New Syllabus

B. Sc. (Computer Science) - Third Year

Data Analysis and Visualization with Python

Module-I

Python Basics : Python Interpreter, Python Idle, Dynamically Typed and Strongly Typed Features, Basic Data Types, Variables, Expressions, Statements, Operators, Flow of Execution. Input and Output Statements, Conditionals : Boolean Values and Operators, Conditional (if), Alternative (else), Chained Conditional (if-elif-else). Iteration : While, For, Break, Continue, Pass, Implementing 'for' through Range(), 'in' and 'not in' Operators for Sequence Traversal. Creating and Executing .py Scripts.

Module-II

Data Structures : Lists : Append, Extend, Insert, Index, Remove, Pop, Count, Sort, Reverse, Slicing, List Comprehension. Copying a List : Deep Copy, Shallow Copy. Tuples : Index, Count, Usage, Use of Tuples as a Swap Function. Dictionaries : Keys, Values, Tuples, Nested Dictionaries, Dictionary Comprehension. Strings : Single Line and Multi-Line Strings, Formatter, isdigit, isalpha, isalnum, islower, istitle, isspace, title, lower, upper, strip, split, splitlines, join etc. Sets : union, intersection, subset, superset, difference, symmetric difference, copy, add, remove, discard etc.

Function & File Handling : Inbuilt Functions : id, len, chr, ord etc, Defining and Calling a Function, Arguments, Global versus Local Variables, Defining and Using Lambda Functions, the map(), filter(), reduce(), functions.

Module-III

NumPy : Introduction to Numpy, NumPy Array in Python, Basics of NumPy Arrays, Comparison of Python Lists with Numpy Arrays.

Array : Array Creation, The Arrange Method, The Zero Method, Numpy Array Filled with All Ones, The Linspace Method, The Eye Method, Numpy Meshgrid Function, Empty and Full NumPy Array. NumPy Array Filled with All Zeros, Numpy Array Filled with All Ones, 2-D Gaussian Array, Creating Vector in Python using NumPy.

Array Indexing, Array Slicing, Data Types, Copy vs. View, Array Shape, Array Reshape, Array Iterating, Array Join, Array Split, Array Search, Array Sort, Array Filter, Concatenation of Two Arrays, Splitting and Comparison of Arrays, Binary Operations, Mathematical Function, String Operations.

Module-IV **Matrix in NumPy :** Matrix Manipulation in Python, empty() Function, zeros() Function, ones() Function, eye() Function, identity() Function, Adding and Subtracting Matrices in Python, Vector Multiplication, Dot Product of Two Arrays.

Operations on NumPy Array : Broadcasting with NumPy Arrays, Sorting, Searching and Counting of NumPy Arrays, Variations in Different Sorting Techniques in Python.

Universal Functions : Creation of ufunc, Simple Arithmetic, Rounding, Trigonometric, Hyperbolic, Set Functions.

Module-V **Data Visualization with Matplotlib :** Overview of Matplotlib and its Capabilities, Creating Line Plots and Scatter Plots, Customizing : Labels, Titles, Colors, Legends, Creating Bar Plots and Histograms, Adding Annotations and Text to Plots, Creating Subplots and Multiple Plots, Saving and Exporting Plots.

Advanced Data Visualization : Creating Pie Charts and Box Plots, Visualizing 3D Data with Matplotlib, Interactive Visualization using Widgets.

Question List

Module - I

Q.1 Write about Introduction to Python.
Q.2 Write about Python Interpreter.
Q.3 Write about Python Idle.
Q.4 Write about Dynamically Typed and Strongly Typed Features in Python.
Q.5 Write about Basic Data Types in Python.
Q.6 Write about Variables in Python.
Q.7 Write about Expressions in Python.

Q.8 Write about Statements in Python.
Q.9 Write about Operators in Python.
Q.10 Write about Flow of Execution in Python.
Q.11 Write about Input and Output Statements in Python.
Q.12 Write about Conditionals in Python.
Q.13 Write about Iteration in Python.
Q.14 Write about Implementing 'for' Through Range( ) in Python.
Q.15 Write about 'in' and 'not in' Operators for Sequence Traversal in Python.
Q.16 Write about Creating and Executing .py Scripts in Python.

Module - II

Q.1 Write about Data Structures in Python.
Q.2 Write about Lists in Python.
Q.3 Write about Tuples in Python.
Q.4 Write about Dictionary in Python.
Q.5 Write about Strings in Python.
Q.6 Write about Sets in Python.
Q.7 Write about Inbuilt Functions in Python.
Q.8 Write about Defining and Calling a Function in Python.
Q.9 Write about Arguments in Python.
Q.10 Write about Global vs. Local Variables in Python.
Q.11 Write about Defining and Using lambda Function in Python.

Module - III

Q.1 Write about Introduction to Numpy in Python.
Q.2 Write about NumPy Array in Python.
Q.3 Write about Array in Python.
Q.4 Write about Indexing and Searching in Array in Python.

Module - IV

Q.1 Write about Matrix in Numpy.

Module - V

Q.1 Write about Matplotlib in Python.
Q.2 Write about Drawing Plots in Python.
Q.3 Write about Data Visualization with Matplotlib in Python.

Module-I

Q.1 Write about Introduction to Python.

Ans. Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactical constructions than other languages.

1 **Python is Interpreted** - Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

2 **Python is Interactive** - You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

3 **Python is Object-Oriented** - Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

4 **Python is a Beginner's Language** - Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features

Python's features include -

1 **Easy-to-learn** - Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

2 **Easy-to-read** - Python code is more clearly defined and visible to the eyes.

3 **Easy-to-maintain** - Python's source code is fairly easy-to-maintain.

4 **A broad standard library** - Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

5 **Interactive Mode** - Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

6 **Portable** - Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

7 **Extendable** - You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

8 **Databases** - Python provides interfaces to all major commercial databases.

9 **GUI Programming** - Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

10 **Scalable** - Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below -

(i) It supports functional and structured programming methods as well as OOP.

(ii) It can be used as a scripting language or can be compiled to byte-code for building large applications.

(iii) It provides very high-level dynamic data types and supports dynamic type checking.

(iv) It supports automatic garbage collection.

(v) It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python is available on a wide variety of platforms including Linux and Mac OS X.

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Setting up PATH

Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The path variable is named as PATH in Unix or Path in Windows (Unix is case sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Python Environment Variables

Here are important environment variables, which can be recognized by Python -

1 **PYTHONPATH** : It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.

2 **PYTHONSTARTUP** : It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.

3 **PYTHONCASEOK** : It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.

4 **PYTHONHOME** : It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Running Python

There are three different ways to start Python-

(i) **Interactive Interpreter :** You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter python the command line.

Start coding right away in the interactive interpreter.

```
spython # Unix/Linux
```


or

```
python% # Unix/Linux
```

or

```
C:> python # Windows/Dos
```

Here is the list of all the available command line options-

Sr.No.	Option & Description	
1	**–d**	It provides debug output
2	**–O**	It generates optimized bytecode (resulting in .pyo files).
3	**–S**	Do not run import site to look for Python paths on startup.
4	**–v**	verbose output (detailed trace on import statements).
5	**–X**	disable class-based built-in exceptions(just use strings); obsolete starting with version 1.6.
6	**–c cmd**	run Python script sent in as cmd string
7	**file**	run Python script from given file

(ii) **Script from the Command-line :** A Python script can be executed at command line by invoking the interpreter on your application, as in the following

```
Spython script .py #Unix/Linux
```

or

```
python% script .py # Unix/Linux
```

or

```
C: >python script .py # Windows/DOS
```

Note- Be sure the file permission mode allows execution.

(iii) **Integrated Development Environment :** You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

(i) **Unix** - IDLE is the very first Unix IDE for Python.

(ii) **Windows** - Python Win is the first Windows interface for Python and is an IDE with a GUI.

(iii) **Macintosh** - The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files

If your are not able to set up the environment properly, then you can take help from your system admin. Make sure the Python environment is properly set up and working perfectly fine.

Q.2 Write about Python Interpreter.

Ans. The default encoding for a Python source file is UTF-8. This is a Unicode Standard variable-width character encoding; it can encode 1,112,064 valid code points in Unicode using up to four 8-bit bytes. Using this encoding, we can use characters of most languages - we can use these in string literals, comments, and identifiers. Since the standard library makes use of ASCII characters only, you must declare the use of this encoding to your editor. This is to ensure that all such characters display without a problem. The font should be such that supports all characters in the file.

We add this comment as the first line of the file we want to use it in-

```
# -*- coding: encoding -*-
```

In this, encoding is a valid codec that Python supports. Similarly, when you want to use the Windows-1252 encoding, you can use this as the first line of code:

```
# -*- coding: cp1252 -*-
```

However, when you want to begin code with a UNIX shebang line, you can put the comment for encoding second-

```
1  #!/usr/bin env python3
2  # -*- coding: cp1252 -*-
```

How to Invoke the Python Interpreter?

On your machine, you can find your interpreter at an address like:

C:\Python36

Or it may reside on the location you selected at the time of installation. Add path using this command:

```
set path="%path%;C:\python36
```

Start the Python Interpreter

On Windows, when you want to run the Python interpreter in the shell, you can type the following:

```
$python
```

To get out of the interpreter in disassembling the Bytecode shell, you can type:

```
>>> quit()
```

Python Multiple Inheritance Python MRO (Method Resolution Order)

Alternately, you can use an 'end-of-file' character at the prompt. Python interpreter exits with a zero exit status. You can use it in a REPL (Read-Evaluate-Print-Loop) fashion. But if you want, you can save your Python code as a script and execute it using the interpreter:


```
$python demo.py
```

To enter interactive mode after running a script, you can pass -i before the script.

The command python -c command [arg] ... executes statements in command, and python -m module [arg] ... executes the source file for module.

Features of Python Interpreter

Some features: of paython interpreter are as follows :

(i) Interactive editing

(ii) History substitution

(iii) Code completion on systems with support for readline

In the first Python prompt, try pressing the following keys:

Ctrl+P

This tells you if your interpreter supports command-line editing. A beep indicates that it does support command-line editing. Otherwise, it will either perform a no-operation or echo ^p to indicate it isn't available.

Passing Arguments

When you pass a script name and additional arguments to the shell when invoking the Python interpreter, it turns these into a list of strings. Then, it assigns these to the variable argv in the sys module. The following command will give us a list of this-

```
import sys
```

Without a script or arguments, sys.argv[0] denotes an empty string. A script name of '-' means that it sets sys.argv[0] to '-', and with '-c', it is set to '-c'. A value of '-m' sets sys.argv[0] to the module's full name. The command/module handles the options after '-c' or '-m'.

Interactive Mode

Python interpreter is in an interactive mode when it reads commands from a tty. The primary prompt is the following:

```
>>>
```

When it shows this prompt, it means it prompts the developer for the next command. This is the REPL. Before it prints the first prompt, Python interpreter prints a welcome message that also states its version number and a copyright notice.

This is the secondary prompt:

```
...
```

This prompt denotes continuation lines.

```
1.  $ python3.7
```

2. Python 3.7 (default, Jul 16 2018, 04:38:07)
3. [GCC 4.8.2] on Windows
4. Type "help", "copyright", "credits" or "license" for more information.
5. >>>

You will find continuation lines when working with a multi-line construct:

1. >>> it_rains =True
2. >>> if it_rains:
3. print("The produce will be good")

You can also use the Python interpreter as a calculator:

1. >>> 2*7
2. 14
3. >>> 4/2
4. 2.0

How Does Python Interpreter Works?

Well, internally, four things happen in a REPL:

i. **Lexing** - The lexer breaks the line of code into tokens.

ii. **Parsing** - The parser uses these tokens to generate a structure.

iii. **Compiling** - The compiler turns this AST into code object(s).

iv. **Interpreting** - The interpreter executes each code object.

Q.3 Write about Python Idle.

Ans. IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python. The Python installer for Windows contains the IDLE module by default.

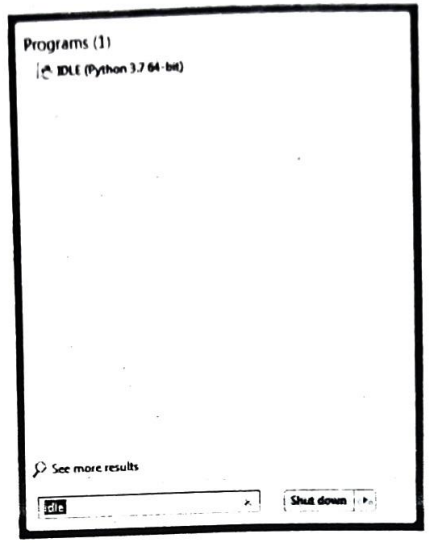
IDLE is not available by default in Python distributions for Linux. It needs to be installed using the respective package managers. For example, in case of Ubuntu:

$ sudo apt-get install idle

IDLE can be used to execute a single statement just like Python Shell and also to create, modify and execute Python scripts. IDLE provides a fully-featured text editor to create Python scripts that includes features like syntax highlighting, autocompletion and smart indent. It also has a debugger with stepping and breakpoints features.

To start IDLE interactive shell, search for the IDLE icon in the start menu and double click on it.

Python IDLE



This will open IDLE, where you can write Python code and execute it as shown below.

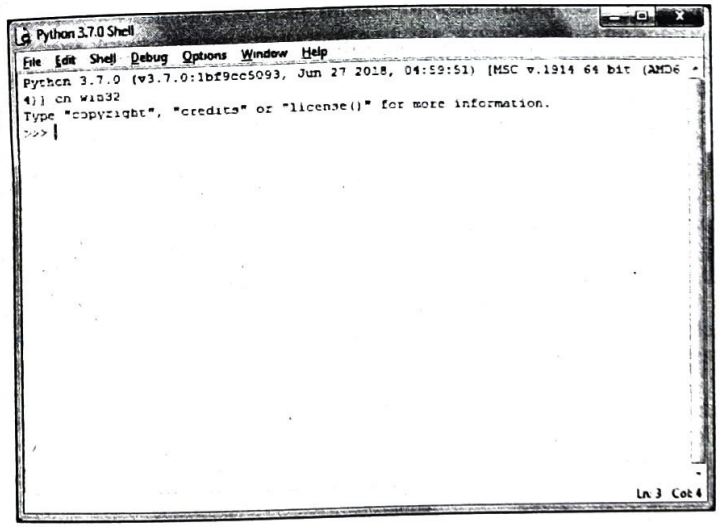


Fig.

Now, you can execute Python statements same as in Python Shell as shown below.

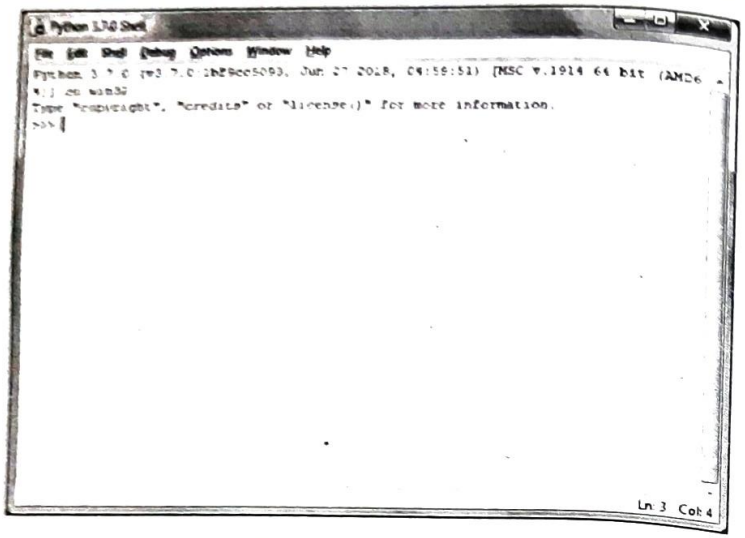


Fig. : Python IDLE

To execute a Python script. create a new file by selecting File -> New File from the menu.

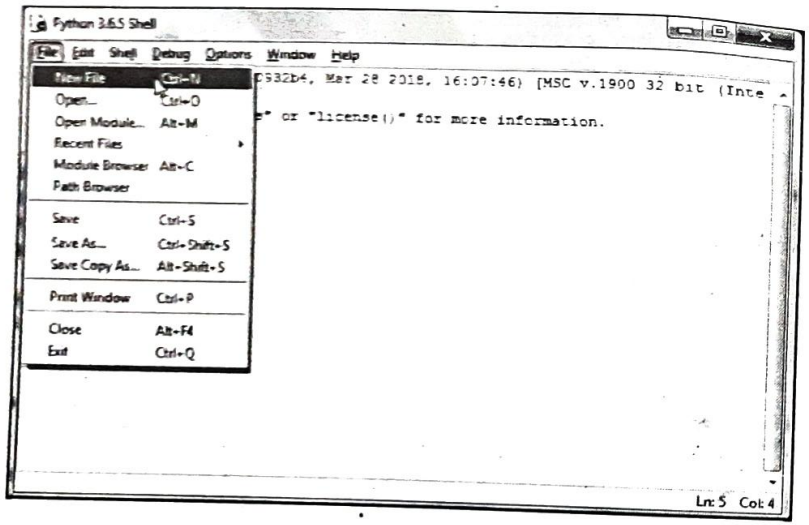


Fig.

Enter multiple statements and save the file with extension .py using File -> Save. For example, save the following code as hello.py.

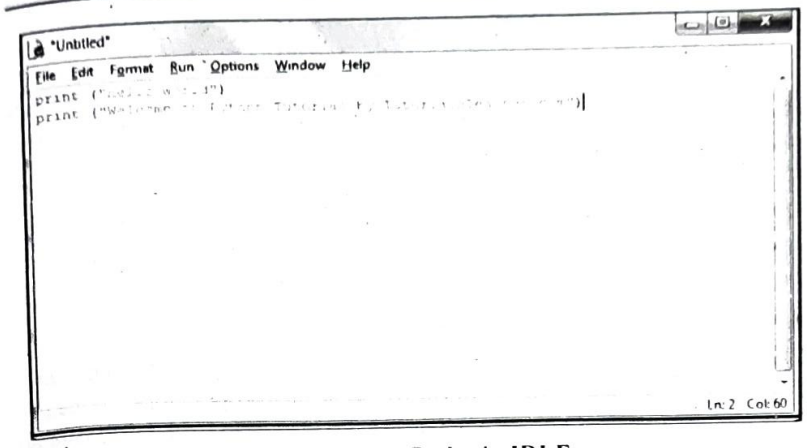


Fig. : Python Script in IDLE

Now, press F5 to run the script in the editor window. The IDLE shell will show the output.

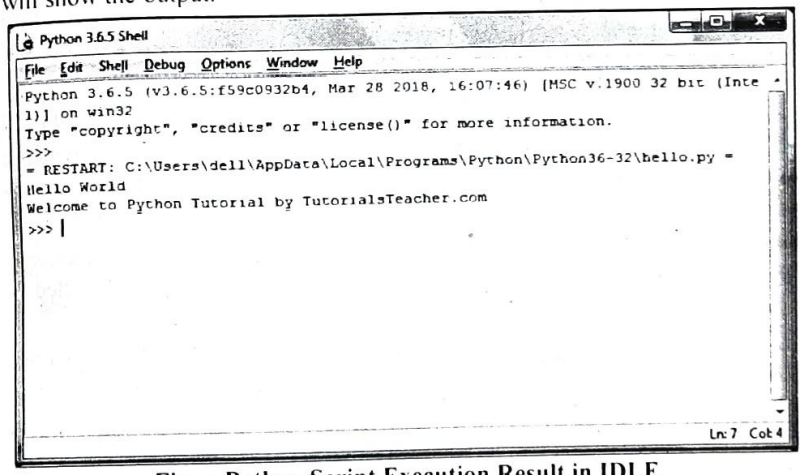


Fig. : Python Script Execution Result in IDLE

Thus, it is easy to write, test and run Python scripts in IDLE.

Q.4 Write about Dynamically Typed and Strongly Typed Features in Python.

Ans. Dynamically typed: Dynamic typed programming languages are those languages in which variables must necessarily be defined before they are used. This implies that dynamic typed languages do not require the explicit declaration of the variables before they're used. Python is an example of a dynamic typed programming language, and so is PHP. Consider the following example:

```
/* Python code */
num = 10 // directly using the variable
```

In the above code fragment, we have directly assigned the variable num the value 10 before initializing it. This is characteristic to dynamic typed programming languages.

Strongly typed: Programming languages in which variables have specific data types are strong typed. This implies that in strong typed languages, variables are necessarily bound to a particular data type. Python is strong typed, and so is Java.

Q.5 Write about Basic Data Types in Python.

Ans. Data types are the classification or categorization of data items. Data types represent a kind of value which determines what operations can be performed on that data. Numeric, non-numeric and Boolean (true/false) data are the most used data types.

Python has the following standard or built-in data types:

(1) **Numeric :** Standard date types,- Numbers, strings, Lists Tuple, Dictionary A numeric value is any representation of data which has a numeric value. Python identifies three types of numbers:

(i) **Integer:** Positive or negative whole numbers (without a fractional part)

(ii) **Float:** Any real number with a floating point representation in which a fractional component is denoted by a decimal symbol or scientific notation

(iii) **Complex number:** A number with a real and imaginary component represented as x+yj. x and y are floats and j is -1(square root of -1 called an imaginary number)

(2) **Boolean :** Data with one of two built-in values True or False. Notice that 'T' and 'F' are capital. true and false are not valid booleans and Python will throw an error for them.

Non-Numerical Data Type

(1) **Sequence Type :** A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types:

(i) **String :** A string value is a collection of one or more characters put in single, double or triple quotes.

(ii) **List :** A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.

(iii) **Tuple :** A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

(2) **Dictionary :** A dictionary object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

type() function : Python has an in-built function 'type()' to ascertain the data type of a certain value. For example, enter type(1234) in Python shell and it will return <class 'int'>, which means 1234 is an integer value. Try and verify the data type of different values in Python shell, as shown below.

```
>>> type(1234)
<class 'int'>
>>> type(55.50)
<class 'float'>
>>> type(6+4j)
<class 'complex'>
>>> type("hello")
<class 'str'>
>>> type([1,2,3,4])
<class 'list'>
>>> type((1,2,3,4))
<class 'tuple'>
>>> type({1:"one", 2:"two", 3:"three"})
<class 'dict'>
```

Mutable and Immutable Objects : Data objects of the above types are stored in a computer's memory for processing. Some of these values can be modified during processing, but contents of others can't be altered once they are created in the memory.

Number values, strings, and tuple are immutable, which means their contents can't be altered after creation.

On the other hand, collection of items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are mutable objects.

Q.6 Write about Variables in Python.

Ans. A variable is something which can change. A variable is a way of referring to a memory location used by a computer program. A variable is a symbolic name for this physical location. This memory location contains values, like numbers, text or more complicated types.

