



COURSE BOOK

IT-140: Introduction to Scripting

ABSTRACT

A supplemental learning resource that keeps pace with IT-140 coursework.

COCE STEM

© 2018 Southern New Hampshire
Universitysnhu.edu

IT-140 COURSE BOOK

Updated: January 5, 2018

This Course Book was created by SNHU faculty to assist students with learning the Python programming language. This Course Book augments, does not replace, the course textbook or other materials presented in the learning modules.

Chapters of this Course Book have been organized to coincide with the content from each module. An additional chapter, Chapter 0, provides some foundational information to help prepare non-programmers for learning Python.

Table of Contents – Quick Look

Chapter 0 - Introduction.....	6
Chapter 1 – Data and Expressions	11
Chapter 2 – Conditional Statements	25
Chapter 3 - Loops.....	42
Chapter 4 – Built-In Data Structures.....	56
Chapter 5 - Functions	68
Chapter 6 – File Handling and Regular Expressions.....	80
Chapter 7 – Beyond IT-140	86

Table of Contents – Detailed

Chapter 0 - Introduction.....	6
Why Python	6
Commenting Your Code.....	7
In-Code Comment Examples.....	7
In-Code Commenting Tips.....	8
Indenting Code.....	8
Tracing Code.....	9
Terminology.....	9
Conclusion	10
Chapter 1 – Data and Expressions	11
Data Types.....	11
Numbers	12
Boolean.....	12
String	13
Variables.....	13
Naming Rules and Conventions	14
Variable Naming Rules	14
Variable Naming Conventions.....	15
Manipulating Strings.....	16
Combining Strings.....	17
Changing Case	17
Using Escape Sequences	19
Introduction to Methods and Functions	19
Creating Your Own Functions.....	20
The Command Line.....	21
dot Notation	22
Terminology.....	23
Conclusion	24
Chapter 2 – Conditional Statements	25
Operators.....	25

Arithmetic Operators	25
Addition	26
Subtraction.....	26
Multiplication	27
Division	27
Modulus	28
Floor Division.....	29
Exponent.....	29
Comparison Operators	30
Assignment Operators	30
Logical Operators.....	32
Membership Operators	32
Order of Operators / Precedence.....	33
Conditional Statements	34
Introducing the if and if/else statements	34
Using Multiple Conditions	35
Complex Conditional Statements.....	35
Nesting Conditional Statements	36
Using the "in" Operator	37
Boolean Logic	38
Truth Tables	38
Math	40
Terminology.....	40
Conclusion	41
Chapter 3 - Loops.....	42
Iteration	42
Loops	42
For Loops	43
While Loops.....	45
Break / Continue.....	46
Nested Loops.....	48

Indexing	48
Ranges	49
Getting User Input.....	51
Factorials	52
Fibonacci Sequence	53
Terminology.....	54
Conclusion	55
Chapter 4 – Built-In Data Structures.....	56
Built-in Data Structures.....	56
List	57
Creating Lists	57
Working with Lists.....	57
Adding Objects to a List	58
Removing Objects from a List	58
Printing from a List.....	59
Slicing.....	60
Tuple.....	61
Creating Tuples	62
Modifying Tuples.....	62
Dictionary	63
Creating Dictionaries.....	63
Working with Dictionaries	64
Modifying Dictionaries	64
Set	65
Terminology.....	67
Conclusion	67
Chapter 5 - Functions	68
Advanced Methods and Functions	68
More on Methods.....	69
Writing Your Own Methods.....	71
More on Functions.....	71

Writing Your Own Functions.....	72
Randomization.....	75
Searching.....	76
Finding a Single Character in a String.....	76
Finding a Substring in a String.....	77
Finding a Number in a List.....	78
Terminology.....	78
Conclusion	79
Chapter 6 – File Handling and Regular Expressions	80
Regular Expressions	80
Pattern Matching	80
match().....	80
search()	81
findall().....	81
sub()	82
Complex Pattern Matching	83
Reading from a File	84
Writing to a File.....	84
Terminology.....	85
Conclusion	85
Chapter 7 – Beyond IT-140	86
Future Coursework.....	86
Learning Additional Programming Languages.....	86
Using What You Learned in the Real World.....	87
Conclusion	87

CHAPTER 0 - INTRODUCTION

Welcome to IT-140 Introduction to Scripting and welcome to Python. The course you are enrolled in focuses on introducing you to Scripting. Python has been selected as the featured scripting language for this course. So, you might ask yourself, why Python? We will answer that question in this chapter and also provide you with foundational information to help you be successful in the IT-140 course and to get the most out of this book.

Specifically, the following topics will be covered in this chapter:

- Why Python
- Commenting Your Code
- Indenting Code
- Tracing Code
- Terminology

WHY PYTHON

The Python programming language was first released in 1991 and its popularity continues to increase today. This popularity is evident in both Universities and real-world use. Data Analytics, for example, is a business and science area where Python is greatly used. Here are a few reasons for the widespread popularity:

- Easy to Learn
- Highly Readable Code
- Increased Efficiency and Productivity
- Includes a Large Library

When we say Python is easy to learn, we really mean it. The language was created with a focus on easy **syntax** and highly readable code. In fact, Python is a **high-level programming language** which means that Python code is easy to read because it makes use of English-like statements. You will see more of this throughout this course book.

Syntax is the set of rules that determines how computer code is written. Syntax is analogous to grammar in the English language.

Python gives us the ability to write code in fewer lines than might be required of other programming languages. Students and professional developers alike appreciate this as it means less work and greater productivity.

The referenced Large Library is a set of functionalities that can be used with the programming language without having to write them yourself. As an example, there is a math **module** that already has mathematical functions. This enables us to focus on the core of our scripting project and not worry about lower level functionality such as multiplication. We will demonstrate this in Chapter 2 in the Math section.

You should not interpret Python's ease of learning or high-level language characteristic as suggesting it is a simple programming language. Python can be used for small, general purpose applications, web programming, and full-scale enterprise applications.

COMMENTING YOUR CODE

It is common practice to insert **comments** in the code we write. These comments are not executed by the computer and are simply there for humans. The purpose of adding comments to our code is to help document what the various sections of code do. This makes it easier for us to go back to our code to update it or **debug** it.

In-code comments also make it easier for others to understand your code. In nearly every case, there is no single right way to program something. Individual programmers can develop a programming style that includes how things are named and organized. This can make it difficult for other developers, even professionals, to easily understand their code. Code-comments mitigate confusion and supports code maintenance. They also help instructors understand what their students intended to do with their code.

IN-CODE COMMENT EXAMPLES

In Python, we can add comments in our code, also referred to as in-code comments, by starting the line with the pound/hash key (#). The image below shows two lines of text in the script file.

```
1 # This line is a comment
2 print('Hello IT-140 Students')
```

Line 1 starts with the hash key and then contains a comment. When this script file is executed, Line 1 will be ignored and only line 2 will be executed.

Here is another example that shows a multi-line code comment.

```
1 # Student Name: Neo Anderson
2 # Date: January 8, 2018
3 # Course: IT-140
4 # Assignment: Milestone 1
5 print('Hello IT-140 Students')
```

As with the previous example, none of the lines starting with the hash key are executed.

Let's look at one more example. The example below shows how to add an in-code comment to the end of a line of code.

```
1 # Student Name: Neo Anderson
2 # Date: January 8, 2018
3 # Course: IT-140
4 # Assignment: Milestone 1
5 print('Hello IT-140 Students') # Assuming all students are enrolled in IT-140
6 print('Good luck this term!')
```


The example above has four lines of in-code comments on lines 1 through 4. There is an additional in-code comment located at the end of the statement on line 5. When the **interpreter** runs into the hash mark, it ignores the rest of the line.

IN-CODE COMMENTING TIPS

The following tips are generally accepted in the software industry. You may find different requirements in your coursework. In those cases, follow the academic requirements.

Tip 1: Do not over comment. Programming takes enough of your time and you do not need to comment every line of code.

Tip 2: Provide header comments. At the top of your file, you can identify yourself, the project, date, and any copyright information.

Tip 3: Think modularly. Provide brief comments for each method, function, or segment of code. This will help ensure relevant comments are copied along with the code.

Tip 4: Be clear. Do not use jargon or nonstandard acronyms. You want your comments to be easily understood.

Tip 5: Think about the reason for your comment. Some programmers subscribe to the theory that if you need to comment your code, the code itself is not clear enough. Coding software so that it is essentially self-documenting is something you do not have to worry about in this course but is significant enough for you to be aware of it.

INDENTING CODE

So far, using this book, you have not written a single line of Python code. Before you start coding, there is one more thing to cover: **Indentation**.

Python uses code indentation to determine which blocks of code to run. The code snippet below is pretty easy to read even if you have not used Python before. There is a **variable** named "age" and we have an initial **value** of 11 assigned to it. Next, we have a **conditional statement** that evaluates the "age >= 18" condition. If that evaluates as true, the first print statement is executed. If the condition evaluates to false, then the second print statement is executed.

```
age = 11
if age >= 18:
    print('You are old enough for this carnival ride.')
else:
    print('You are not yet old enough for this ride.')
```

You can see that the natural indentation makes the code easy to read. It also allows the interpreter the ability to understand blocks of code. In the example below, we removed the whitespace to the left of the first print statement and receive an error when trying to run the script.

```
8     age = 11
9     if age >= 18:
10    print('You are old enough for this carnival ride.')
11    else:
12        print('You are not yet old enough for this ride.')
```

apter00

```
C:\Users\e.lavieri\PycharmProjects\coursebook\venv\Scripts\python.exe C:/Users/e.lavieri/P
File "C:/Users/e.lavieri/PycharmProjects/coursebook/venv/Scripts/Chapter00.py", line 10
    print('You are old enough for this carnival ride.')
    ^
IndentationError: expected an indented block

Process finished with exit code 1
```

The importance of managing your whitespace is critical.

TRACING CODE

Tracing code is simply the process of stepping through code and logging information with the purpose of ensuring the code works as designed. Sometimes you cannot debug code by simply looking at it. Instead, you need to trace the code. This approach has you acting like the Python Interpreter.

It is difficult to demonstrate the process without exposing you to Python code that you are not yet familiar with. The concept is important enough that we mention it in this initial chapter. You can see a short demonstration using the hyperlink below.

<http://bit.ly/TraceCode>

The language featured in the video is Java, and that is okay. The important thing is to understand the basic concept of tracing code.

TERMINOLOGY

As you read through this course book, you will come across several terms that might be new to you or have a different contextual definition than you are familiar with.

In each chapter, as a new term is introduced, it will be displayed in bold. You can turn back a few pages and notice that we did that in this chapter as well. At the end of each chapter, you can find the new terms defined in the Terminology section. Here is this chapter's list of terms.

Conditional Statement A statement containing an 'if' or 'if/else'

Comments Text in code files ignored by the Python interpreter

Debug The process of looking for and removing bugs in your scripts

High-Level Programming languages that have a high level of abstraction from machine language

Indentation	The arrangement and separation of text.
Interpreter	Software that analyzes and executes code one line at a time
Module	A Python source file
Syntax	Set of rules regarding how write code in a given language
Value	A unit of data such as an integer or a string
Variable	A reserved memory location used to store values
Whitespace	The blank space to the left of code to separate code sections

CONCLUSION

In this chapter you were introduced to the Python programming language. You learned why Python was selected for the IT-140 Introduction to Scripting course. You were exposed to some important foundational information about Python to include in-code comments, indentation, and whitespace. An introduction to the importance of tracing code was provided along with a link to a video example. Lastly, the chapter's new terms were defined in the Terminology section.

In the next chapter, you will start learning how to write code in Python. You will start by learning how Python sees, stores, and uses data. You will also explore how to name components of the code you create. The concepts of methods and functions will be examined and you will learn about the command line, escape characters, and .dot notation.

CHAPTER 1 – DATA AND EXPRESSIONS

In the last chapter you were introduced to Python and learned why Python is ideally suited as the programming language of choice for the IT-140 Introduction to Scripting course. You became familiar with comments, indentation, whitespace, and tracing code.

In this chapter you will learn how to read and write Python code. You will start by learning how Python sees, stores, and uses data. You will also explore how to name components of the code you create. The concepts of methods and functions will be examined and you will learn about the command line, escape characters, and dot notation.

Specifically, the following topics will be covered in this chapter:

- Data Types
- Variables
- Naming Rules and Conventions
- Manipulating Strings
- Introduction to Methods and Functions
- The Command Line
- Dot Notation

DATA TYPES

We typically use software applications to interact, analyze, display, or manipulate data. For example, we use a calculator to perform mathematical operations. We input numbers and symbols (such as /, *, -, and +), and then expect the results to be displayed to us. For example, if you wanted to add 33 and 128, you would simply type in the sequence below:

1. 33
2. +
3. 128
4. =

You would be given the result of 161 on the calculator's display. In Python, we can use the single line below to get the same results.

```
print(33 + 128)
```

So, it goes without saying that Python understands numbers. Numbers is one of the many types of data, or **Data Types**, used in Python. In this section, we will cover the following Data Types:

- Numbers
- Boolean
- String

STRING

So far, you have looked at numeric (int and float) and Boolean data types. Working with text is just as common as working with numbers. Python 3 is capable of displaying text in any world language. We will restrict our examples to English.

A string is a data type consisting of a sequence of characters. These are all valid strings:

- Hello
- Coding is Fun
- I ate 23 wings in 2 minutes!

You can see in those examples that strings can contain letters, numbers, and special characters. In the last two examples spaces are part of the string.

Now, let's look at four examples using Python. The four lines of code depicted below are all valid. You can use single quotes ('), double quotes ("), triple single quotes (' ' '), or triple double quotes (" " " "). Python treats all of the strings the same regardless how you provide them.

```
12 print('This is a string')
13
14 print("This is also a string")
15
16 print("""Even this is a string""")
17
18 print('''You guessed it! This is a string too.''')
```

What you did not see in the examples above are two sets of single quotes as with the example below.

```
20 print('' This is invalid syntax '')
```

Using a pair of single quotes for strings will result in an invalid syntax error.

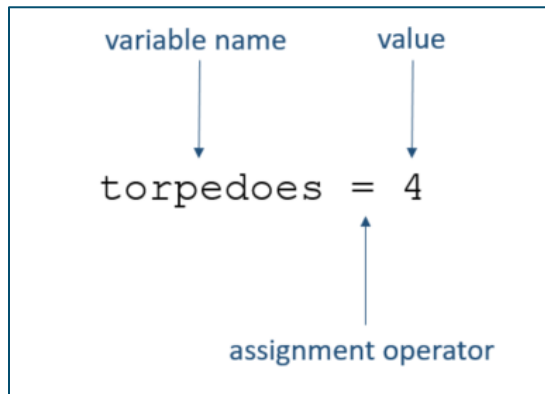
VARIABLES

Variables are memory locations that we create identifiers for so they can store values. Okay, what does that really mean? Let's say we are writing a Sink the Sub video game and want to keep track of how many torpedoes the user has, how many hits, and how many misses. Those are all values that will change throughout the game. So, we can assign variables to keep track of them. Consider the example below.

```
23 torpedoes = 4
24 hits = 0
25 misses = 0
```

You will notice that we do not have to tell Python what type of data we were dealing with. Python automatically detects the type of data we are dealing with and in this case, we are dealing with three integers.

When we created our three variables, we started with the name of the variable, then used the assignment operator, and assigned a value. See below for details.



When Python interprets the line of code it does the following:

- Allocates a memory location
- Creates the torpedoes variable
- Points the torpedoes variable to the allocated memory location
- Sets the data type of torpedoes to int
- Puts the value of 4 in the allocated memory location

That is a lot of work and Python does the hard parts for us. It is important to note how the assignment operator (=) is used. You will learn more about this operator and others in Chapter 2.

Using variables, we can store data and easily refer to it in our code. We might want to use this data, in our Sink the Sub game example, to update on-screen information, detect when the user is out of torpedoes, and calculate a score.

NAMING RULES AND CONVENTIONS

You are probably eager to start writing your code and use variables after having read the previous section. Before we can dive in and start naming our variables, we need to understand the rules for naming them. We will also, in this section, cover several naming conventions.

