



```
def test_discrete_probability():
    X = Geometric('X', Rational(1, 5))
    Y = Poisson('Y', 4)
    @@ -321,76 +293,3 @@ def test_product_spaces()
    assert str(P(Eq(X1 + X2, 3))) == """Sum(
    """X2 <= 2). (0, True)). (X2, 1, oo
    P(Eq(X1 + X2, 3)) <= Rational(1,
```

# INTRODUCTION TO PROGRAMMING WITH PYTHON

**LIBFEXDLBDSIPWP01**



**INTRODUCTION TO**

**PROGRAMMING WITH PYTHON**

## **MASTHEAD**

Publisher:  
The London Institute of Banking & Finance  
8th Floor, Peninsular House  
36 Monument Street  
London  
EC3R 8LJ  
United Kingdom

Administrative Centre Address:  
4-9 Burgate Lane  
Canterbury  
Kent  
CT1 2XJ  
United Kingdom

LIBFEXDLBDSIPWP01  
Version No.: 001-2023-1212  
N. N.

© 2023 The London Institute of Banking & Finance  
This course book is protected by copyright. All rights reserved.  
This course book may not be reproduced and/or electronically edited, duplicated, or distributed in any kind of form without written permission by the The London Institute of Banking & Finance.  
The authors/publishers have identified the authors and sources of all graphics to the best of their abilities. However, if any erroneous information has been provided, please notify us accordingly.

# TABLE OF CONTENTS

## INTRODUCTION TO PROGRAMMING WITH PYTHON

### Introduction

Signposts Throughout the Course Book .....	6
Learning Objectives .....	7

### Unit 1

Introduction to Python .....	9
1.1 Why Python? .....	10
1.2 Obtaining and Installing Python .....	13
1.3 The Python Interpreter, IPython, and Jupyter .....	18

### Unit 2

Variables and Data Types .....	35
2.1 Variables and Value Assignment .....	36
2.2 Numbers .....	48
2.3 Strings .....	52
2.4 Collections .....	64
2.5 Files .....	72

### Unit 3

Statements .....	79
3.1 Assignment and Expressions .....	80
3.2 Conditional Statements and Expressions .....	87
3.3 Loops .....	97
3.4 Iterators and Comprehensions .....	104

### Unit 4

Functions .....	111
4.1 Function Declaration .....	112
4.2 Scope .....	118
4.3 Arguments .....	128

### Unit 5

Errors and Exceptions .....	137
5.1 Errors .....	138
5.2 Exception Handling .....	140
5.3 Logs .....	149

<b>Unit 6</b>	
Modules and Packages	155
6.1 Usage	156
6.2 Namespaces	157
6.3 Documentation	160
6.4 Popular Data Science Packages	164
<b>Appendix</b>	
List of References	176
List of Tables and Figures	177

# INTRODUCTION

# WELCOME

## **SIGNPOSTS THROUGHOUT THE COURSE BOOK**

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!



# LEARNING OBJECTIVES

Python has quickly become one of the most popular and widely used software development languages in the world. In **Introduction to Programming with Python**, you will develop an appreciation for the reasons why Python usage is so prevalent, as well as an understanding of the pros and cons of the Python language. The course will explain how to download and install Python while also providing information about different aspects of the program that will help you learn this amazing language.

Upon completion of the course, you will understand rudimentary Python concepts, such as variables and constants, strings, collections, and file input and output (file i/o). Additionally, you will be able to describe and use different types of Python statements, including assignment statements, expression statements, and various internal function statements. You will also be able to explain and demonstrate the use of conditional statements, loops, and iterators.

The course discusses the usage and purpose of functions within Python and how the pertaining syntactic rules can be used to create your own custom functions. You will be able to implement functions with various parameters and return types, and will also be able to describe scope as it relates to variables and functions.

Finally, you will be able to explain and demonstrate the use of error handling methods and log files along with reasons to use modules, namespaces, and several data science packages.



# UNIT 1

## INTRODUCTION TO PYTHON

### STUDY GOALS

On completion of this unit, you will have learned ...

- the reasons why software developers choose to use Python.
- the strengths and weaknesses of the Python language.
- how to download and install Python.
- how to describe and use the various parts of a Python development environment.

# 1. INTRODUCTION TO PYTHON

## Case Study

Two friends, Kyle and Morgan, are more excited than they have been in years. Despite having discussed ideas for new businesses in the past, neither has acted on any of those ideas so far. However, today is different. As they sit together having lunch and talking about another business idea, there is a different vibe in the room. Both Kyle and Morgan are serious about making this business idea become a reality.

So, what exactly is this new and exciting idea? Having both played soccer together at their university, many of the business ideas Kyle and Morgan have tossed around over the years revolve around soccer. However, they have had a difficult time coming up with a soccer-based idea that might actually make money—until now! Kyle wants to utilize Morgan's background in statistics and his own knowledge of software development to create a soccer player improvement software application. The program will help players and coaches analyze player tendencies through historical data and then identify areas of focus for improvement.

There is no doubt in their minds that this idea is the one that will end up being their successful start-up. Both Kyle and Morgan realize that their application could help many soccer players worldwide and selling it to them could help their bank accounts. To maximize the distribution of the application, it must work on a variety of platforms seamlessly. Additionally, the software will heavily rely on statistical computing power, using a lot of data and complex mathematics. As they contemplated these constraints, Kyle and Morgan identified several questions that will have far-reaching ramifications on their application and its utility:

### Programming Language

One interpreted programming language that is widely used for learning, general purposes, and data science is Python.

- Which **programming language** should they use to build their application and why?
- How can an application be platform independent? What does platform independent even mean?
- Are some programming languages more suitable for data science and mathematics than others? If so, what are some programming languages with strengths in those areas for Kyle and Morgan to consider?
- Once they have decided on a programming language, where do they download the tools for that language and how do they get started?

## 1.1 Why Python?

Congratulations on your decision to learn to develop software using Python! If you are looking for a career in **software development**, you could not have chosen a better path. Software development careers have, for the last decade, consistently found themselves in rankings of the best career options worldwide by analysts and economists. In 2019, *US*

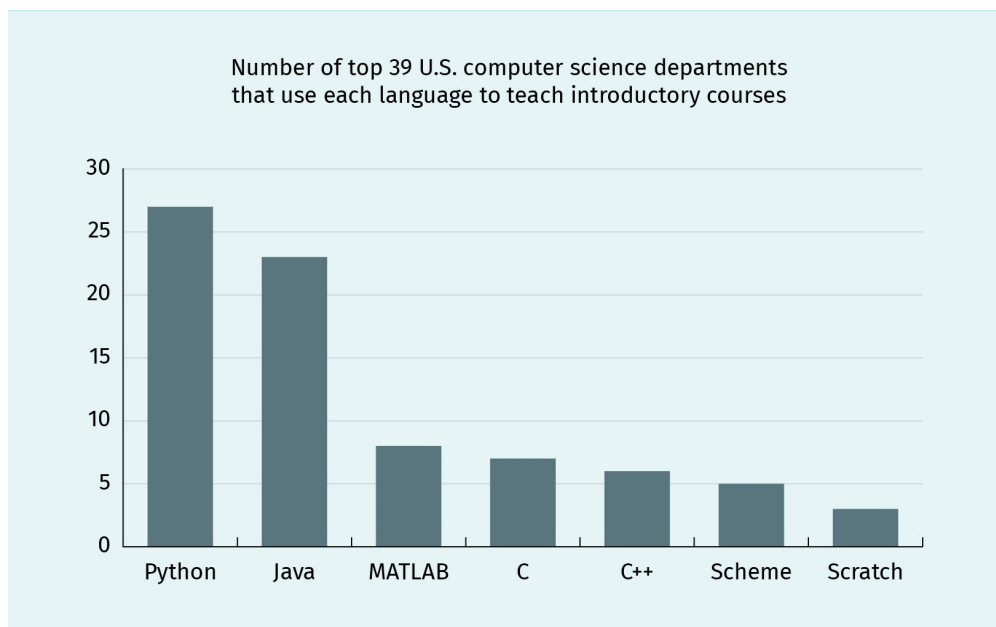
*News & World Report* ranked software developer as the number one career option based on the number of job openings, salary, employment rate, and projected industry growth (US News & World Report, 2019).

Even if a career in computer science does not interest you, successful tech leaders like Bill Gates and Steve Jobs have claimed that having software development skills can make people more valuable to their employers in whatever career they pursue.

Python is a great program for learning software development, particularly if you have never programmed before. Python is widely used as a teaching tool for those learning to program for the first time. In fact, in 2014, Python surpassed Java as the language most used in introductory computer science courses in United States universities (Guo, 2014).

**Software development**  
The process of conceptualizing, designing, programming, and testing to create software applications or components is known as software development.

**Figure 1: Introductory Programming Languages Used in Top 39 U.S. Universities**



Source: Guo, 2014.

The PYPL Popularity of Programming Languages Index ranks programming languages based on how often language tutorials are searched on Google. It is a good indicator of how many people are trying to learn the various programming languages. As of August 2019, Python was ranked first on the index, ahead of Java (second), JavaScript (third), and C# (fourth) (PYPL Index, 2019). Along with the ranking, PYPL provides a trend, comparing each language's ranking to that of a year prior. Python was up 4.5 percent from August 2018 and was the only language in the top 10 that had a positive trend. This indicates that more and more people are trying to learn Python. The fact that it is the only language in the top 10 with a positive trend shows that it is not only growing, but it is taking market share from all the other top languages.

However, do not get the impression that Python is simply a teaching language because it is so popular among introductory computer science classes. Python is one of the most popular programming languages among professionals as well. The TIOBE Programming Community Index ranks software development languages based on the numbers of engineers using each language worldwide, courses teaching each language, and third-party vendors building software for each language (TIOBE, 2019). Python has consistently risen in the TIOBE Index since its inception, jumping from fourth place to third in September of 2018, behind only Java (first) and C (second). The TIOBE index also provides a trend for each language, comparing its rating to that of the previous year. As of August 2019, Python is still in third place (behind Java and C), and it had a positive change of 3.03 percent. Of the top 20 languages, the next-highest growth rate was Groovy at number thirteen, with a growth rate of 1.04 percent.

**Figure 2: TIOBE Index for August 2019**

Aug 2019	Aug 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.028%	-0.85%
2	2		C	15.154%	+0.19%
3	4	▲	Python	10.020%	+3.03%
4	3	▼	C++	6.057%	-1.41%
5	6	▲	C#	3.842%	+0.30%

Source: TIOBE, 2019.

So, what makes Python such a great language to learn? There are many different answers to that question. Some of the most common reasons why people love using Python are as follows:

- Python is free, and a variety of supporting tools, modules, and libraries are available at no cost to aspiring developers.
- Python's **syntax** is concise compared to that of many other languages, which means that you can do more with less, reducing the demand on the programmer.
- Python code is also easier to read than code written in many other languages because one of the central concepts in the creation of Python was that the code should resemble everyday English.
- Python has an active developer community that creates resources for entry-level and expert Python developers.
- Developers can use Python to build applications using procedural, object-oriented, or functional paradigms.

**Syntax**

The rules and symbols used to create programs in a programming language form the syntax of that language.

- Because Python is an **interpreted language**, developers can get immediate feedback without having to wait for an application to compile.

Knowing why Python is a great language to learn can help us to understand why it is so popular among introductory computer science courses at the university level, as well as why Python is the top language on the PYPL index. However, why is Python so popular among professionals and organizations? The reasons cited above, which contribute to Python's success as an introductory language, are also reasons why it is so popular for real-world applications. Here are some other reasons why Python is a great choice for professional and scientific development:

- Despite being relatively easy to use and read, Python is extremely robust and powerful.
- Because Python is so easy to use and is such a concise language, it is a popular choice for quickly building working prototypes.
- Python is the language of choice for **data science**, and is heavily used when working with artificial intelligence.
- Because of its popularity in data science and AI, there is a huge ecosystem of libraries available for Python development in areas such as mathematics, statistics, machine learning, and deep learning.
- There are exceptional libraries and frameworks that facilitate web development in Python (Django and Flask are great examples).
- Python is already extremely popular among educators, learners, and practitioners, and the trends show that this momentum is only increasing. That means that there are more jobs available with higher salaries to Python developers. Learning Python can help you pay the bills!

#### **Interpreted Language**

Python is called an interpreted language because code written in Python can be executed directly without needing to be precompiled into machine language instructions.

#### **Data Science**

This is a field that uses a scientific approach to the organization, representation, and analysis of data through algorithms and computation.

## 1.2 Obtaining and Installing Python

Now that we know why we might choose to learn Python and why we might want to use it in a real-world scenario, let's get started! Although you can download Python as well as find a wide range of valuable Python resources on the Python website, do not download and install Python from that location for this class. In this class, we will be preparing to use Python for data science applications, which will be developed throughout the rest of the data science program. For this reason, we want to install not just Python, but some of the Python libraries for machine learning and data science.

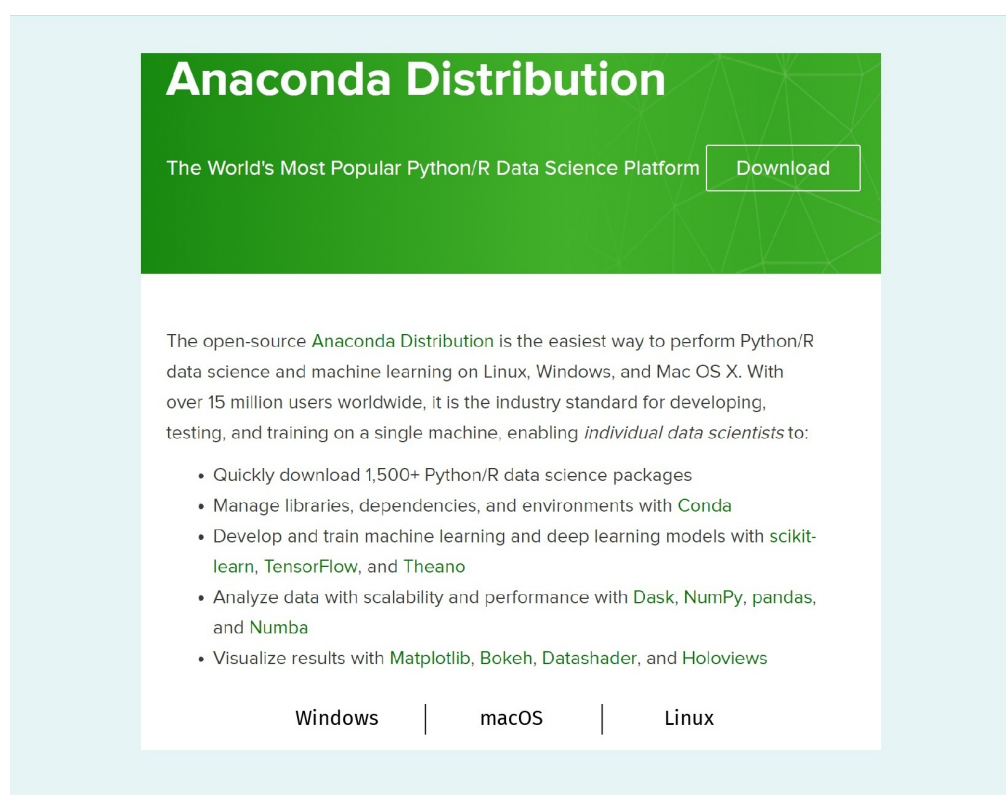
One of the most popular distributions of Python for data science and machine learning is Anaconda. The Anaconda distribution of Python includes some crucial tools you will use throughout the data science program:

- NumPy: An extremely efficient library used for the computation of large data sets and multi-dimensional arrays.
- SciPy: A library used for scientific computing including linear algebra, interpolation, and signal and image processing.
- Pandas: A library used for data manipulation in numerical tables and time series data.

Note that there are other options for working with Python in the data science space. One such option would be to install Python itself and then add whichever libraries you want, such as NumPy and SciPy, to that installation. Python(x,y) is an alternative scientific and numeric computational library. Enthought is a library for data visualization and manipulation. Because of the large and devoted open source development community supporting Python, there are numerous other options that may be worthy of consideration as well. However, for this course, we will be installing the Anaconda distribution.

Visit the Anaconda website and click the download button to find the distribution download. Ensure that your platform (e.g., Windows, MacOS, or Linux) is selected, as per the screenshot below:

**Figure 3: Anaconda - Select Your Platform**

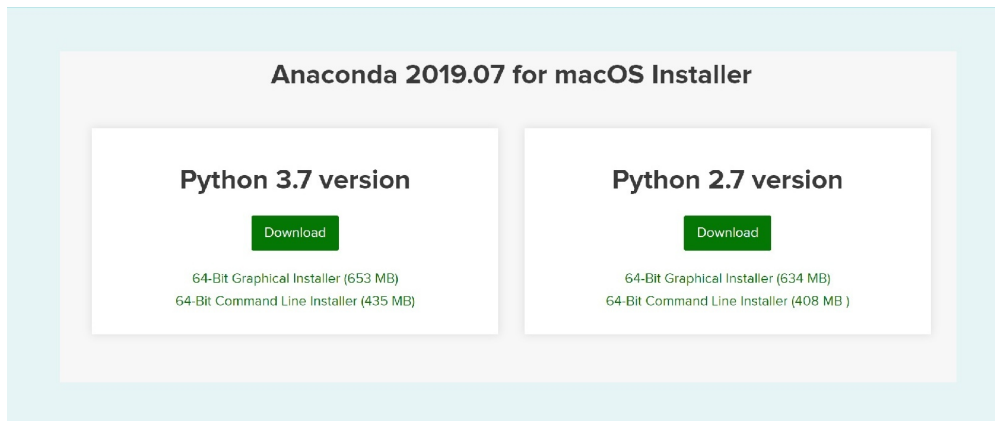


Source: Aaron Reed, 2020.

On the following screen (shown below), you will be presented with a couple of options for your Python version (as of August 2019, the options are Python 3.7 or Python 2.7):



**Figure 4: Anaconda - Select Version**

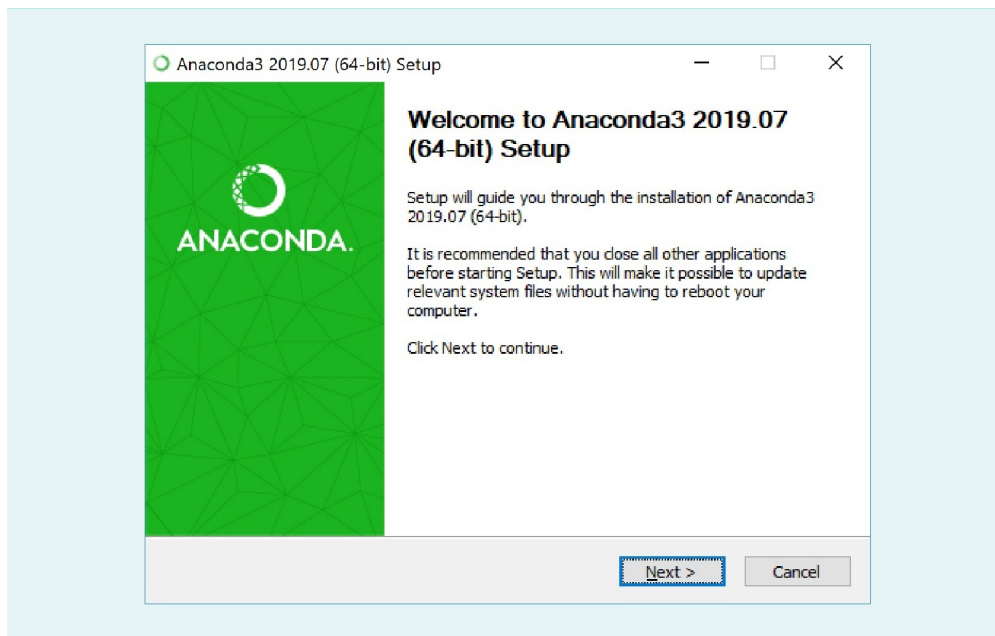


Source: Aaron Reed, 2020.

For this class and throughout the data science program, we will be using Python version 3.x, so make sure you download the Python 3.7 version and not the 2.7 version. Like most software, Python is updated regularly. Version 3 has some significant differences compared to version 2, and many of the course examples may not work correctly if you download an incorrect version.

Once you have downloaded Anaconda, run the install executable. This section will walk you through the installation on a Windows machine. MacOS and Linux installations will be similar. Hit "Next" at the Setup screen:

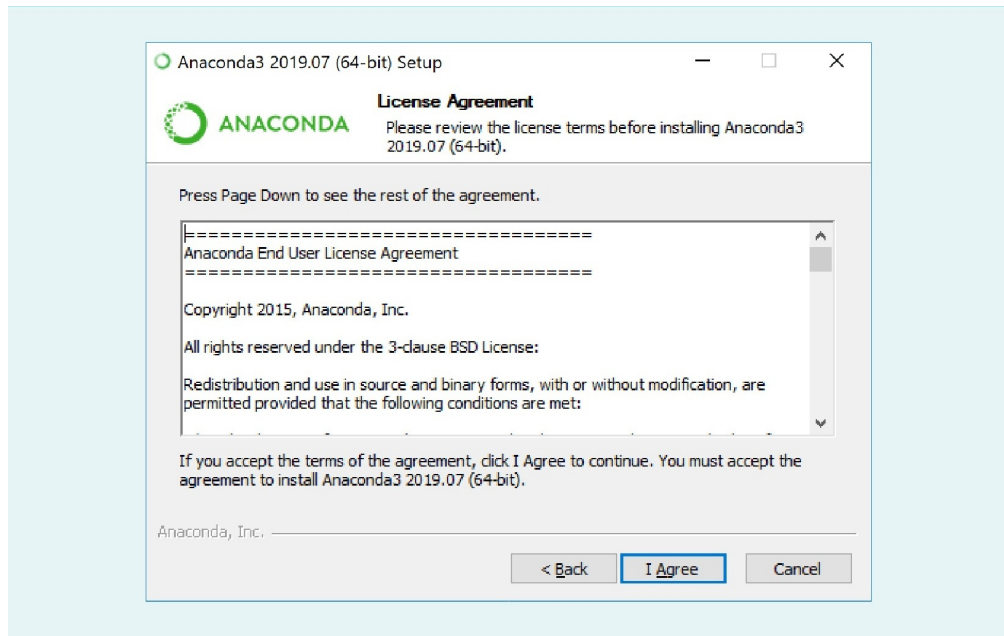
**Figure 5: Anaconda Install - Setup**



Source: Aaron Reed, 2020.

On the next screen, accept the license agreement (after reading it in full, of course):

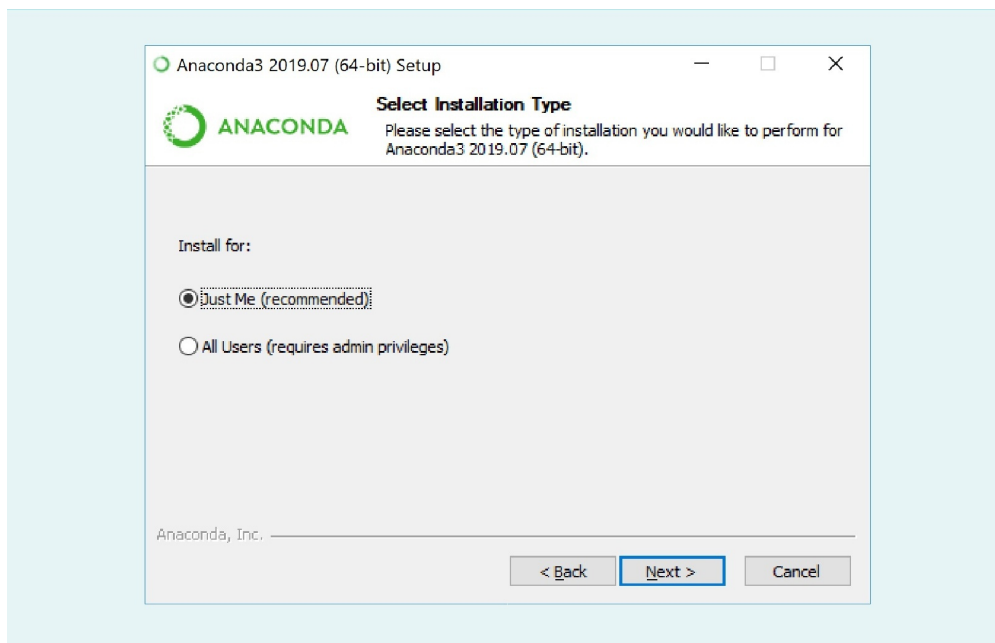
**Figure 6: Anaconda Install - License Agreement**



Source: Aaron Reed, 2020.

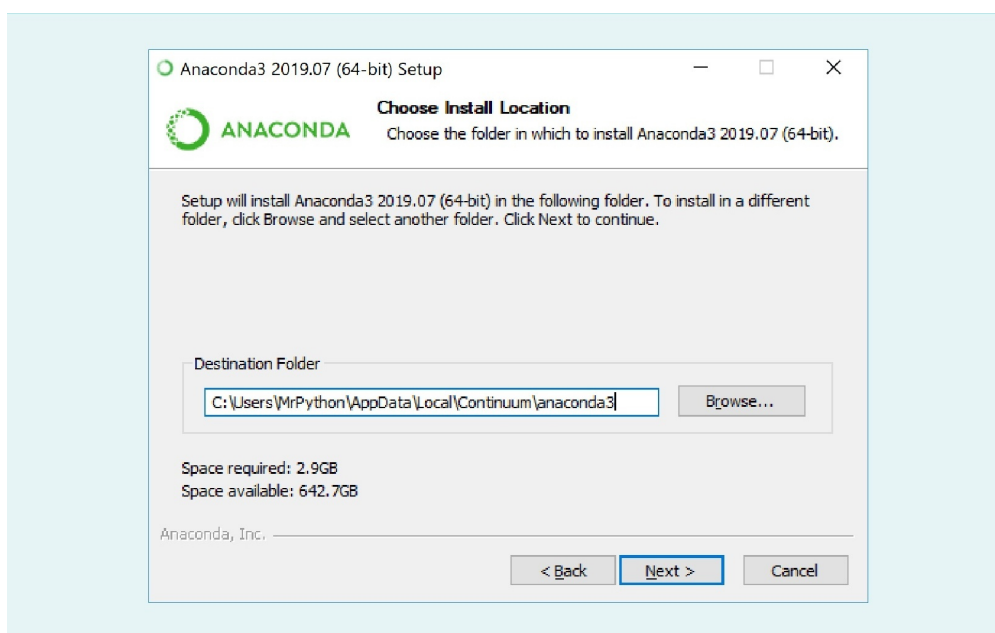
Next, select the permissions for your machine (install it only on your account or make it available to all users):

**Figure 7: Anaconda Install - Installation Type**



Source: Aaron Reed, 2020.

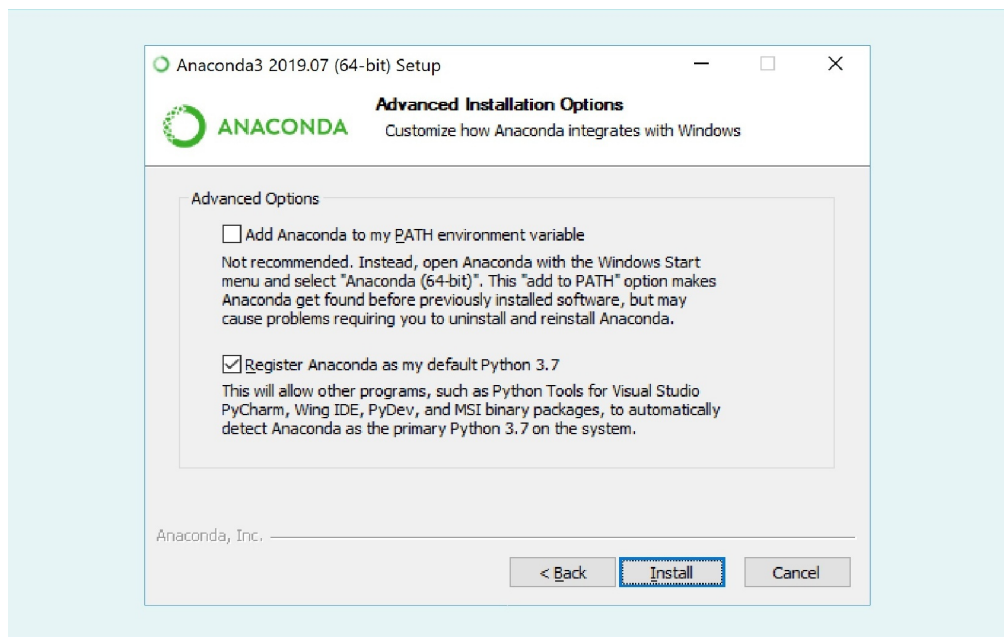
**Figure 8: Anaconda Install - Location**



Source: Aaron Reed, 2020.

On the next screen, accept the default options, and then click the Install button:

Figure 9: Anaconda Install - Advanced Options



Source: Aaron Reed, 2020.

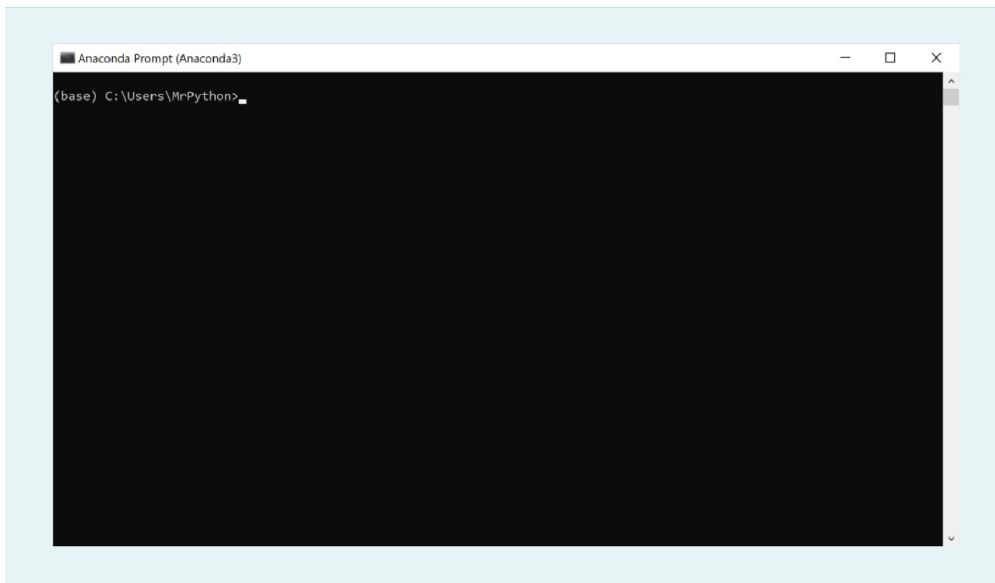
After the installation is complete, click “Next” on the remaining screens and you will have successfully installed Anaconda Python!

## 1.3 The Python Interpreter, IPython, and Jupyter

Now that we have an installation of Python running, let us create our first Python application! The traditional route is to create a “Hello, World” program, so let’s see what that would look like in Python.

To start Python through Anaconda on a Windows machine, click the start button, “Anaconda3,” and then “Anaconda Prompt.” The Anaconda prompt will be displayed:

**Figure 10: Anaconda Prompt**

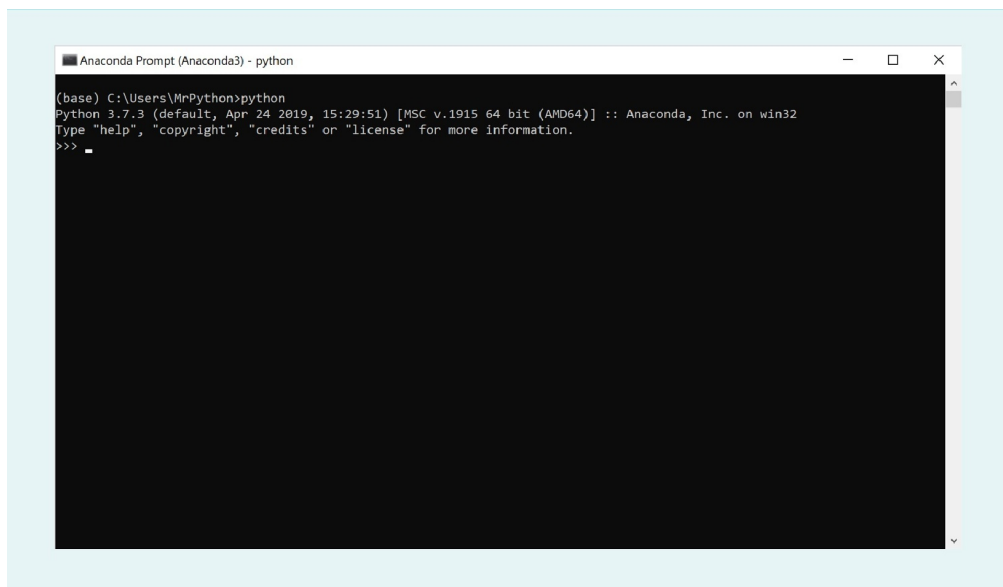


Source: Aaron Reed, 2020.

In Windows, the rest of the code will be entered through the Anaconda prompt. If you are using MacOS, open Launchpad, then click the terminal icon. On MacOS, the rest of the code will be typed into the terminal window. If you are using Linux, open a terminal window; the rest of the code will be typed into the terminal window.

Next, you need to start the Python interpreter. At the prompt, type "Python" to invoke the interpreter—we will explain what an interpreter is in a moment. For now, let's get that "Hello, World" program running. Once you've started the Python interpreter by typing "Python," you should see something similar to the following:

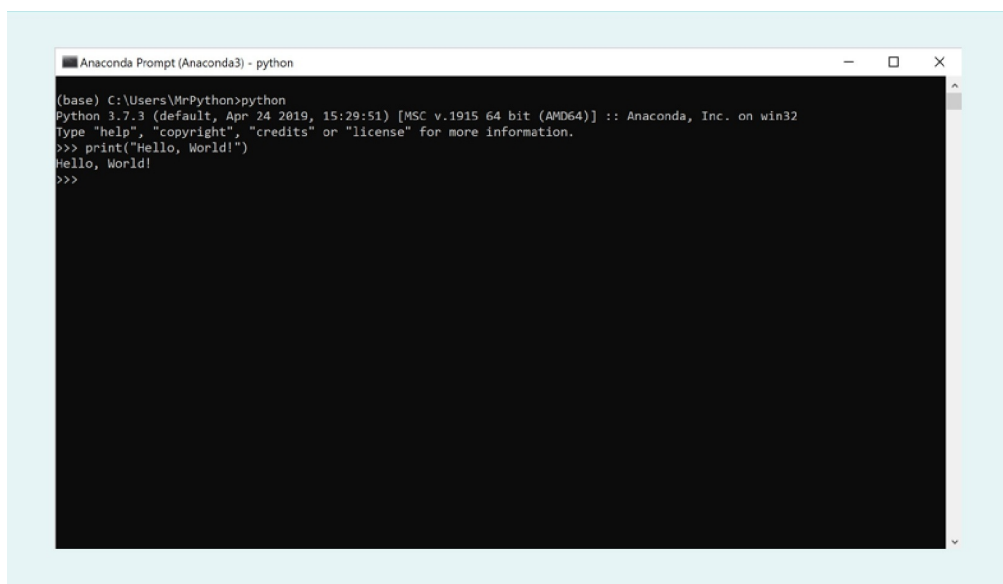
**Figure 11: Anaconda Prompt - Python Interpreter**



Source: Aaron Reed, 2020.

With the Python interpreter running in the Anaconda prompt, you can type code at the prompt, and the interpreter will execute it and display the results. For your first-ever Python application, type `print("Hello, World!")` and hit "Enter."

**Figure 12: Anaconda Prompt - Hello, World! In Python**



Source: Aaron Reed, 2020.

You have created your first Python application; the traditional Hello World is complete!

Your first Python program was literally one line of code—welcome to the power of Python! Remember, we said it was concise, meaning you can do more with less, and recall that we said it reads more like English than other programming languages. So, yes! That’s it! For comparison, let’s look at what it would take to build a Hello World application in Java and C, the two languages that are ahead of Python in the TIOBE index:

**Figure 13: Hello World in Java**

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Source: Aaron Reed, 2020.

**Figure 14: Hello World in C**

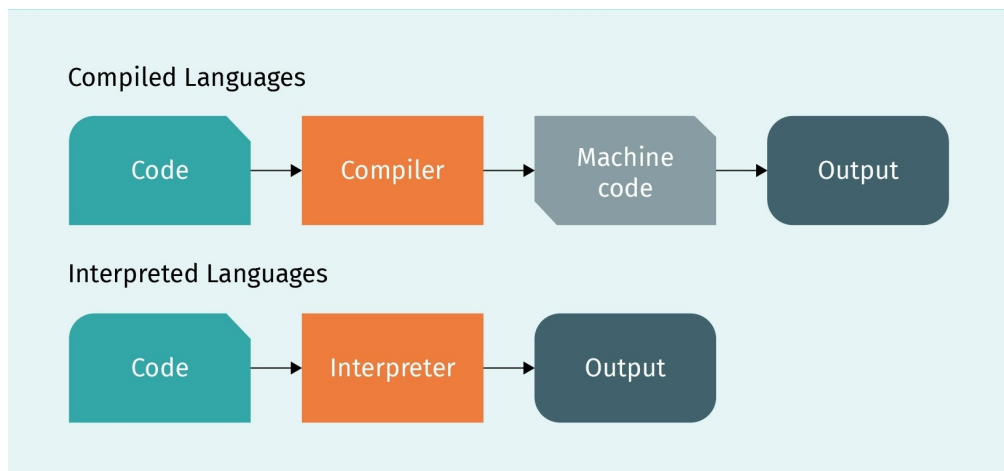
```
#include <stdio.h>  
  
int main() {  
    printf("Hello World!\n");  
}
```

Source: Aaron Reed, 2020.

As you can see, the Python version of Hello World is simpler, easier to read and understand, easier to code, and faster to create. Those are all principles that were fundamental in the conceptualization and creation of Python. No wonder Python is so popular and continues to gain fans worldwide.

So, how does this work? Well, some programming languages use a compiler. The code is sent to the compiler, which translates the entire program into code that is compatible with the target or host machine. The target machine can then execute the code and return the output. In contrast, interpreted languages load an interpreter and return the output without creating executable machine code as an independent artifact. The diagram below illustrates the difference between compiled and interpreted languages.

**Figure 15: Compiled and Interpreted Languages**



Source: Aaron Reed, 2020.

In Python, the interpreter reads the code, translates the code into something called “byte code,” and then executes it in a Python Virtual Machine, returning the output to the user. In our Hello World application, the command `print("Hello, World!")` was read by the interpreter, which first checks to see if it is reading a valid Python command. Once it passes that test, the command is translated into byte code and executed, outputting the result to the screen: Hello, World!

### **IPython and Jupyter**

An alternative to the standard Python command line terminal that has been used in the examples above is IPython. IPython extends the Python terminal into an interactive shell environment. IPython provides support within the terminal for most Linux commands, such as `ls`, `cd`, and more. Moreover, it provides a command history, auto-completion of python commands and many more features. For a full list of features and commands, see the IPython documentation.

To launch IPython, open up the Anaconda Prompt and type “IPython” instead of “Python” and the IPython shell will be invoked. Other key features of IPython include the following:

- Python script debugging,
- the ability to access help by typing a question mark (?) next to commands or object names in the code,
- enhanced feedback from the terminal with indicators for input and output line numbers,
- tab completion in the command line to finish partially typed commands, and
- access to a variety of other tools.



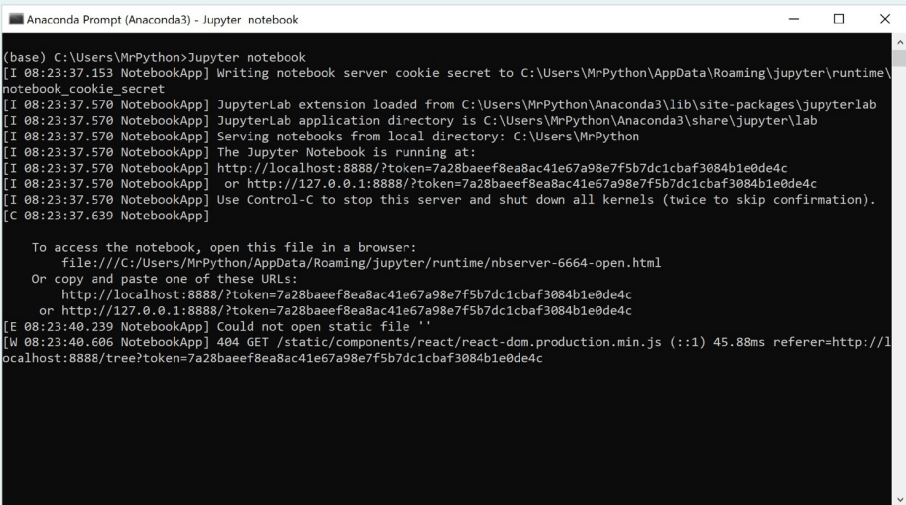
One of the most interesting features of IPython is the IPython notebook, which allows developers to combine rich-formatted text, inline code, mathematical formulas, plots and graphs, and other media into a single document. In 2014, a spinoff to IPython’s notebook called Jupyter was created. Jupyter uses IPython as the underlying Python **kernel** but continues to evolve the notebook under the new name Jupyter.

#### Kernel

A Jupyter kernel is a program that processes the requests from the notebook interface. The notebook and the kernel communicate via the ZeroMQ protocol.

Let’s take a look at Jupyter Notebook. First, pull up an Anaconda Prompt. At the prompt, type “jupyter notebook” and hit “enter.” Your console will start a Jupyter server and will output some text indicating that the server is running. It should look something like this:

Figure 16: Jupyter Notebook — Server



```

Anaconda Prompt (Anaconda3) - Jupyter notebook
(base) C:\Users\MrPython>jupyter notebook
[I 08:23:37.153 NotebookApp] Writing notebook server cookie secret to C:\Users\MrPython\AppData\Roaming\jupyter\runtime\notebook_cookie_secret
[I 08:23:37.570 NotebookApp] JupyterLab extension loaded from C:\Users\MrPython\Anaconda3\lib\site-packages\jupyterlab
[I 08:23:37.570 NotebookApp] JupyterLab application directory is C:\Users\MrPython\Anaconda3\share\jupyter\lab
[I 08:23:37.570 NotebookApp] Serving notebooks from local directory: C:\Users\MrPython
[I 08:23:37.570 NotebookApp] The Jupyter Notebook is running at:
[I 08:23:37.570 NotebookApp] http://localhost:8888/?token=7a28baeef8ea8ac41e67a98e7f5b7dc1cbaf3084b1e0de4c
[I 08:23:37.570 NotebookApp] or http://127.0.0.1:8888/?token=7a28baeef8ea8ac41e67a98e7f5b7dc1cbaf3084b1e0de4c
[I 08:23:37.570 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 08:23:37.639 NotebookApp]

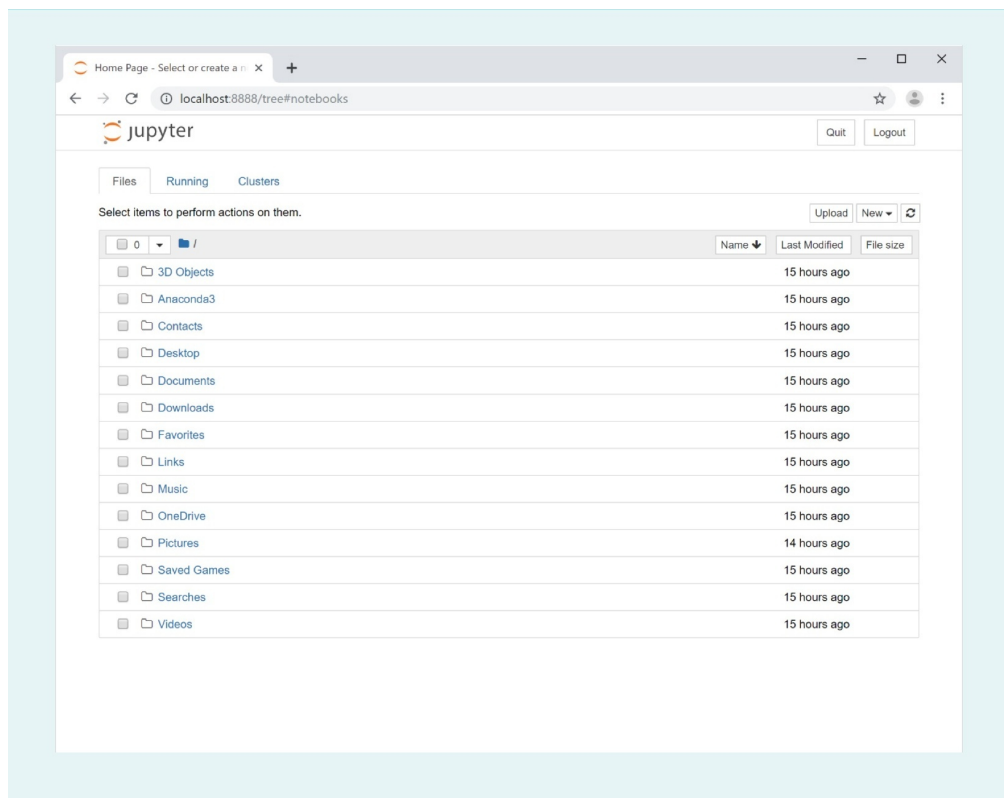
To access the notebook, open this file in a browser:
file:///C:/Users/MrPython/AppData/Roaming/jupyter/runtime/nbserver-6664-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=7a28baeef8ea8ac41e67a98e7f5b7dc1cbaf3084b1e0de4c
or http://127.0.0.1:8888/?token=7a28baeef8ea8ac41e67a98e7f5b7dc1cbaf3084b1e0de4c
[E 08:23:40.239 NotebookApp] Could not open static file ''
[W 08:23:40.606 NotebookApp] 404 GET /static/components/react/react-dom.production.min.js (::1) 45.88ms referer=http://localhost:8888/tree?token=7a28baeef8ea8ac41e67a98e7f5b7dc1cbaf3084b1e0de4c

```

Source: Aaron Reed, 2020.

