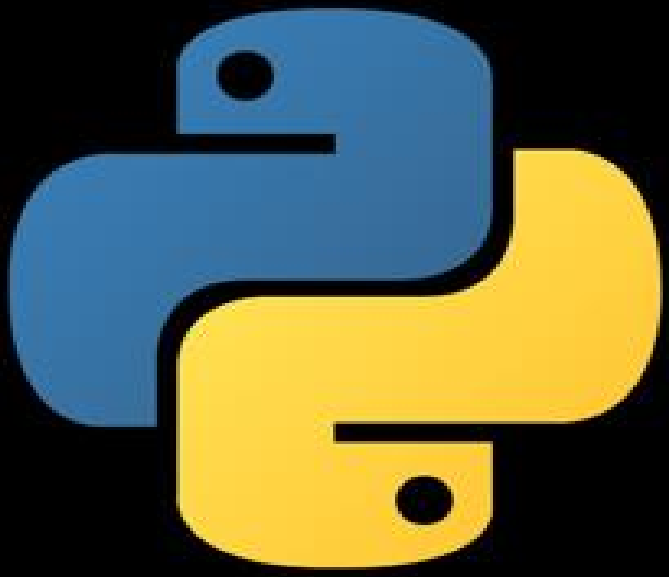
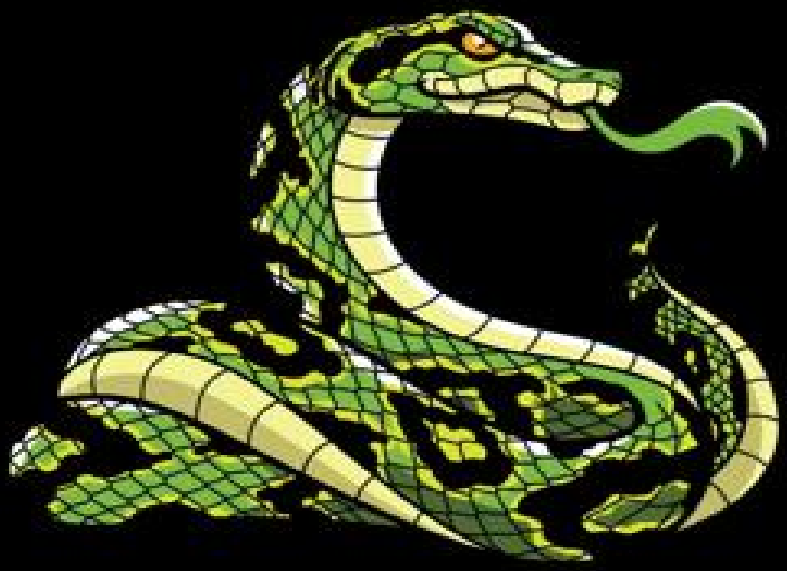
PYTHON PROGRAMMING

FOR BEGINNERS CRASH COURSE WITH HANDS-ON EXERCISES

INCLUDING NUMPY, PANDAS AND MATPLOTLIB





IBNUL JAIF FARABI

TABLE OF CONTENTS

INTRODUCTION TO PYTHON

What is Python programming language?

Python is a high-level programming language that uses a pre-defined set of instructions to teach a computer to perform certain tasks. Python is a free and open-source programming language. It was developed by Guido van Rossum in 1991. Anyone can read, develop, modify and distribute the code of the Python scripts.

In this book, we're going to use Python 3, which is the latest version of Python.

We're going to use JupyterLab for the Python programming.

We can install JupyterLab by running the following command in the terminal.

pip install jupyterlab

```
PowerShell 7.3.0
PS C:\Users\ibnul> pip install jupyterlab
```

Now, we're going to write our first program. We want to display "*Hello World*" as the output. For this, we need to use the print() statement. *print()* is a built-in function that can be used to display stuff on the output screen. And whatever we would like to display, needs to be inside the parenthesis and enclosed within the double quotes " ".

Code:

print("Hello World")

Output:

Hello World

```
print("Hello World")
```

```
Hello World
```

Yay! We've just written our first Python program.

Now, let's talk about the comments in Python. Comments are small pieces of code which will be ignored by the Python interpreter. It makes the code easier to understand. Anything written after the # symbol is considered a comment in Python. Now, we'll add a comment on the above program.

Code:

#This is our first Hello World program.

print("Hello World")

Output:

Hello World

```
#This is our first Hello World program.
print("Hello World")
```

```
Hello World
```

VARIABLES IN PYTHON

In python, the variables are used as containers to store the values. The values can be numbers, or text. Variables are identified by their variable names.

variable_01 = 752

Here, "*variable_01*" is the name of the variable, "=" is the assignment operator, "752" is the value that needs to be assigned to the variable. So, variable_01 has a value of 752.

Types of variables:

String Types: When the variable contains texts or words in general, it's called the String variable. A String is a sequence of characters and are enclosed within the double quotes " " or single quotes ' '.

my_city_01 = "New York"

Here, *my_city_01* is a string with the value "*New York*".

my_city_02 = "64532745"

Here, *my_city_02* is a string with the value "*64532745*". As "64532745" is written inside the double quotes " " it's considered a string.

Numeric Types: There're two types of numeric data types in Python; *Integer* and *Float*. Here, the value of the variable will always be a number.

If the number doesn't have a decimal value, it's an Integer. 5, 65, 124 etc. are Integers.

number_of_planets = 9

If the number has a decimal value, it's Float. 1.25, 243.96 34.26 etc. are Float numbers.

human_body_temperature = 98.4

Note that Python automatically identifies the type from the values, so we don’t need to specify the type manually in the program.

Boolean Types: A variable where the value can be either True or False, is called the Boolean type of variable.

variable_boolean_01 = True

variable_boolean_02 = False

Code:

```
#String type of variable
my_city_01 = "New York"
my_city_02 = "64532745"

print(my_city_01)
print(my_city_02)

#Numeric type of variable
#Integer
number_of_planets = 9
print(number_of_planets)

#Float
human_body_temperature = 98.4
print(human_body_temperature)

#Boolean type of variable
variable_boolean_01 = True
variable_boolean_02 = False
```

```
print(variable_boolean_01)
print(variable_boolean_02)
```

Output:

```
New York
64532745
9
98.4
True
False
```

```python
#String type of variable
my_city_01 = "New York"
my_city_02 = "64532745"

print(my_city_01)
print(my_city_02)

#Numeric type of variable
#Integer
number_of_planets = 9
print(number_of_planets)

#Float
human_body_temperature = 98.4
print(human_body_temperature)

#Boolean type of variable
variable_boolean_01 = True
variable_boolean_02 = False

print(variable_boolean_01)
print(variable_boolean_02)
```

```
New York
64532745
9
98.4
True
False
```

HOW TO TAKE THE USER INPUT IN PYTHON

To get the input from the user in Python, we need to use the *input()* function. We can also add more information inside the parenthesis.

location_name = input(“Where are you from: ”)

Code:

location_name = input("Where are you from: ")

print(location_name)

Output:

Italy


```
location_name = input("Where are you from: ")
print(location_name)
```

```
Where are you from:  Italy
Italy
```

After inputting the answer in the box, the output is displayed. Here, the user input is stored in the variable.

ARITHMETIC OPERATIONS IN PYTHON

In this part, we'll learn the basic arithmetic operations in Python.

Some basic arithmetic operations are the following.

- Addition
- Subtraction
- Multiplication
- Division
- Remainder
- Exponentiation

Addition:

It's the sum of the variables.

Code:

```
#Addition
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

addition = number_01 + number_02

print(addition)
```

Output:

```
Enter Number 01:  20
Enter Number 02:  70
90
```

```python
#Addition
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

addition = number_01 + number_02

print(addition)
```

```
Enter Number 01:  20
Enter Number 02:  70
90
```

Subtraction:

It's the subtraction of the variables.

Code:

```
#Subtraction
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

subtraction = number_01 - number_02

print(subtraction)
```

Output:

```
Enter Number 01:  98
Enter Number 02:  64
34
```

```python
#Subtraction
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

subtraction = number_01 - number_02

print(subtraction)
```

```
Enter Number 01:  98
Enter Number 02:  64
34
```

Multiplication:

It's the multiplication of the variables.

Code:

```
#Multiplication
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

multiplication = number_01 * number_02

print(multiplication)
```

Output:

```
Enter Number 01:  4
Enter Number 02:  8
32
```

```python
#Multiplication
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

multiplication = number_01 * number_02

print(multiplication)
```

```
Enter Number 01:  4
Enter Number 02:  8
32
```

Division:

It's the division of one variable by another variable.

Code:

```
#Division
number_01 = float(input("Enter Number 01: "))
number_02 = float(input("Enter Number 02: "))

division = number_01 / number_02

print(division)
```

Output:

```
Enter Number 01:  54.27
Enter Number 02:  12.65
4.290118577075099
```

```python
#Division
number_01 = float(input("Enter Number 01: "))
number_02 = float(input("Enter Number 02: "))

division = number_01 / number_02

print(division)
```

```
Enter Number 01:  54.27
Enter Number 02:  12.65
4.290118577075099
```

Remainder:

It's the remainder after we divide one variable by another variable.

Code:

```
#Remainder
number = int(input("Enter Number: "))

remainder = number % 4

print(remainder)
```

Output:

```
Enter Number:  10
2
```

```
#Remainder
number = int(input("Enter Number: "))

remainder = number % 4

print(remainder)
```

```
Enter Number:  10
2
```

Exponentiation:

It's the exponentiation of a variable.

Code:

```
#Exponentiation
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

exponentiation = number_01 ** number_02

print(exponentiation)
```

Output:

```
Enter Number 01:  4
Enter Number 02:  2
16
```

```python
#Exponentiation
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

exponentiation = number_01 ** number_02

print(exponentiation)
```

```
Enter Number 01:  4
Enter Number 02:  2
16
```

Here, 4 to the power of 2, where the answer is obviously 16.

COMPARISON AND LOGICAL OPERATORS IN PYTHON

The comparison operators will compare the values on either side of the operand and will determine the relations between them.

Some of the comparison operators are the following.

- **==**
- **!=**
- **<**
- **<=**
- **>**
- **>=**

Comparison operators return the values in *True* or *False*.

Code:

```
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

number_01 == number_02
```

Output:

```
Enter Number 01:  78
Enter Number 02:  78
True
```

```
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

number_01 == number_02
```

```
Enter Number 01:  78
Enter Number 02:  78

True
```

Here, we're checking the condition if *number_01* is equal to *number_02*. Since *number_01* is indeed equal to *number_02*, the output is returned as True.

Code:

number_01 = int(input("Enter Number 01: "))

number_02 = int(input("Enter Number 02: "))

number_01 > number_02

Output:

Enter Number 01: 56

Enter Number 02: 88

False

```
number_01 = int(input("Enter Number 01: "))
number_02 = int(input("Enter Number 02: "))

number_01 > number_02
```

```
Enter Number 01:  56
Enter Number 02:  88

False
```

Here, we're checking the condition if *number_01* is greater than *number_02*. But since *number_01* is actually less than *number_02*, the output is returned as False.

Logical operators in Python are used for conditional statements that're either True or False. Python has the following logical operators.

AND

OR

NOT

AND returns *True* if all the operands are True. Here, both *variable_01* and *variable_02* are True, hence the output is True.

Code:

variable_01 = True

variable_02 = True

variable_01 and variable_02

Output:

True

```
variable_01 = True
variable_02 = True

variable_01 and variable_02
```

```
True
```

OR returns *TRUE* if any of the operands are True. Here, the *variable_01* is True, but *variable_02* is False. But as at least one variable is True, the output is True. The output will be *False* if both *variable_01* and *variable_02* are False.

Code:

variable_01 = True

variable_02 = False

variable_01 or variable_02

Output:

True

```
variable_01 = True
variable_02 = False

variable_01 or variable_02
```

```
True
```

NOT returns *True* if the operand if False, it returns *False* if the operand is True. Here, the variable is True, so the output is False.

Code:

variable = True

not variable

Output:

False

```
variable = True

not variable
```

```
False
```

DECISION MAKING STATEMENTS IN PYTHON

To make decisions in Python, we can use the decision-making statements.

The decision-making statements in Python are the following.

- **if**
- **if-else**
- **elif**

The "**if**" statement is used for checking if a condition is True or False. If True, the block of code following the "if" statement is executed. For indicating the block of code, indentation is used.

Code:

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

if number > 100:
    print("Your Number is greater than 100")
```

Output:

```
Enter Number:  140
Your Number is greater than 100
```

Here, we're checking the condition whether the number is greater than 100. Since, the number that we've entered is indeed greater than 100, so the "if" statement evaluates to be True, and the block of code following the "if" statement gets executed.

An "**else**" statement is used when the condition inside the "if" statement is evaluated to be False, so the statements under the "else" statement gets evaluated.

Code:

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

if number > 100:
    print("Your Number is greater than 100")
else:
    print("Your Number is smaller than 100")
```

Output:

```
Enter Number:  80
Your Number is smaller than 100
```

Here, we're checking the condition whether the number is greater than 100. Since, the number that we've entered is indeed smaller than 100,

so the “if” statement evaluates to be False, and the block of code following the “else” statement gets executed.

If we want to check the multiple conditions, we can use the “**elif**” statements. “elif” will come after the “if” statement and will be checked in the order they appear.

Code:

```
exam_score = int(input("Enter Exam Score: "))

if (exam_score >= 90 and exam_score <= 100):
    print("You've got an A")
elif (exam_score >= 80 and exam_score < 90):
    print("You've got a B")
elif (exam_score >= 70 and exam_score < 80):
    print("You've got a C")
else:
    print("You failed")
```

Output:

```
Enter Exam Score:  76
You've got a C
```

```
exam_score = int(input("Enter Exam Score: "))

if (exam_score >= 90 and exam_score <= 100):
    print("You've got an A")
elif (exam_score >= 80 and exam_score < 90):
    print("You've got a B")
elif (exam_score >= 70 and exam_score < 80):
    print("You've got a C")
else:
    print("You failed")
```

```
Enter Exam Score:  76
You've got a C
```

Here, the third condition was evaluated to be True. If all of the conditions were evaluated to be False, then the “else” block would be executed.

LOOPS IN PYTHON

Loops in Python allows us to execute a statement multiple times.

There're two types of loops in Python.

- **"for" loop**
- **"while" loop**

A "**for**" loop is used for repeating over a certain sequence.

Code:

```
for number in range(20, 30):
    print(number)
```

Output:

```
20
21
22
23
24
25
26
27
28
29
```

```
for number in range(20, 30):
    print(number)
```

```
20
21
22
23
24
25
26
27
28
29
```

Here, the *range()* function takes a starting and an ending value. It generates a list of numbers between them, including the starting value and excluding the ending value. So, the above code will print the numbers from 20 to 29.

A "*step*" inside the *range()* function can be used for incrementing with a certain sequence.

Suppose we want to write a program that'll print all even numbers from 10 to 30. We can do it in the following way.

Code:

```
for number in range(10, 32, 2):
    print(number)
```

Output:

```
10
12
14
16
18
20
22
```

24

26

28

30

```
for number in range(10, 32, 2):
    print(number)
```

```
10
12
14
16
18
20
22
24
26
28
30
```

A “**while**” loop repeatedly executes the statements until the condition is evaluated to be False. So, in general it’ll execute the statements as long as the condition is evaluated to be True.

Code:

```
number = 50
while (number < 70):
    print(number)
    number = number + 3
```

Output:

50

53

56

59

62

65

68

```
number = 50
while (number < 70):
    print(number)
    number = number + 3
```

```
50
53
56
59
62
65
68
```

Here, the variable “*number*” initially has a value of 50. Then the “*while*” loop condition is evaluated to be True, so the block of code after the “*while*” loop gets executed as long as the “*number*” is less than 70. And after the *print()* statement, we’re updating the value of “*number*” by 3, so this’s very similar to the “step” in range() function.

FUNCTIONS IN PYTHON

Functions are a set of instructions that perform a specific task and can be reused over and over again.

Let's say, we want to have a function that'll take temperature input in Celsius and will convert it to Fahrenheit.

"*def*" is used for declaring a function. We can add parameters in function, like in this example we're going to add Celsius as the parameter.

Celsius to Fahrenheit formula is the following.

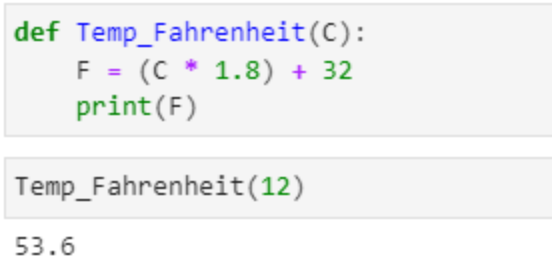
*Fahrenheit = (Celsius * 1.8) + 32*

Code:

```
def Temp_Fahrenheit(C):
    F = (C * 1.8) + 32
    print(F)
```

Output:

```
Temp_Fahrenheit(12)
53.6
```



The above function takes Celsius as the parameter, stores the converted value in Fahrenheit in the variable "F", and then prints it. To call the function, we only need to specify its name and parameters inside the parenthesis. Here, we're adding the value 12 as the Celsius

and printing the output as Fahrenheit. We can call the function as many times as we’d like.

SOME BASIC NUMBERS AND STRINGS OPERATIONS

There're lots of built-in functions in Python. Here, we'll look into some methods of manipulating the numbers and strings.

The *pow(a, b)* function returns the value of a to the power of b ().

Code:

power = pow(4,2)

print(power)

Output:

16

```
power = pow(4,2)
print(power)
```

```
16
```

The *min()* and *max()* functions are used for finding the minimum or maximum value.

Code:

find_minimum = min(20, 34, 12)

find_maximum = max(20, 34, 12)

print(find_minimum)

print(find_maximum)

Output:

12

34

```
find_minimum = min(20, 34, 12)
find_maximum = max(20, 34, 12)

print(find_minimum)
print(find_maximum)
```

```
12
34
```

The *abs()* function prints the absolute value of the given number.

Code:

find_absolute = abs(-25)

print(find_absolute)

Output:

25

```
find_absolute = abs(-25)
print(find_absolute)
```

```
25
```

To use all the numeric functions, we need to import a "*math*" module. Modules are external Python files that include functions and can be used by the Python programs. To import a module, the "*import*" statement is used.

import math

math.floor() takes the number as the parameter and returns the largest integer equal to or less than the given parameter.

math.ceil() takes the number as the parameter and returns the smallest integer equal to or greater than the given parameter.

Code:

import math

a = math.floor(7.4)

b = math.ceil(12.8)

print(a)

print(b)

Output:

7

13

```
import math

a = math.floor(7.4)
b = math.ceil(12.8)

print(a)
print(b)
```

```
7
13
```

The *math.sqrt()* returns the square root of a number.

Code:

import math

find_squareroot = math.sqrt(16)
print(find_squareroot)

Output:

4.0

```
import math

find_squareroot = math.sqrt(16)
print(find_squareroot)
```

```
4.0
```

Now, let's look at some of the most commonly used string functions.

capitalize() function returns the string in sentence with the first letter capitalized and the rest of them are small.

upper() function will convert the string to uppercase.

lower() function will convert the string to lowercase.

count() function returns the number of occurrences of the substring specified in the parameter.

find() function returns the location of the first occurrence of a substring as the parameter whether it exists in the given string. Note that the first character location will always be at 0, so the index starts at 0.

Code:

text = "who does not love python?"

a = text.capitalize() #capitalize the string
print(a)

b = text.upper() #converting the string to uppercase

```
print(b)

c = text.lower() #converting the string to lowercase
print(c)

d = text.count("o") #counting how many times "o" appears in the string
print(d)

e = text.find("o") #finding the first location of "o" in the string
print(e)
```

Output:

```
Who does not love python?
WHO DOES NOT LOVE PYTHON?
who does not love python?
5
2
```

```python
text = "who does not love python?"

a = text.capitalize()  #capitalize the string
print(a)

b = text.upper() #converting the string to uppercase
print(b)

c = text.lower() #converting the string to lowercase
print(c)

d = text.count("o") #counting how many times "o" appears in the string
print(d)

e = text.find("o") #finding the first location of "o" in the string
print(e)
```

```
Who does not love python?
WHO DOES NOT LOVE PYTHON?
who does not love python?
5
2
```

LISTS IN PYTHON

A *List* is a data structure in Python that can hold multiple values. The values can be of different types.

