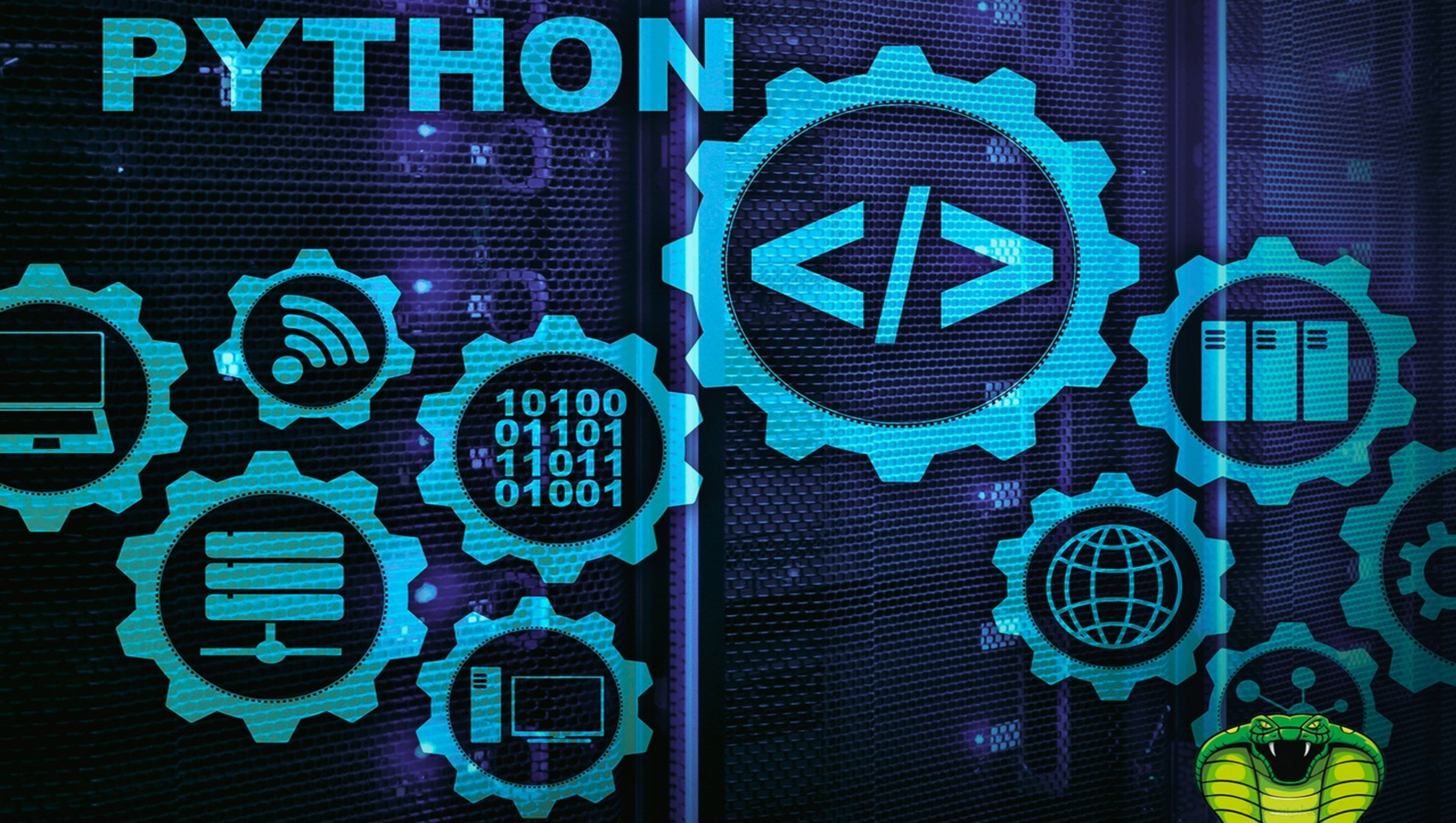
PYTHON for DATA SCIENCE

THE ULTIMATE BEGINNERS' GUIDE TO LEARNING PYTHON DATA SCIENCE STEP BY STEP

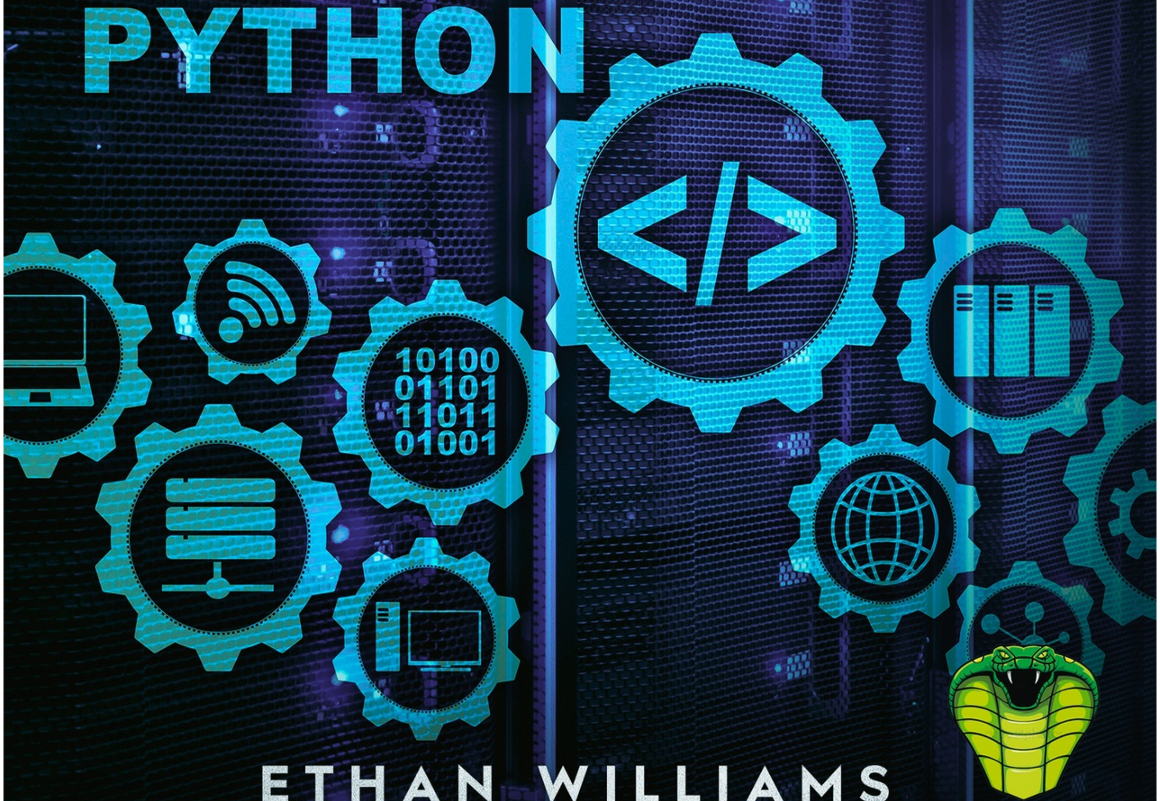


ETHAN WILLIAMS

PYTHON

for DATA SCIENCE

THE ULTIMATE BEGINNERS' GUIDE TO
LEARNING PYTHON DATA SCIENCE
STEP BY STEP



ETHAN WILLIAMS

PYTHON FOR DATA SCIENCE

The Ultimate Beginners' Guide to Learning Python Data Science Step by Step

Ethan Williams

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

TABLE OF CONTENTS

Introduction

According to a report published by LinkedIn, data science is one of the fastest growing tech fields within the past 7 years. The need for companies to have a better understanding of data generated via their business has motivated a lot of interest in the field. However, there is a gap to be breached as the supply of competent data scientists is way lower than the demand. This makes data science a very in-demand skill, with generous compensation for the few that possess the relevant portfolio. On average, a data scientist makes about $109,000/year (according to glassdoor.com); this puts data scientists in the top paid ranks of the tech industry. This tends to bring up certain questions: On a scale of ‘rocket science’ to ‘quantum physics’, how complicated is data science? Well, if you - like many other people (myself included), have wondered what data science is, and why data scientists are so in-demand, then that question is not so far-fetched. On the contrary, however, data science is not that complicated.

To risk over-simplification, data science is just the application of various techniques – usually employing the fast and efficient data organization, visualization and interpretation of computer programs or software to transform raw data into information for decision-making. This type of information is useful to managers in corporate institutions for informed risk-assessments, profit optimization, fraud detection, etc. Imagine the great prospects data science offers such businesses to consistently be ahead of the competition (assuming the competition is not equally leveraging the awesomeness of data-science). Advertisements are better targeted at consumers, companies are more aware of their economic performance and possible trends or options in production, etc. These available prospects serve as excellent motivations for considering a career in data science. However, without the right tools, guidance, and dedication, mastering the skills required for data science would be a very tedious and lengthy task. This is the reason for writing this book; to bring you up to speed on the skills and tools to begin a journey into the exciting and rewarding world of data science.

At this point, you are probably quite excited about data science and the secrets it has to offer, however, you may ask “what does Python have to do with it?”

Remember, data science leverages the exceptional processing and data manipulation capacity of computers? To do this, the data scientist must communicate the task in a clear and logical manner to the computer. The problem arises in the fact that computers do not understand human language; thus, the need for programming languages that allow powerful communications between people and computers.

Python is a high-level programming language that reduces the complexity of coding using almost human expressions in its syntax. This allows programmers to build complex algorithms and systems with exceptional performances, while reducing coding time and resources. Also, Python has an extensive library of tools and applications that makes data science less technical for beginners or intermediate programmers. Therefore, most data scientists have Python as their primary tool for big data, data analysis, etc.

Now, here is the red pill – blue pill moment. This book has been carefully designed to take you on a journey to leveraging the potentials of Python for data science. However, the extent of your progress also depends on you. Each chapter is outlined to introduce a certain concept, and helpful tips would be given to allow you avoid common Noob-mistakes (Noob is a programming jargon for beginner). Also, there would be practice exercises for you to test yourself after each chapter. This is meant to serve as a motivation for practicing the techniques or lessons that were introduced in the chapter.

It is expected that, by following every section of this book and practicing all the lessons presented, as well as practice questions included, you would not only have learned the basic techniques required for using Python for data science, but you would also have confidence to build your own practical systems and projects using Python.

The outline of this book is detailed below, and it is a guide for maximizing your use of this book depending on your level in programming. On this note, I wish you Godspeed as you journey through this book to becoming a data scientist with Python.

Outline:

Chapter 1: Python basics

This chapter introduces you to the basics of Python. The scope of this chapter spans all the necessary details for getting started with Python: downloading and installation of relevant Python packages and tools i.e. Anaconda, managing Python environments, and actual coding with Python. This is especially important for those with little or no programming experience. Even with programming experience (but new to Python), it is advisable to read this chapter. Most of the lessons from this chapter would be recalled in subsequent chapters as we dive deeper into the applications of Python for data science. However, if you already know the basics of Python and can already handle practical problems with Python, then you can skip this chapter. At the end of this chapter, a basic project would be completed using Python, and you can also try out some of the exercises that follow. These exercises would incorporate all the lessons from the chapter and doing them would greatly speed up your progress.

Chapter 2: Data analysis with Python

Here, the skills and techniques learned in chapter 1 are leveraged for starting data science. The most important (and popular) frameworks for analyzing and structuring data with Python are introduced i.e. The NumPy and Pandas frameworks. A few practical examples on using these frameworks will be considered, and practice exercises are provided.

Chapter 3: Data Visualization with Python

This chapter introduces the reader to various plotting techniques in Python. There are various plot libraries for data visualization using Python, these include: Matplotlib, Seaborn, and Pandas, etc. Data visualization is a significant part of data science, as it explains the analyzed data better using graphical representations. For example, it would be much easier to view the trend of user subscription to a service via a chart, graph, or even a pictogram, as opposed to just looking at numbers in a spreadsheet or table! This is the essence of the data visualization libraries available through Python, and we would be exploring their capabilities and specific usage in this chapter. Also, examples would be presented here, and relevant practice questions would be given. At the end of this chapter, it may be wise to review all the lessons from chapter 2, and once you are confident of your competence, to attempt solving some real-life problems using the skills gained from reading these chapters. After all, the whole purpose of data science is to solve real-world problems!

Chapter 4: Bonus chapter – Introduction to Machine learning with Python

This is an extra package added to this book. Machine learning is useful for data science, as it involves the use of powerful statistical techniques and tools for improving outcomes through the evaluation of big data. Through machine learning, the powerful capacity of computers for finding patterns and predicting outcomes via probability theory is leveraged. While this is not a comprehensive study on the subject, the reader is introduced to the underlying concepts, applications, and tools supported through Python for unleashing the power of artificial intelligence.

Chapter 1

Python Basics

Python History

This is a quick recap of the history of Python, along with some common terminologies/references you might find when interacting with other Python programmers.

Python was developed and introduced in the late 1980s by a Dutch programmer **Guido Van Rossum** . It is an object-oriented, interpreted, high-level programming language with high portability i.e. Python code/applications can be run on multiple platforms that have a version of Python installed. The name Python was also adapted from the name of a popular show "Monty Python", which Guido was watching at the time he developed the language. So far, there have been various versions of the language, with the latest being Python 3.7 starting from 2018. While there have been debates on the best version of Python to use i.e. Python version 2xx or 3xx, there are only minor differences in either version. In this book however, we would be using the latest Python 3xx version.

Installing Python

The basic way of getting Python installed on your computer is by visiting the official website for Python www.python.org and downloading the version of Python you need. Remember to download the setup that is specific to your operating system, either Windows, Mac Os or Linux. For Linux or Mac users, you may already have a version of Python installed on your computer. After downloading and installing Python through the set-up file, the installation can be verified by opening a terminal in mac or Linux os, or the command prompt in windows. This example is Windows Os based, since it does not have Python installed by default. However, the same command works on Mac and Linux.

Windows:

Open a search by pressing Windows key + Q, search for command prompt, right-click the result and select ‘run as administrator’. The Command Prompt-Admin window opens, **type: python - -version.**

If Python is properly installed, you should get a result like this: `Python 3.7. x` .

Mac/Linux Os:

Open a terminal by going to ‘Applications >> Utilities, then click on Terminal. Alternatively, you can open a search by pressing the Command key + Spacebar, then search for terminal and press enter. Once the terminal opens, **type: python - -version.**

You are going to get a result like this: `Python 2.7. x` .

This version could however be the basic Python that comes with the Os. To check if Python version 3xx is installed, **type: python3 - -version**

You are going to get a result like this: `Python 3.7. x` .

To start using Python via the command prompt or terminal, **type: python** (without quotes). You should see something like this:

Windows:

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23, 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The three arrowheads/greater-than-signs at the bottom is the trademark of the Python console. This means you are now in a Python environment and can start writing Python codes.

Let us write our first Python code using the command prompt!

At the Python console (after typing ‘python’ at the command window), enter the following code: **print (‘Hello World’)** and press enter**.** You should get a result like this:

```
print (‘Hello world’)
Hello World
```

There, we just wrote our first Python program by displaying a message using the Python print function.

Hint: *It should be noted that, while Python 3xx uses the print function as an object, i.e. print ('Hello World'), the print function is a statement in Python 2xx i.e. print 'Hello World'. This is one (among a few others) of the differences between using Python 2 and Python 3. Hence, to avoid having errors in your code examples, it is important to use Python version 3xx to follow the lessons in this book.*

Now, we have learned how to run Python via the command window. However, to write larger programs that we would prefer to finish developing before running (the command window runs every line of code immediately!), some text editors like notepad, notepad++ and various IDEs have been designed to create and save Python programs. The preferred extension for Python files is '.py' e.g. Hello_world.py, and can be run from the console by typing the full path to the Python file in your command window.

For example, if we saved our Hello_world program on the desktop. The file path could be something like this: C:\Users\yourcomputername\Desktop\Hello_World.py. To run our Python program:

Open the command window, and enter 'file path', you should see something like this:

```
Microsoft Windows [Version 10.0.17134.648]
(c) 2018 Microsoft Corporation. All rights reserved.


C:\Users\Username >C:\Users\Username\Desktop\Hello_World.py
Hello World
```

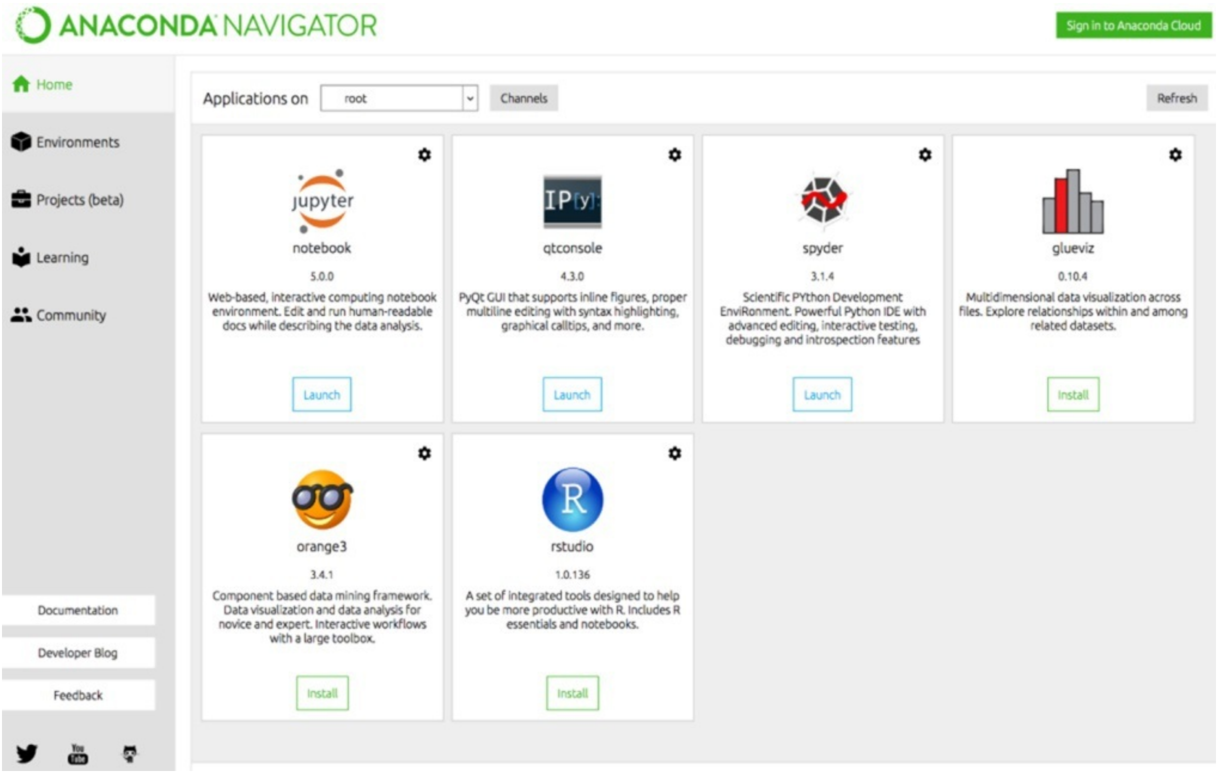
Tip: *To write Python scripts with notepad: create a notepad text file and open it >> Once opened, write your Python code in the file >> at the top left corner, click file and select save as >> select the save location of your code >> In the 'filename' box, name the file with a '.py' extension, e.g. program.py, and in the 'Save as type' drop-down list, select 'All files'. Then click save.*

You should see a Python file in your save location. You can also drag and drop this file into the command window and run it by pressing enter.

Exercise : Try writing fun programs to display text using the print function as described in the example.

The Anaconda navigator

This is a graphical user interface for running Python, and it is specially optimized for data science with Python. It is highly recommended for beginning data science with Python, as it makes the management of important data science packages and Python libraries easy. It is a part of the Anaconda distribution, which comes with a version of Python (the latest Anaconda still comes with Python 3.7 at this time, May,2019) i.e. if you do not have Python installed, anaconda will install a version of Python on your computer.



notebook
5.0.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

qtconsole
4.3.0
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

spyder
3.1.4
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

glueviz
0.10.4
Multidimensional data visualization across files. Explore relationships within and among related datasets.

orange3
3.4.1
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

rstudio
1.0.136
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Figure 1: The anaconda navigator

The most significant advantage of using anaconda is the click-to-launch features it offers for running any package installed in its path. It greatly reduces the need to install packages via command prompt using the pip install method. Also, it is easy to create virtual environments for installing and running various versions of Python, and other data science programs like ‘R’. Virtual environments are paths (or folders) specific to certain applications or packages in an operating system. So, by separating various packages into their respective virtual environment, there would be no risk of conflict in running multiple packages/programs at the same time.

Anaconda also comes with an interactive IDE (Integrated development environment) for running Python codes. There is the Spyder IDE, which uses a console and script interface; the Jupyter notebook, which runs an interactive Python console and GUI in your browser; Jupyter Lab, etc. We would be using the Jupyter notebook for our subsequent Python programming in this book.

To install anaconda navigator, go to www.anaconda.com/downloads and download the latest version of anaconda distribution. After downloading, run the setup file and follow the onscreen instructions to fully install anaconda on your computer.

Once the installation is done, the next step is to check if anaconda is properly installed.

Open a search using **Winkey + Q** , and type in anaconda. You should see the anaconda navigator, and anaconda prompt displayed. Click on the anaconda navigator.

If the anaconda navigator opens up, then it means you have successfully installed anaconda. Alternatively, you can open the anaconda prompt to check for the version of anaconda installed, and also run anaconda navigator.

At the anaconda prompt, **type: conda –version.** You should get a result like this:

```
(base) C:\Users\Username >Conda --version

conda 4.6.14
```

To check the version of Python installed with the Anaconda distribution, **type: python –version** .

```
(base) C:\Users\Oguntuase>python --version

Python 3.7.0
```

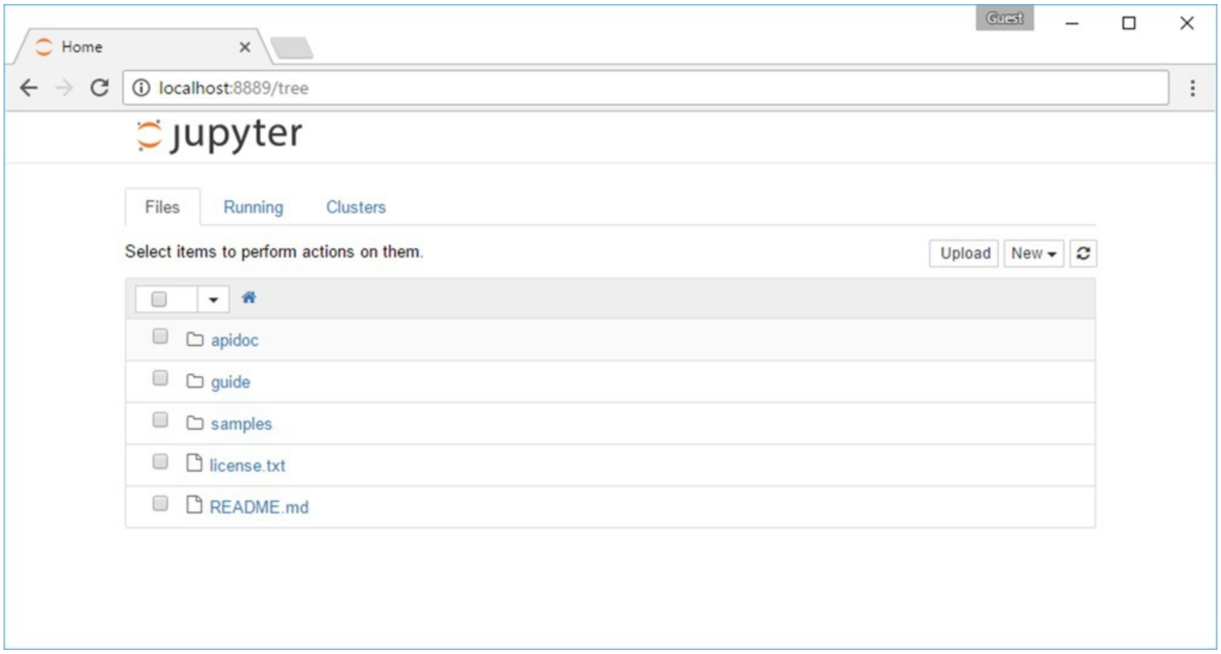
To launch the navigator, **type: anaconda-navigator**

Once the navigator opens, the installation is verified and our work is done.

Now that we have installed Python via anaconda on our computer, we are ready to start our data science journey with Python. However, as stated in the outline, we would be introducing some Python fundamentals which are considered pre-requisites for the data science lessons.

Tip: *While the idea for computer programming is mostly similar, the difference in programming languages exists in their syntax. Things like variable declaration, loops, etc. have valid formats in Python, and we would be exploring that soon. Ensure to follow the examples, and try out the practice exercises.*

Open the Jupyter notebook either by clicking **launch** via the anaconda navigator, or open anaconda prompt and **type: Jupyter notebook** . A browser window will open as shown in figure 2 below.



Select items to perform actions on them.

Figure 2: Jupyter notebook

Once on the Jupyter notebook, at the right side click **New** >> then under notebook, click **python 3/python[default]** .

This will open a Python editor where you can write and execute codes in real-time.

Tip *: Jupyter notebook files are saved as ‘. ipynb’, and can be accessed/opened by browsing to the file location via the Jupyter notebook explorer page. Jupyter also supports Markdown, and other live coding features that make it easier to annotate code. For extra information on using Jupyter, go to:* *https://jupyter.readthedocs.io/en/latest/running.html#running*

Coding with Python: The rudiments

In this section, you can either go through the exercises using Spyder IDE, Jupyter notebook, or IDLE (for those that have Python installed outside anaconda). However, it is recommended you use the Jupyter lab to get familiar with the interface, as we would be using it for the data science section.

Statements, Commands, and Expressions

High-level programming languages are not that different from human language. They follow a particular pattern and rule. Once that rule is missed, it gives an error. In language, it is called a grammatical error; in programming, it is called a syntax error.

A statement in Python is a complete set(s) of instruction that achieves a task. It can be considered as an equivalent to the sentence in the English language that contains a Subject, verb and object.

Recall your first Python program, print (‘Hello World’)? The **print ()** function used is a command. It can be considered as a verb, which does nothing on its own. The **‘Hello World ’** part is the expression, which is the stuff to be *done* . Together they make a complete instruction called a statement, which tells Python exactly what to do, and how!

There are many commands for doing various things in Python, and we would get familiar with using them as we progress. Expressions on the other hand, take on various forms. At times, it can be evaluated e.g. 4+5; other times, like the previous example, it is just some text to be displayed.

Let us try another version of the ‘Hello World’ example. How about we tell Python to display the two most dreadful words you would hope not to see on your screen while playing a game! ‘GAME OVER’

Info: *Python is a great tool for game development as well. This example is a reference to that possibility in the application of Python. However, for standard game development, you would need to learn some specific Python libraries and methods. You can find extra information about this from: https://wiki.python.org/moin/PythonGames*

To do this, we could just modify the expression in our former program.

In Jupyter notebook, **type: print (‘GAME OVER!’)** and press **Shift + Enter** to run**.** You should have a result like this:

```
In [1]:     print (‘GAME OVER!’)

Out[1]:     GAME OVER!
```

Tip: for the print function in Python, you can enclose your expressions in a single quote ‘’, double quote ““, or three quotes “’ “”. However, always make sure to use the same type of quotes to start and terminate the expression.