In a different window, a web browser should open to the Jupyter notebook page, allowing you to select or create a notebook. It should look something like this:

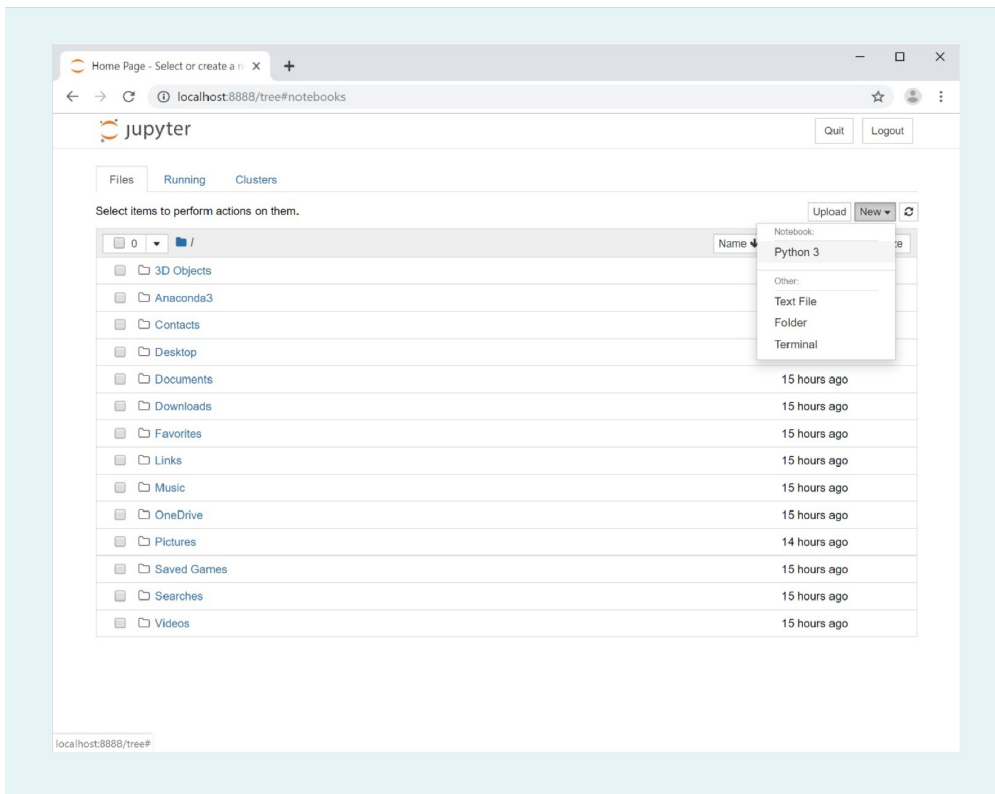
Figure 17: Jupyter Notebook — Select or Create a Notebook



Source: Aaron Reed, 2020.

This should look somewhat familiar; it's a directory listing with a bunch of shortcut directories. The root directory in this listing is the root directory for your Jupyter install on your local computer. Since we don't have any Jupyter notebooks to open, we will create one. Do so by selecting "New" and "Python3."

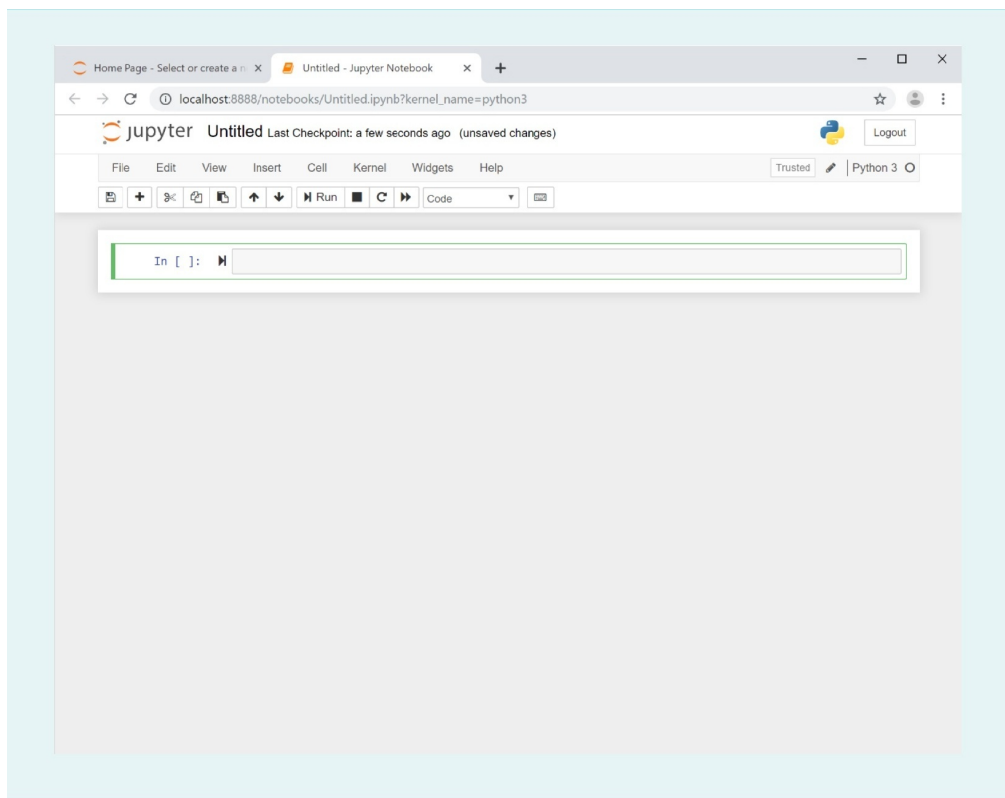
**Figure 18: Jupyter Notebook – New Python 3**



Source: Aaron Reed, 2020.

This will create a new Python 3 Jupyter notebook. The resulting window will look something like this:

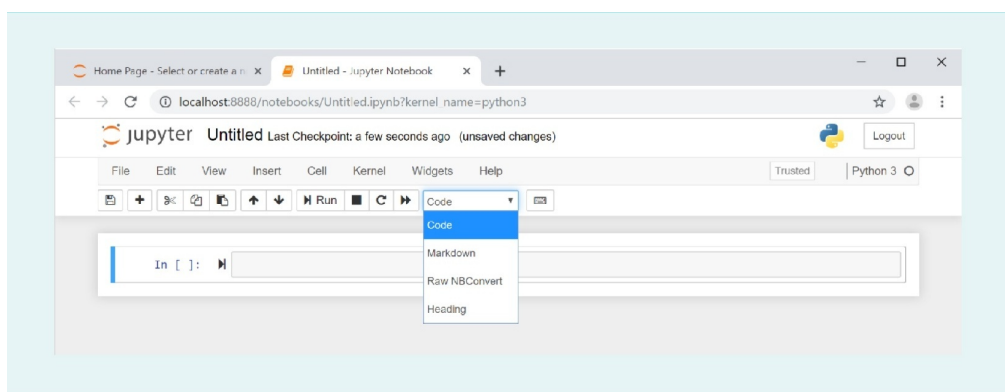
**Figure 19: Jupyter Notebook — Blank Notebook**



Source: Aaron Reed, 2020.

These powerful notebooks allow you to create powerful documents, mixing rich text and code in the same document. You add content in sections called “cells,” and for each cell, you select its type of content. When a cell is selected and highlighted, you can change the content type by using the dropdown menu in the tool bar as seen here:

**Figure 20: Jupyter Notebook - Cell Options**



Source: Aaron Reed, 2020.

So, let's work on our first notebook. Because we did such a great job in our first Python application, it seems fitting to create a page dedicated to our Hello World program. The first cell of our document, which is currently blank, is selected. This is denoted by the box around it with the blue bar on the left border of the box. From the dropdown menu, select **"Markdown"**, which will set the content type of that cell to Markdown. In Markdown cells, we can use Markdown syntax to format our text in a variety of ways. One such way is to mark content as headers, which will make the text larger and bolder. You create a header in Markdown by prefacing text with the hashtag symbol (#). One hashtag is a level-1 header, two hashtags is a level-2 header, and so on. Remember to put a space after the hashtag, or Markdown will think your hashtag is just part of whatever word you're typing. Let's create a header by entering the following text into the cell:

**Markdown**  
A popular markup language with simple syntax for formatting plain text.

#### Code

```
# My Hello World Page
```

Now, let's create a new cell below the current cell. Do so by clicking the Insert Cell button on the toolbar (the button with the plus symbol). Make sure that this new cell is set to use Markdown as well. In this cell, let us add a little more context to our page. To do so, we need to know a couple of other Markdown tricks. First, to make text bold, surround it by double asterisks (\*\*). Second, to create a hyperlink, put the title of the link in brackets, followed by the link itself in parentheses like this: [title](link). Enter the following text in the new cell:

#### Code

```
Welcome to my **Hello World** page. To learn more about Python,  
click [here](http://www.python.org).
```

Let us take a second to figure out what that text is going to do. Based on our newfound knowledge of Markdown, we assume that the Hello World text surrounded by double asterisks will be bolded. The second sentence text should also read "To learn more about Python, click here" with the word here being a hyperlink that will take us to the Python home page. That sounds about right. Now, let's add one more cell by clicking the Insert Cell button again. This time, let's set the cell content type to code. Do so by selecting "Code" from the dropdown menu in the toolbar when the new cell is selected. In a code cell, you can type any Python code you want. Let's add our Hello World code to that cell by typing in the following:

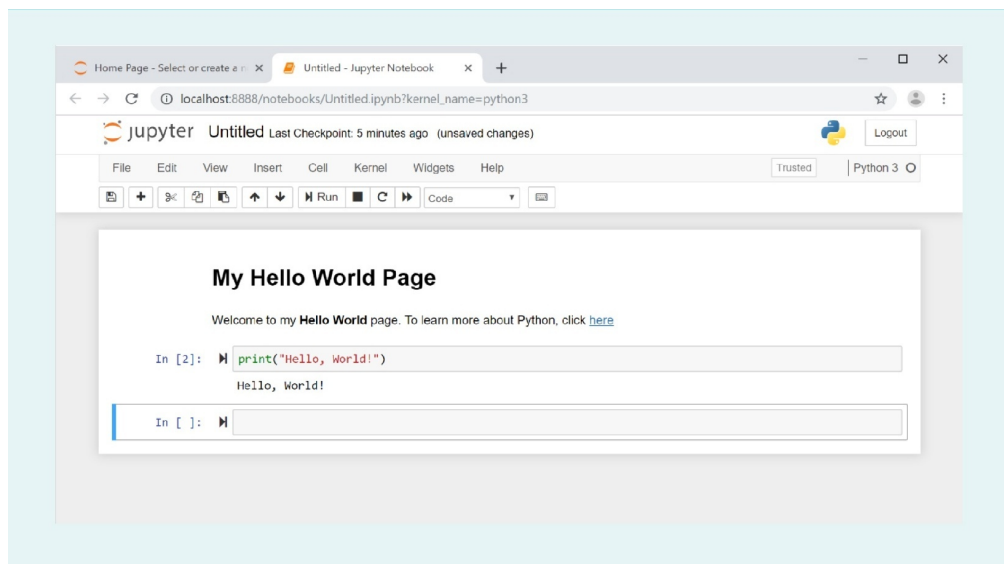
#### Code

```
print("Hello, World!")
```

Now we're ready to run the cells in the document. Running a cell will simply execute whatever is in that cell. If you have a Markdown cell, the Markdown parser will process the text and format it accordingly. Likewise, in a code cell, the Python interpreter will process the code and execute it just as if we were writing the code in the Python Prompt. The code will be executed exactly as it would be in the IPython Prompt because, if you recall, Jupyter uses IPython as the interpreter. That just means you get to use all the features included in IPython with Jupyter.

You can run an individual cell by selecting that cell and then clicking the Run button on the toolbar. Alternatively, when you have multiple cells that need to be run, you can select the Cell menu above the toolbar and select “Run All” from that menu. This will execute the Markdown and code in every cell in the document. Your resulting page should look something like this:

**Figure 21: Jupyter Notebook — Hello World Page Executed**



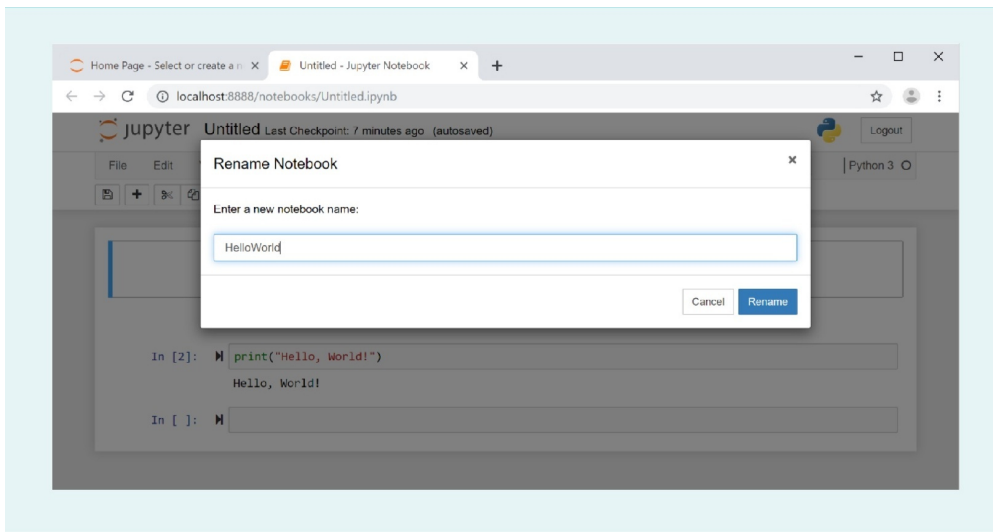
Source: Aaron Reed, 2020.

Congratulations! You have just created your first Jupyter notebook with rich text and code embedded together! A couple of things to note here:

- The formatting of the Markdown cells worked exactly as we anticipated. If your screen doesn't look like the image above, double-check your hashtags and asterisks and make sure you put a space after the hashtags.
- The hyperlink works—it should take you directly to the Python homepage.
- The code looks very similar to what we see in the IPython prompt with input line numbers and output to the console below the executed code.
- Finally, how easy it is to create documents that combine rich text and Python code!

There's one last thing we need to do. We never named our document, so it's currently being auto-saved as Untitled. Since this is an important page for us, we should name it something better. Note the word Untitled at the very top of the document, next to the Jupyter logo. That is our document name. Click that word and rename it to “HelloWorld.”

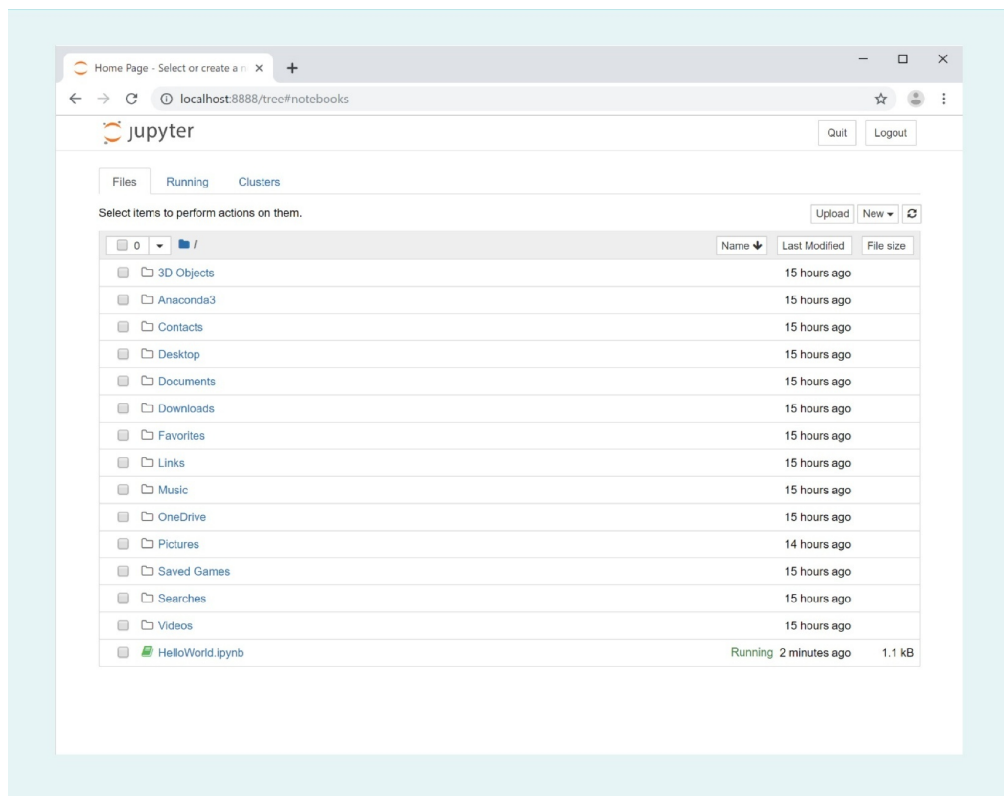
**Figure 22: Jupyter Notebook - Hello World Page Renaming**



Source: Aaron Reed, 2020.

Now we can go ahead and close the document by closing that tab in our browser. Don't worry about the content being saved because Jupyter auto-saves the content for us as long as we're not in the middle of a cell edit. Once we're back at the directory listing page in Jupyter, we can see that our Hello World document was saved as "HelloWorld.ipynb." The IPYNB file extension tells us that it's a notebook file (NB) that uses IPython (IPY). You can continue to create new notebooks the way we created the Hello World notebook, or you can edit an existing notebook by selecting it from this page.

Figure 23: Jupyter Notebook - Select or Create with Hello World Page



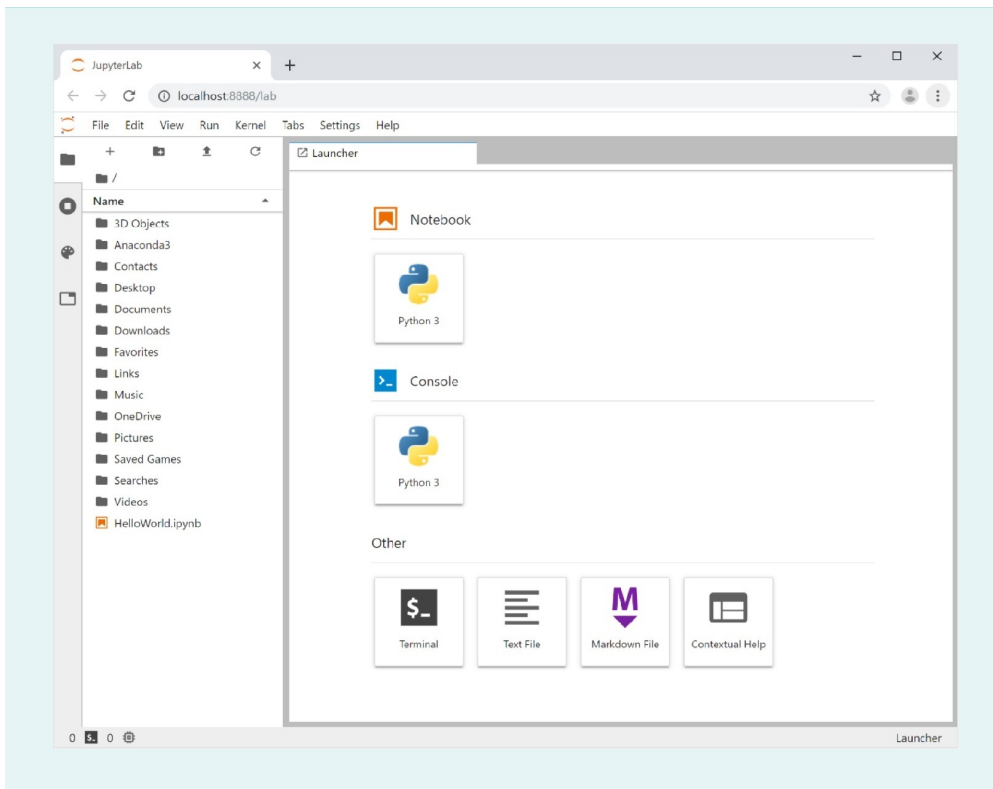
Source: Aaron Reed, 2020.

## Jupyter Labs

As mentioned previously, the Jupyter Project, which branched off from IPython, continues to evolve. The latest version of the Jupyter Project is available as JupyterLab. JupyterLab includes the Jupyter Notebook discussed in the previous section as well as some other very useful tools for Python development. You can launch JupyterLab by opening an Anaconda prompt, typing “jupyter lab,” and hitting “enter.” Just as it did with Jupyter Notebook, the console will display output indicating that a server session has started, and a web browser will open displaying the JupyterLab home page, which should look something like this:



Figure 24: JupyterLab - Home Page

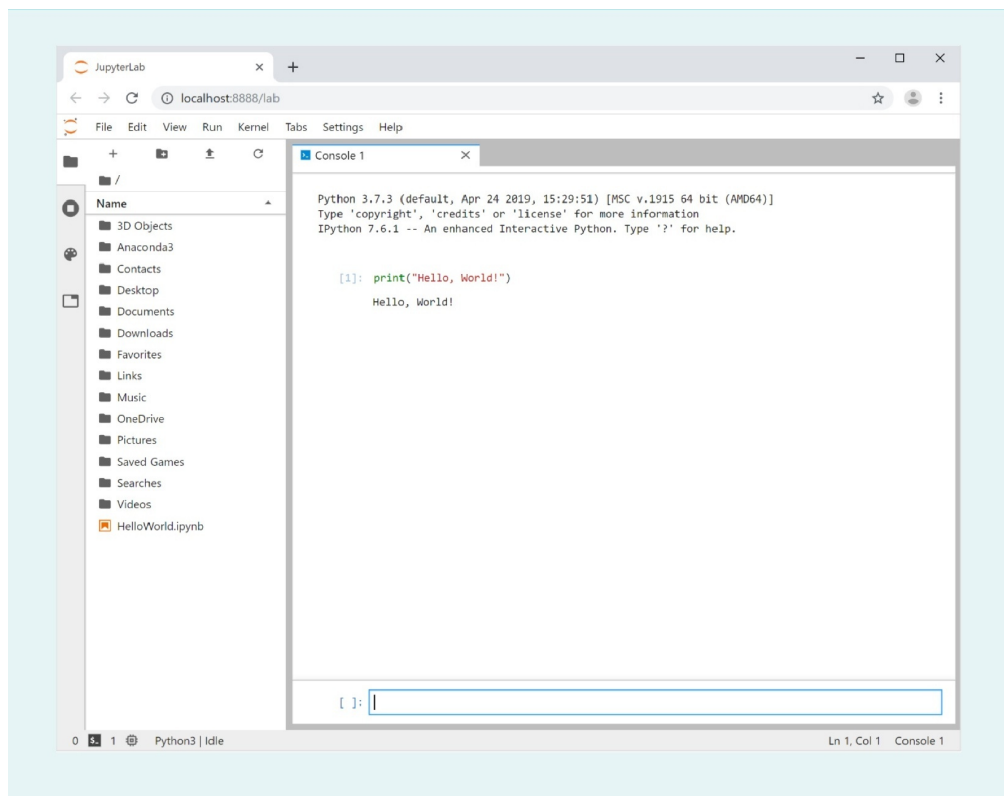


Source: Aaron Reed, 2020.

As you can see in the JupyterLab home screen, we have an interface similar to that of Jupyter Notebook with a directory listing on the left-hand side. You can see our HelloWorld.ipynb notebook in that view, which tells us that this is the same home directory as the one used in Jupyter Notebook. However, you have a few new options on the right-hand side of the screen. The Python 3 Notebook option will create a new notebook using IPython 3 as the underlying interpreter, just like the one we created in the notebook example. There are a few new options available that were not in the standard Jupyter Notebook version, but nothing to be preoccupied with at this time. The biggest change you'll note is that the menu item to run all cells in a notebook has changed in JupyterLab and is now "Run -> Run All Cells." Additionally, from the file menu you can export an open notebook to a variety of formats that you may find beneficial going forward.

The Python 3 Console option on the home page of JupyterLab will open up a console window just like the IPython prompt discussed previously. You can see our Hello World application in the Python 3 Console below:

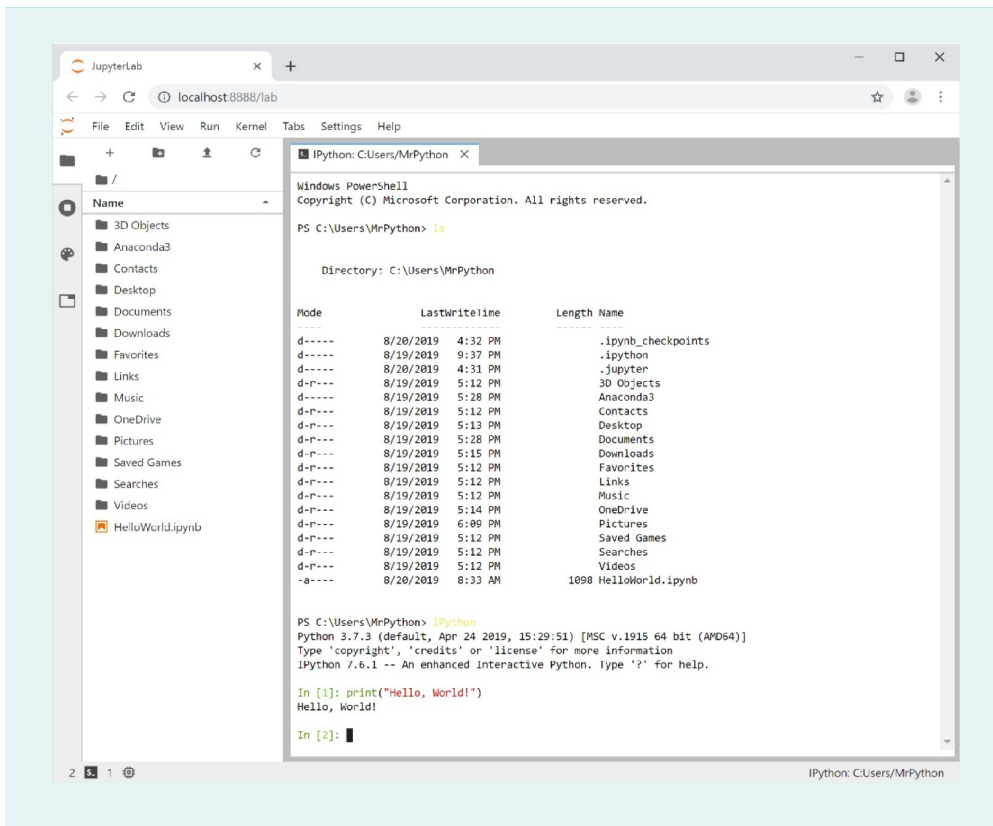
Figure 25: JupyterLab - Python 3 Console



Source: Aaron Reed, 2020.

The Terminal option in the JupyterLab home page opens up a terminal window in the current directory. Below, you can see the open terminal window, the current directory listing being displayed after an ls command, the open IPython prompt, and once again, our beloved Hello World example.

Figure 26: JupyterLab - Terminal



Source: Aaron Reed, 2020.

The other options on the JupyterLab home page let you create other files for raw text and Markdown as well as contextual help that lets you click on various functions and objects in Python code to get helpful information on those objects.

JupyterLab is an integrated development environment meant to facilitate robust development of large projects in Python. Yet, like Python in general, it is easy to use so it makes for a great environment in which to learn and develop your Python skills.

You're now equipped with the tools you need to begin programming in Python! In the coming chapters, we'll move beyond Hello World and into the essentials of Python development as we start building more dynamic and powerful Python code.



## SUMMARY

Software development is one of the most lucrative and rewarding careers available today, recently being ranked as the top career for 2019 by *US News and World Report*. Worldwide, software developers have bountiful job opportunities and high salaries. More and more of the

world is moving toward Python as the language of choice for learning software development principles, in part because of its easy-to-read, concise syntax.

But it's not just educators and learners who are using Python. Python has quickly been climbing the ranks of the most popular languages in the world among professionals. Python is easy to use, robust, and powerful, has an amazing array of freely available tools, and is supported by a large and active community of developers.

Python is also the top programming language in the fields of data science and artificial intelligence. The Python community has created some powerful and extremely efficient libraries for use in areas such as mathematics, statistics, data organization and manipulation, data visualization, machine learning, and deep learning. These tools and libraries make Python a top choice for data science development worldwide.

Python is an interpreted language, meaning that the Python interpreter reads code, executes it, and returns the output to the user. IPython offers some advantages over the standard Python installation which can improve development. Jupyter Notebook and JupyterLab evolved from IPython and, with IPython as the underlying interpreter, provide a rich development environment where Python developers can work with multiple files, combining code, Markup, and text along with terminal windows and other tools.

# UNIT 2

## VARIABLES AND DATA TYPES

### STUDY GOALS

On completion of this unit, you will have learned ...

- how to use variables in Python and how to assign them different values.
- how to work with various numerical data types.
- how to use string and character data types.
- how to store and work with collections of data.
- how to perform basic file input/output operations.

## 2. VARIABLES AND DATA TYPES

### Case Study

Kyle and Morgan have decided on a programming language for their soccer analytics and player improvement application: Python. The two of them appreciate Python's ease of use and feel that it would help them rapidly get to a prototype stage, maybe faster than other languages. Yet they also feel that because Python is so powerful, they would be able to finish the project in Python rather than having to start over with another program after the prototype was finished. Being somewhat new to Python, they feel the wide range of resources available in the Python community would also be of benefit in this project.

Most importantly for Kyle and Morgan, they feel that the libraries available in Python for data manipulation, mathematical operations, data visualization, artificial intelligence, and machine learning would be invaluable in a project like this. They know that they will be collecting massive amounts of data about soccer players. Game conditions, running speed, and all other measurable characteristics will be stored in the application. Then, the application will have to analyze that data in order to help players and coaches identify areas for improvement. That's going to require some serious computational power, and Kyle and Morgan feel that Python would be the perfect language to take on that challenge.

But now, as the two of them start to think about this project, they realize they need some development help. They have asked you to join the project and to start working on a module to store and display player data. You have some early questions about how to store this type of data in Python:

- How can you store data such as player name, height, weight, speed, and age in a Python application?
- How will you store the data for many players at once?
- Once the data is entered into the application, how will you save that data so that it will still be there the next time the application runs?

### 2.1 Variables and Value Assignment

Your first task is to figure out a way to store player data. At the most basic level, all data is stored in something called a "variable." Think of a variable as a container, like an envelope, that holds information. For example, let's say you want to store a player's weight of 73kg. Sure, we could try to have the application somehow remember that a given player weighs 73kg and to output that weight anytime that player shows up in the system, but what happens if the player's weight changes and we need to update it? Forcing an application to always output a given value for something like a certain weight for a player is called **hard-coding**, and it is a dangerous practice. If we hard-coded the player weight at 73kg, then the next time the player changed weight we would need to edit the code—find everywhere the player was mentioned and change 73kg to the new value.

Instead of hard-coding a value such as 73kg every time we see a certain player's name, we use variables to store that data. Imagine if you had some container (again, similar to an envelope) that was called "weight" and it was assigned to a certain player. We could write 73kg on a piece of paper and put it in the player's envelope. Anytime we want to check the player's weight, we just pull out the envelope and look at what's written inside. If we need to update the weight to a new value, we simply open the envelope, throw away the existing paper and value, write a new one, and put it back in the envelope. Now the next time we check that player's weight, the new value will appear. In essence, that's the logic behind the concept of variables.

**Hard-Coding**

A value is considered hard-coded if data is fixed and cannot be changed without editing the program itself.

In Python, you create a variable by typing any name that isn't already the name of another variable you've created and also is not a **reserved word** in Python. For example, we saw in our Hello World example that the word "print" has a special meaning in Python: it prints something out to the console. Hence, print in Python is a reserved word and you cannot create a variable named print. There are a few other rules for variable naming in Python:

**Reserved word**

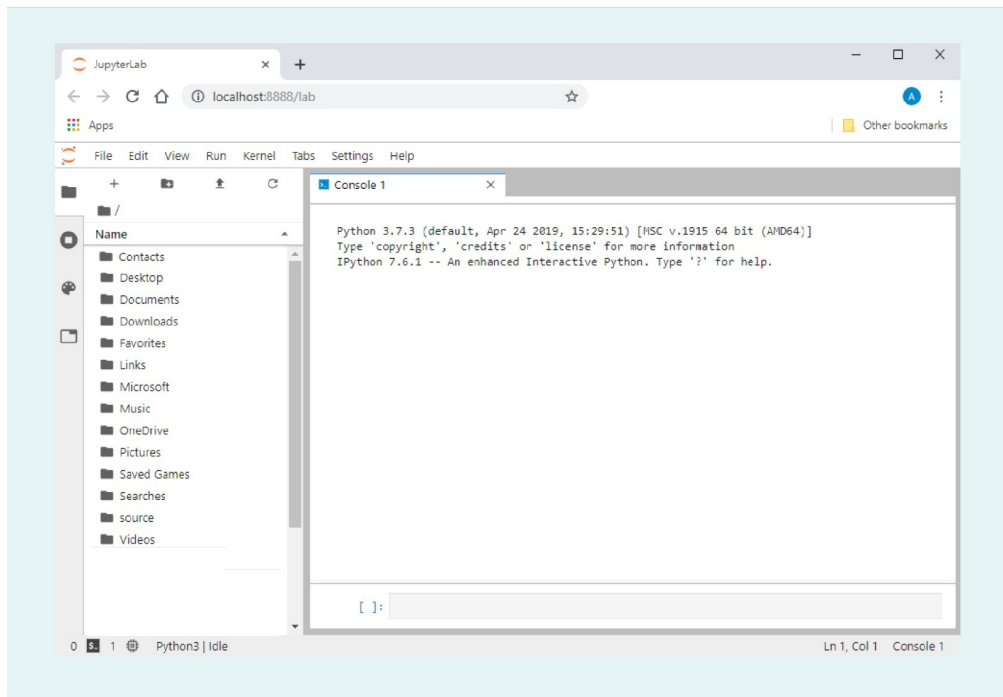
In a programming language, some words already have specific meanings and cannot be reused by programmers. These are known as reserved words.

- Variables must start with either a letter or an underscore (\_)
- After the first character, variable names can consist only of letters, numbers, and underscores.

So, for example, the variable name "weight" is a perfectly valid variable name; the name "\_weight\_" is as well. But the name "5weight\*)" is not valid.

Let's create some variables and get a feel for how things work. Open up JupyterLab by typing "jupyter lab" in an Anaconda Prompt. At the home screen in JupyterLab, open up a Python3 Console. Your screen should look something like this:

Figure 27: Blank Python 3 Console in JupyterLab

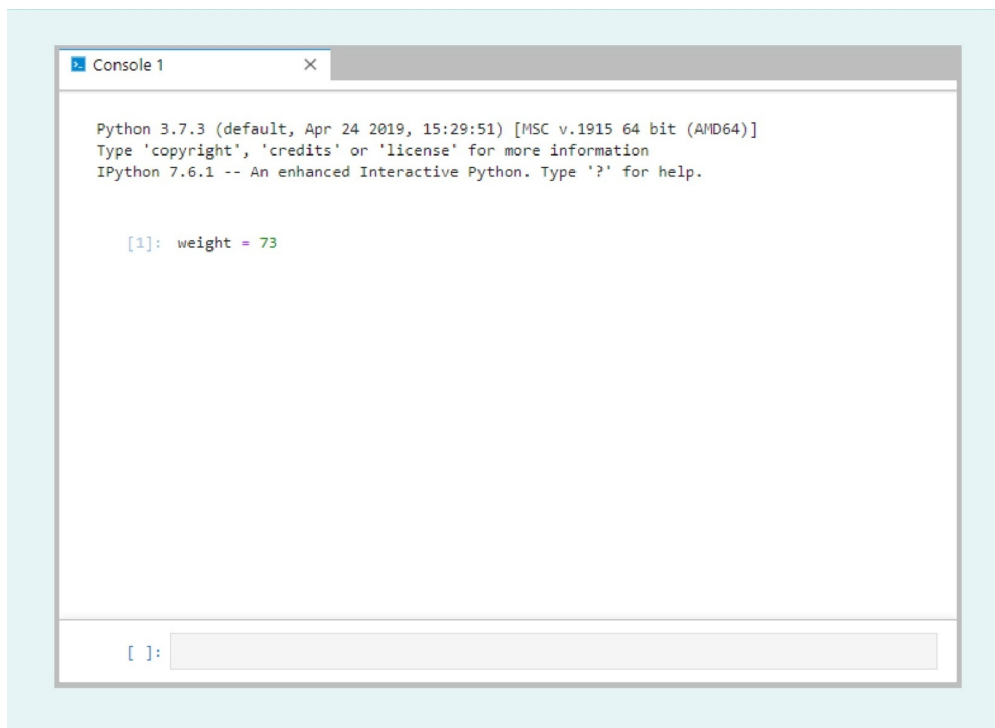


Source: Aaron Reed, 2020.

In the console, at the bottom of the window, type `weight = 73` and hit Shift/Enter to execute the code. You should now see that line of code in the top portion of the console window as shown below:



**Figure 28: Python 3 Console - weight Variable**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

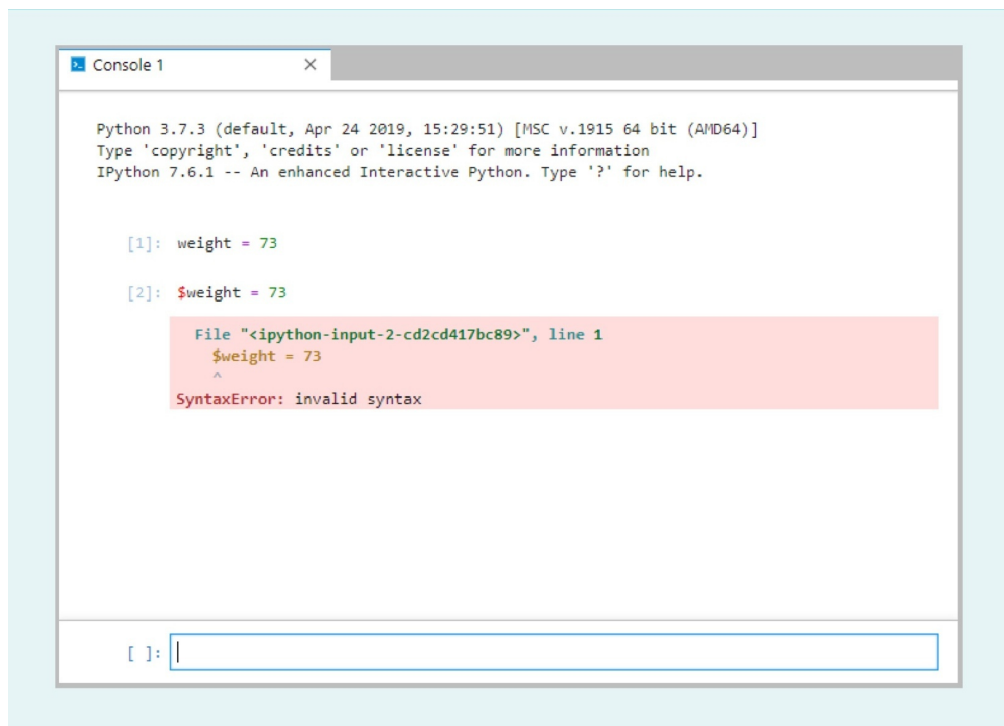
[1]: weight = 73

[ ]:
```

Source: Aaron Reed, 2020.

You've just created your first Python variable! That line of code told the Python interpreter that you want to create a variable, give it a name (`weight`), and assign it the value of `73`. You know that the interpreter read the statement as valid because there are no error messages on the screen. For example, recall the rules for naming variables: must begin with a letter or underscore, can only contain letters, numbers, and underscore characters. Let us see what would happen if we were to break one of those rules. In the console, type `$weight = 73` and hit "Shift/Enter" to execute that code. You should now see an error on your screen indicating that the code you just entered has invalid syntax:

Figure 29: Python 3 Console - Invalid Syntax



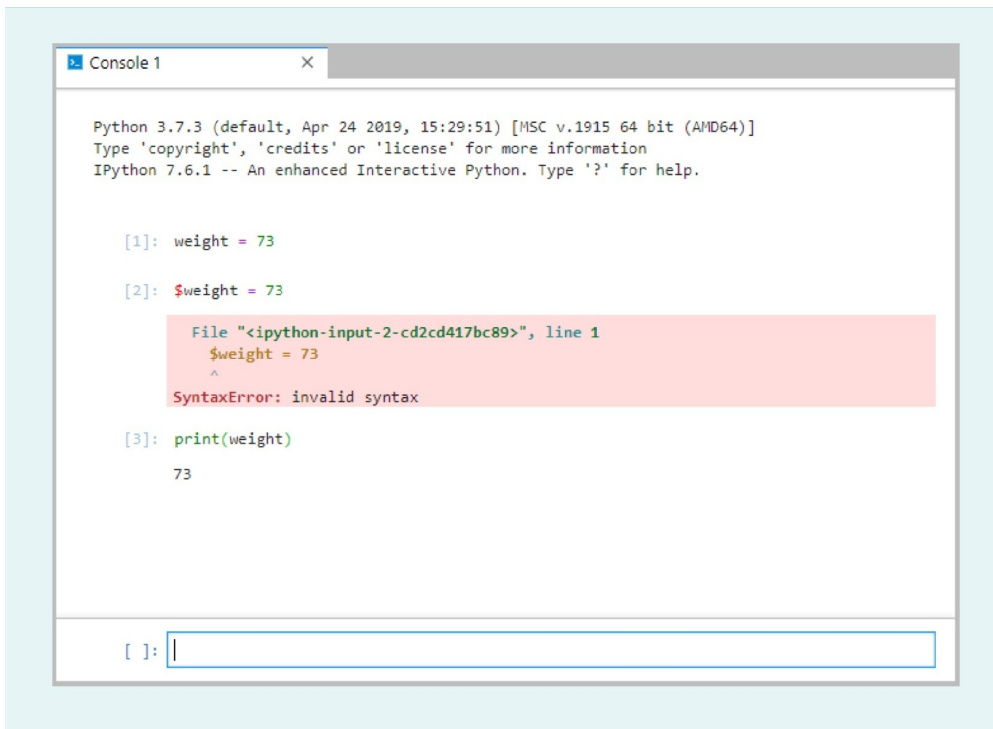
Source: Aaron Reed, 2020.

What just happened? First, recall the variable naming rules we discussed. The code we just entered broke those rules (we did not begin the name of our variable with a letter, we used a \$ symbol instead). Next, it is important to understand that when you hit “Shift/Enter” in the console window, the code you entered is sent to the Python interpreter. Finally, recall the role of the Python interpreter: it reads code, checks to verify that the code is correct, and, if it is, it executes the code and returns the output. If the code is not correct, the interpreter sends us back an error message. That is what happened here. Also recall the term “syntax” and that this term refers to the set of rules used to create code in a programming language. Since our code broke the rules of naming variables, the “invalid syntax” error makes sense.

So far in our code we have created a valid variable named `weight` and assigned it a value of 73, and we have tried unsuccessfully to create the invalidly named variable `$weight`. The console keeps your code in memory throughout a session. That means that even though our last command was to try and create the `$weight` variable, the `weight` variable is still valid and should still contain the value 73. Can you think of any way we can verify that?

If you guessed “print,” you’re right! Remember our Hello World Python code? We made the text “Hello, World!” show up on the screen using the `print` command. I wonder if that would work with our `weight` variable. Let’s give it a try. Type `print(weight)` in the console and hit Shift/Enter.

Figure 30: Python 3 Console - Value of weight



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[1]: weight = 73

[2]: $weight = 73
      ^
SyntaxError: invalid syntax

[3]: print(weight)
      73

[ ]: |
```

Source: Aaron Reed, 2020.

