Python Pandas For Beginners

Pandas Specialization for Data Scientists

Our Books are designed to teach beginners Data Science and AI

PYTHON PANDAS
FOR BEGINNERS

PANDAS SPECIALIZATION
FOR DATA SCIENTISTS

AI PUBLISHING

First Printing, 2021

Edited by AI Publishing
eBook Converted and Cover by AI Publishing Studio
Published by AI Publishing LLC

ISBN-13: 978-1-956591-10-1

How to Contact Us

If you have any feedback, please let us know by emailing contact@aipublishing.io .

Your feedback is immensely valued, and we look forward to hearing from you.
It will be beneficial for us to improve the quality of our books.

To get the Python codes and materials used in this book, please click the link below:

www.aipublishing.io/book-pandas-python

The order number is required.

About the Publisher

At AI Publishing Company, we have established an international learning platform specifically for young students, beginners, small enterprises, startups, and managers who are new to data science and artificial intelligence.

Through our interactive, coherent, and practical books and courses, we help beginners learn skills that are crucial to developing AI and data science projects.

Our courses and books range from basic introduction courses to language programming and data science to advanced courses for machine learning, deep learning, computer vision, big data, and much more, using programming languages like Python, R, and some data science and AI software.

AI Publishing's core focus is to enable our learners to create and try proactive solutions for digital problems by leveraging the power of AI and data science to the maximum extent.

Moreover, we offer specialized assistance in the form of our free online content and eBooks, providing up-to-date and useful insight into AI practices and data science subjects, along with eliminating the doubts and misconceptions about AI and programming.

Our experts have cautiously developed our online courses and kept them concise, short, and comprehensive so that you can understand everything clearly and effectively and start practicing the applications right away.

We also offer consultancy and corporate training in AI and data science for enterprises so that their staff can navigate through the workflow efficiently.

With AI Publishing, you can always stay closer to the innovative world of AI and data science.

If you are eager to learn the A to Z of AI and data science but have no clue where to start, AI Publishing is the finest place to go.

Please contact us by email at
contact@aipublishing.io .

AI Publishing Is Searching for Authors Like You

Interested in becoming an author for AI Publishing?
Please contact us at author@aipublishing.io .

We are working with developers and AI tech professionals just like you to help them share their insights with global AI and Data Science lovers. You can share all your knowledge about hot topics in AI and Data Science.

Table of Contents

From the Same Publisher

Preface

With the rise of data science and high-performance computing hardware, programming languages have evolved as well. Various libraries in different programming languages have been developed that provide a layer of abstraction over complex data science tasks. Python programming language has taken the lead in this regard. More than 50 percent of all data science-related projects are being developed using Python programming.

If you ask a data science expert what the two most common and widely used Python libraries for data science are, the answer would almost invariably be the NumPy library and the Pandas library. And this is what the focus of this book is. It introduces you to the NumPy and Pandas libraries with the help of different use cases and examples.

Thank you for your decision to purchase this book. I can assure you that you will not regret your decision.

§ Book Approach

The book follows a very simple approach. The 1st chapter is introductory and provides information about setting up the installation environment. The 1st chapter also contains a brief crash course on Python, which you can skip if you are already familiar with Python.

Chapter 2 provides a brief introduction to Pandas. Chapter 3 explains how you can manipulate Pandas dataframes, while the 4^{th} chapter focuses on grouping, aggregating, and merging data with Pandas. Finally, the 5^{th} and 6^{th} chapters focus on data visualization and time-series handling with Pandas, respectively.

Each chapter explains the concepts theoretically, followed by practical examples. Each chapter also contains exercises that students can use to evaluate their understanding of the concepts explained in the chapter. The Python notebook for each chapter is provided in the *Codes Folder* that accompanies this book. It is advised that instead of copying the code from the book, you write the code yourself, and in case of an error, you match your code with the corresponding Python notebook, find and then correct the error. The datasets used in this book are either downloaded at runtime or are available in the *Resources* folder.

Do not copy and paste the code from the PDF notebook, as you might face an indentation issue. However, if you have to copy some code, copy it from the Python Notebooks.

§ Who Is This Book For?

The book is aimed ideally at absolute beginners to data science in specific and Python programming in general. If you are a beginner-level data scientist, you can use this book as a first introduction to NumPy and dataframes. If you are already familiar with Python and data science, you can also use this book for general reference to perform common tasks with NumPy and Pandas.

Since this book is aimed at absolute beginners, the only prerequisites to efficiently use this book are access to a computer with the internet and basic knowledge of programming. All the codes and datasets have been provided. However, you will need the internet to download the data preparation libraries.

§ How to Use This Book?

In each chapter, try to understand the usage of a specific concept first and then execute the example code. I would again stress that rather than copying and pasting code, try to write codes yourself. Then, in case of any error, you can match your code with the source code provided in the book as well as in the Python Notebooks in the *Resources* folder.

Finally, answer the questions asked in the exercises at the end of each chapter. The solutions to the exercises have been given at the end of the book.

To facilitate the reading process, occasionally, the book presents three types of box-tags in different colors: **Requirements, Further Readings,** and **Hands-on Time.** Examples of these boxes are shown below.

Requirements

This box lists all requirements needed to be done before proceeding to the next topic. Generally, it works as a checklist to see if everything is ready before a tutorial.

Further Readings

Here, you will be pointed to some external reference or source that will serve as additional content about the specific **Topic** being studied. In general, it consists of packages, documentations, and cheat sheets.

Hands-on Time

Here, you will be pointed to an external file to train and test all the knowledge acquired about a **Tool** that has been studied. Generally, these files are Jupyter notebooks (.ipynb), Python (.py) files, or documents (.pdf).

The box-tag **Requirements** lists the steps required by the reader after reading one or more topics. **Further Readings** provides relevant references for specific topics to get to know the additional content of the topics. **Hands-on Time** points to practical tools to start working on the specified topics. Follow the instructions given in the box-tags to better understand the topics presented in this book.

About the Author

M. Usman Malik holds a Ph.D. in Computer Science from Normandy University, France, with Artificial Intelligence and Machine Learning being his main areas of research. Muhammad Usman Malik has over five years of industry experience in Data Science and has worked with both private and public sector organizations. He likes to listen to music and play snooker in his free time.

You can follow his Twitter handle: @usman_malikk.

Get in Touch With Us

Feedback from our readers is always welcome.

For general feedback, please send us an email at contact@aipublishing.io and mention the book title in the subject line.

Although we have taken extraordinary care to ensure the accuracy of our content, errors do occur. If you have found an error in this book, we would be grateful if you could report this to us as soon as you can.

If you are interested in becoming an AI Publishing author and if you have expertise in a topic and you are interested in either writing or contributing to a book, please send us an email at author@aipublishing.io .

Download the PDF version

We request you to download the PDF file containing the color images of the screenshots/diagrams used in this book here:

www.aipublishing.io/book-pandas-python
The order number is required.

Warning

In Python, indentation is very important. Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular code block. After each loop or if-condition, be sure to pay close attention to the intent.

Example

```
# Python program showing
# indentation

site = 'aisciences'

if site == 'aisciences':
    print('Logging to www.aisciences.io...')
else:
    print('retype the URL.')
print('All set !')
```

To avoid problems during execution, we advise you to download the codes available on Github by requesting access from the link below. Please have your order number ready for access:

www.aipublishing.io/book-pandas-python

Introduction

In this chapter, you will briefly see what NumPy and Pandas libraries are and their advantages. You will also set up the environment that you will need to run the NumPy and Pandas scripts in this book. The chapter concludes with an optional crash course on the Python programming language.

1.1. What Is Pandas?

The Pandas library (https://pandas.pydata.org/) is an open-source, BSD-licensed Python library providing high-performance, simple-to-use data structures and data analysis tools for the versatile Python programming language.

The Pandas library provides data structures that store data in tabular data structures called series and dataframes. With these data structures, you can perform tasks like filtering, merging, and manipulating data based on different criteria.

In Pandas, you can import data from various sources such as flat files (CSV, Excel, etc.), databases, and even online links. Pandas also offer data visualization functionalities. With Pandas, you can plot different types of static plots using a single line of code.

The following are some of the main advantages of the Pandas library in Python:

- Pandas provides built-in features to handle a very large amount of datasets in a fast and memory-optimized manner.

- The Pandas library comes with a myriad of default features for data science and machine learning tasks.
- Pandas library can work with a variety of input data sources.
- Pandas uses labeled indexing for records, which is an extremely intuitive feature for segmenting and subsetting large datasets.

Part II of this book (chapter 7 to chapter 11) is dedicated to the Pandas library, where you will study various components and use cases of Pandas in detail.

You can install the Pandas package in your Python installation via the following pip command in your command terminal.

```
$ pip install pandas
```

If you install the Anaconda distribution (https://bit.ly/3koKSwb) for Python, as you will see in this chapter, the Pandas library will be installed by default.

1.2. Environment Setup and Installation

1.2.1. Windows Setup

The time has come to install Python on Windows using an IDE. We will use Anaconda throughout this book, right from installing Python to writing multithreaded codes. Now let us get going with the installation.

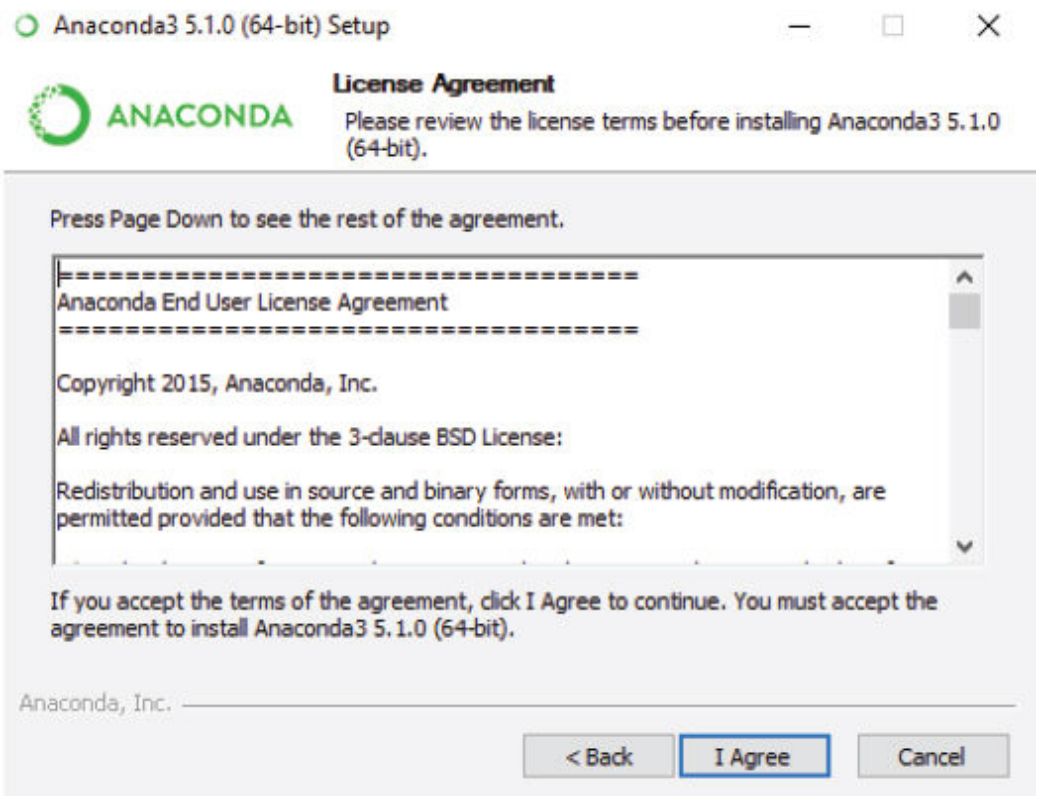
This section explains how you can download and install Anaconda on Windows.

Follow these steps to download and install Anaconda.

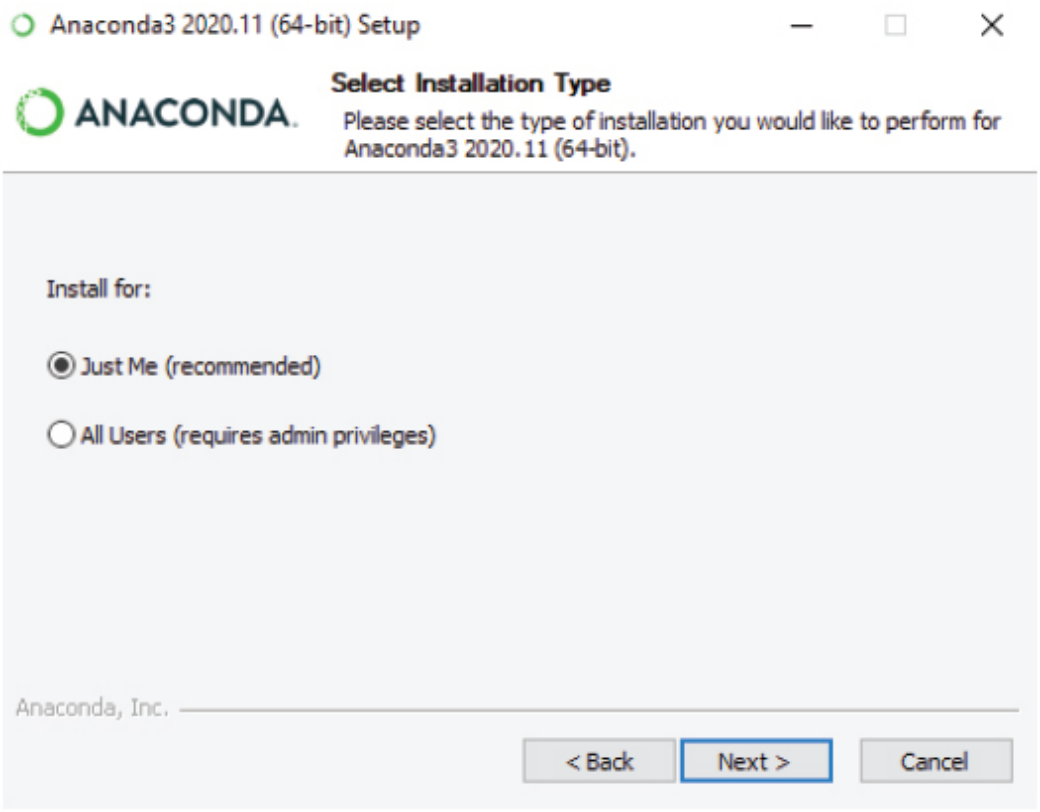
1. Open the following URL in your browser.
   https://www.anaconda.com/products/individual
2. The browser will take you to the following webpage. Depending on your OS, select the 64-bit or 32- bit Graphical Installer file for Windows. The file will download within 2–3 minutes based on the speed of your internet.

3. Run the executable file after the download is complete. You will most likely find the downloaded file in your download folder. The installation wizard will open when you run the file, as shown in the following figure. Click the *Next* button.

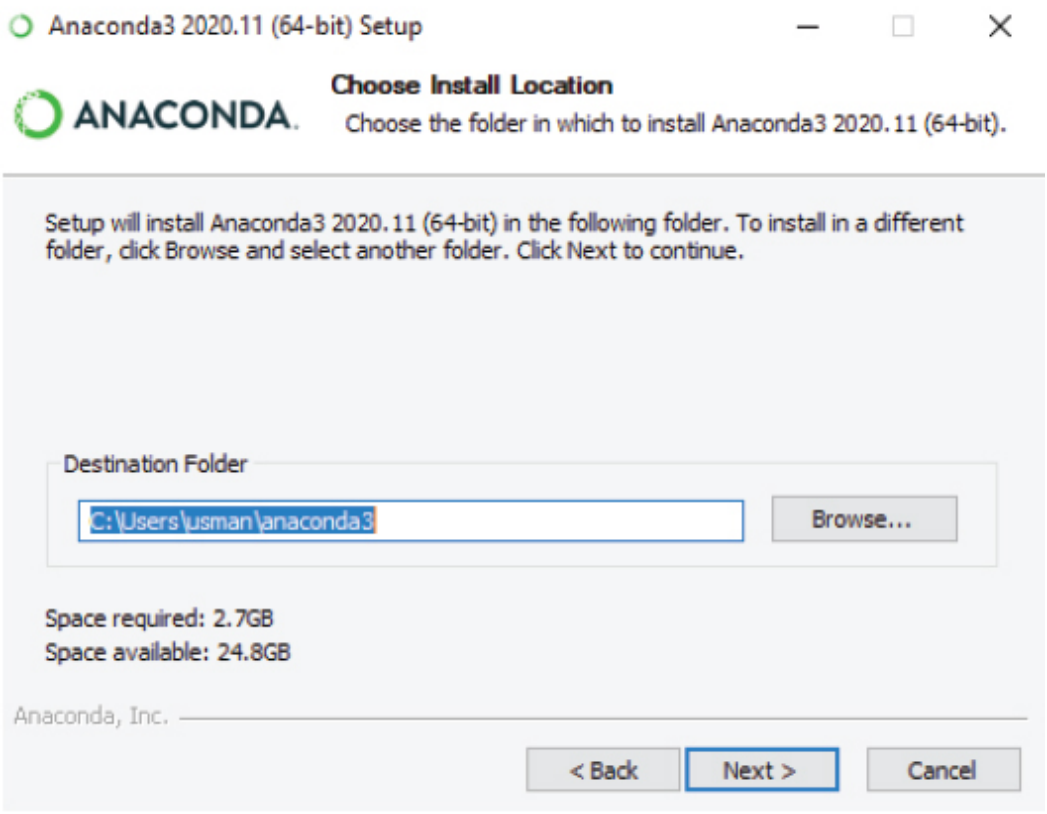
4. Now click *I Agree* on the **License Agreement** dialog, as shown in the following screenshot.



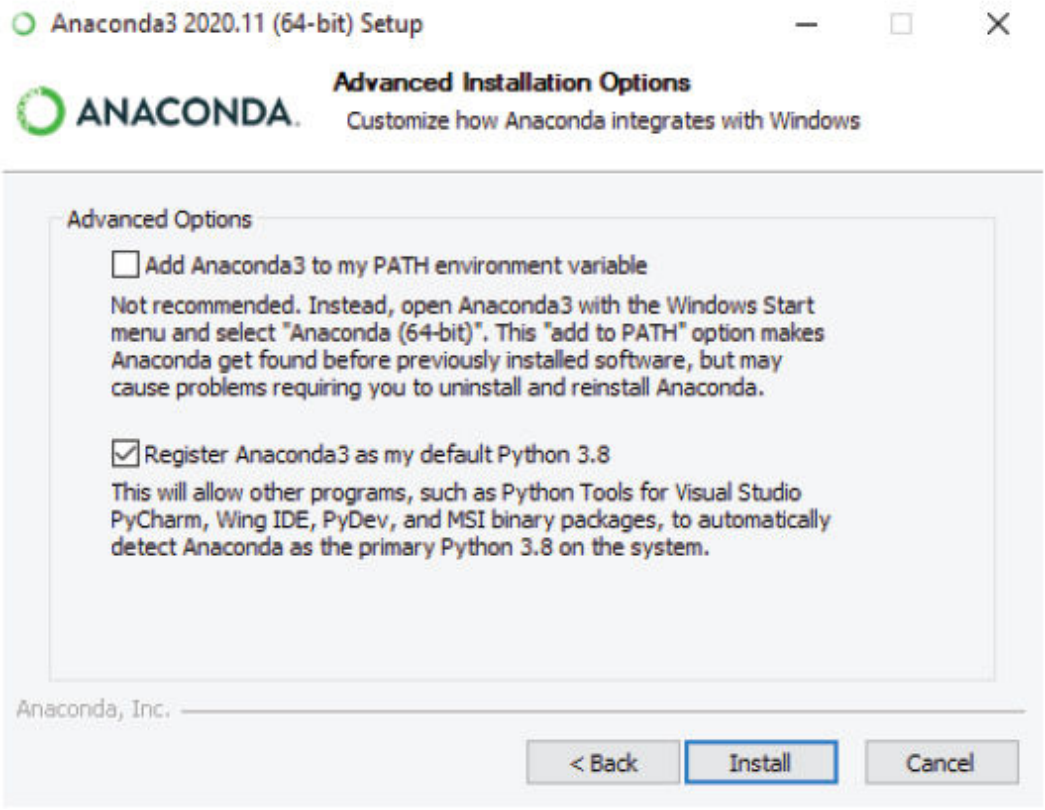
5. Check the *Just Me* radio button from the **Select Installation Type** dialog box. Then, click the *Next* button to continue.



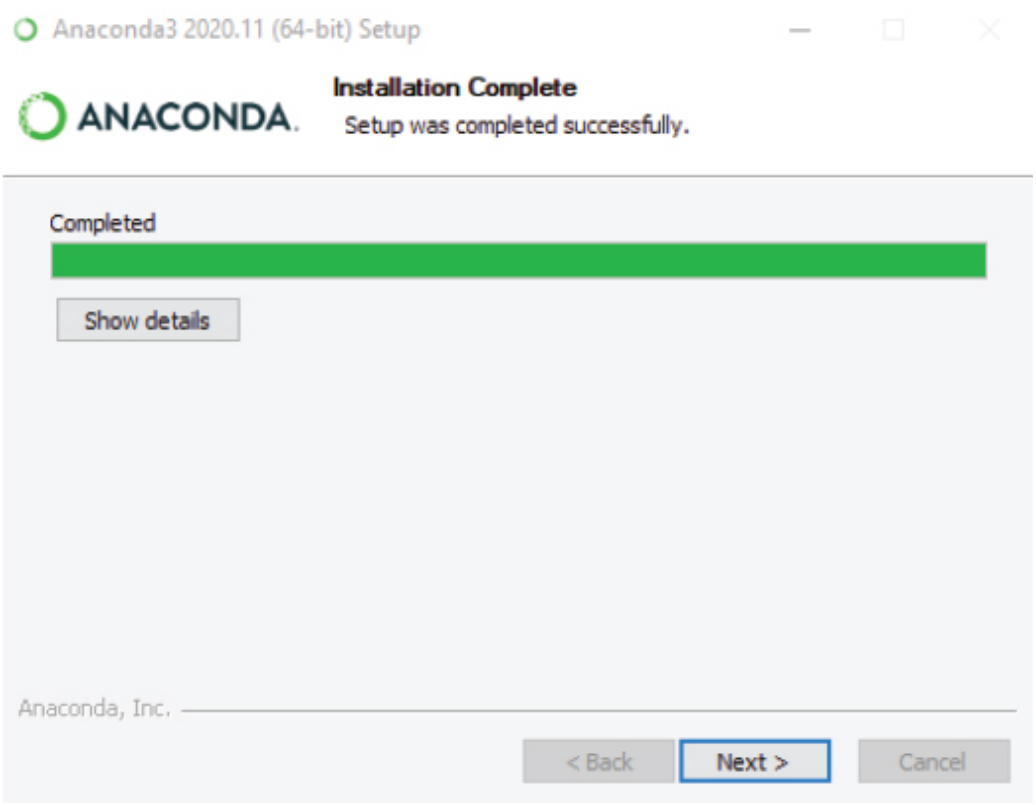
6. Now, the **Choose Install Location** dialog will be displayed. Change the directory if you want, but the default is preferred. The installation folder should have at least 3 GB of free space for Anaconda. Click the *Next* button.



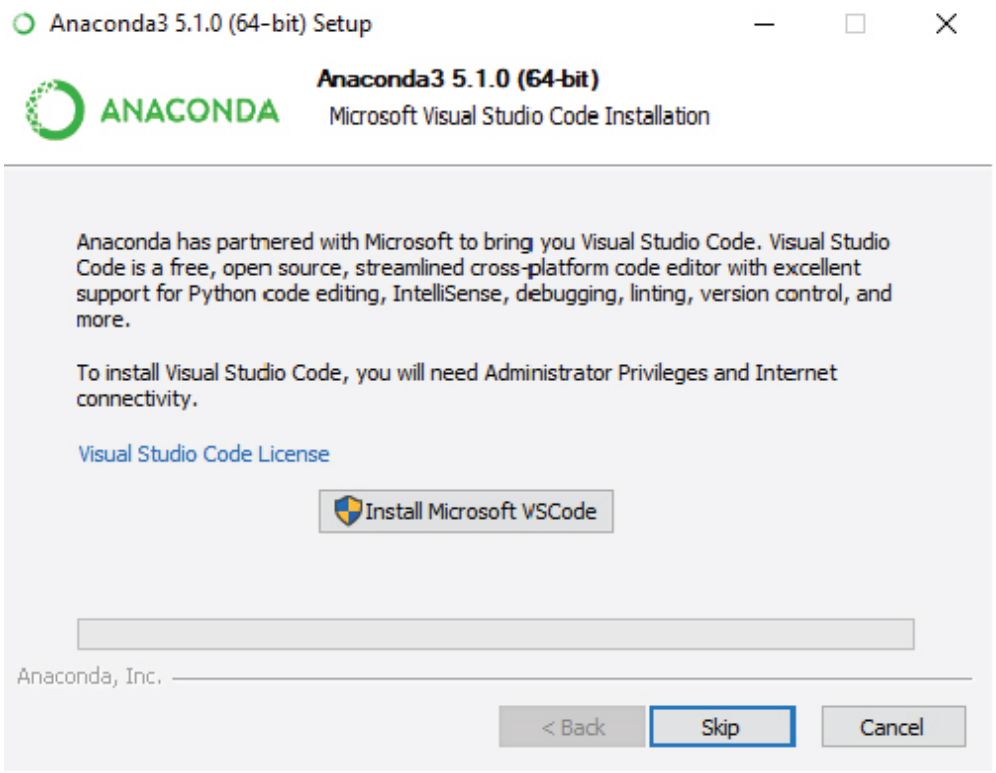
7. Go for the second option, *Register Anaconda as my default Python 3.8*, in the **Advanced Installation Options** dialog box. Click the *Install* button to start the installation, which can take some time to complete.



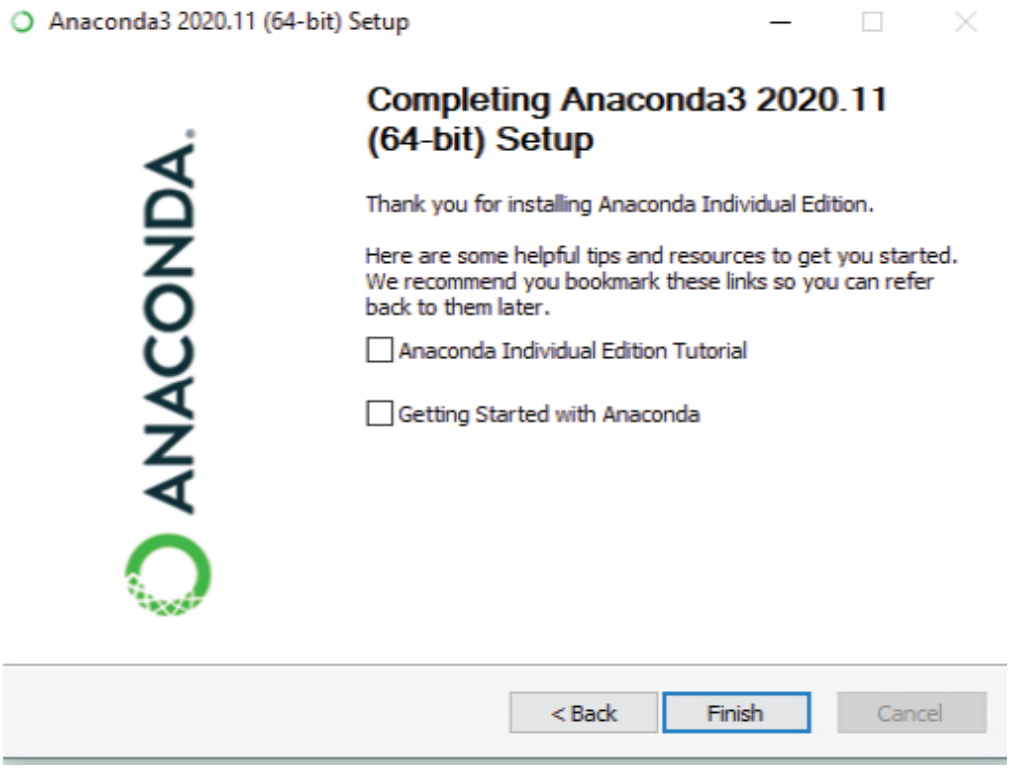
8. Click *Next* once the installation is complete.



9. Click *Skip* on the **Microsoft Visual Studio Code Installation** dialog box.



10. You have successfully installed Anaconda on your Windows. Excellent job. The next step is to uncheck both checkboxes on the dialog box. Now, click on the *Finish* button.



1.2.2. Mac Setup

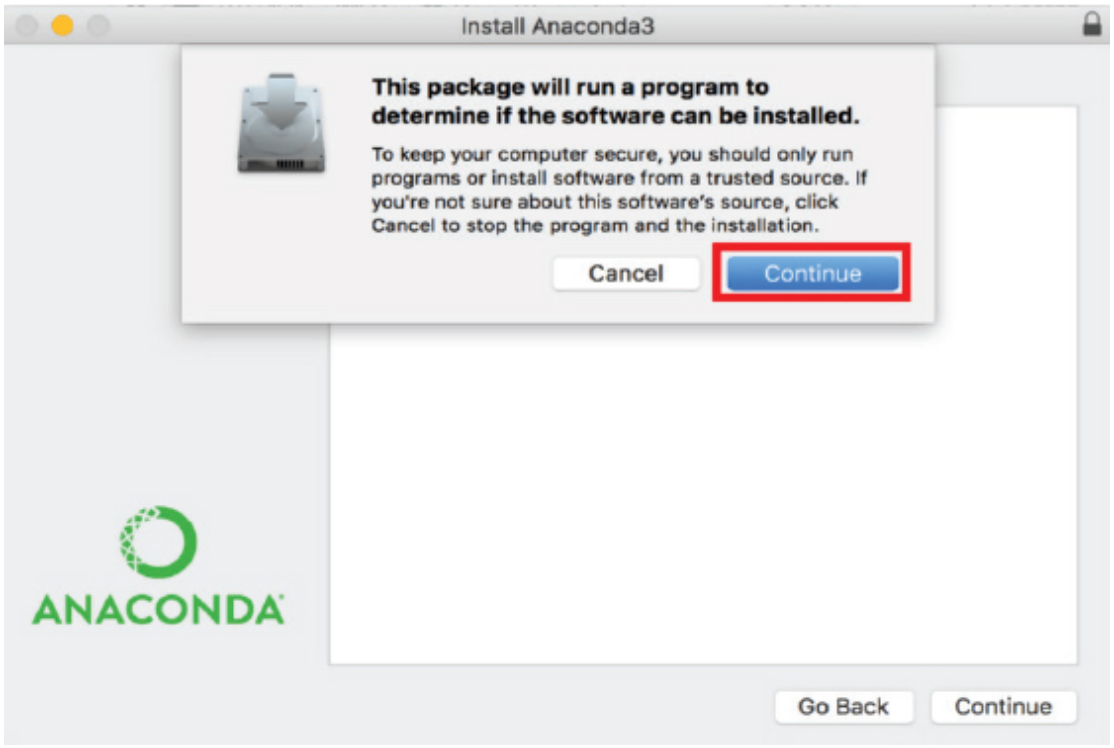
Anaconda's installation process is almost the same for Mac. It may differ graphically, but you will follow the same steps you followed for Windows. The only difference is that you have to download the executable file, which is compatible with Mac operating system.

This section explains how you can download and install Anaconda on Mac.

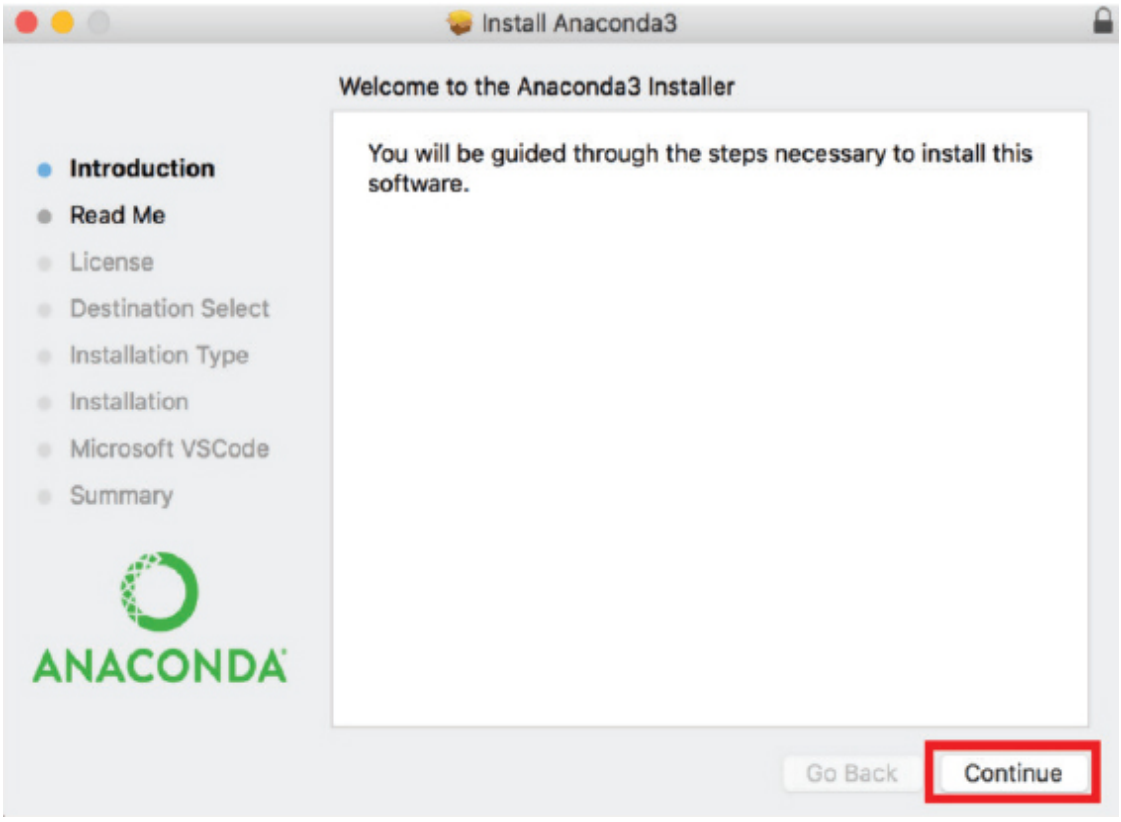
Follow these steps to download and install Anaconda.

1. Open the following URL in your browser. https://www.anaconda.com/products/individual
2. The browser will take you to the following webpage. Depending on your OS, select the 64-bit or 32-bit Graphical Installer file for macOS. The file will download within 2–3 minutes based on the speed of your internet.

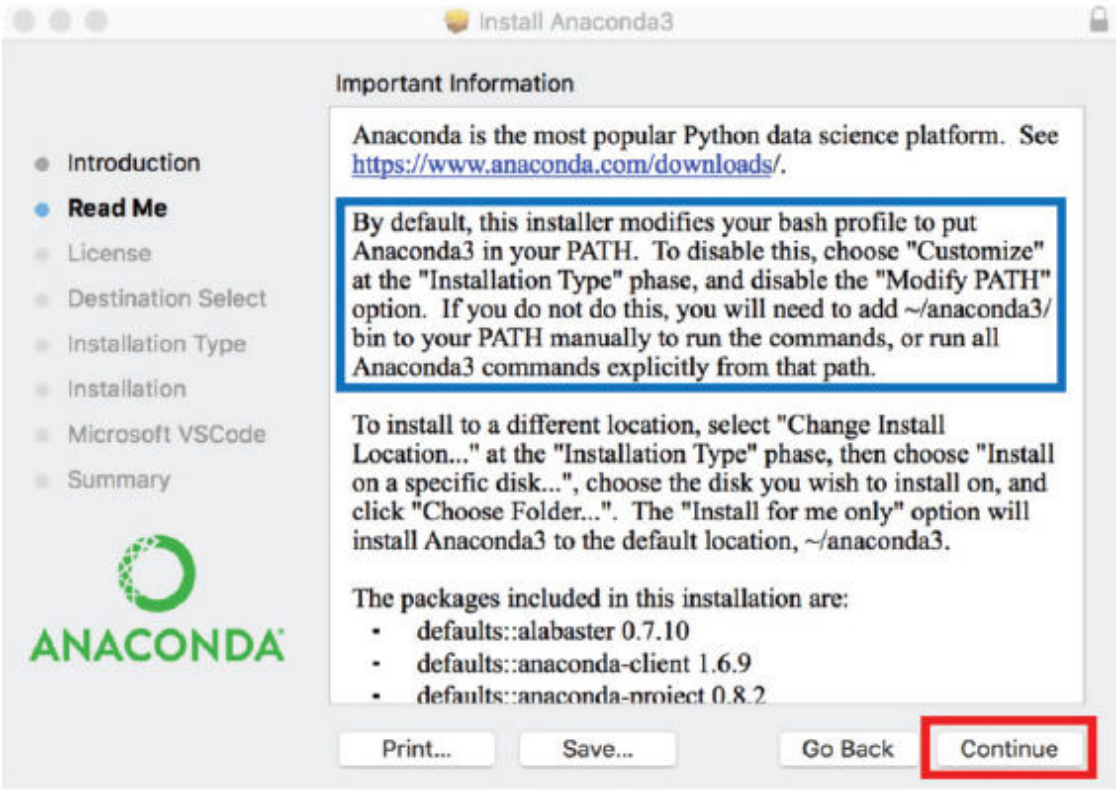
3. Run the executable file after the download is complete. You will most likely find the downloaded file in your download folder. The name of the file should be similar to "Anaconda3-5.1.0-Windows-x86_64." The installation wizard will open when you run the file, as shown in the following figure. Click the *Continue* button.



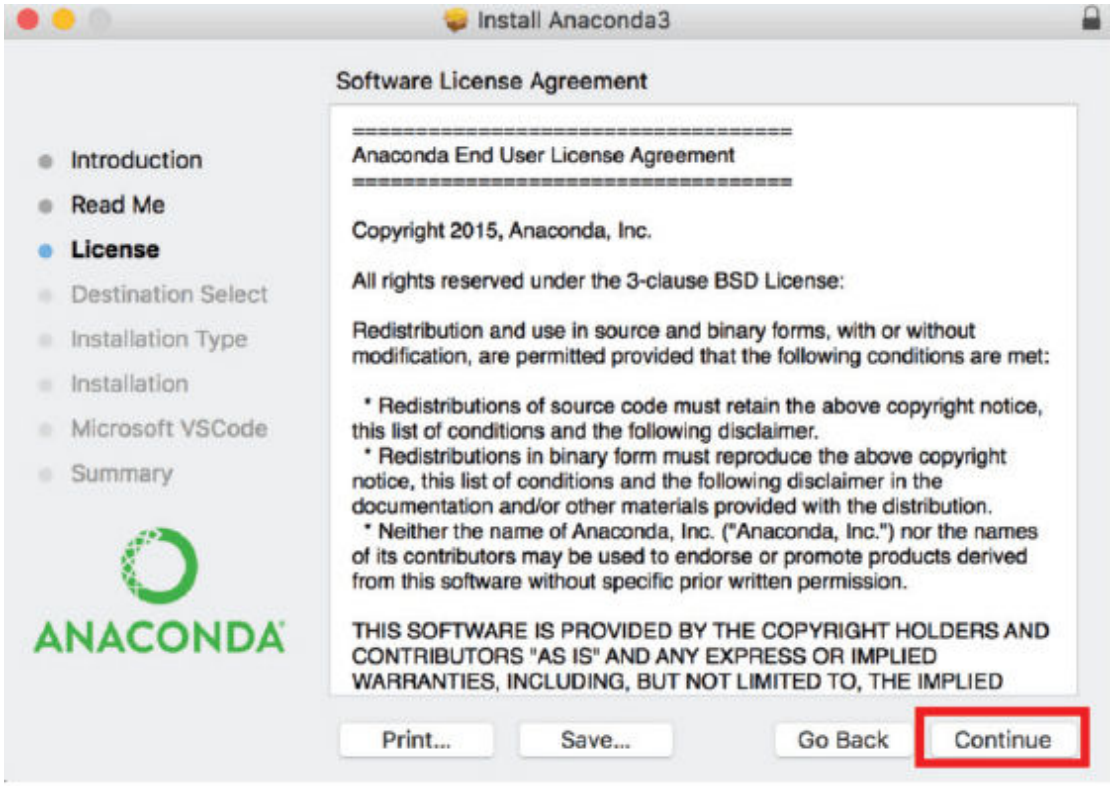
4. Now click *Continue* on the **Welcome to Anaconda 3 Installer** window, as shown in the following screenshot.