A variable can be seen as a container to store certain values. While the program is running, variables are accessed and sometimes changed, i.e. a new value will be assigned to the variable. Python variable is a reserved memory

location to store values. In other words, a variable in a python program gives data to the computer for processing.

Any value of certain type is stored in the computer's memory for processing. Out of available memory locations, one is randomly allocated for storage. In order to conveniently and repeatedly refer to the stored value, it is given a suitable name. A value is bound to a name by the assignment operator '='.

```
>>> myVar=21
```

Here 'myVar' is an identifier (name) referring to integer value 21 (Python treats data value as a value). However, the same identifier can be used to refer to another value. For example, the below-code will assign myVar as the name of a string value, as shown below.

```
>>> myVar="Steve"
```

So, the value being referred can change (or vary), hence it is called a variable. It is important to remember that a variable is a name given to a value, and not to a memory location storing the value.

Variables can be accessed by using their identifier (name) as shown below.

```
>>> myVar="Steve"
>>> myVar
'Steve'
```

Dynamic Typing

One of the important features of Python is that it is a dynamically-typed language. Programming languages such as C, C++, Java, and C# are statically-typed languages. A variable in these languages is a user-friendly name given to a memory location and is intended to store the value of a particular data type.

A variable in Python is not bound permanently to a specific data type. Instead, it only serves as a label to a value of a certain type. Hence, the prior declaration of variable's data type is not possible. In Python, the data assigned to a variable decides its data type and not the other way around.

In the following Python statements, a string value is assigned to a variable 'name'. We can test its type using the 'type()' function.

```
>>> name="Steve"
>>> type(name)
<class 'str'>
```

Now the Python interpreter won't object if the same variable is used to store a reference to an integer.

```
>>> name=1234
>>> type(name)
```



```
<class 'int'>
```

We can see that the data type of a variable has now been changed to integer. This is why Python is called a dynamically-typed language.

Naming Conventions

Any suitable identifier can be used as a name of a variable, based on the following rules:

1. The name of the variable should start with either an alphabet letter (lower or upper case) or an underscore (_), but it cannot start with a digit.
2. More than one alpha-numeric characters or underscores may follow.
3. The variable name can consist of alphabet letter(s), number(s) and underscore(s) only. For example, myVar, MyVar, _myVar, MyVar123 are valid variable names but m*var, my-var, 1myVar are invalid variable names.
4. Identifiers in Python are case sensitive. So, NAME, name, nAME, and nAmE are treated as different variable names.

Starting the name with a single or double underscore has a special meaning in Python.

Q.7 Write about Expressions in Python.

Ans. An expression is a type of Python statement which contains a logical sequence of numbers, strings, objects, and operators. The value in itself is a valid expression and so is a variable.

Using expressions, we can perform operations like addition, subtraction, concatenation and so on. It can also have a call to a function which evaluates results.

Expressions are representations of value. They are different from statement in the fact that statements do something while expressions are representation of value. For example any string is also an expression since it represents the value of the string as well.

Python has some advanced constructs through which you can represent values and hence these constructs are also called expressions.

How to create an expression

Python expressions only contain identifiers, literals, and operators.

Identifiers: Any name that is used to define a class, function, variable, module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

```
>>> test = "Learn Python"
>>> id(test)
6589040
```

The number is the address of the location where the data lives in memory.

Case-2: Right-hand side (RHS) is a current Python variable : Let's take up the next type of assignment statement where the RHS is a current Python variable.

```
>>> another_test = test
```

The above statement won't trigger any new allocation in memory. Both the variables would point to the same memory address. It's like creating an alias to the existing object. Let's validate this by using the id() function.

```
>>> test = "Learn Python"
>>> id(test)
6589424
>>> another_test = test
>>> id(another_test)
6589424
```

Case-3: Right-hand side (RHS) is an operation : In this type of statement, the result would depend on the outcome of the operation. Let's analyze it with the following example.

```
>>> test = 2 * 5 / 10
>>> print(test)
1.0
>>> type(test)
<class 'float'>
```

In the above example, the assignment would lead to the creation of a "float" variable.

```
>>> test = 2 * 5
>>> print(test)
10
>>> type(test)
<class 'int'>
```

In this example, the assignment would lead to the creation of an "int" variable.

(2) **Augmented Assignment Statement :** You can combine arithmetic operators in assignments to form an augmented assignment statement.

Check out the below examples for augmented assignment statement.


```
x += y
```

The above statement is a shorthand for the below simple statement.

```
x = x + y
```

Next one is a bit clearer example where we are appending new elements to the tuple.

```
>>> my_tuple = (10, 20, 30)
>>> my_tuple += (40, 50,)
>>> print(my_tuple)
(10, 20, 30, 40, 50)
```

Next example is using a list of vowels. It is demonstrating the addition of missing vowels to the list.

```
>>> list_vowels = ['a','e','i']
>>> list_vowels += ['o', 'u',]
>>> print(list_vowels)
['a', 'e', 'i', 'o', 'u']
```

(3) **Multi-line Statement in Python :** Usually, every Python statement ends with a newline character. However, we can extend it over to multiple lines using the line continuation character (\). And Python gives us two ways to enable multi-line statements in a program.

(i) **Explicit line continuation :** When you right away use the line continuation character (\) to split a statement into multiple lines.

Example

```
# Initializing a list using the multi-line statement
>>> my_list = [1, \
... 2, 3\
...,4,5 \
... ]
>>> print(my_list)
[1, 2, 3, 4, 5]
# Evalulate an expression using a multi-line statement
>>> eval ( \
... " 2.5 \
... + \
... 3.5")
6.0
```

(ii) **Implicit line continuation :** Implicit line continuation is when you split a statement using either of parentheses ( ), brackets [ ] and braces { }. You need to enclose the target statement using the mentioned construct.

|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

(5) **Assignment operators :** Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

Assignment operators in Python

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

(6) **Special operators :** Python language offers some special type of operators like the identity operator or the membership operator. They are as follows :-

(i) **Identity operators :** 'is' and 'is not' are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Identity operators in Python

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

(ii) **Membership operators :** 'in 'and' not in' are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Q.10 Write about Flow of Execution in Python.

Ans. In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

'Function definitions' do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

Q.11 Write about Input and Output Statements in Python.

Ans. A Program needs to interact with the user to accomplish the desired task; this is done using Input-Output facility. Input means the data entered by the user of the program. In python, we have input() and raw_input ( ) function available for Input.

statement(s)

If the Boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Example:

```
a=10
if a>9:
print("A is Greater than 9")
Output: A is Greater than 9
```

Alternative If(If-Else)

An else statement can be combined with an if statement. An else statement contains the block of code (false block) that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else statement following if.

Syntax:

```
if expression:
statement(s)
else:
statement(s)
```

Example :

```
a=10
b=20
if a>b:
print("A is Greater than B")
else:
print("B is Greater than A")
Output
B is Greater than A
```

Chained Conditional If: (If-Elif-Else)

The elif statement allows us to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE. Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax:

```
if expression1:
statement(s)
elif expression2:
statement(s)
elif expression3:
statement(s)else:
statement(s)
```

Example:

```
var = 100
if var == 200:
print "1 -Got a true expression value"
print var
elif var == 150:
print "2 -Got a true expression value"
print var
elif var == 100:
print "3 -Got a true expression value"
print var
else:
print "4 -Got a false expression value"
print var
print "Good bye!"
Output:
3 -Got a true expression value
100
Good bye!
```

Nested Conditionals

One conditional can also be nested within another.

Syntax:

```
if expression1:
statement(s)
else:
if expression2:
statement(s)
```

```
Current Letter : o
Current Letter : n
```

(5) **Pass**: The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet

Example:

```
for letter in „Python":
if letter == „h":
pass
print „This is pass block"
print „Current Letter :", letter
print "Good bye!"
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye
```

Q.14 Write about Implementing 'for' Through Range( ) in Python.

Ans. In Python, the 'range()' method returns an immutable sequence of numbers. It can be used to control the repetition of a block in the for loop.

Syntax:

```
range([start], stop, [step])
```

All three parameters should be integers. The sequence starts with 0, by default, unless the [start] parameter is provided. The only mandatory parameter for the above function is stop. The last number in the sequence is the number before stop. The numbers in between are incremented with the [step] value, which is 1 by default.

range(5) will generate 0,1,2,3,4

range(1,10) will generate 1,2,3,4,5,6,7,8,9

range(10,21,2) will generate 10,12,14,16,18,20

We can run the for loop over a range to print each number in a given range.

Example: for loop with range()


```
x=range(5)
for num in x:
    print(num)
```

Output :

```
0
1
2
3
4
```

The 'range()' function can also be used directly with the for loop.

Example: for loop with range()

```
for num in range(5):
    print(num)
```

Q.15 Write about 'in' and 'not in' Operators for Sequence Traversal in Python.

Ans.

Python Membership and Identity Operators

Identity operators are used to verify if two variables point to the same memory location or not. Identity operators are of two types:

(1) is and

(2) is not.

The membership operators are used to validate the membership of a value. These operators are used to find out whether a value is a member of a sequence such as string or list. Membership operators are of two types:

(1) in and

(2) not in.

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below -

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here 'in' results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the	x not in y, here 'not in' results in a 1 if x is not a member of sequence

specified sequence y.
and false otherwise.

(1) **in operator :** The 'in' operator is used to check if a value exists in a sequence or not. If it exists in the sequence then it will return a true else it will return a false. Following are two examples that show how this operator can be used. The first example tries to check if a value exists in a list and the second example tries to find out if a value exists in a string.

Exp. 1.

```
lst1 = [`Ajay', `Bobby', `Ashok', `Vijay', `Anil', `Rahul', `Alex', `Christopher']
if `Ajay` in lst1: print(`Name Ajay exists in lst1')
```

Exp. 2

```
if `j` in `Ajay`: print("letter `j` exists in name `Ajay` ")
else: print ("letter `j` does not exists in name `Ajay` ")
if `m` in `Ajay`: print("letter `m` exists in name `Ajay` ")
else: print ("letter `m` does not exists in name `Ajay` ")
```

(2) **not in operator :** The 'not in' operator is the opposite of 'in' operator. So, if a value does not exists in the sequence then it will return a true else it will return a false. Following are two examples that show how this operator can be used. The first example shows how this operator can be used with a list and the second example deals with string.

Exp. 1

```
lst1 = ['Ajay', 'Bobby', 'Ashok', 'Vijay', 'Anil', 'Rahul', 'Alex', 'Christopher']
if 'Raghav' not in lst1: print ('Name Raghav does not exists in lst1')
```

Exp. 2

```
if 'j' not in 'Ajay': print ("letter 'j' does not exists in name `Ajay` ")
else: print ("letter 'j' exists in name 'Ajay' ")
if 'm' not in 'Ajay': print("letter 'm' does not exists in name 'Ajay' ")
else: print ("letter 'm' exists in name 'Ajay' ")
```

Q.16 Write about Creating and Executing .py Scripts in Python.

Ans. A Python script normally can be full of functions that can be imported as a library of functions in other scripts, or a python script can be a command that runs in Rhino.

There is a way to have Python definitions to be both a library of functions and a direct command.

The key is to add these statements to the end of the file:


```
if __name__ == '__main__':
    CreateCircle() # Put the a call to the main function in the file.
```

How it works

When the Python interpreter reads a source file, it executes all of the code found in it.

Before executing the code, it will define a few special variables. If Python is running the file as the main program then Python will create a __name__ variable with the value of main. If python is importing the file as an import into an already running main, the __name__ variable will be set to the module's name in the modules scope.

One of the reasons for doing this is that sometimes you write a module (a .py file) where it can be executed directly. Alternatively, it can also be imported and used in another module. By doing the main check, you can have that code only execute when you want to run the module as a program and not have it execute when someone just wants to import your module and call your functions themselves.

Execute a Python script

We start a Python interactive shell in a terminal with the command "python". It might be necessary to use "python3" to get a Python3 shell:

```
$ python3
(base) bernd@moon:~$ python
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

Module-II

Q.1 Write about Data Structures in Python.

Ans. Data structures are basically just that - they are structures which can hold some data together. In other words, they are used to store a collection of related data.

There are four built-in data structures in Python - list, tuple, dictionary and set.

There are quite a few data structures available. The builtins data structures are: lists, tuples, dictionaries, strings, sets and frozensets.

Lists, strings and tuples are ordered sequences of objects. Unlike strin that contain only characters, list and tuples can contain any type of object Lists and tuples are like arrays. Tuples like strings are immutables. Lists a mutables so they can be extended or reduced at will. Sets are mutable unordere sequence of unique elements whereas frozensets are immutable sets.

Lists are enclosed in brackets:

```
l = [1, 2, "a"]
```

Tuples are enclosed in parentheses:

```
t = (1, 2, "a")
```

Tuples are faster and consume less memory.

Dictionaries are built with curly brackets:

```
d = {"a":1, "b":2}
```

Q.2 Write about Lists in Python.

Ans. Lists in Python are used to store collection of heterogeneous items These are mutable, which means that you can change their content without changing their identity. You can recognize lists by their square brackets [and] that hold elements, separated by a comma(,). Lists are built into Python: you do not need to invoke them separately.

```
x = [ ] # Empty list
type(x)
list
x1 = [1,2,3]
type(x1)
list
x2 = list([1, 'apple', 3])
type(x2)
list
print(x2[1])
apple
x2[1] = 'orange'
print(x2)
[1, 'orange', 3]
```

Note: like you have seen in the above example with x1, lists can also hold homogeneous items and hence satisfying the storage functionality of an array. This is fine unless you want to apply some specific operations to this collection.

Python provides many methods to manipulate and work with lists. Adding new items to a list, removing some items from a list, sorting or reversing a list are common list manipulations.

list.append(x)

Add an item to the end of the list. Equivalent to a[len (a) :] = [x].

The append() method adds an item to the end of the list. The append() method adds a single item to the existing list. It doesn't return a new list; rather it modifies the original list.

The syntax of append() method is:

```
list.append(item)
```

append() Parameters : The append() method takes a single item and adds it to the end of the list. The item can be numbers, strings, another list, dictionary etc.

Return Value from append() : The append() method only modifies the original list. It doesn't return any value.

list.extend(iterable)

Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

The extend() extends the list by adding all items of a list (passed as an argument) to the end.

The syntax of extend() method is:

```
list1.extend(list2)
```

Here, the elements of 'list2' are added to the end of 'list1'.

extend() Parameters : The extend() method takes a single argument (a list) and adds it to the end.

If you need to add elements of other native datatypes (like tuple and set) to the list, you can simply use:

```
# add elements of a tuple to list
list.extend(list(tuple_type))
```

or even easier

```
list.extend(tuple_type)
```

Return Value from extend() : The extend() method only modifies the original list. It doesn't return any value.

list.insert(i, x)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a. insert(0, x) inserts at the front of the list, and a. insert(len(a), x) is equivalent to a. append(x).

The insert() method inserts an element to the list at a given index. T syntax of insert() method is

list.insert(index, element)

insert() Parameters : The insert() function takes two parameters:

(i) **index** - position where an element needs to be inserted

(ii) **element** - this is the element to be inserted in the list

Return Value from insert() : The insert() method only inserts the el ment to the list. It doesn't return anything; returns 'None'.

list. index(x[, start[, end]])

Return zero-based index in the list of the first item whose value is equ to x. Raises a 'ValueError' if there is no such item.

The optional arguments 'start' and 'end' are interpreted as in the slic notation and are used to limit the search to a particular subsequence of the lis The returned index is computed relative to the beginning of the full sequenc rather than the 'start' argument.

The index() method searches an element in the list and returns its index

In simple terms, index() method finds the given element in a list an returns its position.

However, if the same element is present more than once, index() metho returns its smallest/first position.

Note: Index in Python starts from 0 not 1.

The syntax of index() method for list is:

list. index(element)

index() Parameters : The index method takes a single argument:

(1) **element** - element that is to be searched.

Return value from index() : The index() method returns the index o the element in the list.

If not found, it raises a 'ValueError' exception indicating the element is not in the list.

list. remove(x)

Remove the first item from the list whose value is equal to x. It raises a 'ValueError' if there is no such item.

The remove() method searches for the given element in the list and re moves the first matching element.

The syntax of remove() method is:

list.remove(element)

remove() Parameters : The remove() method takes a single element as an argument and removes it from the list.

If the element(argument) passed to the remove() method doesn't exist, 'valueError' exception is thrown.

Return Value from remove() : The remove() method only removes the given element from the list. It doesn't return any value.

list.pop([i])

Removes the item at the given position in the list, and returns it. If no index is specified, a.pop() removes and returns the last item in the list. The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position.

The pop() method removes the item at the given index from the list. The method also returns the removed item.

The syntax of the pop() method is:

list.pop(index)

pop() parameter : The pop() method takes a single argument (index) and removes the item present at that index.

If the index passed to the pop() method is not in range, it throws IndexError: pop index, out of range 'exception'.

The parameter passed to the pop() method is optional. If no parameter is passed, the default 'index -1' is passed as an argument which returns the last item.

Return Value from pop() : The pop() method returns the item present at the given index. This item will be removed the list.

list.count(x)

Return the number of times x appears in the list.

The count() method returns the number of occurrences of an element in a list.

In simple terms, count() method counts how many times an element has occurred in a list and returns it.

The syntax of count() method is:

list.count(element)

count() Parameters : The count() method takes a single argument:

(1) **element** - element whose count is to be found.

Return value from count() : The count() method returns the number of occurrences of an element in a list.

list.sort(key=None, reverse=False)

Sort the items of the list in place (the arguments can be used for sort customization.

The sort() method sorts the elements of a given list.

The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.

The syntax of sort() method is:

```
list.sort(key=..., reverse=...)
```

Alternatively, you can also use Python's in-built function 'sorted()' for the same purpose.

```
sorted(list, key=..., reverse=...)
```

Note : Simplest difference between sort() and sorted() is: sort() doesn't return any value while, sorted() returns an iterable list.

sort() Parameters : By default, sort() doesn't require any extra parameters. However, it has two optional parameters:

(i) **reverse** - If true, the sorted list is reversed (or sorted in Descending order)

(ii) **key** - function that serves as a key for the sort comparison

Return value from sort() : sort() method doesn't return any value. Rather, it changes the original list.

If you want the original list, use sorted().

list.reverse()

Reverse the elements of the list in place. The reverse() method reverses the elements of a given list.

The syntax of reverse() method is:

```
list.reverse()
```

reverse() parameter : The reverse() function doesn't take any argument.

Return Value from reverse() : The reverse() function doesn't return any value. It only reverses the elements and updates the list.

list.slicing()

The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step).

The slice object is used to slice a given sequence (string, bytes, tuple, list or range) or any object which supports sequence protocol (implements __getitem__() and __len__() method).

Slice object represents the indices specified by range(start, stop, step).

The syntax of slice() are:

```
slice(stop)
slice(start, stop, step)
```


slice() Parameters : slice() mainly takes three parameters which have the same meaning in both constructs:

(i) **start** : starting integer where the slicing of the object starts

(ii) **stop** : integer until which the slicing takes place. The slicing stops at index stop - 1.

(iii) **step** : integer value which determines the increment between each index for slicing If a single parameter is passed, 'start' and 'step' are set to None.

Return value from slice() : slice() returns a slice object used to slice a sequence in the given indices.

List Comprehension

List comprehensions provide us with a simple way to create a list based on some iterable. During the creation, elements from the iterable can be conditionally included in the new list and transformed as needed.

The components of a list comprehension are:

(i) Output Expression (Optional)

(ii) Iterable

(iii) Iterator variable which represents the members of the iterable

We can also create more advanced list comprehensions which include a conditional statement on the iterable.

List Comprehensions vs loops : The list comprehensions are more efficient both computationally and in terms of coding space and time than a for loop. Typically, they are written in a single line of code.

Copying a List : Deep Copy and Shallow copy

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations.

Interface summary:

```
copy.copy(x)
```

Return a shallow copy of x.

```
copy.deepcopy(x)
```

Return a deep copy of x.

```
exception copy.error
```

Raised for module specific errors.

Deep copy is a process in which the copying process occurs recursively. It means first constructing a new collection object and then recursively populat-

ing it with copies of the child objects found in the original. In case of deep copy a copy of object is copied in other object. It means that any changes made to copy of object do not reflect in the original object. In python, this is implemented using "deepcopy()" function.

A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original. The copying process does not recurse and therefore won't create copies of the child objects themselves. In case of shallow copy, a reference of object is copied in other object. It means that any changes made to a copy of object do reflect in the original object. In python, this is implemented using "copy()" function.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

(i) A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

(ii) A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

(i) Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.

(ii) Because deep copy copies may copy everything too much, such as data which is intended to be shared between copies.

The deepcopy() function avoids these problems by:

(i) keeping a "memo" dictionary of objects already copied during the current copying pass; and

(ii) letting user-defined classes override the copying operation or the set of components copied.

Q.3 Write about Tuples in Python.

Ans. A tuple is a data structure that is an immutable, or unchangeable, ordered sequence of elements. Because tuples are immutable, their values cannot be modified.

A tuple in Python looks like this:

coral = ('blue coral', 'staghorn coral', 'pillar coral', 'elkhorn coral')

Tuples are used for grouping data. Each element or value that is inside of a tuple is called an item.

Tuples have values between parentheses ( ) separated by commas,. Empty tuples will appear as coral = (), but tuples with even one value must use a comma as in coral = ('blue coral',).

If we print() the tuple above, we'll receive the following output, with the tuple still typed by parentheses:

```
print(coral)
Output
('blue coral', 'staghorn coral', 'pillar coral', 'elkhorn coral')
```

When thinking about Python tuples and other data structures that are types of collections, it is useful to consider all the different collections you have on your computer: your assortment of files, your song playlists, your browser bookmarks, your emails, the collection of videos you can access on a streaming service, and more.

Tuples are similar to lists, but their values can't be modified. Because of this, when you use tuples in your code, you are conveying to others that you don't intend for there to be changes to that sequence of values.

Additionally, because the values do not change, your code can be optimized through the use of tuples in Python, as the code will be slightly faster for tuples than for lists.

Indexing Tuples

As an ordered sequence of elements, each item in a tuple can be called individually, through indexing. Each item corresponds to an index number, which is an integer value, starting with the index number 0. For the 'coral' tuple, the index breakdown looks like this:

'blue coral'	'staghorn coral'	'pillar coral'	'elkhorn coral'
0	1	2	3

The first item, the string 'blue coral' starts at index 0, and the list ends at index 4 with the item 'elkhorn coral'.

Because each item in a Python tuple has a corresponding index number, we're able to access items.

Now we can call a discrete item of the tuple by referring to its index number:

```
print(coral[2])
Output
pillar coral
```

The index numbers for this tuple range from 0-3, as shown in the table below. So to call any of the items individually, we would refer to the index numbers like this:

```
coral[0] = 'blue coral'
coral[1] = 'staghorn coral'
coral[2] = 'pillar coral'
```

coral[3] = 'elkhorn coral'

If we call the tuple 'coral' with an index number of any that is greater than 3, it will be out of range as it will not be valid:

```
print(coral[22])
Output
IndexError: tuple index out of range
```

In addition to positive index numbers, we can also access items from the tuple with a negative index number, by counting backwards from the end of the tuple, starting at -1. This is especially useful if we have a long tuple and we want to pinpoint an item towards the end of a tuple.

For the same tuple 'coral', the negative index breakdown looks like this:

'blue coral'	'staghorn coral'	'pillar coral'	'elkhorn coral'
-4	-3	-2	-1

So, if we would like to print out the item 'blue coral' by using its negative index number, we can do so like this:

```
print(coral[-4])
```

Output :

```
blue coral
```

We can concatenate string items in a tuple with other strings using the + operator:

```
print('This reef is made up of' + coral[1])
```

Output :

```
This reef is made up of staghorn coral
```

We were able to concatenate the string item at index number 0 with the string 'This reef is made up of '. We can also use the + operator to concatenate 2 or more tuples together.

With index numbers that correspond to items within a tuple, we're able to access each item of a tuple discretely.

Python Tuple count()

The count() method returns the number of occurrences of an element in a tuple.

In simple terms, count() method searches the given element in a tuple and returns how many times the element has occurred in it.

The syntax of count() method is:

```
tuple.count(element)
```

count() Parameters : The count() method takes a single parameter:

(1) **element** : element whose count is to be found

Return value from count() : count() method returns the number of occurrences of a given element in the tuple.

Tuples Usage

(i) **Tuples are used for grouping data :** We could group together pairs of values by surrounding with parentheses. A tuple can be used to group any number of items into a single compound value. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
>>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress",
"Atlanta, Georgia")
```

Tuples are useful for representing what other languages often call records - some related information that belongs together, like your student record. There is no description of what each of these fields means, but we can guess. A tuple lets us "chunk" together related information and use it as a single thing.