Comments

Consider this code that is run in Jupyter notebook:

```
In [2]:     # This program says Hello to the world
          # Author: random_Coder_Guy
          print (‘Hello, World!’)
Out[2]:     Hello, World!
```

You can easily notice that the only thing displayed in the result is the ‘Hello, World’ statement. Now let us try this:

```
In [3]:      This program says Hello to the world
           Author: random_Coder_Guy
           print (‘Hello, World!’)


    File "<ipython-input-14-239b196f5fd8>", line 1
              This program says Hello to the world
                           ^
    SyntaxError: invalid syntax
```

We get an error. Python is even kind enough to tell us our error is in the first line! The difference between these two codes is immediately obvious. The first code did not throw an error because of the pound-sign/hashtag (#) in lines 1 and 2. This is called a **comment.**

Comments are meant to illustrate code and improve its readability. When reading the first code, it is obvious what the program is intended to do. However, to prevent Python from executing those lines in our code (thereby leading to syntax error as in the second code), we use the comment sign to distinguish these texts.

A good question would be, are comments compulsory? The only answer is YES! Comments will save your life (while coding at least, it would not do so much in a Mexican standoff), and probably save others from tearing their hair out while trying to understand your code!

Have you ever solved a problem or played a level in a game and years later, you cannot remember how you did it? Now imagine that happening to a very important code you wrote; two years after, you cannot figure out what it does or how it works. That is a form of bad programming, and the lesson to be learned is 'ALWAYS COMMENT YOUR CODE!'

Using the Jupyter notebook also offers more options for improving code readability. Since it uses a web-based framework, it supports 'markdown' (which is just a way of formatting texts on webpages). To fully explore this feature and other cool stuff with Jupyter, visit: https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/

Python data types

Data types are specifications of the kind of operations a programmer expects the computer to perform on/with specific data items. It is what determines the attributes of the elements of an expression, as declared by the programmer i.e. You.

Python, like any other programming language, is also very particular about its data types. The following are the various data types available in Python which we would be using later on:

- Numbers (i.e. integers, floats, complex etc.)
- Strings
- Lists
- Tuples
- Dictionaries
- Booleans
- Print formatting

Numbers

Numbers are represented in Python as either integers, floats (floating-point numbers) or complex numbers.

Integers

Numbers that do not have decimal points are represented as integers in Python with the type **int ().** By default, Python assumes integers to be in decimal (or base 10), but the bases can be changed if there is a need to work in binary, octal or hexadecimal.

You can do arithmetic operations with integers right from the prompt (or code cell in Jupyter notebook).

# Code for arithmetic operations with Python

```
In [4]:     5 + 10
Out[4]:      15
```

```
In [5]:     20 - 12
Out[5]:      8
```

```
In [6]:     3*5
Out[6]:      15
```

```
In [7]:     20/5
Out[7]:      4.0
```

Here we can see that Python can perform basic arithmetic operations with integers. If you observe, it is clear that addition, subtraction, and multiplication of integers with another integer results in an integer. For division, however, the result is a number with a decimal point. This is a floating-point number.

Tip: *There are other arithmetic operations possible with integers. All these operations are similar to those in basic mathematics. Remember Bodmas, Pedmas or Pemdas? I like to use the acronym Bedmas for remembering the priority for Python arithmetic operations.*

B *rackets take first priority, followed by* ***E*** *xponentiation, then* ***D*** *ivision or* ***M*** *ultiplication (they have the same priority and can be interchanged), and finally* ***A*** *ddition and* ***S*** *ubtraction (which are also of same priority).*

This is illustrated in the following code:

```
In [8]:     # This code tries to determine order operations with integers
                (7-2*2)**3 / (12-3**2)
Out[8]:      9.0
```

Now, how does this code work? First, in-case you missed it, the ‘**’ is the sign for exponentiation in Python (some other languages allow the use of ‘^’).

Looking at the first bracket while keeping Bedmas in mind, it becomes obvious that the exponentiation inside the first bracket is treated, making it (7-4) which gives 3. Then 3 to the power of 3 gives 27.

For the second bracket, the same technique applies and results in 3. So, 27/3 gives 9 and Python returns a floating-point value of 9.0 due to the division.

Exercise: Try out a few other arithmetic operations with integers and see what you get. Remember to follow the order of arithmetic operations to avoid errors.

Initially, there was a cap to the number of characters allowable for a type int () data type, but it has been lifted in new releases of Python. Now you can declare an integer type with as many numeric characters as possible (of course, there is the limit of your computer memory!). Also, as indicated earlier, you can perform arithmetic operations on integers in various bases. Python converts these integers to decimal and then performs the arithmetic operation.

Example 1: Assuming we need to multiply 5_2 by 15_8 . 5 in base 2 is ‘101’, and 15 in base 8 is ‘17’

```
In [9]:    # This code multiplies integer in various bases and displays the decimal equivalent
             # to specify 101 in base 2, we use 0b101
             # to specify 17 in base 8, we use 0o17
             0b101 * 0o17
Out[9]:     75
```

Exercise: Try out some random mathematical operations with integers in various bases; hexadecimal is still unused.

Floats

These are numbers that have a decimal point in their representation. They can be considered as real numbers as defined in mathematics. Floats were introduced as a means of increasing the precision of representing numbers in computer programs.

All the numeric operations that are possible with integers can also be replicated with floats, and Python also supports float – integer operations.

Python 3 handles float operations in a straightforward manner. In Python 2, however, when doing division, at least one of the numbers (usually the divisor) must always be represented as a float to avoid truncation error in the result.

Recall that integers have unlimited representation in Python? Floats do not have the same. The maximum value a float can take is less than 1.8e308 (which is considered infinity in Python).

```
In [10]:    1.8e308

Out[10]:    inf
```

On the other end of the number line, floats have a minimum value of 1e-325 which is considered zero.

```
In [11]:    1e-325

Out[11]:    0.0
```

$$\text{floats} = \begin{cases} 0, & \text{value} \leq 1e^{-325} \\ \text{inf}, & \text{value} \geq 1.8e^{308} \end{cases}$$

Tip: *Try out all the operations you have done with integers by using floats.*

Complex numbers

If you know about complex numbers, chances are: you have some mathematics or engineering background. This shows the potential for applying Python to solving abstract engineering or mathematics problems.

Complex numbers are represented by their real and imaginary parts in Python, and all the rules of mathematical operations that apply to integers and floats are still valid with Python's complex numbers.

In Python, complex numbers are represented as {real} + {imaginary} *j* . Although, mathematicians might throw a fit, since they prefer the 'i' operator for imaginary number.

Example 2: write a program to calculate the product of 5+3i and 2+4i (engineering problems might have the form: 5+3j and 2+4j or 5+j3 and 2+j4)

```
In [12]:    # This program calculates the product of complex numbers
```

```
              (5+3j) * (2+4j)
Out[12]:   (-2+26j)
```

Strings

This is the representation of text characters in a programming language. In Python, strings are specified between quotation marks, either single, double or three quotes. Remember the Hello World program we wrote earlier on? Well it was a string!

Example 3: Let us try creating strings with the single, double and three quotes delimiters. You may notice that Python displays the same results.

```
In [13]:   'Michael Foster'
Out[13]:   'Michael Foster'


In [14]:   "David Beckham"
Out[14]:   'David Beckham'


In [15]:   """Guillermo Giovani"""
Out[15]:   'Guillermo Giovani'
```

You may wonder, why strings are equally represented by either single, double or three quotes? Why not stick to one convention? The answer is obvious in the way the English language is written. There are instances when written words require either single or double quotes (at most). The rule is to use a higher number of quotes to wrap the string.

For example, let's say I want Python to print the following statement: Hello Mark, it's nice to meet you!

Let us try the single quotes:

```
In [16]:   'Hello Mark, it's nice to meet you!'
                              File "<ipython-input-23-43365d0419fc>" , line 1
             'Hello Mark, it's nice to meet you!'
              ^
                SyntaxError: invalid syntax
```

Using double quotes, however, gives us the right answer.

```
In [17]:    "Hello Mark, it's nice to meet you!"
Out[17]:    "Hello Mark, it's nice to meet you!"
```

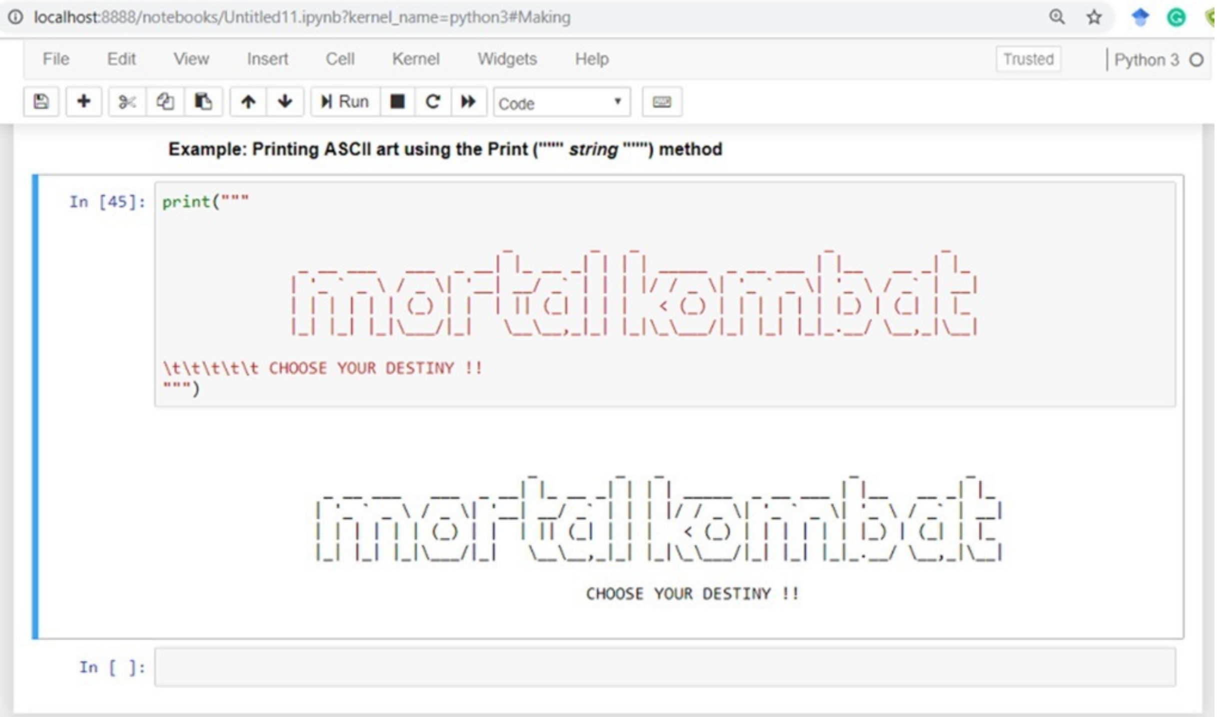
Notice that the output in this case is enclosed in double quotes as well. This is due to the escape of the apostrophe within the expression (an apostrophe is interpreted as a single quote by Python). The same process follows for double quotes in expression.

While it is possible for three quotes to be used for writing multiline strings/comments. It is not a formal way of doing so. The # is still the default comment approach in Python (line by line).

Example 4: Let us try a fun example. We would display a cool Mortal Kombat logo using the print function. This is just a modification of the 'Hello world' program we wrote earlier.

To do this example, we need to have a way to specify the logo as strings. For this, we need ASCII art, which you can design yourself, or copy example art from this repository: https://www.asciiart.eu/video-games/mortal-kombat

The code and the output art can be seen in figure 3.


localhost:8888/notebooks/Untitled11.ipynb?kernel_name=python3#Making
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 ○
Run
Code

Example: Printing ASCII art using the Print (""" *string* """) method

```
In [45]: print("""

                       _        _ _                    _           _   
 _ __ ___   ___  _ __| |_ __ _| | | _____  _ __ ___ | |__   __ _| |_ 
| '_ ` _ \ / _ \| '__| __/ _` | | |/ / _ \| '_ ` _ \| '_ \ / _` | __|
| | | | | | (_) | |  | || (_| | | |   < (_) | | | | | | |_) | (_| | |_ 
|_| |_| |_|\___/|_|   \__\__,_|_| |_|\_\___/|_| |_| |_|_.__/ \__,_|\__|

\t\t\t\t\t CHOOSE YOUR DESTINY !!
""")
```

```

                       _        _ _                    _           _   
 _ __ ___   ___  _ __| |_ __ _| | | _____  _ __ ___ | |__   __ _| |_ 
| '_ ` _ \ / _ \| '__| __/ _` | | |/ / _ \| '_ ` _ \| '_ \ / _` | __|
| | | | | | (_) | |  | || (_| | | |   < (_) | | | | | | |_) | (_| | |_ 
|_| |_| |_|\___/|_|   \__\__,_|_| |_|\_\___/|_| |_| |_|_.__/ \__,_|\__|

                                          CHOOSE YOUR DESTINY !!

In [ ]:
```

Figure 3: Illustrating graphics with the print function and strings

Exercise: Visit the Ascii art repository, and try to replicate displaying graphics using Python.

String Concatenation

While it is not feasible to do arithmetic operations with strings (strings are interpreted as text, you cannot do math with text), strings can be joined together or replicated.

Example 5: Let us assume we want to write a random person's name. We have the First and Last name.

```
In [18]:    'Fire' + ' cracker !'

Out[18]:   'Fire cracker'
```

Tip: *In this example, notice how I introduced a space before the word 'cracker'. This is to ensure the result makes sense, as otherwise Python would have strung everything together like this* ***'Firecracker'*** *.*

Example 6: Now let us try to replicate a string. Imagine you need to type the same sentence 100 times! Only if there was a way to do it just once?

In this example, we would repeat the statement 'I love Python programming, it's awesome', three times. The same method used can be adapted for a larger repetition, it only requires your creativity!

```
                    In [19]:     print("I love Python programming, it's awesome\n" * 3)

Out[19]:   I love Python programming, it's awesome

                I love Python programming, it's awesome

                I love Python programming, it's awesome
```

Tip: *The print function helps to display strings in a proper manner, without quotes. Also, it allows the use of formatting options, like the new line character \n in the code above. Try using this tip to display more elaborate strings.*

Variables

These are placeholders for data of any type i.e. integers, strings, etc. While it is very effective to imagine a variable as a storage space for temporarily keeping data until it is needed, I prefer to think of variables like money.

Wait what? Yes, Money. While it doesn't technically equate in description, it is good for visualization of what a variable does with your data.

First, I believe you are aware of the fact that money only has the value to which a consensus of people, say the world bank, assign to it! (it is more complicated than that, but that is the general idea). That means the $10 note you have today can either be more valuable or less valuable in the next year depending on what happens in the economy of the issuing country, and across the world. It could also be worth nothing if the world so decides.

Now, relating it to variables. As earlier described, they are placeholders for data. To declare a variable, say “X”, you will assign data with the value of a specific data type to it. Then, the value assigned to it becomes the value of your variable. Now, the variable carries the same value and datatype as the original data until you choose to re-assign it, or it gets re-assigned by events in your program.

Makes sense? Let us try a few examples to get a better understanding of variables in Python.

```
In [20]:   x = 5

           y = 10

           x + y

Out[20]:   15
```

As you may observe from the code, x is now of value 5 and y of value 10. If we check the data type of x or y, they would both have the same int() datatype as their assigned data.

```
In [21]:   type(x)

Out[21]:   int
```

Exercise : Try checking the type for the other variable y. Change the values of x and y and perform extra operations with them. Remember that variables can hold strings as well, just remember to put the quotes for declaring strings.

***Tip** : There are a few rules for declaring Python variables: First, Python variables can only begin with an underscore (_) or an alphabet. You cannot start a Python variable with numbers or special characters i.e. @, $, %, etc.*

Second, Python has some reserved names for functions i.e. print, sum, lambda, class, continue, if, else, etc. These names cannot be used for declaring variables to avoid conflicting Python’s operations.

Also, in variable declaration, it is advisable to use a naming convention that best *describes what your variable does. If your choice variable-name contains multiple words, separate them with an underscore like this: variable_Name, matric_Number, total_Number_Of_Cars, etc.*

Example 7: let us build a more elaborate program to illustrate string concatenation.

```
In [22]:   # This program concatenates first and last name and displays fullname.

                    First_name = ‘James’

                      Last_name = ‘Potter’

                      Fullname = First_name + ‘ ‘ + Last_name

             print (Fullname)



Out[22]:   James Potter
```

Tip: *Did you notice there was no use for quotation marks in the print function? This is because the variable ‘Fullname’ already has the quotes as its attributes. By putting strings in the print function, Python would think you wish to print the string **‘Fullname’** !*

Let us check the attribute of the variable ‘Fullname’.

```
In [23]:   type(Fullname)

Out[23]:   str
```

Again, let us build a more complicated version of the name program that accepts user input for first name, surname and age, and then displays something witty.

In this example, the function **input()** would be used, and its syntax can be deduced from the code.

```
                        In [24]: # This program displays name and says witty stuff
                                           First_name = input ('What is your first name ?\t')
                                           Last_name  = input ('Last name too ?\t')
                                           Age        = str (input ('Your age ?\t '))
                                           Fullname = First_name + ' ' + Last_name
                                           print ('Hello '+ Fullname + '\n\n')
                                           print ('It must be nice to be '+ Age + '; I am a computer, I     have no age!')
```

```
Out[24]:   What is your first name ?      David
                  Last name too ?    Maxwell
              Your age ?   25
               Hello David Maxwell
                                                  It must be nice to be 25; I am a computer, I have no age!
```

It might also be easy to observe the datatype conversion of the age variable to a string using the **str ()** function. This is also possible for conversion to other data types i.e. **int ()** for conversion to integers, **float ()** for conversion to floating-point numbers, and **complex ()** for changing to complex numbers.

Exercises

1. Create a simple program that takes the name of three friends, and welcomes them to a restaurant e.g. McDonalds.
2. Write a program that accepts an integer value and displays it as a floating-point number.
3. Write a simple program that accepts a temperature value in Celsius, and displays its Fahrenheit equivalent.

Hint: Conversion from Celsius to Fahrenheit is given by:

$$\frac{(Temperature\ {}^{\circ}\boldsymbol{C}\ \times\ 9)}{5} + 32 = Temperature\ {}^{\circ}\boldsymbol{F}$$

Lists

Python lists are not so different from the general idea of lists in regular language. It is just a collection of values in a sequence separated by commas. Lists can hold values with different data types i.e. a Python list can be a combined sequence of integers, floats, strings etc. Lists can be assigned to variables at their declaration, and a few list operations are illustrated in the following examples:

```
# List examples
```

```
[1,2,3,4,6]                  # List of integers
[12.1,13.42,15.6,18.93,20.0]  # List of Floats
['New','Taken','Extra']       # List of strings
['True','False']              # List of Boolean expressions
['Derek',25,125.50,True,]     # List of different data
 types
```

To declare a Python list, the square brackets **[ ]** are used, and values are separated by commas.

All the lists declared above show how Python lists are flexible for holding data of any type. However, these lists are just declared to be evaluated once at the command window. It is a better practice to assign lists to variables for re-use.

Tip: *To run the above example, copy and run each list individually, if you copy all these lists above into a Jupyter cell, Python will return only the result of the last declared list.*

List indexing

Once you have declared a list, each value in the list can be accessed via a method called indexing. To understand this concept, imagine a list is a library with various types of books. Librarians usually organize books by category/type, and as such, you wouldn't expect to find a science book in the arts section. Also, in each category, the librarian can ease access by allocating a serial number to each book. For example, the book on 'World geography' could have the identifier, 'Science A2' which belongs only to that book in that specific library. In another library, that same identifier can refer to 'Space science' or something else. The point is, each element in a specific list has an identifier for accessing it, and list indexing is the process of retrieving elements of a list using their identifiers.

To use list indexing, the list must first be assigned to a variable. It is also worth noting that Python list indexing is direction dependent. For indexing from left to right, Python starts indexing at '0', while right to left indexing starts at -1.

Example 8: Let us grab the third element from a list.

```
In []:      Random_List = [1,2,3,4,'food']
```

```
                   Random_List [2]
Out[]:      3
```

Alternatively, we could have grabbed the value 3, from **Random_List** by using the command **Random_List [-3]** ; however, it is not a common convention.

There are also other versions of lists that contain other lists. These are called nested lists.

Example 9 : This is a Nested list

```
                    Random_List2 = [[1,2,3,4,'integers'],'food', [12.3,5.2,10.0,'floats']]
```

To understand nested lists, imagine Russian nesting dolls. The name is derived from the fact that the first doll layer contains the other dolls. If the layer is removed, the next layer then houses the rest of the dolls. This goes on until all the layers of dolls are removed to reveal the desired layer of the doll. What this analogy portrays is that the same indexing approach works for lists that are nested within other lists.

Example 10: Let us grab the value 10.0 from Random_List2.

```
                 In []:  Random_List2 = [[1,2,3,4,'integers'],'food',[12.3,5.2,10.0,'floats']]

                         Random_List2 [2][2]

Out[]: 10.0
```

Here is what happened. By using the first command, ***Random_List2 [2]*** *, we were able to grab the nested list containing the desired value 10.0. If you count, that list is the third element in* ***Random_List2*** *. Now, after using the first indexing to grab that list, we can now index that new list as well. Within this new list* ***'Random_List2[2]'*** *the value 10.0 is the 3 rd element, or at index 2 (remember that Python indexes from zero, we could also select it with* ***'-2'*** *), so to grab it, we just index that list at 2:* ***'Random_List2[2][2]'*** *. Easy enough? You can try grabbing other elements of the list using this idea.*

Example 11: Let us create a program that accepts user information and displays some results.

```
                  In []:      # Program that accepts user data

                                  user_Data = eval (input ("'Please enter your information in the following
                    order:
```

```
                    ['Name', Age, Height (in metres), Married (enter True/False)]


           '''))
           Please enter your information in the following order:


           ['Name', Age, Height (in metres), Married (enter True/False)]


           ['James Franco', 41, 1.8, False]
```

Output

Te ‘eval’ method in the code above is used to evaluate any input suggested by the user. This allows Python to accept the input in literal string form and interpret it as Python code. In this instance, Python recognizes the input as a list because of the [ ]. It also assigns the appropriate datatype to each element as specified in the list.

Now that the user_Data list has been prepared, we can now assign each element in the list to their corresponding variables and print our desired output.

```
           In []:       Name             = user_Data[0]
                   Age          = user_Data[1]
                  Height        = user_Data[2]
                  M_Status       = user_Data[3]
                  print(('''Here are your details


                     Name: {}
                    Age: {}
                    Height: {}m
                    Married: {}''').format(Name,Age,Height,M_Status))
```

Output

```
Here are your details


Name: James Franco
Age: 41
Height: 1.8m
```

```
Married: False
```

Now, that was fun right? However, you may notice that a new method was introduced in this section, the **‘. format’** print method. Here is how the code works: From the user_Data list we have declared in the first cell, we grabbed each element and assigned them to variables ‘Name, Age, Height and M_Status’. To display all these elements without the need for concatenating strings or changing data types, we can use the print function with triple quotes to print across multiple lines. Alternatively, we could have used single or double quotes along with newline/tab escape options i.e. \n or \t.

The **‘. format’** method is a way of controlling print. You just need to put { } in the places your values would go, and then specify the order.

Assuming we want to print something like: I am 5 years old, and I have $20.

```
In []:     print (('I am {} years old, and I have {}.').format(5,'$20'))
```

Output

```
I am 5 years old, and I have $20.
```

***Hint** : In the code above, the $20 had to be specified as a string. A better approach would be to use variable assignment i.e. Assigning 5 and $20 to variables and passing them into the **‘.format’** method.*

Exercise: Write a program that displays a customer’s bank account history. This is just a variation of the example program. If your code does not work, review the examples, hints and comments to get a new insight.

Indexing a range (List slicing)

Assuming we need to grab a range of elements from a list, say the first three elements, this can be achieved via range indexing or slicing. The range operator is the colon ‘:’, and the technique has the following syntax: List name [range start : range end +1].

Example 12: Grab the names of domestic animals from the following list [‘Cat’, ‘Dog’, ‘Goat’, ’Jaguar’, ’Lion’] and assign it to a variable **domestic_animals** .

```
In []:     Animals = ['Cat', 'Dog', 'Goat', 'Jaguar', 'Lion']
```

```
            domestic_Animals = Animals [0:3]
            print(domestic_Animals)
```

Output

```
['Cat', 'Dog', 'Goat']
```

Alternatively, if our desired data starts from the first element, there is no need for specifying the index ‘**0** ’ (or index ‘**-1** ’ in right to left indexing), we could have also grabbed the data with **‘domestic_Animals = Animals [: 3]’** .

You may observe that the element ***‘Goat’*** *is at the index ‘2’, however, the range syntax requires the* ***‘range end’*** *argument to be specified as* ***‘range end +1’*** *which is 3 in this case.*

The same approach is used for doing more complicated data grabbing as seen in the next example.

Example 13: Write a program that selects three different treats from a list of food and prints a message.

```
               In []: food_list = ['rice', 'salad’, ['cake’, ‘ice-cream', 'cookies', 'doughnuts'],'Beans']
                            treats = food_list [2] [1:4]
                    print ('I love {}, {}, and {}.’.format(treats[0],treats[1],treats[2]))
```

Output

```
I love ice-cream, cookies, and doughnuts.
```

See how we combined the range indexing with the **.format** method? Try out some fun examples for yourself and consider your proficiency in using these techniques.

String Indexing

Strings are just a bunch of characters ‘*strung’* together. Hence, all list indexing operations are also applicable to strings.

Example 14 : Grab the word ‘except’ from a string.

```
In []:        # This program illustrates string indexing


String = 'Exceptional'
# index_slice
# index -slice
new_string = String[:6]


# Printing the output
print(new_string)
```

Output

```
Except
```

Example 15: Grab the word ‘oats’ from any element in a list.

```
In []:     # grabbing the word 'medical' from a list
              word_list = ['nautical',['medieval', 'extravaganza'],'extra' ]
              word = word_list[1][0][:4] + word_list[0][5:8]
              print(word)
```

Output

```
medical
```

Tip *: keep trying out variations of list and string indexing, they will be very useful in subsequent chapters.*

Now that we have learned how to index lists and strings, let us consider the case where we need to assign new elements to a list.

The first case is element re-assignment. Lists are mutable (strings are not mutable), which means they can allow you to change their initial declaration. You can re-assign list elements using indexing, and you can also add or remove from a list using the **.append** , **.insert** , **.extend** , **.remove** methods.

Example 16: Here is a list of fruits we want to manipulate [‘Apple’, ‘Orange’, ‘Banana’, ‘Cashew’, ‘Almond’].

```
In []: # Declaring the list of the fruits
       fruits = ['Apple', 'Orange', 'Banana', 'Cashew', 'Almond']
       fruits
Out[]: ['Apple', 'Orange', 'Banana', 'Cashew', 'Almond']
```

Now, let us re-assign the second element in the list i.e. ‘Orange’ and change it to

‘Guava’

```
In []: fruits[1] = 'Guava'
fruits
Out[]: ['Apple', 'Guava', 'Banana', 'Cashew', 'Almond']
```

Notice that guava has been indexed as the second element in the list. However, Orange has been removed/replaced.

Hint: *This method uses the idea of variable re-assignment. The element ’Orange’ is a variable in the fruits list and its name is based on its index i.e. fruits[1], just as the variable name for ‘Banana’ would be fruits[2]. So, it's just a matter of re-assigning a value to the variable like we learned in the previous section.*

The same method can be used to re-assign any elements in that list. However, what if we have a nested list?