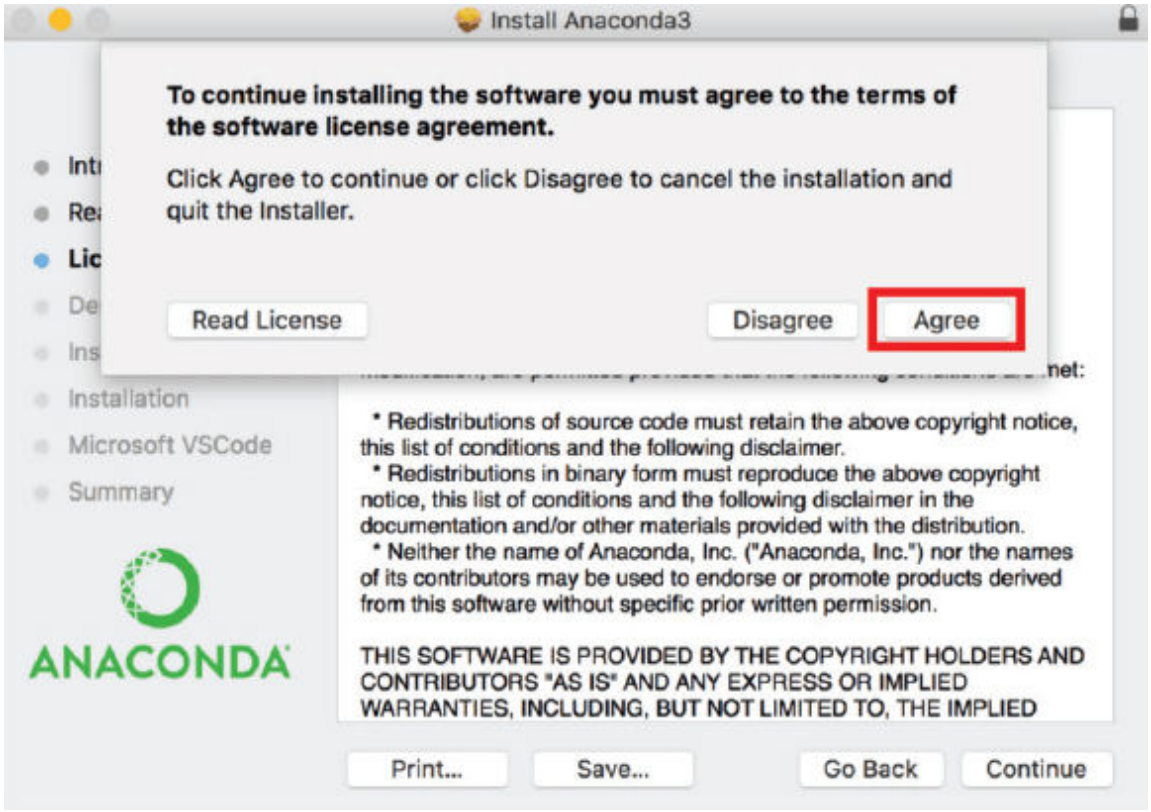
5. The **Important Information** dialog will pop up. Simply click *Continue* to go with the default version, that is, Anaconda 3.



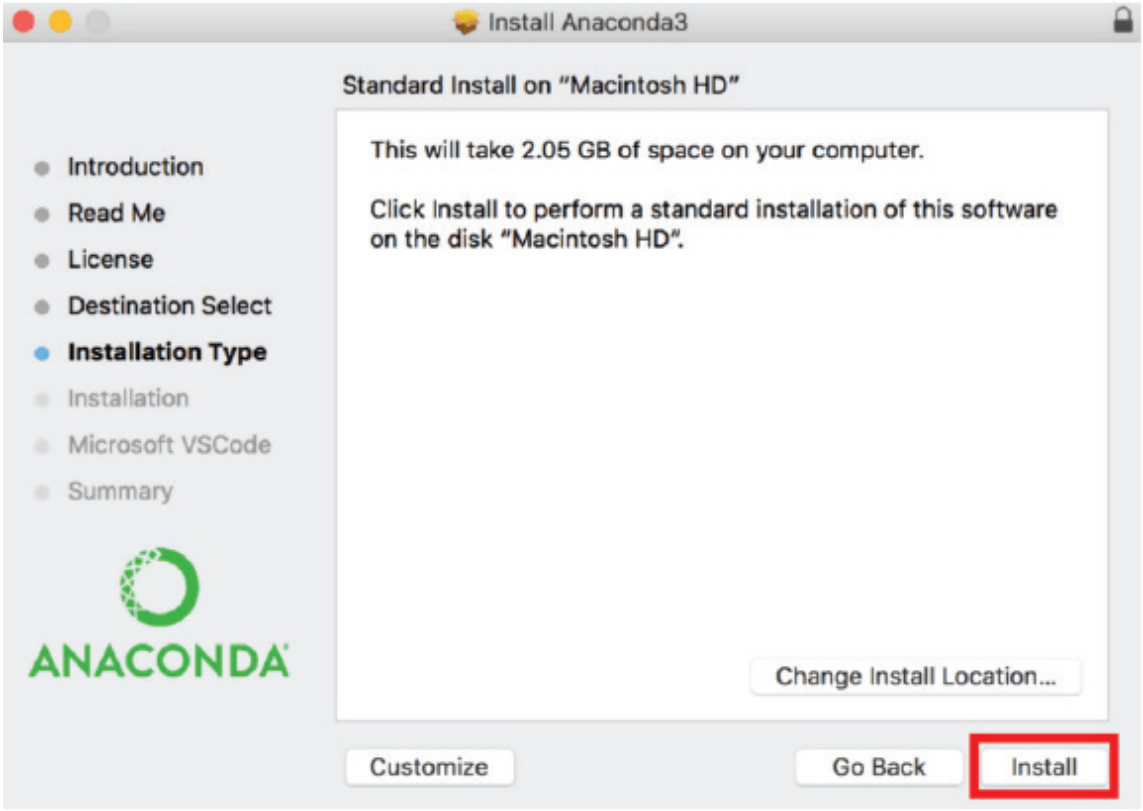
6. Click *Continue* on the **Software License Agreement** dialog.



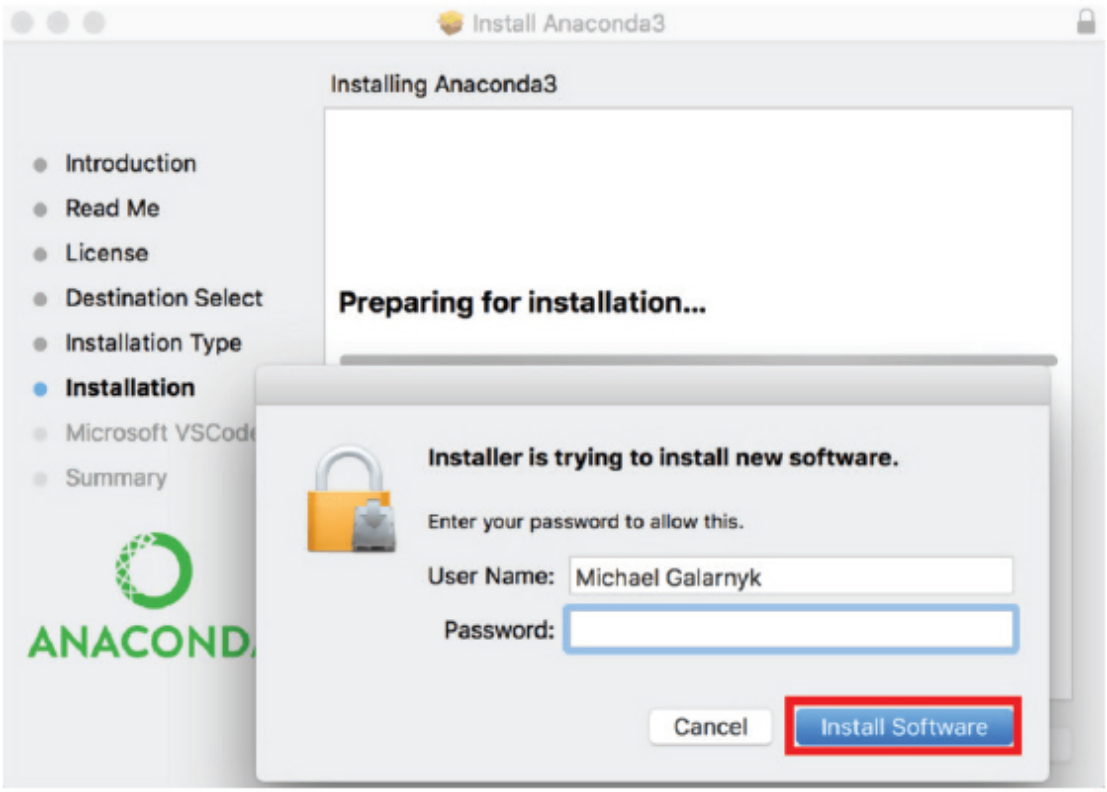
7. It is mandatory to read the license agreement and click the *Agree* button before you can click the *Continue* button again.



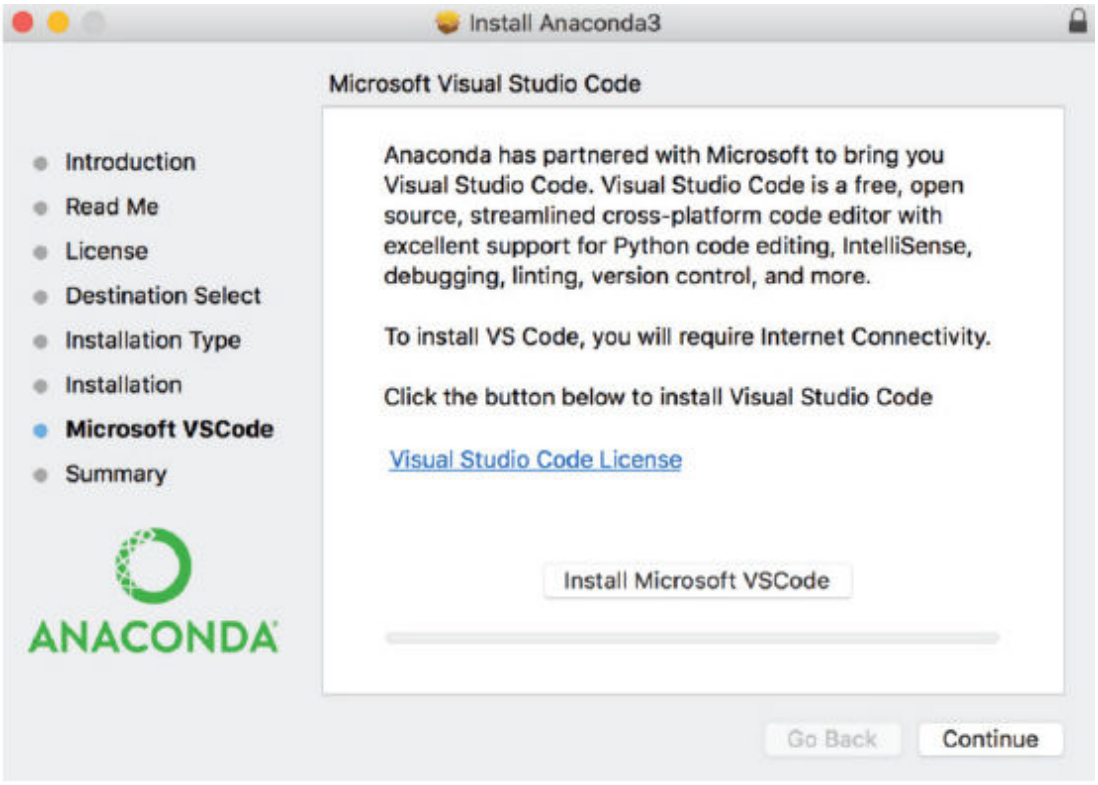
8. Simply click *Install* on the next window that appears.



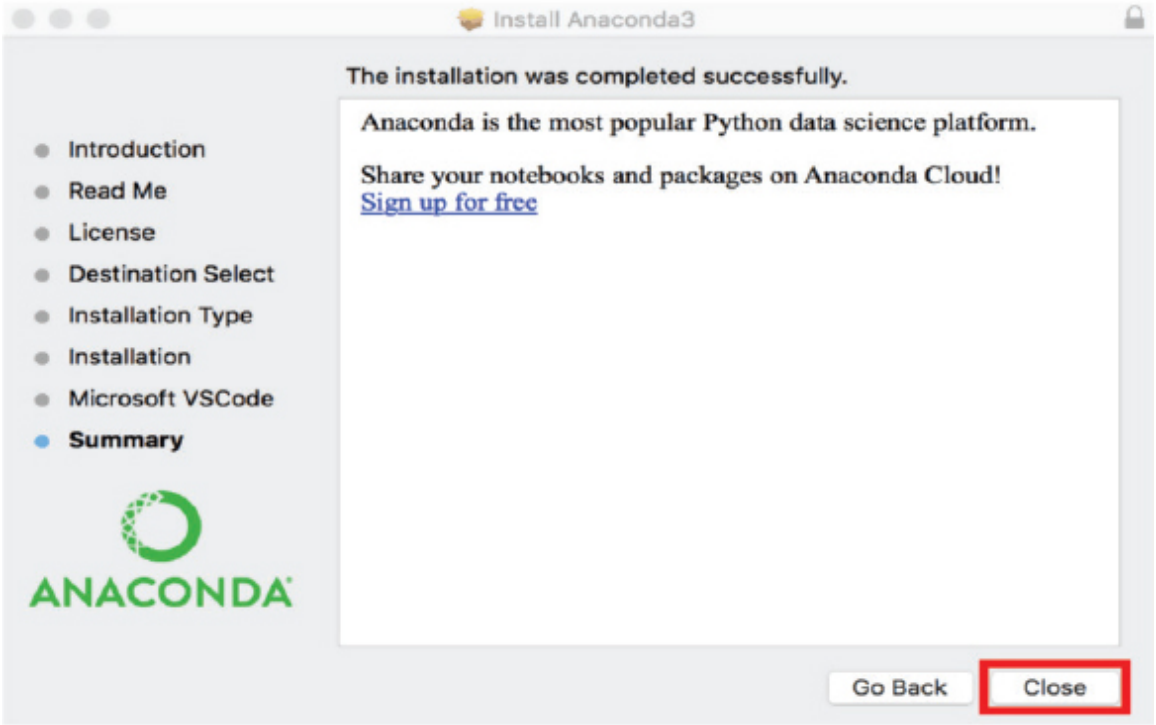
The system will prompt you to give your password. Use the same password you use to log in to your Mac computer. Now, click on *Install Software*.



9. Click *Continue* on the next window. You also have the option to install **Microsoft VSCode** at this point.



The next screen will display the message that the installation has been completed successfully. Click on the *Close* button to close the installer.



There you have it. You have successfully installed Anaconda on your Mac computer. Now, you can write Python code in Jupyter and Spyder the same way you wrote it in Windows.

1.2.3. Linux Setup

We have used Python's graphical installers for installation on Windows and Mac. However, we will use the command line to install Python on Ubuntu or Linux. Linux is also more resource-friendly, and installation of software is particularly easy as well.

Follow these steps to install Anaconda on Linux (Ubuntu distribution).

1. Go to the following link to copy the installer bash script from the latest available version.
   https://www.anaconda.com/products/individual

2. The second step is to download the installer bash script. Log into your Linux computer and open your terminal. Now, go to /temp directory and download the bash you downloaded from Anaconda's home page using curl.

```
$ cd / tmp

$ curl –o https://repo.anaconda.com.archive/Anaconda3-5.2.0-Linux-x86_64.sh
```

3. You should also use the cryptographic hash verification through SHA-256 checksum to verify the integrity of the installer.

```
$ sha256sum Anaconda3-5.2.0-Linux-x86_64.sh
```

You will get the following output.

```
09f53738b0cd3bb96f5b1bac488e5528df9906be2480fe61df40e0e0d19e3d48 Anaconda3-5.2.0-Linux-
    x86_64.sh
```

4. The fourth step is to run the Anaconda Script, as shown in the following figure.

```
$ bash Anaconda3-5.2.0-Linux-x86_64.sh
```

The command line will produce the following output. You will be asked to review the license agreement. Keep on pressing *Enter* until you reach the end.

```
Output

Welcome to Anaconda3 5.2.0

In order to continue the installation process, please review the license agreement.
Please press Enter to continue
>>>
…
Do you approve the license terms? [yes|No]
```

Type *Yes* when you get to the bottom of the License Agreement.

5. The installer will ask you to choose the installation location after you agree to the license agreement. Simply press *Enter* to choose the default location. You can also specify a different location if you want.

```
Output

Anaconda3 will now be installed on the following location: /home/tola/anaconda3

- To confirm the location, press ENTER
- To abort the installation, press CTRL-C
- Otherwise, specify a different location below

[/home/tola/anaconda3] >>>
```

The installation will proceed once you press *Enter*. Once again, you have to be patient as the installation process takes some time to complete.

6. You will receive the following result when the installation is complete. If you wish to use the conda command, type *Yes* .

```
Output
…
Installation finished.
Do you wish the installer to prepend the Anaconda3 install location to a path in your
    /home/tola/.bashrc? [yes|no]
[no]>>>
```

You will have the option to download the Visual Studio Code at this point, as well. Type *yes* or *no* to install or decline, respectively.

7. Use the following command to activate your brand new installation of Anaconda3.

```
$ source `/.bashrc
```

8. You can also test the installation using the conda command.

```
$ conda list
```

Congratulations. You have successfully installed Anaconda on your Linux system.

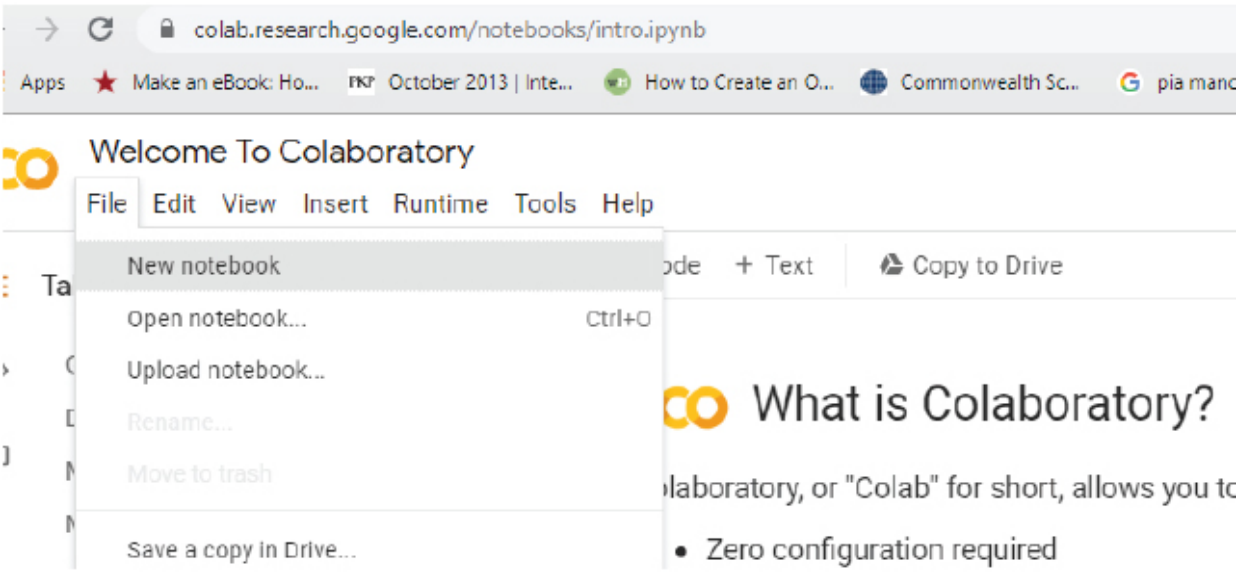
1.2.4. Using Google Colab Cloud Environment

In addition to local Python environments such as Anaconda, you can run deep learning applications on Google Colab as well, which is Google's platform for deep learning with GPU support. All the codes in this book have been run using Google Colab. Therefore, I would suggest that you use Google Colab, too.

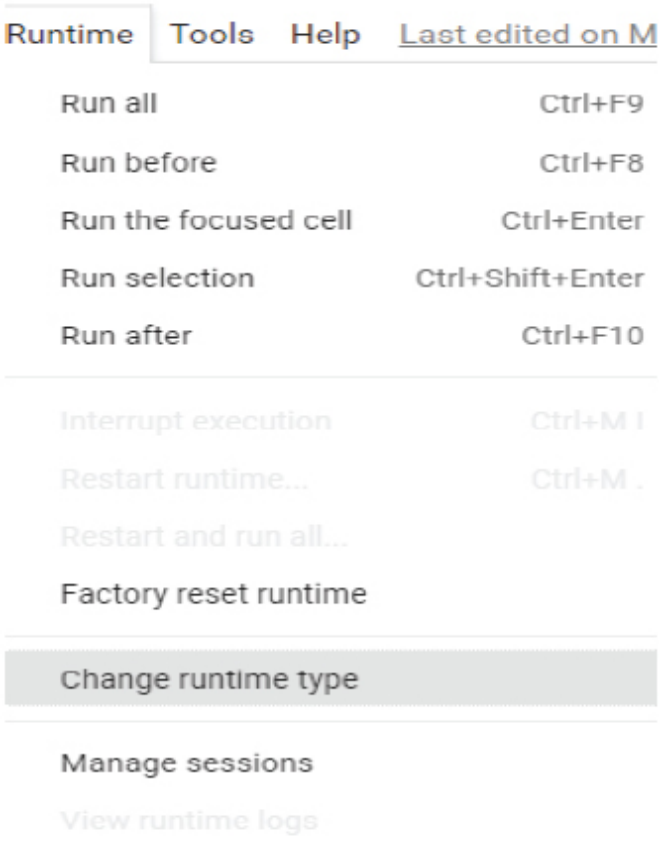
To run deep learning applications via Google Colab, all you need is a Google/Gmail account. Once you have a Google/ Gmail account, you can simply go to:

https://colab.research.google.com/

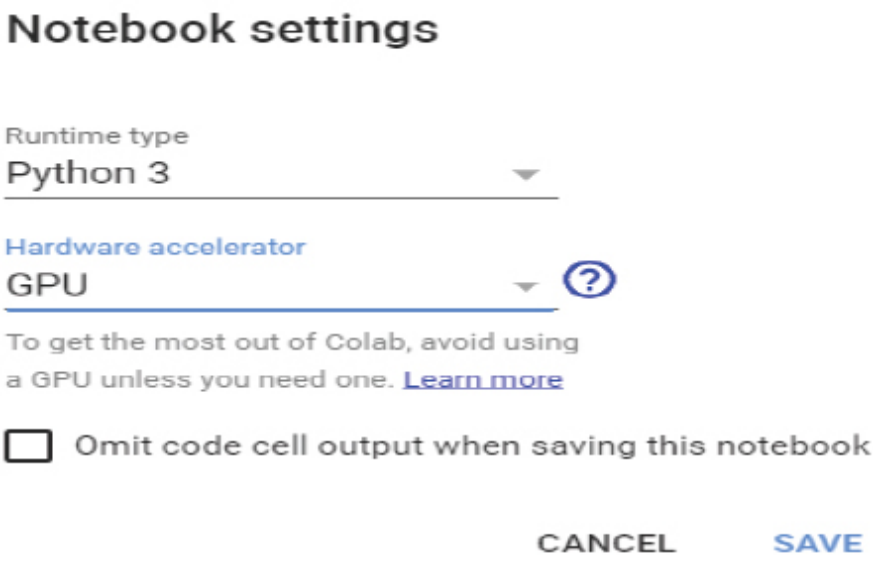
Next, click on File -> New notebook, as shown in the following screenshot.



Next, to run your code using GPU, from the top menu, select Runtime -> Change runtime type, as shown in the following screenshot:



You should see the following window. Here, from the dropdown list, select GPU, and click the *Save* button.



To make sure you are running the latest version of TensorFlow, execute the following script in the Google Colab notebook cell. The following script will

update your TensorFlow version.

```
pip install --upgrade tensorflow
```

To check if you are really running TensorFlow version > 2.0, execute the following script.

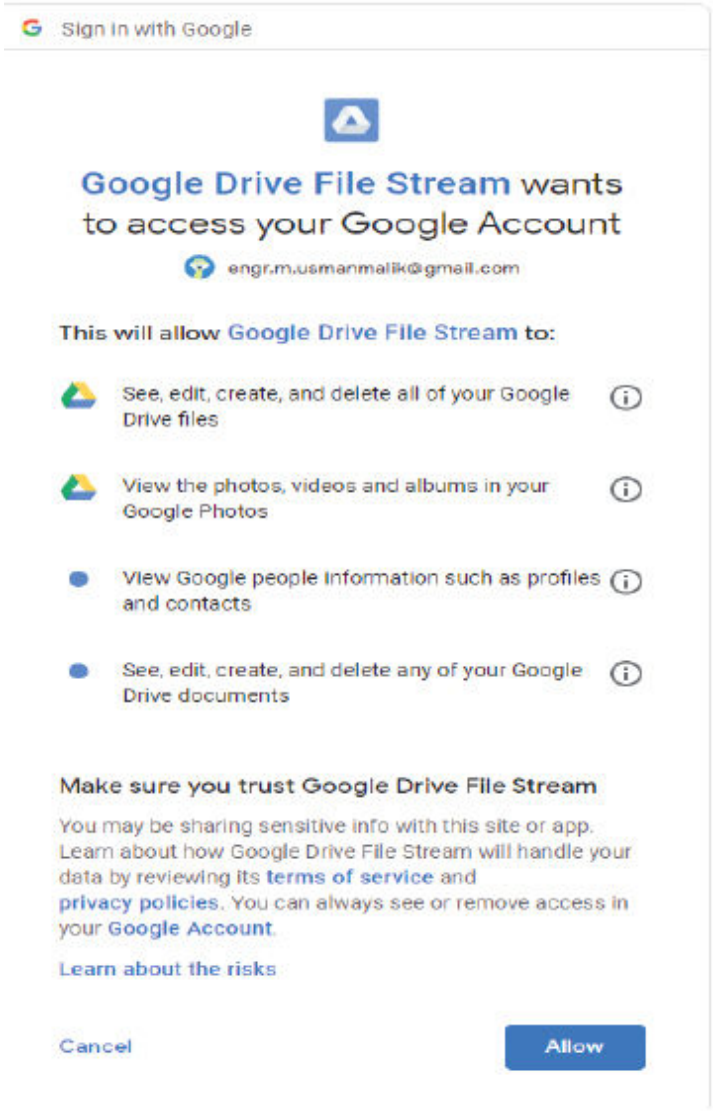
```
import tensorflow as tf
print(tf.__version__)
```

With Google Cloud, you can import the datasets from your Google Drive. Execute the following script. And click on the link that appears, as shown below:

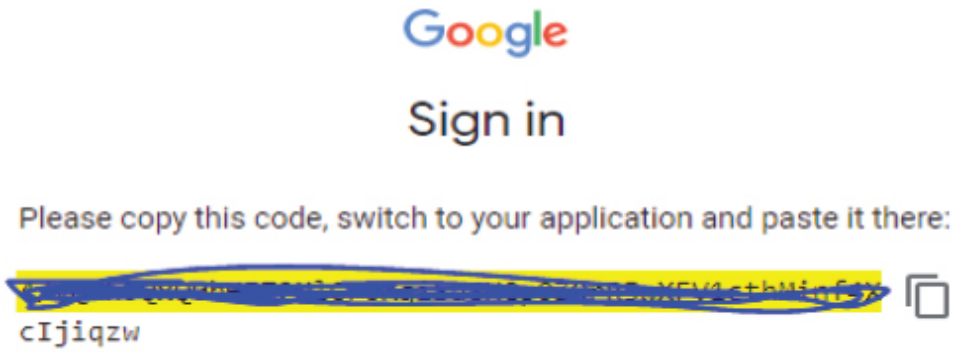
```
from google.colab import drive
drive.mount('/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth

Enter your authorization code:
```

You will be prompted to allow Google Colab to access your Google Drive. Click *Allow* button, as shown below:



You will see a link appear, as shown in the following image (the link has been blinded here).



Copy the link and paste it in the empty field in the Google Colab cell, as shown below:

```
from google.colab import drive
drive.mount('/gdrive')
```

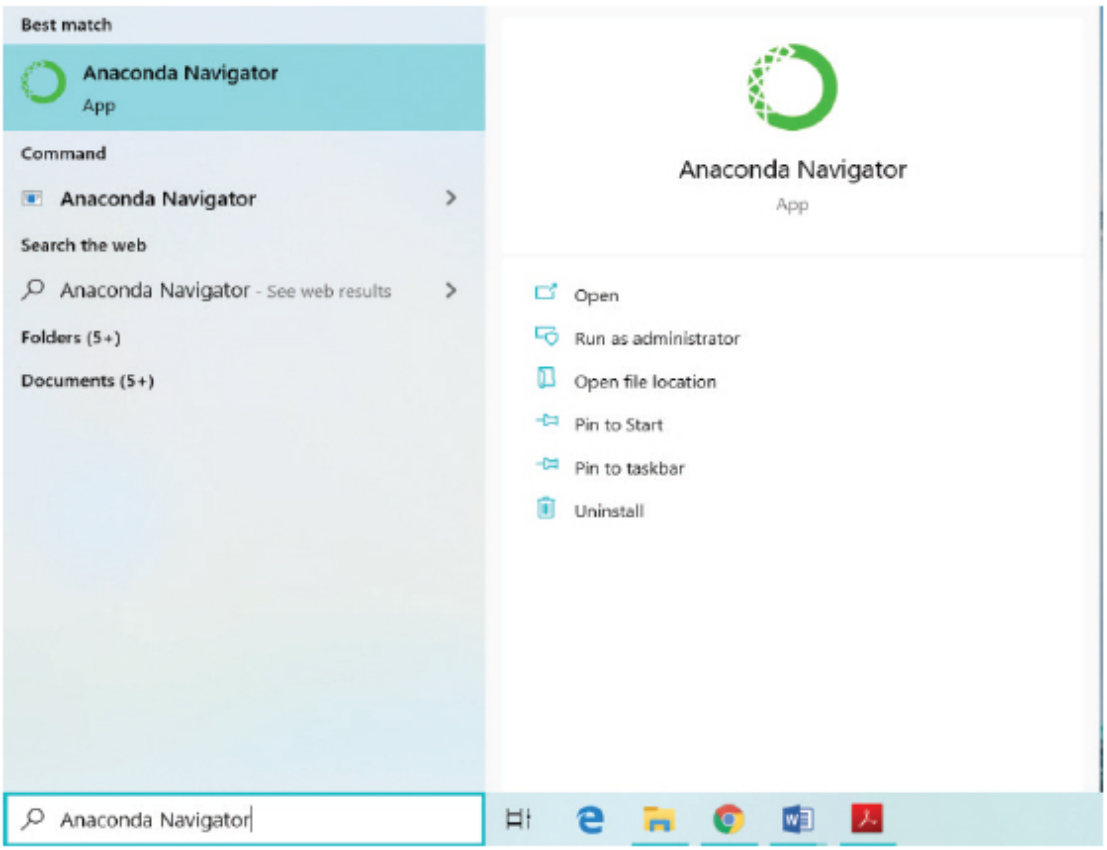
```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth

Enter your authorization code:
```

This way, you can import datasets from your Google Drive to your Google Colab environment.

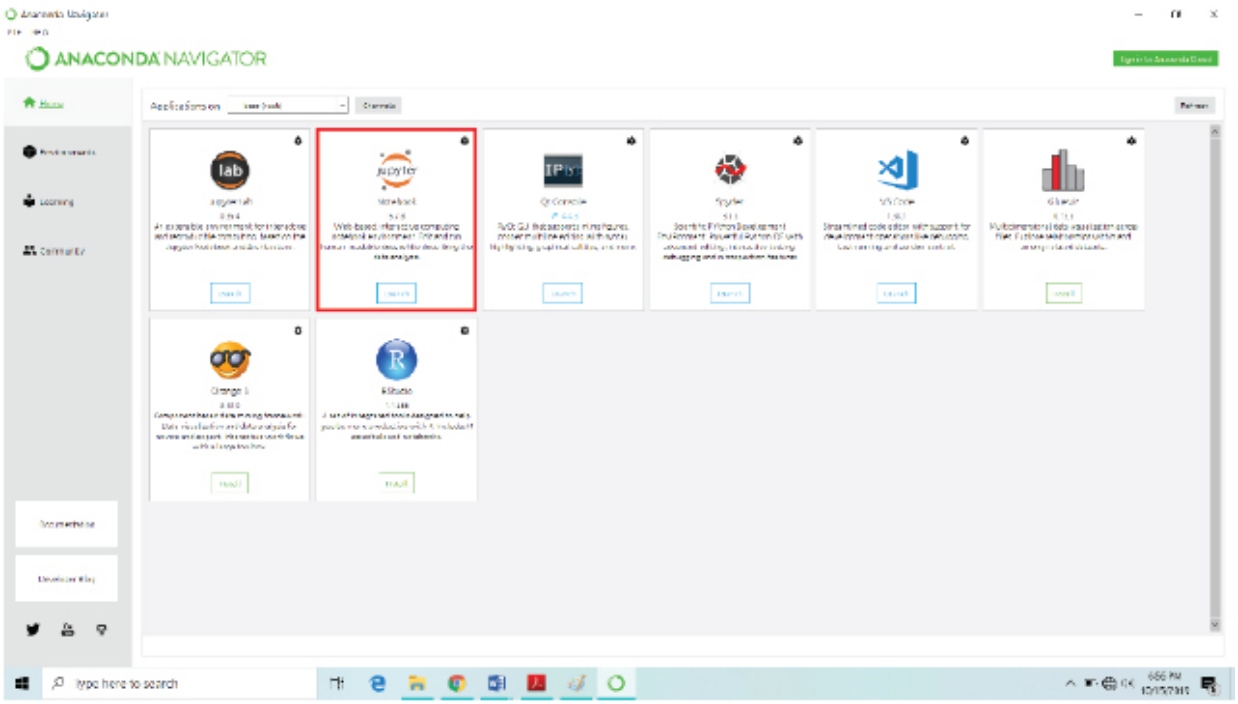
1.2.5. Writing Your First Program

You have installed Python on your computer now and established a distinctive environment in the form of Anaconda. It's now time to write your first program, i.e., the Hello World!

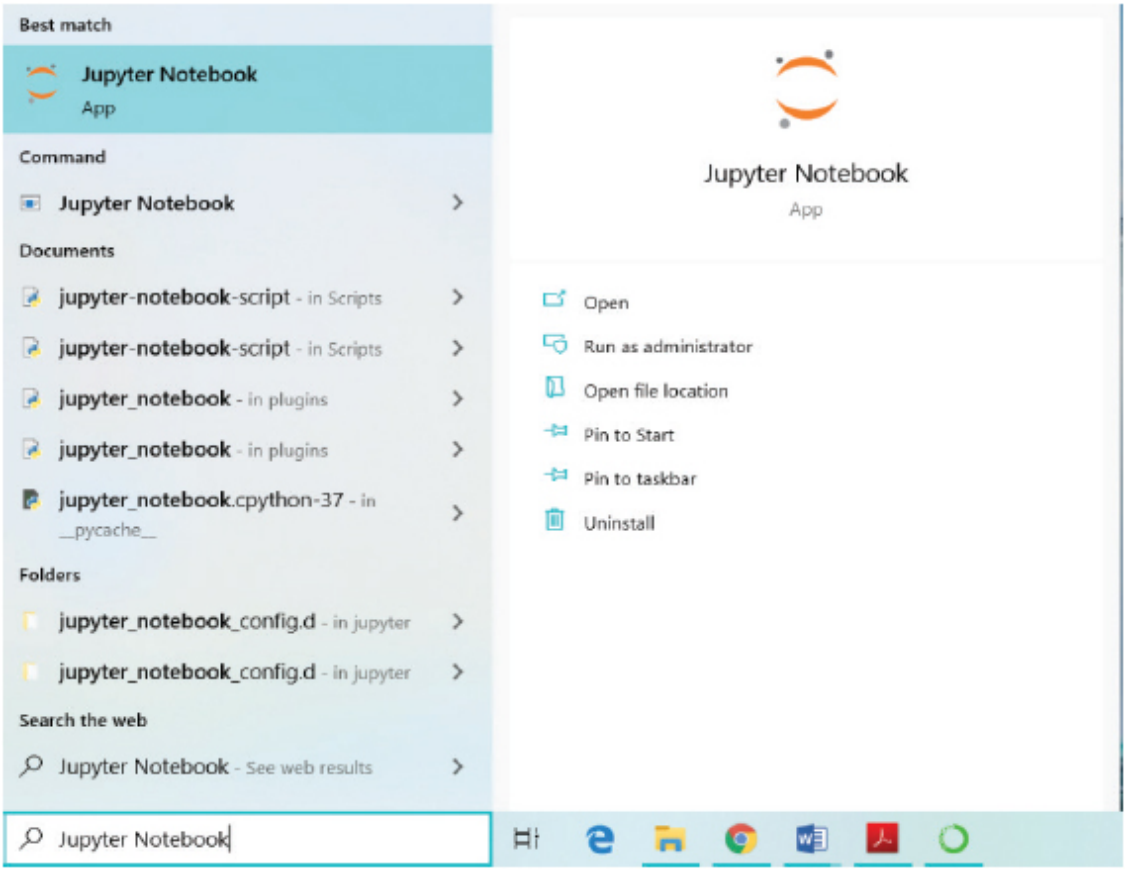
Start by launching the Anaconda Navigator. First, key in "Anaconda Navigator" in your Windows search box. Next, as shown in the following figure, click on the Anaconda Navigator application icon.



