



# Python for Data Science

UNDERSTANDING PYTHON FOR DATA SCIENCE AND MACHINE  
LEARNING PROJECTS

**This is a free eBook from Teksands**



**Teksands is a Deep Tech Learning company dedicated towards creating new age Digital professionals. Our core strength is our Faculty Members who are Research Scholars in the respective fields in top institutes like IIT and NITs.**

**We invite you to our world. We have courses for budding as well as experienced Deep Tech Professionals.**

**Apart from that, Teksands is dedicated to serve the IT profession – we have a huge repository of Deep Tech Articles in our website which anyone can access free.**

**We bring you free eBooks regularly to help with your quest of Tech knowledge. We thank you for your interest in us.**

**Our courses:**

**<https://teksands.ai/courses/>**

**Our Knowledge Repository:**

**<https://teksands.ai/article/>**

# Table of Contents

<b>PYTHON FOR DATA SCIENCE</b>	<b>4</b>
<b>LEARNING OBJECTIVE</b>	<b>4</b>
<b>HISTORY</b>	<b>4</b>
<b>FEATURES OF PYTHON</b>	<b>4</b>
<b>ADVANTAGES OF PYTHON AS THE PREFERRED LANGUAGE FOR DATA SCIENCE</b>	<b>5</b>
<b>INTRODUCTION TO PYTHON PROGRAMMING</b>	<b>5</b>
<i>Variables in Python:.....</i>	<i>5</i>
<i>Python Operators .....</i>	<i>7</i>
<i>Conditional Statements in Python .....</i>	<i>9</i>
<i>Implementing Iterations:.....</i>	<i>12</i>
<b>PYTHON COLLECTION DATA TYPES</b>	<b>19</b>
<i>Python Lists.....</i>	<i>19</i>
<i>Working with Tuples.....</i>	<i>23</i>
<i>Dictionary .....</i>	<i>30</i>
<i>Set.....</i>	<i>36</i>
<b>WORKING WITH PYTHON NUMPY LIBRARY:</b>	<b>43</b>
<i>Installation of NumPy:.....</i>	<i>44</i>
<i>NumPy Arrays.....</i>	<i>44</i>
<i>Printing Array Attributes .....</i>	<i>48</i>
<i>Array Functions (Methods) .....</i>	<i>49</i>
<i>Accessing Array Elements.....</i>	<i>53</i>
<i>Iterating through a Numpy array .....</i>	<i>56</i>
<i>Array Assignment vs Copy .....</i>	<i>60</i>
<b>WORKING WITH THE PANDAS LIBRARY</b>	<b>61</b>
<i>Introduction to Pandas.....</i>	<i>61</i>
<i>Installing Pandas Library .....</i>	<i>62</i>
<i>Working with Series Objects.....</i>	<i>62</i>
<i>Series Object Attributes .....</i>	<i>67</i>
<i>Series Object Methods.....</i>	<i>69</i>
<i>Multi-Indexed Series Objects .....</i>	<i>81</i>

## Python for data science

### Content List:

- Python Data Types
- Conditional Constructs
- Iteration Constructs
- Working with Functions

### Learning Objective

Over the last few years, Python has become the programming language of choice for Developers the world over. For students and professionals of Artificial Intelligence, Machine Learning and Data Science, Python is the de-facto language leaving behind competitors such as “R”, Java, Scala, Julia, MATLAB and others in amount of adoption (60%+ adoption amongst competing languages/products as this is being written in 2020). Python The major strength of Python that has attracted much of the adoption is its simplicity in learning, understanding and implementation. Another strength of Python that has emerged is its versatility – Python is equally adept in a wide range of contexts like Application Development, Web App development, Data Science and ML, DevOps scripting, Web crawling, research projects and prototyping, and much else. Support from a Open Source community has ensured stable versions plus availability of a wide range of libraries for various types of usages, especially Data Science and related fields.

### History

Python was developed by Guido van Rossum, a Dutch programmer back in 1991 when the first version of Python was released. Van Rossum started developing Python as a hobby project to spend his Christmas holiday time and he chose the name of the Language from his liking for “Monty Python”. The base Python language is written in “C”.

The goal Van Rossum had in mind while developing Python are:

- An easy and intuitive language
- Open source
- Easily Understandable Code
- Suitability for everyday tasks, allowing for short development times

### Features of Python

- Highly Portable, works in most of the current Operating Systems, e.g. MacOS, Windows, Linux, Unix, Android
- Suitable for a very wide range of uses, e.g. developing Web Applications, usage in Artificial Intelligence / Machine Learning / Data Science, working with Data files and Databases, Regular Expressions handling
- Free and Open Source. Python source code can be customized by users.
- High Level, no need for compilation
- Interpreted Language
- Procedural as well as Object Orientation

- Dynamically Typed
- Garbage Collected
- Elegant Syntax, easy to read
- Highly maintainable
- Easily extensible by modules written/compiled in other languages

## Advantages of Python as the preferred Language for Data Science

Some of the features that made Python the most preferred Data Science Language are:

- Very small learning curve for beginners due to its overall simplicity
- Extensive Libraries (e.g. Numpy, Pandas, SciPy, Scikit-Learn, Keras, Tensorflow) supporting various Data Science / AI related operations including fantastic visualization libraries and libraries for very complex AI operations
- Solid support ecosystem of volunteers contributing to the language and libraries apart from support forums in problem solving and adoption improvement
- Python being a dynamically typed language, it's handling of data is fairly simple albeit strong

## Introduction to Python Programming

In this section, we will go through some of the basic components of Python programming, such as:

- Declaring Variables
- Printing/Displaying
- Condition handling
- Iterations

## Variables in Python:

As you may know already, a variable is a place in the memory where the computer holds a value. The variable has a designated location, and the content, i.e. the value. Within a programming language like Python, we do not have to worry about the memory location, all we have to know is the Name of the variable. Another attribute of the variable is its type, i.e. the kind of value that it can hold. Python has the following main Variable Types, aka, Data Types.

- **Integer:** Integer variables can hold Integers such as 1, 10, 100, 650, etc. They cannot store values such as 12.5.
- **Float:** Floating point variables can hold values such as 12.5, 126.7897, etc. However, they can also store 1, 2, 3 or 78 (Integers).
- **Boolean:** Boolean variables can hold a Boolean Value, i.e. True or False
- **String:** A string type variable can store value such as "My name is John Doe" or "X".

### Coding Sample:

```
# Creating Variables
i = 5
print(i)
str = "My name is John Doe"
print(str)
flt = 12.678
print(flt)
bln = True
print(bln)
```

### Output:

```
5
My name is John Doe
12.678
True
```

In the above example, we are creating four variables of the four types we have explained, and the output of the print function is shown following the code.

Note that we are directly assigning values to the variables without having to create them like some of the other languages like Java. The Python interpreter dynamically allocates the memory and assigns the type of the variable based on the value passed to it.

You can display the type of the variable using the type function.

### Coding Sample:

```
# Print the Type of each of the variables
print(type(i))
print(type(str))
print(type(flt))
print(type(bln))
```

### Output:

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
```

Interestingly, Python allows the programmer to assign a value to a variable different from the initial type of value assigned. Python dynamically changes the type of the variable at run-time and continues.

### Coding Sample:

```
# A string value is passed without any problem
i = 5
print(i)
i = "Changed to a String"
print(i)
```

## Output:

```
5
Changed to a String
```

## Python Operators

Python supports a number of types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

## Arithmetic Operators

Symbol	Operation	Description
+	Addition	Adds the operands. Can be applied on int, float, string.
Code	<pre>i = 5 j = 6 print(i+j) # str1 = "John " str2 = "Doe" print(str1 + str2)</pre>	Output: 11 'John Doe'
-	Subtraction	Subtracts the operands. Can be applied on int, float.
Code	<pre>i = 5 j = 6 print(i-j)</pre>	Output: -1
*	Multiplication	Multiplies the operands. Applied on numeric values.
Code	<pre>i = 5 j = 6 print(i*j)</pre>	Output: 30

/	Division	Performs division of the operands. Applied on numeric values.
Code	i = 5 j = 6 print(i/j)	Output: 0.8333333333333334
%	Modulus	Divides the operands and returns remainder. Applied on Numeric values.
Code	i = 30 j = 7 print(i%j)	Output: 2
**	Exponent	Finds exponential (power) of second operand over the first.
Code	i = 4 j = 3 print(i**j)	Output: 64
//	Floor Division	Performs division of the operands and returns the quotient as an integer (stripping the decimal if any)
Code	i = 4 j = 3 print(i//j)	Output: 1

### Comparison Operators:

These operators are used in comparing values using condition statements. These operators return a True or False (Boolean) outcome.

Operator	Explanation	Example
==	"Equals" operator – returns True if the two operands are Equal	X == Y
>	"Less Than" operator – returns True if the First operand is Greater than the second	X > Y
<	"Greater Than" operator – returns True if the First operand is smaller than the second	X < Y
>=	"Greater Than or Equal to" operator – returns True if the First operand is greater than or equal to the second	X >= Y
<=	"Less Than or Equal to" operator – returns True if the First operand is smaller than or equal to the second	X <= Y



!= / <>	"Not Equal to" operator – returns True if the operands are not equal	X != Y
---------	--	--------

## Assignment Operators

These operators are used in assigning values to variables.

Operator	Explanation	Example
=	Assigns value from the operand at the left side to the one at the left side (assignee)	X = Y
+=	Increments the left operand by the value of the right operand	X += 5
-=	Subtracts the left operand by the value of the right operand	X -= 5
*=	Multiplies the left operand by the value of the right operand	X *= 5
/=	Divides the left operand by the value of the right operand	X /= 5
%=	Performs modulus operation (find the remainder of division) on the left operand by the value of the right operand	X %= 5
**=	Find the power (exponential) of the left operand by the value of the right operand	X **= 5
//=	Performs a floor division (finds the quotient of a division operation and strip the decimals) of the left operand by the value of the right operand	X //= 5

## Conditional Statements in Python

Like all procedural languages, Python provides the standard conditional constructs in the form of 'if' statements and 'switch' statements. The Conditional statements return a True or False. In case of the statement returning True, the subsequent Block of code is execute.

They are explained with examples as below:

### Coding Sample:

```
# Checking Conditions
x = 7
y = 9
if x == y:
    print("Equal")
else:
    print("Inequal")
```

### Output:

Unequal

#### Coding Sample:

```
# Checking Conditions (Multiple Conditions)
x = 7
y = 9
if x == y:
    print("Equal")
elif x > y:
    print("x is greater than y")
elif x < y:
    print("x is smaller than y")
```

#### Output:

x is smaller than y

#### Coding Sample:

```
# Checking Conditions (Multiple Simultaneous Conditions)
x = 7
y = 9
if x == 7 and y < 10:
    print("Condition Met – Multiline Block")
    print("Condition Met")
else:
    print("Condition not Met – Multiline Block")
    print("Condition not Met")
```

#### Output:

Condition Met – Multiline Block  
Condition Met

Points from the above example:

- Note the use of 'elif' interpreted as 'else if' to check further conditions
- 'else' is used to perform an action if none of the above conditions are met
- 'and' is used to check multiple conditions together
- 'or' is to be used in place of 'and' if the condition is 'if this' or 'that' then perform the subsequent action

**Indentation and Block marking:** Note that after each condition line, a ':' sign is used to denote the line as a conditional construct line.

The action following the condition line, ended by ':' is the action block. Note that unlike C/C++/Java, Python does not use a block denoting character like '{}'. Blocks in Python are to be perfectly indented after the starting line (e.g. 'if'/'elif'/'else'). Each line in a block are to be similarly indented, otherwise it results in

an error. This indentation need or property of Python is also known as the 'Off-side Rule'. Apart from Python, a few other computer languages (such as, Cobra, Coffeescript, Curry, Elixir, etc) follow the Off-side rule.

Further Example:

#### Coding Sample:

```
# Checking for Boolean
x = True
if x:
    print(x)
```

#### Output:

```
True
```

Note:

The 'if'/'elif'/'else' statements results in a True or False (Boolean) outcome. The above example is a demonstration of that.

**Nested Conditions:** Nested conditions are used when further conditions are to be checked based on the True/False status of the outer conditions.

#### Coding Sample:

```
# Nested 'if'
x = 7
y = 9
if x < y:
    if x > 5:
        print("Nested Condition Met")
    else:
        print("Inner nested False Block")
else:
    print("Outer nested False Block")
```

#### Output:

```
Nested Condition Met
```

**Multiple Operations on a Single Line:** It is possible to embed the block action code following the condition on the same line in simple situations.

#### Coding Sample:

```
# Code on same line as the condition
x = 7
if x > 5: print("Condition met - in one line")
```

#### Output:

Condition met - in one line

The above example is from Python's ability to accommodate multiple statements on the same line.

#### Coding Sample:

```
# Multiple statements on the same line  
print(x); print(y)
```

#### Output:

```
7  
9
```

**Conditional Expressions:** Python provides another conditional construct or expression that is also known as a Conditional Operator or Ternary Operator. Conditional Operators evaluate a condition and perform an action (or two) based on the True/False return status of the main condition or another action based on the 'else' condition.

#### Coding Sample:

```
# Conditional Operator  
x = 'Tired'  
print("I want to ", 'go to Sleep' if x == 'Tired' else 'watch TV')
```

#### Output:

```
I want to  watch TV
```

In the below example, an assignment is made by using the Conditional Operator.

#### Coding Sample:

```
# Conditional Operator  
x = 'Tired'  
result = 'Sleep' if x == 'Tired' else 'watch TV'  
print(result)
```

#### Output:

```
Sleep
```

### Implementing Iterations:

Iterations are capabilities of a language to perform a set of actions repeatedly. There are several aspects related to repeated actions in your code:

1. **Entry Criteria** – We need to specify the condition which should be satisfied for the code to enter into the block that needs to be repeated. An incorrect Entry criteria may result in the Repeat Block not getting executed.
2. **Exit Criteria** – A condition needs to be mentioned based on which the repetition will end. A missing or incorrect Exit criteria may lead to the iteration to go on endlessly (Infinite Loop).

3. Set of Code that need to be repeated must be carefully established. Any code that must not be repeated or may have a detrimental effect on the overall code performance or outcome must not be part of the repeat block. This is a common mistake – sometimes code that must reside outside of repeat block as only one-time execution is needed, are placed inside a repeat block.
4. There can also be nested Iteration constructs or nested loops. Setting up the conditions and limits of outer and inner loops must be carefully constructed.
5. Loops or Iterations are very powerful tools of any programming language – they make the code efficient but can be also degrade performance of your application if not correctly used.
6. Normally Loops will take a set of inputs (Multiple similar data) and process through them to produce the outcome.

There are various types of Loops or Iterations supported by Python for different types of uses and based on various types of inputs provided to it.

**Definite Iteration:** When we know the number of times the loop will iterate.

**Coding Sample:**

```
# Example of Definite Iteration
l = 5
m = 0
i = 0
while l > 0:
    m += l
    l-=1
    i+=1
    print("Iteration ", i)
print("Final value of m = ", m)
```

**Output:**

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Final value of m = 15
```

In the above example, the loop is expected to repeat while the value of *l* remains greater than 0 (*l* > 0). We know by looking at the setup of the condition that the loop will repeat 5 time. This is an example of a “Definite” loop when we know at the time of writing the code as to how many times this loop or iteration will repeat.

