’s now Sunday, so change the days appropriately.

As this script gets more complicated, you can see how difficult it would be to maintain. This is one of the purposes of loops. Instead of writing all those lines, you could just write this:

```
i = 1
```

```
while i <= 100
```

```
print("Today is Thursday")
```

```
i += 1
```

That’s it! And you can see how easy it would be to change “Thursday” to “Friday” or make the loop run 1000 times instead of 100. Even adding the second line would be very simple add print(“Tomorrow is Friday”) beneath the “Today” line and you would be done.

But for or now, you don’t worry about understanding the syntax in the example loop above. We will cover loop syntax in detail below.

There are two kinds of loops in python: while and for loops.

The while loop:

The while statement repeatedly executes code as long as a condition is met. It’s syntax looks like this:

```
while condition:
```

```
statement 1
```

```
statement 2
```

```
statement n
```

## end of while loop, rest of program follows

```
statement 1
```

```
...
```

The first line is the while clause, which is similar to a for statement. The condition of the while statement a boolean that evaluates to True or False, again, just like an if statement. The block of indented statements after the while clause are called the loop body, or the while block. The option is where the state of the program is changed. This change will be evaluated by the while condition in the next loop . In the example above this was i += 1, which means increment the variable i by 1. If we did not have something changing the state, then the condition would never evaluates to False, and the loop would run forever. A never ending loop is a bug that can cause programs, and even computers, to crash.

As we have seen before, we continue to use indentation to separate the while code block from the rest of the script.

Now let’s look at the script we used to solve our “Today is Thursday” dilemma. Save it as thusday.py:



```

i = 1

while i <= 100

print("Today is Thursday")

i += 1

```

In the first line the script sets the variable `i` equal to 1. This sets the state that the while condition will test against. When control is passed to the while loop, the condition is evaluated. Because `i = 1`, `i <= 100` evaluates to `True`. If this condition is true, then any statements in the while block are executed. The first statement prints “Today is Thursday”, while the second statement increases the value of `i` by 1. After all the statements are executed, control is passed back up to the while condition and it evaluates again, using the new state (`i` is now equal to 2). If the condition is still true, the statements in the while block are executed again. `i` is increased by 1, and the process continues until `i` is greater than or equal to 100. At this point, the while condition evaluates to `False`, the script exits the while loop, and control transfers to the next statement in the code after the while loop.

Let’s create a new while loop that prints out the sum of numbers between 0 and 100, called `sumto100.py`:

```

i = 1

the_sum = 0

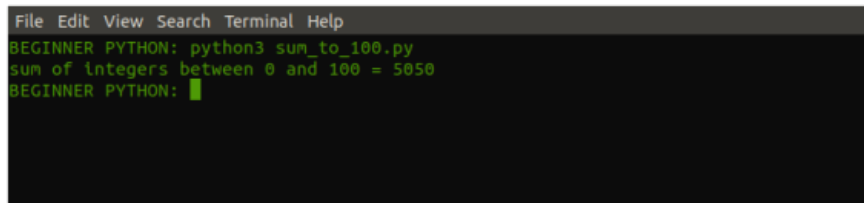
while i < 101:

    thesum += i # same as thesum = the_sum + i

    i += 1

print("sum of integers between 0 and 100 =", thesum) # print the thesum

```



```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 sum_to_100.py
sum of integers between 0 and 100 = 5050
BEGINNER PYTHON:

```

While `i` is less than 100, this while loop continues executing. The variable `thesum` is used to sum up the numbers from 0 to 100. During each iteration, the `i` value is added to `thesum` variable, and `i` is increased by 1. When `i` finally reaches 101 execution stops, the loop terminates, and control passes from the while loop to the rest of the script, in this case executing the `print()` function.

Take a careful look at the code `i += 1` – this line is the same as `i = i + 1`, which simply increased the value of the `i` variable by one at the end of each loop. Above we discussed what happens if the state inside a while loop never changes, meaning the while condition would evaluate to `True` forever. Leaving out the line that increments the `i` variable would leave the code in an infinite loop.

For an experiment, try replacing the line `i += 1` with `print(the_sum)` :

```

i = 1

the_sum = 0

while i < 101:

    thesum += i # same as thesum = the_sum + i

    print(the_sum)

print("sum of integers between 0 and 100 =", thesum) # print the thesum

```

Now run the altered script. You will see the output of `thesum` scroll very quickly down the window. Because `thesum` is still increasing by `i` every loop, it is basically counting from 1 to infinity. Since the variable `i` never changes in value (only `the_sum` is increasing), the while condition will never be met. This is a race condition, an infinite loop that will continue until the system runs out of memory (depending how smart the OS memory management is). At any rate, it is a bad idea and should be stopped as soon as possible. You can use `Ctrl + c` to break out of the loop:

```
File Edit View Search Terminal Help
4226370
4226371
4226372
4226373
4226374
4226375
4226376
4226377
4226378
4226379
4226380
^C4226381
Traceback (most recent call last):
  File "sum_to_100.py", line 5, in <module>
    print(the_sum)
KeyboardInterrupt
BEGINNER PYTHON: █
```

Not all infinite loops are race conditions. Sometimes such a loop is required as part of a system's functionality. For example, consider a script that converts US dollars to Canadian dollars, when the exchange rate is 1.32. Enter the following and save as `dollar_converter.py`:

```
keep_looping = 'y'

while keep_looping == 'y':

    usd = int(input("Enter US dollar amount: "))

    ca = usd*1.32

    print("$",format(usd, "0.2f"), "USD is equivalent to $", format(ca, "0.2f"), "Canadian dollars")

    keep_looping = input("\nConvert another value? Press y for Yes, or any other key to quit:")

print("PROGRAM ENDS")
```

When we run this program, it will ask you to enter in an amount in US dollars. It then converts the USD into Canadian dollars at an exchange of 1.32 % and prints the result. With this loop finished, the program then asks if you'd like to run it again: "Convert another value? Press y for Yes, or any other key to quit:"

If you enter y, the loop will run again and you will be asked to input a new USD amount to be converted. If you enter any other key, however, the loop will terminate, program control comes out of the while loop and "PROGRAM ENDS" is printed.

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 dollar_converter.py
Enter US dollar amount: 10
$ 10.00 USD is equivalent to $ 13.20 Canadian dollars

Convert another value? Press y for Yes, or any other key to quit: y
Enter US dollar amount: 20
$ 20.00 USD is equivalent to $ 26.40 Canadian dollars

Convert another value? Press y for Yes, or any other key to quit: q
PROGRAM ENDS
BEGINNER PYTHON: █
```

We have created an infinite loop here. As long as the user selects y to calculate another currency conversion, the loop will continue forever. However, the decision to loop is in the hands of the user and not up to the program itself, so there is no race condition where the system memory can be overused

The for loop:

In Python for loops iterate code blocks over sequences.

In order to continue with the for loop discussion, we'll have to answer the question "what are sequences?"

A sequence is just a generic term used to refer to collections of data of the following types:

string

list

tuple

dictionary

set

We have only discussed strings from the items above. We will deal with the rest later on.

This is the for loop syntax:

for i in sequence:

statement 1

statement 2

statement 3

statement n

By now, this syntax should be starting to become familiar. The first line is the for clause. It is followed by indented statements, which make up the for code block. This block will be run every time the for loop runs.

Let's use a list sequence with these values: [10, 51, 9, 88, 187]. The code to iterate over this list would be:

for i in [10, 51, 9, 88, 187]

statement 1

statement 2

statement 3

statement n

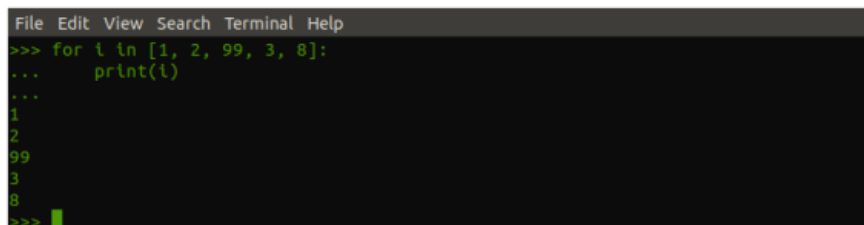
Lists in python are made up of a list of values separated by commas and enclosed in square brackets []. Lists will be discussed in detail later in the book.

Here is how the code above works:

The script reaches the for loop and passes control to it. The first value in the list, 10, is assigned to the variable i, and then the for code block is executed. Once this is done, the second iteration takes place, where i is assigned the value 51 and the for code block is executed. The process repeats for each value until the for loop reaches the end of the list. Once i has been assigned the value 187 and the for code block is executed, the for loop recognizes there are no more values in the list to iterate over, and control is passed from the for loop back out to the rest of the script.

Let's look at some examples:

Here is a for loop that iterates over the items, one by one, in list



```
File Edit View Search Terminal Help
>>> for i in [1, 2, 99, 3, 8]:
...     print(i)
...
1
2
99
3
8
>>>
```

In this for loop, the iteration is over the characters of a string, "apple".

```
File Edit View Search Terminal Help
>>>
>>> for i in "apple":
...     print(i)
...
a
p
p
l
e
>>>
```

Using a for loop and setting parameters with the range() function:

The for loop is useful for iterating over sequences, but what if we want to iterate over a code block a known number of times using a for loop? If we know in advance how many times we want to iterate, we could build a list or other sequence to hold those values, or we could use the built-in python function range(), which makes such things much easier.

The range() function returns an object of type iterable. This is just like a sequence, an iterable object returns items successively as the for loop iterates over it.

The most basic range() syntax looks like this:

range(x)

Where x is a integer value of the int type.

Let's examine a simple example. The following code will create a for loop that iterates over its code block 7 times:

```
File Edit View Search Terminal Help
>>> for i in range(7):
...     print(i)
...
0
1
2
3
4
5
6
>>>
```

In the example above, the built-in function range(7) returns an iterable sequence of numbers from 0 to 6. In the first loop, range() returns the value 0, and this is attached to the variable i. The loop body is then executed, causing the system to print out the value 0 to the console. Then the second iteration takes place. The range() function returns 1, this is assigned to i, and the loop body runs again. The loop finally exits when range() returns 6, this value is assigned to i, and the body of the loop executes for the last time. Control is returned to the rest of the script.

The script above is equivalent to this:

```
File Edit View Search Terminal Help
>>> for i in [0,1,2,3,4,5,6]:
...     print(i)
...
0
1
2
3
4
5
6
>>>
```

But you can see how using the built-in range() function is an advantage. What if you need to run the for loop 100 times? Or 100,000?

There are more complex uses of the range() function. They are:

range(a,b)

range(a,b,c)

range(a,b) returns values starting with a and increments by one until reaching b: a, a+1, a+2 ... b-1

range(a,b,c) is similar to range(a,b), except instead of incrementing by 1, it increments by c: a, a+c, a+2c, a+3c...b-c.

range(0, 16, 4) would return the following iterable sequence: 0, 4, 8, 12

Each number in the sequence returned is separated by the third value, in this case 4, in both directions. That is to say, you could start on the left and continue to add the number for moving right. Or, you could start on the right and delete by 4 moving left.

Let's look at some examples:

```
File Edit View Search Terminal Help
>>> for i in range(10, 18):
...     print(i)
...
10
11
12
13
14
15
16
17
>>>
```

```
File Edit View Search Terminal Help
>>> for i in range(10, 90, 10):
...     print(i)
...
10
20
30
40
50
60
70
80
>>>
```

It is also possible to reverse the direction of the range, like so:

```
File Edit View Search Terminal Help
>>> for i in range(30, 10, -2):
...     print(i)
...
30
28
26
24
22
20
18
16
14
12
>>>
```

In this example, the code will iterate over the range and divide each by 2, outputting the range value and that value divided by 2 (rangebyhalf.py):

```
print("Range\t | Half")
```

```
print("—————")
```

```
for x in range(1, 10):
```

```
print(x, "\t | ", x/2)
```

And here is the output:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 range1.py
Range | Half
-----
1 | 0.5
2 | 1.0
3 | 1.5
4 | 2.0
5 | 2.5
6 | 3.0
7 | 3.5
8 | 4.0
9 | 4.5
BEGINNER PYTHON: █
```

What to take away from this chapter:

The while statement repeatedly executes code until the required condition is no longer met.

Failure to modify the state condition is measured against can lead to an infinite loop race condition, which can fill the system memory and crash the program or even computer.

Not all infinite loops are negative. Sometimes they are needed for software to run properly.

The for loop is used to iterate over sequences

the range() function is designed to be used to iterate over a for loop a known number of times.

## Chapter 10: Break and Continue

The break statement , when its conditions are met, breaks out of the current loop and returns control to the code outside the loop, even if there are many iterations remaining in the loop.

This is the break syntax:

```
break
```

Here is a very basic example of break in action:

```
for x in range(1, 20):
```

```
    if x == 7:
```

```
        break
```

```
    print("x = ",x)
```

```
    print("break has fired, control returned to the script outside the loop")
```

And the output:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 break_example.pl
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
break has fired, control returned to the script outside the loop
BEGINNER PYTHON: █
```

The standard for loop runs, assigning the value 1 to x. The if condition is checked, returns False (1 != 7), so break is not run. The print function displays the current value for x. The first iteration of the for loop is done, so control is returned to the for condition, x is assigned to next value (2), and so on. But on the seventh loop, when x = 7, the if condition is met and break is fired. Control drops out of the outer for loop and back into the script, where "break has fired, control returned to the script outside the loop" is printed.

It's often helpful to consider a real-world, or at least practical example. So let's write a script that identifies if a selected number is a prime number.