It worked! In this case, we have asked the print command to output weight, but the interpreter is smart enough to know that weight is a variable, so instead of printing the word “weight,” it retrieves the value stored in our weight variable and outputs that value to the screen. Nice work!

Let’s talk a little bit more about the naming of variables before we continue. When naming variables in Python, there are coding standards to consider in addition to the Python naming rules. Remember, Python is all about readability. The Python community has created a style guide, which essentially contains a list of best-practice conventions for developers to follow. The premise behind this style guide is that if all Python code were written in a similar way throughout the world, it would increase code readability. The variable naming convention for Python states that variable names should be lowercase, with words separated by underscores as necessary to improve readability. You can read more about the Python style guide by visiting the Python website.

So, as we consider the name of our weight variable, is it the most readable name we could have come up with to reflect that the variable holds a player’s weight? Probably not. The name player\_weight might be better because it better describes the purpose of our variable: to represent the weight of a player. If we follow Python’s guiding principle of readability and the style guide convention, “player\_weight” is probably the better name for our variable.

Because we have both valid (obeys the Python rules) and invalid (does not obey the Python rules) variable names, and we have both conventional (obeys they Python style guide conventions) and unconventional (does not obey the Python style guide conventions) variable names, we can divide variable names into four categories as shown below:

**Figure 31: Categories of Potential Variable Names**

		Python rules	
		Invalid	Valid
Python style guide	Unconventional	<code>\$PlayerWeight</code>	<code>PlayerWeight</code>
	Conventional		<code>player_weight</code>

Source: Aaron Reed, 2020.

We need to avoid all variable names in the invalid column because, well, they’re invalid and they will only result in an error message—as we just saw with our `$weight` variable. If a variable is invalid, it cannot be conventional because convention implies adherence to Python syntax. Hence, there are no invalid/conventional variable names. That leaves us with the valid column. Although the valid/unconventional names will satisfy the rules of Python and will not result in an error, for the sake of readability, we should stick with the valid/conventional names. That means that our variables should:

- begin with a letter or underscore (Python syntax rule),
- contain only letters, numbers, and underscores (Python syntax rule),
- be all lowercase (Python style guide convention),
- be descriptive, even if that means using a multi-word variable name (Python style guide convention), and
- have the words in a multi-word name separated with underscores (Python style guide convention).

Here are some sample variable names and where they rank regarding syntax and style:

**Table 1: Potential Variable Names**

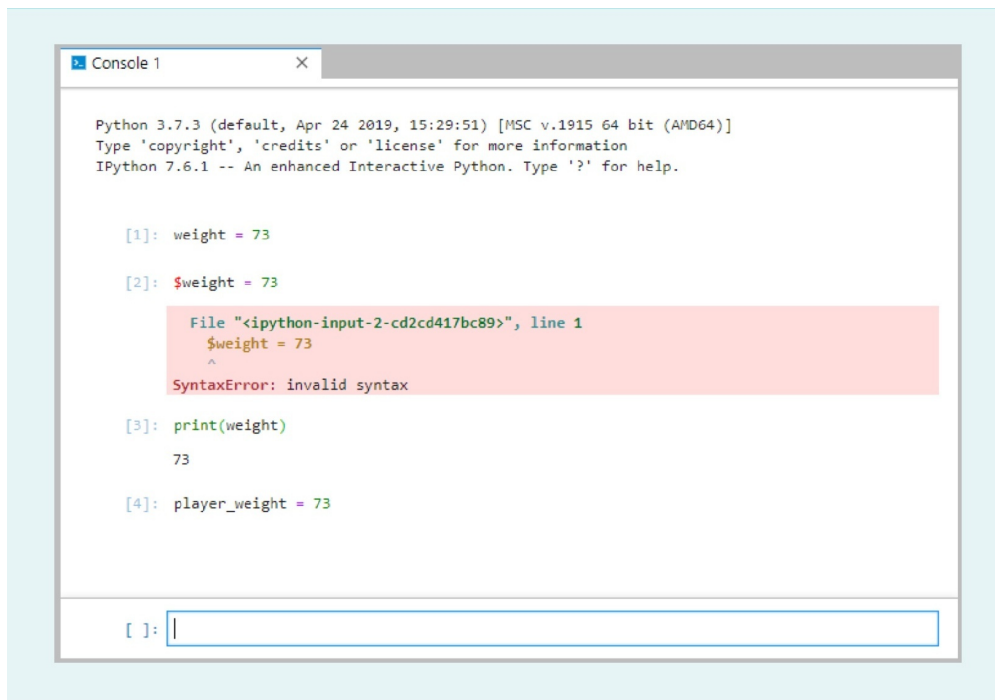
Potential Variable Name	Validity/Convention
<code>#player_age</code>	Invalid
<code>player_age</code>	Valid and Conventional
<code>_player_age_</code>	Valid and Unconventional
<code>7player_age</code>	Invalid
<code>Player_Age</code>	Valid and Unconventional

Potential Variable Name	Validity/Convention
player^age	Invalid

Source: Aaron Reed, 2020.

So, now that we know that a better name for our variable would be “player\_weight,” let’s create that variable. In the console, type `player_weight = 73` and hit “Shift/Enter.”

**Figure 32: Python 3 Console — player\_weight Created**

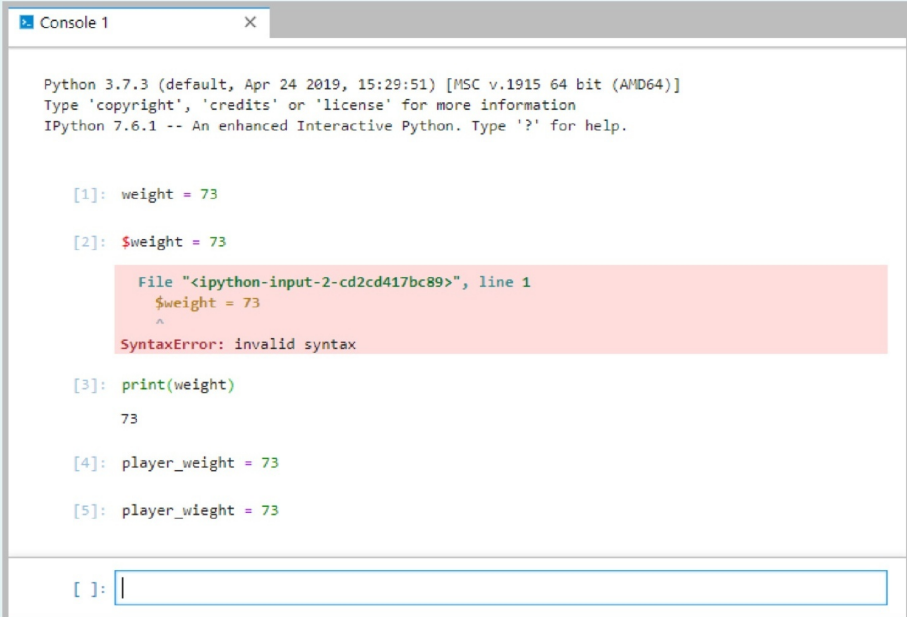


Source: Aaron Reed, 2020.

Note that we are still in the same console session as when we created the weight variable. Also note that the interpreter has no idea that we decided “player\_weight” is a better name for our variable. That is important to understand because when we created `player_weight`, it did not take the place of the `weight` variable. The `weight` variable still exists in this session, and now we have a second variable named “player\_weight.” You want to avoid having extra variables floating around, but in this case, since we’re just learning about variables, it’s not something to worry about.

But it does bring up an interesting point. What if we created an additional variable by accident? For example, type in your console `player_wieght = 73` and hit “Shift/Enter” (note that the `i` and `e` are out of order—that’s intentional to prove a point: type it “player\_wieght” instead of “player\_weight”).

Figure 33: Python 3 Console — player\_wieght Created



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[1]: weight = 73

[2]: $weight = 73
File "<ipython-input-2-cd2cd417bc89>", line 1
    $weight = 73
    ^
SyntaxError: invalid syntax

[3]: print(weight)
73

[4]: player_weight = 73

[5]: player_wieght = 73

[ ]:
```

Source: Aaron Reed, 2020.

What just happened? Well, now we have three variables: `weight` (our old variable that we aren't using anymore), `player_weight` (the new name for our player weight variable), and `player_wieght` (the intentionally misspelled player weight variable). We just created that misspelled variable on purpose, and it works just fine.

In Python, no additional step is required. So, any valid name that you put on the screen can be interpreted as a variable, and if Python hasn't seen that variable name in your code before, instead of throwing you an error, it will create a new variable for you with that name.

Let's say we are using our `player_weight` variable with a value of 73 throughout our code. And let's say at one point we want to update the `player_weight` variable so we add the code `player_wieght = 74`, accidentally misspelling the variable name this time. What would happen? We would now have two variables, one named "player\_weight" with a value of 73 and another named "player\_wieght" with a value of 74. That is problematic, because in this hypothetical case, the "weight" misspelling was an accident and we would probably not even know we had done it. When we go check the value of `player_weight` and expect it to be 74 because we thought we had just updated it, it will still be 73, and we'll be scratching our heads trying to figure out what's wrong with our code.

First of all, don't worry; scratching your head and trying to figure out what's wrong with your code is just part of programming. Second, just as Spider-Man's Uncle Ben said, "With great power comes great responsibility!" That applies to programming as well. Python, in the name of programming speed and concise language, removes the barrier of having to

create a variable before using it. It is like Python is handing you the keys to the car and asking you not to crash it. Programmers should always be aware that a computer will do exactly what they tell it to do, nothing more and nothing less—even if they tell it to do something stupid by mistake, like create a new variable called “player\_wieght” when there is already a player\_weight variable. The computer will do it without question. So just be extra cautious in Python programming to ensure that you name your variables consistently and, when using them again, you use the name you gave it, spelled exactly the same way.

OK. Before we move on, let’s dissect one more aspect of our line of code: `player_weight = 73`. What is actually happening here? Well, we know that the interpreter creates a variable named “player\_weight” and assigns it the value 73. That equal (=) sign is called an assignment operator. There are other operators as well that we will discuss later on, but for now, let’s focus on the assignment operator.

The assignment operator works right-to-left. It takes whatever is on the right-hand side of the operator (in this case, 73) and assigns that value to whatever is on the left-hand side of the operator (in this case, player\_weight). That’s important for a couple of reasons that you need to understand when you work with Python code.

First, assigning from right to left means that the reverse of that statement, `73 = player_weight`, is not valid Python code because you’d be telling the interpreter to take the value of player\_weight and assign it to 73. Even if player\_weight has a valid value, you can’t assign that value to 73. And thank goodness, because that would be a mess. Can you imagine using the number 42 in your code and having to remember that 42 actually means something else, like 6, or 991,421, or “delicious pizza?” That would really make programmers scratch their heads. Numbers, such as 73, are called “literals” in Python, meaning that they have the value that they literally and explicitly should have (in this case, 73) and they cannot be redefined with a different value. Go ahead and try assigning `73 = player_weight` and check out the error you get and see if it makes sense given what we have discussed.

The other reason that it is important to understand the right-to-left processing of the assignment operator is that the interpreter will do calculations on the right-hand side of the operator (the equals sign) until it has a single value, and then it will assign it to the item on the left of the operator. So, that means you can only have one thing on the left-hand side of the operator. But you are not limited to just one thing on the right-hand side. For example, `player_weight = 70 + 3` is perfectly legitimate. The interpreter will calculate `70 + 3`, get 73, and assign it to `player_weight`. We know from our algebra classes that

#### Code

```
player_weight = 70 + 3
```

is the same as

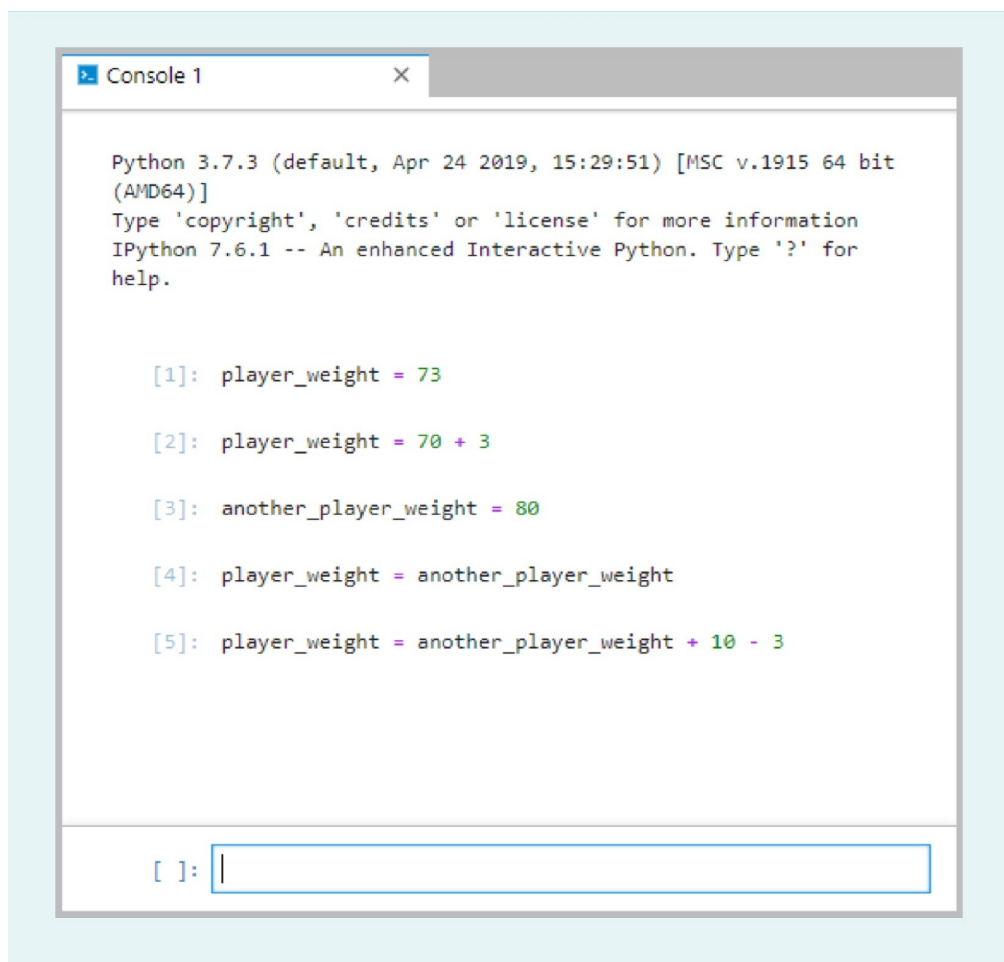
## Code

```
player_weight - 3 = 70
```

But the latter is not a valid Python statement because there is more than one thing on the left-hand side of the assignment operator.

Additionally, you are not restricted to using literals on the right-hand side of the assignment operator. You can have a variable, multiple variables, and even a combination of variables and literals on the right-hand side. Each of the assignment statements below are valid in Python:

**Figure 34: Python 3 Console - Assignment Operator Statements**

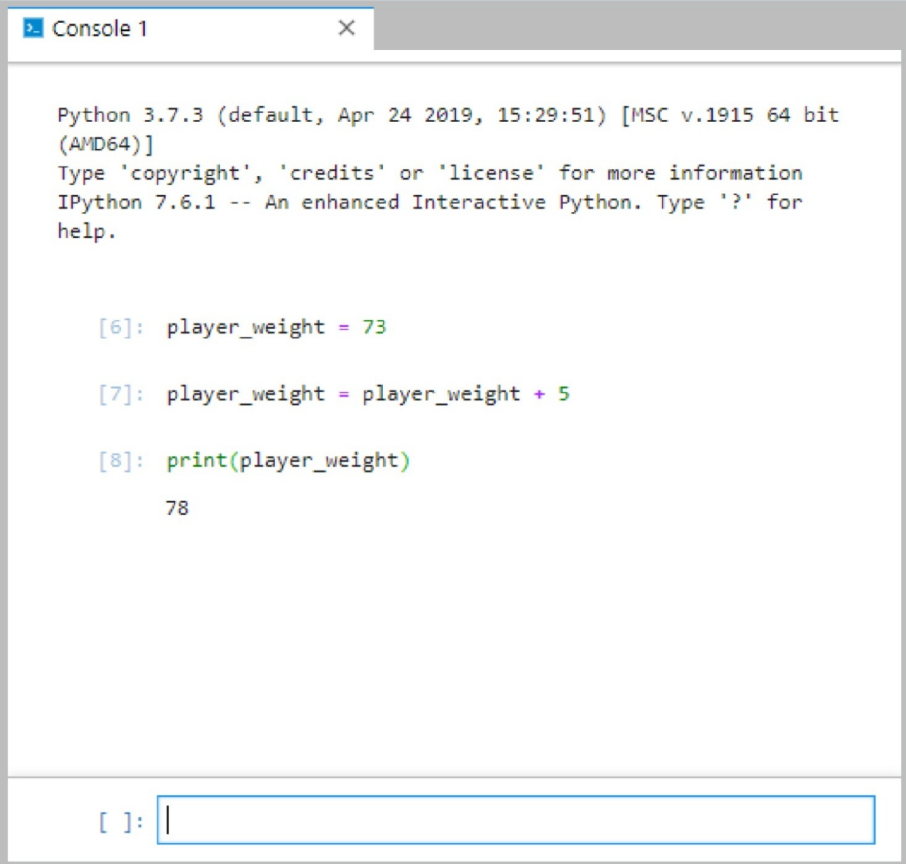


Source: Aaron Reed, 2020.

You can even use the same variable on the left-hand side and the right-hand side. Just remember, the assignment operator processes from right to left, so the value of the variable will be processed on the right first and then assigned to the variable on the left. For example, the statements below are valid:



Figure 35: Python 3 Console - More Assignment Operator Statements



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[6]: player_weight = 73

[7]: player_weight = player_weight + 5

[8]: print(player_weight)
     78

[ ]: |
```

Source: Aaron Reed, 2020.

In the code above, a `player_weight` variable is created and assigned the value 73. Then, the `player_weight` variable is used twice in the second statement (once on the left-hand side of the assignment operator and once on the right). Given that the assignment operator works right-to-left, it will first evaluate the `player_weight` on the right-hand side, at which point the variable's value is 73. The interpreter will take that 73, add 5 to it, and return it to be assigned to the item on the left-hand side of the operator, which is "player\_weight." In the third line of code we see that the `player_weight` variable now has the value 78.

Go ahead and experiment with the assignment operator to get an understanding of some of the basic things you can do.

## 2.2 Numbers

**int**  
In Python, an integer data type that can represent any whole number, positive, negative, or zero, is called an int.

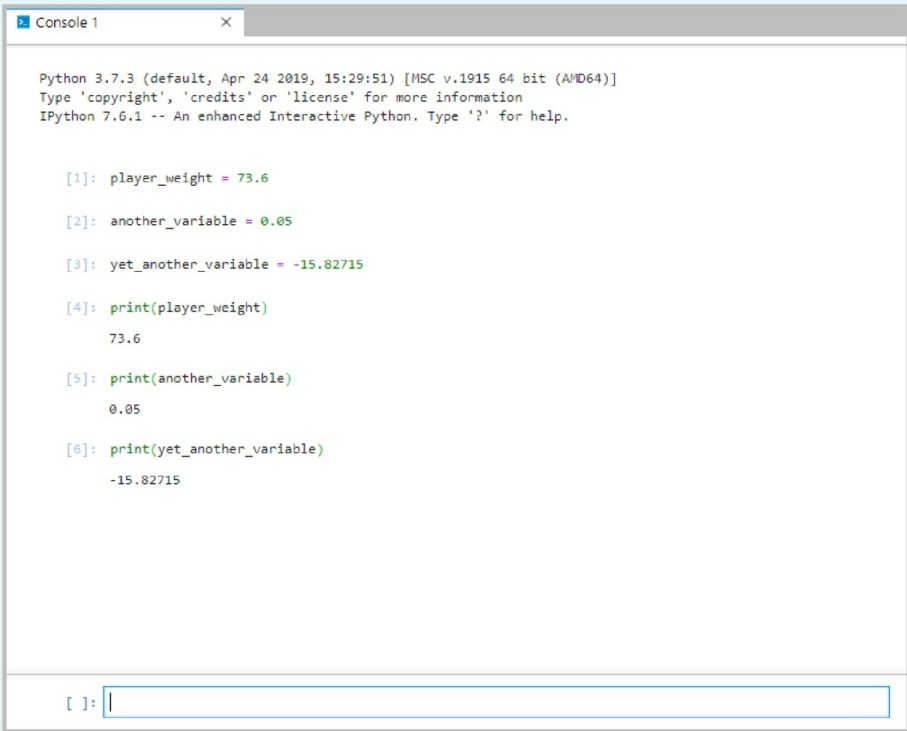
So far, we've created a lot of player weight variables in these examples, and, for whatever reason, we keep setting those variables to 73, representing 73kg. That type of numerical value is known as an integer, or **int**, in Python. Integer values in Python can be positive, negative, or 0. In many languages, an int has a minimum/maximum range representing the values that can be represented by that data type. In Python 2, that was also the case, but in Python 3, the int data type has unlimited size. That is good news for us as programmers because we can use an int to represent any integer value that we could likely ever want to use.

**float**  
This is a numerical data type in Python that represents a value with a decimal point.

But what if we needed to be a bit more nuanced than a whole number? What if we wanted to represent a player weight of 73.6? A number with a decimal point in Python is known as a floating-point number, or a **float**. You can assign float values to variables the same way you do int values. You can use negative or positive float values.

The lines of code below show examples of how to assign float values to variables in Python:

Figure 36: Python - Working with Floating Point Numbers



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[1]: player_weight = 73.6

[2]: another_variable = 0.05

[3]: yet_another_variable = -15.82715

[4]: print(player_weight)
73.6

[5]: print(another_variable)
0.05

[6]: print(yet_another_variable)
-15.82715

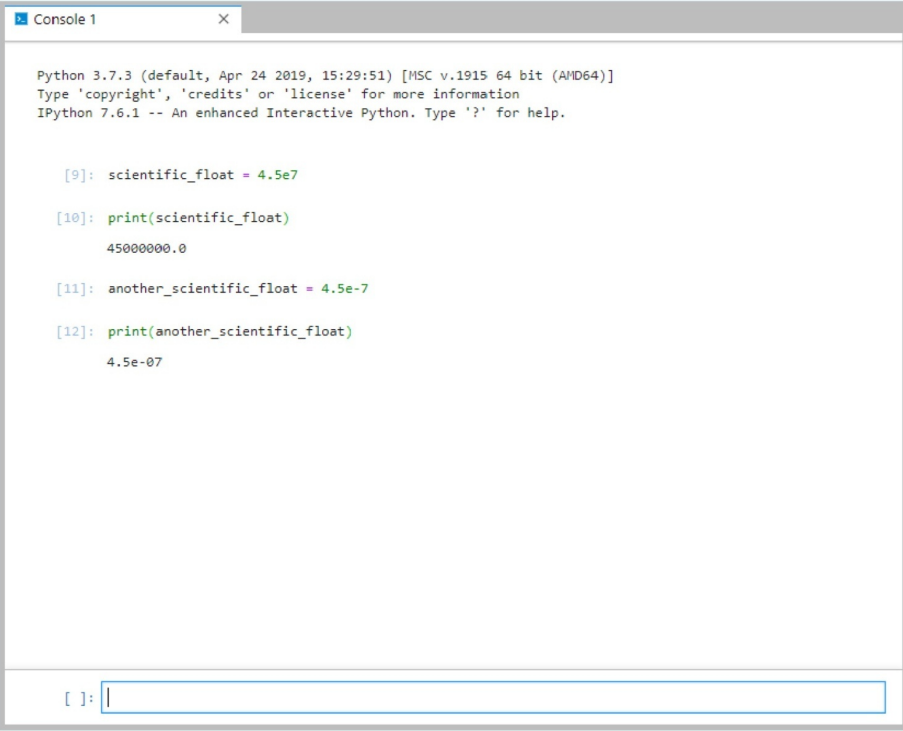
[ ]: |
```

Source: Aaron Reed, 2020.

You can also use the letter e or E to designate **scientific notation** in floating point values. The number after the e or E indicates the number of times to multiply the value by 10 or -10. For example, 4.5e7 is the same as  $4.5 \cdot 10^7$ , or 45,000,000. Below are some examples of scientific notation with floating point numbers:

**Scientific notation**  
A concise way to represent very large or very small numbers is to write it as a number between 1 and 10 multiplied by the appropriate power of ten.

**Figure 37: Python - Working with Scientific Notation Floating Point Numbers**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[9]: scientific_float = 4.5e7

[10]: print(scientific_float)
45000000.0

[11]: another_scientific_float = 4.5e-7

[12]: print(another_scientific_float)
4.5e-07

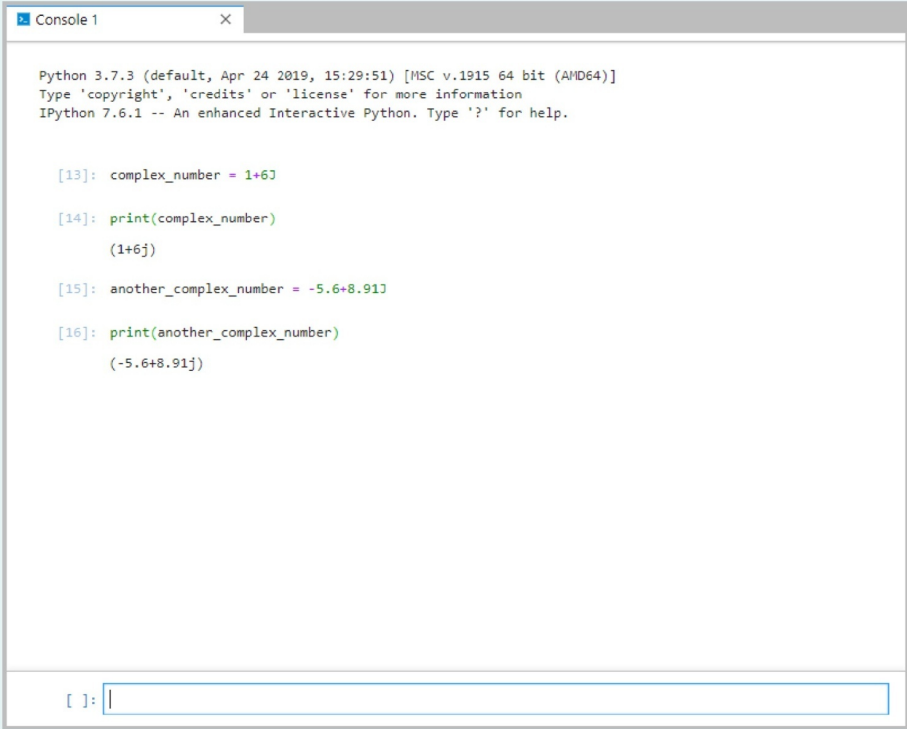
[ ]: |
```

Source: Aaron Reed, 2020.

It is also possible to represent **imaginary numbers** in Python by using a data type called complex (for complex numbers). Complex numbers in Python take the form of  $a + bJ$  where both a and b are floating point numbers and J is the square root of -1, or the imaginary portion of the number.

**Imaginary numbers**  
Numbers that represent the square root of a negative number, typically -1, are called imaginary.

Figure 38: Python - Working with Complex Numbers



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[13]: complex_number = 1+6j

[14]: print(complex_number)
(1+6j)

[15]: another_complex_number = -5.6+8.91j

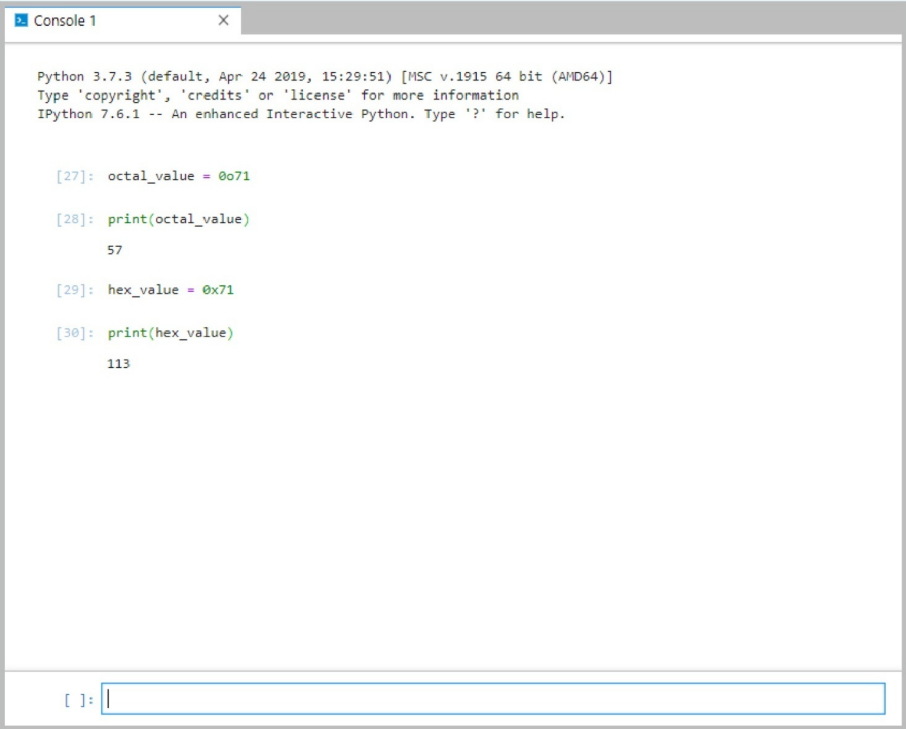
[16]: print(another_complex_number)
(-5.6+8.91j)

[ ]: |
```

Source: Aaron Reed, 2020.

In addition to decimal numbers, Python can also represent hexadecimal and octal values in literals. Hexadecimal is represented by the notation `0x<hex>` where the `<hex>` portion is the hexadecimal number you want to use. Similarly, octal is represented by `0o<octal>`. Below are some examples:

Figure 39: Python - Working with Octal and Hexadecimal Numbers



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[27]: octal_value = 0o71

[28]: print(octal_value)
57

[29]: hex_value = 0x71

[30]: print(hex_value)
113

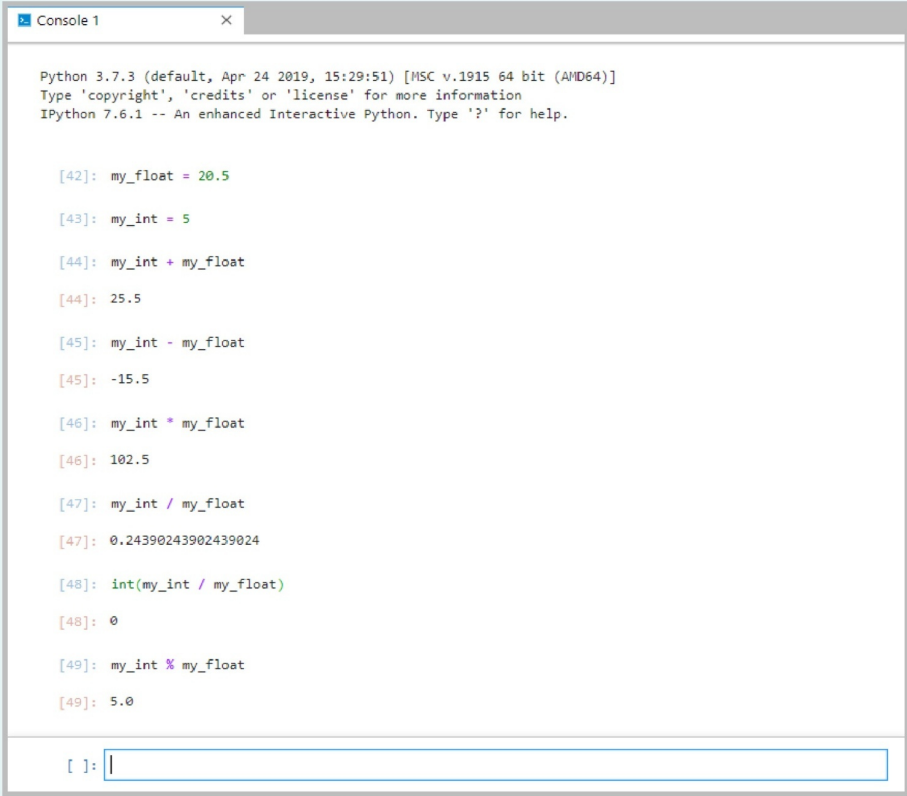
[ ]: |
```

Source: Aaron Reed, 2020.

Finally, you can use the typical arithmetic symbols to calculate values using each of these data types. For simple math, use `+`, `-`, `/`, and `*` for addition, subtraction, division, and multiplication. You can use those operators on mixed data types—for example, multiplying an int and a float is perfectly valid. Just note that when you do so, the result will be a float. For example, `5 * 6.5` will evaluate to `32.5`, which is a floating-point value. If you wanted to just keep the integer portion of that number, you can convert it to an int by using the keyword `int` followed by the number in parentheses, like this: `int(5 * 6.5)`. That expression will result in just the integer value from the result, which is `32`.

Division works the same way. For example, `15 / 6` will yield a floating-point value of `2.5`. To keep just the integer portion of the result, convert it to an int and the result will be just `2`. If you wanted to keep just the remainder from the result, you use the mod operator (`%`) instead of the multiplication operator. For example, `15% 6` results in `3`. Here are some examples:

Figure 40: Python - Simple Arithmetic



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[42]: my_float = 20.5

[43]: my_int = 5

[44]: my_int + my_float
[44]: 25.5

[45]: my_int - my_float
[45]: -15.5

[46]: my_int * my_float
[46]: 102.5

[47]: my_int / my_float
[47]: 0.24390243902439024

[48]: int(my_int / my_float)
[48]: 0

[49]: my_int % my_float
[49]: 5.0

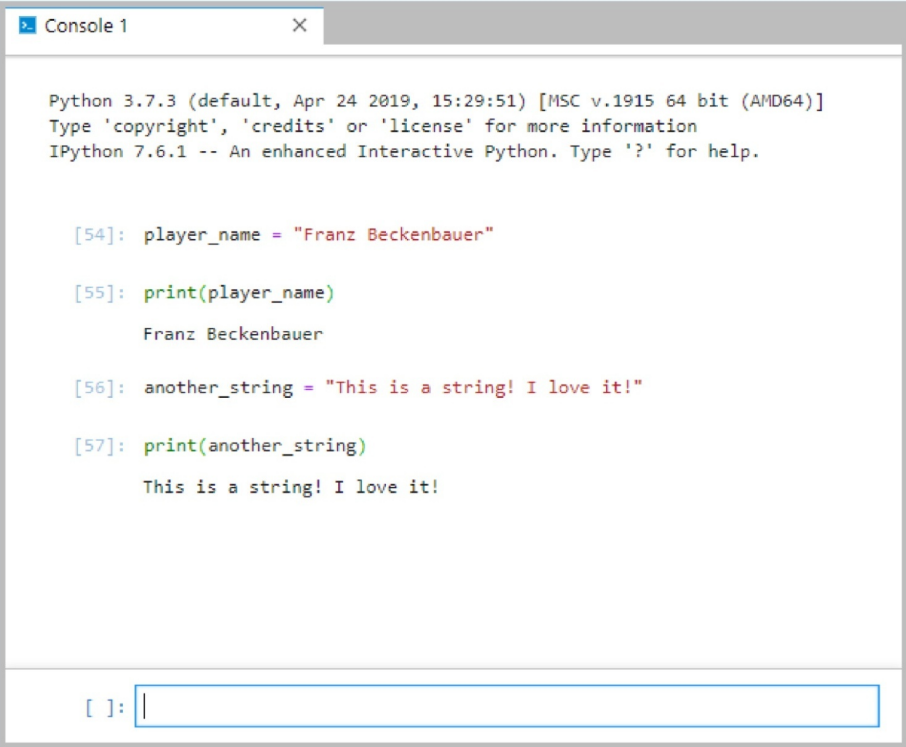
[ ]: |
```

Source: Aaron Reed, 2020.

## 2.3 Strings

So far, we've seen a lot of different numerical data types. Those are great for things like storing a player's weight, height, birth year, jersey number, goal count, and how many pet cats they own. But what about other types of data, such as a player's name? We need a way to represent text. In Python, and in programming in general, text is stored in something called a "string data type" (or `str`). Strings represent a series of characters, including not just upper and lowercase letters but numbers, punctuation marks, spaces, etc. You assign string variables to string data by simply using a new variable name and assigning it some string value contained in quotation marks as shown here:

Figure 41: Python - Simple Strings



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[54]: player_name = "Franz Beckenbauer"

[55]: print(player_name)
      Franz Beckenbauer

[56]: another_string = "This is a string! I love it!"

[57]: print(another_string)
      This is a string! I love it!

[ ]:
```

Source: Aaron Reed, 2020.

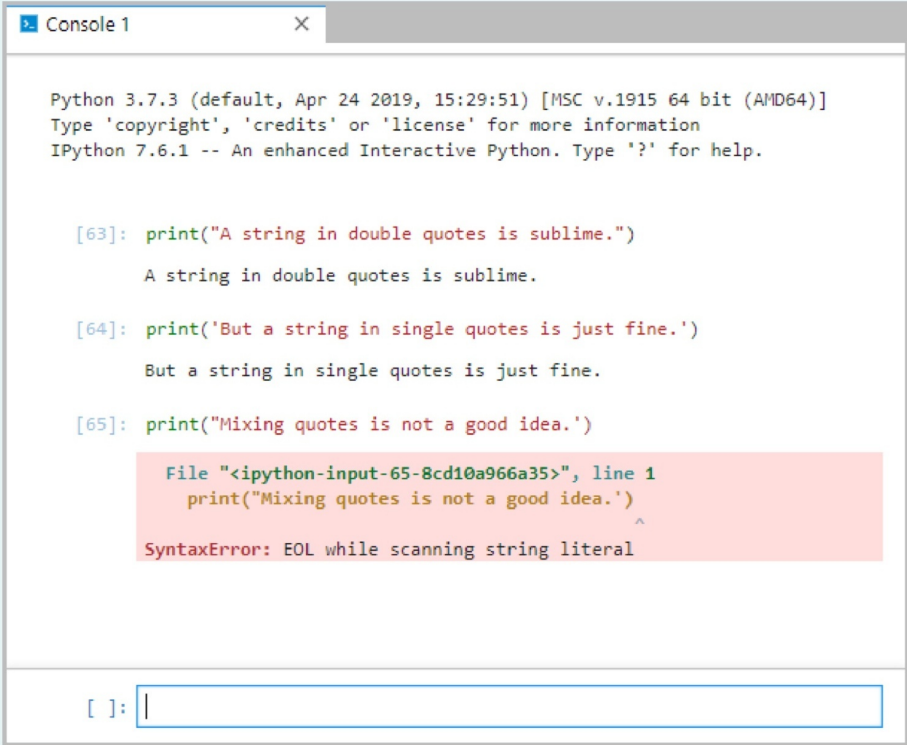
One thing to note is that strings in Python are **immutable**, meaning they cannot be changed. Once you assign a string to a variable, you cannot change that string. You can, however, reassign the variable to a new string. For example, the variable “another\_string” in the previous example was assigned the string “This is a string! I love it!” Once assigned, that text cannot change. But we can assign the variable “another\_string” to a new, different string. The opposite of immutable is mutable, so it would be fair to say that the variable itself is mutable (we can change it and modify it by reassigning it to a new value) but the string it contains is immutable (there’s no way to modify the text assigned to a variable once it is assigned).

It is also important to note that in Python, you can use either double quotes or single quotes to denote a string. The only trick is to ensure that you use the same type on the front and the back of the string:

**Immutable**

In Python, an object that cannot change or be modified is described as immutable.

Figure 42: Python - String Quotes



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[63]: print("A string in double quotes is sublime.")
      A string in double quotes is sublime.

[64]: print('But a string in single quotes is just fine.')
      But a string in single quotes is just fine.

[65]: print("Mixing quotes is not a good idea.")
      File "<ipython-input-65-8cd10a966a35>", line 1
        print("Mixing quotes is not a good idea.")
                                     ^
      SyntaxError: EOL while scanning string literal

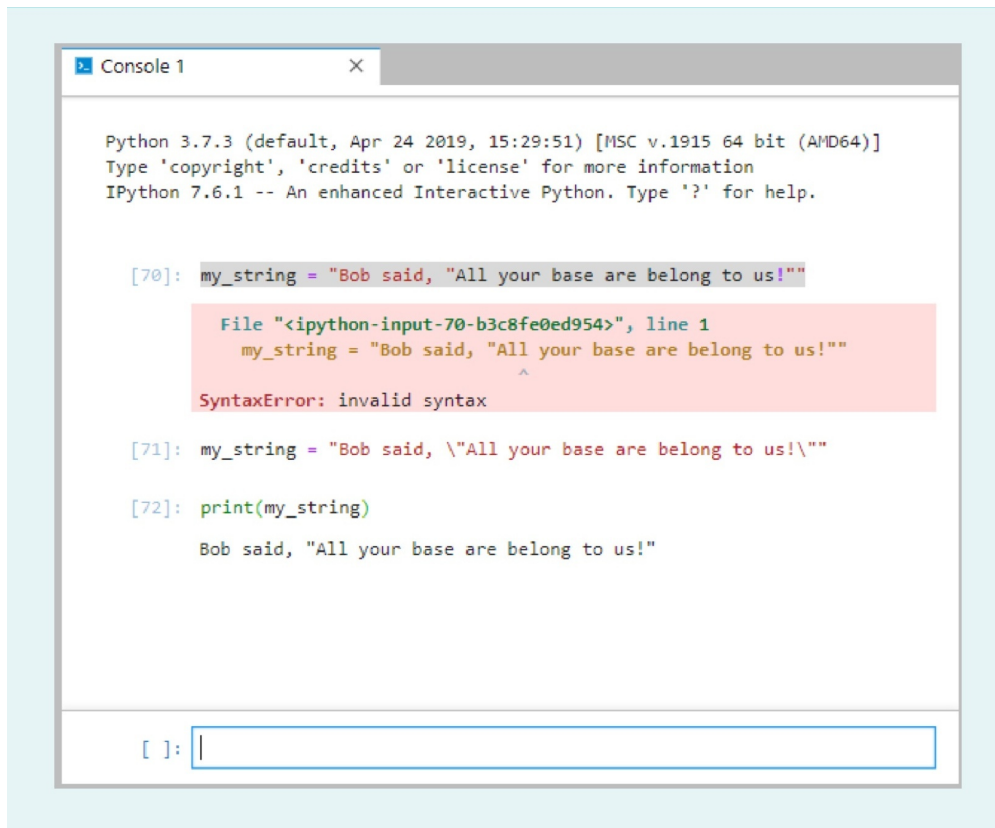
[ ]: |
```

Source: Aaron Reed, 2020.

There are a few important characters that you cannot type into a regular string. Quotation marks, for example, would just end the string if you inserted them in the middle of the string. If we needed them there, we would use what's called an "escape character" to add quotation marks in the middle of our string. The escape character is a backslash (\); to add a quotation mark, you escape it by putting the backslash in front of the quote ("). Just as the backslash character is called an escape character, the combination of the escape character with the quotation mark (or another escape character) is called an escape sequence. For usage, see below:



Figure 43: Python - String with Escaped Quotes



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[70]: my_string = "Bob said, "All your base are belong to us!"
      File "<ipython-input-70-b3c8fe0ed954>", line 1
        my_string = "Bob said, "All your base are belong to us!"
                               ^
      SyntaxError: invalid syntax

[71]: my_string = "Bob said, \"All your base are belong to us!\""

[72]: print(my_string)
      Bob said, "All your base are belong to us!"

[ ]: |
```

Source: Aaron Reed, 2020.