A *List* contains the items that're separated by the commas, and then the items are enclosed within square brackets.

list_01 = [23, 45, 64, 78]

list_02 = [32, "Paris", 44, "Berlin"]

The items in the list are identified using the positions or indexes. Indexes start at 0.

In "*list_01*", *23* is at index 0, *45* is at index 1, *64* is at index 2, *78* is at index 3.

Code:

list_01 = [23, 45, 64, 78]

list_02 = [32, "Paris", 44, "Berlin"]

print(list_01)

print(list_02)

Output:

[23, 45, 64, 78]

[32, 'Paris', 44, 'Berlin']

```
list_01 = [23, 45, 64, 78]
list_02 = [32, "Paris", 44, "Berlin"]

print(list_01)
print(list_02)
```

```
[23, 45, 64, 78]
[32, 'Paris', 44, 'Berlin']
```

The list items are accessed by the index numbers. In "*list_01*", if we want to access the value *64*, we need to write:

list_01[2]

Code:

list_01 = [23, 45, 64, 78]

list_01[2]

Output:

64

```
list_01 = [23, 45, 64, 78]

list_01[2]
```

```
64
```

We can also access multiple items from a list. To do that, we need to specify the starting index and the ending index, separated by a colon. In "*list_01*", if we want to access the values *23*, *45*, *64*, we need to write:

list_01[0:3]

Here, the starting index is inclusive, and the ending index is exclusive.

Code:

list_01 = [23, 45, 64, 78]

list_01[0:3]

Output:

[23, 45, 64]

```
list_01 = [23, 45, 64, 78]

list_01[0:3]
```

```
[23, 45, 64]
```

If we want to access the list items beginning from index 1, we can do it in the following way.

Code:

list_01 = [23, 45, 64, 78]

list_01[1:]

Output:

[45, 64, 78]

```
list_01 = [23, 45, 64, 78]

list_01[1:]
```

```
[45, 64, 78]
```

If we want to access the list items before index 3, we can do it in the following way.

Code:

list_01 = [23, 45, 64, 78]

list_01[:3]

Output:

[23, 45, 64]

```
list_01 = [23, 45, 64, 78]

list_01[:3]
```

```
[23, 45, 64]
```

We can also update the list. To do that, we first need to access the item in the list, then assign a new value to it.

Suppose if we want to update the value of the item at index 2 in *list_01*, we can write:

List_01[2] = 120

Code:

list_01 = [23, 45, 64, 78]

list_01[2] = 120

print(list_01)

Output:

[23, 45, 120, 78]

```
list_01 = [23, 45, 64, 78]

list_01[2] = 120

print(list_01)
```

```
[23, 45, 120, 78]
```

If we want to add an item at the end of the list, we can use the *append()* function.

Code:

list_02 = [32, "Paris", 44, "Berlin"]

list_02.append("Rome")

print(list_02)

Output:

[32, 'Paris', 44, 'Berlin', 'Rome']

```
list_02 = [32, "Paris", 44, "Berlin"]

list_02.append("Rome")

print(list_02)
```

```
[32, 'Paris', 44, 'Berlin', 'Rome']
```

If we want to remove an item at the end of the list, we can use the *remove()* function.

Code:

list_02 = [32, "Paris", 44, "Berlin"]

list_02.remove(44)

print(list_02)

Output:

[32, 'Paris', 'Berlin']

```
list_02 = [32, "Paris", 44, "Berlin"]

list_02.remove(44)

print(list_02)
```

```
[32, 'Paris', 'Berlin']
```

We can add two different lists.

Code:

list_01 = [23, 45, 64, 78]
list_02 = [32, "Paris", 44, "Berlin"]

list_01 + list_02

Output:

[23, 45, 64, 78, 32, 'Paris', 44, 'Berlin']

```
list_01 = [23, 45, 64, 78]
list_02 = [32, "Paris", 44, "Berlin"]

list_01 + list_02
```

```
[23, 45, 64, 78, 32, 'Paris', 44, 'Berlin']
```

We can also have multiple lists within a parent list.

list_03 = [24, 67, [12, 45, 62], 36, ["London", 1940, "Toronto"]]

In this "*list_03*" list, the *[12, 45, 62]* list is at index 2 of the parent *list_03* and the *["London", 1940, "Toronto"]* list is at index 4 of the parent *list_03*. So, if we want to access "*Toronto*" here, we first need to access this list from the main parent list, then the item from the new list.

Code:

list_03 = [24, 67, [12, 45, 62], 36, ["London", 1940, "Toronto"]]

list_03[4][2]

Output:

'Toronto'

```
list_03 = [24, 67, [12, 45, 62], 36, ["London", 1940, "Toronto"]]

list_03[4][2]
```

```
'Toronto'
```

We can also get the index of an item in a list.

Code:

list_01 = [23, 45, 64, 78]

list_01.index(45)

Output:

1

```
list_01 = [23, 45, 64, 78]

list_01.index(45)
```

```
1
```

We can also calculate how many times an item appears in a list.

Code:

list_01 = [23, 45, 67, 45, 64, 72, 78]

list_01.count(45)

Output:

2

```
list_01 = [23, 45, 67, 45, 64, 72, 78]

list_01.count(45)
```

```
2
```

You can also sort and reverse sort the items in a list.

Code:

list_01 = [45, 23, 78, 64]

list_01.sort() #sorting the list_01

list_01

Output:

[23, 45, 64, 78]

```
list_01 = [45, 23, 78, 64]

list_01.sort()  #sorting the list_01

list_01
```

```
[23, 45, 64, 78]
```

Code:

list_01 = [45, 23, 78, 64]

list_01.sort(reverse=True) #reverse sorting the list_01

list_01

Output:

[78, 64, 45, 23]

```
list_01 = [45, 23, 78, 64]

list_01.sort(reverse=True)  #reverse sorting the list_01

list_01
```

```
[78, 64, 45, 23]
```

TUPLES IN PYTHON

A *Tuple* is a data structure like a list that can hold multiple values, and the values may or may not be of the same type.

A *tuple* is a sequence of items that are separated by the commas and the items are enclosed within parentheses.

tuple_01 = (65, 12, 54, 34)

tuple_02 = (16, “Germany”, 38, “USA”)

Just like the lists, tuples are also identified by the indexes and the index start from 0.

In “*tuple_01*”, *65* is at index 0, *12* is at index 1, *54* is at index 2, *34* is at index 3.

Code:

tuple_01 = (65, 12, 54, 34)

tuple_02 = (16, "Germany", 38, "USA")

print(tuple_01)

print(tuple_02)

Output:

(65, 12, 54, 34)

(16, 'Germany', 38, 'USA')

```
tuple_01 = (65, 12, 54, 34)
tuple_02 = (16, "Germany", 38, "USA")

print(tuple_01)
print(tuple_02)
```

```
(65, 12, 54, 34)
(16, 'Germany', 38, 'USA')
```

The tuple items are accessed by the index numbers. In “*tuple_01*”, if we want to access the value *54*, we need to write:

tuple_01[2]

Code:

tuple_01 = (65, 12, 54, 34)

tuple_01[2]

Output:

54

```
tuple_01 = (65, 12, 54, 34)

tuple_01[2]
```

```
54
```

We can also access multiple items from a tuple. To do that, we need to specify the starting index and the ending index, separated by a colon. In “*tuple_01*”, if we want to access the values *65, 12, 54* we need to write:

tuple_01[0:3]

Here, the starting index is inclusive, and the ending index is exclusive.

Code:

tuple_01 = (65, 12, 54, 34)

tuple_01[0:3]

Output:

(65, 12, 54)

```
tuple_01 = (65, 12, 54, 34)

tuple_01[0:3]
```

```
(65, 12, 54)
```

If we want to access the tuple items beginning from index 1, we can do it in the following way.

Code:

tuple_01 = (65, 12, 54, 34)

tuple_01[1:]

Output:

(12, 54, 34)

```
tuple_01 = (65, 12, 54, 34)

tuple_01[1:]
```

```
(12, 54, 34)
```

If we want to access the tuple items before index 3, we can do it in the following way.

Code:

tuple_01 = (65, 12, 54, 34)

tuple_01[:3]

Output:

(65, 12, 54)

```
tuple_01 = (65, 12, 54, 34)

tuple_01[:3]
```

```
(65, 12, 54)
```

But there's a major difference between tuples and lists. Tuples are immutable, which means that the items inside a tuple can't be updated or deleted. However, we can delete the entire tuple. For this, we can use the "*del*" function.

Code:

tuple_01 = (65, 12, 54, 34)

del tuple_01

tuple_01

Output:

NameError: name 'tuple_01' is not defined

```
tuple_01 = (65, 12, 54, 34)
```

```
del tuple_01
```

```
tuple_01
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [3], line 1
----> 1 tuple_01

NameError: name 'tuple_01' is not defined
```

As we've deleted the "*tuple_01*", it doesn't exist anymore.

DICTIONARY IN PYTHON

A *Dictionary* is a data structure that contains data in the combination of pairs of keys and values. A key and value pair is the builds an item in the dictionary.

Items in the dictionary are separated by commas. Keys and values are separated by a colon. And the items in the dictionary are enclosed within the curly brackets.

dictionary_01 = {“food_01”: “Pasta”, “food_02”: “Meatball”, “food_03”: “Baklava”, “food_04”: “Kebab”}

Code:

dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01

Output:

{'food_01': 'Pasta',

'food_02': 'Meatball',

'food_03': 'Baklava',

'food_04': 'Kebab'}

```
dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01
```

```
{'food_01': 'Pasta',
 'food_02': 'Meatball',
 'food_03': 'Baklava',
 'food_04': 'Kebab'}
```

Here, the “*dictionary_01*” contains 4 items.

The first item is “*food_01*” with value “*Pasta*”.

The second item is “*food_02*” with value “*Meatball*”.

The third item is “*food_03*” with value “*Baklava*”.

The fourth item is “*food_04*” with value “*Kebab*”.

The items in the dictionary can be accessed using the keys. In “*dictionary_01*”, if we want to access the value “*Baklava*”, we need to write:

Code:

dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01["food_03"]

Output:

'Baklava'

```
dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01["food_03"]
```

```
'Baklava'
```

In the dictionary, we can add a new key-value pair or update an existing key-value pair.

Suppose in “*dictionary_01*”, we want to update the key “*food_02*” with a new value “*Cheesecake*”. We can do it in the following way.

Code:

dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01["food_02"] = "Cheesecake"

dictionary_01

Output:

{'food_01': 'Pasta',
'food_02': 'Cheesecake',
'food_03': 'Baklava',
'food_04': 'Kebab'}

```
dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01["food_02"] = "Cheesecake"

dictionary_01
```

```
{'food_01': 'Pasta',
 'food_02': 'Cheesecake',
 'food_03': 'Baklava',
 'food_04': 'Kebab'}
```

Now, suppose in "*dictionary_01*", if we want to add a new key "*food_05*" with the value "*Cilantro*", we can do it in the following way.

Code:

dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01["food_05"] = "Cilantro"

dictionary_01

Output:

{'food_01': 'Pasta',
'food_02': 'Meatball',
'food_03': 'Baklava',
'food_04': 'Kebab',
'food_05': 'Cilantro'}

```
dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

dictionary_01["food_05"] = "Cilantro"

dictionary_01
```

```
{'food_01': 'Pasta',
 'food_02': 'Meatball',
 'food_03': 'Baklava',
 'food_04': 'Kebab',
 'food_05': 'Cilantro'}
```

In a dictionary, we can remove an item from the dictionary, as well as delete an entire dictionary.

Suppose in "*dictionary_01*", if we want to delete the newly added item, we can do it in the following way.

Code:

dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab", "food_05": "Cilantro"}

del dictionary_01["food_05"]

dictionary_01

Output:

{'food_01': 'Pasta',
'food_02': 'Meatball',
'food_03': 'Baklava',
'food_04': 'Kebab'}

```
dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab", "food_05": "Cilantro"}

del dictionary_01["food_05"]

dictionary_01
```

```
{'food_01': 'Pasta',
 'food_02': 'Meatball',
 'food_03': 'Baklava',
 'food_04': 'Kebab'}
```

If we want to delete the entire "*dictionary_01*", we can do it in the following way.

Code:

dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}

del dictionary_01

dictionary_01

Output:

NameError: name 'dictionary_01' is not defined

```
dictionary_01 = {"food_01": "Pasta", "food_02": "Meatball", "food_03": "Baklava", "food_04": "Kebab"}
```

```
del dictionary_01
```

```
dictionary_01
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [15], line 1
----> 1 dictionary_01

NameError: name 'dictionary_01' is not defined
```

As we've deleted the "*dictionary_01*", it doesn't exist anymore.

UNIT 02: NUMPY

INTRODUCTION TO NUMPY

"*NumPy*" means "*Numerical Python*". It is a linear algebra library for Python, and commonly is used for linear algebra, matrices and arrays. NumPy is also super-fast. NumPy arrays are about 50 times faster than the Python lists.

We can install NumPy in our system by running the following command in the terminal.

pip install numpy

In this book, we'll mostly work with the NumPy arrays. There're two types of NumPy arrays: *vectors* and *matrices*. Vectors are 1-D arrays and matrices are 2-D arrays. Vectors are matrices are both arrays whether 1-D or 2-D.

Once NumPy is installed, we can import it in the following way.

import numpy as np

```
import numpy as np
```

Suppose we've a Python list, and we can convert it to NumPy array in the following way.

Code:

import numpy as np

list_01 = [26, 87, 24, 56]

np.array(list_01)

Output:

array([26, 87, 24, 56])

```
import numpy as np

list_01 = [26, 87, 24, 56]
np.array(list_01)
```

```
array([26, 87, 24, 56])
```

The output is a 1-D array.

NUMPY ARRAYS

If we have a Python list of lists, we can convert it to 2-D NumPy array or 2-D matrices.

Code:

import numpy as np

list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]

np.array(list_02)

Output:

array([[26, 87, 24, 56],
[15, 67, 45, 98],
[65, 34, 96, 16]])

```
import numpy as np

list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]
np.array(list_02)
```

```
array([[26, 87, 24, 56],
       [15, 67, 45, 98],
       [65, 34, 96, 16]])
```

The output is a 3 X 4 matrix (3 rows and 4 columns).

If we want to create a NumPy array of zeros, we can do it in the following way.

Code:

np.zeros((4,3))

Output:

array([[0., 0., 0.],
[0., 0., 0.],
[0., 0., 0.],
[0., 0., 0.]])

```
np.zeros((4,3))
```

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

Th output is a 4 X 3 matrix (4 rows and 3 columns).

If we want to create a NumPy array of ones, we can do it in the following way.

Code:

np.ones((3,2))

Output:

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

```
np.ones((3,2))
```

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

The output is a 3 X 2 matrix.

We can also create an array with evenly or linearly spaced values. We can do it in the following way.

Code:

np.linspace(2, 26, 10)

Output:

array([2. , 4.66666667, 7.33333333, 10. , 12.66666667,
15.33333333, 18. , 20.66666667, 23.33333333, 26.
])

```
np.linspace(2, 26, 10)
```

```
array([ 2.        ,  4.66666667,  7.33333333, 10.        , 12.66666667,
       15.33333333, 18.        , 20.66666667, 23.33333333, 26.        ])
```

Here, in this 1-D array, the beginning number is 2, the ending number is 26, and there're 10 equally distanced points between 2 and 26.

We can create an array of evenly distributed values in another way.

Code:

np.arange(2, 26, 4)

Output:

array([2, 6, 10, 14, 18, 22])

```
np.arange(2, 26, 4)
```

```
array([ 2,  6, 10, 14, 18, 22])
```

Here, in this 1-D array, the beginning number is 2 and the ending number is 26. But the beginning number is inclusive, and the ending number is exclusive. And 4 is the step number.

If you want to create an identity matrix, you can do it in the following way.

Code:

np.eye(4)

Output:

array([[1., 0., 0., 0.],
[0., 1., 0., 0.],
[0., 0., 1., 0.],
[0., 0., 0., 1.]])

```
np.eye(4)
```

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

The output is a 4 X 4 identity matrix.

If we want to generate an array or random numbers from 0 to 1, we can do it in the following way.

Code:

matrix_1d = np.random.random(4)
matrix_2d = np.random.random((2, 3))

print(matrix_1d)
print("------------------------------")
print("------------------------------")
print(matrix_2d)

Output:

[0.3362997 0.28730457 0.4147856 0.57980998]

[[0.89985886 0.07394658 0.33091669]
[0.59356934 0.44973181 0.88831178]]

```
matrix_1d = np.random.random(4)
matrix_2d = np.random.random((2, 3))

print(matrix_1d)
print("---------------------------------")
print("---------------------------------")
print(matrix_2d)
```

```
[0.3362997  0.28730457 0.4147856  0.57980998]
---------------------------------
---------------------------------
[[0.89985886 0.07394658 0.33091669]
 [0.59356934 0.44973181 0.88831178]]
```

Here, "*matrix_1d*" is a 1-D matrix and "*matrix_2d*" is a 2-D matrix with 2 rows and 3 columns.

If we want to generate an array of 15 random integers from 4 to 67, we can do it in the following way.

Code:

np.random.randint(4, 67, 15)

Output:

array([8, 14, 31, 11, 34, 7, 30, 49, 53, 40, 62, 26, 65, 43, 27])

```
np.random.randint(4, 67, 15)
```

```
array([ 8, 14, 31, 11, 34,  7, 30, 49, 53, 40, 62, 26, 65, 43, 27])
```

Here, 4 is the low number and 67 is the high number. The low number is inclusive and high number is exclusive. So that means 4 has a chance of getting selected while 67 has no chance of getting selected.

We can also reshape an array, for example we can convert a 1-D array into a 2-D array. We need to use the reshape() method for this.