```
num = int(input("Your number: "))
```

```

num_prime = True

for x in range(2, num):

    if num % x == 0:

        num_prime = False

        break

if num_prime:

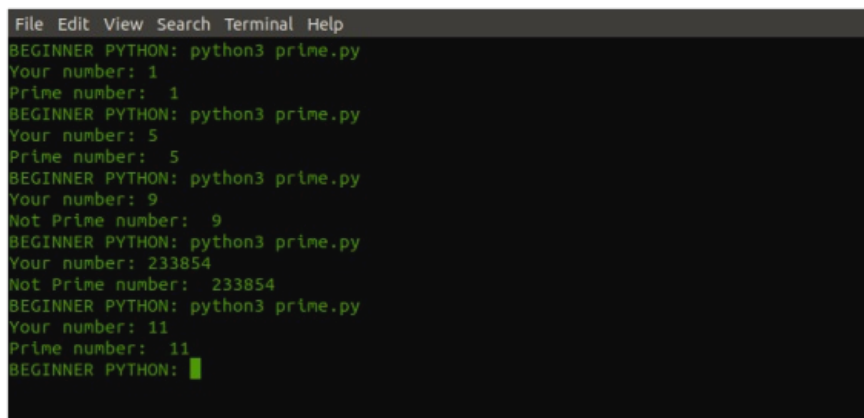
    print("Prime number: ", num)

else:

    print("Not Prime number: ", num)

```

Here is the output:



```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 prime.py
Your number: 1
Prime number: 1
BEGINNER PYTHON: python3 prime.py
Your number: 5
Prime number: 5
BEGINNER PYTHON: python3 prime.py
Your number: 9
Not Prime number: 9
BEGINNER PYTHON: python3 prime.py
Your number: 233854
Not Prime number: 233854
BEGINNER PYTHON: python3 prime.py
Your number: 11
Prime number: 11
BEGINNER PYTHON:

```

Prime numbers are integers that can only be divided by 1 and themselves. If a number can be divided by any other number with a 0 remainder it is not a prime number.

So let's examine the code above. First, the script pauses and requests a number from the user. The variable `num` is set to this value. The variable used to maintain the test state, `num_prime`, is set to `True`. Then the `for` loop begins, using a range from 2 to the variable `num`. The variable `x` is given value provided by `range()`.

The script attempts to divide the `num` variable value by the current range value, `x`. If the division has a non-zero remainder it means `num` cannot be divided by the current value of `x`. The `if` statement's condition is not met, *num\_prime is not changed, and break is not called. The script completes the current loop and returns to the beginning, assigning `x` with the new value returned by `range()`. If during one of these loops the `if` condition is `True`, and dividing `num` by `x` results in a 0 remainder, then the number is not prime, so `num_prime` is set to `False` and `break` is fired (there is no point to continue testing, as just one number is enough to prove the test number is not prime). The loop then ceases and control is returned to the main script.*

Either way, once control is returned to the main script (either because the script has iterated through all the `range()` options returned, or because `num` is shown not to be a prime number) the system then uses another `if` statement to decide if `num_prime` is *true or false. If `True`, the system uses the `print` function to print a notice that the number is prime. If `num_prime` is `False`, the user's number is described as not prime.*

The `break` statement inside a nested loop

The `break` statement only breaks the current loop, even if that loop is running inside another loop.

Here is a simple example, `break_nested.py`

```

for i in range(1, 3):

    print("This is the outer loop, i = ", i)

    for x in range (65, 75):

```

```

print("\nInner loop ASCII chr(x) =", chr(x))

if chr(x) == 'C':

print("\nBREAK OUT INNER LOOP...")

break

print("-----")

```

The script begins with the for condition and iterates over 1...3, setting this value to i. To make what it is doing visible, it prints out the value of i.

The next step is to enter the nested for loop, which iterates over a range between 65...75 and assigns each value returned by range() to x. Now the system prints out the current state of x, but displaying the ASCII character that matches the value of x using the built-in chr() function.

The next line is an if statement, which returns true if the ASCII value of x is equivalent to the letter 'B'. If this statement is False, the code block is not executed, control returns to the inner loop, x is set to the next range() value, and it is tested against 'B' again. But if the ASCII value is equivalent to 'C', the condition evaluates to True and the inner for loop's code block is executed. The script announces it will be breaking out of the loop, and break is called. This exists the current loop and returns control to the outer loop, which updates i with the new value returned by range(), and so on.

Finally, at the end of the outer for loop, a print statement prints a series of dashes to visually separate each outer loop iteration.

Why don't we have a look at the code output:

```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 break_nested.py
This is the outer loop, i = 1

Inner loop ASCII chr(x) = A
Inner loop ASCII chr(x) = B
BREAK OUT INNER LOOP...
-----
This is the outer loop, i = 2

Inner loop ASCII chr(x) = A
Inner loop ASCII chr(x) = B
BREAK OUT INNER LOOP...
-----
BEGINNER PYTHON: █

```

We see the first outside loop begin where i == 1. Then the inner loop starts with x == 65 ('A' in ASCII). This does not meet the break condition, so the nested inner loop continues with x == 66 ('B' in ASCII). Again, this does not meet the break condition. On the third iteration of the inner nested loop x == 67 ('C' in ASCII). Now the condition is met (chr(x) == 'C') and the if condition's code block is executed. The system prints out "BREAK OUT INNER LOOP..." and break is executed. This drops control out of the inner nested loop and back to the outer loop, which sets i to the next returned value of range(), 2, and the process repeats.

The continue statement:

You will find continue statements akin to break statements. Each is called within a loop, usually by a condition being met. Each halts the current loop when it is called. But, unlike break, continue does not return control to the outside script. Instead, control stops any further execution of the current loop code block and returns control back to the beginning of the current loop so it can run the next iteration.

Here is an example, continue\_example.py

```

for x in range(1, 20):

if x % 2 == 0:

continue

print("x =", x)

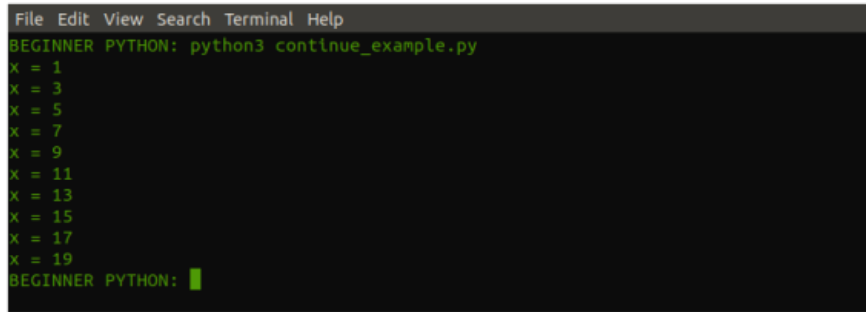
```

Here we see a loop that should run using the output of range() from 1...20. The if statement condition asks if the current value of x is divisible by 2. If this condition evaluates to True, the continue statement is called. As mentioned above, this means control is returned back up to the beginning



of the for loop, x is pointed to the next value provided by range(), and the loop continues. But if the if condition is False, continue is not called and the current value of x is printed in the terminal.

Here is the output from the above code:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 continue_example.py
x = 1
x = 3
x = 5
x = 7
x = 9
x = 11
x = 13
x = 15
x = 17
x = 19
BEGINNER PYTHON: █
```

We have written a simple script to print out odd numbers over a range of integers.

Using continue and break together:

Some code requirements may require using continue and break together. Take a look at this example, `continuebreakexample.py`

```
total = 0

while True:

    val = input("\nEnter a number: ")

    if val == 'q':

        print("Quitting program (break executed)...")

        break

    if not val.isdigit():

        print("Please use whole numbers only (continue executed)...")

        continue

    total += int(val)

    print("Added", val, "to total, which is now:", total)
```

This script will sum any integers provided by the user. The first line defines total, which is used to maintain the running sum of all entered numbers.

The while loop begins the input phase, and the user is asked to provide a number. This is assigned to the variable val, which is then tested. First the program tests to see if val == 'q'. If this condition evaluates to true, the program prints "Quitting program (break executed)..." and break is executed. This moves control out of the while loop and the script ends. If val != 'q' the script continues down the while code block to the next if condition, which is not val.isdigit(). If val is not a digit, then this condition evaluates to True, the script prints out "Please use whole numbers only (continue executed)..." and continue on the next line is executed. This stops the current iteration of the while loop and returns control to the beginning of the while loop for the next iteration. If the condition evaluates to False, then val is a digit and the script continues to add val to the current value of total and print out the current values of val and total. Finally, the script reaches the end of the while block of code and returns to the beginning for the next iteration.

Here is a sample output:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 continue_break_example.py

Enter a number: 5
Added 5 to total, which is now: 5

Enter a number: 7
Added 7 to total, which is now: 12

Enter a number: t
Please use whole numbers only (continue executed)...

Enter a number: q
Quitting program (break executed)...
BEGINNER PYTHON: █
```

What to take away from this chapter:

The break takes stops the current loop and returns control to the code outside script.

If called within a nested loop, break only returns control to the first outer loop

the continue statement also stops the current loop, but unlike break it does not break out of the loop, but instead returns control to the beginning of the loop so the next iteration can take place.

## Chapter 11: Lists

Lists are a sequences . A sequence is a general term used to refer a data type that can hold more than one item of data. There are several types of sequences in python. The three most important are:

lists

tuples

strings

This chapter will focus on the list sequence.

The list in python:

Imagine you wanted to calculate the average amount of groceries purchased by 100 people over one month. A first step to script this problem might be to create a variable to hold the total amount spent by each person over a month, 100 variables:

```
person1 = ""
person2 = ""
person3 = ""
...
person100 = ""
```

But what if you wanted to average 1000 shoppers, or more? Would you set about creating 1000 or more variables. No. As we've learned before, computers and computer languages are built to handle large numbers in simple ways. So for our grocery issue we would create a list.

The list syntax is simple. Within square brackets [] it contains a comma-separated list of items. Here is a generic syntax example.

```
my_list = [item1, item2, item3, ...itemn]
```

Each item stored in a list is an element:

```
groceries = [563.25, 612.35, 492.18, 734.56]
```

The statement above takes 4 grocery amounts and assigns them to a variable called groceries. This could be 40 grocery amounts, or 4000.

Like everything in python, lists are objects. A list's datatype is "list". And, like all variables, groceries above does not store the elements, it stores a reference to the address where the elements are actually stored in memory.

```
type(groceries)
```

```
| | >
```

To display the contents of a list using the python shell, type the name (just like any variable). Or, use the print() function. Again, just like any variable.

```
| | | groceries
```

```
[563.25, 612.35, 492.18, 734.56]
```

```
| | >
```

```
| | | print(groceries)
```

```
[563.25, 612.35, 492.18, 734.56]
```

```
| | >
```

A list is not limited to one data type. You can mix data types inside:

```
mixed_types = ["string here", 1.102, 99]
```

```
| | >
```

```
| | | mixed_types
```

```
["string here", 1.102, 99]
```

```
| | >
```

You can also create empty lists by leaving the content between the crotchets ( [] ) blank:

```
| | | list_empty = []
```

There is built-in python list constructor function:

```
| | | lst1 = list() # creates an empty list
```

```
| | | lst2 = list(["apple", 4, 9.5]) # creates a list with elements of different datatypes
```

```
| | | lst3 = list(["$$$ ", "*", "<<>>"]) # symbol characters are okay if they are part of a string
```

```
| | | lst4 = list("abcdef") # this will create a list from a string
```

```
| | >
```

```
| | | lst1
```

```
[]
```

```
| | >
```

```
| | | lst2
```

```
['apple', 4, 9.5]
```

```
| | >
```

```
| | | lst3
```

```
['$$$ ', '*', '<<>>']
```

```
| | >
```

```
| | | lst4
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

```
| | >
```

You can also use a list as an element in a list:

```
lst5 = [  
... ["a", 2, "3.5"],  
... [99, 999, 109]  
... ]  
  
>  
lst5  
[['a', 2, '3.5'], [99, 999, 109]]  
  
>
```

The variable `lst5` contains 2 elements, both of type list. When you put lists inside lists it's called a nested list or multidimensional list.

Creating lists using the `range()` function:

As we learned above, `range()` can be used to create objects of type iterable. These objects can be large. To use `range()` to create a list, we just pass the iterable object created by `range()` to the list constructor():

```
>  
lst1 = list(range(10))  
lst1  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
>
```

The `range(10)` call generates this sequence of elements:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Here is an example creating a large list:

```
lst2 = list(range(1, 150))  
lst2  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,  
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,  
80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,  
113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140,  
141, 142, 143, 144, 145, 146, 147, 148, 149]  
  
>
```

And using the more advanced `range()` options:

```
lst3 = list(range(0, 200, 25))  
lst3  
[0, 25, 50, 75, 100, 125, 150, 175]  
  
>
```

Common Built-in Functions Available to list:

Here are 4 common functions available to list:

len() - returns # of elements in a sequence

sum() returns sum of all elements in a sequence

max() returns element with greatest value in a sequence

min() returns element with lowest value in a sequence

```
func_list = [1, 100, 5, 66]
>
len(func_list)
```

4

```
>
sum(func_list)
```

172

```
>
max(func_list)
```

100

```
>
min(func_list)
```

1

```
>
```

Lists are similar to strings because both are 0 indexed, meaning that the first element is index[0] (zero), the second element index[1], the third element index[2], and so on. The last index value is always len(list)-1, or one less than the total number of list elements in the list. The syntax to access the list index is:

the\_list[index]

So long as index is an integer. Anything else and the interpreter will

throw an error.

If we create a list with the following statement, we will have a list with 5 elements:

```
list_five = ["x", 43, 1.1, 11, 100]
```

In the above list, *listfive[0]* would return the value "x", *list[1]* would return the value 43, and *listfive[4]* would return the value 100.

```
list_five = ["x", 43, 1.1, 11, 100]
list_five[0]
```

'x'

```
>
list_five[1]
```

43

```
>
list_five[4]
```

```
100
```

```
| | >
```

Remember, the `len()` function built-in can be used to calculate the last index of a list:

```
| | | len(list_five) - 1
```

```
4
```

```
| | >  
| | | listfive[len(listfive) - 1]
```

```
100
```

```
| | >
```

If you try to access an index outside the list you will throw an `IndexError`:

```
| | | list_five[25]
```

Traceback (most recent call last):

File "", line 1, in

`IndexError: list index out of range`

```
| | >
```

Like accessing the elements of a string, elements of a list can also be accessed with negative integers. Negative index values begin at the last element as `-1`, and decrease towards the first element:

```
| | | list_five
```

```
['x', 43, 1.1, 11, 100]
```

```
| | >  
| | | list_five[-1]
```

```
100
```

```
| | >  
| | | list_five[-3]
```

```
1.1
```

```
| | >  
| | | list_five[-5]
```

```
'x'
```

```
| | >
```

Again, should you attempt to access a negative index, one that is outside the valid range of a list), you will throw a `IndexError`:

```
| | | list_five[-22]
```

Traceback (most recent call last):

File "", line 1, in

`IndexError: list index out of range`

```
| | >
```

You can use negative len() to determine a lists first element:

```
> listfive[-len(listfive)]
```

```
'x'
```

```
>
```

Unlike Strings, Lists are Mutable:

If you recall, we discovered in the chapter on strings that the string datatype in python is immutable. If you try to alter the value of a string you are actually creating a new string object. We proved this by examining the memory id of the string variable before and after attempting to change a string's content.

Lists are not the same. Python will allow you to modify lists as you like. For example:

```
nlist = ["apple", "banana", "orange", "grape"]
>
id(nlist)
```

```
140218020185352
```

```
>
nlist[0] = "carrot"
>
nlist
```

```
['carrot', 'banana', 'orange', 'grape']
```

```
>
id(nlist)
```

```
140218020185352
```

```
>
```

As you can see, even after nlist had its 0 index changed from "apple" to "carrot", the id of the variable remained the same. This lets us know no new object has been created, even though the current list object has been modified.