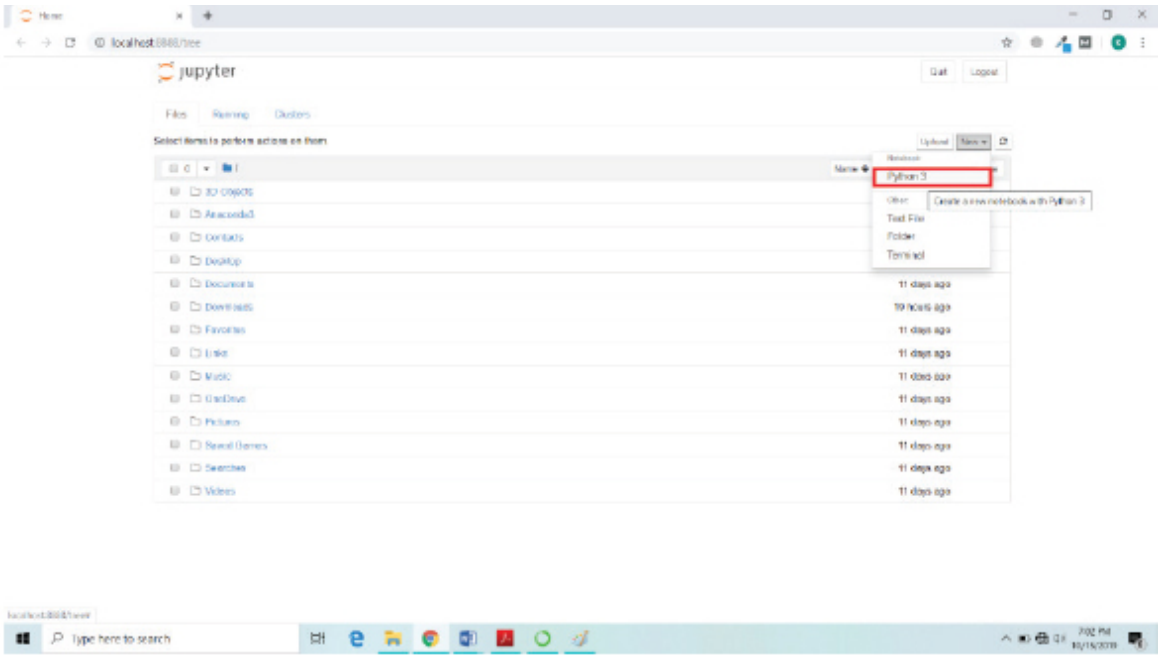
Anaconda's dashboard will open once you click on the application. The dashboard offers you an assortment of tools to write your code. We will use Jupyter Notebook, the most popular of these tools, to write and explain the code throughout this book.



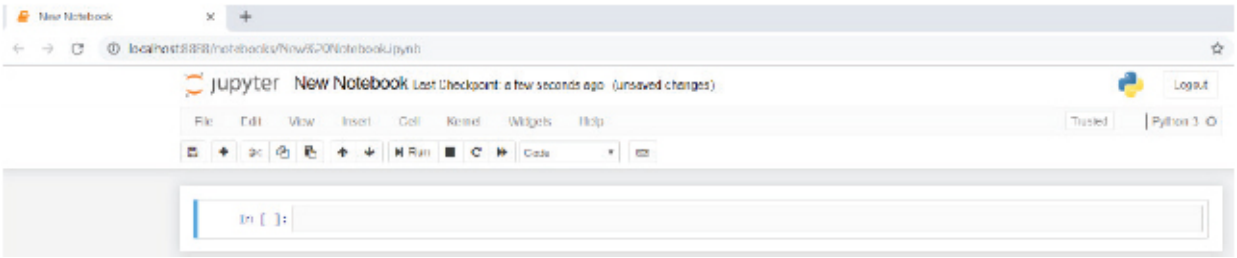
Jupyter Notebook is available in the second position from the top of the dashboard. The key feature of Jupyter Notebook is you can use it even if you don’t have internet access, as it runs right in your default browser. Another method to open Jupyter Notebook is to type Jupyter Notebook in the Windows search bar. Subsequently, click on the Jupyter Notebook application. The application will open in a new tab on your browser.



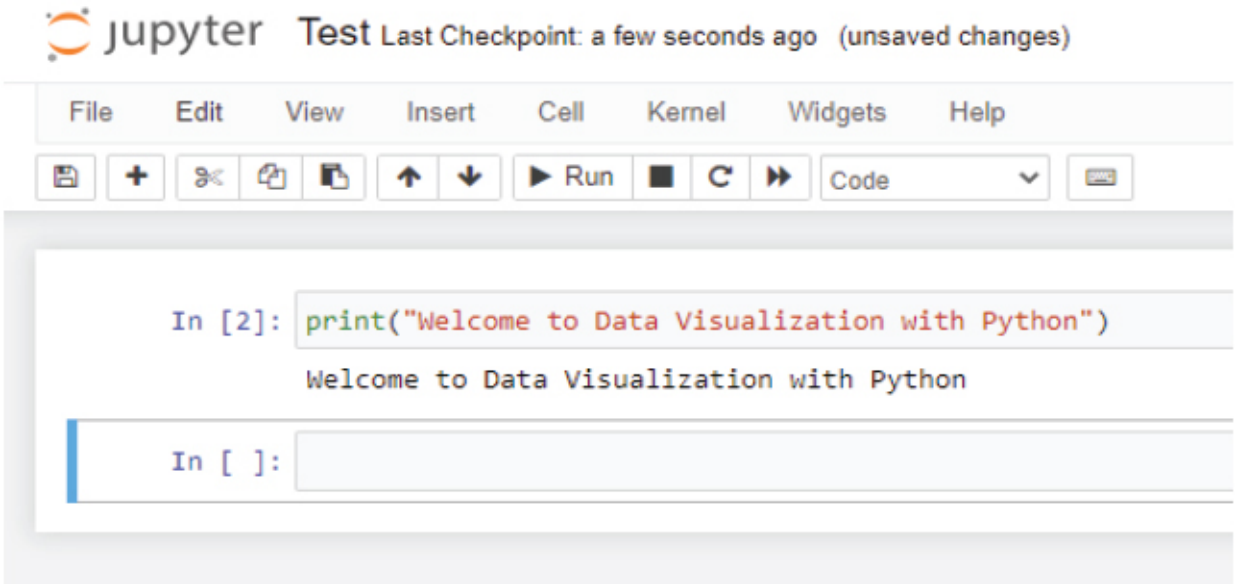
The top right corner of Jupyter Notebook's own dashboard houses a **New** button, which you have to click to open a new document. A dropdown containing several options will appear. Click on *Python 3* .



A new Python notebook will appear for you to write your programs. It looks as follows.



Jupyter Notebook consists of cells, as evident from the above image, making its layout very simple and straightforward. You will write your code inside these cells. Let us write our first ever Python program in Jupyter Notebook.



The above script prints a string value in the output using the **print()** method. The **print()** method is used to print any string passed to it on the console. If you see the following output, you have successfully run your first Python program.

Output:

```
Welcome to Data Visualization with Python
```

1.3. Python Crash Course

In this section, you will see a very brief introduction to various Python concepts. You can skip this section if you are already proficient with basic Python concepts. But if you are new to Python, this section can serve as a basic intro to Python.

Note: Python is a vast language with a myriad of features. This section doesn't serve as your complete guide to Python but merely helps get your feet wet with Python. To learn more about Python, you may check its official documentation, for which the link is given at the end of this section.

1.3.1. Python Syntax

Syntax of a language is a set of rules that the developer or the person writing the code must follow for the successful execution of code. Just like natural languages such as English have grammar and spelling rules, programming languages have their own rules.

Let's see some basic Python syntax rules.

Keywords

Every programming language has a specific set of words that perform specific functions and cannot be used as a variable or identifier. Python has the following set of keywords:

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

For instance, the keyword *class* is used to create a new class in Python (we will see classes in detail in a later chapter). Furthermore, the *If* keyword creates an if condition. If you try to use any of these keywords as variables, you will see errors.

Python Statements

Statements in Python are the smallest unit of executable code. When you assign a value to an identifier, you basically write a statement. For example,

age = 10 is a Python statement. When Python executes this statement, it assigns a value of 10 to the age identifier.

```
age = 10
print(age)
```

The script above has two statements. The first statement assigns a value of 10 to the age identifier. The second statement prints the age identifier.

If your statements are too long, you can span them by enclosing them in parenthesis, braces, or brackets, as shown below:

```
message = ('This is a message '
"it spans multiple lines" )

print(message)
```

Output:

```
This is a message it spans multiple lines
```

Another way to write a statement on multiple lines is by adding a backslash (\) at the end of the first line. Look at the following script:

```
message = 'This is a message ' \
"it spans multiple lines"

print(message)
```

The output is the same as that of the previous script.

Indentation

Indentation is one of those features that distinguish Python from other advanced programming languages such as C++, Java, and C#. In other programming languages, normally, braces ({) are used to define a block of code.

Indentation is used to define a new block of code in Python. A block of code in Python is a set of Python statements that execute together. You will see blocks in action when you study loops and conditional statements.

To define a new block, you have to indent the Python code, one tab (or four spaces) from the left.

```
age = 8
    if age <10 :
    print("Age is less than 10" )
        print("You do not qualify" )
    else:
        print("Age is greater than or equal to 10" )
        print("You do qualify" )
```

Output:

```
Age is less than 10
You do not qualify
```

In the above code, we define an identifier **age** with a value of 8. We then use the *if* statement and check if the age is less than or not. If age is less than 10, then the first block of code executes, which prints two statements on the console. You can see that code blocks have been indented.

Comments

Comments are used to add notes to a program. Comments do not execute, and you don't have to declare them in the form of statements. Comments are used to explain the code so that if you take a look at the code after a long time, you understand what you did.

Comments can be of two types: Single line comments and double-line comments. To add single line comments, you simply have to add #, as shown below:

```
# The following statement adds two numbers

num = 10 + 20 # the result is 30
```

To add multiline comments, you just need to add a # at the start of every line, as shown below:

```
#This is comment 1
#This is comment 2
#This is comment 3
```

1.3.2. Python Variables and Data Types

Data types in a programming language refer to the type of data that the language is capable of processing. The following are the major data types supported by Python:

a. Strings
b. Integers
c. Floating Point Numbers
d. Booleans
e. Lists
f. Tuples
g. Dictionaries

A variable is an alias for the memory address where actual data is stored. The data or the values stored at a memory address can be accessed and updated via the variable name. Unlike other programming languages like C++, Java, and C#, Python is loosely typed, which means that you don't have to define the data type while creating a variable. Instead, the type of data is evaluated at runtime.

The example below demonstrates how to create different data types and how to store them in their corresponding variables. The script also prints the type of the variables via the **type()** function.

Script 1:

```
# A string Variable
```

```
    first_name = "Joseph"
    print(type(first_name))

# An Integer Variable
    age = 20
    print(type(age))

# A floating point variable
    weight = 70.35
    print(type(weight))

# A Boolean variable
    married = False
    print(type(married))

#List
    cars = ["Honda" , "Toyota" , "Suzuki" ]
    print(type(cars))

#Tuples
    days = ("Sunday" , "Monday" , "Tuesday" , "Wednesday" , "Thursday" , "Friday" , "Saturday" )
    print(type(days))

#Dictionaries
    days2 = {1 :"Sunday" , 2: "Monday" , 3: "Tuesday" , 4: "Wednesday" , 5 :"Thursday" , 6 :"Friday" ,
    7 :"Saturday" }
    print(type(days2))
```

Output:

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

1.3.3. Python Operators

Python programming language contains the following types of operators:

a. Arithmetic Operators

b. Logical Operators

c. Comparison Operators

d. Assignment Operators

e. Membership Operators

Let's briefly review each of these types of operators.

Arithmetic Operators

Arithmetic operators are used to execute arithmetic operations in Python. The following table summarizes the arithmetic operators supported by Python. Suppose X = 20 and Y = 10.

Operator Name	Symbol	Functionality	Example
Addition	+	Adds the operands on either side	X+ Y= 30
Subtraction	–	Subtracts the operands on either side	X –Y= 10
Multiplication	*	Multiplies the operands on either side	X * Y= 200
Division	/	Divides the operand on the left by the one on the right	X / Y= 2.0
Modulus	%	Divides the operand on the left by the one on the right and returns the remainder	X % Y= 0
Exponent	**	Takes exponent of the operand on the left to the power of right	X ** Y = $1024 \times e^{10}$

Here is an example of arithmetic operators with output:

Script 2:

```
X = 20
Y = 10
print(X + Y)
print(X - Y)
print(X * Y)
print(X / Y)
print(X ** Y)
```

Output:

```
30
10
200
2.0
10240000000000
```

Logical Operators

Logical operators are used to perform logical AND, OR, and NOT operations in Python. The following table summarizes the logical operators. Here, X is True, and Y is False.

Operator	Symbol	Functionality	Example
Logical AND	and	If both the operands are true, then the condition becomes true.	(X and Y) = False
Logical OR	or	If any of the two operands are true, then the condition becomes true.	(X or Y) = True
Logical NOT	not	Used to reverse the logical state of its operand.	not(X and Y) = True

Here is an example that explains the usage of the Python logical operators.

Script 3:

```
X = True
Y = False
print(X and Y)
print(X or Y)
print(not(X and Y))
```

Output:

```
False
True
True
```

Comparison Operators

Comparison operators, as the name suggests, are used to compare two or more than two operands. Depending upon the relation between the operands,

comparison operators return Boolean values. The following table summarizes comparison operators in Python. Here, X is 20, and Y is 35.

Operator	Symbol	Description	Example
Equality	==	Returns true if values of both the operands are equal	(X == Y) = false
Inequality	!=	Returns true if values of both the operands are not equal	(X = Y) = true
Greater than	>	Returns true if the value of the left operand is greater than the right one	(X> Y) = False
Smaller than	<	Returns true if the value of the left operand is smaller than the right one	(X< Y) = True
Greater than or equal to	>=	Returns true if the value of the left operand is greater than or equal to the right one	(X > =Y) = False
Smaller than or equal to	<=	Returns true if the value of the left operand is smaller than or equal to the right one	(X<= Y) = True

The comparison operators have been demonstrated in action in the following example:

Script 4

```
X = 20
Y = 35

print(X == Y)
```

```
print(X != Y)
print(X > Y)
print(X < Y)
print(X >= Y)
print(X <= Y)
```

Output:

```
False
True
False
True
False
True
```

Assignment Operators

Assignment operators are used to assign values to variables. The following table summarizes the assignment operators. Here, X is 20, and Y is equal to 10.

Operator	Symbol	Description	Example
Assignment	=	Used to assign the value of the right operand to the left operand	R = X+ Y assigns 30 to R
Add and assign	+=	Adds the operands on either side and assigns the result to the left operand	X += Y assigns 30 to X
Subtract and assign	−=	Subtracts the operand on the right from the operand on the left and assigns the result to the left operand	X −= Y assigns 10 to X
Multiply and Assign	*=	Multiplies the operands on either side and assigns the result to the left operand	X *= Y assigns 200 to X
Divide and Assign	/=	Divides the operands on the left by the right and assigns the result to the left operand	X/= Y assigns 2 to X
Take modulus and assign	%=	Divides the operands on the left by the right and assigns the remainder to the left operand	X %= Y assigns 0 to X
Take exponent and assign	**=	Takes exponent of the operand on the left to the power of right and assigns the remainder to the left operand	X **= Y assigns 1024 x e^{10} to X

Take a look at the script below to see Python assignment operators in action.

Script 5:

```
X = 20; Y = 10
R = X + Y
print(R)

X = 20;
Y = 10
X += Y
print(X)

X = 20;
Y = 10
X -= Y
print(X)

X = 20;
Y = 10
X *= Y
print(X)

X = 20;
Y = 10
X /= Y
print(X)

X = 20;
Y = 10
X %= Y
print(X)

X = 20;
Y = 10
X **= Y
print(X)
```

Output:

```
30
30
10
200
2.0
0
10240000000000
```

Membership Operators

Membership operators are used to find if an item is a member of a collection of items or not. There are two types of membership operators: the **in** operator and the **not in** operator. The following script shows **an** operator in action.

Script 6:

```
days = ("Sunday" , "Monday" , "Tuesday" , "Wednesday" , "Thursday" , "Friday" , "Saturday" )
print('Sunday' in days)
```

Output:

```
True
```

And here is an example of the **not in** operator.

Script 7:

```
days = ("Sunday" , "Monday" , "Tuesday" , "Wednesday" , "Thursday" , "Friday" , "Saturday" )
print('Xunday' not in days)
```

Output:

```
True
```

1.3.4. Conditional Statements

Conditional statements in Python are used to implement conditional logic in Python. Conditional statements help you decide whether to execute a certain code block or not. There are three main types of conditional statements in Python:

a. If statement

b. If-else statement

c. If-elif statement

IF Statement

If you have to check for a single condition and you do not concern yourself about the alternate condition, you can use the **if** statement. For instance, if you want to check if 10 is greater than 5 and based on that you want to print a statement, you can use the if statement. The condition evaluated by the **if** statement returns a Boolean value. If the condition evaluated by the **if** statement is true, the code block that follows the **if** statement executes. It is important to mention that in Python, a new code block starts at a new line with a tab indented from the left when compared with the outer block.

In the following example, the condition 10 > 5 is evaluated, which returns true. Hence, the code block that follows the **if** statement executes, and a message is printed on the console.

Script 8:

```
# The if statement

    if 10 > 5 :
        print("Ten is greater than 10" )
```

Output:

```
Ten is greater than 10
```

IF-Else Statement

The **If-else** statement comes in handy when you want to execute an alternate piece of code in case the condition for the *if* statement returns false. For instance, in the following example, the condition 5 < 10 will return false. Hence, the code block that follows the **else** statement will execute.

Script 9:

```
# if-else statement

    if 5 > 10 :
        print("5 is greater than 10" )
    else:
        print("10 is greater than 5" )
```

Output:

```
10 is greater than 5
```

IF-Elif Statement

The **if-elif** statement comes handy when you have to evaluate multiple conditions. For instance, in the following example, we first check if 5 > 10, which evaluates to false. Next, an **elif** statement evaluates the condition 8 < 4, which also returns false. Hence, the code block that follows the last **else** statement executes.

Script 10:

```
#if-elif and else

    if 5 > 10 :
        print("5 is greater than 10" )
    elif 8 < 4 :
        print("8 is smaller than 4" )
    else :
        print("5 is not greater than 10 and 8 is not smaller than 4" )
```

Output:

```
5 is not greater than 10 and 8 is not smaller than 4
```

1.3.5. Iteration Statements

Iteration statements, also known as loops, are used to iteratively execute a certain piece of code. There are two main types of iteration statements in

Python.

a. For loop
b. While Loop

For Loop

The **for loop** is used to iteratively execute a piece of code a certain number of times. You should use **for loop** when you exactly know the number of iterations or repetitions for which you want to run your code. A **for loop** iterates over a collection of items. In the following example, we create a collection of five integers using the **range()** method. Next, a **for loop** iterates five times and prints each integer in the collection.

Script 11:

```
items = range(5 )
    for item in items:
        print(item)
```

Output:

```
0
1
2
3
4
```

While Loop

The **while loop** keeps executing a certain piece of code unless the evaluation condition becomes false. For instance, the **while loop** in the following script keeps executing unless variable c becomes greater than 10.

Script 12:

```
c = 0
    while c < 10 :
```

```
        print(c)
        c = c +1
```

Output:

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

1.3.6. Functions

Functions, in any programming language, are used to implement a piece of code that is required to be executed multiple times at different locations in the code. In such cases, instead of writing long pieces of code, again and again, you can simply define a function that contains the piece of code, and then you can call the function wherever you want in the code.

To create a function in Python, the *def* keyword is used, followed by the name of the function and opening and closing parenthesis.

Once a function is defined, you have to call it to execute the code inside a function body. To call a function, you simply have to specify the name of the function followed by opening and closing parenthesis. In the following script, we create a function named **myfunc** which prints a simple statement on the console using the **print()** method.

Script 13:

```
def myfunc():
        print("This is a simple function" )

    ### function call
    myfunc()
```

Output:

```
This is a simple function
```

You can also pass values to a function. The values are passed inside the parenthesis of the function call. However, you must specify the parameter name in the function definition, too. In the following script, we define a function named **myfuncparam()** . The function accepts one parameter, i.e., **num** . The value passed in the parenthesis of the function call will be stored in this **num** variable and will be printed by the **print()** method inside the **myfuncparam()** method.

Script 14:

```
def myfuncparam(num):
    print("This is a function with parameter value: "+num )

### function call
    myfuncparam("Parameter 1" )
```

Output:

```
This is a function with parameter value:Parameter 1
```

Finally, a function can also return values to the function call. To do so, you simply have to use the return keyword followed by the value that you want to return. In the following script, the **myreturnfunc()** function returns a string value to the calling function.

Script 15:

```
def myreturnfunc():
        return "This function returns a value"

    val = myreturnfunc()
    print(val)
```

Output:

```
This function returns a value
```

1.3.7. Objects and Classes

Python supports object-oriented programming (OOP). In OOP, any entity that can perform some function and have some attributes is implemented in the form of an object.

For instance, a car can be implemented as an object since a car has some attributes such as price, color, model and can perform some functions such as drive car, change gear, stop car, etc.

Similarly, a fruit can also be implemented as an object since a fruit has a price, name and you can eat a fruit, grow a fruit, and perform functions with a fruit.

To create an object, you first have to define a class. For instance, in the following example, a class **Fruit** has been defined. The class has two attributes **name** and **price** and one method, **eat_fruit()** . Next, we create an object **f** of class Fruit and then call the **eat_fruit()** method from the **f** object. We also access the **name** and **price** attributes of the **f** object and print them on the console.

Script 16:

```
class Fruit:

        name = "apple"
        price = 10

        def eat_fruit(self):
            print("Fruit has been eaten" )

    f = Fruit()
    f.eat_fruit()
    print(f.name)
    print(f.price)
```

Output:

```
Fruit has been eaten
apple
10
```

A class in Python can have a special method called a *constructor* . The name of the constructor method in Python is **__init__().** The constructor is called whenever an object of a class is created. Look at the following example to see the constructor in action.

Script 17:

```
class Fruit:

        name = "apple"
        price = 10

        def __init__(self, fruit_name, fruit_price):
            Fruit.name = fruit_name
            Fruit.price = fruit_price

        def eat_fruit(self) :
            print("Fruit has been eaten" )

    f = Fruit("Orange" , 15 )
    f.eat_fruit()
    print(f.name)
    print(f.price)
```

Output:

```
Fruit has been eaten
Orange
15
```

Further Readings - Python [1]

To study more about Python, please check Python 3 Official Documentation (https://bit.ly/3rfaLke). Get used to searching and reading this documentation. It is a great resource of knowledge.

Hands-on Time - Exercise

Now, it is your turn. Follow the instructions in **the exercises below** to check your understanding of the basic Python concepts. The answers to these questions are given at the end of the book.

Exercise 1.1

Question 1

Which iteration should be used when you want to repeatedly execute a code specific number of times?

A. For Loop

B. While Loop

C. Both A & B

D. None of the above

Question 2

What is the maximum number of values that a function can return in Python?

A. Single Value

B. Double Value

C. More than two values

D. None

Question 3

Which of the following membership operators are supported by Python?

A. In

B. Out

C. Not In

D. Both A and C

Exercise 1.2

Print the table of integer 9 using a while loop.

Pandas Basics

In this chapter, you will see a brief introduction to the Pandas series and Dataframes, which are two basic data structures for storing data in Pandas. Next, you will see how to create these data structures and some basic functions that you can perform with Pandas. You will then study how to import datasets into a Pandas dataframe using various input sources. Finally, the chapter concludes with an explanation of the techniques for handling missing data in Pandas dataframes.

Pandas comes installed with default Python installation. You can also install Pandas via the following PIP command:

```
pip install pandas
```

2.1. Pandas Series

A Pandas series is a data structure that stores data in the form of a column. A series is normally used to store information about a particular attribute in your dataset. Let's see how you can create a series in Pandas.

2.1.1. Creating Pandas Series

There are different ways to create a series with Pandas. The following script imports the Pandas module and then calls the Series() class constructor to create an empty series. Here is how to do that:

Script 1:

```
# empty series
```

```
import pandas as pd

my_series = pd.Series()
print (my_series)
```

You can also create a series using a NumPy array. But, first, you need to pass the array to the Series() class constructor, as shown in the script below.

Script 2:

```
# series using numpy array

import pandas as pd
import numpy as np

my_array = np.array([ 10 , 20 , 30 , 40 , 50 ])

my_series = pd.Series(my_array)
print (my_series)
```

Output:

```
0 10
1 20
2 30
3 40
4 50
dtype: int32
```

In the above output, you can see that the indexes for a series start from 0 to 1 less than the number of items in the series. You can also define custom indexes for your series. To do so, you need to pass your list of indexes to the index attribute of the Series class, as shown in the script below:

Script 3:

```
# series with custom indexes

import pandas as pd
import numpy as np

my_array = np.array([ 10 , 20 , 30 , 40 , 50 ])
```

```
my_series = pd.Series(my_array, index = ["num1", "num2", "num3", "num4", "num5"])
print (my_series)
```

Output:

```
num1 10
num2 20
num3 30
num4 40
num5 50
dtype: int32
```

You can also create a series by directly passing a Python list to the Series() class constructor.

Script 4:

```
# series using a list

import pandas as pd
import numpy as np

my_series = pd.Series([ 10 , 20 , 30 , 40 , 50 ], index = ["num1", "num2", "num3", "num4", "num5"])
print (my_series)
```

Output:

```
num1 10
num2 20
num3 30
num4 40
num5 50
dtype: int64
```

Finally, a scaler value can also be used to define a series. In case you pass a list of indexes, the scaler value will be repeated the number of times equal to the items in the index list. Here is an example:

Script 5:

```
# series using a scaler

import pandas as pd
import numpy as np

my_series = pd.Series( 25 , index = ["num1", "num2", "num3", "num4", "num5"])
print (my_series)
```

Output:

```
num1 25
num2 25
num3 25
num4 25
num5 25
dtype: int64
```

Finally, you can also create a series using a dictionary. In this case, the dictionary keys will become series indexes while the dictionary values are inserted as series items. Here is an example:

Script 6:

```
# series using dictionary

my_dict = {'num1' : 6 ,
'num2' : 7 ,
'num3' : 8 }

my_series = pd.Series(my_dict)
print (my_series)
```

Output:

```
num1 6
num2 7
num3 8
dtype: int64
```

7.1.2. Useful Operations on Pandas Series

Let's see some of the useful operations you can perform with the Pandas series.

You can use square brackets as well as index labels to access series items, as shown in the following script:

Script 7:

```
## Accessing Items

import pandas as pd
my_series = pd.Series([ 10 , 20 , 30 , 40 , 50 ], index = ["num1", "num2", "num3", "num4", "num5"])
print (my_series[ 0 ])
print (my_series['num3'])
```

Output:

```
10
30
```

Using the min() and max() functions from the NumPy module, you can find the maximum and minimum values, respectively, from a series. Look at the following script for reference.

Script 8:

```
## Finding Maximum and Minimum Values

import pandas as pd
import numpy as np

my_series = pd.Series([ 5 , 8 , 2 , 11 , 9 ])

print (np.min(my_series))
print (np.max(my_series))
```

Output:

```
2
11
```

Similarly, the mean() method from the NumPy module can find the mean of a Pandas series, as shown in the following script.

Script 9:

```
## Finding Mean

import pandas as pd
import numpy as np

my_series = pd.Series([ 5 , 8 , 2 , 11 , 9 ])

print (my_series.mean())
```

Output:

```
7.0
```

The following script finds the median value of a Pandas series.

Script 10:

```
## Finding Median

import pandas as pd
import numpy as np

my_series = pd.Series([ 5 , 8 , 2 , 11 , 9 ])

print (my_series.median())
```

Output:

```
8.0
```

You can also find the data type of a Pandas series using the dtype attribute. Here is an example:

Script 11:

```
## Finding Data Type

import pandas as pd
import numpy as np

my_series = pd.Series([ 5 , 8 , 2 , 11 , 9 ])

print (my_series.dtype)
```

Output:

```
int64
```

A Pandas series can also be converted to a Python list using the tolist() method, as shown in the script below:

Script 12:

```
## Converting to List

import pandas as pd
import numpy as np

my_series = pd.Series([ 5 , 8 , 2 , 11 , 9 ])

print (my_series.tolist())
```

Output:

```
[5, 8, 2, 11, 9]
```

2.2. Pandas Dataframe

A Pandas dataframe is a tabular data structure that stores data in the form of rows and columns. As a standard, the rows correspond to records while columns refer to attributes. In simplest words, a Pandas dataframe is a collection of series.

2.2.1. Creating a Pandas Dataframe

As is the case with a series, there are multiple ways to create a Pandas dataframe.

To create an empty dataframe, you can use the DataFrame class from the Pandas module, as shown below:

Script 13:

```
# empty pandas dataframe

import pandas as pd

my_df = pd.DataFrame() print (my_df)
```

Output:

```
EmptyDataFrame
Columns: []
Index: []
```

You can create a Pandas dataframe using a list of lists. Each sublist in the outer list corresponds to a row in a dataframe. Each item within a sublist becomes an attribute value.

To specify column headers, you need to pass a list of values to the columns attribute of DataFrame class.

Here is an example of how you can create a Pandas dataframe using a list.

Script 14:

```
# dataframe using list of lists

import pandas as pd

scores = [['Mathematics', 85 ], ['English', 91 ], ['History', 95 ]]

my_df = pd.DataFrame(scores, columns = ['Subject', 'Score'])
```

```
my_df
```

Output:

	Subject	Score
0	Mathematics	85
1	English	91
2	History	95

Similarly, you can create a Pandas dataframe using a dictionary. One of the ways is to create a dictionary where keys correspond to column headers. In contrast, corresponding dictionary values are a list, which corresponds to the column values in the Pandas dataframe.

Here is an example for your reference:

Script 15:

```
# dataframe using dictionaries

import pandas as pd

scores = {'Subject':["Mathematics", "History", "English", "Science", "Arts"],
'Score':[ 98 , 75 , 68 , 82 , 99 ]
    }

my_df = pd.DataFrame(scores)
my_df
```

Output:

	Subject	Score
0	Mathematics	98
1	History	75
2	English	68
3	Science	82
4	Arts	99

Another way to create a Pandas dataframe is using a list of dictionaries. Each dictionary corresponds to one row. Here is an example of how to do that.

Script 16:

```
# dataframe using list of dictionaries

import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 },
    {'Subject':'History', 'Score': 98 },
    {'Subject':'English', 'Score': 76 },
    {'Subject':'Science', 'Score': 72 },
    {'Subject':'Arts', 'Score': 95 },
    ]

my_df = pd.DataFrame(scores)
my_df
```

Output:

	Subject	Score
0	Mathematics	85
1	History	98
2	English	76
3	Science	72
4	Arts	95

The dictionaries within the list used to create a Pandas dataframe need not be of the same size.

For example, in the script below, the fourth dictionary in the list contains only one item, unlike the rest of the dictionaries in this list. The corresponding dataframe will contain a null value in place of the second item, as shown in the output of the script below:

Script 17:

```
# dataframe using list of dictionaries
# with null items

import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 },
    {'Subject':'History', 'Score': 98 },
    {'Subject':'English', 'Score': 76 },
    {'Score': 72 },
    {'Subject':'Arts', 'Score': 95 },
    ]

my_df = pd.DataFrame(scores)
my_df
```

Output:

	Subject	Score
0	Mathematics	85
1	History	98
2	English	76
3	NaN	72
4	Arts	95

2.2.2. Basic Operations on Pandas Dataframe

Let's now see some of the basic operations that you can perform on Pandas dataframes.

To view the top(N) rows of a dataframe, you can call the head() method, as shown in the script below:

Script 18:

```
# viewing header

import pandas as pd

scores = [
     {'Subject':'Mathematics', 'Score': 85 },
     {'Subject':'History', 'Score': 98 },
     {'Subject':'English', 'Score': 76 },
     {'Subject':'Science', 'Score': 72 },
     {'Subject':'Arts', 'Score': 95 },
     ]

my_df = pd.DataFrame(scores)
my_df.head( 2 )
```

Output:

	Subject	Score
0	Mathematics	85
1	History	98

To view the last N rows, you can use the tail() method. Here is an example:

Script 19:

```
# viewing tail

my_df = pd.DataFrame(scores)

my_df.tail( 2 )
```

Output:

	Subject	Score
3	Science	72
4	Arts	95

You can also get a summary of your Pandas dataframe using the info() method.

Script 20:

```
# gettingdataframe info

my_df = pd.DataFrame(scores)

my_df.info()
```

In the output below, you can see the number of entries in your Pandas dataframe, the number of columns along with their column type, and so on.

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Subject  5 non-null      object
 1   Score    5 non-null      int64
dtypes: int64(1), object(1)
memory usage: 208.0+ bytes
```

Finally, to get information such as mean, minimum, maximum, standard deviation, etc., for numeric columns in your Pandas dataframe, you can use the describe() method, as shown in the script below:

Script 21:

```
# getting info about numeric columns

my_df = pd.DataFrame(scores)
```

```
my_df.describe()
```

Output:

	Score
count	5.000000
mean	85.200000
std	11.388591
min	72.000000
25%	76.000000
50%	85.000000
75%	95.000000
max	98.000000

2.3. Importing Data in Pandas

You can import data from various sources into your Pandas dataframe. Some of them are discussed in this section.

2.3.1. Importing CSV Files

A CSV file is a type of file where each line contains a single record, and all the columns are separated from each other via a comma.

You can read CSV files using the **read_csv()** function of the Pandas dataframe, as shown below. The "iris_data.csv" file is available in the *Data* folder of the book resources.

Script 22:

```
import pandas as pd
titanic_data = pd.read_csv(r"D:\Datasets\iris_data.csv")
titanic_data.head()
```

If you print the dataframe header, you should see that the header contains five columns that contain different information about iris plants.

Output:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In some cases, CSV files do not contain any header. In such cases, the read_csv() method treats the first row of the CSV file as the dataframe header.

To specify custom headers for your CSV files, you need to pass the list of headers to the names attribute of the read_ csv() method, as shown in the script below. You can find the "pima-indians-diabetes.csv" file in the *Data* folder of the book resources.

Script 23:

```
headers = ["Preg", "Glucose", "BP", "skinThick", "Insulin", "BMI", "DPF", "Age", "Class"]

patient_data = pd.read_csv(r"D:\Datasets\pima-indians- diabetes.csv", names = headers)

patient_data.head()
```

In the output below, you can see the custom headers that you passed in the list to the read_csv() method's name attribute.

Output:

	Preg	Glucose	BP	skinThick	Insulin	BMI	DPF	Age	Class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

2.3.2. Importing TSV Files

TSV files are similar to CSV files. But in a TSV file, the delimiter used to separate columns is a single tab. The **read_csv()** function can be used to read a TSV file. However, you have to pass "\t" as a value for the "sep" attribute, as shown below.

Note: You can find the "iris_data.tsv" file in the *Data* folder of the book resources.

Script 24:

```
import pandas as pd
patients_csv = pd.read_csv(r"D:\Datasets\iris_data.tsv", sep=' \t ')
patients_csv.head()
```

Output:

	Preg	Glucose	BP	skinThick	Insulin	BMI	DPF	Age	Class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

2.3.3. Importing Data from Databases

Oftentimes, you need to import data from different databases into your Pandas dataframe. In this section, you will see how to import data from various databases into a Python application.

Importing Data from SQL Server

To import data from Microsoft's SQL Server database, you need to first install the "pyodbc" module for Python. To do so, execute the following command on your command terminal.

```
$ pip install pyodbc
```

Next, you need to create a connection with your SQL server database. The **connect()** method of the "pyodbc" module can be used to create a connection. You have to pass the driver name, the server name, and the database name to the connect() method, as shown below.

Note: To run the following script, you need to create a database named *Titanic* with a table named *records* table. Explaining how to create a database and tables is beyond the scope of this book. You find further details at the link below.

Further Readings – CRUD Operations with SQL Server

To see how to create databases and tables with SQL Server, take a look at this link: https://bit.ly/2XwEgAV

In the following script, we connect to the Titanic database.

Script 25:

```
import pandas as pd
import pyodbc

sql_conn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};
    SERVER=HOARE\SQLEXPRESS; DATABASE=Titanic; Trusted_Connection=yes')
```

Once the connection is established, you have to write an SQL SELECT query that fetches the desired record. The following SQL select query fetches all records from a *records* table.

In a Pandas dataframe, the query and the connection object are passed to the **pd_read_sql()** function to store the records returned by the query.

Finally, the dataframe header is printed to display the first five rows of the imported table.

Script 26:

```
query = "SELECT * FROM records;"
titanic_data = pd.read_sql(query, sql_conn)
titanic_data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	False	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.250000	None	S
1	2	True	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.283302	C85	C
2	3	True	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.925000	None	S
3	4	True	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.099998	C123	S
4	5	False	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.050000	None	S

Importing Data from PostgreSQL

To import data from PostgreSQL, you will need to download the *SQLAlchemy* module. Execute the following pip statement to do so.

```
$ pip install SQLAlchemy
```

Next, you need to create an engine, which serves as a connection between the PostgreSQL server and the Python application. The following script shows how to create a connection engine. You need to replace your server and database name in the following script.

Script 27:

```
from sqlalchemy import create_engine
```

```
engine = create_engine('postgresql://postgres:abc123@
    localhost:5432/Titanic')
```

To store the records returned by the query in a Pandas dataframe, the query and the connection object are passed to the **pd_read_sql()** function of the Pandas dataframe. Finally, the dataframe header is printed to display the first five rows of the imported table.

Script 28:

```
import pandas as pd
titanic_data =pd.read_sql_query('select * from "records"',con=engine)
titanic_data.head()
```

Output:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	False	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.250000	None	S
1	2	True	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.283302	C85	C
2	3	True	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.925000	None	S
3	4	True	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.099998	C123	S
4	5	False	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.050000	None	S

Further Readings – CRUD Operations with PostgreSQL

To see how to create databases and tables with PostgreSQL, take a look at this link: https://bit.ly/2XyJr3f

Importing Data from SQLite

To import data from an SQLite database, you do not need any external module. You can use the default *sqlite3* module.

The first step is to connect to an SQLite database. To do so, you can use the **connect()** method of the sqlite3 module, as shown below.

Script 29:

```
import sqlite3
import pandas as pd
# Create your connection.
cnx = sqlite3.connect('E:/Titanic.db')
```

Next, you can call the pd**_read_sql()** function of the Pandas dataframe and pass it to the SELECT query and the database connection. Finally, the dataframe header is printed to display the first five rows of the imported table.