Other escape sequences that should be noted include:

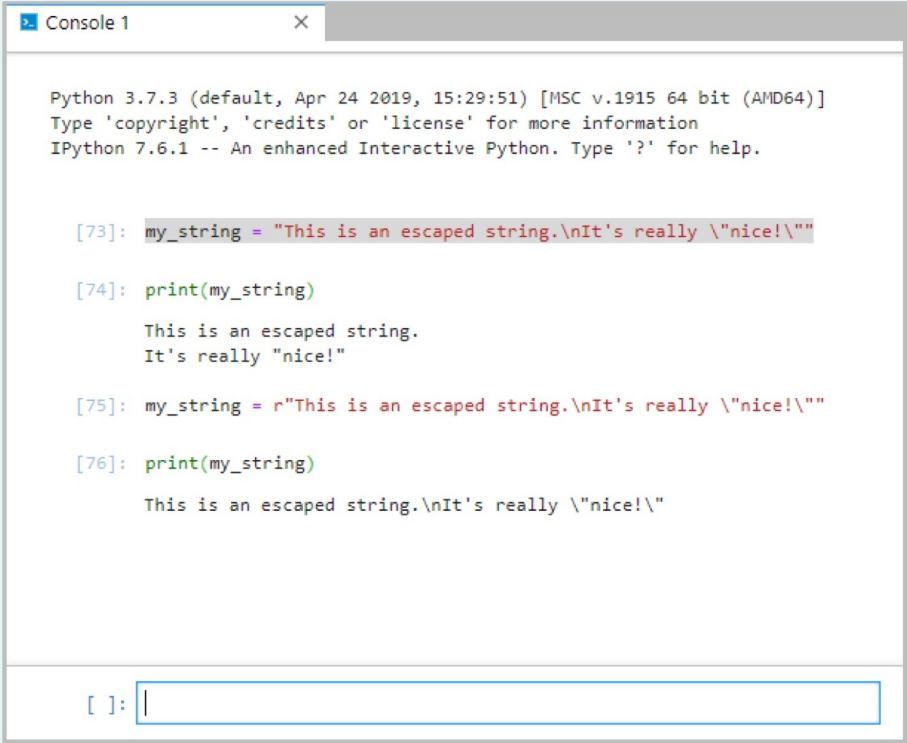
Table 2: Python—Escape Sequences

Escape Sequence	Result
\\	A single backslash character
\'	A single quote
\"	A double quote
\n	An ASCII new line character
\r	An ASCII carriage return character
\t	An ASCII tab character

Source: Aaron Reed, 2020.

If you ever wanted to ignore the escape character in a string, you could do so by using what’s called a “raw string.” When typing the string, preface the first quotation mark with the character r and the string that follows will ignore any escape sequences. For example:

Figure 44: Python - Raw Strings



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[73]: my_string = "This is an escaped string.\nIt's really \"nice!\""

[74]: print(my_string)
      This is an escaped string.
      It's really "nice!"

[75]: my_string = r"This is an escaped string.\nIt's really \"nice!\""

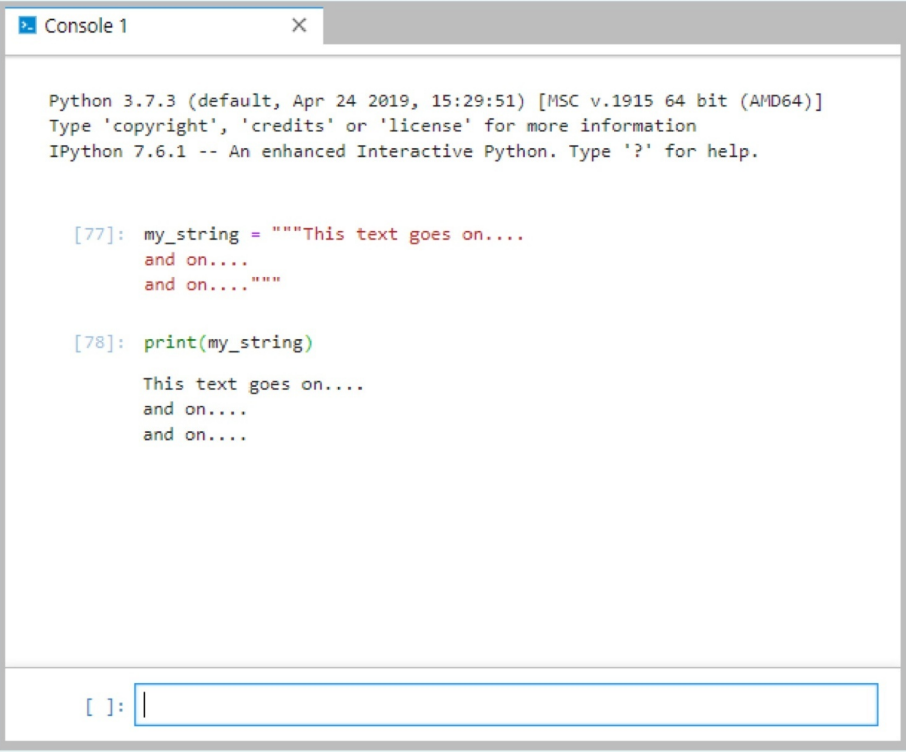
[76]: print(my_string)
      This is an escaped string.\nIt's really \"nice!\"

[ ]: |
```

Source: Aaron Reed, 2020.

You can also use triple quotes (either `"""` or `'''`) to denote text that you want to span multiple lines. See below:

Figure 45: Python - Triple Quoted Strings



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[77]: my_string = """This text goes on...
      and on...
      and on..."""

[78]: print(my_string)

      This text goes on...
      and on...
      and on...

[ ]: |
```

Source: Aaron Reed, 2020.

Finally, there are some important string operations that we should look at. See the table below for a description of each and its syntax usage:

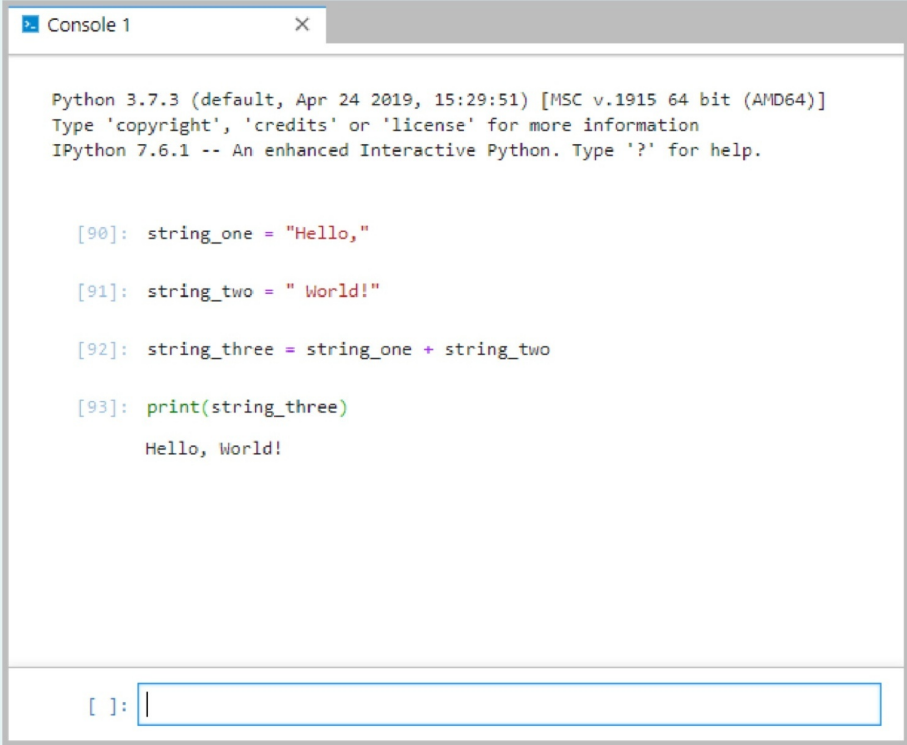
Table 3: Python—String Operations

Purpose	Example	Example Result
Get the length of a string	<code>len("Hello, World!")</code>	13
Get the index – i.e. the position - of a character within a string. Counting starts at 0.	<code>my_string = "Hello, World!"</code> <code>my_string.index('e')</code>	1
Count the number of matching characters in a string	<code>my_string = "Hello, World!"</code> <code>my_string.count('o')</code>	2
Convert all characters to lowercase	<code>my_string = "Hello, World!"</code> <code>my_string.lower()</code>	'hello, world!'
Convert all characters to uppercase	<code>my_string = "Hello, World!"</code> <code>my_string.upper()</code>	'HELLO, WORLD!'

Source: Aaron Reed, 2020.

You can also add two strings together to concatenate them by just using the addition symbol (+) between the two strings, as shown here:

**Figure 46: Python - Concatenating Strings**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

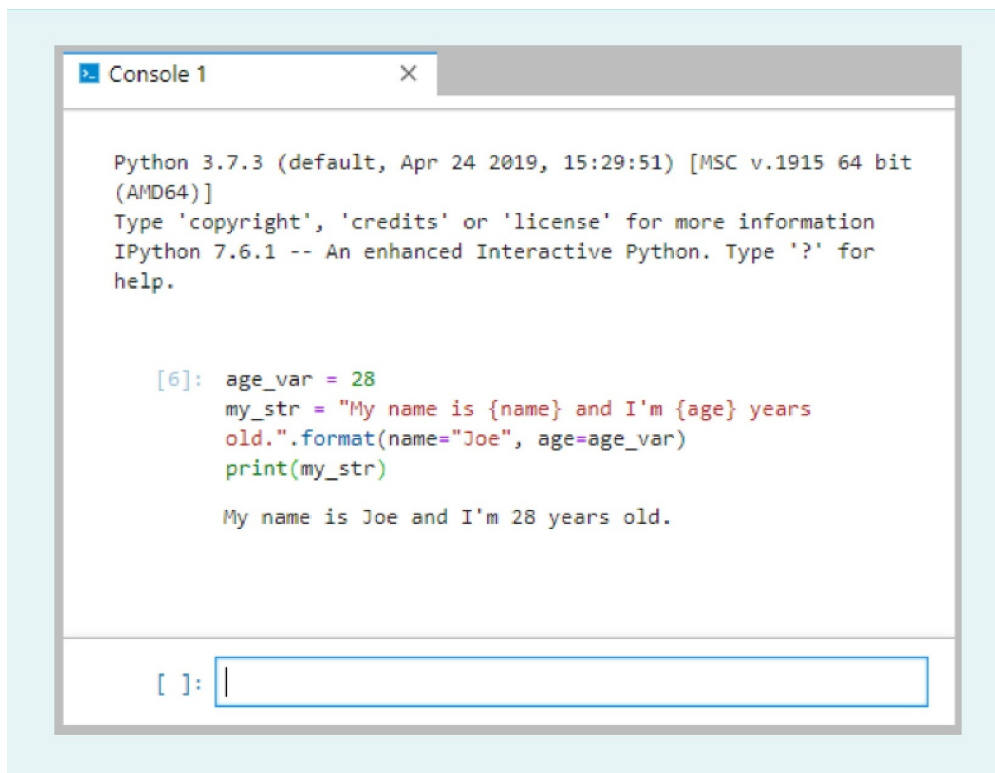
[90]: string_one = "Hello,"
[91]: string_two = " World!"
[92]: string_three = string_one + string_two
[93]: print(string_three)
      Hello, World!

[ ]: |
```

Source: Aaron Reed, 2020.

Another way to concatenate strings is through the format function. In this method, you embed temporary variables within a string by surrounding them with curly braces {}, and then, using the format function, you specify values for those variables. The format function can accept literals or variables as values for the variables in the string. See an example of the string format function below:

Figure 47: Python - String Format Function

A screenshot of a Python console window titled "Console 1". The window shows the Python 3.7.3 startup message, including the version, date, and architecture. It then displays the execution of a code block where a variable 'age\_var' is set to 28, a string 'my\_str' is formatted with 'Joe' and '28', and the result is printed. The output is "My name is Joe and I'm 28 years old." The console prompt "[ ]:" is visible at the bottom.

```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[6]: age_var = 28
     my_str = "My name is {name} and I'm {age} years old.".format(name="Joe", age=age_var)
     print(my_str)

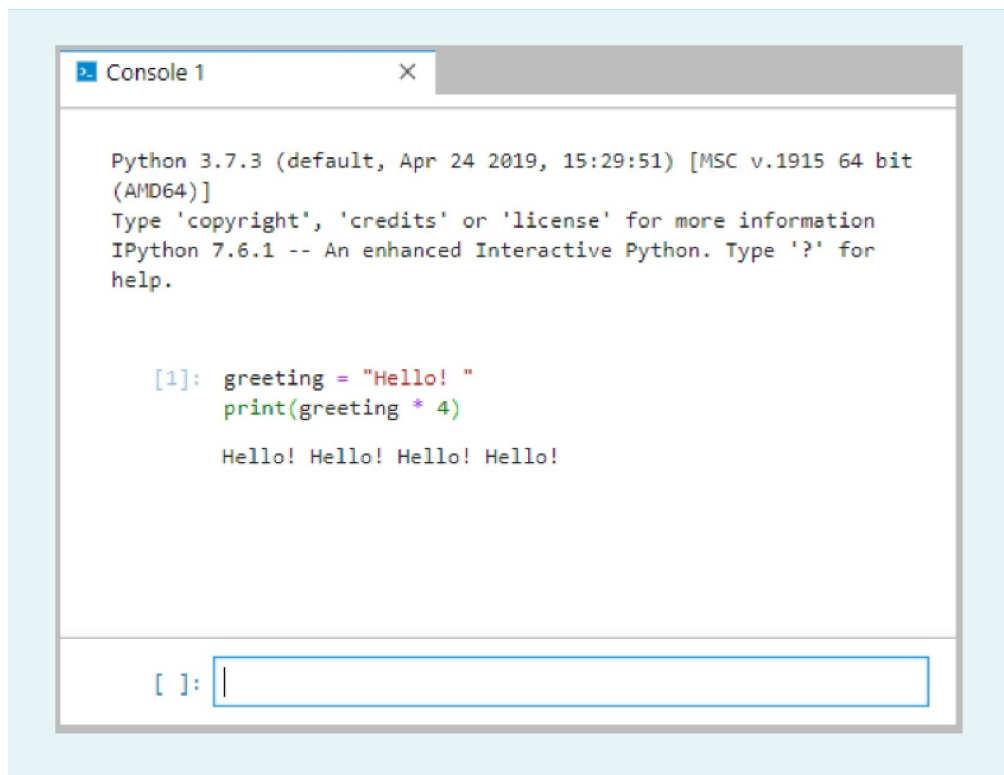
     My name is Joe and I'm 28 years old.

[ ]: |
```

Source: Aaron Reed, 2020.

There may also be times when you want to replicate a string a given number of times. This can be done with the \* operator. For example, see below:

Figure 48: Python - String Replication

A screenshot of a Python console window titled "Console 1". The window shows the Python 3.7.3 startup message, including the version, date, and architecture (AMD64). It also displays the IPython 7.6.1 startup message. The user has entered two lines of code: `greeting = "Hello! "` and `print(greeting * 4)`. The output of the code is `Hello! Hello! Hello! Hello!`. At the bottom of the console, there is a prompt `[ ]:` followed by a text input field containing a vertical bar `|`, indicating the user is ready to enter more code.

```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[1]: greeting = "Hello! "
     print(greeting * 4)

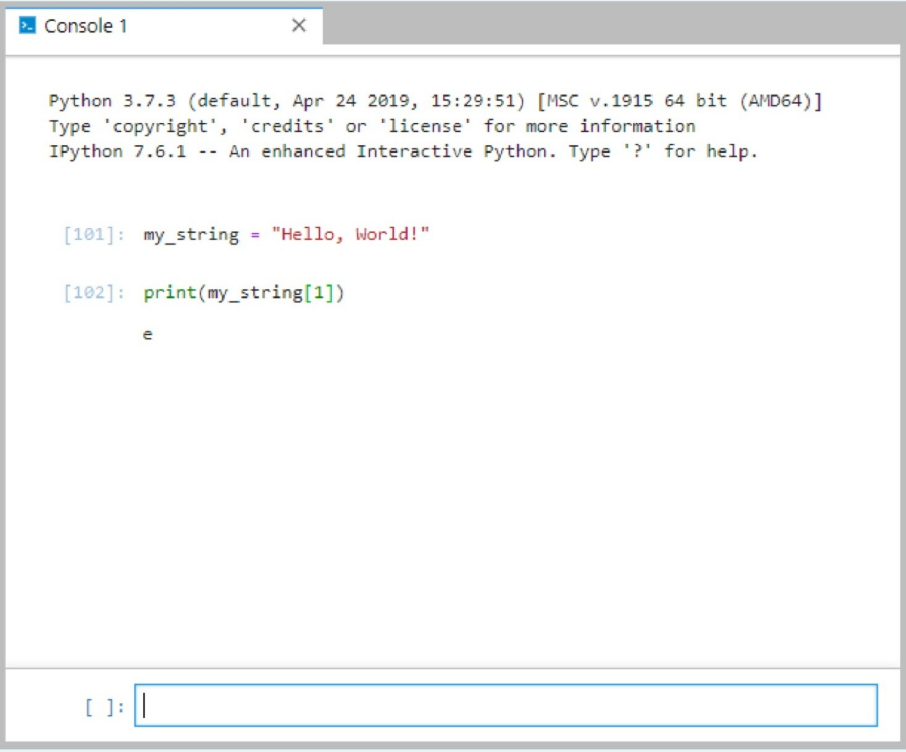
Hello! Hello! Hello! Hello!

[ ]: |
```

Source: Aaron Reed, 2020.

Sometimes it may be helpful to retrieve part of a string from a larger string. This is typically called a “substring operation” in most programming languages. In Python, to get a substring from a string, you use brackets on the original string ([ ]) with up to three parameter values inside those brackets. The first parameter in the brackets indicates the point at which to start retrieving a substring. Using only that parameter will return only the character at that index. For example:

**Figure 49: Python - Substring Part 1**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[101]: my_string = "Hello, World!"

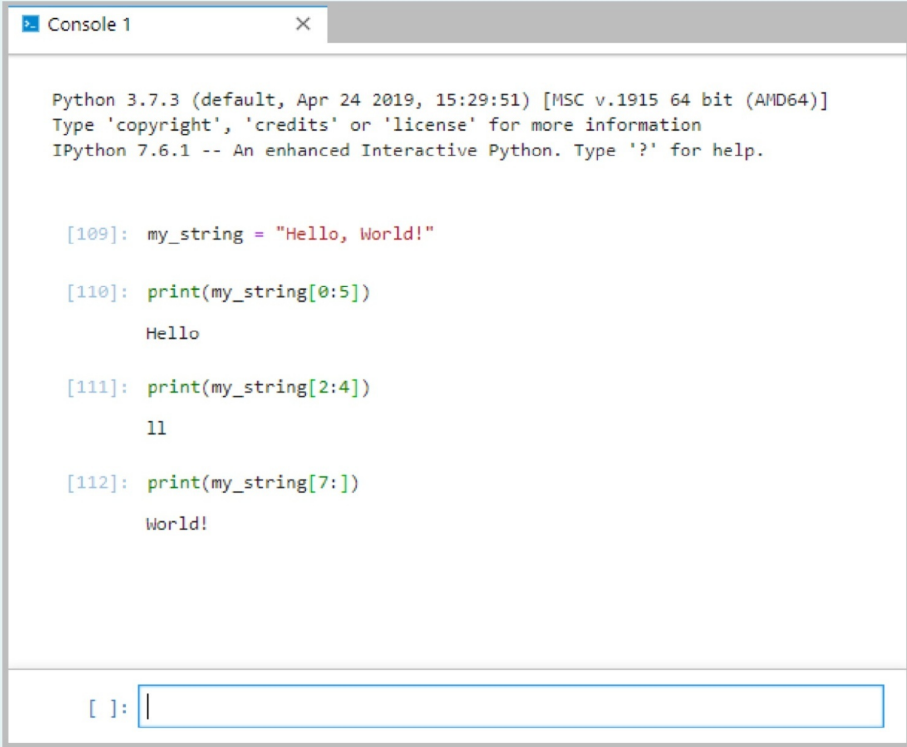
[102]: print(my_string[1])
e

[ ]:
```

Source: Aaron Reed, 2020.

The second parameter of the substring indicates the ending index of the substring to return. You separate the two parameters with a colon (:); if you insert the colon but leave the second parameter blank, it will just return the rest of the string. Here are some examples:

Figure 50: Python - Substring Part 2



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[109]: my_string = "Hello, World!"

[110]: print(my_string[0:5])
Hello

[111]: print(my_string[2:4])
ll

[112]: print(my_string[7:])
World!

[ ]: |
```

Source: Aaron Reed, 2020.

The third parameter in the substring operation is the step. The step allows you to specify an increment that identifies characters to be returned in the substring. Think of it as “return every xth character.” When omitted, the parameter defaults to one, which is to say, return every first character (or all characters). If you use a “2” for the third parameter, it will return every second character or every other character. If you use a “3,” it will return every third character, and so on. See usages here:



Figure 51: Python - Substring Part 3



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[114]: my_string = "Hello, World!"

[115]: print(my_string[0:])
Hello, World!

[116]: print(my_string[0:2])
Hlo ol!

[117]: print(my_string[0:3])
Hl r!

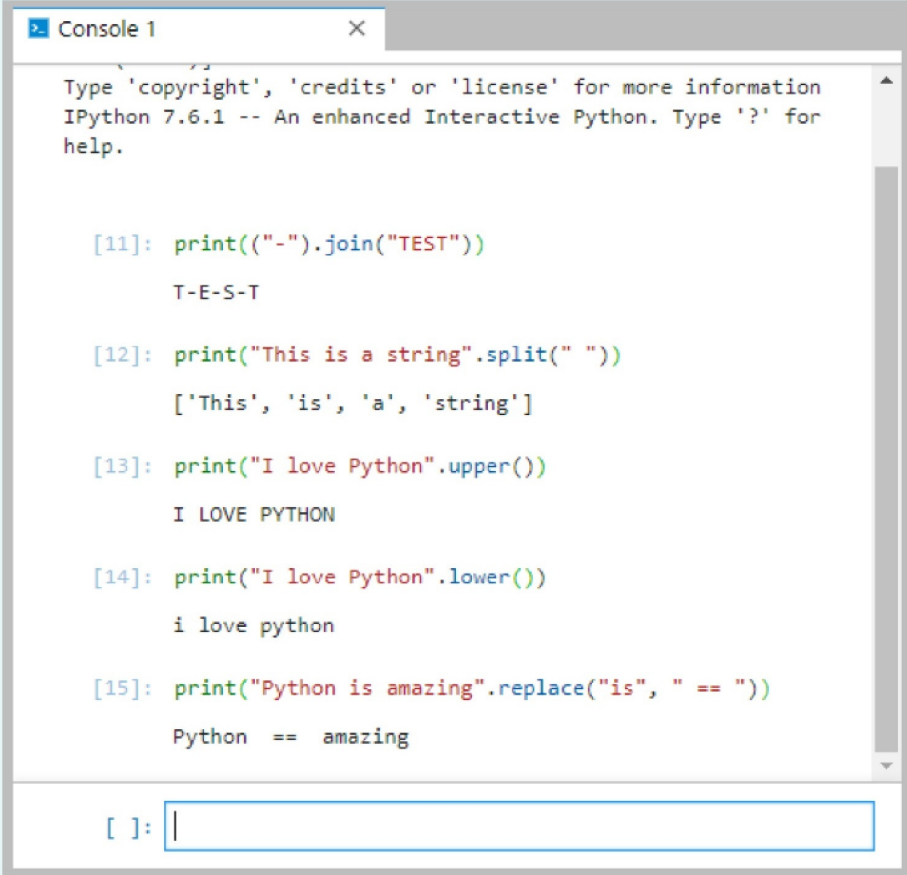
[118]: print(my_string[0:4])
Hoo!

[ ]:
```

Source: Aaron Reed, 2020.

Some other commonly used string manipulation functions include `join()`, `split()`, `upper()`, `lower()`, and `replace()`. With `join()`, you can specify a string to be inserted between every letter of another string. The `split()` function will split a string into multiple strings at every instance of a particular character or substring. The `upper()` and `lower()` functions convert a string to upper- or lowercase, and the `replace()` function will replace a character or substring within a string with another character or substring. Examples of each are given below:

Figure 52: Python - String Manipulation Functions



```
Console 1
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[11]: print("-").join("TEST")
      T-E-S-T

[12]: print("This is a string".split(" "))
      ['This', 'is', 'a', 'string']

[13]: print("I love Python".upper())
      I LOVE PYTHON

[14]: print("I love Python".lower())
      i love python

[15]: print("Python is amazing".replace("is", " == "))
      Python == amazing

[ ]: |
```

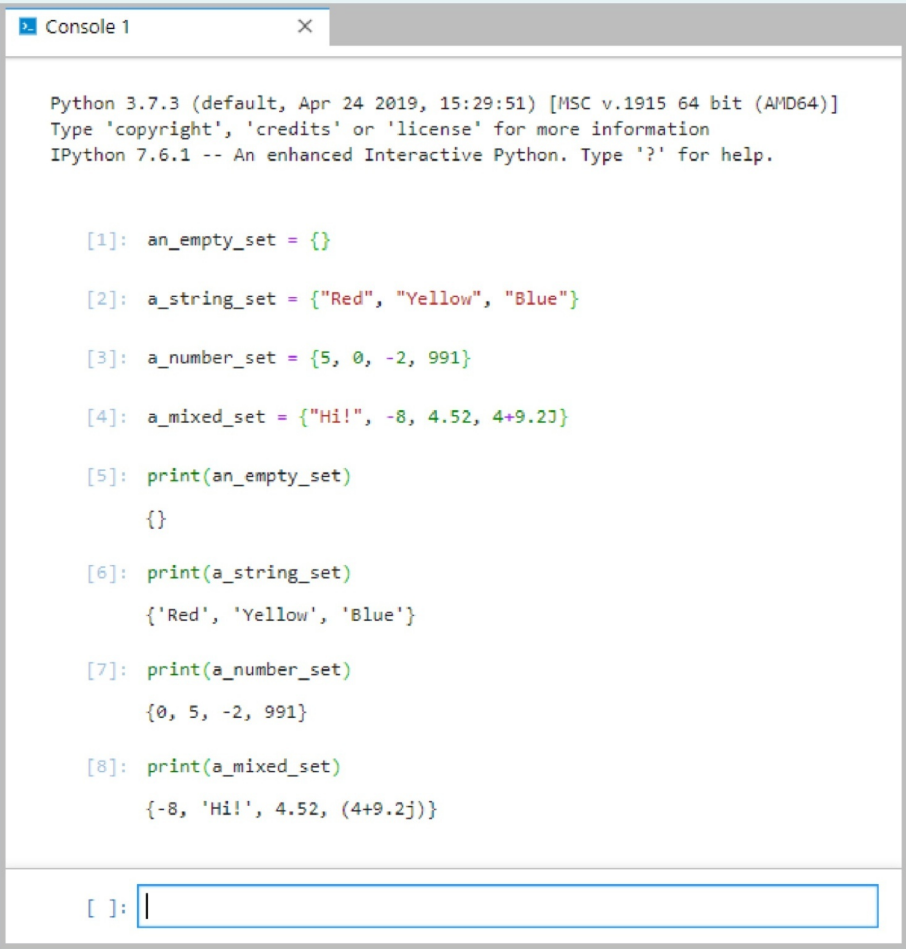
Source: Aaron Reed, 2020.

## 2.4 Collections

We are now equipped to create variables that can store various numerical data types including whole numbers, floating-point values, and any type of string we could hope for. That should come in handy because now we can create variables to store, for instance, a player's name, jersey number, height, weight, and anything else we want. Pretty amazing! But, how many players play on a soccer team? Yeah, we still have some work to do. If, for example, we wanted to store a list of each player on a team, we would have to have 11 different variables for player's names—and that's just for starters! That doesn't sound like the right way to go. There must be an easier way.

Luckily, Python has multiple ways to deal with collections of data. First, let's look at a collection type called "sets." A Python set is a structure that will hold data in an unsorted manner. You create a set by setting a variable equal to curly braces ({} ) and you can include items in the set by entering them between the braces. Here are some examples:

Figure 53: Python — Sets



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[1]: an_empty_set = {}

[2]: a_string_set = {"Red", "Yellow", "Blue"}

[3]: a_number_set = {5, 0, -2, 991}

[4]: a_mixed_set = {"Hi!", -8, 4.52, 4+9.2j}

[5]: print(an_empty_set)
{}

[6]: print(a_string_set)
{'Red', 'Yellow', 'Blue'}

[7]: print(a_number_set)
{0, 5, -2, 991}

[8]: print(a_mixed_set)
{-8, 'Hi!', 4.52, (4+9.2j)}

[ ]: |
```

Source: Aaron Reed, 2020.

In the example above, we have created four sets. First, we created an empty set by using the curly braces with nothing inside of them. Second, we created a set of three strings. Third, we created a set of numbers. And fourth, we created a set of mixed data types (string, int, float, and complex). Note the flexibility you have with Python sets! If you're familiar with other programming languages, you will probably appreciate how nice it is that Python allows mixed data types within the same collection—that is not something you typically see in other languages. Also note the print statement and how it works on a set, outputting the contents of the set, separated by commas, and surrounded by curly braces.

Sets come with a range of methods to help you use them effectively. Think of a method as a way to perform some kind of operation on the set, such as adding or removing an element to/from the set. Some of the more useful methods are shown below:

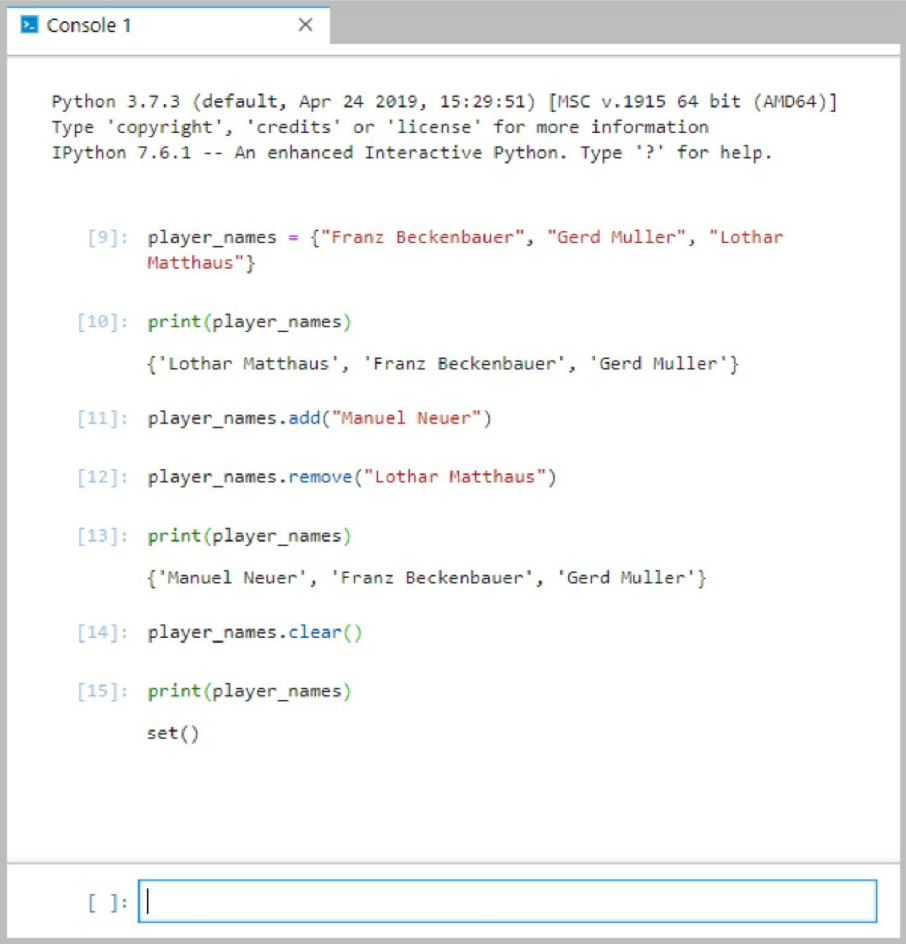
**Table 4: Python—Set Methods**

<b>Method</b>	<b>Purpose</b>
<code>add(<i>element</i>)</code>	Adds an element to the set
<code>remove(<i>element</i>)</code>	Removes a specific element from the set
<code>clear()</code>	Removes all elements from the set

Source: Aaron Reed, 2020.

The example below illustrates how to use these functions with a set called `player_names`:

Figure 54: Python - Set Methods in Use



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[9]: player_names = {"Franz Beckenbauer", "Gerd Muller", "Lothar
    Matthauss"}

[10]: print(player_names)
      {'Lothar Matthauss', 'Franz Beckenbauer', 'Gerd Muller'}

[11]: player_names.add("Manuel Neuer")

[12]: player_names.remove("Lothar Matthauss")

[13]: print(player_names)
      {'Manuel Neuer', 'Franz Beckenbauer', 'Gerd Muller'}

[14]: player_names.clear()

[15]: print(player_names)
      set()

[ ]: |
```

Source: Aaron Reed, 2020.

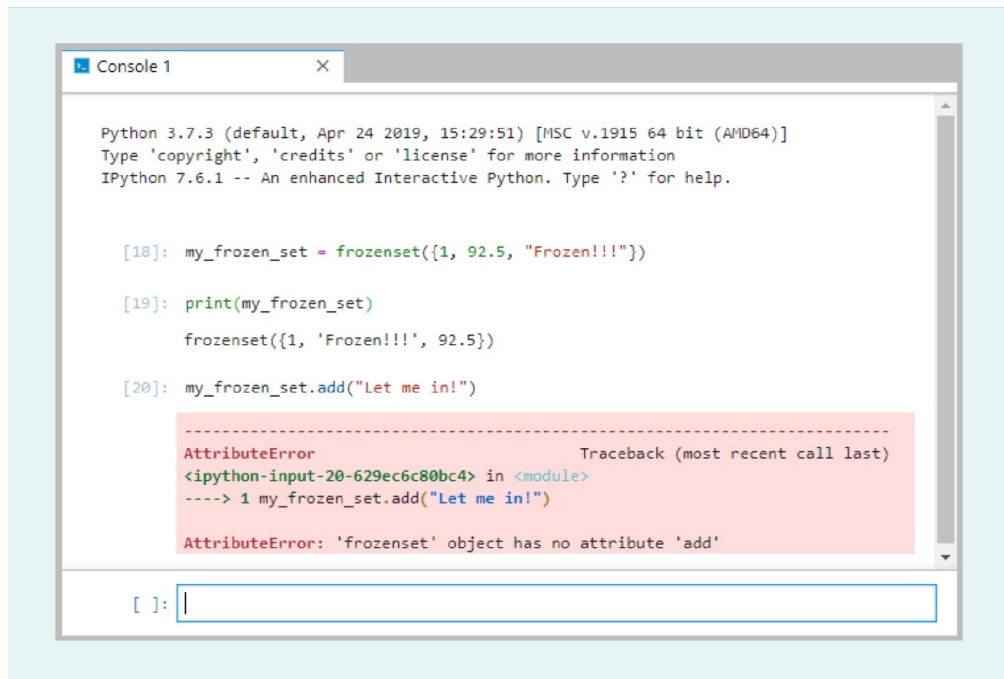
In the code above, a set named `player_names` is created, and it initially contains the names of three soccer players. In line 11, a new player name is added (Manual Neuer), and in line 12, one is removed from the list (Lothar Matthauss). The set is then printed to verify the changes were made correctly. The list is then cleared in line 14 and printed again to verify the list is now empty.

Sets will not allow duplicate elements of the same value. If you try to add a new element with the same value as an existing element, you won't get an error, but the new element will not be added to the list because it is already there.

Go ahead and play with the set functions on your own to get a feel for how they work with various data types.

Sometimes it may be helpful to create a set and not allow changes, making the set immutable. We can create an immutable set in the same way we created the sets above, but we use the keyword “frozenset” and parentheses around the set we are trying to create. See below for usage:

Figure 55: Python — Frozen Set



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[18]: my_frozen_set = frozenset({1, 92.5, "Frozen!!!"})

[19]: print(my_frozen_set)
      frozenset({1, 'Frozen!!!', 92.5})

[20]: my_frozen_set.add("Let me in!")
      -----
      AttributeError                                Traceback (most recent call last)
      <ipython-input-20-629ec6c80bc4> in <module>
      ----> 1 my_frozen_set.add("Let me in!")
      AttributeError: 'frozenset' object has no attribute 'add'
```

Source: Aaron Reed, 2020.

Note the error we get from the interpreter when we try to add an element to the frozen set. If you have a set of data in an application that you never want to change, a frozen set would be a better option than a standard set.

Another collection data type in Python is called a list. Effectively, lists and sets behave similarly with two major exceptions: 1) lists allow duplicate elements within the list and 2) lists are ordered, meaning that each element occupies a specific place in the list. While sets are created using curly braces, lists are created in an almost identical manner, however they are created using square brackets ([ ]) instead of the braces.

There are also a variety of methods for working with lists in Python, for example:

Table 5: Python—List Methods

<code>append(<i>element</i>)</code>	Appends an element to the end of the list
<code>insert(<i>i</i>, <i>element</i>)</code>	Inserts an element at index <i>i</i> in the list
<code>remove(<i>element</i>)</code>	Removes an element from the list

<code>clear()</code>	Removes all elements from the list
<code>count(element)</code>	Returns the number of times a particular element appears in the list

Source: Aaron Reed, 2020.

Review the code below to see how to create and work with lists in Python:

**Figure 56: Python - Lists**

```

Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[7]: my_list = [1, 2.6, "Bob", 5+7.2j]

[8]: print(my_list)
     [1, 2.6, 'Bob', (5+7.2j)]

[9]: my_list.append("Janet")

[10]: print(my_list)
      [1, 2.6, 'Bob', (5+7.2j), 'Janet']

[11]: my_list.insert(2, 99)

[12]: print(my_list)
      [1, 2.6, 99, 'Bob', (5+7.2j), 'Janet']

[13]: my_list.remove(2.6)

[14]: print(my_list)
      [1, 99, 'Bob', (5+7.2j), 'Janet']

[ ]: |

```

Source: Aaron Reed, 2020.

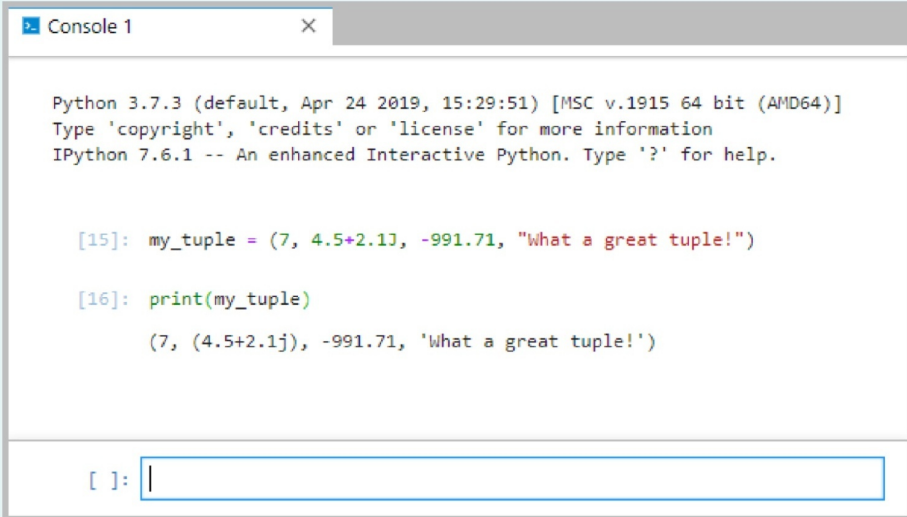
A couple of things to note here. First, notice that the list allows multiple data types just like the set. Also, as you saw in the table of methods, there is no add method for lists. Instead, because the list is ordered, you have two ways to add items: `append`, which adds an element to the back of the list, and `insert`, which adds an element at a specified index.

You may be wondering why we have both sets and lists when they are so similar. Well, for reasons we won't go into deeply at this point in time, sets are much more effective and faster at identifying whether or not an item is found in the set, while lists are much faster at looping through to view or manipulate each item in the list. So really, your choice

depends on the purpose of the set or list. Choosing the right one can have a big impact on the speed of your application. If that doesn't make much sense right now, don't worry—it will later on when we dive into using lists and sets in more depth.

While sets have a frozen set option that creates an immutable set, lists have analogous structures that behave in similar ways but are also immutable: tuples. A tuple is a sequence of immutable objects; they are created using parentheses rather than curly braces or brackets. See below for an example of how to create a tuple:

**Figure 57: Python - Tuples**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[15]: my_tuple = (7, 4.5+2.1j, -991.71, "What a great tuple!")

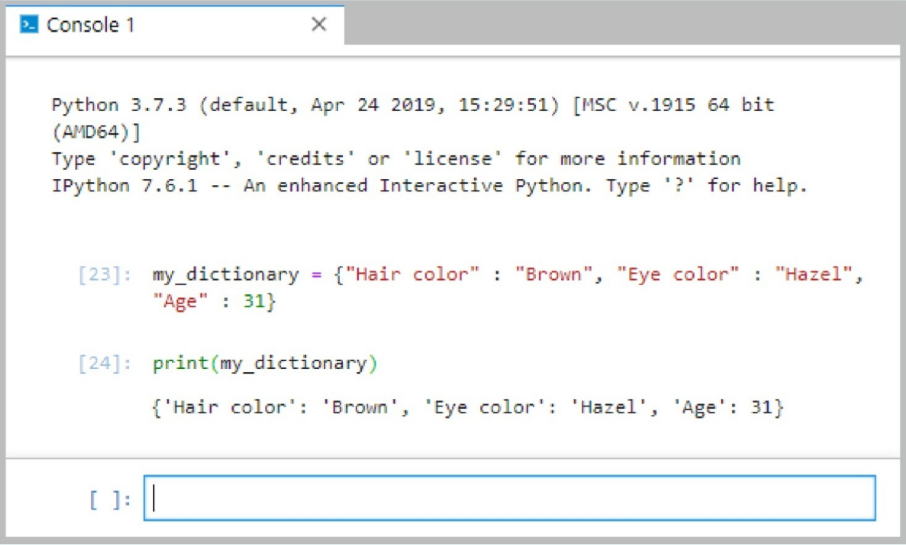
[16]: print(my_tuple)
      (7, (4.5+2.1j), -991.71, 'What a great tuple!')
```

Source: Aaron Reed, 2020.