Example 17: Nested list re-assignments also follow the same idea. First grab the nested element to be re-assigned, and then assign a new value to it.

```
In []: # To re-assign a nested element.
         New_list = ['Apple', ['Orange','Guava'], 'Banana', 'Cashew', 'Almond']
          New_list
Out[]: ['Apple', ['Orange', 'Guava'], 'Banana', 'Cashew', 'Almond']
In []: New_list[1][0] = 'Mango'
    New_list
Out[]: ['Apple', ['Mango', 'Guava'], 'Banana', 'Cashew', 'Almond']
```

Exercise: Now, try changing the element ‘target’ to ‘targeting’ in this list:

```
nest = [1,2,3, [4,5, ['target']]]
```

Using our initial list of fruits, let us use the .append method. This method takes the value passed to it and adds it to the end of the list. It can only add one element to a list at a time. If more than one element is passed into the method, it adds it as a sub-list to the intended list.

Example 18: Add the fruit Pawpaw to the list of fruits and make it the last element.

```
In []: # Declaring the list of the fruits
      fruits = ['Apple', 'Orange', 'Banana', 'Cashew', 'Almond']
             fruits.append ('Pawpaw')
             fruits
Out[]: ['Apple', 'Orange', 'Banana', 'Cashew', 'Almond', 'Pawpaw']
```

To add a sub-list, use the method **.append (‘[sub-list elements]’)**

Example 19: Let us add a sub-list of carbohydrates to this food list

```
In []: #Declaring the food list
      foods = ['Beans','plantain','fish']
            foods.append (['Rice, 'Wheat'])
            foods
Out[]: ['Beans', 'plantain', 'fish', ['Rice', 'Wheat']]
```

Now, let us use another method **‘.extend’** . This method also adds its input values to the end of a list. However, unlike the .append method, in the case of multiple input values, the .extend method adds each element to the list, thereby extending the list index.

Example 20: Use the .extend method to add two elements to the end of a list.

```
In []: # Declaring the food list
       foods = ['Beans','plantain','fish']
       foods.extend(['Rice', 'Wheat'])
  foods
Out[]: ['Beans', 'plantain', 'fish', 'Rice', 'Wheat']
```

Tip: *The .append and .extend methods both accept just one input argument. If it is more than one, enclose the inputs in a square bracket. This is seen in examples 19 and 20.*

Good, you have learned how to add elements to the end of a list using the .append and .extend methods. Imagine a case where you want to add an element to any part of a list without replacing the element at that index. The .insert method is appropriate for this. It also accepts one argument, and inserts that value at the specified index.

The .insert method has the following syntax: **ListName.insert (Desired index, New value)**

The **ListName** is the name of the list to be manipulated. **Desired index** is the expected position of that element (remember indexing starts from 0, or -1 depending on indexing preference), and **New value** is the element that is to be inserted at the specified index.

Example 21: Consider the following list of integers, add the right value to make the sequence correct. Count_to_ten = [1,2,4,5,6,7,8,10].

```
In []: # This program illustrates the .insert method.

        Count_to_ten = [1,2,4,5,6,7,8,10]

        Count_to_ten.insert(2,3)

        Count_to_ten.insert(8,9)

        Count_to_ten

Out[]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Let us review how that worked. The first use of the .insert in example 21 assigns the value 3 to index 2 as specified by the syntax. Now the list 'Count_to_ten' has increased in size and indexing by a value of 1 i.e. **Count_to_ten = [1,2,3,4,5,6,7,8,10]** . Now we can add the value 9 to the proper index of 8 as shown in the 4th line of code.

Tip: *Be aware of the index expansion before assigning elements. In the previous example, if we assigned the value 9 first, the indexing of the other value 3 would not have been affected by the increase in the list size. Then the following code would have achieved the same result:*

```
In []:   Count_to_ten = [1,2,4,5,6,7,8,10]

        Count_to_ten.insert(7,9)

        Count_to_ten.insert(2,3)

        Count_to_ten

Out[]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This is because the insertion of a value in the list shifts the list elements from that index of insertion to the right. Each of the shifted elements now has an index incremented by the number of elements inserted i.e. if an element had an index of 3 in a list, and two items were inserted at its index or before its index, that element now has an index of 5.

The last method we would consider on list operations is the .remove method. This is used for removing an element from a list. It also has a similar syntax with the .insert method and uses list indexing.

Example 22: Let us add the value 11 to the list Count_to_ten and then clear it using the .remove method.

```
In []: # Using the output list from the last example

        Count_to_ten.append(11)

Out[]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11]


In []: # Resetting back to 10

        Count_to_ten.remove(11)

Out[]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

There is something to notice here, the .remove method does not require the specification of the index of the unwanted element. Only the name or value of the unwanted element is needed.

Exercise: Create a list of cars, and practice various manipulations on it. Practice adding to the list, removing elements etc.

Hint: *let your exercise require all the list manipulations we have practiced so far. Get creative!*

Recall how lists are similar to strings in some operations? Well, you can also do list addition and multiplication (repetition) as we did with strings.

Example 23: Here are a few list additions and multiplications

```
In []: # Addition and multiplication of lists
        List1 = [‘A’, ’B’, ’C’, ’D’]
        List2 = [1, 2, 3, 4]
        CombinedList = List1 + List2
```

```
                   CombinedList
        Out[]: [‘A’, ’B’, ’C’, ’D’, 1, 2, 3, 4]
```

```
       In []: # Multiplication of lists
                  List2*2
       Out[]: [1, 2, 3, 4, 1, 2, 3, 4]
```

While there are more operations possible with lists, we have covered enough to serve as a good foundation for further applications with lists.

Tuples

These are similar to lists i.e. a sequence of comma separated values. However, unlike lists, tuples are immutable and they are defined using a parenthesis () . Data can also be grabbed from tuples using the indexing technique. The usefulness of tuples arises from the need to create program parts that cannot be changed by the user/program during operation. Imagine creating a bank account program. Things like your ATM pin could be open for user modification. However, imagine letting people have access to change their account details! This could cause a conflict in the banking program. These are the kinds of data that tuples can hold. Users can retrieve such data; however, they cannot be changed.

Example 24: Let us declare some tuples.

```
         In []: # Tuples can be declared in two ways
                    my_Tuple = (1,2,3,4,5)      # way 1
                     print(my_Tuple)
                     type(my_Tuple)


         Out[]:      (1,2,3,4,5)
                      tuple
```

```
         In []:  my_Tuple2 = 1,2,3,4,5,     # way 2
                     print(my_Tuple2)
                     type(my_Tuple2)
         Out[]:     (1,2,3,4,5)
                      tuple
```

We can consider the two types of tuple declarations shown in the codes above. While the first method is the traditional and recommended method, the other method is equally acceptable. The second method is called ‘tuple packing’, and we would discuss its relevance later.

Any sequence of comma-separated items declared without an operator or special characters like [ ], { }, etc. automatically gets assigned as a tuple. This is important.

As explained earlier, all the regular indexing operations involving data grabbing can be done with tuples, however, it doesn’t support element assignment/re-assignment once declared.

Another reason why a Python programmer might prefer to use a tuple over a list is because tuple operations are faster than with lists as tuples require less memory allocation.

Example 25:

```
In []: # Comparing Lists with tuples

            my_List = ['Men', 'Index', 1,2,3]

            my_tuple = ('Men', 'Index', 1,2,3)



            a = my_tuple.__sizeof__() # get tuple size

            b = my_List.__sizeof__()  # get list size



            print('size of list is {}, and size of tuple is {}.'.format(b,a))



Out[]: size of list is 80, and size of tuple is 64.
```

Even though the elements in a tuple and list are exactly the same, they have different sizes (in terms of memory). This is especially useful for designing systems that have limited resources with high-efficiency requirements.

Example 26: Let us grab some elements from a tuple using indexing

```
In []: # grabbing tuple elements
          A = my_tuple[0]
          B = my_tuple[1]
          print ('The {} of {} is 0.'.format(B,A))

Out[]: The Index of Men is 0.
```

Exercise: Try out other indexing operations with tuples. Create a tuple and try to re-assign one of the elements using indexing. What did you observe and why?

Tuples also allow an operation called ‘packing and unpacking’. Either operation is dependent on the side of the equality operator the tuple is.

Example 27: Let us illustrate tuple packing and unpacking.

```
In []: # tuple packing and unpacking
          Tuple = 'Daniel', 'Dean', 'James'   # Tuple packing
          [Name1, Name2, Name3] = Tuple       # tuple unpacking
          print(Name1)
          print(Name2)
          print(Name3)
Out[]: Daniel
          Dean
          James
```

Tuple packing and unpacking are kind of like a convenient way of using tuples for assignment/creating variables. Packing a tuple is just a matter of declaring the tuple, to unpack however, you just declare a number of choice variables equal to the number of elements in the tuple. Each variable then takes the corresponding value in the tuple according to the order of arrangement. As can be observed, the Name1 variable corresponds to the ‘Daniel’ tuple element, and so on.

Hint: *The square brackets around the declared variables is unnecessary in tuple unpacking. It is just standard practice to use, however, using () or nothing will give the same results. Try it!*

A common use for tuples is to hold value pairs. This is especially useful for collecting data like user information for which each element stores a specific user data.

Example 28:

```
In []: # collecting data with tuples
            User_Info = (('Name','James'),( 'Age',22))                    #Nested tuple
            A,B = User_Info[0]    # Inner Tuple unpacking
            C,D = User_Info[1]    # Inner Tuple unpacking
            print (A, ': ',B)
            print (C, ': ',D)

Out[]:  Name :  James
            Age :  22
```

Dictionaries

These are also similar to lists and tuples in certain ways. Unlike lists and tuples, dictionaries are declared as key-value pairs and by using the curly braces i.e.{ } . Think of dictionaries like safety deposit boxes. You can keep any item you want in it (except your car of course, size does matter), but you can only access it with your key!

In Dictionaries indexing is done with the key associated with the value stored. Dictionaries are mutable in their keys and values (not-so-safe-deposit-box), however, they can be secured by using tuples as their keys or key-value pairs (depending on access level desired).

Example 29: Let us declare a dictionary.

```
In []:  # Dictionary declaration

           my_Dict = {'Key1': 'Movies', 'Key2': ['Iron Man', 'Avengers']}

           print(my_Dict)

           print(type(my_Dict))


Out[]:  {'Key1': 'Movies', 'Key2': ['Iron Man', 'Avengers']}

           <class 'dict'>
```

There, we just declared a dictionary. Notice that the first value is a string, and the second value is a list. The keys can be other datatypes as well, and not necessarily strings; however, string key names are usually preferred for keeping values for easy call and code readability.

To grab elements in a dictionary, the keys holding the value is first called.

Example 30: Grabbing dictionary elements

```
In []: A = my_Dict['Key1']

       B = my_Dict['Key2']

       print (A,': ',B)

Out[]: Movies :  ['Iron Man', 'Avengers']
```

We can see that by calling the dictionary keys, we passed the values held by Key 1 and key 2 to the variables A and B. More complicated dictionary key indexing can be done.

Example 31: Grabbing elements from a nested dictionary.

First, we declare our dictionary:

```
            In []: Acct_Dict = {'Name':'Customer1','Account type'

                            :{'type1':'Savings','type2': 'Current'}}

                 Acct_Dict


            Out[]: {'Name': 'Customer1', 'Account type': {'type1': 'Savings', 'type2': 'Current'}}
```

If we need to print that the user has a savings type of bank account, we would need to grab the Second key 'Account type' first, then grab the first key of that resultant dictionary. Then we can easily pass that into the print statement.

```
             In []: print('You have a '+ Acct_Dict['Account type']['type1']+ ' account')


Out[]:  You have a Savings account
```

Notice how the word 'Savings' has been grabbed using dictionary key-value indexing. Since the name of the keys for a dictionary is important to getting its values, it is important to determine the keys it holds at any point in time. To view a list of the keys a dictionary holds, use the .keys() method.

```
             In []: Acct_Dict.keys()    # Checking keys held by Acct_Dict
```

```
Out[]:  dict_keys(['Name', 'Account type'])
```

```
In []: Acct_Dict['Account type'].keys()    # Checking the inner keys

Out[]:  dict_keys(['type1', 'type2'])
```

***Tip:** There are other methods that can be used with a dictionary, even though this is all we would need for our lessons here. However, to use these methods with a dictionary or to find the methods available for any other datatypes, use the dir() method.*

```
In []: # Checking possible methods with datatypes

            # Let us declare some datatypes first and check their methods

            String = 'Bob'; integer = 100; Float = 25.3

            List = ['Man']; Tuple = 5,   # To declare a single tuple
                                         # put a comma   after the single value

            Dictionary = dict([('Name','Max')])          # Dictionaries can be declared
      # using dict() as well.

            A = dir(String); B = dir(integer); C = dir(Float)

            D = dir(List); E = dir(Tuple) ; F = dir(Dictionary)

            print (''' Here are the methods possible with each type

            Strings
            {}

            Integers

            {}

            Floats

            {}

            Lists

            {}

            Tuples

            {}

            Dictionaries

            {}
            '''.format(A,B,C,D,E,F))
```

Try copying and running the code above, or write your own variation. However, the point is for you to find out all the possible methods available to any data type in Python. This information gives you access to do advanced manipulations. Also notice the comments in the code, some new methods of declaring ‘Tuples’ and ‘Dictionaries’ were introduced. These are extra; just tricks you may wish to use. Same with the use of semi-colon to allow for declaring multiple variables or commands in one line.

Booleans

These are conditional datatypes that are used to determine the state of a statement or block of code. Booleans have two possible values ‘True’ or ‘False’ which have corresponding integer values of 1 and 0 respectively.

```
In []:    A = True   # Boolean values seem like strings but do not enclosed in quotations
              type(A)
Out[]:   bool
```

```
In []:        int(A)   # Integer value of the Boolean
Out[]:         1
```

As can be seen, the corresponding integer value for Boolean ‘True’ is 1. Try the above code for the Boolean ‘False’.

For most operations, Booleans are usually output values used for specifying conditions in a loop statement or checking for the existence of an element or condition. This leads us to the use of comparison and logical operators.

Comparison operators

These operators, as their name implies, are used to check for the validity or otherwise of a comparison. The following are comparison operators:

Table 1: Comparison operators

<	*Less than*
>	*Greater than*
==	*Equal to*
<=	*Less than or equal to*
>=	*Greater than or equal to*
!	*Not operator (can be combined with any of the above)*
	This checks for the existence of an element/value

in

Example 32: Let us check for conditions using the comparison operators

```
In []: #This code illustrates the Boolean comparison output
            print (5 < 10); print (3>4); print('Bob'=='Mary');
            print (True == 1); print (False == 0); print (True != 1)
Out[]: True
            False
            False
            True
            True
            False

In []: #The ‘in’ operator
            print ('Max' in 'Max Payne'); print (2 in [1,3,4,5]); print (True in [1,0])
Out[]: True
            False
            True
```

Logical operators

These are the Python equivalent of logic gates. This is a basic technique for performing logical operations and can also be combined using the interpreted versions of De Morgan’s law.

In case you were wondering, De Morgan is a British guy that found a way to simplify Boolean logic using rules earlier invented by another British guy, George Boole (Boolean is named after this guy).

Table2: Logical operators

and/&	*Evaluates to true if, and only if, both operands are true, false otherwise.*
or/ |	*Evaluates to true if, at least, one of the operands is true, false otherwise.*
	This is an inversion of the value/operation of its operand.

not/ !

Example 33: Let us try out some logical operations (truth table) with the ‘and’ operator .

```
In []: #The ‘and’ operator
          print (True & True); print (True and False);print (False and False)
          print (False and True); print('foo' in 'foobar' and 1<2)
Out[]: True
          False
          False
          False
          True
```

Example 34: Let us try out some logical operations (truth table) with the ‘or’ operator.

```
In []: #The ‘or’ operator
          print (True | True); print (True or False); print (False or False)
          print (False or True); print (('foo' in 'foobar') | 1<2)

Out[]: True
          True
          False
          True
          True
```

For the last operation in the above code, notice how the string operation is enclosed in parenthesis before the ‘logical or’ is used. This is because Python throws an error in the case of comparing string operations directly with other types using the logical or.

Exercises:

1. Here are two food lists: [‘Beans’,’Wheat’,’Bread’] and

[‘Rice’,’Plantain’,’Pizza’,’Spaghetti’]

   a. Write a Python code to check if the word ‘Rice’ exists in both lists.
   b. Write a Python code to check if ‘Pizza’ exits in at least one of the lists.

2. Create a Python dictionary with two keys. Key 1 should be immutable, while key 2 can change. Alternatively, the values held by key 1 and 2 are a sequence of integers; however, the values of key 1 can be edited, while that of key 2 should be immutable. ***Hint:*** *Recall, immutable data types are strings and tuples.*

Now that we have gone through all the data types, let us move on to loops and conditionals.

Conditional statements and Loops

These are a little different from what we have been doing so far. Our previous examples can be described as ‘sequential program execution’, in which expressions are evaluated line by line without any form of control. Conditional statements, however, are used to take control of how and when lines of code are executed. Loops on the other hand, are used to repeat the execution of a specific code or blocks of code. These two different control algorithms are mostly used together to develop programs of varying complexity, although, they can be used independently of each other.

Let us consider the most basic and often used conditional statement – the ‘**IF** ’ statement.

This has a syntax of the form:

```
if  expression :
        statement
```

The expression in this case is usually a Boolean operation, while the statement is a line/block of code to be executed once the Boolean evaluates to either true or false (or is not true or false).

Example 35: Let us write a program that grants a user access when any of three correct passwords is entered.

```
In []: # Grant access if user password is correct
             password_pool = ('Smith_crete','Alex@456','CEO4life')
             user_password = input ('Please enter your password: \t')

             # Now the IF condition
             if user_password in password_pool:
             print('\n Access granted!')

Out[]: Please enter your password: CEO4life


           Access granted!
```

That was a fun program, right? The idea is for Python to check if the password entered by the user is available in the earlier declared password pool (notice that the password pool is a tuple? This is a practical way for creating fixed/secure passwords)

Exercise: Write a program to calculate how much a person owes for keeping a movie past rent-due-date. Let there be an increase in price given the person keeps the movie past 3 days.

Example 36:

```
In []: # Movie rent-overdue price


             price1 = 5      # $ 5 for every day past due within first 3 days
             price2 = 7      # $ 7 for every day past due after first 3 days
             days_past_due = eval(input('How many days past due:\t'))

             # If statement


             if days_past_due <= 3:
                print('\nYou owe $',(days_past_due*price1))
             if days_past_due > 3:
                print('\nYou owe $',(3*price1 + (days_past_due - 3)*price2))
```

```
Out[]: How many days past due:     4


           You owe $ 22
```

See how we can combine multiple if statements to write even more complex code? However, there is a better syntax for evaluating multiple conditional IF statements; the **IF-ELSE** statements.

It has a syntax like this:

```
 if  expression :
       statement
     else:
   alternative statement
```

Example 37: Improved version of example 35 using *IF-ELSE*

```
In []: # Grant access if user password is correct or raise alarm
           password_pool = ('Smith_crete','Alex@456','CEO4life')
           user_password = input ('Please enter your password: \t')

           # Now the IF-ELSE conditions
           if user_password in password_pool:
                print('\n Access granted!')
           else:
                print('\n Access Denied! Calling Security …')

Out[]: Please enter your password:      wrong password


           Access Denied! Calling Security …
```

Now, this seems like a more reasonable security system, right? Still, the IF conditional gets better with the IF-ELIF-ELSE syntax. This allows you to specify actions in multiple situations.

Syntax:

```
 if  expression :
       statement
```

```
elif  expression :
      statement
elif  expression :
      statement
   else:
  alternative/default statement
```

The syntax above can be explained in regular language as “If the first condition is met, then execute the action within the first statement, or else, if the first condition is not met, do the action under the second statement.” This goes on until the else statement that defaults to the inability of the program to fulfill any of the conditions within the IF and ELIF statements.

Example 38: An even better version of example 35.

```
In []: # Grant access and greets user if the user password is correct.
         # Otherwise, raise an alarm.
         password_pool = ('Smith_crete','Alex@456','CEO4life')
         user_password = input ('Please enter your password: \t')

         # Now the IF-ELIF-ELSE conditions
         if user_password in password_pool and user_password == 'Smith_crete':
         print('\nAccess granted! Welcome Dr.Smith')

         elif user_password in password_pool and user_password == 'Alex@456':
         print('\nAccess granted! Welcome Mr. Alexander')

         elif user_password in password_pool and user_password == 'CEO4life':
         print('\nAccess granted! Welcome Mr. CEO')

         else: print('\nAccess Denied! Calling Security …')

Out[]: Please enter your password: wrong password


         Access Denied! Calling Security …
```

```
Out[]: Please enter your password: Alex@456

            Access granted! Welcome Mr. Alexander

Out[]: Please enter your password: Smith_crete

            Access granted! Welcome Dr. Smith

Out[]: Please enter your password: CEO4life

     Access granted! Welcome Mr. CEO
```

Exercise: Now that you have learned the IF conditional statement and its variations, try out some more creative and complicated examples on your own. For example, you can write a program that checks a database, prints students math exam scores and comments on whether they passed the exam or not.

Hint: *The database in your example could be a variation of the 'password pool in example 37'*

Let us now look at loops. We will consider two loops: The **While loop** and the **For loop** . As described earlier, a loop will run indefinitely until a specified condition is met. Due to this, they can be used for automating and performing really powerful operations spanning a large range; however, they must be used with caution to avoid infinite loops!

WHILE loop

This loop is used to execute a set of statements or code as long as a specified condition is true. The conditional statement that controls the algorithm is called a flag and is always true for all non-zero values. When the flag becomes zero, the while loop then passes on to the next line of code following it. There could be a single statement or multiple within the while loop, and Python supports the else statement for the while loop.

A while loop has the following syntax:

```
while flag :
   statement
```

Example 39: Writing a while loop to print a statement 5 times.

```
In []: # while loop to print output 5 times


            i = 1            # counter


            while i < 6:       # flag
            print (i,': I love Python')
                    i = i+1      # This Increments the value of ‘i’
 else:
              print ('\nThe program has completed')


Out[]: 1 : I love Python
            2 : I love Python
            3 : I love Python
            4 : I love Python
            5 : I love Python


            The program has completed
```

Okay, the program worked. But how?

The first thing is the flag. The variable ‘**i** ’ was declared with a value of 1. For the first iteration, Python checks if 1 is less than 6: the flag is true since 1 is less than 6, and the print statement runs (along with the counter we added to print the status of the loop). This continues until the loop is complete (when i=6), then the else statement runs (since the flag is now false). Assuming the else statement was absent, the while statement ends (or in the case of a larger program, it passes on to the next line of code).

Hint: *Notice the increment we included after the print statement? This is important, as it is what makes the value of our counter change to allow for a specific loop duration. It also had to be placed* ***within the While loop*** *such that the increment executes at each iteration. If we remove the increment code, we would have an infinite loop as* ***‘i’*** *(with an initial value of 1) will never be equal* *to 6*

Python loops have some extra control statements for handling their execution. We have the ‘Break’, ’Continue’ and ‘pass’. The break statement is used to stop the execution of the loop once invoked. It could sometimes be useful for safeguarding a loop against defaulting to an infinite loop. The continue statement, however, re-tests the condition of the loop from when it is invoked, this leads to a continuation of the loop from a stop point (usually used to resume the loop after a break statement).

We would not use the pass statement here, as it is only useful for skipping the execution of a loop expression which is otherwise required by the loop syntax.

Example 40: Let us use a while loop along with the continue statement to create a program that prints numbers from 1 to 10, but skips 8.

```
In []: # This program prints 1-10 but skips 8


                        i = 0           # Counter


       while i<=9:
                            i = i+1       # Increment
                            if i == 8:
                                 continue
                            print(i)


                        else: print ('\nThe program has ended')


Out[]: 1
             2
             3
             4
             5
             6
             7
             9
             10
```

```
            The program has ended
```

Tip: *Notice how our increment was placed before the ‘* ***continue*** *statement’? This is important to prevent an infinite loop. In this case, when the counter ‘* ***i*** *’ equals 7, it is incremented to 8 and the* ***if*** *statement executes the ‘* ***continue*** *statement’. The condition is then re-tested, and the increment adds 1 to the value. Since 9 is greater than 8, the rest of the code runs till the loop ends.*

Assuming the increment is after the **if & continue** statements, when **‘i’** is equal to 8, the continue statement re-tests the condition. Now since there is no increment in the value of **‘i’** after this point, the condition 8 <= 9 will always hold true, and the loop continues indefinitely!

Let us write the indefinite loop version of this program and use the break statement to terminate it after 15 iterations.

Example 41: This program uses the break and continue statement to create an indefinite loop and an escape sequence.

```
            In []: # This program is meant to print 1-10 and skip 8
     # It runs indefinitely until a break


                      i = 1       # loop variables
                      j = 1
                      while i<=10:
                        print ('Iteration ',j)
                        if j >= 15:
                       break      # loop escape sequence
                        j = j+1


                        if i == 8:
                            continue      # indefinite loop
                        print('\tvalue =',i)
                        i = i+1
                while: print('\nThe program has ended')


Out[]: Iteration  1
```

```
    value = 1
Iteration  2
    value = 2
Iteration  3
    value = 3
Iteration  4
    value = 4
Iteration  5
    value = 5
Iteration  6
      value = 6
Iteration  7
      value = 7
Iteration  8
Iteration  9
Iteration  10
Iteration  11
Iteration  12
Iteration  13
Iteration  14
Iteration  15
```

See how the loop became indefinite at output ‘7’ as explained earlier.

Tip: *Always make sure your loop can end or insert a break statement to handle such errors.*

For loop

These loops are used for iterating through a sequence of values. It executes a statement of code per each element present in the target sequence. Like while loops, they are also subject to an indefinite execution depending on how they are declared.

The for loop has the following syntax:

```
for iterator in sequence:
        statement
```

Recall the ‘in’ operator from our comparison examples. It is a for loop operator.

The variable or name of iterator does not matter in the for loop, it could be anything; however, the sequence must be properly specified along with the loop statement.

Note: *While it is not common for for loops to be infinite in Python, it is possible. Since the for loop iterates through a sequence, it is logical that the duration of the loop is dependent on the size of the sequence. If the sequence were infinite in length, then the for loop will run forever.*

Below is a sample of an infinite for loop. Notice how the list will grow indefinitely through the increment of the list size and elements via the .append list method.

```
In []:     Indefinite_list = [0]
               for x in Indefinite_list:
               Indefinite_list.append(0)
               print(x)
```

The output is not included here because the code runs forever. You may try running this program, however, when the code runs indefinitely, click the ‘Kernel’ tab at the top of your Jupyter notebook and select ‘restart kernel and clear output’. This will restart that current cell and stop the loop immediately.

Now let us try some for loop examples.

Example 42: Let us write a code that prints all the elements in a list.

```
In []: # This program accepts a list and prints its elements numbers


          List = eval (input ('Please enter a sequence of numbers using [ ] :\t'))


          for value in List:
        print ('\nValue',value,'is : ',List[value-1])


Out[]: Please enter a sequence of numbers using [ ] :          [1,2,3,4,5]


          Value 1 is :  1
```

```
                    Value 2 is :  2

                    Value 3 is :  3

                    Value 4 is :  4

                    Value 5 is :  5
```

Recall, the for loop requires a sequence for its iteration to occur. In cases where you need a certain number of iterations and want to declare an arbitrary sequence of values with the desired length, you can use Python’s range() function.

The range function creates a sequence of numerical values starting from the lower arguments to a number of elements specified by the higher argument.

Syntax: range(x,y)

Here, **x** is the starting value of the sequence with **y** values, and the last value of a range is usually **y -1.**

Example 43: Create a list of 10 separate integers between 0 and 10.