Note the above loop construct uses the ‘while’ clause (called a ‘while’ loop). The while statement above is interpreted as ‘perform the subsequent code block till the condition mentioned along with the ‘while’ clause remains True. The program control will enter inside the loop only when the condition is true and will exit when the condition no longer remains True.

**Indefinite Iteration:** In the case of an Indefinite Iteration, one of the variables used as part of the condition is dynamic and is determined at runtime (e.g. in the above example, if the value of variable ‘*l*’ if not fixed

and is determined dynamically by the program at runtime). Thereby, estimating the number of times the loop will iterate will not be possible by looking at the code.

**A 'for' Loop:** The 'for' loop in Python loops over a range or list of inputs provided along with the 'for' statement. This range or list is called an Iterable.

This is different from the 'while' loop where the repetition is based on a condition provided.

Python provides a number of collection data types, namely, List, Tuple, Set and Dictionary. While they have different attributes and implementations, the one thing that is common is that they can store multiple data. The 'for' loop is specifically designed in Python to iterate over this set of multiple data and process them one by one in the Repeat Block.

The Collection data types are used widely in Python codes as a standard practice for internal processing of datasets and hence the 'for' loop on such Iterables (collection data types) becomes a very powerful feature.

A quick look on how the Collection Data Types look:

List: WeekdaysList = ["Sunday", "Monday", "Tuesday"]

Tuple: CountryTuple = ("India", "US", "UK", "Germany")

Set: PlayerSet = {"Ronaldo", "Messi", "Neymar"}

Dictionary: PersonDict = {"Name": "John Doe", "Age": 30, "Salary": 80000}

#### Coding Sample:

```
# Example of 'for' loop for a List
WeekdaysList = ["Sunday", "Monday", "Tuesday"]
for x in WeekdaysList:
    print(x)
```

#### Output:

```
Sunday
Monday
Tuesday
```

In the above example, the loop picks up the elements in the List sequentially starting from the first element and processes them in each iteration. The WeekdaysList Tuple is the Iterator in this case.

#### Coding Sample: (Iterator is a Tuple)

```
# Example of 'for' loop for a Tuple
CountryTuple = ("India", "US", "UK", "Germany")
for x in CountryTuple:
    print(x)
```

#### Output:

```
India
US
UK
Germany
```

### Coding Sample: (Iterator is a Set)

```
# Example of 'for' loop for a Set
PlayerSet = {"Ronaldo", "Messi", "Neymar"}
for x in PlayerSet:
    print(x)
```

### Output:

```
Neymar
Ronaldo
Messi
```

### Coding Sample: (Iterator is a Dictionary)

```
# Example of 'for' loop for a Dictionary
PersonDict = {"Name": "John Doe", "Age": 30, "Salary": 80000}
for x in PersonDict:
    print(x, PersonDict[x])
```

### Output:

```
Name John Doe
Age 30
Salary 80000
```

A dictionary is a Key-Value pair. While processing a dictionary in a 'for' loop, Python picks only the 'Key' value and not the 'Value'. To print or access the 'Value' for the key being processed, we have to pass the key to the Dictionary.

### Coding Sample: (Iterator is a String)

```
# Example of 'for' loop for a String
str = "John Doe"
for x in str:
    print(x)
```

### Output:

```
J
o
h
n

D
o
e
```

In the above example, the for loop considers the string as the Iterable and processes one character at a time sequentially from the Iterable string.

**For Loops for Numeric Ranges:** The next type of for loop is for a Numeric Range to be the Iterable. 'for' loop processes through each number of the range at a time.

In the following examples, 'range' is a built-in function that returns a sequence for the 'for' loop to iterate over. The sequence returned by the 'range' function depends on a. Start Value (default to 0), b. End value (needed), c. Step Value (default to 1).

#### **Coding Sample: (Iterator is a Numeric Range)**

```
# Example of 'for' loop for a Numeric Range
for x in range(5):
    print(x, " is being processed")
```

#### **Output:**

```
0 is being processed
1 is being processed
2 is being processed
3 is being processed
4 is being processed
```

Note that the range length (or End value) that is input to the 'for' loop is 5. However, the numbers in the range starts from 0. So a range of length 5 will consist of numbers 0-4.

**Specifying Start and End value of Range:** It is possible to mention custom start and end values with the range. As follows:

#### **Coding Sample: (Iterator is a Numeric Range with custom start and end values)**

```
# Example of 'for' loop for a Numeric Range with custom start and end values
for x in range(3,8):
    print(x, " is being processed")
```

#### **Output:**

```
3 is being processed
4 is being processed
5 is being processed
6 is being processed
7 is being processed
```

Note that in the above example, the 'for' loop is interpreted as 'starting from number 3, continue until the 8<sup>th</sup> number (which is 7 on a scale starting from 0)'.

**Specifying Step Value:** With the Start and End values of the Range, we can also specify the step value, i.e., with each iteration, what is the value by which the Range will increment or decrement. By default, it is 1.

#### **Coding Sample: (Iterator is a Numeric Range with custom start and end values)**

```
# Example of 'for' loop for a Numeric Range with custom start and end values and step amounts
for x in range(2,14,2):
    print(x, " is being processed")
```



### Output:

```
2 is being processed
4 is being processed
6 is being processed
8 is being processed
10 is being processed
12 is being processed
```

Note that in the above example, the 'for' loop is interpreted as 'starting from number 2, continue until the 14<sup>th</sup> number incrementing by 2 with each iteration'.

### Coding Sample: (Custom Iterator)

```
# Creating a Custom Range
def cust_range(start_val, end_val, step_val):
    while start_val <= end_val:
        yield start_val
        start_val += step_val

for x in cust_range(5, 25, 5):
    print(x)
```

### Output:

```
5
10
15
20
25
```

In the above example, 'def' statement is used to create a custom Python Function that takes 3 parameters – start value, end value and step value (much like the built-in 'range' function).

**'yield' Statement:** The custom function 'yield's a value to the 'for' loop that mimics the working of the built-in 'range' function. 'yield' is a special Python function (different than 'return') that returns the value specified, waits for the receiver to receive the value and retruns back to processing in the same state. (whereas, 'return' statement returns only once and does not retain state of the processing to come back to the function).

**Break, Continue and Pass Statements:** The 'break', 'continue' and 'pass' statements provided by Python are intervention commands to change the process flow of the loop.

**Break Statement:** The 'break' statement forces the control to come out of the 'for' loop it is part of.

### Coding Sample: (Break statement)

```
# Example of 'break'ing away from a loop
for x in range(5,31,5):
    print(x, " is being processed")
    if x == 20:
        print("Breaking away..")
```

break

#### Output:

```
5 is being processed
10 is being processed
15 is being processed
20 is being processed
Breaking away..
```

In the above example, the loop was initially designed to iterate till it prints 30. However, for some reason, we want to discontinue when the value reaches 20. The 'if' condition with the break provides for that requirement.

Note that if there are multiple levels of nesting in the loops, then the 'break' statement comes out of the loop that is directly part of, and not from its outer layers of nesting.

Note that for the sake of simplicity in the examples, we are using static values like 5, 31, 5 as parameters to the functions we are using and for variable assignment. In real life scenarios, most of the values handled with programs will be dynamic.

**Continue Statement:** The 'continue' statement forces the loop to go back to the top of the loop it is part of from the 'continue' point and not to execute the instructions following the 'continue' statement or point.

#### Coding Sample: (Continue statement)

```
# Example of 'continuing' into a loop
for x in range(5,31,5):
    if x == 20:
        print("Continue to the top of the loop")
        continue
    print(x, " is being processed")
```

#### Output:

```
5 is being processed
10 is being processed
15 is being processed
Continue to the top of the loop
25 is being processed
30 is being processed
```

Note that as we asked Python to 'continue' when it encounters 20, it skipped the subsequent print statement and went back to the top. As a result 20 was not printed.

**Pass Statement:** The 'pass' statement in a loop is actually a null statement. It is usually used as a placeholder for any future changes to be made at the same place.

### Coding Sample: (Pass statement)

```
# Example of 'pass' in a loop
for x in range(5,31,5):
    if x == 20:
        print("Continue to the top of the loop")
        pass
    print(x, " is being processed")
```

### Output:

```
5 is being processed
10 is being processed
15 is being processed
Continue to the top of the loop
20 is being processed
25 is being processed
30 is being processed
```

As we compare the above result with the one prior to it ('continue'), we observe that there was no effect taken in the loop due to the 'pass' statement. Python continued ignoring 'pass'.

## Python Collection Data Types

### Python Lists

A List in Python is a collection object. Lists can hold a sequence of items or elements which can be of same or different data types including other collection objects. Lists elements can be accessed using the Index or relative position numbers starting from the start or end of the List.

Lists are mutable, i.e. they can be changed anytime after they are created.

Lists are a commonly used data structure in Python. Lists are used in scenarios when an array of related data (e.g. Names of customers to be printed together or daily transaction values in a bank for the purpose of various calculations or analysis) are to be held in memory during the runtime of the program, when the data are to be iterated through, are to be displayed together, etc.

Sample Representation of a List:

Index	1	2	3	4	5	6
Value	'Jimi'	5000	True	3.8	2019	300

### Coding Sample:

```
# Create our first List
lst1 = [1,2,3,4]
print(lst1)
```

### Output:

```
[1, 2, 3, 4]
```

In the following example, we are creating a List with multiple types of data. After the List is created, we access and update the value of an element and display that to demonstrate mutability.

#### Coding Sample:

```
# Create a List with Multiple types of Data and Change them after creation
lst1 = ['Jimi', 5000, True, 3.8, 2019, 300]
print(lst1)
lst1[1] = 7000 # Update the 2nd element of the List
print("Demonstrate Mutability: ", lst1)
```

#### Output:

```
['Jimi', 5000, True, 3.8, 2019, 300]
Demonstrate Mutability: ['Jimi', 7000, True, 3.8, 2019, 300]
```

#### Accessing Lists

List elements can be accessed using a relative index value that can be counted from the start (using a positive index number) or at the end (using a negative index number).

To access the 3<sup>rd</sup> element of a List with 6 elements, we have to use Listname[3]

The same can be achieved by counting the position number from the end as Listname[-4]

#### Coding Sample:

```
# Access Lists with relative positioning
lst1 = ['Jimi', 5000, True, 3.8, 2019, 300]
print("Print the 3rd Element of lst1: ", lst1[2]) # Relative indexing from the start
print("Print the 3rd Element of lst1: ", lst1[-4]) # Relative indexing from the end
```

#### Output:

```
Print the 3rd Element of lst1: True
Print the 3rd Element of lst1: True
```

#### Slicing Lists

We can access a cross section of a List by providing a start and an end position.

Syntax: Listname[start:end]

In the next example, we see the following scenarios:

- One where both the start and end positions are specified,
- One where only the end position is specified (List assumes default start as 0)
- One in which only the start position is specified (List assumes default end as all)

#### Coding Sample:

```
# Access cross section of Lists
lst1 = ['Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni']
```

```
print(lst1[1:5]) # Print elements 2 to 5
print(lst1[:5]) # Print elements start to 5 - default start 0
print(lst1[1:]) # Print all elements starting from 2nd - default end is all
```

**Output:**

```
['Gilchrist', 'Waugh', 'Dravid', 'Pollock']
['Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock']
['Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni']
```

**Striding while Slicing**

Lists support stepping multiple items while accessing a cross section by the use of a stride parameter along with the start and end positions. The stride parameter will determine how many elements to jump while printing the next element in the cross section.

**Coding Sample:**

```
# Access cross section of Lists with stride
lst1 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14]
print(lst1[1:12:2]) # Print elements 2 to 12 jumping 2 elements at a time
```

**Output:**

```
[2, 4, 6, 8, 10, 12]
```

Note that the use of negative indexing is also possible while accessing cross sections and with stride.

**Adding Elements in a List**

Elements can be added into a List by using the append method (add one element) or extend method (add several elements).

In the following example, we append single elements into a List using the append method.

**Coding Sample:**

```
# Adding single elements in Lists
lst1 = [1,2,3,4,5,6,7,8,9,10]
for x in range(5):
    lst1.append(x) # Append one element each time to the List
print(lst1)      # Print the final List
```

**Output:**

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

In the following example, we extend the List by several elements by using the extend method.

**Coding Sample:**

```
# Adding multiple elements in Lists
```

```
lst1 = [1,2,3,4,5,6,7,8,9,10]
lst1.extend([11,12,13,14]) # Extend the List by several elements
print(lst1)                # Print the final List
```

**Output:**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

We can also use operators to add two Lists. The += operator adds the right operand List to the left operand (similar to a numeric operation).

**Coding Sample:**

```
# Adding multiple elements in Lists -- another way
lst1 = [1,2,3,4,5,6,7,8,9,10]
lst2 = [11,12,13,14]
lst1 += lst2 # Add lst2 to lst1
print(lst1)  # Print the final List
```

**Output:**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

**Removing an Element from a List**

We can remove elements using the pop method. When a parameter is not specified, by default pop removes the last element in-place in the List. We can also mention an index position as a parameter for Python to remove the List element at the given position.

**Coding Sample:**

```
# Removing an element using the pop method
lst1 = [1,2,3,4,5,6,7,8,9,10]
lst1.pop() # Removes the last element in-place
print("After pop: ", lst1)
lst1.pop(3) # Removes the specified element in-place
print("After pop of 4th element: ", lst1)
```

**Output:**

```
After pop: [1, 2, 3, 4, 5, 6, 7, 8, 9]
After pop of 4th element: [1, 2, 3, 5, 6, 7, 8, 9]
```

The remove method can be used to remove an element of specified value.