Another important collection in Python is the dictionary. Dictionaries are collections that are unordered and changeable (mutable) where each element consists of a key and a value. You create a dictionary in a way almost identical to creating sets, but for each element, you separate the key from the value with a colon (:) like this:



Figure 58: Python - Dictionaries



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[23]: my_dictionary = {"Hair color" : "Brown", "Eye color" : "Hazel",
    "Age" : 31}

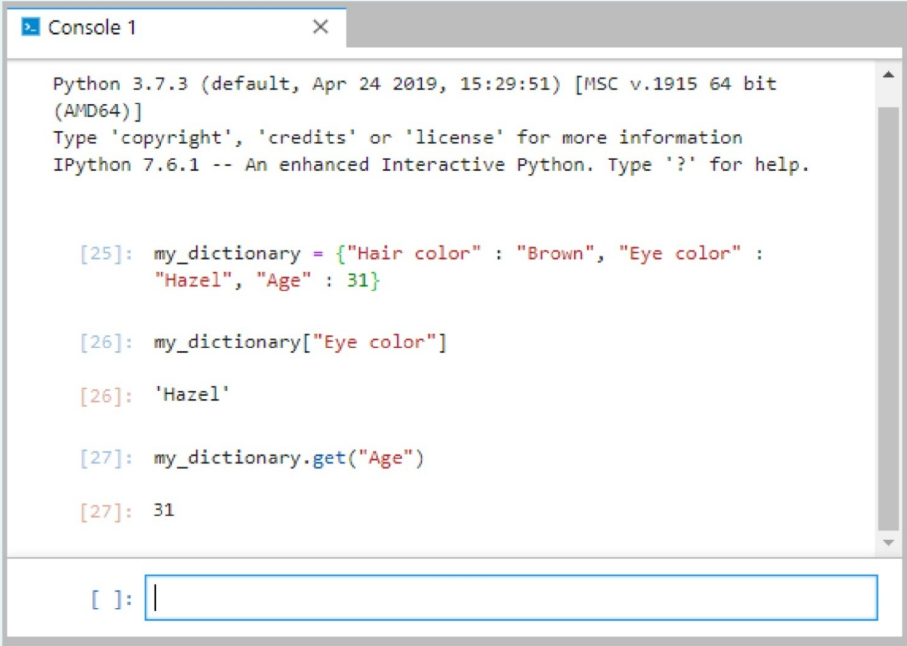
[24]: print(my_dictionary)
{'Hair color': 'Brown', 'Eye color': 'Hazel', 'Age': 31}

[ ]: |
```

Source: Aaron Reed, 2020.

You can then use a dictionary to check the value of a particular key. To do so, you can either put the name of the key in brackets ( [ ] ) or you can use the get method as shown here:

Figure 59: Python - Using Dictionaries



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[25]: my_dictionary = {"Hair color" : "Brown", "Eye color" :
      "Hazel", "Age" : 31}

[26]: my_dictionary["Eye color"]

[26]: 'Hazel'

[27]: my_dictionary.get("Age")

[27]: 31

[ ]: |
```

Source: Aaron Reed, 2020.

There is a lot of information packed into this section. If you are feeling that you don't quite understand all these collections, don't worry right now. The most important thing for you to know is that there are a wide range of collections available in Python and that using those collections can make your applications significantly more powerful.

## 2.5 Files

Now that we understand some different data types in Python and we have a rudimentary understanding of collections, we are poised to start doing some really interesting things such as discovering topics that will help you to build robust Python applications.

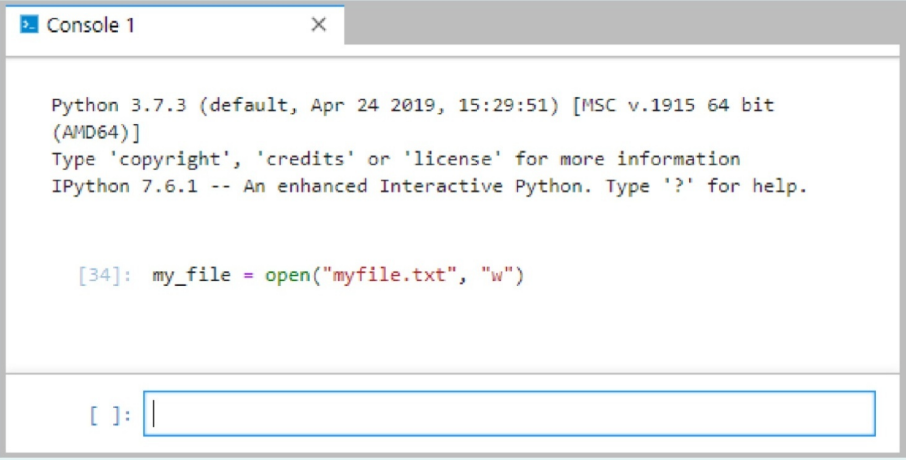
One feature that a lot of powerful applications have is persistent data. What does that mean? Well, imagine typing a document with a word processor. What if every time you used that word processor, you had to create new documents and you could never load saved documents? It would be cumbersome to have to recreate your document every time you wanted to modify it. Instead of putting you through that agony, word processors, like most powerful applications, allow you to save data (in this case, word processing documents) to the hard disk or other permanent storage so you can retrieve that document later. Saving data to files (or some other storage mechanism) and retrieving that data is a critical part of most applications.

Let us look into how you might write data to a file in Python. You can open files in Python by using a method called “open,” specifying a filename and an “x” for “create,” an “a” for “append,” a “w” for “write,” or an “r” for “read”. What do each of those options do?

- “Read” opens the file for reading. Since this is the default mode for file access, it does not have to be specified explicitly.
- “Create” will create a new file with the specified file name. If the file already exists, you will get an error message.
- “Append” will create a new file if one does not exist. If the file does exist, the program will not cause an error but will instead open the file. Anything you write to the file in append mode will be written to the end of the file, keeping any existing file contents safe.
- “Write” will create a new file if one does not exist and open it if it does exist (without causing an error). Unlike append mode, write mode will erase all contents in the existing file and start writing at the beginning of the file.

The open method returns a file object or variable that you need to use in order to write to the file, so you need to use the assignment operator (=) to assign the value from the open method to a variable as follows:

**Figure 60: Python - File Open**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

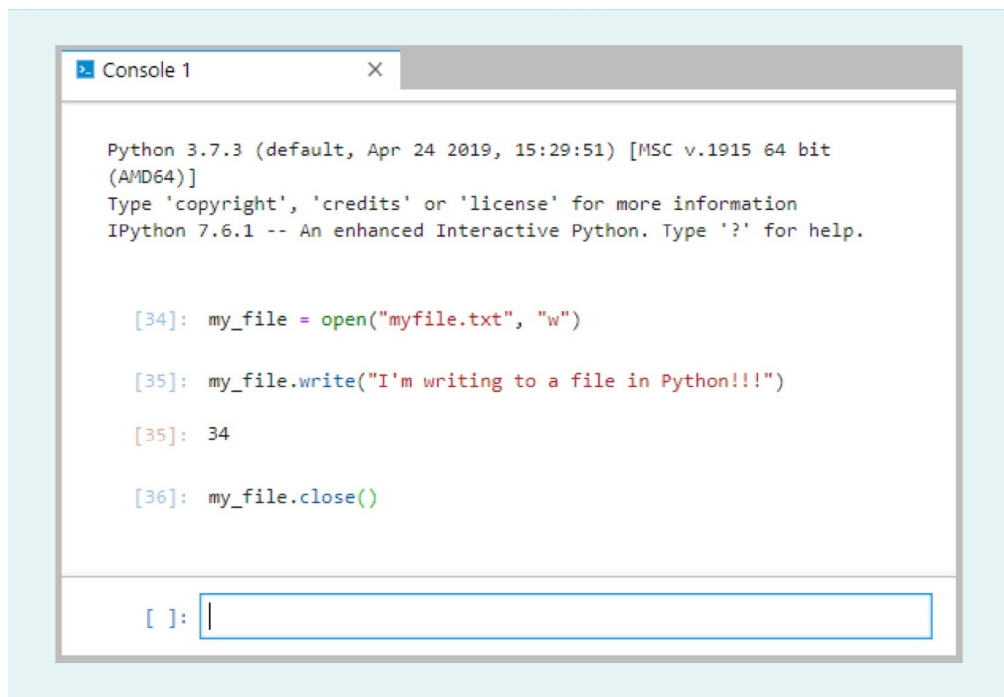
[34]: my_file = open("myfile.txt", "w")

[ ]: |
```

Source: Aaron Reed, 2020.

Now that the file is open, you can write whatever you want to the file. After writing, close the file with the close method. See below:

Figure 61: Python - Writing to a File

A screenshot of a Python console window titled "Console 1". The window shows the Python 3.7.3 startup banner, followed by three lines of code and their outputs. The code opens a file named "myfile.txt" in write mode, writes the string "I'm writing to a file in Python!!!", and then closes the file. The output for the write method is the number 34.

```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[34]: my_file = open("myfile.txt", "w")

[35]: my_file.write("I'm writing to a file in Python!!!")

[35]: 34

[36]: my_file.close()

[ ]: |
```

Source: Aaron Reed, 2020.

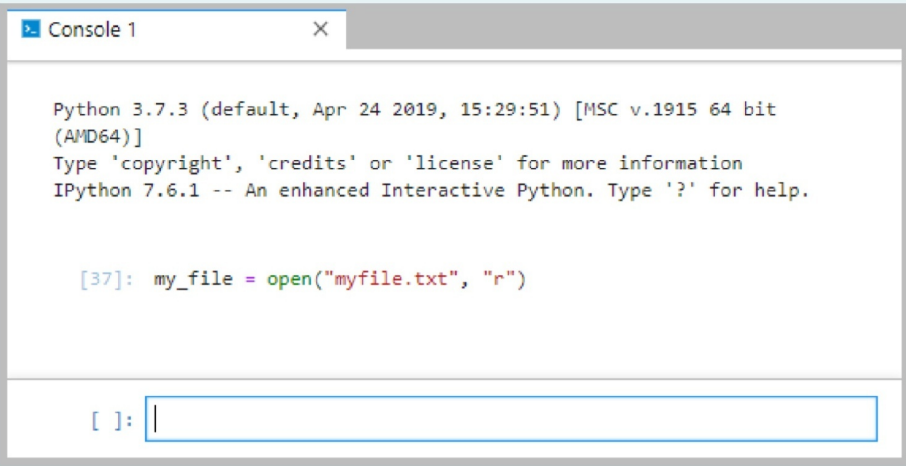
Congratulations! You’ve just written to your first file in Python! Notice the return value from line 35. The write method returned the number 34.

The 34 returned from the write method represents the number of characters successfully written to the file. If you count the characters in the string we wrote to the file, you will find that it is 34 characters long, so it looks like the write method worked! Can you think of a reason why a file write might not work?

Have you ever tried to write to a file in a word processor or some other software tool and found that you could not write to it because it was open in some other application? That is called a “file lock”; developers use these to prevent two (or more) applications from writing to the same files at the same time. A file lock is one possible reason why writing to a file might fail. The bottom line is that we are in control of the code we write. We can make it safe and secure and feel good about it when we go home at night. But the file system is something that is out of our control — we didn’t write it. In this simple exercise, we were just using the file system to gain access to a file, open it, write to it, and then close it. Whenever you start using resources that are out of your control, problems may happen. Later on, you’ll learn about ways to deal with potential problems like this through mechanisms for handling exceptions. For now, however, let us assume our files are ready for us to use!

Now that we have written to a file, let us see if we can read from it and get the text back out of it. To read from a file, you first need to open the file just as you did when you wrote to the file. However, the option you need to specify when reading a file is “r” for read, as shown here:

**Figure 62: Python - Open File for Reading**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

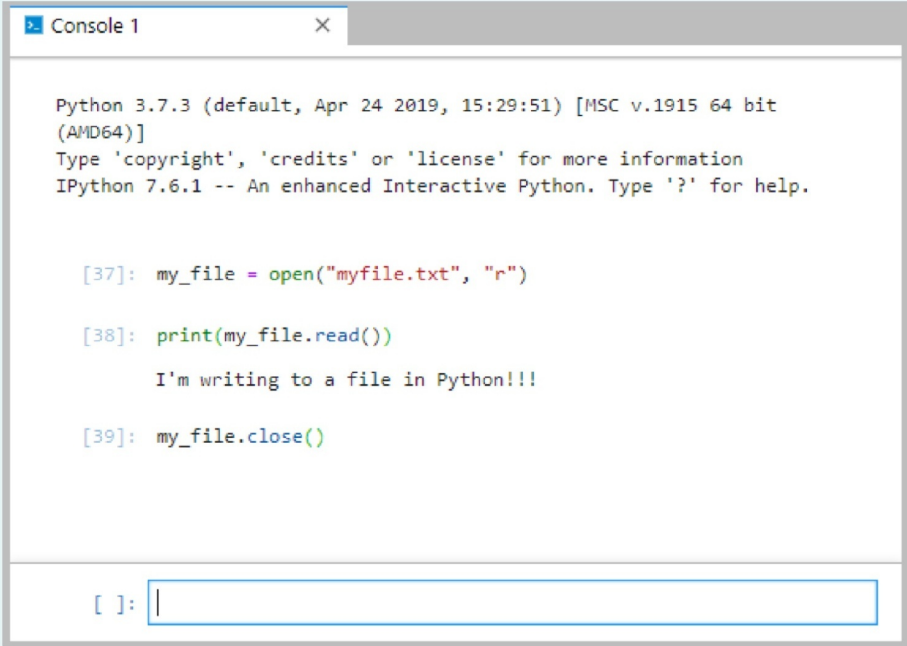
[37]: my_file = open("myfile.txt", "r")

[ ]: |
```

Source: Aaron Reed, 2020.

Now that the file is open for reading, simply use the read method to read the contents of the file. When you are finished, use the close method to close the file after reading. See below:

Figure 63: Python - Reading from a File



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[37]: my_file = open("myfile.txt", "r")

[38]: print(my_file.read())
      I'm writing to a file in Python!!!

[39]: my_file.close()

[ ]: |
```

Source: Aaron Reed, 2020.

Nice work! Now that you know how to read and write to a file, you are ready to take the next step in your Python development training. This chapter covered a wide range of topics that are essential for Python developers. Congratulations for getting through it.

### SUMMARY

Python is a very powerful and dynamic language. It supports a wide range of data types, such as strings, integers, floating-point numbers, and even complex and imaginary numbers.

Variables can be assigned using the assignment operator (=). The assignment operator always processes right to left, meaning that the right-hand side of the operator is evaluated first and then the resulting value is assigned to the variable on the left-hand side of the operator.

There are a wide range of operators in Python that can be used for arithmetic and manipulation of strings and numbers. In addition, a large number of collection data types, such as sets, lists, and dictionaries, can be used to create robust data sets.

Opening, reading to, and writing to files in Python is fairly straightforward and uncomplicated. A challenge exists anytime you use an external resource like a file system because that resource is not fully in your control, which creates opportunity for problems and errors.





# UNIT 3

## STATEMENTS

### STUDY GOALS

On completion of this unit, you will have learned ...

- how to use basic assignments and expressions in Python.
- how and when to use various conditional statements.
- how loops work and how to implement them.
- the basics of iterators and list comprehensions.

## 3. STATEMENTS

### Case Study

Kyle and Morgan are well on their way to creating the app for their new business: a program that stores all kinds of demographic and game-related data about soccer players and then uses that data to suggest areas for coaching and improvement. From the last chapter, they now know how to store that data with variables of different data types, how to make lists of data, and even how to store the data in a file. But what is next?

They need to start thinking about how to write their application. They have figured out how to store data, but now they need to figure out what to do with that data. They need to be able to loop through data, display data, and make decisions based on the data. But how can they do that? Here are some questions that Kyle and Morgan are currently considering:

- How do you make a program take one course if data contains a certain value but another course if the data contains a different value?
- How do applications loop through lists of data?

### 3.1 Assignment and Expressions

In the previous chapter we looked at ways to assign values to variables. We discussed a number of different data types including integers, floating point numbers, strings, and complex numbers. We also talked about the assignment operator as a mechanism for assigning a value to a variable. Remember, the assignment operator works from right to left, meaning that it first calculates what's on the right-hand side of the operator (the = sign) and then assigns that value to the item on the left-hand side of the operator. For example, in the assignment  $a = 4 + 5$ , the right-hand side of the equation, the  $4 + 5$ , will be evaluated first. The result will be the number 9, which will then be assigned to the variable  $a$ . Because the assignment operator expects a single variable on the left-hand side of the equation, something like  $4 + 5 = a$  is an invalid statement in Python. The interpreter will see this statement and attempt to resolve the right-hand side of the equation first. That part is trivial as it would simply retrieve the value currently assigned to the variable  $a$ . It would then attempt to assign that value to whatever is on the left-hand side of the equation. In this case,  $4 + 5$  is on the left-hand side of the equation and those are both literals that cannot be reassigned, which will result in a syntax error from the interpreter.

There are a few other assignment operators we should discuss. In addition to the = sign, which simply assigns the value from the right to the variable on the left, there are special operators that calculate a new value and assign the value simultaneously. The following is a list of assignment operators:

**Table 6: Python Assignment Operators**

Operator	Syntax	Description
=	a = 5	Assigns the value on the right to the variable on the left.
+=	a += 5	Takes the value currently stored in a, adds the value on the right to it, and assigns the new value back to a. This statement is equivalent to: a = a + 5
-=	a -= 5	Takes the value currently stored in a, subtracts the value on the right from it, and assigns the new value back to a. This statement is equivalent to: a = a - 5
*=	a *= 5	Takes the value currently stored in a, multiplies it by the value on the right, and assigns the new value back to a. This statement is equivalent to: a = a * 5
/=	a /= 5	Takes the value currently stored in a, divides it by the value on the right, and assigns the new value back to a. This statement is equivalent to: a = a / 5
%=	a%= 5	Takes the value currently stored in a, computes the modulus of it and the value on the right (remember, the modulus, or mod, is the remainder left after a division operation), and assigns the new value back to a. This statement is equivalent to: a = a% 5
**=	a **= 5	The ** operator is an exponent operator. a ** 2 gives you a to the power of 2, or a <sup>2</sup> . The **= operator takes the value currently stored in a and raises it to the power of the value on the right, then assigns that value back to a. This statement is equivalent to a = a ** 5

Source: Aaron Reed, 2020.

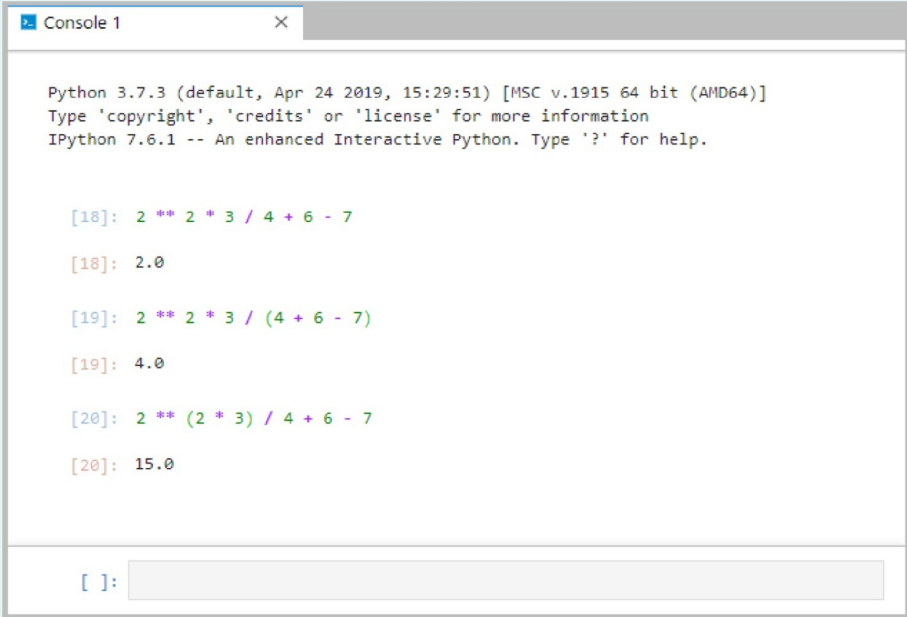
Go ahead and experiment with some of the operators listed above in your Jupyter Lab terminal to get a feel for how they function.

Just like in the mathematics you learned in school, there is an order of operations in Python when calculating the value of an **expression**. This order is based on the precedence of certain operators over others. An easy way to remember this order is by the acronym PEMDAS, which stands for parentheses, exponents, multiplication, division, addition, and subtraction. For example, the following similar expressions evaluate to different values depending on where we decide to place parentheses:

**Expression**

In Python, the combination of literals, variables, and operations that evaluates to a value is known as an expression. For example, a + 5 is an expression.

Figure 64: Python PEMDAS



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[18]: 2 ** 2 * 3 / 4 + 6 - 7
[18]: 2.0

[19]: 2 ** 2 * 3 / (4 + 6 - 7)
[19]: 4.0

[20]: 2 ** (2 * 3) / 4 + 6 - 7
[20]: 15.0

[ ]: 
```

Source: Aaron Reed, 2020.

Let's take a look at what is happening here. In the first expression, there are no parentheses, so we jump right to the exponent and calculate 2 to the power of 2, which is 4. Then we multiply that by 3, getting 12. Then we divide that by 4, resulting in 3, then we add that to 6, which gives us 9, and finally we subtract 7 from 9 and we end up with 2.

In the second example, there are parentheses around the  $4 + 6 - 7$  so we calculate that first, resulting in 3. We then jump over to the exponent and calculate 2 to the power of 2, which is 4. We multiply that by 3, giving us 12. Finally, we divide 12 by 3 (the value from the calculations in parentheses), and we end up with 4.

In the third example, again we have parentheses, so we calculate  $2 \cdot 3$  first, giving us 6. We then take 2 to the power of 6, which gives us 64. Now we divide that by 4, giving us 16, we add 6 to get 22, and subtract 7 to end up with 15.

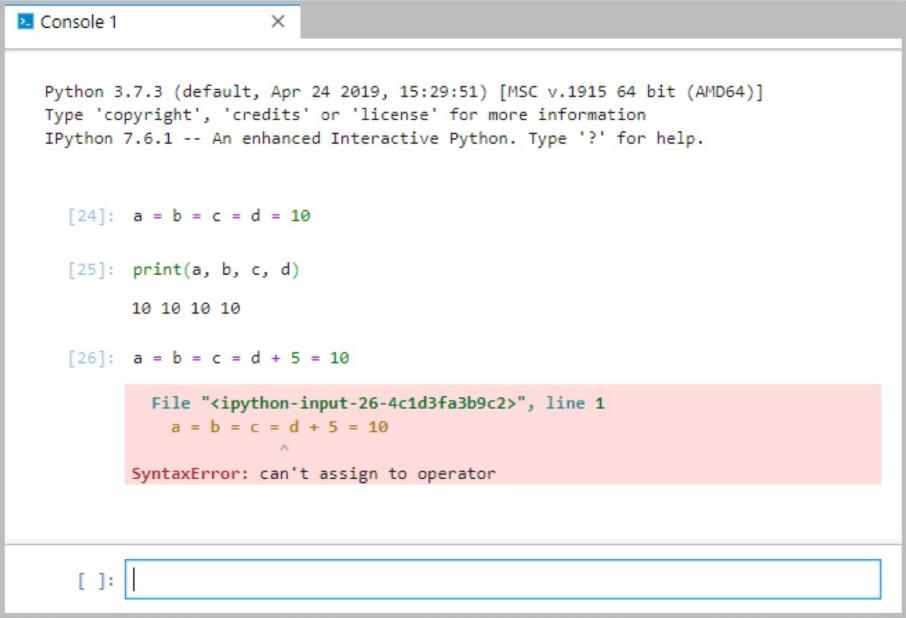
Experiment with the various operators and parentheses to get a feel for how they work together. Remembering the acronym PEMDAS can be helpful as you work through the order precedence.

### Chained Assignment

There may be times when you need to set several variables to the same value. This can be done using something called a "chained assignment." In a chained assignment, you use the assignment operator between multiple variables to set them all to the same value in one statement, like this:  $a = b = c = 1$ .

Effectively, this is equivalent to using the three separate statements `a = 1`, `b = 1`, and `c = 1`. The example below illustrates how chained assignment can be used:

**Figure 65: Python Chained Assignment**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[24]: a = b = c = d = 10

[25]: print(a, b, c, d)
      10 10 10 10

[26]: a = b = c = d + 5 = 10

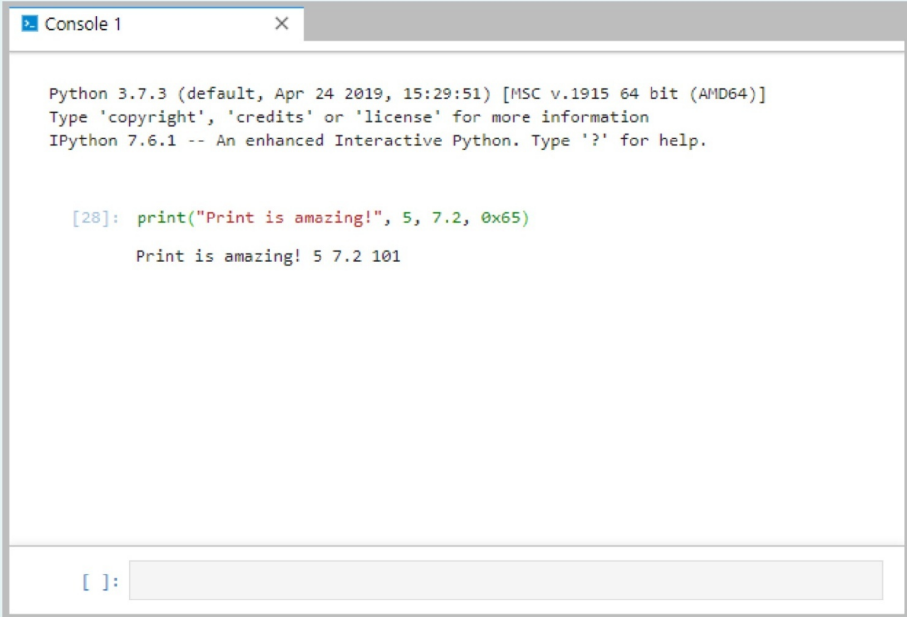
File "<ipython-input-26-4c1d3fa3b9c2>", line 1
      a = b = c = d + 5 = 10
                        ^
SyntaxError: can't assign to operator
```

Source: Aaron Reed, 2020.

The first statement above takes the value 10 and assigns it to `d`, `c`, `b`, and `a`. Note that in the second statement, we print out each of those values and you can see they all contain the value 10. Next, however, look at the third statement. By adding a `d + 5` to the chained assignment, we get a syntax error from the interpreter. Why is that? Well, consider that the first statement breaks down to the equivalent of using four distinct statements: `a = 10`, `b = 10`, `c = 10`, and `d = 10`. The same is true of the third statement except, with the added change, the four distinct statements are: `a = 10`, `b = 10`, `c = 10`, and `d + 5 = 10`. As we know, the assignment operator expects only one variable to be on the left-hand side of the equation, so `d + 5 = 10` will yield a syntax error. When using chained assignments, only a single variable between each assignment operator is allowed.

You also may have noticed the different usage of the `print` command in the previous example. Until now, we've used `print` to output a single value to the console. Here, we see that to output multiple values, you simply separate those values with commas when using the `print` command. This can be handy when trying to output a lot of text to the screen. You can even combine multiple data types in a single call to `print` as shown below:

Figure 66: Python Print



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

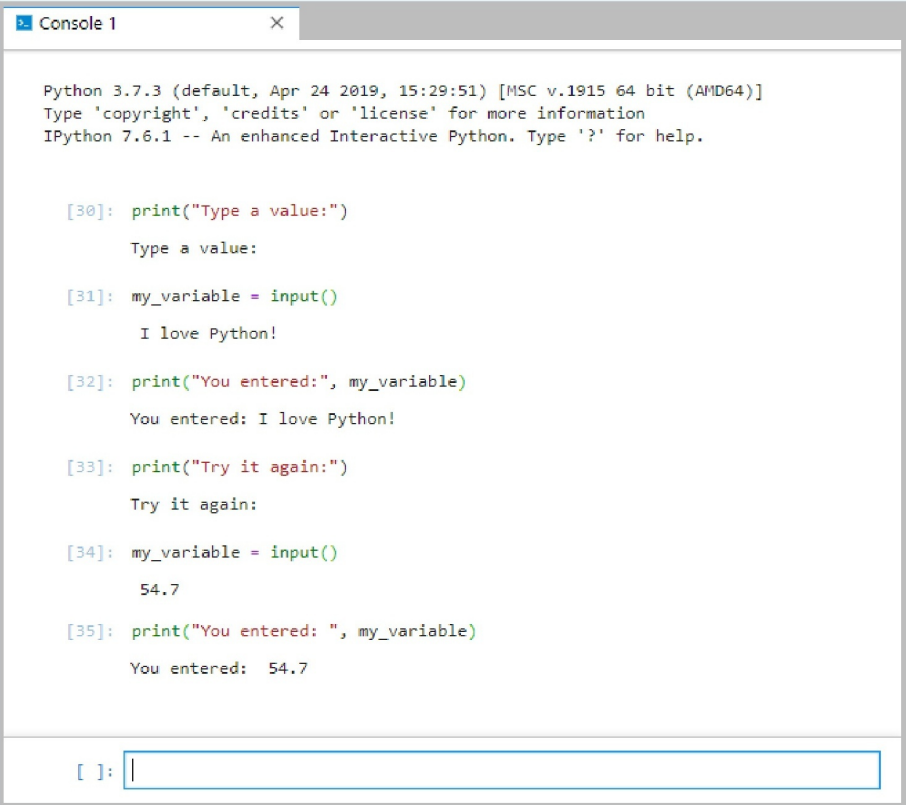
[28]: print("Print is amazing!", 5, 7.2, 0x65)
      Print is amazing! 5 7.2 101

[ ]:
```

Source: Aaron Reed, 2020.

One final note before we move on to the next section: everything we've done thus far has consisted of us creating variables and assigning the value of that variable in code. What if we wanted to let the user choose the value that will be assigned to a variable? To get input from a user, use the input function. The input function takes no parameters and can be used with the assignment operator to read a value from the user via the keyboard and then assign that value to a variable. See an example below:

Figure 67: Python Input



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[30]: print("Type a value:")
      Type a value:

[31]: my_variable = input()
      I love Python!

[32]: print("You entered:", my_variable)
      You entered: I love Python!

[33]: print("Try it again:")
      Try it again:

[34]: my_variable = input()
      54.7

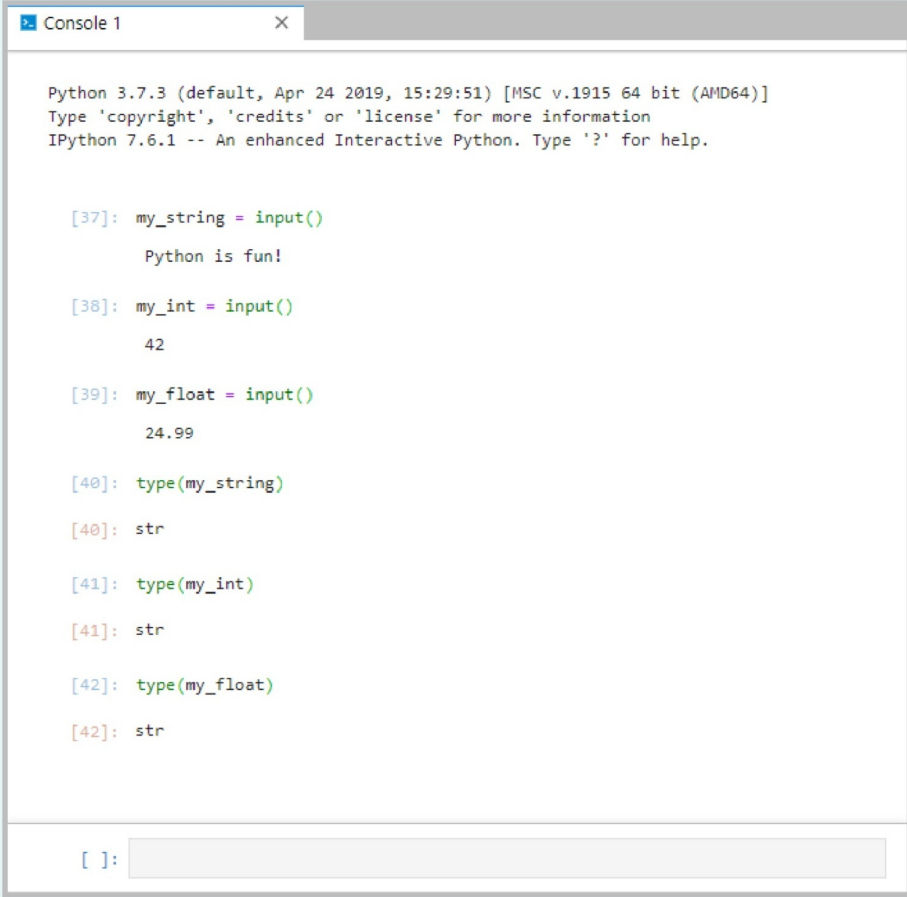
[35]: print("You entered: ", my_variable)
      You entered: 54.7

[ ]: |
```

Source: Aaron Reed, 2020.

Note that via the input command, you can enter numerical and string data. However, let's look a little bit deeper. You can check the type of a variable by using the type function. What happens when we look at the type of a variable read from the user via the input function:

Figure 68: Python More Input



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[37]: my_string = input()
      Python is fun!

[38]: my_int = input()
      42

[39]: my_float = input()
      24.99

[40]: type(my_string)
[40]: str

[41]: type(my_int)
[41]: str

[42]: type(my_float)
[42]: str

[ ]:
```

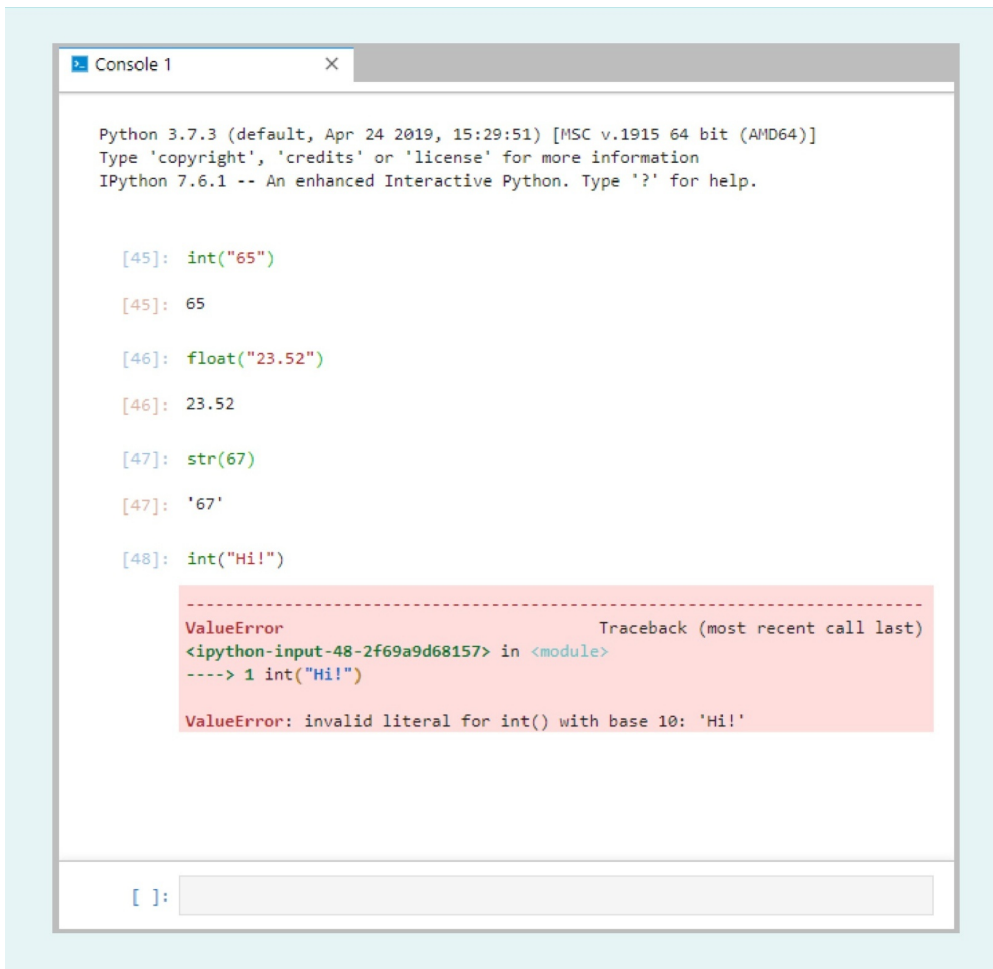
Source: Aaron Reed, 2020.

So, what’s going on here? Why are all three inputs coming back as a type string when we clearly entered a string, an integer, and a floating-point number? Well, the Python interpreter can’t tell what a user intends when they enter data for the input function. When a user enters 42 into the console, it could be that they intended to enter the integer 42. However, it could also be that they intended to enter a string consisting of the characters “42.” But it could also be that they intended to enter a floating-point number with no value after the decimal (42.0). We really have no idea. So, to be safe, Python’s input function always reads data as a string.

Luckily, it’s easy to deal with different data types. If we want to be dealing with an integer, we can convert the data to an integer by using the `int` function. Likewise, we can convert it to floating-point by using the `float` function. We can convert it back to a string using the `str` function. Just be aware that if the data does not match the type of data that you are trying to convert the value to, you will get an error. For example, you cannot convert the string “Hi!” to a number. See below for examples:



Figure 69: Python Type Conversion



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[45]: int("65")
[45]: 65

[46]: float("23.52")
[46]: 23.52

[47]: str(67)
[47]: '67'

[48]: int("Hi!")

-----
ValueError                                Traceback (most recent call last)
<ipython-input-48-2f69a9d68157> in <module>
----> 1 int("Hi!")

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

[ ]:
```

Source: Aaron Reed, 2020.

Now that we understand a little bit more about expressions and assignment operators as well as print, input, and type conversion, we're poised to move on to conditional statements and ways to parse and evaluate data in Python.

## 3.2 Conditional Statements and Expressions

A lot of times you need your program to execute one section of code based on one condition and another section of code based on another condition. For example, in the soccer application Kyle and Morgan are creating, imagine they want to print out a list of players on the roster, but they want to somehow designate the projected starters for a game with some additional text or other indicator. How would they do that? How can you print just a name in one scenario but then print the name and some indicator in another?

Enter the if statement. An if statement is a conditional statement that will evaluate an expression and, if the expression evaluates to True, the code that follows the if statement will be executed. If the expression does not evaluate to True, the code that follows the if statement will be skipped.

Before we dig into the if statement, let us look deeper into the concept of true/false expressions. Note the difference between a statement and an expression: a statement is a line of code that the interpreter can execute, but an expression is a section of code that the interpreter evaluates to a certain value. A boolean expression is one that evaluates to true (represented in Python by True) or false (represented in Python by False). Just like operators exist to assign values to variables, other operators exist to facilitate Boolean expressions. Those operators, called **comparison operators**, in Python are as follows:

**Comparison Operators**

These are operators that facilitate Boolean expressions. These expressions evaluate to either True or False.

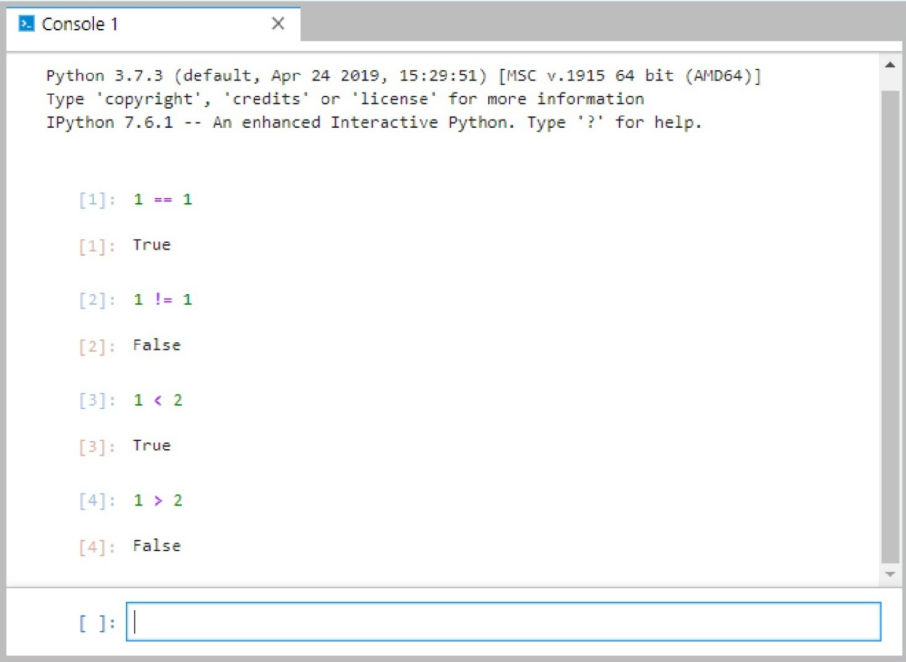
**Table 7: Python Comparison Operators**

Operator	Description	Usage
==	Equal. Checks for equality between the left and right sides of the operator. Returns True if equal, False if not.	a == b
!=	Not equal. Checks for non-equality between the left and right sides of the operator. Returns True if not equal, False if equal.	a != b
>	Greater than. Checks for a greater-than condition. Returns True if the left side is greater than the right, False otherwise.	a > b
>=	Greater than or equal. Checks for a greater-than-or-equal condition. Returns True if the left side is greater than or equal to the right, False otherwise.	a >= b
<	Less than. Checks for a less-than condition. Returns True if the left side is less than the right, False otherwise.	a < b
<=	Less than or equal. Checks for a less-than-or-equal condition. Returns True if the left side is less than or equal to the right, False otherwise.	a <= b

Source: Aaron Reed, 2020.

See the following figure for some of these operators in action:

**Figure 70: Python Comparison Operators in Action**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[1]: 1 == 1
[1]: True

[2]: 1 != 1
[2]: False

[3]: 1 < 2
[3]: True

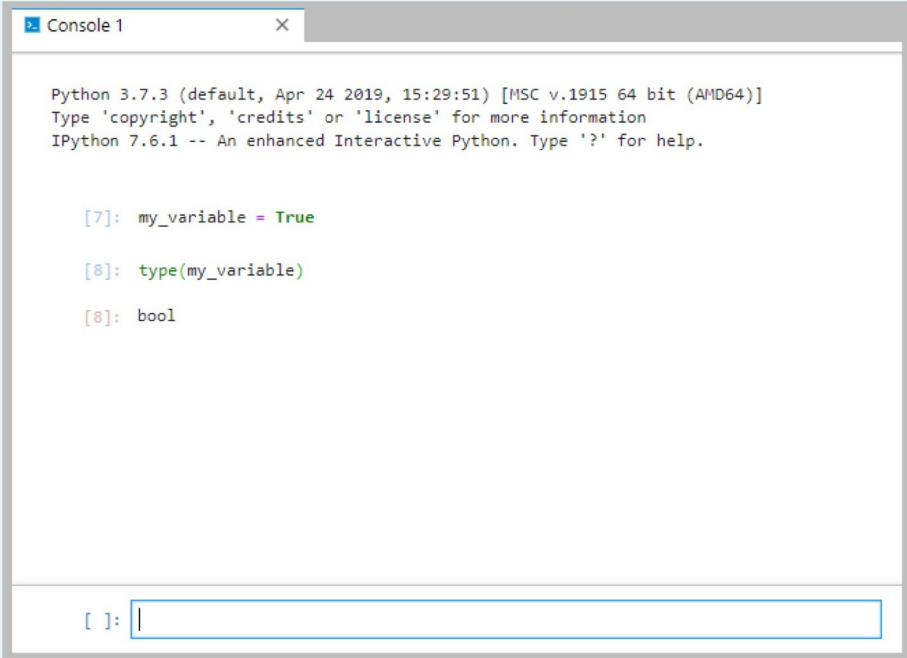
[4]: 1 > 2
[4]: False

[ ]: |
```

Source: Aaron Reed, 2020.

These True/False values actually represent a new data type called “bool”, which can hold only one of two Boolean values: True or False.

**Figure 71: Python Boolean Data Type**



```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

[7]: my_variable = True

[8]: type(my_variable)

[8]: bool

[ ]: |
```

Source: Aaron Reed, 2020.

So, we now know how to create a Boolean expression using comparison operators. Let's go back to the if statement concept and look at how those work.

### **If Statements**

As we described earlier, an if statement evaluates a conditional expression. If the expression evaluates to True, the code following the if statement will be executed. If the expression evaluates to False, the code following the if statement will be skipped.

To see this in action, let's move from the console in Jupyter Lab and start working in a Python notebook in Jupyter Lab. If you have a console window open in Jupyter Lab, close the console window and click on the Python 3 Notebook button to open up a Python 3 notebook. Once you have an open notebook, enter the following code in the first cell in that notebook:

**Figure 72: Python - If Statement**

```
print("Enter a player's name")
player_name = input()
print("Is this player a starter? Enter \'Y\' for yes and \'N\' for no.")
player_starter = input()
print("The player is:")
print(player_name)
if player_starter == "Y":
    print("(Starter)")
```

Source: Aaron Reed, 2020.

Take a look at the code above, and try to figure out what it's going to do.