```
In []: # using the range function
     List = list(range(0,10))
     print(List)


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

Tip: *In this example, the list was created using the list() method. However, the range function has been demonstrated.*

Exercise: Write a for loop that iterates through a List of 10 elements, and prints the value of each element along with the iteration.

Hint: *use the range() function and list() method to generate your list. Also, try different variations of this exercise (more practical and complicated applications).*

List Comprehension

This is a simple way of running for loops and some other conditional operations with lists in Python.

```
Syntax:
    [statement for iterator in sequence if condition]


    for iterator in sequence:    # Equivalent code
                    if condition:
```

Example 44: let us use List comprehension to square all the elements in a list

```
In []: trial_List = list(range(10))
    [(trial_List[item])**2 for item in trial_List]


Out[]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

To use the **IF** statement part of the syntax, we consider a quick **IF** statement syntax allowed by Python.

An if ***condition*** else B

This allows you to write a one-line IF statement that executes the action/statement '**A** ' if ***condition*** is met, else it evaluates **B** . Hence, for the **IF** statement part of List comprehension syntax, we can consider the preceding ' statement for iterator in sequence ' as '**A** ' which executes if a condition is met.

Example 45: Let us use the IF part of list comprehension to grab only the first three letters of each string item in a list.

```
         In []: # Grab first three letters of strings


                List = ['Matrix','Trilogy',1,3.4,'Cattle']


                # grab loop


                new_List = [things[:3] for things in List if type(things)== str]
                print(new_List)
```

```
            Out[]: ['Mat', 'Tri', 'Cat']
```

See how useful this code is and compact too. Here we iterate through the **List** using the ***iterator*** ‘things’, and we asked Python to grab the slice of ‘things’ (which would be the element it indexes at any instant in the loop) up till the third element i.e. things [:3] (recall the slice/range grabbing examples and exercises?). All these actions are then controlled by the last IF statement that checks if the element indexed by ‘things’ has a **type == str** i.e. is a string. Easy enough?

Exercise: To the list in example 45, add the following words ‘Jeep’, ‘Man’ and ‘Go’. Run the same code from example 45 with the updated list, however, only grab the first 3 letters of words that have at least 3 letters i.e. all items in your output list must have three letters.

Hint: *Use the list .extend() method from previous examples to update the list. Also, use the function len() to check for the number of letters in a word. Finally, the IF statement can take logical operators* ***‘and, or, not’*** .

With list comprehension, using the IF-ELSE statement is a little different. In this case, the IF-ELSE statement comes first, and then the for loop.

Syntax: [A if condition else B for iterator in sequence]

This is just like the one-lined IF statement, however with a for loop alongside it.

Example 46: Write a program that accepts user input for the number of list elements. Then the program creates a numeric list from 1 to that number, with values being squares of each odd number, and half for the even numbers.

```
In []: # Create user defined list

             nitems = eval(input('Please enter number of list items: '))

                      new_list  = [x**2 if x%2 != 0 else x/2 for x in range(1,nitems+1)]

                      print('\n  Here is your custom list\n\n',new_list)

             Out[]: Please enter the number of list items: 10
```

```
Here is your custom list

[1, 1.0, 9, 2.0, 25, 3.0, 49, 4.0, 81, 5.0]
```

See how we created such a powerful program in 3 lines. That’s the advantage of list comprehension.

Tip: Take time to review the syntax for list comprehension, as it will be very useful in writing concise codes in the future.

There are other comprehension types in Python, like the dictionary comprehension; however, list comprehension is what we need for now.

Functions

In Python, or any other programming language, the use of functions cannot be underestimated. It allows the programmer to call certain code blocks to perform an action without having to write such code every time. Python has some in-built functions like the ‘range ()’, ‘list ()’, ‘type ()’, etc. that we have used in previous examples. As noticed, using these functions saved us time in their use. However, since Python is more like a tool/template for writing various programs that are supported within its library, it can hardly contain functions that would perform any action a programmer might require. This is why there is an option for defining custom functions in Python. With this option, you can write and call your own code for performing a larger operation.

To define a function in Python, we use the def () keyword. You may choose to interpret the keyword to mean ‘define’. It makes it easier to comprehend.

```
Syntax:          def function_name(‘arguments’):
                   lines of code
```

The **function_name** is the whatever you wish to name your function as. This should be a name that makes sense as regards what the function does. Functions also accept **arguments** , which are inputs for the function to evaluate. Depending on the function you choose to define, the arguments can be of anything, or nothing.

Seems easy enough, right? Let us define some functions and call them to perform certain actions.

Example 47: Define a function that prints (‘Hello User’) when called.

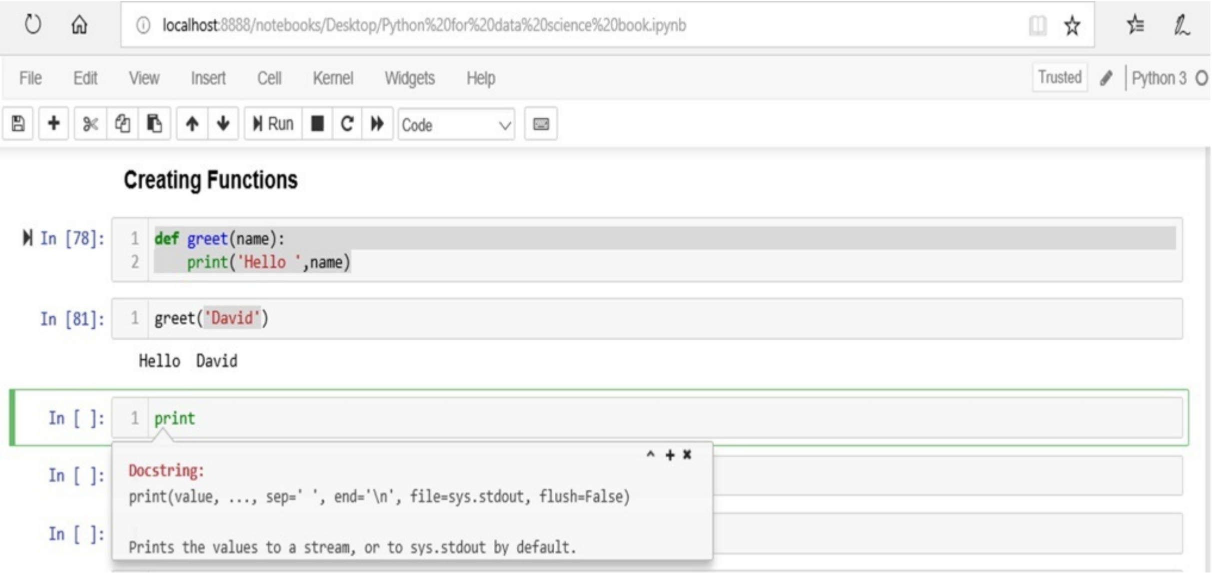
```
In []:      def greet(name):
                print('Hello ',name)
```

Now, let us call the function

```
In []:      greet('David')

                 Hello David


In []:     greet ('Python User guy')


                 Hello  Python User guy
```

See, we only needed to call the function and pass in our argument. With this method, we can output a greeting to any string value without having to write any print statement.

It is good practice to include a ‘docstring’ when writing a function. These are comments (usually multi-line) that tell the user of the function how to call it and what it does. In Jupyter notebook, once the name of a function is typed, press **shift key** + **tab** in windows, and the docstring is shown. You can try this out for any of the functions we have used so far.


localhost:8888/notebooks/Desktop/Python%20for%20data%20science%20book.ipynb
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code

Creating Functions

```
In [78]: def greet(name):
             print('Hello ',name)

In [81]: greet('David')

         Hello  David

In [ ]:  print

Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.

In [ ]:

In [ ]:
```

Figure 4: Docstring example for the print function.

To include a docstring for your function, just add a multi-line comment using the three quotes method. You can now check for the documentation of your function.

Let us try to recreate the example 46 using a function. The function prompts the user for the required list length, and then outputs a custom list. We would also include a documentation.

Example 48:

```
In []:   # function that creates a user-defined list


          def custom_list(nitems):


          """
         This function accepts a numeric value and creates a list.
         The list contains squares of odd numbers within the range
         of 1 and user input, as well as, half values for the even
          numbers in that range.


          syntax: custom_list(numeric value)
```

```
    """
    if nitems > 0:      # This ensures user compliance

   new_list  = [x**2 if x%2 != 0 else x/2 for x in range(1,nitems+1)]

   else: print('Enter a valid number !')  # in case of a fault.

    print('\n Here is your custom list\n\n',new_list)
```

Now, we have created our function. Let us call the function to view the action.

```
In []:        # calling the function
        custom_list(5)

Out[]:      Here is your custom list

                 [1, 1.0, 9, 2.0, 25]
```

The function works! However, since we only printed the output, we cannot access the list created by the function. Assuming we need to use that list for other operations within our code rather than just to view it, we can use the return statement to assign our function output to a variable, which we can then use later.

This is just a variation of the code in example 48.

Example 49:

```
In []: # function that creates a user-defined list and returns a value

        def custom_list(nitems):

         """
        This function accepts a numeric value and returns a list.
        The list contains squares of odd numbers within the range
        of 1 and user input, as well as, half values for the even
         numbers in that range.
```

```
    syntax: List_name = custom_list(numeric value)

    """
    if nitems > 0:      # This ensures user compliance

        new_list  = [x**2 if x%2 != 0 else x/2 for x in range(1,nitems+1)]

    else: print('Enter a valid number !')  # in case of a fault.

    return new_list
```

Now let us call our function to assign the list to a variable ‘List’.

```
In []:    List = custom_list(10)
          List

Out[]: [1, 1.0, 9, 2.0, 25, 3.0, 49, 4.0, 81, 5.0]
```

Notice how calling ‘List’ produces an output? The function works properly.

Exercise: Try creating a function that performs a variation of the action in example 45. The function accepts an input list (of mixed types), and extracts only numbers from the list.

Tip: for detecting numbers, you can use a variation of the method illustrated in example 45 i.e. type(item) == int or double. Another method is to use the .isdigit() method, i.e. item.isdigits

Example 50: Illustrating the .isdigits() method.

```
In []: string = '123456ABC'          # mixed string with digits and characters
            A = [x for x in string if x.isdigit()]
            A
Out[]: ['1', '2', '3', '4', '5', '6']
```

To conclude this introductory chapter to Python programming, we would look at a few more functions that would be useful is the next section. These include: the map function, filter, and lambda expressions. Let us discuss the lambda expressions first, as they are mostly useful to map functions and filters as well.

Lambda expression

This is also called an anonymous function. Lambda expressions are used in instances where you would not want to write a full function. In this case, you only need to use the function temporarily, and defining an entire function for just that seems redundant.

It also reduces the effort in writing code for defining functions. It has the following syntax: `var = lambda argument: statemen t`

A function can equally be expressed in one line as such:

```
def function_name (argument): return output
```

Example 51: let’s write a quick function that squares a number and returns a value.

```
In []:        def square_value (number): return number**2   #definition
      square_value(10)  # calling the function


Out []:      100
```

Alternatively, using lambda expression:

```
In []:       square = lambda number:number**2
      square(10)


Out[]:        100
```

The lambda expression in example 51, can be compared with the square_value function. It can be observed that some words have been eliminated: def, return and function_name. Note that lambda expressions have to be assigned to a variable, and then the variable acts as a function as illustrated by the ‘square’ variable which achieves the same result as the square_value function.

Map function

The map function is another convenient way of working with lists, especially in cases where there is a need to pass the elements of a list to a function in an iteration. As expected, for loops are the first consideration for such a task. However, the map function, which iterates through every element in a list and passes them to a function argument, simplifies such task.

```
Syntax:     map (function, sequence/list)

To return the map as a list of values, use the list () function and assign to a variable.

var =list (map (function, sequence/list))
```

You could pass an actual function to map, or more practically, use a lambda expression.

Example 52: Let us write a code using the map function to square all the elements in a list.

```
In []: # This program squares all the elements in a list
          List = list(range(1,11))

            # using the lambda expression 'square'
          new_list = list(map(square,List))
          new_list


Out[]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Exercise: Using the lambda expression and map function, write a program that creates a list of values and displays all the elements in the list.

Filter function

These are used for extracting specific elements of a sequence based on a Boolean function. The Boolean function can also be created using a lambda expression, and the filter returns the expected values that meet the filter criteria (consider it as a simplified version of an if statement).

Syntax: filter (function, sequence/list)

To return the map as a list of values, use the list () function and assign to a variable.

```
var = list (filter (function, sequence/list))
```

Example 53: Extract only the even numbers from the new_list created from example 52 using the filter function**.**

```
In []: # Extracting even numbers
          criteria = lambda value:value%2==0   # even number identifier
       # using the new_list from example 52
even_list = list(filter(criteria,new_list))
       even_list


Out[]:     [4, 16, 36, 64, 100]
```

The lambda, map and filter are usually used respectively. It is important to master them and it will considerably improve your skills with list manipulation which is very important in the subsequent section.

This concludes the introductory part to Python and all the pre-requisites for following the lessons in the next chapter. Complete the following exercises to test your current knowledge of major lessons from this chapter. Best of luck!

Exercises:

These exercises test your skills on all you have learned in this chapter. Try to solve them yourself, and if you find any difficulty, review the examples and syntax all over.

1. Given the following variables: quantity = ‘Gravity’, unit = ‘m/s^2’, and value = 10, use the .format() method to print the following statement: **Gravity has a value of 10 m/s^2.**

2. Write a function that prints **True** if the word ‘**good** ’ occurs in any list of strings.

3. Use the lambda expression and filter function to extract words that do not start with the letter ‘**b** ’ in the following list [‘bread’,’rice’,’butter’,’beans’,’pizza’,’lasagna’,’eggs’]

4. Given this nested list, use indexing to grab the word "hello Python”.

```
list = [1,2,[3,4],[5,[100,200,['hello Python']],23,11],1,7]
```

5. Given this nested dictionary grab the word "hello Python"

```
d = {'k1':[1,2,3,{'tricky':['oh','man','inception',{'target':[1,2,3,'hello']}]}]}
```

6. Write a function that accepts two arguments. The first argument is the student name, the other is the student score. Let the program print the student name and grade.

The output is dependent on the following criteria:

Criteria:

If the student score is less than 40:	Hello ***student name*** you had an **F**
If the student score is between 40 and 44:	Hello ***student name*** you had an **E**
If the student score is between 45 and 49:	Hello ***student name*** you had a **D**
If the student score is between 50 and 59:	Hello ***student name*** you had a **C**
If the student score is between 60 and 69:	Hello ***student name*** you had a **B**
If the student score is from 70 and above:	Hello ***student name*** you had an **A, Cool!**

Chapter 2

Data Analysis with Python

Data analysis includes all the techniques and processes used in extracting information from raw data. Since raw data is usually unstructured in form, and hardly informative, the need to organize such data becomes even more important. While there are many extra tools that can be used for handling data analysis i.e. Microsoft Excel, R-language, SQL, etc., most data scientist prefer to use Python due to its extensive libraries and support packages for data analysis. The most popular packages/frameworks which we would be exploring in this chapter are NumPy and Pandas.

NumPy

This is the numerical Python package which supports vector and matrix operations. It is a very popular Python package for scientific, mathematical and engineering programming; especially for linear algebraic problems. To a large extent, numeric data can be simplified into arrays (vectors or matrices, depending on dimensions), and this is why NumPy is equally useful in data manipulation and organization.

Package Installation

To get started with NumPy, we have to install the package into our version of Python. While the basic method for installing packages to Python is the `pip install` method, we will be using the `conda install` method. This is the recommended way of managing all Python packages and virtual environments using the anaconda framework.

Since we installed a recent version of Anaconda, most of the packages we need would have been included in the distribution. To verify if any package is installed, you can use the `conda list` command via the anaconda prompt. This displays all the packages currently installed and accessible via anaconda. If your intended package is not available, then you can install via this method:

First, ensure you have an internet connection. This is required to download the target package via conda. Open the anaconda prompt, then enter the following code:

```
Conda install package
```

***Note** : In the code above, ‘package’ is what needs to be installed e.g. NumPy, Pandas, etc.*

As described earlier, we would be working with NumPy arrays. In programming, an array is an ordered collection of similar items. Sounds familiar? Yeah, they are just like Python lists, but with superpowers. NumPy arrays are in two forms: Vectors, and Matrices. They are mostly the same, only that vectors are one-dimensional arrays (either a column or a row of ordered items), while a matrix is 2-dimensional (rows and columns). These are the fundamental blocks of most operations we would be doing with NumPy. While arrays incorporate most of the operations possible with Python lists, we would be introducing some newer methods for creating, and manipulating them.

To begin using the NumPy methods, we have to first import the package into our current workspace. This can be achieved in two ways:

```
import numpy as np

Or

from numpy import *
```

In Jupyter notebook, enter either of the codes above to import the NumPy package. The first method of import is recommended, especially for beginners, as it helps to keep track of the specific package a called function/method is from. This is due to the variable assignment e.g. ‘np’, which refers to the imported package throughout the coding session.

Notice the use of an asterisk in the second import method. This signifies ‘everything/all’ in programming. Hence, the code reads ‘**from NumPy import everything!!** ’

***Tip:** In Python, we would be required to reference the package we are operating with e.g. NumPy, Pandas, etc. It is easier to assign them variable names that can be used in further operations. This is significantly useful in a case where there are multiple packages being used, and the use of standard variable names such* *as: ‘np’ for NumPy, ‘pd’ for Pandas, etc. makes the code more readable.*

Example 55: Creating vectors and matrices from Python lists.

Let us declare a Python list.

```
In []:      # This is a list of integers
                 Int_list = [1,2,3,4,5]
            Int_list


Out[]:       [1,2,3,4,5]
```

Importing the NumPy package and creating an array of integers.

```
In []:      # import syntax
                  import numpy as np
                  np.array(Int_list)


Out[]:       array([1, 2, 3, 4, 5])
```

Notice the difference in the outputs? The second output indicates that we have created an array, and we can easily assign this array to a variable for future reference.

To confirm, we can check for the type.

```
In []:       x = np.array(Int_list)
           type(x)
Out[]:       numpy.ndarray
```

We have created a vector, because it has one dimension (1 row). To check this, the ‘ndim’ method can be used.

```
                    In []:      x.ndim    # this shows how many dimensions the array has
                    Out[]:       1
```

Alternatively, the shape method can be used to see the arrangements.

```
In []:       x.shape   # this shows the shape


Out[]:       (5,)
```

Python describes matrices as (rows, columns) . In this case, it describes a vector as (number of elements,) .

To create a matrix from a Python list, we need to pass a nested list containing the elements we need. Remember, matrices are rectangular, and so each list in the nested list must have the same size.

```
In []: # This is a matrix

        x = [1,2,3]
        y = [4,5,6]

        my_list = [y,x]  # nested list

        my_matrix = np.array(my_list)  # creating the matrix

        A = my_matrix.ndim
        B = my_matrix.shape

        # Printing
        print('Resulting matrix:\n\n',my_matrix,'\n\nDimensions:',A,
        '\nshape (rows,columns):',B)

Out[]: Resulting matrix:

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

         Dimensions: 2
         shape (rows,columns): (2, 3)
```

Now, we have created a 2 by 3 matrix. Notice how the shape method displays the rows and columns of the matrix. To find the transpose of this matrix i.e. change the rows to columns, use the transpose ( ) method.

```
In []:      # this finds the transpose of the matrix
                  t_matrix = my_matrix.transpose()
        t_matrix


Out[]:      array([[4, 1],
                   [5, 2],
                   [6, 3]])
```

Tip: *Another way of knowing the number of dimensions of an array is by counting the square-brackets that opens and closes the array (immediately after the parenthesis). In the vector example, notice that the array was enclosed in single square brackets. In the two-dimensional array example, however, there are two brackets. Also, tuples can be used in place of lists for creating arrays.*

There are other methods of creating arrays in Python, and they may be more intuitive than using lists in some application. One quick method uses the arange( ) function.

Syntax: np.arange(start value, stop value, step size, dtype = ‘type’)

This method is similar to the range( ) method we used in example 43. In this case, we do not need to pass its output to the list function, our result is an array object of a data type specified by ‘dtype’.

Example 56 : Creating arrays with the arange() function.

We will create an array of numbers from 0 to 10, with an increment of 2 (even numbers).

```
In []:      # Array of even numbers between 0 and 10
                  Even_array = np.arange(0,11,2)
                  Even_array


Out[]:      array([ 0,  2,  4,  6,  8, 10])
```

Notice it behaves like the range () method form our list examples. It returned all even values between 0 and 11 (10 being the maximum). Here, we did not specify the types of the elements.

Tip: *Recall, the range method returns value up to the ‘stop value – 1’; hence, even if we change the 11 to 12, we would still get 10 as the maximum.*

Since the elements are numeric, they can either be integers or floats. Integers are the default, however, to return the values as floats, we can also specify the numeric type.

```
          In []:      Even_array2 = np.arange(0,11,2, dtype='float')
         Even_array2


Out[]:     array([ 0.,  2.,  4.,  6.,  8., 10.])
```

Another handy function for creating arrays is linspace( ) . This returns a numeric array of linearly space values within an interval. It also allows for the specification of the required number of points, and it has the following syntax:

```
np.linspace(start value, end value, number of points)
```

At default, linspace returns an array of 50 evenly spaced points within the defined interval.

Example 57 : Creating arrays of evenly spaced points with linspace()

```
In []: # Arrays of linearly spaced points
             A = np.linspace(0,5,5) # 5 equal points between 0 & 5
             B = np.linspace (51,100) # 50 equal points between 51 & 100
       print ('Here are the arrays:\n')
             A
             B


Here are the arrays:




Out[ ]: array([0.  , 1.25, 2.5 , 3.75, 5.  ])
Out[ ]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14., 15., 16., 17., 18., 19.,
  20., 21., 22., 23., 24., 25., 26., 27., 28., 29., 30., 31., 32., 33., 34., 35., 36., 37., 38., 39., 40.,
  41., 42., 43., 44., 45., 46., 47., 48., 49., 50.])
```

Notice how the second use of linspace did not require a third argument. This is because we wanted 50 equally spaced values, which is the default. The ‘dtype’ can also be specified like we did with the range function.

Tip 1: *Linspace arrays are particularly useful in plots. They can be used to create a time axis or any other required axis for producing well defined and scaled graphs.*

Tip 2: *The output format in the example above is not the default way for output in Jupyter notebook. Jupyter displays the last result per cell, at default. To display multiple results (without having to use the print statement every-time), the output method can be changed using the following code.*

```
In[]: # Allowing Jupyter output all results per cell.
      # run the following code in a Jupyter cell.

      from IPython.core.interactiveshell import InteractiveShell
      InteractiveShell.ast_node_interactivity = "all"
```

There are times when a programmer needs unique arrays like the identity matrix, or a matrix of ones/zeros. NumPy provides a convenient way of creating these with the zeros( ) , ones( ) and eye( ) functions.

Example 58: creating arrays with unique elements.

Let us use the zeros () function to create a vector and a matrix.

```
In []: np.zeros(3)  # A vector of 3 elements
       np.zeros((2,3)) # A matrix of 6 elements i.e. 2 rows, 3 columns

Out[]: array([0., 0., 0.])
Out[]: array([[0., 0., 0.],
              [0., 0., 0.]])
```

Notice how the second output is a two-dimensional array i.e. two square brackets (a matrix of 2 columns and 3 rows as specified in the code).

The same thing goes for creating a vector or matrix with all elements having a value of ‘1’.

```
In []: np.ones(3)  # A vector of 3 elements
```

```
                                    np.ones((2,3)) # A matrix of 6 elements i.e. 2 rows, 3 columns


Out[]: array([1., 1., 1.])
Out[]: array([[1., 1., 1.],
                   [1., 1., 1.]])
```

Also, notice how the code for creating the matrices requires the row and column instructions to be passed as a tuple. This is because the function accepts one input, so multiple inputs would need to be passed as tuples or lists in the required order (Tuples are recommended. Recall, they are faster to operate.).

In the case of the identity matrix, the function eye () only requires one value. Since identity matrices are always square, the value passed determines the number of rows and columns.

```
                    In []: np.eye(2)  # A matrix of 4 elements 2 rows, 2 columns
                                    np.eye(3)  # 3 rows, 3 columns


                    Out[]: array([[1., 0.],
                                       [0., 1.]])
                    Out[]: array([[1., 0., 0.],
                                       [0., 1., 0.],
                                       [0., 0., 1.]])
```

NumPy also features random number generators. These can be used for creating arrays, as well as single values, depending on the required application. To access the random number generator, we call the library via np.rando m , and then choose the random method we prefer. We will consider three methods for generating random numbers: rand( ) , randn( ) , and randint( ) .

Example 59: Generating arrays with random values.

Let us start with the rand () method. This generates random, uniformly distributed numbers between 0 and 1.

```
                  In []: np.random.rand (2)    # A vector of 2 random values
                                np.random.rand (2,3)  # A matrix of 6 random values


                  Out[]: array([0.01562571, 0.54649508])
```

```
Out[]: array([[0.22445055, 0.35909056, 0.53403529],
                [0.70449515, 0.96560456, 0.79583743]])
```

Notice how each value within the arrays are between 0 & 1. You can try this on your own and observe the returned values. Since it is a random generation, these values may be different from yours. Also, in the case of the random number generators, the matrix specifications are not required to be passed as lists or tuples, as observed in the second line of code.

The randn () method generates random numbers from the standard normal or Gaussian distribution. You might want to brush up on some basics in statistics, however, this just implies that the values returned would have a tendency towards the mean (which is zero in this case) i.e. the values would be centered around zero.

```
In []: np.random.randn (2)     # A vector of 2 random values
         np.random.randn (2,3)  # A matrix of 6 random values


Out[]: array([ 0.73197866, -0.31538023])
Out[]: array([[-0.79848228, -0.7176693 , 0.74770505],
          [-2.10234448,  0.10995745, -0.54636425]])
```

The randint() method generates random integers within a specified range or interval. Note that the higher range value is exclusive (i.e. has no chance of being randomly selected), while the lower value is inclusive (could be included in the random selection).

Syntax: np.random(lower value, higher value, number of values, dtype)

If the number of values is not specified, Python just returns a single value within the defined range.

```
In []: np.random.randint (1,5)         # A random value between 1 and 5
         np.random.randint (1,100,6)     # A vector of 6 random values
         np.random.randint (1,100,(2,3)) # A matrix of 6 random values
Out[]: 4
Out[]: array([74, 42, 92, 10, 76, 43])
Out[]: array([[92,  9, 99],
        [73, 36, 93]])
```

Tip: *Notice how the size parameter for the third line was specified using a tuple. This is how to create a matrix of random integers using randint.*

Example 59 : Illustrating randint().

Let us create a fun dice roll program using the randint() method. We would allow two dice, and the function will return an output based on the random values generated in the roll.

```
In []: # creating a dice roll game with randint()
           # Defining the function
           def roll_dice():
            """ This function displays a
            dice roll value when called"""

                        dice1 = np.random.randint(1,7) # This allows 6 to be inclusive
             dice2 = np.random.randint(1,7)

                           # Display Condition.
            if dice1 == dice2:
                             print('Roll: ',dice1,'&',dice2,'\ndoubles !')
           if dice1 == 1:
             print('snake eyes!\n')
            else:
                            print('Roll: ',dice1,'&',dice2)
```

```
In []:     # Calling the function
           roll_dice()

Out[]:      Roll:  1 & 1
           doubles !
           snake eyes!
```

Hint: *Think of a fun and useful program to illustrate the use of these random number generators, and writing such programs will improve your chances of comprehension. Also, a quick review of statistics, especially measures of central tendency & dispersion/spread will be useful in your data science journey.*

Manipulating arrays

Now that we have learned how to declare arrays, we would be proceeding with some methods for modifying these arrays. First, we will consider the reshape ( ) method, which is used for changing the dimensions of an array.