Code:

array_01 = np.arange(30)

array_01.reshape(6,5)

Output:

array([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9],
[10, 11, 12, 13, 14],
[15, 16, 17, 18, 19],
[20, 21, 22, 23, 24],
[25, 26, 27, 28, 29]])

```
array_01 = np.arange(30)
array_01.reshape(6,5)
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

Here, *np.arange(30)* generates a 1-D array of 30 numbers starting from 0 and ending at 29. If we use *reshape(6,5)*, we are converting this array into a 6 X 5 matrix. We can convert it to either 5 X 6 or 6 X 5 matrix, because we have to make sure that the new matrix contains the same number of elements as the original 1-D array. If we use reshape(8,7) that means the new matrix has 56 elements, but the original array has 30 elements, so that'll result in an error.

If we want to find the shape of an array, we can do it in the following way.

Code:

list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]

array_02 = np.array(list_02)

array_02.shape

Output:

(3, 4)

```
list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]
array_02 = np.array(list_02)

array_02.shape
```

```
(3, 4)
```

Here, the above array has 3 rows and 4 columns.

We can also get the length, size and datatype of an array. Size means how many elements are in the array.

Code:

list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]
array_02 = np.array(list_02)

array_02_length = len(array_02)
array_02_size = array_02.size
array_02_datatype = array_02.dtype

print("The length of array_02 is: ", array_02_length)
print("The size of array_02 is: ", array_02_size)
print("The datatype of array_02 is: ", array_02_datatype)

Output:

The length of array_02 is: 3
The size of array_02 is: 12
The datatype of array_02 is: int32

```
list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]
array_02 = np.array(list_02)

array_02_length = len(array_02)
array_02_size = array_02.size
array_02_datatype = array_02.dtype

print("The length of array_02 is: ", array_02_length)
print("The size of array_02 is: ", array_02_size)
print("The datatype of array_02 is: ", array_02_datatype)
```

```
The length of array_02 is:  3
The size of array_02 is:  12
The datatype of array_02 is:  int32
```

MATHEMATICS WITH NUMPY ARRAYS

We can do the basic mathematical operations with NumPy arrays.

Code:

```
array_01 = np.arange(2, 27).reshape(5,5)
array_02 = np.arange(9, 34).reshape(5,5)

print("array_01: \n", array_01)  #array_01
print("----------------------")
print("array_02: \n", array_02)  #array_02
print("----------------------")
print("array_01 + array_02: \n", array_01 + array_02)  #Sum of the arrays
print("----------------------")
print("array_02 - array_01: \n", array_02 - array_01)  #Subtraction of the arrays
print("----------------------")
print("array_01 * array_02: \n", array_01 * array_02)  #Multiplication of the arrays
print("----------------------")
print("array_02 / array_01: \n", array_02 / array_01)  #Division of the arrays
print("----------------------")
print("Square root of array_01: \n", np.sqrt(array_01))  #Square root of array_01
print("----------------------")
print("Exponentiation of array_01: \n", np.exp(array_01))  #Exponentiation of array_01
```

Output:

```
array_01:
 [[ 2  3  4  5  6]
 [ 7  8  9 10 11]
[12 13 14 15 16]
[17 18 19 20 21]
[22 23 24 25 26]]
------------------------
array_02:
[[ 9 10 11 12 13]
[14 15 16 17 18]
[19 20 21 22 23]
[24 25 26 27 28]
[29 30 31 32 33]]
------------------------
array_01 + array_02:
[[11 13 15 17 19]
[21 23 25 27 29]
[31 33 35 37 39]
[41 43 45 47 49]
[51 53 55 57 59]]
------------------------
array_02 - array_01:
[[7 7 7 7 7]
[7 7 7 7 7]
[7 7 7 7 7]
[7 7 7 7 7]
[7 7 7 7 7]]
```

```
----------
array_01 * array_02:
 [[ 18  30  44  60  78]
[ 98 120 144 170 198]
[228 260 294 330 368]
[408 450 494 540 588]
[638 690 744 800 858]]
----------
array_02 / array_01:
 [[4.5        3.33333333 2.75       2.4        2.16666667]
 [2.         1.875      1.77777778 1.7        1.63636364]
 [1.58333333 1.53846154 1.5        1.46666667 1.4375    ]
 [1.41176471 1.38888889 1.36842105 1.35       1.33333333]
 [1.31818182 1.30434783 1.29166667 1.28       1.26923077]]
----------
Square root of array_01:
 [[1.41421356 1.73205081 2.         2.23606798 2.44948974]
 [2.64575131 2.82842712 3.         3.16227766 3.31662479]
 [3.46410162 3.60555128 3.74165739 3.87298335 4.        ]
[4.12310563 4.24264069 4.35889894 4.47213595 4.58257569]
 [4.69041576 4.79583152 4.89897949 5.         5.09901951]]
----------
Exponentiation of array_01:
[[7.38905610e+00 2.00855369e+01 5.45981500e+01
1.48413159e+02
  4.03428793e+02]
[1.09663316e+03 2.98095799e+03 8.10308393e+03
2.20264658e+04
  5.98741417e+04]
```

[1.62754791e+05 4.42413392e+05 1.20260428e+06 3.26901737e+06

8.88611052e+06]

[2.41549528e+07 6.56599691e+07 1.78482301e+08 4.85165195e+08

1.31881573e+09]

[3.58491285e+09 9.74480345e+09 2.64891221e+10 7.20048993e+10

1.95729609e+11]]

```
array_01 = np.arange(2, 27).reshape(5,5)
array_02 = np.arange(9, 34).reshape(5,5)

print("array_01: \n", array_01)  #array_01
print("------------------------")
print("array_02: \n", array_02)  #array_02
print("------------------------")
print("array_01 + array_02: \n", array_01 + array_02)  #Sum of the arrays
print("------------------------")
print("array_02 - array_01: \n", array_02 - array_01)  #Subtraction of the arrays
print("------------------------")
print("array_01 * array_02: \n", array_01 * array_02)  #Multiplication of the arrays
print("------------------------")
print("array_02 / array_01: \n", array_02 / array_01)  #Division of the arrays
print("------------------------")
print("Square root of array_01: \n", np.sqrt(array_01))  #Square root of array_01
print("------------------------")
print("Exponentiation of array_01: \n", np.exp(array_01))  #Exponentiation of array_01
```

```
array_01:
 [[ 2  3  4  5  6]
 [ 7  8  9 10 11]
 [12 13 14 15 16]
 [17 18 19 20 21]
 [22 23 24 25 26]]
------------------------
array_02:
 [[ 9 10 11 12 13]
 [14 15 16 17 18]
 [19 20 21 22 23]
 [24 25 26 27 28]
 [29 30 31 32 33]]
------------------------
array_01 + array_02:
 [[11 13 15 17 19]
 [21 23 25 27 29]
 [31 33 35 37 39]
 [41 43 45 47 49]
 [51 53 55 57 59]]
------------------------
array_02 - array_01:
 [[7 7 7 7 7]
 [7 7 7 7 7]
 [7 7 7 7 7]
 [7 7 7 7 7]
 [7 7 7 7 7]]
------------------------
array_01 * array_02:
 [[ 18  30  44  60  78]
 [ 98 120 144 170 198]
 [228 260 294 330 368]
 [408 450 494 540 588]
 [638 690 744 800 858]]
------------------------
array_02 / array_01:
 [[4.5        3.33333333 2.75       2.4        2.16666667]
 [2.         1.875      1.77777778 1.7        1.63636364]
 [1.58333333 1.53846154 1.5        1.46666667 1.4375    ]
 [1.41176471 1.38888889 1.36842105 1.35       1.33333333]
 [1.31818182 1.30434783 1.29166667 1.28       1.26923077]]
```

```
------------------------
Square root of array_01:
 [[1.41421356 1.73205081 2.         2.23606798 2.44948974]
 [2.64575131 2.82842712 3.         3.16227766 3.31662479]
 [3.46410162 3.60555128 3.74165739 3.87298335 4.        ]
 [4.12310563 4.24264069 4.35889894 4.47213595 4.58257569]
 [4.69041576 4.79583152 4.89897949 5.         5.09901951]]
------------------------
Exponentiation of array_01:
 [[7.38905610e+00 2.00855369e+01 5.45981500e+01 1.48413159e+02
  4.03428793e+02]
 [1.09663316e+03 2.98095799e+03 8.10308393e+03 2.20264658e+04
  5.98741417e+04]
 [1.62754791e+05 4.42413392e+05 1.20260428e+06 3.26901737e+06
  8.88611052e+06]
 [2.41549528e+07 6.56599691e+07 1.78482301e+08 4.85165195e+08
  1.31881573e+09]
 [3.58491285e+09 9.74480345e+09 2.64891221e+10 7.20048993e+10
  1.95729609e+11]]
```

We can also do an element-wise comparison between the arrays.

Code:

```
list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]
list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]

array_01 = np.array(list_01)
array_02 = np.array(list_02)

print("array_01: \n", array_01)  #array_01
print("------------------------")
print("array_02: \n", array_02)  #array_02
print("------------------------")
print("array_01 == array_02: \n", array_01 == array_02)
#Checking which elements in array_01 equals to array_02
print("------------------------")
```

```
print("array_01 > 30: \n", array_01 > 30)  #Checking which elements in array_01 are greater than 30
```

Output:

```
array_01:
[[67 21 63 19]
[28 67 42 98]
[65 28 96 38]]
----------
array_02:
[[26 87 24 56]
[15 67 45 98]
[65 34 96 16]]
----------
array_01 == array_02:
[[False False False False]
 [False  True False  True]
 [ True False  True False]]
----------
array_01 > 30:
 [[ True False  True False]
 [False  True  True  True]
 [ True False  True  True]]
```

```
list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]
list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]

array_01 = np.array(list_01)
array_02 = np.array(list_02)

print("array_01: \n", array_01)  #array_01
print("-----------------------")
print("array_02: \n", array_02)  #array_02
print("-----------------------")
print("array_01 == array_02: \n", array_01 == array_02)  #Checking which elements in array_01 equals to array_02
print("-----------------------")
print("array_01 > 30: \n", array_01 > 30)  #Checking which elements in array_01 are greater than 30
```

```
array_01:
 [[67 21 63 19]
 [28 67 42 98]
 [65 28 96 38]]
-----------------------
array_02:
 [[26 87 24 56]
 [15 67 45 98]
 [65 34 96 16]]
-----------------------
array_01 == array_02:
 [[False False False False]
 [False  True False  True]
 [ True False  True False]]
-----------------------
array_01 > 30:
 [[ True False  True False]
 [False  True  True  True]
 [ True False  True  True]]
```

We can also add, subtract, multiply and divide the elements in an array at the same time in the following way.

Code:

list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]

array_02 = np.array(list_02)

print("array_02: \n", array_02) #array_02

print("Adding 5 to all the elements: \n", (array_02 + 5)) #Adding 5 to each element

print("Subtracting 7 from all the elements: \n", (array_02 - 7)) #Subtracting 7 from each element

print("Multiplying all the elements by 4: \n", (array_02 * 4)) #Multiplying each element by 4

print("Dividing all the elements by 8: \n", (array_02 / 8)) #Dividing each element by 8

Output:

```
array_02:
[[26 87 24 56]
[15 67 45 98]
[65 34 96 16]]
Adding 5 to all the elements:
 [[ 31  92  29  61]
 [ 20  72  50 103]
 [ 70  39 101  21]]
Subtracting 7 from all the elements:
[[19 80 17 49]
[ 8 60 38 91]
 [58 27 89  9]]
Multiplying all the elements by 4:
 [[104 348  96 224]
[ 60 268 180 392]
 [260 136 384  64]]
Dividing all the elements by 8:
 [[ 3.25  10.875  3.     7.   ]
 [ 1.875  8.375  5.625 12.25 ]
 [ 8.125  4.25  12.     2.   ]]
```

```
list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]
array_02 = np.array(list_02)

print("array_02: \n", array_02)  #array_02
print("Adding 5 to all the elements: \n", (array_02 + 5))  #Adding 5 to each element
print("Subtracting 7 from all the elements: \n", (array_02 - 7))  #Subtracting 7 from each element
print("Multiplying all the elements by 4: \n", (array_02 * 4))  #Multiplying each element by 4
print("Dividing all the elements by 8: \n", (array_02 / 8))  #Dividing each element by 8
```

```
array_02:
 [[26 87 24 56]
 [15 67 45 98]
 [65 34 96 16]]
Adding 5 to all the elements:
 [[ 31  92  29  61]
 [ 20  72  50 103]
 [ 70  39 101  21]]
Subtracting 7 from all the elements:
 [[19 80 17 49]
 [ 8 60 38 91]
 [58 27 89  9]]
Multiplying all the elements by 4:
 [[104 348  96 224]
 [ 60 268 180 392]
 [260 136 384  64]]
Dividing all the elements by 8:
 [[ 3.25  10.875  3.     7.   ]
 [ 1.875  8.375  5.625 12.25 ]
 [ 8.125  4.25  12.     2.   ]]
```

We can also find the sum, minimum, maximum, mean and median values of an array in the following way.

Code:

list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]

array_01 = np.array(list_01)

print("array_01: \n", array_01) #array_01

print("----------")

print("Sum of all elements in array_01: \n", array_01.sum()) #Sum of all elements in array_01

print("----------")

```
print("Minimum element value in array_01: \n", array_01.min()) #Minimum element value in array_01
print("----------------------")
print("Maximum element value in array_01: \n", array_01.max()) #Maximum element value in array_01
print("----------------------")
print("Mean value in array_01: \n", array_01.mean())  #Mean value in array_01
print("----------------------")
print("Median value in array_01: \n", np.median(array_01)) #Median value in array_01
```

Output:

```
array_01:
[[67 21 63 19]
[28 67 42 98]
[65 28 96 38]]
----------------------
Sum of all elements in array_01:
632
----------------------
Minimum element value in array_01:
19
----------------------
Maximum element value in array_01:
98
----------------------
Mean value in array_01:
52.666666666666664
```

Median value in array_01:

52.5

```
list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]

array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01
print("------------------------")
print("Sum of all elements in array_01: \n", array_01.sum())  #Sum of all elements in array_01
print("------------------------")
print("Minimum element value in array_01: \n", array_01.min())  #Minimum element value in array_01
print("------------------------")
print("Maximum element value in array_01: \n", array_01.max())  #Maximum element value in array_01
print("------------------------")
print("Mean value in array_01: \n", array_01.mean())  #Mean value in array_01
print("------------------------")
print("Median value in array_01: \n", np.median(array_01))  #Median value in array_01
```

```
array_01:
 [[67 21 63 19]
 [28 67 42 98]
 [65 28 96 38]]
------------------------
Sum of all elements in array_01:
 632
------------------------
Minimum element value in array_01:
 19
------------------------
Maximum element value in array_01:
 98
------------------------
Mean value in array_01:
 52.666666666666664
------------------------
Median value in array_01:
 52.5
```

SORTING THE ARRAYS

We can also sort the NumPy arrays.

Code:

```
list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]

array_01 = np.array(list_01)
print("Original array_01: \n", array_01)

print("---------------------------------")

array_01.sort(axis = 0)
print("Sorted each columns in array_01: \n", array_01)
```

Output:

```
Original array_01:
[[67 21 63 19]
[28 67 42 98]
[65 28 96 38]]
---------------------------------
Sorted each columns in array_01:
[[28 21 42 19]
[65 28 63 38]
[67 67 96 98]]
```

```
list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]

array_01 = np.array(list_01)
print("Original array_01: \n", array_01)

print("-----------------------------------")

array_01.sort(axis = 0)
print("Sorted each columns in array_01: \n", array_01)
```

```
Original array_01:
 [[67 21 63 19]
 [28 67 42 98]
 [65 28 96 38]]
-----------------------------------
Sorted each columns in array_01:
 [[28 21 42 19]
 [65 28 63 38]
 [67 67 96 98]]
```

Here, "*axis = 0*" means the matrix will be sorted in each column, from the lowest value to the highest value.

Code:

list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]

array_01 = np.array(list_01)
print("Original array_01: \n", array_01)

print("-----------------------------------")

array_01.sort(axis = 1)
print("Sorted each rows in array_01: \n", array_01)

Output:

Original array_01:

[[67 21 63 19]
[28 67 42 98]
[65 28 96 38]]

Sorted each rows in array_01:
[[19 21 63 67]
[28 42 67 98]
[28 38 65 96]]

```
list_01 = [[67, 21, 63, 19], [28, 67, 42, 98], [65, 28, 96, 38]]

array_01 = np.array(list_01)
print("Original array_01: \n", array_01)

print("-----------------------------------")

array_01.sort(axis = 1)
print("Sorted each rows in array_01: \n", array_01)
```

```
Original array_01:
 [[67 21 63 19]
 [28 67 42 98]
 [65 28 96 38]]
-----------------------------------
Sorted each rows in array_01:
 [[19 21 63 67]
 [28 42 67 98]
 [28 38 65 96]]
```

Here, "*axis = 1*" means the matrix will be sorted in each row, from the lowest value to the highest value.

NUMPY ARRAY INDEXING AND SLICING

Array indexing in NumPy means accessing an array element in the NumPy array. NumPy array indexes start from 0, and we can access any element just by referring to the index number.

Code:

```
list_01 = [26, 87, 24, 56]
array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01

print("1st index of array_01: \n", array_01[0])  #Accesing 1st index of array_01
print("2nd index of array_01: \n", array_01[1])  #Accesing 2nd index of array_01
print("3rd index of array_01: \n", array_01[2])  #Accesing 3rd index of array_01
print("4th index of array_01: \n", array_01[3])  #Accesing 4th index of array_01
```

Output:

```
array_01:
[26 87 24 56]
1st index of array_01:
26
2nd index of array_01:
87
3rd index of array_01:
24
```

4th index of array_01:

56

```
list_01 = [26, 87, 24, 56]
array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01

print("1st index of array_01: \n", array_01[0])  #Accesing 1st index of array_01
print("2nd index of array_01: \n", array_01[1])  #Accesing 2nd index of array_01
print("3rd index of array_01: \n", array_01[2])  #Accesing 3rd index of array_01
print("4th index of array_01: \n", array_01[3])  #Accesing 4th index of array_01
```

```
array_01:
 [26 87 24 56]
1st index of array_01:
 26
2nd index of array_01:
 87
3rd index of array_01:
 24
4th index of array_01:
 56
```

We can do the same thing for the 2-D arrays. We just need to mention the row and the column position of the element. The first element in a row start from 0, and the first element in a column start from 0 as well.

Code:

list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]

array_02 = np.array(list_02)

print("array_02: \n", array_02) #array_02

#Element 45 sits at the 2nd row (index 1) and 3rd column (index 2) in array_02.

print("Accessing 45: \n", array_02[1, 2]) #Accessing the element at index (1, 2)

#Element 34 sits at the 3rd row (index 2) and 2nd column (index 1) in array_02.

print("Accessing 34: \n", array_02[2, 1]) #Accessing the element at index (2, 1)

Output:

array_02:

[[26 87 24 56]

[15 67 45 98]

[65 34 96 16]]

Accessing 45:

45

Accessing 34:

34

```
list_02 = [[26, 87, 24, 56], [15, 67, 45, 98], [65, 34, 96, 16]]
array_02 = np.array(list_02)

print("array_02: \n", array_02)  #array_02

#Element 45 sits at the 2nd row (index 1) and 3rd column (index 2) in array_02.
print("Accessing 45: \n", array_02[1, 2])  #Accessing the element at index (1, 2)

#Element 34 sits at the 3rd row (index 2) and 2nd column (index 1) in array_02.
print("Accessing 34: \n", array_02[2, 1])  #Accessing the element at index (2, 1)
```

```
array_02:
 [[26 87 24 56]
 [15 67 45 98]
 [65 34 96 16]]
Accessing 45:
 45
Accessing 34:
 34
```

We can also slice an array. We need to mention the beginning and ending index. The beginning index is inclusive, and the ending index is

exclusive.

Code:

```
list_01 = [26, 87, 24, 56, 38, 45, 94, 68]
array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01

print("Accessing the 1st three elements: \n", array_01[0: 3]) #Accessing elements from index 0 through index 2
print("Accessing 2nd through 5th elements: \n", array_01[1: 5]) #Accessing elements from index 1 through index 4
print("Accessing the elements before the 6th element: \n", array_01[:5])  #Accessing elements before index 5
print("Accessing the elements from the 3rd element: \n", array_01[2:])  #Accessing elements from index 2
print("Accessing the elements after the 3rd element: \n", array_01[3:])  #Accessing elements after index 2
```

Output:

```
array_01:
[26 87 24 56 38 45 94 68]
Accessing the 1st three elements:
[26 87 24]
Accessing 2nd through 5th elements:
[87 24 56 38]
Accessing the elements before the 6th element:
[26 87 24 56 38]
Accessing the elements from the 3rd element:
[24 56 38 45 94 68]
```

Accessing the elements after the 3rd element:

[56 38 45 94 68]

```
list_01 = [26, 87, 24, 56, 38, 45, 94, 68]
array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01

print("Accessing the 1st three elements: \n", array_01[0: 3])  #Accessing elements from index 0 through index 2
print("Accessing 2nd through 5th elements: \n", array_01[1: 5])  #Accessing elements from index 1 through index 4
print("Accessing the elements before the 6th element: \n", array_01[:5])  #Accessing elements before index 5
print("Accessing the elements from the 3rd element: \n", array_01[2:])  #Accessing elements from index 2
print("Accessing the elements after the 3rd element: \n", array_01[3:])  #Accessing elements after index 2
```

```
array_01:
 [26 87 24 56 38 45 94 68]
Accessing the 1st three elements:
 [26 87 24]
Accessing 2nd through 5th elements:
 [87 24 56 38]
Accessing the elements before the 6th element:
 [26 87 24 56 38]
Accessing the elements from the 3rd element:
 [24 56 38 45 94 68]
Accessing the elements after the 3rd element:
 [56 38 45 94 68]
```

We can also use the comparison operators in the NumPy array to get the Boolean array. In *array_01*, if we want to select the elements that're less than 30, we can do it in the following way.

Code:

list_01 = [26, 87, 24, 56, 38, 45, 94, 68]

array_01 = np.array(list_01)

print("array_01: \n", array_01) #array_01

print("Accessing the elements that're less than 30: \n", array_01[array_01 < 30])

Output:

array_01:

[26 87 24 56 38 45 94 68]

Accessing the elements that're less than 30:

[26 24]