List Element Iteration:

You can use a for loop to repeat over the elements in a list:

```
>
first_floor = [101, 102, 103, 104, 105, 106]
for num in first_floor:
... print(num)
```

```
...
```

```
101
```

```
102
```

```
103
```

```
104
```

105

106

```
| | >
```

The for loop iterates over the variable `first_floor`, beginning with the first `[0]` index and ending with the last `[5]` index, assigning that index value to the variable `num`. Changing the `num` variable has no effect on the list elements, so this is the most common way to iterate over a list when you do not need to modify its elements.

In order to modify a list's elements in a loop, we use the for loop and the `range()` function (or the while loop, see below...):

```
| | | first_floor = [101, 102, 103, 104, 105, 106]
| | | first_floor
```

[101, 102, 103, 104, 105, 106]

```
| | | >
| | | second_floor = first_floor
| | | second_floor
```

[101, 102, 103, 104, 105, 106]

```
| | | >
| | | for num in range(len(second_floor)):
```

```
... second_floor[num] = second_floor[num]+100
```

...

```
| | | second_floor
```

[201, 202, 203, 204, 205, 206]

```
| | >
```

The for loop is most often used to iterate over a list, but it is possible to iterate over a list using while:

```
| | | >
| | | second_floor
```

[201, 202, 203, 204, 205, 206]

```
| | | >
| | | x = 0
| | | >
| | | while x < len(second_floor):
```

```
... print(second_floor[x])
```

```
... x += 1
```

...

201

202

203



204

205

206

```
| | >  
| | >
```

List Slicing:

The slicing operator we discussed in the chapter on strings can also be used on lists. The difference is that instead of returning a slice of a string, it will return a slice of your list. The syntax should be familiar:

`list[start:end]`

```
| | >  
| | | lst1
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
| | >  
| | | lst1[0:5]
```

[0, 1, 2, 3, 4]

```
| | >  
| | | lst1[2:5]
```

[2, 3, 4]

```
| | >  
| | >  
| | | lst1[8:9]
```

[8]

```
| | >
```

Both the start and end indexes are optional. If they are omitted, the default will be the first index, `index[0]` and the last index, `index[len(list)]`:

```
| | >  
| | | lst1
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
| | >  
| | | lst1[:5]
```

[0, 1, 2, 3, 4]

```
| | >  
| | | lst1[2:]
```

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

```
| | >  
| | | lst1[:]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>  
>
```

The Membership Operator: in and not in:

Once again like strings , we can use the in and not in membership operators to check if an element exists in a list:

```
>  
bands = ["pixies", "U2", "Radiohead", "Gorillaz"]  
bands
```

```
['pixies', 'U2', 'Radiohead', 'Gorillaz']
```

```
>  
"pixies" in bands
```

True

```
>  
"Violent Femmes" in bands
```

False

```
>  
"U2" not in bands
```

False

```
>  
"Wham!" not in bands
```

True

```
>  
>
```

List Concatenation:

Lists can be concatenated , joined, using the + operator. If both operands are lists, the + operator creates a new list and simply joins the lists together into one long list:

```
>  
lst1 = [1, 2, 3]  
lst2 = [4, 5, 6]  
>  
id(lst1)
```

```
140218020173384
```

```
id(lst2)
```

```
140218020084872
```

```
>
```

```
lst3 = lst1 + lst2
lst3
```

```
[1, 2, 3, 4, 5, 6]
```

```
>
id(lst3)
```

```
140218078477448
```

```
id(lst1)
```

```
140218020173384
```

```
id(lst2)
```

```
140218020084872
```

```
>
```

Take note that even after we have created the new list, lst3, by joining lst1 and lst2 using the + operator, the id values of lst1 and lst2 have not changed. You can use the += (operator) to concatenate lists. Even when you use the += operator on a list, the existing list is modified. A new list is not created:

```
>
lst1
```

```
[1, 2, 3]
```

```
>
id(lst1)
```

```
140218020173384
```

```
>
lst1 += lst2
>
lst1
```

```
[1, 2, 3, 4, 5, 6]
```

```
>
id(lst1)
```

```
140218020173384
```

```
>
>
```

The statement above, lst1 += lst2 simply adds lst2 to the end of lst1. You can see the memory address of lst1 has not changed, even though it has had elements added to it. Other languages like Java and C have arrays that are fixed in size. In python, lists are not only mutable, they are dynamically sized – that is, the size of the list can grow or shrink as required.

Repetition Operator:

You can also use the \* operator with lists. The syntax:

```
list * n
```

The \* operator copies the list and then joins it to the original, as many times as n:

```
>
lst1 = [1, 2]
>
lst2 = lst1 * 5
>
lst2
```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

```
>
```

You can also use the \* operator as a compound assignment operator \*=. The difference between \* and \*= operators is the latter updates the existing list rather than creating a new one:

```
>
laundry = ["wash", "rinse", "repeat"]
>
id(laundry)
```

```
140218020084872
```

```
>
laundry * 4
```

```
['wash', 'rinse', 'repeat', 'wash', 'rinse', 'repeat', 'wash', 'rinse', 'repeat', 'wash', 'rinse', 'repeat']
```

```
>
laundry
```

```
['wash', 'rinse', 'repeat']
```

```
>
laundry *= 4
>
laundry
```

```
['wash', 'rinse', 'repeat', 'wash', 'rinse', 'repeat', 'wash', 'rinse', 'repeat', 'wash', 'rinse', 'repeat']
```

```
>
id(laundry)
```

```
140218020084872
```

```
>
```

List Comparison:

Like strings, lists can be compared using the relational operators (>, >=, <, <=, ==, !=). List comparison is only possible if the operands have the same type of elements:

```
>
.
```

```
lst1 = ["a", "b", "c"]
lst2 = ["a", "b", 2]

>

lst1 > lst2
```

Traceback (most recent call last):

File "", line 1, in

TypeError: '>' not supported between instances of 'str' and 'int'

```
>
```

Like string comparisons, list comparisons begin by comparing the elements at index[0] in both lists. If they are the same, then the elements at index[1] are compared. The comparison continues until two elements at the same index are different, or the comparison exhausts all the possible index locations.

Check out this example:

```
>

lst1 = ["a", "b", "c"]
lst2 = ["a", "b", "z"]

>

lst1 > lst2
```

False

```
>
>
```

These are the steps the interpreter takes as it compares lst1 and lst2:

Step one, "a" at index[0] from lst1 is compared with "a" at index[0] from lst2. They are the same, so the comparison continues.

Step two, "b" at index[1] from lst1 is compared with "b" at index[1] from lst2. Again, they are the same, so the comparison continues.

Step three, "c" at index[2] from lst1 is compared with "z" at index[2] from lst2. These values are not the same. The (ASCII) value of z is greater than the (ASCII) value of c, so the statement evaluates to False, lst1 is not greater than lst2.

Numerical list comparisons follow the same logic, and the conclusion is reached in the same manner:

```
>

lst1 = ["1", "2", "100"]
lst2 = ["1", "2", "10"]

>

lst1 > lst2
```

True

```
>
```

List Comprehension:

Sometimes you will be required to create lists in which each element in the sequence is the product of an operation, or having each element in the list meet some condition. For example, finding the squared values of a sequence of numbers. We can use list comprehension in these situations. Its syntax is:

[ expression for item in iterable ]

List comprehension functions by assigning a value to item from the iterable object in each iteration. The expression before the for keyword is then evaluated, and the result of the expression is used to populate values for the current index in the list.

Here is an example using squared values from range():

```
>
squared = [ t**2 for t in range(1, 51) ]
>
squared
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500]

```
>
```

Here each value returned by range is squared and added to the list called squared.

It is possible to add an if condition to a list comprehension:

[ expression for variable for iterable if condition ]

This will function the same as the previous comprehension, with the exception that only when condition evaluates to True will the expression be evaluated. Thus:

```
>
squared = [ x**2 for x in range(1, 51) if x % 2 == 0 ]
>
squared
```

[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784, 900, 1024, 1156, 1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500]

```
>
```

This result is the same, except only when x is an even number does the expression get evaluated and added to the squared list.

#### Built in List Methods

Here is a list of the most common built-in list manipulation methods:

append(item) – adds item to the end of list

insert(index, item) – item is inserted at index; if index is larger than last index, item is appended to the end of the list

index(item) – returns index location of the first time item occurs in list

remove(item) – removes first item in the list; if item does not exist in the list it will throw an exception

count(item) – returns number of times item appears in list

clear() – empty list by removing all elements

sort() – sorts list elements in ascending order

reverse() – reverses the element order in a list

extends(sequence) – appends elements in sequence to the end of the list

pop([index]) – removes and then returns element at index; if index is not provided last element is removed and returned; if index is not valid an exception will occur

You have some experience using built-in methods at this point. Try each of the methods above to get a feel for what they do. Remember, methods are invoked using the dot notation: `list.append(item)`.

What to take away from this chapter:

lists are collections of items (a sequence) separated by commas in square brackets[].

lists content can be altered after being created; they are mutable; you can add, edit, and delete list values.

lists can be concatenated.

## Chapter 12: Functions

So far in this book we have used only built-in functions available from the python developers. Now we will discover how to create and then use personal functions. But first, let's look at how functions are valuable in programming.

Consider a program that finds the difference between two numbers. You should have enough experience in writing code that you can write this on your own. It would probably look something like this:

```
diff = 0

num1 = 100

num2 = 95

diff = abs(num1 - num2)

print("Difference between", num1, "and", num2, "is:", diff)
```

This very simple program determines the difference between the two numbers. The `abs()` function removes the negative sign if the `num1` variable happens to be smaller than the `num2` variable (we only want the difference between the two).

But what if you wanted the difference between 1000000 and 993844? One approach would be to simply edit the code and replace the 2 and 100 with these new numbers. But what if we had 1000 sets of numbers we needed to calculate the difference between? Changing the code for each run of the program is impractical. This is the nature of the problems that functions can solve. They allow us to reuse code without having to change anything by hand.

What is a Function?

A function is just a group of statements designed to perform a task. Functions are defined like this:

```
def myfunctionname(arg1, arg2, arg3,...argn):
```

```
...
```

There are two parts here: the definition that begins with reserved keyword “def” (reserved keyword means you should not use them as a variable or function name in your code) which always ends with a colon ( : ), and the function body, which is the list of indented statements beneath the first line.

In the definition, after the `def` keyword, comes the name of the function. It can be any valid string, much like a variable, but it is useful to use a name that describes what the function does. After the name is a list of comma-separated arguments in the parentheses. These arguments are how we pass information to the function. A function can take no arguments, or as many as you need. If you write a function that takes no arguments, you simply leave the parentheses empty.

The function body is a list of statements that defines what the function does. Python uses indentation to identify the function body. Each statement must be indented the same amount, or the interpreter will throw a syntax error.

The last thing to note in a function is the return statement. Returns “return” information back to the code that originally called the function. A return statement is not required and can be omitted. If you omit a return statement in your function it will then return the default, which is “None”. This is an object of type `NoneType`.

Once you know how to create and populate a function, all that's left is to know how to call them. The process of using a function involves coming to a point in your code where you need data manipulated or provided. You “call” a function that does this work, passing along any parameters the function needs to make its calculations (or none if that is the case). The function receives the call (and any arguments you have passed), runs the

block of statements defined in its body, and then returns back to the calling code (if a return is required)

You call a function by simply copying the head of the function, but omit the def and colon:

```
myfunctionname(arg1, arg2, arg3,...argn)
```

Here is a simple statement that prints out what is probably the most common printed code in the software development world:

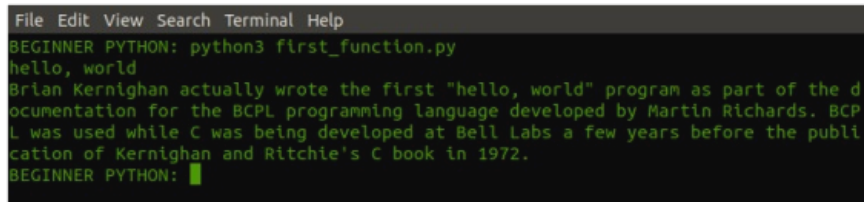
```
def first_hello():
```

```
    print("hello, world\n")
```

```
    print("Brian Kernighan actually wrote the first \"hello, world\" program as part of the documentation for the BCPL programming language developed by Martin Richards. BCPL was used while C was being developed at Bell Labs a few years before the publication of Kernighan and Ritchie's C book in 1972.")
```

```
first_hello()
```

When we run this code, we get this output:

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The prompt 'BEGINNER PYTHON:' is followed by the command 'python3 first\_function.py'. The output consists of two lines: 'hello, world' followed by a newline, and a longer paragraph about Brian Kernighan and the BCPL programming language. The prompt 'BEGINNER PYTHON:' is followed by a green cursor.

Note that the call to a function must come after the function has been defined, otherwise the interpreter will throw a NameError exception:

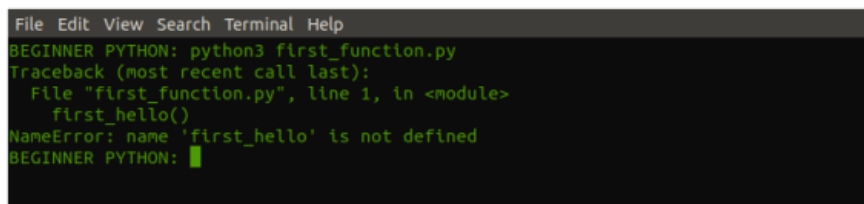
```
first_hello()
```

```
def first_hello():
```

```
    print("hello, world\n")
```

```
    print("Brian Kernighan actually wrote the first \"hello, world\" program as part of the documentation for the BCPL programming language developed by Martin Richards. BCPL was used while C was being developed at Bell Labs a few years before the publication of Kernighan and Ritchie's C book in 1972.")
```

The above code results in:

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The prompt 'BEGINNER PYTHON:' is followed by the command 'python3 first\_function.py'. The output shows a 'Traceback (most recent call last):' followed by 'File "first\_function.py", line 1, in <module>' and 'first\_hello()'. A 'NameError: name 'first\_hello' is not defined' exception is raised. The prompt 'BEGINNER PYTHON:' is followed by a green cursor.

Calling a function interrupts the current script while the function runs, then returns control to the code after the function call:

```
def first_hello():
```

```
    print("hello, world\n")
```

```
    print("Brian Kernighan actually wrote the first \"hello, world\" program as part of the documentation for the BCPL programming language developed by Martin Richards. BCPL was used while C was being developed at Bell Labs a few years before the publication of Kernighan and Ritchie's C book in 1972.")
```

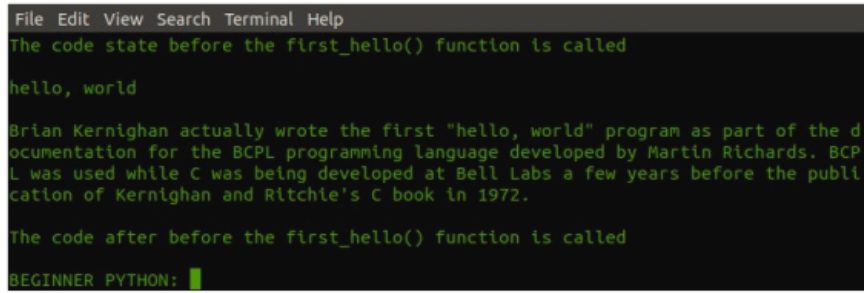
```
    print("The code state before the first_hello() function is called\n")
```

```
first_hello()
```

```
    print("\n\nThe code after before the first_hello() function is called\n")
```



The output from above code looks like this:



```
File Edit View Search Terminal Help
The code state before the first_hello() function is called

hello, world

Brian Kernighan actually wrote the first "hello, world" program as part of the documentation for the BCPL programming language developed by Martin Richards. BCPL was used while C was being developed at Bell Labs a few years before the publication of Kernighan and Ritchie's C book in 1972.