The code first outputs some text directing the user to enter a player's name. Then, using the `input()` function, the second line of code asks the user to enter something via the keyboard, which will be stored in the variable "player\_name." The third line sends more text to the screen telling the user to type "Y" if the player is a starter and "N" if they are not. The fourth line then also uses the `input()` function to ask the user to type something, the value of which will be stored in a variable called "player\_starter." The fifth line of code outputs more text to the screen and the sixth line outputs the contents of the `player_name` variable to the screen. The next line is your first if statement. Note the format of that statement. It starts with the keyword "if," which is followed by a conditional expression using the `==` operator to compare the contents of the variable "player\_starter" with the string "Y." The if statement ends with a colon (:). The colon indicates that anything that follows the if statement in an indented code block is executed when the condition is true.

Below the if statement, you see a line of code that outputs the text "(Starter)" to the screen. This is the text that we will use to designate whether a player is a starter on the roster. Note that the line of code is indented four spaces compared to the rest of the code in this cell. That's how Python knows that the code is part of the True block for the if statement above, meaning that if the expression in the if statement evaluates to True, the indented code will be executed. If the expression evaluates to False, the indented code will be skipped.

You can have as many lines of code as you want in the True block of an if statement, and they will all be executed in order if the Boolean expression evaluates to True. The trick is in the indentation. Python standard is to indent four spaces for an if statement. You can indent more or less than that if you want, but we recommend sticking to the standards. Note that all lines of code in the True block of the if statement must be indented at the same level (e.g., you can't indent one line four spaces and another line five spaces; each line of code must line up with the rest).

Run the code by clicking the Run button at the top of the notebook. Depending on what you enter in the prompts, you may or may not see the "(Starter)" text. Here are a couple of examples, one showing the conditional expression in the if statement evaluating to True and another showing it evaluating to False:

**Figure 73: Python - If Statement Condition True**

```
print("Enter a player's name")
player_name = input()
print("Is this player a starter? Enter \'Y\' for yes and \'N\' for no.")
player_starter = input()
print("The player is:")
print(player_name)
if player_starter == "Y":
    print("(Starter)")
```

```
Enter a player's name
Tyler Huntley
Is this player a starter? Enter 'Y' for yes and 'N' for no.
Y
The player is:
Tyler Huntley
(Starter)
```

Source: Aaron Reed, 2020.

**Figure 74: Python - If Statement Condition False**

```
print("Enter a player's name")
player_name = input()
print("Is this player a starter? Enter \'Y\' for yes and \'N\' for no.")
player_starter = input()
print("The player is:")
print(player_name)
if player_starter == "Y":
    print("(Starter)")
```

```
Enter a player's name
Jason Shelley
Is this player a starter? Enter 'Y' for yes and 'N' for no.
N
The player is:
Jason Shelley
```

Source: Aaron Reed, 2020.

In the first example above, the `player_starter` variable is equal to "Y," so the expression evaluates to True and the "(Starter)" text is outputted to the screen. In the second example, the `player_starter` variable is not equal to "Y," so the expression evaluates to False and all indented lines of code after the if statement (in this case there is just one line) are skipped. In these examples, there is no other code after the if statement, but if there were other lines of code, execution of the program would resume there after the if statement was executed.

Let's look at another example to see multiple lines of code within the True block of an if statement and to see what additional lines of code after the if statement might look like:

**Figure 75: Python – Another If Statement**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
print("We're done!")
```

Source: Aaron Reed, 2020.

In the code listed above, the if statement block contains two lines of code: one that prints a congratulatory message telling the user the number they entered is less than 10 and another line that prints the number itself. If the conditional expression `int(my_number) < 10` evaluates to True, both lines of code will be executed. If the expression evaluates to False, both lines of code will be skipped and execution will resume at the following line, which is the final line of the program. Note the conversion of “my\_number” to an integer by using the `int()` function. Remember that the `input()` function returns a string, so the contents of the `my_number` variable will actually be of a string data type. In order to compare it to the number 10, we have to convert it to an integer by using the `int()` function.

Here are two examples of that code running, one where the expression evaluates to True and one where it evaluates to False:

**Figure 76: Python - Another If Statement Evaluating to True**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
print("We're done!")
```

```
Enter a number
15
We're done!
```

Source: Aaron Reed, 2020.

**Figure 77: Python - Another If Statement Evaluating to False**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
print("We're done!")
```

```
Enter a number
8
Your number is less than 10!
8
We're done!
```

Source: Aaron Reed, 2020.

## Else Statements

Let's take a look at that last example again. If the number entered by the user is less than 10, we output some special text to the user in the if statement block. But what if we also wanted to output other text to the user if the number is not less than 10? We could do something like the following:

**Figure 78: Python — Two If Statements**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
if int(my_number) >= 10:
    print("Your number is not less than 10!")
    print(my_number)
print("We're done!")
```

Source: Aaron Reed, 2020.

In this code we have two if statements: one that executes if `my_number` is less than 10 and one that executes if it is not less than 10 (or, if it is greater than or equal to 10). However, this solution is somewhat clunky; we can accomplish the same thing more efficiently by using the else statement.

When used, an else statement always follows an if statement; you cannot use else without a corresponding and preceding if statement. As you now know, the code immediately following an if statement executes if the if statement's expression evaluates to True. In con-



trast, the code immediately following an else statement executes if the corresponding if statement's expression evaluates to False. The code below illustrates a more effective solution to the two if statements above:

**Figure 79: Python - Else**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
else:
    print("Your number is not less than 10!")
    print(my_number)
print("We're done!")
```

Source: Aaron Reed, 2020.

Notice the format of the else statement. First, it comes immediately after the block of code from the preceding if statement. Next, notice that it also ends with a colon (:). Finally, notice that, just like the if statement, a block of indented code follows the else statement. If the conditional expression in the if statement evaluates to True, the block of code following the if statement is executed and the block of code following the else statement is skipped. However, if the conditional expression evaluates to False, the block of code following the if statement is skipped and the block of code following the else statement is executed.

The images below show the program in execution, first where the conditional expression in the if statement evaluates to True, and second where it evaluates to False:

**Figure 80: Python — Another If True**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
else:
    print("Your number is not less than 10!")
    print(my_number)
print("We're done!")
```

```
Enter a number
5
Your number is less than 10!
5
We're done!
```

Source: Aaron Reed, 2020.

**Figure 81: Python - If False**

```
print("Enter a number")
my_number = input()
if int(my_number) < 10:
    print("Your number is less than 10!")
    print(my_number)
else:
    print("Your number is not less than 10!")
    print(my_number)
print("We're done!")
```

```
Enter a number
100
Your number is not less than 10!
100
We're done!
```

Source: Aaron Reed, 2020.

### **Elif Statements**

There may be times when you will want to execute code based on more than just a true/false evaluation of a single statement. What if, for example, we wanted to evaluate whether a number was negative, 0, or positive? The elif statement can help in such a scenario. Short for “else-if,” the elif statement lets you add an additional conditional expression to be evaluated after an if statement. Like the else statement, an elif requires a preceding if statement. If the conditional expression of the if statement evaluates to False, the elif statement’s conditional expression is evaluated. If the elif statement evaluates to True, the block of code following the elif statement will be executed. See below for an example of an elif statement:

**Figure 82: Python - Elif**

```
print("Enter a number")
my_number = int(input())
if my_number < 0:
    print("Your number is negative.")
elif my_number == 0:
    print("Your number is zero.")
else:
    print("Your number is positive.")
print("Thanks for playing!")
```

Source: Aaron Reed, 2020.

In the example above, the user inputs a number. Notice that we have changed our approach and we are now converting the input to an integer in line two. That will cause the data type for the my\_number variable to be an integer, meaning that we won’t have to continue to convert the variable to test it against different numerical values.

In the if statement, the `my_number` variable is checked to see if it is less than 0. If it is, the code below the if statement will be executed and both the elif and else will be skipped. If `my_number` is not less than 0, the elif will be executed and the variable will be checked to see if it is equal to zero. If it is equal to zero, the code below the elif will be executed and the code below the else statement will be skipped. If the variable is not equal to zero, the code below the else statement will be executed.

You can have multiple elif statements evaluating a chain of conditional expressions. For example, the code below compares the size of a string read from the user and outputs a phrase based on the length of that string. To do so, the code uses an initial if statement, followed by four elif statements, and ending with an else statement:

**Figure 83: Python - Lots of Elif Statements**

```
print("Enter a string:")
my_str = input()
if len(my_str) <= 1:
    print("That's a very short string!")
elif len(my_str) < 5:
    print("That's a fairly short string.")
elif len(my_str) < 25:
    print("That's a good sized string.")
elif len(my_str) < 50:
    print("That's a fairly long string.")
elif len(my_str) < 100:
    print("That's a long string.")
else:
    print("That's a very long string!")
```

Source: Aaron Reed, 2020.

Please note that the else part of the if statement is optional. Thus, if you want your if statement to do nothing whenever neither the if or any elif condition is valid, you can simply leave out the else part.

## 3.3 Loops

As you begin programming, you will find that there are a lot of times when you need to **loop** over a certain set of values or a range of numbers to accomplish a task multiple times. Python has a wide range of options to accommodate this. First, let us look at looping with the range function. The range function will create a range of numbers based on a set of parameters. Those parameters are as follows:

- **Start:** This is an optional numerical parameter that tells you the range function at which number you want to start the set of numbers. If you omit this parameter, the starting point for the range is the number 0.

### **Loop**

One of the most important and powerful tools for programming, a loop will repeat the same set of instructions repeatedly until a specified condition is met.

- Stop: This is the only required parameter. It is also numerical and tells the range function at which number you want to stop the set of numbers. This number will not be included in the set of numbers, but the set will include all numbers until it reaches this one.
- Step: This is an optional numerical parameter that tells the range function how many numbers to skip between each number in the set. If you omit this parameter, the range function will use the number 1.

In the code below, only the required stop parameter is used:

**Figure 84: Python — Range with Stop Parameter Only**

```
for x in range(10):  
    print(x)
```

Source: Aaron Reed, 2020.

Let's take a closer look at the above code. First, note the format of the call to `range(10)`. Remember, the only required parameter in the range function is the stop parameter, so in this case, the start parameter defaults to 0, meaning a set of numbers will be generated starting at 0. The stop parameter is 10, so everything from the start parameter (in this case 0) up to but not including 10 will be in the set of numbers. Finally, since no step parameter is present, the default is 1, meaning we will be counting in stages of 1 in this set of numbers. Hence, we would anticipate the range to include the numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Now, let us look at the for loop in the code above. The for loop lets you loop through all the numbers in a set created by range. The syntax is as follows:

### Code

```
for <variable> in <range>
```

Whatever you put in place of the `<variable>`, which in our code example was the letter `x`, will be created as a variable to temporarily hold the value of each number in the range as you loop through it. Finally, notice that just as you did with `if`, `elif`, and `else`, you end the for statement with a colon (`:`). This colon denotes that the following block of indented code is related to the for statement.

The for statement will loop through the indented code block following the for statement one time for each number in the range. Each time through the loop, the variable `x` will hold the value of the next number in the range sequence. Hence, when we print the value `x` each time through the loop, we see the numbers we expected in the range (0–9) outputted to the screen. See below:

**Figure 85: Python — Range(10) Executed**

---

```
for x in range(10):
    print(x)
```

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

Source: Aaron Reed, 2020.

If we add a start parameter, the range will begin at the parameter specified instead of zero. Below, you will see an example where we create a range from 5 to 9:

**Figure 86: Python - Range with Start and Stop Parameters**

---

```
for x in range(5, 10):
    print(x)
```

5  
6  
7  
8  
9

Source: Aaron Reed, 2020.

The step parameter is the third parameter in the range function; it defaults to 1. If we pass in 2 for the parameter, the range will be created by starting with the first number then including every second number. The code below loops through a range from 5 to 9, in steps of two:

**Figure 87: Python - Range with Start, Stop, and Step Parameters**

```
for x in range(5, 10, 2):  
    print(x)
```

```
5  
7  
9
```

Source: Aaron Reed, 2020.

That's all really fun and all, but why in the world would we ever use something like this for\_range loop? Well, imagine we had a list of players and we wanted to output that list to the screen as a roster. Here's some code that would accomplish that by using a for\_range loop:

**Figure 88: Python - Printing a List with a For\_Range Loop**

```
my_players = ["Franz Beckenbauer", "Gerd Muller",  
              "Lothar Matthaus", "Manuel Neuer"]  
for x in range(4):  
    print("Player " + str(x+1) + ": " + my_players[x])
```

Source: Aaron Reed, 2020.

The code snippet above starts with the definition of a list called my\_players. One interesting thing to note here is that the definition of the list runs over onto two lines of code. That is totally fine and actually encouraged in Python, otherwise your text will run off the edge of the screen. Remember, Python is all about readability, and if your code can't all be visible on the screen without scrolling, it is not as easy to read.

Next, we start a for loop with a range of 0 to 3 by using range(4). Next, each time we go through the for loop, we print some text to the screen. There are a couple of things to note here. Let us look at the my\_players[x] code at the end of the print statement. Using a list or collection with brackets at the end is called "indexing." Essentially, it's a way of accessing a value or values within a list. Remember, we do a very similar thing with strings when trying to access part of the string. When indexing a list, you use the brackets and pass in an index to a value from the list you want to retrieve. Lists are zero-based, meaning the first item at the list is in index position 0, the second at index 1, the third at index 2, and so on. The index for this list is shown in the following figure:

**Figure 89: Python - Indexes for the my\_players list**

Index	Value
0	"Franz Beckenbauer"
1	"Gerd Muller"
2	"Lothar Matthauss"
3	"Manuel Neuer"

Source: Aaron Reed, 2020.

So, the expression `my_players[2]` will result in the value "Lothar Matthauss." It is very important to remember when traversing lists with indexes that the first item in the list is at index 0 and the last item in the list is at index  $n-1$ , where  $n$  is the total number of items in the list. Many programmers have been very stressed over the years with bugs related to this concept. So, the good news is, when you make mistakes on those indexes, you're not alone! The bad news is, you will make mistakes. Everybody does. But just remember, the first item is at index 0, the last at  $n-1$ .

That is why, when we traverse the list, we create a range from 0 to 3 instead of from 1 to 4: because 0-3 are the indexes in this list. But, since we're using `x` to represent the values from 0 to 3 in our loop in the code snippet above, when we print out `x`, we'll get the values 0 to 3. In other words, we'll be outputting something like this:

- Player 0: Franz Benckenbauer
- Player 1: Gerd Muller
- ... etc.

That's not exactly what we want, because it would be confusing to people. Python counts starting at zero, but people typically start with one; seeing "Player 0" would probably make people scratch their heads, wondering what's going on. So, in the middle of the print statement in our code, you'll see that as we print out `x`, we add one to it so the actual text will start with "Player 1," which is much easier for people to understand. The final result when we run that code looks like this:

**Figure 90: Python - Printing a List of Players**

```
my_players = ["Franz Beckenbauer", "Gerd Muller",  
             "Lothar Matthaus", "Manuel Neuer"]  
for x in range(4):  
    print("Player " + str(x+1) + ": " + my_players[x])
```

```
Player 1: Franz Beckenbauer  
Player 2: Gerd Muller  
Player 3: Lothar Matthaus  
Player 4: Manuel Neuer
```

Source: Aaron Reed, 2020.

## While Loops

Another popular method of looping through values in Python is the while loop. The while loop uses the keyword “while” followed by an expression. If the expression evaluates to True, the code attached to the while loop will be executed over and over. Once the expression evaluates to False, the while loop will exit and execution will resume at the point after the while loop and its associated code block.

Below is a code snippet where we print our list of players using a while loop:

**Figure 91: Python - Printing a List of Players Using a While Loop**

```
my_players = ["Franz Beckenbauer", "Gerd Muller",  
             "Lothar Matthaus", "Manuel Neuer"]  
x = 0  
while x < 4:  
    print("Player " + str(x+1) + ": " + my_players[x])  
    x += 1  
print("End of roster")
```

```
Player 1: Franz Beckenbauer  
Player 2: Gerd Muller  
Player 3: Lothar Matthaus  
Player 4: Manuel Neuer  
End of roster
```

Source: Aaron Reed, 2020.

The code above accomplishes the same thing as the for\_range code discussed previously. Take a minute and look at the differences. The creation of the list is exactly the same, as is the print statement. But in order to loop through this list using a while loop, we have to create a variable that we can use to represent the different indexes in the list. The x variable is created and assigned the value 0. Then, each time through the loop, the value of x is incremented by one. This is an absolutely critical step. Why? Remember, the while loop



runs forever until the expression it is evaluating evaluates to False. Since we've initialized the value of `x` to 0, if we never increment that value, `x` will never be greater than or equal to 4 and the while expression will never evaluate to False. What does that mean? The loop will run forever! This is called an infinite loop because, well, it will run an infinite number of times—forever! Just like mistakes made when accessing lists via indexes, infinite loops are another extremely common programming error. Be aware of these issues: whenever you create a while loop, make sure that the code block attached to it will ensure that it does not turn into an infinite loop.

If you do get stuck in an infinite loop in Jupyter Notebook, there is a square button on the toolbar, right next to the run button. This square button will halt execution of any code running on the kernel. That button will get you out of an infinite loop by shutting down the program. That button can be a programmer's best friend. Make sure you know where it is, and get familiar with how it works.

Let's talk about some other keywords that can be handy in loops. These keywords can be used in while and for loops alike. The first is "break." Using the break keyword in a loop will force the loop to exit regardless of the expression being evaluated to execute the loop. For example, let's say we wanted to print the list of players, but after player two, we'd like to exit the loop. Here is how we can accomplish that using the break keyword:

**Figure 92: Python - Break**

```
my_players = ["Franz Beckenbauer", "Gerd Muller",
              "Lothar Matthaus", "Manuel Neuer"]
for x in range(4):
    if x > 1:
        break
    print("Player " + str(x+1) + ": " + my_players[x])
print("End of roster")
```

Player 1: Franz Beckenbauer  
Player 2: Gerd Muller  
End of roster

Source: Aaron Reed, 2020.

Notice that in the code above, we've gone back to the for loop. However, as mentioned, the break statement will also work inside a while loop. We've added an if statement at the start of the for loop that checks to see if `x` is greater than 1. When it is, the code block for the if statement is executed, and the only code in that block is the break keyword. Break will interrupt execution of the for loop and resume execution of the program after the for loop's code block. Hence, when we run the program, we get the first two players in the list outputted to the screen and then the "End of roster" text.

But what if we didn't want to exit the loop altogether, what if we just want to skip execution of the loop once? We can do that with the "continue" keyword. Like break, continue can be used in for loops and while loops. The continue keyword will interrupt a loop, but

instead of exiting the loop, the interpreter just skips the current execution of the loop and returns to the top of the loop. For example, look at the `continue` keyword in the code below:

Figure 93: Python - Continue

```
my_players = ["Franz Beckenbauer", "Gerd Muller",
              "Lothar Matthaus", "Manuel Neuer"]
for x in range(4):
    if x == 2:
        continue
    print("Player " + str(x+1) + ": " + my_players[x])
print("End of roster")
```

```
Player 1: Franz Beckenbauer
Player 2: Gerd Muller
Player 4: Manuel Neuer
End of roster
```

Source: Aaron Reed, 2020.

In this code, the loop will execute when `x` is four different values: 0, 1, 2, and 3. The `if` statement evaluates `x` to see if it is equal to 2; when it is, the `continue` statement is executed. When `x` is equal to 2, `continue` will halt execution of that run through the loop and return execution to the `for` loop with `x` taking on the next value in the range, which is 3. Execution through the loop when `x == 3` happens just as it did when `x` was equal to 0 or 1. The end result is that the `print` statement is skipped when `x == 2` and you have the roster without player 3, Lothar Matthaus—sorry Lothar!

## 3.4 Iterators and Comprehensions

The `for` and `while` loops are essential tools for a programmer. Anytime you need to execute a logical step multiple times in a row, consider the `for` or `while` loop as a possible solution. Sometimes, as we have already seen, when programmers create loops, they do so to loop through all the values of a set or list. Although the `for` and `while` loops are possible solutions in such a case, there is another way to effectively loop through values in a set or list: iterators.

You can use iterators to loop through values in lists, tuples, and dictionaries. To loop through one of these objects using an iterator, encapsulate the name of the list, tuple, or dictionary in the parameter to a call to the `iter` function as shown below:

**Figure 94: Python - Creating an Iterator**

```
my_list = [4, 8, 15, 16, 23, 42]
my_iterator = iter(my_list)
```

Source: Aaron Reed, 2020.

As you can see in the code snippet above, the `iter` function returns an iterator object that you'll need to capture by assigning it a variable name. In this case, the variable "my\_iterator" is an iterator for the list "my\_list." Once you have an iterator object for a list, you can loop through the list by calling the `next` function and passing it the name of the iterator. The first call to the `next` function will return the first value in the list. Each subsequent call to `next` will return the next value in the list. See the code below:

**Figure 95: Python - Looping Through a List with an Iterator**

```
my_list = [4, 8, 15, 16, 23, 42]
my_iterator = iter(my_list)
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

```
4
8
15
16
23
42
```

Source: Aaron Reed, 2020.

As you can see above, the first call to `next` returns the first value in the list (4), and each of the next five calls to `next` return the next value in the list.

But what would happen if we call `next` too many times? Let us see. In the example below, we add one more call to `next` just to see what happens:

Figure 96: Python — Iterator Loop Error

```
my_list = [4, 8, 15, 16, 23, 42]
my_iterator = iter(my_list)
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

4  
8  
15  
16  
23  
42

```
-----  
StopIteration                                Traceback (most recent call last)  
<ipython-input-4-c2ac9dfb00b4> in <module>  
      7 print(next(my_iterator))  
      8 print(next(my_iterator))  
---->  9 print(next(my_iterator))  
  
StopIteration:
```

Source: Aaron Reed, 2020.

We do not like to see that red block. We've found a StopIteration error, which occurs when trying to iterate beyond the bounds of a list, tuple, or dictionary. An effective way to loop through iterations and avoid the StopIteration error is to use the iterator in conjunction with the for loop, as shown below:

Figure 97: Python - Iterator with For Loop

```
my_list = [4, 8, 15, 16, 23, 42]
my_iterator = iter(my_list)
for x in my_iterator:
    print(x)
```

4  
8  
15  
16  
23  
42

Source: Aaron Reed, 2020.

In the code above, you can see that we use a for loop with the same syntax we have already discussed, but in this case, instead of looping through a range of values using the range function, we are looping through the values with an iterator. Each time through the loop, next is called, retrieving the next value in the list and assigning that value to the variable x. This code can be greatly simplified in Python by doing the following:

**Figure 98: Python - For Loop with List**

```
my_list = [4, 8, 15, 16, 23, 42]
for x in my_list:
    print(x)
```

4  
8  
15  
16  
23  
42

Source: Aaron Reed, 2020.

Here, an iterator is created automatically when using the name of the list in a for loop. Just as with the previous example, every time through the loop, the next function is called, and x is assigned the value of the next item in the list.

## Comprehensions

Sometimes you may want to create a list of values based on an existing list of values. For example, let us say you had a list of numbers and you wanted to create a new list of the same numbers multiplied by themselves. Here is one way you could accomplish that:

**Figure 99: Python - Creating a List of Square Numbers**

```
my_numbers = [4, 8, 15, 16, 23, 42]
my_new_numbers = []

for x in my_numbers:
    my_new_numbers.append(x*x)

print(my_new_numbers)
```

[16, 64, 225, 256, 529, 1764]

Source: Aaron Reed, 2020.

With comprehensions, we can create a new list by using values from an existing list, greatly simplifying our code. Here's a way to create the same list of squared numbers using a list comprehension:

**Figure 100: Python - Creating a List of Square Numbers with a List Comprehension**

```
my_numbers = [4, 8, 15, 16, 23, 42]
my_new_numbers = [n*n for n in my_numbers]

print(my_new_numbers)
```

```
[16, 64, 225, 256, 529, 1764]
```

Source: Aaron Reed, 2020.

In the code above, you can see that the contents of the variable “my\_new\_numbers” contains an expression ( $n*n$ ) and a for loop. Effectively, the code works very similarly to the for-loop code we've seen already, it's just formatted differently. As we evaluate the line of code `n*n for n in my_numbers`, it may help to see it in a way we are already familiar with:

**Figure 101: Python - List Comprehension For Loop**

```
for x in my_numbers:
    my_new_numbers.append(n*n)
```

Source: Aaron Reed, 2020.

Notice the similar elements in the code above and in the code for the list comprehension. When we look at the list comprehension definition `n*n for n in my_numbers`, we see that the expression `n*n` is essentially the code that we would have put inside the code block associated with the for loop. Effectively, this code will loop through all the values in `my_numbers`, assigning the next value in the list to the variable `n`. It will then evaluate the expression (in this case, `n*n`), and the resulting value will be added to the `my_new_numbers` list.

Comprehensions can be a powerful way of creating new lists from existing lists. You can do a wide range of things with comprehensions, including, as we've seen, modifying the values in the existing list as you place them in the new list. You can also remove certain values from the list by adding an if statement at the end of the for loop. The code below creates a list of squares from an original list, but only if the original value is less than 20:

**Figure 102: Python - List Creation with Comprehension and Condition**

```
my_numbers = [4, 8, 15, 16, 23, 42]
my_new_numbers = [n*n for n in my_numbers if n < 20]

print(my_new_numbers)
```

```
[16, 64, 225, 256]
```

Source: Aaron Reed, 2020.

With assignments and expressions, conditional statements, loops, iterators, and comprehensions under your belt, you're becoming a powerful Python programmer!



#### **SUMMARY**

In Python, an expression evaluates to a value. Statements perform some function. A line of code that uses the assignment operator is a Python statement that will assign the value of an expression (on the right-hand side of the operator) to a variable on the left-hand side of the operator.

Conditional expressions use operators to compare values, such as == (equals), > (greater than), < (less than), and so forth. They can be used in conditional statements to execute different sections of code, depending on the value of different variables. The if, elif, and else statements are examples of these conditional statements.

At times it is necessary to execute the same line of code, or similar lines of code, multiple times sequentially. Loops such as the for loop and while loop are great tools to accomplish such tasks.

It also may be important to loop through lists of values at times. Iterators provide a concise way to do so in Python. Iterators loop through values in lists, tuples, and dictionaries. When building those lists of values, comprehensions are powerful tools that can facilitate the creation of new lists based on values in existing lists.





# UNIT 4

## FUNCTIONS

### STUDY GOALS

On completion of this unit, you will have learned ...

- what a function is in Python and why you would use one.
- Python scope rules for variables and functions.
- how to use function arguments, such as default arguments.

## 4. FUNCTIONS

### Case Study

Kyle and Morgan are ready to take the next step in the development of their application. They have a solid understanding of how to use different data types, how to save data to files, and how to assign values, compare values in conditional statements, and loop through lists of data.

As they sit down to organize their thoughts on the project, however, one thought keeps occurring to them. They imagine the various functions of their project that would be repeated at different times throughout the application. For example, they want the user to be able to print out a roster for their team. But they also want the user to be able to print out the roster for other teams. Likewise, they believe the user needs to be able to see player statistics such as height, weight, position, speed, goals scored, shots taken, and they believe the user needs to be able to view those statistics for any player.

Kyle and Morgan feel they have the basic knowledge required to write code that prints out a roster for a team. However, they wonder how this will work for every team. Let's say we look at the UEFA Champions League. There are 32 teams in that league; do they have to write that same code 32 times in order to print the roster for every team? Even worse, let's say there are 25 players on each team's roster. That's  $32 \cdot 25 = 800$  players. Do they have to write the code to show player statistics 800 times? And that's only one league! There are hundreds of leagues worldwide. For the first time since conceptualizing their idea, Kyle and Morgan feel their heads spinning.

- How can Kyle and Morgan organize their code in an effective and efficient way?
- Are there ways to reuse code blocks so Kyle and Morgan won't have to write the player-statistics code hundreds of times? If so, what are they?
- Other than requiring less time to write, are there other reasons why reusing code might be beneficial?

### 4.1 Function Declaration

While programming, you'll find there are many times when you need to perform the same task, or a very similar task, multiple times. This is not that different from how our brains work as we navigate everyday life. In many ways, the brain behaves much like a computer. The brain takes input from all kinds of areas including our vision, our hearing, and our sense of touch. With that input, the brain makes some calculations or performs some processing (identifies that a visual shape looks like a pencil, relates a honking sound to automobile's horn, or calculates that  $2 + 2 = 4$ ). The brain then has some form of output as well, sending impulses to different parts of the body to create a reaction to the inputs received.

So, let us look further in to the way the brain works and handles input. If you walk down a street, you don't typically have to think about how to walk. Instead, you give your brain some sort of input signal saying, "I'm going to start walking," you move one foot ahead of the other, and the brain takes over from there. Why? Because the brain has done this walking routine so many times that it is just that: routine. It is a task that is thankfully easy for most brains to perform. But what if you turned at an intersection, and all of a sudden, based on various inputs, the brain realized you were walking down a different street? Would the brain have to invoke some new form of walking routine? Typically not. The brain would likely use the same walking routine regardless of the street upon which you walked. What if you were going to move from pavement to carpet? Would the brain need a new routine for carpet walking? Probably not. Because walking on carpet and walking on concrete are very similar, the brain doesn't need a separate routine for walking on each surface.

You can likely find all kinds of activities that are similar—riding a bicycle, driving a car, eating a sandwich, drinking a glass of water, etc. Each activity has different applications that are very similar (riding a bicycle uphill vs. downhill, driving a car vs. driving a truck, eating a peanut butter sandwich vs. eating a ham sandwich, drinking a glass of water or a glass of milk), and the brain likely uses the same routine to handle the subtle differences.

When programming, we also strive to identify sets of actions that are the same or similar and group them into routines (called functions) that we can reuse over and over. Imagine, for example, that we were building a program to help people learn basic mathematics and in that program we had some code that, given two numbers, would output to the user the results of addition, subtraction, multiplication, and division of those two numbers. It might look something like this:

**Figure 103: Python - Simple Math**

```
int_1 = 1
int_2 = 2
print("Your two numbers are " + str(int_1) + " and " + str(int_2) + ".")
int_3 = int_1 + int_2
print(str(int_1) + " plus " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 - int_2
print(str(int_1) + " minus " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 * int_2
print(str(int_1) + " times " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 / int_2
print(str(int_1) + " divided by " + str(int_2) + " equals " + str(int_3) + ".")
```

```
Your two numbers are 1 and 2.
1 plus 2 equals 3.
1 minus 2 equals -1.
1 times 2 equals 2.
1 divided by 2 equals 0.5.
```

Source: Aaron Reed, 2020.

The code above should be somewhat straightforward by now. We have two variables containing integers (`int_1` holds the value 1 and `int_2` holds the value 2). What follows is a series of print statements and assignments of a third variable, `int_3`, with various expressions using addition, subtraction, multiplication, and division.

Now, suppose we wanted to do this twice, once with the original numbers (1 and 2) and once with a different set of numbers, say 5 and 10. We could simply copy and paste the code and it would work as follows:

**Figure 104: Python - Simple Math and More Simple Math**

```
int_1 = 1
int_2 = 2
print("Your two numbers are " + str(int_1) + " and " + str(int_2) + ".")
int_3 = int_1 + int_2
print(str(int_1) + " plus " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 - int_2
print(str(int_1) + " minus " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 * int_2
print(str(int_1) + " times " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 / int_2
print(str(int_1) + " divided by " + str(int_2) + " equals " + str(int_3) + ".")

int_1 = 5
int_2 = 10
print("Your two numbers are " + str(int_1) + " and " + str(int_2) + ".")
int_3 = int_1 + int_2
print(str(int_1) + " plus " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 - int_2
print(str(int_1) + " minus " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 * int_2
print(str(int_1) + " times " + str(int_2) + " equals " + str(int_3) + ".")
int_3 = int_1 / int_2
print(str(int_1) + " divided by " + str(int_2) + " equals " + str(int_3) + ".")
```

```
Your two numbers are 1 and 2.
1 plus 2 equals 3.
1 minus 2 equals -1.
1 times 2 equals 2.
1 divided by 2 equals 0.5.
Your two numbers are 5 and 10.
5 plus 10 equals 15.
5 minus 10 equals -5.
5 times 10 equals 50.
5 divided by 10 equals 0.5.
```

Source: Aaron Reed, 2020.

The code above works perfectly fine with two sets of numbers. The addition, subtraction, multiplication, and division all appear to work correctly, and the output is accurate.

What if we wanted to do the same calculations and output for not just two sets of numbers but 10? We could again use the same formula and copy/paste the code above to have 10 sets of calculations and print statements. Alternatively, what if I told you that there was a

way to reuse the print/output routine in a Python function much like the way the brain reuses processes to walk or eat? More on functions in a minute, but in the meantime, can you think of any drawbacks to the copy/paste method?

One of the major drawbacks with the copy/paste method comes into play when you need to modify your code. Let's say we found a problem with our code and we needed to change it. Or what if we simply wanted to improve or enhance it? If we had 10 copies of the same code, we would have to make that change in 10 different places. With each change comes an increased opportunity for us to make a mistake and mistype or forget something, introducing more chances for errors in our code. In general, it's a much better practice to reuse similar routines by encapsulating them inside functions.

## Functions

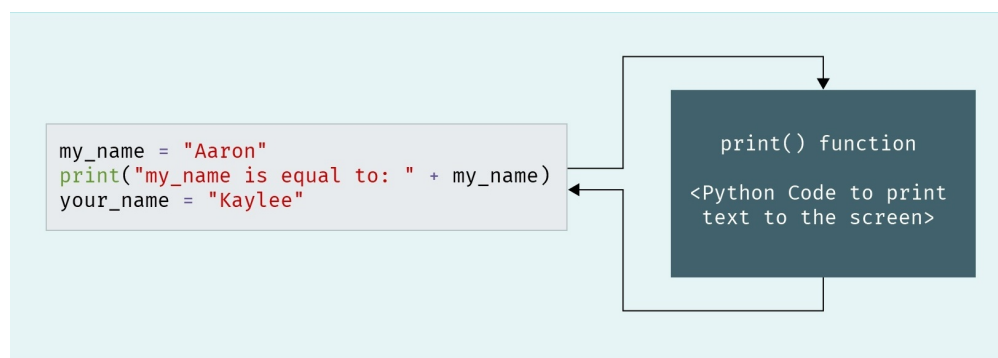
So, what is a function in Python? Well, you've already been using several functions such as print, str, iter, and next. Essentially, a function is a block of code that you've given a name to (such as print, str, iter, or next). When you use that function name in a program, execution of the code jumps to that function block, executes the function, and then returns to the point where the function name was **called**. For instance, the print function isn't some magical line of code that prints text to the screen. Instead, it's a function with an associated code block that takes a string of text and outputs the text to the screen. When you call print, execution of your code is paused and execution of the code within the print function is started. When the print function code block is finished, the execution of the code in your program resumes at the point just after the print statement.

### Calling

We refer to the act of using a function as calling that function. A line of code that uses the print() function is referred to as a function call.

Let's look at the overly simplified code block in the next figure to understand this better:

**Figure 105: Print Function**



Source: Aaron Reed, 2020.

In the example above, a variable called "my\_name" is created and assigned the text "Aaron." Then the print function is called to output that variable and a brief message to the screen. At that point, execution of this code block is paused and execution follows the arrows over to the print function itself. We don't see the code for that function because it was created by the people who wrote Python, but there is code for the print function somewhere, and execution will begin at the top of that function. That function, as we all know, will output text to the screen. After it does so and the function is finished, execution

returns to our code at the point right after the print function was called. The next line of code, which creates a variable called “your\_name” and assigns it the value of “Kaylee,” is then executed.

You typically use functions in Python to accomplish a task. That task could be anything, such as assigning some values to variables, reading or writing data to files, computing some calculations and returning the results, outputting data to the screen, or anything else you can imagine doing in code.

Just like the brain when it reuses the same routine to drive a car or eat a sandwich, you use functions to encapsulate code that will be used repeatedly so you don’t have to type the code multiple times. This reduces the chance of errors in your code, and also makes it more readable.

Let’s build our first Python function! To define your own custom function, you use the keyword `def` followed by the name of the function, open and closed parentheses, and the colon symbol. Just as we saw when we built loops, the colon indicates that an indented section of code will follow. In this case, that indented section of code is the body of your function—the place where you’ll put the code to accomplish the task for which you’ve created the function.

The following code defines one of the world’s simplest functions:

**Figure 106: One of the World’s Simplest Functions**

```
def my_first_function():  
    print("I can't believe I'm in a function!")
```

Source: Aaron Reed, 2020.

In the trivial code above, a function called “my\_first\_function” is defined. The function body is one line of code that outputs something to the screen by calling another function, the print function. We can use this function simply by calling its name and using the open/close parentheses. The following code uses the my\_first\_function function:

**Figure 107: Using One of the World’s Simplest Functions**

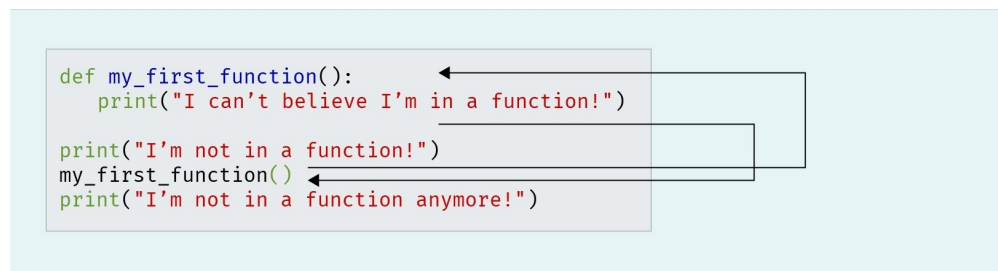
```
def my_first_function():  
    print("I can't believe I'm in a function!")  
  
print("I'm not in a function!")  
my_first_function()  
print("I'm not in a function anymore!")
```

Source: Aaron Reed, 2020.

In this example, we define the `my_first_function` function. Next, we output something to the screen via the `print` function. Then, we call `my_first_function`. Finally, the last line of code uses the `print` function again to output something to the screen.

The execution flow for this code will look like this:

**Figure 108: Execution Flow as We Use One of the World's Simplest Functions**

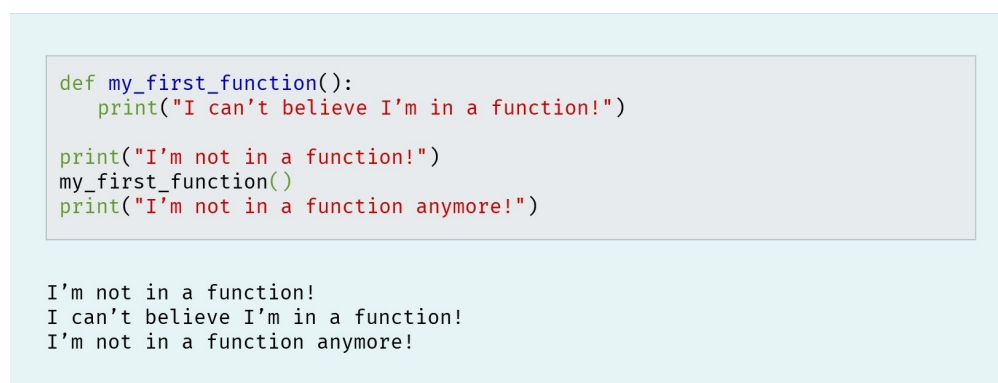


Source: Aaron Reed, 2020.

The first thing that will happen when we run this code is that the first two lines of code will define our `my_first_function` function. Although the `print` statement is part of that function, nothing happens on screen at this point. That's because the interpreter will not execute the code in the `my_first_function` function at this point; it will simply define the function so we can use it later. Then, the third line of code executes, which outputs "I'm not in a function!" to the screen. Then, when we call the `my_first_function` function, execution diverts to start at the beginning of the `my_first_function` function. At this time, the `print` statement will execute, outputting "I can't believe I'm in a function!" to the screen. After that prints, the function is finished, so execution goes back to the point where the function call was made. Then the final line of code prints "I'm not in a function anymore!" to the screen using the `print` function.

When we run this code, the output is seamless:

**Figure 109: Output as We Use One of the World's Simplest Functions**



Source: Aaron Reed, 2020.

Functions are very powerful tools that will be indispensable as you start developing more and more complex programs. They are great ways to organize code and make it more readable. Their greatest benefit, however, is code reusability. Whatever code you place in the function can be used over and over by calling the function as we did in the example above. This eliminates the need to retype code multiple times and, more importantly, reduces potential bugs by placing repetitive code in one location.

## 4.2 Scope

Before we dig further into functions, let's talk a little bit about scope. Scope refers to the definition of variables or functions in Python. If, in a given line of code, the interpreter recognizes a particular variable or function, that variable or function is said to be in scope. If the interpreter does not recognize it, it is said to be out of scope.

For example, look at the following code:

**Figure 110: Very Basic If Statement**

```
if this_new_variable == 5:  
    print("In scope!")
```

Source: Aaron Reed, 2020.

In the code listed above, we simply check to see if a variable named `this_new_variable` is equal to 5. If it is, a message is printed saying that it is in scope. Now, let's run that code and see what happens. You should see something like the image below:

**Figure 111: Very Basic If Statement — Error**

```
if this_new_variable == 5:  
    print("In scope!")  
  
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-3-10ac77fdb8ce> in <module>  
----> 1 if this_new_variable == 5:  
      2 print("In scope!")  
  
NameError: name 'this_new_variable' is not defined
```

Source: Aaron Reed, 2020.



Why do we get an error here? Before we investigate that, note that it is good to get into the habit of learning from these errors. The error message typically has a good amount of information in it to help us figure out where we went wrong. First note that the line of code at which the error occurred is highlighted with an arrow to the left of the code. Above, we can see that the arrow is clearly pointing to line 1 of our code. Next, look at the last line of the error message. It provides a description for the error. In this case, the description tells us the name “this\_new\_variable” is not defined.

But why? You might be thinking back to previous chapters where we discussed the benefits of Python, one of which was that you can simply assign a value to anything you want and it will automatically create a variable for you. Well, that is true, but there is a catch—you have to assign a value to that name in order to create the variable. We never assigned any value to this\_new\_variable. The first time we use that name is in the comparison expression within the if statement. We’re essentially asking the interpreter to tell us if this\_new\_variable is equal to 5 when we’ve never mentioned this\_new\_variable to it before. It therefore has no idea what we’re talking about and gives us an error.

You have probably noticed by now that the Python interpreter reads code from top to bottom. So, if we, for example, define this\_new\_variable later in the code, will it matter? Let us see:

**Figure 112: Very Basic If Statement 2 - Error**

```
if this_new_variable == 5:
    print("In scope!")

this_new_variable = 0
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-dd08ce140f58> in <module>
----> 1 if this_new_variable == 5:
      2 print("In scope!")
      3
      4 this_new_variable = 0

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

Source: Aaron Reed, 2020.

In the code above, we added a line after the if statement where we do indeed define this\_new\_variable. However, running the code yields the same result—an error indicating that this\_new\_variable is not defined. This is because, at the time of the if statement, the variable is out of scope; it has not yet been defined. It is only in scope after the third line of code where this\_new\_variable is defined. Using the variable name at any time when the variable is out of scope will result in an error.

If we were to add a few lines of code just for context and then shade the areas in which this\_new\_variable is in and out of scope, it would look something like this:

Figure 113: Scope for this\_new\_variable

```
print("Welcome to my program!")
print("I'm excited to see if the variable is in scope!") this_new_variable
if this_new_variable == 5:                                is out of scope
    print("In scope!")

this_new_variable = 0
print("We're almost finished!")                          this_new_variable
print("And...now we're done.")                          is in scope
```

Source: Aaron Reed, 2020.

Essentially, you cannot use `this_new_variable` in the red out-of-scope area, but you can use it all you want in the green in-scope area.

The same scope principle applies to functions. Let's take an example from the previous section and modify it so the function call occurs before the function definition. Let us see what happens:

Figure 114: Scope for Functions

```
print("I'm not in a function!")
my_first_function()
print("I'm not in a function anymore!")

def my_first_function():
    print("I can't believe I'm in a function!")
```

I'm not in a function!

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-2-1adb004e58da> in <module>
      1 print("I'm not in a function!")
---->  2 my_first_function()
      3 print("I'm not in a function anymore!")
      4
      5 def my_first_function():

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

Source: Aaron Reed, 2020.

You can see here that the same principle applies to functions regarding scope. In fact, the error message is the same—name <insert name> is not defined. It is telling us that `my_first_function` is not defined and it is correct; we tried to call that function, but we don't define it until later on.

There's another important component of scope that is essential for us to understand before we move on. Variables defined within functions only have a scope within that function. To understand what I'm talking about, let's look first at this simple program:

**Figure 115: A Very Simple Program**

```
def my_function():
    print("Nice function!")
    my_variable = "I love functions!"
    print("my_variable = " + my_variable)

print("Starting program...")
my_function()
print("Ending program...")
```

```
Starting program...
Nice function!
my_variable = I love functions!
Ending program...
```

Source: Aaron Reed, 2020.

In the code above, we first create a function called `my_function`. The body of that function contains three lines of code: a print statement, a variable definition for `my_variable`, and another print statement. Next, we have three other lines of code outside of that function: another print statement, a call to the `my_function` function, and a final print statement. Looking at the output, you should be able to trace through the program and see why the output is as shown.

So, regarding scope of variables, because `my_variable` was created in `my_function`, it only exists within `my_function`. Look at what happens if we try to use it outside of `my_function`:

Figure 116: Function Variable Scope

```
def my_function():
    print("Nice function!")
    my_variable = "I love functions!"
    print("my_variable = " + my_variable)

print("Starting program...")
my_function()
if my_variable == "Quit":
    print("Ending program...")
```

```
Starting program...
Nice function!
my_variable = I love functions!
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-11-f70e85407a6d> in <module>
      6 print("Starting program...")
      7 my_function()
---->  8 if my_variable == "Quit":
      9     print("Ending program...")