```
list_01 = [26, 87, 24, 56, 38, 45, 94, 68]
array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01
print("Accessing the elements that're less than 30: \n", array_01[array_01 < 30])
```

```
array_01:
 [26 87 24 56 38 45 94 68]
Accessing the elements that're less than 30:
 [26 24]
```

In *array_01*, if we want to select the elements that’re greater than 40, we can do it in the following way.

Code:

list_01 = [26, 87, 24, 56, 38, 45, 94, 68]

array_01 = np.array(list_01)

print("array_01: \n", array_01) #array_01

print("Accessing the elements that're greater than 40: \n", array_01[array_01 > 40])

Output:

array_01:

[26 87 24 56 38 45 94 68]

Accessing the elements that're greater than 40:

[87 56 45 94 68]

```
list_01 = [26, 87, 24, 56, 38, 45, 94, 68]
array_01 = np.array(list_01)

print("array_01: \n", array_01)  #array_01
print("Accessing the elements that're greater than 40: \n", array_01[array_01 > 40])
```

```
array_01:
 [26 87 24 56 38 45 94 68]
Accessing the elements that're greater than 40:
 [87 56 45 94 68]
```

UNIT 03: PANDAS

INTRODUCTION TO PANDAS

"*Pandas*" is Python's version of Excel. Pandas is an open-source library that can be used for working with data sets. We can use Pandas to analyze the data really fast, as well as cleaning, exploring, and manipulating the data. We can also do data visualization with Pandas.

We can install Pandas in our system by running the following command in the terminal.

pip install pandas

Once Pandas is installed, we can import it in the following way.

import pandas as pd

```
import pandas as pd
```

In this book, we'll work with the Pandas series, dataframes, various operations as well as the data input and output.

PANDAS SERIES

The Pandas Series is very similar to the NumPy arrays, but in Pandas we can specify both the data and the index.

Suppose we've the following two Python lists.

temperature = [32, 67, 18, 90, 48, 83]

day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

"*temperature*" is a list containing the list of temperatures, and the "*day*" list contains the name of the day of the recorded temperature. The Pandas Series is similar to the column in a table. It is a 1-D array that can have any type of data.

Here, we can consider the "temperature" list as the data and the "day" list as the index. With the index arguments, we can create our own specific labels. If the index isn't specified, the values in the data get the index automatically. The first value is index 0, the second value is index 1 and so on.

Code:

temperature = [32, 67, 18, 90, 48, 83]

day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

pd.Series(data = temperature, index = day)

Output:

day_01 32

day_02 67

day_03 18

day_04 90

day_05 48

day_06 83

dtype: int64

```
temperature = [32, 67, 18, 90, 48, 83]
day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

pd.Series(data = temperature, index = day)
```

```
day_01    32
day_02    67
day_03    18
day_04    90
day_05    48
day_06    83
dtype: int64
```

Here, we can see that the data type is integer.

Now, we can access the data by just referring to the labels. If we want to know the temperature on “day_03”, we can do it in the following way.

Code:

```
temperature = [32, 67, 18, 90, 48, 83]
day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

temperature_data = pd.Series(data = temperature, index = day)
print("Temperature on day_03 is: \n", temperature_data["day_03"])
```

Output:

```
Temperature on day_03 is:
18
```

```
temperature = [32, 67, 18, 90, 48, 83]
day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

temperature_data = pd.Series(data = temperature, index = day)
print("Temperature on day_03 is: \n", temperature_data["day_03"])
```

```
Temperature on day_03 is:
 18
```

Another quick way to create the series is just by mentioning the data and index directly. The correct order is that the data comes first, then comes the index. So, it's fine as long as the correct order is maintained.

Code:

temperature = [32, 67, 18, 90, 48, 83]

day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

temperature_data = pd.Series(temperature, day)

temperature_data

Output:

day_01 32
day_02 67
day_03 18
day_04 90
day_05 48
day_06 83
dtype: int64

```
temperature = [32, 67, 18, 90, 48, 83]
day = ["day_01", "day_02", "day_03", "day_04", "day_05", "day_06"]

temperature_data = pd.Series(temperature, day)
temperature_data
```

```
day_01    32
day_02    67
day_03    18
day_04    90
day_05    48
day_06    83
dtype: int64
```

We can also pass a NumPy array in the Pandas series.

Code:

```
import numpy as np
import pandas as pd

temperature = [32, 67, 18, 90, 48, 83]
temperature_array = np.array(temperature)

temperature_data = pd.Series(temperature_array)
temperature_data
```

Output:

```
0    32
1    67
2    18
3    90
4    48
5    83
```

dtype: int32

```
import numpy as np
import pandas as pd

temperature = [32, 67, 18, 90, 48, 83]
temperature_array = np.array(temperature)

temperature_data = pd.Series(temperature_array)
temperature_data
```

```
0    32
1    67
2    18
3    90
4    48
5    83
dtype: int32
```

Here, since we didn't mention the index, the index gets generated automatically. The first datapoint 32 is at index 0, the second datapoint 67 is at index 1 and so on.

We can also pass a dictionary in the Pandas series. The series will automatically take the dictionary key as the index and set the dictionary value as the data.

Code:

temperature_dictionary = {"day_01": 32, "day_02": 67, "day_03": 18, "day_04": 90, "day_05": 48, "day_06": 83}

pd.Series(temperature_dictionary)

Output:

day_01 32

day_02 67

day_03 18

day_04 90
day_05 48
day_06 83
dtype: int64

```
temperature_dictionary = {"day_01": 32, "day_02": 67, "day_03": 18, "day_04": 90, "day_05": 48, "day_06": 83}
pd.Series(temperature_dictionary)
```

```
day_01    32
day_02    67
day_03    18
day_04    90
day_05    48
day_06    83
dtype: int64
```

We can also add the series. Suppose we've the following three lists.

teams = ["Liverpool", "Chelsea", "Arsenal", "Manchester United"]

trophies_premier_league = [19, 6, 13, 20]

trophies_champions_league = [6, 2, 0, 3]

Now, we can make two series, one for the teams with Premier League trophies and another list for the teams with the Champions League trophies.

Code:

```
teams = ["Liverpool", "Chelsea", "Arsenal", "Manchester United"]
trophies_premier_league = [19, 6, 13, 20]
trophies_champions_league = [6, 2, 0, 3]

series_premier_league = pd.Series(trophies_premier_league, teams)
series_champions_league = pd.Series(trophies_champions_league, teams)
```

```
print(series_premier_league)
print("----------------------------------")
print(series_champions_league)
```

Output:

```
Liverpool          19
Chelsea             6
Arsenal            13
Manchester United    20
dtype: int64
----------------------------------
Liverpool          6
Chelsea            2
Arsenal            0
Manchester United    3
dtype: int64
```

```
teams = ["Liverpool", "Chelsea", "Arsenal", "Manchester United"]
trophies_premier_league = [19, 6, 13, 20]
trophies_champions_league = [6, 2, 0, 3]

series_premier_league = pd.Series(trophies_premier_league, teams)
series_champions_league = pd.Series(trophies_champions_league, teams)

print(series_premier_league)
print("-----------------------------------")
print(series_champions_league)
```

```
Liverpool            19
Chelsea               6
Arsenal              13
Manchester United    20
dtype: int64
-----------------------------------
Liverpool            6
Chelsea              2
Arsenal              0
Manchester United    3
dtype: int64
```

If we want to add the two series together, we can do it in the following way.

Code:

series_total = series_premier_league + series_champions_league

print(series_total)

Output:

Liverpool 25

Chelsea 8

Arsenal 13

Manchester United 23

dtype: int64

```
series_total = series_premier_league + series_champions_league
print(series_total)
```

```
Liverpool            25
Chelsea               8
Arsenal              13
Manchester United    23
dtype: int64
```

Here, we can see the series automatically detected the index as the team names and added the datapoints from the two lists together. As a result, we can see the total number of trophies from each team.

PANDAS DATAFRAME

Pandas DataFrame is a 2-D data structure. It contains a table with rows and columns. The columns can contain different types of data.

Suppose we want to build a DataFrame containing the teams and some information about each team. First, we need to create a dictionary and pass that dictionary into the DataFrame.

Code:

```
team_names = ["Liverpool", "Chelsea", "Arsenal", "Manchester United"]
team_data_dictionary = {"Head Coach": ["Jurgen Klopp", "Graham Potter", "Mikel Arteta", "Erik ten Hag"],
        "Star Player": ["Mohamed Salah", "Mason Mount", "Granit Xhaka", "Harry Maguire"],
        "Annual Spendings": [77220000, 251090000, 118860000, 214220000],
        "Premier League Trophies": [19, 6, 13, 20]}

team_data_dataframe = pd.DataFrame(team_data_dictionary, index = team_names, columns = ["Head Coach", "Star Player", "Annual Spendings", "Premier League Trophies"])
team_data_dataframe
```

Output:

	Head Coach	Star Player	Annual Spendings	Premier League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19
Chelsea	Graham Potter	Mason Mount	251090000	6
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13
Manchester United	Erik ten Hag	Harry Maguire	214220000	20

```
team_names = ["Liverpool", "Chelsea", "Arsenal", "Manchester United"]
team_data_dictionary = {"Head Coach": ["Jurgen Klopp", "Graham Potter", "Mikel Arteta", "Erik ten Hag"],
            "Star Player": ["Mohamed Salah", "Mason Mount", "Granit Xhaka", "Harry Maguire"],
            "Annual Spendings": [77220000, 251090000, 118860000, 214220000],
            "Premier League Trophies": [19, 6, 13, 20]}

team_data_dataframe = pd.DataFrame(team_data_dictionary, index = team_names, columns = ["Head Coach", "Star Player", "Annual Spendings", "Premier League Trophies"])
team_data_dataframe
```

	Head Coach	Star Player	Annual Spendings	Premier League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19
Chelsea	Graham Potter	Mason Mount	251090000	6
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13
Manchester United	Erik ten Hag	Harry Maguire	214220000	20

Here, the team_names is the index, and the team_data_dictionary is the data in the Pandas DataFrame. Each column is a Pandas series, and all the series share a common index.

We can sort and rank the data in the DataFrame. Suppose if we want to sort the DataFrame by the number of Premier League trophies (from highest to lowest), we can do it in the following way.

Code:

team_data_dataframe.sort_values(by = "Premier League Trophies", ascending = False)

Output:

	Head Coach	Star Player	Annual Spendings	Premier League Trophies
Manchester United	Erik ten Hag	Harry Maguire	214220000	20
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13
Chelsea	Graham Potter	Mason Mount	251090000	6

```
team_data_dataframe.sort_values(by = "Premier League Trophies", ascending = False)
```

	Head Coach	Star Player	Annual Spendings	Premier League Trophies
Manchester United	Erik ten Hag	Harry Maguire	214220000	20
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13
Chelsea	Graham Potter	Mason Mount	251090000	6

Now, if we want to know which team does the most annual spendings, we can use the *rank()* function.

Code:

team_data_dataframe["Annual Spendings"].rank(ascending = False).sort_values()

Output:

Chelsea 1.0

Manchester United 2.0

Arsenal 3.0

Liverpool 4.0

Name: Annual Spendings, dtype: float64

```
team_data_dataframe["Annual Spendings"].rank(ascending = False).sort_values()
```

```
Chelsea              1.0
Manchester United    2.0
Arsenal              3.0
Liverpool            4.0
Name: Annual Spendings, dtype: float64
```

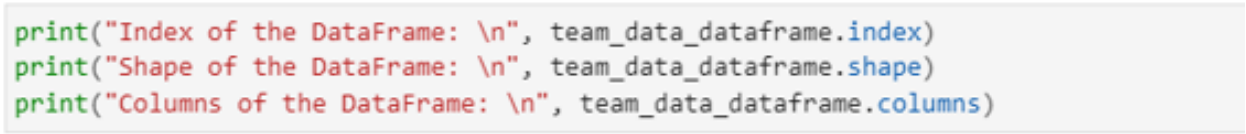
If we want to find the shape, index and columns of a DataFrame, we can do it in the following way.

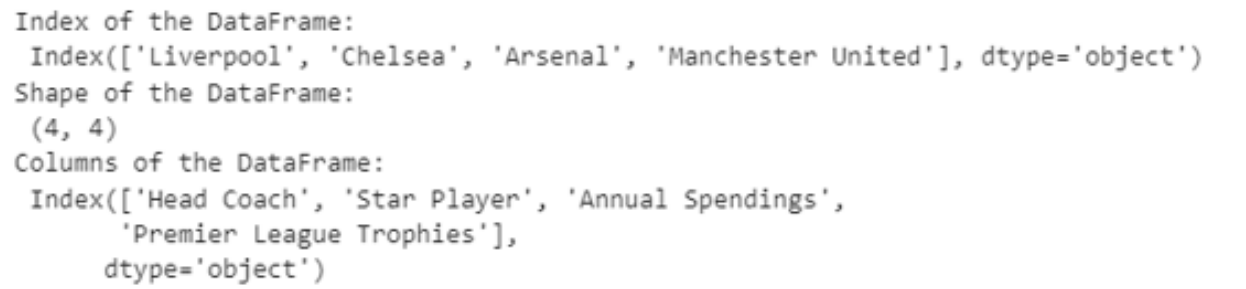
Code:

```
print("Index of the DataFrame: \n", team_data_dataframe.index)
print("Shape of the DataFrame: \n", team_data_dataframe.shape)
print("Columns of the DataFrame: \n",
team_data_dataframe.columns)
```

Output:

```
Index of the DataFrame:
Index(['Liverpool', 'Chelsea', 'Arsenal', 'Manchester United'],
dtype='object')
Shape of the DataFrame:
(4, 4)
Columns of the DataFrame:
Index(['Head Coach', 'Star Player', 'Annual Spendings',
       'Premier League Trophies'],
      dtype='object')
```





If we want to find all the data about Liverpool, we can use the *loc()* function.

Code:

team_data_dataframe.loc["Liverpool"]

Output:

Head Coach Jurgen Klopp
Star Player Mohamed Salah
Annual Spendings 77220000
Premier League Trophies 19
Name: Liverpool, dtype: object

```
team_data_dataframe.loc["Liverpool"]
```

```
Head Coach                  Jurgen Klopp
Star Player                Mohamed Salah
Annual Spendings                77220000
Premier League Trophies               19
Name: Liverpool, dtype: object
```

Here, “Liverpool” is acting as the index.

Suppose we want to find the “Star Player” in “Liverpool”. We can do it in the following way.

Code:

team_data_dataframe.loc["Liverpool"]["Star Player"]

Output:

'Mohamed Salah'

```
team_data_dataframe.loc["Liverpool"]["Star Player"]
```

```
'Mohamed Salah'
```

Here, “Liverpool” is acting as the index, and “Star Player” is acting as the datapoint.

If we want to return any specific column, we can do it in the following way.

Code:

team_data_dataframe["Head Coach"]

Output:

Liverpool Jurgen Klopp
Chelsea Graham Potter
Arsenal Mikel Arteta
Manchester United Erik ten Hag
Name: Head Coach, dtype: object

```
team_data_dataframe["Head Coach"]
```

```
Liverpool            Jurgen Klopp
Chelsea             Graham Potter
Arsenal              Mikel Arteta
Manchester United    Erik ten Hag
Name: Head Coach, dtype: object
```

Here, we’re returning only the “Head Coach” column.

If we want to select multiple columns from the DataFrame, we can do it in the following way.

Code:

team_data_dataframe[["Head Coach", "Star Player"]]

Output:

	Head Coach	Star Player
Liverpool	Jurgen Klopp	Mohamed Salah
Chelsea	Graham Potter	Mason Mount
Arsenal	Mikel Arteta	Granit Xhaka
Manchester United	Erik ten Hag	Harry Maguire

```
team_data_dataframe[["Head Coach", "Star Player"]]
```

	Head Coach	Star Player
Liverpool	Jurgen Klopp	Mohamed Salah
Chelsea	Graham Potter	Mason Mount
Arsenal	Mikel Arteta	Granit Xhaka
Manchester United	Erik ten Hag	Harry Maguire

Here, we're returning the "Head Coach" and the "Star Player" columns.

We can add new columns in the DataFrame. To do this, we can simply add the new column thinking that it already exists, then pass on to the values. Suppose we want to add another column "Champions League Trophies", we can do it in the following way.

Code:

team_data_dataframe["Champions League Trophies"] = [6, 2, 0, 3]

team_data_dataframe

Output:

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6
Chelsea	Graham Potter	Mason Mount	251090000	6	2
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13	0
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3

```
team_data_dataframe["Champions League Trophies"] = [6, 2, 0, 3]

team_data_dataframe
```

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6
Chelsea	Graham Potter	Mason Mount	251090000	6	2
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13	0
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3

We can also create a new column that's the sum of other columns. Suppose we want to create another column "Total Trophies" which is the sum of the columns "Premier League Trophies" and "Champions League Trophies".

Code:

team_data_dataframe["Total Trophies"] = team_data_dataframe["Premier League Trophies"] + team_data_dataframe["Champions League Trophies"]

team_data_dataframe

Output:

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies	Total Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6	25
Chelsea	Graham Potter	Mason Mount	251090000	6	2	8
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13	0	13
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3	23

```
team_data_dataframe["Total Trophies"] = team_data_dataframe["Premier League Trophies"] + team_data_dataframe["Champions League Trophies"]

team_data_dataframe
```

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies	Total Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6	25
Chelsea	Graham Potter	Mason Mount	251090000	6	2	8
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13	0	13
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3	23

If we want to delete a column, we can use the *drop()* function. Suppose we want to delete the newly added “Total Trophies” column. We can do it in the following way.

Code:

team_data_dataframe.drop("Total Trophies", axis = 1)

Output:

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6
Chelsea	Graham Potter	Mason Mount	251090000	6	2
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13	0
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3

```
team_data_dataframe.drop("Total Trophies", axis = 1)
```

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6
Chelsea	Graham Potter	Mason Mount	251090000	6	2
Arsenal	Mikel Arteta	Granit Xhaka	118860000	13	0
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3

Here, the "*axis = 1*" refers to the columns, and "*axis = 0*" will refer to the rows.

We can also filter the Pandas DataFrame and apply different conditions. Suppose we want to know which teams have more than 14 Premier League trophies.

Code:

team_data_dataframe["Premier League Trophies"] > 14

Output:

Liverpool True
Chelsea False
Arsenal False
Manchester United True
Name: Premier League Trophies, dtype: bool

```
team_data_dataframe["Premier League Trophies"] > 14
```

```
Liverpool             True
Chelsea              False
Arsenal              False
Manchester United     True
Name: Premier League Trophies, dtype: bool
```

Here, we can see that Liverpool and Manchester United both have more than 14 Premier League trophies.

Now if we want to return all the data about the teams that have more than 14 Premier League trophies, we can do it in the following way.

Code:

team_data_dataframe[team_data_dataframe["Premier League Trophies"] > 14]

Output:

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3

```
team_data_dataframe[team_data_dataframe["Premier League Trophies"] > 14]
```

	Head Coach	Star Player	Annual Spendings	Premier League Trophies	Champions League Trophies
Liverpool	Jurgen Klopp	Mohamed Salah	77220000	19	6
Manchester United	Erik ten Hag	Harry Maguire	214220000	20	3

We can also add more steps to the conditions in the DataFrame. Suppose we want to know the names of the star players of each team that has more than 14 Premier League trophies. We can do it in the following way.

Code:

team_data_dataframe[team_data_dataframe["Premier League Trophies"] > 14]["Star Player"]

Output:

Liverpool Mohamed Salah

Manchester United Harry Maguire

Name: Star Player, dtype: object

```
team_data_dataframe[team_data_dataframe["Premier League Trophies"] > 14]["Star Player"]
```

```
Liverpool            Mohamed Salah
Manchester United    Harry Maguire
Name: Star Player, dtype: object
```

We can add multiple conditions to the DataFrame. Suppose we want to know which team has more than 14 Premier League trophies and more than 4 Champions League trophies.

Code:

(team_data_dataframe["Premier League Trophies"] > 14) & (team_data_dataframe["Champions League Trophies"] > 4)

Output:

Liverpool True

Chelsea False

Arsenal False

Manchester United False

dtype: bool