VARIABLE NAMING RULES

Rules, in the context of naming variables, are things you cannot do. If you violate a rule Python will let you know and by generating an error. Fortunately, there are not too many rules and they are easy to remember. Here are the rules regarding naming variables:

- Cannot begin with a number

- Cannot contain special characters other than the underscore (_)
- Cannot be the same as a Python **keyword**

Python has several reserved names, known as keywords that you cannot use for your own variable names. Here is the list of keywords:

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

Based on that limited rule set, we know that the following are true about variable names:

- Can be of any length
- Can contain letters and numbers
- Can contain upper and lower case
- Can use keywords as long as the case is different (i.e. use false instead of False)

Just because we have great flexibility in naming our variables does not mean we should not show some discipline and consistency in how we name them. That is the subject of the next section.

VARIABLE NAMING CONVENTIONS

In addition to the rules for naming variables, there are also several **naming conventions**. Naming conventions are a set of rules just like the rules for naming variables in the previous section. The difference is that naming conventions are not enforced by Python. Following naming conventions is considered good programming practice. Let’s review the most common naming conventions.

Convention 1. Do not use a modified form of a keyword. For example, do not use While, _while, or while_ because while is a keyword.

Convention 2. Avoid lengthy variable names. While Python will not restrict the length of your variable names, they should not be longer than necessary. Common convention is to limit variable names to 79 characters, but even that length should be avoided. Here are some examples:

Use this	Instead of this
studentAge	the_student_age_as_of_application_run_date
organCount	internal_organ_count_for_the_currently_selected_zoo_animal

Convention 3. Avoid unnecessary mixture of letters and numbers. Sometimes it makes sense to combine letters and numbers. Employ this strategy as needed and ensure the purpose is clear for doing so.

Convention 4. Variable names should be self-descriptive. When we use self-descriptive variable names, our code is easier to read and debug. While a variable name such as ptr3 is legal, it does not tell us what

it is being used for. One might guess that it refers to printer port number 3, when it is actually referring to a golf putting wedge. Refer to the table below for good and bad examples of this convention.

Legal but Bad Examples	Good Examples
nhit	numberOfHits
m	missleCount

The more descriptive our variable names are, the less we need to comment our code. So, by following this convention, we will save ourselves time and make our code more readable!

Convention 5. Use all uppercase for **constants**. Constants are variables that do not change during an applications execution. For example, `PLANET_NAME = "Earth"` let's anyone viewing your code know that `PLANET_NAME` is a constant variable. Also, you will note that there are underscores separating each word in the variable name.

Convention 6. Avoid excessive mixed-case. While we can name our variables with both upper and lower case, we should avoid misuse of this affordance. For example, `mIsslpl` is a valid variable name, but contains unnecessary mixed-case.

Convention 7. Select a single naming schema. There are several variable naming schemas that you can use or even create your own. The most important aspect is that you are consistent in your code. This helps improve readability. Let's review some of the most common naming schemas.

Upper Camel Case. This schema has every word of a variable starting with a capital letter with no underscores between words. Example: `AverageLifeExpectancy`. This is also referred to as Pascal Case.

Lower Camel Case. This schema differs from upper camel case in that the first word starts with a lower case letter. Example: `averageLifeExpectancy`.

Underscore Case. This schema has an underscore between each word in a variable. The first word always begins with a lower case letter and the second and subsequent words can start with an upper or lower case letter. This is also referred to as Snake Case.

***Tip on Naming Conventions.** Employers and instructors can mandate a specific set of naming conventions, so having a general understanding of this concept is sure to set you up for success.*

MANIPULATING STRINGS

You have already learned that strings are a data type consisting of a sequence of characters. In this section, we will further explore strings and learn how to manipulate them.

Strings are a very useful data type. They can be empty, contain one character, multiple characters, several words, and multiple lines of text. The one thing we cannot do is change them. I know that sounds like it would be impossible, but it is true. Strings are an **immutable** data type. This means that strings cannot be changed.

Strings are immutable to provide fixed storage requirements.

Let's look at some ways we can manipulate string data.

COMBINING STRINGS

Combining strings is referred to as string **concatenation**. This is a simple operation. On the following line of code, we add three strings.

```
36 firstName = 'Neo'
37 middleName = 'The One'
38 lastName = 'Anderson'
39 fullName = firstName + middleName + lastName
40 print(fullName)
```

OUTPUT: NeoThe OneAnderson

That output is not the best and we can do better. Let's modify line 39 to include spaces.

```
36 firstName = 'Neo'
37 middleName = 'The One'
38 lastName = 'Anderson'
39 fullName = firstName + " " + middleName + " " + lastName
40 print(fullName)
```

OUTPUT: Neo The One Anderson

CHANGING CASE

Python makes it easy for us to change the case of strings. In the example below, there is one string, `brutalTruth` with a simple 'ships sink subs' value assigned to it. On line 28, the `print()` function outputs the phrase as it exists on line 27.

```
27 brutalTruth = 'ships sink subs'
28 print(brutalTruth)
```

OUTPUT: ships sink subs

We can easily print the entire phrase in upper case using the code below.

```
27 brutalTruth = 'ships sink subs'
28 #print(brutalTruth) # Prints original string
29 print(brutalTruth.upper())
```

OUTPUT: SHIPS SINK SUBS

In this next example, we will print the string in all lower case characters.

```
27 brutalTruth = 'ships sink subs'  
28 #print(brutalTruth) # Prints original string  
29 #print(brutalTruth.upper()) # Prints all caps  
30 print(brutalTruth.lower())
```

OUTPUT: ships sink subs

We can also capitalize the first letter in each word.

```
27 brutalTruth = 'ships sink subs'  
28 #print(brutalTruth) # Prints original string  
29 #print(brutalTruth.upper()) # Prints all caps  
30 #print(brutalTruth.lower()) # Prints all lower case  
31 print(brutalTruth.title())
```

OUTPUT: Ships Sink Subs

If we were to print the brutalTruth again, we would be presented with the original results. Nothing has changed. Let's try it with the code below.

```
27 brutalTruth = 'ships sink subs'  
28 #print(brutalTruth) # Prints original string  
29 #print(brutalTruth.upper()) # Prints all caps  
30 #print(brutalTruth.lower()) # Prints all lower case  
31 print(brutalTruth.title()) # Prints title case  
32 print(brutalTruth)
```

OUTPUT:

Ships Sink Subs
ships sink subs

Now, let's permanently change the value that brutalTruth points to. Review the following code to see how, on line 32, we made the change.

```
27 brutalTruth = 'ships sink subs'  
28 #print(brutalTruth) # Prints original string  
29 #print(brutalTruth.upper()) # Prints all caps  
30 #print(brutalTruth.lower()) # Prints all lower case  
31 #print(brutalTruth.title()) # Prints title case  
32 brutalTruth = brutalTruth.title()  
33 print(brutalTruth)
```

OUTPUT: Ships Sink Subs

USING ESCAPE SEQUENCES

When working with strings, we often want to insert special things into them such as a tab, quote, or line feed. As you likely suspect, this is definitely possible in Python. To accomplish this, we use the **escape character** and **escape sequences**. In Python, the escape character is the backslash and there are several escape sequences. Let's look at a quick example.

```
print('Neo Anderson is: \tThe One!')
```

OUTPUT: Neo Anderson is: The One!

As you can see by the output, there is a tab inserted prior to The One!. We accomplished this by using the escape character (backslash) immediately followed by the escape sequence. In this case, the escape sequence is the letter t for tab.

Let's look at another example.

```
print('Neo Anderson is \"The One!\")')
```

OUTPUT: Neo Anderson is "The One!"

The output shows that quotes were included in the output surrounding the "The One!" text.

This is as easy as it seems and there is a finite list of escape sequences. The five you are most likely to use are listed in the table below.

Escape Sequence	Results
\\	Backslash
\'	Single Quote
\"	Double Quote
\n	New Line
\t	Tab

INTRODUCTION TO METHODS AND FUNCTIONS

Methods and functions are similar and easily confused. So, let's start off with a clear distinction so you are not susceptible to this confusion.

Throughout this chapter we have made liberal use of the print() function. This function, like all functions, has the following characteristics:

1. A piece of code
2. Is called by name

3. Can be passed data (**parameters**)
4. Can provide **return data**

The four criteria for a function are numbered above simply for reference purposes and does not represent a hierarchy or precedence. Let's look at each of these criteria in relation to the `print()` function that you have already been exposed to:

1. The `print()` function code is a core part of Python. The creators of Python wrote the code for that function.
2. We call `print()` by its name.
3. We can pass data to the function but are not required to. If we simply use `print()` a blank line will print. So, in that case, no parameters were passed. As another example, we can pass a number to the function by placing it in inside the parenthesis such as with `print(5)`. Anything we put inside the parentheses are referred to as the parameters.
4. If warranted, the `print()` function can provide return data. When we pass `5 + 7` as parameters, with `print(5 + 7)`, the return data is 12.

Methods are both similar and different from functions. Here are the characteristics of a method:

1. A piece of code
2. Is called by name
3. Is associated with an object
4. Can operate on data within the class

The first two characteristics of methods and functions are the same. The second two characteristics of methods are different than that of functions. You will not work with objects or classes in this course.

CREATING YOUR OWN FUNCTIONS

We are not restricted to the functions available in Python; we can write our own. Let's say we want an easy way to calculate sales prices with a discount. We can do that with the code below.

```
50 def applyDiscount(price, discountRate):
51     return price - (price * discountRate)
52     salePrice = 29.99
53     discount = 0.10
54     print(applyDiscount(salePrice, discount))
```

OUTPUT: 26.991

On line 50 we start defining our function with "def" and then the function name. Our function name is `applyDiscount`. Still on line 50, we add a set of open and close parentheses with two parameters named `price` and `discountRate`. That tells Python that the `applyDiscount` rate expects to receive two parameters each time it is called. We end this line with a colon.

On line 51 we have the rest of the function. This could have been multiple lines and if it was, each of those lines would be aligned with preceding whitespace as with line 51. On that line we are returning the results of the `price - (price * discountRate)` mathematical equation.

Line 52 declares the salePrice variable and sets its value to 29.99.

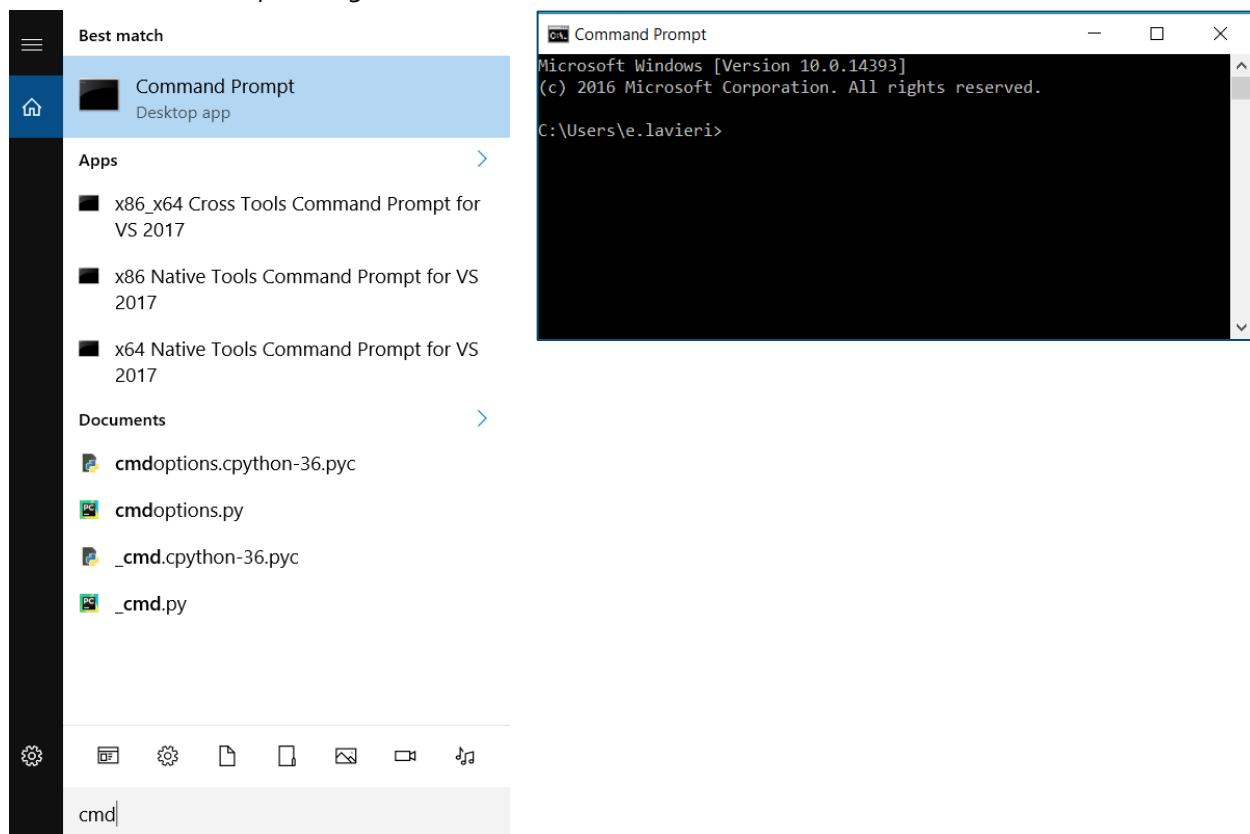
Line 53 declares the discount variable and sets its value to 0.10.

The final line, line 54, uses the print() function to call our applyDiscount function and print the results. As you can see in our function call, we pass salePrice and discount as parameters to our applyDiscount function.

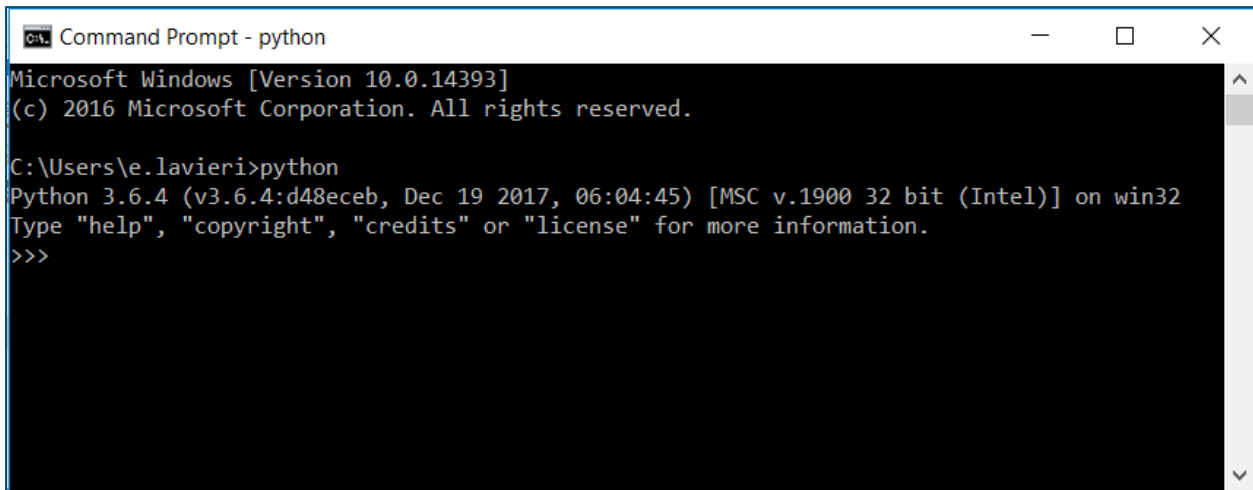
There it is, you wrote your first Python function!

THE COMMAND LINE

The command line refers to using your operating system's console window. This console is referred to as the command prompt in Windows. You can access the command prompt in multiple ways. Perhaps the easiest is to click the search icon in the system taskbar, type cmd, and press return. This will result in a Command Prompt dialog window.



If you have Python properly installed on your computer, you should be able to type "python" and hit return. This will open the Python interpreter. You will notice, as illustrated below, that the command line is preceded with ">>>" and is visually different than the Windows operating system prompt.