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

Source: Aaron Reed, 2020.

As you can see, we have the same error once again. It's telling us that `my_variable` is not defined. Look at the output from the program. It runs just as you would expect until the point where we try to access `my_variable` outside of the function, and then the error occurs.

It's important to remember the rules of scope as you define functions and determine where to create variables. Let us look at one more example before we move on. Review the code below and try to determine what the output will look like:

Figure 117: Fun with Variable Scope

```
def my_function():
    my_variable = "Inside of the function"
    print("my_variable = " + my_variable)

my_variable = "Outside of the function"
print("my_variable = " + my_variable)
my_function()
print("my_variable = " + my_variable)
```

Source: Aaron Reed, 2020.

What do you think will happen here? The tricky thing is that now we are defining `my_variable` twice: once inside the function and once outside the function. I'd encourage you to take a piece of paper and jot down what you think the output will be.

Now, let's see what the output really is:

**Figure 118: Fun with Variable Scope - Output**

```
def my_function():
    my_variable = "Inside of the function"
    print("my_variable = " + my_variable)

my_variable = "Outside of the function"
print("my_variable = " + my_variable)
my_function()
print("my_variable = " + my_variable)
```

my\_variable = Outside of the function  
my\_variable = Inside of the function  
my\_variable = Outside of the function

Source: Aaron Reed, 2020.

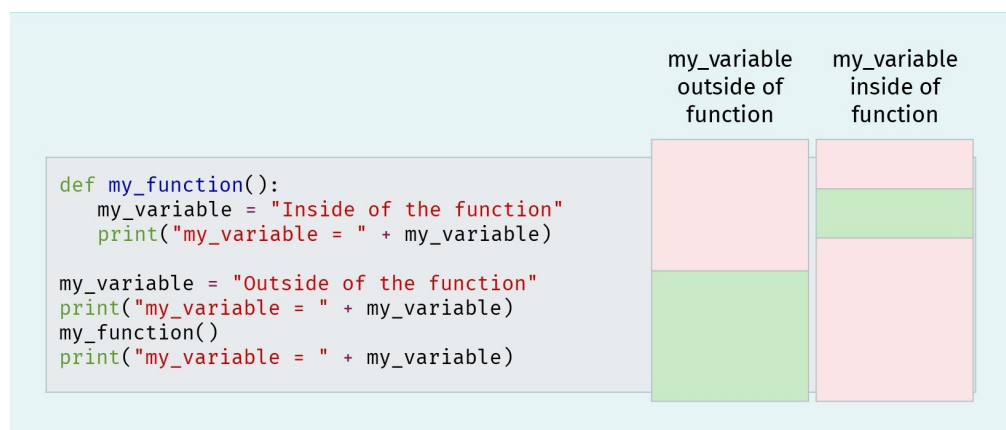
Did you get it right? If so, congratulations! If not, don't worry about it. Scope rules can be somewhat confusing at first. You'll pick it up as you go along. The trick here is to remember that when you define a variable within a function, it only has scope within that function. So, in the code above, we define `my_variable` outside of the function. We then define it again within the function. Remember, because the variable within the function only exists within the function, to the rest of the program it is as if that variable does not exist. That's why when we change the value of that variable, it does not change the value of the variable outside of the function. The interpreter actually creates two variables with the same name, one within the function and one outside of the function. Changing the value of one will not affect the other.

If we were to outline the scope of both **instances** of `my_variable`, it would look something like this, where red is out of scope and green is in scope:

**Instance**

The term instance refers to a unique representative of a data type. When you define a variable, you create an instance of that variable's type.

Figure 119: Scope of Two Instances of my\_variable



Source: Aaron Reed, 2020.

As you can see, the instance of `my_variable` that was created outside of the function is not in scope anywhere other than in the code outside of the function after the variable has been defined. Alternatively, the instance of `my_variable` that was defined in the function is only available within that function after the variable was defined.

Go ahead and take some time to experiment with this in Python. Scope rules can be confusing but they are also essential to understand as you start programming more complicated applications.

In Python, you can also create nested functions. A nested function is a function that is defined within another function. Just as a variable defined within a function has local scope to that function alone, a nested function has a scope of its parent function alone. That means it can only be called within the function in which it is created. See an example of the flow of a program with a nested function below:

**Figure 120: Scope of Nested Function**

```
def my_function():  
    print("In my function.")  
  
    def my_nested_function():  
        print("In my nested function.")  
        print("Exiting my nested function.")  
  
    print("About to call the nester function.")  
    my_nested_function()  
    print("Finishing call to nested function.")  
    print("Exiting my function.")  
  
print("About to call my function.")  
my_function()  
print("Finished call to my function.")
```

```
About to call my function.  
In my function.  
About to call the nested function.  
In my nested function.  
Exiting my nested function.  
Finished call to nesting function.  
Exiting my function.  
Finished call to my function.
```

Source: Aaron Reed, 2020.

Thinking of variable scope can start to get confusing when looking at code that uses nested functions. That's because as more and more levels of scope are added, it can be more difficult to identify the level that a variable or function belongs to. Look at the code below to see how variable scope works with nested functions:

**Figure 121: Variable Scope within Nested Functions**

```
x = 1

def my_function():
    x = 10
    print("In my_function, x=" + str(x))

    def my_nested_function():
        x = 100
        print("In my_nested_function, x=" + str(x))

    my_nested_function()

my_function()
print("Outside of functions, x=" + str(x))
```

```
In my_function, x=10
In my_nested_function, x=100
Outside of functions, x=1
```

Source: Aaron Reed, 2020.

Notice that despite using the variable `x` in each function, the global variable `x` is not affected as we modify `x`. This is because, in each instance, the functions are creating their own local version of `x` when the variable `x` is used with the assignment operator. There is a way to prevent this and to use the global `x` instead. You use the `global` keyword to preface the variable you're going to use. This will tell the interpreter that you want to use the global version of that variable instead of creating a new local version. See below:



**Figure 122: Accessing Global Variables within Functions**

```
x = 1

def my_function():
    global x
    x = 10
    print("In my_function, x=" + str(x))

    def my_nested_function():
        global x
        x = 100
        print("In my_nested_function, x=" + str(x))

    my_nested_function()

my_function()
print("Outside of functions, x=" + str(x))
```

```
In my_function, x=10
In my_nested_function, x=100
Outside of functions, x=100
```

Source: Aaron Reed, 2020.

There's also a way to access a function's local variable from a nested function within that function. To do this, you use the `nonlocal` keyword. Just like the `global` keyword, this tells the interpreter that you don't want to create a local variable within the nested function and you also don't want to access a global version of that variable, but instead that you want to use the version created in the parent function. See below:

Figure 123: Nonlocal Variable within Nested Functions

```
x = 1

def my_function():
    x = 10
    print("In my_function, x=" + str(x))

    def my_nested_function():
        nonlocal x
        x = 100
        print("In my_nested_function, x=" + str(x))

    my_nested_function()
    print("At end of my_function, x=" + str(x))

my_function()
print("Outside of functions, x=" + str(x))
```

```
In my_function, x=10
In my_nested_function, x=100
At end of my_function, x=100
Outside of functions, x=1
```

Source: Aaron Reed, 2020.

## 4.3 Arguments

Until now, our function definitions have been fairly simplistic. We have created some functions that print text or that create some variables, add them, and print the result, but we have not gone much beyond that. The real power of functions comes in passing values to and from functions. Functions can receive data from the calling program through parameters. The values that are passed into the function via the parameters are called **arguments**. A function's parameters are defined between the open and closed parentheses in the function definition. A function can have zero parameters, denoted by empty parentheses as we've seen thus far, or it can have as many arguments as you'd like. Separate each parameter in the function definition with a comma.

When calling a function that has one or more parameters, put arguments, i.e. values for the parameters in the parentheses when you call that function. In this way, the value becomes an argument to the function. You've already done this many times with several functions including the print function. When you call print, you pass it a string by putting that string inside the parentheses when you call print. That's how print knows what it is you would like to print.

Let's take a look at an example:

### Arguments

Parameters, or arguments, are essential components of functions. Functions can receive data from the calling code if data is sent through arguments.

**Figure 124: A Function with Three Arguments**

```
def add_three_numbers(a, b, c):
    the_sum = a + b + c
    print("The sum of " + str(a) + ", " + str(b) + ", and " +
          str(c) + " is: " + str(the_sum))

add_three_numbers(1, 2, 3)
add_three_numbers(10, 20, 30)
add_three_numbers(-123, 9271, -712)
```

```
The sum of 1, 2, and 3 is: 6
The sum of 10, 20, and 30 is: 60
The sum of -123, 9271, and -712 is: 8436
```

Source: Aaron Reed, 2020.

In the code above, we've defined a function called "add\_three\_numbers" just as we have defined functions previously, with one difference. We now have three variables as arguments for that function: a, b, and c. The code in the body of the function adds the three numbers and then uses a print statement to output the result to the screen. The other three lines of code simply call the add\_three\_numbers function with different values. The result is shown below the code.

One key thing to remember is that the number of arguments passed into a function must be equal to the number of arguments specified in the function definition. For example, below we have the same code but with one difference: the first call to add\_three\_numbers has only two arguments instead of three. Notice what happens when we run that code:

Figure 125: Missing Argument Error

```
def add_three_numbers(a, b, c):
    the_sum = a + b + c
    print("The sum of " + str(a) + ", " + str(b) + ", and " +
          str(c) + " is: " + str(the_sum))

add_three_numbers(1, 2)
add_three_numbers(10, 20, 30)
add_three_numbers(-123, 9271, -712)
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-22-de45088852a3> in <module>
      4             str(c) + "is: " + str(the_sum))
      5
----> 6 add_three_numbers(1, 2)
      7 add_three_numbers(10, 20, 30)
      8 add_three_numbers(-123, 9271, -712)
TypeError: add_three_numbers() missing 1 required positional argument: 'c'
```

Source: Aaron Reed, 2020.

Again, notice how informative these error messages are. The line causing the error is marked with the arrow (line 6). The error message tells us that the function call to `add_three_numbers` is missing one argument. Pay close attention to error messages when you see them as they can really help you save time when trying to figure out what's wrong with your code.

Just as calling a function with too few arguments causes an error, calling it with too many will also cause an error:

**Figure 126: Too Many Arguments Error**

```
def add_three_numbers(a, b, c):
    the_sum = a + b + c
    print("The sum of " + str(a) + ", " + str(b) + ", and " +
          str(c) + " is: " + str(the_sum))

add_three_numbers(1, 2, 4, 5)
add_three_numbers(10, 20, 30)
add_three_numbers(-123, 9271, -712)
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-22-de45088852a3> in <module>
      4             str(c) + "is: " + str(the_sum))
      5
----> 6 add_three_numbers(1, 2, 4, 5)
      7 add_three_numbers(10, 20, 30)
      8 add_three_numbers(-123, 9271, -712)
TypeError: add_three_numbers() takes 3 positional arguments but 4 were given
```

Source: Aaron Reed, 2020.

Parameters can be of any type, just like variables. Below, you can see an example of a function that requires four arguments, the first of which are strings and the fourth of which is an integer:

**Figure 127: Different Data Types in Arguments**

```
def person_info(name, phone, address, age):
    print(name)
    print("Phone: " + phone)
    print("Address: " + address)
    print("Age: " + str(age))

person_info("Alyssa", "423-192-2311", "152 Wander Lane", 23)
```

```
Alyssa
Phone: 423-192-2311
Address: 152 Wander Lane
Age: 23
```

Source: Aaron Reed, 2020.

Thus far, we have passed arguments to functions in the same order that they are defined. For instance, the function defined in the code above has four arguments that are (in order) name, phone, address, and age. When you pass arguments to the function in the function call, the data we pass to the function lines up with the function arguments in order. In

other words, “Alyssa” is the first thing we pass to the function call, so it is assigned to the first argument, which is “name.” The second thing we pass is assigned to the second argument, and so on.

There is another way we can pass arguments, however. We can call them by name specifically in the function call. When we do so, we can specify them in any order we want. See below:

**Figure 128: Passing Arguments in Various Orders**

```
def person_info(name, phone, address, age):
    print(name)
    print("Phone: " + phone)
    print("Address: " + address)
    print("Age: " + str(age))

person_info("Alyssa", "423-192-2311", "152 Wander Lane", 23)
person_info(name="Hayden", phone="312-910-2238",
            address="51 Cosmo Drive", age=45)
person_info(age=32, name="Rylan",
            address="8162 Grizzly Way", phone="763-821-0028")
```

```
Alyssa
Phone: 423-192-2311
Address: 152 Wander Lane
Age: 23
Hayden
Phone: 312-910-2238
Address: 51 Cosmo Drive
Age: 45
Rylan
Phone: 763-821-0028
Address: 8162 Grizzly Way
Age: 32
```

Source: Aaron Reed, 2020.

Notice the three function calls in the code above. The first call uses the same format as the previous example. This type of argument setup is called “positional arguments.” We are passing a set of arguments, and they line up with the function’s arguments based on the position in which we arranged them.

Next, look at the final two function calls. Instead of just passing in the arguments, we name the argument from the function definition and assign that name a value. This is an example of using named arguments because we specify the name of the argument in the function call. Pay close attention to the final function call: the argument names are out of order—(age, name, address, and phone) instead of (name, phone, address, age). Yet look at the output; it is all lined up correctly with the right arguments within the function. When you pass data through named arguments, you don’t have to follow a specified order of arguments.

Remember when I said you have to make sure your function calls have the same number of arguments as your function definitions? Well, that is only mostly true. There is something called a default argument in Python. You can specify a default value for an argument in a function definition, and if that argument is not passed in during the function call, the argument will assume the default value. If the argument is passed in during the function call, the passed in value overrides the default value. You specify a default value for an argument by setting it equal to a value in the function definition. See below:

**Figure 129: Default Arguments**

```
def do_math(a, b=1, c=2, d=3):
    result = a + b + c + d
    print("Sum is: " + str(result))
    result = a * b * c * d
    print("Product is: " + str(result))

do_math(1)
do_math(1, 5)
do_math(1, 5, 10)
do_math(1, 5, 10, 20)
```

```
Sum is: 7
Product is: 6
Sum is: 11
Product is: 30
Sum is: 19
Product is: 150
Sum is: 36
Product is: 1000
```

Source: Aaron Reed, 2020.

Notice in the code above that we have defined a function called “do\_math” that takes four arguments. However, notice that the final three arguments are default arguments, meaning we’ve assigned them default values. The only required parameter when you call do\_math is a, because it has no default value. Notice the four function calls to do\_math in the code above. The first one uses only the first argument; the second passes in two arguments; the third, three; and the fourth, four. Look at the output and compare the math. See if you can trace through each function call and understand what values a, b, c, and d are assigned within the function.

One thing to note about default arguments is that all default arguments have to be the rightmost arguments in the function definition. Meaning, in the code above, if a was not a default argument, b and c were, and d was not, you would get an error. All default arguments have to be on the right-hand side of the function’s argument list. Think about all we’ve discussed so far and see if you can figure out why that is.

Remember that there are two ways to pass in arguments in a function call. You can pass in arguments positionally, where the arguments are assigned based on position, and you can pass in arguments by name. If Python did not force you to have all default arguments on the right-hand side of the list, it would get really confusing when passing by position. For

example, imagine you had a function where the first two arguments were default arguments and the last two were not. Then let us say you called that function and passed in three values. To which of the four arguments would you assign those three values? The final two would likely have to be assigned to the two non-default parameters. But what about the first? Does it get assigned to the first argument or do you put it in the second so it is closer to the other two arguments? To avoid this confusion, all default arguments are specified on the right-hand side of the argument list.

## Return Values

Not only can you pass values into functions, you can also get values back. Functions do this via something called a “return statement.” A return statement simply uses the keyword “return” followed by a value or set of values that you want to return to the program at the point you called the function. One thing to note: the return statement will exit your function, so make sure you do not have code that you want to run placed inside your function after your return statement—that code will never be executed!

Let’s look at a couple of examples of return statements:

**Figure 130: Return Values**

```
def return_one_thing(a, b):
    result = a + b
    return result

def return_two_things(a, b):
    result1 = a + b
    result2 = a - b
    return result1, result2

print(return_one_thing(5, 6))
print(return_two_things(5, 6))
```

11  
(11, -1)

Source: Aaron Reed, 2020.

In the code above, the function “return\_one\_thing” will compute the sum of *a* and *b* and return the value. Notice that with the function call to `return_one_thing`, we have simply placed that call within a call to `print`. Because `return_one_thing` returns a value, the value returning from `return_one_thing` will be printed here.

Next, look at the function “return\_two\_things.” This function will add *a* and *b* and store it in a variable but then also store the value of *a* – *b* in a different variable. It will then return both values. Notice that the resulting output when printing the value returned from `return_two_things` is enclosed by parentheses. This is because the function is returning a tuple data type as it returns multiple values.



Functions are extremely powerful tools and are essential in most complex programs. Spend some time going over the examples in this section and making your own examples to get a better feel for how they work and what you can do with functions, arguments, and return values.



#### **SUMMARY**

Functions are a critically important concept in programming. They help to organize code by separating it from the main block of the program. They facilitate code reusability because blocks of code within functions can be called repeatedly. This also reduces the chances of errors in the code because you only need to build code for repeatable routines once.

Scope is also a critically important concept to understand. Scope, as it pertains to variables and functions, identifies where those variables and functions are defined and where they are not defined. When variables or functions are not defined, they cannot be used.

Arguments and return statements provide more utility to functions. You can pass data to a function through its list of arguments. Arguments can be passed positionally or by name. Arguments can also have default values. Return statements return values from functions and enable the function to send data back to the block of code that called the function.



# UNIT 5

## ERRORS AND EXCEPTIONS

### STUDY GOALS

On completion of this unit, you will have learned ...

- how to interpret error messages and trace the error to the root cause.
- about exception handling, what it is, and how to implement it.
- how to create and use logs to improve understanding of program flow.

## 5. ERRORS AND EXCEPTIONS

### Case Study

Kyle, Morgan, and their team are busy working on their application. They feel energized because they are armed with the basics of Python. They know how to use the data types they need in order to make their soccer application a reality. They also understand program flow and how to use if/elif/else statements and various looping techniques to achieve the results they want as they build their app. They know how to use functions to improve the readability of their code, maximize reusability, and minimize errors. They feel like they know everything they need to know in order to build a robust and powerful customer-ready application.

However, although some of the techniques they've learned can minimize errors, they quickly find that they cannot eliminate them. Errors pop up in code, and Kyle and Morgan find themselves struggling to deal with them.

- How can Kyle and Morgan identify the causes of errors and fix them more effectively?
- Are there ways to stop errors from affecting the application and/or the end-user—particularly when the program anticipates certain errors at specific points in the application?
- What other techniques can help to identify errors and understand the general flow of the program?

### 5.1 Errors

**Errors**  
Syntax errors, exceptions, and logic errors occur frequently when programming. Computers will do exactly what you tell them, and humans make mistakes. The best remedy for errors is to practice fixing them when they occur.

Programming **errors** are unavoidable. Even creating the simplest of programs yields a chance for some form of error. You have likely already experienced numerous errors as you've learned Python throughout this book. We have even discussed a few of them intentionally to help you understand how errors happen and how to deal with them. This chapter will go into more detail on errors, what we do when we find them, and steps we can take to reduce them.

For our purposes, we can think of errors in two categories: syntax errors and exceptions. Syntax errors will be discussed in this section, and exceptions will be addressed in the following section of this chapter.

A syntax error occurs when code does not conform to syntax rules. For example, remember the syntax rules for if statements? If statements use the keyword “if,” followed by a conditional expression, followed by a colon, followed by a block of indented code. If those rules are broken, a syntax error will occur.

Below, you will see an example of a syntax error. You've seen many of these already as we have learned the basics of Python together. In this case, we've omitted the colon after the if statement, resulting in a syntax error.

**Figure 131: Syntax Error**

```
value = int(input())
if value == 10
    print("You entered 10!")

File "<ipython-input-1-66f5a16cf215>", line 2
    if value == 10
                ^
SyntaxError: invalid syntax
```

Source: Aaron Reed, 2020.

Syntax errors are found by the Python interpreter. As the interpreter reads code, it checks for fidelity to the Python syntax rules. If a syntax error is found, the interpreter will display a syntax error message like the one shown above.

Take some time to look at the error message above. Try to pick out all the important information. The phrase “syntax error” itself helps you to understand that something in the code goes against Python programming rules. Typically, our goal when we get an error like this would be to find and fix the error. In that light, “important information” would include anything that leads us to the cause of the error.

Here are some things to focus on. The first line of the error message is critically important because it tells us the exact line of our code in which the error occurred. In this case, the error occurred in line 2. The second line of the error message shows the line of code in question and includes an arrow that points to where the interpreter guesses that the error occurs. Why would the interpreter “guess” where the error is found? The truth is, the interpreter really doesn’t know what you’re trying to do—only you know that. So, all the interpreter can do is point to where it sees a problem, and then you as the programmer must figure out if that is really the problem or if it is something else. The message here is that you should not just automatically look at the place in code that interpreter points to and assume that is where the error is. Instead, look at the general area of code starting with that point, branching out to the entire statement, and considering even surrounding statements as you try to ascertain the real problem.

An example where the interpreter may not point at the exact problem in the code can be found below:

Figure 132: Syntax Error — Interpreter Pointing In Wrong Spot

```
value = int(input())
value == 10:
    print("You entered 10!")

File "<ipython-input-9-8c58821f1f90>", line 2
    value == 10:
            ^
SyntaxError: invalid syntax
```

Source: Aaron Reed, 2020.

The code above is very similar to the previous example. However, instead of missing the colon at the end of the if statement, we're missing the if keyword itself. You can see, however, that although the interpreter identified the correct line as causing an error, the arrow is pointing to the wrong location. Why would the interpreter point in the wrong location? Again, it comes down to the fact that the interpreter does not know what you are trying to do—only you know that. Without the keyword “if,” the interpreter has no idea that this is supposed to be an if statement. As such, when it reaches the end of the line, it's not sure what it just tried to interpret, and it throws the error. However, even though the error message doesn't point us to the exact location of the error, it does get us close enough that we can figure it out by looking at the line of code pointed to by the interpreter. When you get an error message, make sure you read the error message carefully and then look for possible causes for the error at the location of the interpreter's arrow, the line of code in question, and surrounding code.

## 5.2 Exception Handling

Sometimes, code that conforms to Python programming rules and does not cause a syntax error will still break during execution of the program. An error that occurs during program execution and disrupts the normal flow of a program is called an **exception**. If an exception was not planned for appropriately by a programmer, the exception will cause an application to crash and cease execution. Programs do this because, by nature, an exception occurs when program execution hits a fatal error. For example, a common exception occurs when a program tries to divide any number by zero. If you go back to your math days, you'll remember that the answer to any number being divided by zero is undefined. Hence, when a program attempts to do this, the program simply does not know what to do next or even how to store a result that is undefined and instead, an exception, which in Python is called `ZeroDivisionError`, occurs.

Another example of a common exception can be found when trying to open, read from, or write to a file that does not exist. If such an error occurs, the Python program will show a `FileNotFoundError` exception.

**Exception**  
Runtime errors in code, known as “exceptions,” disrupt program execution. There are numerous types of exceptions, each designed to represent common runtime errors, such as dividing by zero or trying to open a file that does not exist.

These exceptions may sound dangerous and scary. Although exceptions should be carefully anticipated, detected, and handled, programmers should not view them as the end of the world. In reality, exceptions happen all the time when programming. The trick is to be good at knowing when they may occur and putting the right code in place to deal with them so your application will not crash. More on that in a bit. First, let's look at a couple of exceptions and how to detect them:

**Figure 133: Dividing by a Number**

```
value = input()
result = str(5 / int(value)):
print("5 divided by " + value + " = " + result)
```

10  
5 divided by 10 = 0.5

Source: Aaron Reed, 2020.

In the code snippet above, you see we first get a value from the user via the input function and assign that to a variable called "value." Next, we convert that value variable to an integer and use it as the denominator in a division problem dividing 5 by our value. We convert that value to a string and store it in the result variable. Finally, we print out text saying that 5 divided by our number is equal to the result.

You can see the code in action below the code snippet. When the user enters 10, the result shows that 5 divided by 10 is 0.5.

That code seems to work perfectly, right? Well, what if the user enters zero? The second line of the code will attempt to divide 5 by 0, which is undefined. Let's look at the result of such an action:

**Figure 134: ZeroDivisionError**

```
value = input()
result = str(5 / int(value)):
print("5 divided by " + value + " = " + result)

0

-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-4-d3ba6166a0d9> in <module>
      1 value = input()
---->  2 result = str(5 / int(value))
      3 print("5 divided by " + value + " = " + result)
ZeroDivisionError: division by zero
```

Source: Aaron Reed, 2020.

You can see in the image above that when the user enters zero for the input, the application crashes. The error provided by the interpreter is the `ZeroDivisionError` exception. That exception name gives us some idea as to why this crashed—it implies that the program attempted to divide something by zero. If you ever come across an error that you don't understand, you can simply search for that error on the internet to find some great help resources on that error.

Let's look at a few other things in this error message. At the bottom of the error, you see a cleaner description of the error. In this case, that description says "division by zero," which may further clarify things if you were confused. Next, just as it does with syntax errors, the interpreter provides an arrow pointing to the place in the code at which the error occurred. In this case, we can see that in line 2 we are dividing by a variable and, when that variable equals zero, we are going to have this error.

Let's look at another exception example:

**Figure 135: Opening and Reading a File**

```
file_name = input()
my_file = open(file_name, "r")
print(my_file.read())
my_file.close()
```

Source: Aaron Reed, 2020.



In the code above, the user is prompted to enter a value, which we store in a variable called “file\_name.” Then, we open a file with the name stored in file\_name. The second argument in the open function, “r,” indicates that we’re opening the file for reading. We then read the file and print out the resulting data before closing the file.

If you run this code and enter a valid filename, the result should be that the contents of the file will be printed to the screen. However, if you enter an invalid file name, an exception will occur. See below:

**Figure 136: FileNotFoundError**

```
file_name = input()
my_file = open(file_name, "r")
print(my_file.read())
my_file.close()
```

DeathStarPlans.txt

```
-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-10-967fbe0fbbb6> in <module>
      1 file_name = input()
---->  2 my_file = open(file_name, "r")
      3 print(my_file.read())
      4 my_file.close()

FileNotFoundError: [Errno 2] No such file or directory: 'DeathStarPlans.txt'
```

Source: Aaron Reed, 2020.

In the image above, you can see that the user entered the file name “DeathStarPlans.txt” when prompted. Unfortunately, it was a trap! The file does not exist. The result is a FileNotFoundError exception. Again, let’s look at some hints here to help us figure out what is going on. The description at the bottom of the error is very helpful in this case: No such file or directory: 'DeathStarPlans.txt'. This tells us exactly what the problem is and, if we really expected the file to be there, we could go looking for it and try to ascertain the problem. The interpreter also points to the line of code (line 2) at which the error occurred. Armed with this information, we should be able to figure out what is going on and then build a plan to fix it.

It’s important to note that when an exception occurs, the application execution stops. The error above occurred in line 2, and lines 3 and 4 were never executed. Why? Well, let’s look at the rest of the code. If a FileNotFoundError occurred on line 2, that means the file was never opened because it does not exist. If the interpreter continued executing our code, line 3 would ask it to try to read from that file. That would also then result in an error because we’re trying to read from a file that isn’t open and doesn’t even exist. Line 4 then tries to close the file. This would also result in an error because the file isn’t open to begin with. So, mercifully, the interpreter instead just shuts down execution of the program when it finds an exception.

But how do you fix or avoid an exception? What if we need to ask the user for a number and then use it as the denominator in a division problem? Or what if we need to ask the user for a filename and then try to read from it? Are we doomed to have our application fail based on certain input? Luckily, no. Like I said, exceptions are, unfortunately, a normal occurrence in programming. The trick is to anticipate them and code in a way that will not make your application crash when one occurs. Let's look deeper into how we can do that.

We can code for exceptions, and if we handle them correctly, we can prevent the application from crashing when an exception occurs. The first step in doing so is to use the "try" statement preceding any block of code in which we think an exception may occur. The try statement consists of simply the word "try" followed by a colon. Any code in the indented block that follows the try statement will be set up for the proper handling of exceptions. Next, you use the "except" clause after the try block of code. The except clause will catch any exceptions of the type specified that occur within the try block. For example, see the modified divide by zero code below:

**Figure 137: Divide by Zero Code with Try/Except**

```
try:
    value = input()
    result = str(5 / int(value))
    print("5 divided by " + value + " = " + result)
except ZeroDivisionError:
    print("You tried to divide by zero!")
    print("We're now at the end of the program.")
```

Source: Aaron Reed, 2020.

First notice the try statement at the beginning of the code. The three indented lines of code that follow are part of this try block of code; if an exception occurs within that block, it can be captured in the except clause that follows. Note the format of the except clause: the word "except," followed by a type of error, followed by a colon. In the case above, the type of error specified is ZeroDivisionError. That means that if a ZeroDivisionError occurs in the try block, instead of crashing the application, execution of the program will jump to the except block. Any code in the except block (the indented code after the except clause) will be executed at that point. After either the try code is executed without an exception error or a ZeroDivisionError occurs and the except block is executed, the program will resume normal operation after the except clause.

Let's take a look at the code above running without an error. See the following figurecxc:

**Figure 138: Try Block Executing without Error**

```
try:
    value = input()
    result = str(5 / int(value))
    print("5 divided by " + value + " = " + result)
except ZeroDivisionError:
    print("You tried to divide by zero!")
print("We're now at the end of the program.")
```

50  
5 divided by 50 = 0.1  
We're now at the end of the program.

Source: Aaron Reed, 2020.

In the code snippet above, the user entered `50` when prompted. Notice that each line of the try block is executed, the except block is not executed, and the print statement at the end of the program is executed.

Now let's see what happens when the same code is run with an error in the try block:

**Figure 139: Try Block Executing with Error**

```
try:
    value = input()
    result = str(5 / int(value))
    print("5 divided by " + value + " = " + result)
except ZeroDivisionError:
    print("You tried to divide by zero!")
print("We're now at the end of the program.")
```

0  
You tried to divide by zero!  
We're now at the end of the program.

Source: Aaron Reed, 2020.

In the code above, the user entered `0` when prompted. In line 2 of that block, this generates an exception. Note that the print statement in the try block is never executed. This is because the error occurred before this line and execution immediately jumps to the except block. Note that the except block is executed and then the print statement at the end of the block is executed.

It's also important to note that the error type specified in the except clause is optional. You can instead just use the "except" keyword followed by a colon. In that case, any exception that is thrown will be caught in the except clause. It is considered a better practice, however, to be as specific as possible with exceptions. You don't want to just capture all excep-

tions indifferently because you won't really know what has happened in your program and why it could not continue normal execution. That's not a comforting thought as a programmer. Instead, you want to anticipate potential exceptions and capture them specifically.

Let's look at the file open exception from the previous example and how we might use try/except to handle the potential FileNotFoundError exception.

**Figure 140: Try Block - Reading a File Successfully**

```
try:
    file_name = input()
    my_file = open(file_name, "r")
    print(my_file.read())
    my_file.close()
except FileNotFoundError:
    print("File not found")
print("End of program")
```

StarkillerBasePlans.txt  
This is basically a giant Death Star and you destroy it the same way.  
See DeathStarPlans.txt for info on how to destroy.  
End of program

Source: Aaron Reed, 2020.

When executing the code above, if the user enters a valid filename, the file contents will be printed as execution of the program continues through the try block, then the except block will be skipped, and the last line of the program will print an "End of program" message. For the example above, we have created a file called StarKillerBasePlans.txt with a couple of lines of text. You see from the program output that those lines of text have printed, followed by the "End of program" message.

Let's look at what would happen if the file did not exist:

**Figure 141: Try Block - Reading a File with Error**

```
try:
    file_name = input()
    my_file = open(file_name, "r")
    print(my_file.read())
    my_file.close()
except FileNotFoundError:
    print("File not found")
print("End of program")
```

DeathStarPlans.txt  
File not found  
End of program

Source: Aaron Reed, 2020.

In the example above, the file `DeathStarPlans.txt` does not exist. The execution of the program stops at the “file open” command in line 2 of the try block because a `FileNotFoundError` exception was generated. Execution of the program then jumps to the except block, which is why we see “File not found” in the output. At the end of the program, we see the final “End of program” message.

In some cases, you may want to ensure some code runs as part of the try statement regardless of whether or not an exception occurred. You can do this by adding a “finally” clause to the end of the try/except block. Code in the finally block will always run after a try/except block regardless of errors. See a couple of examples below:

**Figure 142: Finally Block without Error**

```
try:
    my_variable = 10/10
except ZeroDivisionError:
    print("Divided by zero!")
finally:
    print("Here's our finally block...")
print("End of program")
```

Here's our finally block...  
End of program

Source: Aaron Reed, 2020.

**Figure 143: Finally Block with Error**

```
try:
    my_variable = 10/0
except ZeroDivisionError:
    print("Divided by zero!")
finally:
    print("Here's our finally block...")
print("End of program")
```

```
Divided by zero!
Here's our finally block...
End of program
```

Source: Aaron Reed, 2020.

In the first example, we divide 10 by 10, which does not generate an error. You can see from the program output that the finally block executes and then the “End of program” message is generated. In the second example, we divide 10 by 0, which does generate an error. You can see from the program output that both the except block and the finally block are executed before the program terminates.

You may find times when it would be important for you to raise an exception on your own as the programmer. You can do this by using the “raise” command, followed by the exception that you’d like to raise. See an example below:

**Figure 144: Exception Raised by Programmer**

```
try:
    raise ZeroDivisionError
finally:
    print("Here's our finally block...")
print("End of program")
```

```
Here's our finally block...
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-5-e66d22b1a14b> in <module>
      1 try:
----> 2     raise ZeroDivisionError
      3 finally:
      4     print("Here's our finally block...")
      5 print("End of program")

ZeroDivisionError:
```

Source: Aaron Reed, 2020.

In the example above, you see that we raise the `ZeroDivisionError` by using the `raise` command. However, note that we have a `finally` clause but no `except` clause. Without the `except` clause, we aren't capturing that exception, so the program will terminate due to the error we raised. An important thing to note, however, is that even though an exception was raised, the `finally` block still executes before the program terminates.

## 5.3 Logs

Thus far, we've talked about syntax errors and exceptions. Syntax errors occur when code does not meet Python syntax, which prevents the code from being executed. Exceptions occur when code is properly formed syntactically but an error occurs that would cause program termination. Another type of coding error can occur when there is a logical error in the code even though it might be syntactically correct and might not cause an exception. For example, what if you wanted to add two variables (`a` and `b`) and instead of typing `a + b`, you accidentally typed `a - b`? This code is still syntactically correct, and, assuming `a + b` wouldn't cause an exception, `a - b` won't cause an exception either. Yet when you run this code, you are likely to find that the result is very different from what you anticipated. This type of error is called a "logical error."

Logical errors are perhaps the most difficult errors to find in programming because you don't get the benefit of the interpreter pointing to the likely location of the error. Instead, the only way to tell if you have a logical error is to carefully compare obtained results to expected results to make sure the program is doing what you wanted it to do. Once you identify that you have an unintended result, you have to dig into the code and try to figure out where and when the logical error occurs.

This process of identifying and fixing the cause of a logical error is called **debugging**, which can be a painstaking process. Luckily, there are some built-in tools that can help us through the debugging process. Logging, or writing data to log files as the program executes, is one such tool. Logging allows a programmer to get a better sense of program flow and what is happening within the program at critical points.

### Debugging

The process of locating and rectifying errors, flaws, faults, and defects in code, known as "debugging," is improved if it is done systematically and methodically.

In order to use logging in Python, we have to import the logging library to our program. This is done with one line of code: `import logging`. Put that line at the top of your program and you will then be able to write code using Python's built-in logging features.

Logging in Python has five different levels of severity in terms of which information is being logged. Those are, in order of lowest severity to highest, the following:

- `debug`
- `info`
- `warning`
- `error`
- `critical`

When programming, you can log messages through one of those severity levels. You can also set a severity level for your program so that during program execution, only the severity levels equal to or higher than that specified will be logged. For example, let's look at the code and output below:

**Figure 145: Logging Levels**

```
import logging

logging.debug("Debug is the lowest log level in severity")
logging.info("Info is the second lowest log level")
logging.warning("Warning is the third level")
logging.error("Error is the fourth level")
logging.critical("Critical is the fifth and highest level")
```

```
WARNING:root:Warning is the third level
ERROR:root:Error is the fourth level
CRITICAL:root:Critical is the fifth and highest level
```

Source: Aaron Reed, 2020.

As you can see in the code snippet above, the way to create a log entry is to use the logging object followed by a period and the level of the log entry you wish to create (debug, info, warning, error, or critical), followed by the log message in parentheses. Notice also that the only messages that appeared in the output (the red box below the code snippet) are those of the warning, error, and critical levels. That is because the Python default logging level is "warning," meaning that only messages of a level equal to or more severe than the warning level will be logged.

To customize the logging function within Python, use the `logging.basicConfig()` method. The `basicConfig` method allows you to change the level of log output as well as write logs to files. To change the logging level, set the `level` parameter in the call to `basicConfig` as shown below:



**Figure 146: Changing the Logging Level**

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("Debug is the lowest log level in severity")
logging.info("Info is the second lowest log level")
logging.warning("Warning is the third level")
logging.error("Error is the fourth level")
logging.critical("Critical is the fifth and highest level")
```

```
DEBUG:root:Debug is the lowest log level in severity
INFO:root:Info is the second lowest log level
WARNING:root:Warning is the third level
ERROR:root:Error is the fourth level
CRITICAL:root:Critical is the fifth and highest level
```

Source: Aaron Reed, 2020.

Above, you'll see the logging level was set to DEBUG, meaning any logging equal to or more severe than debug will be shown. In the output, we can see that all five levels of debugging are shown when the level is set to DEBUG. An example of writing logs to a file can be seen below:

**Figure 147: Writing Log Data to a File**

```
import logging

logging.basicConfig(filename="mylog.log", filemode="w", level=logging.DEBUG)

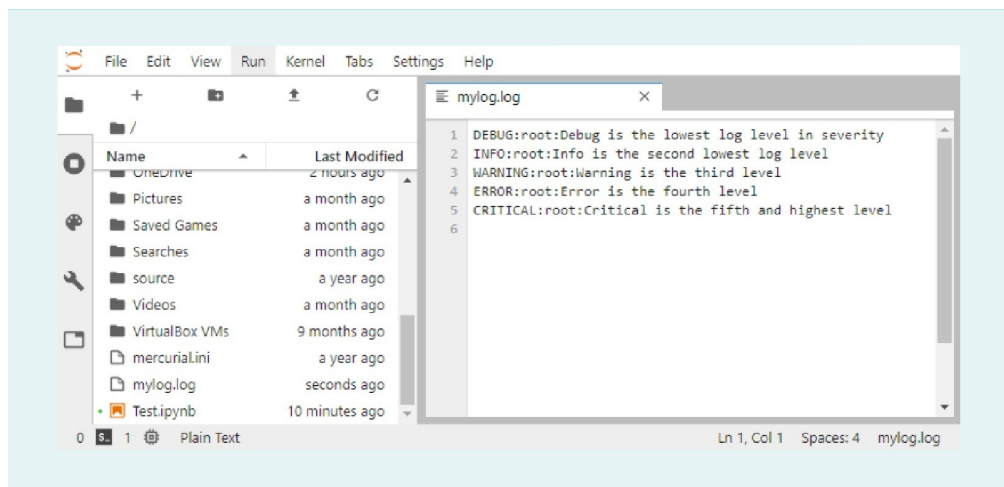
logging.debug("Debug is the lowest log level in severity")
logging.info("Info is the second lowest log level")
logging.warning("Warning is the third level")
logging.error("Error is the fourth level")
logging.critical("Critical is the fifth and highest level")
```

Source: Aaron Reed, 2020.

When you run the code above, there is no output. Why is that? Well, first, the code itself does not have any output (e.g., there are no print statements that would output anything to the console), and second, when we direct log info to a file, it writes the data to a file instead of the console.

After running the code above, you should see a new file with the name "mylog.log" in the list of files in Jupyter Lab. Opening that file will show the contents, where you can see the output from the log statements we created. See below:

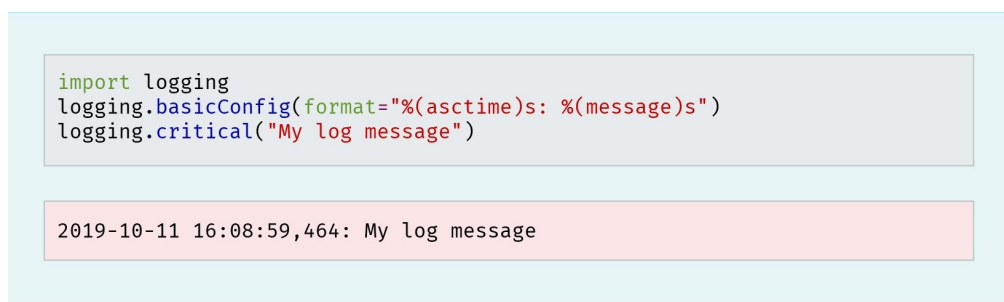
**Figure 148: Opening and Reading a Log File**



Source: Aaron Reed, 2020.

There are also other ways to customize your log messages. You can use various attributes to get more detailed info on what is going on in your program. You set these attributes in the `basicConfig` function by assigning a string to the `format` parameter as follows:

**Figure 149: Formatting Log Output**



Source: Aaron Reed, 2020.

The format string uses a certain syntax for message formatting as shown above. Using keywords within parentheses preceded by a percent symbol (%) and followed by a character denoting the data type, you can add different info to your log records. The “asctime” attribute above will output the date and time when the log was created. The “message” attribute above is a placeholder for the actual log message.

There are a number of other log attributes that can be used to pull data into your logs. Some of the more popular ones are listed below:

**Table 8: Logging Format Attributes**

Attribute	Format	Description
asctime	%(asctime)s	Gets the log entry creation date and time
filename	%(filename)s	Gets the name of the file
funcName	%(funcName)s	Gets the name of the function in which the log entry was created
lineno	%(lineno)d	Gets the line of code where the log entry was created
process	%(process)d	Gets the process ID that created the log entry
processName	%(processName)s	Gets the process name that created the log entry

Source: Aaron Reed, 2020.

Logging is a very effective tool for developers that can lead to a better understanding of application flow and improved bug and error tracking.



#### **SUMMARY**

Programmers make mistakes. That's just the way it is. No matter how small the program, there will almost always be errors. There are several different types of common errors, and there are built-in ways that Python helps us deal with them.

Syntax errors occur when code does not conform to Python syntax rules. These are detected by the interpreter and prevent program execution. The interpreter provides an error message to the programmer that contains helpful information on the syntax error, such as the line number and the nature of the error. By analyzing the message provided by the interpreter, programmers can track down and rectify syntax errors more effectively.

Exceptions occur when the code meets the syntax rules, but an error occurs when the program runs. Exceptions end program execution. The interpreter provides information to the programmer or user regarding the nature of the exception and the location of the error. Exception handling provides a way for programmers to plan for and manage exceptions.

Logging is a technique that can be valuable for any programmer. Through log information, programmers can gain a better sense of application flow and logic. Some errors do not manifest themselves in appli-

cation crashes but instead show up only in program logic. Logging can help programmers trace through code to better understand the program behavior and track down and fix logic errors.

# UNIT 6

## MODULES AND PACKAGES

### STUDY GOALS

On completion of this unit, you will have learned ...

- what Python namespaces are and how to use them.
- how to navigate and use Python documentation.
- about various data science packages that can be used with Python.

## 6. MODULES AND PACKAGES

### Case Study

Kyle and Morgan are working diligently on their soccer application. They have a much better sense of Python as a language, what it can do, and how it works. They now know how to deal with errors as they arise in coding and during application execution. They are ready to take the next step as developers. As they continue their journey, and as their application becomes more complex, a few questions pop into their heads:

- Are there ways to compartmentalize code for reusability? If so, how does that work?
- Regarding function and variable visibility, what are the Python visibility rules beyond scope as we currently understand it?
- What do we know about popular data science packages for Python and how they are used?

### 6.1 Usage

To use logging features in Python, we must add the line `import logging` to the code. This line imports libraries of code that enable logging features.

This import feature is an effective way of creating separation in code as well as enabling its reuse. The code that handles the logging, for example, is not visible to us as programmers. But the import feature allows the code to be visible to our program, enabling us to use the functions associated with logging.

#### Modules

At its core, a module is just a Python file with code in it. You can import entire files or specific functions from those files for use in other applications.

You can create your own custom **modules** in Python as well. Why would you do that? Well, let's say as Morgan and Kyle build their soccer analytics program, they come up with some code that calculates the likelihood that one player will score against another. They would likely use that code in their soccer application, since that application is meant to help soccer players learn more about and improve their gameplay. However, they may also have plans to build another application later that uses the same functionality. What if, for example, they wanted to use that code for some applications designed for fans to see what player combinations might be the best against another team? Or, maybe they will want to use that same code in a soccer game at some point.

Modules allow you to take code, such as the soccer code mentioned above, put it in a separate module, and reuse it in multiple applications. Creating a module is easy. Simply write the code you want and save it in a `.py` file. You can then use `import` to access functions in that file from other files by using the syntax `from <filename without the .py extension> import <function name>`.

## 6.2 Namespaces

When importing functions from modules, the function is usable within your code if you simply call the function by name. For example, the math library in Python contains a function called “floor” that will return the largest integer less than or equal to a value passed in. So, if 6.5 is passed to floor, floor returns 6.

However, if you try to call floor without importing it, you will get a syntax error because no function called floor exists in your program unless you import it from math or define one yourself. See below:

**Figure 150: Function Not Defined: Floor**

```
floor(6.5)

-----
NameError                                Traceback (most recent call last)
<ipython-input-2-b7d7628089a2> in <module>
----> 1 floor(6.5)

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

Source: Aaron Reed, 2020.