```
(team_data_dataframe["Premier League Trophies"] > 14) & (team_data_dataframe["Champions League Trophies"] > 4)
```

```
Liverpool             True
Chelsea              False
Arsenal              False
Manchester United    False
dtype: bool
```

Here, we used the “&” symbol which means that, the output will return True if both of the conditions are satisfied.

Suppose we want to know which team either has more than 14 Premier League trophies or more than 4 Champions League trophies.

Code:

(team_data_dataframe["Premier League Trophies"] > 14) | (team_data_dataframe["Champions League Trophies"] > 4)

Output:

Liverpool True
Chelsea False
Arsenal False
Manchester United True
dtype: bool

```
(team_data_dataframe["Premier League Trophies"] > 14) | (team_data_dataframe["Champions League Trophies"] > 4)
```

```
Liverpool            True
Chelsea             False
Arsenal             False
Manchester United    True
dtype: bool
```

Here, we used the “|” symbol which means that , the output will return True if either of the conditions are satisfied.

PANDAS OPERATIONS

We can do various mathematical and statistical operations in Pandas.

Suppose we want to build a DataFrame containing the months and the income of persons in each month. First, we need to create a dictionary and pass that dictionary into the DataFrame.

Code:

```
month_names = ["January", "February", "Month", "April", "May", "June"]
income = {"Person A": [658, 243, 982, 473, 385, 864],
        "Person B": [126, 927, 532, 763, 352, 984],
        "Person C": [327, 736, 635, 873, 673, 523],
        "Person D": [128, 476, 897, 253, 874, 693]}

monthly_income_persons = pd.DataFrame(income, index = month_names, columns = income)

monthly_income_persons
```

Output:

	Person A	Person B	Person C	Person D
January	658	126	327	128
February	243	927	736	476
Month	982	532	635	897
April	473	763	873	253
May	385	352	673	874
June	864	984	523	693

```
month_names = ["January", "February", "Month", "April", "May", "June"]
income = {"Person A": [658, 243, 982, 473, 385, 864],
          "Person B": [126, 927, 532, 763, 352, 984],
          "Person C": [327, 736, 635, 873, 673, 523],
          "Person D": [128, 476, 897, 253, 874, 693]}

monthly_income_persons = pd.DataFrame(income, index = month_names, columns = income)

monthly_income_persons
```

	Person A	Person B	Person C	Person D
January	658	126	327	128
February	243	927	736	476
Month	982	532	635	897
April	473	763	873	253
May	385	352	673	874
June	864	984	523	693

If we want to know how much each person made from January through June, we need to use the sum() function.

Code:

monthly_income_persons.sum(axis = 0)

Output:

Person A 3605
Person B 3684
Person C 3767
Person D 3321
dtype: int64

```
monthly_income_persons.sum(axis = 0)
```

```
Person A    3605
Person B    3684
Person C    3767
Person D    3321
dtype: int64
```

Since we're calculating for each column, the "*axis = 0*". By default, the axis is always 0.

Now, if we want to know how much was made by all the persons in each month, we can do it in the following way.

Code:

monthly_income_persons.sum(axis = 1)

Output:

January 1239
February 2382
Month 3046
April 2362
May 2284
June 3064
dtype: int64

```
monthly_income_persons.sum(axis = 1)
```

```
January     1239
February    2382
Month       3046
April       2362
May         2284
June        3064
dtype: int64
```

Since we're calculating for each row, the "*axis = 1*".

If we want to know the mean income of each person, we can do it in the following way.

Code:

monthly_income_persons.mean()

Output:

Person A 600.833333

Person B 614.000000

Person C 627.833333

Person D 553.500000

dtype: float64

```
monthly_income_persons.mean()
```

```
Person A    600.833333
Person B    614.000000
Person C    627.833333
Person D    553.500000
dtype: float64
```

If we want to know the median income of each person, we can do it in the following way.

Code:

monthly_income_persons.median()

Output:

Person A 565.5

Person B 647.5

Person C 654.0

Person D 584.5

dtype: float64

```
monthly_income_persons.median()
```

```
Person A    565.5
Person B    647.5
Person C    654.0
Person D    584.5
dtype: float64
```

If we want to find the minimum and the maximum monthly income of each person, we can do it in the following way.

Code:

```
print(monthly_income_persons.min())
print("----------------------------")
print(monthly_income_persons.max())
```

Output:

Person A 243

Person B 126

Person C 327

Person D 128

dtype: int64

Person A 982

Person B 984

Person C 873

Person D 897

dtype: int64

```
print(monthly_income_persons.min())
print("-----------------------------")
print(monthly_income_persons.max())
```

```
Person A    243
Person B    126
Person C    327
Person D    128
dtype: int64
-----------------------------
Person A    982
Person B    984
Person C    873
Person D    897
dtype: int64
```

If we want to find the summary statistics of the DataFrame, we can do it in the following way.

Code:

monthly_income_persons.describe()

Output:

	Person A	Person B	Person C	Person D
count	6.000000	6.00000	6.000000	6.000000
mean	600.833333	614.00000	627.833333	553.500000
std	285.931052	337.81119	187.290594	321.723328
min	243.000000	126.00000	327.000000	128.000000
25%	407.000000	397.00000	551.000000	308.750000
50%	565.500000	647.50000	654.000000	584.500000
75%	812.500000	886.00000	720.250000	828.750000
max	982.000000	984.00000	873.000000	897.000000

```
monthly_income_persons.describe()
```

	Person A	Person B	Person C	Person D
count	6.000000	6.00000	6.000000	6.000000
mean	600.833333	614.00000	627.833333	553.500000
std	285.931052	337.81119	187.290594	321.723328
min	243.000000	126.00000	327.000000	128.000000
25%	407.000000	397.00000	551.000000	308.750000
50%	565.500000	647.50000	654.000000	584.500000
75%	812.500000	886.00000	720.250000	828.750000
max	982.000000	984.00000	873.000000	897.000000

We can also apply functions to each element in the DataFrame. Suppose we want to multiply the income of “Person A” by 5. We can do it in the following way.

Code:

def multiply_by_5(x):

return x * 5

monthly_income_persons["Person A"].apply(multiply_by_5)

Output:

January 3290
February 1215
Month 4910
April 2365
May 1925
June 4320
Name: Person A, dtype: int64

```
def multiply_by_5(x):
    return x * 5

monthly_income_persons["Person A"].apply(multiply_by_5)
```

```
January     3290
February    1215
Month       4910
April       2365
May         1925
June        4320
Name: Person A, dtype: int64
```

We can also elementwise apply the function to all the columns simultaneously.

Code:

monthly_income_persons.applymap(multiply_by_5)

Output:

	Person A	Person B	Person C	Person D
January	3290	630	1635	640
February	1215	4635	3680	2380
Month	4910	2660	3175	4485
April	2365	3815	4365	1265
May	1925	1760	3365	4370
June	4320	4920	2615	3465

```
monthly_income_persons.applymap(multiply_by_5)
```

	Person A	Person B	Person C	Person D
January	3290	630	1635	640
February	1215	4635	3680	2380
Month	4910	2660	3175	4485
April	2365	3815	4365	1265
May	1925	1760	3365	4370
June	4320	4920	2615	3465

We can also apply the *lambda* functions directly to the elements.

Code:

monthly_income_persons.applymap(lambda x: x * 5)

Output:

	Person A	Person B	Person C	Person D
January	3290	630	1635	640
February	1215	4635	3680	2380
Month	4910	2660	3175	4485
April	2365	3815	4365	1265
May	1925	1760	3365	4370
June	4320	4920	2615	3465

```
monthly_income_persons.applymap(lambda x: x * 5)
```

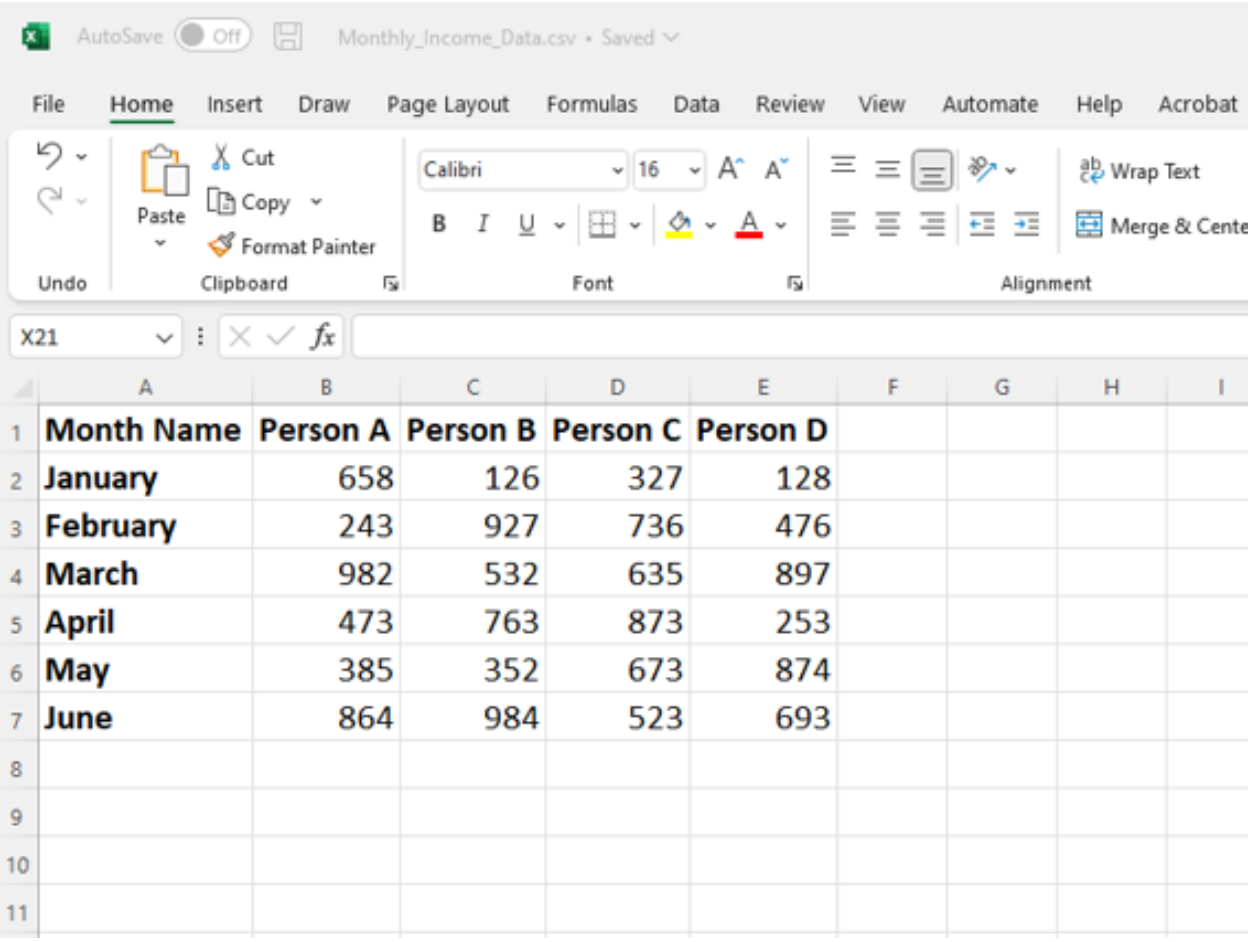
	Person A	Person B	Person C	Person D
January	3290	630	1635	640
February	1215	4635	3680	2380
Month	4910	2660	3175	4485
April	2365	3815	4365	1265
May	1925	1760	3365	4370
June	4320	4920	2615	3465

IMPORTING AND EXPORTING DATA IN PANDAS

We can use Pandas to read from or write to external data sources. If we want to read/write from a CSV or Excel file, we have to make sure that file exists in the same directory as the JupyterLab notebook we're working on.

Suppose we've a CSV file with the following data.

Now, we've "*Monthly_Income_Data.csv*" CSV file in the same directory.



	A	B	C	D	E	F	G	H	I
1	**Month Name**	**Person A**	**Person B**	**Person C**	**Person D**				
2	**January**	658	126	327	128				
3	**February**	243	927	736	476				
4	**March**	982	532	635	897				
5	**April**	473	763	873	253				
6	**May**	385	352	673	874				
7	**June**	864	984	523	693				
8									
9									
10									
11									

If we want to open and read data from this CSV file, we can do it in the following way.

Code:

pd.read_csv("Monthly_Income_Data.csv")

Output:

	Month Name	Person A	Person B	Person C	Person D
0	January	658	126	327	128
1	February	243	927	736	476
2	March	982	532	635	897
3	April	473	763	873	253
4	May	385	352	673	874
5	June	864	984	523	693

```
pd.read_csv("Monthly_Income_Data.csv")
```

	Month Name	Person A	Person B	Person C	Person D
0	January	658	126	327	128
1	February	243	927	736	476
2	March	982	532	635	897
3	April	473	763	873	253
4	May	385	352	673	874
5	June	864	984	523	693

Suppose we want to add a new column “Person E” and write this data to the CSV file. We can do it in the following way.

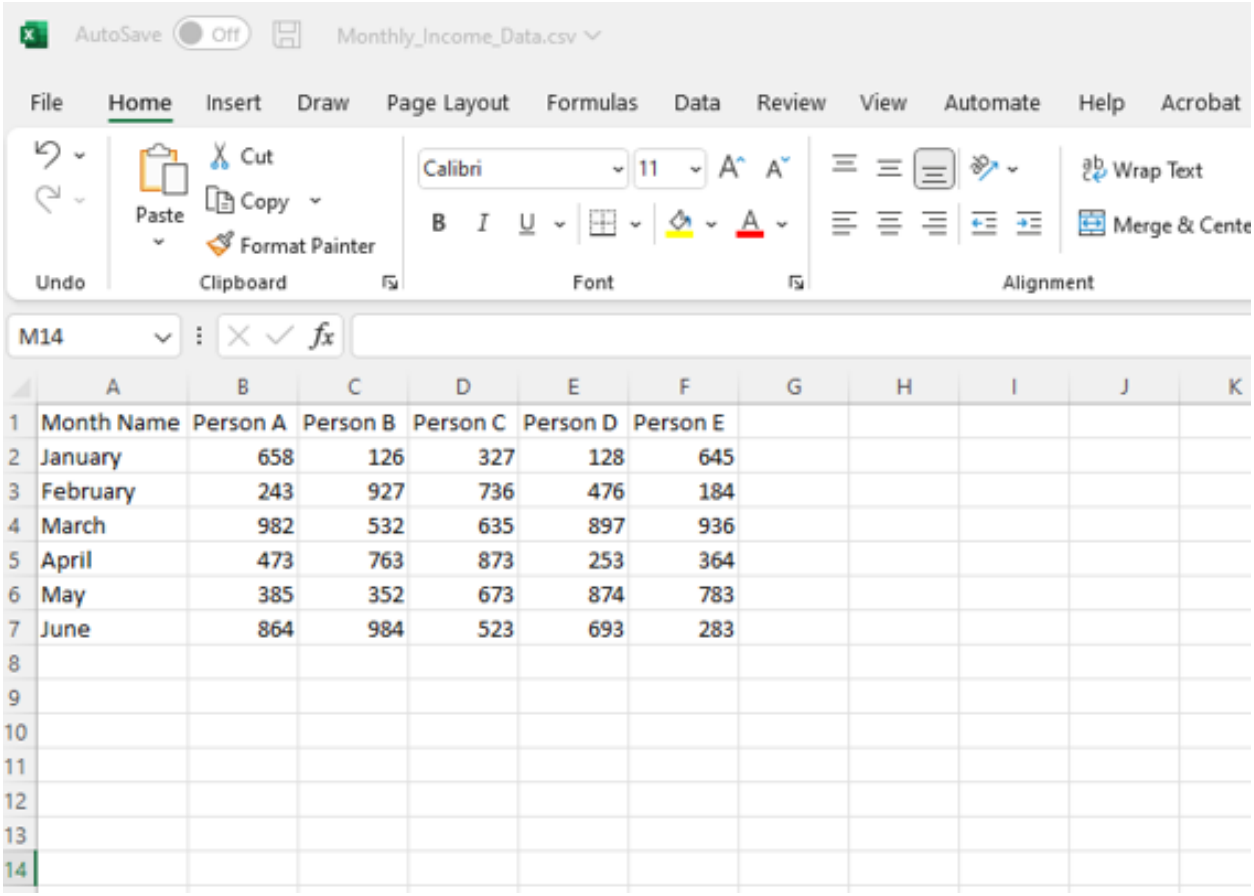
Code:

pd.read_csv("Monthly_Income_Data.csv")
df = pd.read_csv("Monthly_Income_Data.csv")

df["Person E"] = [645, 184, 936, 364, 783, 283]
df.to_csv("Monthly_Income_Data.csv", index = False)

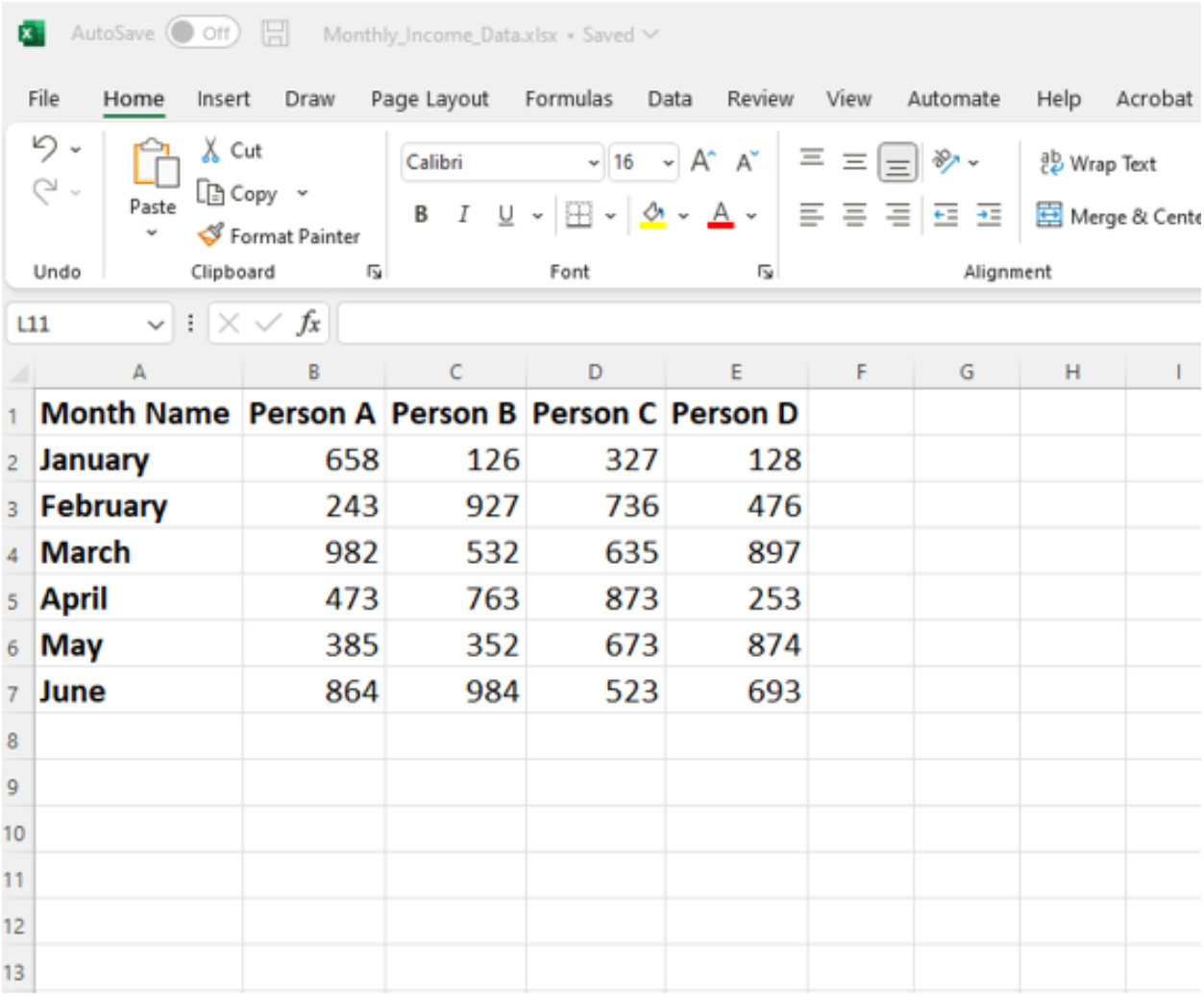
```
pd.read_csv("Monthly_Income_Data.csv")
df = pd.read_csv("Monthly_Income_Data.csv")

df["Person E"] = [645, 184, 936, 364, 783, 283]
df.to_csv("Monthly_Income_Data.csv", index = False)
```

We converted the imported CSV into a DataFrame, then added a new column. When writing to the CSV file, "*index = False*" means that it'll not add any extra index columns into the CSV file.



	A	B	C	D	E	F	G	H	I	J	K
1	Month Name	Person A	Person B	Person C	Person D	Person E					
2	January	658	126	327	128	645					
3	February	243	927	736	476	184					
4	March	982	532	635	897	936					
5	April	473	763	873	253	364					
6	May	385	352	673	874	783					
7	June	864	984	523	693	283					
8											
9											
10											
11											
12											
13											
14											

We can also use Pandas to open and read from the Excel files. Now, we've "*Monthly_Income_Data.xlsx*" Excel file in the same directory.



	A	B	C	D	E	F	G	H	I
1	**Month Name**	**Person A**	**Person B**	**Person C**	**Person D**				
2	**January**	658	126	327	128				
3	**February**	243	927	736	476				
4	**March**	982	532	635	897				
5	**April**	473	763	873	253				
6	**May**	385	352	673	874				
7	**June**	864	984	523	693				
8									
9									
10									
11									
12									
13									

Code:

pd.read_excel("Monthly_Income_Data.xlsx")

Output:

	Month Name	Person A	Person B	Person C	Person D
0	January	658	126	327	128
1	February	243	927	736	476
2	March	982	532	635	897
3	April	473	763	873	253
4	May	385	352	673	874
5	June	864	984	523	693