Tuples support the same sequence operations as strings. The index operator selects an element from a tuple.

```
>>> julia[2]
1967
```

But if we try to use item assignment to modify one of the elements of the tuple, we get an error:

```
>>> julia[0] = "X"
TypeError: 'tuple' object does not support item assignment.
```

So like strings, tuples are immutable. Once Python has created a tuple in memory, it cannot be changed.

Of course, even if we can't modify the elements of a tuple, we can always make a new tuple holding different information. To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So if 'julia' has a new recent film, we could change her variable to reference a new tuple that used some information from the old one:

```
>>> julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
>>> julia
("Julia", "Roberts", 1967, "Eat Pray Love", 2010, "Actress", "Atlanta, Georgia")
```

To create a tuple with a single element, we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses:

```
>>> tup = (5,)
>>> type (tup)
```

```
<class 'tuple'>
>>> x = (5)
>>> type(x)
<class 'int'>
```

Use of Tuples as a Swap Function

Tuple can be used as swap function. Once in a while, it is useful to swap he values of two variables.

This code reverses the contents of 2 variables x and y:

```
>>> (x, y) = (y, x)
```

Consider the following function:

```
def swap (a, b):
    (b, a) = (a, b)
then:
    a = 2
    b = 3
    swap (a, b)
```

#a is still 2 and b still 3 !! a and b are indeed passed by value not reference.

With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

If we have to do this often, this approach becomes cumbersome. Python provides a form of 'tuple assignment' that solves this problem neatly:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b, c, d = 1, 2, 3
```

ValueError: unpack tuple of wrong size

Q.4 Write about Dictionary in Python.

Ans. A dictionary object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

Another useful data type built into Python is the dictionary. Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

Python's dictionaries are kind of hash - table type. They work like associative arrays or hashes found in Perl and consist of key - value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the 'in' keyword.

The dict() constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionary : Keys

The keys() method returns a view object that displays a list of all t keys in the dictionary

The syntax of keys() is:

```
dict.keys()
```

keys() Parameters : The keys() doesn't take any parameters.

Return Value from keys() : The keys() returns a view object that displa a list of all the keys. When the dictionary is changed, the view object also reflec these changes.

Example : How keys() works?

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}
print(person.keys())
empty_dict = {}
print(empty_dict.keys())
```

When you run the program, the output will be:

```
dict_keys(['name', 'salary', 'age'])
dict_keys()
```

Dictionary : values

The values() method returns a view object that displays a list of all the values in the dictionary.

The method values() returns a list of all the values available in a given dictionary.

Syntax : Following is the syntax for values() method-

```
dict.values()
```

values() Parameters : The values() method doesn't take any parameters.

Return Value : This method returns a list of all the values available in a given dictionary.

Nested Dictionary

In Python, a nested dictionary is a dictionary inside a dictionary. It's a collection of dictionaries into one single dictionary.

```
nested_dict = { 'dictA': {'key_1': 'value_1'},
                'dictB': {'key_2': 'value_2'}}
```

Here, the 'nested_dict' is a nested dictionary with the dictionary 'dictA' and 'dictB'. They are two dictionary each having own key and value.

Create a Nested Dictionary : We're going to create dictionary of people within a dictionary.

Example : How to create a nested dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
print(people)
```

When we run above program, it will output:

{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}

In the above program, 'people' is a nested dictionary. The internal dictionary 1 and 2 is assigned to 'people'. Here, both the dictionary have key 'name, age, sex' with different values. Now, we print the result of 'people'.

Access elements of a Nested Dictionary : To access element of a nested dictionary, we use indexing [ ] syntax in Python.

Exp. : Access the elements using the [] syntax

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
print(people[1]['name'])
print(people[1]['age'])
print(people[1]['sex'])
```

When we run above program, it will output:

```
John
27
Male
```

In the above program, we print the value of key 'name' using i.e. 'people[1]['name']' from internal dictionary 1. Similarly, we print the value of 'age' and 'sex' one by one.

Add element to a Nested Dictionary

Exp : Add elements in a nested dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
people[3] = {}
people[3]['name'] = 'Luna'
people[3]['age'] = '24'
people[3]['sex'] = 'Female'
people[3]['married'] = 'No'
print(people[3])
```

When we run above program, it will output:

{'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'}

In the above program, we create an empty dictionary 3 inside the dictionary 'people'.

Then, we add the 'key:value' pair i.e 'people[3]['Name'] = 'Luna'' inside the dictionary 3. Similarly, we do this for key 'age, sex and married' one by one. When we print the people[3], we get 'key:value' pairs of dictionary 3

Deleting elements from a Nested Dictionary : In Python, we use "del" statement to delete elements from nested dictionary.

Exp : Deleting elements from a nested dictionary

```
2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'},
4: {'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}}
    del people[3]['married']
    del people[4]['married']
    print(people[3])
    print(people[4])
```

When we run above program, it will output:

```
{'name': 'Luna', 'age': '24', 'sex': 'Female'}
{'name': 'Peter', 'age': '29', 'sex': 'Male'}
```

In the above program, we delete the 'key:value' pairs of 'married' from internal dictionary 3 and 4. Then, we print the 'people[3] 'and' people[4]' to confirm changes.

Exp : Deleting dictionary from a nested dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female'},
          4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}
del people[3], people[4]
print(people)
```

When we run above program, it will output:

```
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie',
'age': '22', 'sex': 'Female'}}
```

In the above program, we delete both the internal dictionary 3 and 4 using 'del' from the nested dictionary 'people'. Then, we print the nested dictionary 'people' to confirm changes.

Iterating Through a Nested Dictionary : Using the for loops, we can iterate through each element in a nested dictionary.

Exp : iterating through a Nested dictionary

script.py


```
people = {1: {'Name': 'John', 'Age': '27', 'Sex': 'Male'},
          2: {'Name': 'Marie', 'Age': '22', 'Sex': 'Female'}}
for p_id, p_info in people.items():
    print("\nPerson ID:", p_id)
    for key in p_info:
    print(key + ':', p_info[key])
```

When we run above program, it will output:

```
Person ID: 1
Name: John
Age: 27
Sex: Male
Person ID: 2
Name: Marie
Age: 22
Sex: Female
```

In the above program, the first loop returns all the keys in the nested dictionary 'people'. It consist of the IDs 'p_id' of each person. We use these IDs to unpack the information 'p_info' of each person.

The second loop goes through the information of each person. Then, it returns all of the keys 'name, age, sex' of each person's dictionary.

Now, we print the key of the person's information and the value for that key.

Key Points to Remember

1. Nested dictionary is an unordered collection of dictionary
2. Slicing Nested Dictionary is not possible.
3. We can shrink or grow nested dictionary as need.
4. Like Dictionary, it also has key and value.
5. Dictionary is accessed using key.

Dictionary Comprehension

Dictionary comprehension is a method for transforming one dictionary into another dictionary. During this transformation, items within the original dictionary can be conditionally included in the new dictionary and each item can be transformed as needed.

A good list comprehension can make your code more expressive and thus, easier to read.

The way to do dictionary comprehension in Python is to be able to access the 'key' objects and the 'value' objects of a dictionary.

You can use the built-in methods for the same:

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
# Put all keys of 'dict1' in a list and returns the list
dict1.keys()
dict_keys(['c', 'd', 'a', 'b'])
# Put all values saved in 'dict1' in a list and returns the list
dict1.values()
dict_values([3, 4, 1, 2])
```

So, now that you know how to access all the keys and their values in a dictionary. You can also access each key-value pair within a dictionary using the 'items()' method:

```
dict1.items()
dict_items([('c', 3), ('d', 4), ('a', 1), ('b', 2)])
```

This is the general template you can follow for dictionary comprehension in Python:

```
dict_variable = {key:value for (key,value) in dictonary.items()}
```

This can serve as the basic and the most simple template. This can get more and more complex as you add conditionalities to it.

Why Use Dictionary Comprehension? Dictionary comprehension is a powerful concept and can be used to substitute 'for' loops and 'lambda' functions. However, not all 'for loop' can be written as a dictionary comprehension but all 'dictionary comprehension' can be written with a 'for loop'.

Q.5 Write about Strings in Python.

Ans. In computer programming language, strings are nothing but a single dimension array of charateers. You can define a string in easy words as "a sequence of characters".

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example -

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings : Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example -

```
#!/usr/bin/python
var1 = 'Hello World!'
```



```
var2 = "Python Programming"
print "var1[0]:", var1[0]
print "var2[1:5]:", var2[1:5]
```

When the above code is executed, it produces the following result -

```
var1[0]: H
var2[1:5]: ytho
```

Updating Strings : You can "update" an existing string by assigning or reassigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example -

```
#!/usr/bin/python
var1 = 'Hello World!'
print "Updated String :-" , var1[:6] + 'Python'
```

When the above code is executed, it produces the following result -

```
Updated String :- Hello Python
```

Single Line and Multi Line String

In computer programming language, strings are nothing but a single dimension array of characters. You can define a string in easy words as "a sequence of characters". In Python, a string can be categorized as

1. A single Line string and
2. A multiline string

To work with Single Line String in Python Programming : In Python programming language there are multiple ways of handling a string. Among those the most common ways are:

1. Enclose the string in Double Quotes and
2. Enclose the string in Single Quotes

Apart from these two options, Python Interpreter provides us one more way which is for handling multi-line string.

How Python Interpreter Processes a String? : Whenever Python interpreter starts processing a string it looks for a quotation mark, it could either be a single quote or double quote. The opening quote indicates the starting of the string and closing quote indicates the end of the string. If either of these goes missing then it will raise an alarm which will end up in an error.

When To Use Single or Double Quotes? : In such scenario where you want to print a string which contains punctuation marks like double quotes ("...") then always enclose your string in a pair of single quotes. That way the opening single quote will indicate the starting of the string and the interpreter will look for the closing single quote sign, which will indicate the end of your string. Any punctuation marks that comes in the middle will be bypassed.

Multiline Strings with Triple Quotes

While you can use the \n escape character to put a newline into a string, it is often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks do not apply to lines inside a multiline string.

Ordinarily, strings do not extend across multiple lines, causing an error if you try.

But, starting and ending a string """ or ''' tells Python to allow the string to cross multiple lines.

Any character other than ''' (or """ , if that is how the string started) is allowed inside the string.

Example,

```
>>> s1 = """ This
is a multi-line
string."""
>>> s1
'This\nis a multi-line\nstring.'
>>> print s1
This
is a multi-line
string.
>>>
```

Notice the \n when we ask Python for the value of the string (instead of printing it). This is an 'escape character',

Multiline Comments : While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines. The following is perfectly valid Python code:

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com
This program was designed for Python 3, not Python 2.
"""
def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```


Formatter

The built-in string class provides the ability to do complex variable substitutions and value formatting via the 'format()' method. The 'Formatter' class in the 'string' module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in 'format()' method.

class : string.Formatter

The 'Formatter' class has the following public methods:

(1) **format(format_string, *args, **kwargs) :** format( ) is the primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. 'format()' is just a wrapper that calls 'vformat()'.

(2) **vformat(format_string, args, kwargs) :** This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the '*args 'and' **kwargs' syntax. 'vformat()' does the work of breaking up the format string into character data and replacement fields.

In addition, the 'Formatter' defines a number of methods that are intended to be replaced by subclasses:

(1) **parse(format_string) :** Loop over the format_string and return an iterable of tuples (literal_text, field_name, format_spec, conversion). This is used by 'vformat()' to break the string into either literal text, or replacement fields.

(2) **get_field(field_name, args, kwargs) :** Given field_name as returned by parse(), convert it to an object to be formatted. Returns a tuple (obj, used_key). The default version takes strings of the form, such as "0[name]" or "label.title". args and kwargs are as passed in 'to vformat()'. The return value used_key has the same meaning as the 'key' parameter to 'get_value()'.

(3) **get_value(key, args, kwargs) :** Retrieve a given field value. The 'key' argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in args; if it is a string, then it represents a named argument in 'kwargs'.

The 'args' parameter is set to the list of positional arguments to 'vformat()', and the 'kwargs' parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; Subsequent components are handled through normal attribute and indexing operations.

If the index or keyword refers to an item that does not exist, then an 'IndexError' or 'KeyError' should be raised.

(4) check_unused_args(used_args, args, kwargs) : Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the 'args' and 'kwargs' that was passed to 'vformat'. The set of unused args can be calculated from these parameters 'check_unused_args()' is assumed to raise an exception if the check fails.

(5) format_field(value, format_spec) : 'format_field()' simply calls the global 'format()' built-in. The method is provided so that subclasses can override it.

(6) convert_field(value, conversion) : Converts the value (returned by get_field()) given a conversion type (as in the tuple returned by the 'parse()' method). The default version understands 's' (str), 'r' (repr) and 'a' (ascii) conversion types.

isdigit

The isdigit() method returns True if all characters in a string are digits. If not, it returns False.

The syntax of isdigit() is

string.isdigit()

isdigit() Parameters : The isdigit() doesn't take any parameters.

Return Value from isdigit() : The isdigit() returns:

(i) True if all characters in the string are digits.

(ii) False if at least one character is not a digit.

Example : Working of isdigit()

```
s = "28212"
print(s.isdigit())
# contains alphabets and spaces
s = "Mo3 nicaG el l22er"
print(s.isdigit())
```

When you run the program, the output will be:

```
True
False
```

A digit is a character that has property value:

(i) Numeric_Type=Digit

(ii) Numeric_Type=Decimal

In Python, superscript and subscripts (usually written using unicode) are also considered digit characters. Hence, if the string contains these characters along with decimal characters, isdigit() returns True. The roman numerals,

currency numerators and fractions (usually written using unicode) are considered numeric characters but not digits. The isdigit() returns False if the string contains these characters.

To check whether a character is numeric character or not, you can use isnumeric() method.

isalpha

In Python, isalpha() is a built-in method used for string handling.

The isalpha() method returns "True" if all characters in the string are alphabets, Otherwise, It returns "False".

This function is used to check if the argument contains any alphabets characters such as :

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

Syntax : string.isalpha()

Parameters: isalpha() does not take any parameters.

Returns :

1. True- If all characters in the string are alphabet.
2. False- If the string contains 1 or more non-alphabets.

Errors and Exceptions

1. It contains no arguments, therefore an error occurs if a parameter is passed.
2. Both uppercase and lowercase alphabets return "True".
3. Space is not considered to be alphabet, therefore it returns "False".

Application : Given a string in python, count number of alphabets in the string and print the alphabets.

isalnum

isalnum() function in Python programming language checks whether all the characters in a given string is alphanumeric or not. The isalnum() method returns True if all characters in the string are alphanumeric (either alphabets or numbers). If not, it returns False.

Alphanumeric: A character that is either a letter or a number

Syntax: string_name.isalnum()

string_name is the name of the string

Parameter: isalnum() method takes no parameters

Return Value from isalnum() : The isalnum() returns:

(i) **True:** If all the characters are alphanumeric

(ii) **False:** If one or more characters are not alphanumeric

islower

Python islower is one of the Python String Methods which is used to check whether the given string has at least one character, and the character is

either in lowercase, or not. If it is lower then the function will return True otherwise, it returns False. The method islower() checks whether all the case-based characters (letters) of the string are lowercase.

Syntax : Following is the syntax for islower() method-

str.islower()

Parameters: islower() does not take any parameters

Returns :

1. **True** - If all characters in the string are lower.
2. **False** - If the string contains 1 or more non-lowercase characters.

Examples:

Input : string = 'geeksforgeeks'

Output : True

Input : string = 'GeeksforGeeks'

Output : False

Errors And Exceptions

1. It returns "True" for whitespaces
2. It does not take any arguments, Therefore, It returns an error if a parameter is passed.
3. Digits and symbols return "True", Only a lowercase letter returns "false".

istitle

Titlecased String has the first character in each word Uppercase and remaining all characters Lowercase alphabets. The method istitle() checks whether all the case-based characters in the string following non-case-based letters are uppercase and all other case-based characters are lowercase.

Syntax

Following is the syntax for istitle() method -

str.istitle( )

Parameters : istitle( ) does not take any parameters

Return Value : This method returns true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns false otherwise.

Example : The following example shows the usage of istitle() method.

```
#!/usr/bin/python
str = "This Is String Example...Wow!!!";
print str.istitle()
str = "This is string example....wow!!!";
print str.istitle()
```

When we run above program, it produces following result -

```
True
False
```


isspace

The method isspace() checks whether the string consists of whitespace.

Syntax : Following is the syntax for isspace() method-

str.isspace()

Parameters : Isspace() does not take any parameters

Return Value : This method returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

Example : The following example shows the usage of isspace() method.

```
#!/usr/bin/python
str ="          ";
print str.isspace()
str = "This is string example....wow!!!";
print str.isspace()
```

When we run above program, it produces following result -

```
True
False
```

title

Title function in python is the Python String Method which is used to convert the first character in each word to Uppercase and remaining characters to Lowercase in string and returns new string. The method title() returns a copy of the string in which first characters of all the words are capitalized.

Syntax : Following is the syntax for title() method -

str.title();

Parameters : title() does not take any parameters

Return Value : This method returns a copy of the string in which first characters of all the words are capitalized.

Example : The following example shows the usage of title() method.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print str.title()
```

When we run above program, it produces following result -

```
This Is String Example....Wow!!!
```

lower

In Python, lower() is a built-in method used for string handling.

The lower() method returns the lowercased string from the given string. It converts all uppercase characters to lowercase. If no uppercase characters exist, it returns the original string.

Syntax : string.lower()

Parameters: lower() does not take any parameters

Returns : It converts the given string in into lowercase and returns the string.

Examples:

Input : string = 'GEEKSFORGEEKS'
Output : geeksforgeeks
Input : string = 'GeeksforGeeks'
Output : geeksforgeeks

Errors And Exceptions

1. It does not take any arguments, Therefore, It returns an error if a parameter is passed.
2. Digits and symbols return are returned as it is, Only an uppercase letter is returned after converting to lowercase.

upper

In Python, upper() is a built-in method used for string handling.

The upper() method returns the uppercased string from the given string. It converts all lowercase characters to uppercase. If no lowercase characters exist, it returns the original string.

Syntax : string.upper()

Parameters: upper() does not take any parameters

Returns : It converts the given string in into uppercase and returns the string.

Examples:

Input : string = 'geeksforgeeks'
Output : GEEKSFORGEEKS
Input : string = 'My name is ayush'
Output : MY NAME IS AYUSH

Errors And Exceptions

1. It does not take any arguments, Therefore, It returns an error if a parameter is passed.

2. Digits and symbols return are returned as it is, Only a lowercase letter is returned after converting to uppercase.

strip

The strip() method returns a copy of the string with both leading and trailing characters removed (based on the string argument passed). The strip() removes characters from both left and right based on the argument (a string specifying the set of characters to be removed). The method strip() returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

Syntax : Following is the syntax for strip() method -

```
str.strip([chars]) ;
```

Parameters :

Chars : The characters to be removed from beginning or end of the string.

Return Value : This method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string.

Example : The following example shows the usage of strip() method.

```
#!/usr/bin/python
str= "0000000this is string example.... wow!!!0000000";
print str.strip( '0' )
```

When we run above program, it produces following result -

```
this is string example....wow!!!
```

split()

split() method returns a list of strings after breaking the given string by the specified separator. The method split() returns a list of all the words in the string, using 'str' as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to num.

Syntax : Following is the syntax for split() method -

```
str.split(str=" ", num=string.count(str)).
```

Parameters

1. **str** - This is any delimeter, by default it is space.
2. **num** - this is number of lines minus one

Return Value : This method returns a list of lines.

Example : The following example shows the usage of split() method.

```
#!/usr/bin/python
str= "Line1-abcdef \nLine2-abc \nLine4-abcd";
print str.split( )
```

```
print str.split(' ', 1 )
```

When we run above program, it produces following result -

```
['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
['Line1-abcdef', '\nLine2-abc \nLine4-abcd']
```

splitlines

splitline() method is used to split the lines at line boundaries. The function returns a list of lines in the string, including the line break(optional). The method splitlines() returns a list with all the lines in string, optionally including the line breaks (if num is supplied and is true)

Syntax : Following is the syntax for splitlines() method -

```
str.splitlines( num=string.count('\n'))
```

Parameters

1. **num** - This is any number, if present then it would be assumed that line breaks need to be included in the lines.

2. **keepends** (optional): When set to True, line breaks are included in the resulting list. This can be a number, specifying the position of line break or, can be any Unicode characters, like "\n", "\r", "\r\n", etc as boundaries for strings.

Return Value : This method returns true if found matching string otherwise false.

Example : The following example shows the usage of splitlines() method.

```
#!/usr/bin/python
str = "Line1-a b c d e f\nLine2- a b c\n\nLine4- a b c d";
print str.splitlines( )
print str.splitlines( 0 )
print str.splitlines( 3 )
print str.splitlines( 4 )
print str.splitlines( 5 )
```

When we run above program, it produces following result -

```
['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
```

join

The join() method is a string method and returns a string in which the elements of sequence have been joined by str separator. The method join() returns a string in which the string elements of sequence have been joined by str separator.

