Semester-wise Revised Syllabus under CBCS, 2020-21
Four -year B.Sc.(Hons)
Domain Subject: **Computer Science**
IV Year B. Sc.(Hons) – Semester – V

Course Code: Max. Marks : 100 + 50

Course 7C: Python for Data Science

Unit - I (10 hours)

Introduction to Data Science - Why Python? - Essential Python libraries - Python Introduction- Features, Identifiers, Reserved words, Indentation, Comments, Built-in Data types and their Methods: Strings, List, Tuples, Dictionary, Set - Type Conversion- Operators. Decision Making- Looping- Loop Control statement- Math and Random number functions. User defined functions - function arguments & its types.

UNIT –II (10 hours)

User defined Modules and Packages in Python- Files: File manipulations, File and Directory related methods - Python Exception Handling.
OOPs Concepts -Class and Objects, Constructors – Data hiding- Data Abstraction- Inheritance.

UNIT –III (10 hours)

NumPy Basics: Arrays and Vectorized Computation- The NumPy ndarray- Creating ndarrays- Data Types for ndarrays- Arithmetic with NumPy Arrays- Basic Indexing and Slicing - Boolean Indexing-Transposing Arrays and Swapping Axes.

UNIT –IV (10 hours)

Introduction to pandas Data Structures: Series, Data Frame and Essential Functionality: Dropping Entries- Indexing, Selection, and Filtering- Function Application and Mapping- Sorting and Ranking.

Summarizing and Computing Descriptive Statistics- Unique Values, Value Counts, and Membership. Reading and Writing Data in Text Format

UNIT –V (10 hours)

Data Cleaning and Preparation: Handling Missing Data - Data Transformation: Removing Duplicates, Transforming Data Using a Function or Mapping, Replacing Values, Detecting and Filtering Outliers- String Manipulation: Vectorized String Functions in pandas.

References

1. Y. Daniel Liang, "Introduction to Programming using Python", Pearson, 2012.
2. Wes McKinney, "Python for Data Analysis: Data Wrangling with Pandas,NumPy, and IPython", O'Reilly, 2nd Edition, 2018.
3. Jake VanderPlas, "Python Data Science Handbook: Essential

Tools for Working with Data", O'Reilly, 2017.
4. Wesley J. Chun, "Core Python Programming", Prentice Hall, 2006.
5. Mark Lutz, "Learning Python", O'Reilly, 4th Edition, 2009.

UNIT – I

1Q) Introduction to Data Science

Ans

- Data science is the process of deriving knowledge and insights from a huge and diverse set of data through organizing, processing and analysing the data.
- It involves many different disciplines like mathematical and statistical modelling, extracting data from it source and applying data visualization techniques.
- Often it also involves handling big data technologies to gather both structured and unstructured data. Below we will see some example scenarios where Data science is used.

1. **Recommendation systems**
   As online shopping becomes more prevalent, the e-commerce platforms are able to capture users shopping preferences as well as the performance of various products in the market. This leads to creation of recommendation systems which create models predicting the shoppers needs and show the products the shopper is most likely to buy.
2. **Financial Risk management**
   The financial risk involving loans and credits are better analysed by using the customers past spend habits, past defaults, other financial commitments and many socio-economic indicators. These data is gathered from various sources in different formats. Organising them together and getting insight into customers profile needs the help of Data science.
3. **Improvement in Health Care services**
   The health care industry deals with a variety of data which can be classified into technical data, financial data, patient information, drug information and legal rules. All this data need to be analysed in a coordinated manner to produce insights that will save cost both for the health care provider and care receiver while remaining legally compliant.
4. **Computer Vision**
   The advancement in recognizing an image by a computer involves processing large sets of image data from multiple objects of same category. For example, Face recognition. These data sets are modelled, and algorithms are created to apply the model to newer images to get a satisfactory result. Processing of these huge data sets and creation of models need various tools used in Data science.
5. **Efficient Management of Energy**
   As the demand for energy consumption soars, the energy producing

companies need to manage the various phases of the energy production and distribution more efficiently. This involves optimizing the production methods, the storage and distribution mechanisms as well as studying the customers consumption patterns. Linking the data from all these sources and deriving insight seems a daunting task. This is made easier by using the tools of data science.

2Q) Write about PYTHON?

Ans:

- ⇨ PYTHON is a **general purpose**, **dynamic**, **high-level** and **interpreted programming language**.
- ⇨ It supports OOP approach to develop applications.
- ⇨ It is simple and easy to learn and provides lots of high-level data structures.
- ⇨ It is easy to learn yet powerful and versatile scripting language, which makes it attractive for application development.
- ⇨ Python's syntax and dynamic typing with its interpreted nature make it an ideal language for scripting and rapid application development.
- ⇨ It supports **multiple programming pattern, including Object Oriented, imperative** and **function or procedural programming styles**.
- ⇨ It is not intended to work in a particular area, such as web programming. That is why it is known as multipurpose programming language because it can be used with web, enterprise, 3D CAD/CAM, etc.
- ⇨ We don't need to use data type to declare variables because it is dynamically typed so we can write **a=10** to assign an integer value in an **integer variable**
- ⇨ **It makes the development and debugging fast because there is no compilation step included in python development and edit-test-debug cycle is very fast.**

History:

- ⇨ **Python was invented by GUIDO VAN ROSSUM in 1991 at CWI in Netherland.**
- ⇨ The idea of python programming language has taken from the **ABC programming language** or we can say that ABC is predecessor of python language.
- ⇨ **There is also a fact behind the choosing name Python. GUIDO VAN ROSSUM was a fan of the popular BBC comedy show of that time, "Monty Python's Flying circus". So he decided to pick the name Python for his newly created programming language.**

3Q) Write the Features of Python?

Ans:

1. **Easy to Learn and Use**
   - It is easy to learn as compared to other programming languages.
   - Its syntax is straight forward and must the same as the English language.
   - There is no use of the semi-colon or curly bracket, the indentation defines the code block.
   - It is recommended programming language for beginners
2. **Expressive language**
   - Python can perform complex tasks using a few lines of code.

Eg: Print("Hari")

3. **Interpreted Language**
   - It is an interpreted language. It means the python program is executed one line at a time.
   - The advantage of being interpreted language, it makes debugging easy and portable.
4. **Cross-Platform Language**
   - Python can run equally on different platforms such as windows, Linux, Unix and Macintosh, etc. So we can say that python is a portable language.
   - It enables programs to develop the software for several competing platforms by writing a program once.
5. **Free and Open source**
   - Python is freely available for everyone.
   - It is freely available on its official website www.python.org.
   - It has a large community across the world that is dedicatedly working towards make new python modules and functions.
   - The open source means, "Anyone can download its source code without paying any amount".
6. **Object Oriented language**
   - Python supports Object Oriented language and concepts of classes and objects come into existence.
   - It supports Polymorphism, inheritance and encapsulation
   - The object oriented procedure helps to programmer to write reusable code and develop applications in less code.
7. **Extensible**
   - It implies that other language such as C/C++ can be used to compile the code and thus it can be used further in our python code.
   - It converts the program into byte code, and any platform can use that byte code.
8. **Large standard Library**
   - It provides a vast range of libraries for the various fields such as machine learning, web developer and also scripting.
   - There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras and Pytorch, etc. Django, flask, pyramids and other popular framework for python web development.
9. **GUI Programming support**
   - GUI is used for the development of desktop applications and web applications.
10. **Integrated programming language**
    - It can be easily integrated with languages like C, C++, Java,etc.
    - Python runs code line by line like C/C++/Java. It makes easy to debug the code.
11. **Embeddable**
    - The code of the other programming language can use in the python source code.

- We can use python source code in other programming language as well. It can embed other language into our code.

12. Dynamic memory allocation

- In python, we don't need to specify the data type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time.

4Q) Python Popular Frameworks and Libraries

Ans:

Python has wide range of libraries and frameworks widely used in various fields such as machine learning, artificial intelligence, web applications, etc. We define some popular frameworks and libraries of Python as follows.

- **Web development (Server-side) -** Django Flask, Pyramid, CherryPy
- **GUIs based applications -** Tk, PyGTK, PyQt, PyJs, etc.
- **Machine Learning -** TensorFlow, PyTorch, **Scikit-learn**, Matplotlib, Scipy, etc.
- **Mathematics -** Numpy, Pandas, etc.

5Q) PYTHON TOKENS

Ans:

- The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
- The token is the smallest unit inside the given program.
- There are following tokens in Python:
  1. Identifiers
  2. Keywords.
  3. Operators.
  4. Data types
  5. Literals
  6. Comments

1) Identifiers

- It is a variable name, function name or constant name.
- An Identifier is used to identify the literals used in the program. To define identifiers the following rules to be followed. They are
  1. The first character of the identifier name must be an alphabet or underscore.
  2. All the characters except the first character may be an alphabet of lower-case (a-z), upper-case (A-Z), underscore, or digit (0-9).
  3. Identifier name must not contain any white-space, or special character
  4. Identifier name must not be similar to any keyword defined in the language.
  5. Identifier names are case sensitive

2) Keywords

- Python Keywords are special reserved words that convey a special meaning to the compiler/interpreter.
- Each keyword has a special meaning and a specific operation.
- These keywords can't be used as a variable.
- There are **35 Keywords** in Python 3.9 version

True	False	None	And	As
Assert	Def	class	continue	Break
Else	finally	elif	Del	Except
Global	for	if	From	Import
Raise	try	or	return	Pass
nonlocal	in	not	is	Lambda
Await	Async	While	With	yield

3) Operators

- The operator can be defined as a symbol which is responsible for a particular operation between two operands.
- Operators are the pillars of a program on which the logic is built in a specific programming language.
- Python provides a variety of operators, such as
  1. Arithmetic operators
  2. Comparison operators
  3. Assignment Operators
  4. Logical Operators
  5. Bitwise Operators
  6. Membership Operators
  7. Identity Operators

Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations between two operands.

Operator	Meaning	Examples
+	Addition	A=10,B=20,C=A+B=>30
-	Subtraction	A=10,B=20,C=A-B=>-10
*	Multiplication	A=10,B=20,C=A*B=>200
/	Division	A=10,B=2,C=A/B=>5
%	Modulus	A=10,B=3,C=A%B=>1
**	**Exponent**	A=5,b=3, a**b =>125
//	**Floor division**	

Comparison operator

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly.

Operator	Meaning	Examples
>	Greater than	A=10,b=5,a>b =>true
<	Less than	A=10,b=5,A<b =>false
>=	Greater than equals to	A=10,b=5,a>=b =>true
<=	Less than equals to	A=10,b=5,A<=b =>false
==	equals to	A=10,b=5,A= =b =>false
!=	not equals to	A=10,b=5,A!=b =>true

Logical Operators

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

<table>
<tr><th>Operator</th><th>Meaning</th><th>Example</th></tr>
<tr><td>And</td><td>And</td><td>
<table>
<tr><th>Exp1</th><th>And</th><th>Exp2</th><th>Result</th></tr>
<tr><td>T</td><td>And</td><td>T</td><td>T</td></tr>
<tr><td>T</td><td>And</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>And</td><td>T</td><td>F</td></tr>
<tr><td>F</td><td>And</td><td>F</td><td>F</td></tr>
</table>
</td></tr>
<tr><td>Or</td><td>Or</td><td>
<table>
<tr><th>Exp1</th><th>Or</th><th>Exp2</th><th>Result</th></tr>
<tr><td>T</td><td>Or</td><td>T</td><td>T</td></tr>
<tr><td>T</td><td>Or</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>Or</td><td>T</td><td>F</td></tr>
<tr><td>F</td><td>Or</td><td>F</td><td>F</td></tr>
</table>
</td></tr>
<tr><td>Not</td><td>Not</td><td>
<table>
<tr><th>Not Exp</th><th>Result</th></tr>
<tr><td>T</td><td>F</td></tr>
<tr><td>F</td><td>T</td></tr>
</table>
</td></tr>
</table>

Assignment Operators

The assignment operators are used to assign RHS value to LHS either before calculation or after calculation

Operator	Meaning	Examples
=	Assigns RHS value to LHS	**a=10**
+=	Assigns RHS value to LHS after addition	**a=10 a+=3=>13**
-=	Assigns RHS value to LHS after subtraction	**a=10 a - =3=>7**
*=	Assigns RHS value to LHS after multiplication	**a=10 a*=3=>30**
/=	Assigns RHS value to LHS after Division	**a=10 a/=2=>5**
%=	Assigns RHS value to LHS after modulus	**a=10 a%=3=>1**
**=	Assigns RHS value to LHS after calculating the power for a given number	a = 4, b =2, a**=b =>16
//=		a = 4, b = 3, a//=b => 1

Bitwise Operators

The bitwise operators perform bit by bit operation on the values of the two operands.

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.

<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Membership Operators

Membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
In	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

Identity Operators

The identity operators are used to decide whether an element certain class or type.

Operator	Description
Is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both sides do not point to the same object.

4) Data Types

- Variables can hold values, and every value has a data-type.
- **Python is a dynamically typed language;** hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

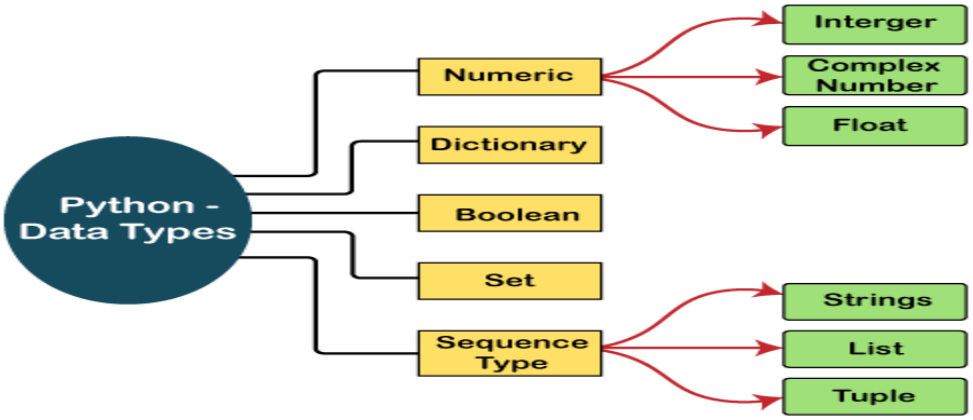
```
a = 5
```

- Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Standard data types

- A variable can hold different types of values

- Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.
  1. Numbers
  2. Sequence Type
  3. Boolean
  4. Set
  5. Dictionary



1. Numbers

- Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type.
- Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Eg:

```
a = 5
print("The type of a", type(a))
b = 40.5
print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

- **Python supports three types of numeric data.**
  1. **Int**
     - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc.
     - Python has no restriction on the length of an integer.
     - Its value belongs to **int**
  2. **Float**
     - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc.

- It is accurate upto 15 decimal points.

3. **complex**
   - A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

2. Sequence Type

1) String

- The string can be defined as the sequence of characters represented in the quotation marks.
- In Python, we can use single, double, or triple quotes to define a string.
- String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.
- In the case of string handling, the operator + is used to concatenate two strings as the operation "*hello*"+" *python*" returns "*hello python*".
- The operator * is known as a repetition operator as the operation "Python" *2 returns 'Python Python'.

2) List

- Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].
- We can use slice [:] operators to access the data of the list.
- The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Eg:

```
list1  = [1, "hi", "Python", 2]
print(type(list1))    #Checking type of given list
print (list1)  #Printing the list1
print (list1[3:])  # List slicing
print (list1[0:2])   # List slicing
print (list1 + list1)  # List Concatenation using + operator
print (list1 * 3)  # List repetation using * operator
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

3) Tuple

- A tuple is similar to the list in many ways.
- Like lists, tuples also contain the collection of the items of different data types.
- The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

- A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Eg: tup = ("hi", "Python", 2)
print (tup) **#Printing the tuple**
print (tup[1:]) **# Tuple slicing**
print (tup[0:1])
print (tup + tup) **# Tuple concatenation using + operator**
print (tup * 3) **# Tuple repatation using * operator**
tup[2] = "hi" **# Adding value to tup. It will throw an error.**
Output:
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)
Traceback (most recent call last):
File "main.py", line 14, in <module>
tup[2] = "hi";
TypeError: 'tuple' object does not support item assignment

3. Dictionary

- Dictionary is an unordered set of a key-value pair of items.
- It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.
- The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Eg: d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
print (d) **# Printing dictionary**
print("1st name is "+d[1]) **# Accesing value using keys**
print("2nd name is "+ d[4])
print (d.keys())
print (d.values())
Output:
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
1st name is Jimmy
2nd name is mike
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])

4. Boolean

- Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false.
- It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.

Eg:# Python program to check the boolean type
print(type(True))
print(type(False))

print(false)

Output:

<class 'bool'>

<class 'bool'>

NameError: name 'false' is not defined

5. Set

- Python Set is the unordered collection of the data type.
- It is iterable, mutable(can modify after creation), and has unique elements.
- In set, the order of the elements is undefined; it may return the changed sequence of the element.
- The set is created by using a built-in function set(), or a sequence of elements is passed in the curly braces and separated by the comma.
- It can contain various types of values. Consider the following example.