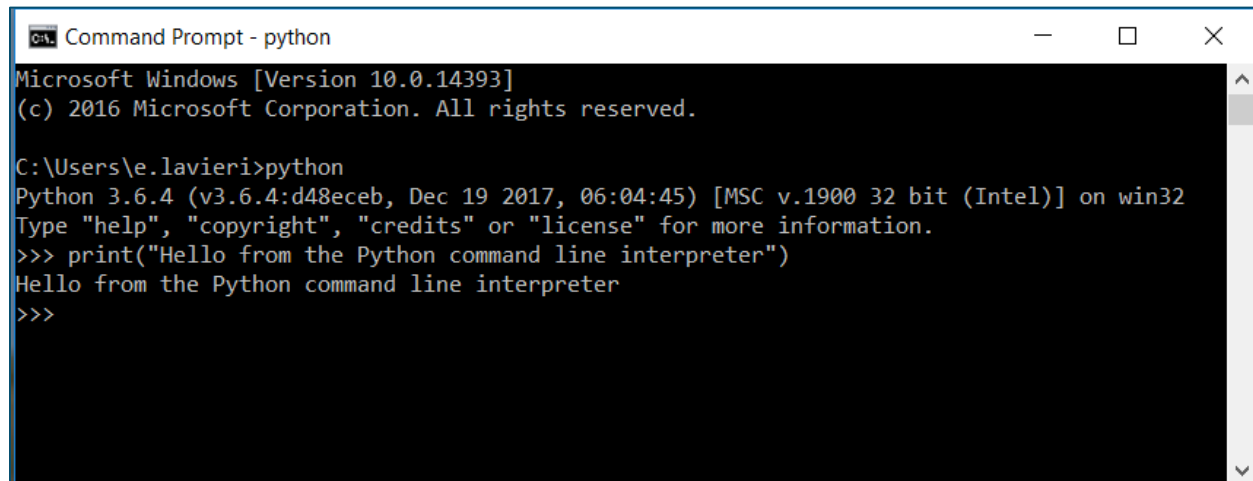


```
Command Prompt - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\e.lavieri>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

From this command line, you can start typing Python commands directly. For example, we will enter the following line of code, press enter, and observe the results:

```
print("Hello from the Python command line interpreter")
```



```
Command Prompt - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\e.lavieri>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello from the Python command line interpreter")
Hello from the Python command line interpreter
>>>
```

You can exit out of the Python interpreter by typing `quit()` and pressing enter. To exit out of the Command Prompt dialog window, simply type `exit` and press return or, of course, click the X in the upper-right corner of the window.

DOT NOTATION

An important part of the Python syntax is the use of **dot notation**. We have already used it in this chapter but did not make a big deal about it. Now it is time to make a big deal about dot notation by reviewing an earlier code snippet.

```

27 brutalTruth = 'ships sink subs'
28 print(brutalTruth) # Prints original string
29 print(brutalTruth.upper()) # Prints all caps
30 print(brutalTruth.lower()) # Prints all lower case
31 print(brutalTruth.title()) # Prints title case

```

On lines 29, 30, and 31, you can see that have a string variable name (`brutalTruth`) followed by a dot (period) and then a method. If we were to read it aloud, we would say “`brutalTruth dot upper.`” When the Python interpreter sees the dot, it refers to the item on the left before proceeding to the right. With line 29 as an example, the `upper()` method refers to the data pointed to by the `brutalTruth` variable, then it executes the function.

This is a relatively simple concept and you will see an increasing number of dot notation as you progress through this course book.

TERMINOLOGY

Boolean logic	Algebraic constructs resulting in one of two values: true or false
Concatenate	Linking strings together in a series
Constant	A variable whose value does not change
Data Type	The category of data such as bytes, numbers, or strings
Dot Notation	Permits
Escape Character	The black slash used to invoke an alternative interpretation of the subsequent character(s)
Escape Sequence	The character(s) immediately following the escape character
Float	A number with a decimal point
Function	A callable object
Immutable	Unable to be changed
Integer	A whole number
Keyword	Reserved names used by Python
Method	A piece of callable code associated with an object

Naming Convention	A set of rules for naming variables
Parameter	A data element passed to a method or function
Return Value	Results of a function returned to the calling function
String	A string of characters
Type	[Data Type] The category of data such as bytes, numbers, or strings
Variable	A storage location identified by an identifier containing a value

CONCLUSION

In this chapter you learned how to read and write Python code. You started by learning several data types. You also discovered how to name variables, while adhering to rules and naming conventions. Extra time was spent on the string data type to include instructions for manipulating them. The concepts of methods and functions were introduced and you even wrote your first function. You also gained exposure to the command line, escape characters, and dot notation.

In the next chapter you will learn about operators and their order of precedence. You will also explore conditional statements, Boolean logic, and truth tables. The chapter will include a section on using math in Python and give you the opportunity to see operator precedence first hand.

CHAPTER 2 – CONDITIONAL STATEMENTS

In the last chapter reviewed some simple Python code. You learned important concepts including variables, data types, methods, and functions. You gained exposure to the command line, escape characters, and dot notation.

In this chapter, you will continue your exploration of the Python programming language. As we introduce new concepts such as operators, order of operations, conditional statements, and Boolean logic, you will be energized with their relative simplicity. We will also introduce truth tables and ensure you are competent with using Python for mathematical computations. Your understanding of Python will significantly increase throughout this chapter.

Specifically, the following topics will be covered in this chapter:

- Operators
- Order of Operators / Precedence
- Conditional Statements
- Boolean Logic
- Truth Tables
- Math

OPERATORS

In the last chapter, we used the calculator metaphor referencing some of the keys you find on every calculator. Specifically, we used the plus (+), divide (/), and multiplication (*) **operators**. Operators are symbols used to perform mathematical or logical computations. You already used three of them and I bet you can think of others on your own. In this section we will cover basic operators and provide examples of each one.

Before we go further, let's introduce a new vocabulary word, **operand**. Operands are the objects that are manipulated in an expression containing an operator. In the $x + y$ example, both x and y are operands and $+$ is the operator.

Operators can be categorized by their use: Arithmetic, Comparison, Assignment, Logical, and Membership. Each of these operator categories is covered in its own section. Additional operator categories, not covered in this course book are Identity, Unary, and Bitwise.

ARITHMETIC OPERATORS

Python's **arithmetic operators** are used for, you guessed it, performing arithmetic. The table below contains the arithmetic operators.

Operator	Description	Example Usage	Results from Example
+	Addition	5 + 5	5 plus 5 equals 10
-	Subtraction	10 - 2	10 minus 2 equals 8
*	Multiplication	10 * 2	10 times 2 equals 20
/	Division	10 / 2	10 divided by 2 equals 5.0
%	Modulus	10 / 3	The remainder of 10 divided by 3 is 1
//	Floor Division	10 // 3	10 divided by 3 equals 3 without the remainder
**	Exponent	10**3	10 times 10 times 10 equals 1000

Let's look at examples of each one of the arithmetic operators.

ADDITION

This section provides four examples. An explanation of each example is provide below the screenshot from the terminal window.

```
>>> 5 + 5
10
```

In the first example, $5 + 5$, we see the expected result of 10.

```
>>> 3 + 1.9
4.9
```

The second example, $3 + 1.9$, results in 4.9. This example shows that we can add a mixed set of integers and floats and the result will be a float.

```
>>> 6.3 + 6.7
13.0
```

The example above adds two floats. Although the result is a whole number (nothing past the decimal point), Python keeps the result as a float as indicated by the decimal point in the result 13.0.

```
>>> 4 + 6 + 20 + 100
130
```

Our final addition example demonstrates how we can have multiple operations in a sequence.

SUBTRACTION

Subtraction operations are not going to surprise you. They work just as if you were entering the operands and operators into a calculator. Let's review three examples.

```
>>> 100 - 25
75
```

In this first subtraction example, we are subtracting 25 from 100 and receive the expected result of 75.

```
>>> 50 - 7.1
42.9
```

The example above illustrates that when at least one of the operands is a float, the result will be a float.

```
>>> 40 - 10 - 5
25
```

Our final subtraction example shows multiple sequential subtraction operations.

MULTIPLICATION

Multiplication in Python is straight forward. As you will see in the follow examples, our calculator metaphor still applies. You are welcome to double check the math using an actual calculator.

```
>>> 3 * 20
60
```

In this first example, we are simply multiplying 3 by 20 with the result of 60.

```
>>> 4 * 5.5
22.0
```

The example above demonstrates how if at least one of the operands is a float, the result of the multiplication will also be a float.

```
>>> 4 * 3 * 5
60
```

Our final multiplication example shows multiple sequential multiplication operations.

DIVISION

Working with division in Python is unsurprisingly easy. Let's look at four examples.

```
>>> 50 / 2
25.0
```

Dividing 50 by 2 results in 25.0. This shows how Python treats division results as a float.

```
>>> 50 / 3
16.666666666666668
```

Dividing 50 by 3 results in a float

```
>>> 2 / 30
0.06666666666666667
```

When the denominator is greater than the numerator, our results will be a float that is less than 1. This is how we calculate percentages. In this example, 2 is essentially 6.7% of 30.

```
>>> 40 / 4 / 2
5.0
```

Our final division example shows multiple sequential division operations.

MODULUS

So far, we have only used operators that you were already familiar with. The modulus operator is perhaps the first operator that is new to you. Fortunately, it is not complex. The modulus operator uses the percent (%) symbol. It provides the remainder of a division operation.

Consider this division example: 10 / 3

Here we are dividing 10 by 3, which, as you know, results in 3.3333333333333335. You can also say the result is 3 with a remainder of 1. Using modulus, instead of division, gives us that remainder. Here it is from the terminal window:

```
>>> 10 / 3
3.3333333333333335
>>>
>>> 10 % 3
1
>>>
```

As you can see above, we demonstrated the division and then the modulus. The modulus operator is typically used to determine if one operand is evenly divisible by another.

In the example below, the modulus operation results in zero indicating that the numerator was evenly divisible by the denominator.

```
>>> 10 % 2
0
```

FLOOR DIVISION

Floor division uses the double slash (`//`) as its operator symbol. When we use floor division, we are only interested in the results to the left of the decimal point. Let's look at three examples.

```
>>> 10 // 2
5
>>> 10 // 3
3
>>> 3 // 30
0
```

The first example, `10 // 2` results in 5. With regular division `10 / 2` would also result in 5.

The second example, `10 // 3` results in 3. If we were using regular division, `10 / 3` would result in 3.3333333333333335. With floor division, we only care about what is to the left of the decimal, so our result is 3.

The third floor division example is `3 // 30`. We know that with regular division, `3 / 30` would result in 0.1. Because floor division only cares about what is to the left of the decimal point, the result is 0.

EXPONENT

The exponent operator is double asterisks (`**`). It is used to calculate exponential power. For example, 5 to the power of 2. In mathematics, this is written as 5^2 and spoken as 5 to the power of 2. In Python, we would write that expression as `5**2`. Here are two examples.

```
>>> 5**2
25
>>> 5**3
125
```

In the first example, we demonstrate 5 to the power of 2 with `5**2`. This equates to $5 * 5$ which results in 25.

In the second example, we demonstrate 5 to the power of 3 with `5**3`. This equates to $5 * 5 * 5$ which results in 125.

COMPARISON OPERATORS

Python's **comparison operators** are used extensively in programming and are sometimes referred to as **relational operators**. They allow us to compare values and make decisions regarding their relationship. As you review the table below, you will realize how basic comparison operators are. The table below contains comparison operators.

Operator	Comparison	Example Usage	Results from Example
==	If both operands are equal	5 == 10	False
!=	If both operands are not equal	5 != 10	True
>	If the left operand is greater than the right operand	5 > 10	False
<	If the left operand is less than the right operand	5 < 10	True
>=	If the left operand is greater than or equal to the right operand	5 >= 10	False
<=	If the left operand is less than or equal to the right operand	5 <= 10	True

We will use these comparison operators later in this chapter when we cover **conditional statements**.

ASSIGNMENT OPERATORS

Python's **assignment operators** are used to assign new values to variables. You have already seen the equal sign (=) used to assign values. That operator is among the list of assignment operators in the table below.

Operator	Description	Example Usage	Equivalent Usage
=	Assigns value on right to operand on left	price = \$29.99	price = \$29.99
+=	Adds the right operand to the left and assigns the result to the operand on the left	newPrice += tax	newPrice = newPrice + tax
-=	Subtracts the right operand from the left and assigns the	newPrice -= discount	newPrice = newPrice - discount

	result to the operand on the left		
<code>*=</code>	Multiplies the right operand by the left and assigns the result to the operand on the left	<code>newPrice *= quantity</code>	<code>newPrice = newPrice * quantity</code>
<code>/=</code>	Divides the left operand by the right and assigns the result to the operand on the left	<code>newPrice /= special</code>	<code>newPrice = newPrice / special</code>
<code>%=</code>	Assigns the modulus of the two operands to the operand on the left	<code>leftValue %= rightValue</code>	<code>leftValue = leftValue % rightValue</code>
<code>//=</code>	Assigns the floor division results from the two operands to the operand on the left	<code>leftValue//= rightValue</code>	<code>leftValue = leftValue // rightValue</code>
<code>**=</code>	Assigns the exponential calculation results to the left operand	<code>leftValue**= rightValue</code>	<code>leftValue = leftValue ** rightValue</code>

Let's look at the first four of these assignment operators in code. Review the code below on your own, then review the results.

```

1  # Operator: =
2  price = 29.99
3  discount = 0.10
4  taxRate = 0.095
5  quantity = 3
6  print(' 6. Unit Price: ', price)
7  print(' 7. Tax Rate: ', taxRate)
8
9  # Operator: -=
10 price -= discount
11 print('11. New Price after discount: ', price)
12
13 # Operator: +=
14 tax = price * taxRate
15 price += tax
16 print('16. New Price with Tax: ', price)
17
18 # Operator: *=
19 price *= quantity
20 print('20. New Price with Quality: ', price)

```

As you review the code output below, you will notice that each line of output is preceded by the line of code the print function call is on. This is just done for easy reference.


```
6. Unit Price: 29.99
7. Tax Rate: 0.095
11. New Price after discount: 29.889999999999997
16. New Price with Tax: 32.729549999999996
20. New Price with Quality: 98.18865

Process finished with exit code 0
```

LOGICAL OPERATORS

Python’s **logical operators** are used to execute code based on multiple conditions. For example, if two conditions are true, then execute a section of code, otherwise bypass the code. The table below contains logical operators.

Operator	Example	Evaluates as True if:
and	A and B	both A and B are true
or	C or D	both C or D are true

We will demonstrate the use of logical operators in the Conditional Statements section of this chapter.

MEMBERSHIP OPERATORS

Python’s **membership operators** are used to determine if one operand is or is not contained in the second operand. The table below contains the two membership operators.

Operator	Description	Example Usage
in	Will evaluate to true if the left operand is a member of the right operand	vowel in word
not in	Will evaluate to true if the left operand is not a member of the right operand	vowel not in word

We will demonstrate the use of membership operators in the Conditional Statements section of this chapter.

Even More Operators. Python includes additional operators from what is presented in this chapter. We shared the most common operators. If you are hungry for more, consult the Python documentation and search for Identity, Unary, and Bitwise operators.

ORDER OF OPERATORS / PRECEDENCE

We covered 25 operators in this chapter and, as indicated, there are even more used in the Python language. Even with our 25, we can easily misuse them resulting in less than desired results. Consider the following simple math statement. What do you think the result is?

$$5 + 3 * 2$$

Did you say 16? If you did, let me explain why that is the incorrect answer. In Python, there is a distinct **order of operations**, also known as **operator precedence**. The Python interpreter does not simply execute the math operations from left to right. In our example, multiplication has a higher priority than addition, so $3 * 2$ is executed first then the addition. Here are the steps:

1. $3 * 2 = 6$
2. $5 + 6 = 11$

The acronym PEMDAS is often used to help students learn the order of operations. PEMDAS stands for **P**arenthesis, **E**xponents, **M**ultiply, **D**ivide, **A**dd, **S**ubtract. Interestingly, PEMDAS has an alternate meaning to help students remember the acronym: **P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally. This is helpful but not as informative of the table provided below that shows the order of operations from highest priority to the lowest.

Priority	Operator	Remarks
1	()	Parenthesis
2	**	Exponent
3	* / % //	Multiply, Divide, Modulus, Floor Division
4	+ -	Addition, Subtraction
5	in not in < <= > >= != ==	Membership and Equality Tests
6	not	Boolean not
7	and	Boolean and
8	or	Boolean or

Important Notes:

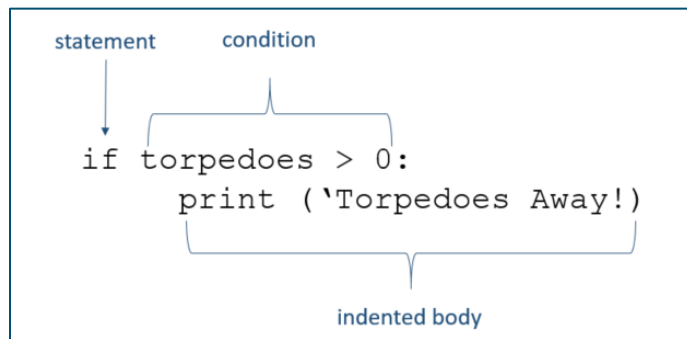
1. When more than one operator has the same precedence, Python will execute them from left to right.
2. The use of parentheses can be used for clarity and to dictate precedence since they have the highest precedence.
3. The order of operators applies inside parenthesis.
4. Not all operators are listed here. The list has been condensed to prevent providing you with more information that you require for this course.

CONDITIONAL STATEMENTS

Conditional statements are usually a core part of any application. If this is true, do that; otherwise do something else or nothing at all. That makes sense. We use conditional statements in our lives, so they are easy to understand. If Grace finished her vegetables, give her desert. It really is that simple.

INTRODUCING THE IF AND IF/ELSE STATEMENTS

In Python we use the “if” statement combined with operands and operators to form our conditional statements. The following graphic shows a basic conditional statement and identifies the individual components.



The conditional statement above uses the “if” statement. The condition, `torpedoes > 0`, will resolve to either True or False. This is a **Boolean expression**. If the Boolean expression evaluates to True, then the indented body will be executed. We could have more indented lines and they would each be executed, as illustrated in the following example.