Syntax : Following is the syntax for join() method -

```
str.join(sequence)
```

Parameters

1. **sequence** - This is a sequence of the elements to be joined.

2. The join() method takes iterable - objects capable of returning its members one at a time. Some examples are List, Tuple, String, Dictionary and Set

Return Value : This method returns a string, which is the concatenation of the strings in the sequence seq. The separator between elements is the string providing this method.

Type Error : If the iterable contains any non-string values, it raises a TypeError exception.

Example : The following example shows the usage of join() method.

```
#!/usr/bin/python
s = "-";
seq = ("a", "b", "c"); # This is sequence of strings.
print s.join( seq )
```

When we run above program, it produces following result -

```
a-b-c
```

Q.6 Write about Sets in Python.

Ans. Mathematically a set is a collection of items not in any particular order. A Python set is similar to this mathematical definition with below additional conditions.

(i) The elements in the set cannot be duplicates.

(ii) The elements in the set are immutable(cannot be modified) but the set as a whole is mutable.

(iii) There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it's elements and carry out these mathematical operations.

(1) **Creating a set :** A set is created by using the set() function or placing all the elements within a pair of curly braces.

```
Days=set(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
Months={"Jan", "Feb", "Mar"}
Dates={21,22,17}
```

```
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

(2) **Accessing Values in a Set :** We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
for d in Days:
    print(d)
```

When the above code is executed, it produces the following result.

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

Sets : Union

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element "Wed" is present in both the sets. Returns a union of two sets. Using the '|' operator between 2 sets is the same as writing set1.union(set2)

```
people = {"Jay", "Idrish", "Archil"}
vampires = {"Karan", "Arjun"}
population = people.union(vampires)
OR
population = people|vampires
```

population set will have components of both people and vampire

Sets : intersection

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element "Wed" is present in both the sets. It returns an intersection of two sets. The

'&' operator can also be used in this case.

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one "wed".

```
set(['Wed'])
```

Sets : subset

The issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

Set A is said to be the subset of set B if all elements of A are in B.

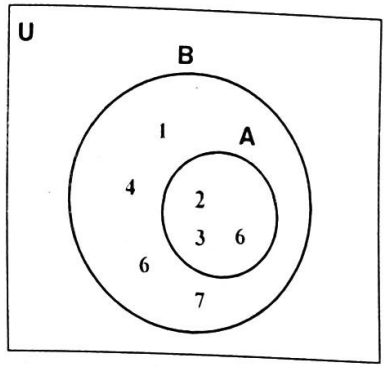


Fig.

Here, set A is a subset of B.

The syntax of issubset() is:

```
A.issubset(B)
```

Return Value from issubset() : The issubset() returns

(i) True if A is a subset of B

(ii) False if A is not a subset of B

Exp :

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}
C = {1, 2, 4, 5}
# Returns True
print(A.issubset(B))
# Returns False
```

```
# B is not subset of A
print(B.issubset(A))
# Returns False
print(A.issubset(C))
# Returns True
print(C.issubset(B))
```

When you run the program, the output will be:

```
True
False
False
True
```

Sets : superset

The issuperset() method returns True if a set has every elements of another set (passed as an argument). If not, it returns False.

Set A is said to be the superset of set B if all elements of B are in A.

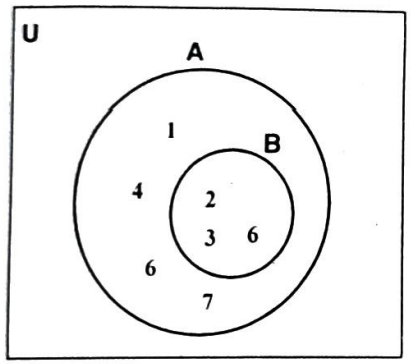


Fig.

Here, set A is a superset of set B and set B is a subset of set A.

The syntax of issuperset() is:

```
A.issuperset(B)
```

Return Value from issuperset() : The issuperset() returns

(i) True if A is a superset of B

(ii) False if A is not a superset of B

Exp :

```
A = {1, 2, 3, 4, 5}
B = {1, 2, 3}
C = {1, 2, 3}
```



```
# Returns True
print(A.issuperset(B))
# Returns False
print(B.issuperset(A))
# Returns True
print(C.issuperset(B))
```

When you run the program, the output will be:

```
True
False
True
```

Sets : difference

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element "Wed" is present in both the sets so it will not be found in the result set. It returns a set containing all the elements of invoking set but not of the second set. We can use '-' operator here.

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has none "wed".

```
set(['Mon', 'Tue'])
```

Sets : symmetric difference

This in-built function of Python Set helps us to get the symmetric difference between two sets, which is equal to the elements present in either of the two sets, but not common to both the sets. Let's look at the Venn diagram of the symmetric_difference between two sets.

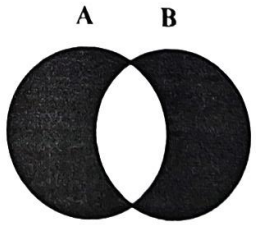


Fig.

Symmetric Difference is marked as dark.

If there is a set_A and set_B, then the symmetric difference between them will be equal to the union of set_A and set_B without the intersection between the two.

```
// Takes a single parameter that has to be
// a set and returns a new set which is the
// symmetric difference between the two sets.
set_A.symmetric_difference(set_B)
```

Exp :

Input: set_A = {1, 2, 3, 4, 5}
set_B = {6, 7, 3, 9, 4}
Output : {1, 2, 5, 6, 7, 9}

Explanation: The common elements {3, 4} are discarded from the output.

Sets : copy

The copy() method returns a shallow copy of the set in python. If we use "=" to copy a set to another set, when we modify in the copied set, the changes are also reflected in the original set. So we have to create a shallow copy of the set such that when we modify something in the copied set, changes are not reflected back in the original set.

Syntax: set_name.copy()

set_name: Name of the set whose copy we want to generate.

Parameters: The copy() method for sets doesn't take any parameters.

Return value: The function returns a shallow copy of the original set.

Below is the implementation of the above function:

```
# Python3 program to demonstrate the use
# of join () function
set1 = {1, 2, 3, 4}
# function to copy the set
set2 = set1. copy ()
# prints the copied set
print(set2)
```

Output:

{1, 2, 3, 4}

Sets : add

We can add elements to a set by using add() method. This adds the item x to set if it is not already present in the set.

```
Days=set(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat"])
Days.add("Sun")
```



```
print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Sets : remove

We can remove elements from a set by using discard() method.

```
Days=set(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat"])
Days.discard("Sun")
print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Sets : discard

The built-in method, discard() in Python, removes the element from the set only if the element is present in the set. If the element is not present in the set, then no error or exception is raised and the original set is printed.

If the element is present in the set:

```
# Python program to remove random elements of choice
# Function to remove elements using discard()
def Remove(sets):
    sets.discard(20)
    print (sets)
# Driver Code
sets = set([10, 20, 26, 41, 54, 20])
Remove(sets)
```

Output:

{41, 10, 26, 54}

If the element is not present in the set:

```
# Python program to remove random elements of choice
# Function to remove elements using discard()
def Remove(sets):
    sets.discard(21)
    print (sets)
# Driver Code
sets = set([10, 20, 26, 41, 54, 20])
Remove(sets)
```

Output:

{41, 10, 26, 20, 54}

Q.7 Write about Inbuilt Functions in Python.

Ans. A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.

Following are main in built functions in Python:

(1) **Id :** The id() function returns a unique id for the specified object. All objects in Python has its own unique id. The id is assigned to the object when it is created. The id is the object's memory address, and will be different for each time you run the program. (except for some object that has a constant unique id, like integers from -5 to 256)

Syntax : id(object)

Parameter Values :

Parameter	Description
object	Any object, String, Number, List, Class etc.

(2) **len :** The len ( ) function returns the number of items in an object. When the object is a string, the len() function returns the number of characters in the string.

Syntax : len(object)

Parameter Values :

Parameter	Description
object	Required. An object. Must be a sequence or a collection

Example : Return the number of characters in a string:

```
mylist = "Hello"
x = len(mylist)
```

(3) **chr :** The chr() function returns the character that represents the specified unicode.

Syntax : chr(number)

Parameter Values :

Parameter	Description
number	An integer representing a valid Unicode code point

(4) **ord :** The ord () function returns the number representing the unicode code of a specified character.

Syntax : ord(character)

Parameter Values :

Parameter	Description
character	String, any character

Python Function Types

Python comes preloaded with many function-definition that you can use as per your needs. You can even create new functions. Broadly, Python functions can belong to one of the following three categories :

1. **Built-in functions** : These are pre-defined functions and are always available for use. Some of them are - len( ), type( ), int( ), input( ) etc.
2. **Functions defined in modules** : These functions are pre-defined in particular modules and can only be used when the corresponding module is imported.
3. **User defined functions** : These are defined by the programmer. As programmers you can create your own functions.

Python Built-In Functions

1. **abs() :** The abs() is one of the most popular Python built-in functions, which returns the absolute value of a number.
2. **all() :** The all() function takes a container as an argument. This Built in Function returns True if all values in a python iterable have a Boolean value of True. An empty value has a Boolean value of False.
3. **any() :** Like all(), it takes one argument and returns True if, even one value in the iterable has a Boolean value of True.
4. **ascii() :** It returns a printable representation of a python object (like a string or a Python list).
5. **bin() :** bin() converts an integer to a binary string.
6. **bool() :** bool() converts a value to Boolean.
7. **bytearray() :** bytearray() returns a python array of a given byte size.
8. **bytes() :** bytes() returns an immutable bytes object.
9. **callable() :** callable() tells us if an object can be called.
10. **chr() :** chr() Built In function returns the character in python for an ASCII value.

11. **classmethod()** : classmethod() returns a class method for a given method.

12. **compile()** : compile() returns a Python code object. We use this Python in built function to convert a string code into object code.

13. **complex()** : complex() function creates a complex number.

14. **delattr()** : delattr() takes two arguments- a class, and an attribute in it. It deletes the attribute.

15. **dict()** : dict(), creates a python dictionary.

16. **dir()** : dir() returns an object's attributes.

17. **divmod()** : divmod(), takes two parameters, and returns a tuple of their quotient and remainder. In other words, it returns the floor division and the modulus of the two numbers.

18. **enumerate()** : This Python Built In function returns an enumerate object. In other words, it adds a counter to the iterable.

19. **eval()** : This Function takes a string as an argument, which is parsed as an expression.

20. **exec()** : exec() runs Python code dynamically.

21. **filter()** : filter() filters out the items for which the condition is True.

22. **float()** : This Python Built In function converts an int or a compatible value into a float.

23. **format()** : The format() function formats a specified value into a specified format.

24. **frozenset()** : frozenset() returns an immutable frozenset object.

25. **getattr()** : getattr() returns the value of an object's attribute.

26. **globals()** : This Python built-in function, returns a dictionary of the current global symbol table.

27. **hasattr()** : Like delattr() and getattr(), hasattr() Python built-in function, returns True if the object has that attribute.

28. **hash()** : hash() function returns the hash value of an object. And in Python, everything is an object.

29. **help()** : To get details about any module, keyword, symbol, or topic, we use the help() function.

30. **hex()** :hex() Python built-in function, converts an integer to hexadecimal.

31. **id()** : id() returns an object's identity.

32. **input()** : Input() Python built-in function, reads and returns a line of string.

33. **int()** : int() converts a value to an integer.

34. **isinstance()** : isinstance() takes a variable and a class as arguments. Then, it returns True if the variable belongs to the class. Otherwise, it returns False.

35. **issubclass()** : This Python Built In function takes two arguments- two python classes. If the first class is a subclass of the second, it returns True. Otherwise, it returns False.

36. **iter()** : Iter() Python built-in function, returns a python iterator for an object.

37. **len()** : len() returns the length of an object.

38. **list()** : list() creates a list from a sequence of values.

39. **locals()** : This function returns a dictionary of the current local symbol table.

40. **map()** : Like filter(), map() Python built-in function, takes a function and applies it on an iterable. It maps True or False values on each item in the iterable.

41. **max()** : max() returns the item, in a sequence, with the highest value of all.

42. **memoryview()** : memoryview() shows us the memory view of an argument.

43. **min()** : min() returns the lowest value in a sequence.

44. **next()** : This Python Built In function returns the next element from the iterator.

45. **object()** : object() Python built-in function, creates a featureless object.

46. **oct()** : oct() converts an integer to its octal representation.

47. **open()** : The open() function opens a file, and returns it as a file object. open() lets us open a file.

48. **ord()** : The function ord() returns an integer that represents the Unicode point for a given Unicode character.

49. **pow()** : pow() takes two arguments- say, x and y. It then returns the value of x to the power of y.

50. **print()** : The print() function prints the specified message to the screen or other standard output device.

51. **property()** : The function property() returns a property attribute.

52. **range()** : The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default) and ends at a specified number.

53. **repr()** : repr() returns a representable string of an object.

54. **reversed() :** This function reverses the contents of an iterable and returns an iterator object.

55. **round() :** round() rounds off a float to the given number of digits (given by the second argument).

56. **set() :** set() returns a set of the items passed to it.

57. **setattr() :** Like getattr(), setattr() sets an attribute's value for an object.

58. **slice() :** slice() returns a slice object that represents the set of indices specified by range(start, stop, step).

59. **sorted() :** sorted() prints out a sorted version of an iterable. It does not, however, alter the iterable.

60. **staticmethod() :** staticmethod() creates a static method from a function. A static method is bound to a class rather than to an object. But it can be called on the class or on an object.

61. **str() :** str() takes an argument and returns the string equivalent of it.

62. **sum() :** The function sum() takes an iterable as an argument, and returns the sum of all values.

63. **super() :** super() returns a proxy object to let you refer to the parent class.

64. **tuple() :** The function tuple() lets us create a tuple.

65. **type() :** The type() function returns the type of the specified object.

66. **vars() :** vars() function returns the __dict__ attribute of a class.

67. **zip() :** zip() returns us an iterator of tuples.

Q.8 Write about Defining and Calling a Function in Python.

Ans. A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

(i) Function blocks begin with the keyword 'def' followed by the function name and parentheses (( )).

(ii) Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

(iii) The first statement of a function can be an optional statement - the documentation string of the function or docstring.

(iv) The code block within every function starts with a colon (:) and is indented.

(v) The statement, return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax :

```
def functionname( parameters ):
"function_docstring"
function_suite
return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example : The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ) :
"This prints a passed string into this function"
print str
return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function -

```
#!/usr/bin/python
# Function definition is here
def printme( str ) :
"This prints a passed string into this function"
print str
return;
# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result -

```
I'm first call to user defined function!
Again second call to the same function
```

Q.9 Write about Arguments in Python.

Ans. A parameter is a variable in a method definition. When a method is called, then the argument are the data you pass into the methods parameter.

Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.

Python refers to the values being passed as arguments

Arguments in python can be one of these value types

1. Literals
2. variables
3. expressions

The alternatives names for arguments are actual parameter and actual argument.

Function Arguments : You can call a function by using the following types of formal arguments -

(i) Required arguments
(ii) Keyword arguments
(iii) Default arguments
(iv) Variable-length arguments

(1) **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.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
"This prints a passed string into this function"
    print str
    return;
# Now you can call printme function
    printme()
```

When the above code is executed, it produces the following result -

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```


(2) **Keyword arguments :** Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways -

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
"This prints a passed string into this function"
    print str
    return;
# Now you can call printme function
    printme( str = "My string")
```

When the above code is executed, it produces the following result -

```
My string
```

(3) **Default arguments :** A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed -

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age = 35 ):
"This prints a passed info into this function"
   print "Name:", name
   print "Age", age
   return;
# Now you can call printinfo function
   printinfo( age=50, name="miki" )
   printinfo( name="miki" )
```

When the above code is executed, it produces the following result -

```
Name: miki
Age 50
Name: miki
Age 35
```

(4) **Variable-length arguments :** You may need to process a function for more arguments than you specified while defining the function. These argu-

ments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this -

```
def functionname([formal_args,] *var_args_tuple ) :
"function_docstring"
function_suite
return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example :

```
#!/usr/bin/python
# Function definition is here
def printinfo( arg1, *vartuple ) :
"This prints a variable passed arguments"
print "Output is:"
print arg1
for var in vartuple:
print var
return;
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result -

Output is:

```
10
```

Output is:

```
70
60
50
```

Q.10 Write about Global vs. Local Variables in Python.

Ans. All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python -

(i) Global variables

(ii) Local variables

(i) **Local variables :** Local variables can only be reached in their scope. The example below has two local variables: x and y.

```
def sum(x,y):
sum = x + y
return sum
print(sum(8, 6))
```

The variables x and y can only be used inside the function sum, they don't exist outside of the function. Local variables cannot be used outside of their scope, this line will not work:

```
print(x)
```

(ii) **Global variables :** A global variable can be used anywhere in the code. In the example below we define a global variable z

```
z = 10
def afunction() :
global z
print(z)
afunction()
print(z)
```

The global variable z can be used all throughout the program, inside functions or outside.A global variable can be modified inside a function and changed for the entire program:

```
z = 10
def afunction() :
    global z
    z = 9
afunction()
print(z)
```

After calling a function(), the global variable is changed for the entire program.

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example -

```
#!/usr/bin/python
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ) :
# Add both the parameters and return them."
total = arg1 + arg2; # Here total is local variable.
print "Inside the function local total :", total
return total;
# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total :", total
```

When the above code is executed, it produces the following result -

```
Inside the function local total : 30
Outside the function global total : 0
```

The way Python uses global and local variables is excellent. While in many or most other programming languages variables are treated as global if not otherwise declared, Python deals with variables the other way around. They are local, if not otherwise declared. The driving reason behind this approach is that global variables are generally bad practice and should be avoided. In most cases where you are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out. Like in many other program structures, Python also imposes good programming habit by design.

So when you define variables inside a function definition, they are local to this function by default. This means that anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name. This means that the function body is the scope of such a variable, i.e. the enclosing context where this name with its values is associated.

All variables have the scope of the block, where they are declared and defined in. They can only be used after the point of their declaration.

Q.11 Write about Defining and Using lambda Function in Python.

Ans. In Python, anonymous function means that a function is without a name. 'def' keyword is used to define the normal functions and the 'lambda' keyword is used to create anonymous functions. It has the following syntax:

```
lambda arguments: expression
```

This function can have any number of arguments but only one expression, which is evaluated and returned. One is free to use 'lambda' functions

wherever function objects are required. You need to keep in your knowledge that 'lambda' functions are syntactically restricted to a single expression. It has various uses in particular fields of programming besides other types of expressions in functions.

Let's look at this example and try to understand the difference between a normal def defined function and 'lambda' function. This is a program that returns the cube of a given value:

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().
def cube(y):
    return y*y*y;
g = lambda x: x*x*x
print(g(7))
print(cube(5))
```

Output:

```
343
125
```

1. **Without using 'Lambda' :** Here, both of them returns the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the 'return' keyword.

2. **Using 'Lambda' :** Lambda definition does not include a "return" statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all. This is the simplicity of lambda functions.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Use of lambda() with filter()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True. Here is a small program that returns the odd numbers from an input list:

```
# Python code to illustrate
# filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x%2 != 0), li))
```

```
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

Use of lambda() with map()

The map() function in Python takes in a function and a list as arguments. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

Example:

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2, li)) print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

Use of lambda() with reduce()

The reduce() function in Python takes in a function and a list as arguments. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list. This is a part of functools module. Example:

```
# Python code to illustrate
# reduce() with lambda()
# to get sum of a list
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

Output:

```
193
```

Here the results of previous two elements are added to the next element and this goes on till the end of the list like (((((5+8)+10)+20)+50)+100).

Module-III

Q.1 Write about Introduction to Numpy in Python.

Ans. NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely. NumPy stands for Numerical Python.

Uses of NumPy

(i) In Python we have lists that serve the purpose of arrays, but they are slow to process.

(ii) NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

(iii) The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

(iv) Arrays are very frequently used in data science, where speed and resources are very important.

Data Science is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

NumPy is Faster Than Lists

(i) NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

(ii) This behavior is called locality of reference in computer science.

(iii) This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Language in NumPy

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Q.2 Write about NumPy Array in Python.

Ans.

Introduction to NumPy

NumPy (Numerical Python) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures. It provides ndarray, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

If you already have Python, you can install NumPy with:

```
conda install numpy
```

or

```
pip install numpy
```

To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

We shorten the imported name to np for better readability of code using NumPy.

NumPy Array in Python

NumPy arrays can be defined using Python sequences such as lists and tuples. Lists and tuples are defined using [...] and (...), respectively. Lists and tuples can define ndarray creation:

(i) a list of numbers will create a 1D array,

(ii) a list of lists will create a 2D array,

(iii) further nested lists will create higher-dimensional arrays. In general, any array object is called an ndarray in NumPy.

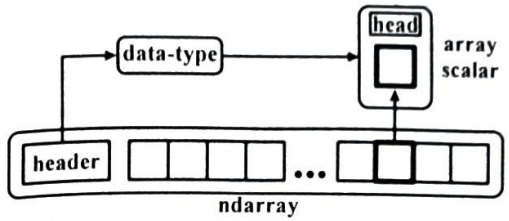
```
>>> a1D = np.array([1, 2, 3, 4])
>>> a2D = np.array([[1, 2], [3, 4]])
>>> a3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

The homogeneous multidimensional array is the main object of NumPy. It is basically a table of elements which are all of the same type and indexed by a tuple of positive integers. The dimensions are called axis in NumPy.

The NumPy's array class is known as ndarray or alias array. The numpy.array is not the same as the standard Python library class array.array. The array.array handles only one-dimensional array and provides less functionality.

Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called dtype).

Any item extracted from ndarray object (by slicing) is represented by a Python object of one of array scalar types. The following diagram shows a relationship between ndarray, data type object (dtype) and array scalar type:



An instance of ndarray class can be constructed by different array creation routines. The basic ndarray is using an array function in NumPy as follows:

```
numpy.array
```

It creates an ndarray from any object exposing array interface, or from any method that returns an array.

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

The above constructor takes the following parameters :

	Parameter	Description
(1)	object	Any object exposing the array interface method returns an array, or any (nested) sequence.
(2)	dtype	Desired data type of array, optional
(3)	copy	Optional. By default (true), the object is copied
(4)	order	C (row major) or F (column major) or A (any) (default)
(5)	subok	By default, returned array forced to be a base class array. If true, sub-classes passed through
(6)	ndmin	Specifies minimum dimensions of resultant array

Ex. 1:

```
import numpy as np
a = np.array([1,2,3])
print a
```

The output is as follows:

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

Ex. 2:

```
# more than one dimensions
import numpy as np
```

```
a = np.array([[1, 2], [3, 4]])
print a
```

The output is as follows:

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

Ex. 3: # minimum dimensions

```
import numpy as np
a = np.array([1, 2, 3, 4, 5], ndmin = 2)
print a
```

The output is as follows:

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

Ex. 4: # dtype parameter

```
import numpy as np
a = np.array([1, 2, 3], dtype = complex)
print a
```

The output is as follows:

```
[1. + 0.j, 2. + 0.j, 3. + 0.j]
```

The ndarray object consists of contiguous one-dimensional segment of computer memory, combined with an indexing scheme that maps each item to a location in the memory block. The memory block holds the elements in a row-major order (C style) or a column-major order (FORTRAN or MatLab style).