```
pd.read_excel("Monthly_Income_Data.xlsx")
```

	Month Name	Person A	Person B	Person C	Person D
0	January	658	126	327	128
1	February	243	927	736	476
2	March	982	532	635	897
3	April	473	763	873	253
4	May	385	352	673	874
5	June	864	984	523	693

Suppose we want to add a new column “Person E” and write this data to the Excel file. We can do it in the following way.

Code:

pd.read_excel("Monthly_Income_Data.xlsx")

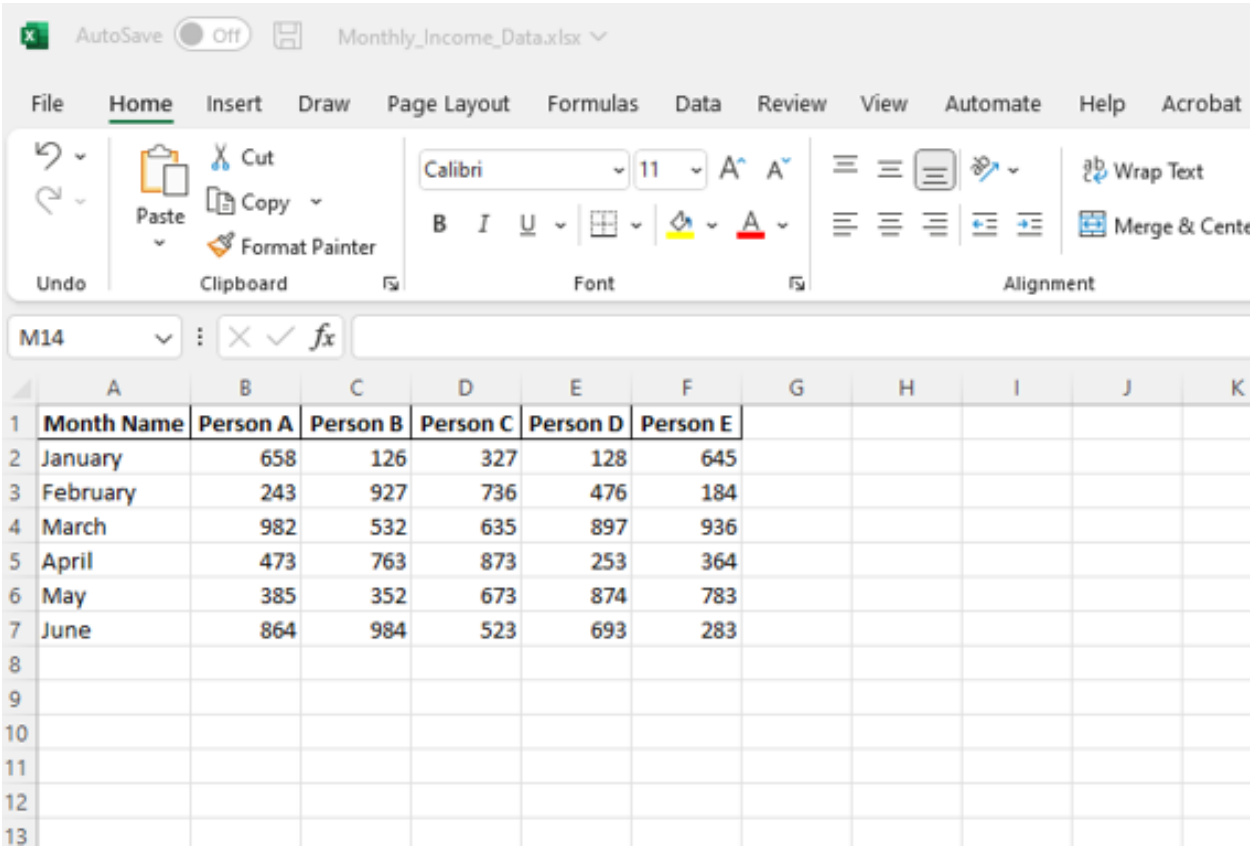
df = pd.read_excel("Monthly_Income_Data.xlsx")

df["Person E"] = [645, 184, 936, 364, 783, 283]

df.to_excel("Monthly_Income_Data.xlsx", index = False)

```
pd.read_excel("Monthly_Income_Data.xlsx")
df = pd.read_excel("Monthly_Income_Data.xlsx")

df["Person E"] = [645, 184, 936, 364, 783, 283]
df.to_excel("Monthly_Income_Data.xlsx", index = False)
```

We converted the imported Excel into a DataFrame, then added a new column. When writing to the Excel file, “index = False” means that it’ll not add any extra index columns into the Excel file.



	A	B	C	D	E	F	G	H	I	J	K
1	**Month Name**	**Person A**	**Person B**	**Person C**	**Person D**	**Person E**					
2	January	658	126	327	128	645					
3	February	243	927	736	476	184					
4	March	982	532	635	897	936					
5	April	473	763	873	253	364					
6	May	385	352	673	874	783					
7	June	864	984	523	693	283					
8											
9											
10											
11											
12											
13											

UNIT 04: MATPLOTLIB

INTRODUCTION TO MATPLOTLIB

"*Matplotlib*" is one of the most popular open-source libraries in Python for creating plots and data visualizations. It is a massive, powerful library for creating all sorts of visualizations.

We can install *Matplotlib* in our system by running the following command in the terminal.

pip install matplotlib

Once Matplotlib is installed, we can import it in the following way.

import matplotlib.pyplot as plt

Note that, most of the Matplotlib functionalities fall under the pyplot submodule.

In this book, we'll work with the different Matplotlib plotting basics, markers, lines, labels, grids, subplots, as well as bar graphs, scatter plots, histograms and the pie charts.

PLOTS IN MATPLOTLIB

Suppose we want to plot the following equation in Matplotlib.

Here, x has the following values (5, 26, 48, 65, 72). First, we need to convert the list to a NumPy array, then use the following two functions to plot it.

plt.plot()

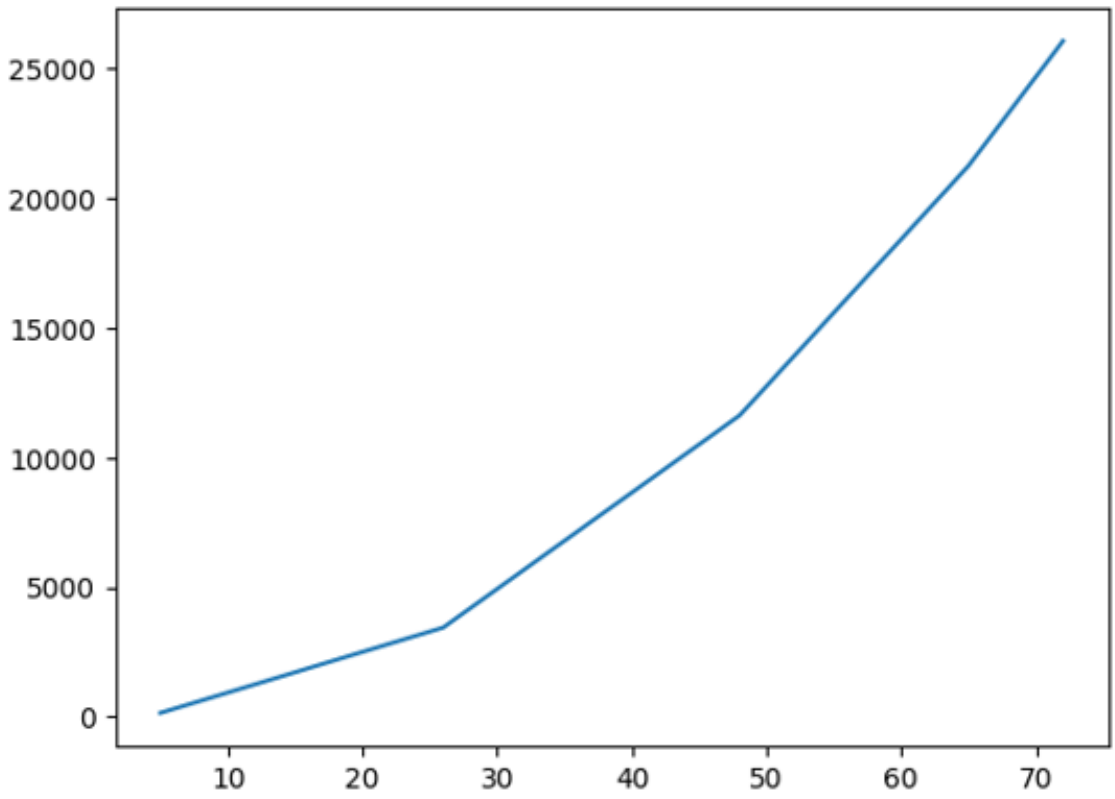
plt.show()

The *plot()* function is used for drawing the points in the diagram. This takes the parameters for indicating to the points in the diagram. The first parameter is for the x-axis, and the second parameter is for the y-axis.

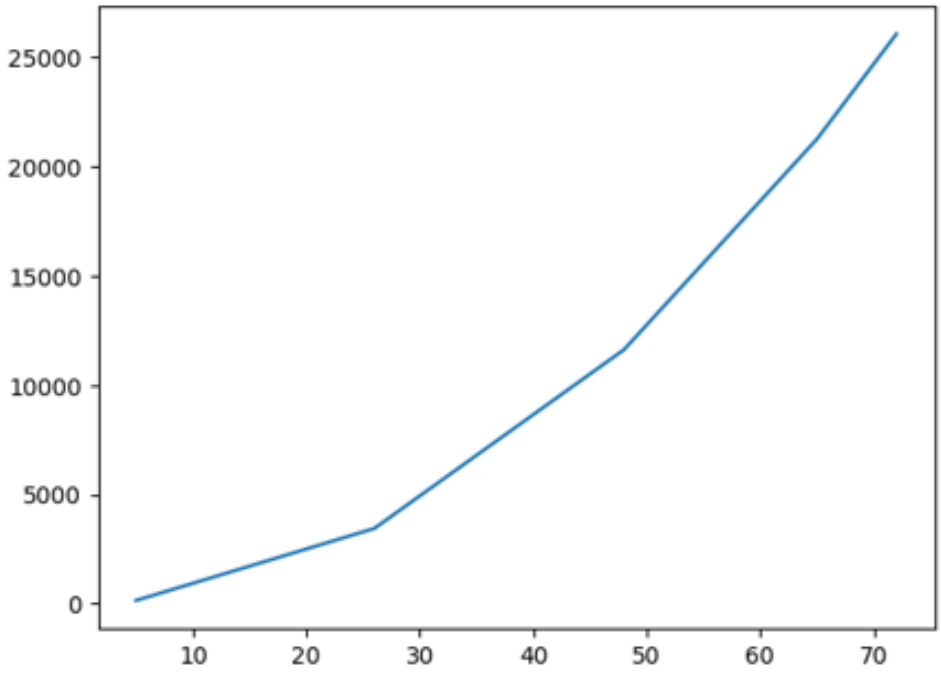
Code:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([5, 26, 48, 65, 72])  #x values
y = (5 * x ** 2) + (2 * x) + 7  #y values

plt.plot(x, y)
plt.show()
```

Output:


25000
20000
15000
10000
5000
0
10
20
30
40
50
60
70

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.array([5, 26, 48, 65, 72])  #x values
y = (5 * x ** 2) + (2 * x) + 7  #y values

plt.plot(x, y)
plt.show()
```



MATPLOTLIB MARKERS, LINES, GRIDS AND LABELS

We can use the "*marker*" keyword argument to specify each point.

Some of the popular markers are the following.

"o" **circle**

"s" **square**

"p" **pentagon**

"*" **star**

We can use the "*linestyle*" or "*ls*" keyword argument to change the plotted line style.

Some of the popular linestyles are the following.

"solid" **solid**

"dotted" **dotted**

"dashed" **dashed**

"dashdot" **dashdot**

We can use the "*color*" keyword argument to change the color of the plotted line. We can use any CSS colors as well as the Hexadecimal color codes.

We can use the "*linewidth*" keyword argument to change the width of the plotted line.

We can use *plt.plot()* function to plot as many lines as we want.

We can use the *xlabel()* and *ylabel()* functions for labelling the x-axis and y-axis. We can also use the

title() function for setting up the title. We can use the "*loc*" parameter in the *title()* function to reposition the title. We can use "*left*", "*right*", and "*center*" in the "*loc*" parameter.

We can use the *grid()* function to apply grid lines on the plot. We can add the different properties in grid() function as well.

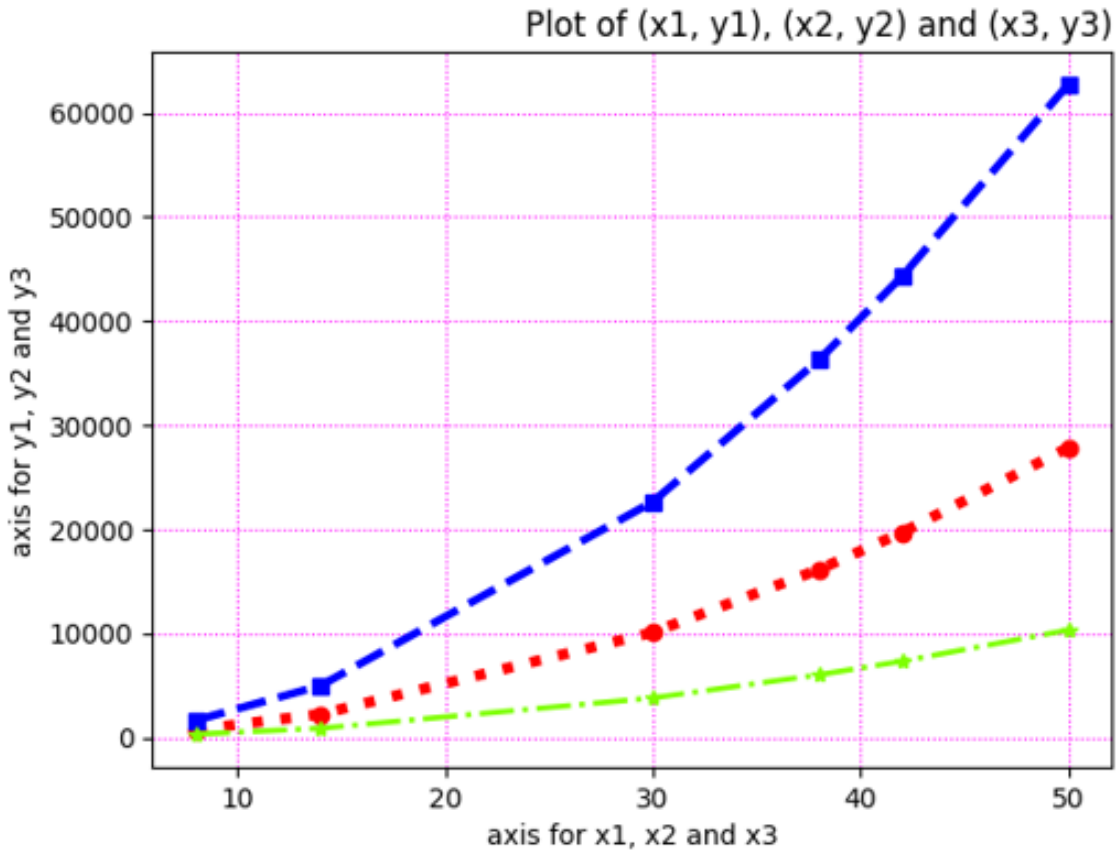
Suppose we want to plot the following three equations.

Code:

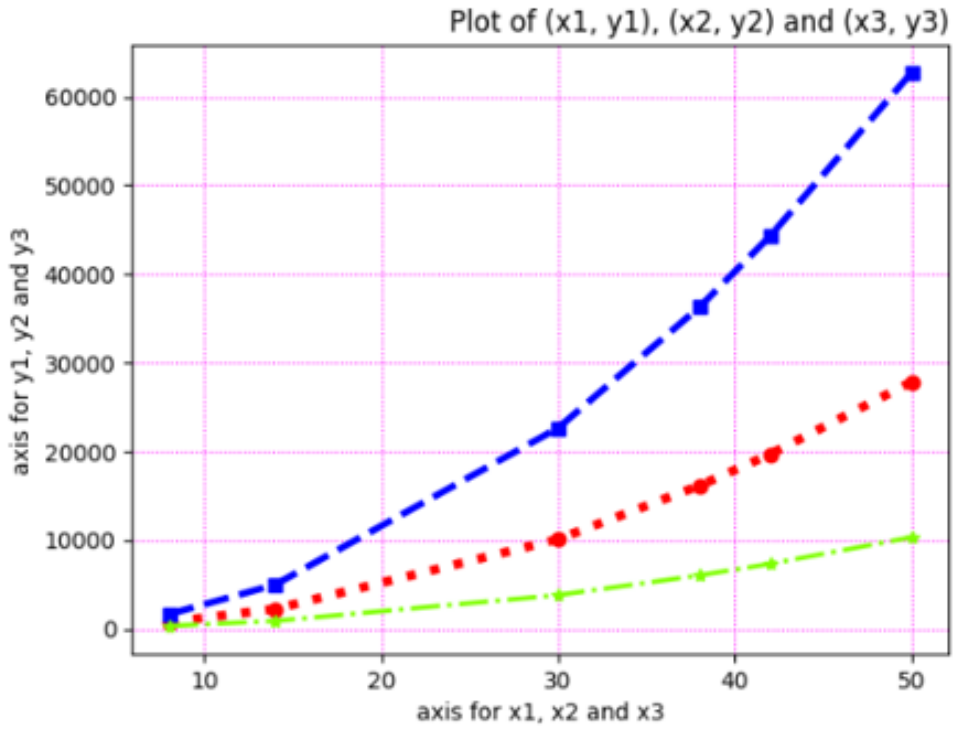
```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.array([8, 14, 30, 38, 42, 50])  #x1 values
y1 = (11 * x1 ** 2) + (7 * x1) - 20  #y1 values

x2 = np.array([8, 14, 30, 38, 42, 50])  #x2 values
y2 = (25 * x2 ** 2) + (4 * x2) + 10  #y2 values

x3 = np.array([8, 14, 30, 38, 42, 50])  #x3 values
y3 = (4 * x3 ** 2) + (6 * x3) + 14  #y3 values

plt.plot(x1, y1, marker = "o", linestyle = "dotted", color = "red", linewidth = "4")
```