We can import floor from math by using the syntax mentioned in the previous section: from <library> import <function>. Below, we import floor from the math library, enabling us to use the floor function:

**Figure 151: Importing Floor From Math**

```
from math import floor

floor(6.5)

6
```

Source: Aaron Reed, 2020.

Importing code from libraries is essential as you start building more and more complex applications. If something is already written, why write it again? Particularly if you trust the code you’re importing, you can assume some level of quality and efficiency in that code that might take you a very long time to match. Instead, import existing code and rely on it in your application to save time and effort.

One complication that can arise when importing a lot of code is name duplication. As we've discussed, it is important to name variables and functions in ways that make sense and help you understand the code. That is typically easy to do in a small program. But some programs comprise millions of lines of code—that's a lot of variables and functions! As your programs grow in size, you will likely have times when you will struggle to come up with names because the names you want to use already exist. Importing code further complicates things because you will be importing names into your code that maybe you would have wanted to use yourself. For example, the code below imports `floor` from `math`, but then creates a `floor` function to return a string representation of the floor of a building:

Figure 152: Redefining Floor

```
from math import floor

print(floor(6.5))

def floor(a):
    if a == 1:
        return "First floor"
    elif a == 2:
        return "Second floor"
    elif a == 3:
        return "Third floor"

print(floor(2))
```

6  
Second floor

Source: Aaron Reed, 2020.

Notice what happens when we define the `floor` function and then call it. The new `floor` function is used, and the imported `math` function is no longer available. This problem is caused because both floors are in the same namespace.

#### Namespace

In Python, a namespace is a system to help make the names of functions and variables unique and to prevent their duplication across modules and packages.

In Python, a **namespace** represents a way of outlining categories of names to reduce name reuse and overwriting. Python namespaces include the following:

- built-in namespace: contains all built-in functions and exceptions.
- global namespace: contains all the names of variables and functions you create in your program that exist outside of functions
- local namespace: contains only names within a given function.

For example, look at the code snippet below:



**Figure 153: Namespaces**

```
my_str = "Hello, World!"
print(my_str)

def print_it_x_times(str, x):
    a = 0
    while a < x:
        print(str)
        a += 1

print_times = 5
print_it_x_times("Hello again, World!", print_times)
```

Hello, World!  
Hello again, World!  
Hello again, World!  
Hello again, World!  
Hello again, World!  
Hello again, World!

Source: Aaron Reed, 2020.

In the code above, the following names belong to the following namespaces:

**Table 9: Namespace Analysis**

Namespace	Namespace Definition	Names
Built-in	Contains all built-in functions and exceptions	print
Global	Contains all the names of variables and functions you create in your program that exist outside of functions	my_str print_it_x_times print_times
Local	Local namespace: Contains only names within a given function	str x a

Source: Aaron Reed, 2020.

Again, the reason namespaces exist is to provide you with different levels of names for variables and functions. This concept can be illustrated by importing math in a different way. Previously, we imported select functions from math, such as `from math import floor`. This brings the floor function into the global namespace but also creates a conflict with any other reference to a name of floor in your global namespace.

By importing the entire math library instead, we can avoid these duplicate names. For example, the code below imports the math library in its entirety:

Figure 154: Import Math Library

```
import math
math.floor(6.5)

6
```

Source: Aaron Reed, 2020.

The benefit of importing the entire library is that now the names for variables and functions in the library are accessed with dot notation represented by <library name>.<function or variable name> as can be seen above with the call to `math.floor`. Importing libraries this way allows you to have a floor reference in your own global namespace and eliminates conflict with a library function of the same name.

## 6.3 Documentation

We've talked about code readability as we've progressed through this book. Improved readability was one of the fundamental principles behind Python's creation. Creating readable code is critical in any program for improving maintainability and reducing the time needed to track down bugs or add enhancements.

### Commenting

The act of commenting code is a critical part of programming in any language. Although comments do not impact the execution of a program, they can dramatically improve code readability, which significantly reduces debugging and maintenance time.

By **commenting** your code in Python, you can improve its readability. You are effectively telling anybody reading your code why you're doing what you're doing with that code. A comment in Python code is like an author's note, and the comment will be completely ignored by the interpreter. To create a comment, use the # symbol. Anything to the right of the # symbol is part of the comment. Combining the practice of solid naming of variables and functions with the inclusion of comments in code can greatly improve code readability. For example, look at these two functions that do the same thing. Spend a few minutes looking at the first function before moving to the second. Try to figure out what it does and why. Then move to the second and do the same thing.

Figure 155: Difficult to Read Code

```
def do_math(a, b):
    c = a * b
    return c
```

Source: Aaron Reed, 2020.

**Figure 156: Easy to Read Code**

```
def get_sales_tax(item_price, tax_rate):  
    # This function returns the tax for the purchase of an item  
    # item_price is the price of the item being purchased  
    # tax_rate should be a floating-point tax rate (e.g., .07)  
    tax = item_price * tax_rate  
    return tax
```

Source: Aaron Reed, 2020.

Both functions in the previous two examples do exactly the same thing: they calculate the tax on the sale of an item. But as you examined the first function, you likely had to stop and think about what this function was doing, what its purpose was, and how and when you might use it. As you read the second function, you likely had to spend a lot less time figuring it out. That's the power of properly named variables and functions and informative comments.

There is another tool that can be quite useful in creating readable code. Python has a built-in way of creating easy-to-use documentation for your code. You can tap into this built-in documentation by using docstrings. Docstrings are created by using triple double quotes (""") before and after the docstring. A typical usage of docstrings is at the start of a function. In this case, you would put the docstring in the first lines of a function. Let's look at an example below:

**Figure 157: Simple Docstring**

```
def add_two_numbers(a, b):  
    """ Adds two numbers together and returns the result. """  
    return a + b
```

Source: Aaron Reed, 2020.

The single-line docstring above looks like a comment, and like a comment, it was not picked up by the interpreter. We know it was ignored by the interpreter because that line does not follow Python syntax but we did not get a syntax error. So, it must have been ignored, right? Well, not quite. Yes, the docstring was not interpreted as code by the compiler, but it is treated differently than a comment. Specifically, there are some functions that can be used to retrieve the docstrings in code and to help you and others better understand your code.

Let's say you wanted to see what the `add_two_numbers` function was supposed to do. You can use the `help` function to get a sense of the purpose of the function and how to use it from your docstring. See below:

**Figure 158: Simple Docstring - Using help**

```
def add_two_numbers(a, b):  
    """ Adds two numbers together and returns the result. """  
    return a + b  
  
help(add_two_numbers)
```

```
Help on function add_two_numbers in module __main__:  
  
add_two_numbers(a, b)  
    Adds two numbers together and returns the result.
```

Source: Aaron Reed, 2020.

Notice the output when calling `help`. The name of the function and its arguments are given, followed by your docstring. This may seem like a trite use for docstrings at the moment, and it probably is. The real power of having code documented with docstrings comes when you import code from another module. You may or may not have access to the code itself in that module, but if you want to see what a particular function does, you can use the `help` function to better understand it—as long as the developers documented that code with docstrings.

Let's say Morgan and Kyle create a module for their soccer application that calculates various soccer statistics. In that module, they create a function called `score_chance` that returns the scoring chance of a particular player in a penalty kick scenario. If they were good about documenting their code, an overly simple version of that function may look something like this:

**Figure 159: Multiline Docstring**

```
def score_chance(player):  
    """  
    Gets a player's chance of scoring on penalty kick.  
  
    This function returns the penalty kick scoring chance  
    for a player specified in the function argument.  
  
    Arguments:  
    player(string) – the name of a player  
  
    Returns:  
    float representing the scoring chance of that player  
    """  
    if player == "Joe":  
        return .54  
    elif player == "Bob":  
        return .46  
    else:  
        return 0
```

Source: Aaron Reed, 2020.

Notice a couple of things about the docstring above. First, it's multiline. Multiline docstrings are created by placing the triple double quotes on the first and last line of the docstring. Second, notice the depth of description for that function. If a user were to ask for help on the `score_chance` function, there would be little doubt of the purpose of that function and how to use it. Here's what would be outputted when using `help(score_chance)`:

**Figure 160: Multiline Docstring — Using help**

```
help (score_chance)  
  
Help on function score_chance in module __main__:  
  
score_chance(player)  
    Gets a player's chance of scoring.  
  
    This function returns the scoring chance for a player  
    specified in the function argument.  
  
    Arguments:  
    player(string) – the name of a player  
  
    Returns:  
    float representing the scoring chance of that player
```

Source: Aaron Reed, 2020.

Docstrings provide a powerful way to help programmers who will use your code to better understand the purpose of that code even when that code exists elsewhere in an imported module.

## 6.4 Popular Data Science Packages

In this section, we will take a look at a few important libraries for data science programming in Python. Python has a very robust and active support community, and some very helpful libraries for use in data science programming in Python are available at no cost. Those include NumPy, Matplotlib, SciPy, pandas, and scikit-learn. Each of these will be briefly discussed in this section.

### NumPy

NumPy is one of the most important libraries for data science in Python. It provides a variety of data types and functions for use in computations with arrays. NumPy provides developers with the following:

- Built-in, robust data types for storing and working with data sets that are more efficient than Python's built-in lists
- Improved speed over standard Python when manipulating large amounts of data
- A wide range of built-in functions for dealing with statistical analysis on large data sets

You can use NumPy by importing the NumPy library (`import numpy`). The code snippet below shows a very basic example of some features of NumPy.

Figure 161: NumPy – A Few Examples

```
import numpy

# Create a simple NumPy array
my_array = numpy.array([0, 5, 10, 15, 20, 25])

# Print the data type (ndarray)
print(type(my_array))

# Multiply the array by 5 and assign it to a new array
my_squares_array = my_array * 5

# Print the array values
print("First array=" + str(my_array))
print("Second array=" + str(my_squares_array))

# Compute and print the sums
print("First array sum=" + str(my_array.sum()))
print("Second array sum=" + str(my_squares_array.sum()))

# Compute and print the means
print("First array mean=" + str(my_array.mean()))
print("Second array mean=" + str(my_squares_array.mean()))

# Compute and print the standard deviations
print("First array standard deviation=" + str(my_array.std()))
print("Second array standard deviation=" + str(my_squares_array.std()))

<class 'numpy.ndarray'>
First array=[ 0  5 10 15 20 25 ]
Second array=[ 0 25 50 75 100 125 ]
First array sum=75
Second array sum=375
First array mean=12.5
Second array mean=62.5
First array standard deviation=8.539125638299666
Second array standard deviation=42.69562819149833
```

Source: Aaron Reed, 2020.

As you can see from the code above, NumPy uses its own data types for arrays. These data types are compact and require less storage than Python's built-in data types. NumPy provides a range of functions for use in manipulating and computing data on lists in these built-in data types.

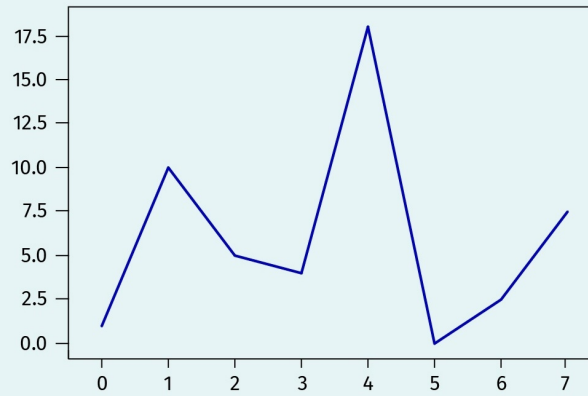
## Matplotlib

Matplotlib is a library that provides a wide range of flexible, customizable, and easy-to-use plotting functions for displaying data. Matplotlib includes functions for graphs, histograms, bar charts, scatterplots, and more. The library can plot a wide range of data types, including arrays from NumPy.

The example below plots a simple NumPy array:

Figure 162: Matplotlib Simple Plot

```
import numpy
import matplotlib.pyplot as plt
my_array = numpy.array([1, 10, 5, 4, 19, 0, 2, 8])
plt.plot(my_array)
plt.show()
```

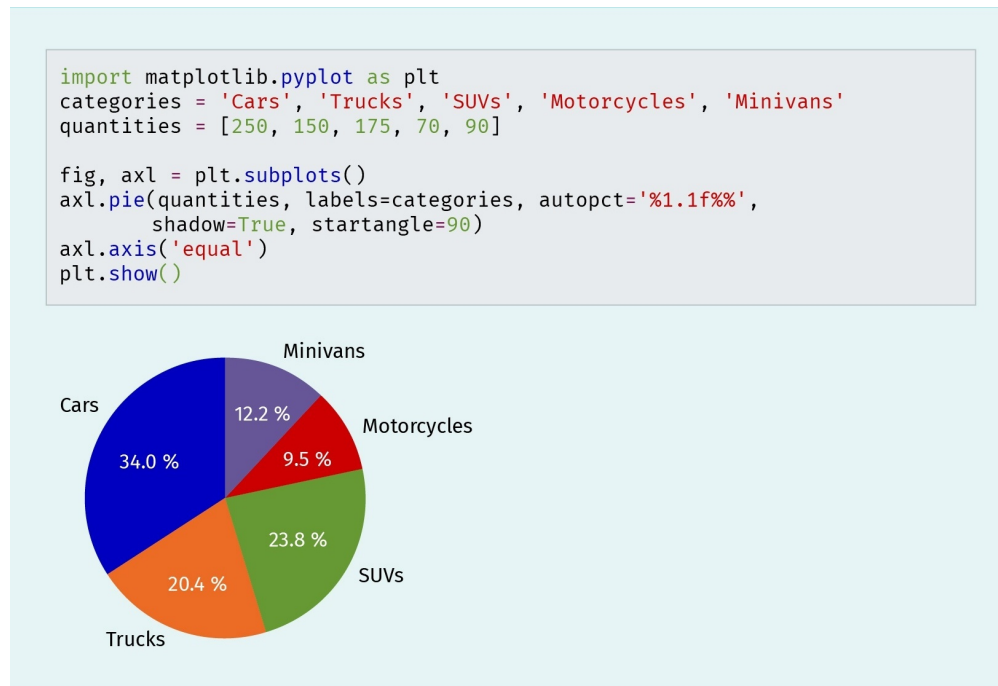


Source: Aaron Reed, 2020.

Here's another example of a simple pie chart based on categories and quantities of items:



**Figure 163: Matplotlib Simple Pie Chart**



Source: Aaron Reed, 2020.

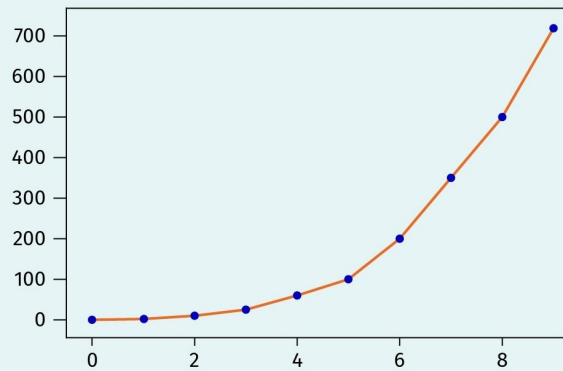
## SciPy

SciPy offers a range of functions used for scientific computations. With SciPy, you can implement linear algebraic functions, interpolation, signal processing, and more. Interpolation is a way of estimating the value of a function between two data points. An example of SciPy interpolation plotted with Matplotlib is shown below:

Figure 164: SciPy Interpolation Plot

```
import numpy
import matplotlib.pyplot as plt
from scipy import interpolate

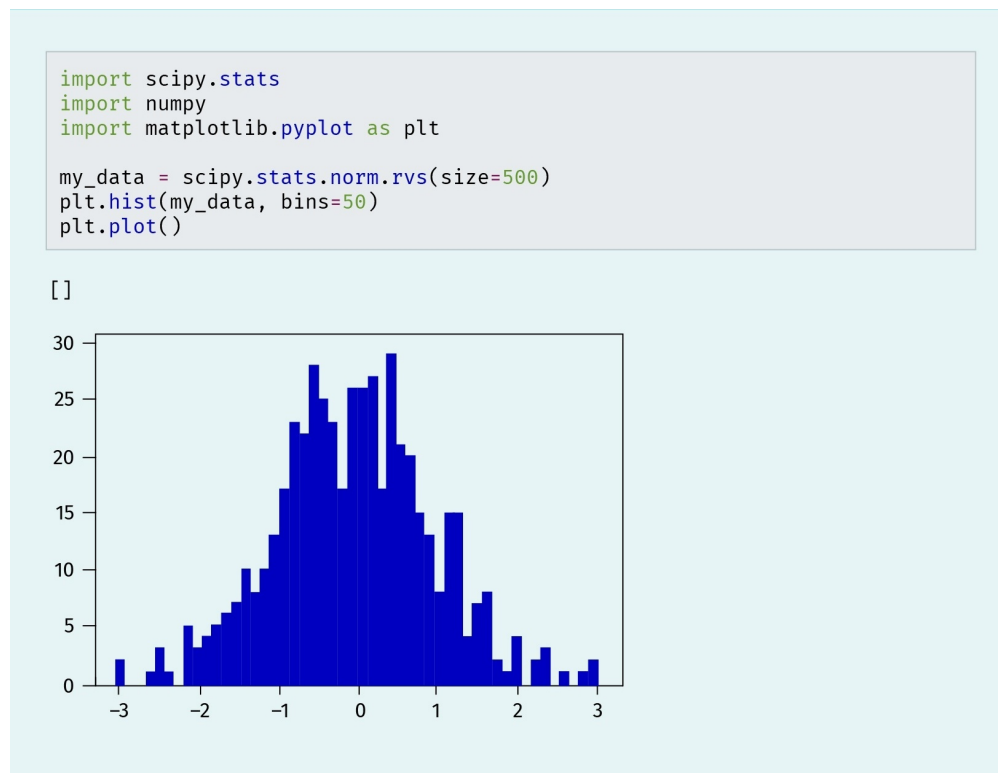
x = numpy.arange(0, 10)
y = x * x * x
f = interpolate.interp1d(x, y)
intx = np.arange(0, 9, 0.1)
inty = f(intx)
plt.plot(x, y, "o", intx, inty, "-")
plt.show()
```



Source: Aaron Reed, 2020.

SciPy allows you to quickly build random data sets from which you can run calculations and try out functionality of the library. The example below creates a sample of 500 data points in a normal distribution, graphed in a histogram with Matplotlib:

**Figure 165: SciPy Normal Distribution Histogram**



Source: Aaron Reed, 2020.

## **Pandas**

Pandas provides a variety of data structures for use in scientific applications. Pandas data structures are high-performance, extremely efficient, and fairly easy to use. The library also includes a range of functions for manipulating data in those data structures.

The pandas DataFrame is a two-dimensional, table-like structure that is heavily used in scientific applications. With pandas, you can easily read .csv files into DataFrame structures for processing in your application. The example below reads a .csv file with car data and shows the top five rows of that data once read into a DataFrame:

Figure 166: Pandas Car Data

```
import pandas
car_data = pandas.read_csv('mtcars.csv')
car_data.head()
```

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Source: Aaron Reed, 2020.

Pandas allows you to quickly sort, filter, and manipulate the data once it is in a DataFrame. The example below shows the same car data but filtered to display only those cars that have mpg > 25:

Figure 167: Pandas Efficient Cars Data

```
import pandas
car_data = panda.read_csv('mtcars.csv')
efficient_cars = car_data["mpg"] > 25
print(car_data[efficient_cars])
```

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	\
17	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	
18	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	
19	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	
25	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	
26	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	
27	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	

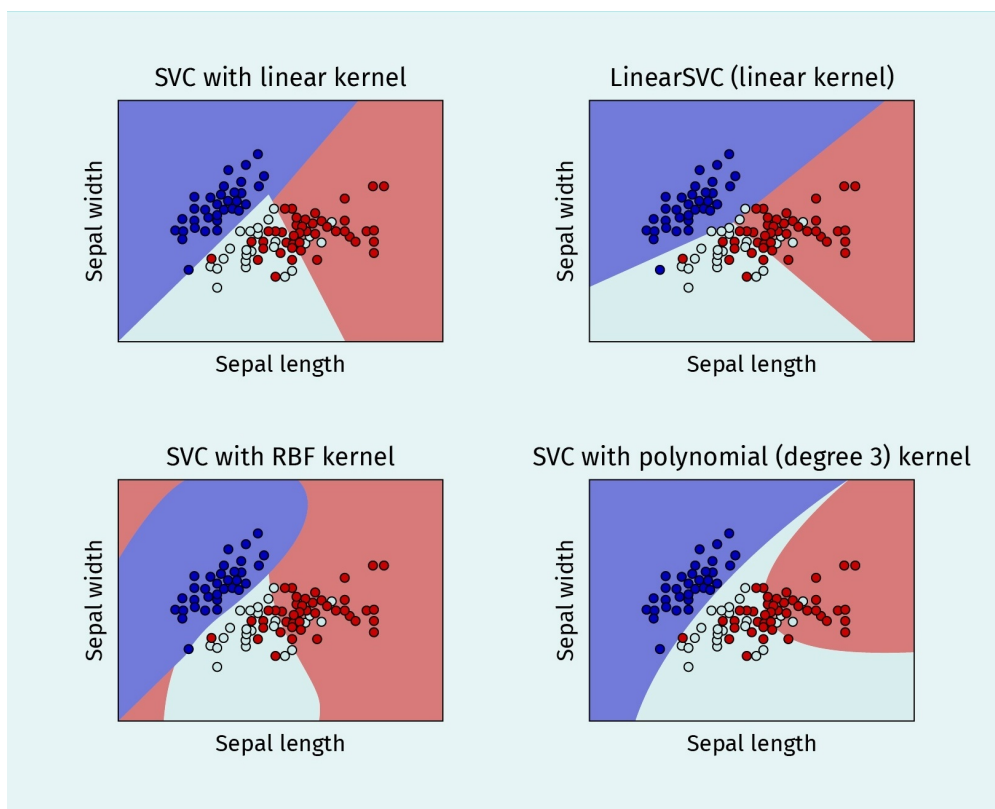
Source: Aaron Reed, 2020.

## Scikit-Learn

Scikit-Learn provides functions to support machine learning in Python. It includes algorithms for classification, regression, and clustering.

Classification is a machine-learning technique where algorithms attempt to classify data based on certain characteristics. For example, you may be given a set of data with attributes. Based on that data, the algorithm can classify a given record in that data as describing a person, animal, or plant. Once the algorithm has “learned” to classify, you can send it new rows of data and it will classify those rows according to what it has learned about the data. Below is an image from scikit-learn depicting various classification algorithms:

**Figure 168: Classification Algorithms**

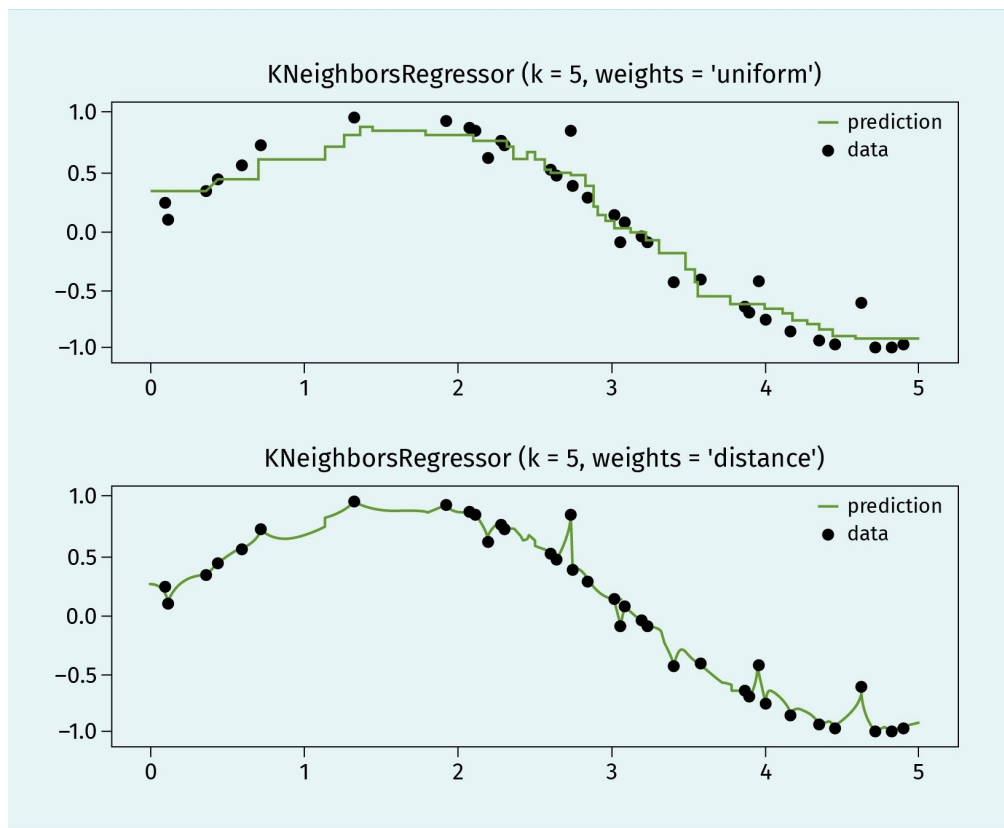


Source: Scikit, 2019

Regression algorithms are designed to analyze a series of inputs and, based on those inputs, predict the output of new values added to the system. For example, a dataset may contain a large number of attributes about parents and the hair colors of their children. Based on the inputs, the algorithm would learn that certain combinations of attributes in parents led to certain outcomes in terms of child hair color. Then, as new data is added, the algorithm can predict the hair color of children based on the attributes of parents.

The image below from scikit-learn shows the results of a regression algorithm called “nearest neighbors”:

Figure 169: Regression Algorithms

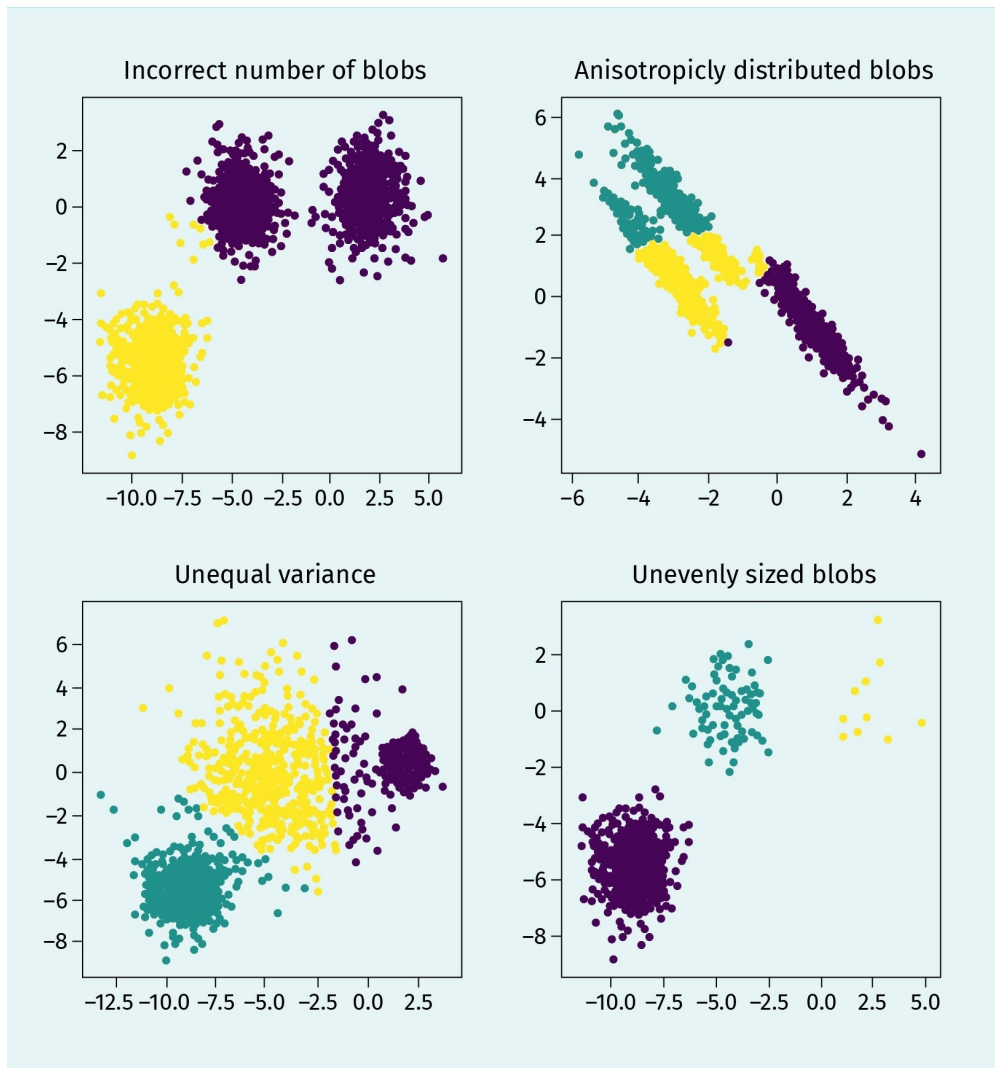


Source: Scikit, 2019

Clustering is a technique where algorithms attempt to group data based on a set of input attributes. Through these groups, algorithms then attempt to predict outcomes based on new inputs.

Below is an image from scikit-learn depicting various clustering algorithms at work:

Figure 170: Clustering Algorithms



Source: Scikit, 2019

One of Python's greatest strengths is its powerful and robust utility in the field of data science. NumPy, Matplotlib, SciPy, pandas, and scikit-learn are tremendous libraries to facilitate complex and effective scientific computing in Python.



#### SUMMARY

Naming variables and functions with clearly defined, distinct names can be a daunting task once applications reach a certain complexity and length. Namespaces separate naming schemes to help prevent naming conflicts.

Documenting code through comments and docstrings is an essential part of any programmer's job. Docstrings can be used to generate documentation help, particularly for modules developed by third parties.

Python's popularity is partly due to its impressive performance in data science applications. There are abundant resources for Python developers in data science and machine learning. NumPy, Matplotlib, SciPy, pandas, and scikit-learn are all packages that can facilitate data science development in Python.



# BACKMATTER

# LIST OF REFERENCES

- Guo, P. (2014, July 7). Python is now the most popular introductory teaching language at top U.S. universities [blog]. Retrieved from <https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext>
- PYPL Index. (2019). PYPL popularity of programming language [index]. Retrieved from <http://pypl.github.io/PYPL.html>
- Python. (2019). PEP 8—Style guide for Python code [guide]. Retrieved from <https://www.python.org/dev/peps/pep-0008/>
- Scikit. (2019a). Clustering [report]. Retrieved from <https://scikit-learn.org/stable/modules/clustering.html>
- Scikit. (2019b). Nearest neighbors [report]. Retrieved from <https://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbors-regression>
- Scikit. (2019c). Support vector machines [report]. Retrieved from <https://scikit-learn.org/stable/modules/svm.html#classification>
- TIOBE. (2019). TIOBE Index for August 2019 [index]. Retrieved from <https://www.tiobe.com/tiobe-index/>
- US News and World Report (2019). 100 best jobs [website]. Retrieved from <https://money.usnews.com/careers/best-jobs/rankings/the-100-best-jobs>

# LIST OF TABLES AND FIGURES

Figure 1: Introductory Programming Languages Used in Top 39 U.S. Universities .....	11
Figure 2: TIOBE Index for August 2019 .....	12
Figure 3: Anaconda - Select Your Platform .....	14
Figure 4: Anaconda - Select Version .....	15
Figure 5: Anaconda Install - Setup .....	15
Figure 6: Anaconda Install - License Agreement .....	16
Figure 7: Anaconda Install - Installation Type .....	17
Figure 8: Anaconda Install - Location .....	17
Figure 9: Anaconda Install - Advanced Options .....	18
Figure 10: Anaconda Prompt .....	19
Figure 11: Anaconda Prompt - Python Interpreter .....	20
Figure 12: Anaconda Prompt - Hello, World! In Python .....	20
Figure 13: Hello World in Java .....	21
Figure 14: Hello World in C .....	21
Figure 15: Compiled and Interpreted Languages .....	22
Figure 16: Jupyter Notebook — Server .....	23
Figure 17: Jupyter Notebook — Select or Create a Notebook .....	24
Figure 18: Jupyter Notebook — New Python 3 .....	25
Figure 19: Jupyter Notebook — Blank Notebook .....	26
Figure 20: Jupyter Notebook - Cell Options .....	26

Figure 21: Jupyter Notebook — Hello World Page Executed .....	28
Figure 22: Jupyter Notebook - Hello World Page Renaming .....	29
Figure 23: Jupyter Notebook - Select or Create with Hello World Page .....	30
Figure 24: JupyterLab - Home Page .....	31
Figure 25: JupyterLab - Python 3 Console .....	32
Figure 26: JupyterLab - Terminal .....	33
Figure 27: Blank Python 3 Console in JupyterLab .....	38
Figure 28: Python 3 Console - weight Variable .....	39
Figure 29: Python 3 Console - Invalid Syntax .....	40
Figure 30: Python 3 Console - Value of weight .....	41
Figure 31: Categories of Potential Variable Names .....	42
Table 1: Potential Variable Names .....	42
Figure 32: Python 3 Console — player_weight Created .....	43
Figure 33: Python 3 Console — player_wieght Created .....	44
Figure 34: Python 3 Console - Assignment Operator Statements .....	46
Figure 35: Python 3 Console - More Assignment Operator Statements .....	47
Figure 36: Python - Working with Floating Point Numbers .....	48
Figure 37: Python - Working with Scientific Notation Floating Point Numbers .....	49
Figure 38: Python - Working with Complex Numbers .....	50
Figure 39: Python - Working with Octal and Hexadecimal Numbers .....	51
Figure 40: Python - Simple Arithmetic .....	52
Figure 41: Python - Simple Strings .....	53
Figure 42: Python - String Quotes .....	54

Figure 43: Python - String with Escaped Quotes .....	55
Table 2: Python—Escape Sequences .....	55
Figure 44: Python - Raw Strings .....	56
Figure 45: Python - Triple Quoted Strings .....	57
Table 3: Python—String Operations .....	57
Figure 46: Python - Concatenating Strings .....	58
Figure 47: Python - String Format Function .....	59
Figure 48: Python - String Replication .....	60
Figure 49: Python - Substring Part 1 .....	61
Figure 50: Python - Substring Part 2 .....	62
Figure 51: Python - Substring Part 3 .....	63
Figure 52: Python - String Manipulation Functions .....	64
Figure 53: Python — Sets .....	65
Table 4: Python—Set Methods .....	66
Figure 54: Python - Set Methods in Use .....	67
Figure 55: Python — Frozen Set .....	68
Table 5: Python—List Methods .....	68
Figure 56: Python - Lists .....	69
Figure 57: Python - Tuples .....	70
Figure 58: Python - Dictionaries .....	71
Figure 59: Python - Using Dictionaries .....	72
Figure 60: Python - File Open .....	73
Figure 61: Python - Writing to a File .....	74

Figure 62: Python - Open File for Reading .....	75
Figure 63: Python - Reading from a File .....	76
Table 6: Python Assignment Operators .....	81
Figure 64: Python PEMDAS .....	82
Figure 65: Python Chained Assignment .....	83
Figure 66: Python Print .....	84
Figure 67: Python Input .....	85
Figure 68: Python More Input .....	86
Figure 69: Python Type Conversion .....	87
Table 7: Python Comparison Operators .....	88
Figure 70: Python Comparison Operators in Action .....	89
Figure 71: Python Boolean Data Type .....	90
Figure 72: Python - If Statement .....	91
Figure 73: Python - If Statement Condition True .....	92
Figure 74: Python - If Statement Condition False .....	92
Figure 75: Python — Another If Statement .....	93
Figure 76: Python - Another If Statement Evaluating to True .....	93
Figure 77: Python - Another If Statement Evaluating to False .....	94
Figure 78: Python — Two If Statements .....	94
Figure 79: Python - Else .....	95
Figure 80: Python — Another If True .....	95
Figure 81: Python - If False .....	96
Figure 82: Python - Elif .....	96

Figure 83: Python - Lots of Elif Statements .....	97
Figure 84: Python --- Range with Stop Parameter Only .....	98
Figure 85: Python — Range(10) Executed .....	99
Figure 86: Python - Range with Start and Stop Parameters .....	99
Figure 87: Python - Range with Start, Stop, and Step Parameters .....	100
Figure 88: Python - Printing a List with a For_Range Loop .....	100
Figure 89: Python - Indexes for the my_players list .....	101
Figure 90: Python - Printing a List of Players .....	102
Figure 91: Python - Printing a List of Players Using a While Loop .....	102
Figure 92: Python - Break .....	103
Figure 93: Python - Continue .....	104
Figure 94: Python - Creating an Iterator .....	105
Figure 95: Python - Looping Through a List with an Iterator .....	105
Figure 96: Python — Iterator Loop Error .....	106
Figure 97: Python - Iterator with For Loop .....	106
Figure 98: Python - For Loop with List .....	107
Figure 99: Python - Creating a List of Square Numbers .....	107
Figure 100: Python - Creating a List of Square Numbers with a List Comprehension ...	108
Figure 101: Python - List Comprehension For Loop .....	108
Figure 102: Python - List Creation with Comprehension and Condition .....	109
Figure 103: Python - Simple Math .....	113
Figure 104: Python - Simple Math and More Simple Math .....	114
Figure 105: Print Function .....	115

Figure 106: One of the World's Simplest Functions .....	116
Figure 107: Using One of the World's Simplest Functions .....	116
Figure 108: Execution Flow as We Use One of the World's Simplest Functions .....	117
Figure 109: Output as We Use One of the World's Simplest Functions .....	117
Figure 110: Very Basic If Statement .....	118
Figure 111: Very Basic If Statement — Error .....	118
Figure 112: Very Basic If Statement 2 - Error .....	119
Figure 113: Scope for this_new_variable .....	120
Figure 114: Scope for Functions .....	120
Figure 115: A Very Simple Program .....	121
Figure 116: Function Variable Scope .....	122
Figure 117: Fun with Variable Scope .....	122
Figure 118: Fun with Variable Scope - Output .....	123
Figure 119: Scope of Two Instances of my_variable .....	124
Figure 120: Scope of Nested Function .....	125
Figure 121: Variable Scope within Nested Functions .....	126
Figure 122: Accessing Global Variables within Functions .....	127
Figure 123: Nonlocal Variable within Nested Functions .....	128
Figure 124: A Function with Three Arguments .....	129
Figure 125: Missing Argument Error .....	130
Figure 126: Too Many Arguments Error .....	131
Figure 127: Different Data Types in Arguments .....	131
Figure 128: Passing Arguments in Various Orders .....	132



Figure 129: Default Arguments .....	133
Figure 130: Return Values .....	134
Figure 131: Syntax Error .....	139
Figure 132: Syntax Error — Interpreter Pointing In Wrong Spot .....	140
Figure 133: Dividing by a Number .....	141
Figure 134: ZeroDivisionError .....	142
Figure 135: Opening and Reading a File .....	142
Figure 136: FileNotFoundError .....	143
Figure 137: Divide by Zero Code with Try/Except .....	144
Figure 138: Try Block Executing without Error .....	145
Figure 139: Try Block Executing with Error .....	145
Figure 140: Try Block - Reading a File Successfully .....	146
Figure 141: Try Block - Reading a File with Error .....	147
Figure 142: Finally Block without Error .....	147
Figure 143: Finally Block with Error .....	148
Figure 144: Exception Raised by Programmer .....	148
Figure 145: Logging Levels .....	150
Figure 146: Changing the Logging Level .....	151
Figure 147: Writing Log Data to a File .....	151
Figure 148: Opening and Reading a Log File .....	152
Figure 149: Formatting Log Output .....	152
Table 8: Logging Format Attributes .....	153
Figure 150: Function Not Defined: Floor .....	157

Figure 151: Importing Floor From Math .....	157
Figure 152: Redefining Floor .....	158
Figure 153: Namespaces .....	159
Table 9: Namespace Analysis .....	159
Figure 154: Import Math Library .....	160
Figure 155: Difficult to Read Code .....	160
Figure 156: Easy to Read Code .....	161
Figure 157: Simple Docstring .....	161
Figure 158: Simple Docstring - Using help .....	162
Figure 159: Multiline Docstring .....	163
Figure 160: Multiline Docstring — Using help .....	163
Figure 161: NumPy — A Few Examples .....	165
Figure 162: Matplotlib Simple Plot .....	166
Figure 163: Matplotlib Simple Pie Chart .....	167
Figure 164: SciPy Interpolation Plot .....	168
Figure 165: SciPy Normal Distribution Histogram .....	169
Figure 166: Pandas Car Data .....	170
Figure 167: Pandas Efficient Cars Data .....	170
Figure 168: Classification Algorithms .....	171
Figure 169: Regression Algorithms .....	172
Figure 170: Clustering Algorithms .....	173





**The London Institute of Banking & Finance**

8th Floor, Peninsular House  
36 Monument Street  
London  
EC3R 8LJ  
United Kingdom



**Administrative Centre Address:**

4-9 Burgate Lane  
Canterbury  
Kent  
CT1 2XJ  
United Kingdom



media@libf.ac.uk  
<https://www.libf.ac.uk/>



**Help & Contacts (FAQ)**

On myCampus you can always find answers to questions concerning your studies.