 # Creating Empty set

```
set1 = set()
set2 = {'James', 2, 3,'Python'}
  print(set2)
set2.add(10)
  print(set2)
  set2.remove(2)
  print(set2)
```

Output:

{3, 'Python', 'James', 2}

{'Python', 'James', 3, 2, 10}

{'Python', 'James', 3, 10}

5) Literals

- Python Literals can be defined as data that is given in a variable or constant. Python supports the following literals:
    1) String Literals
    2) Numeric Literals
    3) Boolean Literals
    4) Special Literals
    5) Collection Literals

1. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Types of Strings:

There are two types of Strings supported in Python:

a) Single-line String

Strings that are terminated within a single-line are known as Single line Strings.

Example: text1='hello'

b) **Multi-line String –**

A piece of text that is written in multiple lines is known as multiple lines string. There are two ways to create multiline strings:

1) Adding black slash at the end of each line.

```
text1='hello\
      user'
print(text1)
```
Output: hellouser

2) **Using triple quotation marks:-**

```
str2='''welcome
      to  SSSIT'''
      print str2
```
Output:

Welcome
to SSSIT

2. **Numeric literals:**

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

1) **Signed integers**
   Numbers can be both positive and negative with no fractional part.
   eg: 100
2) **Long Integers**
   Integers of unlimited size followed by lowercase or uppercase L
   eg: 87032845L
3) **Float:** Real numbers with both integer and fractional part
   eg: -26.2
4) **Complex**
   In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number
   eg: 3.14j

3. **Boolean literals:**

A Boolean literal can have any of the two values: True or False.

Example - Boolean Literals

```
x = (1 == True)
y = (2 == False)
z = (3 == True)
a = True + 10
b = False + 10
print("x is", x)  print("y is", y)
print("z is", z)  print("a:", a)  print("b:", b)
```
Output:

x is True
y is False
z is False
a: 11
b: 10

4. **Special literals**

- Python contains one special literal i.e., **None.**
- None is used to specify to that field that is not created.
- It is also used for the end of lists in Python.

Example - Special Literals

```
val1=10
val2=None
print(val1, val2)
```

Output: 10 None

5. **Literal Collections**

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

1) **List:**
   - List contains items of different data types. Lists are mutable i.e., modifiable.
   - The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

     Example - List literals

     ```
     list=['John',678,20.4,'Peter']
     list1=[456,'Andrew']
     print(list)
     print(list + list1)
     ```

     Output:

     ['John', 678, 20.4, 'Peter']

     ['John', 678, 20.4, 'Peter', 456, 'Andrew']

2) **Dictionary:**
   - Python dictionary stores the data in the key-value pair.
   - It is enclosed by curly-braces {} and each pair is separated by the commas(,).

     Example

     ```
     dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}
     print(dict)
     ```

     Output: {'name': 'Pater', 'Age': 18, 'Roll_nu': 101}

3) **Tuple:**
   - Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.
   - It is enclosed by the parentheses () and each element is separated by the comma(,).

     Example

     ```
     tup = (10,20,"Dev",[2,3,4])
     print(tup)
     ```

     Output: (10, 20, 'Dev', [2, 3, 4])

4) **Set:**
   - Python set is the collection of the unordered dataset.
   - It is enclosed by the {} and each element is separated by the comma(,).

Example: - Set Literals

```
set = {'apple','grapes','guava','papaya'}
print(set)
```

Output: {'guava', 'apple', 'papaya', 'grapes'}

6) Comments

Comments are essential for defining the code and help us and other to understand the code. By looking the comment, we can easily understand the intention of every line that we have written in code. We can also find the error very easily, fix them, and use in other applications.

In Python, we can apply comments using the # hash character. The Python interpreter entirely ignores the lines followed by a hash character.

Types of Comment

Python provides the facility to write comments in two ways-

1. Single line comment and
2. Multi-line comment.

Single-Line Comment

Single-Line comment starts with the hash # character followed by text

Eg: name = "Thomas" # Assigning string value to the name variable

Multi-Line Comments

Python doesn't have explicit support for multi-line comments but we can use hash # character to the multiple lines.

For example -

```
# we are defining for loop
# To iterate the given list.
# run this code.
```

We can also use another way.

```
" " "
This is an example
Of multi-line comment
Using triple-quotes
" " "
```

5Q) Python String

Ans:

- Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.
- Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

  Syntax: str = "Hi Python !"

  Here, if we check the type of the variable str using a Python script

  print(type(str)), then it will print a string (str).

- In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or docstrings.

Eg:

```
str1 = 'Hello Python'   #Using single quotes
print(str1)
str2 = "Hello Python"   #Using double quotes
print(str2)
#Using triple quotes
str3 = """Triple quotes are generally used for
   represent the multiline or
   docstring'''
print(str3)
```

Output:

```
Hello Python
Hello Python
Triple quotes are generally used for
   represent the multiline or
   docstring
```

Strings indexing and splitting

- Like other languages, the indexing of the Python strings starts from 0.
- For example, The string "HELLO" is indexed as given in the below figure.

str = "HELLO"

H	E	L	L	O
0	1	2	3	4

str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

Example:

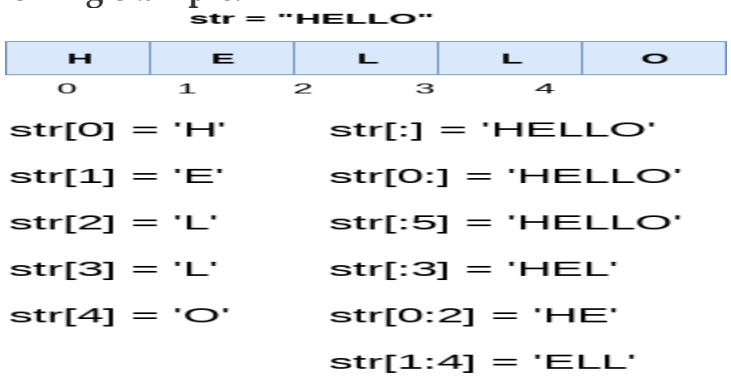
```
str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
print(str[6])  # It returns the IndexError because 6th index doesn't exist
```

Output:

```
H
```

E
L
L
O
IndexError: string index out of range

Slice Operator:
The slice operator [] is used to access the individual characters of the string. However, we can use the : (colon) operator in Python to access the substring from the given string.
Consider the following example.



Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.

Consider the following example:

```
str = "FUTURESOFT"  # Given String
print(str[0:])  # Start Oth index to end
print(str[1:5])  # Starts 1th index to 4th index
print(str[2:4])  # Starts 2nd index to 3rd index
print(str[:3])  # Starts 0th to 2nd index
print(str[4:7])  #Starts 4th to 6th index
```

Output:
FutureSoft
utur
tu
Fut
res
We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on.
Consider the following image.

str = “HELLO”

H	E	L	L	O
-5	-4	-3	-2	-1

str[-1] = ‘O’ str[-3:-1] = ‘LL’

str[-2] = ‘L’ str[-4:-1] = ‘ELL’

str[-3] = ‘L’ str[-5:-3] = ‘HE’

str[-4] = ‘E’ str[-4:] = ‘ELLO’

str[-5] = ‘H’ str[::-1] = ‘OLLEH’

Example

```
str = 'FUTURESOFT'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string
print(str[::-1])
print(str[-12])
```

Output:

```
T
O
FT
SOF
URESO
TOFOSERUTUF
IndexError: string index out of range
```

Deleting the String

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the del keyword.

Eg-1:

```
str = "FUTURESOFT"
del str[1]
```

Output:

```
TypeError: 'str' object doesn't support item deletion
```

Now we are deleting entire string.

Eg-2:

```
str1 = "FUTURESOFT”
del str1
print(str1)
```

Output:

```
NameError: name 'str1' is not defined
```

String Operators

Operator	Description

+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
In	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

Eg:

```
str = "Hello"
str1 = " world"
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
print(r'C://python37') # prints C://python37 as it is written
print("The string str : %s"%(str)) # prints The string str : Hello
```

Output:

```
HelloHelloHello
Hello world
o
ll
False
False
```

C://python37
The string str : Hello
Python String Formatting

6Q) Write about Escape Sequences?

Ans:

These are non-printable characters but its functionality can be observed on the output screen. The backslash(/) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently. The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

Example -

```
print('''They said, "What's there?"''')
print('They said, "What\'s going on?"')
print("They said, \"What's going on?\"")
```

Output:

They said, "What's there?"
They said, "What's going on?"
They said, "What's going on?"

The list of an escape sequence is given below:

Sr.	Escape Sequence	Description	Example
1.	\newline	It ignores the new line.	print("Python1 \ Python2 \ Python3") **Output:** Python1 Python2 Python3
2.	\\	Backslash	print("\\") **Output:** \
3.	\'	Single Quotes	print('\'') **Output:** '
4.	\\"	Double Quotes	print("\"") **Output:** "
5.	\a	ASCII Bell	print("\a")
6.	\b	ASCII Backspace(BS)	print("Hello \b World") **Output:** Hello World
7.	\f	ASCII Formfeed	print("Hello \f World!") Hello World!

8.	\n	ASCII Linefeed	print("Hello \n World!") **Output:** Hello World!
9.	\r	ASCII Carriege Return(CR)	print("Hello \r World!") **Output:** World!
10.	\t	ASCII Horizontal Tab	print("Hello \t World!") **Output:** Hello World!
11.	\v	ASCII Vertical Tab	print("Hello \v World!") **Output:** Hello World!
12.	\ooo	Character with octal value	print("\110\145\154\154\157") **Output:** Hello
13	\xHH	Character with hex value.	print("\x48\x65\x6c\x6c\x6f") **Output:** Hello

7Q) Write about the format() method

Ans:

- The format() method is the most flexible and useful method in formatting strings.
- The curly braces {} are used as the placeholder in the string and replaced by the format() method argument. Let's have a look at the given an example:

Eg:

```
# Using Curly braces
print("{} and {} both are the best friend".format("Devansh","Abhishek"))
#Positional Argument
print("{1} and {0} best players ".format("Virat","Rohit"))
#Keyword Argument
print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))
```

Output:

Devansh and Abhishek both are the best friend
Rohit and Virat best players
James,Peter,Ricky

Python String Formatting Using % Operator

- Python allows us to use the format specifiers used in C's printf statement.
- The format specifiers in Python are treated in the same way as they are treated in C. However, Python provides an additional operator %, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

  Consider the following example.

  Integer = 10;

```
Float = 1.290
String = "Devansh"
print("Hi I am Integer ... My value is %d\nHi I am float ... My value is
%f\nHi I am string ... My value is %s"%(Integer,Float,String))
```

Output:

```
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
Hi I am string ... My value is Devansh
```

8Q) Conditional / Decision Making Statements

Ans:

The conditional statements are used to control the flow of execution of statements of a program. Python support different conditional statements. They are

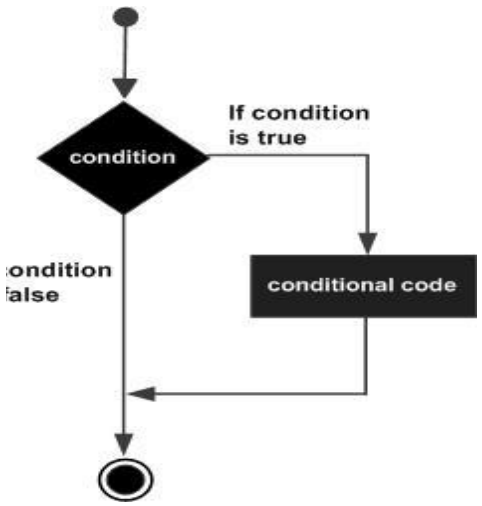
1. Simple if
2. If..else
3. If..elif..else
4. Nested if

1) Simple if:

- This conditional statement executes true statements only when the condition is true, otherwise it skips the if statement
- The if statement checks the condition first.
- If the condition evaluates to True, it executes the statements in the if-block. Otherwise, it ignores the statements.
- Note that the colon (:) that follows the condition is very important. If you forget it, you'll get a syntax error.

Syn:

```
if expression:
        Statement1;
        Statement2;
```

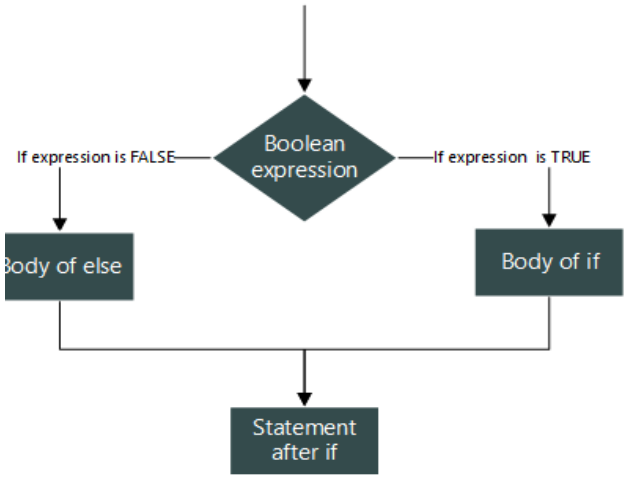


2) if .. else:

This conditional statement can execute true statements only when the condition is true otherwise it executes false statements.

Syn:

```
if expression:
        statement1;
        statement2;
else:
        statement1;
        statement2;
```



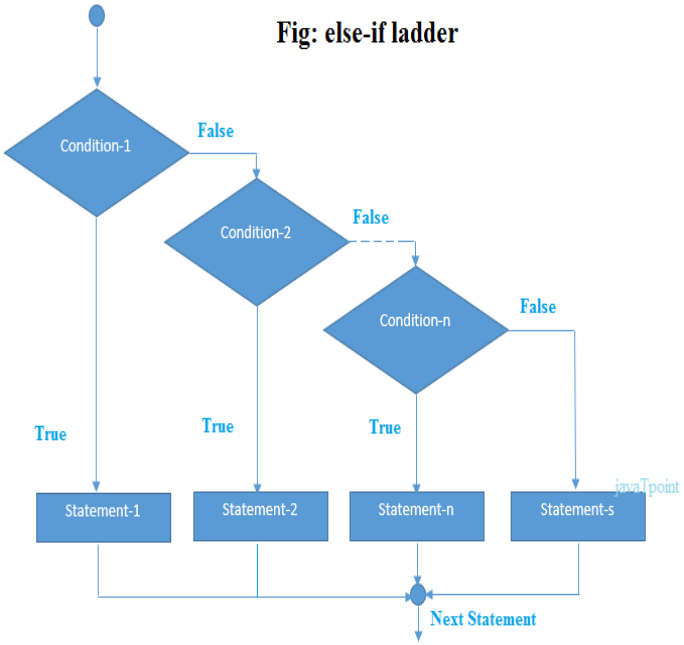
3) if .. elif.. else:

It is also called as a branching statement or Ladder statement. This conditional statement executes statements based on its respective condition.

Syn: if expression1:

```
        statement1;
        statement2;
 elif expression2:
        statement1;
        statement2;
else:
        statement1;
        statement2;
```

Fig: else-if ladder

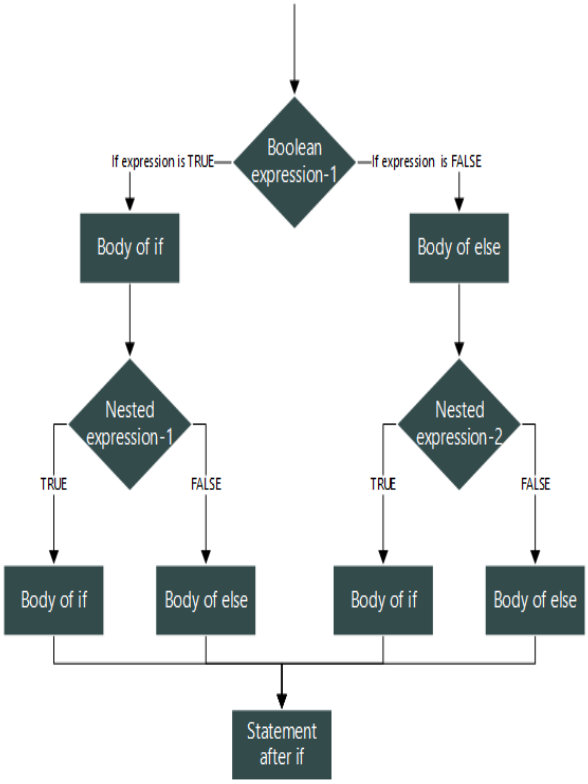


4) Nested if:

A if statement which can execute within if statement itself it is called as nested if conditional statement. It is also called as a multi-level branching statement. In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syn:

```
if exp1:
        if exp1.1:
                statement1
                statement2
        elif exp1.2:
                statement1
                statement2
          else:
                statement1;
                statement2;
elif exp2:
     if exp2.1:
                statement1;
                statement2;
        elif exp1.2:
                statement1;
                statement2;
```



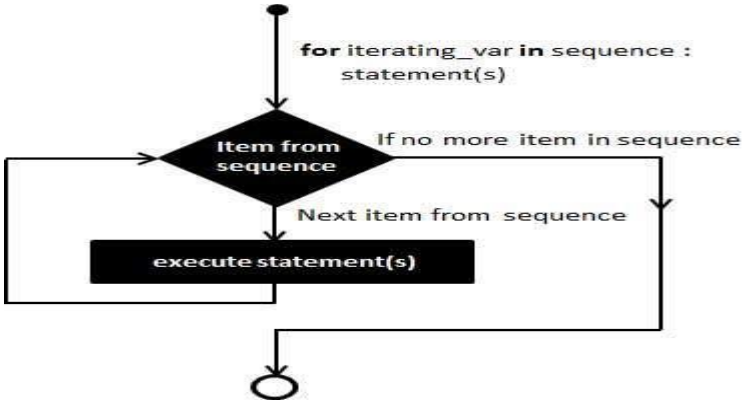
9Q) Write about Loops?