```
plt.plot(x2, y2, marker = "s", linestyle = "dashed", color = "#0000FF", linewidth = "3")
plt.plot(x3, y3, marker = "*", linestyle = "dashdot", color = "Chartreuse", linewidth = "2")

plt.title("Plot of (x1, y1), (x2, y2) and (x3, y3)", loc = "right")
plt.grid(color = "magenta", linestyle = "dotted")
plt.xlabel("axis for x1, x2 and x3")
plt.ylabel("axis for y1, y2 and y3")
plt.show()
```

Output:



```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.array([8, 14, 30, 38, 42, 50])  #x1 values
y1 = (11 * x1 ** 2) + (7 * x1) - 20  #y1 values

x2 = np.array([8, 14, 30, 38, 42, 50])  #x2 values
y2 = (25 * x2 ** 2) + (4 * x2) + 10  #y2 values

x3 = np.array([8, 14, 30, 38, 42, 50])  #x3 values
y3 = (4 * x3 ** 2) + (6 * x3) + 14  #y3 values

plt.plot(x1, y1, marker = "o", linestyle = "dotted", color = "red", linewidth = "4")
plt.plot(x2, y2, marker = "s", linestyle = "dashed", color = "#0000FF", linewidth = "3")
plt.plot(x3, y3, marker = "*", linestyle = "dashdot", color = "Chartreuse", linewidth = "2")

plt.title("Plot of (x1, y1), (x2, y2) and (x3, y3)", loc = "right")
plt.grid(color = "magenta", linestyle = "dotted")
plt.xlabel("axis for x1, x2 and x3")
plt.ylabel("axis for y1, y2 and y3")
plt.show()
```



SUBPLOTS IN MATPLOTLIB

If we want to draw multiple plots in one figure, we need to use the *subplot()* function. The *subplot()* function takes three arguments to describe the layout of the figure.

plt.subplot(a, b, c)

Here, a is the number of rows, b is the number of columns and c is the plot number.

We can add titles to the individual plots, and we can add a title to the entire figure by using the suptitle() function.

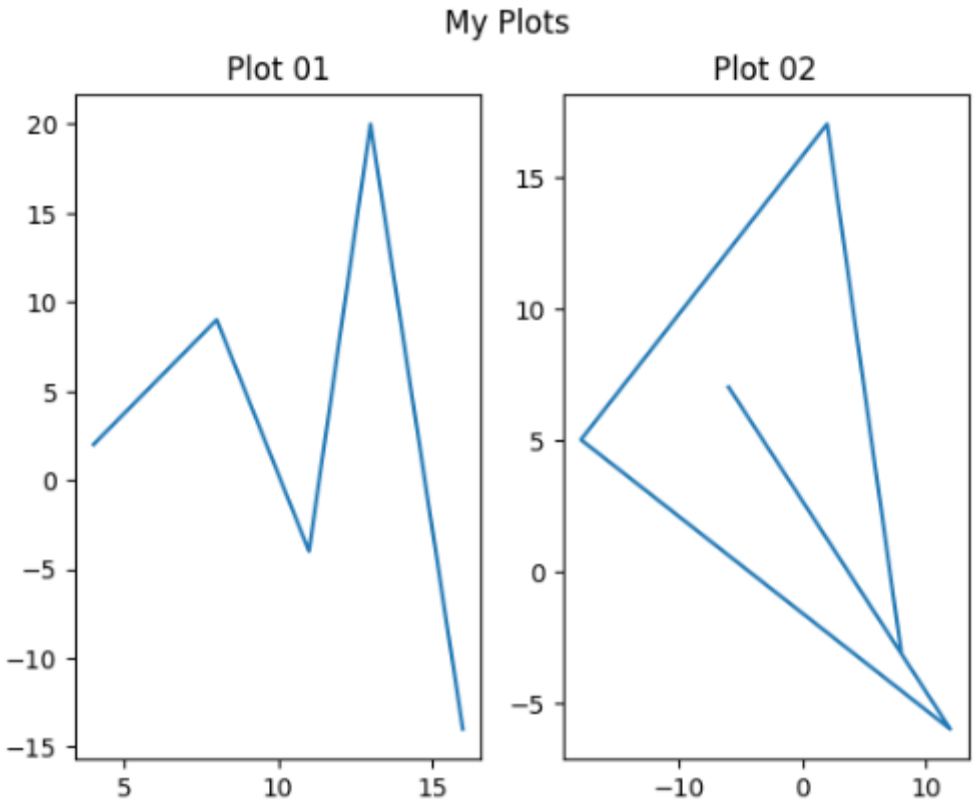
Code:

```
import numpy as np
import matplotlib.pyplot as plt

#Plot_01
x = np.array([4, 8, 11, 13, 16])
y = np.array([2, 9, -4, 20, -14])

plt.subplot(1, 2, 1)
plt.plot(x, y)
plt.title("Plot 01")

#Plot_02
x = np.array([8, 2, -18, 12, -6])
y = np.array([-3, 17, 5, -6, 7])

plt.subplot(1, 2, 2)
plt.plot(x, y)
```

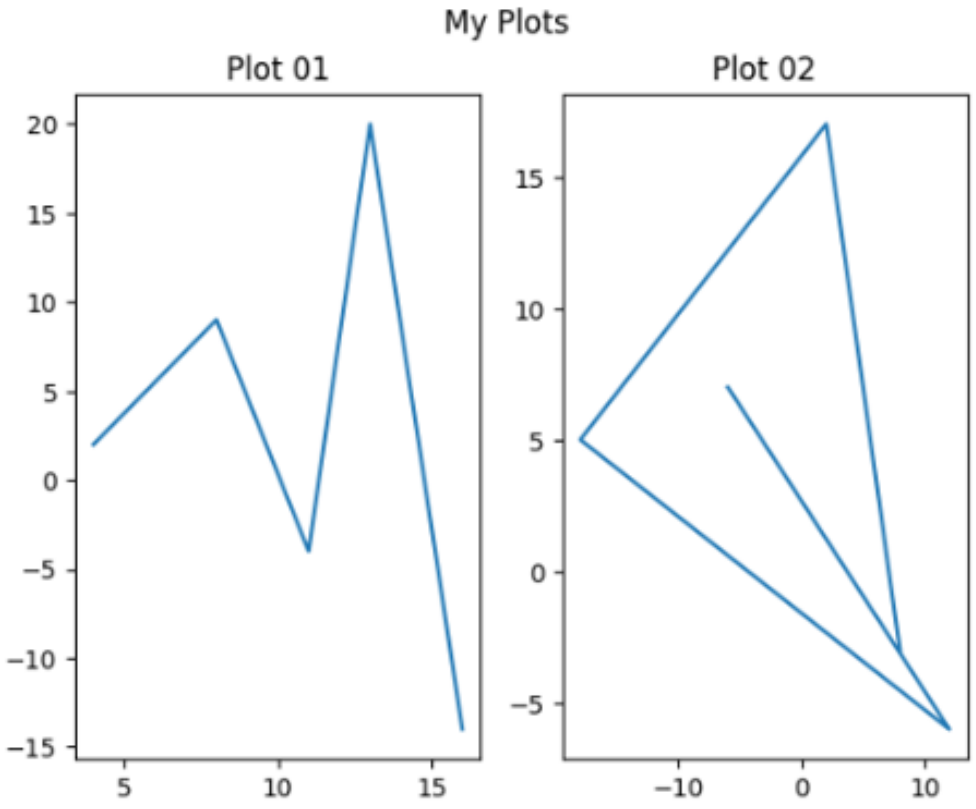
plt.title("Plot 02")

plt.suptitle("My Plots")
plt.show()

Output:



```python
import numpy as np
import matplotlib.pyplot as plt

#Plot_01
x = np.array([4, 8, 11, 13, 16])
y = np.array([2, 9, -4, 20, -14])

plt.subplot(1, 2, 1)
plt.plot(x, y)
plt.title("Plot 01")

#Plot_02
x = np.array([8, 2, -18, 12, -6])
y = np.array([-3, 17, 5, -6, 7])

plt.subplot(1, 2, 2)
plt.plot(x, y)
plt.title("Plot 02")

plt.suptitle("My Plots")
plt.show()
```



MATPLOTLIB SCATTER PLOTS, BAR GRAPHS, HISTOGRAMS AND PIE CHARTS

Scatter Plots:

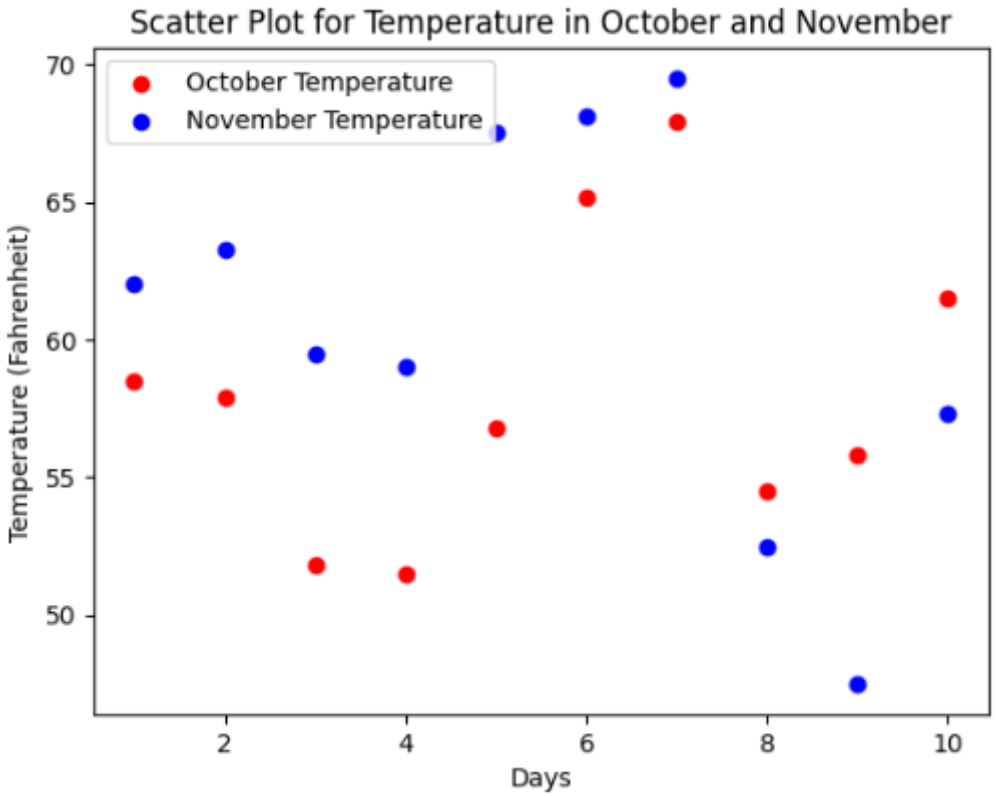
We can use the *scatter()* function to draw a scatter plot. This function plots one dot for each observation. We can add our preferred color to each scatter plot. We can add legends to the plot using the *legend()* function.

Code:

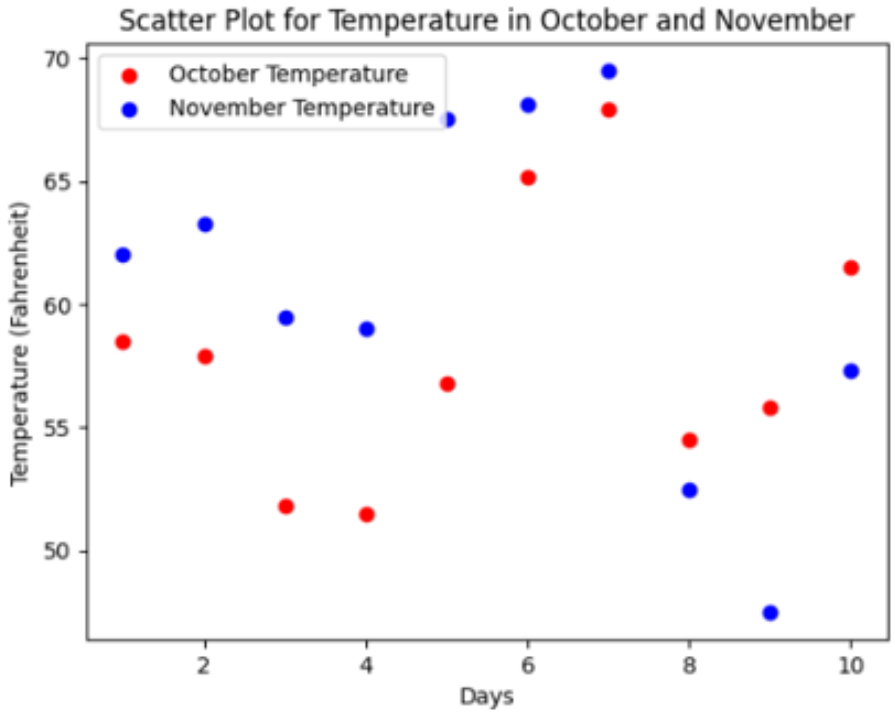
```
import numpy as np
import matplotlib.pyplot as plt

#temperature data in october
days_october = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
temperature_october = np.array([58.5, 57.9, 51.8, 51.5, 56.8, 65.2, 67.9, 54.5, 55.8, 61.5])

plt.scatter(days_october, temperature_october, color = "Red", label = "October Temperature")

#temperature data in november
days_november = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
temperature_november = np.array([62.0, 63.3, 59.5, 59.0, 67.5, 68.1, 69.5, 52.5, 47.5, 57.3])

plt.scatter(days_november, temperature_november, color = "Blue", label = "November Temperature")

plt.title("Scatter Plot for Temperature in October and November")
```

```
plt.xlabel("Days")
plt.ylabel("Temperature (Fahrenheit)")
plt.legend()
plt.show()
```

Output:



```python
import numpy as np
import matplotlib.pyplot as plt

#temperature data in october
days_october = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
temperature_october = np.array([58.5, 57.9, 51.8, 51.5, 56.8, 65.2, 67.9, 54.5, 55.8, 61.5])

plt.scatter(days_october, temperature_october, color = "Red", label = "October Temperature")

#temperature data in november
days_november = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
temperature_november = np.array([62.0, 63.3, 59.5, 59.0, 67.5, 68.1, 69.5, 52.5, 47.5, 57.3])

plt.scatter(days_november, temperature_november, color = "Blue", label = "November Temperature")

plt.title("Scatter Plot for Temperature in October and November")
plt.xlabel("Days")
plt.ylabel("Temperature (Fahrenheit)")
plt.legend()
plt.show()
```



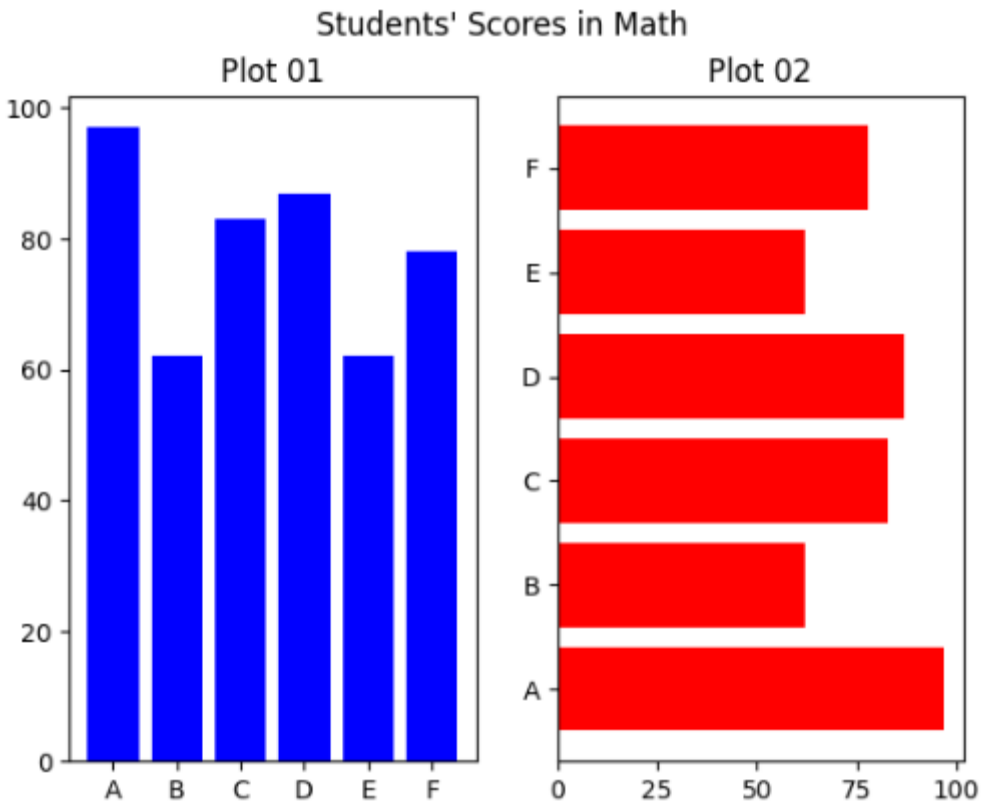
Bar Graphs:

If we want to draw the bar graphs, we can use the *bar()* function. If we want to display the bar horizontally, we can use the *barh()* function. We can add color to the bars.

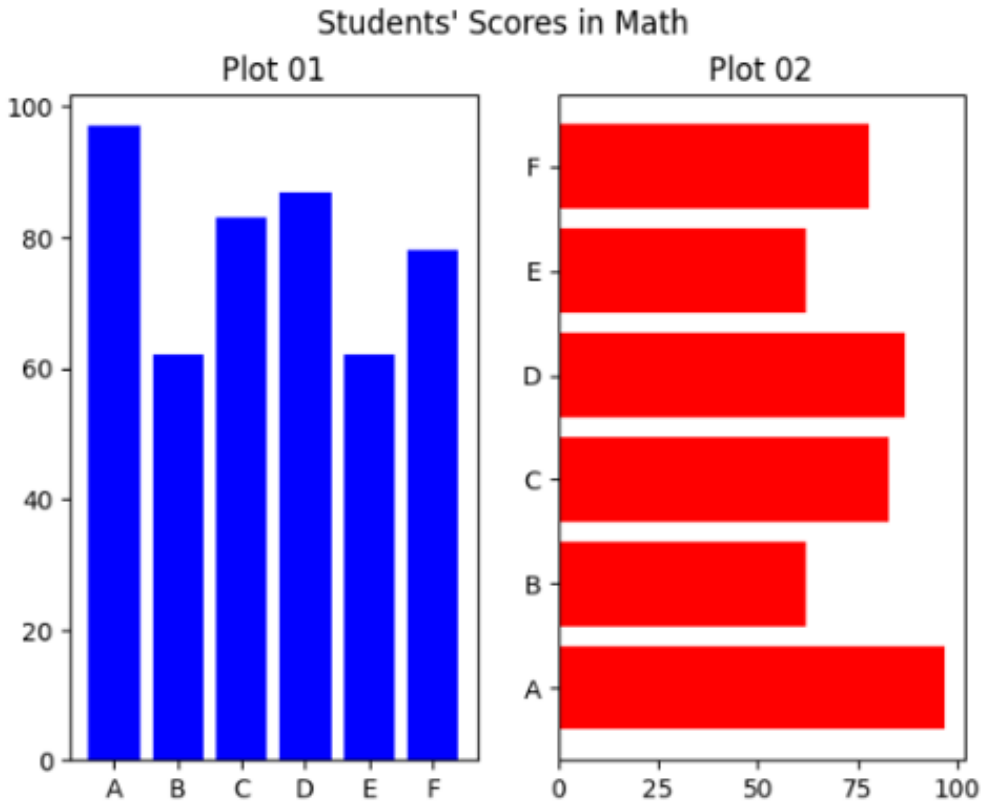
Code:

```
import numpy as np
import matplotlib.pyplot as plt

#Plot 01
students = np.array(["A", "B", "C", "D", "E", "F"])
scores_math = np.array([97, 62, 83, 87, 62, 78])

plt.subplot(1, 2, 1)
plt.bar(students, scores_math, color = "Blue")
plt.title("Plot 01")