Comparison of Python List with Numpy Array

	Numpy Array		Python List
(1)	It is the core Library of python which is used for scientific computing.	(1)	The core library of python provides list.
(2)	It can contain similar datatypes.	(2)	It Contains different types of datatypes.
(3)	We need Numpy Library to access Numpy Arrays.	(3)	It is built-in function of python.
(4)	It is Homogeneous.	(4)	It is both homogeneous and heterogeneous.
(5)	In this Element wise operation is possible.	(5)	Element wise operation is not possible on the list.

(6)	By using numpy.array() we can create N-Dimensional array.	(6)	It is by default 1-dimensional. In some cases, we can create an N-Dimensional list. But it is a long process.
(7)	It requires smaller memory consumption as compared to Python List.	(7)	It requires more memory as compared to Numpy Array.
(8)	In this each item is stored in a sequential manner.	(8)	It stores item in random location of the memory.
(9)	It is faster as compared to list.	(9)	It is slow as compared to NumPy Array.
(10)	It also have some optimism function.	(10)	It does not have some optimism function.

Attributes of NumPy Array

Various array attributes of NumPy are:

ndarray.shape

This array attribute returns a tuple consisting of array dimensions. It can also be used to resize the array.

Ex. 1:

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print a.shape
```

The output is as follows:

```
(2, 3)
```

Ex. 2: # this resizes the ndarray

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
a.shape = (3,2)
print a
```

The output is as follows:

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

Ex. 3: NumPy also provides a reshape function to resize an array.

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

```
b = a.reshape(3, 2)
print b
```

The output is as follows:

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

ndarray.ndim

This array attribute returns the number of array dimensions.

Ex. 1: # an array of evenly spaced numbers

```
import numpy as np
a = np.arange(24)
print a
```

The output is as follows:

```
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

Ex. 2: # this is one dimensional array

```
import numpy as np
a = np.arange(24)
a.ndim

# now reshape it
b = a.reshape(2,4,3)
print b
# b is having three dimensions
```

The output is as follows:

```
[[[0, 1, 2]
  [3, 4, 5]
  [6, 7, 8]
  [9, 10, 11]]
 [[12, 13, 14]
  [15, 16, 17]
  [18, 19, 20]
  [21, 22, 23]]]
```

numpy.itemsize

This array attribute returns the length of each element of array in bytes.

Ex. 1: # dtype of array is int8 (1 byte)

```
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.int8)
print x.itemsize
```

The output is as follows:

```
1
```

Ex. 2: # dtype of array is now float32 (4 bytes)

```
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.float32)
print x.itemsize
```

The output is as follows:

```
4
```


numpy.flags

The ndarray object has the following attributes. Its current values are returned by this function.

	Attribute	Description
(1)	C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment
(2)	F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment
(3)	OWNDATA (O)	The array owns the memory it uses or borrows it from another object
(4)	WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only
(5)	ALIGNED (A)	The data and all elements are aligned appropriately for the hardware
(6)	UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array

Ex. : The following example shows the current values of flags.

```
import numpy as np
x = np.array([1,2,3,4,5])
print x.flags
```

The output is as follows:

```
C_CONTIGUOUS : True
F_CONTIGUOUS : True
OWNDATA : True
```

WRITEABLE : True
ALIGNED: True
UPDATEIFCOPY : False

Data types

A data type is a way to specify the type of data that will be stored in an ray. For example,

array1 = np.array([2, 4, 6])

Here, the array1 array contains three integer elements, so the data type Integer(int64)), by default.

NumPy supports a much greater variety of numerical types than Python does. The following table shows different scalar data types defined in NumPy.

	Data Types	Description
(1)	bool_	Boolean (True or False) stored as a byte
(2)	int_	Default integer type (same as C long; normally either int64 or int32)
(3)	intc	Identical to C int (normally int32 or int64)
(4)	intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
(5)	int8	Byte (–128 to 127)
(6)	int16	Integer (–32768 to 32767)
(7)	int32	Integer (–2147483648 to 2147483647)
(8)	int64	Integer (–9223372036854775808 to 9223372036854775808)
(9)	uint8	Unsigned integer (0 to 255)
(10)	uint16	Unsigned integer (0 to 65535)
(11)	uint32	Unsigned integer (0 to 4294967295)
(12)	uint64	Unsigned integer (0 to 18446744073 709551615)
(13)	float_	Shorthand for float64
(14)	float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
(15)	float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
(16)	float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
(17)	complex_	Shorthand for complex128

(18)	complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
(19)	complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

NumPy numerical types are instances of dtype (data-type) objects, each having unique characteristics. The dtypes are available as np.bool_, np.float32, etc.

Data Type Objects (dtype)

A data type object describes interpretation of fixed block of memory corresponding to an array, depending on the following aspects:

(i) Type of data (integer, float or Python object)
(ii) Size of data
(iii) Byte order (little-endian or big-endian)
(iv) In case of structured type, the names of fields, data type of each field and part of the memory block taken by each field.
(v) If data type is a subarray, its shape and data type

The byte order is decided by prefixing '<' or '>' to data type. '<' means that encoding is little-endian (least significant is stored in smallest address). '>' means that encoding is big-endian (most significant byte is stored in smallest address).

A dtype object is constructed using the following syntax:

```
numpy.dtype(object, align, copy)
```

The parameters are:

(i) **Object:** To be converted to data type object
(ii) **Align:** If true, adds padding to the field to make it similar to C-struct
(iii) **Copy:** Makes a new copy of dtype object. If false, the result is reference to builtin data type object

Ex.:

```
# using array-scalar type
import numpy as np
dt = np.dtype(np.int32)
print dt
```

The output is as follows:

```
int32
```

Ex.: Creating NumPy Arrays With a Defined Data Type

In NumPy, we can create an array with a defined data type by passing the dtype parameter while calling the np.array() function. For example,

```
import numpy as np
# create an array of 32-bit integers
array1 = np.array([1, 3, 7], dtype='int32')
print(array1, array1.dtype)
```

Output:

```
[1 3 7] int32
```

In the above example, we have created a NumPy array named array1 with a defined data type.

Notice the code,

```
np.array([1, 3, 7], dtype='int32')
```

Here, inside np.array(), we have passed an array [1, 3, 7] and set the dtype parameter to int32.

Since we have set the data type of the array to int32, each element of the array is represented as a 32-bit integer.

Ex.: Creating NumPy Arrays With a Defined Data Type

```
import numpy as np
# create an array of 8-bit integers
array1 = np.array([1, 3, 7], dtype='int8')
# create an array of unsigned 16-bit integers
array2 = np.array([2, 4, 6], dtype='uint16')

# create an array of 32-bit floating-point numbers
array3 = np.array([1.2, 2.3, 3.4], dtype='float32')
# create an array of 64-bit complex numbers
array4 = np.array([1 +2j, 2+3j, 3+4j], dtype='complex64')
# print the arrays and their data types
print(array1, array1.dtype)
print(array2, array2.dtype)
print(array3, array3.dtype)
print(array4, array4.dtype)
```

Output

```
[1 3 7] int8
[2 4 6] uint16
[1.2 2.3 3.4] float32
[1.+2.j 2.+3.j 3.+4.j] complex64
```


Q.3 Write about Array in Python.

Ans.

NumPy Array Functions

NumPy array functions are the built-in functions provided by NumPy that allow us to create and manipulate arrays, and perform different operations on them.

We will discuss some of the most commonly used NumPy array functions.

Common NumPy Array Functions

There are many NumPy array functions available but here are some of the most commonly used ones.

	Array Operations	Functions
1.	Array Creation Functions	np.array(), np.zeros(), np.ones(), np.empty(), etc.
2.	Array Manipulation Functions	np.reshape(), np.transpose(), etc.
3.	Array Mathematical Functions	np.add(), np.subtract(), np.sqrt(), np.power(), etc.
4.	Array Statistical Functions	np.median(), np.mean(), np.std(), and np.var().
5.	Array Input and Output Functions	np.save(), np.load(), np.loadtxt(), etc.

Array Creation

A new ndarray object can be constructed by any of the following array creation routines or using a low-level ndarray constructor.

numpy.empty

It creates an uninitialized array of specified shape and dtype. It uses the following constructor:

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

The constructor takes the following parameters.

	Parameter	Description
(1)	Shape	Shape of an empty array in int or tuple of int
(2)	Dtype	Desired output data type. Optional
(3)	Order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Ex.: The following code shows an example of an empty array.

```
import numpy as np
```

```
x = np.empty([3,2], dtype = int)
print x
```

The output is as follows:

```
[[22649312      1701344351]
 [1818321759    1885959276]
 [16779776      156368896]]
```

Note: The elements in an array show random values as they are not initialized.

numpy.zeros

Returns a new array of specified size, filled with zeros.

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

The constructor takes the following parameters.

	Parameter	Description
(1)	Shape	Shape of an empty array in int or sequence of int
(2)	Dtype	Desired output data type. Optional
(3)	Order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Ex. 1 : # array of five zeros. Default dtype is float

```
import numpy as np
x = np.zeros(5)
print x
```

The output is as follows:

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

Ex. 2 : import numpy as np

```
x = np.zeros((5,), dtype = np.int)
print, x
```

Now, the output would be as follows:

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

numpy.ones

Returns a new array of specified size and type, filled with ones.

```
numpy.ones(shape, dtype = None, order = 'C')
```

The constructor takes the following parameters.

	Parameter	Description
(1)	Shape	Shape of an empty array in int or tuple of int

(2)	Dtype	Desired output data type. Optional
(3)	Order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Ex. 1: # array of five ones. Default dtype is float

```
import numpy as np
x = np.ones(5)
print x
```

The output is as follows:

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

Ex. 2: import numpy as np

```
x = np.ones([2,2], dtype = int)
print x
```

Now, the output would be as follows:

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

Array From Existing Data

NumPy provides us the way to create an array by using the existing data.

numpy.asarray

This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.

The syntax to use the asarray() routine is given below.

```
numpy.asarray(sequence, dtype = None, order = None)
```

It accepts the following parameters.

1. **sequence:** It is the python sequence which is to be converted into the python array.
2. **dtype:** It is the data type of each item of the array.
3. **order:** It can be set to C or F. The default is C.

Ex.: creating numpy array using the list

```
import numpy as np
l=[1,2,3,4,5,6,7]
a = np.asarray(l):
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
```

[1 2 3 4 5 6 7]

Ex.: creating a numpy array using Tuple

```
import numpy as np
I=(1,2,3,4,5,6,7)
a = np.asarrayt1);
print(type(a))
print(a)
```

Output

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

Ex.: creating a numpy array using more than one list

```
import numpy as np
I= [[1,2,3,4,5,6,7],[8,9]]
a = np.asarray(I);
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]
```

numpy.frombuffer

This function is used to create an array by using the specified buffer. The syntax to use this buffer is given below.

```
numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)
```

It accepts the following parameters.

(i) **buffer:** It represents an object that exposes a buffer interface.

(ii) **dtype:** It represents the data-type of the returned data type array. The default value is 0.

(iii) **count:** It represents the length of the returned ndarray. The default value is -1.

(iv) **offset:** It represents the starting position to read from. The default value is 0.

Ex.:

```
import numpy as np
I = b'hello world'
print(type(I))
a = np.frombuffer(I, dtype = "S1")
print(a)
print(type(a))
```


Output:

```
<class 'bytes'>
[b'h' b'e' b'l' b'l' b'o' b' ', b'w' b'o' b'r' b'l' b'd']
<class 'numpy.ndarray'>
```

numpy.fromiter

This routine is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarray object.

The syntax is given below.

```
numpy.fromiter(iterable, dtype, count = - 1)
```

It accepts the following parameters.

(i) **Iterable:** It represents an iterable object.

(ii) **dtype:** It represents the data type of the resultant array items.

(iii) **count:** It represents the number of items to read from the buffer in the array.

Ex.:

```
import numpy as np
list = [0,2,4,6]
it = iter(list)
x = np.fromiter(it, dtype = float)
print(x)
print(type(x))
```

Output:

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

numpy.arrange

This function returns an ndarray object containing evenly spaced values within a given range. The format of the function is as follows:

```
numpy.arrange(start, stop, step, dtype)
```

The constructor takes the following parameters.

	Parameter	Description
(1)	start	The start of an interval. If omitted, defaults to 0
(2)	stop	The end of an interval (not including this number)
(3)	step	Spacing between values, default is 1
(4)	dtype	Data type of resulting ndarray. If not given, data type of input is used

The following examples show how you can use this function.

Ex. 1:
```
import numpy as np
x = np.arange(5)
print x
```
Its output would be as follows:
```
[0 1 2 3 4]
```
Ex. 2:
```
import numpy as np
# dtype set
x = np.arange(5, dtype = float)
print x
```
Here, the output would be:
```
[0. 1. 2. 3. 4.]
```
Ex. 3:
```
# start and stop parameters set
import numpy as np
x = np.arange(10,20,2)
print x
```
Its output is as follows:
```
[10 12 14 16 18]
```

numpy.linspace

This function is similar to arange() function. In this function, instead of step size, the number of evenly spaced values between the interval is specified. The usage of this function is as follows :

```
numpy.linspace(start, stop, num, endpoint, retstep, dtype)
```

The constructor takes the following parameters.

	Parameter	Description
(1)	start	The starting value of the sequence
(2)	stop	The end value of the sequence, included in the sequence if endpoint set to true
(3)	num	The number of evenly spaced samples to be generated. Default is 50
(4)	endpoint	True by default, hence the stop value is included in the sequence. If false, it is not included
(5)	retstep	If true, returns samples and step between the consecutive numbers
(6)	dtype	Data type of output ndarray

The following examples demonstrate the use linspace function.

Ex. 1 :
```
import numpy as np
x = np.linspace(10,20,5)
print x
```
Its output would be:
```
[10.  12.5  15.  17.5  20.]
```
Ex. 2 :
```
# endpoint set to false
import numpy as np
x = np.linspace(10,20, 5, endpoint = False)
print x
```
Its output would be:
```
[10.  12.  14.  16.  18.]
```
Ex. 3 :
```
# find retstep value
import numpy as np

x = np.linspace(1,2,5, retstep = True)
print x
# retstep here is 0.25
```
Now, the output would be:
```
(array([1., 1.25, 1.5, 1.75, 2. ]), 0.25)
```

numpy.logspace

This function returns an ndarray object that contains the numbers that are evenly spaced on a log scale. Start and stop endpoints of the scale are indices of the base, usually 10.

```
numpy.logspace(start, stop, num, endpoint, base, dtype)
```

Following parameters determine the output of logspace function.

	Parameter	Description
(1)	start	The starting point of the sequence is basestart
(2)	stop	The final value of sequence is basestop
(3)	num	The number of values between the range. Default is 50
(4)	endpoint	If true, stop is the last value in the range
(5)	base	Base of log space, default is 10
(6)	dtype	Data type of output array. If not given, it depends upon other input arguments

The following examples will help you understand the logs pace function.

Ex. 1:
```
import numpy as np
# default base is 10
a = np.logspace(1.0, 2.0, num = 10)
print a
```

Its output would be as follows:

```
[10          12.91549665  16.68100537  21.5443469   27.82559402
35.93813664  46.41588834  59.94842503  77.42636827  100 ]
```

Ex. 2:
```
# set base of log space to 2
import numpy as np
a = np.logspace(1,10,num = 10, base = 2)
print a
```

Now, the output would be -

```
[ 2.   4.   8.   16.   32.   64.   128.   256.   512.
1024.]
```

The Eye method

The eye() method creates a 2D array with 1s on the diagonal and 0s elsewhere.

Syntax : The syntax of eye() is:

```
numpy.eye(N, M = None, k = 0, dtype = float, order = 'C')
```

Note: Here, M = None implies that M = N by default.

eye() Arguments

The eye() method takes the following arguments:

(i) **N:** Number of rows in the output (can be int)

(ii) **M (optional):** Number of columns in the output (can be int)

(iii) **k (optional):** the diagonal in question (can be int)

(iv) **dtype (optional):** the datatype of the resultant array

(v) **order (optional):** the memory layout of the array (can be 'C' or 'F')

Notes:

(i) By default k = 0 and represents the main diagonal.

(ii) k > 0 represents diagonals above the main diagonal

(iii) k < 0 represents diagonals below the main diagonal.

The eye() method returns the resultant array with shape NxM with all elements as 0 except the kth diagonal where all elements are 1.

Ex. 1: Create a Square Array Using eye()


```
import numpy as np

# create a square 2x2 array with 1s at diagonal
array1 = np.eye(2)

print('2x2 square array\n', array1)

# create a square 2x2 array with 1 above diagonal
array2 = np.eye(2, k=1)

print('2x2 square array with k = 1\n', array2)

# create a square 2x2 array with 1 below diagonal
array3 = np.eye(2, k=-1)
print('2x2 square array with k = -1\n', array3)
```

Output

```
2x2 square array
    [[1. 0.]
    [0. 1.]]
2x2 square array with k = 1
    [[0. 1.]
    [0. 0.]]
2x2 square array with k = -1
    [[0. 0.]
    [1. 0.]]
```

full() method

The full() method creates a new array of given shape and type, filled with a given fill value.

The syntax of full() is:

```
numpy.empty(shape, fill_value, dtype None, order = 'C', like = None)
```

full() Arguments

The full() method takes the following arguments:

(i) **shape:** desired new shape of the array can be integer or sequence of integers)

(ii) **fill_value:** value to fill the array with

(iii) **dtype (optional):** datatype of the array

(iv) **order (optional):** specifies the order in which the uninitialized values are filled

(v) **like (optional):** reference object to allow the creation of arrays that are not NumPy arrays

full() Return Value

The full() method returns the array of given shape, order, and datatype filled with a fill value.

Ex. 1: Create Array with full()

```
import numpy as np

# create a 1D array of five 2s
array1 = np.full(5,2)
print('1D Array:',array1)
# create a 2D array of 2.0s
array2 = np.full((3,2), 2.0)
print('2D Array: \n',array2)
# create an nd-array of (1,2)
array3 = np.full((2, 2, 2), (1, 2))
print('n-d array:\n',array3)
```

Output:

```
1D Array: [2 2 2 2 2]
2D Array:
[[2. 2.]
 [2. 2.]
 [2. 2.]]
n-d array:
[[[1 2]
  [1 2]]

 [[1 2]
  [1 2]]]
```

If unspecified, the default dtype is the data type of fill value.

Ex. 2: Change the Data Type of Array Elements

You can change the data type of array elements by using the dtype argument.

```
import numpy as np
# create a 1D array of five 2s
# pass the dtype argument to change the data type
array1 = np.full(5, 2, dtype = 'float')
print('1D Array:',array1)
```

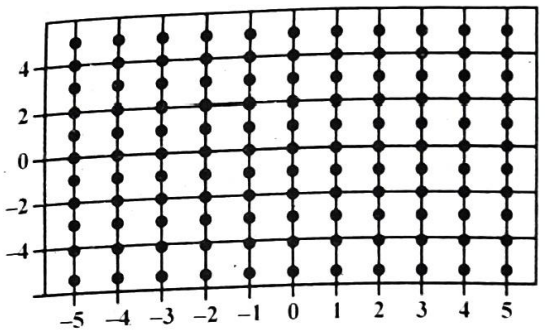
Output:

```
1D Array: [2. 2. 2. 2. 2.]
```

Here, the fill value is 2, so the default dtype is an integer. However, we changed the data type of the filled value to floating-point by passing dtype = 'float'.

Numpy Meshgrid Function

The meshgrid() method takes two or more 1D arrays representing coordinate values and returns a rectangular grid of a pair of 2D arrays. The numpy module of Python provides meshgrid() function, for creating a rectangular grid with the help of the given 1-D arrays that represent the Matrix indexing or Cartesian indexing. MATLAB somewhat inspires the meshgrid() function. From the coordinate vectors, the meshgrid() function returns the coordinate matrices.



In the above figure, the x-axis is ranging from –5 to 5, and the y-axis is ranging from –5 to 5. So, there is a total of 121 points marked in the figure, each with x-coordinate and y coordinate. For any line parallel to the x-axis, the x-coordinates of the marked points are –5, –4, –3, –2, –1, 0, 1, 2, 3, 4, and 5 respectively. On the other hand, for any line parallel to the y-axis, the y-coordinates of the marked points from bottom to top are –5, –4, –3, –2, –1, 0, 1, 2, 3, 4, and 5 respectively.

The syntax of meshgrid() is:

```
numpy.meshgrid(*xi, copy= True, sparse = False, indexing='xy')
```

meshgrid() Argument

The meshgrid() method takes the following arguments:

(i) ***xi:** 1D arrays representing the coordinates of a grid

(ii) **indexing(optional):** specifies index of the grid ('xy'(Cartesian, default) or 'ij'(matrix))

(iii) **sparse(optional):** a bool value, if True a sparse grid is returned

(iv) **copy(optional):** create a copy if True(default) and return view if False

meshgrid() Return Value

The meshgrid() method returns coordinate matrices from coordinate vectors.

Ex. 1: Create a 2-D Grid

```
import numpy as np
# create 1D arrays
x = np.array([1, 2, 3])
y = np.array([1, 2, 3])
# create a 2D grid using meshgrid
X, Y = np.meshgrid(x, y)
# print the created grid
print("X values:\n", X)
print("Y values:\n", Y)
```

Output :

```
X values:
[[1 2 3]
 [1 2 3]
 [1 2 3]]
Y values:
 [[1 1 1]
 [2 2 2]
 [3 3 3]]
```

The code creates a 2D grid where the values of X and Y present the coordinates of the grid points.

The X array represents the x-coordinates of the grid points, where each row contains the same x values [1, 2, 3]. The Y array represents the y-coordinates of the grid points, where each column contains the same y values [1, 2, 3].

2-D Gaussian Array

To create a 2 D Gaussian array using the Numpy python module.

Functions used:

(i) **numpy.meshgrid():** It is used to create a rectangular grid out of two given one-dimensional arrays representing the Cartesian indexing or Matrix indexing.

Syntax:

```
numpy.meshgrid(*xi, copy= True, sparse = False, indexing='xy')
```

(ii) **numpy.linspace():** returns number spaces evenly w.r.t interval.

Syntax:

```
numpy.linspace(start, stop, num = 50, endpoint = True, retstep =
False, dtype = None)
```

(iii) **numpy.exp():** this mathematical function helps the user to calculate the exponential of all the elements in the input array.

Syntax:

```
numpy.exp(array, out = None, where = True, casting =
'same_kind', order 'K', dtype None)
```

Ex. 1: Python3

```
# Importing Numpy package
import numpy as np

# sigma(standard deviation) and muu(mean) are the parameters
of gaussian

def gaussuian_filter(kernel_size, sigma=1, muu=0):

    # Initializing value of x,y as grid of kernel size
    # in the range of kernel size

    x, y = np.meshgrid(np.linspace(-1, 1, kernel_size),
           np.linspace(-1, 1, kernel_size))
    dst = np.sqrt(x**2+y**2)

    # lower normal part of gaussian
    normal = 1/(2, 0 * np.pi * sigma**2)

    # Calculating Gaussian filter
    gauss = np.exp(-((dst-muu)**2 / (2.0 * sigma**2))) *
    normal

kernel_size=5
gaussian = gaussian _filter(kernel_size)
    print("gaussian filter of{} X {} :". format (kernel_ size,
```

```
                kernel_size))
                print(gaussian)
```

Output:

```
gaussian filter of 5 X 5 :
[[0.00291502 0.01306423 0.02153928 0.01306423 0.00291502]
[0.01306423 0.05854983 0.09653235 0.05854983 0.01306423]
[0.02153928 0.09653235 0.15915494 0.09653235 0.02153928]
[0.01306423 0.05854983 0.09653235 0.05854983 0.01306423]
[0.00291502 0.01306423 0.02153928 0.01306423 0.00291502]]
```

Creating Vectors in Python using Numpy

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python. Numpy is basically used for creating array of n dimensions.

Vector are built from components, which are ordinary numbers. We can think of a vector as a list of numbers, and vector algebra as operations performed on the numbers in the list. In other words vector is the numpy 1-D array.

In order to create a vector, we use np.array method.

Syntax: np.array(list)

Argument: It take 1-D list it can be 1 row and n columns or n rows and 1 column

Return : It returns vector which is numpy.ndarray

Note: We can create vector with other method as well which return 1-D numpy array for example np.arange(10), np.zeros((4, 1)) gives 1-D array, but most appropriate way is using np.array with the 1-D list.

Ex.: In this example we will create a horizontal vector and a vertical vector

```
# importing numpy
import numpy as np

# creating a 1-D list (Horizontal)
list1 = [1, 2, 3]

# creating a 1-D list (Vertical)
list2 = [[10],
         [20],
         [30]]
# creating a vector1
# vector as row
vector1 = np.array(list1)

# creating a vector 2
# vector as column
vector2 = np.array(list2)

# showing horizontal vector
print("Horizontal Vector")
print(vector1)

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

# showing vertical vector
print("Vertical Vector")
print(vector2)
```

Output :

```
Horizontal Vector
[1 2 3]
---------------
Vertical Vector
[[10]
 [20]
 [30]]
```


Q.4 Write about Indexing and Searching in Array in Python.

Ans. In NumPy, each element in an array is associated with a number. The number is known as an array index.

Let's see an example to demonstrate NumPy array indexing.

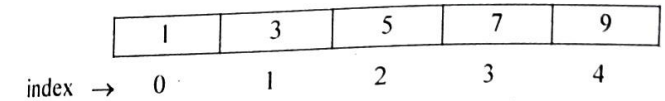


Fig.: Array Indexing in NumPy

In the above array, 5 is the 3rd element. However, its index is 2.

This is because the array indexing starts from 0, that is, the first element of the array has index 0, the second element has index 1, and so on.

Now, we'll see now we can access individual items from the array using the index number.

Access Array Elements Using Index

We can use indices to access individual elements of a NumPy array.

Suppose we have a NumPy array:

```
array1 = np.array([1, 3, 5, 7, 9)]
```

Now, we can use the index number to access array elements as:

(i) array1[0] - to access the first element, i.e. 1

(ii) array1[2] - to access the third element, i.e. 5

(iii) array1[4] - to access the fifth element, i.e. 9

Ex.: Access Array Elements Using Index

```
import numpy as np
array1 = np.array([1, 3, 5, 7, 9])
# access numpy elements using index
print(array1[0])      # prints 1
print(array1[2])      # prints 5
print(array1[4])      # prints 9
```

Note: Since the last element of array1 is at index 4, if we try to access the element beyond that, say index 5, we will get an index error: IndexError: index 5 is out of bounds for axis 0 with size 5

Modify Array Elements Using Index

We can use indices to change the value of an element in a NumPy array. For example,

```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10])
# change the value of the first element
numbers[0] = 12
print("After modifying first element:",numbers) # prints [12 4 6 8 10]
# change the value of the third element
numbers[2] = 14
print("After modifying third element:",numbers)# prints [12 4 14 8 10]
```


Output:

```
After modifying first element: [12 4 6 8 10]
After modifying third element: [12 4 14 8 10]
```

In the above example, we have modified elements of the numbers array using array indexing.

(i) numbers[0] = 12 - modifies the first element of numbers and sets its value to 12

(ii) numbers[2] = 14 - modifies the third element of numbers and sets its value to 14

NumPy Negative Array Indexing

NumPy allows negative indexing for its array. The index of –1 refers to the last item, –2 to the second last item and so on.

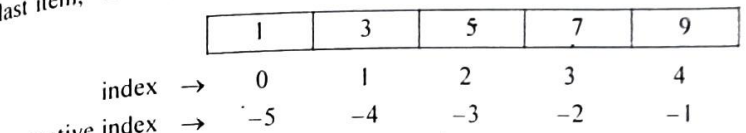


Fig.: NumPy Array Negative Indexing

Let's see an example.

```
import numpy as np
# create a numpy array
numbers = np.array([1, 3, 5, 7, 9])
# access the last element
print(numbers[–1])  # prints 9
# access the second-to-last element
print(numbers[–2])  # prints 7
```

Output:

```
9
7
```

Modify Array Elements Using Negative Indexing

Similar to regular indexing, we can also modify array elements using negative indexing. For example,

```
import numpy as np
# create a numpy array
numbers = np.array([2, 3, 5, 7, 11])

# modify the last element
numbers[–1] = 13
```

```
print(numbers)      # Output: [2 3 5 7 13]
# modify the second-to-last element
numbers[-2] = 17
print(numbers)      # Output: [2 3 5 17 13]
```

Here, numbers[-1] = 13 modifies the last element to 13 and numbers[-2] = 17 modifies the second-to-last element to 17.

Note : Unlike regular indexing, negative indexing starts from -1 (not 0) and it starts counting from the end of the array.

2-D NumPy Array Indexing

Array indexing in NumPy allows us to access and manipulate elements in a 2-D array.

To access an element of array1, we need to specify the row index and column index of the element. Suppose we have following 2-D array,

```
array1 = np.array([[1, 3, 5],
                   [7, 9, 2],
                   [4, 6, 8]])
```

Now, say we want to access the element in the third row and second column we specify the index as:

```
array1[2, 1] # returns 6
```

Since we know indexing starts from 0. So to access the element in the third row and second column, we need to use index 2 for the third row and index 1 for the second column respectively.

Ex.: 2-D NumPy Array Indexing

```
import numpy as np
# create a 20 array
array1 = np.array([[1, 3, 5, 7],
                   [9, 11, 13, 15],
                   [2, 4, 6, 8]])
# access the element at the second row and fourth column
element1 = array1 [1, 3] # returns 15
print("4th Element at 2nd Row:",element1)

# access the element at the first row and second column
element2 = array1 [0, 1] # returns 3
print("2n Element at First Row:",element2)
```

Output:

```
4th Element at 2nd Row: 15
2nd Element at First Row: 3
```


Access Row or Column of 2D Array Using Indexing

In NumPy, we can access specific rows or columns of a 2-D array using array indexing.

Let's see an example.

```
import numpy as np
# create a 2D array
array1 = np.array([[1, 3, 5],
                   [7, 9, 2],
                   [4, 6, 8]])
# access the second row of the array
second_row = array1[1, :]
print("Second Row:", second_row) # Output: [7 9 2]
# access the third column of the array
third_col = array1 [:, 2]
print("Third Column:", third_col) # Output: [5 2 8]
```

Output:

```
Second Row: [7 9 2]
Third Column: [5 2 8]
```

Here,

(i) array1 [1, :] - access the second row of array1

(ii) array1 [:, 2] - access the third column of array1

3-D NumPy Array Indexing

We learned how to access elements in a 2D array. We can also access elements in higher dimensional arrays.

To access an element of a 3D array, we use three indices separated by commas.

(i) The first index refers to the slice

(ii) The second index refers to the row

(iii) The third index refers to the column.

Note: In 3D arrays, slice is a 2D array that is obtained by taking a subset of the elements in one of the dimensions.

Let's see an example.

```
import numpy as np
# create a 3D array with shape (2, 3, 4)
array1 = np.array([[[1, 2, 3, 4],
```

```
        [5, 6, 7, 8],
        [9, 10, 11, 12]],
            [[13, 14, 15, 16],
            [17, 18, 19, 20],
            [21, 22, 23, 24]]])
# access a specific element of the array
element = array1 [1, 2, 1]
# print the value of the element
print(element)
```

Output: 22

Here, we created a 3D array called array1 with shape (2, 3, 4). This array contains 2 2D arrays, each with 3 rows and 4 columns.

Then, we used indexing to access a specific element of array1. Notice the code,

```
array1[1, 2, 1]
```

Here,

(i) array1[1, , ,] - access the second 2D array, i.e.
```
[13, 14, 15, 16],
[17, 18, 19, 20],
[21, 22, 23, 24]
```

(ii) array1 [,2,] - access the third row of the 2D array, i.e.
```
[21, 22, 23, 24]
```

(iii) array1 [, ,1] - access the second element of the third row, i.e.
```
[22]
```

Array Slicing

Array Slicing is the process of extracting a portion of an array. With slicing, we can easily access elements in the array. It can be done on one or more dimensions of a NumPy array.

Syntax of NumPy Array Slicing:

```
array[start:stop:step]
```

Here,

(i) **start:** index of the first element to be included in the slice

(ii) **stop:** index of the last element (exclusive)

(iii) **step:** step size between each element in the slice

Note: When we slice arrays, the start index is inclusive but the stop index is exclusive.

(i) If we omit start, slicing starts from the first element

(ii) If we omit stop, slicing continues up to the last element

(iii) If we omit step, default step size is 1

1D NumPy Array Slicing

In NumPy, it's possible to access the portion of an array using the slicing operator :. For example,

```
import numpy as np
# create a 1D array
array1 = np.array([1, 3, 5, 7, 8, 9, 2, 4, 6])

# slice array1 from index 2 to index 6 (exclusive)
print(array1 [2 :6]) # [5 7 8 9]
# slice array1 from index 0 to index 8 (exclusive) with a step size of 2
print(array1 [0:8:2]) # [1 5 8 2]
# slice array1 from index 3 up to the last element
print(array1 [3:]) # [7 8 9 2 4 6]
# items from start to end
print(array1[:])      # [1 3 5 7 8 9 2 4 6]
```

In the above example, we have created the array named array1 with 9 elements.

Then, we used the slicing operator : to slice array elements.

(i) array1[2:6] - slices array1 from index 2 to index 6, not including index 6

(ii) array1[0:8:2] - slices array1 from index 0 to index 8, not including index 8

(iii) array1[3:] - slices array1 from index 3 up to the last element

(iv) array1 [:] - returns all items from beginning to end

Modify Array Elements Using Slicing

With slicing, we can also modify array elements using:

1. **Using start Parameter**

```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])
# modify elements from index 3 onwards
numbers[3:] = 20
print(numbers)
```

Output: [2 4 6 20 20 20]

Here, numbers[3:] = 20 replaces all the elements from index 3 onwards with new value 20.

2. **Using stop Parameter**

```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])
# modify the first 3 elements
numbers[:3] = 40
print(numbers)
```

Output: [40 40 40 8 10 12]

Here, numbers[:3] = 20 replaces the first 3 elements with the new value 40.

3. **Using start and stop parameter**

```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])
# modify elements from indices 2 to 5
numbers[2:5] = 22
print(numbers)
```

Output: [2 4 22 22 22 12]

Here, numbers[2:5] = 22 selects elements from index 2 to index 4 and replaces them with new value 22.

4. **Using start, stop, and step parameter**

```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])
# modify every second element from indices 1 to 5
numbers[1:5:2] = 16
print(numbers)
```

Output: [2 16 6 16 10 12]

In the above example, numbers[1:5:2] = 16 modifies every second element from index 1 to index 5 with a new value 16.

NumPy Array Negative Slicing

We can also use negative indices to perform negative slicing in NumPy arrays. During negative slicing, elements are accessed from the end of the array.

Let's see an example.


```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])
# slice the last 3 elements of the array
# using the start parameter
print(numbers[-3:]) # [8 10 12]
# slice elements from 2nd-to-last to 4th-to-last element
# using the start and stop parameters
print(numbers[-5:-2])       # [4 6 8]
# slice every other element of the array from the end
# using the start, stop, and step parameters
print(numbers[-1 ::-2]) # [12 8 4]
```

Output:

```
Using numbers[-3:]- [ 8 10 12]
Using numbers[-5:-2]- [4 6 8]
Using numbers[-1: :-2]- [12 8 4].
```

Here,

(i) numbers[-3:] - slices last 3 elements of numbers

(ii) numbers[-5:-2) - slices numbers elements from 5th last to 2nd last(excluded)

(iii) numbers[-1 ::-2) - slices every other numbers elements from the end with step size 2

Reverse NumPy Array Using Negative Slicing

In NumPy, we can also reverse array elements using the negative slicing. For example,

```
import numpy as np
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])
# generate reversed array
reversed_numbers = numbers[::-1]
print(reversed_ numbers)
```

Output: [12 10 8 6 4 2]

Here, the slice numbers[: :-1] selects all the elements of the array with a step size of -1, which reverses the order of the elements.

2D NumPy Array Slicing

A 2D NumPy array can be thought of as a matrix, where each element has two indices, row index and column index.

To slice a 2D NumPy array, we can use the same syntax as for slicing a 1D NumPy array. The only difference is that we need to specify a slice for each dimension of the array.

Syntax of 2D NumPy Array Slicing:

```
array[row_start:row_stop:row_
step, col_start:col_stop:col_step]
```

Here,

(i) **row_start,row_stop,row_step:** specifies starting index, stopping index, and step size for the rows respectively

(ii) **col_start,col_stop,col_step:** specifies starting index, stopping index, and step size for the columns respectively

Let's understand this with an example.

```
# create a 2D array
array1 = np.array([[1, 3, 5, 7],
        [9, 11, 13, 15]])
print(array1[:2, :2])
```

Output:

```
[[1 3]
[9 11]]
```

Here, the , in [:2, :2] separates the rows of the array. The first :2 returns first 2 rows i.e., entire array1. This results in

```
[1 3]
```

The second : 2 returns first 2 columns from the 2 rows. This results in

```
[9 11]
```

NumPy Array Reshaping

NumPy array reshaping simply means changing the shape of an array without changing its data.

Let's say we have a 1D array.

```
np.array([1, 3, 5, 7, 2, 4, 6, 8])
```

We can reshape this 1D array into N-d array as

```
# reshape 1D into 2D array
# with 2 rows and 4 columns
[[1 3 5 7]
[2 4 6 8]]

# reshape 1D into 3D array
# with 2 rows, 2 columns, and 2 layers
```



```
[[[1 3]
  [5 7]]
 [[2 4]
  [6 8]]]
```

Here, we can see that the 1D array has been reshaped into 2D and 3D arrays without altering its data.

Syntax NumPy Array Reshaping: The syntax of NumPy array reshaping is

```
np.reshape(array, newshape, order = 'C')
```

Here,

(i) **array:** input array that needs to be reshaped,

(ii) **newshape:** desired new shape of the array

(iii) **order (optional):** specifies the order in which the elements of the array should be arranged. By default it is set to 'C'

Note: The order argument can take one of three values: 'C', 'F', or 'A'.

Reshape 1D Array to 2D Array in NumPy

We use the reshape() function to reshape a 1D array into a 2D array. For example,

```
import numpy as np
array1 = np.array([1, 3, 5, 7, 2, 4, 6, 8])
# reshape a 1D array into a 2D array
# with 2 rows and 4 columns
result = np.reshape(array1, (2, 4))
print(result)
```

Output:

```
[[1 3 5 7]
[2 4 6 8]]
```

In the above example, we have used the reshape() function to change the shape of the 1D array named array1 into the 2D array. Notice the use of the reshape() function,

```
np.reshape(array1, (2, 4))
```

Here, reshape() takes two parameters,

(i) **array1:** array to be reshaped

(ii) **(2, 4):** new shape of array1 specified as a tuple with 2 rows and 4 columns.

Ex.: Reshape 1D Array to 2D Array in NumPy

```
import numpy as np
```

```
# reshape a 1D array into a 2D array
# with 4 rows and 2 columns
array1 = np.array([1, 3, 5, 7, 2, 4, 6, 8])
result1 = np.reshape(array1, (4, 2))
print("With 4 rows and 2 columns: \n",result1)
# reshape a 1D array into a 2D array with a single row
result2 = np.reshape(array1, (1, 8))
print("\nWith a single row: \n",result2)
```

Output:

```
With 4 rows and 2 columns:
[[1 3]
 [5 7]
 [2 4]
 [6 8]]
With a single row:
 [[1 3 5 7 2 4 6 8]]
```

Note:

(i) We need to make sure that the new shape in reshape() has the same number of elements as the original array, else a ValueError will be raised.

(ii) For example, reshaping an 8 element 10 array into a 2D array of 2 rows and 4 columns is possible, but reshaping it into a 2D array of 3 rows and 3 columns is not possible as that would require 3x3 = 9 elements.

Array Shape

To get the shape or dimensions of a NumPy Array, use ndarray.shape where ndarray is the name of the numpy array you are interested of ndarray.shape returns a tuple with dimensions along all the axis of the numpy array.

In the following example, we have initialized a multi-dimensional numpy array. Of course, we know the shape of the array by its definition. But, we will use ndarray.shape property to get the shape of the array programmatically.

Python Program

```
import numpy as np

# Initialize an array
arr = np.array([[[11, 11, 9, 9],
   [11, 0, 2, 0]
            ],
   [[10, 14, 9, 14],
```



```
    [0, 1, 11, 11]]])

# Get array shape
shape = arr.shape
print(shape)
```

Output:

```
(2, 2, 4)
```

Copy Vs View in NumPy

NumPy consists of different methods to duplicate an original array. The two main functions for this duplication are copy and view. The duplication of the array means an array assignment.

When we duplicate the original array, the changes made in the new array may or may not reflect. The duplicate array may use the same location or may be at a new memory location.

The main highlight difference between a copy and view it in its memory location. The copy of an array is a new array. The view, on the other hand, is just a view of the original array.

We store the copy at a new memory location. A copy returns the data stored at the new location. The changes made in the copy data does not reflect in the original array. The copy of an array owns its own data.

The view is a view of the original array at the same memory location. A view returns the data stored at the original memory location. The changes made in the original data are reflected in the view and vice versa. The view of an array does not own its own data.

Copy or Deep Copy in NumPy

It returns a copy of the original array stored at a new location. The copy doesn't share data or memory with the original array. The modifications are not reflected. The copy function is also known as deep copy.

```
import numpy as np
arr = np.array([20,30,50,70])
a = arr.copy()
#changing a value in original array
arr[0] = 100
print(arr)
print(a)
```

Output:

```
[100 30 50 70]
[20 30 50 70]
```

Changes made in the original array are not reflected in the copy. Similarly, changes made in copy are not reflected in the original array.

View or Shallow copy in NumPy

It returns a view of the original array stored at the existing location. The view doesn't have its own data or memory but uses the original array. The modifications reflect in both. We also call the view function shallow copy.

```
import numpy as np
arr = np.array([20,30,50,70])
a= arr.view()
#changing a value in original array
arr[0] = 100
print(arr)
print(a)
```

Output :

```
[100 30 50 70]
[100 30 50 70]
```

Changes made in the original array reflect in the view. In the same way, changes made in the view are reflected in the original array.

No copy in NumPy

When we make assignments to an array does not create a copy of the array. It actually accesses the original array through its id(). The id() element is equivalent to pointers in c /c+ + language. They just point to the original array.

```
import numpy as np
#declaring an array
arr1 = np.arange(10)
print(arr1)
#applying id() function
print (id(arr1))
#assign the array to another variable
arr2 = arr1
print (arr2)
#we print the id() for arr2
print( id(arr2) )
```

Output:

```
[0 1 2 3 4 5 6 7 8 9]
139825053003648
```



```
[0 1 2 3 4 5 6 7 8 9]
139825053003648
```

NumPy View Creation

We can create a view via two methods. We can either create it by slicing the original array or changing its data type.

NumPy Slice view: In this type of view creation, we perform slicing of the original array. We can then address the view by offsets, strides, and counts of the original array.

```
import numpy as np
arr= np.arange(10)
print(arr)
#slicing of original array to create a view
v=arr[1:10:2]
print(v)
```

Output:

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

NumPy dtype view: In this case we change the data type of the original array and generate a view.

```
import numpy as np
arr = np.arange(10,dtype='int32')
print(arr)
#changing datatype of original array to create a view
v=arr.view('int16')
print(v)
```

Output :

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

Array Iterating

Iterating means going through elements one by one. As we use multi-dimensional array in python.(numpy), we can use for loop.

Iterating 1-D: For 1-D array, iteration happens one by one.

```
import numpy as np
arr=np.array([1,2,3])
print("Original array:",arr)
print("After iteration:")
```

```
for x in arr:
    print(x)
```

Iterating 2-D: For 2-D array, iteration happens passing through all the row.

```
import numpy as np
arr=np.array([[1,2,3],[4,5,6]])
print("Original array:\n",arr)
print("After iteration:")
for x in arr:
    print(x)
# to iterate on each scalar:
print("After iterating on each scalar:")
for x in arr:
    for y in x:
        print(y)
```

Iterating 3-D: For 3-D array, iteration happens passing through all the 2-D array.