Ans: A loop statement allows us to execute a statement or group of statements multiple times. Python programming language provides following types of loops to handle looping requirements.

1) For..loop
2) While..loop
3) Nested loop

1. For..loop

- Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.
  Syn: for value in sequence:
  code block
- In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.
- Loop iterates until the final item of the sequence are reached.



Examples:

Print the text "Hello, World!" 5 times.

```
list = [1, 2, 3, 4, 5]
for num in list:
   print("Hello, World!")
```

Note: The variable num is not used in the code, so we can use the below syntax (use underscore):

```
list = [1, 2, 3, 4, 5]
for _ in list:
   print("Hello, World!")
```

```
# Prints out the numbers 0,1,2,3,4
for x in range(5):
   print(x)
# Prints out 3,4,5
for x in range(3, 6):
   print(x)
# Prints out 3,5,7
for x in range(3, 8, 2):
   print(x)
```

```
# Program to find the sum of all numbers stored in a list
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0
# iterate over the list
for val in numbers:
   sum = sum+val
print("The sum is", sum)
```

```
# Program to Create a list of all the even numbers between 1 and 10
even_nums = []
for i in range(1, 11):
   if i % 2 == 0:
      even_nums.append(i)
print("Even Numbers: ", even_nums)
for letter in 'Python':     # First Example
  print 'Current Letter :', letter
fruits = ['banana', 'apple',  'mango']
```

```
for fruit in fruits:        # Second Example
   print 'Current fruit :', fruit
print "Good bye!"
```

o/p:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Eg:

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]
print "Good bye!"
```

o/p:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

for loop with else

- A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.
- The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs.

Eg: to Use else with a for loop

```
iterator = (1, 2, 3, 4)
for item in iterator:
    print(item)
else:
    print("No more items in the iterator")
```

The range() function

- We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.
- The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and __getitem__ operations.

- This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.
- To force this function to output all the items, we can use the function list().
  Eg:
  print(range(10))
  print(list(range(10)))
  print(list(range(2, 8)))
  print(list(range(2, 20, 3)))

2. While Loop

While loops are used in Python to iterate until a specified condition is met. However, the statement in the program that follows the while loop is executed once the condition changes to false.

Syn: while <condition>:
code block

 # Python program to show how to use a while loop

```
counter = 0
while counter < 10:
   counter = counter + 3
   print("Python Loops")
```

Output:

```
Python Loops
Python Loops
Python Loops
Python Loops
```

Using else Statement with while Loops

We can use the else statement with the while loop also. It has the same syntax.

Syn: while <condition>:
code block
else:
statements

#Python program to show how to use else statement with the while loop

```
counter = 0
while (counter < 10):
   counter = counter + 3
   print("Python Loops") # Executed until condition is met
else:
   print("Code block inside the else statement")
```

Output:

```
Python Loops
Python Loops
Python Loops
Python Loops
Code block inside the else statement
```

Single statement while Block

- The loop can be declared in a single statement, as seen below.
- This is similar to the if-else block, where we can write the code block in a single line.

 # Python program to show how to write a single statement while loop

```
counter = 0
while (count < 3): print("Python Loops")
```

3. Nested loop

Python programming language allows to use one loop inside another loop.

Syn: for iterating_var in sequence:

```
        for iterating_var in sequence:
                statements(s)
                statements(s)
```

Syn: while expression:

```
        while expression:
                statement(s)
                statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Eg:

```
i = 2
while(i < 50):
  j = 2
  while(j <= (i/j)):
    if not(i%j): break
    j = j + 1
  if (j > i/j) : print i, " is prime"
  i = i + 1
print "Good bye!"
```

10Q) Numeric Functions

Ans: These functions are used to represent numbers in different forms. The methods are like below −

S.No.	Function	Description
1	**ceil(x)**	Return the Ceiling value. It is the smallest integer, greater or equal to the number x.
2	**fabs(x)**	Returns the absolute value of x.
3	**factorial(x)**	Returns factorial of x. where x ≥ 0
4	**floor(x)**	Return the Floor value. It is the largest integer, less or equal to the number x.

5	**fsum(iterable)**	Find sum of the elements in an iterable object
6	**Fmod(x,y)**	Return a floating point number value after calculating module of given parameters x and y.
7	**gcd(x, y)**	Returns the Greatest Common Divisor of x and y
8	**isnan(x)**	Checks whether x is not a number.
9	**remainder(x, y)**	Find remainder after dividing x by y.
Power and Logarithmic Functions		
1	**Exp(a)**	This function returns the value of **e raised to the power a (e**a)**
2	**Log(a,b)**	This function returns the logarithmic **value of a with base b**. If base is not mentioned, the computed value is of natural log.
3	**Log2(a)**	This function computes value of **log a with base 2**.
4	**Log10(a)**	his function computes value of **log a with base 10**.
5	**pow(x, y)**	Return the x to the power y value.
6	**Sqrt(x)**	Finds the square root of x
Trigonometric Functions		
1	**Sin(x)**	Return the sine of x in radians
2	**Cos(x)**	Return the cosine of x in radians
3	**Tan(x)**	Return the tangent of x in radians

11Q) Random Numbers

Ans:

- Python defines a set of functions that are used to generate or manipulate random numbers through the **random module.**
- Functions in the random module rely on a pseudo-random number generator function **random()**, which generates a random float number between 0.0 and 1.0. These particular type of functions is used in a lot of games, lotteries, or any application requiring a random number generation.

Eg: for Generating Random Number using random() function

```
import random
```

```
num = random.random()
print(num)
```
Output:
0.30078080420602904

```
# Python3 program to demonstrate the use of# choice() method
import random
list1 = [1, 2, 3, 4, 5, 6]
print(random.choice(list1))
string = "striver"
print(random.choice(string))
```
Output
5
T

Method 2: Generating random number list in Python randrange(beg, end, step)
The random module offers a function that can generate random numbers from a specified range and also allows room for steps to be included, called **randrange().**

```
# Python code to demonstrate the working of choice() and randrange()
import random
print("A random number from list is : ", end="")
print(random.choice([1, 4, 8, 10, 3]))
print("A random number from range is : ", end="")
print(random.randrange(20, 50, 3))
```
Output:
A random number from list is : 4
A random number from range is : 41

Method 3: Generating random number list in Python using seed()
The seed function is used to save the state of a random function so that it can generate some random numbers on multiple executions of the code on the same machine or on different machines (for a specific seed value). The seed value is the previous value number generated by the generator. For the first time when there is no previous value, it uses the current system time.

```
# Python code to demonstrate the working of random() and seed()
import random
print("A random number between 0 and 1 is : ", end="")
print(random.random())
random.seed(5)
print("The mapped random number with 5 is : ", end="")
print(random.random())
random.seed(7)
print("The mapped random number with 7 is : ", end="")
print(random.random())
random.seed(5)
```

```
print("The mapped random number with 5 is : ", end="")
print(random.random())
random.seed(7)
print("The mapped random number with 7 is : ", end="")
print(random.random())
```

Output:

```
A random number between 0 and 1 is : 0.510721762520941
The mapped random number with 5 is : 0.6229016948897019
The mapped random number with 7 is : 0.32383276483316237
The mapped random number with 5 is : 0.6229016948897019
The mapped random number with 7 is : 0.32383276483316237
```

Method 4: Generating random number list in Python using shuffle()

- It is used to shuffle a sequence (list).
- Shuffling means changing the position of the elements of the sequence. Here, the shuffling operation is in place.

Eg:

```
import random
sample_list = ['A', 'B', 'C', 'D', 'E']
print("Original list : ")
print(sample_list)
random.shuffle(sample_list)
print("\nAfter the first shuffle : ")
print(sample_list)
random.shuffle(sample_list)
print("\nAfter the second shuffle : ")
print(sample_list)
```

Output:

```
Original list:
['A', 'B', 'C', 'D', 'E']
After the first shuffle :
['A', 'B', 'E', 'C', 'D']
After the second shuffle :
['C', 'E', 'B', 'D', 'A']
```

12Q) What is a function in Python? Write the advantages of Functions in Python?

Ans:

- A function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.

Advantages

1. By including functions, we can prevent repeating the same code block repeatedly in a program.

2. Python functions, once defined, can be called many times and from anywhere in a program.
3. If our Python program is large, it can be separated into numerous functions which is simple to track.
4. The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

13Q) Write about types of functions

Ans:

The functions are of two types. They are

1. Built in functions
2. User-defined Functions

1) Built in functions

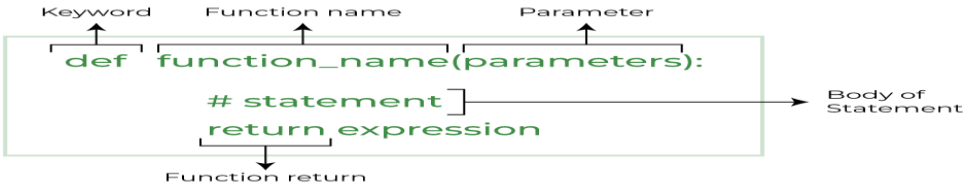
The functions which are already existed in python, they are called as built in functions.

Eg: abs(), bin(), ascii(), bool(), chr(), dict(), dir(), divmod(), exec(), sin(), cos(),

2) User-Defined Fucntions:

Functions that we define ourselves to do certain specific task are referred as user-defined functions.

Syn:



- **A function definition that consists of the following components**
  - **Keyword def** that marks the start of the function header
  - A **function name** to uniquely identify the function
  - **Parameters** (arguments) through which we pass values to a function. They are optional.
  - **A colon (:)** to mark the end of the function header
  - One or more valid python statements that make up the function body, Statements must have the same indentation level (usually 4 spaces).
  - An optional return statement to return a value from the function

14Q) Write about the Function Arguments?

Ans: The following are the types of arguments that we can use to call a function:

1. Default arguments
2. Keyword arguments
3. Required arguments
4. Variable-length arguments

1) Default Arguments

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called.

 # Python program to demonstrate default arguments

```
def myFun(x, y=50):
        print("x: ", x)
        print("y: ", y)

myFun(10)
```

Output

```
x:  10
y:  50
```

2) **Keyword arguments**

The idea is to allow the caller to specify the argument name with values so that caller does not need to remember the order of parameters.

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
        print(firstname, lastname)
student(firstname='Future', lastname='Practice')
student(lastname='Practice', firstname='Future')
```

Output

```
Future Practice
Future Practice
```

3) Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

We must send two arguments to the function function() in the correct order, or it will return a syntax error,

```
# Python code to demonstrate the use of default arguments
def function( num1, num2 ):
            print("num1 is: ", num1)
             print("num2 is: ", num2)
print( "Passing out of order arguments" )
function( 30, 20 )
# Calling function and passing only one argument
print( "Passing only one argument" )
try:
   function( 30 )
except:
   print( "Function needs two positional arguments" )
```

4) Variable-length arguments

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args (Non-Keyword Arguments)
- **kwargs (Keyword Arguments)

```
# Python program to illustrate *args for variable number of arguments
def myFun(*argv):
```

```
        for arg in argv:
            print(arg)
myFun('Hello', 'Welcome', 'to', 'Future Soft')
```

Output

```
Hello
Welcome
to
Future Soft
```

```
# Python program to illustrate *kwargs for variable number of keyword
# arguments
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))

myFun(first='Future', mid='for', last='Future')
```

Output

```
first == Future
mid == for
last == Future
```

15Q) Write about Calling a Function?

Ans:

- A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.
- After a function's fundamental framework is complete, we can call it from anywhere in the program. The following is an example of how to use the a_function function.

Eg:

```
def a_function( string ):
    "This prints the value of length of string"
    return len(string)
print( "Length of the string Functions is: ", a_function( "Functions" ) )
print( "Length of the string Python is: ", a_function( "Python" ) )
```

Output:

```
Length of the string Functions is:  9
Length of the string Python is:  6
```

16Q) Write about Pass by Reference vs. Value?

Ans: In the Python programming language, all arguments are supplied by reference. It implies that if we modify the value of an argument within a function, the change is also reflected in the calling function.

Eg

```
def square( my_list ):
   squares = []
   for l in my_list:
      squares.append( l**2 )
   return squares
# calling the defined function
list_ = [45, 52, 13];
```

```
result = square( list_ )
print( "Squares of the list is: ", result )
```

Output:

Squares of the list is: [2025, 2704, 169]

UNIT –II

1Q) What are Python Modules?

Ans:

Modules provide us with a way to share reusable functions. A module is simply a "Python file" which contains code we can reuse in multiple Python programs. A module may contain functions, classes, lists, etc.

Modules in Python can be of two types:

- Built-in Modules.
- User-defined Modules.

1. **Built-in Modules in Python**
   One of the many superpowers of Python is that it comes with a "rich standard library". This rich standard library contains lots of built-in modules. Hence, it provides a lot of reusable code.

   To name a few, Python contains modules like "os", "sys", "datetime", "random". You can import and use any of the built-in modules whenever you like in your program.

2. **User-Defined Modules in Python**
   Another superpower of Python is that it lets you take things in your own hands. You can create your own functions and classes, put them inside modules and voila! You can now include hundreds of lines of code into any program just by writing a simple import statement.
   To create a module, just put the code inside a .py file.

Create a simple Python module

```
# A simple module, calc.py
def add(x, y):
        return (x+y)

def subtract(x, y):
        return (x-y)
```

Import Module in Python

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax: import module

Importing modules in Python

Now, we are importing the **calc** that we created earlier to perform add operation.

```
# importing module calc.py
import calc
print(calc.add(10, 2))
```

The from-import Statement in Python

Python's *from* statement lets you import specific attributes from a module without importing the module as a whole.

Eg: Importing specific attributes from the module

Here, we are importing specific sqrt and factorial attributes from the math module.

```
# importing sqrt() and factorial from the module math
from math import sqrt, factorial
print(sqrt(16))
```

```
print(factorial(6))
```

Import all Names

The * symbol used with the from import statement is used to import all the names from a module to a current namespace.

Syntax: from module_name import *

Eg: From import *

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the module math
from math import *
print(sqrt(16))
print(factorial(6))
```

Renaming the Python module

We can rename the module while importing it using the keyword.

Syntax: *Import* ***Module_name*** *as* ***Alias_name***

```
# importing sqrt() and factorial from the module math
import math as mt
print(mt.sqrt(16))
print(mt.factorial(6))
```

2Q) File Handling

Ans The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.
Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

- Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed.
- The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syn: file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

Access mode	Description
R	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
Rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
W	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
Wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
A	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
Ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Eg:

```
fileptr = open("file.txt","r")
if fileptr:
   print("file is opened successfully")
```

Output:

```
 <class '_io.TextIOWrapper'>
 file is opened successfully
```

close(): This method is used to close the file which was opened

Syn: fileobject.close()

Eg:

```
fileptr = open("file.txt","r")
if fileptr:
   print("file is opened successfully")
fileptr.close()
```

The with statement

- The **with** statement was introduced in python 2.5.
- The with statement is useful in the case of manipulating the files.
- It is used in the scenario where a pair of statements is to be executed with a block of code in between.

  Syn: with open(<file name>, <access mode>) as <file-pointer>:

Example

```
with open("file.txt",'r') as f:
   content = f.read();
   print(content)
```

Example of writing the file

```
fileptr = open("file2.txt", "w")
fileptr.write("""Python is the modern day language. It makes things so simple.
It is the fastest-growing programing language""")
fileptr.close()
```

Output:

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

Example 2

```
    fileptr = open("file2.txt","a")
    fileptr.write(" Python has an easy syntax and user-friendly interaction.")
    fileptr.close()
```

Output:

Python is the modern day language. It makes things so simple.

Read():

- To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

  Syn: fileobj.read(<count>)

  Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Example

```
fileptr = open("file2.txt","r")
content = fileptr.read(10)
print(type(content))
print(content)
fileptr.close()
```

Output:

```
<class 'str'>
Python is
```

Read file through for loop: We can read the file using for loop.

eg:

```
fileptr = open("file2.txt","r");
for i in fileptr:
   print(i) # i contains each line of the file
```

Output:

Python is the modern day language.
It makes things so simple.
Python has easy syntax and user-friendly interaction.

Read Lines of the file

- Python facilitates to read the file line by line by using a function **readline()** method.
- The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Example 1: Reading lines using readline() function

```
fileptr = open("file2.txt","r");
content = fileptr.readline()
content1 = fileptr.readline()
print(content)
print(content1)
fileptr.close()
```

Output:

Python is the modern day language.