**Coding Sample:**

```
# Removing an element using the remove method
lst1 = ['Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni']
lst1.remove('Waugh') # Removes the specified element in-place
print(lst1)
```

### Output:

```
['Gayle', 'Gilchrist', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni']
```

The del method can be used to remove multiple values.

### Coding Sample:

```
# Removing an element using the del keyword
lst1 = ['Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni']
del lst1[2:4] # Removes the elements at the specified index range
print(lst1)
```

### Output:

```
['Gayle', 'Gilchrist', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni']
```

### Common List Methods Summary

Method	Description
append	Append an item at the end of the List
extend	Extend the List with all the items in an iterable such as another List, array
insert	Insert an element in a given position
remove	Remove the first element in the List matching the value passed as parameter
pop	Drop the last element (by default) or drop the element at the specified index.
clear	Clears the entire List
index	Returns the zero-based (positive) index of the first element matching the value of the parameter.
count	Returns the number of times the passed value appears in the List
sort	Sort the List (in-place) in ascending (default) descending if specified as reverse=True as the parameter
reverse	Reverses the order of elements
copy	Make a physically separate copy

### Working with Tuples

Tuples are also collection objects in Python similar to Lists.

A comparison between Tuples and Lists:

Tuple	List
Tuples are Immutable, i.e. once created, Items in Tuples cannot be updated.	List Items are updateable or mutable.
Traversing through Tuples are computationally faster	Traversing through Lists are slower than Tuples
Usage: Enclosed in round brackets	Enclosed in square brackets

Use Tuple when you are sure not to have to change the Data after creation	Use when you expect to update the items values during runtime
Both are collection or sequence of elements or items like an array	
Each Item in a Tuple or List can of any data type.	
Elements both in a Tuple or List can be accessed through Indexes	

## Tuple Immutability

Please note that we are saying Tuple items are Updateable, this means the following:

This operation to update an element is not allowed:

### Coding Sample:

```
# Tuple Immutability
tup1 = (1,2,3,4,5)
tup1[2] = 4    # Not allowed
```

### Output:

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-2-e16bb25e1f4f> in <module>
      1 # Tuple Immutability
      2 tup1 = (1,2,3,4,5)
----> 3 tup1[2] = 4

TypeError: 'tuple' object does not support item assignment
```

But this is allowed:

### Coding Sample:

```
# Tuple Immutability
tup1 = (1,2,3,4,5)
print("Value of first object: ", tup1)
print("Memory Address of first object: ", id(tup1))
tup1 = (4,5,6,7,8)
print("Value of second object: ", tup1)
print("Memory Address of second object: ", id(tup1))
tup1
```

### Output:

```
Value of first object: (1, 2, 3, 4, 5)
Memory Address of first object: 112150118128
Value of second object: (4, 5, 6, 7, 8)
Memory Address of second object: 112150298992
```

In the second example above, when we try to create the Tuple again, Python does not throw any error. This is because in the second time, Python creates a completely different Tuple object at a different



memory location and tup1 will continue to refer to the second object – refer to the memory addresses printed to demonstrate that. But, when we try to update an item in an existing element by accessing it through an index value, that is not allowed due to the immutable nature of the Tuple object.

Exception: If the Tuple contains a mutable object within itself, e.g. a List. The List within the Tuple will remain mutable while the rest of the Tuple will not.

## Creating Tuples

### Coding Sample:

```
# Creating Tuples in various ways
tup1 = () # Create an empty Tuple
print("Contents of an empty Tuple: ", tup1)
tup1 = (1,2,3,4,5)
print("Contents of a simple Tuple: ", tup1)
tup1 = (1,2,(5,6,7),4,5)
print("Contents of a Nested Tuple: ", tup1)
tup1 = (1,2,(5,6,7),4,[9,8,7])
print("Contents of a Tuple having a Nested List: ", tup1)
tup1 = ('Rohit', 'Karnataka', True, 2000)
print("Contents of a Tuple having mixed Data Types: ", tup1)
```

### Output:

```
Contents of an empty Tuple: ()
Contents of a simple Tuple: (1, 2, 3, 4, 5)
Contents of a Nested Tuple: (1, 2, (5, 6, 7), 4, 5)
Contents of a Tuple having a Nested List: (1, 2, (5, 6, 7), 4, [9, 8, 7])
Contents of a Tuple having mixed Data Types: ('Rohit', 'Karnataka', True, 2000)
```

We can also create Tuples from Lists, String and Dictionaries using the **'tuple'** function. The tuple function converts these other DataTypes into Tuples.

### Coding Sample:

```
# Creating Tuple using the tuple function
lst1 = [1,2,3,4,5] # Create a List
tup1 = tuple(lst1)
print("Print Tuple Created from a List", tup1)
dict1 = {'Name': 'Robert', 'Salary': 2000} # Dictionary are key:value pairs separated by comma
tup2 = tuple(dict1) # Note that the Tuple is created with the values of the Dictionary Indexes
print("Print Tuple Created from a Dictionary", tup2)
str1 = "Christiano Ronaldo"
tup3 = tuple(str1) # Tuple created from a String
print("Print Tuple Created from a String", tup3)
```

### Output:

```
Print Tuple Created from a List (1, 2, 3, 4, 5)
Print Tuple Created from a Dictionary ('Name', 'Salary')
Print Tuple Created from a String ('C', 'h', 'r', 'i', 's', 't', 'i', 'a', 'n', 'o', ' ', 'R', 'o', 'n', 'a', 'l', 'd', 'o')
```

## Accessing Tuple Elements

Like Lists, Tuple elements can be accessed using a relative index value that can be counted from the start (using a positive index number) or at the end (using a negative index number).

To access the 3<sup>rd</sup> element of a Tuple with 6 elements, we have to use `TupleName[3]`

The same can be achieved by counting the position number from the end as `TupleName[-4]`

### Coding Sample:

```
# Access Tuples with relative positioning
tup1 = ('Jimi', 5000, True, 3.8, 2019, 300)
print("Print the 3rd Element of Tup1: ", tup1[2]) # Relative indexing from the start
print("Print the 3rd Element of Tup1: ", tup1[-4]) # Relative indexing from the end
```

### Output:

```
Print the 3rd Element of Tup1: True
Print the 3rd Element of Tup1: True
```

## Slicing Tuples

We can access a cross section of a Tuple by providing a start and an end position.

Syntax: `TupleName[start:end]`

In the next example, we see the following scenarios:

- One where both the start and end positions are specified,
- One where only the end position is specified (Tuple assumes default start as 0)
- One in which only the start position is specified (Tuple assumes default end as all)

### Coding Sample:

```
# Access cross section of Tuples
tup1 = ('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni')
print(tup1[1:5]) # Print elements 2 to 5
print(tup1[:5]) # Print elements start to 5 - default start 0
print(tup1[1:]) # Print all elements starting from 2nd - default end is all
```

### Output:

```
('Gilchrist', 'Waugh', 'Dravid', 'Pollock')
('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock')
('Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni')
```

## Striding while Slicing

Tuples support stepping multiple items while accessing a cross section by the use of a stride parameter along with the start and end positions. The stride parameter will determine how many elements to jump while printing the next element in the cross section.

### Coding Sample:

```
# Access cross section of Tuples with stride
tup1 = (1,2,3,4,5,6,7,8,9,10,11,12,13,14)
print(tup1[1:12:2]) # Print elements 2 to 12 jumping 2 elements at a time
```

### Output:

```
(2, 4, 6, 8, 10, 12)
```

Note that the use of negative indexing is also possible while accessing cross sections and with stride.

## Operations on Elements in a Tuple

As Tuples are immutable elements, the following operations are not possible on Tuples:

- Adding one more elements
- Removing one or more elements
- Updating any element

## Common Tuple Methods and Operations Summary

Method	Description
index	<p>Returns the zero-based (positive) index of the first element matching the value of the parameter.</p> <pre>tup1 = (1, 3, 7, 8, 7, 5) print(tup1.index(8))</pre> <p>Returns - 3</p>
count	<p>Returns the number of times the a value passed as a parameter appears on the Tuple.</p> <pre>tup1 = (1,2,3,4,5,6,7,8,9,10,11,12,13,14) print(tup1.count(5))</pre> <p>Returns - 1</p>
any	<p>The any method returns True if any of the elements is True.</p> <pre>tup1 = (1,2,3,4,5,6,7,8) print(any(tup1))</pre>

	<p>Returns - True</p>
all	<p>Returns True if all the elements in the Tuple are True.</p> <pre>tup1 = (1,2,3,4,0,6,7,8) print(all(tup1))</pre> <p>Returns - False</p>
Enumerate	<p>The Enumerate method takes in an iterable (Tuple in this case), adds a sequence number to it and returns an Enumerate object.</p> <pre>tup1 = ('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni') print(list(enumerate(tup1)))</pre> <p>Returns –</p> <pre>[(0, 'Gayle'), (1, 'Gilchrist'), (2, 'Waugh'), (3, 'Dravid'), (4, 'Pollock'), (5, 'Tendulkar'), (6, 'Gavaskar'), (7, 'Dhoni')]</pre> <p>Note that the Enumerate object had to be converted into a List to print it.</p>
Zip	<p>Zip is a built in Python function that takes in two sequences and returns a combined pair set.</p> <pre>tup1 = (1,2,3,4,5) tup2 = ('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock') print(list(zip(tup1, tup2)))</pre> <p>Returns –</p> <pre>[(1, 'Gayle'), (2, 'Gilchrist'), (3, 'Waugh'), (4, 'Dravid'), (5, 'Pollock')]</pre> <p>Note, that the zip function returns a zip object, that needs to be converted into a List object for printing.</p>
Sorted	<p>Returns a sorted List of the passed Tuple.</p> <pre>res = sorted(('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock')) print("Sorted: ", res) rev = sorted(('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock'), reverse=True) print("Reverse Sorted: ", rev)</pre> <p>Returns –</p> <pre>Sorted: ['Dravid', 'Gayle', 'Gilchrist', 'Pollock', 'Waugh'] Reverse Sorted: ['Waugh', 'Pollock', 'Gilchrist', 'Gayle', 'Dravid']</pre>
reversed	<p>Returns a reversed iterator of the sequence that is passed as parameter.</p> <pre>tup1 = ('Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock') res = list(reversed(tup1)) print("Reversed: ", res)</pre> <p>Returns -</p>

	Reversed: ['Pollock', 'Dravid', 'Waugh', 'Gilchrist', 'Gayle']  The Returned Iterator has to be converted into a list to print the values.
Max	Returns the maximum value from the Tuple.  <code>tup1 = (1,2,3,4,5,6,7,8,9,10,11,12,13,14)</code> <code>print(max(tup1))</code>  Returns - 14
Min	Returns the maximum value from the Tuple.  <code>tup1 = (1,2,3,4,5,6,7,8,9,10,11,12,13,14)</code> <code>print(min(tup1))</code>  Returns - 1
Sum	Returns the maximum value from the Tuple.  <code>tup1 = (1,2,3,4,5,6,7,8,9,10,11,12,13,14)</code> <code>print(sum(tup1))</code>  Returns - 105

<< Parking – Slice, others >>

### Copying a Tuple

As Tuples are mutable, there cannot be much of a rationale to create a copy of a Tuple. The assignment operator '=' and 'deepcopy' works but they simply create a new reference to the same memory location.

### Coding Sample:

```
# Copy Using Deepcopy
from copy import deepcopy
tup1 = (1, 2, 3, 4, 5)
tup2 = deepcopy(tup1)
print("Copy: ", tup2)
print(id(tup1))
print(id(tup2))

# Copy using assignment
tup3 = (1, 2, 3, 4, 5)
tup4 = deepcopy(tup3)
print("Copy: ", tup4)
print(id(tup3))
print(id(tup4))
```

**Output:**

```
Copy: (1, 2, 3, 4, 5)
112150347216
112150347216
Copy: (1, 2, 3, 4, 5)
112150118128
112150118128
```

You can see that, after the “=” and “deepcopy”, the memory reference returned are the same as original.

## Dictionary

A Dictionary in Python is a sequence or collection of Key:Value pairs. A dictionary is expressed within curling braces {}. The key is called the Index and the values can be accessed using the Index value from any specific pair.

Dictionaries “values” are mutable, i.e. values against a key can be updated after the dictionary is created and can be of any data type. However, the “Key” cannot be updated (immutable). The Key cannot be duplicated also. i.e. there cannot be two keys in the same dictionary with the same name.

Dictionary can hold “values” as other collections such as a Numpy Array, List or Set.

Sample Representation of a List:

The following example shows how the key-value pairs are stored in Python.

## Creating Dictionaries

The following example demonstrate a few ways / types Dictionaries can be created.

The following example is of a straightforward way of assigning values to keys in a Dictionary.

**Coding Sample:**

```
# Create our First Dictionary
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'ATM_Location': 'HSR Layout', 'Acc_Type': 'Savings' ,
'Amount_Drawn': 8000}
print(dict_tran)
```

**Output:**

```
{'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'ATM_Location': 'HSR Layout', 'Acc_Type': 'Savings',
'Amount_Drawn': 8000}
```

In the following example, we are creating a dictionary having a List as one of the Values against a key.

**Coding Sample:**

```
# Create a Dictionary with a List as one of the values
dict_tran = {'Cust_id' : 'C0789', 'Cust_Trans': ['T001', 'T003', 'T006']}
print(dict_tran)
```

**Output:**

```
{'Cust_id': 'C0789', 'Cust_Trans': ['T001', 'T003', 'T006']}
```

In the following example, we are creating a dictionary using the dict method from a List containing Tuple of key, value pairs.

**Coding Sample:**

```
# Create a Dictionary from a set of Tuples in a List using the Dict Method
dict_tran = dict([('Cust_id', 'C0789'), ('Cust_Trans', 'T001')])
print(dict_tran)
```

**Output:**

```
{'Cust_id': 'C0789', 'Cust_Trans': 'T001'}
```

In the following example, we are creating an empty dictionary and building it by populating values against keys.

**Coding Sample:**

```
# Populate from an empty dictionary
dict_tran = {}
dict_tran['Cust_id'] = 'C0789'
dict_tran['Cust_Trans'] = 'T001'
print(dict_tran)
```

**Output:**

```
{'Cust_id': 'C0789', 'Cust_Trans': 'T001'}
```

**Accessing Dictionaries**

Dictionary elements / values can be accessed by using the Keys.

**Coding Sample:**