Script 30:

```
titanic_data = pd.read_sql_query("SELECT * FROM records", cnx)
titanic_data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	False	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.250000	None	S
1	2	True	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.283302	C85	C
2	3	True	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.925000	None	S
3	4	True	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.099998	C123	S
4	5	False	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.050000	None	S

Further Readings – CRUD Operations with SQLite

To see how to create databases and tables with SQLite, take a look at this link: https://bit.ly/2BAXZXL

2.4. Handling Missing Values in Pandas

Missing values, as the name suggests, are those observations in the dataset that doesn't contain any value. Missing values can change the data patterns, and, therefore, it is extremely important to understand why missing values occur in the dataset and how to handle them.

In this section, you will see the different techniques with examples to handle missing values in your Pandas dataframes.

2.4.1. Handling Missing Numerical Values

To handle missing numerical data, we can use statistical techniques. The use of statistical techniques or algorithms to replace missing values with statistically generated values is called imputation.

In this section, you will see how to do median and mean imputation. Mean or median imputation is one of the most commonly used imputation techniques for handling missing numerical data.

In mean or median imputation, missing values in a column are replaced by the mean or median of all the remaining values in that particular column.

For instance, if you have a column with the following data:

Age
15
NA
20
25
40

In the above Age column, the second value is missing. Therefore, with mean and median imputation, you can replace the second value with either the mean or median of all the other values in the column. For instance, the following column contains the mean of all the remaining values, i.e., 25 in the second row. You could also replace this value with the median if you want.

Age
15
25
20
25
40

Let's see a practical example of mean and median imputation. First, we will import the Titanic dataset and find the columns that contain missing values. Then, we will apply mean and median imputation to the columns containing missing values. Finally, we will see the effect of applying mean and median imputation to the missing values.

You do not need to download the Titanic dataset. If you import the Seaborn library, the Titanic data will be downloaded with it. The following script imports the Titanic dataset and displays its first five rows.

Script 31:

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = [ 8 , 6 ]
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')

titanic_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

Let's filter some of the numeric columns from the dataset and see if they contain any missing values.

Script 32:

```
titanic_data = titanic_data[["survived", "pclass", "age", "fare"]]
titanic_data.head()
```

Output:

	survived	pclass	age	fare
0	0	3	22.0	7.2500
1	1	1	38.0	71.2833
2	1	3	26.0	7.9250
3	1	1	35.0	53.1000
4	0	3	35.0	8.0500

To find missing values from the aforementioned columns, you need to first call the **isnull()** method on the **titanic_data** dataframe, and then you need to call the **mean()** method, as shown below.

Script 33:

```
titanic_data.isnull().mean()
```

Output:

```
survived 0.000000
```

```
pclass 0.000000
age 0.198653
fare 0.000000
dtype: float64
```

The above output shows that only the *age* column contains missing values. And the ratio of missing values is around 19.86 percent.

Let's now find out the median and mean values for all the non- missing values in the *age* column.

Script 34:

```
median = titanic_data.age.median()
print (median)

mean = titanic_data.age.mean()
print (mean)
```

Output:

```
28.0
29.69911764705882
```

The age column has a median value of 28 and a mean value of 29.6991.

To plot the kernel density plots for the actual age and median and mean age, we will add columns to the Pandas dataframe.

Script 35:

```
import numpy as np

titanic_data['Median_Age'] = titanic_data.age.fillna(median)

titanic_data['Mean_Age'] = titanic_data.age.fillna(mean)

titanic_data['Mean_Age'] = np.round(titanic_data['Mean_Age'], 1 )

titanic_data.head( 20 )
```

The above script adds **Median_Age** and **Mean_Age** columns to the **titanic_data** dataframe and prints the first 20 records. Here is the output of the above script:

Output:

	survived	pclass	age	fare	Median_Age	Mean_Age
0	0	3	22.0	7.2500	22.0	22.0
1	1	1	38.0	71.2833	38.0	38.0
2	1	3	26.0	7.9250	26.0	26.0
3	1	1	35.0	53.1000	35.0	35.0
4	0	3	35.0	8.0500	35.0	35.0
5	0	3	NaN	8.4583	28.0	29.7
6	0	1	54.0	51.8625	54.0	54.0
7	0	3	2.0	21.0750	2.0	2.0
8	1	3	27.0	11.1333	27.0	27.0
9	1	2	14.0	30.0708	14.0	14.0
10	1	3	4.0	16.7000	4.0	4.0
11	1	1	58.0	26.5500	58.0	58.0
12	0	3	20.0	8.0500	20.0	20.0
13	0	3	39.0	31.2750	39.0	39.0
14	0	3	14.0	7.8542	14.0	14.0
15	1	2	55.0	16.0000	55.0	55.0
16	0	3	2.0	29.1250	2.0	2.0
17	1	2	NaN	13.0000	28.0	29.7
18	0	3	31.0	18.0000	31.0	31.0
19	1	3	NaN	7.2250	28.0	29.7

The highlighted rows in the above output show that NaN, i.e., null values in the **age** column, have been replaced by the median values in the **Median_Age** column and by mean values in the **Mean_Age** column.

The mean and median imputation can affect the data distribution for the columns containing the missing values. Specifically, the variance of the column is decreased by mean and median imputation now since more values are added to the center of the distribution. The following script plots the distribution of data for the **age** , **Median_Age** , and **Mean_Age** columns.

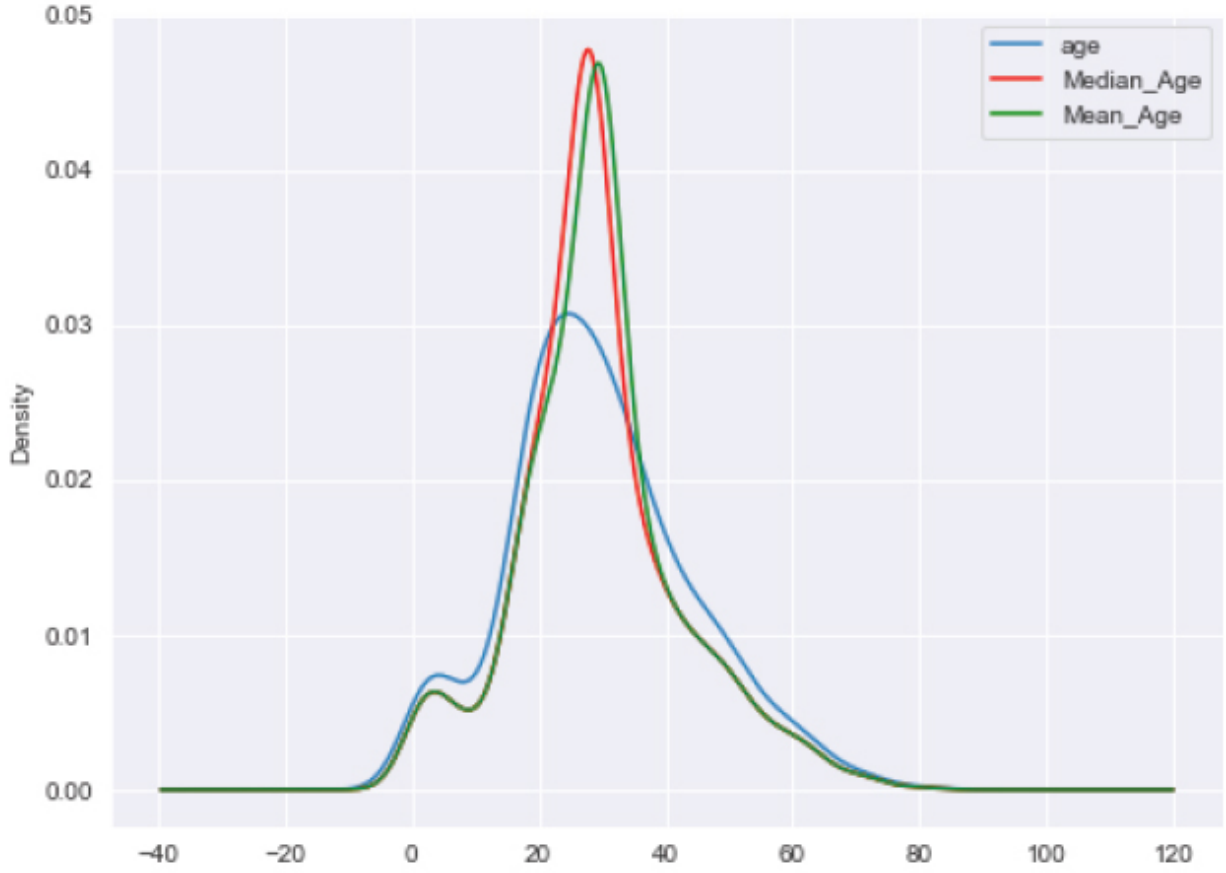
Script 36:

```
fig = plt.figure()
ax = fig.add_subplot( 111 )

titanic_data['age'] .plot(kind='kde', ax=ax)

titanic_data['Median_Age'] .plot(kind='kde', ax=ax, color='red')

titanic_data['Mean_Age'] .plot(kind='kde', ax=ax, color='green')

lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

Here is the output of the script above:

Output:

```
<matplotlib.legend.Legend at 0x1b1414a3c70>
```



You can see that the default values in the *age* columns have been distorted by the mean and median imputation, and the overall variance of the dataset has also been decreased.

Recommendations

Mean and Median imputation could be used for the missing numerical data in case the data is missing at random. If the data is normally distributed, mean imputation is better, or else, median imputation is preferred in case of skewed distributions.

2.4.2. Handling Missing Categorical Values

Frequent Category Imputation

One of the most common ways of handling missing values in a categorical column is to replace the missing values with the most frequently occurring values, i.e., the mode of the column. It is for this reason, frequent category imputation is also known as mode imputation. Let's see a real-world example of the frequent category imputation.

We will again use the Titanic dataset. We will first try to find the percentage of missing values in the *age* , *fare*, and *embarked_ town* columns.

Script 37:

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = [ 8 , 6 ]
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')

titanic_data = titanic_data[["embark_town", "age", "fare"]]
titanic_data.head()
titanic_data.isnull().mean()
```

Output:

```
embark_town    0.002245
age    0.198653
fare    0.000000
dtype:  float64
```

The output shows that *embark_town* and *age* columns have missing values. The ratio of missing values for the *embark_ town* column is very less.

Let's plot the bar plot that shows each category in the *embark_town* column against the number of passengers.

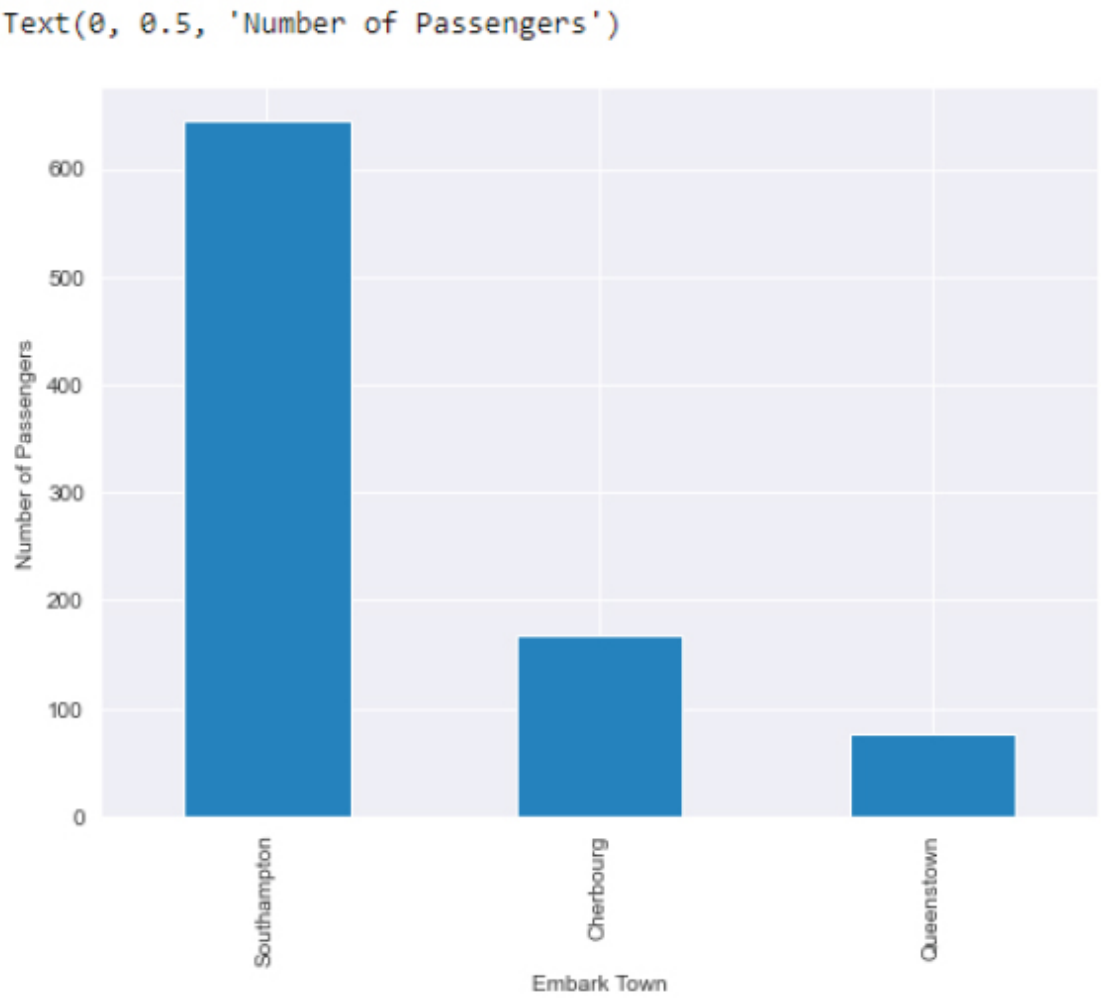
Script 38:

```
titanic_data.embark_town.value_counts().sort_values(ascending=False).plot.bar()
plt.xlabel('Embark Town')
```

```
plt.ylabel('Number of Passengers')
```

The output below clearly shows that most of the passengers embarked from Southampton.

Output:



Let's make sure if *Southampton* is the mode value for the *embark_town* column.

Script 39:

```
titanic_data.embark_town.mode()
```

Output:

```
0 Southampton
dtype:object
```

Next, we can simply replace the missing values in the *embark town* column by *Southampton* .

Script 40:

```
titanic_data.embark_town.fillna('Southampton', inplace=True)
```

Let's now find the mode of the *age* column and use it to replace the missing values in the *age* column.

Script 41:

```
titanic_data.age.mode()
```

Output:

```
0 24.0
dtype: float64
```

The output shows that the mode of the *age* column is 24. Therefore, we can use this value to replace the missing values in the *age* column.

Script 42:

```
import numpy as np

titanic_data['age_mode'] = titanic_data.age.fillna( 24 )

titanic_data.head( 20 )
```

Output:

	embark_town	age	fare	age_mode
0	Southampton	22.0	7.2500	22.0
1	Cherbourg	38.0	71.2833	38.0
2	Southampton	26.0	7.9250	26.0
3	Southampton	35.0	53.1000	35.0
4	Southampton	35.0	8.0500	35.0
5	Queenstown	NaN	8.4583	24.0
6	Southampton	54.0	51.8625	54.0
7	Southampton	2.0	21.0750	2.0
8	Southampton	27.0	11.1333	27.0
9	Cherbourg	14.0	30.0708	14.0
10	Southampton	4.0	16.7000	4.0
11	Southampton	58.0	26.5500	58.0
12	Southampton	20.0	8.0500	20.0
13	Southampton	39.0	31.2750	39.0
14	Southampton	14.0	7.8542	14.0
15	Southampton	55.0	16.0000	55.0
16	Queenstown	2.0	29.1250	2.0
17	Southampton	NaN	13.0000	24.0
18	Southampton	31.0	18.0000	31.0
19	Cherbourg	NaN	7.2250	24.0

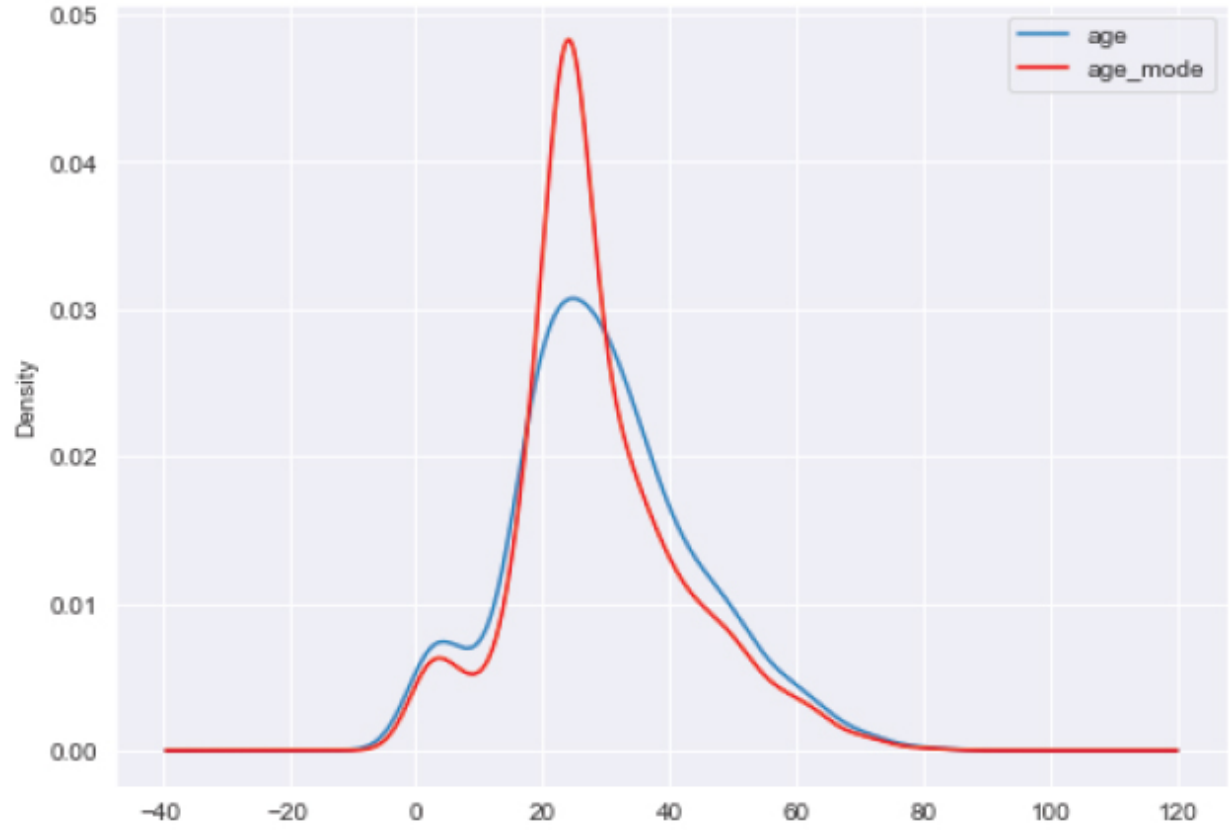
Finally, let's plot the kernel density estimation plot for the original *age* column and the *age* column that contains the mode of the values in place of the missing values.

Script 43:

```
plt.rcParams["figure.figsize"] = [ 8 , 6 ]

fig = plt.figure()
ax = fig.add_subplot( 111 )

titanic_data['age'] .plot(kind='kde', ax=ax)

titanic_data['age_mode'] .plot(kind='kde', ax=ax, color='red')

lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

Output:

```
<matplotlib.legend.Legend at 0x1b1416139a0>
```



Missing Category Imputation

Missing value imputation adds an arbitrary category, e.g., *missing* in place of the missing values. Take a look at an example of missing value imputation.

Let's load the Titanic dataset and see if any categorical column contains missing values.

Script 44:

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = [ 8 , 6 ]
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')
titanic_data = titanic_data[["embark_town", "age", "fare"]]
titanic_data.head()
titanic_data.isnull().mean()
```

Output:

```
embark_town 0.002245
age 0.198653
fare 0.000000
dtype: float64
```

The output shows that the *embark_town column* is a categorical column that contains some missing values too. We will apply the missing value imputation to this column.

Script 45:

```
titanic_data.embark_town.fillna('Missing', inplace=True)
```

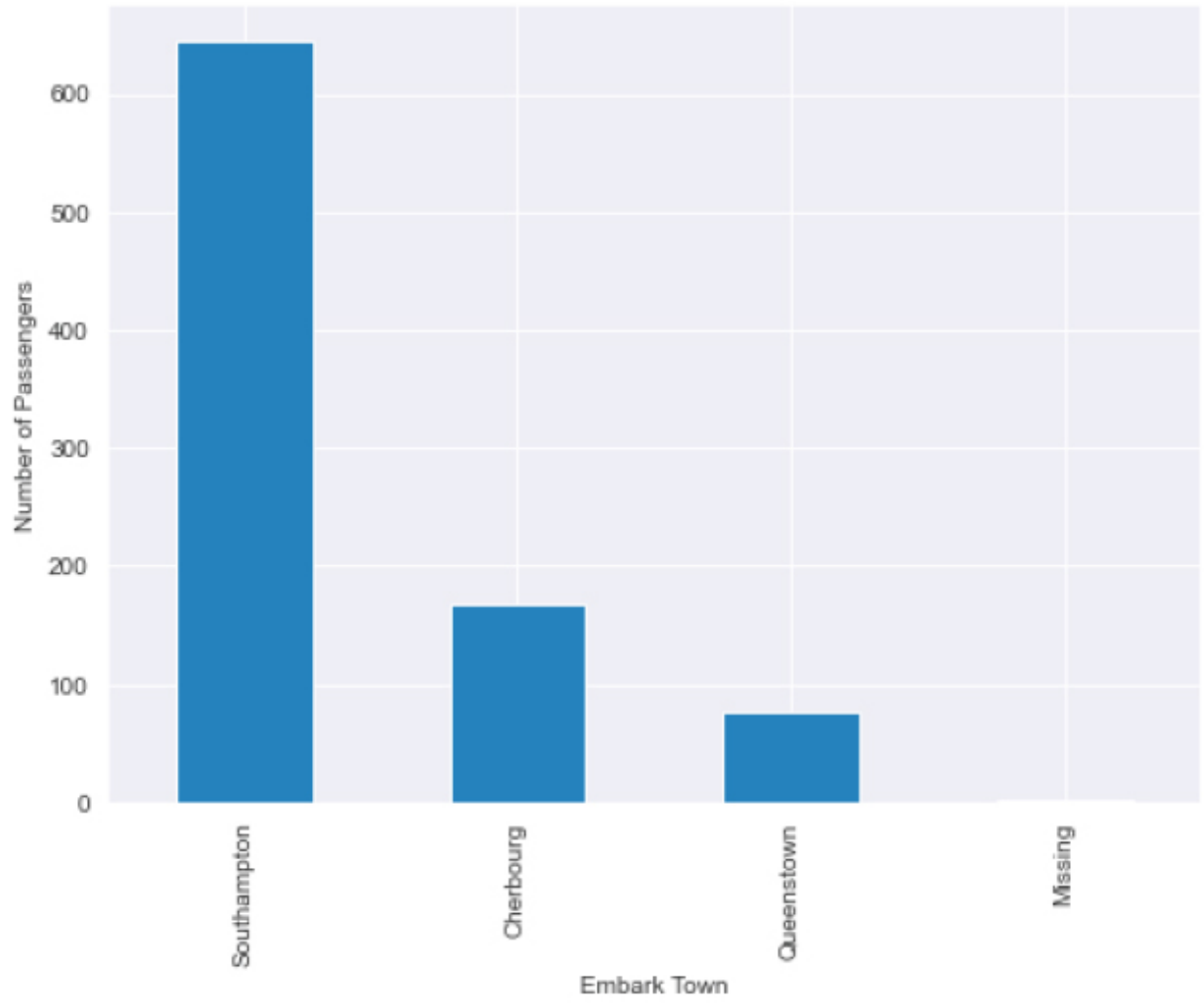
After applying missing value imputation, plot the bar plot for the *embark_town* column. You can see that we have a very small, almost negligible plot for the *missing* column.

Script 46:

```
titanic_data.embark_town.value_counts().sort_values(ascending=False).plot.bar()
plt.xlabel('Embark Town')
plt.ylabel('Number of Passengers')
```

Output:

```
Text(0, 0.5, 'Number of Passengers')
```



Further Readings – Basics of Pandas

1. Check the official documentation here (https://bit.ly/3mQnfOE) to learn more about the Pandas basics.

2. You can learn more about Matplotlib for data plotting at this link (https://matplotlib.org/).

Hands-on Time – Exercises

Now, it is your turn. Follow the instructions in **the exercises below** to check your understanding of Pandas basics that you learned in this chapter. The answers to these questions are given at the end of the book.

Exercise 2.1

Question 1:

What is the major disadvantage of mean and median imputation?

A. Distorts the data distribution
B. Distorts the data variance
C. Distorts the data covariance
D. All of the Above

Question 2:

How do you display the last three rows of a Pandas dataframe named "my_df"?

A. my_df.end(3)
B. my_df.bottom(3)
C. my_df.top(-3)
D. my_df.tail(3)

Question 3:

You can create a Pandas series using a:

A. NumPy Array
B. List

C. Dictionary

D. All of the Above

Exercise 2.2

Replace the missing values in the “deck” column of the Titanic dataset with the most frequently occurring categories in that column. Plot a bar plot for the updated “deck” column. The Titanic dataset can be downloaded using this Seaborn command:

```
import seaborn as sns

sns.load_dataset('titanic')
```

Manipulating Pandas Dataframes

Once you have loaded data into your Pandas dataframe, you might need to further manipulate the data and perform a variety of functions such as filtering certain columns, dropping the others, selecting a subset of rows or columns, sorting the data, finding unique values, and so on. You are going to study all these functions in this chapter.

You will first see how to select data via indexing and slicing, followed by a section on how to drop unwanted rows or columns from your data. You will then study how to filter your desired rows and columns. The chapter concludes with an explanation of sorting and finding unique values from a Pandas dataframe.

3.1. Selecting Data Using Indexing and Slicing

Indexing refers to fetching data using index or column information of a Pandas dataframe. Slicing, on the other hand, refers to slicing a Pandas dataframe using indexing techniques.

In this section, you will see the different techniques of indexing and slicing Pandas dataframes.

You will be using the Titanic dataset for this section, which you can import via the Seaborn library's load_dataset() method, as shown in the script below.

Script 1:

```
import matplotlib.pyplot as plt
import seaborn as sns

# sets the default style for plotting
```

```
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')
titanic_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

3.1.1. Selecting Data Using Brackets []

One of the simplest ways to select data from various columns is by using square brackets. To get column data in the form of a series from a Pandas dataframe, you need to pass the column name inside square brackets that follow the Pandas dataframe name.

The following script selects records from the class column of the Titanic dataset.

Script 2:

```
print (titanic_data["class"])
type (titanic_data["class"])
```

Output:

```
0 Third
1 First
2 Third
3 First
4 Third
      …
886 Second
887 First
888 Third
889 First
```

```
890 Third
Name: class, Length: 891, dtype: category
Categories (3, object): ['First', 'Second', 'Third']
Out[2]:
pandas.core.series.Series
```

You can select multiple columns by passing a list of column names inside a string to the square brackets. You will then get a Pandas dataframe with the specified columns, as shown below.

Script 3:

```
print (type (titanic_data[["class", "sex", "age"]]))
titanic_data[["class", "sex", "age"]]
```

Output:

	class	sex	age
0	Third	male	22.0
1	First	female	38.0
2	Third	female	26.0
3	First	female	35.0
4	Third	male	35.0
...	...	...	...
886	Second	male	27.0
887	First	female	19.0
888	Third	female	NaN
889	First	male	26.0
890	Third	male	32.0

You can also filter rows based on some column values. For doing this, you need to pass the condition to the filter inside the square brackets. For instance, the script below returns all records from the Titanic dataset where the sex column contains the value “male.”

Script 4:

```
my_df = titanic_data[titanic_data["sex"] == "male"]
my_df.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True
5	0	3	male	NaN	0	0	8.4583	Q	Third	man	True	NaN	Queenstown	no	True
6	0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no	True
7	0	3	male	2.0	3	1	21.0750	S	Third	child	False	NaN	Southampton	no	False

You can specify multiple conditions inside the square brackets. The following script returns those records where the sex column contains the string "male," while the class column contains the string "First."

Script 5:

```
my_df = titanic_data[(titanic_data["sex"] == "male") & (titanic_data["class"] == "First")]
my_df.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
6	0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no	True
23	1	1	male	28.0	0	0	35.5000	S	First	man	True	A	Southampton	yes	True
27	0	1	male	19.0	3	2	263.0000	S	First	man	True	C	Southampton	no	False
30	0	1	male	40.0	0	0	27.7208	C	First	man	True	NaN	Cherbourg	no	True
34	0	1	male	28.0	1	0	82.1708	C	First	man	True	NaN	Cherbourg	no	False

You can also use the isin() function to specify a range of values to filter records. For instance, the script below filters all records where the age column contains the values 20, 21, or 22.

Script 6:

```
ages = [ 20 , 21 , 22 ]
age_dataset = titanic_data[titanic_data["age"].isin(ages)]
age_dataset.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.25	S	Third	man	True	NaN	Southampton	no	False
12	0	3	male	20.0	0	0	8.05	S	Third	man	True	NaN	Southampton	no	True
37	0	3	male	21.0	0	0	8.05	S	Third	man	True	NaN	Southampton	no	True
51	0	3	male	21.0	0	0	7.80	S	Third	man	True	NaN	Southampton	no	True
56	1	2	female	21.0	0	0	10.50	S	Second	woman	False	NaN	Southampton	yes	True

3.1.2. Indexing and Slicing Using loc Function

The loc function from the Pandas dataframe can also be used to filter records in the Pandas dataframe.

To create a dummy dataframe used as an example in this section, run the following script:

Script 7:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]

my_df = pd.DataFrame(scores)
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

Let's now see how to filter records. To filter the row at the second index in the my_dfdataframe, you need to pass 2 inside the square brackets that follow the loc function. Here is an example:

Script 8:

```
print (my_df.loc[ 2 ])
type(my_df.loc[ 2 ])
```

In the output below, you can see data from the row at the second index (row 3) in the form of a series.

Output:

```
Subject English
Score 76
Grade C
Remarks Fair
Name: 2, dtype: object
Out[7]:
pandas.core.series.Series
```

You can also specify the range of indexes to filter records using the loc function. For instance, the following script filters records from index 2 to 4.

Script 9:

```
my_df.loc[ 2 : 4 ]
```

Output:

	Subject	Score	Grade	Remarks
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

Along with filtering rows, you can also specify which columns to filter with the loc function.

The following script filters the values in columns Grade and Score in the rows from index 2 to 4.

Script 10:

```
my_df.loc[ 2 : 4 , ["Grade", "Score"]]
```

Output:

	Grade	Score
2	C	76
3	C	72
4	A	95

In addition to passing default integer indexes, you can also pass named or labeled indexes to the loc function.

Let’s create a dataframe with named indexes. Run the following script to do so:

Script 11:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]

my_df = pd.DataFrame(scores, index = ["Student1", "Student2", "Student3", "Student4", "Student5"])
my_df
```

From the output below, you can see that the my_dfdataframe now contains named indexes, e.g., Student1, Student2, etc.

Output:

	Subject	Score	Grade	Remarks
Student1	Mathematics	85	B	Good
Student2	History	98	A	Excellent
Student3	English	76	C	Fair
Student4	Science	72	C	Fair
Student5	Arts	95	A	Excellent

Let's now filter a record using Student1 as the index value in the loc function.

Script 12:

```
my_df.loc["Student1"]
```

Output:

```
Subject Mathematics
Score      85
Grade       B
```

```
Remarks     Good
Name: Student1, dtype: object
```

As shown below, you can specify multiple named indexes in a list to the loc method. The script below filters records with indexes *Student1* and *Student2* .

Script 13:

```
index_list = ["Student1", "Student2"]
my_df.loc[index_list]
```

Output:

	Subject	Score	Grade	Remarks
Student1	Mathematics	85	B	Good
Student2	History	98	A	Excellent

You can also find the value in a particular column while filtering records using a named index.

The script below returns the value in the Grade column for the record with the named index *Student1* .

Script 14:

```
my_df.loc["Student1", "Grade"]
```

Output:

```
'B'
```

As you did with the default integer index, you can specify a range of records using the named indexes within the loc function.

The following function returns values in the Grade column for the indexes from Student1 to Student2.

Script 15:

```
my_df.loc["Student1":"Student2", "Grade"]
```

Output:

```
Student1 B
Student2 A
Name: Grade, dtype: object
```

Let’s see another example.

The following function returns values in the Grade column for the indexes from Student1 to Student4.

Script 16:

```
my_df.loc["Student1":"Student4", "Grade"]
```

Output:

```
Student1 B
Student2 A
Student3 C
Student4 C
Name: Grade, dtype: object
```

You can also specify a list of Boolean values that correspond to the indexes to select using the loc method.

For instance, the following script returns only the fourth record since all the values in the list passed to the loc function are false, except the one at the fourth index.

Script 17:

```
my_df.loc[[False, False, False, True, False ]]
```

Output:

	Subject	Score	Grade	Remarks
Student4	Science	72	C	Fair

You can also pass dataframe conditions inside the loc method. A condition returns a boolean value which can be used to index the loc function, as you have already seen in the previous scripts.

Before you see how loc function uses conditions, let's see the outcome of a basic condition in a Pandas dataframe. The script below returns index names along with True or False values depending on whether the Score column contains a value greater than 80 or not.

Script 18:

```
my_df[«Score»]> 80
```

You can see Boolean values in the output. You can see that indexes Student1, Student2, and Student5 contain True.

Output:

```
Student1 True
Student2 True
Student3 False
Student4 False
Student5 True
Name: Score, dtype: bool
```

Now, let's pass the condition "my_df["Score"]> **80** " to the loc function.

Script 19:

```
my_df.loc[my_df["Score"]> 80 ]
```

In the output, you can see records with the indexes Student1, Student2, and Student5.

Output:

	Subject	Score	Grade	Remarks
Student1	Mathematics	85	B	Good
Student2	History	98	A	Excellent
Student5	Arts	95	A	Excellent

You can pass multiple conditions to the loc function. For instance, the script below returns those rows where the Score column contains a value greater than 80, and the Remarks column contains the string *Excellent*.

Script 20:

```
my_df.loc[(my_df["Score"]> 80 ) & (my_df["Remarks"] == "Excellent")]
```

Output:

	Subject	Score	Grade	Remarks
Student2	History	98	A	Excellent
Student5	Arts	95	A	Excellent

Finally, you can also specify column names to fetch values from, along with a condition.

For example, the script below returns values from the Score and Grade columns, where the Score column contains a value greater than 80.

Script 21:

```
my_df.loc[my_df["Score"]> 80 , ["Score","Grade"]]
```

Output:

	Score	Grade
Student1	85	B
Student2	98	A
Student5	95	A

Finally, you can set values for all the columns in a row using the loc function. For instance, the following script sets values for all the columns for the record at index *Student4* as 90.

Script 22:

```
my_df.loc["Student4"] = 90
my_df
```

Output:

	Subject	Score	Grade	Remarks
Student1	Mathematics	85	B	Good
Student2	History	98	A	Excellent
Student3	English	76	C	Fair
Student4	90	90	90	90
Student5	Arts	95	A	Excellent

3.1.3. Indexing and Slicing Using iloc Function

You can also use the iloc function for selecting and slicing records using index values. However, unlike the loc function, where you can pass both the string indexes and integer indexes, you can only pass the integer index values to the iloc function.

The following script creates a dummy dataframe for this section.

Script 23:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]
my_df = pd.DataFrame(scores)
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

Let’s filter the record at index 3 (row 4).

Script 24:

```
my_df.iloc[ 3 ]
```

The script below returns a series.

Output:

```
Subject Science
Score     72
Grade     C
Remarks   Fair
Name: 3, dtype: object
```

If you want to select records from a single column as a dataframe, you need to specify the index inside the square brackets and then those square brackets inside the square brackets that follow the iloc function, as shown below.

Script 25:

```
my_df.iloc[[ 3 ]]
```

Output:

	Subject	Score	Grade	Remarks
3	Science	72	C	Fair

You can pass multiple indexes to the iloc function to select multiple records. Here is an example:

Script 26:

```
my_df.iloc[[ 2 , 3 ]]
```

Output:

	Subject	Score	Grade	Remarks
2	English	76	C	Fair
3	Science	72	C	Fair

You can also pass a range of indexes. In this case, the records from the lower range to 1 less than the upper range will be selected.

For instance, the script below returns records from index 2 to index 3 (1 less than 4).

Script 27:

```
my_df.iloc[ 2 : 4 ]
```

Output:

	Subject	Score	Grade	Remarks
2	English	76	C	Fair
3	Science	72	C	Fair

In addition to specifying indexes, you can also pass column numbers (starting from 0) to the iloc method.

The following script returns values from columns number 0 and 1 for the records at indexes 2 and 3.

Script 28:

```
my_df.iloc[[ 2 , 3 ], [ 0 , 1 ]]
```

Output:

	Subject	Score
2	English	76
3	Science	72

You can also pass a range of indexes and columns to select. The script below selects columns 1 and 2 and rows 2 and 3.

Script 29:

```
my_df.iloc[ 2 : 4 , 0 : 2 ]
```

Output:

	Subject	Score
2	English	76
3	Science	72

3.2. Dropping Rows and Columns with the drop() Method

Apart from selecting columns using the loc and iloc functions, you can also use the drop() method to drop unwanted rows and columns from your dataframe while keeping the rest of the rows and columns.

3.2.1. Dropping Rows

The following script creates a dummy dataframe that you will use in this section.

Script 30:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
```

```
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]

my_df = pd.DataFrame(scores)
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

The following script drops records at indexes 1 and 4.

Script 31:

```
my_df2 = my_df.drop([ 1 , 4 ])
my_df2.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
2	English	76	C	Fair
3	Science	72	C	Fair

From the output above, you can see that the indexes are not in sequence since you have dropped indexes 1 and 4.

You can reset dataframe indexes starting from 0, using the reset_index().

Let's call the reset_index() method on the my_df2 dataframe. Here, the value True for the inplace parameter specifies that you want to remove the records in place without assigning the result to any new variable.

Script 32:

```
my_df2.reset_index(inplace=True )
my_df2.head()
```

Output:

	index	Subject	Score	Grade	Remarks
0	0	Mathematics	85	B	Good
1	2	English	76	C	Fair
2	3	Science	72	C	Fair

The above output shows that the indexes have been reset. Also, you can see that a new column *index* has been added, which contains the original index. If you only want to reset new indexes without creating a new column named *index* , you can do so by passing True as the value for the drop parameter of the reset_index method.

Let's again drop some rows and reset the index using the reset_index() method by passing True as the value for the drop attribute. See the following two scripts:

Script 33:

```
my_df2 = my_df.drop([ 1 , 4 ])
my_df2.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
2	English	76	C	Fair
3	Science	72	C	Fair

Script 34:

```
my_df2.reset_index(inplace=True , drop = True )
my_df2.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	English	76	C	Fair
2	Science	72	C	Fair

By default, the drop method doesn't drop rows in place. Instead, you have to assign the result of the drop() method to another variable that contains the records with dropped results.

For instance, if you drop the records at indexes 1, 3, and 4 using the following script and then print the dataframe header, you will see that the rows are not removed from the original dataframe.

Script 35:

```
my_df.drop([ 1 , 3 , 4 ])
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

If you want to drop rows in place, you need to pass True as the value for the inplace attribute, as shown in the script below:

Script 36:

```
my_df.drop([ 1 , 3 , 4 ], inplace = True )
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
2	English	76	C	Fair

3.2.1. Dropping Columns

You can also drop columns using the drop() method.

The following script creates a dummy dataframe for this section.

Script 37:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
```

```
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]

my_df = pd.DataFrame(scores)
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

To drop columns via the drop() method, you need to pass the list of columns to the drop() method, along with 1 as the value for the axis parameter of the drop method.

The following script drops the columns Subject and Grade from our dummy dataframe.

Script 38:

```
my_df2 = my_df.drop(["Subject", "Grade"], axis = 1 )
my_df2.head()
```

Output:

	Score	Remarks
0	85	Good
1	98	Excellent
2	76	Fair
3	72	Fair
4	95	Excellent

You can also drop the columns inplace from a dataframe using the inplace = True parameter value, as shown in the script below.

Script 39:

```
my_df.drop(["Subject", "Grade"], axis = 1 , inplace = True)
my_df.head()
```

Output:

	Score	Remarks
0	85	Good
1	98	Excellent
2	76	Fair
3	72	Fair
4	95	Excellent

3.3. Filtering Rows and Columns with Filter Method

The drop() method drops the unwanted records, and the filter() method performs the reverse tasks. It keeps the desired records from a set of records in a Pandas dataframe.

3.3.1. Filtering Rows

Run the following script to create a dummy dataframe for this section.

Script 40:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]

my_df = pd.DataFrame(scores)
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

To filter rows using the filter() method, you need to pass the list of row indexes to filter to the filter() method of the Pandas dataframe. Along with that, you need to pass 0 as the value for the axis attribute of the filter() method. Here is an example. The script below filters rows with indexes 1, 3, and 4 from the Pandas dataframe.

Script 41:

```
my_df2 = my_df.filter([ 1 , 3 , 4 ], axis = 0 )
my_df2.head()
```

Output:

	Subject	Score	Grade	Remarks
1	History	98	A	Excellent
3	Science	72	C	Fair
4	Arts	95	A	Excellent

You can also reset indexes after filtering data using the reset_ index() method, as shown in the following script:

Script 42:

```
my_df2 = my_df2.reset_index(drop=True)
my_df2.head()
```

Output:

	Subject	Score	Grade	Remarks
0	History	98	A	Excellent
1	Science	72	C	Fair
2	Arts	95	A	Excellent

3.3.2. Filtering Columns

The dummy dataframe for this section is created using the following script:

Script 43:

```
import pandas as pd

scores = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Science', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Arts', 'Score': 95 , 'Grade': 'A','Remarks': 'Excellent'},
    ]

my_df = pd.DataFrame(scores)
my_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Science	72	C	Fair
4	Arts	95	A	Excellent

To filter columns using the filter() method, you need to pass the list of column names to the filter method. Furthermore, you need to set 1 as the value for the axis attribute.