#Plot 02
students = np.array(["A", "B", "C", "D", "E", "F"])
scores_math = np.array([97, 62, 83, 87, 62, 78])

plt.subplot(1, 2, 2)
plt.barh(students, scores_math, color = "Red")
plt.title("Plot 02")

plt.suptitle("Students' Scores in Math")
plt.show()
```

Output:


Students' Scores in Math
Plot 01
Plot 02
100
80
60
40
20
0
A
B
C
D
E
F
0
25
50
75
100

```python
import numpy as np
import matplotlib.pyplot as plt

#Plot 01
students = np.array(["A", "B", "C", "D", "E", "F"])
scores_math = np.array([97, 62, 83, 87, 62, 78])

plt.subplot(1, 2, 1)
plt.bar(students, scores_math, color = "Blue")
plt.title("Plot 01")

#Plot 02
students = np.array(["A", "B", "C", "D", "E", "F"])
scores_math = np.array([97, 62, 83, 87, 62, 78])

plt.subplot(1, 2, 2)
plt.barh(students, scores_math, color = "Red")
plt.title("Plot 02")

plt.suptitle("Students' Scores in Math")
plt.show()
```



Histogram:

We can use histograms in Matplotlib to observe the frequency distributions. Suppose we've a random set of 100 float numbers ranging from 0 to 1. We can use the *hist()* function to create a histogram.

Code:

```
import numpy as np
import matplotlib.pyplot as plt

random_numbers = np.random.random_sample((100,))
print("The Random Numbers: \n", random_numbers)
histogram_random_numbers = plt.hist(random_numbers)
print("Histogram of Random Numbers: \n",
histogram_random_numbers)
plt.show()
```

Output:

```
The Random Numbers:
[0.99970645 0.38652378 0.26376309 0.97650318 0.21282455
0.93609989
0.01223652 0.88489606 0.73211523 0.26084752 0.62088388
0.41359238
 0.70213179 0.03422516 0.13440504 0.5853184  0.49709348
0.39339666
0.84807557 0.85636143 0.01245723 0.29687602 0.97005694
0.16364311
0.98697444 0.07801281 0.19173311 0.78471766 0.76962146
0.37585719
0.15308166 0.49032123 0.42528288 0.07984182 0.54199524
0.88284878
```

0.29071074 0.27209094 0.12813872 0.75666989 0.9718723 0.37809274

0.95016316 0.06033715 0.25698112 0.15825548 0.92286199 0.33953332

0.5304793 0.4489419 0.78688742 0.43615918 0.46884775 0.85564764

0.21361248 0.94279661 0.2365931 0.8471353 0.08195358 0.58759777

0.35441567 0.77005144 0.68029704 0.06640786 0.49001966 0.79096429

0.22677418 0.89353091 0.35366981 0.05670551 0.50206579 0.6712167

0.52644735 0.16960066 0.26493867 0.79679 0.28080317 0.58669749

0.08164664 0.16046696 0.29420099 0.75995852 0.49935822 0.20840543

0.19143555 0.89876253 0.93875386 0.72610141 0.55456369 0.43709338

0.27846346 0.35975292 0.95325262 0.54736884 0.507812 0.64013932

0.89198432 0.81918986 0.1126081 0.74015938]

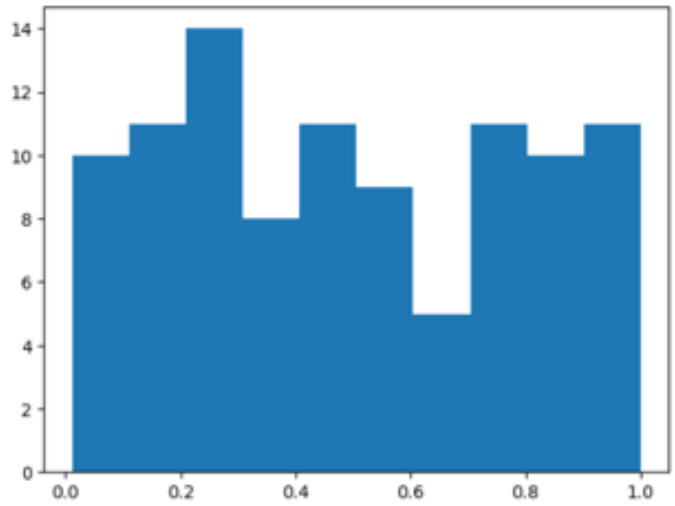
Histogram of Random Numbers:

(array([10., 11., 14., 8., 11., 9., 5., 11., 10., 11.]), array([0.01223652, 0.11098351, 0.20973051, 0.3084775 , 0.40722449,

0.50597148, 0.60471848, 0.70346547, 0.80221246, 0.90095945,

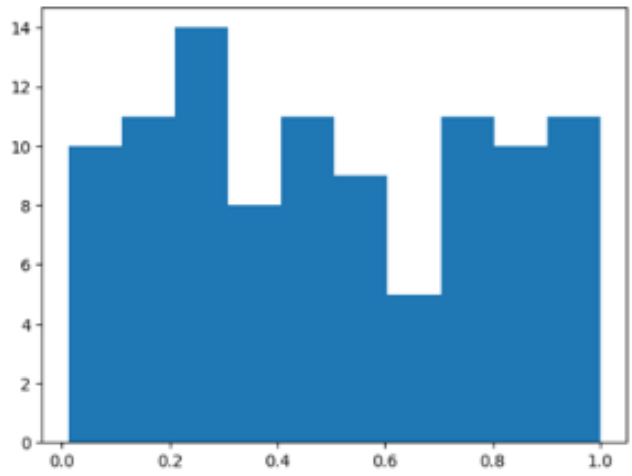
0.99970645]), <BarContainer object of 10 artists>)

```
The Random Numbers:
 [0.99970645 0.38652378 0.26376309 0.97650318 0.21282455 0.93609989
 0.01223652 0.88489606 0.73211523 0.26084752 0.62088388 0.41359238
 0.70213179 0.03422516 0.13440504 0.5853184  0.49709348 0.39339666
 0.84807557 0.85636143 0.01245723 0.29687602 0.97005694 0.16364311
 0.98697444 0.07801281 0.19173311 0.78471766 0.76962146 0.37585719
 0.15308166 0.49032123 0.42528288 0.07984182 0.54199524 0.88284878
 0.29071074 0.27209094 0.12813872 0.75666989 0.9718723  0.37809274
 0.95016316 0.06033715 0.25698112 0.15825548 0.92286199 0.33953332
 0.5304793  0.4489419  0.78688742 0.43615918 0.46884775 0.85564764
 0.21361248 0.94279661 0.2365931  0.8471353  0.08195358 0.58759777
 0.35441567 0.77005144 0.68029704 0.06640786 0.49001966 0.79096429
 0.22677418 0.89353091 0.35366981 0.05670551 0.50206579 0.6712167
 0.52644735 0.16960066 0.26493867 0.79679    0.28080317 0.58669749
 0.08164664 0.16046696 0.29420099 0.75995852 0.49935822 0.20840543
 0.19143555 0.89876253 0.93875386 0.72610141 0.55456369 0.43709338
 0.27846346 0.35975292 0.95325262 0.54736884 0.507812   0.64013932
 0.89198432 0.81918986 0.1126081  0.74015938]
Histogram of Random Numbers:
 (array([10., 11., 14.,  8., 11.,  9.,  5., 11., 10., 11.]), array([0.01223652, 0.11098351, 0.20973051, 0.3084775 , 0.40722449,
        0.50597148, 0.60471848, 0.70346547, 0.80221246, 0.90095945,
        0.99970645]), <BarContainer object of 10 artists>)
```



```
import numpy as np
import matplotlib.pyplot as plt

random_numbers = np.random.random_sample((100,))
print("The Random Numbers: \n", random_numbers)
histogram_random_numbers = plt.hist(random_numbers)
print("Histogram of Random Numbers: \n", histogram_random_numbers)
plt.show()
```

```
The Random Numbers:
 [0.99970645 0.38652378 0.26376309 0.97650318 0.21282455 0.93609989
 0.01223652 0.88489606 0.73211523 0.26084752 0.62088388 0.41359238
 0.70213179 0.03422516 0.13440504 0.5853184  0.49709348 0.39339666
 0.84807557 0.85636143 0.01245723 0.29687602 0.97005694 0.16364311
 0.98697444 0.07801281 0.19173311 0.78471766 0.76962146 0.37585719
 0.15308166 0.49032123 0.42528288 0.07984182 0.54199524 0.88284878
 0.29071074 0.27209094 0.12813872 0.75666989 0.9718723  0.37809274
 0.95016316 0.06033715 0.25698112 0.15825548 0.92286199 0.33953332
 0.5304793  0.4489419  0.78688742 0.43615918 0.46884775 0.85564764
 0.21361248 0.94279661 0.2365931  0.8471353  0.08195358 0.58759777
 0.35441567 0.77005144 0.68029704 0.06640786 0.49001966 0.79096429
 0.22677418 0.89353091 0.35366981 0.05670551 0.50206579 0.6712167
 0.52644735 0.16960066 0.26493867 0.79679    0.28080317 0.58669749
 0.08164664 0.16046696 0.29420099 0.75995852 0.49935822 0.20840543
 0.19143555 0.89876253 0.93875386 0.72610141 0.55456369 0.43709338
 0.27846346 0.35975292 0.95325262 0.54736884 0.507812   0.64013932
 0.89198432 0.81918986 0.1126081  0.74015938]
Histogram of Random Numbers:
 (array([10., 11., 14.,  8., 11.,  9.,  5., 11., 10., 11.]), array([0.01223652, 0.11098351, 0.20973051, 0.3084775 , 0.40722449,
       0.50597148, 0.60471848, 0.70346547, 0.80221246, 0.90095945,
       0.99970645]), <BarContainer object of 10 artists>)
```



Pie Charts:

We can use the *pie()* function to draw pie charts in Matplotlib. Suppose we've the following data of world cup winning teams.

Team Name	World Cup Trophies
Brazil	5
Italy	4
Germany	4

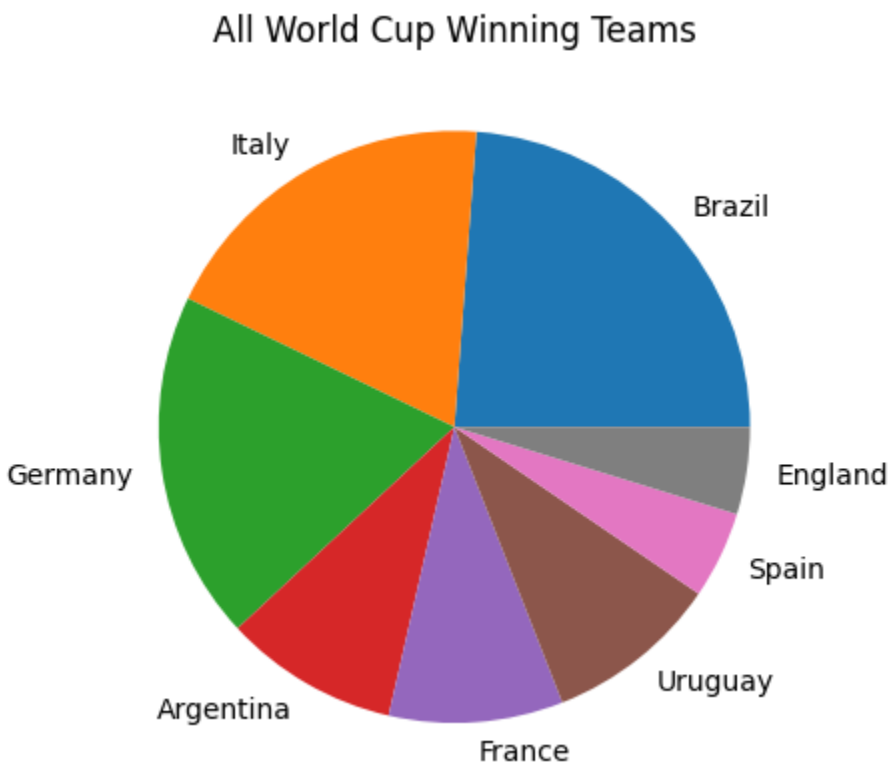
Argentina	2
France	2
Uruguay	2
Spain	1
England	1

We can create a pie chart from this data in the following way.

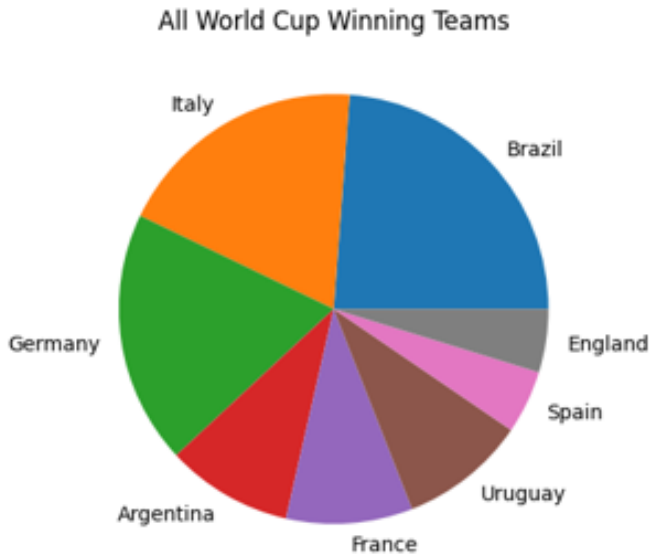
Code:

```
import numpy as np
import matplotlib.pyplot as plt

world_cup_trophies = np.array([5, 4, 4, 2, 2, 2, 1, 1])
team_names = np.array(["Brazil", "Italy", "Germany",
"Argentina", "France", "Uruguay", "Spain", "England"])

plt.title("All World Cup Winning Teams")
plt.pie(world_cup_trophies, labels = team_names)
plt.show()
```

Output:


All World Cup Winning Teams
Italy
Brazil
Germany
England
Spain
Uruguay
Argentina
France

```
import numpy as np
import matplotlib.pyplot as plt

world_cup_trophies = np.array([5, 4, 4, 2, 2, 2, 1, 1])
team_names = np.array(["Brazil", "Italy", "Germany", "Argentina", "France", "Uruguay", "Spain", "England"])

plt.title("All World Cup Winning Teams")
plt.pie(world_cup_trophies, labels = team_names)
plt.show()
```



If we want one of the wedges to stand out, we can use the "*explode*" parameter. Suppose we want "France" and "Spain" to stand out. We can do it in the following way.

Code:

import numpy as np

import matplotlib.pyplot as plt

world_cup_trophies = np.array([5, 4, 4, 2, 2, 2, 1, 1])

team_names = np.array(["Brazil", "Italy", "Germany", "Argentina", "France", "Uruguay", "Spain", "England"])

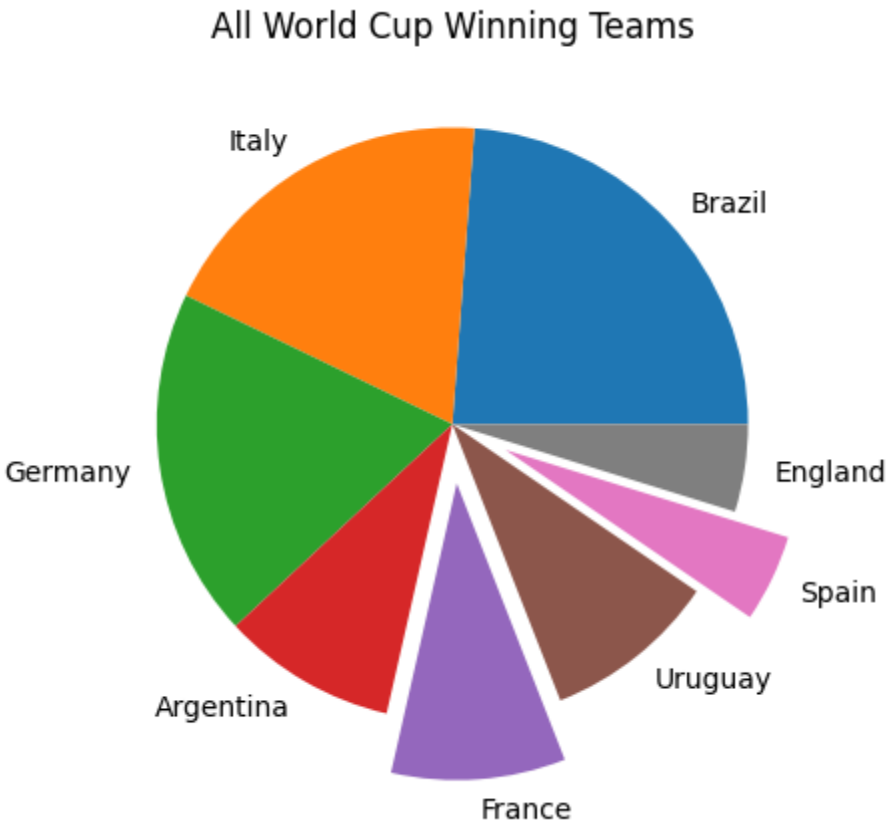
preferred_teams = np.array([0, 0, 0, 0, 0.2, 0, 0.2, 0])

plt.title("All World Cup Winning Teams")

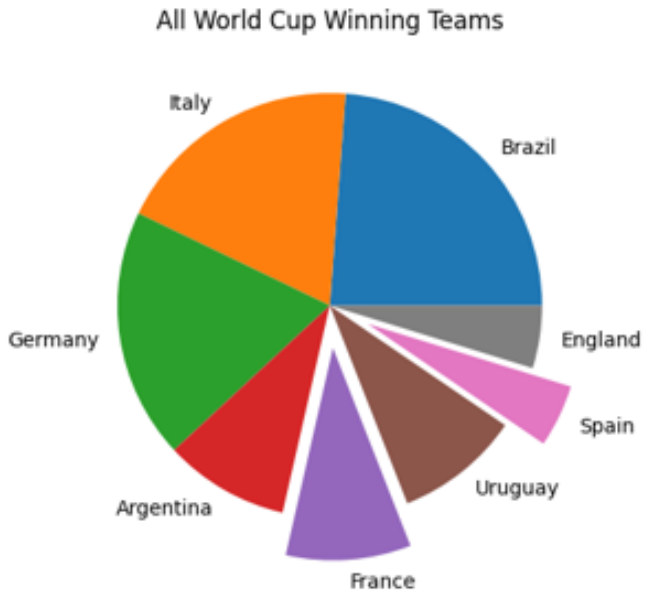
plt.pie(world_cup_trophies, labels = team_names, explode = preferred_teams)

plt.show()

Output:



```
import numpy as np
import matplotlib.pyplot as plt

world_cup_trophies = np.array([5, 4, 4, 2, 2, 2, 1, 1])
team_names = np.array(["Brazil", "Italy", "Germany", "Argentina", "France", "Uruguay", "Spain", "England"])
preferred_teams = np.array([0, 0, 0, 0, 0.2, 0, 0.2, 0])

plt.title("All World Cup Winning Teams")
plt.pie(world_cup_trophies, labels = team_names, explode = preferred_teams)
plt.show()
```



DON'T MISS OUT!

Click the button below and you can sign up to receive emails whenever Ibnul Jaif Farabi publishes a new book. There's no charge and no obligation.

https://books2read.com/r/B-H-DDHI-RQKDC

Sign Me Up!

https://books2read.com/r/B-H-DDHI-RQKDC

BOOKS 2 READ

Connecting independent readers to independent writers.

Also by Ibnul Jaif Farabi

Learn HTML and CSS In 24 Hours and Learn It Right | HTML and CSS For Beginners with Hands-on Exercises

Python Programming for Beginners Crash Course with Hands-On Exercises, Including NumPy, Pandas and Matplotlib

ABOUT THE AUTHOR

Ibnul Jaif Farabi is a Web Designer and Developer, UI / UX and Mobile App Developer based in New York City.

He has a wide depth of knowledge and expertise in using his technical skills in the fields of electrical engineering, computer science and software development to help organizations increase productivity, as well as accelerating business performance. He also has a deep passion for troubleshooting technical issues and brainstorming new ideas that can lead to unique solutions. He works effectively and efficiently as a team player as well as being purpose driven and result oriented.