Example 2: Reading Lines Using readlines() function

```
fileptr = open("file2.txt","r");
   content = fileptr.readlines()
print(content)
fileptr.close()
```

Output:

['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']

Example 1: Creating a new file

```
fileptr = open("file2.txt","x")
print(fileptr)
if fileptr:
print("File created successfully")
```

Output:

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
File created successfully
```

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

```
fileptr = open("file2.txt","r")
print("The filepointer is at byte :",fileptr.tell())
content = fileptr.read();
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

Syntax: <file-ptr>.seek(offset[, **from**)

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Example

```
fileptr = open("file2.txt","r")
print("The filepointer is at byte :",fileptr.tell())
fileptr.seek(10);
#tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

3Q) What is an Exception Handling in Python?

Ans: An exception is an error which happens at the time of execution of a program. However, while running a program, Python generates an exception that should be handled to avoid your program to crash. In Python language, exceptions trigger automatically on errors, or they can be triggered and intercepted by your code.

The exception indicates that, although the event can occur, this type of event happens infrequently. When the method is not able to handle the exception, it is thrown to its caller function. Eventually, when an exception is thrown out of the main function, the program is terminated abruptly.

4Q) Common Examples of Exception:

Ans:

- Division by Zero
- Accessing a file which does not exist.
- Addition of two incompatible types
- Trying to access a nonexistent index of a sequence
- Removing the table from the disconnected database server.
- ATM withdrawal of more than the available amount

5Q) Python Exception Handling Mechanism

Ans: Exception handling is managed by the following 5 keywords:

1. try
2. catch
3. finally
4. throw

Python Try Statement

- A try statement includes keyword try, followed by a colon (:) and a suite of code in which exceptions may occur. It has one or more clauses.
- During the execution of the try statement, if no exceptions occurred then, the interpreter ignores the exception handlers for that specific try statement.
- In case, if any exception occurs in a try suite, the try suite expires and program control transfers to the matching except handler following the try suite.

  Syntax: try:

  statement(s)
- In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

 # Python code to catch an exception and handle it using try and except code blocks

```
a = ["Python", "Exceptions", "try and except"]
try:
   for i in range( 4 ):
     print( "The index and element from the array is" i, a[i] )
```

except:
print ("Index out of range")
Output:
The index and element from the array is 0 Python
The index and element from the array is 1 Exceptions
The index and element from the array is 2 try and except
Index out of range

The catch Statement
Catch blocks take one argument at a time, which is the type of exception that it is likely to catch. These arguments may range from a specific type of exception which can be varied to a catch-all category of exceptions.
Rules for catch block:

- You can define a catch block by using the keyword catch
- Catch Exception parameter is always enclosed in parentheses
- It always represents the type of exception that catch block handles.
- An exception handling code is written between two {} curly braces.
- You can place multiple catch block within a single try block.
- You can use a catch block only after the try block.
- All the catch block should be ordered from subclass to superclass exception.

Syntax

```
try:
   #Code that may raise an error
   #Raising your own errors
   raise ErrorType("Error message")
except ErrorType as e: #code to run if error occurs
   #code to run if error is raised
else:
   #code to run if no error is raised
```

Finally

- Finally block always executes irrespective of an exception being thrown or not. The final keyword allows you to create a block of code that follows a try-catch block.
- Finally, clause is optional. It is intended to define clean-up actions which should be that executed in all conditions.

Eg:

```
try:
        raise KeyboardInterrupt
finally:
   print 'welcome, world!'
```

Output
Welcome, world!
KeyboardInterrupt
Finally, clause is executed before try statement.

Raise Statement in Python

The raise statement specifies an argument which initializes the exception object. Here, a comma follows the exception name, and argument or tuple of the argument that follows the comma.

Syntax: raise [Exception [, args [, traceback]]]

In this syntax, the argument is optional, and at the time of execution, the exception argument value is always none.

5Q) What is an Error? Explain different types of Errors in Python?

Ans:

The most common reason of an error in a Python program is when a certain statement is not in accordance with the prescribed usage. Such an error is called a syntax error. The Python interpreter immediately reports it, usually along with the reason.

Types of Errors

1) **Syntax errors / Parsing Errors**
   - Syntax errors are the most basic type of error.
   - They arise when the Python parser is unable to understand a line of code.
   - Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.
   - In IDE, it will highlight where the syntax error is.
   - Most syntax errors are typos, incorrect indentation, or incorrect arguments.
2) **Logical Errors**

   These are the most difficult type of error to find, because they will give unpredictable results and may crash your program. A lot of different things can happen if you have a logic error.
3) **Run Time Errors**

   Many times though, a program results in an error after it is run even if it doesn't have any syntax error. Such an error is a runtime error, called an exception. A number of built-in exceptions are defined in the Python library. Some of the built in exceptions in Python are

Exception	Description
AttributeError	Raised on the attribute assignment or reference fails.
EOFError	Raised when the input() function hits the end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raised when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+c)
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in the local or global scope.

Exception	Description
NotImplementedError	Raised by abstract methods.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
RuntimeError	Raised when an error does not fall under any other category.
SyntaxError	Raised by the parser when a syntax error is encountered.
IndentationError	Raised when there is an incorrect indentation.
SystemError	Raised when the interpreter detects internal error.
SystemExit	Raised by the sys.exit() function.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of a division or module operation is zero.

6Q) Python Classes and Objects

Ans:

Class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
  Eg.: Myclass.Myattribute
- A class is a user-defined blueprint or prototype from which objects are created.
- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new type of object, allowing new instances of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.
- Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.
  Syn:
  class ClassName:
  obj = ClassName()
  print(obj.atrr)

Class Objects

- It is a basic unit of OOP and represents the real life entities.
- It is an instance of class
- A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example:

Eg: # Python3 program to demonstrate instantiating a class

```
class Dog:
        attr1 = "mammal"
        attr2 = "dog"
        def fun(self):
                print("I'm a", self.attr1)
                print("I'm a", self.attr2)
Rodger = Dog()
print(Rodger.attr1)
Rodger.fun()
```

Python Class Variables

- In Object-oriented programming, when we design a class, we use instance variables and class variables.
- In Class, attributes can be defined into two parts:
  - **Instance variables**: If the value of a variable varies from object to object, then such variables are called instance variables.
  - **Class Variables**: A class variable is a variable that is declared inside of class, but outside of any instance method or __init__() method.

Create Class Variables

- A class variable is declared inside of class, but outside of any instance method or __init__() method.
- By convention, typically it is placed right below the class header and before the constructor method and other methods.

Accessing Class Variables

- We can access static variables either by class name or by object reference, but it is recommended to use the class name.
- In Python, we can access the class variable in the following places
  - Access inside the constructor by using either self parameter or class name.
  - Access class variable inside instance method by using either self of class name
  - Access from outside of class by using either object reference or class name.

Example 1: Access Class Variable in the constructor

```
class Student:
    school_name = 'ABC School '
# constructor
    def __init__(self, name):
        self.name = name
        print(self.school_name)
        print(Student.school_name)
s1 = Student('Emma')
```

Output

```
ABC School
ABC School
```

Example 2: Access Class Variable in Instance method and outside class

```
class Student:
    school_name = 'ABC School '
    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
    # Instance method
    def show(self):
        print('Inside instance method')
        print(self.name, self.roll_no, self.school_name)
        print(Student.school_name)
s1 = Student('Emma', 10)
s1.show()
print('Outside class')
print(s1.school_name)
print(Student.school_name)
```

Output

```
Inside instance method
Emma 10 ABC School
ABC School
Outside class
ABC School
```

ABC School

7Q) Differences between Instance variables and Class Variables

Ans:

Instance Variable	Class Variable
Instance variables are not shared by objects. Every object has its own copy of the instance attribute	Class variables are shared by all instances.
Instance variables are declared inside the constructor i.e., the __init__() method.	Class variables are declared inside the class definition but outside any of the instance methods and constructors.
It is gets created when an instance of the class is created.	It is created when the program begins to execute.
Changes made to these variables through one object will not reflect in another object.	Changes made in the class variable will reflect in all objects.

8Q) What is Class Method in Python

Ans:

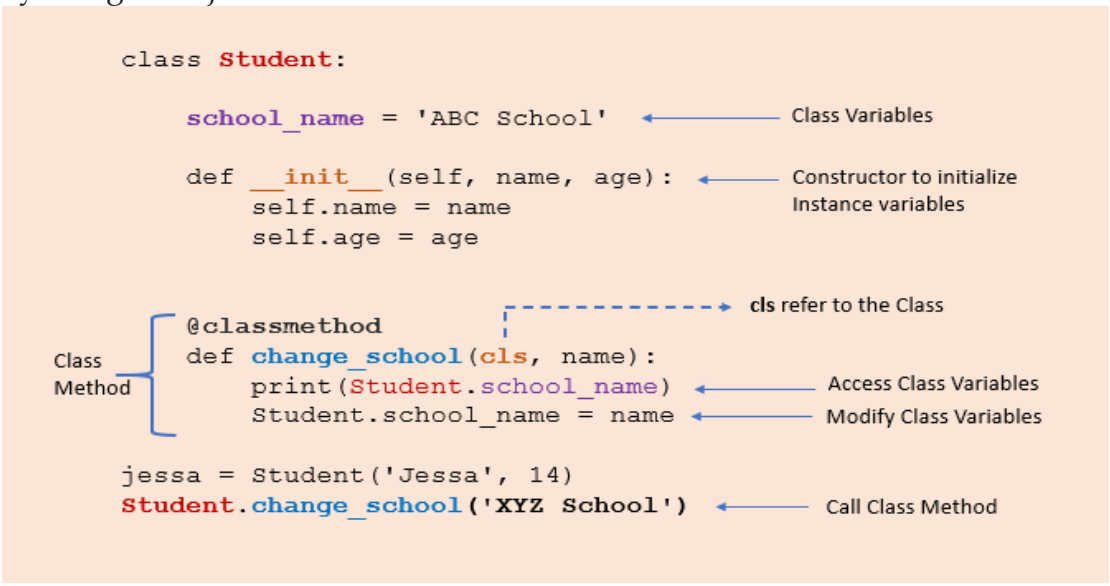
In Object-oriented programming, we use instance methods and class methods. Inside a Class, we can define the following three types of methods.

1. **Instance method**: Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods. It must have a self parameter to refer to the current object.
2. **Class method**: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method. The class method has a cls parameter which refers to the class.
3. **Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't take any parameters like self and cls.

Class Method:

- Class methods are methods that are called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.
- **A class method is bound to the class** and not the object of the class. It can access only class variables.
- It can modify the class state by changing the value of a class variable that would apply across all the class objects.
- In method implementation, if we use only class variables, we should declare such methods as class methods. The class method has a cls as the first parameter, which refers to the class.
- Class methods are used when we are **dealing with factory methods**. Factory methods are those methods that **return a class object for different use cases**. Thus, factory methods create concrete implementations of a common interface.

- The class method can be called using ClassName.method_name() as well as by using an object of the class.



- Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method using the @classmethod decorator or classmethod() function.
- Class methods are defined inside a class, and it is pretty similar to defining a regular function.
- The class method can only access the class attributes, not the instance attributes

Example 1: Create Class Method Using @classmethod Decorator

- To make a method as class method, add @classmethod decorator before the method definition, and add cls as the first parameter to the method.
- The @classmethod decorator is a built-in function decorator. In Python, we use the @classmethod decorator to declare a method as a class method.
- The @classmethod decorator is an expression that gets evaluated after our function is defined.

#Example to create a Student class object using the class method

```
from datetime import date
class Student:
   def __init__(self, name, age):
      self.name = name
      self.age = age
@classmethod
   def calculate_age(cls, name, birth_year):
      return cls(name, date.today().year - birth_year)
   def show(self):
      print(self.name + "'s age is: " + str(self.age))
jessa = Student('Jessa', 20)
jessa.show()
joy = Student.calculate_age("Joy", 1995)
joy.show()
```

Output

Jessa's age is: 20

John's age is: 26

Example 2: Create Class Method Using classmethod() function

Apart from a decorator, the built-in function classmethod() is used to convert a normal method into a class method. The classmethod() is an inbuilt function in Python, which returns a class method for a given function.

Syntax: classmethod(function)

```
class School:
   name = 'ABC School'
   def school_name(cls):
      print('School Name is :', cls.name)
School.school_name = classmethod(School.school_name)
School.school_name()
```

Output

School Name is : ABC School

9Q) Python Constructor

Ans:

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.
- In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.
- Constructors can be of two types.
  1. Parameterized Constructor
  2. Non-parameterized Constructor

Creating the constructor in python

- In Python, the method the **__init__()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self-**keyword as a first argument which allows accessing the attributes or method of the class.
- We can pass any number of arguments at the time of creating the class object, depending upon the **__init__()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Example

```
class Employee:
   def __init__(self, name, id):
      self.id = id
      self.name = name
   def display(self):
      print("ID: %d \nName: %s" % (self.id, self.name))
emp1 = Employee("John", 101)
emp2 = Employee("David", 102)
```

 # accessing display() method to print employee 1 information

```
emp1.display()
emp2.display()
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class.

Example

```
class Student:
   count = 0
   def __init__(self):
      Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

Example

```
class Student:
   # Constructor - non parameterized
   def __init__(self):
      print("This is non parametrized constructor")
   def show(self,name):
      print("Hello",name)
student = Student()
student.show("John")
```

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the self.

Example

```
class Student:
   # Constructor - parameterized
   def __init__(self, name):
      print("This is parametrized constructor")
      self.name = name
   def show(self):
      print("Hello",self.name)
student = Student("John")
student.show()
```

Output:
This is parametrized constructor
Hello John

Python Default Constructor
When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects.
Example

```
class Student:
    roll_num = 101
    name = "Joseph"
    def display(self):
        print(self.roll_num,self.name)
st = Student()
st.display()
```

Output:
101 Joseph
More than One Constructor in Single class

```
class Student:
    def __init__(self):
        print("The First Constructor")
    def __init__(self):
        print("The second contructor")
st = Student()
```

Output:
The Second Constructor
Note:
In the above code, the object st called the second constructor whereas both have the same configuration. The first method is not accessible by the st object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

10Q) Python Inheritance

Ans:

- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name

Syn: class **derived-**class**(base** class**):**
<**class**-suite>

- A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Example 1

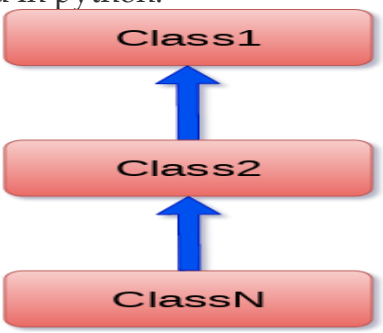
```
class Animal:
def speak(self):
print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
def bark(self):
print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

Dog barking
Animal Speaking

Multi-Level inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



Syn:

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
.
```

Example

```
class Animal:
def speak(self):
print("Animal Speaking")
```

```
#The child class Dog inherits the base class Animal
class Dog(Animal):
def bark(self):
print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
def eat(self):
print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

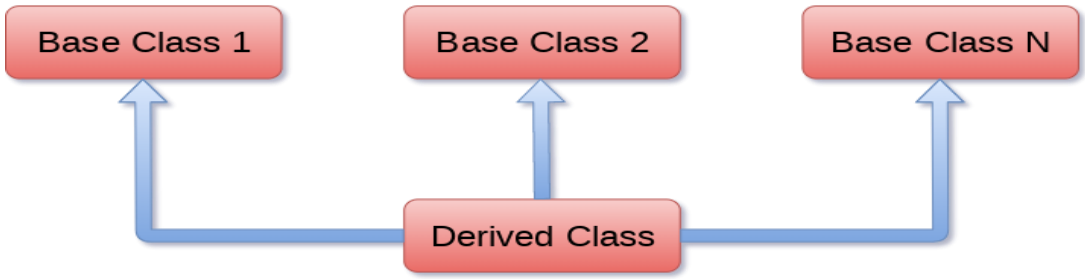
Dog barking

Animal speaking

Eating Bread…

Multiple inheritance

- Python provides us the flexibility to inherit multiple base classes in the child class.



Syntax

```
class Base1:
<class-suite>

class Base2:
<class-suite>
.
.
.
class BaseN:
<class-suite>

class Derived(Base1, Base2, ..... BaseN):
<class-suite>
```

Example

```
class Calculation1:
    def Summation(self,a,b):
            return a+b;
class Calculation2:
```

```
        def Multiplication(self,a,b):
                return a*b;
    class Derived(Calculation1,Calculation2):
        def Divide(self,a,b):
                return a/b;
    d = Derived()
    print(d.Summation(10,20))
    print(d.Multiplication(10,20))
    print(d.Divide(10,20))
```

Output:

30
200
0.5

11Q) Data abstraction in python

Ans:

- Abstraction is an important aspect of object-oriented programming.
- In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Example

```
class Employee:
   __count = 0;
   def __init__(self):
      Employee.__count = Employee.__count+1
   def display(self):
      print("The number of employees",Employee.__count)
emp = Employee()
emp2 = Employee()
try:
   print(emp.__count)
finally:
   emp.display()
```

Output:

The number of emploees 2
AttributeError: 'Employee' object has no attribute '_count'

UNIT – III

1Q) What is NumPy? Write its advantages?

Ans:

- NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.
- **Travis Oliphant** created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.
- It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed.
- NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices.
- Nowadays, NumPy in combination with SciPy and Mat-plotlib is used as the replacement to MATLAB as Python is more complete and easier programming language than MATLAB.

Advantages

There are the following advantages of using NumPy for data analysis.

1. NumPy performs array-oriented computing.
2. It efficiently implements the multidimensional arrays.
3. It performs scientific computations.
4. It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
5. NumPy provides the in-built functions for linear algebra and random number generation.

2Q) Write about NumPy Ndarray?

Ans:

- Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements. In other words, we can define a ndarray as the collection of the data type (dtype) objects.
- The ndarray object can be accessed by using the 0 based indexing. Each element of the Array object contains the same size in the memory.