```
torpedoes = 0
if torpedoes > 1:
    print('Torpedo launched!')
    torpedoes -= 1
```

Rule Note. Python requires you to have at least one indented line in the indented block.

Let’s expand our example and produce a message to let the user know s/he is out of torpedoes.

```
torpedoes = 0
if torpedoes > 0:
    print('Torpedo launched!')
    torpedoes -= 1
else:
    print('You do not have any torpedoes to launch!')
```

In the example above, we introduced the “else” making this statement an if / else statement. As you can see, the statement evaluates the condition (`torpedoes > 0`) and does one thing if it evaluates to True and something else if it evaluates to False.

Remember, with Boolean expressions, we are evaluating a condition to see if it is True or False.

USING MULTIPLE CONDITIONS

We can add more branches to our if/else statements by using "elif" which is short for 'else if.' Here is the syntax for using elif.

```
if <condition>:  
    indented body  
  
elif <condition>:  
    indented body  
  
else:  
    indented body
```

Rule Notes.

1. *You are not required to use "else"*
2. *You can use "elif" unlimited times*

Using this statement allows us to check for multiple conditions. Based on the syntax above, if the first condition is evaluated as True the first indented body would be executed and the elif and else portions of the statement would not be executed. If the first condition evaluated to False, the elif condition will be evaluated. If that is True its indented body is executed. Only if all if and elif conditions evaluate to False, will the indented body under the else be executed. Let's look at an example.

```
torpedoes = 0  
if torpedoes > 1:  
    print('Torpedo launched!')  
    torpedoes -= 1  
elif torpedoes == 1:  
    print('You launched your last torpedo!')  
    torpedoes -= 1  
else:  
    print('You do not have any torpedoes to launch!')
```

COMPLEX CONDITIONAL STATEMENTS

The previous torpedo examples were basic and certainly easy to read. We can get a bit more complex by using **compound conditions**, which is more than one grouped condition. For example, we might say, if Grace ate her vegetables and finished her homework, then give her ice cream. Let's look at how that looks in code.

```
vegetables = False
homework = False
if vegetables and homework:
    print("You deserve ice cream.")
elif vegetables or homework:
    print("Did you forget something?")
else:
    print("Vegetables and homework first!")
```

In the example above, we declared two Boolean variables: `vegetables` and `homework`. The first conditional statement checks if both values are `True`. The `elif` condition will be `True` if either variable is `True`.

In the next section, we will look at a more complex example to ensure you understand how to read (and write) compound condition statements.

NESTING CONDITIONAL STATEMENTS

Python gives us great control of our conditional statements. We can even nest them, which simply means to subordinate conditional statements in others. Here is an example:

```
If Grace did her Homework:
    If Grace did her Chores:
        If Grade ate her Vegetables:
            Give Grace Ice Cream
```

The example above has three levels or three nested conditional statements. If the first one (the outer one) is `False`, then there is no need to go further down the chain.

Let's now look at an example using Python. We will start with a scenario for our application.

Scenario: There is a Rapid River Rafting Recreational Retreat (R5) with a set of Requirements. Here are those requirements:

```
# Rapid River Rafting Requirements:
# Total weight of group must be at least 150 lbs
# Total weight of group must not exceed 300 lbs
# No individual weight can exceed 125 lbs
# At least one group member must be at least 10 years old
```

We will use `kid1Age`, `kid1Height`, `kid1Weight` for our variables and replicate that for `kid2` and `kid3`. Here is the Python code to determine if this group of three kids can rent a raft and travel down the river:

```

# Start by ensuring no kid is over the weight of 125
if (kid1Weight <= 125) or (kid2Weight <= 125) or (kid3Weight <= 125):
    print('Requirement Met: No individual weight can exceed 125 lbs')
    # Check to ensure total weight is within the range of 150 to 300 lbs
    if ((kid1Weight + kid2Weight + kid3Weight) >= 150) and ((kid1Weight + kid2Weight + kid3Weight) <= 300):
        print('Requirement Met: Total weight of group must be 150-300 lbs')
        # Check to ensure at least one kid is 10 or older
        if (kid1Age >= 10 or kid2Age >= 10 or kid3Age >= 10):
            print('Requirement Met: At least one member must be at least 10 years old')
            print('All Aboard! Enjoy your journey down the river.')

```

Read the three nested conditional statements in the code snippet above and make sure you understand what it is accomplishing. Each conditional statement has a preceding in-code comment line to help clarify the code. Once you understand the code, review the below variables, then see if you can figure out if the kids can ride down the river or not.

```

# Data on kid1
kid1Age = 8
kid1Height = 42
kid1Weight = 60
# Data on kid2
kid2Age = 9
kid2Height = 44
kid2Weight = 70
# Data on kid3
kid3Age = 10
kid3Height = 50
kid3Weight = 80

```

USING THE "IN" OPERATOR

The "in" operator is pretty special and highly useful. The simple example below illustrates how easy it is to use and I am sure you can think of great ways to use it in applications you might write.

```

password = "do not change my initial password"
if "pass" in password:
    print('change it buddy')
else:
    print('password check passed')

```

As you can see, the condition checks to see if "pass" is contained in the value assigned to the password string. If it evaluates as true, the first print() function will be called; otherwise, the second print() function is called.

It really is that easy!

BOOLEAN LOGIC

In the previous chapter, you learned that Boolean is a data type and can have one of two values: True or False. That makes it a very basic data type, but nonetheless very important.

There are several **gates** that govern Boolean logic and you do not need to memorize them. The gates are: AND, OR, XOR, NOT, NOR, XNOR, and NAND. To give you some insight into Boolean gates, you should realize that the binary nature of Boolean (True or False) directly applies to digital circuitry where there are two binary conditions: 0 and 1.

The important thing for you to learn is how to evaluate Boolean expressions. You can do that with the aid of **truth tables**. Those are conveniently provided in the next section.

TRUTH TABLES

Truth tables are a visual way to document all variations of inputs and their corresponding output. We can use these tables to determine what combination of variables are True. Let's look at the "AND" truth table so we know how to read them.

AND	Is True?
True and False	False
True and True	True
False and True	False
False and False	False

We can see in the table above that if both operands in an expression are False, then the entire expression is False. You can read the first line, given an (X AND Y) condition, that if X is True and Y is False, then the expression (X AND Y) is False. It is as simple as that.

Now that you understand how to read a truth table, here are the rest of them.

OR	Is True?
True or False	True
True or True	True
False or True	True
False or False	False

NOT	Is True?
not False	True
not True	False

NOT OR	Is True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	Is True?
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

!=	Is True?
1 != 0	True
1 != 1	False
0 != 1	True
0 != 0	False

==	Is True?
1 == 0	False
1 == 1	True
0 == 1	False
0 == 0	True

You do not need to memorize these tables, but you should understand how to read them and the truth behind each table.

MATH

Mathematicians love Python because of how easy it is to perform math and statistical computations. You do not have to be a mathematician or statistician to enjoy Python's mathematical prowess.

You have already done a lot of basic math in your course, albeit restricted to basic functionality such as addition, subtraction, multiplication, and division. With Python, we can go beyond the basics.

Python comes with a standard library of math functions. To gain access to them, we simply need to import the math module. That is accomplished by entering the following line of code.

```
import math
```

This statement is the same if you are using a terminal window or an integrated development environment. This gives us immediate access to math functions. For example how would you compute the square root of 54? Is the answer using prime factorization? Maybe for a mathematician, but in Python we can simply use the following:

```
math.sqrt(54)
```

Pretty powerful. This is just the tip of the iceberg. Python has a plethora of logarithmic functions, power functions, trigonometric functions, angular conversion functions, hyperbolic functions, representation functions, number-theoretic functions, special functions, and even constants such as `math.pi`.

This section is only intended to open your eyes to how easy math can be in Python.

Need more Math?

Visit: <https://docs.python.org/3/library/math.html>

TERMINOLOGY

Arithmetic Operator	A symbol used to perform a calculation on two operands
Assignment Operator	A symbol used to assign a new value to a variable
Boolean Expression	An expression that evaluates to True or False

Comparison Operator	A symbol used to compare values and make decisions regarding their relationship
Compound Condition	More than one grouped condition
Conditional Statement	Statements that test for a condition and provide execution conditional upon the results
Gate	A Boolean construct that is either open/on or closed/off
Identity Operator	A symbol representing a mathematical operation
Logical Operator	A symbol used in conditional statements
Membership Operator	Used to determine if a value is the member of a set or sequence
Operand	The objects manipulated in an expression containing an operator
Operator	A symbol used for mathematical or logical computation
Operator Precedence	The order in which operations in mathematical expressions.
Order of Operations	The order in which operations in mathematical expressions.
Relational Operator	The same as a comparison operator (just another name)
Truth Table	A table shows all combinations of input with corresponding output

CONCLUSION

In this chapter you continued your exploration of the Python programming language. You learned about operators, order of operations, conditional statements, and Boolean logic. As the topics became more complex, you absorbed the content along with easy to follow examples. You were also introduced to truth tables and learned how easy math can be with Python.

In the next chapter you focus on how to create loops in your Python code. There are multiple types of loops and different strategies for each one. This will be a fun chapter which will include information on how to get user input.

CHAPTER 3 - LOOPS

In the last chapter, you moved your understanding of the Python programming language forward and gained comprehension in the areas of operators, operator precedence, conditional statements, and Boolean logic. You also learned how to perform basic and advanced mathematical calculations in Python.

In this chapter, you will be introduced to the concept of iteration, learn how to use loops, and gain exposure to indexing and ranges. There are also three bonus sections on obtaining user input, factorials, and the Fibonacci sequence.

Specifically, the following topics will be covered in this chapter:

- Iteration
- For Loops
- While Loops
- Break / Continue
- Nested Loops
- Indexing
- Ranges
- Getting User Input
- Factorials
- Fibonacci Sequence

ITERATION

One of the efficiencies we are provided in programming languages such as Python is the ability to reuse our code within an application. We can define a set of related statements and use that set repeatedly. An example in the real world would be defining a set of steps used for washing cars. For brevity, let's say those steps are: Rinse, Soap, Scrub, Rinse, and Dry. We have a long line of cars and we can apply the set of steps for each car. Each pass through the set of steps is an **iteration**. As we work through the list of cars, we are **iterating** through the list.

In Python, we can iterate a specified number of times, iterate through a list of things, and iterate while a certain condition is true. This is all possible with the use of **loops** which we cover in the next section.

LOOPS

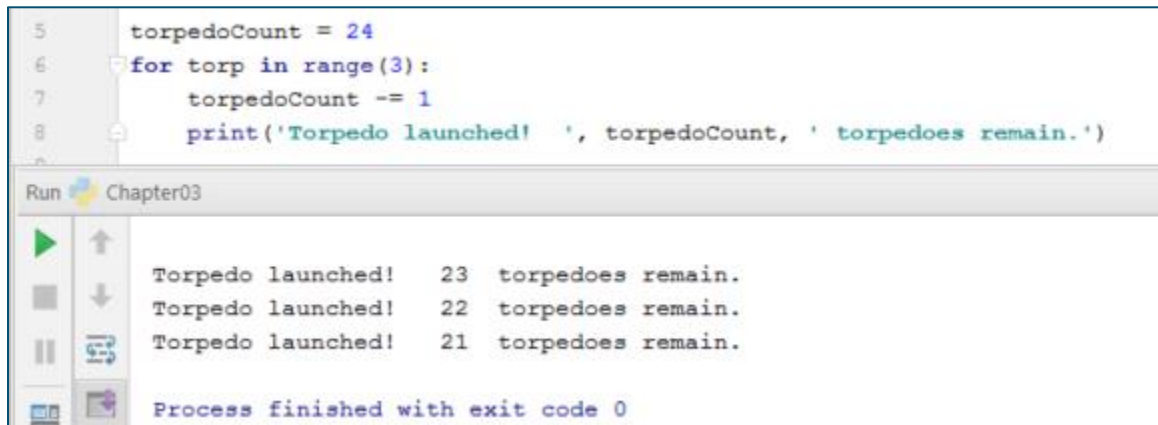
Python makes it easy for use to iterate through data structures (strings, lists, etc.) even when we do not know how many elements there are. In our car washing example, we did not know how many cars were in the line. We iterate through a data structure by looping through from start to finish.

There are two types of loops in Python: the **for loop** and the **while loop**. Let's examine each of these with some Python scripting.

FOR LOOPS

The for loop allows us to repeatedly execute a block of code. This type of control is especially useful when we know how many iterations we want. Here is a quick example along with the output:

```
5 torpedoCount = 24
6 for torp in range(3):
7     torpedoCount -= 1
8     print('Torpedo launched! ', torpedoCount, ' torpedoes remain.')
```

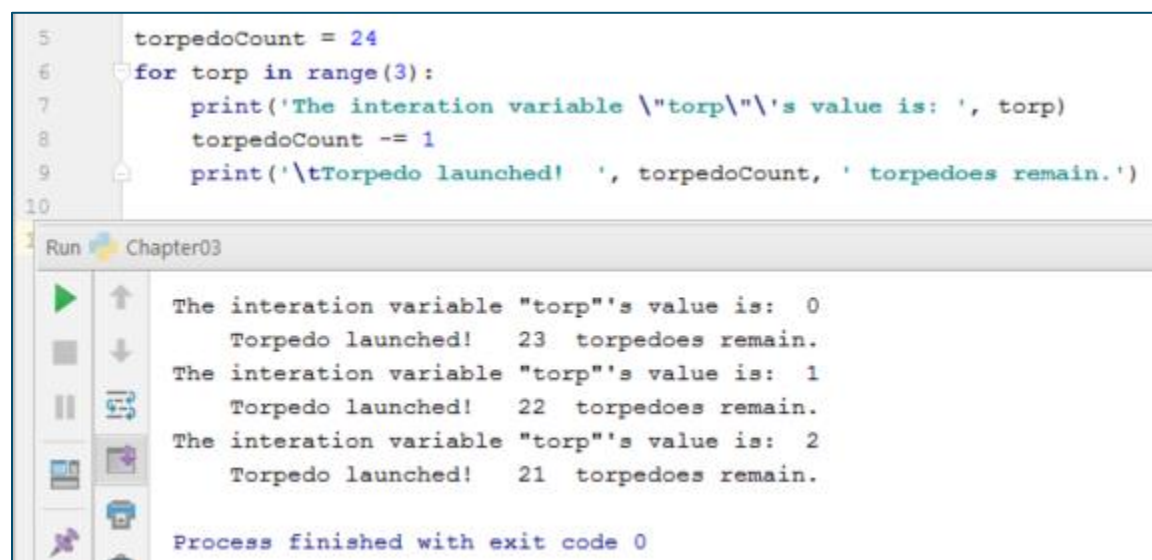


```
Run Chapter03
Torpedo launched! 23 torpedoes remain.
Torpedo launched! 22 torpedoes remain.
Torpedo launched! 21 torpedoes remain.
Process finished with exit code 0
```

The example above executes the loop three times. On line 6 we indicate that we want to loop three times using the **range()** function. This is an important function, so it has its own section in this chapter. For now, accept that if you use **range(#)**, the loop will iterate that number of times. If you just cannot wait, you can jump to the ranges section now, but make sure you come back!

By now you are looking at Python code with a sharp eye and are wondering what the “torp” is on line 6 of our example. It is not used anywhere else in our script and there were no errors. This is our **iteration variable**. This variable, using our example, starts with zero (0) and is incremented by one each time the loop is iterated. Here is an updated example to demonstrate that.