Example 60 : Using the reshape() method.

Let us declare a few arrays and call the reshape method to change their dimensions.

```
In []: freq = np.arange(10);values = np.random.randn(10)
       freq; values


Out[]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])


Out[]: array([ 1.33534821,  1.73863505,  0.1982571 , -0.47513784, 1.80118596, -1.73710743,
            -0.24994721, 1.41695744, -0.28384007,  0.58446065])
```

Using the reshape method, we would make ‘freq’ and ‘values’ 2 dimensional.

```
In []: np.reshape(freq,(5,2))


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

```
In []: np.reshape(values,(2,5))


Out[]: array([[ 1.33534821,  1.73863505,  0.1982571 , -0.47513784,  1.80118596],
            [-1.73710743, -0.24994721,  1.41695744,    -0.28384007,  0.58446065]])
```

Even though the values array still looks similar after reshaping, notice the two square brackets that indicate it has been changed to a matrix. The reshape method comes in handy when we need to do array operations, and our arrays are inconsistent in dimensions. It is also important to ensure the new size parameter passed to the reshape method does not differ from the number of elements in the original array. The idea is simple: when calling the reshape method, the product of the size parameters must equal the number of elements in the original array.

As seen in Example 60, the size parameter passed as a tuple to the reshape methods gives a value of 10 when multiplied, and this is also the number of elements in ‘freq’ and ‘values’ respectively.

There are times when we may need to find the maximum and minimum values within an array (or real-world data), and possibly the index of such maximum or minimum values. To get this information, we can use the .max( ) , .min( ) , .argmax( ) and .argmin( ) methods respectively.

Example 61:

Let us find the maximum and minimum values in the ‘values’ array, along with the index of the minimum and maximum within the array.

```
In []: A = values.max();B = values.min();
       C = values.argmax()+1; D = values.argmin()+1


    print('Maximum value: {}\nMinimum Value: {}\
 \nItem {} is the maximum value, while item {}\
       is the minimum value'.format(A,B,C,D))
```

Output

```
Maximum value: 1.8011859577930067
Minimum Value: -1.7371074259180737
Item 5 is the maximum value, while item 6 is the minimum value
```

A few things to note in the code above: The variables C&D, which defines the position of the maximum and minimum values are evaluated as shown [by adding 1 to the index of the maximum and minimum values obtained via argmax ( ) and argmin ( )], because Python indexing starts at zero. Python would index maximum value at 4, and minimum at 5, which is not the actual positions of these elements within the array (you are less likely to start counting elements in a list from zero! Unless you are Python, of course.).

Another observation can be made in the code. The print statement is broken across a few lines using enter. To allow Python to know that the next line of code is a continuation, the backslash ‘\’ is used. Another way would be to use three quotes for a multiline string.

Indexing and selecting arrays

Array indexing is very much similar to List indexing with the same techniques of item selection and slicing (using square brackets). The methods are even more similar when the array is a vector.

Example 62:

```
In []: # Indexing a vector array (values)
       values
       values[0]   # grabbing 1st item
       values[-1]  # grabbing last item
       values[1:3] # grabbing 2nd & 3rd item
       values[3:8] # item 4 to 8

Out[]: array([ 1.33534821,  1.73863505,   0.1982571 , -0.47513784,  1.80118596,
  -1.73710743,      -0.24994721,  1.41695744, -0.28384007,  0.58446065])

Out[]: 1.3353482110285562

Out[]: 0.5844606470172699

Out[]: array([1.73863505, 0.1982571 ])

Out[]: array([-0.47513784,  1.80118596, -1.73710743, -0.24994721,  1.41695744])
```

The main difference between arrays and lists is in the broadcast property of arrays. When a slice of a list is assigned to another variable, any changes on that new variable does not affect the original list. This is seen in the example below:

```
In []: num_list = list(range(11))  # list from 0-10
       num_list                    # display list
       list_slice = num_list[:4]   # first 4 items
       list_slice                  # display slice
                                            list_slice[:] = [5,7,9,3]    # Re-
                                   assigning elements
                                            list_slice              # display
                                   updated values
```

```
                # checking for changes
                print(' The list changed !') if list_slice == num_list[:4]\
                else print(' no changes in original list')

Out[]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Out[]: [0, 1, 2, 3]

Out[]: [5, 7, 9, 3]

               no changes in the original list
```

For arrays, however, a change in the slice of an array also updates or broadcasts to the original array, thereby changing its values.

```
In []: # Checking the broadcast feature of arrays

                num_array = np.arange(11)     # array from 0-10

                num_array                     # display array

                array_slice = num_array[:4]  # first 4 items

                array_slice                  # display slice

                                                       array_slice[:] = [5,7,9,3]    # Re-
                                                  assigning elements

                                                        array_slice                # display
                                                  updated values
                num_array
```

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

Out[]: array([0, 1, 2, 3])

Out[]: array([5, 7, 9, 3])
```

```
Out[]: array([ 5,  7,  9,  3,  4,  5,  6,  7,  8,  9, 10])
```

This happens because Python tries to save memory allocation by allowing slices of an array to be like shortcuts or links to the actual array. This way it doesn't have to allocate a separate memory location to it. This is especially ingenious in the case of large arrays whose slices can also take up significant memory. However, to take up a slice of an array without broadcast, you can create a 'slice of a copy' of the array. The array.copy( ) method is called to create a copy of the original array.

```
In []:   # Here is an array allocation without broadcast
              num_array     # Array from the last example


          # copies the first 4 items of the array copy
          array_slice = num_array.copy()[:4]


                                              array_slice               # display
                                          array
                                              array_slice[:] = 10         # re-
                                          assign array
                                              array_slice               # display
                                          updated values
                                              num_array                 # checking
                                          original list


Out[]:   array([ 5,  7,  9,  3,  4,  5,  6,  7,  8,  9, 10])


Out[]:   array([5, 7, 9, 3])


Out[]:   array([10, 10, 10, 10])


Out[]:   array([ 5,  7,  9,  3,  4,  5,  6,  7,  8,  9, 10])
```

Notice that the original array remains unchanged.

For two-dimensional arrays or matrices, the same indexing and slicing methods work. However, it is always easy to consider the first dimension as the rows and the other as the columns. To select any item or slice of items, the index of the rows and columns are specified. Let us illustrate this with a few examples:

Example 63 : Grabbing elements from a matrix

There are two methods for grabbing elements from a matrix: array_name[row][col] or array_name[row,col] .

```
In []: # Creating the matrix
          matrix = np.array(([5,10,15],[20,25,30],[35,40,45]))

          matrix      #display matrix
          matrix[1]   # Grabbing second row
          matrix[2][0] # Grabbing 35
          matrix[0:2]  # Grabbing first 2 rows
          matrix[2,2]   # Grabbing 45

Out[]: array([[ 5, 10, 15],
               [20, 25, 30],
               [35, 40, 45]])

Out[]: array([20, 25, 30])

Out[]: 35

Out[]: array([[ 5, 10, 15],
        [20, 25, 30]])

Out[]: 45
```

***Tip:** It is recommended to use the array_name[row,col] method, as it saves typing and is more compact. This will be the convention for the rest of this section.*

To grab columns, we specify a slice of the row and column. Let us try to grab the second column in the matrix and assign it to a variable column_slice.

```
In []: # Grabbing the second column
          column_slice = matrix[:,1:2] # Assigning to variable
          column_slice

Out[]: array([[10],
                        [25],
                        [40]])
```

Let us consider what happened here. To grab the column slice, we first specify the row before the comma. Since our column contains elements in all rows, we need all the rows to be included in our selection, hence the ‘ : ’ sign for all. Alternatively, we could use ‘0 : ’ , which might be easier to understand. After selecting the row, we then choose the column by specifying a slice from ‘1:2 ’ , which tells Python to grab from the second item up to (but not including) the third item. Remember, Python indexing starts from zero.

Exercise: Try to create a larger array, and use these indexing techniques to grab certain elements from the array. For example, here is a larger array:

```
In []: # 5 × 10 Array of even numbers between 0 and 100.
large_array = np.arange(0,100,2).reshape(5,10)
large_array     # show


      Out[]: array([[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
             [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
             [40, 42, 44, 46, 48, 50, 52, 54, 56, 58],
             [60, 62, 64, 66, 68, 70, 72, 74, 76, 78],
             [80, 82, 84, 86, 88, 90, 92, 94, 96, 98]])
```

Tip: *Try grabbing single elements and rows from random arrays you create. After getting very familiar with this, try selecting columns. The point is to try as many combinations as possible to get you familiar with the approach. If the slicing and indexing notations are confusing, try to revisit the section under list or string slicing and indexing.*

Click this link to revisit the examples on slicing: *List indexing*

Conditional selection

Consider a case where we need to extract certain values from an array that meet a Boolean criterion. NumPy offers a convenient way of doing this without having to use loops.

Example 64: Using conditional selection

Consider this array of odd numbers between 0 and 20. Assuming we need to grab elements above 11. We first have to create the conditional array that selects this:

```
                                              In []: odd_array = np.arange(1,20,2)   #
                                               Vector of odd numbers
          odd_array                 # Show vector
                                                     bool_array = odd_array >
                                               11     # Boolean conditional array
          bool_array


     Out[]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])


     Out[]: array([False, False, False, False, False, False,  True, True, True, True])
```

Notice how the bool_array evaluates to True at all instances where the elements of the odd_array meet the Boolean criterion.

The Boolean array itself is not usually so useful. To return the values that we need, we will pass the Boolean_array into the original array to get our results.

```
     In []: useful_Array = odd_array[bool_array]  # The values we want
          useful_Array


     Out[]: array([13, 15, 17, 19])
```

Now, that is how to grab elements using conditional selection. There is however a more compact way of doing this. It is the same idea, but it reduces typing.

Instead of first declaring a Boolean_array to hold our truth values, we just pass the condition into the array itself, like we did for useful_array.

```
     In []: # This code is more compact
          compact = odd_array[odd_array>11] # One line
          compact


     Out[]: array([13, 15, 17, 19])
```

See how we achieved the same result with just two lines? It is recommended to use this second method, as it saves coding time and resources. The first method helps explain how it all works. However, we would be using the second method for all other instances in this book.

Exercise: The conditional selection works on all arrays (vectors and matrices alike). Create a two 3 × 3 array of elements greater than 80 from the ‘large_array’ given in the last exercise.

Hint: *use the reshape method to convert the resulting array into a 3 × 3 matrix.*

NumPy Array Operations

Finally, we will be exploring basic arithmetical operations with NumPy arrays. These operations are not unlike that of integer or float Python lists.

Array – Array Operations

In NumPy, arrays can operate with and on each other using various arithmetic operators. Things like the addition of two arrays, division, etc.

Example 65:

```
In []: # Array - Array Operations

          # Declaring two arrays of 10 elements
          Array1 = np.arange(10).reshape(2,5)
          Array2 = np.random.randn(10).reshape(2,5)
          Array1;Array2          # Show the arrays

          # Addition
          Array_sum = Array1 + Array2
          Array_sum              # show result array

          #Subtraction
          Array_minus = Array1 - Array2
          Array_minus            # Show array

          # Multiplication
          Array_product = Array1 * Array2
          Array_product          # Show
```

```
                # Division
                Array_divide = Array1 / Array2
                Array_divide            # Show


        Out[]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])


        array([[ 2.09122638,  0.45323217, -0.50086442,  1.00633093,  1.24838264],  [
          1.64954711,     -0.93396737,  1.05965475,  0.78422255,        -1.84595505]])


        array([[2.09122638, 1.45323217, 1.49913558, 4.00633093, 5.24838264], [6.64954711,
          5.06603263, 8.05965475, 8.78422255, 7.15404495]])


        array([[-2.09122638,  0.54676783,  2.50086442,  1.99366907,  2.75161736], [
          3.35045289,  6.93396737,  5.94034525,  7.21577745, 10.84595505]])


        array([[  0.      ,   0.45323217,  -1.00172885,   3.01899278, 4.99353055],
          [  8.24773555,      -5.60380425,   7.41758328,   6.27378038,     -16.61359546]])


        array([[ 0.      ,  2.20637474, -3.99309655,  2.9811267 ,  3.20414581],  [
          3.03113501,     -6.42420727,  6.60592516, 10.20118591,       -4.875525  ]])
```

Each of the arithmetic operations performed are element-wise. The division operations require extra care however. In Python, most arithmetic errors in code throw a run-time error, which helps in debugging. For NumPy, however, the code could run with a warning issued.

Array – Scalar operations

Also, NumPy supports scalar with Array operations. A scalar in this context is just a single numeric value of either integer or float type. The scalar – Array operations are also element-wise, by virtue of the broadcast feature of NumPy arrays.

Example 66:

```
        In []: #Scalar- Array Operations
                new_array = np.arange(0,11)     # Array of values from 0-10
                print('New_array')
                new_array                   # Show
```

```
          Sc = 100                    # Scalar value

          # let us make an array with a range from 100 - 110 (using +)
          add_array = new_array + Sc      # Adding 100 to every item
          print('\nAdd_array')
          add_array                   # Show

          # Let us make an array of 100s (using -)
          centurion = add_array - new_array
          print('\nCenturion')
          centurion                   # Show
          # Let us do some multiplication (using *)

          multiplex = new_array * 100
          print('\nMultiplex')
          multiplex                   # Show

          # division [take care], let us deliberately generate
          # an error. We will do a divide by Zero.

          err_vec = new_array / new_array
          print('\nError_vec')
          err_vec                     # Show

          New_array

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

          Add_array

Out[]: array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110])

          Centurion

Out[]: array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100])
```

```
                Multiplex


        Out[]: array([   0,  100,  200,  300,  400,  500,  600,  700,  800,  900, 1000])


                Error_vec
            C:\Users\Oguntuase\Anaconda3\lib\site-
         packages\ipykernel_launcher.py:27:            RuntimeWarning: invalid value encountered in
         true_divide


        array([nan,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Notice the runtime error generated? This divide by zero value was caused by the division of the first element of new_array by itself, i.e. 0/0 . This would give a divide by zero error in normal Python environment and would not run the code. NumPy, however, ran the code and indicated the divide by zero in the Error_vec array as a 'nan' type (not-a-number). This also goes for values that evaluate to infinity, which would be represented by the value '+/- inf' (try 1/0 using NumPy array-scalar or array-array operation.).

Tip: *Always take caution when using division to avoid such runtime errors that could later bug your code.*

Universal Array functions

These are some built-in functions designed to operate in an element-wise fashion on NumPy arrays. They include mathematical, comparison, trigonometric, Boolean, etc. operations. They are called using the np.function_name(array) method.

Example 67 : A few Universal Array functions (U-Func)

```
        In []: # Using U-Funcs
                U_add = np.add(new_array,Sc)  # addition
                U_add                    # Show


                U_sub = np.subtract(add_array,new_array)
                U_sub                    # Show
        U_log = np.log(new_array)      # Natural log
        U_log                          # Show

        sinusoid = np.sin(new_array)  # Sine wave
        sinusoid                       # Show

        # Alternatively, we can use the .method
        new_array.max()                # find maximum
        np.max(new_array)              # same thing

    Out[]: array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110])
    Out[]: array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100])

    C:\Users\Oguntuase\Anaconda3\lib\site-
      packages\ipykernel_launcher.py:8:     RuntimeWarning: divide by zero encountered in log

    Out[]: array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791, 1.79175947,
      1.94591015, 2.07944154, 2.19722458, 2.30258509])

    Out[]: array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
      -0.95892427,        -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402111])
    Out[]: 10
    Out[]: 10
```

There are still many more functions available, and a full reference can be found in the NumPy documentation for Universal functions here: https://docs.scipy.org/doc/numpy/reference/ufuncs.html

Now that we have explored NumPy for creating arrays, we would consider the Pandas framework for manipulating these arrays and organizing them into data frames.

Pandas

This is an open source library that extends the capabilities of NumPy. It supports data cleaning and preparation, with fast analysis capabilities. It is more like Microsoft excel framework, but with Python. Unlike NumPy, it has its own built-in visualization features and can work with data from a variety of sources. It is one of the most versatile packages for data science with Python, and we will be exploring how to use it effectively.

To use pandas, make sure it is currently part of your installed packages by verifying with the conda list method. If it is not installed, then you can install it using the conda install pandas command; you need an internet connection for this.

Now that Pandas is available on your PC, you can start working with the package. First, we start with the Pandas series.

Series

This is an extension of the NumPy array. It has a lot of similarities, but with a difference in indexing capacity. NumPy arrays are only indexed via number notations corresponding to the desired rows and columns to be accessed. For Pandas series, the axes have labels that can be used for indexing their elements. Also, while NumPy arrays -- like Python lists, are essentially used for holding numeric data, Pandas series are used for holding any form of Python data/object.

Example 68 : Let us illustrate how to create and use the Pandas series

First, we have to import the Pandas package into our workspace. We will use the variable name pd for Pandas, just as we used np for NumPy in the previous section.

```
In []: import numpy as np   #importing numpy for use
          import pandas as pd  # importing the Pandas package
```

We also imported the numpy package because this example involves a numpy array.

```
In []: # python objects for use
            labels = ['First','Second','Third']
                    # string list
            values = [10,20,30]          # numeric list
            array = np.arange(10,31,10)     # numpy array
            dico = {'First':10,'Second':20,'Third':30}
                  # Python dictionary


            # create various series
            c = pd.Series(values)
            print('Default series')
            A                       #show

            B = pd.Series(values,labels)
            print('\nPython numeric list and label')
            B                       #show

            C = pd.Series(array,labels)
            print('\nUsing python arrays and labels')
            C                       #show

            D = pd.Series(dico)
            print('\nPassing a dictionary')
            D                       #show

            Default series

Out[]: 0    10
           1    20
            2    30
            dtype: int64

            Python numeric list and label

Out[]:       First     10
            Second    20
            Third     30
            dtype: int64

            Using python arrays and labels

Out[]:       First     10
            Second    20
            Third     30
```

```
                 dtype: int32


                 Passing a dictionary


            Out[]: First     10
                 Second    20
                 Third     30
                 dtype: int64
```

We have just explored a few ways of creating a Pandas series using a numpy array, Python list, and dictionary. Notice how the labels correspond to the values? Also, the dtypes are different. Since the data is numeric and of type integer, Python assigns the appropriate type of integer memory to the data. Creating series from NumPy arrays returns the smallest integer size (int 32). The difference between 32 bits and 64 bits unsigned integers is the corresponding memory allocation. 32 bits obviously requires less memory (4bytes, since 8bits make a byte), and 64 bits would require double (8 bytes). However, 32bits integers are processed faster, but have a limited capacity in holding values, as compared with 64 bits.

Pandas series also support the assignment of any data type or object as its data points.

```
In []: pd.Series(labels,values)


Out[]: 10     First
   20    Second
   30     Third
   dtype: object
```

Here, the string elements of the label list are now the data points. Also, notice that the dtype is not ‘object’.

This kind of versatility in item operation and storage is what makes pandas series very robust. Pandas series are indexed using labels. This is illustrated in the following examples:

Example 69:

```
            In []: # series of WWII countries
```

```
                pool1 = pd.Series([1,2,3,4],['USA','Britain','France','Germany'])
                pool1       #show
                print('grabbing the first element')
                pool1['USA']  # first label index

Out[]:          USA        1
                Britain    2
                France     3
                Germany    4
                dtype: int64
                grabbing the first element
Out[]: 1
```

As shown in the code above, to grab a series element, use the same approach as the numpy array indexing, but by passing the label corresponding to that data point. The data type of the label is also important, notice the ‘USA’ label was passed as a string to grab the data point ‘1’. If the label is numeric, then the indexing would be similar to that of a numpy array. Consider numeric indexing in the following example:

```
In []: pool2 = pd.Series(['USA','Britain','France','Germany'],[1,2,3,4])
                pool2      #show
                print('grabbing the first element')
                pool2[1]  #numeric indexing

Out[]: 1       USA
                2    Britain
                3     France
                4    Germany
                dtype: object

                grabbing the first element

Out[]: 'USA'
```

Tip: *you can easily know the data held by a series through the dtype. Notice how the dtype for pool1 and pool2 are different, even though they were both created from the same lists. The difference is that pool2 holds strings as its data points,* *while pool1 holds integers (int64).*

Panda series can be added together. It works best if the two series have similar labels and data points.

Example 70 : Adding series

Let us create a third series ‘pool 3’. This is a similar series as pool1, but Britain has been replaced with ‘USSR’, and a corresponding data point value of 5.

```
In []: pool3 = pd.Series([1,5,3,4],['USA','USSR','France',
  'Germany'])

         pool3


Out[]: USA       1

         USSR      5

         France    3

         Germany   4

         dtype: int64
```

Now adding series:

```
In []:   # Demonstrating series addition

         double_pool = pool1 + pool1

         print('Double Pool')

         double_pool


         mixed_pool = pool1 + pool3

         print('\nMixed Pool')

         mixed_pool


         funny_pool = pool1 + pool2
         print('\nFunny Pool')
         funny_pool
         Double Pool

Out[]: USA       2
         Britain    4
         France     6
         Germany    8
         dtype: int64

         Mixed Pool

Out[]: Britain    NaN
         France     6.0
         Germany    8.0
         USA        2.0
         USSR       NaN
         dtype: float64

         Funny Pool

C:\Users\Oguntuase\Anaconda3\lib\site-     packages\pandas\core\indexes\base.py:3772:
  RuntimeWarning: '<' not supported between instances of 'str' and 'int', sort order is undefined
  for incomparable objects
        return this.join(other, how=how, return_indexers=return_indexers)

Out[]: USA       NaN
         Britain    NaN
         France     NaN
         Germany    NaN
         1          NaN
         2          NaN
         3          NaN
         4          NaN
         dtype: object
```

By adding series, the resultant is the increment in data point values of similar labels (or indexes). A ‘NaN’ is returned in instances where the labels do not match.

Notice the difference between Mixed_pool and Funny_pool: In a mixed pool, a few labels are matched, and their values are added together (due to the add operation). For Funny_pool, no labels match, and the data points are of different types. An error message is returned and the output is a vertical concatenation of the two series with ‘NaN’ Datapoints.

Tip: *As long as two series contain the same labels and data points of the same type, basic array operations like addition, subtraction, etc. can be done. The order of the labels does not matter, the values will be changed based on the operator being used. To fully grasp this, try running variations of the examples given above.*

Data frames

A Pandas data frame is just an ordered collection of Pandas series with a common/shared index. At its basic form, a data frame looks more like an excel sheet with rows, columns, labels and headers. To create a data frame, the following syntax is used:

```
pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

Usually, the data input is an array of values (of whatever datatype). The index and column parameters are usually lists/vectors of either numeric or string type.

If a Pandas series is passed to a data frame object, the index automatically becomes the columns, and the data points are assigned accordingly.

Example 71 : Creating a data frame

```
In []: df = pd.DataFrame([pool1])       # passing a series
 df                          # show


 # two series
 index = 'WWI WWII'.split()
 new_df = pd.DataFrame([pool1,pool3],index)
 new_df                      # show
```

Output:

USA	Britain	France	Germany	
0	1	2	3	4

	USA	Britain	France	Germany
WWI	1	2	3	4
WWII	5	1	3	4

We have created two data frames from the pool 1 and pool 3 series we created earlier. Notice how the first data frame assigns the series labels as column headers, and since no index was assigned, a value of ‘0’ was set at that index i.e. row header.

For the second data frame, the row labels were specified by passing a list of strings [‘WWI’,’WWII’].

***Tip:** The .split( ) string method is a quick way of creating lists of strings. It works by splitting a string into its component characters, depending on the delimiter passed to the string method.*

For example, let us split this email ‘pythonguy@gmail.com’ into a list containing the username and the domain name.

```
In []:  # Illustrating the split() method


            email = 'pythonguy@gmail.com'
            string_vec = email.split('@')
             string_vec     # show
            A = string_vec[0]; B = string_vec[1] # Extracting values
            print('Username:',A,'\nDomain name:',B)


Out[]:  ['pythonguy', 'gmail.com']
          Username: pythonguy
          Domain name: gmail.com
```

To create a data frame with an array, we can use the following method:

```
# Creating dataframe with an array

Array = np.arange(1,21).reshape(5,4)  # numpy array
row_labels = 'A B C D E'.split()
col_labels = 'odd1 even1 odd2 even2'.split()
Arr_df = pd.DataFrame(Array,row_labels,col_labels)
Arr_df
```

Output:

```
     odd1   even1   odd2   even2
A    1      2       3      4
B    5      6       7      8
C    9      10      11     12
D    13     14      15     16
E    17     18      19     20
```

Notice how this is not unlike how we create spreadsheets in excel. Try playing around with creating data frames.

Exercise: Create a data frame from a 5 × 4 array of uniformly distributed random values. Include your choice row and column names using the .split( ) method.

Hint: *use the rand function to generate your values, and use the reshape method to form an array.*

Now that we can conveniently create Data frames, we are going to learn how to index and grab elements off them.

Tip: *Things to note about data frames.*

- *They are a collection of series (more like a list with Pandas series as its elements).*
- *They are similar to numpy arrays i.e. they are more like n × m dimensional matrices, where ‘n’ are the rows and ‘m’ are the columns.*

Example 72 : Grabbing elements from a data frame.

The easiest elements to grab are the columns. This is because, by default, each column element is a series with the row headers as labels. We can grab them by using a similar method from the series – indexing by name.

```
In []: # Grab data frame elements

          Arr_df['odd1']    # grabbing first column

Out[]: A     1
          B     5
          C     9
          D    13
          E    17
          Name: odd1, dtype: int32
```

Pretty easy, right? Notice how the output is like a Pandas series. You can verify this by using the type(Arr_df[**'odd1'**]) command.

When more than one column is grabbed, however, it returns a data frame (which makes sense, since a data frame is a collection of at least two series). To grab more than one column, pass the column names to the indexing as a list. This is shown in the example code below:

```
In []:    # Grab two columns

          Arr_df[['odd1','even2']] # grabbing first and last columns
```

Output:

	odd1	**even2**
A	**1**	**4**
B	**5**	**8**
C	**9**	**12**
D	**13**	**16**
E	**17**	**20**

To select a specific element, use the double square brackets indexing notation we learned under array indexing. For example, let us select the value 15 from Arr_df.

```
In []: Arr_df['odd2']['D']

Out[]: 15
```

You may decide to break the steps into two, if it makes it easier. This method is however preferred as it saves memory from variable allocation. To explain, let us break it down into two steps.

```
In []: x = Arr_df['odd2']

       x


Out[]: A    3

       B    7

       C   11

       D   15

       E   19

       Name: odd2, dtype: int32
```

See that the first operation returns a series containing the element ‘15’. This series can now be indexed to grab 15 using the label ‘D’.

```
In []: x['D']

Out[]: 15
```

While this approach works, and is preferred by beginners, a better approach is to get comfortable with the first method to save coding time and resources.

To grab rows, a different indexing method is used.
You can use either data_frame_name.loc[‘row_name’] or data_frame_name.iloc[‘row_index’] .

Let us grab the row E from Arr_d f .

```
In []: print("using .loc['E']")

   Arr_df.loc['E']


   print('\nusing .iloc[4]')

   Arr_df.iloc[4]
```

```
        using .loc['E']
Out[]:
      odd1     17
      even1    18
      odd2     19
      even2    20
      Name: E, dtype: int32


        using .iloc[4]
Out[]:
      odd1     17
      even1    18
      odd2     19
      even2    20
      Name: E, dtype: int32
```

See, the same result!