Creating a ndarray object

The ndarray object can be created by using the array routine of the numpy module. For this purpose, we need to import the numpy.

```
>>> a = numpy.array
```

We can also pass a collection object into the array routine to create the equivalent n-dimensional array.

Syntax:

```
>>> numpy.array(object, dtype = None, copy = True, order = None, subok = False, n
dmin = 0)
```

The parameters are described in the following table.

Parameter	Description
object	It represents the collection object. It can be a list, tuple, dictionary, set, etc.
dtype	We can change the data type of the array elements by changing this option to the specified type. The default is none.
Copy	It is optional. By default, it is true which means the object is copied.
order	There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any)
subok	The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true.
ndmin	It represents the minimum dimensions of the resultant array.

Eg:

```
>>> a = numpy.array([1, 2, 3])
```

```
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([1,2,3])
>>> print(a)
[1 2 3]
>>> print(type(a))
<class 'numpy.ndarray'>
>>>
```

To create a multi-dimensional array object, use the following syntax.

```
>>> a = numpy.array([[1, 2, 3], [4, 5, 6]])
```

```
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([[1,2,3],[4,5,6]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
>>>
```

To change the data type of the array elements, mention the name of the data type along with the collection.

```
>>> a = numpy.array([1, 3, 5, 7], complex)
```

```
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([1, 3, 5, 7],dtype = complex)
>>> print(a)
[1.+0.j 3.+0.j 5.+0.j 7.+0.j]
>>>
```

Finding the dimensions of the Array

The **ndim** function can be used to find the dimensions of the array.

```
>>> import numpy as np
>>> arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])
>>> print(arr.ndim)
```

Finding the size of each array element

The itemsize function is used to get the size of each array item. It returns the number of bytes taken by each array element.

Example

```
import numpy as np
a = np.array([[1,2,3]])
print("Each item contains",a.itemsize,"bytes")
```

Output:

Each item contains 8 bytes.

Finding the data type of each array item

To check the data type of each array item, the dtype function is used. Consider the following example to check the data type of the array items.

Example

```
import numpy as np
a = np.array([[1,2,3]])
print("Each item is of the type",a.dtype)
```

Output:

Each item is of the type int64

Finding the shape and size of the array

To get the shape and size of the array, the size and shape function associated with the numpy array is used.

Example

```
import numpy as np
a = np.array([[1,2,3,4,5,6,7]])
print("Array Size:",a.size)
print("Shape:",a.shape)
```

Output:

Array Size: 7

Shape: (1, 7)

Reshaping the array objects

By the shape of the array, we mean the number of rows and columns of a multi-dimensional array. However, the numpy module provides us the way to reshape the array by changing the number of rows and columns of the multi-dimensional array.

The reshape() function associated with the ndarray object is used to reshape the array. It accepts the two parameters indicating the row and columns of the new shape of the array.

Let's reshape the array given in the following image.

2 X 3 3 X 2

Example

```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]])
print("printing the original array..")
print(a)
a=a.reshape(2,3)
print("printing the reshaped array..")
print(a)
```

Output:

printing the original array..
[[1 2]
 [3 4]
 [5 6]]
printing the reshaped array..
[[1 2 3]
 [4 5 6]]

Slicing in the Array

- Slicing in the NumPy array is the way to extract a range of elements from an array.
- Slicing in the array is performed in the same way as it is performed in the python list.

  #Consider the following example to print a particular element of the array.
  import numpy as np
  a = np.array([[1,2],[3,4],[5,6]])
  print(a[0,1])
  print(a[2,0])
  Output:
  2
  5

Note: The above program prints the 2nd element from the 0th index and 0th element from the 2nd index of the array.

Linspace

The linspace() function returns the evenly spaced values over the given interval.
The following example returns the 10 evenly separated values over the given interval 5-15

Example

```
import numpy as np
a=np.linspace(5,15,10) #prints 10 values which are evenly spaced over the given interval 5-15
print(a)
```

Output:

[5. 6.11111111 7.22222222 8.33333333 9.44444444 10.55555556
 11.66666667 12.77777778 13.88888889 15.]

Finding the maximum, minimum, and sum of the array elements

The NumPy provides the max(), min(), and sum() functions which are used to find the maximum, minimum, and sum of the array elements respectively.

Example

```
import numpy as np
a = np.array([1,2,3,10,15,4])
print("The array:",a)
print("The maximum element:",a.max())
print("The minimum element:",a.min())
print("The sum of the elements:",a.sum())
```

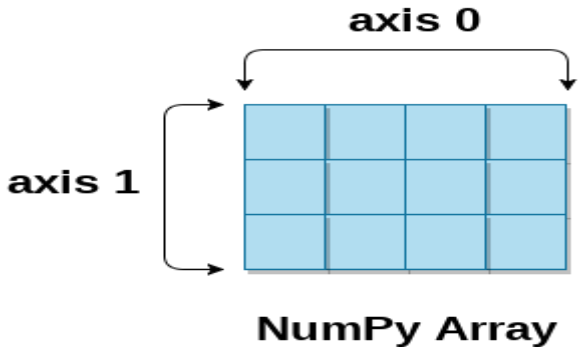
Output:

The array: [1 2 3 10 15 4]

The maximum element: 15
The minimum element: 1
The sum of the elements: 35

NumPy Array Axis

A NumPy multi-dimensional array is represented by the axis where axis-0 represents the columns and axis-1 represents the rows. We can mention the axis to perform row-level or column-level calculations like the addition of row or column elements.



To calculate the maximum element among each column, the minimum element among each row, and the addition of all the row elements,

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
print("The array:",a)
print("The maximum elements of columns:",a.max(axis = 0))
print("The minimum element of rows",a.min(axis = 1))
print("The sum of all rows",a.sum(axis = 1))
```

Output:

```
Arrays vertically concatenated
 [[ 1  2 30]
 [10 15  4]
 [ 1  2  3]
 [12 19 29]]
Arrays horizontally concatenated
 [[ 1  2 30  1  2  3]
 [10 15  4 12 19 29]]
```

3Q) NumPy Datatypes

Ans:

The NumPy provides a higher range of numeric data types than that provided by the Python. A list of numeric data types is given in the following table.

Data type	Description
bool_	It represents the boolean value indicating true or false. It is stored as a byte.

int_	It is the default type of integer. It is identical to long type in C that contains 64 bit or 32-bit integer.
Intc	It is similar to the C integer (c int) as it represents 32 or 64-bit int.
Intp	It represents the integers which are used for indexing.
int8	It is the 8-bit integer identical to a byte. The range of the value is -128 to 127.
int16	It is the 2-byte (16-bit) integer. The range is -32768 to 32767.
int32	It is the 4-byte (32-bit) integer. The range is -2147483648 to 2147483647.
int64	It is the 8-byte (64-bit) integer. The range is -9223372036854775808 to 9223372036854775807.
uint8	It is the 1-byte (8-bit) unsigned integer.
uint16	It is the 2-byte (16-bit) unsigned integer.
uint32	It is the 4-byte (32-bit) unsigned integer.
uint64	It is the 8 bytes (64-bit) unsigned integer.
float_	It is identical to float64.
float16	It is the half-precision float. 5 bits are reserved for the exponent. 10 bits are reserved for mantissa, and 1 bit is reserved for the sign.
float32	It is a single precision float. 8 bits are reserved for the exponent, 23 bits are reserved for mantissa, and 1 bit is reserved for the sign.
float64	It is the double precision float. 11 bits are reserved for the exponent, 52 bits are reserved for mantissa, 1 bit is used for the sign.
complex_	It is identical to complex128.
complex64	It is used to represent the complex number where real and imaginary part shares 32 bits each.
complex128	It is used to represent the complex number where real and imaginary part shares 64 bits each.

4Q) What is NumPy dtype?

Ans:

All the items of a numpy array are data type objects also known as numpy dtypes. A data type object implements the fixed size of memory corresponding to an array.
We can create a dtype object by using the following syntax.

numpy.dtype(object, align, copy)

The constructor accepts the following object.
Object: It represents the object which is to be converted to the data type.
Align: It can be set to any boolean value. If true, then it adds extra padding to make it equivalent to a C struct.
Copy: It creates another copy of the dtype object.

Example 1

```
import numpy as np
d = np.dtype(np.int32)
print(d)
```

Output:

```
int32
```

Example 2

```
import numpy as np
d = np.int32(i4)
print(d)
```

Output:

```
int32
```

Creating a Structured data type

We can create a map-like (dictionary) data type which contains the mapping between the values. For example, it can contain the mapping between employees and salaries or the students and the age, etc.

Example 1

```
import numpy as np
d = np.dtype([('salary',np.float)])
print(d)
```

Output:

```
[('salary', '
```

Example 2

```
import numpy as np
d=np.dtype([('salary',np.float)])
arr = np.array([(10000.12,),(20000.50,)],dtype=d)
print(arr['salary'])
```

Output:

```
[(10000.12,) (20000.5 ,)]
```

5Q) Numpy Array Creation
Ans:
The ndarray object can be constructed by using the following routines.

1) Numpy.empty

As the name specifies, The empty routine is used to create an uninitialized array of specified shape and data type.

Syn: numpy.empty(shape, dtype = float, order = 'C')

It accepts the following parameters.

- Shape: The desired shape of the specified array.
- dtype: The data type of the array items. The default is the float.
- Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.empty((3,2), dtype = int)
print(arr)
```

Output:

```
[[ 140482883954664          36917984]
 [ 140482883954648     140482883954648]
 [6497921830368665435  17202647269960427 2]]
```

2) NumPy.Zeros

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 0.

Syn: numpy.zeros(shape, dtype = float, order = 'C')

It accepts the following parameters.

- Shape: The desired shape of the specified array.
- dtype: The data type of the array items. The default is the float.
- Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.zeros((3,2), dtype = int)
print(arr)
```

Output:

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

3) NumPy.ones

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 1.

Syn: numpy.ones(shape, dtype = none, order = 'C')

It accepts the following parameters.

- Shape: The desired shape of the specified array.
- dtype: The data type of the array items.
- Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.ones((3,2), dtype = int)
print(arr)
```

Output:

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

UNIT – IV

1Q) Python Pandas Introduction

Ans:

- Pandas is defined as an open-source library that provides high-performance data manipulation in Python. The name of Pandas is derived from the word **Panel Data**, which means **an Econometrics from Multidimensional data**. It is used for data analysis in Python and developed by **Wes McKinney** in **2008**.
- Data analysis requires lots of processing, such as **restructuring, cleaning** or **merging**, etc. There are different tools are available for fast data processing, such as **Numpy, Scipy, Cython**, and **Panda**. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.
- Pandas is built on top of the **Numpy** package, means **Numpy** is required for operating the Pandas.
- Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., **load, manipulate, prepare, model, and analyze**.

Key Features of Pandas

1. It has a fast and efficient DataFrame object with the default and customized indexing.
2. Used for reshaping and pivoting of the data sets.
3. Group by data for aggregations and transformations.

4. It is used for data alignment and integration of the missing data.
5. Provide the functionality of Time Series.
6. Process a variety of data sets in different formats like matrix data, tabular heterogeneous, time series.
7. Handle multiple operations of the data sets such as subsetting, slicing, filtering, groupBy, re-ordering, and re-shaping.
8. It integrates with the other libraries such as SciPy, and scikit-learn.
9. Provides fast performance, and If you want to speed it, even more, you can use the **Cython**.

2Q) Python Pandas Data Structure

Ans: The Pandas provides two data structures for processing the data, i.e., **Series** and **DataFrame**

1) Series

It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the **index**. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:

Data: It can be any list, dictionary, or scalar value.

Creating Series from Array:

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

```
import pandas as pd
import numpy as np
info = np.array(['P','a','n','d','a','s'])
a = pd.Series(info)
print(a)
```

Output

```
0  P
1  a
2  n
3  d
4  a
5  s
dtype: object
```

2) DataFrame

- It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns).
- DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index.
- It consists of the following properties:
- The columns can be heterogeneous types like int, bool, and so on.

- It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

#Create a DataFrame using List:

```
import pandas as pd
x = ['Python', 'Pandas']
df = pd.DataFrame(x)
print(df)
```

Output

```
    0
0   Python
1   Pandas
```

3Q) Python Pandas Series

Ans:

- The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types.
- We can easily convert the list, tuple, and dictionary into series using "series' method.
- The row labels of series are called the index.
- A Series cannot contain multiple columns. It has the following parameter:
  data: It can be any list, dictionary, or scalar value.
  index: The value of the index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default np.arrange(n) will be used.
  dtype: It refers to the data type of series.
  copy: It is used for copying the data.

Creating a Series:

We can create a Series in two ways:

1. Create an empty Series
2. Create a Series using inputs.

Create an Empty Series:

We can easily create an empty series in Pandas which means it will not have any value.

Syn: <series object> = pandas.Series()

Eg:

```
import pandas as pd
x = pd.Series()
print (x)
```

Output

```
Series([], dtype: float64)
```

Creating a Series using inputs:

We can create Series by using various inputs:

- Array
- Dict
- Scalar value

Creating Series from Array:

- Before creating a Series, firstly, we have to import the numpy module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.
- If we do not pass an index, then by default index of range(n) is being passed where n defines the length of an array, i.e., [0,1,2,....range(len(array))-1].

Example

```
import pandas as pd
import numpy as np
info = np.array(['P','a','n','d','a','s'])
a = pd.Series(info)
print(a)
```

Output

```
0  P
1  a
2  n
3  d
4  a
5  s
dtype: object
```

Create a Series from dict

- We can also create a Series from dict. If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index.
- If index is passed, then values correspond to a particular label in the index will be extracted from the dictionary.

```
#import the pandas library
import pandas as pd
import numpy as np
info = {'x' : 0., 'y' : 1., 'z' : 2.}
a = pd.Series(info)
print (a)
```

Output

```
x  0.0
y  1.0
z  2.0
dtype: float64
```

Create a Series using Scalar:

- If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

```
#import pandas library
```

```
import pandas as pd
import numpy as np
x = pd.Series(4, index=[0, 1, 2, 3])
print (x)
```

Output

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

Accessing data from series with Position:

- Once you create the Series type object, you can access its indexes, data, and even individual elements.
- The data in the Series can be accessed similar to that in the ndarray.

```
import pandas as pd
x = pd.Series([1,2,3],index = ['a','b','c'])
#retrieve the first element
print (x[0])
```

Output

```
1
```

Series object attributes

The Series attribute is defined as any information related to the Series object such as size, datatype. etc. Below are some of the attributes that you can use to get the information about the Series object:

Attributes	Description
Series.index	Defines the index of the Series.
Series.shape	It returns a tuple of shape of the data.
Series.dtype	It returns the data type of the data.
Series.size	It returns the size of the data.
Series.empty	It returns True if Series object is empty, otherwise returns false.
Series.hasnans	It returns True if there are any NaN values, otherwise returns false.
Series.nbytes	It returns the number of bytes in the data.

Series.ndim	It returns the number of dimensions in the data.
Series.itemsize	It returns the size of the datatype of item.

Series Functions

There are some functions used in Series which are as follows:

Functions	Description
Pandas Series.map()	Map the values from two series that have a common column.
Pandas Series.std()	Calculate the standard deviation of the given set of numbers, DataFrame, column, and rows.
Pandas Series.to_frame()	Convert the series object to the dataframe.
Pandas Series.value_counts()	Returns a Series that contain counts of unique values.

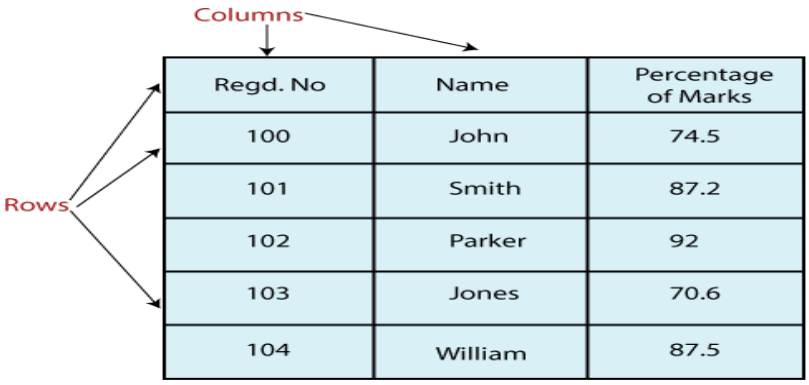
4Q) Python Pandas DataFrame

Ans:

- Pandas DataFrame is a widely used data structure which works with a two-dimensional array with labeled axes (rows and columns).
- DataFrame is defined as a standard way to store data that has two different indexes, i.e., row index and column index.
- It consists of the following properties:
    - The columns can be heterogeneous types like int, bool, and so on.
    - It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

Parameter & Description:

- ⇨ **data:** It consists of different forms like ndarray, series, map, constants, lists, array.
- ⇨ **index:** The Default np.arrange(n) index is used for the row labels if no index is passed.
- ⇨ **columns:** The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.
- ⇨ dtype: It refers to the data type of each column.
- ⇨ copy(): It is used for copying the data.