The code after before the first_hello() function is called

BEGINNER PYTHON: █
```

You can see how the flow of the script works. First the print line is reached and “The code state before the *firsthello()* function is called” outputs to the terminal. Then the function call is made and control is passed to the function’s code body. This outputs “hello, world” and the historical note about the phrase. Finally, the function completes its code and returns control back to the script, and “The code after before the *firsthello()* function is called” is printed out.

A python program could have hundreds or even thousands of functions. It is standard practice to include a function called `main()` in python, and have it call all other functions. The script simply calls `main()` and control is then passed off to this function:

```
import datetime

def first_hello(thename):

    print("\nhello, world")

    print("and hello", thename, ", today is", datetime.datetime.now(), "\n")
```

---

```
def main():

    print("calling main() function")

    first_hello("Henry")

    print("main() function complete")

main()
```

Here we are including a module, `datetime`, and using it to get the current date and time. A variable is passed from the *firsthello()* call, *thename*, and received as an argument in the function called *firsthello(thename)*. A function called `main()` has been created to call the `first_hello()` function, and the only expression in the script outside of functions is the call to `main()`, which gets the script started.

Variable Scope: local and global variables:

Variable scope defines the part of a program where a variable can be accessed.

Variables created inside functions are local. This means they can only be used from the time they are created (after the function begins) until the moment the function ends. These variables are removed by the garbage collector after the function in which they reside is finished. Trying to access a variable beyond its scope you will cause an error.

The other kind of variable scope is global. As the terms suggests, global variables can be access at any time anywhere in a program. A global variable is available to the entire program the moment they are defined, and will remain available until the end of the program.

Here is an example:

```
global_variable = 10

def variable_scope():

    local_variable = 99

    print("inside variablescope(), localvariable = ", local_variable)
```

```
print("inside variablescope(), globalvariable = ", global_variable)

def main():

    variable_scope()

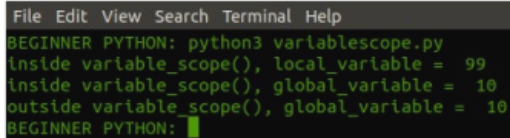
print("outside variablescope(), globalvariable = ", global_variable)
```

**print("inside variablescope(), localvariable = ", local\_variable)**

---

```
main()
```

Here is the output:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 variablescope.py
inside variable_scope(), local_variable = 99
inside variable_scope(), global_variable = 10
outside variable_scope(), global_variable = 10
BEGINNER PYTHON:
```

Notice the commented line in the code above. Let's see what happens if we remove the comment:

```
global_variable = 10

def variable_scope():

    local_variable = 99

    print("inside variablescope(), localvariable = ", local_variable)

    print("inside variablescope(), globalvariable = ", global_variable)

def main():

    variable_scope()

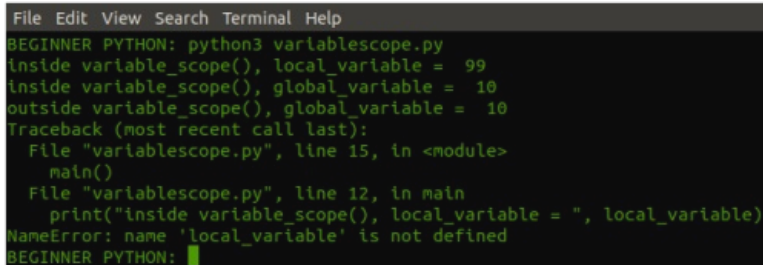
    print("outside variablescope(), globalvariable = ", global_variable)

    print("inside variablescope(), localvariable = ", local_variable)
```

---

```
main()
```

And the output:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 variablescope.py
inside variable_scope(), local_variable = 99
inside variable_scope(), global_variable = 10
outside variable_scope(), global_variable = 10
Traceback (most recent call last):
  File "variablescope.py", line 15, in <module>
    main()
  File "variablescope.py", line 12, in main
    print("inside variable_scope(), local_variable = ", local_variable)
NameError: name 'local_variable' is not defined
BEGINNER PYTHON:
```

Because the variable *localvariable* is *defined inside the actual function variablescope()*, it is only available inside that function. Thus, when it is referenced in *main()*, a *NameError* is thrown.

Variable scope allows us to use the same name for a variable in a local and global context. If there is a name conflict inside a function, the local value will be used. Because of variable scope, it is possible to reuse variable names in more than one function:

```
my_var = 10
```

```

def variable_scope1():

my_var = 99

print("inside variablescope1(), myvar = ", my_var)

def variable_scope2():

my_var = "apple"

print("inside variablescope2(), myvar = ", my_var)

def main():

variable_scope1()

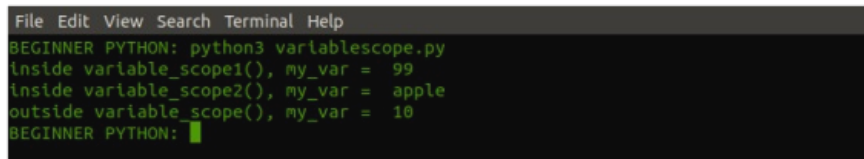
variable_scope2()

print("outside variablescope(), myvar = ", my_var)

main()

```

And here is the output:



```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 variablescope.py
inside variable_scope1(), my_var = 99
inside variable_scope2(), my_var = apple
outside variable_scope(), my_var = 99
BEGINNER PYTHON:

```

The same variable name, *myvar*, is used in two different functions and globally. In each function *myvar* is redefined differently and printed out. As you can see, the local variable overrides the global value inside each function, but when control is returned to the *main()* function the global *my\_var* still retains its original value.

Pass by Value and Mutability:

Passing a variable to a function is called a pass by value. In fact we are not passing a value at all, but a pointer to the memory location of the variable. This matters because this means if an argument passed to a function will result in different effects if the argument is mutable or immutable. Consider an integer, which we know is immutable:

```

def modify_val(arg1):

print("id of arg1:", id(arg1))

arg1 += 1

print("value of arg1", arg1)

print("id of arg1", id(arg1))

def main():

my_val = 1

print("value of myval", myval)

print("id of myval", id(myval))

modifyval(myval)

print("value of myval", myval)

print("id of myval", id(myval))

main()

```

And the output:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 passbyvalue.py
value of my_val 1
id of my_val 10910400
id of arg1: 10910400
value of arg1 2
id of arg1 10910432
value of my_val 1
id of my_val 10910400
BEGINNER PYTHON: █
```

As long as the value of `arg1` is not modified, it remains the same id, meaning it is the same object. However, as soon as the function `modifyval()` changes the value of `arg1`, it becomes a new object, hence it has a new id. Yet, when we return to the `main()` function after the `modifyval()` function completes, the original value and id of `my_val` remains the same.

So what happens when a mutable variable is passed to a function?

```
def modify_val(arg1):

    print("id of arg1:", id(arg1))

    arg1.append(99)

    print("value of arg1", arg1)

    print("id of arg1", id(arg1))

def main():

    my_val = [1, 2, 3, 4]

    print("value of myval", myval)

    print("id of myval", id(myval))

    modifyval(myval)

    print("value of myval", myval)

    print("id of myval", id(myval))

main()
```

We know lists are mutable, and here is the output:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 passbyvalue2.py
value of my_val [1, 2, 3, 4]
id of my_val 140374746231752
id of arg1: 140374746231752
value of arg1 [1, 2, 3, 4, 99]
id of arg1 140374746231752
value of my_val [1, 2, 3, 4, 99]
id of my_val 140374746231752
BEGINNER PYTHON: █
```

The id value of the variable remains the same after it is modified, and even in the `main()` function after the `modifyvar()` function completes, `myval` now retains its new value (99 has been appended to the list), and its id remains the same.

Positional and Keyword Arguments:

We can pass multiple arguments to a function:

```
some_function(arg1, arg2, arg3)
```

And at the time we define our function we must provide it the same number of arguments:

```
def some_function(arg1, arg2, arg3):
```

Not that it is the position of the arguments, not their names, that is important. If we run this code:

```
def my_func(arg3, arg2, arg1):
```

```
print(arg3, " ", arg2, " ", arg1)
```

```
def main():
```

```
arg3 = 3
```

```
arg2 = 2
```

```
arg1 = 1
```

```
my_func(arg1, arg2, arg3)
```

```
main()
```

The output is:

```
1, 2, 3
```

This is why it is called positional arguments. The first argument in a function definition receives the first argument passed to it, regardless of its name.

In contrast to positional arguments, keyword arguments use key-value pairs to pass an argument by name:

```
def my_func(height, width, length):
```

```
print("height =", height, " width =", width, " length =", length)
```

```
def main():
```

```
my_func(length=22, height=9, width=37)
```

```
main()
```

The output is:

```
height = 9 width = 37 length = 22
```

You can also mix positional and keyword passing:

```
def my_func(height, width, length):
```

```
print("height =", height, " width =", width, " length =", length)
```

```
def main():
```

```
my_func(9, 37, length=22)
```

```
main()
```

Produces the same output:

```
height = 9 width = 37 length = 22
```

You can even mix up the keywords. Using:

```
my_func(9, length=22, width=37)
```

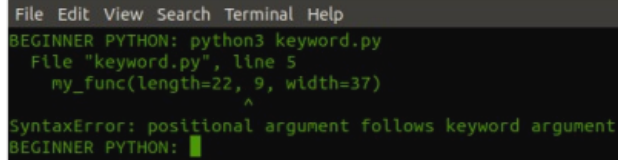
Again produces the same result:

```
height = 9 width = 37 length = 22
```

But one thing you cannot do is this:

```
my_func(length=22, 9, width=37)
```

This will result in an error:

A screenshot of a terminal window with a dark background. The window has a menu bar at the top with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the output of running a Python script: 'BEGINNER PYTHON: python3 keyword.py', followed by 'File "keyword.py", line 5' and 'my\_func(length=22, 9, width=37)'. A caret '^' is placed under the space between '9' and 'width=37'. Below this, a red error message is displayed: 'SyntaxError: positional argument follows keyword argument'. The prompt 'BEGINNER PYTHON: ' is visible at the bottom.

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 keyword.py
File "keyword.py", line 5
  my_func(length=22, 9, width=37)
                        ^
SyntaxError: positional argument follows keyword argument
BEGINNER PYTHON: █
```

Positional arguments must always appear first, followed by keyword arguments, in any order.

Function Return Values:

So far we have created functions that don't return values. These are known as void functions. The syntax to return a value from a function is:

```
return [expression]
```

The expression is option (this is what the [] brackets means). If there is no expression included, the value None is returned.

A function that returns a function is called a value-returning function. The return statement can be used anywhere in a function. When a return statement is encountered inside the running function, the running function terminates and the expression is returned to the line that called it.

We can deal with return values in 3 ways. First, we can generate a new variable and assign the returned value to it.

```
def do_sum(arg1, arg2):
```

```
    return arg1 + arg2
```

```
def main():
```

## assign return to va

```
var = do_sum(10, 100)
```

```
print(var)
```

## treat function call as variable

```
print(do_sum(1,2))
```

## ignore return value

```
do_sum(1000, 99)
```

```
main()
```

As you can see in the comments above, the return value of a function can be assigned to a variable, we can use the return as a variable, or we can simply ignore the return value.

A function can return any data type.

Using return with no value returns the special value None. In fact, if you have a function that does not use the return statement, it still returns None. This is called a void function. Void functions are generally called without being applied to a variable:

```
calling_function(arg1, arg2)
```

Multiple Returns:

To return more than one value, simply separate them with commas:

```
return val1, val2, val3
```

And call the function with as many variables needed:

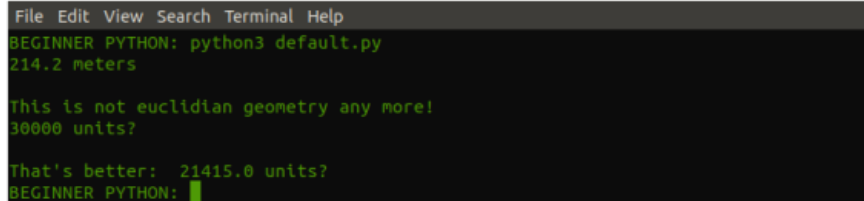
```
var1, var2, var3 = my_function(args)
```

Default argument values:

Sometimes it is useful to assign default argument values to a function:

```
def calccirccleradius(radius=10, pi=2.1415):  
  
    area = pi * radius**2  
  
    return area  
  
def main():  
  
    circarea = calccircle_radius()  
  
    print(format(circ_area, "0.1f"), "meters")  
  
    circarea = calccircle_radius(100, 3)  
  
    print("\nThis is not euclidian geometry any more!")  
  
    print(circ_area, "units?")  
  
    circarea = calccircle_radius(100)  
  
    print("\nThat's better: ", format(circ_area, "0.1f"), "units?")  
  
    main()
```

The above produces the following output:



```
File Edit View Search Terminal Help  
BEGINNER PYTHON: python3 default.py  
214.2 Meters  
  
This is not euclidian geometry any more!  
30000 units?  
  
That's better: 21415.0 units?  
BEGINNER PYTHON: █
```

The first time the function above is called, it is not passed any arguments, so it uses a radius of 10 and a close approximation of pi to generate the area of a circle.

In the second call a measure of 100 units is passed, and pi is defined as 6, meaning this calculation is basically meaningless in terms of geometry.

Finally, the third call supplies only the first argument, for units, which is then calculated and returned.

But wait, did you catch the error? Look again at the `calccirccleradius()` function. It has defined pi as 2.1415, which is incorrect. It should be 3.1415.

In fact, python's math module has a built in pi value, which is accurate to much more than 4 decimal places. The `calccirccleradius()` function is poorly written, because by not taking advantage of built in modules it has made a calculation error. Also, pi is not a variable. It cannot change. A proper rewrite of this function might look like this:

```
import math  
  
def calccirccleradius(radius=10):  
  
    area = math.pi * radius**2  
  
    return area  
  
def main():  
  
    circarea = calccircle_radius()
```

```
print(format(circ_area, "0.1f"), "meters")
```

```
circarea = calccircle_radius(100, 6)
```

```
print("\nThis is not euclidian geometry any more!")
```

```
print(circ_area, "units?")
```

```
circarea = calccircle_radius(100)
```

```
print("\nThat's better: ", format(circ_area, "0.1f"), "meters")
```

```
main()
```

What to take away from this chapter:

A function is a collection of one or several statements that are meant to perform a task.