You can also use the row indexing method to select single items.

```
In []: Arr_df.loc['E']['even2']
        # or
          Arr_df.iloc[4]['even2']


Out[]: 20


Out[]: 20
```

Moving on, we will try to create new columns in a data frame, and also delete a column.

```
                   In []: # Let us add two sum columns to Arr_df


                                  Arr_df['Odd sum'] = Arr_df['odd1']+Arr_df['odd2']
                                  Arr_df['Even sum'] = Arr_df['even1']+Arr_df['even2']


                    Arr_df
```

Output:

	odd1	even1	odd2	even2	Odd sum	Even sum
A	**1**	**2**	**3**	**4**	**4**	**6**
B	**5**	**6**	**7**	**8**	**12**	**14**
C	**9**	**10**	**11**	**12**	**20**	**22**
D	**13**	**14**	**15**	**16**	**28**	**30**
E	**17**	**18**	**19**	**20**	**36**	**38**

Notice how the new columns are declared. Also, arithmetic operations are possible with each element in the data frame, just like we did with the series.

Exercise: Add an extra column to this data frame. Call it Total Sum, and it should be the addition of Odd sum and Even sum.

To remove a column from a data frame, we use the data_frame_name.drop( ) method.

Let us remove the insert a new column and then remove it using the .drop( ) method.

```
                                        In []: Arr_df['disposable'] = np.zeros(5)   # new
                                         column

   Arr_df   #show
```

Output:

odd1even1odd2even2				Odd sum	Even sum	disposable
A1	**2**	**3**	**4**	**4**	**6**	**0.0**
B5	**6**	**7**	**8**	**12**	**14**	**0.0**
C9	**10**	**11**	**12**	**20**	**22**	**0.0**
D13	**14**	**15**	**16**	**28**	**30**	**0.0**
E17	**18**	**19**	**20**	**36**	**38**	**0.0**

To remove the unwanted column:

```
In []: # to remove
            Arr_df.drop('disposable',axis=1,inplace=True)
            Arr_df
```

Output:

	odd1	**even1**	**odd2**	**even2**	**Odd sum**	**Even sum**
A	**1**	**2**	**3**	**4**	**4**	**6**
B	**5**	**6**	**7**	**8**	**12**	**14**
C	**9**	**10**	**11**	**12**	**20**	**22**
D	**13**	**14**	**15**	**16**	**28**	**30**
E	**17**	**18**	**19**	**20**	**36**	**38**

Notice the ‘axis=1’ and ‘inplace = True’ arguments. These are parameters that specify the location to perform the drop i.e. axis (axis = 0 specifies row operation), and intention to broadcast the drop to the original data frame, respectively. If ‘inplace= False’, the data frame will still contain the dropped column.

Tip: *The ‘inplace = False’ method is used for assigning an array to another variable without including certain columns.*

Conditional selection

Similar to how we conditional selection works with NumPy arrays, we can select elements from a data frame that satisfy a Boolean criterion.

You are expected to be familiar with this method, hence, it will be done in one step.

Example 72: Let us grab sections of the data frame ‘Arr_df’ where the value is > 5.

```
In []: # Grab elements greater than five


Arr_df[Arr_df>5]
```

Output:

	odd1	even1	odd2	even2	Odd sum	Even sum
A	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**	**6**
B	**NaN**	**6.0**	**7.0**	**8.0**	**12.0**	**14**
C	**9.0**	**10.0**	**11.0**	**12.0**	**20.0**	**22**
D	**13.0**	**14.0**	**15.0**	**16.0**	**28.0**	**30**
E	**17.0**	**18.0**	**19.0**	**20.0**	**36.0**	**38**

Notice how the instances of values less than 5 are represented with a ‘NaN’.

Another way to use this conditional formatting is to format based on column specifications.

You could remove entire rows of data, by specifying a Boolean condition based off a single column. Assuming we want to return the Arr_df data frame without the row ‘C’. We can specify a condition to return values where the elements of column ‘odd1’ are not equal to ‘9’ (since row C contains 9 under column ‘odd1’).

```
In []: # removing row C through the first column
      Arr_df[Arr_df['odd1']!= 9]
```

Output:

	odd1	even1	odd2	even2	Odd sum	Even sum
A	**1**	**2**	**3**	**4**	**4**	**6**
B	**5**	**6**	**7**	**8**	**12**	**14**
D	**13**	**14**	**15**	**16**	**28**	**30**
E	**17**	**18**	**19**	**20**	**36**	**38**

Notice that row ‘C’ has been filtered out. This can be achieved through a smart conditional statement through any of the columns.

```
In []:  # does the same thing : remove row ‘C’
       # Arr_df[Arr_df['even2']!= 12]


In[]: # Let us remove rows D and E through 'even2'
       Arr_df[Arr_df['even2']<= 12]
```

Output

	odd1	**even1**	**odd2**	**even2**	**Odd sum**	**Even sum**
A	**1**	**2**	**3**	**4**	**4**	**6**
B	**5**	**6**	**7**	**8**	**12**	**14**
C	**9**	**10**	**11**	**12**	**20**	**22**

Exercise: Remove rows C, D, E via the ‘Even sum’ column. Also, try out other such operations as you may prefer.

To combine conditional selection statements, we can use the ‘logical and, i.e. &’, and the ‘logical or, i.e. |’ for nesting multiple conditions. The regular ‘and’ and ‘or’ operators would not work in this case as they are used for comparing single elements. Here, we will be comparing a series of elements that evaluates to true or false, and those generic operators find such operations ambiguous.

Example 73: Let us select elements that meet the criteria of being greater than 1 in the first column, and less than 22 in the last column. Remember, the ‘and statement’ only evaluates to true if both conditions are true.

```
In []:         Arr_df[(Arr_df['odd1']>1) & (Arr_df['Even sum']<22)]
```

Output:

	odd1	**even1**	**odd2**	**even2**	**Odd sum**	**Even sum**
B	**5**	**6**	**7**	**8**	**12**	**14**

Only the elements in Row ‘B’ meet this criterion, and were returned in the data frame.

This approach can be expounded upon to create even more powerful data frame filters.

Missing Data

There are instances when the data being imported or generated into pandas is incomplete or have missing data points. In such a case, the likely solution is to remove such values from the dataset, or to fill in new values based on some statistical extrapolation techniques. While we would not be fully exploring statistical measures of extrapolation (you can read up on that from any good statistics textbook), we would be considering the use of the .dropna( ) and .fillna( ) methods for removing and filling up missing data points respectively.

To illustrate this, we will create a data frame – to represent imported data with missing values, and then use these two data preparation methods on it.

Example 73: Another way to create a data frame is by using a dictionary. Remember, a python dictionary is somehow similar to a Pandas series in that they have key-value pairs, just as Pandas series are label-value pairs (although this is a simplistic comparison for the sake of conceptualization).

```
In []:   # First, our dictionary
             dico = {'X':[1,2,np.nan],'Y':[4,np.nan,np.nan],'Z':[7,8,9]}
             dico #show

             # passing the dictionary to a dataframe
             row_labels = 'A B C'.split()
             df = pd.DataFrame(dico,row_labels)
             df #show
```

Output:

```
         {'X': [1, 2, nan], 'Y': [4, nan, nan], 'Z': [7, 8, 9]}

         X            Y            Z
A        1.0          4.0          7
B        2.0          NaN          8
```

C	**NaN**	**NaN**	**9**

Now, let us start off with the .dropna( ) method. This removes any ‘null’ or ‘nan’ values in the data frame it’s called off, either column-wise or row-wise, depending on the axis specification and other arguments passed to the method. It has the following default syntax:

```
df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
```

The ‘df’ above is the data frame name. The default axis is set to zero, which represent row-wise operation. Hence, at default, the method will remove any row containing ‘nan’ values.

Let us see what happens when we call this method for our data frame.

```
In []:  # this removes ‘nan’ row-wise
        df.dropna()
```

Output:	**X**	**Y**	**Z**
A	**1.0**	**4.0**	**7**

Notice that rows B and C contain at least a ‘nan’ value. Hence, they were removed.

Let us try a column-wise operation by specifying the axis=1.

```
In []:  # this removes ‘nan’ column-wise
        df.dropna(axis=1)
```

Output:

	Z
A	**7**
B	**8**
C	**9**

As expected, only the column ‘Z’ was returned.

Now, in case we want to set a condition for a minimum number of ‘non-nan’ values/ actual data points required to make the cut (or escape the cut, depending on your perspective), we can use the ‘thresh’ (short for threshold) parameter to specify this.

Say, we want to remove ‘nan’ row-wise, but we only want to remove instances where the row had more than one actual data point value. We can set the threshold to 2 as illustrated in the following code:

```
In []: # drop rows with less than 2 actual values

       df.dropna(thresh = 2)
```

Output:

	X	Y	Z
A	**1.0**	**4.0**	**7**
B	**2.0**	**NaN**	**8**

Notice how we have filtered out row C, since it contains only one actual value ‘9’.

Exercise: Filter out columns in the data frame ‘df’ containing less than 2 actual data points

Next, let us use the .fillna( ) method to replace the missing values with our extrapolations.

This method has the following syntax:

```
                    df.fillna(value=None, method=None, axis=None, inplace=False, limit=None,
                     downcast=None, **kwargs)
```

Tip: *Reminder, you can always use shift + ta b to check the documentation of methods and functions to know their syntax.*

Let us go ahead and replace our ‘NaN’ values with an ‘x’ marker. We can specify the ‘X’ as a string, and pass it into the ‘value’ parameter in .fillna( ) **.**

```
In []: # filling up NaNs
```

```
df.fillna('X')
```

Output:

	X	Y	Z
A	**1**	**4**	**7**
B	**2**	**X**	**8**
C	**X**	**X**	**9**

While marking missing data with an ‘X’ is fun, it is sometimes more intuitive (for lack of a better statistical approach), to use the mean of the affected column as a replacement for the missing elements.

Example 74 : Filling up missing data.

Let us first use the mean method to fill up column ‘X’, then based off that simple step, we will use a for loop to automatically fill up missing data in the data frame.

```
In []: # Replacing missing values with mean in column ‘X’
            df['X'].fillna(value = df['X'].mean())
Out[]: A    1.0
            B    2.0
            C    1.5
            Name: X, dtype: float64
```

Notice that the value of the third element in column ‘X’ has changed to 1.5. This is the mean of that column. The one line code that accomplished this could have been broken down into multiple line for better understanding. This is shown below:

```
In []:    # variables

            xcol_var = df['X']

            xcol_mean = xcol_var.mean()  # or use mean(xcol_var)



            # instruction

            xcol_var.fillna(value = xcol_mean)
```

```
Out[]: A    1.0
       B    2.0
       C    1.5
       Name: X, dtype: float64
```

Same results, but more coding and more memory use via variable allocation.

Now, let us automate the entire process.

```
In []: for i in 'X Y Z'.split():            # loop
     df[i].fillna(value = df[i].mean(),inplace=True)
     df                # show
```

	X	**Y**	**Z**
A	**1.0**	**4.0**	**7**
B	**2.0**	**NaN**	**8**
C	**NaN**	**NaN**	**9**

Output:

	X	**Y**	**Z**
A	**1.0**	**4.0**	**7**
B	**2.0**	**4.0**	**8**
C	**1.5**	**4.0**	**9**

New data frame   Old data frame

While the output only displays the data frame on the left, the original data frame is put here for comparison. Notice the new values replacing the NaNs. For the column ‘Y’, the mean is 4.0, since that is the only value present.

This is a small operation that can be scaled for preparing and formatting larger datasets in Pandas.

Tip: *The other arguments of the .fillna( ) method can be explored, including the fill methods: for example, forward-fill - which fills the missing value with the previous row/column value based on the value of the limit parameter i.e. if limit*

= 1, it fills the next 1 row/column with the previous row/column value; also, the back-fill - which does the same as forward-fill, but backwards.

Group-By

This Pandas method, as the name suggests, allows the grouping of related data to perform combined/aggregate operations on them.

Example 75: Creating a data frame of XYZ store sales.

```
In []: # Company XYZ sales information
    # Dictionary containing needed data
          data = {'Sales Person':'Sam Charlie Amy Vanessa Carl Sarah'.split(),
         'Product':'Hp Hp Apple Apple Dell Dell'.split(),
         'Sales':[200,120,340,124,243,350]}
          print('XYZ sales information\n________________')  # print info.

          serial = list(range(1,7))        # row names from 1-6
          df = pd.DataFrame(data,serial)   # build data frame
          df
```

Output:

XYZ sales information

____

	Sales Person	Product	Sales
1	**Sam**	**Hp**	**200**
2	**Charlie**	**Hp**	**120**
3	**Amy**	**Apple**	**340**
4	**Vanessa**	**Apple**	**124**
5	**Carl**	**Dell**	**243**
6	**Sarah**	**Dell**	**350**

From our dataset, we can observe some common items under the product column. This is an example of an entry point for the group-by method in a data set. We can find information about the sales using the product grouping.

```
In []: # finding sales information by product


          print('Total items sold: by product')
          df.groupby('Product').sum()
```

Total items sold: by product

	Sales
Product	
Apple	**464**
Dell	**593**
Hp	**320**

This is an example of an aggregate operation using groupby. Other functions can be called to display interesting results as well. For example, .count( ) :

```
In []:  df.groupby('Product').count()
```

Output:

Sales Person	Sales	
Product		
Apple	**2**	**2**
Dell	**2**	**2**
Hp	**2**	**2**

While the previous operation could not return the ‘Sales person’ field, since a numeric operation like ‘sum’ cannot be performed on a string, the count method returns the instances of each product within both categories. Via this output, we can easily infer that XYZ company assigns two salespersons per product, and that each of the sales persons made a sale of the products. However, unlike the sum method, this count method does not give a clearer overview of the sales. This is why so many methods are usually called to explain certain aspects of data. A very useful method for checking multiple information at a time is the .describe() method.

```
In []:  #Getting better info using describe ()
         df.groupby('Product').describe()
```

Output:

	Sales							
	count	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
Product								
Apple	**2.0**	**232.0**	**152.735065**	**124.0**	**178.00**	**232.0**	**286.00**	**340.0**
Dell	**2.0**	**296.5**	**75.660426**	**243.0**	**269.75**	**296.5**	**323.25**	**350.0**
Hp	**2.0**	**160.0**	**56.568542**	**120.0**	**140.00**	**160.0**	**180.00**	**200.0**

Now, this is more informative. It says a lot about the data at a glance. Individual products can also be selected: `df.groupby(‘Product’).describe()[‘Product name e.g. ‘Apple’ ]` .

Concatenate, Join and Merge

These are methods for combining multiple data frames, or data sets into a single one. They differ in syntax, and achieve specific combinations of data frames based on the intended output.

Concatenation allows datasets to be ‘glued’ together, either row-wise or column-wise. Here, the dimensions of the data frames must be the same along the axis of concatenation, i.e. row-wise concatenation requires the two data frames to have the same number of columns, and vice versa.

Example 76: Let us create two example data frames and use the concatenate method.

```
In []:  # Defining a dictionary of values
            d1 = {'A':'A1 A2 A3'.split(),'B':'B1 B2 B3'.split(),
              'C':'C1 C2 C3'.split()}
            d2 = {'A':'A4 A5 A6'.split(),'B':'B4 B5 B6'.split(),
              'C':'C4 C5 C6'.split()}

            # Now the data frames
            df1 = pd.DataFrame(d1,index= '1 2 3'.split())
            df2 = pd.DataFrame(d2,index= '4 5 6'.split())

            # concatenating
            pd.concat([df1,df2])    # row-wise axis = 0
            pd.concat([df1,df2],axis=1,sort=True) # col-wise axis = 1
Out[]:
```

	A	B	C
1	**A1**	**B1**	**C1**
2	**A2**	**B2**	**C2**
3	**A3**	**B3**	**C3**
4	**A4**	**B4**	**C4**
5	**A5**	**B5**	**C5**
6	**A6**	**B6**	**C6**

Out[]:

	A	B	C	A	B	C
1	**A1**	**B1**	**C1**	**NaN**	**NaN**	**NaN**
2	**A2**	**B2**	**C2**	**NaN**	**NaN**	**NaN**
3	**A3**	**B3**	**C3**	**NaN**	**NaN**	**NaN**

4	**NaN**	**NaN**	**NaN**	**A4**	**B4**	**C4**
5	**NaN**	**NaN**	**NaN**	**A5**	**B5**	**C5**
6	**NaN**	**NaN**	**NaN**	**A6**	**B6**	**C6**

Notice how the result of the row-wise concatenation is just like placing the first data frame on top of the other. The attempt at column concatenation for these data frames resulted in NaNs because they have varying indices. While df1 has its index from 1-3, with corresponding values; df2 has its own index ranging from 4-6. To allow for proper column concatenation, we have to specify a similar index as below.

```
In []: df2.index = '1 2 3'.split() # setting same index as df1


            # This should then work
            pd.concat([df1,df2],axis=1) # col-wise axis = 1
```

```
Out[21]:
```

	A	**B**	**C**	**A**	**B**	**C**
1	**A1**	**B1**	**C1**	**A4**	**B4**	**C4**
2	**A2**	**B2**	**C2**	**A5**	**B5**	**C5**
3	**A3**	**B3**	**C3**	**A6**	**B6**	**C6**

Now, this is a nicely combined data set. This is more likely the kind of concatenation you would encounter while working with data sheets, as it allows for a more flexible naming for the columns, while sharing similar row-indices.

Exercise: Change the last three column names in the above output table to ‘D E F’, i.e. the table should now have columns ‘A B C D E F’.

Hint: *use df2.columns = ‘value ’ to change the column names.*

Merge and Join on the other hand, are more similar. They take two data frames and combine them together column-wise. They both require a specification for a right and left data frame respectively to determine the organization. Also, they require an extra specification for an entry point (usually called a key) where the join or merge operation occurs. While the Merge and Join methods are powerful, we will not be exploring them.

Reading and Writing data

In real-world applications, data comes in various formats. These are the most common ones: CSV, Excel spreadsheets (xlsx / xls), HTML and SQL. While Pandas can read SQL files, it is not necessarily the best for working with SQL databases, since there are quite a few SQL engines: SQL lite, PostgreSQL, MySQL, etc. Hence, we will only be considering CSV, Excel and HTML.

Read

The pd.read_file_type(‘file_name’) method is the default way to read files into the Pandas framework. After import, pandas displays the content as a data frame for manipulation using all the methods we have practiced so far, and more.

CSV (comma separated variables) & Excel

Create a CSV file in excel and save it in your python directory. You can check where your python directory is in Jupyter notebook by typing: pwd() . If you want to change to another directory containing your files (e.g. Desktop), you can use the following code:

```
In []: import os
            os.chdir('C:\\Users\\Username\\Desktop')
```

To import your CSV file, type: pd.read_csv(‘csv_file_name’). Pandas will automatically detect the data stored in the file and display it as a data frame. A better approach would be to assign the imported data to a variable like this:

```
In []:   Csv_data = pd.read_csv(‘example file.csv’)
         Csv_data          # show
```

Running this cell will assign the data in ‘example file.csv’ to the variable Csv_data, which is of the type data frame. Now it can be called later or used for performing some of the data frame operations.

For excel files (.xlsx and .xls files), the same approach is taken. To read an excel file named ‘class data.xlsx’, we use the following code:

```
In []:     Xl_data = pd.read_excel(‘class data.xlsx’)
               Xl_data          # show
```

This returns a data frame of the required values. You may notice that an index starting from 0 is automatically assigned at the left side. This is similar to declaring a data frame without explicitly including the index field. You can add index names, like we did in previous examples.

Tip: in case the excel spreadsheet has multiple sheets filled. You can specify the sheet you need to be imported. Say we need only sheet 1, we use: sheetname = ‘Sheet 1’ . For extra functionality, you may check the documentation for read_excel() by using shift+tab .

Write

After working with our imported or pandas-built data frames, we can write the resulting data frame back into various formats. We will, however, only consider writing back to CSV and excel. To write a data frame to CSV, use the following syntax:

```
In []:     Csv_data.to_csv(‘file name ’,index = False)
```

This writes the data frame ‘Csv_data’ to a CSV file with the specified filename in the python directory. If the file does not exist, it creates it.

Tip: *You can also use this method to create spreadsheet files via Python. The False index parameter is to ensure that the automatic indexing of Jupyter is not written to the file, thereby messing up its formatting.*

For writing to an excel file, a similar syntax is used, but with sheet name specified for the data frame being exported.

```
In []:   Xl_data.to_excel(‘file name.xlsx’,sheet_name = ‘Sheet 1’)
```

This writes the data frame Xl_data to sheet one of ‘file name.xlsx’ .

Html

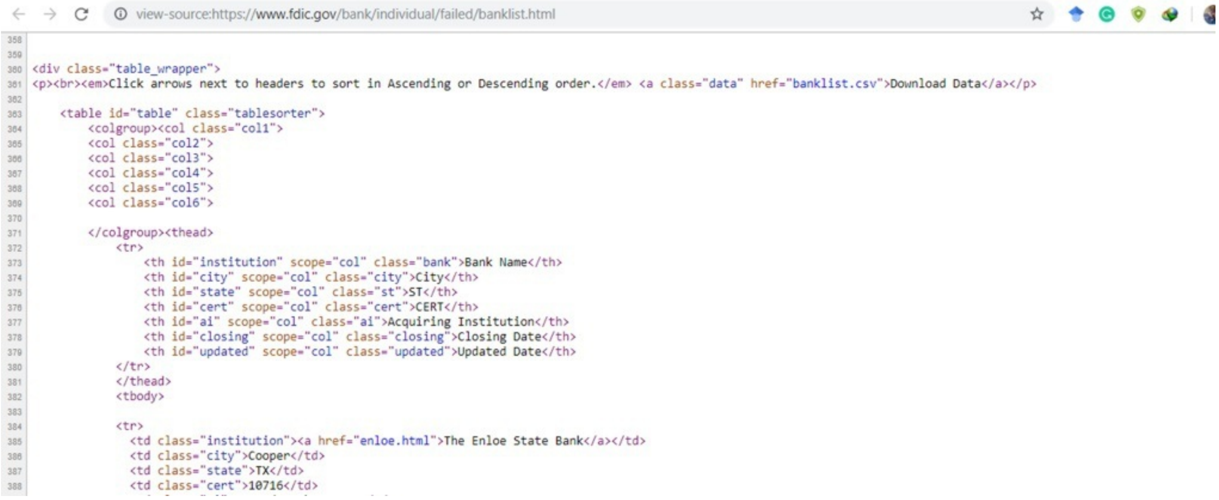
Reading Html files through pandas requires a few libraries to be installed: htmllib5, lxml, and BeautifulSoup4. Since we installed the latest Anaconda, these libraries are likely to be included. Use conda list to verify, and conda instal l to install any missing ones.

Html tables can be directly read into pandas using the pd.read_html (‘sheet url’) method. The sheet url is a web link to the data set to be imported. As an example, let us import the ‘Failed bank lists’ dataset from FDIC’s website and call it w_data.

```
In []: w_data = pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')


 w_data[0]
```

To display the result, here we used w_data [0] . This is because the table we need is the first sheet element in the webpage source code. If you are familiar with HTML, you can easily identify where each element lies. To inspect a web page source code, use Chrome browser. On the web page >> right click >> then, select ‘view page source ’ . Since what we are looking for is a table-like data, it will be specified like that in the source code. For example, here is where the data set is created in the FDIC page source code:

FDIC page source via chrome



view-source:https://www.fdic.gov/bank/individual/failed/banklist.html

```


<div class="table_wrapper">
<p><br><em>Click arrows next to headers to sort in Ascending or Descending order.</em> <a class="data" href="banklist.csv">Download Data</a></p>

    <table id="table" class="tablesorter">
        <colgroup><col class="col1">
        <col class="col2">
        <col class="col3">
        <col class="col4">
        <col class="col5">
        <col class="col6">

        </colgroup><thead>
            <tr>
                <th id="institution" scope="col" class="bank">Bank Name</th>
                <th id="city" scope="col" class="city">City</th>
                <th id="state" scope="col" class="st">ST</th>
                <th id="cert" scope="col" class="cert">CERT</th>
                <th id="ai" scope="col" class="ai">Acquiring Institution</th>
                <th id="closing" scope="col" class="closing">Closing Date</th>
                <th id="updated" scope="col" class="updated">Updated Date</th>
            </tr>
            </thead>
            <tbody>

            <tr>
              <td class="institution"><a href="enloe.html">The Enloe State Bank</a></td>
              <td class="city">Cooper</td>
              <td class="state">TX</td>
              <td class="cert">10716</td>
```

This section concludes our lessons on the Pandas framework. To test your knowledge on all that has been introduced, ensure to attempt all the exercises below. In the next chapter, we will be exploring some data visualization frameworks.

For the exercise, we will be working on an example dataset. A salary spreadsheet from Kaggle.com. Go ahead and download the spreadsheet from this link: www.kaggle.com/kaggle/sf-salaries

Note: *You might be required to register before downloading the file.* Download the file to your python directory and extract the file.

Exercises: We will be applying all we have learned here.

1. Import pandas as pd
2. Import the CSV file into Jupyter notebook, assign it to a variable 'Sal', and display the first 5 values.

Hint: *use the .head( ) method to display the first 5 values of a data frame. Likewise, .tail( ) is used for displaying the last 5 results. To specify more values, pass **'n=value'** into the method.*

3. What is the highest pay (including benefits)? ***Answer: 567595.43***

Hint: *Use data frame column indexing and .max( ) method.*

4. According to the data, what is 'MONICA FIELDS's Job title, and how much does she make plus benefits? ***Answer: Deputy Chief of the Fire Department, and $ 261,366.14*** .

Hint: *Data frame column selection and conditional selection works (conditional selection can be found under Example 72. Use column index ==’string’ for the Boolean condition).*

5. Finally, who earns the highest basic salary (minus benefits), and by how much is their salary higher than the average basic salary. ***Answer: NATHANIEL FORD earns the highest. His salary is higher than the average by $ 492827.1080282971.***

Hint: *Use the .max( ) , and .mean( ) methods for the pay gap. Conditional selection with column indexing also works for the employee name with the highest pay.*

Best of luck.

Chapter 3

Data Visualization with Python

Data visualization can be described as the various ways by which analyzed data i.e. information is displayed. Sometimes, even well-analyzed data is not informative enough at a glance. With data visualization, which includes line graphs, bar charts, pictograms, etc. the results/ analysis being presented become less abstract to the end-user, and decision-making is enhanced. In this chapter, we will be learning various techniques for displaying the results of our analysis with NumPy and Pandas frameworks.

Matplotlib

This is a python library for producing high-quality 2D plots. For those that have some MATLAB experience, the plotting techniques and visualizations here will seem familiar. Matplotlib offers a lot of flexibility with plots, in terms of control over things like the axes, fonts, line styles and size, etc. However, all these require writing extra lines of code. So, if you do not mind going the extra mile (with typing code) to fully specify your plots, then matplotlib is your friend. For extra information about this package, visit the official page at www.matplotlib.org

There are basically two approaches to plotting data in matplotlib: The Functional and the object-oriented (OO) approach, respectively. You might encounter the two terms consistently as you interact with programmers and other programming languages, but they are just two slightly different approaches to programming. We will only be considering the functional approach here, as it is easy to understand for beginners and also requires writing fewer lines of code. The OO method, however, offers more control over the plots at the consequence of writing more lines of code.

To start off, let us create a simple cosine plot using the functional approach.