Regd. No	Name	Percentage of Marks
100	John	74.5
101	Smith	87.2
102	Parker	92
103	Jones	70.6
104	William	87.5

Create a DataFrame

We can create a DataFrame using following ways:

- dict
- Lists
- Numpy ndarrrays
- Series

Create an empty DataFrame

The below code shows how to create an empty DataFrame in Pandas:

```
# importing the pandas library
import pandas as pd
df = pd.DataFrame()
print (df)
```

Output

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame using List:

We can easily create a DataFrame in Pandas using list.

```
# importing the pandas library
import pandas as pd
# a list of strings
x = ['Python', 'Pandas']
df = pd.DataFrame(x)
print(df)
```

Output

```
     0
0  Python
1  Pandas
```

Create a DataFrame from Dict of ndarrays/ Lists

```
# importing the pandas library
import pandas as pd
info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}
df = pd.DataFrame(info)
print (df)
```

Output

```
        ID     Department
```

```
0       101     B.Sc
1       102     B.Tech
2       103     M.Tech
```

Create a DataFrame from Dict of Series:

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1)
```

Output

```
     one       two
a     1.0        1
b     2.0        2
c     3.0        3
d     4.0        4
e     5.0        5
f     6.0        6
g     NaN        7
h     NaN        8
```

Column Selection

We can select any column from the DataFrame. Here is the code that demonstrates how to select a column from the DataFrame.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}
d1 = pd.DataFrame(info)
print (d1 ['one'])
```

Output

```
a     1.0
b     2.0
c     3.0
d     4.0
e     5.0
f     6.0
g     NaN
h     NaN
Name: one, dtype: float64
```

Column Addition

We can also add any new column to an existing DataFrame. The below code demonstrates how to add any new column to an existing DataFrame:

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
  'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
# Add a new column to an existing DataFrame object
print ("Add new column by passing series")
df['three']=pd.Series([20,40,60],index=['a','b','c'])
print (df)
print ("Add new column using existing DataFrame columns")
df['four']=df['one']+df['three']
print (df)
```

Output

Add new column by passing series

	one	two	three
a	1.0	1	20.0
b	2.0	2	40.0
c	3.0	3	60.0
d	4.0	4	NaN
e	5.0	5	NaN
f	NaN	6	NaN

Add new column using existing DataFrame columns

	one	two	three	four
a	1.0	1	20.0	21.0
b	2.0	2	40.0	42.0
c	3.0	3	60.0	63.0
d	4.0	4	NaN	NaN
e	5.0	5	NaN	NaN
f	NaN	6	NaN	NaN

Column Deletion:

We can also delete any column from the existing DataFrame. This code helps to demonstrate how the column can be deleted from an existing DataFrame:

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2], index= ['a', 'b']),
  'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}
df = pd.DataFrame(info)
print ("The DataFrame:")
print (df)

# using del function
```

```
print ("Delete the first column:")
del df['one']
print (df)
# using pop function
print ("Delete the another column:")
df.pop('two')
print (df)
```

Output

```
The DataFrame:
    one  two
a   1.0   1
b   2.0   2
c   NaN   3

Delete the first column:
   two
a   1
b   2
c   3

Delete the another column:
Empty DataFrame
Columns: []
Index: [a, b, c]
```

Row Selection, Addition, and Deletion

Row Selection:

We can easily select, add, or delete any row at anytime. First of all, we will understand the row selection. Let's see how we can select a row using different ways that are as follows:

Selection by Label:

We can select any row by passing the row label to a loc function.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)
print (df.loc['b'])
```

Output

```
one   2.0
two   2.0
Name: b, dtype: float64
```

Selection by integer location:

The rows can also be selected by passing the integer location to an iloc function.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df.iloc[3])
```

Output

```
one   4.0
two   4.0
Name: d, dtype: float64
```

Slice Rows

It is another method to select multiple rows using ':' operator.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df[2:5])
```

Output

```
    one   two
c   3.0   3
d    4.0   4
e   5.0   5
```

Addition of rows:

We can easily add new rows to the DataFrame using append function. It adds the new rows at the end.

```
# importing the pandas library
import pandas as pd
d = pd.DataFrame([[7, 8], [9, 10]], columns = ['x','y'])
d2 = pd.DataFrame([[11, 12], [13, 14]], columns = ['x','y'])
d = d.append(d2)
print (d)
```

Output

```
    x    y
0   7    8
1   9    10
0   11   12
1   13   14
```

Deletion of rows:

We can delete or drop any rows from a DataFrame using the index label. If in case, the label is duplicate then multiple rows will be deleted.

```
# importing the pandas library
import pandas as pd
a_info = pd.DataFrame([[4, 5], [6, 7]], columns = ['x','y'])
```

```
b_info = pd.DataFrame([[8, 9], [10, 11]], columns = ['x','y'])
a_info = a_info.append(b_info)
# Drop rows with label 0
a_info = a_info.drop(0)
```

Output

```
x   y
1   6   7
1   10  11
```

DataFrame Functions

There are lots of functions used in DataFrame which are as follows:

Functions	Description
Pandas DataFrame.append()	Add the rows of other dataframe to the end of the given dataframe.
Pandas DataFrame.apply()	Allows the user to pass a function and apply it to every single value of the Pandas series.
Pandas DataFrame.assign()	Add new column into a dataframe.
Pandas DataFrame.astype()	Cast the Pandas object to a specified dtype.astype() function.
Pandas DataFrame.concat()	Perform concatenation operation along an axis in the DataFrame.
Pandas DataFrame.count()	Count the number of non-NA cells for each column or row.
Pandas DataFrame.describe()	Calculate some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame.
Pandas DataFrame.drop_duplicates()	Remove duplicate values from the DataFrame.
Pandas DataFrame.groupby()	Split the data into various groups.
Pandas DataFrame.head()	Returns the first n rows for the object based on position.
Pandas DataFrame.hist()	Divide the values within a numerical variable into "bins".
Pandas DataFrame.iterrows()	Iterate over the rows as (index, series) pairs.

Pandas DataFrame.mean()	Return the mean of the values for the requested axis.
Pandas DataFrame.melt()	Unpivots the DataFrame from a wide format to a long format.
Pandas DataFrame.merge()	Merge the two datasets together into one.
Pandas DataFrame.pivot_table()	Aggregate data with calculations such as Sum, Count, Average, Max, and Min.
Pandas DataFrame.query()	Filter the dataframe.
Pandas DataFrame.sample()	Select the rows and columns from the dataframe randomly.
Pandas DataFrame.shift()	Shift column or subtract the column value with the previous row value from the dataframe.
Pandas DataFrame.sort()	Sort the dataframe.
Pandas DataFrame.sum()	Return the sum of the values for the requested axis by the user.
Pandas DataFrame.to_excel()	Export the dataframe to the excel file.
Pandas DataFrame.transpose()	Transpose the index and columns of the dataframe.
Pandas DataFrame.where()	Check the dataframe for one or more conditions.

5Q) Python Pandas - Descriptive Statistics

Ans:

- A large number of methods collectively compute descriptive statistics and other related operations on DataFrame.
- Most of these are aggregations like sum(), mean(), but some of them, like sumsum(), produce an object of the same size.
- Generally speaking, these methods take an axis argument, just like ndarray.{sum, std, ...}, but the axis can be specified by name or integer

DataFrame − "index" (axis=0, default), "columns" (axis=1)

Example

```
import pandas as pd
import numpy as np
#Create a Dictionary of series
```

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65]) }
```

#Create a DataFrame

```
df = pd.DataFrame(d)
print df
```

O/P:

	Age	Name	Rating
0	25	Tom	4.23
1	26	James	3.24
2	25	Ricky	3.98
3	23	Vin	2.56
4	30	Steve	3.20
5	29	Smith	4.60
6	23	Jack	3.80
7	34	Lee	3.78
8	40	David	2.98
9	30	Gasper	4.80
10	51	Betina	4.10
11	46	Andres	3.65

sum()

Returns the sum of the values for the requested axis. By default, axis is index (axis=0).

Eg:

```
import pandas as pd
import numpy as np
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65]) }
#Create a DataFrame
df = pd.DataFrame(d)
print df.sum()
```

o/p:

```
Age                               382
Name    TomJamesRickyVinSteveSmithJackLeeDavidGasperBe...
Rating                          44.92
dtype: object
```

Each individual column is added individually (Strings are appended).

axis=1

```
import pandas as pd
import numpy as np
#Create a Dictionary of series
```

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])  }
```
#Create a DataFrame
```
df = pd.DataFrame(d)
print df.sum(1)
```
o/p:
```
0   29.23
1   29.24
2   28.98
3   25.56
4   33.20
5   33.60
6   26.80
7   37.78
8   42.98
9   34.80
10  55.10
11  49.65
dtype: float64
```

mean(): Returns the average value

Eg:
```
import pandas as pd
import numpy as np
```
#Create a Dictionary of series
```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}
```
#Create a DataFrame
```
df = pd.DataFrame(d)
print df.mean()
```
o/p:
```
Age      31.833333
Rating    3.743333
dtype: float64
```

std(): Returns the Bressel standard deviation of the numerical columns.

Eg:
```
import pandas as pd
import numpy as np
```
#Create a Dictionary of series
```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
```

```
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65]) }
```

#Create a DataFrame

```
df = pd.DataFrame(d)
print df.std()
```

o/p:

```
Age      9.232682
Rating   0.661628
dtype: float64
```

Functions & Description

The functions under Descriptive Statistics in Python Pandas. The following table list down the important functions −

S.No.	Function	Description
1	count()	Number of non-null observations
2	sum()	Sum of values
3	mean()	Mean of Values
4	median()	Median of Values
5	mode()	Mode of values
6	std()	Standard Deviation of the Values
7	min()	Minimum Value
8	max()	Maximum Value
9	abs()	Absolute Value
10	prod()	Product of Values
11	cumsum()	Cumulative Sum
12	cumprod()	Cumulative Product

7Q) **Summarizing Data**

Ans: The describe() function computes a summary of statistics pertaining to the DataFrame columns.

Eg:

```
import pandas as pd
import numpy as np
```

#Create a Dictionary of series

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
```

```
  'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65]) }
#Create a DataFrame
df = pd.DataFrame(d)
print df.describe()
```

o/p:

	Age	Rating
count	12.000000	12.000000
mean	31.833333	3.743333
std	9.232682	0.661628
min	23.000000	2.560000
25%	25.000000	3.230000
50%	29.500000	3.790000
75%	35.500000	4.132500
max	51.000000	4.800000

This function gives the mean, std and IQR values. And, function excludes the character columns and given summary about numeric columns. 'include' is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.

object − Summarizes String columns
number − Summarizes Numeric columns
all − Summarizes all columns together (Should not pass it as a list value)

Now, use the following statement in the program and check the output −

```
import pandas as pd
import numpy as np
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
  'Lee','David','Gasper','Betina','Andres']),
  'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
  'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}
#Create a DataFrame
df = pd.DataFrame(d)
print df.describe(include=['object'])
```

o/p:

	Name
count	12
unique	12
top	Ricky
freq	1

Now, use the following statement and check the output −

```
import pandas as pd
import numpy as np
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
  'Lee','David','Gasper','Betina','Andres']),
```

```
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}
#Create a DataFrame
df = pd.DataFrame(d)
print df. describe(include='all')
```

o/p:

	Age	Name	Rating
count	12.000000	12	12.000000
unique	NaN	12	NaN
top	NaN	Ricky	NaN
freq	NaN	1	NaN
mean	31.833333	NaN	3.743333
std	9.232682	NaN	0.661628
min	23.000000	NaN	2.560000
25%	25.000000	NaN	3.230000
50%	29.500000	NaN	3.790000
75%	35.500000	NaN	4.132500
max	51.000000	NaN	4.800000

UNIT – V

1Q) What is data cleaning?

Ans:

- *Data cleaning* is the process of changing or eliminating garbage, incorrect, duplicate, corrupted, or incomplete data in a dataset.
- Data cleansing, data cleansing, or data scrub is that the initiative among the general data preparation process.
- Data cleaning plays an important part in developing reliable answers and within the analytical process and is observed to be a basic feature of the info science basics.
- The motive of data cleaning services is to construct uniform and standardized data sets that enable data analytical tools and business intelligence easy access and perceive accurate data for each problem.

2Q) Write about Data Cleaning Cycle?

Ans:

- It is the method of analyzing, distinguishing, and correcting untidy, raw data.
- Data cleaning involves filling in missing values, distinguish and fix errors present in the dataset. Whereas the techniques used for data cleaning might vary in step with different types of datasets, the following are standard steps to map out data cleaning:

Data cleaning step by step

- To start working with Pandas we need to import it. We are using Google Colab as IDE, so we will import Pandas in Google Colab.

```
#importing module
import pandas as pd
```

Import Dataset

- To import the dataset we use the read_csv() function of pandas and store it in the DataFrame named as data. As the dataset is in tabular format, when working with tabular data in Pandas it will be automatically converted in a DataFrame.
- DataFrame is a two-dimensional, mutable data structure in Python. It is a combination of rows and columns like an excel sheet.

- **The head() function** is a built-in function in pandas for the dataframe used to display the rows of the dataset. We can specify the number of rows by giving the number within the parenthesis. By default, it displays the first five rows of the dataset. If we want to see the last five rows of the dataset we use the tail()function of the dataframe like this:

#displayinf last five rows of dataset
data.tail()

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

Merge Dataset

- Merging the dataset is the process of combining two datasets in one, and line up rows based on some particular or common property for data analysis. We can do this by using the merge() function of the dataframe.

Syn:

DataFrame_name.merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Rebuild Missing Data

To find and fill the missing data in the dataset we will use another function. There are 4 ways to find the null values if present in the dataset.

1. **Using isnull() function:**

data.isnull()

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	False	False
...	...	...	...	...	...	...
145	False	False	False	False	False	False
146	False	False	False	False	False	False
147	False	False	False	False	False	False
148	False	False	False	False	False	False
149	False	False	False	False	False	False

150 rows × 6 columns

This function provides the boolean value for the complete dataset to know if any null value is present or not.

2. **Using isna() function:**

data.isna()

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	False	False
...	...	...	...	...	...	...
145	False	False	False	False	False	False
146	False	False	False	False	False	False
147	False	False	False	False	False	False
148	False	False	False	False	False	False
149	False	False	False	False	False	False

150 rows × 6 columns

3. **Using isna().any()**

data.isna().any()

```
Id               False
SepalLengthCm    False
SepalWidthCm     False
PetalLengthCm    False
PetalWidthCm     False
Species          False
dtype: bool
```

This function also gives a boolean value if any null value is present or not, but it gives results column-wise, not in tabular format.

4. **Using isna(). sum()**

data.isna().sum()

```
Id               0
SepalLengthCm    0
SepalWidthCm     0
PetalLengthCm    0
PetalWidthCm     0
Species          0
dtype: int64
```

This function gives the sum of the null values preset in the dataset column-wise.

Using isna().any().sum()

data.isna().any().sum()

```
data.isna().any().sum()
```

```
0
```

This function gives output in a single value if any null is present or not.

There are no null values present in our dataset. But if there are any null values preset we can fill those places with any other value using the fillna() function of DataFrame.Following is the syntax of fillna() function:

Syn:

DataFrame_name.fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

This function will fill NA/NaN or 0 values in place of null spaces.

Standardization and Normalization

Data Standardization and Normalization is a common practice in machine learning.

Standardization is another scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling.

De-Duplicate

- De-Duplicate means remove all duplicate values. There is no need for duplicate values in data analysis. These values only affect the accuracy and efficiency of the analysis result.
- To find duplicate values in the dataset we will use a simple dataframe function i.e. duplicated().

data.duplicated()

```
0      False
1      False
2      False
3      False
4      False
       ...
145    False
146    False
147    False
148    False
149    False
Length: 150, dtype: bool
```

- This function also provides bool values for duplicate values in the dataset. As we can see that dataset doesn't contain any duplicate values.
- If a dataset contains duplicate values it can be removed using the drop_duplicates() function.

Syn:

DataFrame_name.drop_duplicates(*subset=None*, *keep='first'*, *inplace=False*, *ignore_index=False*)

Verify and Enrich

After removing null, duplicate, and incorrect values, we should verify the dataset and validate its accuracy. In this step, we have to check that the data cleaned so far is making any sense. If the data is incomplete we have to enrich the data again by data gathering activities like approaching the clients again, re-interviewing people, etc. Completeness is a little more challenging to achieve accuracy or quality in the dataset.

Export Dataset

This is the last step of the data cleaning process. After performing all the above operations, the data is transformed into clean the dataset and it is ready to export for the next process in Data Science or Data Analysis.

3Q) Data Transformation

Ans:

Most real-world dataset is dirty. Before analyzing the dataset, you need to transform this dataset. This process is called data transformation.