Variables created inside a function are only available inside that function. These are local variables.

Variable created outside functions are available to all functions. These are global variables.

Mutability determines if an argument passed to a function will be changed in the function only (immutable), or in the originating function as well (mutable).

All functions return a value. If none returned deliberately then None is returned.

#### Chapter 13: Modules

Python uses modules to help organize large programs. We have used built in modules already, like math and datetime. If you write a program that is several pages long, or if you have code you would like to create reusable code for other python programs, you should use modules.

A module is standard text file with a .py extension. Inside it we can define classes, functions, variables, and constants. To use the code inside a module, we simply import it into our current script:

```
import nameofmodule
```

Importing a module means the current script parses and executes the module's code, making it available to the importing script. To use the features of a module, simply use the dot notation. For example, to call a function called *myfunction* in the a module called *nameof\_module*

```
nameofmodule.my_function()
```

If the script cannot find the module to import, the interpreter will generate an ImportError.

Create a new file and call it my\_module.py:

```
CONSTANT_VAL = 99
```

```
var = "variable"
```

```
def my_output():
```

```
    print("This is the my_output() function")
```

```
def my_arg(arg):
```

```
    print("my_arg() called with an argument:", arg)
```

Now create another text file named test\_import.py. Make it in the same directory:

```
import my_module
```

```
mymodule.myoutput()
```

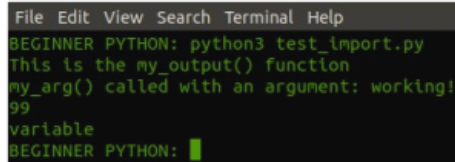


```
mymodule.myarg("working!")

print(mymodule.CONSTANTVAL)

print(my_module.var)
```

When we run test\_import.py, we get this output:

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The output text is: 'BEGINNER PYTHON: python3 test\_import.py', 'This is the my\_output() function', 'my\_arg() called with an argument: working!', '99', 'variable', and 'BEGINNER PYTHON: ' followed by a cursor.

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 test_import.py
This is the my_output() function
my_arg() called with an argument: working!
99
variable
BEGINNER PYTHON: █
```

You must use unique names for classes, functions and variables in your modules. If you do not, python will use the last item and ignore any previous ones:

```
CONSTANT_VAL = 99

var = "variable"

def my_output():

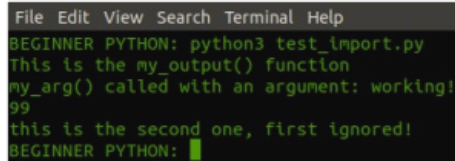
print("This is the my_output() function")

def my_arg(arg):

print("my_arg() called with an argument:", arg)

var = "this is the second one, first ignored!"
```

This is identical to our original module, except there is another var variable defined differently at the bottom. When we run our test again:

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The output text is: 'BEGINNER PYTHON: python3 test\_import.py', 'This is the my\_output() function', 'my\_arg() called with an argument: working!', '99', 'this is the second one, first ignored!', and 'BEGINNER PYTHON: ' followed by a cursor.

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 test_import.py
This is the my_output() function
my_arg() called with an argument: working!
99
this is the second one, first ignored!
BEGINNER PYTHON: █
```

Selecting Identifiers to Import:

Perhaps you don't want to load every identifier from a module. Perhaps you only need two. You can accomplish this with a different form of input statement:

```
from module-name import identifier1, identifier2, ... identifierN
```

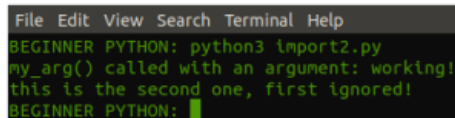
If you use this form of import, then you no longer need to prefix the module name. Here is an example:

```
from mymodule import myarg, var

my_arg("working!")

print(var)
```

Which would output:

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The output text is: 'BEGINNER PYTHON: python3 import2.py', 'my\_arg() called with an argument: working!', 'this is the second one, first ignored!', and 'BEGINNER PYTHON: ' followed by a cursor.

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 import2.py
my_arg() called with an argument: working!
this is the second one, first ignored!
BEGINNER PYTHON: █
```

If you want to have access to every identifier but want to avoid using the dot notation to call them, just import everything using this other syntax:

```
from module-name import *
```

This makes everything available to use in without having to preface it with the module name and a period.

What to take away from this chapter:

Modules allow for code reuse.

Specific identifies can be imported from modules, if you don't to import all of them.

## Chapter 14: Objects and Classes

Ever heard the joke that starts: "There are two types of people..."? Well, the same thing applies to programmers, and it's no joke. Two different schools of thought in programming are:

procedural programming

object oriented programming

What's the difference?

Procedural Programming:

Procedural programming employs a series of steps to tell a computer what to do. You can think of a procedure as a function, and imagine a series of functions calling each other to accomplish the program's purpose. The central goal in procedural programming to do develop reusable code. And while there is nothing wrong with the procedural approach, it tends to get unwieldy as programs get large. What do you do, for example, when you need a function to be slightly different than one you have already written?

Object Oriented Programming:

Object Oriented Programming (often shortened to oop) is a much more structured approach to software development. In oop a set of objects are used to manipulate data. Objects contain variables and procedures to operate on that data. These are known as attributes and methods. To create an object, we have to first define a class. A class is an empty template that defines attributes and methods. By defining a class a new data type is actually created. But keep in mind, a class is just a template, a blueprint, that does nothing on its own. To use a class we have to create objects based on that class, known as a class instance, or just instance. When we create an object like this we call it instantiating a class. We can create as many objects as needed.

Try not to get caught up in the debate. The answer to the question: "What is better, procedural or oop?" is "both." If you are building a short script to achieve a well-defined task and don't expect to build it out in the future, then a procedural approach will probably be quicker getting code down and working. You can avoid all the complex preparation and thought that should go in to an oop project. On the other hand, a large, complex project that could require changes in the future is a better candidate for an oop approach.

In other words, use what works as long as it meets the KISS criteria: keep it simple, stupid.

Defining Class:

The class creation syntax is:

```
class class-name(parent-class)
```

The class definition can be divided into two pieces, a header section and a body section.

Class headers start with the "class" keyword, then the class name, optionally followed by a parent class in parenthesis.

Using a parent class allows you to inherit from it. This is inheritance, which we will discuss later. If no parent class is defined, the parent is set to object.

After the class header comes the class body, which is where the class methods are defined. Remember in python about indentation: method definitions have to be indented the same or you will receive a syntax error.

Here is a class called LightBulb:

```
class LightBulb:
```

```

def init(self, brightness):

self.brightness = brightness

def turn_on(self):

print("Light turned on and brightness = ", self.brightness)

return self.brightness

def turn_off(self):

self.brightness = 0

print("Light turned off and brightness = ", self.brightness)

return self.brightness

def dim(self, brightness):

if(self.brightness == 0):

print("Light is currently off, so turn on:")

self.turn_on()

self.brightness = brightness

print("dimmer set to", self.brightness)

return self.brightness

```

We defined the class name in the class head, where the line begins with “class” and ends with a colon.

Next we define the **init()** method, which is a special method known as the constructor. It is optional. If you do not include one in your class, python will insert an **init()** method that does nothing. The constructor is used to set initial values when an object is created. In our case, **init()** expects two parameters, self and brightness. In python self is required for every method. When you call a method, you do not have to pass any value to self. The interpreter binds the self parameter to the object automatically when a method is called.

Object attributes are called instance variables. The methods that operate on these instance variables are instance methods. The LightBulb class has three instance methods: *turnon()*, *turnoff()*, and *dim()*.

This class describes a light bulb with a dimmer switch. The constructor takes an initial value for brightness, from 0 (off) to 100 (full brightness). Here self.brightness is initialized to the value of the brightness variable passed in during construction.

```
self.brightness = brightness
```

The value self.brightness is an instance variable available to the entire class, while the variable brightness on the right side is a local variable available only to the **init()** constructor method.

Our three methods turn the light on, off, or set its brightness. The first two methods take no arguments (other than the required self), because the state of turning a light bulb on or off is known. In our case, fully off is 0 and on is at the current brightness level.

The dim() class receives an additional argument, brightness, which is an integer between 0 and 100 to dim the bulb to that setting. It also has a check. If the brightness has been set to 0, then the bulb is off. This method first invoices the turn\_on() method to turn the light on, then sets the brightness to whatever value has been passed as an argument to the method.

Creating Objects:

This is simple. You create a class by calling it just like you would a function:

```
ClassName()
```

If you have defined a constructor, you must call the class name with any arguments:

```
ClassName(arguments)
```

In our case, we might instantiate our class like this:

```
LightBulb(50)
```

Accessing Attributes and Methods:

Once we have created an object of a class, this object can be used to access the attributes and methods defined in the class:

```
object.attribute
```

```
object.method(args)
```

In our LightBulb example, we can interact with it like this:

Open your text editor and in a new file called bulb.py put in the following code:

```
from light_bulb import *
```

```
bulb = LightBulb(50)
```

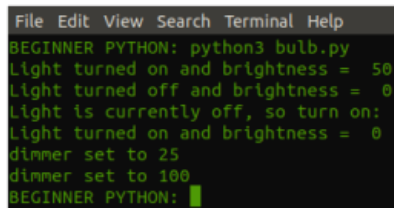
```
bulb.turn_on()
```

```
bulb.turn_off()
```

```
bulb.dim(25)
```

```
bulb.dim(100)
```

When we run the file, we get this output:



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 bulb.py
Light turned on and brightness = 50
Light turned off and brightness = 0
Light is currently off, so turn on:
Light turned on and brightness = 0
dimmer set to 25
dimmer set to 100
BEGINNER PYTHON: █
```

Our script has created a LightBulb object, originally set the brightness to 50, then turned the light on, turned it off, set the dimmer to 25 (which turns the light on if it is not on, before setting the dimmer), and finally set the dimmer to 100.

You can see the power of the oop model. The bulb.py script can create as many LightBulb objects as required, and through the dot notation define the state and brightness of each bulb independantly:

```
from light_bulb import *
```

```
bulb = LightBulb(50)
```

```
print("Turn on bulb")
```

```
bulb.turn_on()
```

```
print("\nTurn on bulb2")
```

```
bulb2 = LightBulb(100)
```

```
bulb2.turn_on()
```

```
print("\nTurn on bulb3")
```

```
bulb3 = LightBulb(0)
```

```
bulb3.turn_on()
```

And the output of this code:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 bulb.py
Turn on bulb
Light turned on and brightness = 50

Turn on bulb2
Light turned on and brightness = 100

Turn on bulb3
Light turned on and brightness = 0
BEGINNER PYTHON: █
```

Take a look at another class, PiggyBank:

```
class PiggyBank:

    def init(self, the_balance):

        self.the_balance = thebalance

    def make_deposit(self, amount):

        self.the_balance += amount

    def make_withdrawal(self, amount):

        if self.the_balance < amount:

            print("Error: Not enough change")

        else:

            print("Successfully withdrawn $", format(amount, "0.2f"), sep="")

            self.the_balance -= amount

    def get_balance(self):

        return self.the_balance
```

Here is our code to access the class:

```
from piggybank import *

my_piggy = PiggyBank(10)

print("Current Balance: $", format(mypiggy.getbalance(), "0.2f"), sep="")

print("Withdrawing $11.50 ...")

mypiggy.makewithdrawal(11.50)

print("Lets try withdrawing $2.50 ...")

mypiggy.makewithdrawal(2.5)

print("Now Current Balance: $", format(mypiggy.getbalance(), "0.2f"), sep="")

print("Depositing $9.95 ...")

mypiggy.makedeposit(9.95)

print("Now Current Balance: $", format(mypiggy.getbalance(), "0.2f"), sep="")
```

And the output is:

```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 getpiggy.py
Current Balance: $10.00
Withdrawing $11.50 ...
Error: Not enough funds
Lets try withdrawing $2.50 ...
Successfully withdrawn $2.50
Now Current Balance: $7.50
Depositing $9.95 ...
Now Current Balance: $17.45
BEGINNER PYTHON: █
```

The workings of PiggyBank should be self-explanatory.

What to take away from this chapter:

Procedural program is fine, but tends to get unwieldy as it gets large

Oop programming allows for code reuse and almost limitless program size.

Classes are templates that define attributes and methods.

Once an object is created from a class, its attributes and methods can be accessed to achieve the program goals.

## Chapter 15: Inheritance and Polymorphism

Inheritance allows the creation of new classes based on an existing class. This is commonly known as creating a child class from a parent class. The child class inherits all the parent attributes and parent class methods of the parent class.

The real value in inheritance is that it allows you to expand on an existing class without having to stuff it full of methods it doesn't need, while at the same time you can still you the attributes and methods of the parent class without duplicating its code in the client class.

Let's use our LightBulb class as a simple example. As it is, the class works well to turn lights on and off, and to set the brightness level. But what if you wanted to change the color of the light? This is not something you will do often, so it doesn't really belong in the main LightBulb class. At the same time, creating a new class with all the same code as LightBulb just to add the color option seems like a waster of time. Inheritance lets us have the best of both worlds

Here is the syntax for class inheritance:

```
class Parent():
```

```
class Child(Parent):
```

When the Child class inherits from the Parent class as we have above, we say the Child class extends the Parent.

Let's revisit our LightBulb class:

```
class LightBulb:
```

```
def init(self, brightness=100):
```

```
self.__brightness = brightness
```

```
def turn_on(self):
```

```
print("Light turned on and brightness = ", self.__brightness)
```

```
return self.__brightness
```

```
def turn_off(self):
```

```
self.__brightness = 0
```

```
print("Light turned off and brightness = ", self.__brightness)
```

```
return self.__brightness
```

```
def set_brightness(self, brightness):
```

```

if(self.__brightness == 0):

print("Light is currently off, so turn on:")

self.turn_on()

self.__brightness = brightness

print("dimmer set to", self.__brightness)

return self.__brightness

class ColorBulb(LightBulb):

def init(self, color='white'):

super().init()

self.__color = color

def get_color(self):

return self.__color

def set_color(self, color):

self.__color = color

```

Pay attention to this line:

```
super().init()
```

This is the python syntax for a child class to call the parent class's methods and access its attributes. The `super()` function calls the parent's `init()` method.

We can now access our classes:

```

from light_bulb import *

bulb = ColorBulb()

print("Turn on bulb")

bulb.turn_on()

print("What's the current color?", bulb.get_color())

print("Set the color to red:")

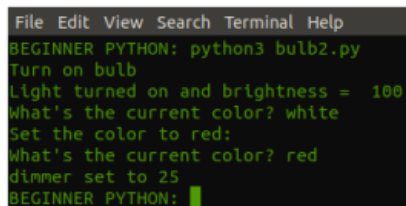
bulb.set_color("red")

print("What's the current color?", bulb.get_color())

bulb.set_brightness(25)

```

And get this output:



```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 bulb2.py
Turn on bulb
Light turned on and brightness = 100
What's the current color? white
Set the color to red:
What's the current color? red
dimmer set to 25
BEGINNER PYTHON: █

```

In python a class can also inherit from multiple parents. The syntax is:

Class Child(Parent1, Parent2, Parent3)

Polymorphism and Method Overriding:

A complex sounding name for a fairly simple idea. If you have a child class inherit from a parent, that parent has a method called *classx*, *but you want your child to have a method called classx*, but for it to be different in function, you can. This child class will “override” the parent class. So any object instance of the child class will have access to the child’s version of *classx*, *while instances of the parent class will maintain the original classx* method.

What to take away from this chapter:

Classes can inherit (child) from other classes (parent).

inheritance makes all parent class attributes and all parent class methods available to the child class.

A child class can override a parent method by defining it differently. This is known as polymorphism.

## Chapter 16: Operator Overloading

Operator overloading allows us to change the definition of the meaning of an operator inside your class. Operator overloading is what allows us to use the same operator ( + ) to add integer values numerically while at the same time concatenate strings.

Operator overloading is done by defining a special method in your class definition. These methods start and end with two underscores ( \_\_ ). The method used in overloading the + operator is **add()**. The int and str classes each implement the **add()** method. Where the int class simply uses the **add()** method to add two numbers numerically, the string **add()** method concatenates two strings.

We learned earlier that variable names in classes that begin with two underscores are private and therefor inaccessible outside the class. This is not the case with variables that have two underscores at the end as well as two at the beginning. So, **and** and other operators are not private.

To python, i + x is interpreted as i.add(x) – which type of method is called depends on what type i and x are:

```
a, q = 1, 10
>
a+q
```

11

```
>
a.add(q)
```

11

```
a, q = "this", "orthat"
>
a+q
```

'thisorthat'

```
>
a.add(q)
```

'thisorthat'

```
>
```

Here is an example of operator overloading within a class, overloading.py:

```
import math
```

```
class MyCircle:
```



```

def init(self, the_radius):

self._theradius = the_radius

def settheRadius(self, the_radius):

self._theradius = radius

def gettheRadius(self):

return self._theradius

def area(self):

return math.pi * self._theradius ** 2

def add(self, circle2):

return MyCircle( self.theradius + circle2.theradius )

circ1 = MyCircle(9)

print(circ1.gettheRadius())

circ2 = MyCircle(57)

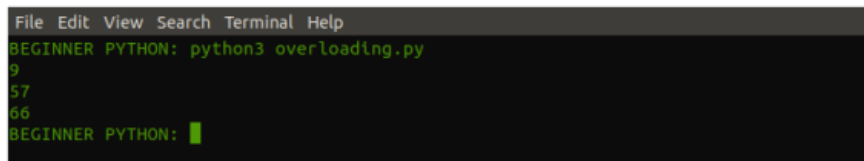
print(circ2.gettheRadius())

circ3 = circ1 + circ2 # Only possible because of the add overriding above

print(circ3.gettheRadius())

```

And the output:



```

File Edit View Search Terminal Help
BEGINNER PYTHON: python3 overloading.py
9
57
66
BEGINNER PYTHON: █

```

What to take away from this chapter:

Operator overloading allows us to custom define operators inside classes

although the special methods that define operators begin with a double underscore, they are not private variables because they also end with two underscores.

## Chapter 17: File Handling

Being able to handle files is a useful tool for any software developer. File manipulation in python occurs in 3 steps:

open a file

act on the file(read, write, etc)

close the file

There are two types of files: binary and text. Text files are like those we've been working on in this book. Text files are essentially a sequence of ASCII characters. Binary files, on the other hand, store information in a binary format. Common forms of binary file are mp3 and images files.

In this book we will only be looking at the manipulation of text files.

Opening a File:

```
thefile = open(filename , mode)
```

"filename" is the path and name of the file. The mode is a switch that identifies what kind of operation you want to perform on the file: r = open for

reading (if there is no such file you will get an error); w = opens for writing (the file will be created if it does not exist; if the file does exist its contents will be lost); a = append mode (the file will be created if it does not exist; if the file already exists data will be added to the file's end, rather than destroying the original file contents)

You can also add t or b to set the mode. The t stands for "text file" while the b stands for a "binary file". The default mode is text, so it is not usually added.

```
f = open("dog_breeds.txt", "a")
```

When finished, you close the file:

```
f.close()
```

Although python automatically closes when the program ends, it is good practice to close files when you're done using them. Opening files can consume significant resources, which could have a negative effect on your programs.

The open() function produces a file object. It's type is `__io.TextIOWrapper`. This class contains the methods and attributes you will use to read and manipulate the contents of files. As you see above, close() is one of those methods. Here is a list of them all:

read([num]) – reads [num] characters; returns them as string; if omitted, read() returns the whole file.

readline() – reads one line and returns a string.

readlines() – reads every line; returns a list.

write(str) – writes str to the file; returns the # of characters written.

seek(offset, origin) – moves the file pointer offset from origin.

tell() – returns the current file pointer position.

close() – closes file.

Writing to a Text File:

```
f = open("myfile.txt", "w")
```

```
f.write("This is line one\n")
```

```
f.write("This is line two\n")
```

```
f.write("This is line three\n")
```

```
f.close()
```

You will see this in your terminal:

```
f = open("myfile.txt", "w")
```

```
    f.write("This is line one\n")
```

```
23
```

```
    f.write("This is line two\n")
```

```
24
```

```
    f.write("This is line three\n")
```

```
23
```

```
    f.close()
```

Open the file with your text editor. You should see this inside myfile.txt:

This is line one

This is line two

This is line three

## Reading Data from a Text File

Here we will be using the `r` mode. Make sure your file actually exists, or you code will throw a `FileNotFoundError`. You can use the function `isfile()` to determine if a file actually exists. `isfile()` is available in the `os` module:

```
import os
```

```
os.path.isfile("path/to/file/myfile.txt")
```

```
True
```

Best practice is to check if a file exists before trying to open it for reading.

```
f = open("myfile.txt", "r")  
print(f.read())
```

This is line one

This is line two

This is line three

```
f.close()  
>
```

We did not provide an index to the `read()` module, so the whole file is read into memory. We then printed the file contents and closed the file.

To get a better understanding of how read functions, take a look at this output:

```
f = open("myfile.txt", "r")  
some_chars = f.read(15)  
for x in some_chars:
```

```
... print(x)
```

```
...
```

```
T
```

```
h
```

```
i
```

```
s
```

```
i
```

```
s
```

```
l
```

```
i
```

```
n
```

```
e
```

```
o
```

```
n
```

```
- - -
```

```
f.close()
```

And here is an example of `readlines()`:

```
f = open("myfile.txt", "r")  
  
all = f.readlines()  
  
f.close()  
  
print(all)
```

```
['This is line one\n', 'This is line two\n', 'This is line three\n']
```

```
>
```

What to take away from this chapter:

There are two kinds of files, binary and text.

The process of accessing a file in python involves opening the file, reading or manipulating the file, and closing the file.

Leaving large files open in long-running programs can cause heavy system resource use.

## Chapter 18: Exception Handling

We use exception handling to deal gracefully with errors while a program is running. Being preemptive in this way can allow our programs to recover from errors that would ordinarily stop them from continuing. The alternative is to have the program execution end abruptly and unpredictably. This rarely produces a good user experience.

Runtime Errors:

These are errors that fire when program's are in a running state. These are not syntax errors. The interpreter understands your code, but is just incapable of running it. We've encountered these already, like when we tried to add an integer and a string. Other runtime errors are division by zero, accessing an index that doesn't exist, or opening a file in read mode that does not exist.

These errors are known as exceptions. Python creates an object that contains the exception details, including line number, any error messages, and so on. By default these types of errors will halt program execution. Exception handling is a way to manage these types of errors gracefully, that is, without breaking out of the program.

try-except Statement:

This is the statement python uses for exception handling. Here is the syntax:

try:

## try block

## possible exception code

except ExceptionType:

## except block

## handle exception here

The handler begins with `try` – a reserved keyword – followed by a colon. After the `try` statement is the code that could throw an exception. This is followed by `except` – another reserved keyword – followed by the exception type and a colon. The `except` block has the code to deal with the possible exception. The exception block will only fire if the exception thrown is the same identified at the beginning of the exception block. Other

types of exception will halt program execution as usual.

Here is an example of handling division by zero:

try:

```
num = int(input("Enter a number: "))
```

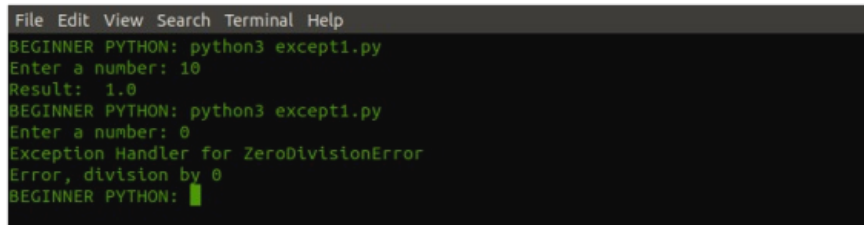
```
result = 10/num
```

```
print("Result: ", result)
```

```
except ZeroDivisionError:
```

```
print("Exception Handler for ZeroDivisionError")
```

```
print("Error division by 0")
```



```
File Edit View Search Terminal Help
BEGINNER PYTHON: python3 except1.py
Enter a number: 10
Result: 1.0
BEGINNER PYTHON: python3 except1.py
Enter a number: 0
Exception Handler for ZeroDivisionError
Error, division by 0
BEGINNER PYTHON: 
```

On the first run, a ten was entered and the program ran successfully. On the second run, a 0 was entered and the exception block was fired. Note that if you run the above code and enter a string rather than a number, it will fail and throw ValueError error. This error does not match our error catching block, so it is not caught by the code.

Multiple Exceptions:

Here is the syntax to catch multiple possible exception errors:

try:

## try block

except :

## except block

## handle ExceptionType1

except :

## except block

## handle ExceptionType2

except :

## except block

## handle ExceptionType3

except:

# handle any type

If there is an exception, python will go through each exception type until it finds a match. If it does not, the final except: line is a catch-all for any unanticipated errors. This is a last-ditch attempt to catch an exception before it halts program execution.

A try-exception statement can also include an else and/or finally clause. The else block will only be executed if there is no exception. The finally clause runs after either case. It is often used for cleanup.

Accessing an Exception Object:

The syntax to access an exception object is:

```
except ExceptionType as e
```

as in:

try:

```
except RuntimeError as e:
```

```
print("Error:", e)
```

What to take away from this chapter:

Exception handling allows a program to catch exceptions and handle them without halting the program's execution.

Runtime errors happen during a running program. They are not syntax errors: the interpreter has reached a statement it is incapable of running.

Runtime errors are known as exceptions.

Exceptions objects can be accessed in order to print out their error messages.

## Chapter 19: Tuples

A tuple functions exactly like a list, with the only difference being a tuple is immutable. You cannot add, remove, or modify the contents of a tuple. This immutability makes them very efficient pragmatically when compared to lists.

You can create a tuple in various ways:

```
my_tup = ("apple", "banana", "dragon fruit")
```

```
>
```

```
my_tup
```

```
('apple', 'banana', 'dragon fruit')
```

```
>
```

```
type(my_tup)
```

```
>
```

```
empty_tuple = ()
```

```
>
```

```
empty_tuple
```

```
()
```

```
>
```

```
type(empty_tuple)
```

```
- -
```

```
| | >
```

You can also use constructor functions:

```
| | | tup2 = tuple("xyz")
| | | >
| | | tup2
```

('x', 'y', 'z')

```
| | | >
| | | | tup3 = tuple(range(5, 15))
| | | >
| | | | tup3
```

(5, 6, 7, 8, 9, 10, 11, 12, 13, 14)

```
| | | >
| | | | tup4 = tuple(["a", "bb", "xxx", "yyyy"])
| | | >
| | | | tup4
```

('a', 'bb', 'xxx', 'yyyy')

```
| | >
```

List comprehension can be used to create tuples as well:

```
| | | >
| | | | tup_comp = tuple([i * 2 for i in range(2, 15)])
| | | >
| | | | tup_comp
```

(4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28)

```
| | >
```

Be careful when making a tuple with only one value. You must include the trailing comma:

```
| | | one_tup = (1,)
| | | >
| | | | one_tup
```

(1,)

```
| | | type(one_tup)
```

```
| | | >
| | | | one_tup = (1)
| | | | one_tup
```

1

```
| | | type(one_tup)
```

-- --

The parenthesis are optional when creating tuples. So this is valid:

```
pnottup = "apple", "banana", "dragon fruit"

>

pnottup
```

('apple', 'banana', 'dragon fruit')

```
>

type(pnottup)

>
```

As with creating tuple with one item in parenthesis, you must include the trailing comma if you create a tuple with a single item using the no parenthesis method.

Tuple Unpacking:

You can create multiple tuples at once:

```
one, two, three = ("apple", "banana", "dragon fruit")

>

one
```

'apple'

```
two
```

'banana'

```
three
```

'dragon fruit'

```
>
```

The number of variables (left side) and elements (right site) must be the same, or the interpreter will throw an error. As with other methods of tuple creation, parenthesis are optional.

```
one, two, three = ("apple", "banana", "dragon fruit", "show")
```

Traceback (most recent call last):

File "", line 1, in

ValueError: too many values to unpack (expected 3)

```
>

one, two, three = "apple", "banana", "dragon fruit"

>

one
```

'apple'

```
two
```

'banana'

```
three
```



```
'dragon fruit'
```

```
>
```

Operations on Tuples:

Since a tuple is essentially an immutable list, most of the operators you can use on lists you can use on tuples. The only exception is methods used to modify lists. So methods like these:

```
insert(),
```

```
append(),
```

```
remove(),
```

```
reverse(),
```

```
sort(),
```

cannot be used on tuples.

What to take away from this chapter:

Tuples are like immutable lists.

Tuples are much more efficient to work with than lists.

Most methods that work on lists will work on tuples, with the exception of methods used to modify lists. These will not work because of the tuple's immutability.

## Chapter 20: Sets

Sets are very much like lists. They are mutable and store a collection of items. But they differ from lists in two important ways:

each element in a set must be unique

elements are stored in no fixed order

Thus, if your program does not care what the order of the items are, use a set. Sets are far more efficient than lists when you manipulate elements.

Creating Sets:

Syntax for creating a set:

```
my_set = {34, 5, 67, 99}
```

Note the curly braces.