The script below filters the Score and Grade columns from your dummy dataframe.

Script 44:

```
my_df2 = my_df.filter(["Score","Grade"], axis = 1 )
my_df2.head()
```

Output:

	Score	Grade
0	85	B
1	98	A
2	76	C
3	72	C
4	95	A

3.4. Sorting Dataframes

You can also sort records in your Pandas dataframe based on values in a particular column. Let's see how to do this.

For this section, you will be using the Titanic dataset, which you can import using the Seaborn library using the following script:

Script 45:

```
import matplotlib.pyplot as plt
import seaborn as sns

# sets the default style for plotting
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')
titanic_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

To sort the Pandas dataframe, you can use the **sort_values()** function of the Pandas dataframe. The list of columns used for sorting needs to be passed to the **by** attribute of the **sort_ values()** method.

The following script sorts the Titanic dataset in ascending order of the passenger's age.

Script 46:

```
age_sorted_data = titanic_data.sort_values(by=['age'])
age_sorted_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
803	1	3	male	0.42	0	1	8.5167	C	Third	child	False	NaN	Cherbourg	yes	False
755	1	2	male	0.67	1	1	14.5000	S	Second	child	False	NaN	Southampton	yes	False
644	1	3	female	0.75	2	1	19.2583	C	Third	child	False	NaN	Cherbourg	yes	False
469	1	3	female	0.75	2	1	19.2583	C	Third	child	False	NaN	Cherbourg	yes	False
78	1	2	male	0.83	0	2	29.0000	S	Second	child	False	NaN	Southampton	yes	False

To sort by descending order, you need to pass False as the value for the **ascending** attribute of the **sort_values()** function.

The following script sorts the dataset by descending order of age.

Script 47:

```
age_sorted_data = titanic_data.sort_values(by=['age'], ascending = False)
age_sorted_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
630	1	1	male	80.0	0	0	30.0000	S	First	man	True	A	Southampton	yes	True
851	0	3	male	74.0	0	0	7.7750	S	Third	man	True	NaN	Southampton	no	True
493	0	1	male	71.0	0	0	49.5042	C	First	man	True	NaN	Cherbourg	no	True
96	0	1	male	71.0	0	0	34.6542	C	First	man	True	A	Cherbourg	no	True
116	0	3	male	70.5	0	0	7.7500	Q	Third	man	True	NaN	Queenstown	no	True

You can also pass multiple columns to the **by** attribute of the **sort_values()** function. In such a case, the dataset will be sorted by the first column, and in the case of equal values for two or more records, the dataset will be sorted by the second column and so on.

The following script first sorts the data by Age and then by Fare, both by descending orders.

Script 48:

```
age_sorted_data = titanic_data.sort_values(by=['age','fare'], ascending = False)
age_sorted_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
630	1	1	male	80.0	0	0	30.0000	S	First	man	True	A	Southampton	yes	True
851	0	3	male	74.0	0	0	7.7750	S	Third	man	True	NaN	Southampton	no	True
493	0	1	male	71.0	0	0	49.5042	C	First	man	True	NaN	Cherbourg	no	True
96	0	1	male	71.0	0	0	34.6542	C	First	man	True	A	Cherbourg	no	True
116	0	3	male	70.5	0	0	7.7500	Q	Third	man	True	NaN	Queenstown	no	True

3.5. Pandas Unique and Count Functions

In this section, you will see how you can get a list of unique values, the number of all unique values, and records per unique value from a column in a Pandas dataframe.

You will be using the Titanic dataset once again, which you download via the following script.

Script 49:

```
import matplotlib.pyplot as plt
import seaborn as sns

# sets the default style for plotting
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')
titanic_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

To find the number of all the unique values in a column, you can use the unique() function. The script below returns all the unique values from the

class column from the Titanic dataset.

Script 50:

```
titanic_data["class"].unique()
```

Output:

```
['Third', 'First', 'Second']
Categories (3, object): ['Third', 'First', 'Second']
```

To get the count of unique values, you can use the nunique() method, as shown in the script below.

Script 51:

```
titanic_data["class"].nunique()
```

Output:

```
3
```

To get the count of non-null values for all the columns in your dataset, you may call the count() method on the Pandas dataframe. The following script prints the count of the total number of non-null values in all the columns of the Titanic dataset.

Script 52:

```
titanic_data.count()
```

Output:

```
survived    891
```

```
pclass    891
sex    891
age    714
sibsp    891
parch    891
fare    891
embarked    889
class    891
who    891
adult_male    891
deck    203
embark_town    889
alive    891
alone    891
dtype: int64
```

Finally, if you want to find the number of records for all the unique values in a dataframe column, you may use the value_counts() function.

The script below returns counts of records for all the unique values in the class column.

Script 53:

```
titanic_data["class"].value_counts()
```

Output:

```
Third    491
First    216
Second    184
Name: class, dtype: int64
```

Further Readings – Pandas Dataframe Manipulation

Check the official documentation here (https://bit.ly/3kguKgb) to learn more about the Pandas dataframe manipulation functions.

Hands-on Time – Exercises

Now, it is your turn. Follow the instructions in **the exercises below** to check your understanding of Pandas dataframe manipulation techniques that you learned in this chapter. The answers to these questions are given at the end of the book.

Exercise 3.1

Question 1:

Which function is used to sort Pandas dataframe by a column value?

A. sort_dataframe()
B. sort_rows()
C. sort_values()
D. sort_records()

Question 2:

To filter columns from a Pandas dataframe, you have to pass a list of column names to one of the following methods:

A. filter()
B. filter_columns()
C. apply_filter ()
D. None of the above

Question 3:

To drop the second and fourth rows from a Pandas dataframe named my_df, you can use the following script:

A. my_df.drop([2,4])
B. my_df.drop([1,3])
C. my_df.delete([2,4])
D. my_df.delete([1,3])

Exercise 3.2

From the Titanic dataset, filter all the records where the fare is greater than 20 and the passenger traveled alone. You can access the Titanic dataset using the following Seaborn command:

```
import seaborn as sns

titanic_data = sns.load_dataset('titanic')
```

Data Grouping, Aggregation, and Merging with Pandas

Often, you need to group data based on a particular column value. Take the example of the Titanic dataset that you have seen in previous chapters. What if you want to find the maximum fare paid by male and female passengers? Or, you want to find which embarked town had the oldest passengers? In such cases, you will need to create groups of records based on the column values. You can then find the maximum, minimum, mean, or other information within that group.

Furthermore, you might want to merge or concatenate dataframes if your dataset is distributed among multiple files. Finally, you might need to change the orientation of your dataframe and discretize certain columns.

In this chapter, you will see the answers to all these questions. You will study how to group, concatenate, and merge your dataframe. You will also see how to change your dataframe orientation. The chapter concludes with a brief introduction to binning and discretization with Pandas dataframes.

4.1. Grouping Data with GroupBy

You will be using the Titanic dataset for various GroupBy functions in this section. Import the Titanic dataset using the following script:

Script 1:

```
import matplotlib.pyplot as plt
import seaborn as sns

import pandas as pd
```

```
# sets the default style for plotting
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')
titanic_data.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

To group data by a column value, you can use the groupby() function of the Pandas dataframe. You need to pass the column as the parameter value to the groupby() function.

The script below groups the data in the Titanic dataset by the class column. Next, the type of object returned by the groupby() function is also printed.

Script 2:

```
titanic_gbclass = titanic_data.groupby("class")
type(titanic_gbclass)
```

Output:

```
pandas.core.groupby.generic.DataFrameGroupBy
```

The above output shows that the groupby() function returns the DataFrameGroupBy object. You can use various attributes and functions of this object to get various information about different groups.

For instance, to see the number of groups, you can use the ngroups attribute, which returns the number of groups (unique values). Since the number of

groups in the class columns is 3, you will see 3 printed in the output of the script.

Script 3:

```
titanic_gbclass.ngroups
```

Output:

```
3
```

You can use the size() function to get the number of records in each group. This is similar to the value_counts() function that you saw in the previous chapter.

Script 4:

```
titanic_gbclass.size()
```

Output:

```
class
First   216
Second  184
Third   491
dtype: int64
```

Finally, you can also get the row indexes for records in a particular group. The script below returns the row indexes for rows where the class column contains “First.”

Script 5:

```
titanic_gbclass.groups["First"]
```

Output:

```
Int64Index([1, 3, 6, 11, 23, 27, 30, 31, 34, 35,
    …
    853, 856, 857, 862, 867, 871, 872, 879, 887, 889],
dtype='int64', length=216)
```

You can also get the first or last record from each group using the first() and last() functions, respectively.

As an example, the following script returns the last record from each group in the class column.

Script 6:

```
titanic_gbclass.last()
```

Output:

class	survived	pclass	sex	age	sibsp	parch	fare	embarked	who	adult_male	deck	embark_town	alive	alone
First	1	1	male	26.0	0	0	30.00	C	man	True	C	Cherbourg	yes	True
Second	0	2	male	27.0	0	0	13.00	S	man	True	E	Southampton	no	True
Third	0	3	male	32.0	0	0	7.75	Q	man	True	E	Queenstown	no	True

You can also get a dataframe that contains records belonging to a subgroup using the get_group() function. The following script returns all records from the group named “Second” from the class column.

Script 7:

```
titanic_second_class = titanic_gbclass.get_group("Second")
titanic_second_class.head()
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	who	adult_male	deck	embark_town	alive	alone
9	1	2	female	14.0	1	0	30.0708	C	child	False	NaN	Cherbourg	yes	False
15	1	2	female	55.0	0	0	16.0000	S	woman	False	NaN	Southampton	yes	True
17	1	2	male	NaN	0	0	13.0000	S	man	True	NaN	Southampton	yes	True
20	0	2	male	35.0	0	0	26.0000	S	man	True	NaN	Southampton	no	True
21	1	2	male	34.0	0	0	13.0000	S	man	True	D	Southampton	yes	True

The DataFrameByGroup object can also be used to perform aggregate functions. For instance, you can get the maximum age in all the groups in the class column using the max() function, as shown in the following script.

Script 8:

```
titanic_gbclass.age.max()
```

Output:

```
class
First    80.0
Second   70.0
Third    74.0
Name: age, dtype: float64
```

Similarly, you can also get information based on various aggregate functions bypassing the list of functions to the agg() method.

For instance, the script below returns the maximum, minimum, median, and mean, and count of age values in different groups in the class column.

Script 9:

```
titanic_gbclass.fare.agg(['max', 'min', 'count', 'median', 'mean'])
```

Output:

class	max	min	count	median	mean
First	512.3292	0.0	216	60.2875	84.154687
Second	73.5000	0.0	184	14.2500	20.662183
Third	69.5500	0.0	491	8.0500	13.675550

4.2. Concatenating and Merging Data

4.2.1. Concatenating Data

As you did previously, you will be using the Titanic dataset for this section as well. Run the following script to import the Titanic dataset.

Script 10:

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic_data = sns.load_dataset('titanic')
```

Concatenating Rows

Oftentimes, you need to concatenate or join multiple Pandas dataframes horizontally or vertically. So, let's first see how to concatenate or join Pandas dataframes vertically or in row order.

Using Titanic data, we will create two Pandas dataframes. The first dataframe consists of rows where the passenger class is *First* , while the second dataframe consists of rows where the passenger class is *Second* .

Script 11:

```
titanic_pclass1_data = titanic_data[titanic_data["class"] == "First"]
print (titanic_pclass1_data.shape)

titanic_pclass2_data = titanic_data[titanic_data["class"] == "Second"]
print (titanic_pclass2_data.shape)
```

Output:

```
(216, 15)
(184, 15)
```

The output shows that both the newly created dataframes have 15 columns. It is important to mention that while concatenating data vertically, both the dataframes should have an equal number of columns.

There are two ways to concatenate datasets horizontally. You can call the **append()** method via the first dataframe and pass the second dataframe as a parameter to the **append()** method. Look at the following script:

Script 12:

```
final_data = titanic_pclass1_data.append(titanic_pclass2_data,
ignore_index=True)
print (final_data.shape)
```

The output now shows that the total number of rows is 400, which is the sum of the number of rows in the two dataframes that we concatenated.

Output:

```
(400, 15)
```

The other way to concatenate two dataframes is by passing both the dataframes as parameters to the **concat()** method of the Pandas module. The following script shows how to do that.

Script 13:

```
final_data = pd.concat([titanic_pclass1_data, titanic_pclass2_data])
print (final_data.shape)
```

Output:

```
(400, 15)
```

Concatenating Columns

To concatenate dataframes horizontally, make sure that the dataframes have an equal number of rows. You can use the **concat()** method to concatenate dataframes horizontally as well. However, you will need to pass 1 as the value for the **axis** attribute. Furthermore, to reset dataset indexes, you need to pass True as the value for the **ignore_index** attribute.

Script 14:

```
df1 = final_data[: 200 ]
print (df1.shape)
df2 = final_data[ 200 :]
print (df2.shape)

final_data2 = pd.concat([df1, df2], axis = 1 , ignore_index = True )
print (final_data2.shape)
```

Output:

```
(200, 15)
(200, 15)
(400, 30)
```

4.2.2. Merging Data

You can merge multiple dataframes based on common values between any columns of the two dataframes.

Let’s create two dataframes: scores1 and scores2.

Script 15:

```
import pandas as pd

scores1 = [
    {'Subject':'Mathematics', 'Score': 85 , 'Grade': 'B', 'Remarks': 'Good', },
```

```
    {'Subject':'History', 'Score': 98 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'English', 'Score': 76 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Chemistry', 'Score': 72 , 'Grade': 'C','Remarks': 'Fair'},
    ]

scores2 = [
    {'Subject':'Arts', 'Score': 70 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'Physics', 'Score': 75 , 'Grade': 'C','Remarks': 'Fair'},
    {'Subject':'English', 'Score': 92 , 'Grade': 'A','Remarks': 'Excellent'},
    {'Subject':'Chemistry', 'Score': 91 , 'Grade': 'A','Remarks': 'Excellent'},
]

scores1_df = pd.DataFrame(scores1)
scores2_df = pd.DataFrame(scores2)
```

The following script prints the header of the scores1 dataframe.

Script 16:

```
scores1_df.head()
```

Output:

	Subject	Score	Grade	Remarks
0	Mathematics	85	B	Good
1	History	98	A	Excellent
2	English	76	C	Fair
3	Chemistry	72	C	Fair

Script 17:

```
scores2_df.head()
```

The following script prints the header of the scores2 dataframe

Output:

	Subject	Score	Grade	Remarks
0	Arts	70	C	Fair
1	Physics	75	C	Fair
2	English	92	A	Excellent
3	Chemistry	91	A	Excellent

You can see that the two dataframes, scores1 and scores2, have the same columns. You can merge or join two dataframes on any column. In this section, you will be merging two dataframes on the Subject column.

The merge() function in Pandas can be used to merge two dataframes based on common values between columns of the two dataframes. The merge() is similar to SQL JOIN operations.

Merging with Inner Join

The script below merges scores1 and scores2 dataframe using the INNER JOIN strategy.

To merge two dataframes, you call the merge() function on the dataframe that you want on the left side of the join operation. The dataframe to be merged on the right is passed as the first parameter. Next, the column name on which you want to apply the merge operation is passed to the “on” attribute. Finally, the merge strategy is passed to the “how” attribute, as shown in the script below.

With INNER JOIN, only those records from both the dataframes are returned, where there are common values in the column used for merging the dataframes.

For instance, in the scores1 and scores2 dataframes, English and Chemistry are two subjects that exist in the Subject column of both the dataframes.

Therefore, in the output of the script below, you can see the merged dataframes containing only these two records. Since the column names, Score, Grade, and Remarks, are similar in both the dataframes, the column names from the dataset on the left (which is scores1 in the following script) have

been appended with "_x". While the column names in the dataframe on the right are appended with "_y".

In case both the dataframes in the merge operation contain different column names, you will see the original column names in the merged dataframe.

Script 18:

```
join_inner_df = scores1_df.merge(scores2_df, on='Subject', how='inner')
join_inner_df.head()
```

Output:

	Subject	Score_x	Grade_x	Remarks_x	Score_y	Grade_y	Remarks_y
0	English	76	C	Fair	92	A	Excellent
1	Chemistry	72	C	Fair	91	A	Excellent

Merging with Left Join

When you merge two Pandas dataframes using a LEFT join, all the records from the dataframe to the left side of the merge() function are returned, whereas, for the right dataframe, only those records are returned where a common value is found on the column being merged.

Script 19:

```
join_inner_df = scores1_df.merge(scores2_df, on='Subject', how='left')
join_inner_df.head()
```

In the output below, you can see the records for the Mathematics and History subjects in the dataframe on the left. But since these records do not exist in the dataframe on the right, you see null values for columns Score_y, Grade_y, and Remarks_y. Only those records can be seen for the right dataframe where there are common values in the Subject column from both the dataframes.

Output:

	Subject	Score_x	Grade_x	Remarks_x	Score_y	Grade_y	Remarks_y
0	Mathematics	85	B	Good	NaN	NaN	NaN
1	History	98	A	Excellent	NaN	NaN	NaN
2	English	76	C	Fair	92.0	A	Excellent
3	Chemistry	72	C	Fair	91.0	A	Excellent

Merging with Right Join

In merging with right join, the output contains all the records from the right dataframe while only those records are returned from the left dataframe where there are common values in the merge column.

Here is an example:

Script 20:

```
join_inner_df = scores1_df.merge(scores2_df, on='Subject', how='right')
join_inner_df.head()
```

Output:

	Subject	Score_x	Grade_x	Remarks_x	Score_y	Grade_y	Remarks_y
0	Arts	NaN	NaN	NaN	70	C	Fair
1	Physics	NaN	NaN	NaN	75	C	Fair
2	English	76.0	C	Fair	92	A	Excellent
3	Chemistry	72.0	C	Fair	91	A	Excellent

Merging with Outer Join

With outer join, all the records are returned from both right and left dataframes. Here is an example:

Script 21:

```
join_inner_df = scores1_df.merge(scores2_df, on='Subject', how='outer')
join_inner_df.head()
```

Output:

	Subject	Score_x	Grade_x	Remarks_x	Score_y	Grade_y	Remarks_y
0	Mathematics	85.0	B	Good	NaN	NaN	NaN
1	History	98.0	A	Excellent	NaN	NaN	NaN
2	English	76.0	C	Fair	92.0	A	Excellent
3	Chemistry	72.0	C	Fair	91.0	A	Excellent
4	Arts	NaN	NaN	NaN	70.0	C	Fair

4.3. Removing Duplicates

Your datasets will often contain duplicate values, and frequently, you will need to remove these duplicate values. In this section, you will see how to remove duplicate values from your Pandas dataframes.

The following script creates a dummy dataframe for this section.

Script 22:

```
import pandas as pd

scores = [['Mathematics', 85 , 'Science'],
     ['English', 91 , 'Arts'],
     ['History', 95 , 'Chemistry'],
     ['History', 95 , 'Chemistry'],
     ['English', 95 , 'Chemistry'],
     ]

my_df = pd.DataFrame(scores, columns = ['Subject', 'Score', 'Subject'])
my_df.head()
```

Output:

	Subject	Score	Subject
0	Mathematics	85	Science
1	English	91	Arts
2	History	95	Chemistry
3	History	95	Chemistry
4	English	95	Chemistry

From the above output, you can see that there are some duplicate rows (index 2,3), as well as duplicate columns (Subject) in our dataset.

4.3.1. Removing Duplicate Rows

To remove duplicate rows, you can call the drop_duplicates() method, which keeps the first instance and removes all the duplicate rows.

Here is an example that removes the row at index 3, which is a duplicate of the row at index 2, from our dummy dataframe.

Script 23:

```
result = my_df.drop_duplicates()
result.head()
```

Output:

	Subject	Score	Subject
0	Mathematics	85	Science
1	English	91	Arts
2	History	95	Chemistry
4	English	95	Chemistry

If you want to keep the last instance and remove the remaining duplicates, you need to pass the string “last” as the value for the keep attribute of the

drop_duplicates() method.

Here is an example:

Script 24:

```
result = my_df.drop_duplicates(keep='last')
result.head()
```

Output:

	Subject	Score	Subject
0	Mathematics	85	Science
1	English	91	Arts
3	History	95	Chemistry
4	English	95	Chemistry

Finally, if you want to remove all the duplicate rows from your Pandas dataframe without keeping any instance, you can pass the Boolean value False as the value for the keep attribute, as shown in the example below.

Script 25:

```
result = my_df.drop_duplicates(keep=False)
result.head()
```

Output:

	Subject	Score	Subject
0	Mathematics	85	Science
1	English	91	Arts
4	English	95	Chemistry

By default, the drop_duplicates() method only removes duplicate rows, where all the columns contain duplicate values. If you want to remove rows based on duplicate values in a subset of columns, you need to pass the column list to the subset attribute.

For instance, the script below removes all rows where the Score column contains duplicate values.

Script 26:

```
result = my_df.drop_duplicates(subset=['Score'])
result.head()
```

Output:

	Subject	Score	Subject
0	Mathematics	85	Science
1	English	91	Arts
2	History	95	Chemistry

4.3.2. Removing Duplicate Columns

There are two main ways to remove duplicate columns in Pandas. You can remove two columns with the duplicate name, or you can remove two columns containing duplicate values for all the rows.

Let's create a dummy dataset that contains duplicate column names and duplicate values for all rows for different columns.

Script 27:

```
import pandas as pd

scores = [['Mathematics', 85 , 'Science', 85 ],
     ['English', 91 , 'Arts', 91 ],
     ['History', 95 , 'Chemistry', 95 ],
     ['History', 95 , 'Chemistry', 95 ],
```

```
    ['English', 95 , 'Chemistry', 95 ],
    ]

my_df = pd.DataFrame(scores, columns = ['Subject', 'Score', 'Subject', 'Percentage'])
my_df.head()
```

Output:

	Subject	Score	Subject	Percentage
0	Mathematics	85	Science	85
1	English	91	Arts	91
2	History	95	Chemistry	95
3	History	95	Chemistry	95
4	English	95	Chemistry	95

You can see that the above dataframe contains two columns with the name *Subject* . Also, the Score and Percentage columns have duplicate values for all the rows.

Let's first remove the columns with duplicate names. Here is how you can do that using the duplicated() method.

Script 28:

```
result = my_df.loc[:,~my_df.columns.duplicated()]
result.head()
```

Output:

	Subject	Score	Percentage
0	Mathematics	85	85
1	English	91	91
2	History	95	95
3	History	95	95
4	English	95	95

To remove the columns with the same values, you can convert columns to rows using the "T" attribute and then call drop_ duplicates() on the transposed dataframe. Finally, you can again transpose the resultant dataframe, which will have duplicate columns removed. Here is a sample script on how you can do that.

Script 29:

```
result = my_df.T.drop_duplicates().T
result.head()
```

Output:

	Subject	Score	Subject
0	Mathematics	85	Science
1	English	91	Arts
2	History	95	Chemistry
3	History	95	Chemistry
4	English	95	Chemistry

4.4. Pivot and Crosstab

You can pivot a Pandas dataframe using a specific column or row. With pivoting, you can set values in columns as index values, as well as column headers.

The following script imports the *Flights* dataset from the seaborn library.

Script 30:

```
import matplotlib.pyplot as plt
import seaborn as sns

flights_data = sns.load_dataset('flights')

flights_data.head()
```

Output:

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121

You will be pivoting the above dataframe. To pivot a dataframe, you can use the pivot_table() function.

For instance, the pivot_table() function in the following script returns a dataframe where rows represent months, columns represent years from 1949 to 1960, and each cell contains the number of passengers traveling in a specific month of a specific year.

Script 31:

```
flights_data_pivot =flights_data.pivot_table(index='month', columns='year', values='passengers')
flights_data_pivot.head()
```

Output:

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
Jan	112	115	145	171	196	204	242	284	315	340	360	417
Feb	118	126	150	180	196	188	233	277	301	318	342	391
Mar	132	141	178	193	236	235	267	317	356	362	406	419
Apr	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472

The crosstab() function is used to plot cross-tabulation between two columns. Let's import the Titanic dataset from the Seaborn library and plot a cross tab matrix between passenger class and age columns for the Titanic dataset.

Look at the following two scripts on how to do that:

Script 32:

```
import matplotlib.pyplot as plt
import seaborn as sns

import pandas as pd

# sets the default style for plotting
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')
```

Script 33:

```
pd.crosstab(titanic_data["class"], titanic_data["age"], margins=True )
```

Output:

age	0.42	0.67	0.75	0.83	0.92	1.0	2.0	3.0	4.0	5.0	...	63.0	64.0	65.0	66.0	70.0	70.5	71.0	74.0	80.0	All
class																					
First	0	0	0	0	1	0	1	0	1	0	...	1	2	2	0	1	0	2	0	1	186
Second	0	1	0	2	0	2	2	3	2	1	...	0	0	0	1	1	0	0	0	0	173
Third	1	0	2	0	0	5	7	3	7	3	...	1	0	1	0	0	1	0	1	0	355
All	1	1	2	2	1	7	10	6	10	4	...	2	2	3	1	2	1	2	1	1	714

4 rows × 89 columns

4.5. Discretization and Binning

Discretization or binning refers to creating categories or bins using numeric data. For instance, based on age, you may want to assign categories such as toddler, young, adult, and senior to the passengers in the Titanic dataset. You can do this using binning.

Let's see an example. The following script imports the Titanic dataset.

Script 34:

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic_data = sns.load_dataset('titanic')
```

You can use the cut method from the Pandas dataframe to perform binning. First, the column name is passed to the "x" attribute. Next, a list containing ranges for bin values is passed to the "bins" attribute, while the bin or category names are passed to the "labels" parameter.

The following script assigns a category *toddler* to passengers between the ages 0–5, *young* to passengers aged 5–20, *adult* to passengers aged 20–60, and *senior* to passengers aged 60–100.

Script 35:

```
titanic_data['age_group']=pd.cut(x = titanic_data['age'], bins = [ 0 , 5 , 20 , 60 , 100 ], labels =
    ["toddler", "young", "adult","senior"])

titanic_data['age_group'].value_counts()
```

Output:

```
adult    513
young    135
toddler   44
senior    22
Name: age_group, dtype: int64
```

Further Readings – Pandas Grouping and Aggregation

1. Check the official documentation here (https://bit.ly/3qnp87I) to learn more about Merging and Concatenating dataframes.
2. To learn more about the GroupBy statement in Pandas, check this official documentation link (https://bit.ly/31zNGzU).

Hands-on Time – Exercises

Now, it is your turn. Follow the instructions in **the exercises below** to check your understanding of Pandas techniques that you learned in this chapter. The answers to these questions are given at the end of the book.

Exercise 4.1

Question 1:

To horizontally concatenate two Pandas (pd) dataframes A and B, you can use the following function:

A. pd.concat([A, B], ignore_index = True)

B. pd.concat([A, B], axis = 1, ignore_index = True)

C. pd.append([A, B] ignore_index = True)

D. pd.join([A, B], axis = 1, ignore_index = True)

Question 2:

To find the number of unique values in column X of Pandas dataframe df, you can use the groupby clause as follows:

A. df.groupby(«X»).nunique

B. df.groupby(«X»).unique

C. df.groupby(«X»).ngroups

D. df.groupby(«X»).nvalues

Question 3:

To remove all the duplicate rows from a Pandas dataframe df, you can use the following function:

A. df.drop_duplicates(keep=False)

B. df.drop_duplicates(keep='None')

C. df.drop_duplicates(keep='last')

D. df.drop_duplicates()

Exercise 4.2

From the Titanic dataset, find the minimum, maximum, median, and mean values for ages and fare paid by passengers of different genders. You can access the Titanic dataset using the following Seaborn command:

```
import seaborn as sns

titanic_data = sns.load_dataset('titanic')
```

Pandas for Data Visualization

5.1. Introduction

In the previous chapters, you have seen how the Pandas library can be used to perform different types of data manipulation and analysis tasks. Data visualization is another extremely important data analysis task. Luckily enough for us, the Pandas library offers data visualization functionalities, as well, in the form of various charts.

In this chapter, you will see how the Pandas library can be used to plot different types of visualizations. The Pandas library is probably the easiest library for data plotting, as you will see in this chapter.

5.2. Loading Datasets with Pandas

Before you can plot any visualization with the Pandas library, you need to read data into a Pandas dataframe. The best way to do so is via the read_csv() method. The following script shows how to read the Titanic dataset into a dataframe named titanic_data. You can give any name to the dataframe.

Note: In the previous chapters, you have been importing the Titanic dataset directly from the Seaborn library. However, for practice, in this chapter, the Titanic dataset is imported via the titanic_data.csv file, which you can find in the *Data* folder of the book resources.

Script 1:

```
import pandas as pd
titanic_data = pd.read_csv(r"D:\Datasets\titanic_data.csv")
titanic_data.head()
```

Output:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

5.3. Plotting Histograms with Pandas

Let's now see how to plot different types of plots with the Pandas dataframe. The first one we are going to plot is a Histogram.

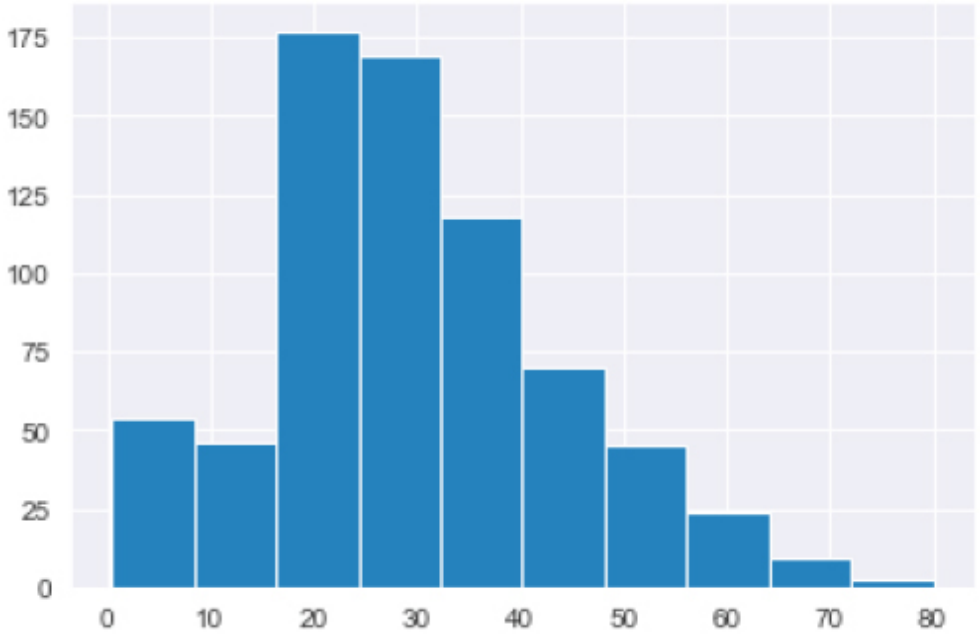
There are multiple ways to plot a graph in Pandas. The first way is to select the dataframe column by specifying the name of the column in square brackets that follows the dataframe name and then append the plot name via dot operator.

The following script plots a histogram for the Age column of the Titanic dataset using the hist() function. It is important to mention that behind the scenes, the Pandas library makes use of the Matplotlib plotting functions. Therefore, you need to import Matplotlib's pyplot module before you can plot Pandas visualizations.

Script 2:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("darkgrid")
titanic_data['Age'].hist()
```

Output:

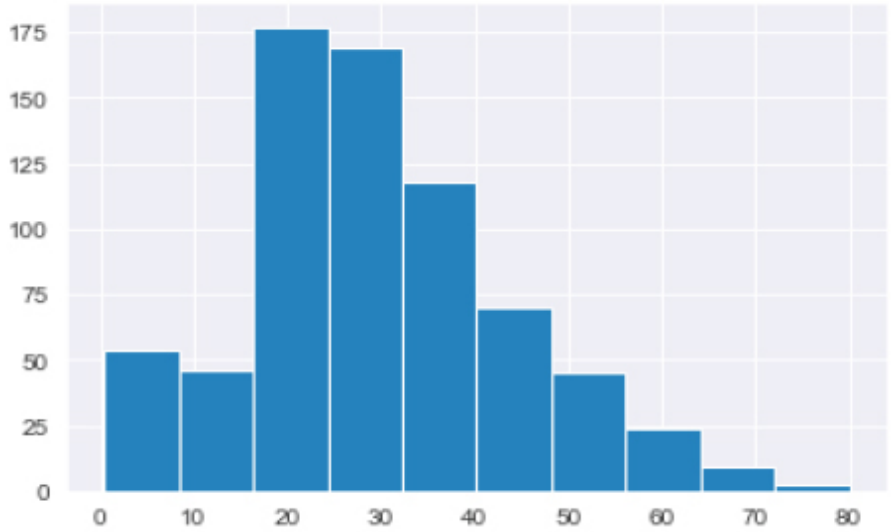


The other way to plot a graph via Pandas is by using the plot() function. The type of plot you want to plot is passed to the kind attribute of the plot() function. The following script uses the plot() function to plot a histogram for the Age column of the Titanic dataset.

Script 3:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("darkgrid")
titanic_data['Age'].plot(kind='hist')
```

Output:

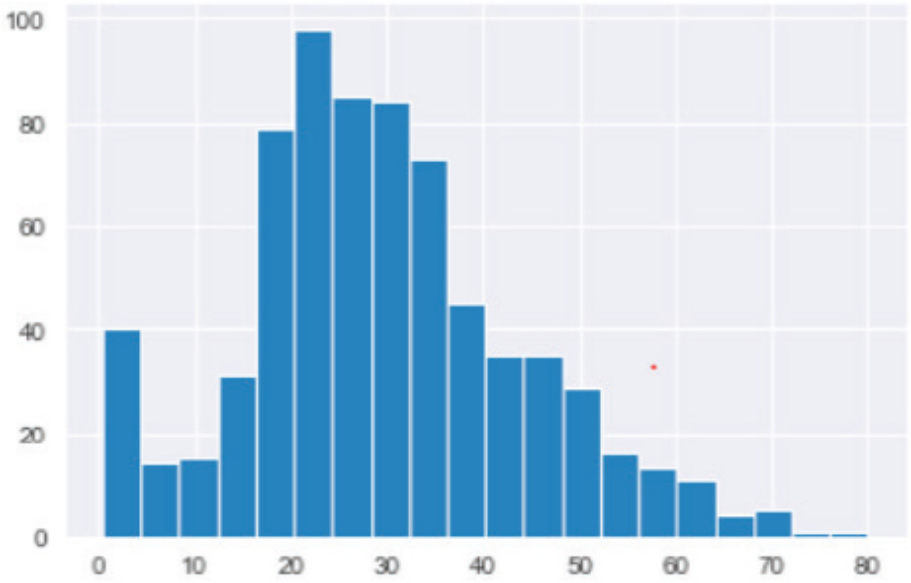


By default, a Pandas histogram divides the data into 10 bins. You can increase or decrease the number of bins by passing an integer value to the bins parameter. The following script plots a histogram for the Age column of the Titanic dataset with 20 bins.

Script 4:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('darkgrid')
titanic_data['Age'].hist(bins = 20 )
```

Output:



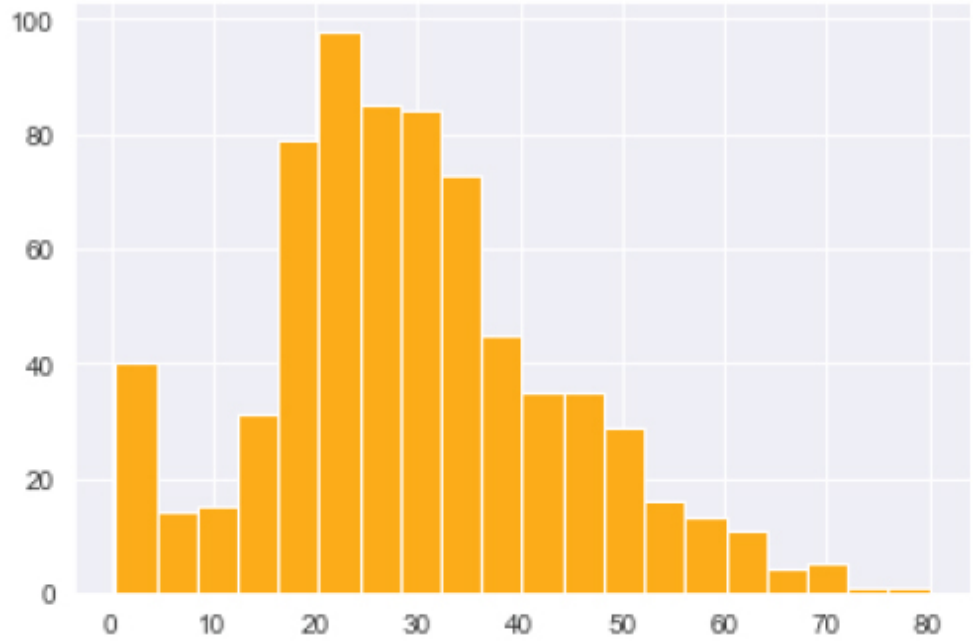
Finally, you can change the color of your histogram by specifying the color name to the color attribute, as shown below.

Script 5:

```
titanic_data['Age'].hist(bins = 20 , color = 'orange')
```

Output:

```
<AxesSubplot:>
```



5.4. Pandas Line Plots

To plot line plots via a Pandas dataframe, we will use the *Flights* dataset. The following script imports the Flights dataset from the built-in Seaborn library.

Script 6:

```
flights_data = sns.load_dataset('flights')

flights_data.head()
```

Output:

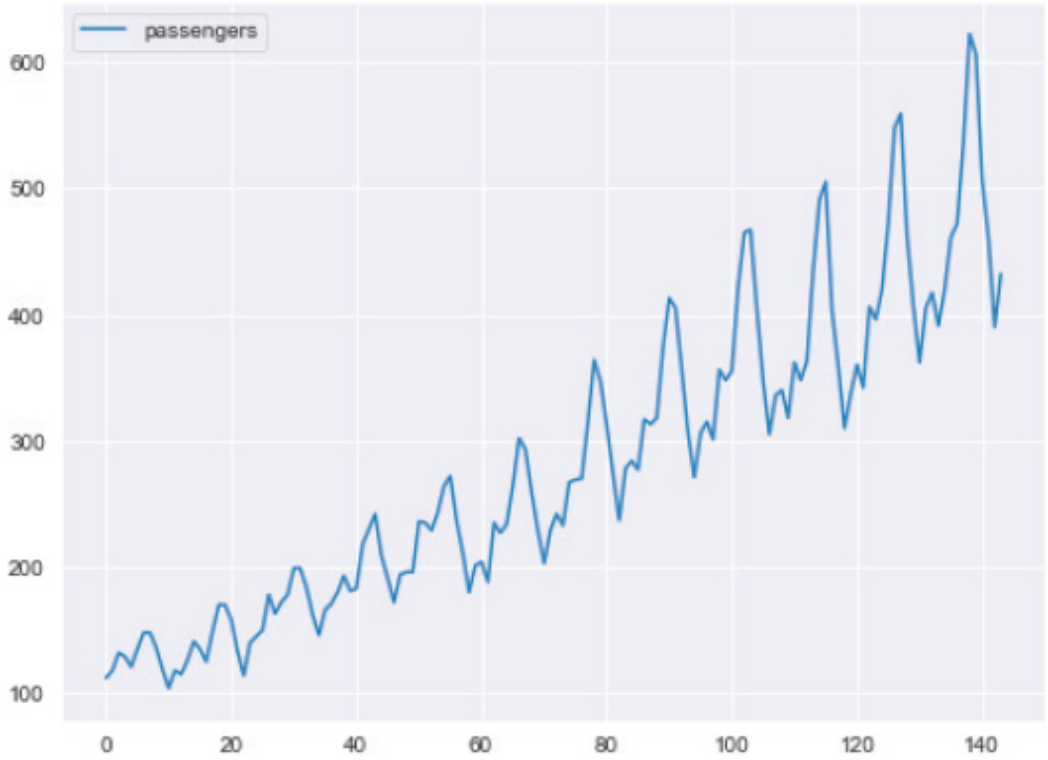
	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

By default, the index serves as the x-axis. In the above script, the left-most column, i.e., containing 0,1,2 … is the index column. To plot a line plot, you have to specify the column names for the x and y axes. If you specify only the column value for the y-axis, the index is used as the x-axis. The following script plots a line plot for the passengers column of the Flights data.