```
5 torpedoCount = 24
6 for torp in range(3):
7     print('The iteration variable \'torp\'s value is: ', torp)
8     torpedoCount -= 1
9     print('\tTorpedo launched! ', torpedoCount, ' torpedoes remain.')
```



```
Run Chapter03
The iteration variable "torp"'s value is: 0
Torpedo launched! 23 torpedoes remain.
The iteration variable "torp"'s value is: 1
Torpedo launched! 22 torpedoes remain.
The iteration variable "torp"'s value is: 2
Torpedo launched! 21 torpedoes remain.
Process finished with exit code 0
```

As you can see, the iteration variable is incremented for us and when it is no longer in the range of 3, the loop ends.

Let's take a look at a more robust example and have some fun at the same time. Consider the Python script below. Can you see what is happening? What do you think the output will be?

```
13 garbledPhrase = "43d6_o@8&or!8@3d42o80@-n+=o<>t@t#1h$e$R~e@i99s@n(}|o@t?+r?*~y||"
14 cleanPhrase = ""
15
16 for i in range(len(garbledPhrase)):
17     if garbledPhrase[i].isalpha():
18         cleanPhrase += garbledPhrase[i]
19     elif garbledPhrase[i] == "@":
20         cleanPhrase += " "
21 print(cleanPhrase.title())
```

Let's review this line, by line. We start with two variables on lines 13 and 14. Our for loop starts on line 16 and has "i" as the iteration variable. Our range is set to the length of the garbledPhrase variable. The len() method returns the length of a string, that is the number of characters in the string. This is great because we are unsure what the length of the variable garbledPhrase might be. Okay, it is 63, but if we are getting our input from a user, that will change.

Tip: Lowercase "i" is commonly used as the iteration variable, but is not required or considered a convention.

We now know that our for loop will iterate 63 times, once for each character in the garbledPhrase string variable. Lines 17 through 20 represent our loop's body. The if statement on line 17 checks to see if the current character in garbledPhrase is a letter or not. We determine this by calling the isalpha() method which returns True or False. You will also note that we use [i] after garbledPhrase to indicate the current **index** position of the string. We will talk more about indexing later in the chapter.

If the current character in garbledPhrase is a letter, it is added to the cleanPhrase string. If it is not a letter, we check, on line 19, to see if it is the @ character. If it is, we add a space to cleanPhrase.

The loop continues iterating through the end of the garbledPhrase and then exits. After our loop, on line 21, we have a print statement that prints the cleanPhrase string we constructed with our loop and it calls the title() method on it.

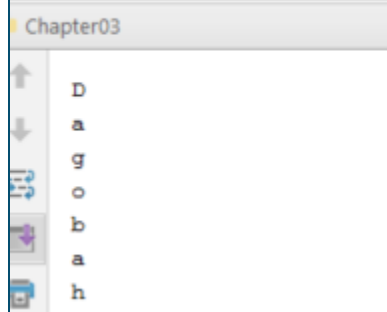
In case you are still wondering, here is the output from our script.

```
Do Or Do Not There Is No Try
```

Both examples provided in this section used the "in range" construct. Let's look at a couple more examples using a different strategy.


The example below loops through the string "Dagobah" using "i" as the iteration variable. So, it iterates seven times.

```
for i in "Dagobah":  
    print(i)
```

A screenshot of a Python IDE window titled 'Chapter03'. The code editor shows a for loop that iterates over the string 'Dagobah' and prints each character. The output console shows the characters 'D', 'a', 'g', 'o', 'b', 'a', and 'h' printed on separate lines.

The example below uses the "in" operator and a list of strings.

```
for name in ["Ada", "Grace", "Jack", "Tim"]:  
    print(name)
```

A screenshot of a Python IDE window titled 'Chapter03'. The code editor shows a for loop that iterates over a list of names: ['Ada', 'Grace', 'Jack', 'Tim'] and prints each name. The output console shows the names 'Ada', 'Grace', 'Jack', and 'Tim' printed on separate lines.

The for loop is indeed a very handy and powerful component of the Python programming language. Remember, we typically use the for loop when we know how many times we want to iterate through the loop. Even though we did not know the length of the garbledPhrase string, we knew that we wanted to iterate the number of times that matches the length, so this counts as knowing how many times we wanted to iterate.

WHILE LOOPS

While loops have similarities to for loops. The major difference is that while loops are used when we want to iterate until a certain condition is met. As you will see, while loops are less complex than for loops, so let's take a look at an example.

The code below starts on line 30 where an int named number is declared with zero as a value. Our while loop starts on line 31 and has the condition of "number <= 5" which means the loop will continue until the variable number's value is greater than 5. Inside the loop, we have a print statement on line 32 and, on line 33, we increment the number variable by one.

```
30     number = 0
31     while number <= 5:
32         print(number, ' x ', number, ' = ', number*number)
33         number += 1
```

Chapter03

```
0 x 0 = 0
1 x 1 = 1
2 x 2 = 4
3 x 3 = 9
4 x 4 = 16
5 x 5 = 25
```

Process finished with exit code 0

Warning: Ensure your loops have a natural exit point. In our example above, if we did not have the `number += 1` statement inside the loop, the value of `number` would never have changed and the loop would have never ended.

There are two important things to remember about while loops:

1. They are typically used when you do not know how many times you want to iterate through a loop.
2. They iterate until a certain condition is met. Examples include: “while `x < 10`” and “while `True`.”
3. Ensure you do not create infinite loops.

BREAK / CONTINUE

Loops are certainly powerful programming constructs and are used extensively in programming. We covered `for` and `while` loops individually. In this section we will discover additional methods of controlling flow with those loops.

We can use the **break** statement to exit out of a loop at any time. In the example below, we have a `for` loop that iterates through a list of names. For each iteration, the current name is checked to see if it consists of alpha characters (letters). If a non-alpha name is found, an error message is printed from line 40 and, on line 41, the loop is exited.

```
36 for name in ["Ada", "Grace", "Jack", "444", "Tim"]:
37     if name.isalpha():
38         print(name)
39     else:
40         print("Invalid name detected. Program excution terminated.")
41         break
```

Chapter03

```
Ada
Grace
Jack
Invalid name detected. Program excution terminated.

Process finished with exit code 0
```

The break statement can be used in both for and while loops.

Another flow control statement used in loops, both for and while, is the **continue** statement. This statement returns control back to the beginning of the loop and ignores any loop processing below it. Let's look at an example.

```
43 number = 0
44 while number <= 11:
45     number += 1
46     if number % 2 == 0:
47         continue
48     else:
49         print(number, " is odd.")
50
```

Chapter03

```
1 is odd.
3 is odd.
5 is odd.
7 is odd.
9 is odd.
11 is odd.

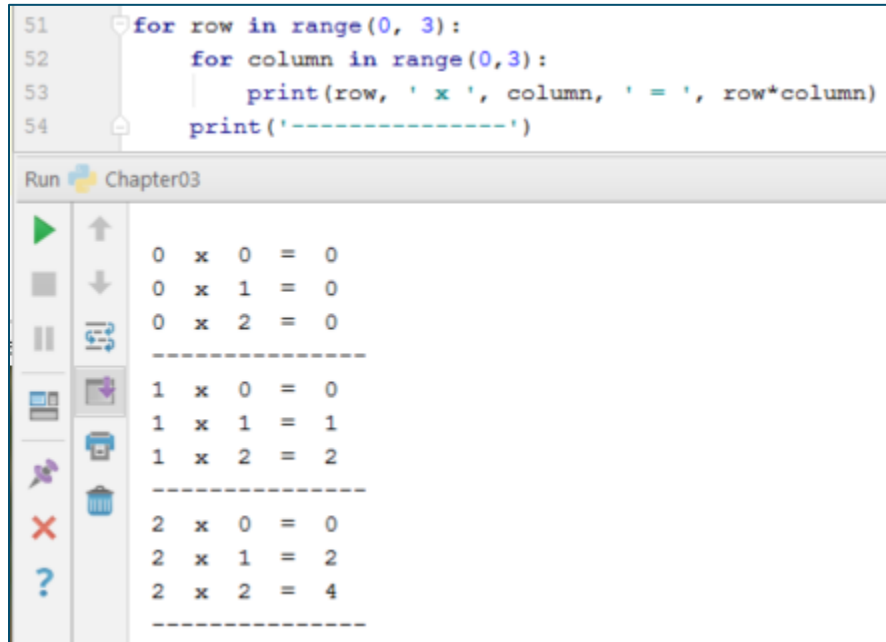
Process finished with exit code 0
```

In the example above, we have simple while loop that checks the iteration variable's value and produces output if the number is odd. If the number is not odd, determined by the conditional statement on line 46, then the continue statement is executed on line 47 and processing resumes at the top of the loop.

NESTED LOOPS

You had some nesting fun with if statements in the previous chapter, so let's have some more nesting fun here. You guessed it, we can nest loops. Let's look at an example that generates a times table.

```
51 for row in range(0, 3):
52     for column in range(0,3):
53         print(row, ' x ', column, ' = ', row*column)
54     print('-----')
```



The two nested for loops in the example above are used to print a simple multiplication table.

Nesting loops can provide solutions to many programming needs. It is important to proceed with caution when nesting loops. They can easily become unreadable and the introduction of infinite loops is always a concern.

INDEXING

You have been exposed to some **indexing** already. So, let's review what we already know.

Remember the statement below? We used this when we first started exploring for loops.

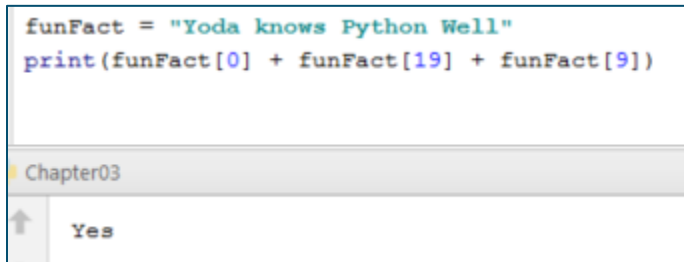
```
if garbledPhrase[i].isalpha():
```

The [i] after garbledPhrase indicates the index position of the string. Indices start with 0. If we have a string with "Yoda knows Python" the first character ("Y") would be at index position 0. Refer to the following graphic for details.

Y	o	d	a		k	n	o	w	s		P	y	t	h	o	n	← Sequence
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	← Index

Now, let's use this in Python to further demonstrate indexing. In the script below, we print three specific letters from the funFact string using their index values. The output is provided at the end of the screenshot.

```
funFact = "Yoda knows Python Well"
print(funFact[0] + funFact[19] + funFact[9])
```



The screenshot shows a code editor window with the following content:

```
funFact = "Yoda knows Python Well"
print(funFact[0] + funFact[19] + funFact[9])
```

Below the code, there is a tab labeled "Chapter03" and a button labeled "Yes".

As you will see in the next chapter, indexing is critical for managing lists and dictionaries.

RANGES

You have already been exposed to ranges using loops, and we will complete our coverage of ranges in this section.

Range is a function and can accept up to three parameters. Here is the syntax:

`range ([start], stop, [step])`

When syntax is provided as with the line above, the parameters are encased in the (parenthesis). Any optional parameters will be bracketed. So, for the range function, the start and step parameters are optional and the stop parameter is mandatory. Here is a quick look at each of the parameters:

Parameter	Requirement	Description
start	Optional	The starting number
stop	Mandatory	The ending number (up to, but not including this number)
step	Optional	Difference between each iteration value

Let's look at a series of range() function examples.

```
# Range Example 1
print('Range Example 1:')
for i in range(3):
    print (i)
```

Chapter03

```
Range Example 1:
0
1
2
```

In Range Example 1, we pass a single parameter, the stop parameter. This causes the for loop to iterate three times. So, the stop parameter had the loop iterate up to but not including the number 3.

```
# Range Example 2
print('Range Example 2:')
for i in range(5, 10):
    print (i)
```

Chapter03

```
Range Example 2:
5
6
7
8
9
```

In Range Example 2, we used both the start and stop parameters, requiring the loop to start at 5 and go up to, but not including 10.

```
# Range Example 3
print('Range Example 3:')
for i in range(1, 11, 2):
    print (i)
```

Chapter03

```
Range Example 3:
1
3
5
7
9
```

The Range Example 3 includes the step parameter. Here, the range starts at 1, goes up to but not including 11, and has a step value of 2. The default step value is 1, so instead of using 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 we will use 1, 3, 5, 7, 9.

```
)# Range Example 4
print('Range Example 4:')
for i in range(10, 0, -1):
    print (i)
```

Chapter03

```
Range Example 4:
10
9
8
7
6
5
4
3
2
1
```

In our final example, Range Example 4, we set the start at 10, the stop at 0, and the step at -1 (negative 1). This causes, as you can determine by the output, the loop to go in reverse index order.

GETTING USER INPUT

Users are arguably the most important aspect of any application. That often requires us to obtain input from them. This section provides an easy method of doing just that. Here is an example that demonstrates how easy it is to get user input.

```
print('Welcome to the System.')
name = input("Please enter your name: ")
print ('Thank you, ', name)
```

Chapter03

```
Welcome to the System.
Please enter your name: Neo
Thank you, Neo
```

Our script prompted the user for input and when the user pressed the enter key, their input was stored in the name string variable. You can see that we subsequently used that input to personally thank the user.

This type of user input is using a terminal window. Using buttons and inform forms is beyond the scope of this course book and the class.

FACTORIALS

Factorials are the product of an integer and all numbers below it. Here are some examples:

Factorial of 3: $1 * 2 * 3 = 6$

Factorial of 4: $1 * 2 * 3 * 4 = 24$

Factorial of 5: $1 * 2 * 3 * 4 * 5 = 120$

We can use the math library's factorial function to calculate these for us. Here is a screenshot from a terminal window.

```
>>> import math
>>> math.factorial(3)
6
>>> math.factorial(4)
24
>>> math.factorial(5)
120
>>> math.factorial(99)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089
4146397615651828625369792082722375825118521091686400000000000000000000
>>>
```

We can also calculate the factorial of a number based on user input. Let's demonstrate how that is done while incorporating two concepts you learned in this chapter: while loop and user input.

```
89 # Factorial Example
90 print('Personal Factorial Calculator')
91 userNumber = int(input('Enter a number: '))
92 userFactorial = 1
93 i = 1
94 while i <= userNumber:
95     userFactorial *= i
96     i += 1
97 print('The factorial of ', userNumber, ' is ', userFactorial)
```

Chapter03

```
Personal Factorial Calculator
Enter a number: 5
The factorial of 5 is 120
```

Let's look at each line:

90. An initial print statement to provide the user with contextual output.
91. Here we obtain user input and store it in an int variable called userNumber. In our case, the user entered 5.

92. We create an int, userFactorial, and set the initial value to 1. We will use this to hold the factorial value as it is being calculated.
93. Create in iteration variable and set the value to 1.
94. This is our while statement. We are using "i" as the iteration variable and will iterate while that variable is less than or equal to userNumber.
95. Here we use the userFactorial *= i statement to start building the factorial.
96. On this line we increment the iteration variable by one.
97. Once execution leaves the while loop, we print the results.

FIBONACCI SEQUENCE

The Fibonacci sequence or **Fibonacci numbers** is a series of numbers where each number is the sum of the two preceding numbers. For example 1, 1, 2, 3, 5, 8, 13, 21, 34 . . .

***Fun Fact:** The Fibonacci sequence is said to have first appeared 450 BC.*

Why is this important? Fibonacci numbers give us a great opportunity to practice using what we have learned up to this point about Python. So, why not have some fun while we practice?

Here is one method you can use to program a Fibonacci sequence in Python:

```
# Fibonacci Sequence Example 1
nbr1 = 1
nbr2 = 1
while nbr1 < 100:
    tmpNbr = nbr1
    nbr1 = nbr1 + nbr2
    nbr2 = tmpNbr
    print(nbr1)
```

Chapter03

```
2
3
5
8
13
21
34
55
89
144
```

In the interest of showing off how efficient programming can be in Python, take a look at the modified version of our code in the next screenshot.