```
my_set = {1, b, 3, d}
```

```
>
```

```
type(my_set)
```

```
>
```

```
my_set
```

```
{1, b, 3, d}
```

```
>
```

```
set2 = {"apple", 99, "red balloons", "2.5"}
```

```
>
```

```
set2
```

```
{99, '2.5', 'apple', 'red balloons'}
```

```
>
```

Like lists and tuples, you the `set()` function and list comprehension can be used to generate sets:

```
s1 = set({22, 33, 66, 5})  
s1
```

```
{33, 66, 5, 22}
```

```
>  
s2 = set("789xyz")  
s2
```

```
{'9', 'y', '8', 'x', '7', 'z'}
```

```
>  
s3 = set(['one', 'two', 3, 4])  
s3
```

```
{'one', 3, 4, 'two'}
```

```
>  
s4 = set(("shoe", "sock", "ankle"))  
s4
```

```
{'sock', 'ankle', 'shoe'}
```

```
>  
s5 = set([q*2 for q in range(1, 10)])  
s5
```

```
{2, 4, 6, 8, 10, 12, 14, 16, 18}
```

```
>
```

If you try to make a set with more than once element the same, python will drop the identical values:

```
>  
ident_set = {"blue", "orange", "red", "orange", "orange", "blue"}  
ident_set
```

```
{'blue', 'red', 'orange'}
```

```
>
```

As will lists, there are many built-in functions you can use to examine sets. You can also use the methods that manipulate lists with sets, as well as looping and membership operators (`in` and `not in`).

What to take away from this chapter:

Sets are mutable like lists.

The sequence of a set must consist of unique elements.

Set elements are stored in no particular order.

Sets are much more efficient to use than lists.

## Chapter 21: Dictionary

### Accessing Values from Dictionaries:

This is very similar to lists again, but instead of access a particular element by using an index, we use the element's key:

```
dict_name[key]
```

Dictionaries are mutable, meaning you can add, modify, and remove elements from them.

Here's an example:

```
my_dic = {  
... 'fruit' : 'apple',  
... 'vegetable' : 'carrot',  
... 'meat' : 'steak'  
... }  
  
>  
my_dic  
{'fruit': 'apple', 'vegetable': 'carrot', 'meat': 'steak'}
```

```
>  
my_dic['meat']  
'steak'
```

```
>  
my_dic['fruit']  
'apple'
```

```
>  
my_dic['seed']
```

Traceback (most recent call last):

File "", line 1, in

KeyError: 'seed'

```
>
```

As demonstrated in the final access attempt above, if there is no such key available in the dictionary the interpreter will throw a `KeyError`.

### Adding and Modifying Dictionary Values:

```
my_dic  
{'fruit': 'apple', 'vegetable': 'carrot', 'meat': 'steak'}  
  
>  
my_dic['nut'] = 'peanut'  
>  
.
```

```
my_dic
```

```
{'fruit': 'apple', 'vegetable': 'carrot', 'meat': 'steak', 'nut': 'peanut'}
```

```
>  
my_dic['fruit'] = 'DRAGON FRUIT'  
>  
my_dic
```

```
{'fruit': 'DRAGON FRUIT', 'vegetable': 'carrot', 'meat': 'steak', 'nut': 'peanut'}
```

```
>
```

As you can see above, if the key already exists it will be updated.

To delete a particular element from a dictionary sequence, use the following syntax:

```
del dictionary['key']
```

Here is an example using our dictionary:

```
del my_dic['fruit']  
>  
my_dic
```

```
{'vegetable': 'carrot', 'meat': 'steak', 'nut': 'peanut'}
```

```
>
```

As with lists, a mutable dictionary can have the same built-in methods to view and manipulate. New methods to be aware of are the built-in `key()`, `values()`, and `items()` methods:

```
my_dic.keys()
```

```
dict_keys(['vegetable', 'meat', 'nut'])
```

```
>  
type(my_dic.keys())
```

```
>  
my_dic.values()
```

```
dict_values(['carrot', 'steak', 'peanut'])
```

```
>  
type(my_dic.values())
```

```
>  
my_dic.items()
```

```
dict_items([('vegetable', 'carrot'), ('meat', 'steak'), ('nut', 'peanut')])
```

```
>
```

The `keys()` and `values()` methods return lists of keys or values from the dictionary. These types are like immutable lists that you can use to loop through a dictionary's contents.

The `items()` method returns tuples, where each tuple contains the key and value of a single dictionary element.

What to take away from this chapter:

Dictionaries use key-value pairs to store elements, in no fixed order

dictionaries are mutable, so elements can be added, edited, or removed.

The keys() method returns a dictionary's keys as a list; the values() method returns a dictionary's values as a list.

the items() method returns a dictionary as an iterable sequence of key-value tuples.

## Chapter 22: Black hat hacking tips

Blackhat hacking techniques don't descend from the other variety of hacking. Learning hacking is learning black hat hacking . You don't wish to attack alternative servers from the primary day, right? build Without expertise, you may in all probability make a bunch of mistakes that may end you up in jail. Having aforementioned that I will tell you current ways to start out learning hacking, especially black hat hacking .

For today's article, we'll talk about the 6 hacking tips you may use that'll help you to become a certified black hat hacker.

Are you ready? check out the list of tips below:

### 1. Use Breach to Encrypt HTTPS Streams

A new hacking technique dubbed BREACH will extract login tokens, session ID numbers and alternative sensitive info from SSL/TLS encrypted net traffic in exactly thirty seconds.

The technique was incontestable at the Black Hat security conference in an urban center in Las Vegas (Presentation PDF & Paper) by Gluck and researchers Neal Harris and Angelo Prado, that permits hackers to decode encrypted knowledge that online banks Associate in Nursing e-commerce sites from an HTTPS channel.

BREACH (Browser reconnaissance mission and Exfiltration via Adaptive Compression of Hypertext) is extremely targeted and don't rewrite the complete channel. BREACH manipulates knowledge compression to pry out doses {of information, and of knowledge } from HTTPS-protected data, together with email addresses, security tokens, and other plain text strings.

Angelo Prado told The Hacker News, "We square measure employing a compression oracle is an investment of the building blocks from CRIME, on a different compression context." i.e. To execute the Oracle attack, BREACH exploits the standard Deflate compression algorithm used by many websites to conserve bandwidth.

The assailant simply needs to regularly snoop on the encrypted traffic between a victim and an internet server before and also the exploit needs that a victim 1st access a malicious link, this could be done by embedding an iframe tag in a page the victim visits.

The recovery of secret authentication cookies opens the door for attackers to create as their victims and hijack attested net sessions. It is vital to notice that the attack is agnostic to the version of TLS/SSL, and does not require TLS-layer compression. Additionally, the attack works against any cipher suite.

### 2. Make Use of DDoS Attacking Tools

Aside from the Breach tool, to execute the black hat hacking , you may also use the Dos or Denial-of-service tool. Let's learn what this is...

Denial-of-service (DoS) attacks square measure the precursor to DDoS attacks. Historically, DoS attacks were a primary method for disrupting computer systems on a network. DoS attacks originate from a single machine and can be very simple; a basic ping flood attack can be accomplished by sending more ICMP (ping) requests to a targeted server then it's ready to method and answers expeditiously. Just about anyone with a networked machine is ready to launch this kind of attack by the exploitation of integral terminal commands. More complex DoS attacks may involve using packet fragmentation, such as the now largely defunct "Ping of Death" attack.

A number of various attack tools or "stressors" square measure out, there for are complimentary on the web. At their core, a number of these tools have legitimate functions, as security researchers and network engineers may at times perform stress tests against their own networks. Some attack tools square measures specialized and solely specialize in a selected space of the protocol stack, where as others are going to be designed to permit for multiple attack vectors.

Attack tools are loosely characterized by many groups:

#### Low and slow attack tools

As the name implies, these types of attack tools both use a low volume of data and operate very slowly. Designed to send little amounts of information across multiple connections so as to stay ports on a targeted server open as long as doable, these tools still utilize server resources till

a targeted server is unable to take care of extra connections. Uniquely, low and slow attacks could sometimes be effective even once not employing a distributed system like a botnet and square measure normally utilized by one machine.

#### Application layer (L7) attack tools

These tools target layer 7 of the OSI model, where Internet-based requests such as HTTP occur. Using a variety of protocol flood attack to overwhelm a target with protocol GET and POST requests, a malicious actor will launch attack traffic that's tough to tell apart from normal requests made by actual visitors.

#### Protocol and transport layer (L3/L4) attack tools

Going additional down the protocol stack, these tools utilize protocols like UDP to send large volumes of traffic to a targeted server, such as during a UDP flood. While typically ineffective severally, these attacks are typically found in the form of DDoS attacks where the benefit of additional attacking machines increases the effect.

A few commonly used tools include:

##### Low Orbit Ion Cannon (LOIC)

The LOIC is an open-source stress testing application. It permits for each transmission control protocol and UDP protocol layer attacks to be dole out employing an easy application interface. Due to the recognition of the first tool, derivatives have been created that allow attacks to be launched using a web browser.

##### High Orbit Ion Cannon (HOIC)

This attack tool was created to switch the LOIC by increasing its capabilities and adding customizations. By utilizing the HTTP protocol, the HOIC is able to launch targeted attacks that are difficult to mitigate. The software package is meant to own a minimum of fifty folks operating along in a very coordinated attack effort.

##### Slowloris

Apart from being a slow primate, Slowloris is an application designed to instigate a low and slow attack on a targeted server. The magnificence of Slowloris is that the restricted quantity of resources it has to consume so as to form a dangerous impact.

##### R.U.D.Y (R-U-Dead-Yet)

R.U.D.Y. is another low and slow attack tool designed to permit the user to simply launch attacks employing an easy point-and-click interface. By gap multiple communications protocol POST requests so keeping those connections open as long as potential, the attack aims to slowly overwhelm the targeted server.

#### 1. Make Use of Blue Box Hacker for Phreaking

Now, if you're more on the phreaking side of your black hat hacking , you may consider using the Blue Box method.

A blue box is one in every of the oldest tools utilized and it comes within the type of an electronic sort of equipment. It was named in and of itself thanks to the proof that was initially discovered by the Bell System security, that was really a blue plastic. Blue box functions in such the simplest way that it creates a replica of the sound that's made by the dialing console of a manipulator.

The feature of a blue box includes the imitation of the tones that square measure used in swapping the long-distance calls. In addition, this ancient phreaking tool is additionally used in routing a substituted decision with the decision being created by the user of the blue box. Hence, the standard procedure of decision shift is also circumvented with the employment of this phreaking equipment.

Their square measure varied varieties of tools utilized by the phreaks and every one has its own distinctive purpose. According to the specialists in phreaking, the foremost common use of this blue box is to create free phone calls. On the other hand, the other phreaking tool black box allows an individual to accept free calls that are made by the person on the other end of the line.

As of this times, however, the blue box is not anymore used in most of the countries in the western part of the world. This is thanks to the changed method applied on this shift systems once calls square measure being created. Telecommunication firms shifted from the in-band communication, which may be simulated by the blue box, to the more advanced digital switching systems.

Signaling takes place these days on an out-of-band channel, which is not similar in any way to the one being used by companies during the previous years. Out-of-band channel disagrees since this could ne'er be accessed through the road being used by a selected caller. This is famed to be because of the Common Channel Interoffice communication or CCIS.

## 1. Secure Boot to Remain Undetected

Furthermore, one of our greatest concern in black hat hacking is to remain undetected, right? With that said, secure boot can be a great help!

Secure Boot may be security normal that's a part of UEFI designed to limit what gets loaded throughout the boot time of the device.

Microsoft introduced the feature in Windows eight back in 2011, and every client or server version of Windows supported it since then.

Microsoft expressed in the past that it had been up to the manufacturer of the device to ship it with controls to show Secure Boot off.

Without those controls, it's uphill to use load operative systems that don't seem to be expressly allowed. In the worst case, it would mean that only one particular flavor of Windows can be run on a device.

This is as an example the case on Windows RT or Windows Phone devices. Secure Boot is turned off on PCs and notebooks but, a minimum of for the nonce.

Researchers discovered the simplest way to govern Secure Boot on Windows devices, effectively rendering it useless.

### 1. Make Use of Bochspxn; a Tool Released by Google, Inc.

Bochspxn may be a system-wide instrumentation project designed to log memory accesses performed by OS kernels and examine them in search of patterns indicating the presence of sure bugs, such as "double fetches". Information regarding memory references is obtained by running the guest operative systems among the Bochs IA-32 someone with the custom instrumentation part compiled in. It was written in 2013 and was used to discover over 50 race conditions in the Windows kernel, fixed across numerous security bulletins (MS13-016, MS13-017, MS13-031, MS13-036).

The toolset isn't actively maintained, and its ASCII text file is discharged "as is", mostly for reference purposes. It was originally released as kfetch-toolkit in 2013 after the Black Hat USA talk, together with comprehensive documentation at DOCUMENTATION.old.md (now partially obsolete). In 2017, we have a tendency to revised the ASCII text file of the project and enforced many new features:

Information about the address space layout of kernel drivers is stored in a separate file ( modules. In by default&#41, and each driver is referenced by its index in the main log file. This was done to save disk space, by preventing the redundant information (image names and base addresses) from being needlessly saved for every stack trace item in the log.

Information regarding the presence of a full of the life exception handler in every stack frame was acquisitions to the access log protocol buffer, permitting the United States to observe a variety of native Windows DoS vulnerabilities (see examples 1, 2, 3, 4).

Information about the value of Previous Mode at the time of the memory access in Windows was added to the protocol buffer.

The "online" double-fetch detection mode was far away from the code, as it was deemed too slow to be practically useful.

Some symbolization-related and alternative minor bugs were mounted within the code.

The instrumentation was conjointly ported to Bochs version two.6.9, the latest one at the time of this writing.

### 1. Don't Forget the CreepyDOL Tool

The last important tip for this article is about the use of the tool CreepyDOL. Let's know important ideas about this tool and how it is use primarily for black hat hacking.

Brendan Flannery O'Connor is an unembarrassed hacker who has worked for DARPA and has taught at the. United States military's cybersecurity faculty. CreepyDOL (Creepy Distributed Object Locator) , his new personal tracking system, allows a user to track, locate, and break into an individual's smartphone. "For a number of hundred bucks," he says, "I can track your every movement, activity, and interaction until I find whatever it takes to blackmail you."

Privacy is turning into ever tougher to ensure in today's connected world. It is not clear whether or not it's governments or businesses that square measure additional inquisitive about your innermost secrets, however, each have a fairly smart handle on most people.

CreepyDOL and similar systems currently threaten to form the power to find somebody's personal affairs accessible to anyone with the inclination and a couple of hundred bucks to spare.

CreepyDOL may be a network of sensors that communicate with a data-processing server. The sensor network runs on boxes about the size of a small external hard drive, with each node containing a Raspberry Pi Model A, two Wi-Fi adapters, and a USB hub. Previously developed by a writer, these are was known as F-BOMBs (Falling/Ballistically-launched Object that creates Backdoors) and are sufficiently rugged to be thrown, or perhaps born from a UAV. Each F-BOMB cost just over US\$50, giving a network of 10 a price of around \$500.

## Final Thoughts on Python Black Hat Hacking Tips

There you have it!

In this article I talked about the easy and achievable tips for python black hat hacking. Indeed, Learning hacking is learning how to do black hat hacking.

As a recap, below are the 6 important black hat hacking tips to utilize:

Use Breach to Encrypt HTTPS Streams,

Make Use of DDoS Attacking Tools,

Make Use of Blue Box Hacker for Phreaking,

Secure Boot to Remain Undetected,

Make Use of BochsPwn; a Tool Released by Google, Inc.,

Don't Forget the CreepyDOL Tool,

Conclusion

Thank for making it through to the end of The Ultimate Python Programming Guide for Beginners ; let's hope it was informative and able to provide you with a toolkit you can use to achieve your programming goals, whatever they may be.

You should now have a basic and fundamental understanding of the python programming language, as well as a start in understanding coding best practices and how to achieve software requirements with proper coding techniques. Software development is difficult, and a life-long practice. Feel free to use this book as a reference now, returning to it as often as you need in order to brush up on concepts and increase your understanding of the language.

The internet is a vast resource for many programming languages, and no less so for python. If you get stumped or confused, search for solutions others have made and learn how and why they did what they did. Next to having a personal tutor, other software developers on the internet are your best resource for increasing your skills in python. You will be exposed to not only code with working answers, but oftentimes more than one solution is presented to a problem. Examine these answers and try to understand where the programmer was coming from in writing that solution. Read what others say about a given solution, and try to understand their criticism or praise for a particular piece of code. If you do not understand, join the conversation and ask. Developers are willing to help out beginners, even if they appear exacting and harsh at first. After all, every developer had to start at the beginning, just like you are now.

The next step is to take what you have learned and start building code. Think of a problem you would like to solve and see if you can solve it in python. The best way to write software is not to start writing code right away, but instead you should write down the requirements of your software. Oftentimes you will find that what you thought was a simple problem, or a problem that required a particular solution, is far more complex than it appeared at the beginning. When you know what your software should do, try to break down these requirements into steps: modules first, then classes, and finally methods in those classes. When you are done with this work, you should have a very clear picture of what you are going to write to achieve your requirements. If you find out what you are working on will not do because it is missing some fundamental piece, step back, write down your requirements again, and break these down into steps before getting back to writing code. You will find every hour you spend in preparation will save you many hours of frustration in having to re-write code that does not meet your needs.

Another important next step you should take is to learn how to access an sql database with python. This will require some understand of sql (mysql is the most common sql database on the web), which is outside the purview of this book. But interacting with a database is absolutely essential to build larger python programs, in particular those that save and retrieve large amounts of information.

If this book has helped you please leave a generous review.

Thank you and good luck!

---

THE END

---

KEY BOARD SYBOLS

tilde ~



acute ` quote double quote “

exclamation ! apostrophe or single quote ‘

inverted ¡ less than <

at @ more than greater than >

hash # question ?

dollar \$ inverted ¢

percent % period dot .

caret ^ ellipse ...

ampersand & asterisk \*

left parenthesis underscore \_

or bracket ( plus +

right ) dash -

equal = left curly brace {

Or or vertical bar | right curly brace }

open or left square bracket [ forward slash /

right or right square bracket ] back slash \

colon : semi colon ;

---

Computer Programming: From Beginner to Badass—JavaScript, HTML, CSS, & SQL

A Step-by-Step Guide for Beginners 2019

---

By Zack Fleming and Steven Webber