First, let us import the relevant libraries and create plot data:

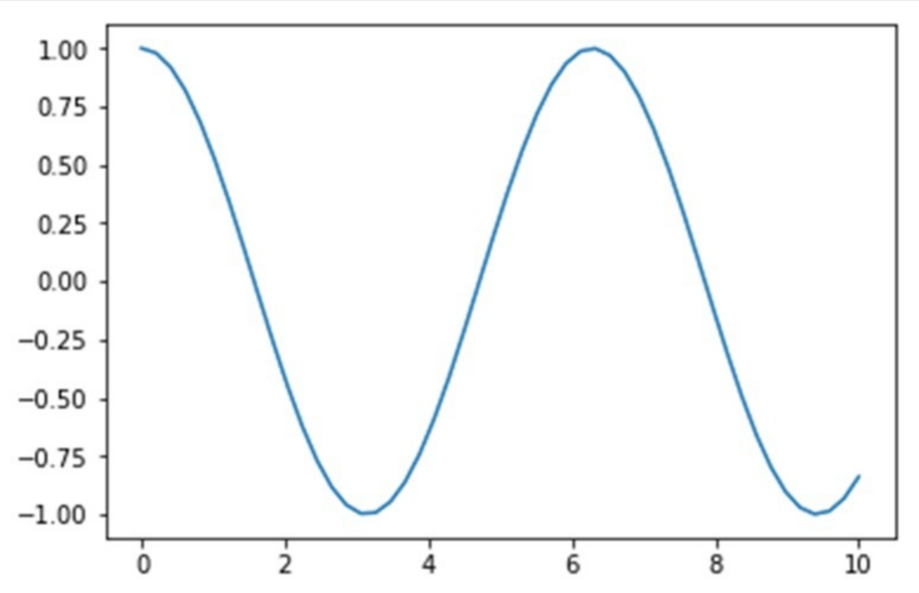
```
In []: import matplotlib.pyplot as plt
```

```
import numpy as np
%matplotlib inline
# creating plot values
x = np.linspace(0,10)  # x-axis/time-scale
y = np.cos(x)    # corresponding cosine values
```

The %matplotlib inline option in the code ensures all our plots are displayed as we run each cell. If you are running a different python console, you can put plt.show( ) at the end of your code to display your plots. plt.show( ) is the print( ) function equivalent for matplotlib plots.

Functional method

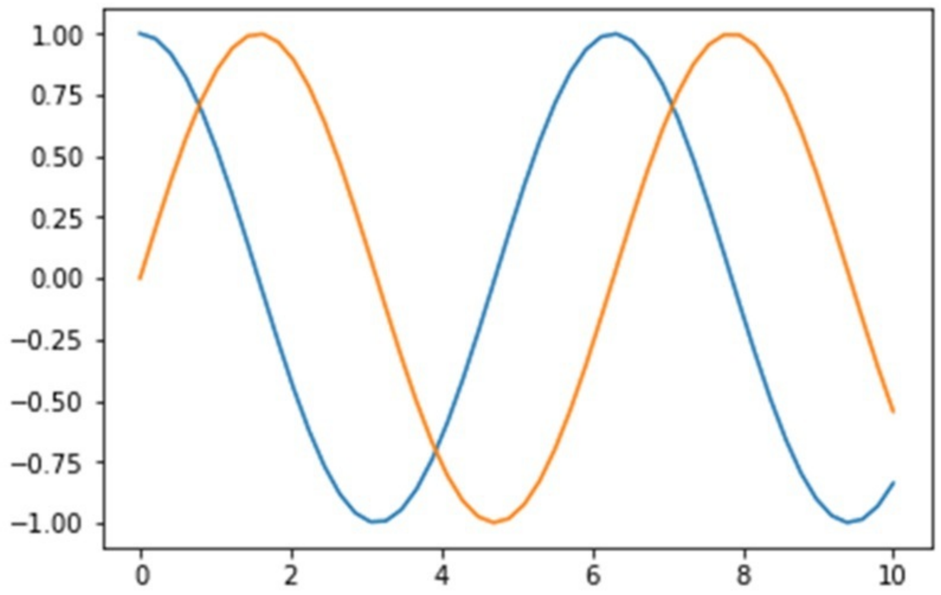
```
In []: # functional plot
                                   plt.plot(x,y)


Out[]: [<matplotlib.lines.Line2D at 0x25108cf27f0>]
```



Notice that we get an Out[ ] statement. This is because we did not print the result using plt.show( ) . While this is not significant if you are using Jupyter, it might be required for other consoles.

We can also plot multiple functions in one graph.

```
                  In []:        z = np.sin(x)  # Adding an extra plot variable
                                 plt.plot(x,y,x,z);plt.show()
```

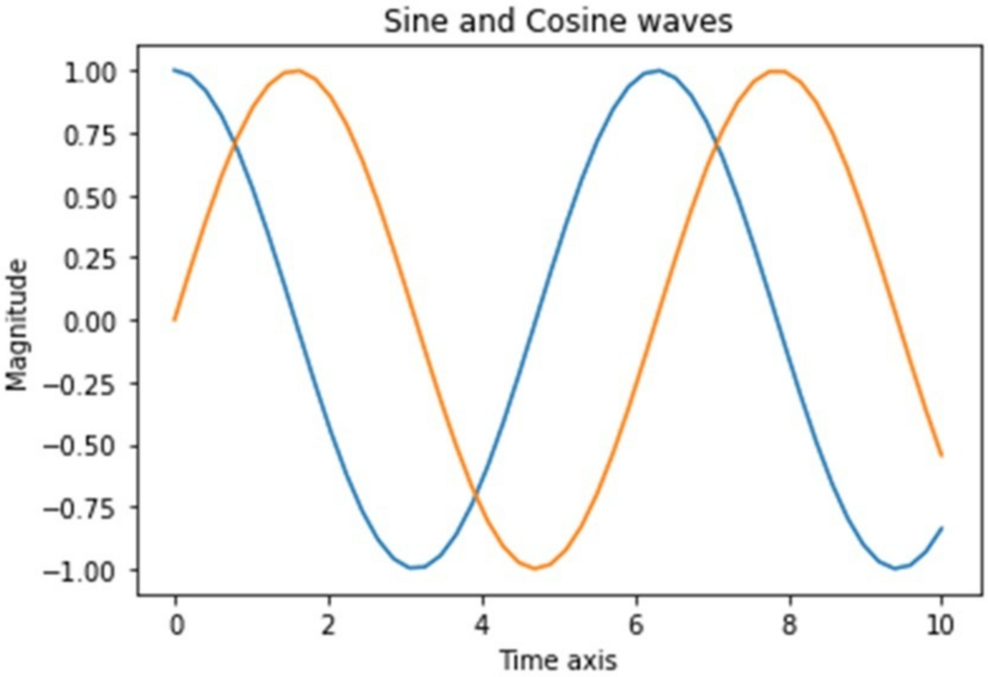


To print multiple graphs, just pass each argument to the plot statement, and separate the plots with commas.

To make our graphs more meaningful, we can label the axes and give the graph a title.

```
In []: plt.plot(x,y,x,z)
 plt.xlabel('Time axis')  # labelling x-axis
 plt.ylabel('Magnitude')  # labelling y-axis
 plt.title('Sine and Cosine waves')  # graph title
 plt.show()  # printing
```

Output:

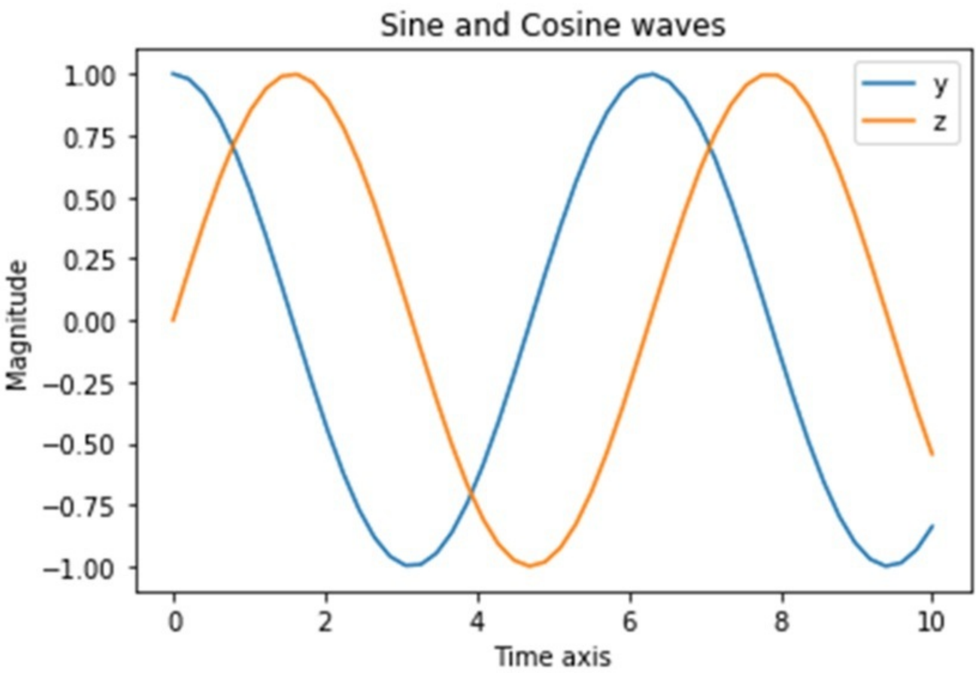


Now, this is a better figure. Since we added more than one plot, there is an extra functionality called ‘legend’ which helps differentiate between the plots.

The legend function takes in two arguments. The first is usually a string argument for labeling the graphs in order. The second is for extra functionality, like where the legend should be. The location argument is specified using ‘loc=value’. For value = 1, upper right corner; 2, for upper left corner; 3, for lower left corner; and 4 for the lower right corner. However, if you would rather let matplotlib decide the best location, use ‘value=0’.

```
In []: plt.plot(x,y,x,z)
       plt.xlabel('Time axis')
       plt.ylabel('Magnitude')
       plt.title('Sine and Cosine waves')
                                                  plt.legend(['y','z'],loc=0)    #
                                          loc=0 means best location
       plt.show()
```

Output:



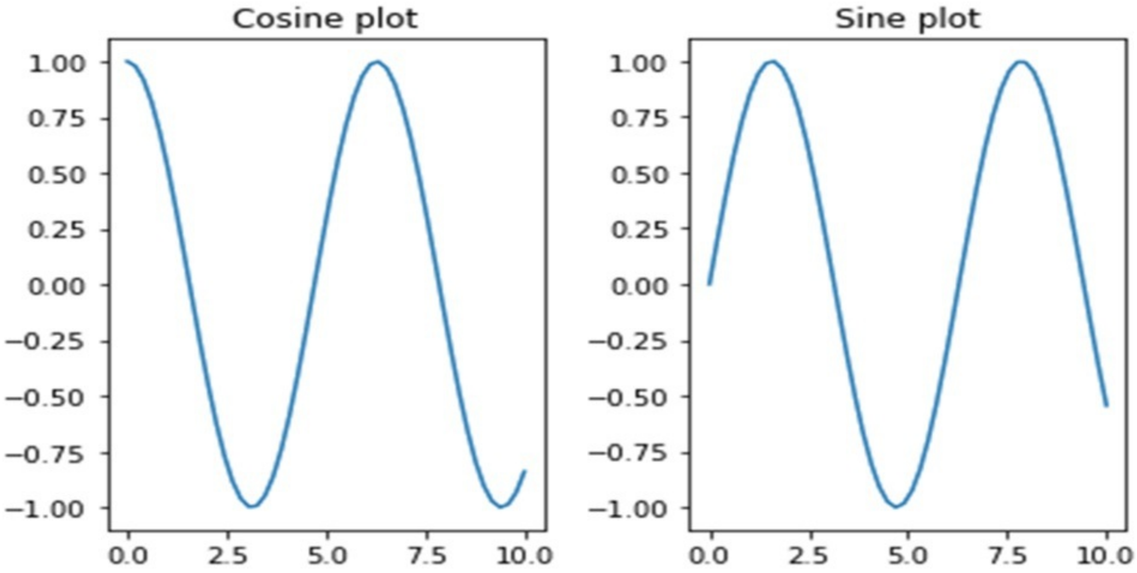
Assuming we wish to plot both the cosine and sine wave above, but side by side.

We can use the subplot command to do this. Think of the subplot as an array of figures with a specification of the number of rows and columns. So, if we want just two graphs beside each otherhat can be considered as a 1 row, and 2 columns array.

```
In []:  # subplotting
           plt.subplot(1,2,1)    # plot 1
           plt.plot(x,y)
           plt.title('Cosine plot')

           plt.subplot(1,2,2)    # plot 2
           plt.plot(x,z)
           plt.title('Sine plot')

           plt.tight_layout()   # avoid overallping plots
           plt.show()
```

Tip: *The tight_layout() line ensures that all subplots are well spaced. Always use this when sub-plotting to make your plots nicer. Try removing that line and compare the output!*

Output:



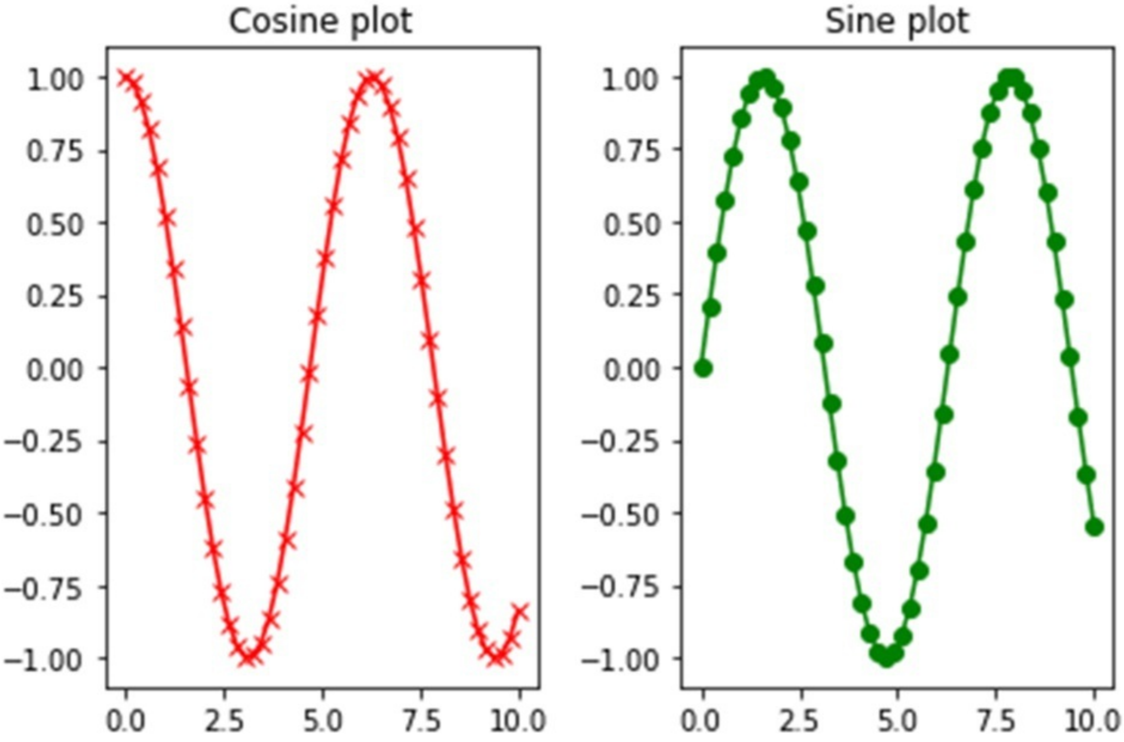
To explain the subplot line i.e. subplot(1,2,1) and subplot(1,2,2): The first two values are the number of rows and columns of the subplot. As seen in the above result, the plot is on one row, and two columns. The last value specifies the order of the plot; hence (1,2,1) translates to plotting that figure in the first row, and the first column out of two.

We can specify the line colors in our plots, as well as the line style. This is passed as a string after the plot argument. Since all the plot options, including marker style and the likes are exactly same as for Matlab, here is a link to Matlab plot documentation to explore all the extra customization feature you can port to your matplotlib plots: https://www.mathworks.com/help/matlab/ref/plot.html#btzpndl-1

Let us change the color and fonts in our subplots to illustrate this.

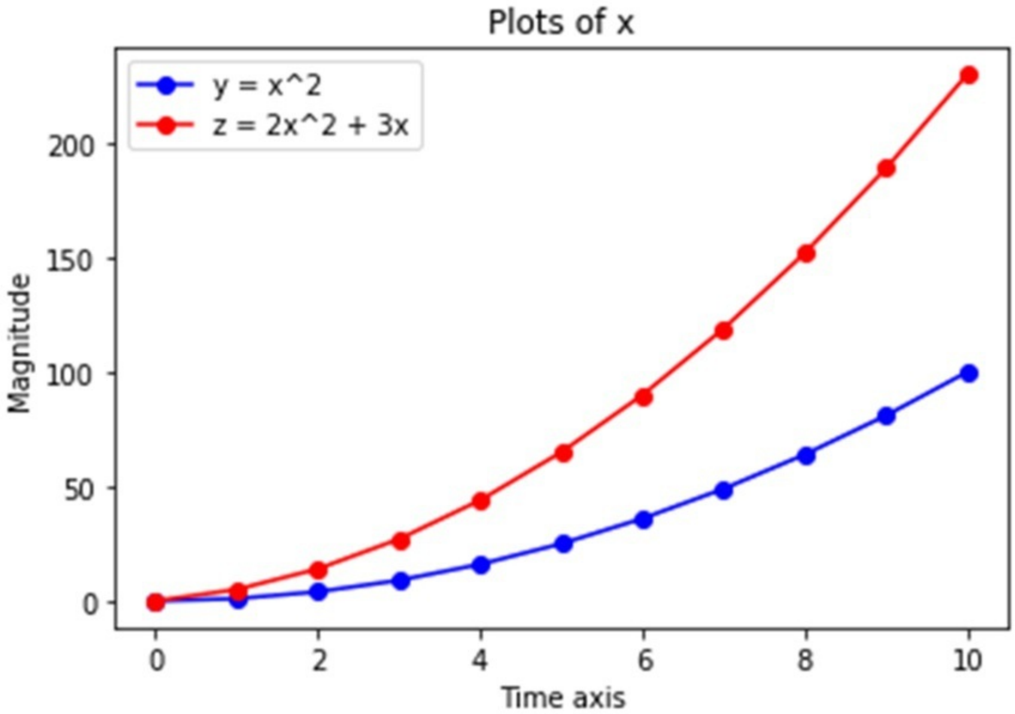
```
In []: plt.subplot(1,2,1)    # plot 1
 plt.plot(x,y,'r-x')   # red plot with -x marker
 plt.title('Cosine plot')


        plt.subplot(1,2,2)    # plot 2
 plt.plot(x,z,'g-o')   # green plot with -o marker
 plt.title('Sine plot')


 plt.tight_layout()   # still avoiding overallping plots
plt.show()
```

Output:



Exercise: Now that you have learned how to plot using the functional approach, test your skills.

Plot this:



Tip: *use* np.arange(0,10,11) *for the x-axis.*

After creating your plot, you may need to import it into your documents or just save it on your device. To do this, you can right-click the image in your Jupyter notebook and click copy. Then, you can paste the copied image into your document.

If you prefer to save, you can use the plt.savefig('figurename.extension', DPI = value) method. Here, figurename is the desired name for your saved image; The extension is the desired format i.e. PNG, JPG, BMP, etc. Finally, the DPI specifies the quality of the image, the higher the better. Usually for standard printing quality, about 300 is good enough.

Tip: *Learning a bit more about these specifications can really help you generate better images from your plots.*

Seaborn

This is another data visualization library that extends the graphical range of the matplotlib library. A lot of methods from matplotlib are applicable here, for customizing plots. However, it generates high quality, dynamic plots in fewer lines of code.

Seaborn is more optimized for plotting trends in datasets, and we are going to explore a dataset using this library. Since, Seaborn is pre-loaded with a few datasets (it can call and load certain datasets from its online repository), we will just load up one of these for our example.

Example 77 : Loading a seaborn dataset and plotting trends

As with other packages, we have to import seaborn using the standard variable name ‘sns’.

```
In []: # importing seaborn
            import seaborn as sns
            %matplotlib inline
```

Next, we will load the popular ‘tips’ dataset from Seaborn. This dataset contains information about a restaurant, the tips given to the waiters, amount of tip, size of the customer group (e.g. a party of 3 people) etc.

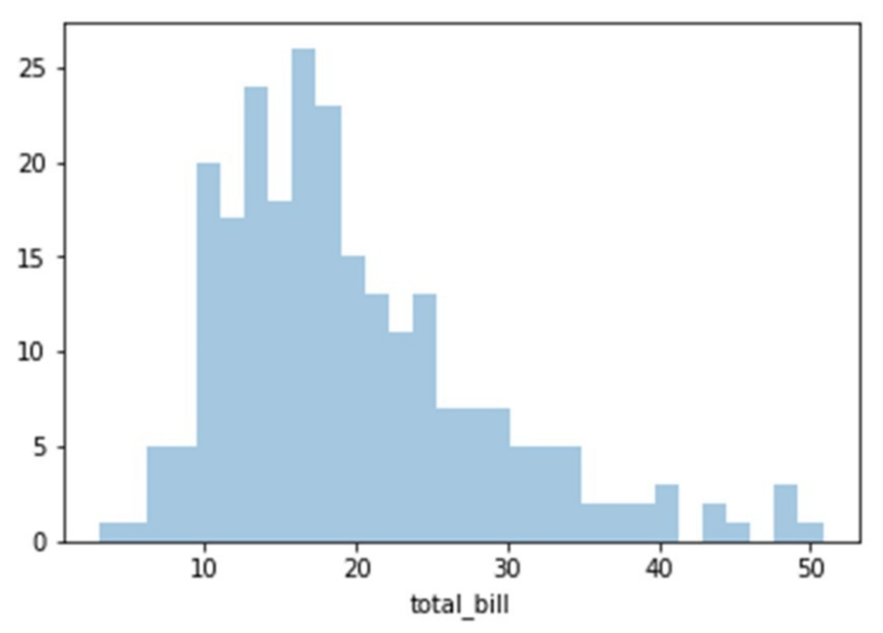
```
In []:   # loading a dataset from seaborn
            tips_dataset = sns.load_dataset('tips')
            tips_dataset.head()


Out[]:
```

	total_bill	tip	sex	smoker	day	time	size
0	**16.99**	**1.01**	**Female**	**No**	**Sun**	**Dinner**	**2**
1	**10.34**	**1.66**	**Male**	**No**	**Sun**	**Dinner**	**3**
2	**21.01**	**3.50**	**Male**	**No**	**Sun**	**Dinner**	**3**
3	**23.68**	**3.31**	**Male**	**No**	**Sun**	**Dinner**	**2**
4	**24.59**	**3.61**	**Female**	**No**	**Sun**	**Dinner**	**4**

Next, we can find trends in the data using different kinds of plots. Let us use the dist_plot (distribution plot) to observe how the total_bill is distributed across the dataset.

```
In []:   #Use distplot
            sns.distplot(tips_dataset['total_bill'],bins = 30,kde=False)
```

```
Out[]:<matplotlib.axes._subplots.AxesSubplot at 0x28d3867d6d8>
```

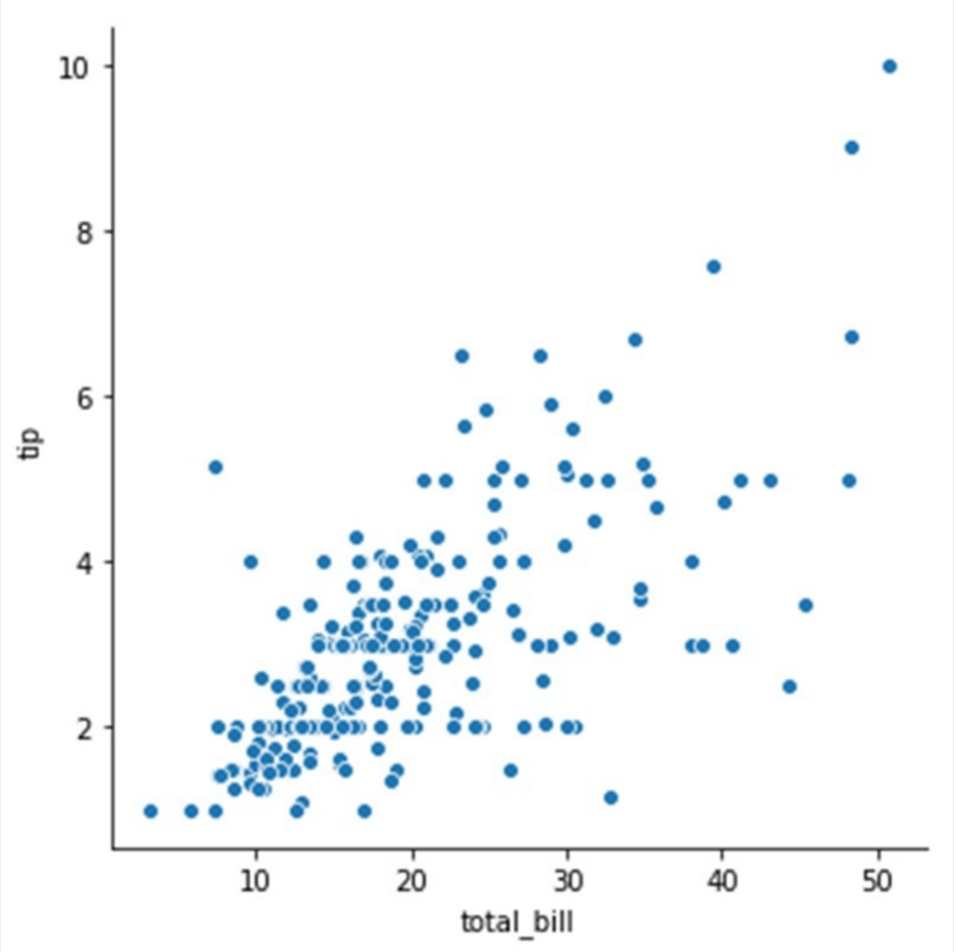


The distplot shows the distribution of certain trends within data. According to the plot above, we can infer that most of the bills are within the range of $10 - $20, since they have the tallest bins within the distribution. You may notice some extra arguments in the distplot code, the bins argument controls the number of histograms that are shown within the population. The higher the value, the more histograms. Although, sometimes higher can make the data less obvious to read, so finding a balance is important. The next argument is the ‘kde’, which means kernel density estimate. It is sometimes preferred over histograms, or along with histograms for a more accurate interpretation of the data. It is mostly an estimate of the probability density function of any variable within the distribution, more like the histogram but smoother. You can read up for a more thorough statistical background on some of these things.

Another useful plot is the relplot, which shows the relationship between two variables within a data_set. It is good for comparison, and can sort results based off categories i.e. sex, age, etc. within the data.

Let us demonstrate how the total_bill relates with the estimated tip, and sort by category male/female.

```
In []: # estimating tip with respect to total_bill

        sns.relplot(x ="total_bill", y="tip", data = tips_dataset)


out []:
```



Here is a basic plot without the category argument. This tells us that the tips generally increase with respect to the total bill. The higher bills correspond to higher tips, and lower bills to lower tips. Adding categories makes the data more interesting as we can see the category that tips more or less.