```
# Fibonacci Sequence Example 2
nbr1 = nbr2 = 1
while nbr1 < 100:
    nbr1, nbr2 = nbr1 + nbr2, nbr1
    print(nbr1)
```

Chapter03

2
3
5
8
13
21
34
55
89
144

The output in our second example is the same as with the first example. We were able to do it in four lines of code instead of seven. It is okay if the second example does not make perfect sense to you. It is meant to expose you to the code efficiency that is possible with Python.

TERMINOLOGY

Break	A statement that causes the immediate exit from a loop
Continue	A statement that returns to control to the beginning of a loop
Factorial	The product of an integer and all the integers below it
Fibonacci Numbers	A series of numbers where each number is the sum of the two preceding numbers.
For Loop	A control flow statement
Index	Notation used to identify elements of a data structure
Indexing	(see index)
Iterating	Repeated sequencing through a set of statements (see iteration)
Iteration	A single pass through a set of statements (see iterate)
Iteration Variable	The variable that is updated for each loop iteration as a method of controlling the loop's termination condition

Loop	A programming construct that repeats a set of statements until a specified condition is met
Range	A built-in Python function
While Loop	A control flow statement

CONCLUSION

In this chapter you were introduced to the concept of iteration, learned how to use for and while loops, and gained exposure to indexing and ranges. You also explored how to obtain and use user input as well as how to calculate factorials and Fibonacci sequences.

In the next chapter you will learn about list and dictionary data structures. You will also make use of the knowledge you already have with loops and the “in” operator.

CHAPTER 4 – BUILT-IN DATA STRUCTURES

In the last chapter you were introduced to the concept of iteration and learned how to use for and while loops. You made use of the break and continue statements as you learned about loop flow control. Additionally, you gained exposure to indexing and ranges. Your Python exploration in the last chapter included prompting for and getting user input, factorial calculation, and scripting the creation of Fibonacci sequences.

In this chapter we will examine Python's four built-in data structures: list, tuple, dictionary, and set. We will make extensive use of conditional statements, operators such as the "in" operator, and loops. This is where your previous work pays big dividends.

Specifically, the following topics will be covered in this chapter:

- Lists
- Tuples
- Dictionaries
- Sets

BUILT-IN DATA STRUCTURES

You have come to understand how powerful Python is and how easy some programming tasks are with this programming language. When you least expect it, Python surprises you again. This time with four **built-in** data structures that result in tremendous capability for you, the Python programmer.

Each of the four built-in **data structures** (**list**, **tuple**, **dictionary**, and **set**) will be covered in its own section. You will learn how to create, edit, and use each of these data types. As an overview, you can refer to the following table and the Terminology section at the end of this chapter.

	Data Type	Description	Dynamic	Mutable
Ordered	List	An indexed collection of related objects. Similar to arrays in other programming languages.	Yes	Yes
	Tuple	Similar to lists but are immutable.	No	No
Unordered	Dictionary	An unordered set of key/value pairs with each key being unique	Yes	Yes
	Set	An unordered set of related unique objects	Yes	Yes

The table above mentions four terms that you might not be familiar with. These are defined below:

Dynamic – Size (number of elements can increase and decrease during program execution.

Mutable – Ability to be changed. If a data structure is not mutable, it is immutable.

Ordered – Items stored in an organized, usually numbered, order.

Unordered – Items stored without a specific or ordered organization.

LIST

Lists in Python are a collection of related objects. They are ordered for easy reference and manipulation. What are these objects? They can be anything you can think of. They might be zip codes, names, student IDs, product SKUs, etc. Let's look at how to create and manipulate them.

CREATING LISTS

A basic example of a list is shown below. As you can see, we create this in a manner similar to other variables. We have a variable name on the left, the assignment operator, and then a comma-separated list of objects, in this case, strings. You will also note that the objects are provided inside a pair of square brackets.

```
coloniesOfRobol = ['Aerilon', 'Gemenon', 'Sagittaron', 'Aquaria',  
                  'Leonis', 'Scorpia', 'Canceron', 'Libran',  
                  'Tauron', 'Caprica', 'Picon', 'Virgon']
```

You might also notice that this list took multiple lines in our script, and that is okay. Python knows how to read this.

We can also create an empty list as illustrated below.

```
myEmptyList = []
```

The following example illustrates how to create a list of numbers.

```
jerseyNumbers = [ 44, 8, 24, 32, 42, 33 ]
```

In the next section, we will start using the lists we created.

WORKING WITH LISTS

In this section we will explore several examples of using lists. Our first example has a list of jersey numbers. The script iterates through the list looking for 8 or 24. The output results are provided.

```
jerseyNumbers = [ 44, 8, 24, 32, 42, 33 ]  
for jersey in jerseyNumbers:  
    if ((jersey == 8) or (jersey == 24)):  
        print('Jersey Number ', jersey, ' belongs to Kobe Bryant.')
```

Chapter04


```
Jersey Number 8 belongs to Kobe Bryant.  
Jersey Number 24 belongs to Kobe Bryant.
```

We can easily determine how many objects are in our list with a simple call to the len() function. Using our example, we could use len(jerseyNumbers) to discover that there are 6 objects in the list.

ADDING OBJECTS TO A LIST

So, we have a list with selected jersey numbers for the Los Angeles Lakers. What if we wanted to add Elgin Baylor's jersey number (22) to the list? We can use the append() method as illustrated below.

```
print(len(jerseyNumbers))
jerseyNumbers.append(22)
print(len(jerseyNumbers))
```



Chapter04

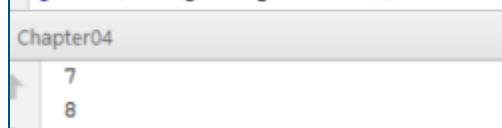
6

7

As you can see in the above example, we print the number of objects in our list (6), use the append() method to add jersey #22, and then print the new length of the list (7).

Earlier, we used the "in" operator to see if a jersey number was a member of the list. Let's check to see if a specific jersey number is not in the list. The example below checks to see if A.C. Green's jersey number (45) is not in the list.

```
print(len(jerseyNumbers))
if 45 not in jerseyNumbers:
    jerseyNumbers.append(45)
print(len(jerseyNumbers))
```



Chapter04


7

8

In the example above, we used a conditional statement with the "not in" operator to quickly determine if 45 was in the list. We printed the length of the list before and after and can see that the new basketball jersey number was added.

We can perform an additional test and print our list. The statement for that is simply print() with the name of our list as a parameter. See the example below.

```
print(jerseyNumbers)
```



Chapter04

[44, 8, 24, 32, 42, 33, 22, 45]

As you can see, the new jersey number (45) is now part of our list.

REMOVING OBJECTS FROM A LIST

We can remove objects from a list just as easily as we can add them. Let's say we decide that we only want jersey numbers from players that are still alive. That means we want to remove jersey number 22.

```
print(jerseyNumbers)
jerseyNumbers.remove(22)
print(jerseyNumbers)
```

Chapter04

```
[44, 8, 24, 32, 42, 33, 22, 45]
[44, 8, 24, 32, 42, 33, 45]
```

Using the dot notation, we merely called the `remove()` method and passed the value of the object we wanted to remove.

Let's revisit our `coloniesOfKobol` list to demonstrate that the `remove()` method works equally as well on strings.

```
coloniesOfKobol = [ 'Aerilon', 'Gemenon', 'Sagittaron', 'Aquaria',
                   'Leonis', 'Scorpia', 'Canceron', 'Libran',
                   'Tauron', 'Caprica', 'Picon', 'Virgon' ]
print(coloniesOfKobol)
coloniesOfKobol.remove("Gemenon")
print("Gemenon removed.")
print(coloniesOfKobol)
```

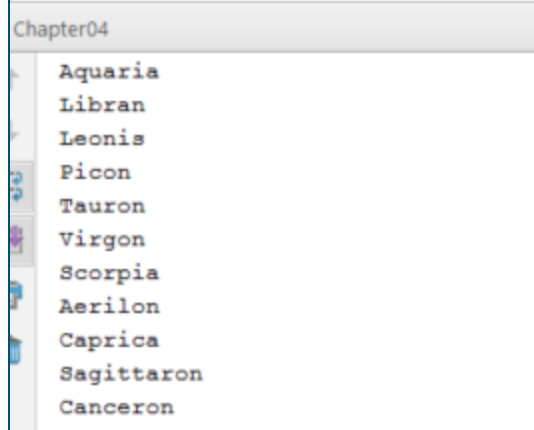
Chapter04

```
['Aerilon', 'Gemenon', 'Sagittaron', 'Aquaria', 'Leonis', 'Scorpia', 'Canceron', 'Libran',
 'Tauron', 'Caprica', 'Picon', 'Virgon']
Gemenon removed.
['Aerilon', 'Sagittaron', 'Aquaria', 'Leonis', 'Scorpia', 'Canceron', 'Libran', 'Tauron',
 'Caprica', 'Picon', 'Virgon']
```

PRINTING FROM A LIST

As you can see in the script below, printing the objects in a list is very easy. Thank you, Python.

```
colonyList = list(coloniesOfKobol)
for colony in colonyList:
    print(colony)
```



Chapter04

- Aquaria
- Libran
- Leonis
- Picon
- Tauron
- Virgon
- Scorpia
- Aerilon
- Caprica
- Sagittaron
- Canceron

Our for loop above uses “colony” as the iteration variable and colonyList as the list that is being iterated over. In each iteration, the colony variable refers to the current object.

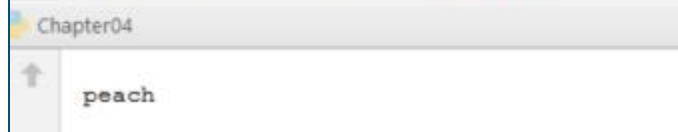
SLICING

Sometimes we want to pull out a **slice** of a sequence. For example, we might have a long string and we just want to get a section, or slice of it. The process of getting the slice is referred to as **slicing** and we use a special **slicing notation** to accomplish it. We will look at how to do this with a string and then look at how to slice a list.

For our first example, we will extract a slice from a string. Consider this code:

```
def saveThePrincess(sequence, princess):
    theStart = sequence.find(princess)
    theEnd = theStart + len(princess)
    print(sequence[theStart : theEnd])

rawData = "monsterpeachmonster"
saveThePrincess(rawData, "peach")
```



Chapter04

↑ peach

In the example above, we defined a saveThePrincess function that takes in a reference to a string (our sequence) and the subset we want to extract. The function looks for the subset inside the string and determines where in the sequence it starts and stops. If we knew the placement already, we could have just used this line of code:

```
print(rawData[7 : 12])
```

As you can see, we used [start : end] as our notation. The complete slicing notation is [start : end : step]. We can also use [:] to extract the entire sequence.

We can also slice a list. For fun, let's start with a string containing a phrase and then convert it to a list. Here is how we do that:

```
badNews = "Princess Peach was taken once again in New Super Mario Bros."
badNewsList = list(badNews)
print('HERE IS THE STRING')
print(badNews)
print('HERE IS THE LIST')
print(badNewsList)
```

Chapter04

↑
HERE IS THE STRING
↓
Princess Peach was taken once again in New Super Mario Bros.
HERE IS THE LIST
['P', 'r', 'i', 'n', 'c', 'e', 's', 's', ' ', 'P', 'e', 'a', 'c', 'h', ' ', 'w', 'a', 's', ' ', 't', 'a', 'k', 'e', 'n', ' ', 'o', 'n', 'c', 'e', ' ', 'a', 'g', 'a', 'i', 'n', ' ', 'i', 'n', ' ', 'N', 'e', 'w', ' ', 'S', 'u', 'p', 'e', 'r', ' ', 'M', 'a', 'r', 'i', 'o', ' ', 'B', 'r', 'o', 's', '.']

Okay, for the rest of our example, we will use the badNewsList list.

```
badNews = "Princess Peach was taken once again in New Super Mario Bros."
badNewsList = list(badNews)
marioResponse = ''.join([badNewsList[25], badNewsList[13],
                        badNewsList[8], badNewsList[3], badNewsList[53]])
print('Mario said: ', marioResponse)
```

Chapter04

↑
Mario said: oh no

There are two things to notice about the code above. First, we used the slicing notation to get a slice of the badNewsList list using badNewsList[#]. For instance, badNewsList[13] returned the "h" we used as we built the marioResponse string. We also used the join method which returns a string from the elements it was passed. We could have used standard concatenation as shown below to achieve the same results.

```
marioResponse = badNewsList[25] + badNewsList[13] + \
                badNewsList[8] + badNewsList[3] + badNewsList[53]
```

TUPLE

You will recall that tuples are similar to lists with the exception that they are **immutable**. As you plan your data structure use for programming project, decide if you have lists of data that you need in your program that you know will not change.

You already employ this strategy when naming variables. If you have an individual data component that you know will not change, then you make it a constant versus a regular variable.

There are several benefits for using tuples as long as the immutable characteristic suits your needs. Here are primary benefits:

- Protects data from being accidentally or maliciously changed.
- Tuples offer greater processing efficiency over lists.
- Helps avoid application bloat (where it is bigger than it needs to be).

CREATING TUPLES

When we create a tuple, we use parenthesis instead of curly brackets. The below code shows a tuple of colony names. We know those will not change.

```
theImmutableColoniesOfKobol = ( 'Aerilon', 'Gemenon', 'Sagittaron', 'Aquaria',
                                'Leonis', 'Scorpia', 'Canceron', 'Libran',
                                'Tauron', 'Caprica', 'Picon', 'Virgon')
print(theImmutableColoniesOfKobol)
print(type(theImmutableColoniesOfKobol))
```

Chapter04

```
('Aerilon', 'Gemenon', 'Sagittaron', 'Aquaria', 'Leonis', 'Scorpia', 'Canceron',
 'Libran', 'Tauron', 'Caprica', 'Picon', 'Virgon')
<class 'tuple'>
```

The example above shows the creation of the tuple using parentheses to encase the objects, or elements of the tuple. We then use the same method of printing the values as with a list. Finally, we call the type() function from within the print() function. The type() function returns the type of the object. In our case, we have confirmation that it is a tuple.

MODIFYING TUPLES

Tuples are immutable and cannot be changed. Here is an example of an error we receive when trying to remove an object from the tuple.

```
theImmutableColoniesOfKobol.remove("Gemenon")
```

Chapter04

```
AttributeError: 'tuple' object has no attribute 'remove'
```

DICTIONARY

Dictionaries are a unique data type and present an interesting set of uses. This data structure is named dictionary because of its look up nature. Python dictionaries consist of two columns of data. The first column contains a unique key and the second column contains values associated with the keys. Each set of key and value is referred to as a **key/value** pair. Here is an example of a player's data for a computer game.

Key	Value
Gamer Tag	Slasher
Current Level	19
Score	30019
Health	75
Energy	100
Player Class	Archer

The information in this table is apt to change frequently, at least some of the components, during game play, so we want a data structure that will support lightning fast look up. Lists cannot provide the quickness we seek, but dictionaries can.

CREATING DICTIONARIES

The script below shows how we would create this dictionary using Python.

```
playerData = {'Gamer Tag' : 'Slasher',
              'Current Level' : '19',
              'Score' : '30019',
              'Health' : '75',
              'Energy' : '100',
              'Player Class' : 'Archer'}
print(type(playerData))
```

Chapter04

```
<class 'dict'>
```

To create the dictionary, we assigned a variable name followed by the assignment operator and then the dictionary contents within curly brackets. Each key/value pair was formatted using the following syntax:

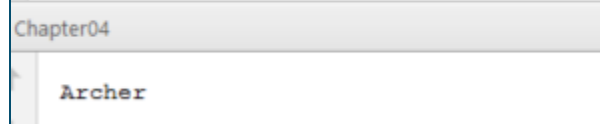
'Key Name' : 'Value'

Multiple key/value pairs were separated with commas. Our script's final statement tested playerData's type, resulting in 'dict' for dictionary being displayed.

WORKING WITH DICTIONARIES

We can obtain a key's value with the following statement:

```
print(playerData['Player Class'])
```

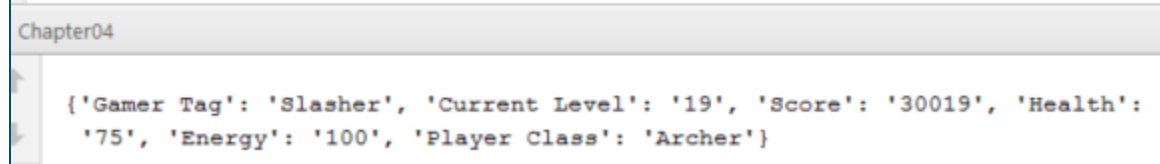


Chapter04

Archer

We can also extract all of a dictionary's data using the statement below.