Script 7:

```
flights_data.plot.line(y='passengers', figsize=( 8 , 6 ))
```

Output:

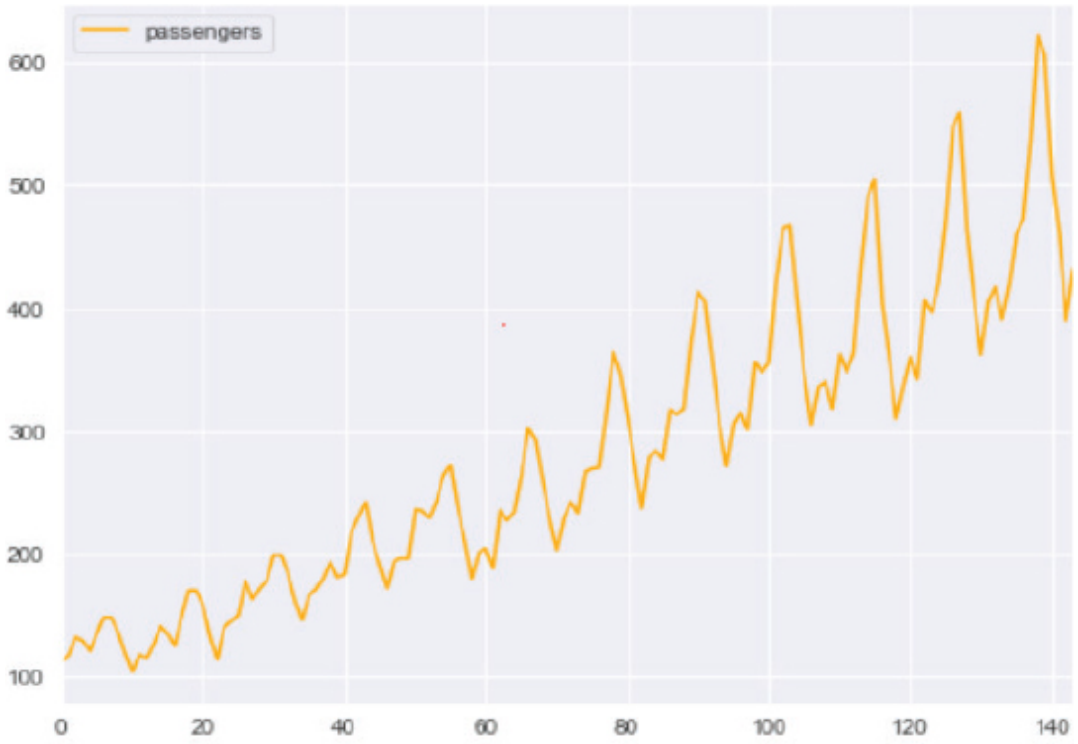


Similarly, you can change the color of the line plot via the color attribute, as shown below.

Script 8:

```
flights_data.plot.line( y='passengers', figsize=( 8 , 6 ), color = 'orange')
```

Output:

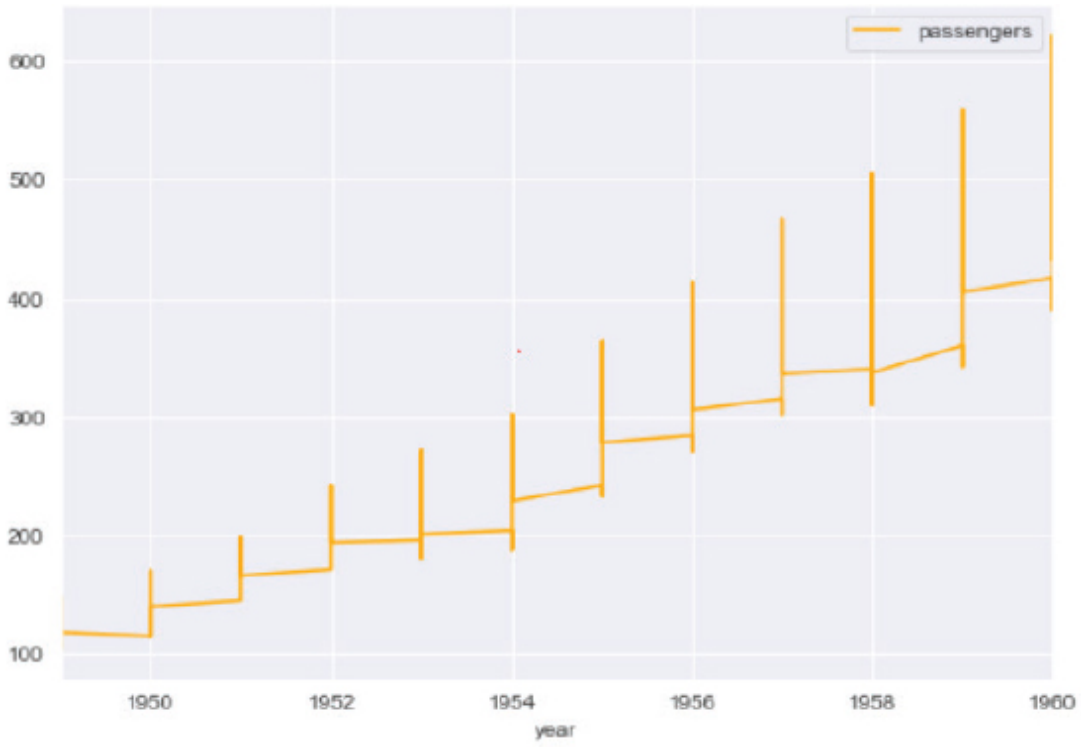


In the previous examples, we didn't pass the column name for the x-axis. So let's see what happens when we specify the year as the column name for the x-axis.

Script 9:

```
flights_data.plot.line(x ='year', y='passengers', figsize=( 8 , 6 ), color = 'orange')
```

Output:



The output shows that for each year, we have multiple values. This is because each year has 12 months. However, the overall trend remains the same, and the number of passengers traveling by air increases as the years pass.

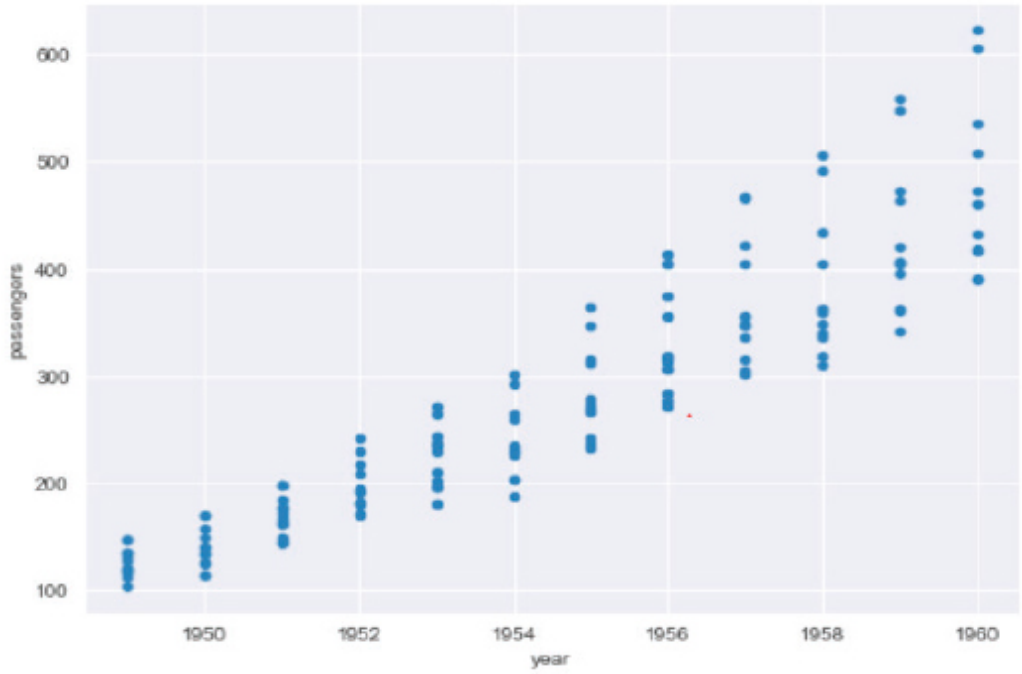
5.5. Pandas Scatter Plots

To plot scatter plots with Pandas, the scatter() function is used. The following script plots a scatter plot containing the year on the x-axis and the number of passengers on the y-axis.

Script 10:

```
flights_data.plot.scatter(x='year', y='passengers', figsize=( 8 , 6 ))
```

Output:

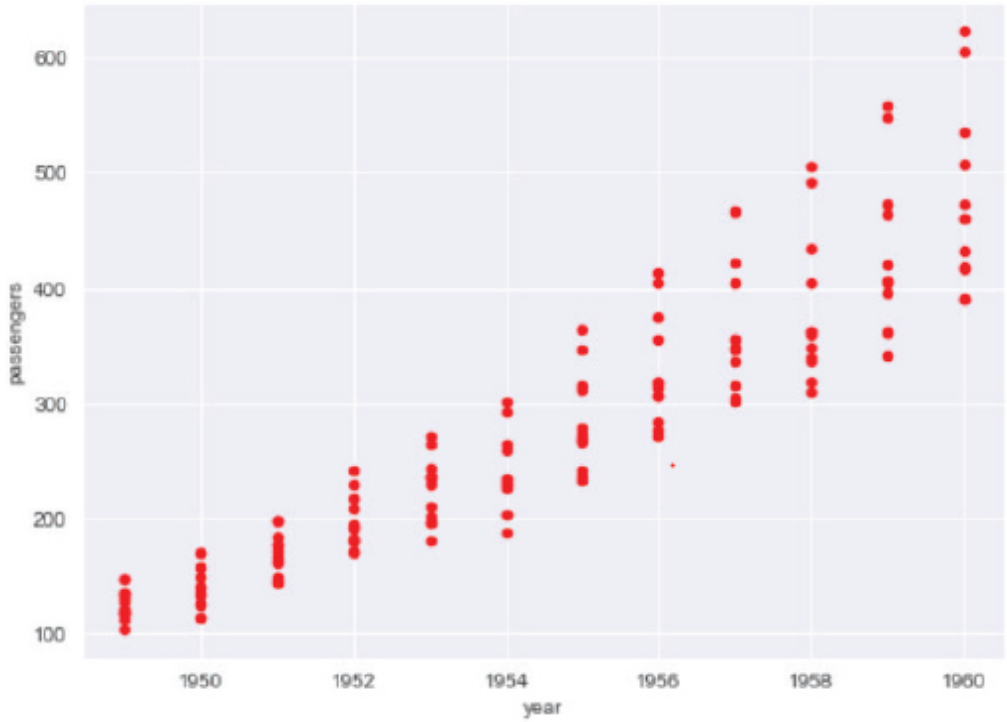


Like a line plot and histogram, you can also change the color of a scatter plot by passing the color name as the value for the color attribute. Look at the following script.

Script 11:

```
flights_data.plot.scatter(x='year', y='passengers', color='red', figsize=( 8 , 6 ))
```

Output:



5.6. Pandas Bar Plots

To plot bar plots with Pandas, you need a list of categories and a list of values. This list of categories and values must have the same length. Let's plot a bar plot that shows the average age of male and female passengers.

First, we need to calculate the mean age of both male and female passengers traveling in the unfortunate Titanic ship. The groupby() method of the Pandas dataframe can be used to apply aggregate function concerning categorical columns. The following script returns the mean values for the ages of male and female passengers on *Titanic* .

Script 12:

```
titanic_data = pd.read_csv(r"D:\Datasets\titanic_data.csv") titanic_data.head()
sex_mean = titanic_data.groupby("Sex")["Age"].mean()

print (sex_mean)
print (type(sex_mean.tolist()))
```

Output:

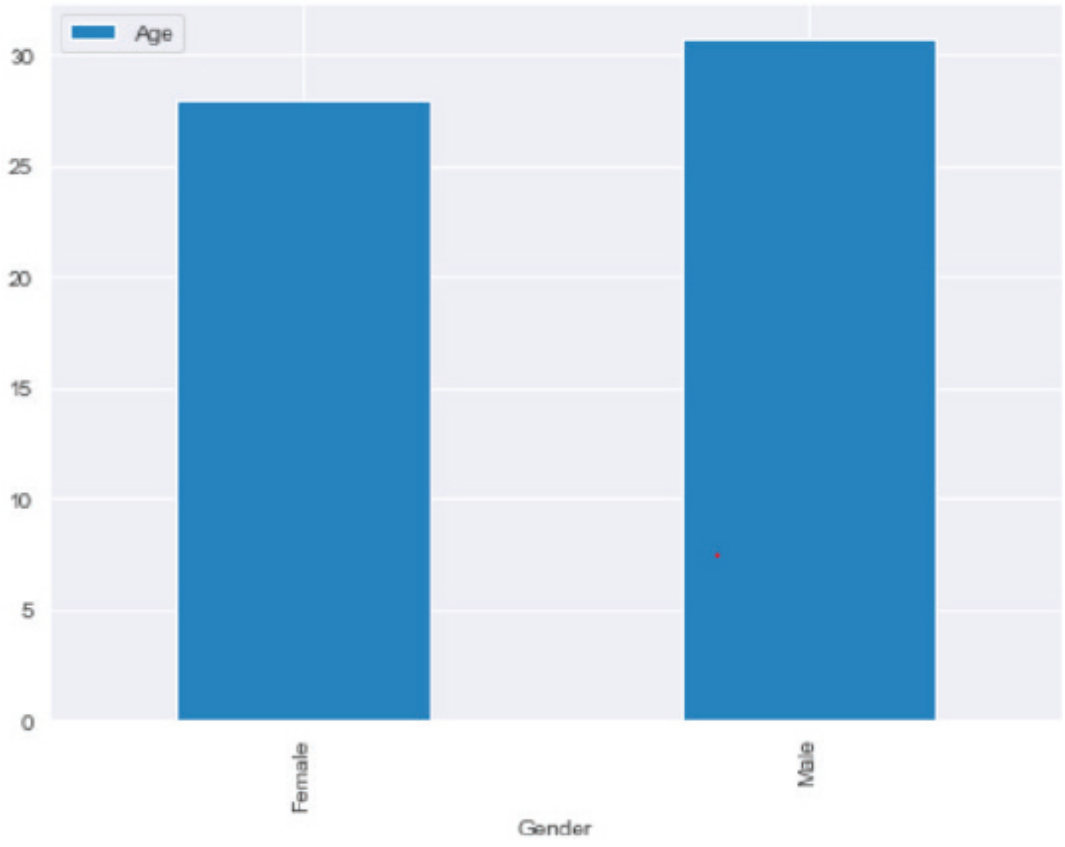
```
Sex
female 27.915709
male   30.726645
Name:  Age, dtype: float64
<class 'list'>
```

Next, we need to create a new Pandas dataframe with two columns: Gender and Age. Then, we can simply use the bar() method to plot a bar plot that displays the average ages of male and female passengers on *Titanic* .

Script 13:

```
df = pd.DataFrame({'Gender':['Female', 'Male'], 'Age':sex_mean.tolist()})
ax = df.plot.bar(x='Gender', y='Age', figsize=( 8 , 6 ))
```

Output:

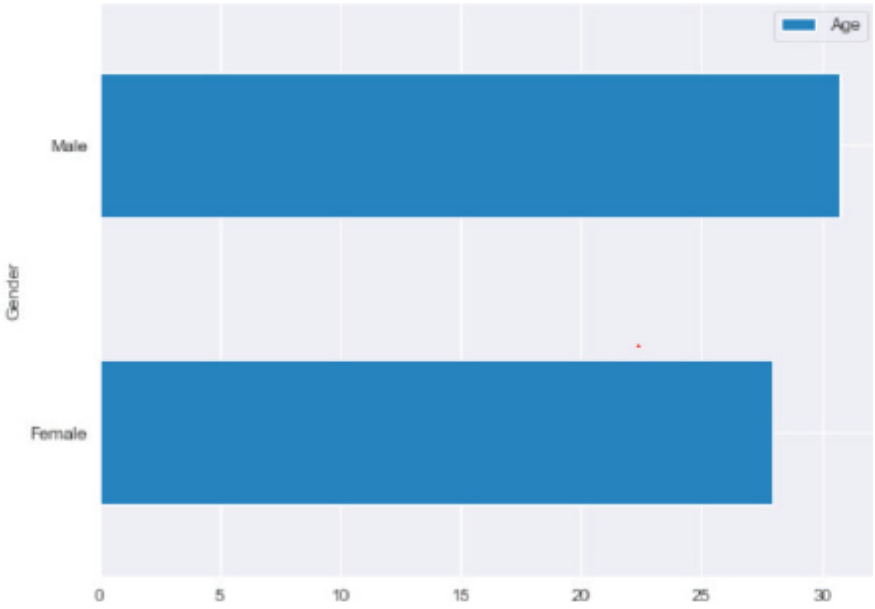


You can also plot horizontal bar plots via the Pandas library. To do so, you need to call the barh() function, as shown in the following example.

Script 14:

```
df = pd.DataFrame({'Gender':['Female', 'Male'], 'Age':sex_ mean.tolist()})
ax = df.plot.barh(x='Gender', y='Age', figsize=( 8 , 6 ))
```

Output:

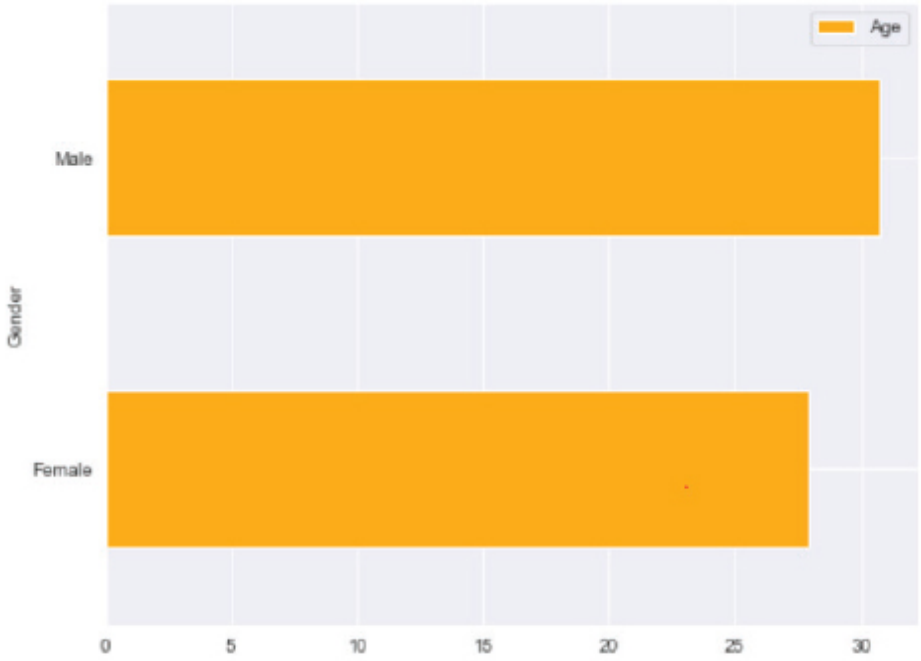


Finally, like all the other Pandas plots, you can change the color of both vertical and horizontal bar plots by passing the color name to the color attribute of the corresponding function.

Script 15:

```
df = pd.DataFrame({'Gender':['Female', 'Male'], 'Age':sex_ mean.tolist()})
ax = df.plot.barh(x='Gender', y='Age', figsize=( 8 , 6 ), color = 'orange')
```

Output:



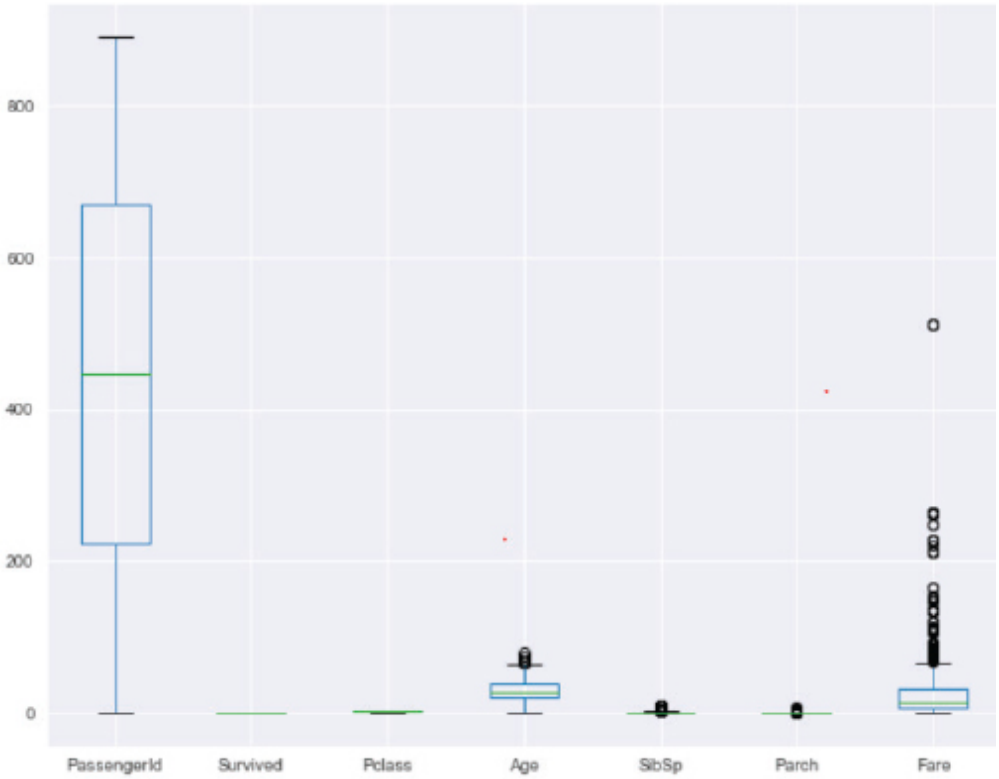
5.7. Pandas Box Plots

To plot box plots via the Pandas library, you need to call the box() function. The following script plots box plots for all the numeric columns in the Titanic dataset.

Script 16:

```
titanic_data = pd.read_csv(r"D:\Datasets\titanic_data.csv")
titanic_data.plot.box(figsize=( 10 , 8 ))
```

Output:



5.8. Pandas Hexagonal Plots

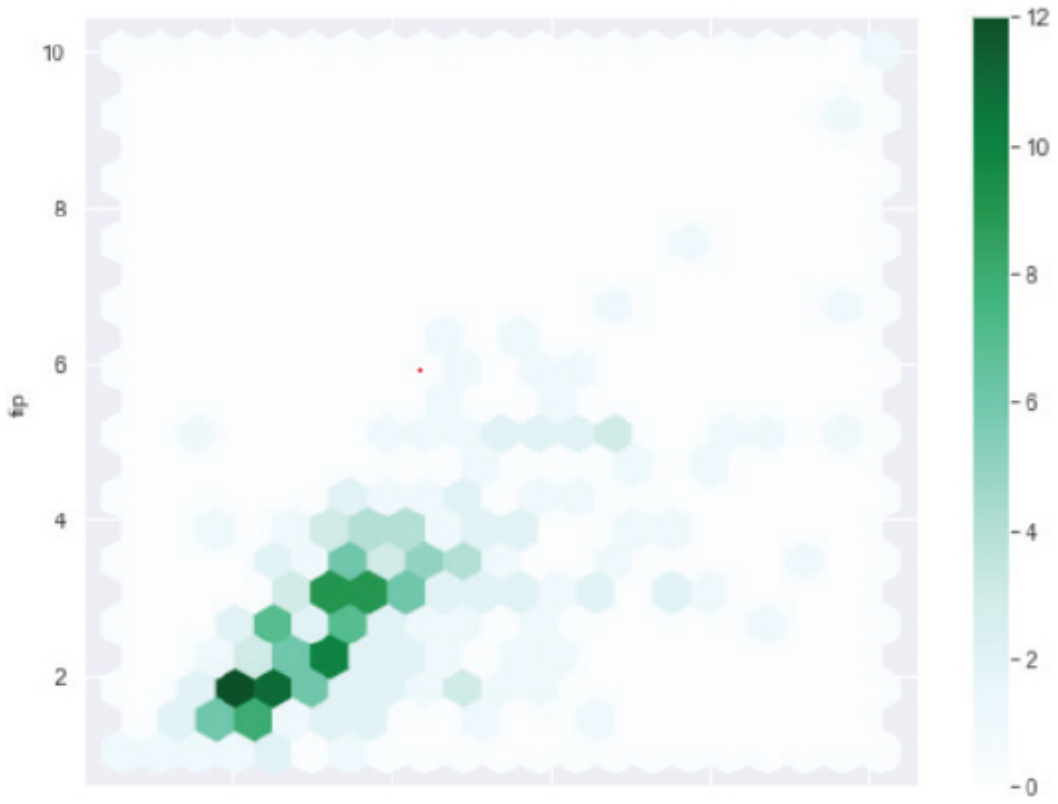
Hexagonal plots are used to plot the density of occurrence of values for a specific column. The hexagonal plots will be explained with the help of the *Tips* dataset. The following script loads the Tips dataset from the Seaborn library and then plots a hexagonal plot that shows values from the total_ bill column on the x-axis and values from the tip column on the y-axis.

Script 17:

```
tips_data = sns.load_dataset('tips')

tips_data.plot.hexbin(x='total_bill', y='tip', gridsize= 20 , figsize=( 8 , 6 ))
```

The output shows that most of the time, the tip is between two and four dollars.

Output:

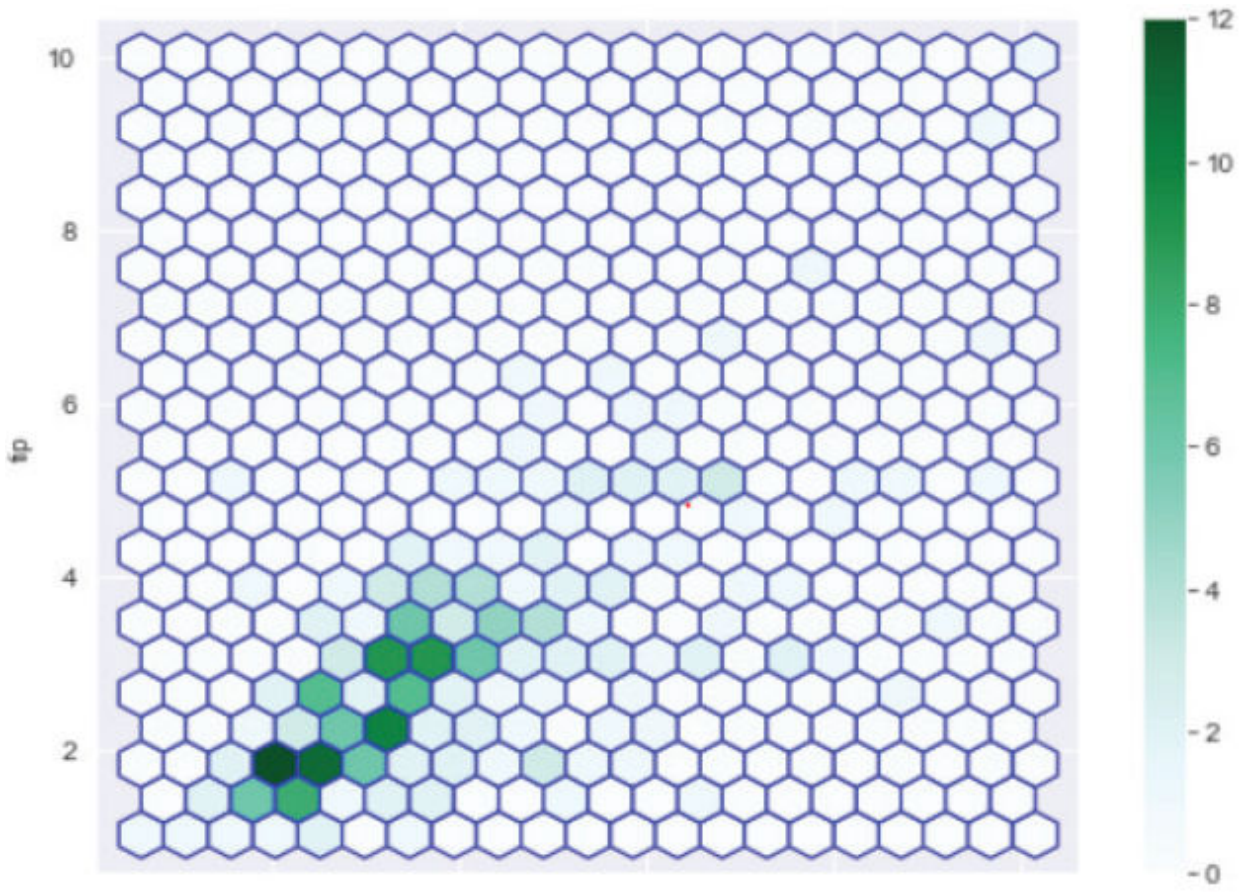


As always, you can change the color of the hexagonal plot by specifying the color name for the color attribute, as shown below.

Script 18:

```
tips_data.plot.hexbin(x='total_bill', y='tip', gridsize= 20 ,
figsize=( 8 , 6 ), color = 'blue')
```

Output:



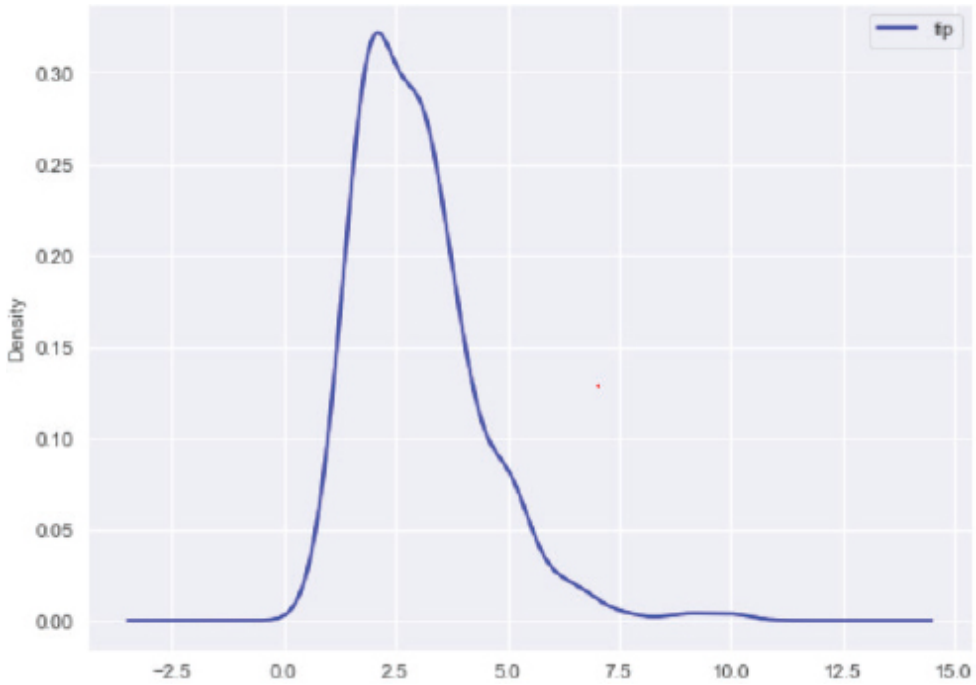
5.9. Pandas Kernel Density Plots

You can also plot Kernel Density Estimation plots with the help of the Pandas kde() function. The following script plots a KDE for the tip column of the Tips dataset.

Script 19:

```
tips_data.plot.kde( y='tip', figsize=(8,6), color = 'blue')
```

Output:

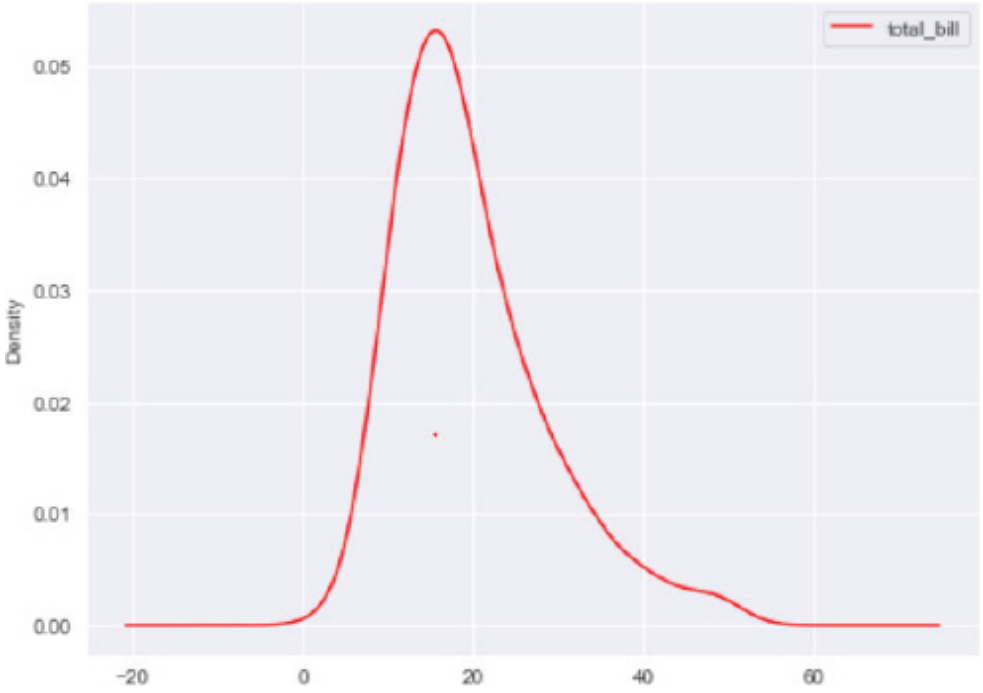


To change the color of a KDE plot, all you have to do is pass the color name to the color attribute of the kde() function, as shown below.

Script 20:

```
tips_data.plot.kde( y='total_bill', figsize=( 8 , 6 ), color = 'red')
```

Output:



5.10. Pandas Pie Charts

You can also plot a pie chart with Pandas. To do so, you need to pass “pie” as the value for the kind attribute of the plot() function. The plot() function is needed to be called via an object, which contains categories and the number of items per category.

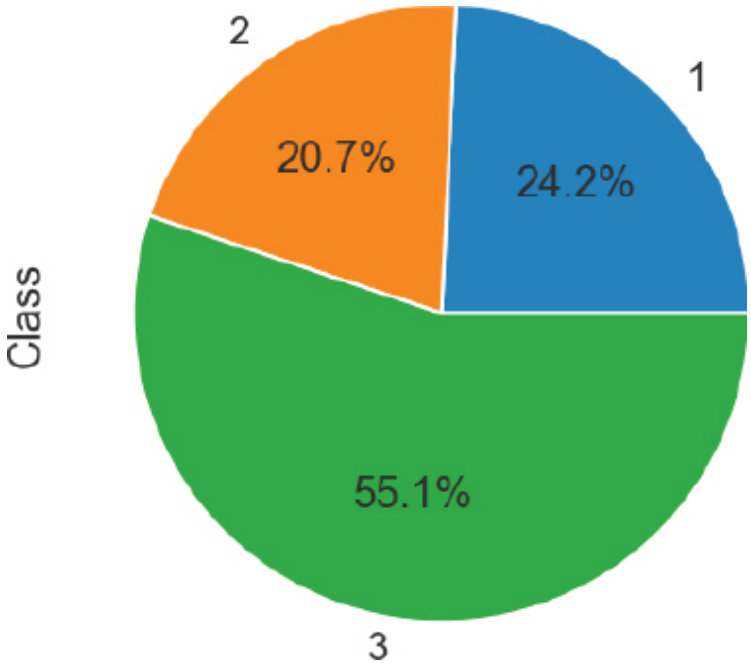
For instance, the script below plots a pie chart that shows the distribution of passengers belonging to different classes.

Script 21:

```
import seaborn as sns
import pandas as pd

sns.set_style("darkgrid")
sns.set_context("poster")

titanic_data = pd.read_csv(r"D:\Datasets\titanic_data.csv")
titanic_data.head()

titanic_data.groupby('Pclass').size().plot(kind='pie',
                y = "PClass",
                label = "Class",
```

```
autopct='%1.1f%%',
figsize=( 10 , 8 ))
```

Output:



Further Readings – Pandas Data Visualization

Check the official documentation here (https://bit.ly/3EQWwI2) to learn more about data visualization with Pandas.

Hands-on Time – Exercises

Now, it is your turn. Follow the instructions in **the exercises below** to check your understanding of Pandas data visualization techniques that you learned in this chapter. The answers to these questions are given at the end of the book.

Exercise 5.1

Question 1:

Which attribute is used to change the color of the Pandas graph?

A. set_color()
B. define_color()
C. color()
D. None of the above

Question 2:

Which Pandas function is used to plot a horizontal bar plot?

A. horz_bar()
B. barh()
C. bar_horizontal()
D. horizontal_bar()

Question 3:

Which attribute is used to define the number of bins in a Pandas histogram plot?

A. n_bins
B. bins
C. number_bins
D. None of the above

Exercise 5.2

Display a bar plot using the Titanic dataset that displays the average age of the passengers who survived vs. those who did not survive.

You can find the “titanic_data.csv” file in the Data folder of the book resources.

Handling Time-Series Data with Pandas

Time-series data is a type of data that is dependent on time and changes with time. For instance, the hourly temperature for a specific place changes after every hour and is dependent on time. Similarly, the stock prices of a particular company change with every passing day.

In this chapter, you will see how Pandas handles time-series data. You will start by creating the TimeStamp data type in Pandas. Next, you will see how Pandas allows you to perform time-sampling and time-shifting on time series data. Finally, you will study rolling window functions on time-series data with Pandas. Along the way, you will also study how to plot the time-series data using Pandas.

6.1. Introduction to Time-Series in Pandas

The TimeStamp data type in Pandas is the most basic unit for storing time-step data. Let's see this with the help of an example.

The following script uses the date_range() function to create a collection that contains dates in the form of time stamps.

Script 1:

```
import pandas as pd
import numpy as np

from datetime import datetime

dates = pd.date_range(start='1/1/2021', end='6/30/2021')
print (len(dates))
print (dates)
```

Output:

```
181
DatetimeIndex(['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04',
               '2021-01-05', '2021-01-06', '2021-01-07', '2021-01-08',
               '2021-01-09', '2021-01-10',
               ...
               '2021-06-21', '2021-06-22', '2021-06-23', '2021-06-24',
               '2021-06-25', '2021-06-26', '2021-06-27', '2021-06-28',
               '2021-06-29', '2021-06-30'],
              dtype='datetime64[ns]', length=181, freq='D')
```

Let's check the data type of the first item in our date range.

Script 2:

```
type (dates[ 0 ])
```

You can see the date range collection stores data in the form of TimeStamp data type.

Output:

```
pandas._libs.tslibs.timestamps.Timestamp
```

Let's now create a dataframe that contains the date range that we just created and some random values between 0–50. The date range column is named as *Date* , while the column for random values is named as *Temperature* .