```
import numpy as np
arr=np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
print("Original array:\n",arr)
print("After iteration:")
for x in arr:
    print(x)
# to iterate on each scalar:
print("After iterating on each scalar:")
for x in arr:
    for y in x:
        for z in y:
            print(z)
```

Using nditer ( ):

NumPy package provides an iterator object called nditer. It is a multi-dimensional iterator object used to iterate the elements of an array. Each element of an array is visited using Python's standard Iterator interface.

Let us create a 3X4 array using arange() function and iterate over it using nditer.

Ex. :

```
import numpy as np
a = np.arange(0,60,5)
```



```
a = a.reshape(3,4)
print 'Original array is:'
print a
print '\n'
print 'Modified array is:'
for x in np.nditer(a):
    print x,
```

The output of this program is as follows:

```
Original array is:
[[ 0 5 10 15]
[20 25 30 35]
[40 45 50 55]]
Modified array is:
0 5 10 15 20 25 30 35 40 45 50 55
```

Array Join

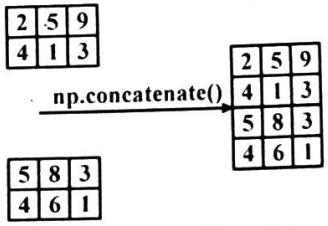
Joining means putting contents of two or more arrays in a single array. In SQL, we join tables based on a key, whereas in NumPy we join arrays by axes.

The concatenate() function is a function from the NumPy package. This function essentially combines NumPy arrays together. This function is basically used for joining two or more arrays of the same shape along a specified axis.

There are the following things which are essential to keep mind:

(1) NumPy's concatenate() is not like a traditional database join. It is like stacking NumPy arrays.

(2) This function can operate both vertically and horizontally. This means we can concatenate arrays together horizontally or vertically.



The concatenate() function is usually written as np.concatenate(), but we can also write it as numpy.concatenate(). It depends on the way of importing the numpy package, either import numpy as np or import numpy, respectively.

Syntax:

numpy.concatenate((a1, a2, ...), axis)

Parameters

(1) (a1, a2, ...): This parameter defines the sequence of arrays. Here, a1, a2, a3... are the arrays which have the same shape, except in the dimension corresponding to the axis.

(2) axis : int(optional): This parameter defines the axis along which the array will be joined. By default, its value is 0.

Result:

It will return a ndarray containing the elements of both the arrays.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Ex. 1: numpy.concatenate()

```
import numpy as np
x = np.array([[1,2],[3,4]])
y=np.array([[12,30]])
z=np.concatenate((x, y))
z
```

In the above code

(i) We have imported numpy with alias name np.

(ii) We have created an array 'x' using np.array() function.

(iii) Then, we have created another array 'y' using the same np.array() function.

(iv) We have declared the variable 'z' and assigned the returned value of np.concatenate() function.

(v) We have passed the array 'x' and 'y' in the function.

(vi) Lastly, we tried to print the value of 'z'.

In the output, values of both the arrays, i.e., 'x' and 'y' shown as per the axis=0.

Output:

```
array([[ 1, 2],
       [3, 4],
       [12, 30]])
```

Ex. 2: numpy.concatenate() with axis=1

```
import numpy as np
x = np.array([[1,2],[3 4]])
y= np.array([[12,30]])
```



```
z=np.concatenate((x,y.T), axis=1)
z
```

Output:

```
array([[ 1, 2, 12],
       [3, 4, 30]])
```

In the above example, the '.T' used to change the rows into columns and columns into rows.

Array Split

Splitting is reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

Ex.: Split the array in 3 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
```

Note: The return value is a list containing three arrays.

If the array has less elements than required, it will adjust from the end accordingly.

NumPy Sort, Search and Count Functions

NumPy contains functions to perform search sort, and count operations. There are a wide variety of functions to get the output. We can use the functions accordingly to produce the most optimum output. We can implement these functions on the array object.

The functions can be used along with each other on the same data. We use NumPy search to find a particular element from the array. We use Numpy sort function for arranging the array in a particular order. NumPy counting function returns the count of a particular value.

Array Search

We perform NumPy search operation to determine the position of a given element or value inside an array. There are functions to find the maximum, minimum, or a value satisfying a particular condition. The search turns out to be successful if the value is found.

1. np.argmax(): This function returns the index of the maximum value in the array list. The search can be specific to a particular axis or the entire array.

```
import numpy as np
```

```
arr= np.array([[74,23],[56,98]])
print(np.argmax(arr))
```

Output: 3

2. **np.nanargmax():** This function returns the index of the maximum alue element after ignoring 'Not a Number' (NaN) values present in the list.

```
import numpy as np
arr= np.array([[74,np.nan,59],[56,98,np.nan]])
print(np.nanargmax(arr))
```

Output : 4

3. **np.argmin():** This function returns the index of the minimum value in the array list. The search can be specific to a particular axis or the entire array.

```
import numpy as np
arr=np.array([[74,59],[56,98]])
print(np.argmin(arr))
```

Output: 2

4. **np.nanargmin():** This function returns the index of the minimum value element after ignoring 'Not a Number' (NaN) values present in the list.

```
import numpy as np
arr=np.array([[74, np.nan,59],[56,98,np.nan]])
print(np.nanargmin(arr))
```

Output: 3

5. **np.extract():** This function returns the array values that satisfy a particular condition.

```
import numpy as np
arr=np.array([[74,33],[59,98]])
a=np.mod(arr,2) = =0
print(np.extract(a,arr))
```

Output : [74 98]

6. **np.nonzero():** This function search returns values that are non-zero.

```
import numpy as np
arr=np.array([0,56,89,0,23])
print(np.nonzero(arr))
```

Output: (array([1, 2, 4]))

	Function	Description
(1)	np.argwhere()	It is used to group elements, having non-zero indices.
(2)	np.flatnonzero()	It returns a flattened array consisting of elements with non-zero indices.
(3)	np.where()	It returns elements following a condition.
(4)	np.searchsorted()	It returns indices of elements that are in sorted manner.

Array Sort

There are a wide variety of sorting functions in NumPy. These NumPy Sort functions arrange the data in a particular order. We choose the best sorting algorithm depending on the output criteria. We can apply for any order over the data. There are various sorting functions available.

1. **np.sort():** This function returns an array in sorted format.

```
import numpy as np
arr = np.array([[16,1,47,53,87,28]])
print(np.sort(arr))
```

Output: [[1 16 28 47 53 87]]

2. **np.argsort():** This function returns the indices of the sorted array. We can perform this along a particular axis.

```
import numpy as np
arr = np.array([[16,1,47,53,87,28]])
print(np.argsort(arr))
```

Output: [[1 0 5 2 3 4]]

3. **np.lexsort():** This function returns the indices of array elements indirectly sorted by using a key sequence. We can consider the last key as the primary one.

```
import numpy as np
arr1 =np.array([[16,1,47,53,87,28]])
arr2 = np.array([[56,90,44,32,77,45]])
#sort by arr1 and then arr2
print(np.lexsort((arr2,arr1)))
```

Output: [[1 0 5 2 3 4]]

	Function	Description
(1)	np.ndarraysort()	It is for sorting an array in place.

(2)	np.msort()	It returns an array which sorted along the first axis
(3)	np.partition()	It returns a copy of array partition.
(4)	np.argpartition()	It returns the partition of an array along a specific axis.
(5)	np.sort_complex()	It sorts the real and imaginary parts separately.

Variations in Different Sorting Techniques

Let 'a' be a numpy array

a.sort()

(i) Sorts the array in-place & returns None

(ii) Return type is None

(iii) Occupies less space. No copy created as it directly sorts the original array

(iv) Faster than sorted(a)

sorted(a)

(i) Creates a new list from the old & returns the new one, sorted

(ii) Return type is a list

(iii) Occupies more space as copy of original array is created and then sorting is done

(iv) Slower than a.sort()

np.argsort(a)

(i) Returns the indices that would sort an array

(ii) Return type is numpy array

(iii) Occupies space as a new array of sorted indices is returned

np.lexsort((b, a))

(i) Perform an indirect sort using a sequence of keys

(ii) Sort by a, then by b

(iii) Return type ndarray of ints Array of indices that sort the keys along the specified axis

(iv) Occupies space as a new array of sorted indices pair wise is returned.

NumPy Counting Function

We can apply NumPy Count to count a specific kind of value from the array list.

1. **np.count_nonzero():** This function returns the count of all the non-zero values from the array. We can apply this function along a specific axis.

```
import numpy as np
```



```
arr= np.array([[16,0,56],[2,34,0],[90,87,0]])
print(np.count_nonzero(arr))
print(np.count_nonzero(arr,axis=0))
```

Output:

```
6
[3 2 1]
```

Splitting Arrays

	Array	Description
(1)	split	Splits an array into multiple sub-arrays
(2)	hsplit	Splits an array into multiple sub-arrays horizontally (column-wise)
(3)	vsplit	Splits an array into multiple sub-arrays vertically (row-wise)

Comparing and Filtering NumPy array

The NumPy library has a wide range of tools which perform the comparison and filtering of the arrays. The comparison of arrays will be done element by element in row wise and column wise based on the dimension of the array. When we want to retrieve a particular array element, then we can apply the filtering.

NumPy is abbreviated as Numerical python, which is used to perform mathematical and scientific calculations on multidimensional arrays and matrices. There are different functions and methods available in NumPy to perform the filtering and comparison of the arrays.

Comparing NumPy Arrays

Following are the methods available in NumPy library to perform the comparison operations on the arrays.

(i) equal()

(ii) greater()

(iii) array_equal()

(iv) all_close()

(1) **Using equal() Method:** The equal() method in NumPy library compares the two NumPy arrays element-wise and returns a Boolean output for each comparison performed. To elaborate, when one element from the first array is compared with its corresponding element in the second array, 'True' value is returned if they are equal and 'False', otherwise. This process is repeated for all the elements in both these arrays.

All these Boolean values obtained are stored in another array and displayed as the output. If the output array contains all 'True' values, then these arrays are said to be identical and 'False', otherwise.

Ex. : Let us see a simple example comparing two NumPy arrays element by element, using the equal() method -

```
import numpy as np
a = np.array([1,2,3])
b = np.array([34,2,10])
out = np.equal(a,b)
print("The Boolean output of the equal function:",out)
```

Output: The following is the output of the equal function applied on the arrays and we can observe the Boolean output.

The Boolean output of the equal function: [False True False]

(2) **Using greater() Method:** The greater() method is used to compare any two NumPy arrays element-wise. This method returns True if the element in first array is greater than its corresponding element in the second array; and False, otherwise.

Ex. In this example, we are trying to compare elements of two NumPy arrays using the greater() method. The return values after each comparison are stored in another array.

```
import numpy as np
a = np.array([1,2,3])
b = np.array([34,2,10])
out = np.greater(a,b)
print("The Boolean output of the greater function:",out)
```

Output : The following is the output of the greater function when applied on the two arrays.

The Boolean output of the greater function: [False False False]

(3) **Using array_equal() Method:** The array_equal() method is used to compare two NumPy arrays element-wise to check whether both the arrays are equal or not. If both arrays are equal, the return value is True and if not, the return value is False.

Note: Both arrays are said to be equal only if all the elements in the arrays are equal.

Ex.: When we pass the two arrays of same shape and size to the array_equal() function then the output will be in Boolean format.

```
import numpy as np
a = np.array([1,2,3])
b = np.array([34,2,10])
out = np.array_equal(a,b)
print("The Boolean output of the array_equal function:",out)
```

Output: The following is the output of the array_equal() function.

The Boolean output of the array_equal function: False

(4) **Using allclose() Method:** The allclose() method compares two NumPy arrays element-wise and checks which element is close to each other.

Ex.: In this example if we pass the two input arrays to the allclose() function then it will gives true when the elements are close otherwise gives false.

```
import numpy as np
a = np.array([1,2,3])
b = np.array([3,4,2])
out = np.allclose(a,b)
print("The Boolean output of the allclose function:",out)
```

Output: The following is the output of the allclose() function and returns the Boolean output.

The Boolean output of the allclose function: False

Filtering NumPy Arrays

The following are the functions to perform the filtering operations on the arrays.

(i) **Boolean Indexing:** Boolean Indexing allows us to select elements from an array based on a Boolean condition, i.e. only those elements that satisfy the Boolean conditions are extracted from the array. This condition, which needs to be satisfied with the array elements, is also known as Boolean mask.

Ex.: In this example, we are trying to retrieve the filtered elements of an array using Boolean indexing. To filter elements, we will first create a Boolean mask, with a condition a > 2, then the elements satisfying this condition are extracted.

```
import numpy as np
a = np.array([34,2,10])
mask = a > 2
filtered_array = a[mask]
print("The output of the Boolean indexing:",filtered_array)
```

Output: The following is the output of the Boolean indexing applied on the array.

The output of the Boolean indexing: [34 10]

(ii) **Using where() Method:** The where() method is used to filter the elements from an array, based on a condition given by the user. It returns the indices of the array elements that satisfy the given condition.

Ex.: Following example shows how to filter elements in a NumPy array using the where() method. Here, we are passing a condition (a > = 2) as an argument to the where() method and only those value that satisfy this condition are displayed in the output array.

```
import numpy as np
a = np.array([34,2,10])
filtered_array = np.where(a < = 2)
print("The output of the where function :",filtered_array)
```

Output: The following is the output of the where function applied on the input arrays.

```
The output of the where function : (array([1]),)
```

(iii) **Using extract() Method:** The extract() method extracts all the elements that satisfy the given condition, as its name implies.

Ex.: Here, we are passing an array along with the condition as arguments to the extract() function. The elements in this array that satisfy the given condition are expected to be extracted.

```
import numpy as np
a = np.array([34,2,10])
filtered_array = np.extract(a < = 2, a)
print("The output of the extract function :",filtered_array)
```

Output: The below is the output of the extract function applied on the given input array.

```
The output of the extract function : [2]
```

(iv) **Using delete() Method:** The delete() method is used to delete elements from a NumPy array, based on a condition specified by the user.

The return value will be the array after deleting the elements in the given array depending on the condition. The condition argument we will use in this method is obtained from the where() method.

Ex.: Here in this example, when we pass the input array along with the condition (obtained from where() method) to the delete() method, the array elements that satisfy the given condition are deleted.

```
import numpy as np
a = np.array([34,2,10])
filtered_array = np.delete(a, np.where(a = = 2))
print("The output of the delete function :",filtered_array)
```


Output: The following is the output of the delete() function.

```
The output of the delete function : [34 10]
```

Module-IV

Q.1 Write about Matrix in Numpy.

Ans.

Binary Operations

Binary operators acts on bits and performs bit by bit operation. Binary operation is simply a rule for combining two values to create a new value.

Following are the functions for bitwise operations available in NumPy package.

	Operation	Description
(1)	bitwise_and	Computes bitwise AND operation of array elements
(2)	bitwise_or	Computes bitwise OR operation of array elements
(3)	invert	Computes bitwise NOT
(4)	left_shift	Shifts bits of a binary representation to the left
(5)	right_shift	Shifts bits of binary representation to the right

Ex.:

```
# Python program explaining
# bitwise_and() function

import numpy as geek
in_num1 = 10
in_num2 = 11

print ("Input number1 :", in_num1)
print ("Input number2 :", in_num2)

out_num = geek.bitwise_and(in_num1, in_num2)
print ("bitwise_and of 10 and 11 :", out_num)
```

Mathematical Functions

NumPy contains a large number of various mathematical operations. NumPy provides standard trigonometric functions, functions for arithmetic operations, handling complex numbers, etc.

Let's explore three different types of math functions in NumPy:

1. Trigonometric Functions
2. Arithmetic Functions
3. Rounding Functions

1. **Trigonometric Functions:** NumPy provides a set of standard trigonometric functions to calculate the trigonometric ratios (sine, cosine, tangent, etc.)

Here's a list of commonly used trigonometric functions in NumPy.

Trigonometric Function	Computes (in radians)
sin()	the sine of an angle
cos()	cosine of an angle
tan()	tangent of an angle
arcsin()	the inverse sine
arccos()	the inverse cosine
arctan()	the inverse tangent
degrees()	converts an angle in radians to degrees
radians()	converts an angle in degrees to radians

Let's see the examples.

```
import numpy as np
# array of angles in radians
angles = np.array([0, 1, 2])
print("Angles:", angles)
# compute the sine of the angles
sine_values = np.sin(angles)
print("Sine values:", sine_values)
# compute the inverse sine of the angles
inverse_sine = np.arcsin(angles)
print("Inverse Sine values:", inverse_sine)
```

Output

```
Angles: [0 1 2]
Sine values: [0.        0.84147098 0.90929743]
Inverse Sine values: [0.        1.57079633  nan]
```

In this example, the sin() and arcsin() functions calculate the sine and inverse sine values, respectively, for each element in the angles array.

The resulting values are in radians.

Now let's see the examples for degrees() and radians().


```
import numpy as np
# define an angle in radians
angle = 1.57079633
print("Initial angle in radian:", angle)
# convert the angle to degrees
angle_degree = np.degrees(angle)
print("Angle in degrees:", angle_degree)
        # convert the angle back to radians
        angle_radian = np.radians(angle_degree)
        print("Angle in radians (after conversion):", angle_radian)
```

Output

```
Initial angle in radian: 1.57079633
Angle in degrees: 90.0000001836389
Angle in radians (after conversion): 1.57079633
```

Here, we first initialized an angle in radians. Then we converted it to degrees using the degrees() function.

Similarly, we used radians() to convert the degrees back to radians.

2. Arithmetic Functions: NumPy provides a wide range of arithmetic functions to perform on arrays.

Here's a list of various arithmetic functions along with their associated operators:

Operation	Arithmetic Function	Operator
Addition	add()	+
Subtraction	subtract()	-
Multiplication	multiply()	*
Division	divide()	/
Exponentiation	power()	**
Modulus	mod()	%

Let's see the examples :

```
import numpy as np
first_array = np.array([1, 3, 5, 7])
second_array = np.array([2, 4, 6, 8])

# using the add() function
result2 = np.add(first_array, second_array)
print("Using the add() function:",result2)
```

Output:

Using the add() function : [3 7 11 15]

In the above example, first we created two arrays named: first_array and second_array. Then, we used the add() function to perform element-wise addition respectively.

3. **Rounding Functions:** We use rounding functions to round the values in an array to a specified number of decimal places.

Here's a list of commonly used NumPy rounding functions:

	Rounding Functions	Functions
(1)	round()	returns the value rounded to the desired precision
(2)	floor()	returns the values of array down to the nearest integer that is less than each element
(3)	ceil()	returns the values of array up to the nearest integer that is greater than each element.

Let's see an example %

```
import numpy as np
numbers = np.array([1.23456, 2.34567, 3.45678, 4.56789])

# round the array to two decimal places
rounded_array = np.round(numbers, 2)

print(rounded_array)
# Output: [1.23 2.35 3.46 4.57]
```

Here, we used the round() function to round the values of array numbers. Notice the line,

np.round(numbers, 2)

We've given two arguments to the round() function.

(i) **numbers :** the array whose values are to be rounded

(ii) **2 :** denotes the number of decimal places to which the array is rounded

Now, let's see the example of other NumPy rounding functions.

```
import numpy as np
array1 = np.array([1.23456, 2.34567, 3.45678, 4.56789])
print("Array after floor():", np.floor(array1))
print("Array after ceil():", np.ceil(array1))
```


Output:

Array after floor(): [1. 2. 3. 4.]

Array after ceil(): [2. 3. 4. 5.]

In the above example, the floor() function rounds the values of array1 down to the nearest integer that is less than or equal to each element.

Whereas, the ceil() function rounds the values of array1 up to the nearest integer that is greater than or equal to each element.

String Functions

The following functions are used to perform vectorized string operations for arrays of dtype numpy.string_ or numpy.unicode_. They are based on the standard string functions in Python's built-in library.

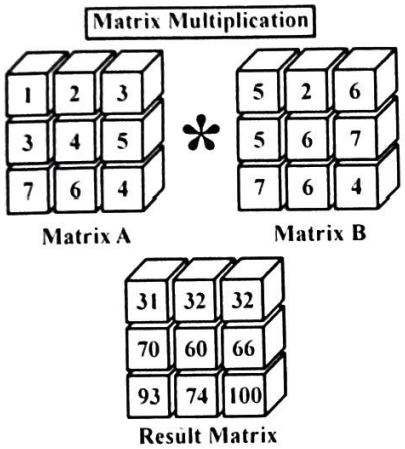
	Function	Description
(1)	add()	Returns element-wise string concatenation for two arrays of str or Unicode
(2)	multiply()	Returns the string with multiple concatenation, element-wise
(3)	center()	Returns a copy of the given string with elements centered in a string of specified length
(4)	capitalize()	Returns a copy of the string with only the first character capitalized
(5)	title()	Returns the element-wise title cased version of the string or unicode
(6)	lower()	Returns an array with the elements converted to lowercase
(7)	upper()	Returns an array with the elements converted to uppercase
(8)	split()	Returns a list of the words in the string, using separator delimiter
(9)	splitlines()	Returns a list of the lines in the element, breaking at the line boundaries
(10)	strip()	Returns a copy with the leading and trailing characters removed
(11)	join()	Returns a string which is the concatenation of the strings in the sequence
(12)	replace()	Returns a copy of the string with all occurrences of substring replaced by the new string

(13)	decode()	Calls str.decode element-wise
(14)	encode()	Calls str.encode element-wise

These functions are defined in character array class numpy.char). The older Numarray package contained chararray class. The above functions in numpy.char class are useful in performing vectorized string operations.

NumPy Matrix Multiplication in Python

Multiplication of matrix is an operation which produces a single matrix by taking two matrices as input and multiplying rows of the first matrix to the column of the second matrix. Note that we have to ensure that the number of rows in the first matrix should be equal to the number of columns in the second matrix.



In Python, the process of matrix multiplication using NumPy is known as vectorization. The main objective of vectorization is to remove or reduce the loops which we were using explicitly. By reducing 'for' loops from programs gives faster computation. The build-in package NumPy is used for manipulation and array-processing.

These are three methods through which we can perform numpy matrix multiplication.

1. First is the use of multiply() function, which perform element-wise multiplication of the matrix.
2. Second is the use of matmul() function, which performs the matrix product of two arrays.

3. Last is the use of the dot() function, which performs dot product of two arrays.

Ex. 1: Element-wise matrix multiplication

```
import numpy as np
array1 = np.array([[1,2,3],[4,5,6],[7,8,9]],ndmin=3)
array2 = np.array([[9,8,7],[6,5,4],[3,2,1]],ndmin=3)
result = np.multiply(array1,array2)
result
```

In the above code

(i) We have imported numpy with alias name np.

(ii) We have created an array1 and array2 using numpy.array() function with dimension 3.

(iii) We have created a variable result and assigned the returned value of np.multiply() function.

(iv) We have passed both the array array1 and array2 in np.multiply().

(v) Lastly, we tried to print the value of the result.

In the output, a three-dimensional matrix has been shown whose elements are the result of the element-wise multiplication of both array1 and array2 elements.

Output:

```
array([[[ 9, 16, 21),
        [24, 25, 24],
        [21, 16, 9]]])
```

Ex. 2: Matrix product

```
import numpy as np
array1 = np.array([[1,2,3],[4,5,6],[7,8,9]],ndmin = 3)
array2 = np.array([[9,8,7],[6,5,4),[3,2,1]],ndmin = 3)
result = np.matmul(array1,array2)
result
```

Output:

```
array([[[ 30, 24, 18],
        [84, 69, 54],
        [138, 114, 90]]])
```

In the above code

(i) We have imported numpy with alias name np.