1. Finding the duplicate values
2. Mapping
3. Replacing
4. Renaming
5. Cutting
6. Finding the specific values
7. Selecting
8. Creating dummy variables

First of all, let me import Pandas.

```
In [1]:  1  import pandas as pd
```

1. Finding the duplicate values

⇨ **Let's create a dataset.**

```
In [2]:  1  data=pd.DataFrame({"a":["one","two"]*3,
         2                     "b":[1,1,2,3,2,3]})
         3  data
```

Out[2]:

	a	b
0	one	1
1	two	1
2	one	2
3	two	3
4	one	2
5	two	3

⇨ To see whether the row is repeated or not, you can use the duplicated method.

```
In [3]:  1  data.duplicated()
```

```
Out[3]: 0    False
        1    False
        2    False
        3    False
        4     True
        5     True
        dtype: bool
```

⇨ To remove duplicate rows, you can use the drop_duplicates method which returns a data frame.

```
In [4]: data.drop_duplicates()
```

Out[4]:

	a	b
0	one	1
1	two	1
2	one	2
3	two	3

⇨ To find duplicates on specific columns, you can use the duplicated method. To show this, let me add a column to the dataset.

```
In [5]: data["c"]=range(6)
        data
```

Out[5]:

	a	b	c
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	2	4
5	two	3	5

For example, let's find the duplicate values according to column c.

```
In [6]: data.duplicated(["a","b"],keep="last")
```

```
Out[6]: 0    False
        1    False
        2     True
        3     True
        4    False
        5    False
        dtype: bool
```

2. Mapping

You can transfer the values of data to the dataset with a function or a mapping. To show that, let me create a dataset.

```
In [7]: df=pd.DataFrame({"names":["Tim","tom","Sam",
                                  "kate","Kim"],
                         "scores":[60,50,70,80,40]})
        df
```

Out[7]:

	names	scores
0	Tim	60
1	tom	50
2	Sam	70
3	kate	80
4	Kim	40

Let's add a column to show each student's class.

```
In [8]: classes={"Tim":"A","Tom":"A","Sam":"B",
                "Kate":"B","Kim":"B"}
```

Let's convert the first character of the names to a capital (uppercase) letter so that the names are the same.

```
In [9]: n=df["names"].str.capitalize()
```

Now, let's add a branch variable to the df dataset using the map() method.

```
In [10]: df["branches"]=n.map(classes)
```

```
In [11]: df
```

Out[11]:

	names	scores	branches
0	Tim	60	A
1	tom	50	A
2	Sam	70	B
3	kate	80	B
4	Kim	40	B

3. Replacing

As you know, you can use the **fillna()** method for missing data. You can also the replace method for missing data. To show this, let me create a series named s.

```
In [12]:  1 s=pd.Series([80,70,90,60])
          2 s

Out[12]: 0    80
         1    70
         2    90
         3    60
         dtype: int64
```

I'm going to add missing data to this data. To do this, let me use Numpy. First, I'm going to import NumPy.

```
In [13]:  1 import numpy as np
```

Let's assign the missing data to 70 with the replace method.

```
In [14]:  1 s.replace(70,np.nan)

Out[14]: 0    80.0
         1     NaN
         2    90.0
         3    60.0
         dtype: float64
```

Using the replace method, you can assign different values to each value. For example, let's assign missing data instead of 70 and 0 instead of 60.

```
In [15]:  1 s.replace([70,60],[np.nan,0])

Out[15]: 0    80.0
         1     NaN
         2    90.0
         3     0.0
         dtype: float64
```

To replace, you can use the dictionary structure.

```
In [16]:  1 s.replace({90:100,60:0})

Out[16]: 0     80
         1     70
         2    100
         3      0
         dtype: int64
```

4. Renaming

You can rename axes in the dataset with a function or mapping. To show this, let's create a dataset.

```
In [17]:  1 df=pd.DataFrame(
          2       np.arange(12).reshape(3,4),
          3       index=[0,1,2],
          4       columns=["tim","tom","kim","sam"])
          5 df

Out[17]:
```

	tim	tom	kim	sam
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Now, let's create an s variable and pass it to this dataset.

```
In [18]:  1 s=pd.Series(["one","two","three"])
          2 df.index=df.index.map(s)
```

Let's take a look at the dataset.

```
In [19]:  1 df

Out[19]:
```

	tim	tom	kim	sam
one	0	1	2	3
two	4	5	6	7
three	8	9	10	11

You can capitalize the row and column names with the rename method,

```
In [20]: df.rename(index=str.title,columns=str.upper)
```

Out[20]:

	TIM	TOM	KIM	SAM
One	0	1	2	3
Two	4	5	6	7
Three	8	9	10	11

You can change the row or column names using the dictionary structure with the rename method.

```
In [21]: df.rename(index={"one":"ten"},
                   columns={"sam":"kate"},
                   inplace=True)
         df
```

Out[21]:

	tim	tom	kim	kate
ten	0	1	2	3
two	4	5	6	7
three	8	9	10	11

5. Cutting

To group data periodically, you can use the cut method. To show this, let's create a data.

```
In [22]: sc=[30,80,40,90,60,45,95,75,55,100,65,85]
```

Let's split this data into certain intervals. For this, let's create an x variable and specify the intervals.

```
In [23]: x=[20,40,60,80,100]
```

Now, let's determine the interval of each value in sc according to these intervals.

```
In [24]: y=pd.cut(sc,x)
         y
```

```
Out[24]: [(20, 40], (60, 80], (20, 40], (80, 100], (40, 6
         0], ..., (60, 80], (40, 60], (80, 100], (60, 8
         0], (80, 100]]
         Length: 12
         Categories (4, interval[int64]): [(20, 40] < (4
         0, 60] < (60, 80] < (80, 100]]
```

You can use the codes attribute to see the categorical code for each value.

```
In [25]: y.codes
```

```
Out[25]: array([0, 2, 0, 3, 1, 1, 3, 2, 1, 3, 2, 3], dtyp
         e=int8)
```

You can use the categories attribute to see intervals.

```
In [26]: y.categories
```

```
Out[26]: IntervalIndex([(20, 40], (40, 60], (60, 80], (8
         0, 100]],
                       closed='right',
                       dtype='interval[int64]')
```

Let's see the frequency of the categories.

```
In [27]:  1  pd.value_counts(y)
```

```
Out[27]: (80, 100]    4
         (60, 80]     3
         (40, 60]     3
         (20, 40]     2
         dtype: int64
```

Note that the intervals start with normal brackets and end with square brackets. To change this, you can use the option right = False.

```
In [28]:  1  y=pd.cut(sc,x,right=False)
          2  y
```

```
Out[28]: [[20, 40), [80, 100), [40, 60), [80, 100), [60,
         80), ..., [60.0, 80.0), [40.0, 60.0), NaN, [60.
         0, 80.0), [80.0, 100.0)]
         Length: 12
         Categories (4, interval[int64]): [[20, 40) < [4
         0, 60) < [60, 80) < [80, 100)]
```

You can name the intervals. Let me show this.

```
In [29]:  1  nm=["low", "medium", "high", "very high"]
          2  pd.cut(sc,x,labels=nm)
```

```
Out[29]: ['low', 'high', 'low', 'very high', 'medium',
         ..., 'high', 'medium', 'very high', 'high', 'ver
         y high']
         Length: 12
         Categories (4, object): ['low' < 'medium' < 'hig
         h' < 'very high']
```

You can categorize the data by entering the number of intervals.

```
In [30]:  1  pd.cut(sc,10)
```

```
Out[30]: [(29.93, 37.0], (79.0, 86.0], (37.0, 44.0], (86.
         0, 93.0], (58.0, 65.0], ..., (72.0, 79.0], (51.
         0, 58.0], (93.0, 100.0], (58.0, 65.0], (79.0, 8
         6.0]]
         Length: 12
         Categories (10, interval[float64]): [(29.93, 37.
         0] < (37.0, 44.0] < (44.0, 51.0] < (51.0, 58.0]
         ... (72.0, 79.0] < (79.0, 86.0] < (86.0, 93.0] <
         (93.0, 100.0]]
```

You can use the qcut method to divide the data into quarter intervals. To show this, first, let's generate data from the normal distribution and then use the qcut method.

```
In [31]:  1  data=np.random.randn(100)
          2  c=pd.qcut(data,4)
          3  c
```

```
Out[31]: [(-0.135, 0.631], (0.631, 2.286], (-0.629, -0.13
         5], (0.631, 2.286], (-2.186, -0.629], ..., (-0.6
         29, -0.135], (0.631, 2.286], (0.631, 2.286], (-
         0.135, 0.631], (-0.629, -0.135]]
         Length: 100
         Categories (4, interval[float64]): [(-2.186, -0.
         629] < (-0.629, -0.135] < (-0.135, 0.631] < (0.6
         31, 2.286]]
```

Now, let's take a look at how many values fall in each quarter interval.

```
In [32]: pd.value_counts(c)
```

```
Out[32]: (0.631, 2.286]      25
         (-0.135, 0.631]     25
         (-0.629, -0.135]    25
         (-2.186, -0.629]    25
         dtype: int64
```

6. Finding the specific values in a dataset

You can find specific values in the dataset. To show this, let's generate data from the normal distribution.

```
In [33]: data=pd.DataFrame(np.random.randn(1000,4))
         data.head()
```

Out[33]:

	0	1	2	3
0	0.661162	-1.315550	0.138893	-2.186859
1	-0.422096	0.587658	-0.478577	-0.285737
2	-0.283092	-0.021623	1.194335	-0.197599
3	-1.545286	-0.219977	0.353704	0.424970
4	-0.196521	3.491917	0.016217	-0.464119

You can use the describe method to see summary statistics. Let me show this.

```
In [34]: data.describe()
```

Out[34]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.048690	0.030168	-0.050958	-0.004881
std	0.991747	0.996656	0.992911	1.021762
min	-2.999425	-3.514219	-2.864107	-3.186480
25%	-0.723203	-0.622904	-0.726046	-0.752119
50%	-0.058605	0.027484	-0.036939	0.023319
75%	0.560961	0.654890	0.605512	0.739495
max	3.952527	3.491917	3.074154	3.002287

For example, let's find values whose absolute value exceeds 3 in a column with 1 index. To show this, I'm going to assign the column with the 1st index of the data to the col variable.

```
In [35]: col=data[1]
```

Now, let's find the values whose absolute value is greater than 3.

```
In [36]: col[np.abs(col)>3]
```

```
Out[36]: 4       3.491917
         117     3.098920
         311    -3.067095
         903    -3.514219
         Name: 1, dtype: float64
```

To find rows with at least one absolute value exceeding 3 in the entire data set, you can use the any method.

```
In [37]: data[(np.abs(data)>3).any(1)]
```

Out[37]:

	0	1	2	3
4	-0.196521	3.491917	0.016217	-0.464119
117	-1.360727	3.098920	-0.902404	-1.874759
311	-1.408380	-3.067095	-0.034621	-1.377992
492	-0.508447	-0.264550	-2.246989	-3.096799
510	3.952527	-0.307437	1.160524	1.022866
547	-0.146700	-0.594890	3.074154	-0.198825
642	3.116662	-0.466845	-0.543486	0.038652
867	0.222411	-0.131135	0.618451	3.002287
903	-0.917298	-3.514219	-2.500715	0.259489
956	0.115918	1.314808	-0.220663	-3.186480

You can use the sign method to see positive or negative values.

```
In [38]:  1  np.sign(data).head()
```

Out[38]:

	0	1	2	3
0	1.0	-1.0	1.0	-1.0
1	-1.0	1.0	-1.0	-1.0
2	-1.0	-1.0	1.0	-1.0
3	-1.0	-1.0	1.0	1.0
4	-1.0	1.0	1.0	-1.0

7. Selecting

You can use the permutation to sort randomly. To show this, I'm going to create a data frame.

```
In [39]:  1  data=pd.DataFrame(
          2       np.arange(12).reshape(4,3))
          3  data
```

Out[39]:

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

Now, let's use the permutation.

```
In [40]:  1  rw=np.random.permutation(4)
          2  rw
```

```
Out[40]: array([2, 3, 0, 1])
```

To change the order of the row in the data, let's use the take method.

```
In [41]:  1  data.take(rw)
```

Out[41]:

	0	1	2
2	6	7	8
3	9	10	11
0	0	1	2
1	3	4	5

To randomly select a row, you can use the sample method.

```
In [42]:  1  data.sample()
```

Out[42]:

	0	1	2
3	9	10	11

You can select random rows.

```
In [43]:  1  data.sample(n=2)
```

Out[43]:

	0	1	2
0	0	1	2
2	6	7	8

8. Converts categorical data into dummy variables

You can use the get_dummy method to convert a categorical variable into a "dummy" or "indicator". To show this, let me create a dataset.

```
In [44]:
data=pd.DataFrame(
    {"letter":["c","b","a","b","b","a"],
                    "number":range(6)})
data
```

Out[44]:

	letter	number
0	c	0
1	b	1
2	a	2
3	b	3
4	b	4
5	a	5

Let's convert the letter column into the dummy variables with the get_dummies method.

```
In [45]:
pd.get_dummies(data["letter"])
```

Out[45]:

	a	b	c
0	0	0	1
1	0	1	0
2	1	0	0
3	0	1	0
4	0	1	0
5	1	0	0

You can use the dummy variable to find the interval in which values fall in a dataset. To show this, let's generate data from the normal distribution.

```
In [46]:
data=np.random.randn(10)
data
```

```
Out[46]: array([-0.47399816, -0.4700948 , -0.10090975, -
0.49105714,  0.85748633,
        -2.17384891, -1.89062041,  1.15155524, -
1.12043372,  0.82935199])
```

Now let's divide the dataset into 4 intervals and see which interval each value falls into. To do this, I'm going to use the cut method.

```
In [47]:
pd.get_dummies(pd.cut(data,4))
```

Out[47]:

	(-2.177, -1.342]	(-1.342, -0.511]	(-0.511, 0.32]	(0.32, 1.152]
0	0	0	1	0
1	0	0	1	0
2	0	0	1	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0
6	1	0	0	0
7	0	0	0	1
8	0	1	0	0
9	0	0	0	1

4Q) Detecting and Filtering Outliers

Ans:

Identifying and dealing with outliers can be tough, but it is an essential part of the data analytics process, as well as for feature engineering for machine learning.

Outlier:

When exploring data, the outliers are the extreme values within the dataset. That means the outlier data points vary greatly from the expected values—either being much larger or significantly smaller. For data that follows a normal distribution, the values that fall more than three standard deviations from the mean are typically considered outliers.

Outliers can find their way into a dataset naturally through variability, or they can be the result of issues like human error, faulty equipment, or poor sampling. Regardless of how they get into the data, outliers can have a big impact on statistical analysis and machine learning because they impact calculations like mean and standard deviation, and they can skew hypothesis tests.

A data analyst should use various techniques to visualize and identify outliers before deciding whether they should be dropped, kept, or modified.

Using pandas describe() to find outliers

After checking the data and dropping the columns, use .**describe()** to generate some summary statistics. Generating summary statistics is a quick way to help us determine whether or not the dataset has outliers.

df.describe()[['fare_amount', 'passenger_count']]

	fare_amount	passenger_count
count	200000.000000	200000.000000
mean	11.359955	1.684535
std	9.901776	1.385997
min	-52.000000	0.000000
25%	6.000000	1.000000
50%	8.500000	1.000000
75%	12.500000	2.000000
max	499.000000	208.000000

df.describe()

As we can see, the **fare_amount** and **passenger_count** columns have outliers. For example, the max fare_amount is 499 while its mean is 11.36. The mean is sensitive to outliers, but the fact the mean is so small compared to the max value indicates the max value is an outlier. Similarly, the max passenger_count is 208 while the mean is 1.68. Since this value is entered by the driver, my best guess for the passenger_count outlier is human error. As we explore the data using additional methods, we can decide how to handle the outliers.

How do you find outliers in your dataset?

Finding outliers in your data should follow a process that combines multiple techniques performed during your exploratory data analysis. I recommend following this plan to find and manage outliers in your dataset:

- Use data visualization techniques to inspect the data's distribution and verify the presence of outliers.
- Use a statistical method to calculate the outlier data points.
- Apply a statistical method to drop or transform the outliers.

We will explore three different visualization techniques that tackle outliers. After visualizing the data, depending on the distribution of values, we will pick a technique to calculate the outlier data points. Finally, after calculating the outliers, we will discuss three techniques for handling in preparation for data modeling.

How do you visualize outliers?

Now that we've taken a quick look at the statistics, let's perform exploratory data analysis using visualizations to get a better look at the outliers compared to the rest of the data points. There are several different visualizations that will help us understand the data and the outliers. The type of plot you pick will depend on the number of variables you're analyzing. These are a few of the most popular visualization methods for finding outliers in data:

- Histogram
- Box plot
- Scatter plot

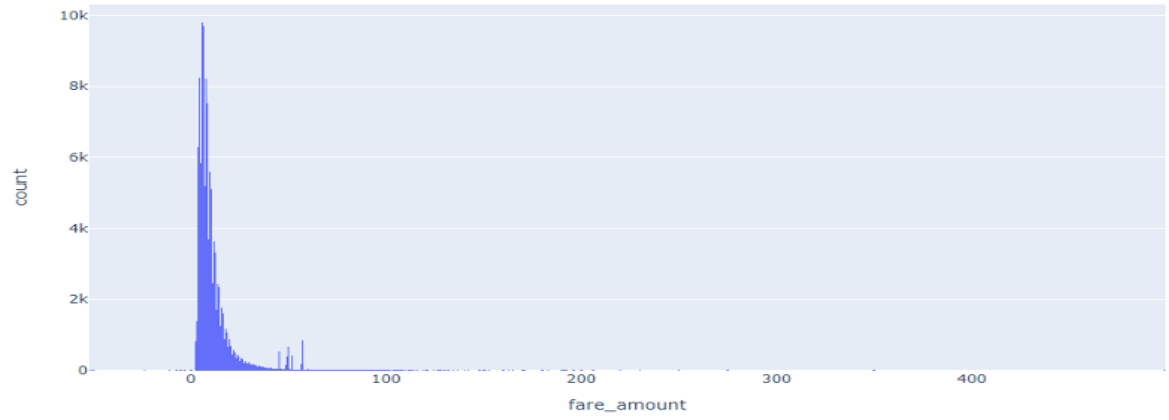
I prefer to use the Plotly express visualization library because it creates interactive visualizations in just a few lines of code, allowing us to zoom in on parts of the chart if needed.