Script 3:

```
date_df = pd.DataFrame(dates, columns=['Date'])
date_df['Temperature'] = np.random.
    randint( 0 , 50 ,size=(len(dates)))
date_df.head()
```

Output:

	Date	Temperature
0	2021-01-01	32
1	2021-01-02	13
2	2021-01-03	8
3	2021-01-04	1
4	2021-01-05	2

Most of the time, you will need to convert your TimeStamps into dates and then set the converted (date type column) as the index column. Here is how you can do that.

Script 4:

```
date_df['Date'] = pd.to_datetime(date_df['Date'])
date_df = date_df.set_index('Date')

date_df.head()
```

In the output below, you can see that your Date column is set as the index column.

Output:

	Temperature
Date	
2021-01-01	32
2021-01-02	13
2021-01-03	8
2021-01-04	1
2021-01-05	2

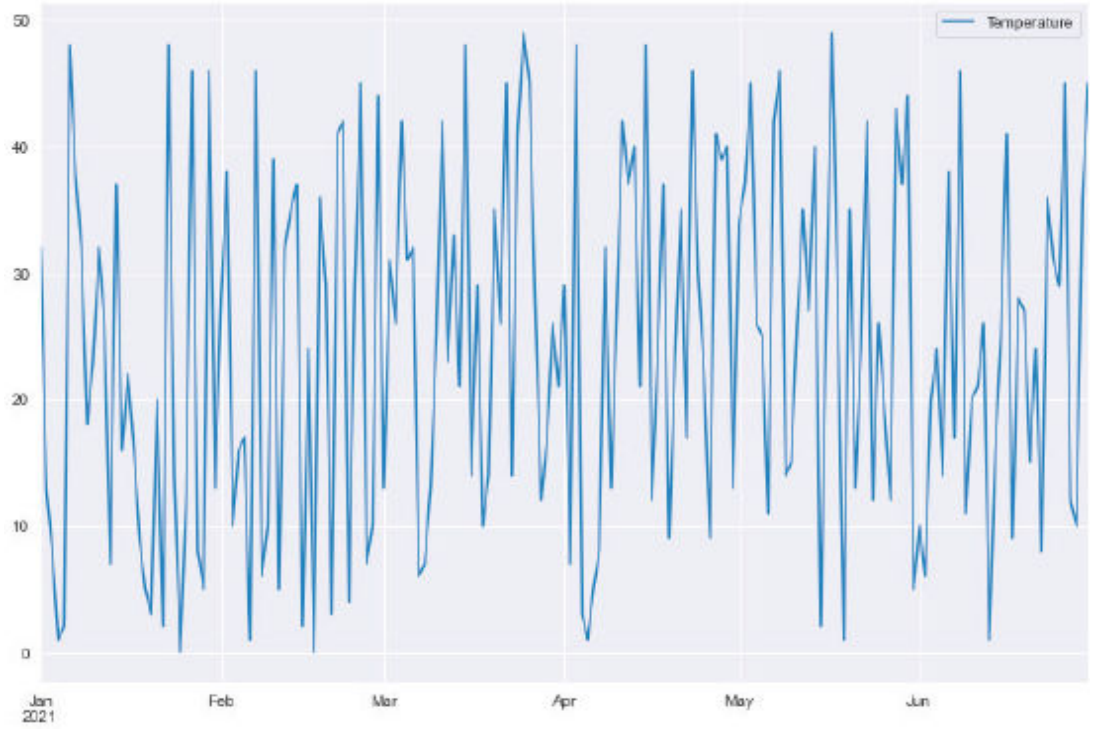
Finally, you can plot your time-series data (values concerning some time unit) using one of the Pandas plots that you studied in the previous chapter.

For instance, the script below plots a line chart that shows the Temperature against date values in the Date column.

Script 5:

```
import seaborn as sns
sns.set_style("darkgrid")
date_df.plot.line( y='Temperature', figsize=( 12 , 8 ))
```

Output:



6.2. Time Resampling and Shifting

In this section, you will see how to resample and shift the time series data with Pandas.

You will work with Google Stock Price data from 6th January 2015 to 7th January 2020. The dataset is available in the *Data* folder of the book resources by the name google_data.csv. The following script reads the data into a Pandas dataframe.

Script 6:

```
import seaborn as sns
import pandas as pd

sns.set_style("darkgrid")

google_stock = pd.read_csv(r"D:Datasets\google_data.csv")
google_stock.head()
```

Output:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-01-06	513.589966	514.761719	499.678131	500.585632	500.585632	2899900
1	2015-01-07	505.611847	505.855164	498.281952	499.727997	499.727997	2065000
2	2015-01-08	496.626526	502.101471	489.655640	501.303680	501.303680	3353500
3	2015-01-09	503.377991	503.537537	493.435272	494.811493	494.811493	2071300
4	2015-01-12	493.584869	494.618011	486.225067	491.201416	491.201416	2326700

If you look at the dataset header, the index column by default is the left-most column. The x-axis will use the index column to plot a line plot.

However, we want to plot stock prices concerning the date. To do so, we first need to set the date as the index column. The date column currently contains dates in a string format.

We first need to convert values in the date column to date format. We can use pd.to_datetime() function for that purpose. Next, to set the Date column as the index column, we can use the set_index() function, as shown below. We can then simply use the line() function and pass the column name to visualize the y parameter.

The following script prints the opening stock prices of Google stock over five years.

Script 7:

```
google_stock['Date'] = google_stock['Date'].apply(pd.to_datetime)
google_stock.set_index('Date', inplace=True)
google_stock.plot.line( y='Open', figsize=( 12 , 8 ))
```

Output:

Let's now see how to perform time sampling and time-shifting with time-series data.

6.2.1. Time Sampling with Pandas

Time sampling refers to grouping data over a certain period using an aggregate function such as min, max, count, mean, etc.

To do resampling, you have to use the resample() function. The timeframe is passed to the rule attribute of the resample() function. Finally, you have to append the aggregate function at the end of the resample() function.

The following script shows the average values for all the columns of Google stock data, grouped by year. In the output, you can see five rows since our dataset contains five years of Google stock prices. Here, we pass A as the value for the rule attribute, which refers to yearly data.

Script 8:

```
google_stock.resample(rule='A').mean()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume
2015-12-31	602.676217	608.091468	596.722047	602.678382	602.678382	2.071960e+06
2016-12-31	743.732459	749.421629	737.597905	743.486707	743.486707	1.832266e+06
2017-12-31	921.121193	926.898963	915.331412	921.780837	921.780837	1.476514e+06
2018-12-31	1113.554101	1125.777606	1101.001658	1113.225134	1113.225134	1.741965e+06
2019-12-31	1187.009821	1196.787599	1178.523734	1188.393057	1188.393057	1.414085e+06
2020-12-31	1346.470011	1379.046672	1345.697998	1374.079997	1374.079997	1.441767e+06

Similarly, to plot the monthly mean values for all the columns in the Google stock dataset, you will need to pass **M** as a value for the **rule** attribute, as shown below.

Script 9:

```
google_stock.resample(rule='M').mean()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume
2015-01-31	510.388728	515.352041	503.988300	510.248006	510.248006	2.595550e+06
2015-02-28	534.448454	540.111910	530.943141	536.519088	536.519088	1.715495e+06
2015-03-31	558.825290	562.627577	554.057018	558.183871	558.183871	1.756709e+06
2015-04-30	539.966811	543.839108	535.114912	539.304467	539.304467	2.017938e+06
2015-05-31	535.470502	539.167248	530.856650	535.238998	535.238998	1.593295e+06
...	...	...	...	...	...	...
2019-09-30	1217.599005	1228.892249	1209.628491	1220.839520	1220.839520	1.344970e+06
2019-10-31	1230.809995	1242.260774	1223.923043	1232.711744	1232.711744	1.250361e+06
2019-11-30	1302.348492	1311.498956	1296.424707	1304.278992	1304.278992	1.246170e+06
2019-12-31	1340.861415	1348.178531	1334.039190	1340.867635	1340.867635	1.302719e+06
2020-01-31	1346.470011	1379.046672	1345.697998	1374.079997	1374.079997	1.441767e+06

61 rows × 6 columns

In addition to aggregate values for all the columns, you can resample data concerning a single column. For instance, the following script prints the yearly mean values for the opening stock prices of Google stock for five years.

Script 10:

```
google_stock['Open'].resample('A').mean()
```

Output:

```
Date
2015-12-31   602.676217
2016-12-31   743.732459
2017-12-31   921.121193
2018-12-31  1113.554101
2019-12-31  1187.009821
2020-12-31  1346.470011
Freq: A-DEC, Name: Open, dtype: float64
```

The list of possible values for the rule attribute is given below:

B – business day frequency

C – custom business day frequency (experimental)
D – calendar day frequency
W – weekly frequency
M – month-end frequency
SM – semi-month end frequency (15th and end of the month)
BM – business month-end frequency
CBM – custom business month-end frequency
MS – month start frequency
SMS – semi-month start frequency (1st and 15th)
BMS – business month start frequency
CBMS – custom business month start frequency
Q – quarter-end frequency
BQ – business quarter-end frequency
QS – quarter start frequency
BQS – business quarter start frequency
A – year-end frequency
BA – business year-end frequency
AS – year start frequency
BAS – business year start frequency
BH – business hour frequency
H – hourly frequency
T – minutely frequency
S – secondly frequency
L – milliseonds

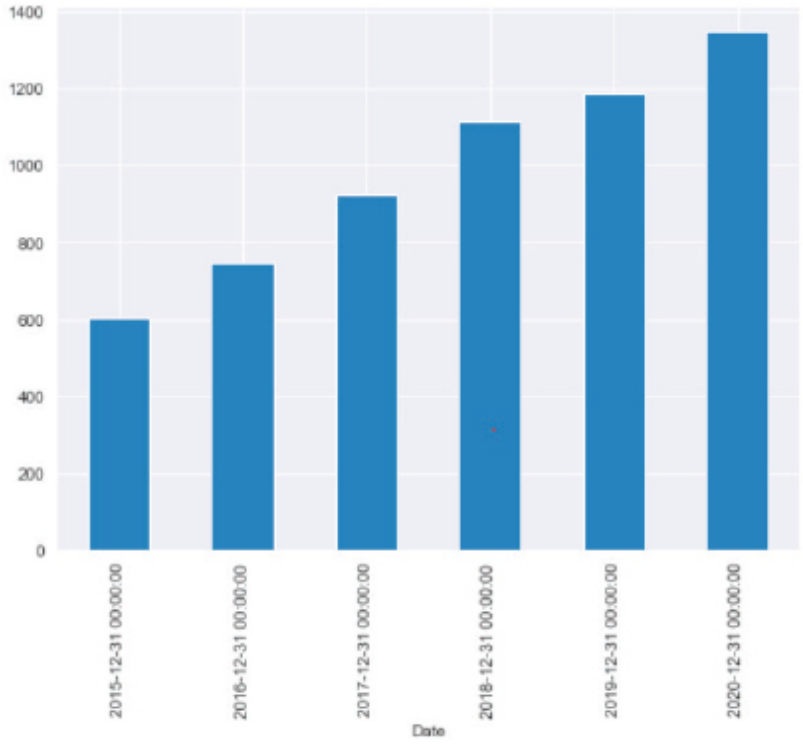
U – microseconds
N – nanoseconds

You can also append plot functions with the **resample()** function to plot the different types of plots based on aggregate values. For instance, the following script plots a bar plot for the opening stock price of Google over five years.

Script 11:

```
google_stock['Open'].resample('A').mean().plot(kind='bar', figsize=( 8 , 6 ))
```

Output:

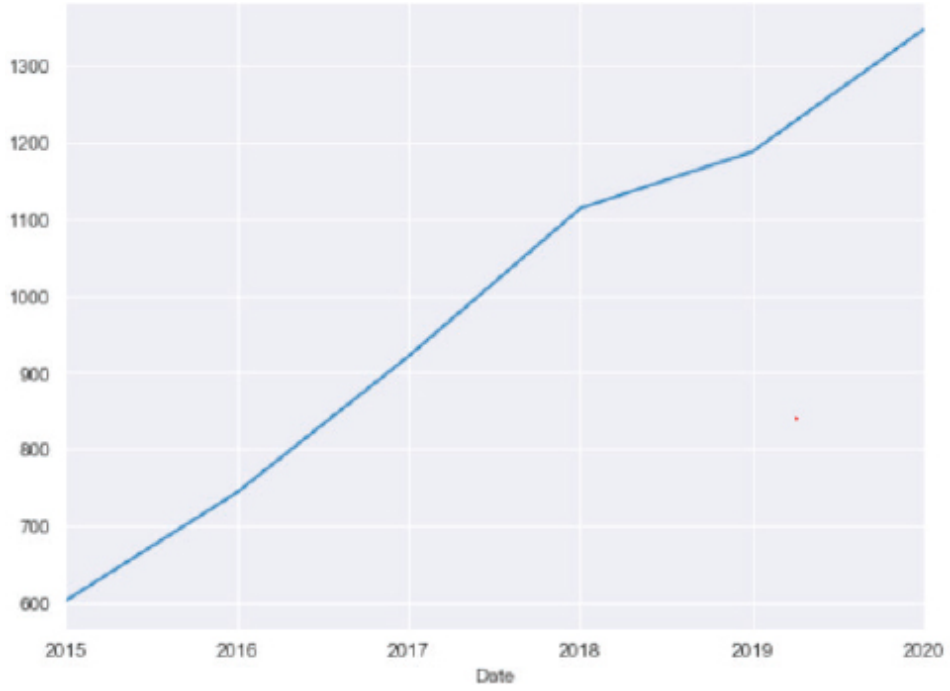


Similarly, here is the line plot for the yearly mean opening stock prices for Google stock over five years.

Script 12:

```
google_stock['Open'].resample('A').mean().plot(kind='line', figsize=( 8 , 6 ))
```

Output:



6.2.2. Time Shifting with Pandas

Time-shifting refers to shifting rows forward or backward. To shift rows forward, you can use the **shift()** function and pass it a positive value. For instance, the following script shifts three rows ahead and prints the header of the dataset.

Script 13:

```
google_stock.shift( 3 ).head()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume
2015-01-06	NaN	NaN	NaN	NaN	NaN	NaN
2015-01-07	NaN	NaN	NaN	NaN	NaN	NaN
2015-01-08	NaN	NaN	NaN	NaN	NaN	NaN
2015-01-09	513.589966	514.761719	499.678131	500.585632	500.585632	2899900.0
2015-01-12	505.611847	505.855164	498.281952	499.727997	499.727997	2065000.0

You can see that the first three rows now contain null values, while what previously was the first record has now been shifted to the 4th row.

In the same way, you can shift rows backward. To do so, you have to pass a negative value to the shift function.

Script 14:

```
google_stock.shift(- 3 ).tail()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume
2019-12-30	1347.859985	1372.5	1345.543945	1360.660034	1360.660034	1186400.0
2019-12-31	1350.000000	1396.5	1350.000000	1394.209961	1394.209961	1732300.0
2020-01-02	NaN	NaN	NaN	NaN	NaN	NaN
2020-01-03	NaN	NaN	NaN	NaN	NaN	NaN
2020-01-06	NaN	NaN	NaN	NaN	NaN	NaN

6.3. Rolling Window Functions

Rolling window functions are aggregate functions applied on a set of a specific number of records, which is called the window for a window function. For instance, with rolling window functions, you can find the average of values in a specific column for the previous two rows.

Let's see some examples of rolling window functions in Pandas. For this section, you will again be using the "google_data.csv," which you can find in the Data folder of the book resources. The following script imports this file:

Script 15:

```
google_stock = pd.read_csv(r"D:\Datasets\google_data.csv")

google_stock['Date'] = google_stock['Date'].apply(pd.to_ datetime)
google_stock.set_index('Date', inplace=True )

google_stock.head()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume
2015-01-06	513.589966	514.761719	499.678131	500.585632	500.585632	2899900
2015-01-07	505.611847	505.855164	498.281952	499.727997	499.727997	2065000
2015-01-08	496.626526	502.101471	489.655640	501.303680	501.303680	3353500
2015-01-09	503.377991	503.537537	493.435272	494.811493	494.811493	2071300
2015-01-12	493.584869	494.618011	486.225067	491.201416	491.201416	2326700

To apply rolling window functions in Pandas, you can use the rolling(). You need to pass the window size as a parameter value to the function. Finally, you need to concatenate the type of operation that you need to perform, e.g., mean(), median(), min(), max(), etc., with the rolling() function.

As an example, the script below finds the rolling average (mean) of the values in the *Volume* column of the previous two records in our Google stock price dataframe. Then, the rolling values are added in a new column named *Roll. Volume Avg* .

Script 16:

```
google_stock['Roll. Volumne Avg'] = google_stock['Volume'].rolling( 2 ).mean()
```

```
google_stock.head()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume	Roll. Volumne Avg
2015-01-06	513.589966	514.761719	499.678131	500.585632	500.585632	2899900	NaN
2015-01-07	505.611847	505.855164	498.281952	499.727997	499.727997	2065000	2482450.0
2015-01-08	496.626526	502.101471	489.655640	501.303680	501.303680	3353500	2709250.0
2015-01-09	503.377991	503.537537	493.435272	494.811493	494.811493	2071300	2712400.0
2015-01-12	493.584869	494.618011	486.225067	491.201416	491.201416	2326700	2199000.0

From the above output, you can see that the *Roll. Volume Avg* column contains the rolling average (mean) of the values in the *Volume* column of the previous two records. For instance, the average of values 289900 and 2065000 is 2482450.

Similarly, you can find the rolling sum of the previous two records using the sum() function with the rolling(2) function, as shown in the script below.

Script 17:

```
google_stock['Roll. Sum Avg'] = google_stock['Volume'].rolling( 2 ).sum()
google_stock.head()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume	Roll. Volumne Avg	Roll. Sum Avg
2015-01-06	513.589966	514.761719	499.678131	500.585632	500.585632	2899900	NaN	NaN
2015-01-07	505.611847	505.855164	498.281952	499.727997	499.727997	2065000	2482450.0	4964900.0
2015-01-08	496.626526	502.101471	489.655640	501.303680	501.303680	3353500	2709250.0	5418500.0
2015-01-09	503.377991	503.537537	493.435272	494.811493	494.811493	2071300	2712400.0	5424800.0
2015-01-12	493.584869	494.618011	486.225067	491.201416	491.201416	2326700	2199000.0	4398000.0

The min() function can be chained with the rolling() function to find the rolling minimum values for a range of values, as shown in the script below.

Script 18:

```
google_stock['Roll. Min Avg'] = google_stock['Volume'].
    rolling( 3 ).min()
google_stock.head()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume	Roll. Volumne Avg	Roll. Sum Avg	Roll. Min Avg
2015-01-06	513.589966	514.761719	499.678131	500.585632	500.585632	2899900	NaN	NaN	NaN
2015-01-07	505.611847	505.855164	498.281952	499.727997	499.727997	2065000	2482450.0	4964900.0	NaN
2015-01-08	496.626526	502.101471	489.655640	501.303680	501.303680	3353500	2709250.0	5418500.0	2065000.0
2015-01-09	503.377991	503.537537	493.435272	494.811493	494.811493	2071300	2712400.0	5424800.0	2065000.0
2015-01-12	493.584869	494.618011	486.225067	491.201416	491.201416	2326700	2199000.0	4398000.0	2071300.0

Finally, you can find the rolling standard deviation via the following script.