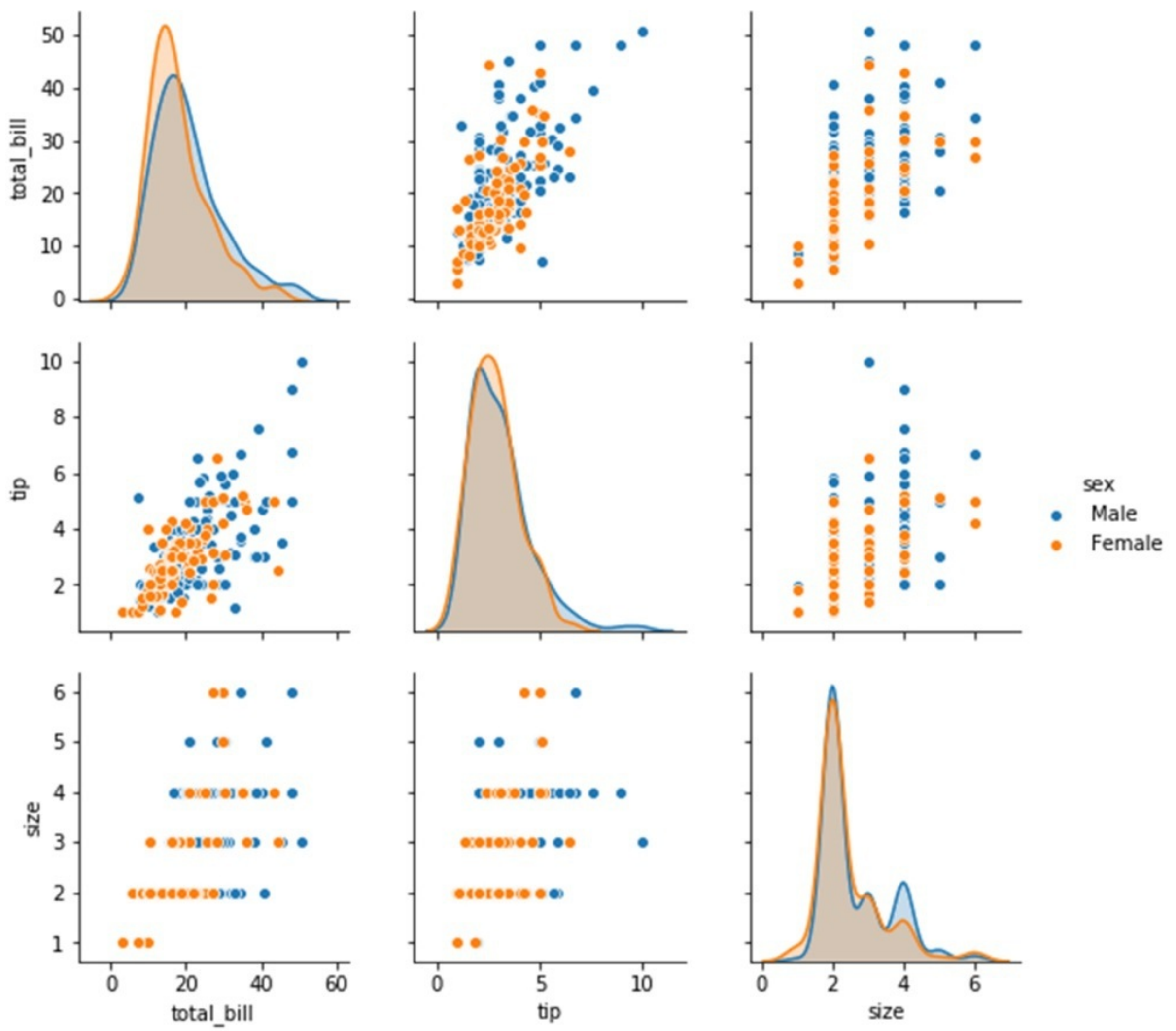
```
In []: sns.relplot(x ="total_bill", y="tip", data = tips_dataset, hue = 'sex')


Out[]: <seaborn.axisgrid.FacetGrid at 0x28d3b9cedd8>
```

See how this is a more informative data. This tells how male customers tipped higher per total_bill on average than the females.

Now this idea can be extended and applied to more advanced data. You can further explore various plotting options with relplot via this link: www.seaborn.pydata.org/tutorial/relational.html

The relplot is even extended with the pairplot option, which relates everything in a data set in one plot. It is a great way to get a quick overview of important trends within your data.

```
In []: sns.pairplot(tips_dataset,hue = 'sex', palette = 'coolwarm')


Out[]:<seaborn.axisgrid.PairGrid at 0x28d412f2e48>
```

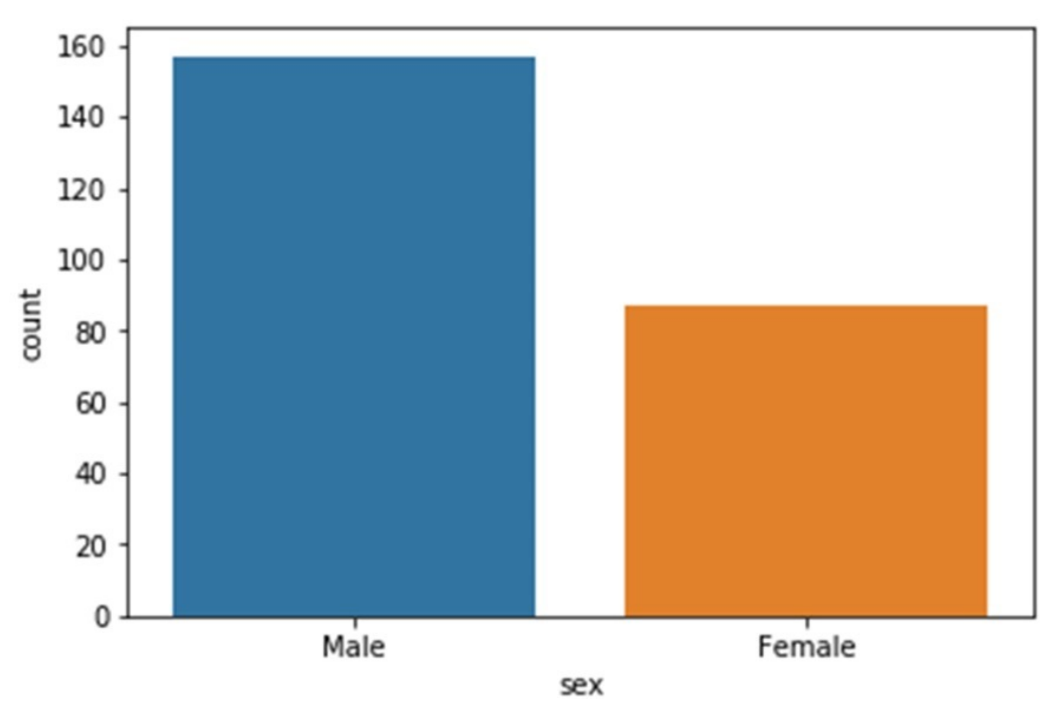


See how easy it is to observe the variations between the three main parameters within the data: total_bill, tip and size. The category ‘sex’ has also been passed to observe the trends in that wise. For each instance where a variable is being compared with itself, we get a kernel density estimate, or a histogram if specified. The other comparisons are made via scatterplots.

From the pairplot, we can quickly infer that the tips do not necessarily increase with increasing party size, considering that the largest tip is within the party size of 3. This inference can be found by observing graphs 2,3, and 3,2 (row,column).

You may wonder, could we also find the population size within a dataset by category? Well, countplot is very useful for that. It is common to see such kinds of plots within a document like the US census report. It basically displays a bar chart, with the height corresponding to the population of a category within the dataset.

```
In []:    sns.countplot(x ='sex', data = tips_dataset)

Out[]: <matplotlib.axes._subplots.AxesSubplot at 0x28d41990978>
```



To validate this, we can use the pandas groupby method along with count. Hope you can recall these methods!

```
In []: import pandas as pd   #importing pandas to use groupby()

            tips_dataset.groupby('sex').count()


Out[]:
```

	total_bill	tip	smoker	day	time	size
sex						
Male	**157**	**157**	**157**	**157**	**157**	**157**
Female	**87**	**87**	**87**	**87**	**87**	**87**

As expected, notice how the male and female counts of 157 and 87 respectively correspond with the countplot above.

These and many more are data visualization capabilities of Seaborn. For now, these are some basic examples to get you started; you may visit the seaborn official documentation gallery to explore more plots styles and options via this link: https://seaborn.pydata.org/examples/index.html

Pandas

Well, it's our friendly pandas again. The library also has some highly functional visualization capabilities. It is quite intuitive at the time to use these built-in visualization options while working with pandas, unless something more specialized is required.

First, we import a few familiar libraries:

```
In []: # importing all necessary libraries

        import numpy as np

        import pandas as pd

        import matplotlib.pyplot as plt

        %matplotlib inline

        import seaborn as sns
```

You may wonder why all the other libraries apart from Pandas are imported. Well, your outputs will look much better with these libraries synchronized. Pandas plots using the matplotlib library functionality -- even though it doesn’t directly call it, and the seaborn library makes the graphs/plots look better.

Let us work with a different dataset. We can create our own data frame and call plots off it.

We will create a data frame from a uniform distribution.

```
In []: # let us create our dictionary

        d = {'A':np.random.rand(5),

       'B':np.random.rand(5),

       'C':np.random.rand(5),

       'D':np.random.rand(5)}



        # now creating a data frame
```

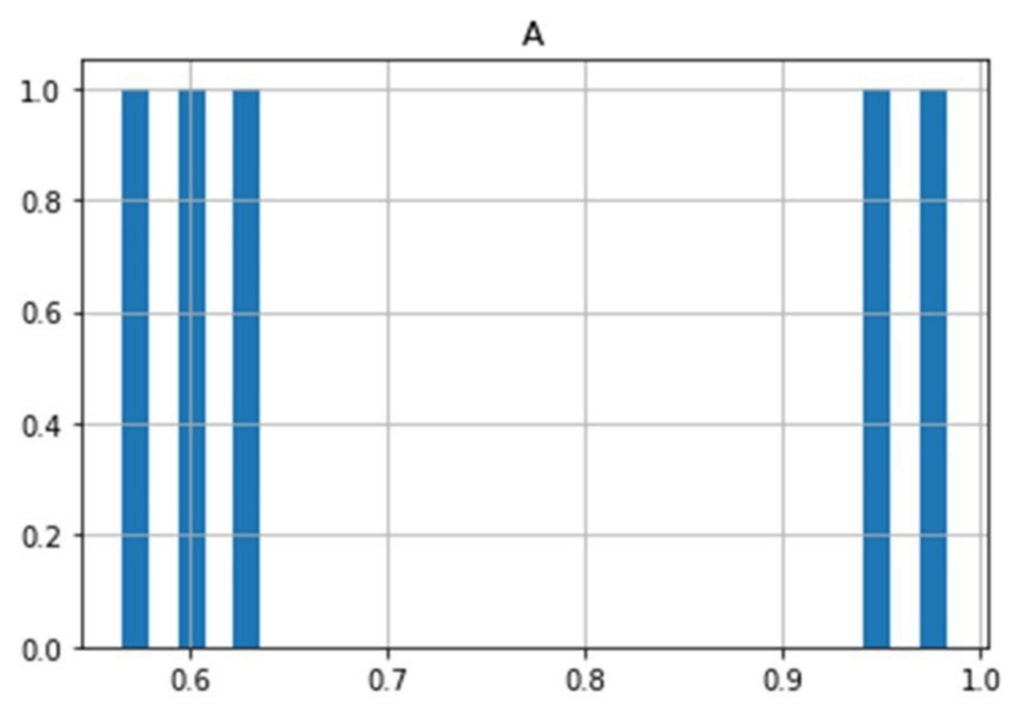
```
df = pd.DataFrame(d)
df
```

```
Out[]:
```

	A	B	C	D
0	**0.982520**	**0.469717**	**0.973735**	**0.397019**
1	**0.602272**	**0.148608**	**0.433559**	**0.929647**
2	**0.566168**	**0.737165**	**0.040840**	**0.435978**
3	**0.632309**	**0.772419**	**0.341389**	**0.603980**
4	**0.949631**	**0.906318**	**0.895018**	**0.679825**

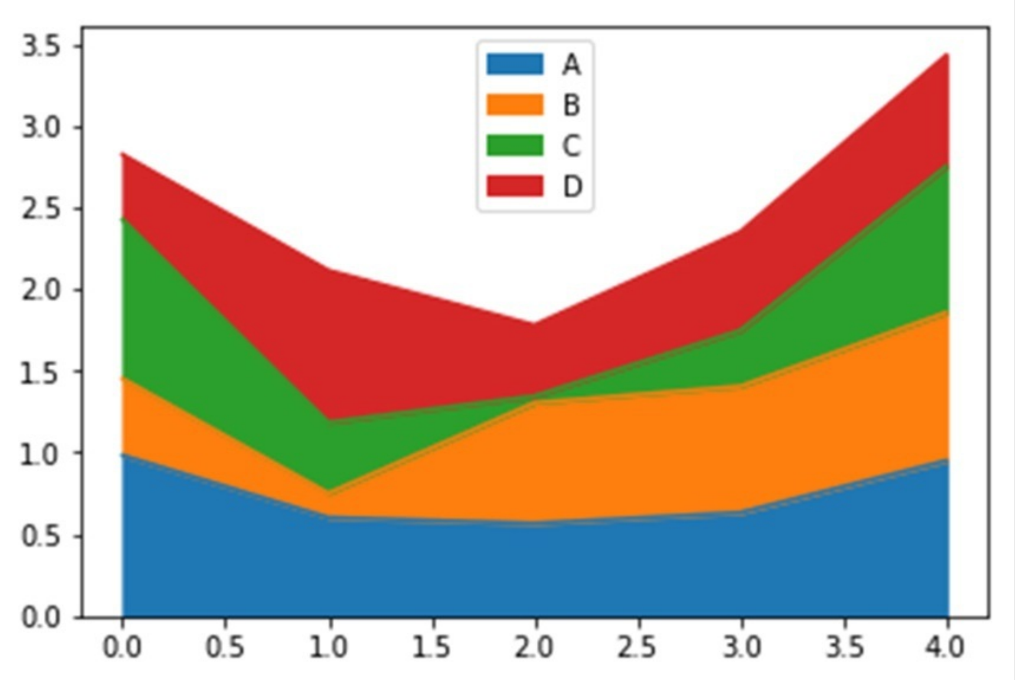
With our data frame, we can now observe trends. To create a histogram plot using pandas, use the hist() function. Also, you can pass some matplotlib arguments like ‘bins’

```
In []: df[['A']].hist(bins=30)
Out[]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0000028D432E50B8>]],
      dtype=object)
```



We can do an area plot of the values as well, which is essentially a line graph of the values with the area underneath shaded:

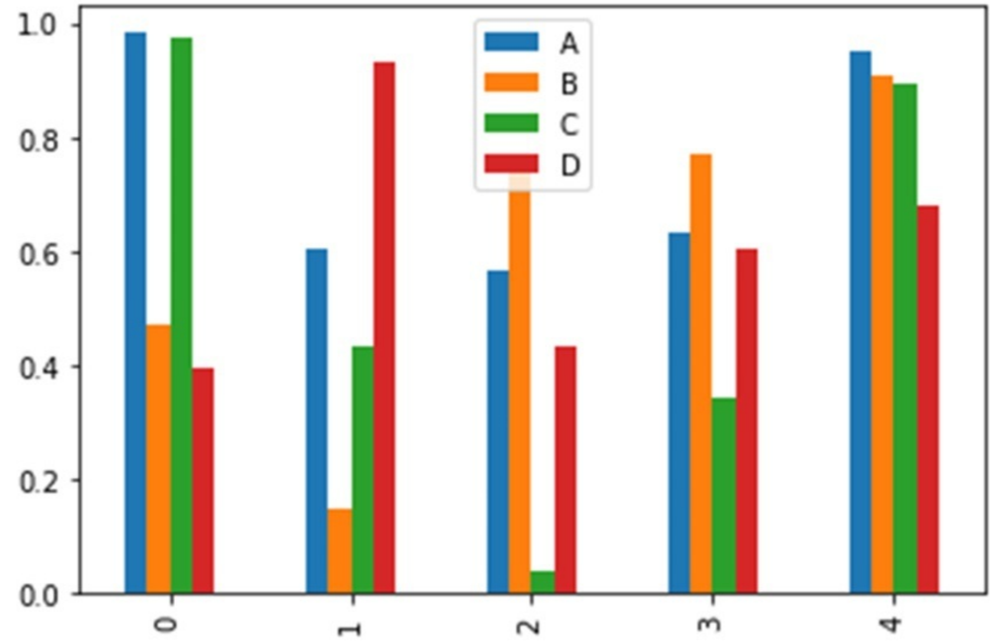
```
In []: df.plot.area()

Out[]: <matplotlib.axes._subplots.AxesSubplot at 0x28d433b2438>
```



The transparency settings of this graph can be set with the argument ‘alpha = value’.

We can also do a bar plot which can categorize our data based off of our row_index.

```
In []: df.plot.bar()

Out[]: <matplotlib.axes._subplots.AxesSubplot at 0x28d435fe828>
```

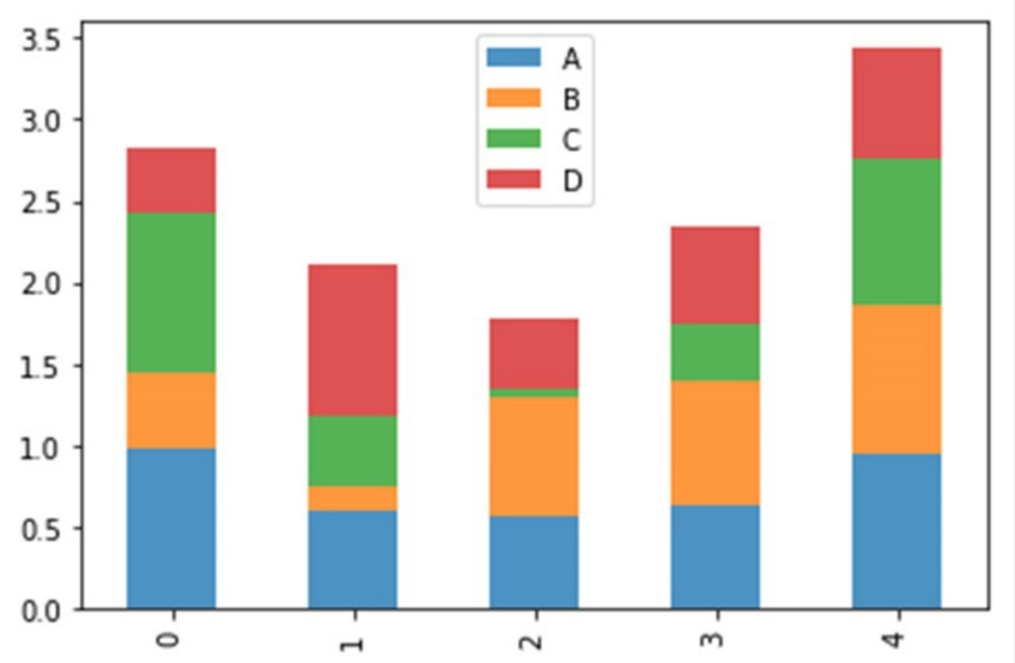


See, our x-axis has the row index, and the y-axis shows the value of in each column per index.

This kind of plot can be useful for things like, sales trends per month (with sales as values and months as row_index), school attendance per day, etc. Our current plots might not be too informative since we are using random data, however, an actual data set would reveal more details.

If you prefer, the bar plots can be stacked to give better visualization:

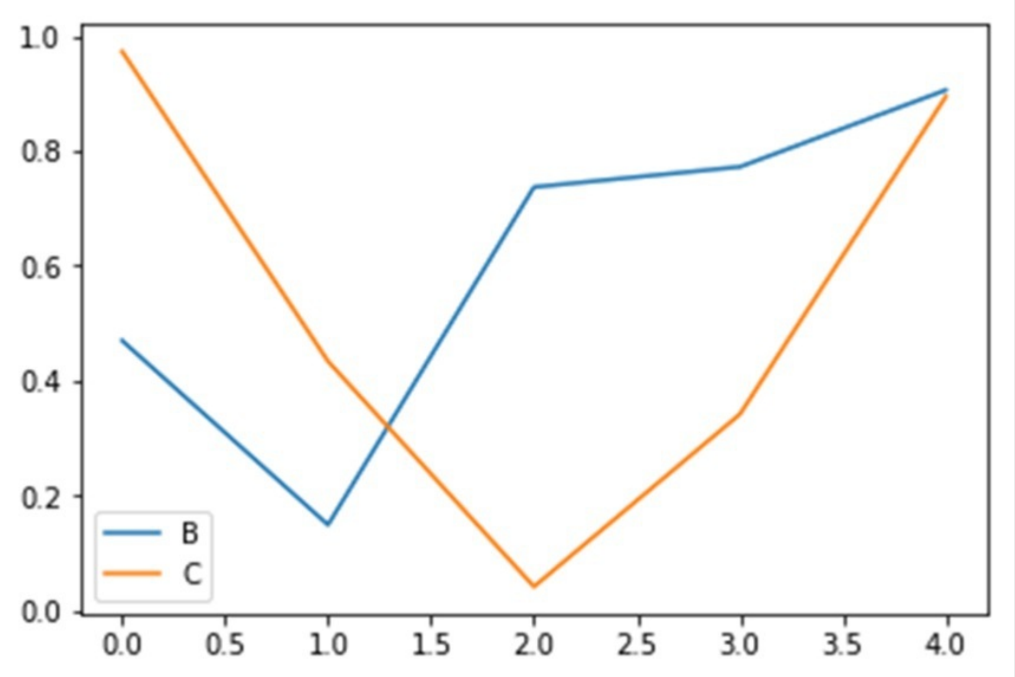
```
In []: df.plot.bar(stacked = True, alpha = 0.8)

Out[]: <matplotlib.axes._subplots.AxesSubplot at 0x28d43a6e048>
```



This kind of plot gives us an idea of the total values per category, as well as the percentages that account for that total. We can still observe that value in column ‘A’ contributes the most in category ‘0’, followed by ‘C’, and so forth.

Line plot:

```
In []: df.plot.line(y =['B','C'])

Out[]: <matplotlib.axes._subplots.AxesSubplot at 0x28d43b6fc88>
```



The line plot takes positional arguments of the x and y-axis. In this case, the y-axis has been specified. Other specifications like: line width ‘lw’, figsize, etc. can be included as well.

We can also make a scatterplot, box plots and a few other plots that can be useful for interpreting data. Depending on your choice, and proficiency with these plotting techniques, you will be able to master data and the information it contains.

Go ahead and check these useful links for extra information on plotting with pandas:
https://towardsdatascience.com/introduction-to-data-visualization-in-python-89a54c97fbed
https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

With these data visualization options, you may start to test your skills by displaying data in a whole range of formats. It is quite obvious that a good knowledge of statistical methods would be very useful for excelling as a data scientist, since data science mostly deals with statistical data. While statistics can be intimidating without graphical aides, your own approach will be better as you now have the full potential of matplotlib, seaborn and pandas to visualize your lessons. For interactive visualization options (not covered here), you can check out ‘plotly and cufflinks’ libraries from this link: https://plot.ly/ipython-notebooks/cufflinks/

In the next section, however, we give a brief introduction into the world and concepts of machine learning.

Chapter 4

Machine Learning with Python

So, what is machine learning?

Machine learning is an advanced form of data analysis and computation that utilizes the exceptional processing speed and pattern recognition techniques of computers for finding and learning new trends in data. Quite long-winded huh?

Simply put, it is an artificial-intelligence-inspired technique of programming that allows computers to improve their learning capabilities through the data they are fed, or can access. This is much like the way human beings develop through life.

While Machine learning is not a new concept or technology -- the term was in fact first introduced at IBM in 1959 by American gaming and AI expert, Arthur Samuel – it has recently gained a lot of interests in terms of research and applications across various fields. This is mostly due to the improvement in processing capabilities of modern generation computers, along with portable, high-level, and fast programming languages that support advanced pattern recognition. The concept behind the technique is consistently being improved and tested, and it will be a key player in the bigger technology revolution of the future.

Okay, machine learning is cool. How is it related to data science?

Imagine this: You are a football player (or soccer in some countries, good grief!), and you play defense. You have tried marking the opposing team's winger three times, and at each encounter, he did a step-over and dribbled to the left. Now, your team is defending, and that striker takes on your co-defender; you are close enough to shout instructions at him/her, what do you say?

Mark to the left!!!

Right? Unfortunately, he dribbles to the right and scores. Everyone blames you.

Okay, that story has a bad ending, but the point is that you made a prediction based on what you have learned about that striker during your previous encounters.

What if, before the game, you had watched year-long footage of that striker’s dribbling techniques and tricks? They would have a very low chance of getting past you, right? How about 10 years long footage (which you could binge watch like The Flash!)? No chance, at all!

This is where the capability of machine learning applies to data science. Computers, while not as smart as humans, are exceptionally great at finding trends (of course they are, computer operations are all about pattern recognition via one’s and zero’s). Now, imagine that machine having access to a large database, like all the tweets on twitter for the past 5 years. It would learn quite a lot about pop-culture, that’s for sure (Financial stock predictions, fraud detection, etc.). These are the most popular applications of machine learning, and the technique is based off extracting information from data; data science!

It is thus important for any current or aspiring data scientist to join the growing machine learning community, and contribute a quota to improving the technology. There are a few programming tools that are optimized for machine learning, and python is one of those.

Python and machine learning

As with most applications of python, there is a library for machine learning and it’s called Scikit Learn. The Scikit learn package is most likely included in your distribution of anaconda, and you can go ahead and verify with conda lis t at the anaconda prompt. You can check out Scikit-learn’s official documentation via this link: https://scikit-learn.org/stable/

Types of machine learning

There are currently four generalized categories of machine learning, and this knowledge is important depending on the application in mind. We have: The supervised, semi-supervised, un-supervised and Reinforcement machine learning.

In supervised learning, the desired output is known by the trainer (you, or whoever is behind the keyboard). The machine is trained using labeled inputs which it associates with corresponding outputs. Through this, the machine develops a predictive model for linking those inputs with specific outputs over a period of iterative learning. It is not so different from the way we learn in a classroom, with a teacher available to correct mistakes. This is the easiest, but usually expensive approach to machine learning.

For unsupervised learning, there is no specific goal in mind. The trainer at times does not know the right answer, and the computer only finds interesting trends in the data based on a training algorithm (usually a clustering technique). This is similar to the informal learning process in humans, where we learn based on our interaction with our environment.

The semi-supervised approach is just a scaled-down version of the supervised, which is useful in the absence of a complete labeled training data set. In this case, the machine has to make some approximations to compensate for the unlabeled data. It is cheaper than supervised learning, but slower and relatively less efficient.

Finally, the reinforcement learning technique is a trial-and-error approach based on the points-reward system in gaming (It is actually used in new gaming engines for creating competitive bosses). Here, the goal is to find the best possible route to achieve a goal. This includes using the minimum resources, while maximizing time. It is also very useful in modern robotics. Here, the machine learns from its experience while interacting with the environment.

All of these applications of machine learning are possible, and ready for exploring via Scikit learn with python. All you need is a good resource, time, dedication and all the knowledge of basic data analysis and visualization from the previous sections of this book; this is because data cleaning and preparation is a big part of machine learning.

Machine learning, like other elements of data science requires a good background in statistical analysis. Things like regression analysis i.e. linear and logistic regression, K-means clustering, K-NN (nearest neighbor), etc. Here is a link to download a good, and free book that can introduce you to some of these statistical concepts:http://www-bcf.usc.edu/~gareth/ISL/ISLR%20Seventh%20Printing.pdf

Conclusion

We hope you have learned a ton from this book. The mastery of a skill is not just in knowing, but continuous practice, wherein lies uniqueness of skill and competence. With time, you will keep pursuing that never-ending expanse of knowledge that is data science. At least, you are no longer a novice!