```
print(playerData)
```



Chapter04

```
{'Gamer Tag': 'Slasher', 'Current Level': '19', 'Score': '30019', 'Health': '75', 'Energy': '100', 'Player Class': 'Archer'}
```

MODIFYING DICTIONARIES

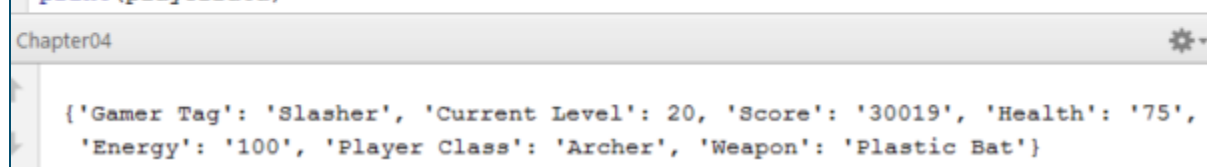
We can edit and add key/value pairs to dictionaries with relative ease. Let's continue our player data example and assume that the player leveled up to level 20. In the game, level 20 players earn their first weapon and we want to track that.

So, we need to perform two actions:

- Change the 'Current Level' key's value from 19 to 20.
- Add a 'Weapon' key with an associated value of 'Plastic Bat.'

```
playerData['Current Level'] = 20
playerData['Weapon'] = 'Plastic Bat'

print(playerData)
```



Chapter04

```
{'Gamer Tag': 'Slasher', 'Current Level': 20, 'Score': '30019', 'Health': '75', 'Energy': '100', 'Player Class': 'Archer', 'Weapon': 'Plastic Bat'}
```

As you can see, we used two statements, one for each of the actions we wanted to take. They are written using the same syntax. You can see by the output that our changes were successful.

Let's modify our dictionary by identifying the Current Level, Score, Health, and Energy as integers instead of strings. As you can see below, we simply took those values out of the single quotes that previously encased them.

```
playerData = {'Gamer Tag' : 'Slasher',  
             'Current Level' : 19,  
             'Score' : 30019,  
             'Health' : 75,  
             'Energy' : 100,  
             'Player Class' : 'Archer'}
```

Now we can update the Current Level using a more mathematical approach. See how we accomplished this in the following screenshot.

```
playerData['Current Level'] += 1  
print(playerData)
```

Chapter04

```
{'Gamer Tag': 'Slasher', 'Current Level': 20, 'Score': 30019, 'Health': 75,  
 'Energy': 100, 'Player Class': 'Archer'}
```

SET

Sets are our final built-in data structure. Sets are a particularly useful data structure. If we have large volumes of data that we need to find the unique values for, using a set data structure would be the ideal solution. Let's look at an example.

```
ages = { 4, 7, 2, 9, 3, 4, 4, 9, 4, 5, 6, 6, 7, 3, 2, 4}  
print(type(ages))
```

Chapter04

```
<class 'set'>
```

The set named ages contains 16 ages for students. We created the set by using curly brackets and separating each element with a comma. You can see that, with our second statement, we checked to see what type Python thinks we created and it knows it is a set.

Are you ready for some quick magic? Look at the print(ages) statement below and the results.

```
ages = { 4, 7, 2, 9, 3, 4, 4, 9, 4, 5, 6, 6, 7, 3, 2, 4}
#print(type(ages))

print(ages)
```

apter04

```
{2, 3, 4, 5, 6, 7, 9}
```

Only unique values were printed.

You can do a lot more with sets and hopefully this section provided enough insight to get you interested in using sets in your applications.

TERMINOLOGY

Built-In	Functionality that is part of the core Python programming language and is available without having to import modules
Data Structure	Formalized structure for the organization and storage of data
Dictionary	An unordered set of key/value pairs
Dynamic	Size (number of elements can increase and decrease during program execution)
Immutable	Inability to be changed
Key/Value	A pair of consisting of a unique key and associated value
List	An indexed collection of related objects
Mutable	Ability to be changed
Ordered	Items stored in an organized, usually numbered, order
Set	An unordered set of related unique objects
Slice	A subset of sequence
Slice Notation	The syntax used to obtain a slice using slicing (see slicing)
Slicing	Obtaining a subset using slice notation (see slice notation)
Tuple	Similar to lists but are immutable (see list)
Unordered	Items stored without a specific or ordered organization

CONCLUSION

In this chapter you examined Python's four built-in data structures: list, tuple, dictionary, and set. You learned the applicability of each data structure, how to create them, how to work with them, and which ones are mutable. You benefited from multiple scripting examples featuring each of these data structures.

In the next chapter you extend what you already know about methods and functions. You will also experiment with randomization with scripts and learn how to program searches using Python.

CHAPTER 5 - FUNCTIONS

In the last chapter you examined Python's four built-in data structures: list, tuple, dictionary, and set. You learned the benefits of each of these data structures, their mutability/immaturity, how to create and work with them. You gained exposure and knowledge on how to script operations with these data structures using the Python programming language.

Throughout the previous chapters, you have been exposed to several Python methods and functions. In this chapter you will expand your understanding of how methods and functions work. We will use several examples to solidify your handle on these important Python constructs. You will learn how to use randomization in your applications. In addition, this chapter contains a section specific to searching. In that section, you will learn several strategies for searching various objects in Python.

Specifically, the following topics will be covered in this chapter:

- Advanced Methods and Functions
- Randomization
- Searching

ADVANCED METHODS AND FUNCTIONS

You first learned about methods and functions in Chapter 1 and have used several of them as you progressed through the book. You even wrote your own function. We will build on this and explore more about methods and functions to increase your understanding of them and to inform your scripting endeavors.

First, let's recap what we know about methods and functions. We group those two programming constructs together because they are so similar. Here are more robust definitions of each:

Method	A callable chunk of code that operates on the associated object.
Function	A callable chunk of code that can receive parameters to operate on and optionally can have return data.

So, both methods and functions are chunks of code that do something. The key difference between them is that methods are associated with **objects**. In object-oriented-programming (**OOP**), we say that 'everything is an object' and this is true in Python. You have already worked with several different types of objects including strings, numbers, lists, and functions.

To help you remember the difference between them, consider these two facts:

- Methods are associated with an object; functions are not.
- Functions have return values; methods do not.

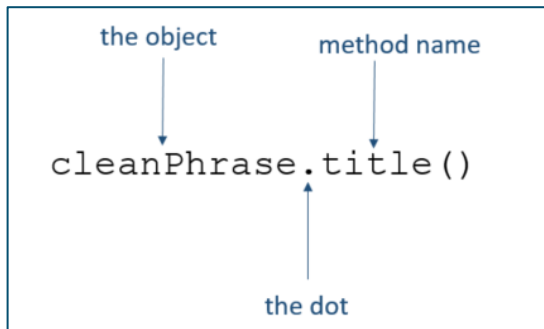
In the next two sections we will individual cover methods and functions.

MORE ON METHODS

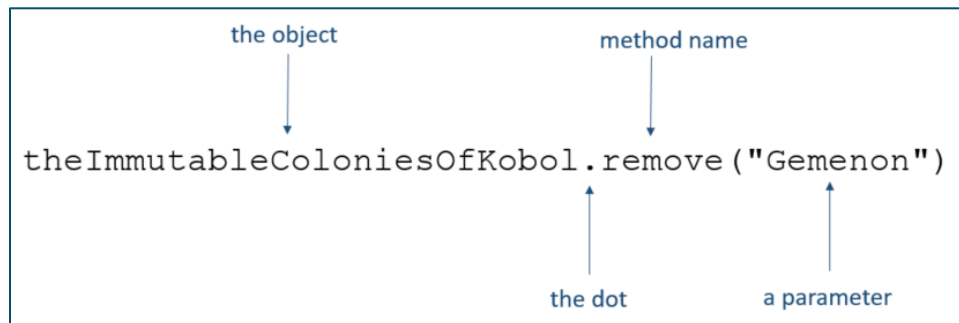
We call methods by referencing the object, followed by a dot (remember the dot notation), the method name, and optional parameters. Here is the basic syntax:

```
object.methodName()
```

The following graphic illustration contains a method call we used in Chapter 3. As you can see, the `cleanPhrase` (a string) is the object and `title()` is the method name.



Our next example uses a line of code from Chapter 4. In this example, our method takes a parameter.



Python has a plethora of built-in methods and you do not need to memorize them. When you are using an editor to enter your Python code, you can easily see what built-in methods are available to you. In the example below, we created a string object called `myObject`. Next, we typed the name of our object followed by a dot. When we type that dot, the editor provided a scrollable list of methods and functions. The methods have a lower case circled-m on the left.

```
myObject = "Python is named after the Monty " \
           "Python Flying Circus TV show."

myObject.
```

m	isalpha(self)	str
m	title(self)	str
m	lower(self)	str
m	upper(self)	str
m	capitalize(self)	str
m	casefold(self)	str
m	center(self, width, fillchar)	str
m	count(self, x, __start, __end)	str
m	encode(self, encoding, errors)	str
m	endswith(self, suffix, start, end)	str
m	expandtabs(self, tabsize)	str
m	find(self, sub, __start, __end)	str
m	format(self, args, kwargs)	str
m	format_map(self, map)	str
m	index(self, sub, __start, __end)	str
m	isalnum(self)	str
m	isdecimal(self)	str
m	isdigit(self)	str
m	isidentifier(self)	str
m	islower(self)	str
m	isspace(self)	str
m	istitle(self)	str
m	isupper(self)	str
m	join(self, iterable)	str
m	ljust(self, width, fillchar)	str
m	lstrip(self)	str
m	partition(self, sep)	str
m	replace(self, old, new)	str
m	rfind(self, sub, __start, __end)	str
m	rindex(self, sub, __start, __end)	str
m	rjust(self, width, fillchar)	str
m	rpartition(self, sep)	str
m	rstrip(self)	str
m	split(self, sep=None, maxsplit=-1)	str
m	splitlines(self, keepends=False)	str
m	startswith(self, prefix, start, end)	str
m	strip(self)	str
m	swapcase(self)	str
m	tabexpand(self, tabsize)	str
m	tabsplit(self, maxsplit=-1)	str
m	translate(self, table)	str
m	uncommonprefix(self, other)	str
m	zfill(self, width)	str

Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>

The list shows any parameters that are supported. When you see (self), as with .upper(self), we do not need to include a parameter because the method refers to the object on the left of the dot. See below for an example.

```
myObject = "Python is named after the Monty " \
           "Python Flying Circus TV show."

myObject.upper()
print(myObject)
```

Chapter05

```
Python is named after the Monty Python Flying Circus TV show.
```

Wait! Something is not right. You know the upper() method should result in the myObject string being in all caps. You will remember that strings are immutable. Let's try this again, this time with a different approach.

```
myObject = "Python is named after the Monty " \
           "Python Flying Circus TV show."

print(myObject.upper())
```

Chapter05

```
PYTHON IS NAMED AFTER THE MONTY PYTHON FLYING CIRCUS TV SHOW.
```

WRITING YOUR OWN METHODS

Most of the method-related work you will be doing in Python will consist of using built-in methods. We can write our own methods. We accomplish this by creating our own **classes** and then defining methods within them.

We can create a class to define a non-built-in object type and assign **attributes** and **behaviors** to it. This is provided as a peep hole in the door leading to object oriented programming. This concept is beyond the scope for this class.

MORE ON FUNCTIONS

You are well versed with functions already. You have used the print() function numerous times as well as several others. You even wrote your own function in Chapter 1. Let's look at that function to refresh our memories on how we did it.

```
def applyDiscount(price, discountRate):
    return price - (price * discountRate)
```

We used the "def" keyword which is short for define. This tells Python that we are creating a function. Immediately after the def keyword, we identified the function name (applyDiscount). Inside the parentheses we indicated we expect two parameters and, internal to our function, we will refer to them as price and discountRate. The last line calculated and return a value.

Warning: *If you pass two few parameters to a function, you will receive an error.*

We are ready to go beyond this simple example. In the next section we will script several functions to work with a dictionary data structure we create.

WRITING YOUR OWN FUNCTIONS

Our scenario is that we want to keep track of the dogs in a kennel. First we will create a dictionary data structure for the dogs. This will be the template we use for each dog. The data fields should be self-explanatory.

```
# Create dictionary template
dog0 = {'Name' : '',
        'Age' : 0,
        'Breed' : '',
        'Color' : '',
        'Boarded' : False,
        'Room' : 0}
```

Next, we will write six functions to allow us to update the individual key/value pairs in our dictionary. Here is an overview of those functions.

Function Name	Parameter(s)	Functionality
updateName	theDog – reference to specific dog newName – the new name for the dog	Change the name of the specified dog.
updateAge	theDog – reference to specific dog newAge – the new name for the dog	Change the age of the specified dog.
updateColor	theDog – reference to specific dog newColor – the new color for the dog	Change the color of the specified dog.
updateBreed	theDog – reference to specific dog newBreed – the new breed for the dog	Change the breed of the specified dog.
boardDog	theDog – reference to specific dog roomNumber – the number of the room the dog is staying in	Change the status of Boarded to True and update the room number for the specified dog.
releaseDog	theDog – reference to specific dog	Change the status of Boarded to False and reset the room number to zero for the specified dog.

Here is the Python code for each of those functions.

```

# Function to update dog Name
def updateName (theDog, newName):
    theDog['Name'] = newName

# Function to update dog Age
def updateAge (theDog, newAge):
    theDog['Age'] = newAge

# Function to update dog Breed
def updateBreed (theDog, newBreed):
    theDog['Breed'] = newBreed

# Function to update dog Color
def updateColor (theDog, newColor):
    theDog['Color'] = newColor

# Function to board dog
def boardDog (theDog, roomNumber):
    theDog['Room'] = roomNumber
    theDog['Borded'] = True

# Function to release dog
def releaseDog (theDog):
    theDog['Room'] = 0
    theDog['Borded'] = False

```

Next, we will create two functions to print a specific dog record. The first one is the most basic and is illustrated below.

```

# Function to print dog record
def printDog (theDog):
    print (theDog)

```

Let's quickly try this printDog function with our dictionary template. As you can see below, the output is accurate, but very crude.

```

printDog(dog0)
Chapter05
{'Name': '', 'Age': 0, 'Breed': '', 'Color': '', 'Borded': False, 'Room': 0}

```

Our last function for our dog kennel application will result in a nicer output of data on a specific dog. Here is the Python code.

```
# Function to print dog record in more readable format
def printDogNicer(theDog) :
    print("Welcome to We Pet 'em Kennel\n")

    if theDog['Name'] == '':
        print("Dog Name: Not provided")
    else:
        print("Dog Name: ", theDog['Name'])

    if theDog['Age'] == 0:
        print("Dog Age: Not provided")
    else:
        print("Dog Age: ", theDog['Age'])

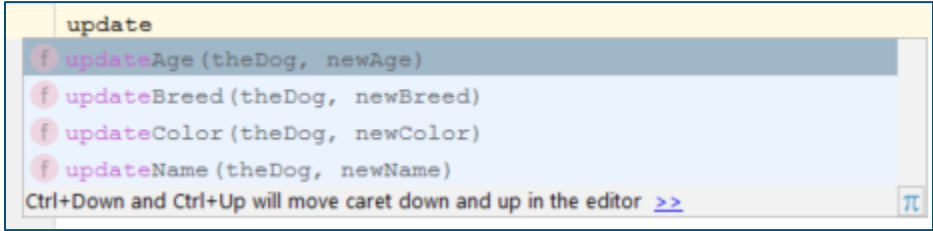
    if theDog['Breed'] == '':
        print("Dog Breed: Not provided")
    else:
        print("Dog Breed: ", theDog['Breed'])

    if theDog['Color'] == '':
        print("Dog Color: Not provided")
    else:
        print("Dog Color: ", theDog['Color'])

    if theDog['Borded'] == True:
        print("Boarding Status: Currently in Room ", theDog['Room'])
    else:
        print("Boarding Status: Not Boarded")
```

As you can see in the script above, we have several conditional statements and provide a nicely formatted output. Before we test this, let's use our other functions to populate a dog dictionary.

Using an editor to type our code, as soon as we type "update," we receive a list of functions starting with "update." As you can see below, all four of our update functions are listed.