Script 19:

```
google_stock['Roll. std Avg'] = google_stock['Volume'].
rolling( 3 ).std() google_stock.head()
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume	Roll. Volumne Avg	Roll. Sum Avg	Roll. Min Avg	Roll. std Avg
2015-01-06	513.589966	514.761719	499.678131	500.585632	500.585632	2899900	NaN	NaN	NaN	NaN
2015-01-07	505.611847	505.855164	498.281952	499.727997	499.727997	2065000	2482450.0	4964900.0	NaN	NaN
2015-01-08	496.626526	502.101471	489.655640	501.303680	501.303680	3353500	2709250.0	5418500.0	2065000.0	653585.396104
2015-01-09	503.377991	503.537537	493.435272	494.811493	494.811493	2071300	2712400.0	5424800.0	2065000.0	742103.853918
2015-01-12	493.584869	494.618011	486.225067	491.201416	491.201416	2326700	2199000.0	4398000.0	2071300.0	678673.244893

Further Readings – Handling Time-series Data with Pandas

1. Check the official documentation (https://bit.ly/3C2d8us) to learn

more about time-series data handling with Pandas.

2. To study more about the Pandas time shifting functions for time-series data analysis, please check Pandas' official documentation for the shift() function (https://bit.ly/3riI8mq).

Hands-on Time – Exercises

Now, it is your turn. Follow the instructions in **the exercises below** to check your understanding of Pandas time-series handling techniques that you learned in this chapter. The answers to these questions are given at the end of the book.

Exercise 6.1

Question 1:

In a Pandas dataframe df, how would you add a column "B," which contains the rolling sum of the previous three rows in the column "C"?

A. df["B"] = df["C"].roll(3).sum()

B. df["B"] = df["C"].rolling(3).add()

C. df["B"] = df["C"].rolling(3).sum()

D. df["C"] = df["B"].rolling(3).sum()

Question 2:

To resample time-series data by year, you have to use the following rule in the resample() function:

A. "Year"

B. "Years"

C. "A"

D. "Annual"

Question 3

How to time-shift Pandas dataframe five rows back?

A. shift_back(5)
B. shift(5)
C. shift_behind(-5)
D. shift(-5)

Exercise 6.2

Using the Pandas dataframe, read the “titanic_data.csv” file from the data folder. Convert the “date” column to the date type column and then set this column as the index column.

Add a column in the dataframe which displays the maximum value from the “Open” column for the last 5 days.

Appendix:

Working with Jupyter Notebook

All the scripts in this book are executed via Jupyter Notebook that comes with Anaconda or via the Colab Environment, which contains a Jupyter Notebook-like interface of its own. Therefore, it makes sense to give a brief overview of Jupyter Notebook.

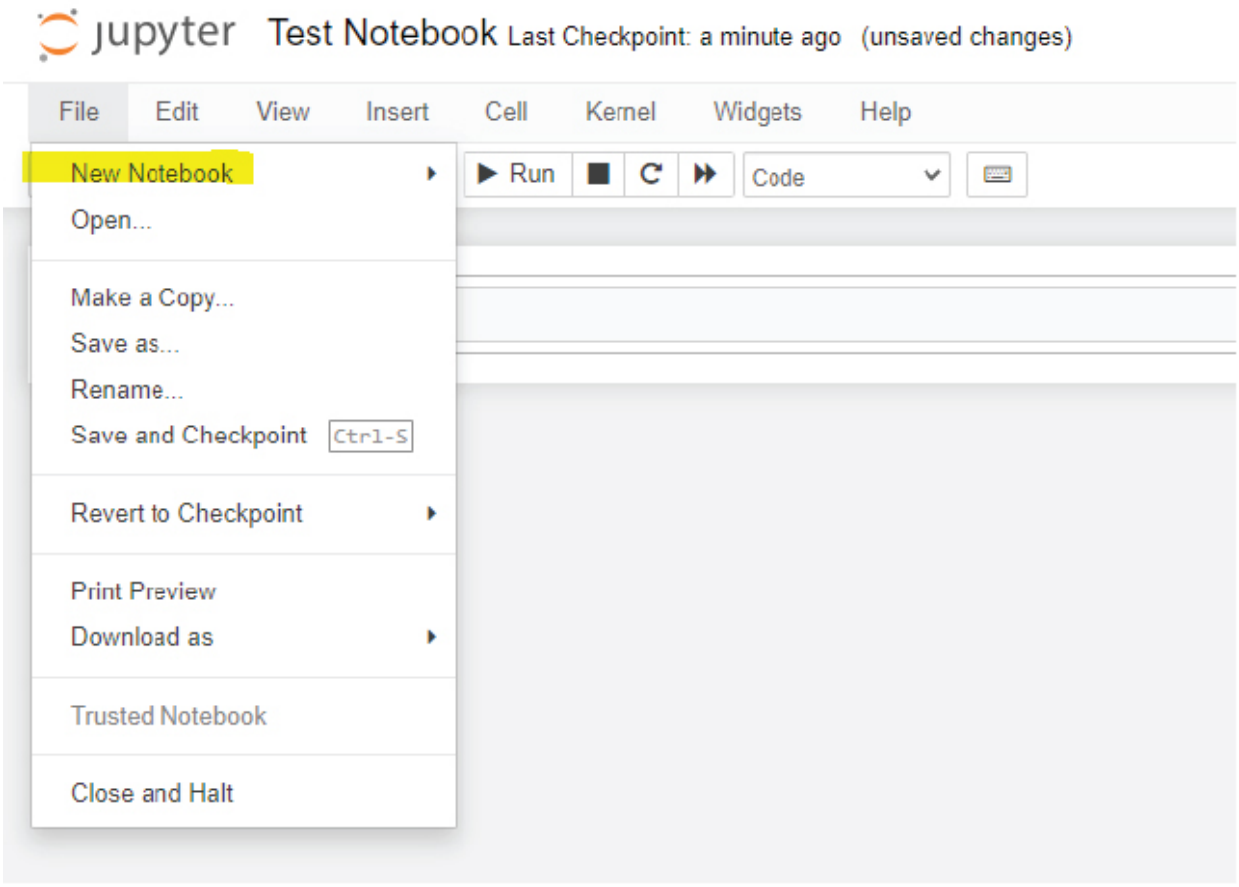
In chapter 1, you have already seen how to run a very basic script with the Jupyter Notebook from Anaconda. In this section, you will see some other common functionalities of the Jupyter Notebook.

Creating a New Notebook

There are two main ways to create a new Python Jupyter notebook.

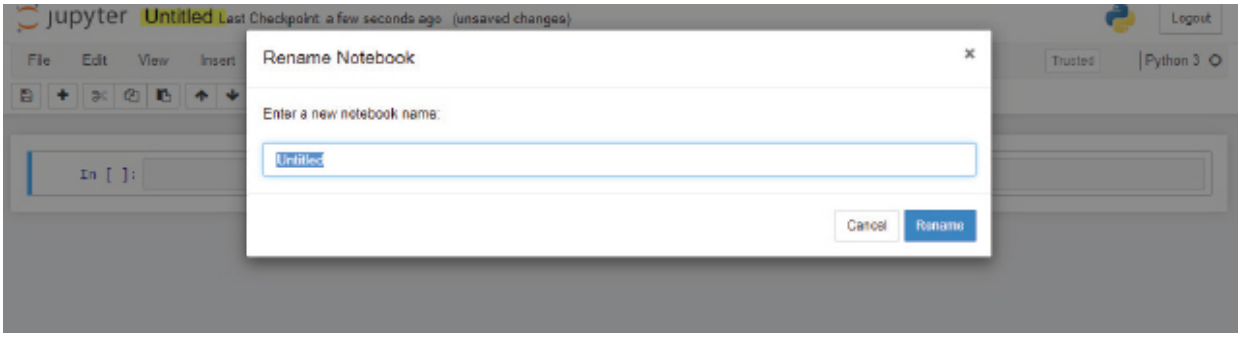
1. From the home page of the Jupyter notebook, click the **New** button at the top right corner and then select *Python 3* , as shown in the following screenshot.

2. If you have already opened a Jupyter notebook, and you want to create a new Jupyter notebook, select “File - > New Notebook” from the top menu. Here is a screenshot of how to do this.


jupyter
Test Notebook
Last Checkpoint: a minute ago
(unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
New Notebook
Open...
Make a Copy...
Save as...
Rename...
Save and Checkpoint
Ctrl-S
Revert to Checkpoint
Print Preview
Download as
Trusted Notebook
Close and Halt
Run
Code

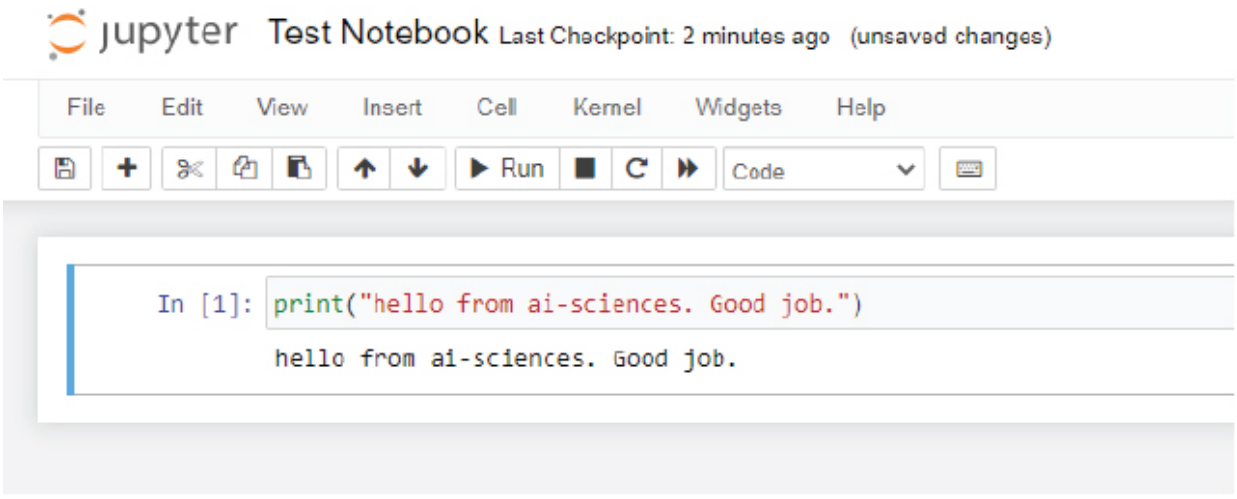
Renaming a Notebook

To rename a new Jupyter notebook, click on the Jupyter notebook name from the top left corner of your notebook. A dialog box will appear containing the old name of your Jupyter notebook, as shown below (the default name for a Jupyter notebook will be "Untitled"). Here, you can enter a new name for your Jupyter notebook.



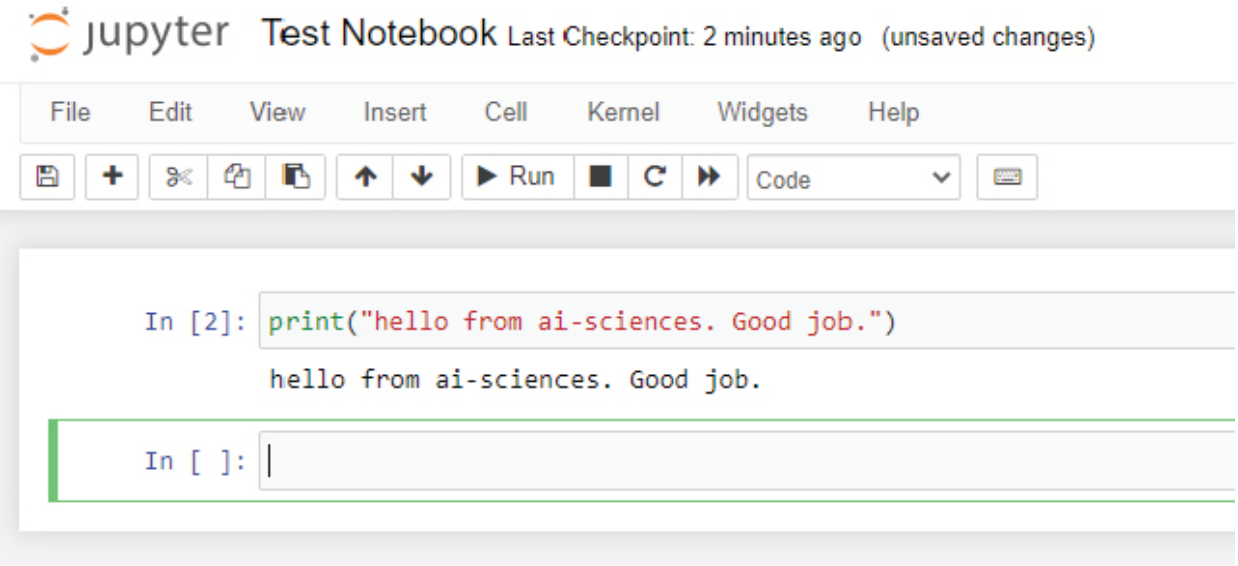
Running Script in a Cell

To run a script inside a cell, select the cell and then press "CTRL + Enter" from your keyboard. Your script will execute. However, a new cell will not be created automatically once the current cell executes.

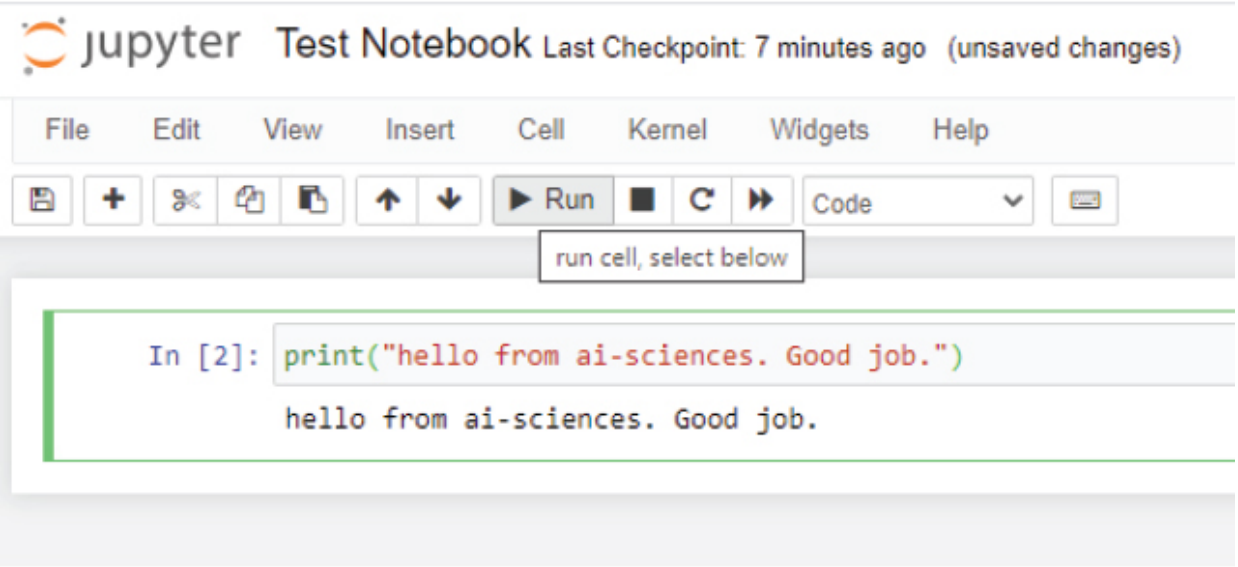


If you want to automatically create a new cell with the execution of a cell, you can select a cell and execute it using "SHIFT + ENTER" or "ALT + ENTER".

You will see that the current cell is executed, and a new cell is automatically created, as shown below:

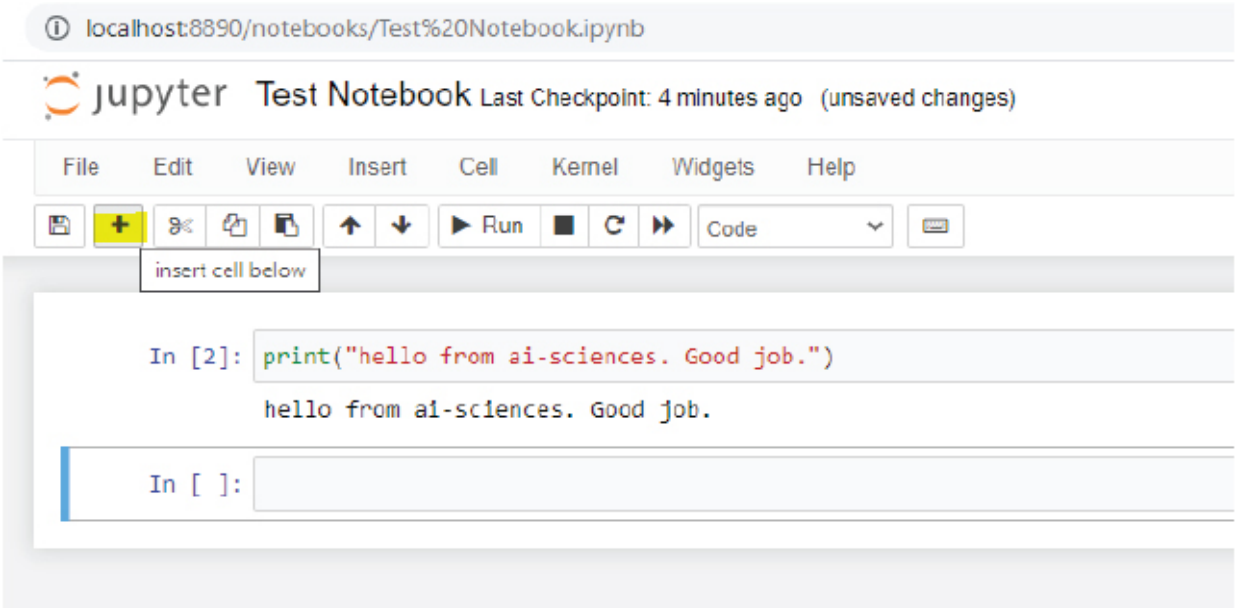


You can also run a cell by selecting a cell and then by clicking the **Run** option from the top menu, as shown below.

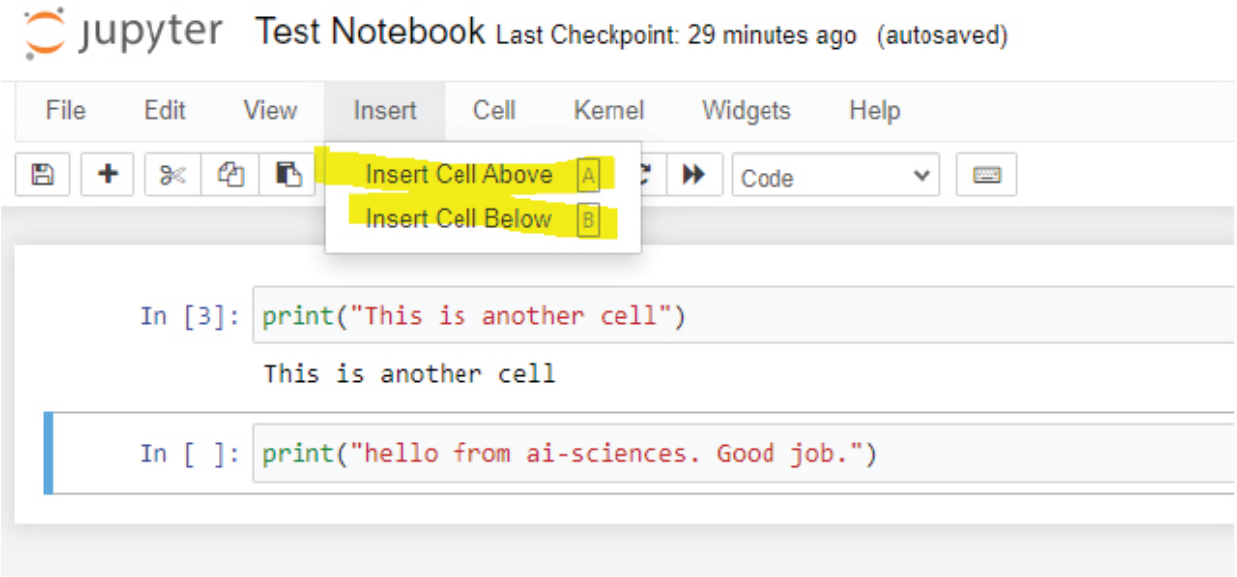


Adding a New Cell

The plus "+" symbol from the top menu allows you to add a new cell below the currently selected cell. Here is an example.

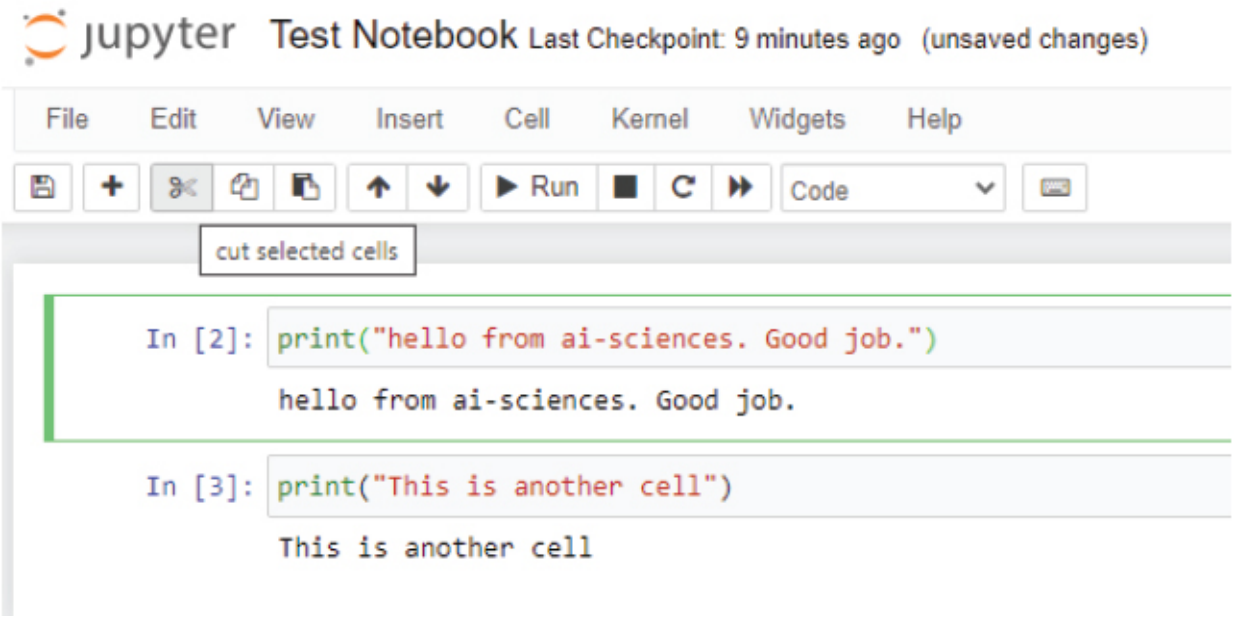


You can also insert cells above or below any selected cells via the *Insert Cell Above* and *Insert Cell Below* commands from the *Insert* option in the top menu, as shown below:



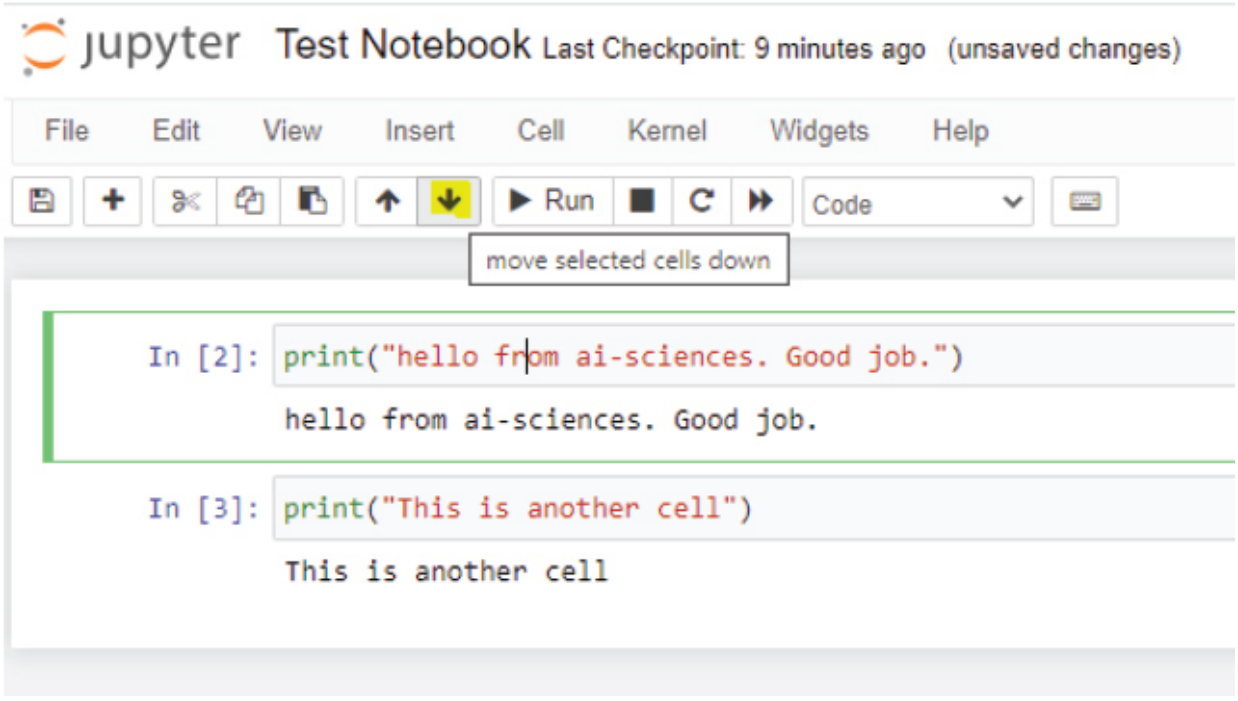
Deleting a New Cell

To delete a cell, simply select the cell and then click the scissor icon from the top menu of your Jupyter notebook, as shown in the following screenshot.

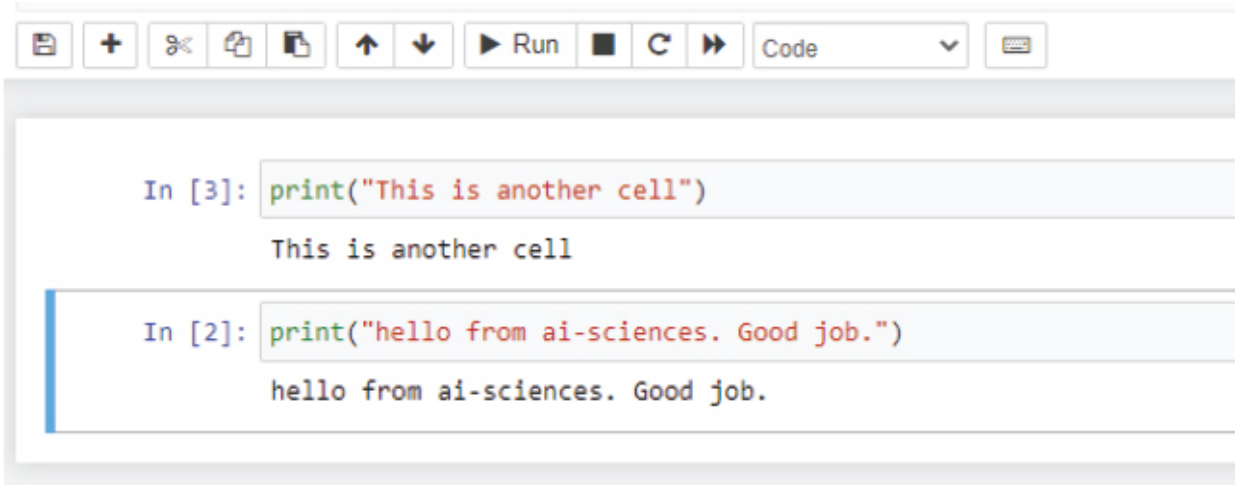


Moving Cells Up and Down

To move a cell down, select a cell and then click the downward arrow. Here is an example. Here, cell number 2 is moved one position below.



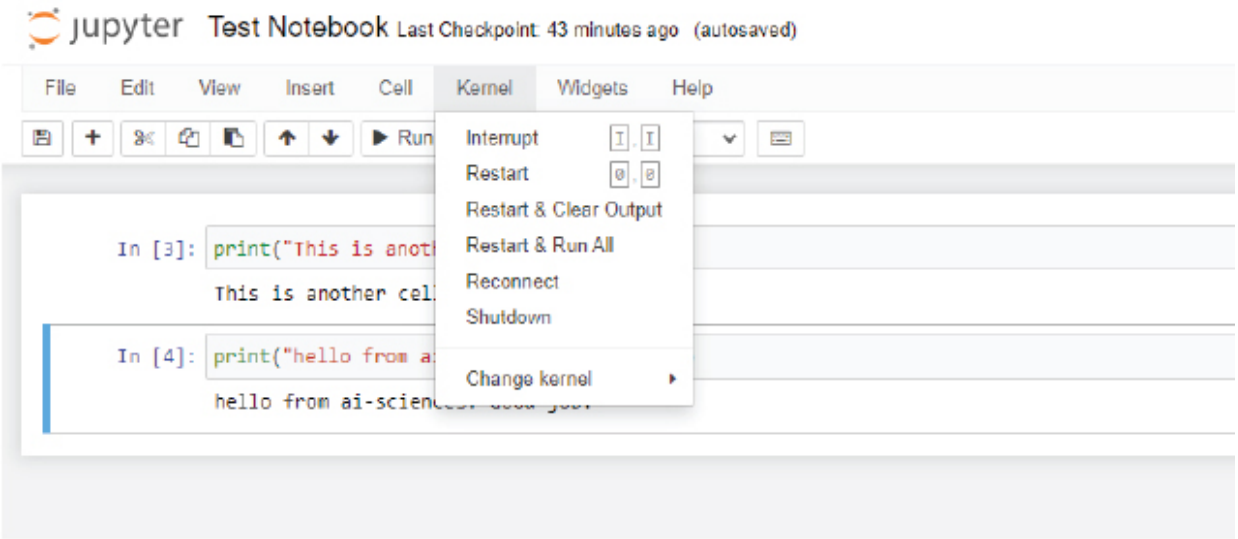
In the output, you can see that cell 2 is now below cell 3.



In the same way, you can click on the upward arrow to move a cell up.

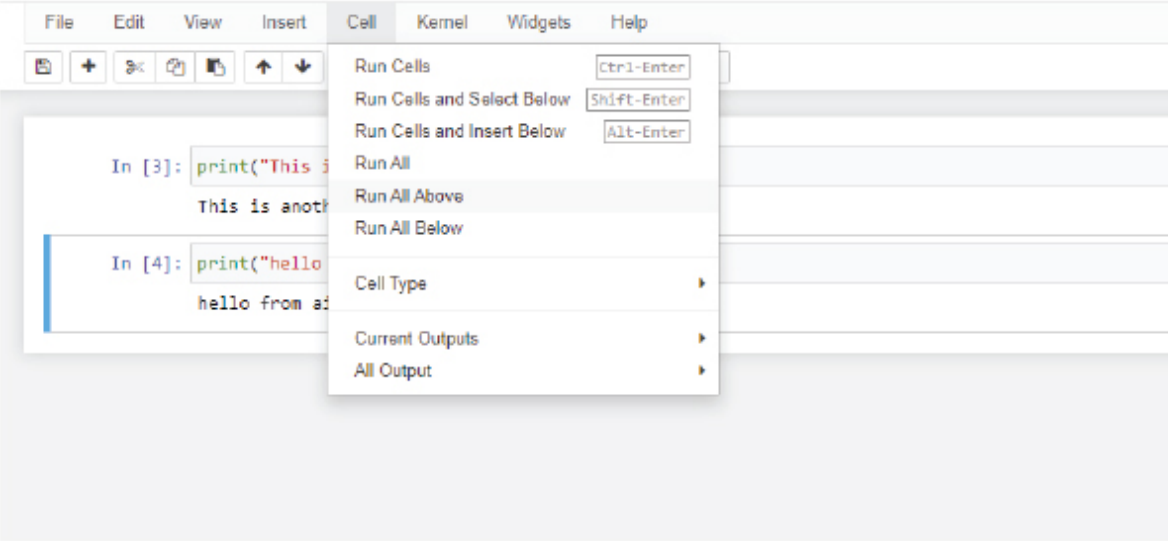
Miscellaneous Kernel Options

To see miscellaneous kernel options, click the *Kernel* button from the top menu. A dropdown list will appear, where you can see options related to interrupting, restarting, reconnecting, and shutting down the kernel used to run your script. Clicking "Restart & Run" will restart the kernel and run all the cells once again.



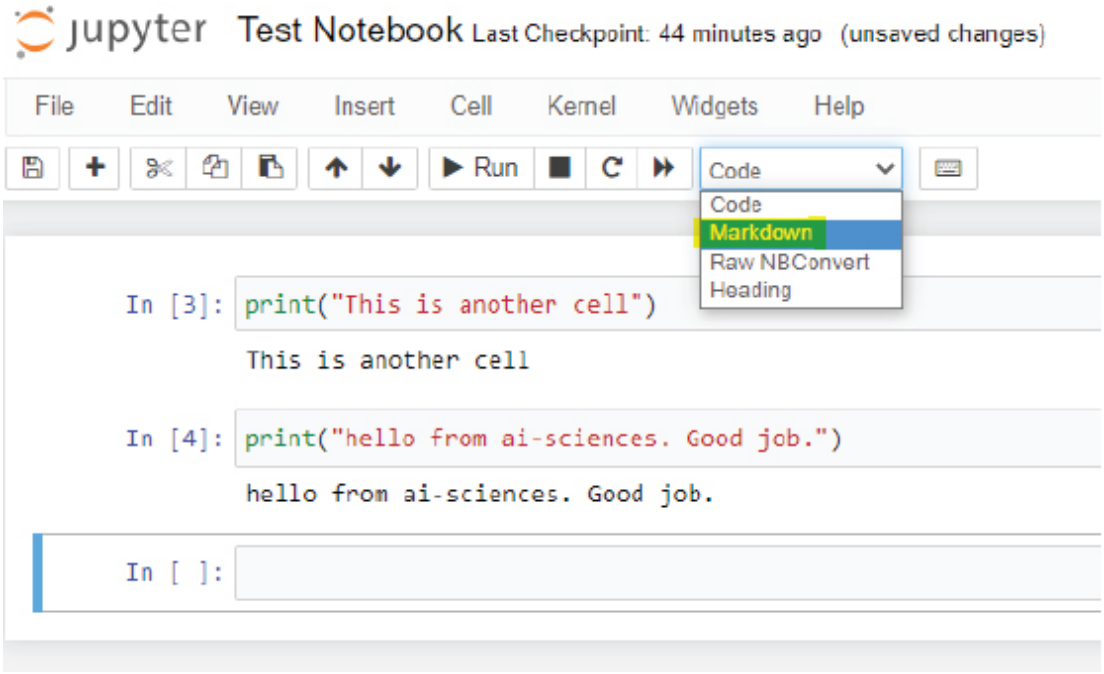
Miscellaneous Cell Options

Clicking the Cell button from the top menu reveals the location of options related to running a particular cell in a Jupyter notebook. Look at the following screenshot for reference.

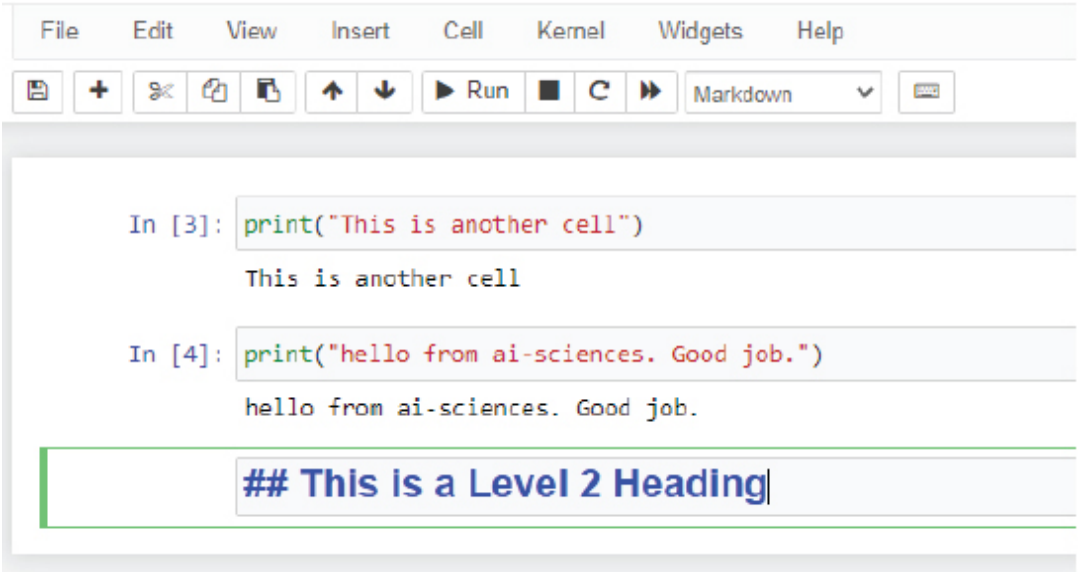


Writing Markdown in Jupyter Notebook

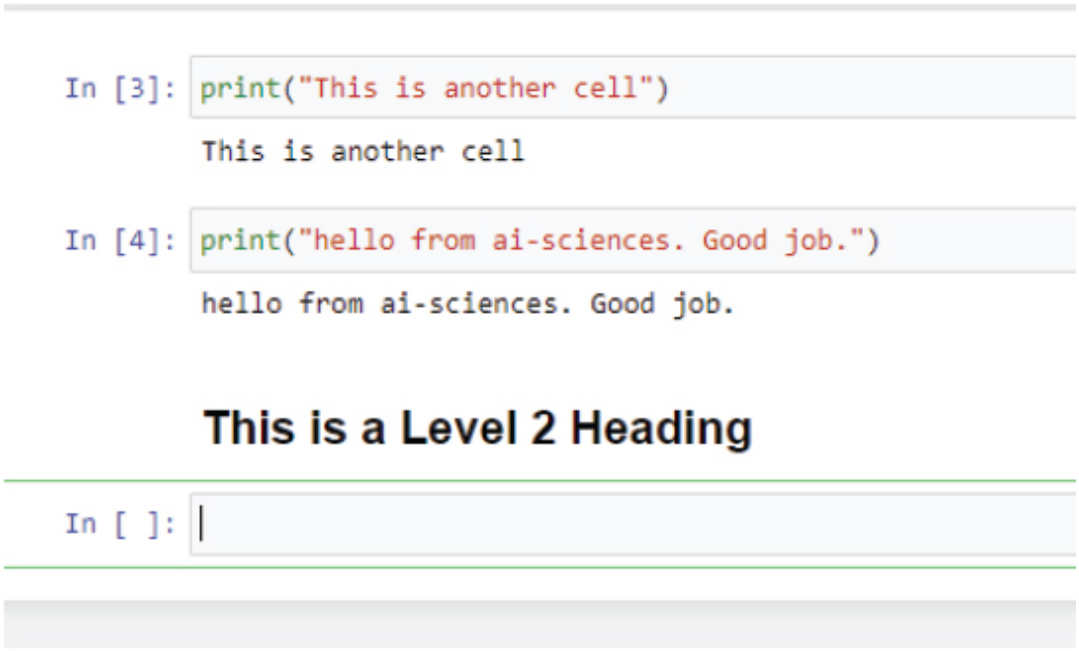
Apart from writing Python scripts, you can also write markdown content in the Jupyter notebook. To do so, you have to first select a cell where you want to add your markdown content, and then you have to select the *Markdown* option from the dropdown list, as shown in the following script.



Next, you need to enter the markdown content in the markdown cell. For instance, in the following screenshot, the markdown content h2 level heading is added to the third cell.

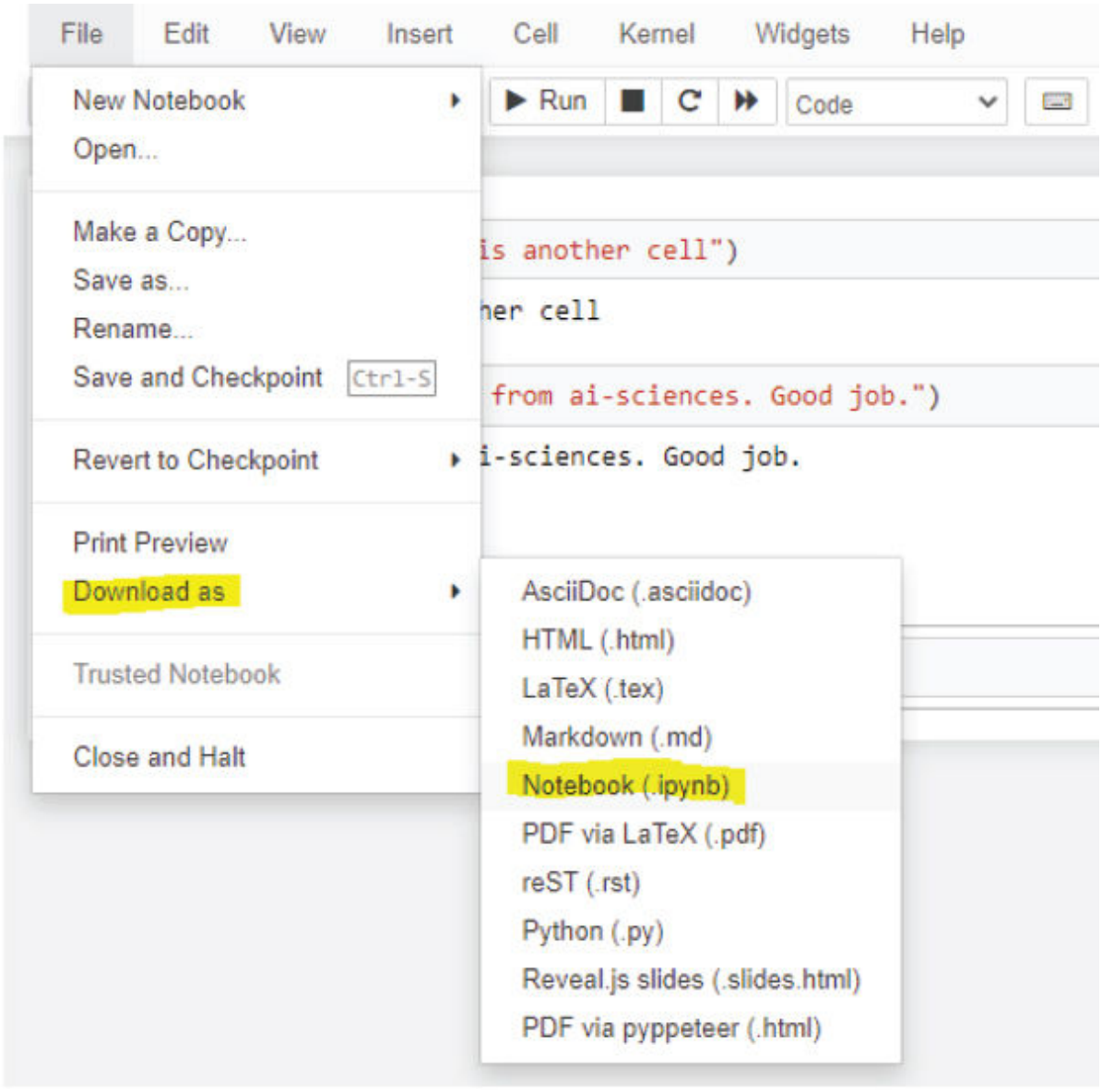


When you run the 3rd cell in the above script, you will see the compiled markdown content, as shown below:



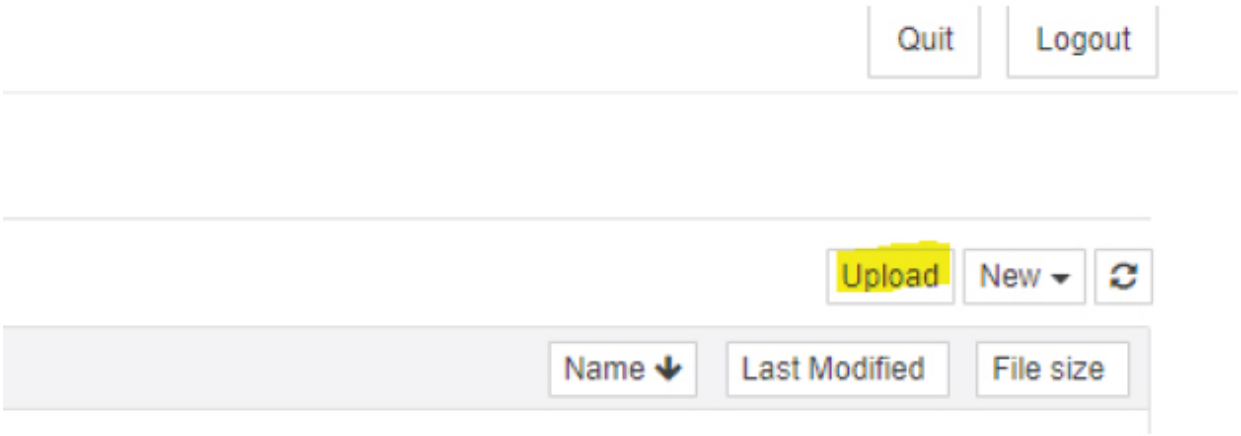
Downloading Jupyter Notebooks

To download a Jupyter notebook, click the "File -> Download as" option from the top menu. You can download Jupyter notebooks in various formats, e.g., HTML, PDF, Python Notebook (ipynb), etc.


File
Edit
View
Insert
Cell
Kernel
Widgets
Help
New Notebook
Open...
Make a Copy...
Save as...
Rename...
Save and Checkpoint
Ctrl-S
Revert to Checkpoint
Print Preview
Download as
Trusted Notebook
Close and Halt
Run
Code
is another cell")
her cell
from ai-sciences. Good job.")
i-sciences. Good job.
AsciiDoc (.asciidoc)
HTML (.html)
LaTeX (.tex)
Markdown (.md)
Notebook (.ipynb)
PDF via LaTeX (.pdf)
reST (.rst)
Python (.py)
Reveal.js slides (.slides.html)
PDF via pyppeteer (.html)

Uploading an Existing Notebook

Similarly, you can upload an existing notebook to Anaconda Jupyter. To do so, you need to click the *Upload* button from the main Jupyter dashboard. Look at the screenshot below for reference.



Exercise Solutions

Exercise 2.1

Question 1:

What is the major disadvantage of mean and median imputation?

A. Distorts the data distribution
B. Distorts the data variance
C. Distorts the data covariance
D. All of the Above

Answer: D

Question 2:

How to display the last three rows of a Pandas dataframe named “my_df”?

A. my_df.end(3)
B. my_df.bottom(3)
C. my_df.top(-3)
D. my_df.tail(3)

Answer: D

Question 3:

You can create a Pandas series using a:

A. NumPy Array
B. List
C. Dictionary

D. All of the Above

Answer: C

Exercise 2.2

Replace the missing values in the “deck” column of the Titanic dataset with the most frequently occurring categories in that column. Plot a bar plot for the updated “deck” column. The Titanic dataset can be downloaded using this Seaborn command:

```
import seaborn as sns

sns.load_dataset('titanic')
```

Solution:

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = [ 8 , 6 ]
sns.set_style("darkgrid")

titanic_data = sns.load_dataset('titanic')

titanic_data =titanic_data[["deck"]]
titanic_data.head()
titanic_data.isnull().mean()

titanic_data.deck.value_counts().sort_values(ascending=False).plot.bar()

plt.xlabel('deck')
plt.ylabel('Number of Passengers')

titanic_data.deck.mode()

titanic_data.deck.fillna('C', inplace=True)

titanic_data.deck.value_counts().sort_values(ascending=False).plot.bar()
plt.xlabel('deck')
plt.ylabel('Number of Passengers')
```

Exercise 3.1

Question 1:

Which function is used to sort Pandas dataframe by a column value?

A. sort_dataframe()
B. sort_rows()
C. sort_values()
D. sort_records()

Answer: C

Question 2:

To filter columns from a Pandas dataframe, you have to pass a list of column names to one of the following method:

A. filter()
B. filter_columns()
C. apply_filter ()
D. None of the above()

Answer: A

Question 3:

To drop the second and fourth rows from a Pandas dataframe named my_df, you can use the following script:

A. my_df.drop([2,4])
B. my_df.drop([1,3])
C. my_df.delete([2,4])
D. my_df.delete([1,3])

Answer: B

Exercise 3.2

From the Titanic dataset, filter all the records where the fare is greater than 20 and the passenger traveled alone. You can access the Titanic dataset using the following Seaborn command:

```
import seaborn as sns

titanic_data = sns.load_dataset('titanic')
```

Solution:

```
import seaborn as sns

titanic_data = sns.load_dataset('titanic')

my_df = titanic_data[(titanic_data["fare"] > 50 ) & (titanic_data["alone"] == True )]
my_df.head()
```

Exercise 4.1

Question 1:

To horizontally concatenate two Pandas (pd) dataframes A and B, you can use the following function:

A. pd.concat([A, B], ignore_index = True)

B. pd.concat([A, B], axis = 1, ignore_index = True)

C. pd.append([A, B] ignore_index = True)

D. pd.join([A, B], axis = 1, ignore_index = True)

Answer: B

Question 2:

To find the number of unique values in column X of Pandas dataframe df, you can use the groupby clause as follows:

A. df.groupby(«X»).nunique

B. df.groupby(«X»).unique

C. df.groupby(“X”).ngroups

D. df.groupby(“X”).nvalues

Answer: C

Question 3:

To remove all the duplicate rows from a Pandas dataframe df, you can use the following function:

A. df.drop_duplicates(keep=False)

B. df.drop_duplicates(keep=’None’)

C. df.drop_duplicates(keep=’last’)

D. df.drop_duplicates()

Answer: A

Exercise 4.2

From the Titanic dataset, find the minimum, maximum, median, and mean values for ages and fare paid by passengers of different genders. You can access the Titanic dataset using the following Seaborn command:

```
import seaborn as sns

titanic_data = sns.load_dataset('titanic')
```

Solution:

Finding max, min, count, median, and mean for “sex” column.

```
titanic_gbsex= titanic_data.groupby("sex")

titanic_gbsex.age.agg(['max', 'min', 'count', 'median', 'mean'])
```

Finding max, min, count, median, and mean for the “fare” column.

```
titanic_gbsex.fare.agg(['max', 'min', 'count', 'median', 'mean'])
```

Exercise 5.1

Question 1:

Which attribute is used to change the color of a Pandas graph?

A. set_color()
B. define_color()
C. color()
D. None of the above

Answer: C

Question 2:

Which Pandas function is used to plot a horizontal bar plot?

A. horz_bar()
B. barh()
C. bar_horizontal()
D. horizontal_bar()

Answer: B

Question 3:

Which attribute is used to define the number of bins in a Pandas histogram plot?

A. n_bins
B. bins
C. number_bins
D. None of the above

Answer: B

Exercise 5.2

Display a bar plot using the Titanic dataset that displays the average age of the passengers who survived vs. those who did not survive.

```
import seaborn as sns
import pandas as pd

sns.set_style("darkgrid")
sns.set_context("poster")

titanic_data = pd.read_csv(r"D:Datasets\titanic_data.csv")
titanic_data.head()
surv_mean = titanic_data.groupby("Survived")["Age"].mean()

df = pd.DataFrame({'Survived':['No', 'Yes'], 'Age':surv_mean. tolist()})
ax = df.plot.bar(x='Survived', y='Age', figsize=( 8 , 6 ))
```

Exercise 6.1

Question 1:

In a Pandas dataframe df, how would you add a column “B,” which contains the rolling sum of the previous three rows in the column “C”?

A. df[“B”] = df[“C”].roll(3).sum()

B. df[“B”] = df[“C”].rolling(3).add()

C. df[“B”] = df[“C”].rolling(3).sum()

D. df[“C”] = df[“B”].rolling(3).sum()

Answer: C

Question 2:

To resample time-series data by year, you have to use the following rule in the resample() function:

A. “Year”

B. “Years”

C. “A”

D. “Annual”

Answer: C

Question 3

How to time shift Pandas dataframe five rows back?

A. shift_back(5)

B. shift(5)

C. shift_behind(-5)

D. shift(-5)

Answer: D

Exercise 6.2

Using the Pandas dataframe, read the “titanic_data.csv” file from the data folder. Convert the “date” column to the date type column and then set this column as the index column.

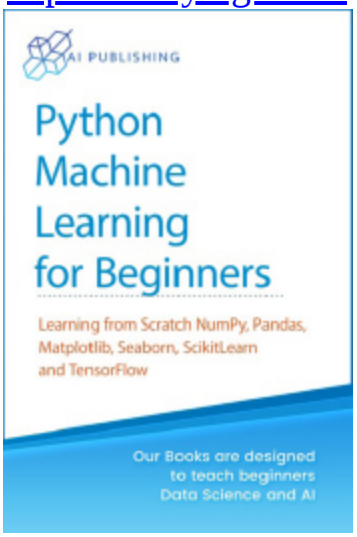
Add a column in the dataframe which displays the maximum value from the “Open” column for the last 5 days.

Solution:

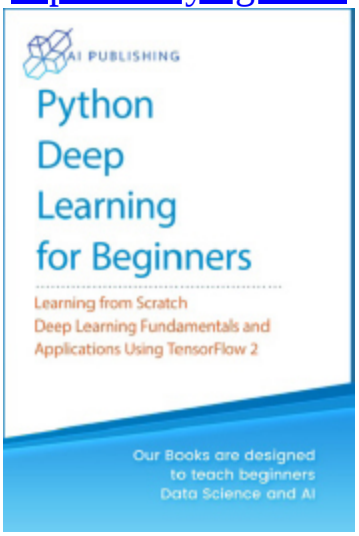
```
google_stock = pd.read_csv(r"D:\Datasets\google_data.csv")
google_stock['Date'] = google_stock['Date'].apply(pd.to_ datetime)
google_stock.set_index('Date', inplace=True)

google_stock['Roll. Max Avg'] = google_stock['Open'].rolling( 5 ).max()
google_stock.head( 15 )
```

From the Same Publisher

Python Machine Learning
https://bit.ly/3gcb2iG

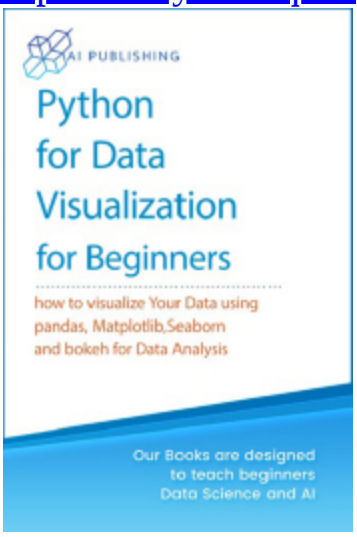


Python Deep Learning
https://bit.ly/3gci9Ys



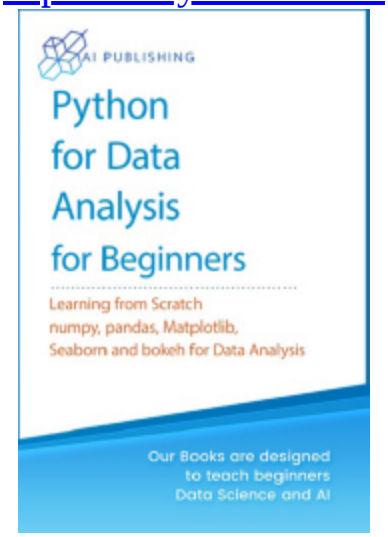
Python Data Visualization

https://bit.ly/3wXqDJI



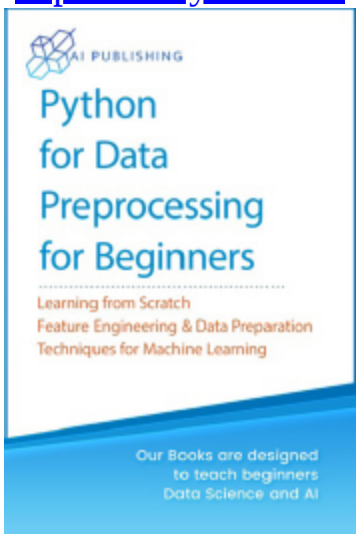
Python for Data Analysis

https://bit.ly/3wPYEM2



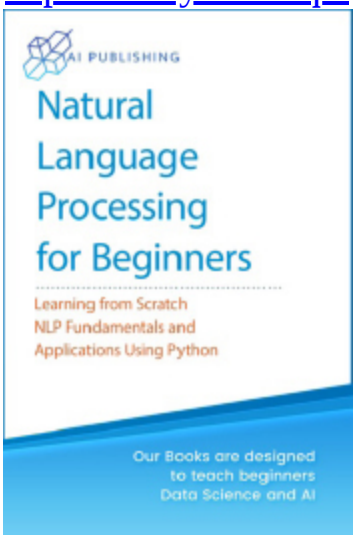
Python Data Preprocessing

https://bit.ly/3fLV3ci



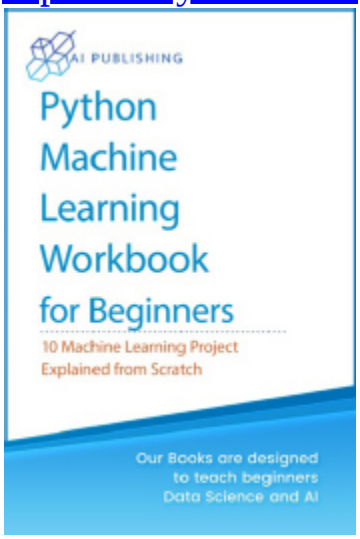
Python for NLP

https://bit.ly/3chlTqm



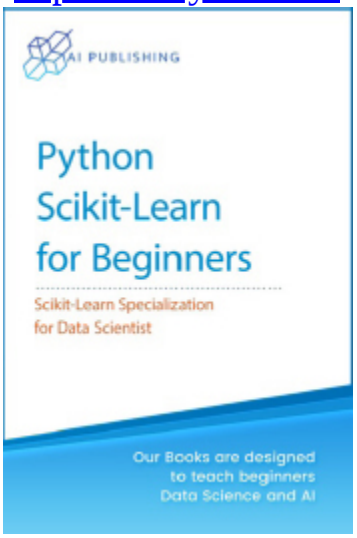
10 ML Projects Explained from Scratch

https://bit.ly/34KFsDk



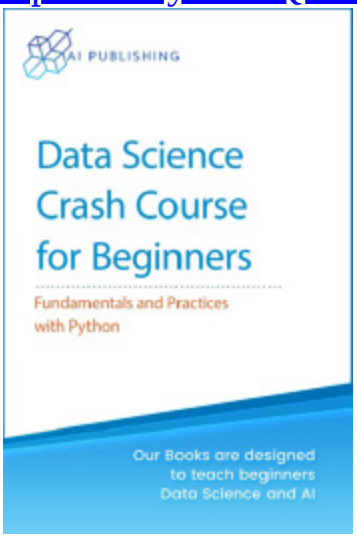
Python Scikit-Learn for Beginners

https://bit.ly/3fPbtRf



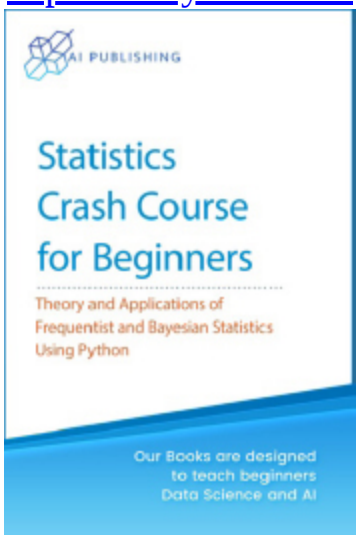
Data Science with Python

https://bit.ly/3wVQ5iN



Statistics with Python

https://bit.ly/3z27KHt



How to Contact Us

If you have any feedback, please let us know by sending an email to contact@aipublishing.io.

Your feedback is immensely valued, and we look forward to hearing from you. It will be beneficial for us to improve the quality of our books.