```
31 def __init__(self, settings):
32     self.file = None
33     self.fingerprints = set()
34     self.logdupes = True
35     self.debug = debug
36     self.logger = logging.getLogger(__name__)
37     if path:
38         self.file = open(os.path.join(path, "requests.txt"),
39                           "a")
40         self.file.seek(0)
41         self.fingerprints.update(set(request_fingerprint(r) for r in self.requests))
42
43 @classmethod
44 def from_settings(cls, settings):
45     debug = settings.getbool("debug", False)
46     return cls(job_dir(settings), debug)
47
48 def request_seen(self, request):
49     fp = self.request_fingerprint(request)
50     if fp in self.fingerprints:
51         return True
52     self.fingerprints.add(fp)
53     if self.file:
54         self.file.write(fp + os.linesep)
55
56 def request_fingerprint(self, request):
57     return request_fingerprint(request)
```

The following eBook is reproduced below with the goal of providing information that is as accurate and reliable as possible. Regardless, purchasing this eBook can be seen as consent to the fact that both the publisher and the author of this book are in no way experts on the topics discussed within and that any recommendations or suggestions that are made herein are for entertainment purposes only. Professionals should be consulted as needed prior to undertaking any of the action endorsed herein.

This declaration is deemed fair and valid by both the American Bar Association and the Committee of Publishers Association and is legally binding throughout the United States.

Furthermore, the transmission, duplication, or reproduction of any of the following work including specific information will be considered an illegal act irrespective of if it is done electronically or in print. This extends to creating a secondary or tertiary copy of the work or a recorded copy and is only allowed with the express written consent from the Publisher. All additional right reserved.

The information in the following pages is broadly considered a truthful and accurate account of facts and as such, any inattention, use, or misuse of the information in question by the reader will render any resulting actions solely under their purview. There are no scenarios in which the publisher or the original author of this work can be in any fashion deemed liable for any hardship or damages that may befall them after undertaking information described herein.

Additionally, the information in the following pages is intended only for informational purposes and should thus be thought of as universal. As befitting its nature, it is presented without assurance regarding its prolonged validity or interim quality. Trademarks that are mentioned are done without written consent and can in no way be considered an endorsement from the trademark holder.

For those listening to the audio book version it is recommended that you also purchase the paperback or kindle eBook version in order to view figures, illustrations and to see exactly how to type different codes and instructions in a precise manner.

## Description

Inside, you will find an introduction to JavaScript, HTML, CSS, and SQL. These are computer programming languages. Some of them are more precisely referred to as scripting languages.

Starting with JavaScript, I will introduce you to the above-mentioned programming languages. Hopefully, by the end of this book, you will have the answers to the questions of what it is and what you can do with it.

JavaScript is a programming language that allows for the implementation of complex items on static web pages. Every time you look at a web page that does something besides display static information, it is a pretty safe assumption to make that JavaScript is involved. It is often referred to as the third layer of the cake.

HTML is a relatively simple programming language that, at its core, consists of elements. These elements allow you to distinguish different sections of text as different aspects on your page. Whether it be a paragraph, a heading, a column, or whatever you need it to be—this is accomplished with the use of these elements.

This language is also where you get the ability to create hyperlinks. The use of hyperlinks is very important, especially when you want to be able to direct someone to a specific part of your text or page.

CSS is a programming language that is used to style and structure the layout of a web page. If you want to change your font, color, and background, you may do so by adding an animation or design a specific outline. This is the language you're going to make use of to accomplish those things.

CSS makes use of the box model—most elements are represented as a box with the content, padding, and borders built-up like layers on an onion. You need to understand the box model before you can begin to understand how to create CSS layouts.

Meanwhile, SQL is a programming language that is designed to work with sets of facts and how they relate to each other. As you might expect, relational database programs make use of SQL. Like many other computer languages, SQL is, in fact, an international standard. However, SQL is easily read and understood even by beginners.

Data sets are described in SQL using SELECT statements. An SQL statement is like a sentence and consists of clauses. Some clauses are required in a SELECT statement.

## Table of Contents

### Introduction

### Chapter 1—The Basics in Programming with JavaScript

Chapter 2—The Basics in Programming with HTML

Chapter 3—The Basics in CSS

Chapter 4—The Essentials for SQL

Chapter 5—Becoming Efficient with JavaScript

Chapter 6—Applying What You Have Learned

Conclusion

Introduction

Congratulations on downloading Computer Programming: From Beginner to Badass—JavaScript, HTML, CSS, & SQL , and thank you for doing so! The world is growing increasingly digital, and downloading this book is the first step you can take towards actually doing something about keeping pace with it. The first step is also never the easiest, however, which is why the information you can find in the following chapters is so important to take to heart, as they are a roadmap for your journey.

Starting with no knowledge in computer programming, you will be taught by this book about how to understand the different programming languages you will encounter and show you how to construct a web page. Learning the principles and practicing with them will give you the keys to unlock a whole new level of understanding of computers and how they work.

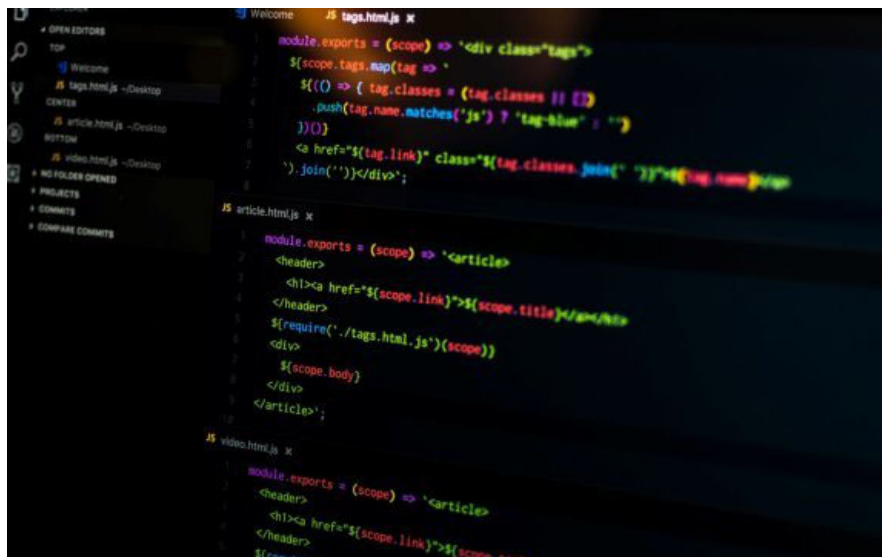
A key part of computer programming is the ability to fix the programs, also known as debugging—finding the errors and being able to correct them. I will include some instruction in this area, in the hopes of you being able to be well-rounded in your knowledge when you are finished reading this book.

I have tried to start with the basics to ensure that you have a solid foundation. Then, I have layered key elements in that are essential to possess in order for it to work for you. I have not included every possible nuance, as there just isn't enough room in the book. I do believe that you will have a solid understanding of the components necessary for you to be able to begin practicing. After all, that is where all the experience and the best understanding come from.

I have tried to make this book not only informative but also interesting as well as efficient. I have tried not to ramble on too much—sometimes you get on to a point that you find so moving and you go on and on about it and everybody else is like, “Okay, got it already. Let's move on.” I tried to avoid those types of scenarios while keeping it from becoming too dry at the same time.

I will spend some time focusing on testing web projects on different browser platforms. I will touch on how to mitigate different types of code and how to go about doing testing. I will delve into using automation in your testing and show you how it can speed the process up for you. I will also mention which tools will be the most useful in fixing problems as well as in testing for them. In spoken language, we tend to emphasize certain words or syllables to manipulate the meaning of what we're saying. It is no different with a programming language, except that the method we tend to stress the words with is italics. In HTML, you would use the *element to perform this task*. *This will make the document more interesting to read—screen readers recognize these, and they will be spoken in a different tone.*

Chapter 1—The Basics in Programming with JavaScript



*"What is JavaScript?" you may be asking. It is a computer programming language. It is used to add interaction to your website. This can take many forms—whether it be a game, a reaction when a button is pushed, or perhaps someone entering data in a form.*

*These may be some of the questions you're thinking to yourself, among a myriad of others that you have no answers to. While it may not be entirely necessary when it comes to learning the language, I thought in the interest of being thorough, that perhaps I should start you off with a little bit of history.*

*In the early 1990s, there was a flourish of activity and businesses being formed. The main area of activity was in the browsers that were competing to be number one. In 1994, Mosaic Communications was founded in California. Its authors were tasked with creating Mosaic Netscape and ensuring that it contained no code that had been previously used in NCSA Mosaic, which was the first popular graphical web browser that had been released. An interesting note that shows the mindset of the times is that the company internally codenamed their browser "Mozilla," which meant "Mosaic killer."*

*In 1995, Brendan Eich was recruited by Netscape Communications. In May of that year, it became apparent that in order to defend the language they wanted to create, they would need a prototype. The writing of this prototype was accomplished by Eich in ten days. Although developed under the name Mocha, by the time it was deployed in December, it had been renamed JavaScript.*

*JavaScript was adopted by Microsoft in 1996 and was part of Internet Explorer 3. However, it was still rapidly gaining a reputation for being a roadblock to a cross-platform standards-driven Web. In the attempt to obtain standardization, Netscape turned to ECMA International—and in December 1999, ECMAScript3 was released. This is the modern-day baseline for JavaScript.*

*JavaScript is almost entirely object-based—and whereas many of the competitive languages use classes for inheritance, JavaScript uses prototypes.*

*JavaScript is extremely versatile, making it an excellent place for the beginning programmer to start. Start small with perhaps an image gallery or programming a response to a button click—and then as your experience expands, you'll be able to create more complex, detailed projects as well as perhaps some database-driven apps, if you so desire. The sky is the limit—well, that and your desire to learn.*

*JavaScript is extremely flexible while remaining fairly compact. There have been a large number of tools developed that operate on the basic JavaScript language. This allows for unlocking incredible amounts of functionality with a minimal amount of effort. Some examples would be Application Programming Interfaces commonly referred to as API's as well as third-party Application Programming Interfaces. There are also third-party frameworks as well as libraries, which add more functionality to the language.*

*Some of these functions can be used in conjunction with HTML, CSS, and SQL, but we'll delve into that a little further in later chapters. I will cover Application Programming Interfaces more thoroughly at a later date. For now, suffice it to say that APIs will expand your knowledge and thus your abilities once you're ready to get to them. At this point, they would likely just confuse you and slow you down.*

*At this point, you are most likely really excited, and you should be! JavaScript and the learning of it will unlock the unlimited potential for you as far as programming is concerned. JavaScript can take a little more time in acclimating to than some of the other languages, however, if you start small and consistently move forward, you will get there.*

*Let's start with something simple—a commonly used first example in many programming languages is the "Hello World" example. Hence, let's do that one—here it is in a few simple steps.*

*A common place to begin is to create a folder, and typically, I would give it the name "scripts" without the quotations. Then, open that folder and create another new file. Give this one the name of "main.js" and again leaving out the quotations. Now, once you are ready and have already saved this, you should notice that unless you specify otherwise, it will save in your scripts folder. Don't specify anywhere else.*

*Next, you will need to open your index.html file. Once it is open, you will need to scroll to the bottom of the body content and enter the following element:*

*This is how you activate the JavaScript you have just written to make it do something on the page. Without this, it would have no effect on anything on the page.*

*Ensure that your work is all saved, then click on your browser's navigation bar and enter index.htm. Once you press enter, this should produce your "Hello world!" greeting.*

*Congratulations, you've just done your first programming in JavaScript. This is probably the best time to point out that there is a reason that the*