```
# Access Dictionary Elements by using 'Keys' to access the 'Values'
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
print("Printing Value of Tran_id: ", dict_tran['Tran_id'])
print("Printing Value of Cust_Id: ", dict_tran['Cust_Id'])
```

**Output:**

```
Printing Value of Tran_id: T0089
Printing Value of Cust_Id: C000876
```

The following example demonstrates accessing Values in a Nested dictionary. As mentioned earlier, Dictionary values can be collection objects including Dictionary objects. A nested value can be accessed by qualifying the nested level Key preceded by the upper level key.

**Coding Sample:**

```
# Access Dictionary Elements by using 'Keys' to access the 'Values' -- Nested Dictionaries
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Tran_info': {'Acc_Type': 'Savings', 'Amount_Drawn': 8000}}
print("Printing Value of Tran_id: ", dict_tran['Tran_id'])
print("Printing Value of Tran_info: ", dict_tran['Tran_info'])
print("Printing Value of Amount_Drawn: ", dict_tran['Tran_info']['Amount_Drawn'])
```

**Output:**

```
Printing Value of Tran_id: T0089
Printing Value of Tran_info: {'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
Printing Value of Amount_Drawn: 8000
```

**Adding Elements in a Dictionary**

Elements or new key:value pairs can be added into a Dictionary by assigning the value against the new key. The following example demonstrates.

**Coding Sample:**

```
# Adding an element (key:value) by value assignment against a new key
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
dict_tran['ATM_Location'] = 'Bengaluru'
print("Printing the updated dictionary: ", dict_tran)
```

**Output:**

```
Printing the updated dictionary: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings',
'Amount_Drawn': 8000, 'ATM_Location': 'Bengaluru'}
```

Another way of adding elements to a dictionary is to use the 'update' method of dictionary object. The 'update' method takes in a dictionary and adds it into the original dictionary.

**Coding Sample:**

```
# Adding an element (key:value) by using the update method of dictionary
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
dict_tran.update({'ATM_Location': 'Bengaluru', 'Amount_Given': 6000})
print("Printing the updated dictionary: ", dict_tran)
```

**Output:**

```
Printing the updated dictionary: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings',
'Amount_Drawn': 8000, 'ATM_Location': 'Bengaluru', 'Amount_Given': 6000}
```



### Removing an Element from a Dictionary

Removing an element from a dictionary requires identifying the element through the 'key'. The 'del' statement removed the element in-place.

In the following example, we are deleting the key:value pair identified with the key 'Acc\_Type'.

#### Coding Sample:

```
# Deleting an Element from a Dictionary
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
del dict_tran['Acc_Type']
print("Printing the updated dictionary: ", dict_tran)
```

#### Output:

```
Printing the updated dictionary: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Amount_Drawn': 8000}
```

We can also use the 'pop' method of dictionary and specify the 'key' to be removed. The pop keyword taken in the key, removed the key:value pair and returns the value.

#### Coding Sample:

```
# Deleting an Element from a Dictionary using 'pop'
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
popped = dict_tran.pop('Acc_Type')
print("Printing the updated dictionary: ", dict_tran)
print("What did I pop? ", popped)
```

#### Output:

```
Printing the updated dictionary: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Amount_Drawn': 8000}
What did I pop? Savings
```

### Trapping KeyError in a Dictionary

Most operations on a Python Dictionary requires a 'key' to access the value, hence there is probability of specifying a wrong key. A dictionary operation throws a 'KeyError' when a wrong key is specified and it is advisable to trap it, especially if the key is provided dynamically in the program.

In the following example, we use the Python exception trapping mechanism of a 'try-except' block. The code to be executed needs to be placed inside the 'try' block whereas the exception handling 'code' needs to be placed inside the 'except' block.

#### Coding Sample:

```
# Deleting an Element from a Dictionary using 'pop' while trapping 'KeyError'
dict_tran = {'Tran_id' : 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
try:
    popped = dict_tran.pop('Acc_Type')
    print("Printing the updated dictionary: ", dict_tran)
    print("What did I pop? ", popped)
except KeyError:
```

```
print("Key Specified is not found in dict_tran")
```

#### Output:

```
Printing the updated dictionary: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Amount_Drawn': 8000}
What did I pop? Savings
```

As a valid key is provided, the actual code (one within the 'try' block) is performed. We would now pass a wrong key with the 'pop' method.

#### Coding Sample:

```
# Deleting an Element from a Dictionary using 'pop' while trapping 'KeyError'
dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
try:
    popped = dict_tran.pop('Acc_Typ1')
    print("Printing the updated dictionary: ", dict_tran)
    print("What did I pop? ", popped)
except KeyError:
    print("Key Specified is not found in dict_tran")
```

#### Output:

```
Key Specified is not found in dict_tran
```

Python traps the KeyError generated due to the use of the 'try-except' block and in the above example, the code inside the 'except' block executes.

#### Other Common Dictionary Methods Summary

Method	Description
Clear	Deletes all elements in the Dictionary # 'Clear' the Dictionary dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_tran.clear() print(dict_tran) Outcome: {}
Copy	Returns a copy of the Dictionary # 'Copy' the Dictionary dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_copy = dict_tran.copy() print(dict_copy) Outcome: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
Fromkeys	Creates a dictionary from a specified sequence of keys. # Create a new Dictionary with a specified sequence of key-values

	<pre>dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} keys = {'Cust_Id', 'Acc_Type'} dict_copy = dict_tran.fromkeys(keys) print(dict_copy) Outcome: {'Acc_Type': None, 'Cust_Id': None}</pre>
Get	<p>Returns the value of a key specified.</p> <pre># 'get' the value of a specified key dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_get = dict_tran.get('Cust_Id') print(dict_get) Outcome: C000876</pre>
Items	<p>Returns all the key:value pairs in the dictionary in the form of a List of Tuples.</p> <pre># Return the key:value pairs from the dictionary as a list of key:value Tuples dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_items = dict_tran.items() print(dict_items) Outcome: dict_items([('Tran_id', 'T0089'), ('Cust_Id', 'C000876'), ('Acc_Type', 'Savings'), ('Amount_Drawn', 8000)])</pre>
Keys	<p>Returns a list containing all the keys of the Dictionary.</p> <pre># Return all the Keys of the dictionary dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_keys = dict_tran.keys() print(dict_keys) Outcome: dict_keys(['Tran_id', 'Cust_Id', 'Acc_Type', 'Amount_Drawn'])</pre>
Values	<p>Returns all the values</p> <pre># Return all the Values of the dictionary dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_vals = dict_tran.values() print(dict_vals) Outcome: dict_values(['T0089', 'C000876', 'Savings', 8000])</pre>
Popitem	<p>Removes the last Inserted key:value pair</p> <pre># Deleting the last Element from a Dictionary using 'popitem' dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} popped = dict_tran.popitem() print("Printing the updated dictionary: ", dict_tran) print("What did I pop? ", popped) Outcome:</pre>

	Printing the updated dictionary: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings'} What did I pop? ('Amount_Drawn', 8000)
Setdefault	Insert a specified key:value pair. If the Key exists, returns the value. # Insert a key:value pair. If the pair exists, return the value dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000} dict_tran.setdefault('New_field', 6766) print("Printing the updated dictionary - pass 1: ", dict_tran) ret = dict_tran.setdefault('Amount_Drawn', 6766) print("Printing the updated dictionary - pass 2: ", dict_tran) print("Checking the returned value: ", ret) Outcome: Printing the updated dictionary - pass 1: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000, 'New_field': 6766} Printing the updated dictionary - pass 2: {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000, 'New_field': 6766} Checking the returned value: 8000 There are two examples above – in pass 1, we are passing a new:value pair and the dictionary is updated with them. In pass 2, we are passing an existing key with a new value. In this case, the existing key:value is not updated and the existing value is returned.

## Iterating through a Dictionary

Traversing through a dictionary is similar to traversing through other iterators in Python. While traversing through a dictionary in a for loop, python returns the current Key. The value as usual can be extracted.

### Coding Sample:

```
# Iterating through a Dictionary
dict_tran = {'Tran_id': 'T0089', 'Cust_Id': 'C000876', 'Acc_Type': 'Savings', 'Amount_Drawn': 8000}
for x in dict_tran:
    print("Key: ", x, " -- Value: ", dict_tran[x])
```

### Output:

```
Key: Tran_id -- Value: T0089
Key: Cust_Id -- Value: C000876
Key: Acc_Type -- Value: Savings
Key: Amount_Drawn -- Value: 8000
```

In the above example, we print 'x', which is the key of the current pair of the dictionary in the for loop. 'x' is then used as the key to extract and print the value against it.

## Set

Sets are collection objects containing unique mutable elements. Set objects are used in Python the purpose of carrying out special set operations such as finding unions, differences, intersections, symmetric

intersections, etc between datasets. Sets are also used for membership testing, i.e. testing for whether an element is part of the set or not. Another key property of sets is that elements in a set are unordered. Hence, it is not possible to access elements in a set using indexes, or carrying out other sequence operations like slicing, unlike other collection objects in Python such as Lists, Tuples, Dictionaries and Numpy Arrays.

### Creating Sets

Sets are created by using the set method. In the following example, we create a set using the set method. Notice that the order in which the input elements are passed are not same as in the output. This proves the theory of sets not being ordered collections.

#### Coding Sample:

```
# Creating a Set
set1 = set(['Gayle', 'Gilchrist', 'Waugh', 'Dravid', 'Pollock', 'Tendulkar', 'Gavaskar', 'Dhoni'])
print(set1)
```

#### Output:

```
{'Gavaskar', 'Gayle', 'Waugh', 'Dravid', 'Gilchrist', 'Dhoni', 'Tendulkar', 'Pollock'}
```

### Finding the Intersections of two Sets

Intersections return the elements that are common across the two sets used as operands. The following code demonstrates two ways intersections between two sets are found.

1. Use the '&' operator that returns the elements common between set1 & set2.
2. Use the Intersection method of set object.

At the end, we compare the result sets (int1 and int2) to find if they are the same.

#### Coding Sample:

```
# Finding an Intersection
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
set2 = set(['a', 'b', 'c', 'e', 'f', 'h', 'm', 't', 'q'])
print("Set 1: ", set1)
print("Set 2: ", set2)
int1 = set1 & set2    # Use the '&' - intersection operator
int2 = set1.intersection(set2)  # Use the intersection method of set object
print("Intersection using the & Operator: ", int1)
print("Intersection using the Intersection method: ", int2)
if int1 == int2:
    print("The Intersections are same")
```

#### Output:

```
Set 1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}
Set 2: {'e', 'c', 'm', 'f', 'h', 'a', 'q', 't', 'b'}
Intersection using the & Operator: {'c', 'e', 'f', 'h', 'a', 'b'}
Intersection using the Intersection method: {'c', 'e', 'f', 'h', 'a', 'b'}
The Intersections are same
```

## Finding the Union of two Sets

Union return unique occurrences of all elements among the two sets used as operands.  
The following code demonstrates two ways unions between two sets are found.

3. Use the '|' operator that returns the union between set1 & set2. Interpreted as 'return all which are in either set1 or set2 plus the unique of the ones common across both sets'.
4. Use the Union method of set object.

At the end, we compare the result sets (uni1 and uni2) to find if they are the same.

### Coding Sample:

```
# Finding an Union
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
set2 = set(['a', 'b', 'c', 'e', 'f', 'h', 'm', 't', 'q'])
print("Set 1: ", set1)
print("Set 2: ", set2)
uni1 = set1 | set2 # Use the '|' - union operator
uni2 = set1.union(set2) # Use the union method of set object
print("Union using the | Operator: ", uni1)
print("Union using the Union method: ", uni2)
if uni1 == uni2:
    print("The Unions are same")
```

### Output:

```
Set 1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}
Set 2: {'e', 'c', 'm', 'f', 'h', 'a', 'q', 't', 'b'}
Union using the | Operator: {'e', 'c', 'm', 'f', 'h', 'a', 'q', 'g', 't', 'b', 'd'}
Union using the Union method: {'e', 'c', 'm', 'f', 'h', 'a', 'q', 'g', 't', 'b', 'd'}
The Unions are same
```

## Finding the Difference of two Sets

The difference operation on sets, set1 and set2 return all elements that are found in set1 but not there in set2.

The following code demonstrates two ways differences between two sets are found.

1. Use the '-' operator that returns the difference between set1 & set2. Interpreted as 'return all elements that are found in set1 but not in set2'.
2. Use the difference method of set object.

At the end, we compare the result sets (dif1 and dif2) to find if they are the same.

### Coding Sample:

```
# Finding a Difference
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
set2 = set(['a', 'b', 'c', 'e', 'f', 'h', 'm', 't', 'q'])
print("Set 1: ", set1)
print("Set 2: ", set2)
dif1 = set1 - set2    # Use the '-' (Difference operator)
dif2 = set1.difference(set2)    # Use the Difference method of set object
print("Difference using the - Operator: ", dif1)
print("Difference using the Difference method: ", dif2)
if dif1 == dif2:
    print("The Differences are same")
```

### Output:

```
Set 1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}
Set 2: {'e', 'c', 'm', 'f', 'h', 'a', 'q', 't', 'b'}
Difference using the - Operator: {'g', 'd'}
Difference using the Difference method: {'g', 'd'}
The Differences are same
```

### Finding the Symmetric Difference of two Sets

The symmetric difference operation on sets, set1 and set2 return all elements that are found in set1 and set2 but not in both.

The following code demonstrates two ways symmetric differences between two sets are found.

3. Use the '^' operator that returns the symmetric difference between set1 & set2. Interpreted as 'return all elements that are found in set1 and set2 but not in both'.
4. Use the symmetric\_difference method of set object.

At the end, we compare the result sets (dif1 and dif2) to find if they are the same.

### Coding Sample:

```
# Finding Symmetric Difference
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
set2 = set(['a', 'b', 'c', 'e', 'f', 'h', 'm', 't', 'q'])
print("Set 1: ", set1)
print("Set 2: ", set2)
dif1 = set1 ^ set2    # Use the '^' (Symmetric Difference operator)
dif2 = set1.symmetric_difference(set2)    # Use the symmetric_difference method of set object
print("Symmetric Difference using the ^ Operator: ", dif1)
print("Symmetric Difference using the symmetric_difference method: ", dif2)
if dif1 == dif2:
```