Find outliers and view the data distribution using a histogram

Using a histogram, we can see how the data is distributed. Having data that follows a normal distribution is necessary for some of the statistical techniques used to detect outliers. If the data doesn't follow a normal distribution, the z-score calculation shouldn't be used to find the outliers.

Use a **px.histogram()** to plot to review the *fare_amount* distribution.

```
#create a histogram
fig = px.histogram(df, x='fare_amount')
fig.show()
```



fare_amount histogram

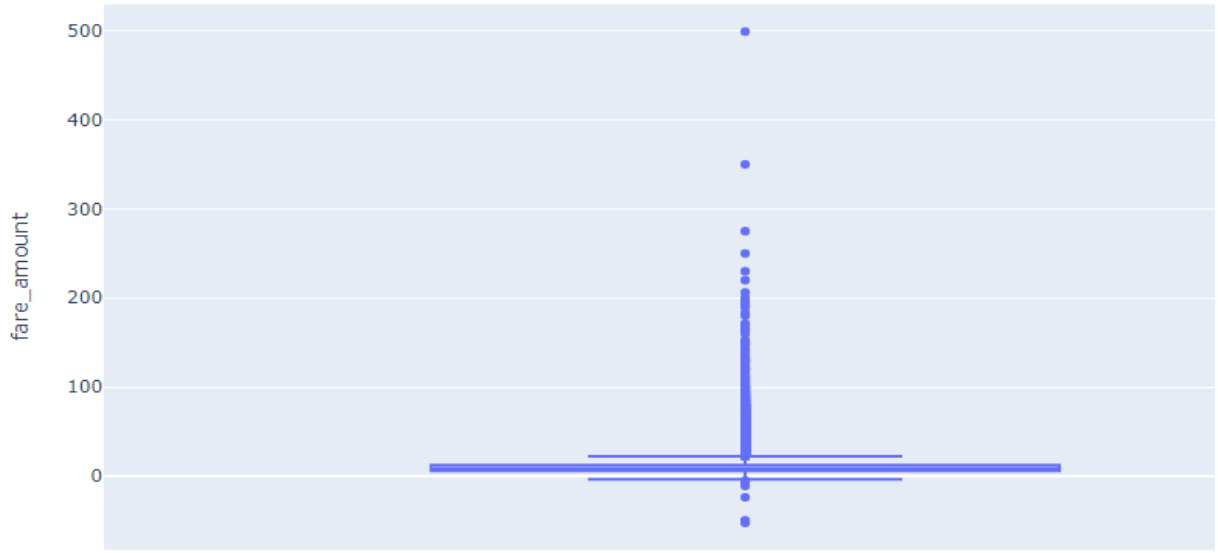
Notice the data does not follow a normal distribution. Since the data is skewed, instead of using a z-score we can use interquartile range (IQR) to determine the outliers. We will explore using IQR after reviewing the other visualization techniques.

Find outliers in data using a box plot

Begin by creating a box plot for the **fare_amount** column. A box plot allows us to identify the univariate outliers, or outliers for one variable. Box plots are useful because they show minimum and maximum values, the median, and the interquartile range of the data. In the chart, the outliers are shown as points which makes them easy to see.

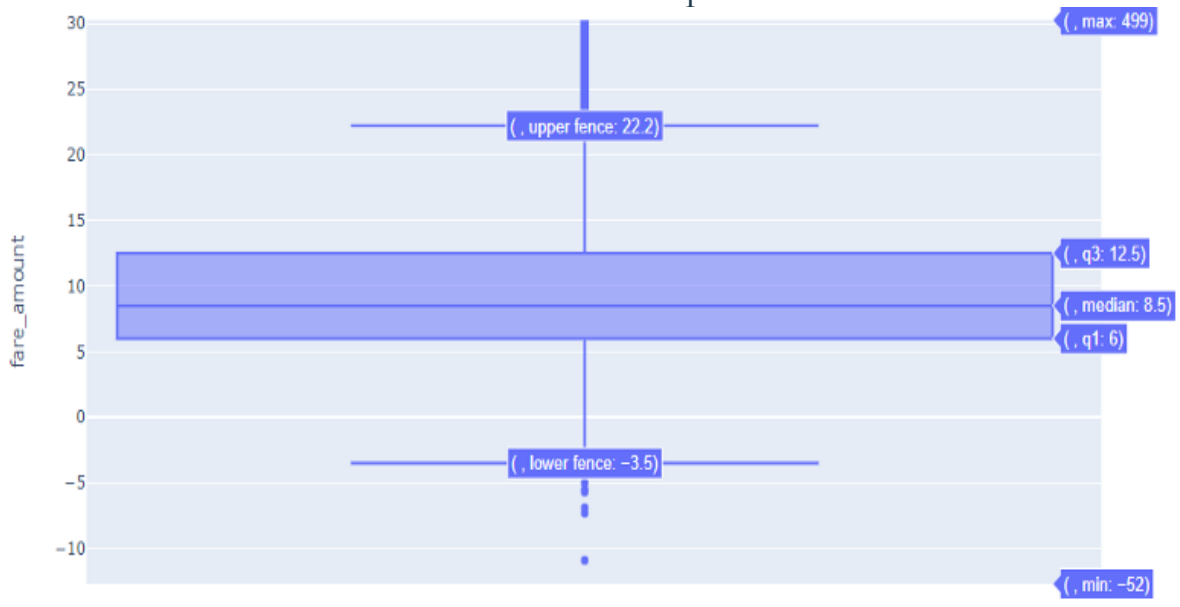
Use **px.box()** to review the values of fare_amount.

```
#create a box plot
fig = px.box(df, y="fare_amount")
fig.show()
```



fare_amount box plot

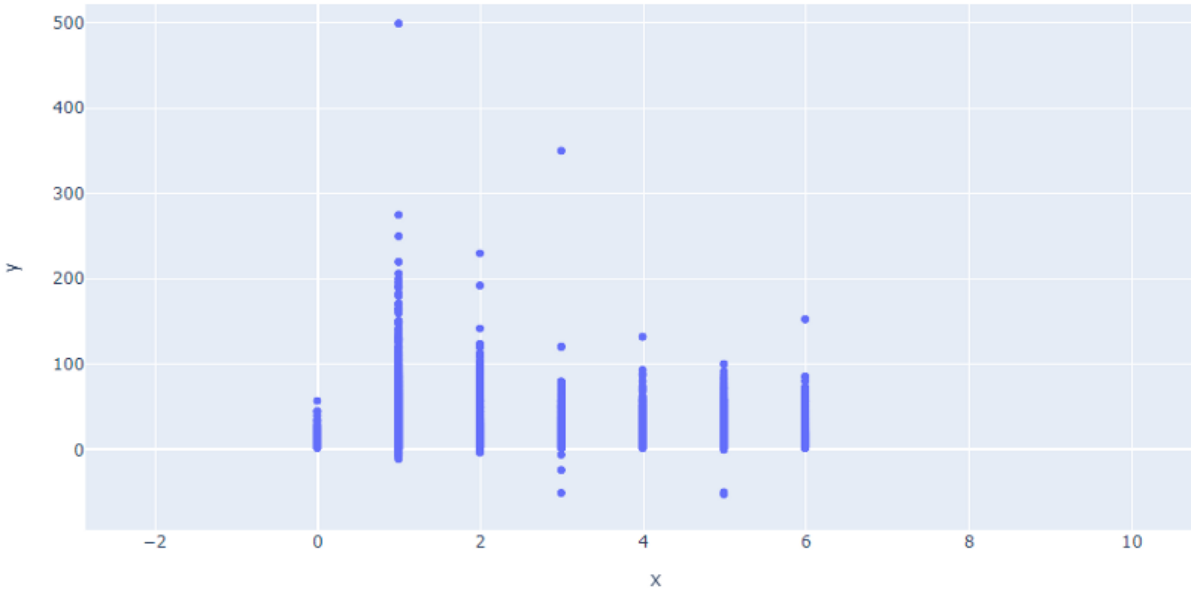
As we can see, there are a lot of outliers. That thick line near 0 is the box part of our box plot. Above the box and upper fence are some points showing outliers. Since the chart is interactive, we can zoom to get a better view of the box and points, and we can hover the mouse on the box to view of the box plot values:



Find multivariate outliers using a scatter plot

Using a Scatter plot, it is possible to review multivariate outliers, or the outliers that exist in two or more variables. For example, in our dataset we see a fare_amount of -52 with a passenger_count of 5. Both of those values are outliers in our data. On the x-axis use the **passenger_count** column. On the y-axis use the **fare_amount** column. Use **px.scatter()** to review passenger_count and fare_amount.

```
fig = px.scatter(x=df['passenger_count'], y=df['fare_amount'])
fig.show()
```



Scatter plot

Since the plot needs to include the 208 passenger_count outlier, I recommend zooming in to get a better look at the distribution of the data in the scatter plot.

Finding outliers using statistical methods

Since the data doesn't follow a normal distribution, we will calculate the outlier data points using the statistical method called interquartile range (IQR) instead of using Z-score. Using the IQR, the outlier data points are the ones falling below Q1–1.5 IQR or above Q3 + 1.5 IQR. The Q1 is the **25th percentile** and Q3 is the **75th percentile** of the dataset, and IQR represents the interquartile range calculated by Q3 minus Q1 (Q3–Q1).

Using the convenient pandas **.quantile()** function, we can create a simple Python function that takes in our column from the dataframe and outputs the outliers:

```
#create a function to find outliers using IQR
def find_outliers_IQR(df):
  q1=df.quantile(0.25)
  q3=df.quantile(0.75)
  IQR=q3-q1
  outliers = df[((df<(q1-1.5*IQR)) | (df>(q3+1.5*IQR)))]
  return outliers
```

Notice using .**quantile()** we can define Q1 and Q3. Next we calculate IQR, then we use the values to find the outliers in the dataframe. Since it takes a dataframe, we can input one or multiple columns at a time.

First run fare_amount through the function to return a series of the outliers.

```
outliers = find_outliers_IQR(df["fare_amount"])
print("number of outliers: "+ str(len(outliers)))
print("max outlier value: "+ str(outliers.max()))
print("min outlier value: "+ str(outliers.min()))
outliers
```

```
number of outliers: 17167
max outlier value: 499.0
min outlier value: -52.0

6         24.50
30        25.70
34        39.50
39        29.00
48        56.80
          ...
199976    49.70
199977    43.50
199982    57.33
199985    24.00
199997    30.90
```

validating the find_outliers_IQR function

Using the IQR method, we find 17,167 fare_amount outliers in the dataset. I printed the min and max values to verify they match the statistics we saw when using the pandas describe() function, which helps confirm we calculated the outliers correctly. We can also pass both fare_amount and passenger_count through the function to get back a dataframe of all rows instead of just the outliers. If the value is not an outlier, it will display as NaN (not a number):

```
outliers = find_outliers_IQR(df[["passenger_count","fare_amount"]])
outliers
```

	passenger_count	fare_amount
0	NaN	NaN
1	NaN	NaN
2	NaN	NaN
3	NaN	NaN
4	5.0	NaN
...	...	...
199995	NaN	NaN
199996	NaN	NaN
199997	NaN	30.9
199998	NaN	NaN
199999	NaN	NaN

find_outliers_IQR dataframe

Working with outliers using statistical methods

After identifying the outliers, we need to decide what to do with them. Unfortunately, there is no straightforward "best" solution for dealing with outliers because it depends on the severity of outliers and the goals of the analysis. For example, since we think the value 208 in the passenger_count was caused by human error, we should treat that outlier differently than the outliers for fare_amount. Here are three techniques we can use to handle outliers:

- Drop the outliers
- Cap the outliers
- Replace outliers using imputation as if they were missing values

Drop the outliers

Using this method, we essentially drop all the outliers from the data, excluding them from the analysis and modeling. Although this technique is quick and easy, it isn't always the right solution and can reduce the amount of data if there are a lot of outliers present. For example, using the IQR method to identify the outliers, we will lose 17,167 rows.

Copy and paste the find_outliers_IQR function so we can modify it to return a dataframe with the outliers removed. Rename it **drop_outliers_IQR**. Inside the function we create a dataframe named *not_outliers* that replaces the outlier values with a NULL. Then we can use **.dropna()**, to drop the rows with NULL values.

```
def drop_outliers_IQR(df):
  q1=df.quantile(0.25)
  q3=df.quantile(0.75)
  IQR=q3-q1
  not_outliers = df[~((df<(q1-1.5*IQR)) | (df>(q3+1.5*IQR)))]
  outliers_dropped = outliers.dropna().reset_index()
  return outliers_dropped
```

	index	Unnamed: 0	key	fare_amount	pickup_datetime	passenger_count
0	0	24238194	2015-05-07 19:52:06.0000003	7.5	2015-05-07 19:52:06 UTC	1.0
1	1	27835199	2009-07-17 20:04:56.0000002	7.7	2009-07-17 20:04:56 UTC	1.0
2	2	44984355	2009-08-24 21:45:00.00000061	12.9	2009-08-24 21:45:00 UTC	1.0
3	3	25894730	2009-06-26 08:22:21.0000001	5.3	2009-06-26 08:22:21 UTC	3.0
4	5	44470845	2011-02-12 02:27:09.0000006	4.9	2011-02-12 02:27:09 UTC	1.0
...	...	...	...	...	...	...
162273	199994	3189201	2014-01-31 14:42:00.000000181	12.0	2014-01-31 14:42:00 UTC	1.0
162274	199995	42598914	2012-10-28 10:49:00.00000053	3.0	2012-10-28 10:49:00 UTC	1.0
162275	199996	16382965	2014-03-14 01:09:00.0000008	7.5	2014-03-14 01:09:00 UTC	1.0
162276	199998	20259894	2015-05-20 14:56:25.0000004	14.5	2015-05-20 14:56:25 UTC	1.0
162277	199999	11951496	2010-05-15 04:08:00.00000076	14.1	2010-05-15 04:08:00 UTC	1.0

162278 rows × 6 columns

Dataframe with outliers dropped

Notice the dataframe is only 162,278 rows once all the outliers have been dropped from fare_amount and passeger_count. After dropping the outliers, it is best to create new visualizations and reexamine the statistics.

Cap the outliers

In this technique, we essentially set a limit for the min and max outlier values. Anything above or below the cap gets set to the capped min or max respectively. For example, if we set the cap max for fare_amount at 20, any outlier above 20 will be set to 20. This technique is used when you can assume that all outliers express the same behaviors or patterns, meaning the model wouldn't learn anything new by allowing the outliers to remain.

To cap the outliers, calculate a upper limit and lower limit. For the upper limit, we will use the *mean* plus three standard deviations. For the lower limit, we will calculate it as the mean minus 3 standard deviations. Keep in mind, the calculation you use can depend on the data's distribution.

```
upper_limit = df['fare_amount'].mean() + 3*df['fare_amount'].std()
print(upper_limit)
lower_limit = df['fare_amount'].mean() – 3*df['fare_amount'].std()
print(lower_limit)
```

Based on our calculated limits, any outliers above 41.06 will be set to 41.06. Likewise, any outlier below -18.34 will be set to -18.34.

After calculating the upper and lower limit, we use the numpy **.where()** function to apply the limits to fare_amount.

```
df['fare_amount'] = np.where(df['fare_amount'] > upper_limit,
  upper_limit,
  np.where(
     df['fare_amount'] < lower_limit,
     lower_limit,
     df['fare_amount']
  )
)
```

We can use .describe() to verify the min and max values have been capped as expected:

```
df.describe()[['fare_amount']]
```

	fare_amount
count	200000.000000
mean	11.008988
std	8.088084
min	-18.345373
25%	6.000000
50%	8.500000
75%	12.500000
max	41.065284

Replace outliers using imputation as if they were missing values

The third technique for handling outliers is similar to capping the values. Instead of using a capping calculation, use whatever imputation technique is being used on the missing values. For example, if the fare_amount column had missing values, we might find it appropriate to fill in the missing values using the mean. Since that is how we treat the missing values, we would do the same thing for the outliers.

Use a function to find the outliers using IQR and replace them with the mean value. Name it impute_outliers_IQR. In the function, we can get an upper limit and a lower limit using the .**max()** and .**min()** functions respectively. Then we can use numpy **.where()** to replace the values like we did in the previous example.

```
def impute_outliers_IQR(df):
  q1=df.quantile(0.25)
  q3=df.quantile(0.75)
  IQR=q3-q1
  upper = df[~(df>(q3+1.5*IQR))].max()
  lower = df[~(df<(q1-1.5*IQR))].min()
  df = np.where(df > upper,
     df.mean(),
     np.where(
        df < lower,
        df.mean(),
        df
        )
     )
  return df
```

We can pass fare_amount through the impute_outliers_IQR function to transform the outliers into the mean value. We can use .describe() to verify the function works.

```
df['fare_amount'] = impute_outliers_IQR(df['fare_amount'])
df.describe()['fare_amount']
```

```
count    200000.000000
mean          9.147804
std           4.019032
min          -3.500000
25%           6.000000
50%           8.500000
75%          11.359955
max          22.200000
```

df.describe()
['fare_amount']

As we can see, there are still more than 200,000 rows, the **min** is our lower limit and the **max** is the upper limit. That means the function was successful.