```
update
f updateAge(theDog, newAge)
f updateBreed(theDog, newBreed)
f updateColor(theDog, newColor)
f updateName(theDog, newName)
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>
```

In the screenshot below, you can see we first copied the dog dictionary template and then called the four update functions on the new dog dictionary. We next boarded the dog and assigned the dog to room 3. Lastly, we called the printDogNicer function and verified the output.

```
dog1 = dog0.copy() # create a new dog from the template

updateName(dog1, 'Muzz')
updateAge(dog1, 9)
updateBreed(dog1, 'Shar Pei / Golden Mix')
updateColor(dog1, 'Blonde')
boardDog(dog1, 3)

printDogNicer(dog1)
```

Chapter05

```
Welcome to We Pet 'em Kennel

Dog Name: Muzz
Dog Age: 9
Dog Breed: Shar Pei / Golden Mix
Dog Color: Blonde
Boarding Status: Currently in Room 3
```

Before we end this section, let's send the dog Muzz back home by calling the `releaseDog()` function and reprinting her information.

```
releaseDog(dog1)
printDogNicer(dog1)
```

Chapter05

```
Welcome to We Pet 'em Kennel

Dog Name: Muzz
Dog Age: 9
Dog Breed: Shar Pei / Golden Mix
Dog Color: Blonde
Boarding Status: Not Boarded
```

At this point, you should have a strong grasp of Python functions.

RANDOMIZATION

Often times you will have the need to generate a random number within a specific range as part of your application programming. This will section provides you with ways to accomplish that task using Python.

Python includes a module called `random`. To use that module, we need to use the `import random` statement in our code. Here is an example of using the `random.randint()` function to generate a random number between 1 and 10.

```
import random
print(random.randint(1, 10))
```

Chapter05

4

Let's use this in a more contextual example. We will prompt the user to guess our random number and display the results.

```
#
# Randomization
#
import random
answer = (random.randint(1, 10))
guess = int(input('Enter a Your Guess: '))
if answer == guess:
    print("You win.")
else:
    print("You are wrong. Correct answer was ", answer)
```

Chapter05

Enter a Your Guess: 3
You are wrong. Correct answer was 7

We only used one function, but the random module has a long list of functions that you can explore should you need to go beyond the basic random number generation. The URL below is for the official documentation on Python's random module.

<https://docs.python.org/3/library/random.html>

SEARCHING

Python makes searching for things (numbers, strings, objects, etc.) easy. In this section we will review three specific examples. Those examples are:

- Finding a Single Character in a String
- Finding a Substring in a String
- Finding a Number in a List

We will cover additional searching examples in Chapter 6, *File Handling and Regular Expressions*.

FINDING A SINGLE CHARACTER IN A STRING

The following example creates a myString variable and assigns the value of "Python is named after the Monty Python Flying Circus TV show." We also create a variable named aChar that contains the single letter V for a value.

Next, we create a "findIt" function that takes two parameters. We use a while loop to step through the myString string looking for the V letter.

Lastly, we evaluate and output the results. This is an easy method of search a string for a letter.

```
myString = "Python is named after the Monty " \
           "Python Flying Circus TV show."
aChar = 'V'
def findIt(character, phrase):
    i = 0
    while i < len(phrase):
        if phrase[i] == character:
            return i
        i += 1
    return -1

searchResult = findIt(aChar, myString)
if (searchResult == -1):
    print(aChar, 'was found not found.')
else:
    print(aChar, 'was found at index position', searchResult)
```

Chapter05

V was found at index position 54

If our letter is not found, we have the function returning a -1. We know that indexing starts at 0 and any found character will have an index value of 0 or larger. So, if a -1 is returned, we are assured no match was found.

FINDING A SUBSTRING IN A STRING

We will now explore how to find a substring within a string. In the example below, we use the find() function to determine if a substring is within a string. The function can optionally take starting and ending index positions as parameters. When those parameters are not provided, the entire string is searched.

```
myString = "Python is named after the Monty " \
           "Python Flying Circus TV show."
mySub = 'ing'

if (myString.find(mySub) == -1):
    print(mySub, 'was not found.')
else:
    print(mySub, 'was found')
```

Chapter05

↑

↓ ing was found

The code above shows that the substring was found in the string. The find() function returns -1 if no match was found and the index number where the substring match was found (starting point) if a match was found.

We can shorten this code by using the "in" operator as provided below.

```
if mySub in myString:
    print(mySub, 'was found.')
else:
    print(mySub, 'was not found')
```

Chapter05

↑

↓ ing was found.

FINDING A NUMBER IN A LIST

We can search for numbers as well. Our example below uses the "in" operator to determine if a specific grade is located in a list.

```
grades = [88, 74, 92, 65, 23, 95, 99, 100, 88, 97]

if 65 in grades:
    print('65 found in grades list.')
else:
    print('65 not found in grades list.')
```

Chapter05

↑

↓ 65 found in grades list.

TERMINOLOGY

Attribute	A property of an object
Behavior	A method that performs an operation on a class attribute
Class	An object template definition
Function	A callable chunk of code that can receive parameters to operate on and optionally can have return data
Method	A callable chunk of code that operates on the associated object
Object	All data in Python is an object and has a type, identity, and value
OOP	An approach to programming organized by objects
Parameter	A data element passed to a method or function
Return Data	Results of a function returned to the calling function

CONCLUSION

In this chapter you expanded your understanding of methods and functions, focusing on the similarities and differences of them. You worked through additional method and function examples including built-in method and functions as well as several scratch-made functions. In the chapter you also covered randomization and searching.

In the next chapter you will learn what Regular Expressions are and how to use them. We will also explore reading and writing files.

CHAPTER 6 – FILE HANDLING AND REGULAR EXPRESSIONS

In the last chapter you expanded your understanding of methods and functions, focusing on the similarities and differences of them. You worked through additional method and function examples including built-in method and functions as well as several scratch-made functions. In the chapter you also covered randomization and searching.

In this chapter you will learn what Regular Expressions are and how to use them. We will also explore reading and writing files.

Specifically, the following topics will be covered in this chapter:

- Regular Expressions
- Reading from a File
- Writing to a File

REGULAR EXPRESSIONS

Regular expressions refer to Python's `re` module, which we can use when we include "import re" in our script file. That module can be used to perform simple and complex **pattern matching**.

In this section we will cover several regular expression methods for simple pattern matching and then look at a more complex example.

PATTERN MATCHING

For our testing, we require a source to search. We will use a speech from the fictional character Cersei Lannister and name it `cerseiSpeech`. Using that lengthy string as our source, we will experiment with the following regular expressions: `match()`, `search()`, `findall()`, and `sub()`.

In the initial section of our script, we see the import statement and creation of the source string.

```
import re
cerseiSpeech = "The Mad King's daughter has ferried an army of savages to " \
    "our shores, mindless unsullied soldiers who would destroy " \
    "your castles and your holdfasts, Dothraki heathens who " \
    "will burn your villages to the ground, rape and enslave " \
    "your women, and butcher your children without a second " \
    "thought.. You remember the mad king, you remember the " \
    "horrors he inflicted on his people. His daughter is no " \
    "different. In Essos her brutality is already legendary. " \
    "She crucified hundreds of noblemen in Slavers Bay. And " \
    "when she got bored of that, she fed them to her dragons. " \
    "It is my solemn duty to protect the people and I will, but " \
    "I need your help my lords. We must stand together, all of " \
    "us, if we hope to stop her."
```

MATCH()

For our first example, we will see if the string 'Mad' is found in our source. Here is the code:

```
# match()
if re.match('Mad', cerseiSpeech):
    print("Pattern matched!")
else:
    print("No pattern Match.")
```

Chapter06

↑ No pattern Match.

Did you notice that the match failed? This is because the match() method only determine if the source begins with the pattern. In our case, the string 'Mad' was the pattern. Let's try this again and change the pattern to 'The' and change that to a variable.

```
pattern = "The"
if re.match(pattern, cerseiSpeech):
    print("Pattern matched!")
else:
    print("No pattern Match.")
```

Chapter06

↑ Pattern matched!

This time, our pattern match was successful.

SEARCH()

We can replicate the way we used the match() method for the search() method. Reviewing the results below reveals that pattern match was successful.

```
# search()
pattern = "your"
if re.search(pattern, cerseiSpeech):
    print("Pattern matched!")
else:
    print("No pattern Match.")
```

Chapter06

↑
↓ Pattern matched!

The search() method returns the first match, if one is found.

FINDALL()

The findall() method is a fun one to use. It searches for the pattern in the source and returns a list (the data type) of all instances it found. Here is an example where we search for a pattern and then print the results.

```
# findall()
pattern = "your"
results = re.findall(pattern, cerseiSpeech)
print(results)
```

Chapter06

```
['your', 'your', 'your', 'your', 'your', 'your']
```

Because you are already familiar with lists, you know you can do great things with this. Let's have a little fun with the next example.

```
pattern1 = "I "
pattern2 = "your"
pattern3 = "we"
results1 = re.findall(pattern1, cerseiSpeech)
results2 = re.findall(pattern2, cerseiSpeech)
results3 = re.findall(pattern3, cerseiSpeech)

print ('Cersei used', pattern1, len(pattern1), 'times and',
      pattern2, '&', pattern3, (len(pattern2)+len(pattern3)), 'times.')
if (len(pattern2)+len(pattern3)) >= (len(pattern1)):
    print("That is hardly a sign of a selfish ruler.")
```

Chapter06

```
Cersei used I 2 times and your & we 6 times.
That is hardly a sign of a selfish ruler.
```

With the preceding example, we established three patterns and evaluated the results to generate informational output.

SUB()

The sub() method works like a global find and replace operation. In our example below, we take two passes on the source file. Initially, we search for the 'her' pattern and replace all instances, if found, with 'HER.' In our second pass, we search for 'she' and replace any instances with 'SHE.' Finally, we output the final results.

```
# sub()
results1 = re.sub('her', 'HER', cerseiSpeech)
results2 = re.sub('she', 'SHE', results1)
print(results2)
```

Chapter06

The Mad King's daughter has ferried an army of savages to our shores, mindless unsullied soldiers who would destroy your castles and your holdfasts, Dothraki heathens who will burn your villages to the ground, rape and enslave your women, and butcHER your children without a second thought... You remember the mad king, you remember the horrors he inflicted on his people. His daughter is no different. In Essos HER brutality is already legendary. She crucified hundreds of noblemen in Slavers Bay. And when SHE got bored of that, SHE fed them to HER dragons. It is my solemn duty to protect the people and I will, but I need your help my lords. We must stand together, all of us, if we hope to stop HER.

COMPLEX PATTERN MATCHING

The regular expressions module gives us great flexibility and functionality. One of the common programming tasks we have is to validate a user's email address. In this section we will use regex to validate a list of email addresses based on our specified formatting.

Here are our emails for what we will consider a valid email address:

- The user name must be at least one character in length
- The user name must be followed by the @ symbol
- There must be a domain name
- There must be a .com or other . top-level domain provided

Here is the code to create a list of emails, establish a pattern, then validate the emails using the pattern.

```
# email validation
emailList = ['cersei.lannister@got.eu', 'j.a.m.i.eLan@goteu', 'nedstark_got.eu']
emailValidation = re.compile(r'([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4})', re.VERBOSE)

for email in emailList:
    if re.match(emailValidation, email):
        print(email, "is valid")
    else:
        print(email, "is invalid")
```

Chapter06

cersei.lannister@got.eu is valid
j.a.m.i.eLan@goteu is invalid
nedstark_got.eu is invalid

It is okay if the emailValidation pattern line of code is not completely understandable. It looks cryptic but is really just the pattern based on our rules. This was provided to help you appreciate the power of Python.

Want to Know More?

Visit: <https://docs.python.org/3/howto/regex.html>

READING FROM A FILE

Reading from a file is a common programming operation. The first thing we need to do is to open the file. We will use the following syntax:

```
ourVariable = open(filename, mode)
```

We have several modes to open our file with: r, w, x, and a.

- r read mode
- w write mode (creates new file and overwrites existing file)
- x write mode (only for new file)
- a append mode (creates new file or adds content to end of existing file)

We can use a second mode indicator (t for text or b for binary) to indicate the file type. So available options are rt, rb, wt, wb, xt, xb, at, and ab.

With our file open, we can use read(), readline(), and readlines() to read data from the file.

When we have completed the file operations, we need to close the file using the ourVariable.close() statement.

WRITING TO A FILE

Writing to files is pretty straight forward. Let's say that we are writing a sfogliatelle recipe and have stored it the secretRecipe string. We could simply use the following statements to write the entire recipe to file:

```
myFile = open('sfoglitelle', 'wt')
myFile.write(secretRecipe)
myFile.close()
```

Reading and writing to files is relatively straight forward, but can be a bit complex when dealing with different formats and file systems. The previous two sections (reading and writing) were intended to give you an overview of the process.

TERMINOLOGY

Pattern	A sequence of tokens forming a pattern
Pattern Matching	Checking if a string pattern exists in a source string
RE	Short for Regular Expressions (see regular expressions)
Regular Expressions	A Python module containing methods for pattern matching

CONCLUSION

In this chapter you learned what Regular Expressions were and how to use them. You used the `match()`, `search()`, `findall()`, and `sub()` methods and took an advanced look at how to use regular expressions to validate email addresses. You also learned the basics of reading and writing files.

In the next chapter you will learn how the knowledge you learned in the class and with this course book can benefit you beyond this course.

CHAPTER 7 – BEYOND IT-140

This final chapter does not introduce any new Python related content. For that reason, and for the information you are about to read, feel free to consider this the Good News chapter.

In this brief chapter, you will learn how the information you learned in this course will help you in other courses, when (if) you learn additional programming languages, and in the real-world.

The chapter is therefore organized into these three sections:

- Future Coursework
- Learning Additional Programming Languages
- Using What You Learned in the Real World

FUTURE COURSEWORK

As you take non-programming classes you are apt to benefit from your new ability to look at a problem and break it down into component parts. That is how computer programmers do their work. They analyze the problem and turn it into a set of smaller problems. Let's face it, smaller problems are easier to solve than larger ones.

You also have the ability to look at data in a new way. You understand data types and data structures. Those are certainly not unique to the programming world.

Perhaps the most important take away for you will be that you can take any course, even if it does not seem interesting to you, and be successful. We are able to use resources such as course material, course books, textbooks, peers, instructors, and even Google search results to augment our learning.

LEARNING ADDITIONAL PROGRAMMING LANGUAGES

Are you ready for some really good news? Nearly every concept, idea, and term used in this course book is NOT unique to Python. Programming languages have unique features and syntax that are easily learned. The difficult part in learning a programming language is understanding the basic concepts such as variables, data types, data structures, loops, etc. You have already been exposed to that.

So, if you were to learn a new programming language, you do not need a "...For Dummies book." Let's say that you want to learn Java next. You do not need to wonder what kinds of data it has. You already know the answer: integers, floats, strings, etc. Sure, there are some differences, but there are far more similarities in programming languages than there are differences.

Shed the fear of learning new programming language and believe in the old adage that once you learn one programming language, learning additional ones is easy. Full confidence is approved!

USING WHAT YOU LEARNED IN THE REAL WORLD

Okay, so you might not want a career in computer programming, but if you took this class, you are likely to have a career in or around Information Technology. You might even have developers (programmers) working for you or be on a team with them. You understand their language, Python or otherwise. You truly have taken a great glimpse into their world and that will help you communicate with them more effectively.

Finally, when we program, we think of issues and how to solve them. Our minds are analytical and we have the cognitive ability to create organization and solutions from a mess of problems and uncertainty. You are truly ready for anything!

CONCLUSION

We hope this course book was a helpful component of your course resources. Your feedback is welcome. Feel free to provide feedback to your instructor via email. S/he will forward your email to the right person. Alternatively, or additionally, you can include feedback regarding this course book in your end of course survey. Trust us, we want to continually improve and are happy to update this course book as needed.