```
print("The Symmetric Differences are same")
```

**Output:**

```
Set 1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}
Set 2: {'e', 'c', 'm', 'f', 'h', 'a', 'q', 't', 'b'}
Symmetric Difference using the ^ Operator: {'m', 'q', 'g', 't', 'd'}
Symmetric Difference using the symmetric_difference method: {'m', 'q', 'g', 't', 'd'}
The Symmetric Differences are same
```

**Adding an element in a Set**

Adding an element into a Set can be achieved through the add method of set. Following example adds an element into set1.

Note that adding a set to another set is achieved by the union method or '&' operator which we have learnt before. A set addition is essentially an union as sets only contain unique elements.

**Coding Sample:**

```
# Add an element in a Set
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print("Set 1: ", set1)
set1.add('i') # Use the add method of set object to add the new element 'i'
print("Updated Set1: ", set1)
```

**Output:**

```
Set 1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}
Updated Set1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd', 'i'}
```

**Test of Membership**

We can check *if an element is part of a set* by using the 'in' or 'not in' statements. A Boolean is returned based on whether or not the specified element is part of the set in question.

**Coding Sample:**

```
# Check if element 'c' is part of set1
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print("Whether c is present in Set1: ", 'c' in set1)
print("Whether c is present in Set1...negative way: ", 'c' not in set1)
```

**Output:**

```
Whether c is present in Set1: True
Whether c is present in Set1...negative way: False
```

The condition whether 'c' is in set1 is checked twice in the above example by once using 'in' and another time using the 'not in' clauses. The return values are different but answers the asked question correctly. We can also check *whether a set is a subset or superset or another specified set* by the use of 'issubset' or 'issuperset' set methods.



## Coding Sample:

```
# Check for subset or superset
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
set2 = set(['b', 'e', 'f', 'h'])
print("Set 1: ", set1)
print("Set 2: ", set2)
print("Whether Set2 is a subset of set1: ", set2.issubset(set1))
print("Whether Set1 is a superset of set2: ", set1.issuperset(set2))
```

## Output:

```
Set 1: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}
Set 2: {'h', 'f', 'b', 'e'}
Whether Set2 is a subset of set1: True
Whether Set1 is a superset of set2: True
```

The 'issubset' method in the above example establishes whether every element in set2 is found in set1. Likewise, the 'issuperset' method establishes whether set2 is completely contained within set1, essentially the same condition as the above.

## Other Common Set Methods

Method	Description
Update	<p>The 'update' function updates multiple elements into a set.</p> <p># Adding Multiple Elements using the 'update' method</p> <pre>set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) set1.update(['w', 'x', 'm', 'a']) print(set1)</pre> <p>Outcome:</p> <pre>{'e', 'c', 'm', 'f', 'h', 'x', 'a', 'w', 'g', 'b', 'd'}</pre> <p>Please note that the 'union' and 'update' methods are functionally similar. The only difference is that 'union' returns a new set while 'update' updates the set in place.</p>
Len	<p>The 'len' function returns the number of elements in a set.</p> <p># Finding the number of elements in a Set</p> <pre>set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) print(len(set1))</pre> <p>Outcome:</p> <pre>8</pre>
Copy	<p>The 'copy' function creates a Shallow copy of the set.</p> <p>A "=" assignment operator creates only a reference (a different variable name on the same memory location). A new set created hence using "=" will change when the original set changes.</p> <p>To create a physically separate new copy, or a shallow copy, the 'copy' method has to be used.</p> <p># Creating a Shallow Copy</p> <pre>set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])</pre>

	<pre> set2 = set1.copy() print("Printing original set: ", set1) print("Printing the copied set: ", set2) print("Memory Address of Set1: ", id(set1)) print("Memory Address of Set2: ", id(set2)) if id(set1) != id(set2):     print("Shallow Copy") </pre> <p>Outcome:          Printing original set: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}          Printing the copied set: {'e', 'c', 'f', 'h', 'a', 'g', 'b', 'd'}          Memory Address of Set1: 4596733904          Memory Address of Set2: 4596735104          Shallow Copy</p> <p>In the above example, the memory address of the new set is different, hence establishing that it's a physically different copy than the original.</p>
remove	<p>The 'remove' method removes an element from the set. 'remove' raises a KeyError if the element is not present.</p> <pre> # Remove an element from the Set set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) set1.remove('d') print(set1) </pre> <p>Outcome:          {'e', 'c', 'f', 'h', 'a', 'g', 'b'}</p>
Discard	<p>Similar to 'remove'. Only difference is that it does not raise KeyError if the element is not present.</p>
Clear	<p>Removes all elements from the set</p>
Pop	<p>Removes an arbitrary element from the set and returns it.</p> <pre> # Pop an element from the Set set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) x = set1.pop() print("Popped Item: ", x) print("New Set: ", set1) </pre> <p>Outcome:          Popped Item: e          New Set: {'c', 'f', 'h', 'a', 'g', 'b', 'd'}</p>
intersection_update	<p>The Intersection_update method returns set1 keeping only those elements that are also found in set2.</p> <pre> # Intersection_update --- Return set1 keeping only those elements also found in set2 set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) set2 = set(['b', 'e', 'f', 'h', 'u', 'j']) set1.intersection_update(set2) print("Intersection Update: ", set1) </pre> <p>Outcome:          Intersection Update: {'h', 'f', 'b', 'e'}</p>
Difference_update	<p>The difference_update method returns those elements in set1 after removing those also found in set2.</p>

	<pre># Difference_update --- Return set1 after removing those elements also found in set2 set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) set2 = set(['b', 'e', 'f', 'h', 'u', 'j']) set1.difference_update(set2) print("Difference Update: ", set1) Outcome: Difference Update: {'c', 'a', 'g', 'd'}</pre>
Symmetric_difference_update	<pre>Updates set1 with elements found in both set1 or set2 but not both. # Symmetric_Difference_update --- Updates set1 keeping those elements found in Set1 and Set2 but not both set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']) set2 = set(['b', 'e', 'f', 'h', 'u', 'j']) set1.symmetric_difference_update(set2) print("Symmetric Difference Update: ", set1) Outcome: Symmetric Difference Update: {'c', 'u', 'a', 'j', 'g', 'd'}</pre>

## Iterating through a Set

Traversing through a set is similar to traversing through other iterators in Python. With each iteration, Python returns an element from the set till all elements are exhausted.

### Coding Sample:

```
# Traversing through a set
set1 = set(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
for x in set1:
    print(x)
```

### Output:

```
e
c
f
h
a
g
b
d
```

## Working with Python NumPy Library:

Data that we deal with comes in various shapes and form and originate from varied sources, however at the end all of them can be fundamentally represented as arrays of Numeric digits.

Some examples as follows:

- Data that are fundamentally numeric
  - Financial Transaction amounts
  - Your Monthly income

- Number of times you visited a website in last 7 days
- Sound: Stored as a one dimensional array of frequency vs time
- Images: Stored as two Dimensional numeric arrays representing brightness values of each pixel over an area
- Text: Can be represented as Numeric digits as well in various ways

Numeric representation of all kinds of data are essential to the heart of Data Analysis / Data Science as creation of effective models of predictions and decisions are dependent on all data being available in Numeric form. Note that Prediction Modelling can take only Numeric input variables.

Numpy is the foundational library of Python. NumPy provides for a N-dimensional array storage, powerful set of array handling functions and strong support of mathematical operations on arrays including Linear Algebra, **Fourier Transformation** and Random Number capabilities.

### Installation of NumPy:

If you are using Anaconda, NumPy comes pre-installed. However, just in case, you do not have the NumPy library installed, here is what you can do:

```
$ pip install numpy
or
$ brew install numpy
Or
$ conda install numpy
```

(Assuming you have pip, homebrew or conda already)

### Importing NumPy

You need to import the NumPy library in your Python code before using it:

```
import numpy
or
import numpy as np
```

The difference between the above two commands is that with the first command, you need to specify the full 'numpy' qualifier for all NumPy functions whereas in the second one, you can specify only 'np' as the qualifier – short and sweet.

### NumPy Arrays

Numpy supports a multidimensional homogenous array. The index of a numpy array is a Tuple of integers starting from 0. The number of dimensions is called the *rank* of the array. The shape of the array is also represented by the Tuple of integers denoting the length of each dimension.

## Creating Arrays:

A common way of creating a numpy array is from Python Tuples. For multiple dimensions, multiple Tuples are used.

### Coding Sample:

```
# Create a numpy Array from Tuple
arr = np.array([[1,2,3,4,5], [2,3,4,5,6]])
print(arr)
print(arr.shape)
```

### Output:

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

In the above example, we are creating a two dimensional array from two similar sized Tuples.

The shape attribute of the array returns the dimension, i.e. (2, 5) in this case.

The following set of examples elicit various ways of creating Numpy Arrays. Note that the Numpy array is always homogenous and will store data of the same type.

### Coding Sample:

```
# Create an array of all zeros
arr = np.zeros((4,4))
print(arr)
```

### Output:

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

### Coding Sample:

```
# Create an array of all ones
arr = np.ones((3,4))
print(arr)
```

### Output:

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

### Coding Sample:

```
# Create an array filling all elements with a specified value
arr = np.full((4,3), 9)
print(arr)
```

**Output:**

```
[[9 9 9]
 [9 9 9]
 [9 9 9]
 [9 9 9]]
```

**Coding Sample:**

```
# Create a 4x4 identity matrix
arr = np.eye(4)
print(arr)
```

**Output:**

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

**Coding Sample:**

```
# Create an array filled with random values
arr = np.random.random((3,3))
print(arr)
```

**Output:**

```
[[0.83039916 0.72889966 0.92177252]
 [0.4749109  0.81077047 0.55353536]
 [0.13588012 0.18089611 0.04976572]]
```

**Coding Sample:**

```
# Create an array with sequential values starting from 0 to a given range limit
arr = np.arange(30)
arr
```

**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])
```

**Coding Sample:**

```
# Create an array with sequential values starting from a given value to a final value and with a step value
arr = np.arange(0, 30, 2)
arr
```

**Output:**

```
array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

**Coding Sample:**

```
# Reshape the array to a new specified dimension
arr = np.arange(30)
arr = arr.reshape(5,6)
arr
```

**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]])
```

**Coding Sample:**

```
# Reshape the array to a new specified dimension -- a shorter way
arr = np.arange(30).reshape(5,6)
arr
```

**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]])
```

**Coding Sample:**

```
# Creating a string array
```

```
arr = np.array(['John', 'Robert'], ['Pranav', 'Tauseef'])
arr
```

### Output:

```
array(['John', 'Robert',
       ['Pranav', 'Tauseef']], dtype='<U7')
```

### Printing Array Attributes

Numpy provides several attributes of the Array Object. The following table Tuples some of the key attributes:

```
# Create an array of dimension 5,6 using the arange function.
arr = np.arange(30).reshape(5,6)
arr
```

Considering the above array is created, the Output column in the following table captures the output of the attribute named in the first column.

Attribute Name	Description	Output
T	Short for 'transpose'. Converts the array into it's transpose (rows into columns and columns into rows)	array([[ 0,  6, 12, 18, 24],        [ 1,  7, 13, 19, 25],        [ 2,  8, 14, 20, 26],        [ 3,  9, 15, 21, 27],        [ 4, 10, 16, 22, 28],        [ 5, 11, 17, 23, 29]])
dtype	Returns the data type of the array	dtype('int64')
real	Returns the real values of the array. Equivalent to print(arrayname).	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]])
size	Returns the size of the array	30
itemsize	Size of each element in bytes	8
nbytes	Total storage consumed by the array in bytes	240
ndim	Number of dimensions	2



shape	Shape of the array	(5,6)
-------	--------------------	-------

## Array Functions (Methods)

The following table Tuples some of the key array functions

Function Name	Parameters	Description
astype	Data type	Return the array after casting it in the specified Type
copy		Return a copy of the array
cumprod	Axis, Dtype	Return the cumulative product of the elements along the given axis.
cumsum	Axis, Dtype	Return the cumulative product of the elements along the given axis.
diagonal	Offset, axis1, axis2	Return specified Diagonals
dot	Second Array	Return the dot product of two arrays
fill	value	Fill the array with a specified value
flatten		Return the array flattened into one dimension
max		Return the maximum value from the array
mean		Return the mean of all elements in the array
min		Return the minimum value from the array
reshape	dimensions	Return the array reshaped to the mentioned dimensions
resize	dimensions	Change the shape of the array in-place
round	decimals	Return the array with each element rounded to the specified decimal
sort	axis	Sort the array, in-place along the specified axis
std	axis	Returns the standard deviation of the elements of the array along the specified axis
sum	axis	Returns the sum of the elements of the array along the specified axis
tofile	filename, separator	Write the array into the specified file using the separator

toTuple		Returns the array as a Tuple (nested Tuple in case of multidimensional array)
transpose		Returns the transpose of the array
var	axis	Returns the variance of the array along the given axis

The following code examples demonstrate some of the above functions in action:

#### Coding Sample:

```
# Return the dot product of two arrays
arr1 = np.arange(30).reshape(5,6)
arr2 = arr1.T
arr_res = arr1.dot(arr2)
print(arr_res)
```

#### Output:

```
[[ 55 145 235 325 415]
 [ 145 451 757 1063 1369]
 [ 235 757 1279 1801 2323]
 [ 325 1063 1801 2539 3277]
 [ 415 1369 2323 3277 4231]]
```

#### Coding Sample:

```
# Flatten an Array
arr1 = np.arange(30).reshape(5,6)
arr = arr1.flatten()
print(arr)
```

#### Output:

```
[ 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]
```

#### Coding Sample:

```
# Find the Min, Max, Mean, Standard Deviation and Variance of the elements
arr1 = np.arange(30).reshape(5,6)
minm = arr1.min()
maxm = arr1.max()
mean = arr1.mean()
std = arr1.std(axis=1) # Along Columns
```

```
var = arr1.var(axis=0) # Along Rows
print("Minimum: ", minm)
print("Maximum: ", maxm)
print("Mean: ", mean)
print("Standard Deviation: ", std)
print("Variance: ", var)
```

**Output:**

```
Minimum: 0
Maximum: 29
Mean: 14.5
Standard Deviation: [1.70782513 1.70782513 1.70782513 1.70782513 1.70782513]
Variance: [72. 72. 72. 72. 72. 72.]
```

**Coding Sample:**

```
# Create a copy
arr1 = np.arange(30).reshape(5,6)
arrc = arr1.copy()
print(arrc)
```

**Output:**

```
[[ 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]]
```

**Coding Sample:**

```
# Convert an Array to a Tuple / Nested Tuple
arr1 = np.arange(30).reshape(5,6)
arrlst = arr1.toTuple()
print(arrlst)
```

**Output:**

```
[[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]]
```

**Coding Sample:**

```
# Rounding elements
arr1 = np.random.random((3,3))
print("Array 1: \n", arr1)
arr2 = arr1.round(2)
```

```
print("Array 2: \n", arr2)
```

**Output:**

```
Array 1:
[[0.15820549 0.13700026 0.65842054]
 [0.57790003 0.21375053 0.53365548]
 [0.34936194 0.3160926  0.1099425 ]]
Array 2:
[[0.16 0.14 0.66]
 [0.58 0.21 0.53]
 [0.35 0.32 0.11]]
```

**Coding Sample:**

```
# Converting Data Types with astype
arr1 = np.arange(30).reshape(5,6)
print("Array 1: \n", arr1)
arr2 = arr1.astype(str) # Convert elements from Int to String
print("Array 2: \n", arr2)
arr3 = arr2.astype(int) # Convert String elements to Int again
print("Array 3: \n", arr3)
```

**Output:**

```
Array 1:
[[ 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 2:
[['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 3:
[[ 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]]
```

## Accessing Array Elements

Array elements are accessed through the use of indexes. Array index position numbers start from 0. The 2<sup>nd</sup> row, 3<sup>rd</sup> columns will be denoted as [1,2]. For an element to be accessed, it's position along the dimensions have to be specified within square brackets.

The full syntax for index accessing of arrays:

*Arrayname[starting row position: ending row position, starting column position: ending column position]*

Let's understand the mechanism and syntax through the following examples.

### Coding Sample:

```
# Accessing Array Elements - 2nd row, 3rd column
arr1 = np.arange(30).reshape(5,6)
print("Array 1: \n", arr1)
print("Element at the 2nd row, 3rd column: ", arr1[1, 2])
```

### Output:

```
Array 1:
[[ 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]]
Element at the 2nd row, 3rd column: 8
```

### Coding Sample:

```
# Accessing Array Elements
arr1 = np.arange(30).reshape(5,6)
print("Array 1: \n", arr1)
print("Element at the 2nd row, 3rd column: ", arr1[1,2])
print("All Elements on the 2nd row: ", arr1[1]) # Print the 2nd row
print("Element on the 3rd column: ", arr1[:, 2]) # Print the 3rd column
print("All Elements: \n", arr1[:,:]) # Another way of doing this is not mentioning the indexes
```

### Output:

```
Array 1:
[[ 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]]
Element at the 2nd row, 3rd column: 8
All Elements on the 2nd row: [ 6  7  8  9 10 11]
Element on the 3rd column: [ 2  8 14 20 26]
```

All Elements:

```
[[ 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]]
```

The following example demonstrates

- How to access a cross section of elements in a given row: By mentioning the row position + start:end positions of elements on that row (`arr1[1, 2:5]`)
- How to access a cross section of elements in a given column: By mentioning the column position + start:end positions of elements on that column (`arr1[2:5, 2]`)
- How to access a cross section of elements in a given row and column (like a tabular cross section from the array): By mentioning the row start:end positions + column start:end positions (`arr1[2:5, 2:5]`)

#### Coding Sample:

```
# Accessing Array Elements
arr1 = np.arange(30).reshape(5,6)
print("Array 1: \n", arr1)
print("Elements on the 2nd row, 3rd to 5th positions: ", arr1[1, 2:5])
print("Elements on the 3rd column, 3rd to 5th row positions: ", arr1[2:5, 2])
# Elements on the 2nd row, 3rd to 5th positions, Elements on the 3rd column, 3rd to 5th row positions
print("Elements on the 2nd row, 3rd to 5th positions , Elements on the 3rd column, 3rd to 5th row positions :")
print(arr1[2:5, 2:5])
```

### Output:

```

Array 1:
[[ 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]]
Elements on the 2nd row, 3rd to 5th positions: [ 8  9 10]
Elements on the 3rd column, 3rd to 5th row positions: [14 20 26]
Elements on the 2nd row, 3rd to 5th positions , Elements on the 3rd column, 3rd to 5th row positions :
[[14 15 16]
 [20 21 22]
 [26 27 28]]

```

The following example elicits how elements or cross sections can be extracted from a 3-Dimensional array.

### Coding Sample:

```

# Accessing Array Elements in a 3D array
arr1 = np.arange(64).reshape(4,4,4)
print("Array 1: \n", arr1)
print("Elements at the positions: first dimension: 0, second dimension: 3, third dimension: 2 :")
print(arr1[0, 3, 2])
print("Elements at the positions: first dimension: 0, second dimension: positions 1st to 2nd, third dimension: 2 :")
print(arr1[0, 0:2, 2])

```

### Output:

```

Array 1:
[[[ 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 30 31]]

 [[32 33 34 35]
 [36 37 38 39]
 [40 41 42 43]
 [44 45 46 47]]

 [[48 49 50 51]
 [52 53 54 55]
 [56 57 58 59]]

```

```
[60 61 62 63]]]
```

Elements at the positions: first dimension: 0, second dimension: 3, third dimension: 2 :

```
14
```

Elements at the positions: first dimension: 0, second dimension: positions 1st to 2nd, third dimension: 2 :

```
[2 6]
```

From the above examples, now we can derive the generalised syntax of accessing an n-dimensional array as:

*Arrayname[starting D1 position: ending D1 position, starting D2 position: ending D2 position, starting D3 position: ending D3 position, ..... , starting D-n position: ending D-n position]*

### Iterating through a Numpy array

The following examples demonstrate how to traverse through rows or columns in an array.

#### Coding Sample:

```
# Iterating through Rows
arr1 = np.arange(30).reshape(5,6)
print("Printing Through Rows:")
for row in arr1:    # Returns one row at a time - default
    print(row)
# Iterating through Rows
print("Printing Through Columns:")
cols = arr1.shape[1] # Returns number of columns
for n in range(cols):
    print(arr1[:,n]). # Prints all elements on n-th column
```

#### Output:

```
Printing Through Rows:
[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]
Printing Through Columns:
[ 0  6 12 18 24]
[ 1  7 13 19 25]
[ 2  8 14 20 26]
[ 3  9 15 21 27]
[ 4 10 16 22 28]
[ 5 11 17 23 29]
```

<https://numpy.org/devdocs/user/quickstart.html#array-creation>



## Stacking arrays

The `hstack` and `vstack` functions of Numpy are used to stack multiple arrays together. The stacking can be horizontal (using `hstack` function) or vertical (using `vstack` function).

In the following example, two arrays are stacked or joined together horizontally.

### Coding Sample:

```
# Horizontal Stacking of Two Arrays
arr1 = np.arange(30).reshape(5,6)
print("Array 1 :\n", arr1)
arr2 = np.arange(100, 130).reshape(5,6)
print("Array 2 :\n", arr2)
print("Stacked Array :\n", np.hstack((arr1, arr2)))
```

### Output:

```
Array 1 :
[[ 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 2 :
[[100 101 102 103 104 105]
 [106 107 108 109 110 111]
 [112 113 114 115 116 117]
 [118 119 120 121 122 123]
 [124 125 126 127 128 129]]
Stacked Array :
[[ 0  1  2  3  4  5 100 101 102 103 104 105]
 [ 6  7  8  9 10 11 106 107 108 109 110 111]
 [12 13 14 15 16 17 112 113 114 115 116 117]
 [18 19 20 21 22 23 118 119 120 121 122 123]
 [24 25 26 27 28 29 124 125 126 127 128 129]]
```

Example of vertical stacking:

### Coding Sample:

```
# Vertical Stacking of Two Arrays
arr1 = np.arange(30).reshape(5,6)
print("Array 1 :\n", arr1)
arr2 = np.arange(100, 130).reshape(5,6)
print("Array 2 :\n", arr2)
print("Stacked Array :\n", np.vstack((arr1, arr2)))
```

## Output:

```

Array 1 :
[[ 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 2 :
[[100 101 102 103 104 105]
 [106 107 108 109 110 111]
 [112 113 114 115 116 117]
 [118 119 120 121 122 123]
 [124 125 126 127 128 129]]
Stacked 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]
 [100 101 102 103 104 105]
 [106 107 108 109 110 111]
 [112 113 114 115 116 117]
 [118 119 120 121 122 123]
 [124 125 126 127 128 129]]

```

The `vstack` or `hstack` functions are particularly useful when joining multiple array datasets together.

## Splitting arrays

Numpy also allows for arrays to be split vertically or horizontally using the `vsplit` or `hsplit` functions. `Vsplit` and `hsplit` functions take in arguments of the array name and the number of equal splits that are to be made. The second argument necessitates that the array has a dimension that can be equally split into the specified number of sections.

E.g. `np.vsplit(arrayname, 3)` cannot be applied on a array with dimension of (5, 5). A more appropriate dimension of 3 splits is (9, 9).

## Coding Sample:

```

# Vertical Splitting of Two Arrays
arr1 = np.arange(36).reshape(6,6)
print("Array 1 :\n", arr1)
print("Split Array :\n", np.vsplit(arr1, 2))

```

## Output:

```

Array 1 :
[[ 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]
[30 31 32 33 34 35]]
Split Array :
[array([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17]]), array([[18, 19, 20, 21, 22, 23],
        [24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35]])]
```

In the above example, the array is split to form a Tuple containing two arrays – the first one taken from the first 3 rows of the original (input) array and the second output array taken from the last 3 rows of the input array – hence representing two equal splits of the input array along the rows.

### Coding Sample:

```
# Horizontal Splitting of Two Arrays
arr1 = np.arange(36).reshape(6,6)
print("Array 1 :\n", arr1)
print("Split Array :\n", np.hsplit(arr1, 2))
```

### Output:

```
Array 1 :
[[ 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]
 [30 31 32 33 34 35]]
Split Array :
[array([[ 0,  1,  2],
        [ 6,  7,  8],
        [12, 13, 14],
        [18, 19, 20],
        [24, 25, 26],
        [30, 31, 32]]), array([[ 3,  4,  5],
        [ 9, 10, 11],
        [15, 16, 17],
        [21, 22, 23],
        [27, 28, 29],
        [33, 34, 35]])]
```

In the above example of horizontal splitting, the input array is split to form a Tuple containing two arrays – the first one taken from the first 3 columns of the original (input) array and the second output array taken from the last 3 columns of the input array – hence representing two equal splits of the input array along the columns.

## Array Assignment vs Copy

When an array is assigned into another using the "=" operator, it creates a reference to the original array object and does not create a new copy. Hence, any change in either of the original or copy arrays will reflect in the other. Developers need to be particularly mindful of this to avoid any inadvertent mistake while working with array copies.

### Coding Sample:

```
# Demonstrate Array assignment
arr1 = np.arange(36).reshape(6,6)
arr2 = arr1    # arr2 is a reference to arr1, and not a physically different copy
arr1[1,1] = 99 # Change reflected in both arr1 and arr2
print("Array 1: \n", arr1)
print("Array 2: \n", arr2)
```

### Output:

```
Array 1:
[[ 0  1  2  3  4  5]
 [ 6 99  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
Array 2:
[[ 0  1  2  3  4  5]
 [ 6 99  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
```

To get around this and be able to create physically different copies, the copy function is used.

### Coding Sample:

```
# Demonstrate Array Copy
arr1 = np.arange(36).reshape(6,6)
arr2 = arr1.copy() # A physically different copy of the input array is created
arr1[1,1] = 99    # Change reflected in both arr1 and arr2
print("Array 1: \n", arr1)
print("Array 2: \n", arr2)
```

### Output:

```
Array 1:
```

```
[[ 0  1  2  3  4  5]
 [ 6 99  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
Array 2:
[[ 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]
 [30 31 32 33 34 35]]
```

Note that the change in `arr1[1,1]` is reflected only in `arr1`, and not `arr2`. In the earlier case of assignment ("`=`"), the change was reflected in both arrays as `arr2` was only a memory reference to `arr1`.

## Working with the Pandas library

### Introduction to Pandas

Pandas is another library that has become almost all pervading in Data Analysis using Python. Pandas offer two powerful Data Structures named, Series and DataFrame. In this chapter we will look into the features and uses of these two data structures in detail. Before that, some of the key points about the Pandas library:

1. Pandas is open source, BSD-licensed.
2. Pandas is built on top of the Numpy Library.
3. Pandas offer Series, a one-dimensional array of labelled or indexed data. In some effect, that makes the Series object act like a Dictionary.
4. DataFrames are multidimensional arrays. DataFrames provide row and column labels or indexes that can be used for efficient access of rows, columns or elements.
5. DataFrames allow for working with missing data.
6. DataFrames act similarly to DataBase tables or spreadsheets and offer similar features to them such as data grouping, group operations, filtering, efficient condition-based data manipulations, creating pivot tables, etc.
7. Pandas offer features to manipulate data structures and interoperability between various data structures.
8. Pandas offer efficient date/time operations and handling of time-indexed data.
9. In both Series and DataFrame, indexes can be explicitly defined and manipulated.
10. Pandas offer integration with several data/file formats such as csv, excel, json. Import from or export to these data formats are efficient and easy.
11. Pandas also offer integration with various visualization tools such as matplotlib or seaborn libraries.

Pandas data structures and operations are particularly useful when working with tabular data similar to databases or spreadsheets and similar operations are needed.

## Installing Pandas Library

If you are using the Anaconda tool, Pandas comes pre-installed. However, just in case, you do not have the Pandas pre-installed in your system, here is what you can do:

```
$ pip install pandas  
or  
$ brew install pandas  
Or  
$ conda install pandas
```

(Assuming you have pip, homebrew or conda already)

## Importing Pandas

You need to import the Pandas library in your Python code before using it:

```
import pandas  
or  
import pandas as pd
```

The difference between the above two commands is that with the first command, you need to specify the full 'pandas' qualifier for all Pandas functions whereas in the second one, you can specify only 'pd' as the qualifier

## Working with Series Objects

Pandas Series is a one-dimensional indexed object that is mutable, iterable and can hold elements of various types. Each Series item is indexed – by default, elements assume a serial numeric index starting with 0. Indexes can be specified explicitly as well. A key difference between a Numpy array and Pandas Series is the use of index.

## Creating Series Objects

A Series object can be created using the `pandas.Series` method. A Series can be created by specifying explicit values or importing values from other collection or iterable objects. Alongside the values, we may also want to specify corresponding Index values unless we want Pandas to assign them automatically. Note that in the first of the following examples below, although we are specifying explicit values to assign to the new Series object elements, the values are enclosed within a List object. Series only accepts values input through another collection object. The left-hand column are the index values while the second column are the Series element values.

In the second example, we assign Index values explicitly.

## Coding Sample:

```
# Create my first Series Object  
ser1 = pd.Series([3, 4, 8, 9, 5])  
print(ser1)  
ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e']) # Specify the Index values
```

```
print(ser1)
```

#### Output:

```
4    3
4    4
2    8
3    9
4    5
dtype: int64
a    3
b    4
c    8
d    9
e    5
dtype: int64
```

In the following example, we create a Series object from a Numpy Array.

#### Coding Sample:

```
# Create a Series Object using a Numpy Array
arr = np.array([1,2,3,4,5])
ser1 = pd.Series(arr)
print(ser1)
```

#### Output:

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

In the next example, we create a Series object from a Dictionary. Note, how the keys of a dictionary become the indexes of a Series and the values will become the Series elements values.

In the second example below, we import only specified indexes from dictionary by using the Index parameter and mentioning the index values to import.

#### Coding Sample:

```
# Create a Series Object using a Dictionary
dict1 = {'a':1, 'b':2, 'c':3, 'd':4}
ser1 = pd.Series(dict1)
print("Import the whole Dictionary: \n", ser1)
ser1 = pd.Series(dict1, index=['c', 'b']) # Explicitly mention which keys to import
print("Import Only specified Indexes from teh dictionary: \n", ser1)
```

### Output:

```

Import the whole Dictionary:
a  1
b  2
c  3
d  4
dtype: int64
Import Only specified Indexes from teh dictionary:
c  3
b  2
dtype: int64

```

### Accessing Series Elements

Series objects can be accessed by specifying index values of the position of the element to be accessed. Slice of the Series object can also be accessed by specifying the range.

### Coding Sample:

```

# Access elements of a Series object
ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e'])
print("Print the whole Series:\n", ser1)
print("Print the 3rd Element:\n", ser1[2]) # Access a single element
print("Print the elements between the 2nd to 4th positions:\n", ser1[1:4]) # Access a slice

```

### Output:

```

Print the whole Series:
a  3
b  4
c  8
d  9
e  5
dtype: int64
Print the 3rd Element:
8
Print the elements between the 2nd to 4th positions:
b  4
c  8
d  9
dtype: int64

```

In the following example, we take a slice from the Series using explicite indexes similar to the way we would have used implicit number indexes.

### Coding Sample:

```

# Access elements of a Series object by specifying Index values
ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e'])

```



```
print("Print a slice using explicit indexes: \n", ser1['b':'d'])
```

#### Output:

```
Print a slice using explicit indexes:
b  4
c  8
d  9
dtype: int64
```

### Accessing Series objects using 'loc' and 'iloc' Indexers

Python provides a host of Indexers for selection and accessing of elements from Series and DataFrame objects. In this section, we discuss how we can extract single or range of elements from a Series using the 'loc' and 'iloc' indexer methods.

Python provides the 'loc' method which takes **explicit** or label-based index values or range of index values to access corresponding elements. The explicit Index labels are the ones that are mentioned explicitly for the Series object.

#### Label / Index Based Indexing using 'loc'

In the following example, we create a Series object with explicit labels and access them with 'loc'. In the first of the two examples in the following code block, we access a single element whereas in the second example, we access a range of values from the Series by specifying the starting label and ending label. These mechanisms are much the same as without using the indexer.

#### Coding Sample:

```
# Access elements of a Series object by using the 'loc'
ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e'])
print("Print a single element using explicit indexes with 'loc': \n", ser1.loc['b'])
print("Print a slice of elements using explicit indexes with 'loc': \n", ser1.loc['b':'d'])
```

#### Output:

```
Print a single element using explicit indexes with 'loc':
4
Print a slice of elements using explicit indexes with 'loc':
b  4
c  8
d  9
dtype: int64
```

## Implicit or Integer based Indexing using 'iloc'

The 'iloc' indexer takes *implicit* index or range of indexes to access corresponding elements.

### Coding Sample:

```
# Access elements of a Series object by using the 'iloc'
ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e'])
print("Print a single element using implicit indexes with 'iloc': \n", ser1.iloc[2])
print("Print a slice of elements using implicit indexes with 'iloc': \n", ser1.iloc[1:4])
```

### Output:

```
Print a single element using implicit indexes with 'iloc':
8
Print a slice of elements using implicit indexes with 'iloc':
b  4
c  8
d  9
dtype: int64
```

The loc/iloc indexers will raise IndexError in case an invalid out of range index value is used. A suggested way in case of a possibility is to trap this error using a "try-except" block.

The reason Pandas offer loc / iloc method is to bring in clarity when developers tend to use implicit / explicit indexes interchangeably and the indexes are numeric. E.g.

Implicit – 0 onwards

Explicit – 1 onwards

Using loc/iloc clarify whether an implicit or an explicit index is being used.

## Implicit or Integer based Indexing using 'iat'

Another indexer provided by Pandas is the 'iat' indexer, that is similar to 'iloc', but can only extract single elements.

### Coding Sample:

```
# Access single element of a Series object by using the 'iat'
ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e'])
print(ser1.iat[2])
```

### Output:

```
8
```

## Series Object Attributes

The following table details a list of key Series object attributes.

Attributes	Description
T	Returns the transpose of the Series. However, Series being one dimensional, the transpose is the same as the original Series.
Array	The 'array' attribute returns the array equivalent of the Series object. # Return the Python Array equivalent of the Series arr1 = ser1.array print(arr1) Outcome: <PandasArray> [3, 4, 8, 9, 5] Length: 5, dtype: int64
Axes	Returns the Row Axes Label, i.e. for a Series object, returns the index labels. # Return the Row Axes labels axes1 = ser1.axes print(axes1) Outcome: [Index(['a', 'b', 'c', 'd', 'e'], dtype='object')]
Dtypes	The dtypes attributes return the datatype of the underlying values in the Series. In case of heterogeneity of data types in the Series, dtypes return an Object type. # Return the datatype of the underlying data with dtypes attribute ser1 = pd.Series([3, 4, 8, 9, 5]) # All elements are of type int ser2 = pd.Series(['a', 'b', 'c', 'd', 'e']) # All elements are of type char ser3 = pd.Series([12.3, 34.56, 45.999]) # All elements are of type float ser4 = pd.Series([1, 's', 12.3, True]) print("Dtype of ser1 - ", ser1.dtypes) print("Dtype of ser2 - ", ser2.dtypes) print("Dtype of ser3 - ", ser3.dtypes) print("Dtype of ser4 - ", ser4.dtypes) Outcome: Dtype of ser1 - int64 Dtype of ser2 - object Dtype of ser3 - float64 Dtype of ser4 - object
Hasnans	The hasnans attribute return whether or not there is any NaN value in the Series. NaN in Python data structures denote a missing value. # Return True if the Series has any NaN values, otherwise False ser1 = pd.Series([3, 4, 8, 9, 5]) print("Check whether ser1 has NaN values: ", ser1.hasnans) ser1[2] = np.nan # Assign a NaN value to an element in ser1 print("Print ser1 \n", ser1) print("Check whether ser1 has NaN values: ", ser1.hasnans) Outcome: Check whether ser1 has NaN values: False Print ser1 0 3.0

	<pre> 1  4.0 2  NaN 3  9.0 4  5.0 dtype: float64 Check whether ser1 has NaN values: True </pre>
Index	<p>Returns the index values of the Series object.</p> <pre> # Print the Index labels of the Series object ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e']) print(ser1.index) </pre> <p>Outcome:</p> <pre> Index(['a', 'b', 'c', 'd', 'e'], dtype='object') </pre>
Is_monotonic	<p>Returns Boolean result of if the values of the elements in the Series are increasing monotonic.</p> <pre> # Print if the Series values are monotonic increasing ser1 = pd.Series([1, 2, 3, 4, 5, 6]) print("Check if ser1 is monotonic Increasing ", ser1.is_monotonic) ser1 = pd.Series([1, 2, 3, 4, 7, 5, 6]) print("Check if ser1 is monotonic Increasing after the change ", ser1.is_monotonic) </pre> <p>Outcome:</p> <pre> Check if ser1 is monotonic Increasing True Check if ser1 is monotonic Increasing after the change False </pre>
Is_monotonic_increasing	<p>Returns Boolean True of if the values of the elements in the Series are increasing monotonic.</p>
Is_monotonic_decreasing	<p>Returns Boolean True of if the values of the elements in the Series are decreasing monotonic.</p>
Is_unique	<p>Returns True if the values of the Series elements are unique, otherwise False.</p> <pre> # Print if the values of the Series are unique ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser1.is_unique </pre> <p>Outcome:</p> <pre> False </pre>
Nbytes	<p>Returns the total number of memory bytes occupied by the Series.</p> <pre> # Print the total storage bytes taken by ser1 ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser1.nbytes </pre> <p>Outcome:</p> <pre> 48 </pre>
size	<p>Returns the size of the Object in terms of total number of elements.</p> <pre> # Print the size of the Series object ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser1.size </pre> <p>Outcome:</p> <pre> 6 </pre>

Values	Returns the values of the elements. # Print the values of the Series object ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser1.values Outcome: array([1, 2, 2, 4, 4, 6])
--------	--

## Series Object Methods

The following are the main methods supported by Pandas for the Series object. Note that only a limited list of commonly used methods are listed here. Pandas support a huge list of methods, covering all of which are outside the scope of this lesson.

1. Methods	Description
2. Add	The 'add' method adds corresponding elements from two Series objects.  # Check out the 'add' method to add corresponding elements from two Series objects ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser2 = pd.Series([10, 20, 20, 40, 40, 60]) print(ser1.add(ser2))  Outcome: 0 11 1 22 2 22 3 44 4 44 5 66 dtype: int64
3. Append	The 'append' method concatenates one Series object at the end of the other.  # Use of the 'append' method to concatenate (add one after the other) ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser2 = pd.Series([10, 20, 20, 40, 40, 60]) print(ser1.append(ser2))  Outcome: 0 1 1 2 2 2 3 4 4 4 5 6 0 10 1 20

	<pre> 2  20 3  40 4  40 5  60 dtype: int64 </pre>
4. Astype	<p>The 'astype' method is used to cast the elements of a Pandas Series to another Data Type. In the following example, we are converting int elements to a string type.</p> <pre> # Use of the 'astype' method to cast the elements to another data type ser1 = pd.Series([1, 2, 2, 4, 4, 6]) print(list(ser1.astype(str))) </pre> <p>Outcome: ['1', '2', '2', '4', '4', '6']</p>
5. At_time	<p>Extracts values of an element at a specified date/time in a Date/Time Index Series.</p> <pre> # Filter/Extract specific elements with date/time index Series dt_tm = pd.date_range('2020-04-01', periods=5, freq='6H') ser1 = pd.DataFrame({'A': ['Sleep', 'Breakfast', 'Lunch', 'Snacks', 'Dinner']}, index=dt_tm) print("Series: \n", ser1) print("\n", ser1.at_time('18:00')) </pre> <p>Note the use the of the date_range method above to create a Series with Date/Time values passing a Date, number of periods and period increment durations.</p> <p>Outcome: Series:</p> <pre>           A 2020-04-01 00:00:00    Sleep 2020-04-01 06:00:00  Breakfast 2020-04-01 12:00:00    Lunch 2020-04-01 18:00:00    Snacks 2020-04-02 00:00:00    Dinner </pre> <p>A</p> <pre> 2020-04-01 18:00:00  Snacks </pre>
6. Between	<p>The 'between' method returns a Boolean vector corresponding to all the elements in the passed Series as to whether or not each of the elements are between two other passed values.</p> <pre> # Use the 'between' method to create a Boolean vector corresponding to all the elements in the passed Series # as to whether or not each of the elements are between two other passed values. ser1 = pd.Series([1, 2, 3, 4, 5, 6]) </pre>

	<pre>print(ser1.between(2,4))</pre> <p>Outcome:</p> <pre>0  False 1   True 2   True 3   True 4  False 5  False dtype: bool</pre>
7. Copy	<p>Copies one Series to a new one.</p> <p>Deep copy: Creates a new object with values and indexes of the original Series. Change in any element in the original Series will not reflect in the Deep copy.</p> <p>Shallow Copy: Creates a new object with references of values and indexes of the original Series. Change in any element in the original Series will reflect in the Shallow copy.</p> <pre># Series Deep Copy &amp; Shallow Copy ser1 = pd.Series([3, 4, 5]) ser2 = ser1.copy(deep=True) ser1[2] = 8 print("Ser1: \n", ser1) print("Ser2 - Deep Copy: \n", ser2) ser3 = ser1.copy(deep=False) ser1[2] = 8 print("Ser3 - Shallow Copy: \n", ser3)</pre> <p>Outcome:</p> <pre>Ser1: 0  3 1  4 2  8 dtype: int64 Ser2 - Deep Copy: 0  3 1  4 2  5 dtype: int64 Ser3 - Shallow Copy: 0  3 1  4 2  8 dtype: int64</pre>
8. count	<p>Returns the number of elements in the Series.</p> <pre># Count of Elements</pre>

	<pre>ser1 = pd.Series([3, 4, 5]) print(ser1.count())</pre> <p>Outcome: 3</p>
9. describe	<p>Prints the Descriptive statistics of the Series elements.</p> <pre># Print Discriptive Statistics of the Series elements ser1 = pd.Series([12, 21, 14, 10, 15, 19]) ser1.describe()</pre> <p>Outcome: count    6.000000 mean     15.166667 std       4.167333 min       10.000000 25%      12.500000 50%      14.500000 75%      18.000000 max       21.000000 dtype: float64</p>
10. Div	<p>Divides corresponding elements from two Series Objects:</p> <pre># 'div' method divides corresponding elements from two Series objects ser1 = pd.Series([5, 4, 52, 64, 47, 45]) ser2 = pd.Series([10, 20, 20, 40, 40, 60]) print(round(ser1.div(ser2),4))</pre> <p>Outcome: 0    0.500 1    0.200 2    2.600 3    1.600 4    1.175 5    0.750 dtype: float64</p>
11. Drop	<p>The 'drop' method removes the specified elements from a Series object. You can mention multiple labels/indexes as a List object in the parameter. It returns the updated Series after the drop operation.</p> <pre># Drop elements with specified label(s) ser1 = pd.Series([3, 4, 8, 9, 5], index=['a', 'b', 'c', 'd', 'e']) ser2 = ser1.drop(labels=['c', 'e']) print(ser2)</pre> <p>Outcome: a    3 b    4 d    9</p>



	dtype: int64
12. Dropna	<p>The 'dropna' method drops the 'NaN' (Null or missing) values from the Series.</p> <pre># Drops the NaN elements in a Series ser1 = pd.Series([3, 4, 8, 9, 5]) ser1[2] = np.nan # Assign a NaN value to an element in ser1 print("Print ser1 \n", ser1) print("Check the Series after dropping NaNs: \n", ser1.dropna())</pre> <p>Outcome:</p> <p>Print ser1</p> <pre>0  3.0 1  4.0 2  NaN 3  9.0 4  5.0 dtype: float64</pre> <p>Check the Series after dropping NaNs:</p> <pre>0  3.0 1  4.0 3  9.0 4  5.0 dtype: float64</pre>
13. Equals	<p>Checks if two Series objects passed to it are equal.</p> <pre># Validates if two Series objects are equal ser1 = pd.Series([3, 4, 8, 9, 5]) ser2 = pd.Series([3, 4, 8, 9, 5]) print(ser1.equals(ser2))</pre> <p>Outcome:</p> <p>True</p>
14. Fillna	<p>Fills the NaN values in the Series object with a specified value.</p> <pre># Fills the NaN elements in a Series with a Specified value ser1 = pd.Series([3, 4, 8, 9, 5]) ser1[2] = np.nan # Assign a NaN value to an element in ser1 print("Print ser1 \n", ser1) print("Check the Series after filling NaNs: \n", ser1.fillna(12))</pre> <p>Outcome:</p> <p>Print ser1</p> <pre>0  3.0 1  4.0 2  NaN 3  9.0</pre>

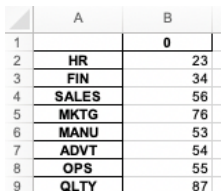
	<pre> 4    5.0 dtype: float64 Check the Series after filling NaNs: 0    3.0 1    4.0 2   12.0 3    9.0 4    5.0 dtype: float64 </pre>
15. floordiv	<p>The 'floordiv' method divides corresponding elements from two Series objects and returns the interger of the quotient.</p> <p># 'floordiv' method divides corresponding elements from two Series objects and returns the interger of the quotient</p> <pre> ser1 = pd.Series([5, 4, 52, 64, 47, 45]) ser2 = pd.Series([10, 20, 20, 40, 40, 60]) print(ser1.floordiv(ser2)) </pre> <p>Outcome:</p> <pre> 0    0 1    0 2    2 3    1 4    1 5    0 dtype: int64 </pre>
16. Groupby	<p>The 'groupby' method groups elements of the same label together to enable performing a group operation such as mean, sum, etc. In the following example, we find the 'mean' of all the elements with HR and SALES labels respectively. Note, in this example, we also establish that Series elements can have same labels.</p> <p># Grouping of series elements</p> <pre> ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'HR', 'SALES', 'HR', 'SALES', 'HR', 'HR', 'SALES']) ser1.groupby(level=0).mean() </pre> <p>Outcome:</p> <pre> HR      48.400000 SALES   65.333333 dtype: float64 </pre>
17. Head	<p>Print the first 5 elements from a Series. Usually used on Series with large numbers for ease of visual inspection.</p> <p># Print the first 5 elements</p> <pre> ser1 = pd.Series([5, 4, 52, 64, 47, 45, 20, 55, 54, 42, 88]) ser1.head() </pre>

	<p>Outcome:</p> <pre>0    5 1    4 2   52 3   64 4   47 dtype: int64</pre>
18. Idxmax	<p>The 'idxmax' method returns the label or index of the maximum value in a Series.</p> <p># Return the label of the maximum value</p> <pre>ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1.idxmax()</pre> <p>Outcome:</p> <p>'QLTY'</p>
19. Idxmin	<p>The 'idxmin' method returns the label or index of the minimum value in a Series.</p> <p># Return the label of the minimum value</p> <pre>ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1.idxmin()</pre> <p>Outcome:</p> <p>'HR'</p>
20. Isin	<p>Returns a Series/vector of whether a specified value equals to each of the corresponding Series object elements.</p> <p>In the next example, we check if 87 is part of ser1 and it checks against each element and returns a Boolean vector.</p> <p># Check for membership (if it is part of) of a specified value in a Series object</p> <pre>ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87]) ser1.isin([87])</pre> <p>Outcome:</p> <pre>0    False 1    False 2    False 3    False 4    False 5    False 6    False 7     True dtype: bool</pre>
21. Isna	<p>Checks each element in a Series on whether it is a NaN value and returns a Series/vector of Boolean results.</p>

	<pre># Check for existence of NaN value against each of the elements ser1 = pd.Series([23, 34, 56, None, 53, 54, 55, 87]) ser1.isna()</pre> <p>Outcome:</p> <pre>0  False 1  False 2  False 3   True 4  False 5  False 6  False 7  False dtype: bool</pre>
22. Max	Return the 'max', 'mean', 'min' and 'median' of the values from the Series object.
23. Mean	
24. Min	
25. Median	<pre># Return 'max', 'min', 'mean', 'median' values of a Series ser1 = pd.Series([23, 34, 56, 43, 53, 54, 55, 87]) print("The Maximum Value element: ", ser1.max()) print("The Minimum Value element: ", ser1.min()) print("The Mean of all elements: ", ser1.mean()) print("The Median value among all elements: ", ser1.median())</pre> <p>Outcome:</p> <pre>The Maximum Value element: 87 The Minimum Value element: 23 The Mean of all elements: 50.625 The Median value among all elements: 53.5</pre>
26. Multiply	<p>The 'multiply method multiplies corresponding elements from two Series objects.</p> <pre># Use of the 'multiply' method to multiply corresponding elements from two Series objects ser1 = pd.Series([1, 2, 2, 4, 4, 6]) ser2 = pd.Series([10, 20, 20, 40, 40, 60]) print(ser1.multiply(ser2))</pre> <p>Outcome:</p> <pre>0   10 1   40 2   40 3  160 4  160 5  360 dtype: int64</pre>

27. Replace	<p>The 'replace' method replaces a specified value in the Series with another. The 'replace' method returns a modified Series and the original series remains unchanged. The 'inplace' parameter has to be set to True if the original Series object is to be updated.</p> <pre># Replace a specified value in the Series with another ser1 = pd.Series([23, 34, 56, 66, 53, 54, 55, 87]) ser2 = ser1.replace({34:55}) print("Printing ser2: \n", ser2) ser1.replace({34:55}, inplace=True) print("Printing ser1 after replacing inplace:\n", ser1)</pre> <p>Outcome:</p> <p>Printing ser2:</p> <pre>0  23 1  55 2  56 3  66 4  53 5  54 6  55 7  87 dtype: int64</pre> <p>Printing ser1 after replacing inplace:</p> <pre>0  23 1  55 2  56 3  66 4  53 5  54 6  55 7  87 dtype: int64</pre>
28. Round	<p>The 'round' method rounds off the series elements to a specified position. In the following example, it rounds off to 2nd decimal place.</p> <pre># Rounding off Series elements ser1 = pd.Series([23.8567876, 34.76465354, 56.53439, 43.09775454, 53.63654, 5.8755424, 5.858765, 8.656547]) print(ser1.round(2))</pre> <p>Outcome:</p> <pre>0  23.86 1  34.76 2  56.53 3  43.10 4  53.64 5  5.88</pre>

	<pre>6  5.86 7  8.66 dtype: float64</pre>
29. Std	<p>The 'std' method returns the Standard Deviation of all the elements of a Series Object.</p> <pre># Find the standard deviation using the std method ser1 = pd.Series([23, 34, 56, 66, 53, 54, 55, 87]) stdv = ser1.std() print(round(stdv, 2))</pre> <p>Outcome: 19.27</p>
30. str	<p>'str' enables String functions to be applied on Series objects.</p> <pre># Enables operation of string functions on string type elements ser1 = pd.Series(['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1_str = ser1.str.replace('M', 'Y') print(ser1_str)</pre> <p>Outcome:</p> <pre>0    HR 1    FIN 2  SALES 3  YKGT 4  YANU 5  ADVT 6    OPS 7  QLTY dtype: object</pre> <p>In the above example, the String function 'replace' cannot be directly applied on the Series object and hence enabled by 'str'.</p>
31. Sub	<p>The 'sub' function subtracts numeric elements of one Series element from corresponding elements of the original one.</p> <pre># Subtraction of one Series elements from another ser1 = pd.Series([5, 4, 52, 64, 47, 45, 20, 55, 54, 42, 88]) ser2 = pd.Series([1, 3, 35, 33, 89, 57, 46, 22, 34, 32, 44]) print(ser1.sub(ser2))</pre> <p>Outcome:</p> <pre>0    4 1    1 2   17 3   31 4  -42 5  -12</pre>

	<pre> 6 -26 7 33 8 20 9 10 10 44 dtype: int64 </pre>
32. To_dict	<p>The 'to_dict' method converts a Series object to a Python Dictionary. The Indexes get converted into 'key's and the Series element values are copied as 'values' into the output dictionary.</p> <pre> # Convert a Series object to a Dictionary ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1dict = ser1.to_dict() print(ser1dict) </pre> <p>Outcome:</p> <pre> {'HR': 23, 'FIN': 34, 'SALES': 56, 'MKTG': 76, 'MANU': 53, 'ADVT': 54, 'OPS': 55, 'QLTY': 87} </pre>
33. To_excel	<p>The 'to_excel' method converts a Series object to an Excel sheet – external file.</p> <pre> # Convert a Series object to an Excel sheet (External file) ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1dict = ser1.to_excel('ser1.xls') </pre> <p>Outcome:</p>  <p>The above screenshot is of the resultant excel file.</p>
34. To_list	<p>The 'to_list' method converts a Series object to a List object. Note that in the conversion, we lose the Series 'indexes'. Only the values get converted as the List elements.</p> <pre> # Convert a Series object to a List object ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1lst = ser1.to_list() print(ser1lst) </pre> <p>Outcome:</p> <pre> [23, 34, 56, 76, 53, 54, 55, 87] </pre>
35. To_string	<p>The 'to_string' method converts a Series object to it's string representation.</p>

	<pre># Convert a Series object to String Representation ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87], index=['HR', 'FIN', 'SALES', 'MKTG', 'MANU', 'ADVT', 'OPS', 'QLTY']) ser1.to_string() print(ser1)</pre> <p>Outcome:</p> <pre>HR    23 FIN    34 SALES  56 MKTG   76 MANU   53 ADVT   54 OPS    55 QLTY   87</pre> <p>Note that the 'print' function is rendering the string well into a table like format due to the presence of '\n' – newline characters after every line/row from the original Series.</p>
36. truncate	<p>The 'truncate' method truncates a Series object based on a 'before' or 'after' index values. 'before=3' parameter indicates truncation of all elements before index number 3. 'after=3' indicates truncation of all elements after the index value 3.</p> <pre># Truncate a Series Object ser1 = pd.Series([23, 34, 56, 76, 53, 54, 55, 87]) sert1 = ser1.truncate(before=3) print(sert1) sert2 = ser1.truncate(after=3) print(sert2)</pre> <p>Outcome:</p> <pre>3    76 4    53 5    54 6    55 7    87 dtype: int64 0    23 1    34 2    56 3    76 dtype: int64</pre>
37. Unique	<p>The 'unique' method returns unique elements from the Series object after removing all the duplicates.</p> <pre># Find Unique elements in a Series ser1 = pd.Series([23, 34, 23, 76, 53, 53, 55, 87]) ser1unq = ser1.unique()</pre>



	<pre>print(ser1unq)</pre> <p>Outcome: [23 34 76 53 55 87]</p>
38. Var	<p>The 'var' method finds the Variance of all numeric elements in a Series.</p> <pre># Find Variance of all elements in a Series ser1 = pd.Series([23, 34, 23, 76, 53, 53, 55, 87]) variance = ser1.var() print(round(variance, 2))</pre> <p>Outcome: 542.86</p>

### Multi-Indexed Series Objects

In scenarios when we are required to have multi-level indexes for values/elements, a smart way provided by Pandas is the Multi-indexed Series objects.

E.g. We would like to have runs scored by cricketers in Test matches as well as One Day matches. In this case, we have two levels of indexes or data labels – Cricketer Name and the inner level is Type of Matches (i.e. ODI or Tests).

In the following example, we are creating such a Series object that has these two-level indexes.

#### Coding Sample:

```
# Create a Multi-level Index Series Object
index = [('Tendulkar', 'Tests'), ('Tendulkar', 'ODIs'), ('Sangakkara', 'Tests'), ('Sangakkara', 'ODIs'),
('Ponting', 'Tests'), ('Ponting', 'ODIs')]
runs = [16000, 18000, 12000, 14000, 13000, 13000]
index = pd.MultiIndex.from_tuples(index)
records = pd.Series(runs, index=index)
print(records)
```

#### Output:

```
Tendulkar Tests 16000
          ODIs 18000
Sangakkara Tests 12000
          ODIs 14000
Ponting Tests 13000
          ODIs 13000
dtype: int64
```

The output display is quite obvious – the Outer levels are not duplicated through all the inner levels, making the display visually apparent of the multi-level.

## Accessing Multi-Level Series Elements

The usual methods of accessing Series elements apply. Here, we are showing examples of 'loc', 'iloc' and direct index methods for accessing Multi-Indexed Series elements.

### Coding Sample:

```
print("Printing all of Sangakkara's Records: \n", records.loc['Sangakkara'])
print("Printing Tendulkar's ODI record: \n", records.loc['Tendulkar', 'ODIs'])
print("Printing Ponting's Test record: \n", records.iloc[4])
print("Printing Ponting's Test record - direct Index Access: \n", records[4])
```

### Output:

```
Printing all of Sangakkara's Records:
Tests   12000
ODIs    14000
dtype: int64
Printing Tendulkar's ODI record:
18000
Printing Ponting's Test record:
13000
Printing Ponting's Test record - direct Index Access:
13000
```

**Send us your feedback on [info@teksands.ai](mailto:info@teksands.ai)**

# Explore Teksands.ai

## Learn from Researchers from IITs/NITs



<https://www.facebook.com/Teksands>



<https://www.linkedin.com/company/teksands/>



<https://twitter.com/teksands>



<https://www.instagram.com/teksands.ai/>

**Call or WhatsApp us to Enquire about our Machine Learning/AI and Data Science Courses**



<https://api.whatsapp.com/send?phone=916362732428>



**EXPLORE OUR MACHINE  
LEARNING AND DATA  
SCIENCE COURSES**