(ii) We have created array1 and array2 using numpy.array() function with dimension 3.

(iii) We have created a variable result and assigned the returned value of the np.matmul() function.

(iv) We have passed both the array array1 and array2 in np.matmul().

(v) Lastly, we tried to print the value of the result.

In the output, a three-dimensional matrix has been shown whose elements are the product of both array1 and array2 elements.

Ex. 3: Dot product

These are the following specifications for numpy.dot:

(i) When both a and b are 1-D (one dimensional) arrays → Inner product of two vectors (without complex conjugation)

(ii) When both a and b are 2-D (two dimensional) arrays → Matrix multiplication

(iii) When either a or b is 0-D (also known as a scalar) → Multiply by using numpy.multiply (a, b) or a * b.

(iv) When a is an N-D array and b is a 1-D array → Sum product over the last axis of a and b.

) When a is an N-D array and b is an M-D array provided that M >= 2 → Sum product over the last axis of a and the second-to-last axis of b:

Also, dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

```
import numpy as np
array1 = np.array([[1,2,3],[4,5,6],[7,8,9]],ndmin =3)
array2 = np.array([[9,8,7],[6,5,4 ],[3,2,1]],ndmin=3)
result= np.dot(array1,array2)
result
```

In the above code

(i) We have imported numpy with alias name np.

(ii) We have created array1 and array2 using numpy. array() function with dimension 3.

(iii) We have created a variable result and assigned the returned value of the np.dot() function.

(iv) We have passed both the array array1 and array2 in np.dot().

(v) Lastly, we tried to print the value of the result.

In the output, a three-dimensional matrix has been shown whose elements are the dot product of both array1 and array2 elements.

Output:

```
array([[[[ 30, 24, 18]],
        [[ 84, 69, 54]],
```



```
        [[138, 114, 90]]])
```

NumPy Matrix Library

NumPy contains a matrix library, i.e. numpy.matlib which is used to configure matrices instead of ndarray objects.

numpy.matlib.empty() function

This function is used to return a new matrix with the uninitialized entries. The syntax to use this function is given below.

```
numpy.matlib.empty(shape, dtype, order)
```

It accepts the following parameter.

(1) **shape:** It is the tuple defining the shape of the matrix.

(2) **dtype:** It is the data type of the matrix.

(3) **order:** It is the insertion order of the matrix. i.e. C or F.

Consider the following example.

Ex.:

```
import numpy as np
import numpy.matlib
print(numpy.matlib.empty((3,3)))
```

Output:

```
[[6.90262230e-310 6.90262230e-310 6.90262304e-310]
[6.90262304e-310 6.90261674e-310 6.90261552e-310]
[6.90261326e-310 6.90262311e-310 3.95252517e-322]]
```

(i) **numpy.matlib.zeros() function:** This function is used to create the matrix where the entries are initialized to zero.

Consider the following, example:

Ex.:

```
import numpy as np
import numpy.matlib
print(numpy.matlib.zeros((4,3)))
```

Output:

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

(ii) **umpy.matlib.ones() function:** This function returns a matrix with all the elements initialized to 1.

Consider the following example :

Ex.:

```
import numpy as np
import numpy.matlib
```

```
print(numpy.matlib.ones((2,2)))
```

Output:

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

(iii) **numpy.matlib.eye() function:** This function returns a matrix with the diagonal elements initialized to 1 and zero elsewhere. The syntax to use this function is given below.

```
numpy.matlib.eye(n, m, k, dtype)
```

It accepts the following parameters.

1. **n:** It represents the number of rows in the resulting matrix.
2. **m:** It represents the number of columns, defaults to n.
3. **k:** It is the index of diagonal.
4. **dtype:** It is the data type of the output

Consider the following example:

Ex.:

```
import numpy as np
import numpy.matlib
print(numpy.matlib.eye(n = 3, M = 3, k = 0, dtype = int))
```

Output:

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

(iv) **numpy.matlib.identity() function:** This function is used to return an identity matrix of the given size. An identity matrix is the one with diagonal elements initializes to 1 and all other elements to zero.

Consider the following example.

Ex.:

```
import numpy as np
import numpy.matlib
print(numpy.matlib.identity(5, dtype = int))
```

Output:

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

(v) **numpy.matlib.rand() function:** This function is used to generate a matrix where all the entries are initialized with random values.

Consider the following example :

Ex.:

```
import numpy as np
import numpy.matlib
print(numpy.matlib.rand(3,3))
```

Output:

```
[[0.86201511 0.86980769 0.06704884]
[0.80531086 0.53814098 0.84394673]
[0.85653048 0.8146121  0.35744405]]
```

Addition of Two Matrices

We use + operator to add corresponding elements of two NumPy matrices.

```
import numpy as np
A = np.array([[2, 4], [5, -6]])
B = np.array([[9, -3], [3, 6]])
C = A + B     # element wise addition
print(C)
'''
```

Output:

```
[[11  1]
 [8  0]]
'''
```

Subtracting Matrices

For subtracting two matrices, we will use the subtract() function in NumPy which will subtract the two matrices and return the result.

Algorithm

Follow the algorithm to understand how to use this function.

Step 1: Import NumPy into the program
Step 2: Declare matrix 1 and give values
Step 3: Declare matrix 2 and give values
Step 4: Call the subtract() function
Step 5: Print the result

Program to subtract two matrices:

```
#subtract two matrices
import numpy

# matrix 1
```

```
A=[ [1, 2, 3], [3, 4, 5], [6, 7, 8] ]
# matrix 2
B=[ [5, 6, 7], [1, 2, 3], [5, 3, 8] ]
print("Result:")
print(numpy.subtract(A, B))
```

Output:

```
Result:
[[-4 -4 -4]
[2 2 2]
[1 4 0]]
```

Vector Multiplication

Vector multiplication can be performed in three ways:

(i) Scalar Multiplication

(ii) Dot Product Multiplication

iii) Cross Product Multiplication

dot() product of vectors: In Python, you can use the numpy.dot function o calculate the dot product of two vector arrays:

In this, we have taken two complex vectors and calculated their dot duct using the numpy.dot() function.

```
import numpy as np
vector_x = 5 + 2j
vector_y = 2 + 3j
product = np.dot(vector_x, vector_y)
print("Dot Product :", product)
```

Output of the above code:

```
Dot Product : (4 + 19j)
```

In Python 3.5 and later, you can also use the @ operator to calculate the dot product:

```
In [ ]:
import numpy as np

v = np.array([2,1])
s = np.array([-3,2])
d = v @ s
print (d)
```


Dot Product of two arrays

Python provides the numpy.dot() function to return the dot product of two arrays.

NumPy is the fundamental package for scientific computing with Python. It is a highly optimised library for numerical operations. The support of NumPy makes the task easier. Several libraries, like OpenCV, SciPy, and Matplotlib are available.

Syntax of numpy.dot()

```
numpy.dot(x, y, out=None)
```

Here, x and y are two input arrays. Both arrays should be 1-D or 2-D. The out is the output argument for 1-D array scalar to be returned. It returns a dot product of two arrays, x and y. If both the arrays 'x' and 'y' are 1-D arrays, the dot() function performs the inner product of vectors and returns a scalar. If both the arrays 'x' and 'y' are 2-D arrays, the dot() function performs the matrix multiplication. If 'x' is an N-dimensional array and 'y' is a 1-dimensional array, then the dot() function performs the sum-product over the last axis of x and y. For N-dimensional arrays, it is a sum product over the last axis of x and the second-last axis of y.

Ex.: numpy.dot() product of 1D array

In the given example, we have taken two numpy one-dimensional arrays and calculated their dot product using the numpy.dot() function. The output would be an inner product of these two vectors.

```
import numpy as np
#initialize arrays
A = np.array([5, 2, 9, 3])
B = np.array([6, 9, 1, 7])
#dot product
output = np.dot(A, B)

print(output)
```

Output of the above code:

```
78
```

Ex.: numpy.dot() product of 1D array

Here is another example of numpy dot product of two one-dimensional arrays.

```
import numpy.matlib
import numpy as np
x = np.array([11,23])
```

```
y = np.array([31,25])
print(np.dot(x,y))
```

Output of the above code:

```
916
```

Ex.: numpy.dot() product of 2D array

Here is an example of numpy.dot() of 2-D array. In this, we have taken two two-dimensional arrays and calculated their dot product using numpy.dot() function. The output would be a matrix multiplication of the two input arrays.

```
import numpy.matlib
import numpy as np
x = np.array([[11,23],[2,3]])
y = np.array([[31,25],[4,1]])
print(np.dot(x, y))
```

Output of the above code:

```
[[433 298]
[74 53]]
```

Ex.: numpy.dot() product of 3D array

Here is an example of numpy.dot() of two 3-D arrays.

```
import numpy.matlib
import numpy as np
x = np.array([[2,4,6],[3,1,4],[8,5,3]])
y = np.array([[3,7,2],[2,4,7],[3,5,1]])
print(np.dot(x, y))
```

Output of the above code :

```
[[32 60 38]
[23 45 17]
[43 91 54]]
```

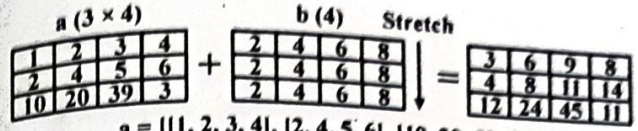
Broadcasting with NumPy Arrays

Broadcasting in NumPy is a very useful concept. The array size is a very important concept when performing arithmetic operations on arrays. Arrays cannot always be of the same shape.

We perform the array operations on corresponding array elements. Hence, we broadcast the smaller array along with the larger array. This is done to achieve similar dimensions.

This functionality helps to perform arithmetic operations with ease. We use it to broadcast the arrays to convert them into similar shapes.

Ex.:



a = [[1, 2, 3, 4], [2, 4, 5, 6], [10, 20, 39, 3]]
b = [[2, 4, 6, 8]]

The following rules are applicable on the arrays for broadcasting:

(1) If both the arrays do not have similar shapes, then we prepend the array of lower rank with 1s. This process continues until both the arrays have the same shape.

(2) We can consider the arrays compatible if they have the same length in a dimension or if one of the arrays has size as 1.

(3) Broadcasting is applicable if arrays are compatible in all the dimensions.

(4) The shape of the arrays turns out to be equivalent element-wise after we perform broadcasting.

(5) The smaller array appears as if it is a copy along the dimension.

(6) The length of each dimension is either the same or 1.

(a) Broadcasting array with the same shape

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b
```

Output :

```
array([ 2., 4., 6.])
```

(b) Broadcasting an array along with a scalar value

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

Output:

```
array([ 2., 4., 6.])
```

In both of the above examples, we have done broadcasting keeping in mind the broadcasting rules.

NumPy Broadcasting in 1-D array

Array broadcasting when one of the array dimensions are 1. The array with the smaller dimension is then broadcasted along the larger dimension.

```
import numpy as np
arr1 = np.array([56,78,94])
```

```
print(arr1)
arr2 = 5
print(arr2)
a=arr1+arr2
print(a)
```

Output

```
[56 78 94]
5
[61 83 99]
```

Here we carry out broadcasting over the smaller array. The broadcasting is due to the miss-match shape of the arrays.

NumPy Broadcasting in 2-D array

One of the arrays is a 2-dimensional array. We perform broadcasting on the smaller array.

```
import numpy as np
arr1 = np.array(([45,90,87],[56,78,94]))
print(arr1)
arr2 = 5
print(arr2)
a=arr1+arr2
print(a)
```

Output

```
[[45 90 87]
[56 78 94]]
5
[[50 95 92]
[61 83 99]]
```

Similarly we perform broadcasting on 3 or more dimensional arrays.

NumPy Broadcasting in 3-D array

We take a 3-dimensional array and broadcast along with a 1-D array.

```
import numpy as np
A = np.arange(24).reshape(2,3,4)
A
A.shape
B = np.arange(4)
B
```



```
B.shape
A+B
```

Output:

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

Universal Functions

A universal function (or ufunc) is a function that operates on ndarrays element by element, providing array broadcasting, type casting, and a variety of other standard features. A ufunc, in other words, is a "vectorized" wrapper around a function that accepts a fixed number of scalar inputs and returns a fixed number of scalar outputs.

They are simple mathematical functions in the NumPy library. Numpy includes a number of universal functions that support a wide range of operations.

These functions contain standard trigonometric functions, arithmetic operations, complex number handling, statistical functions, and so forth. The following are some of the characteristics of universal functions:

(i) These functions work with ndarray (N-dimensional array), which is Numpy's array class.

(ii) It provides quick array operations on elements.

(iii) It provides a variety of functions such as array broadcasting, type casting, and so on.

(iv) Numpy universal functions are objects in the numpy.ufunc class.

(v) Python functions can also be made universal by utilizing the from pyfunc library function.

(vi) When the corresponding array arithmetic operator is applied, some ufuncs are called automatically. When two arrays are added element by element using the '+' operator, np.add() is called internally.

NumPy universal functions are mathematical functions that allow vectorization. Vectorization refers to performing element-wise operations on arrays.

NumPy Universal Functions

The universal functions in NumPy include:

(i) Trigonometric functions like sin(), cos(), and tan()

(ii) Arithmetic functions like add(), subtract(), and multiply()

(iii) Rounding functions like floor(), ceil() and around()

(iv) Aggregation functions like mean(), min(), and max().

Creation of ufunc

To create your own ufunc, you have to define a function, like you do with normal functions in Python, then you add it to your NumPy ufunc library with the frompyfunc() method.

The frompyfunc() method takes the following arguments:

1. **function** : the name of the function.
2. **inputs** : the number of input arguments (arrays).
3. **outputs** : the number of output arrays.

Ex.: Create your own ufunc for addition:

```
import numpy as np

def myadd(x, y):
return x+y

myadd = np.frompyfunc(myadd, 2, 1)

print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
```

Simple Arithmetic

You could use arithmetic operators + – * / directly between NumPy arrays, but this section discusses an extension of the same where we have functions that can take any array-like objects e.g. lists, tuples etc. and perform arithmetic conditionally.

Arithmetic Conditionally: means that we can define conditions where the arithmetic operation should happen.

Addition: The add() function sums the content of two arrays, and return the results in a new array.

Ex.: Add the values in arr1 to the values in arr2:

```
import numpy as np

arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 =np.array([20, 21, 22, 23, 24, 25])

newarr = np.add(arr1, arr2)

print(newarr)
```


The example above will return [30 32 34 36 38 40] which is the sums of 10+20,11+21,12+22 etc.

Rounding Decimals

There are primarily five ways of rounding off decimals in NumPy:

(i) truncation

(ii) fix

(iii) rounding

(iv) floor

(v) ceil

(i) Truncation: Remove the decimals, and return the float number closest to zero. Use the trunc() and fix() functions.

Ex.: Truncate elements of following array:

```
import numpy as np
arr = np.trunc([-3.1666, 3.6667])
print(arr)
```

Ex.: Same example, using fix():

```
import numpy as np
arr = np.fix([-3.1666, 3.6667])
print(arr)
```

(ii) Rounding: The around() function increments preceding digit or decimal by 1 if > = 5 else do nothing.

E.g. round off to 1 decimal point, 3.16666 is 3.2

Ex.:

Round off 3.1666 to 2 decimal places:

```
import numpy as np
arr = np.around(3.1666, 2)

print(arr)
```

Floor

The floor() function rounds off decimal to nearest lower integer.

E.g. floor of 3.166 is 3.

Ex.: Floor the elements of following array:

```
import numpy as np
arr = np.floor([-3.1666, 3.6667])
print(arr)
```

Ceil: The ceil() function rounds off decimal to nearest upper integer.

E.g. ceil of 3.166 is 4.

Ex.: Ceil the elements of following array:

```
import numpy as np
arr = np.ceil([-3.1666, 3.6667])
print(arr)
```

Trigonometric Functions

NumPy provides the ufuncs sin(), cos() and tan() that take values in radians and produce the corresponding sin, cos and tan values.

Ex.: Trigonometric Functions

```
import numpy as np
# array of angles in radians
angles = np.array([0, 1, 2])
print("Angles:", angles)

# compute the sine of the angles
sine_values = np.sin(angles)
print("Sine values:", sine_values)
# compute the inverse sine of the angles
inverse_sine = np.arcsin(angles)
print("Inverse Sine values:", inverse_sine)
```

Output :

```
Angles: [0 1 2]
Sine values: [0.         0.84147098 0.90929743]
Inverse Sine values: [0.         1.57079633  nan]
```

In this example, we have used universal functions sin() and arcsin() to compute the sine and inverse sine values respectively.

When we perform the sin() function to the array angles, an element-wise operation is performed to entire elements of the array. This is called vectorization.

Hyperbolic Functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles. NumPy provides the ufuncs sinh(), cosh() and tanh() that take values in radians and produce the corresponding sinh, cosh and tanh values.

Ex.: Find sinh value of PI/2:

```
import numpy as np
x = np.sinh(np.pi/2)
print(x)
```


Ex.: Find cosh values for all of the values in arr:

```
import numpy as np
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])
x = np.cosh(arr)
print(x)
```

Finding Angles

Finding angles from values of hyperbolic sine, cos, tan. E.g. sinh, cosh and tanh inverse (arcsinh, arccosh, arctanh).

Numpy provides ufuncs arcsinh(), arccosh() and arctanh() that produce radian values for corresponding sinh, cosh and tanh values given.

Ex.:

Find the angle of 1.0:

```
import numpy as np

x = np.arcsinh(1.0)

print(x)
```

Set Functions

A set in mathematics is a collection of unique elements. Sets are used for operations involving frequent intersection, union and difference operations.

Create Sets in NumPy

We can use NumPy's unique() method to find unique elements from any array. E.g. create a set array, but remember that the set arrays should only be 1-D arrays.

Ex.: Convert following array with repeated elements to a set:

```
import numpy as np
arr = np.array([1, 1, 1, 2, 3, 4, 5, 5, 6, 7])
x = np.unique(arr)
print(x)
```

(1) **Finding Union:** To find the unique values of two arrays, use the union1d() method.

Ex.: Find union of the following two set arrays:

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([3, 4, 5, 6])
newarr = np.union1d(arr1, arr2)
```

```
print(newarr)
```

(2) **Finding Intersection:** To find only the values that are present in both arrays, use the intersect1d() method.

Ex.: Find intersection of the following two set arrays:

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([3, 4, 5, 6])

newarr = np.intersect1d(arr1, arr2, assume_unique=True)

print(newarr)
```

Note: the intersect1d() method takes an optional argument assume_unique, which if set to True can speed up computation. It should always be set to True when dealing with sets.

(3) **Finding Difference:** To find only the values in the first set that is not present in the seconds set, use the setdiff1d() method.

Ex.: Find the difference of the set1 from set2:

```
import numpy as np
set1 = np.array([1, 2, 3, 4])
set2 = np.array([3, 4, 5, 6])
newarr = np.setdiff1d(set1, set2, assume_unique=True)
print(newarr)
```

Note: the setdiff1d() method takes an optional argument assume_unique, which if set to True can speed up computation. It should always be set to True when dealing with sets.

(4) **Finding Symmetric Difference:** To find only the values that are not present in both sets, use the setxor1d() method.

Ex.: Find the symmetric difference of the set1 and set2:

```
import numpy as np

set1 = np.array([1, 2, 3, 4])
set2 = np.array([3, 4, 5, 6])

newarr = np.setxor1d(set1, set2, assume_unique=True)

print(newarr)
```


Module-V

Q.1 Write about Matplotlib in Python.

Ans. Matplotlib is an open-source 2D plotting library that helps in visualizing figures. Matplotlib is used in Python as it is a robust, free and easy library for data visualization. It is easy to learn and understand.

Matplotlib library is used for creating static, animated and interactive 2D-plots or figures in Python.

Matplotlib is the most popular plotting library for Python. It gives us control over every aspect of a figure. It supports interactive and non-interactive plotting and can save images in several output formats (PNG, PS and others). It was originally written by J.D. Hunter and is actively being developed by a full-fledged community. It is distributed under a BSD-Style Licence.

Matplotlib is the Python package/library used to create 2D graphs and plots using Python scripts. Pyplot is a module in Matplotlib which supports a very wide variety of graphs and plots, namely histogram, bar charts, pie charts, powerspectra, error charts, etc. It is used along with NumPy library to provide an effective open-source alternative environment for MatLab. MatLab is a proprietary programming platform, designed specifically for engineers and scientists, which allows the most natural expression of computational mathematics.

CTM: Matplotlib is a Python 2D plotting library for creating publication-quality figures. Pyplot is a module of matplotlib library (of Python) containing collection of methods which allows a user to create 2D plots and graphs easily and interactively.

Pyplot provides the state-machine interface to the plotting library in Matplotlib. It means that figures and axes are implicitly and automatically created to achieve the desired plot. For example, calling plot() function from pyplot module will automatically create the necessary figure and axes to achieve the desired plot. The pyplot interface is generally preferred for non-interactive plotting (i.e., scripting).

Each pyplot function makes some changes to a figure, like creating a figure, creating a plotting area in a figure, plotting some lines in a plotting area, decorating the plot with labels, etc.

CTM: A plot is a graphical representation technique for representing a data set, usually as a graph, showing the relationship between two or more variables.

NumPy

Another library which helps in the process of plotting graphs/ charts using pyplot is NumPy. NumPy stands for Numerical Python. NumPy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object and tools for working with these arrays.

Using NumPy, a developer can perform the following operations:

(i) Mathematical and logical operations on arrays.

(ii) Fourier transforms and routines for shape manipulation.

(iii) Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

We can install NumPy using the popular Python package installer, pip. Type the following command at the command prompt:

```
> C:\pip install numpy
```

NumPy will get installed onto your system and will be ready to be used.

Installing Matplotlib

Before we start plotting graphs in Matplotlib, it needs to be installed first. For the installation of Matplotlib, follow the steps listed below:

Step 1: Open cmd (command prompt) and run command prompt as an Administrator.

Step 2: Type cd\ to move to the root directory.

Step 3: Type: pip install matplotlib (with internet connection) as shown in the given window.

Note: Please ensure that the system has Python Shell installed before installing Matplotlib library.

Step 4: Installation of Matplotlib will start.

After the installation is successfully completed on the system, an appropriate message shall be displayed in the window.

After the successful installation of the above Library package (Matplotlib), we can plot various types of graphs in Python shell using pyplot methods.

Q.2 Write about Drawing Plots in Python.

Ans.

Basic Nomenclature of a Plot

Pyplot module of Matplotlib contains a collection of functions that can be used to work on a plot. The plot() function of the pyplot module is used to create a figure. A figure is the overall window where the outputs of pyplot functions are plotted. A figure contains a plotting area, legend, axis labels, ticks, title, etc., as explained.

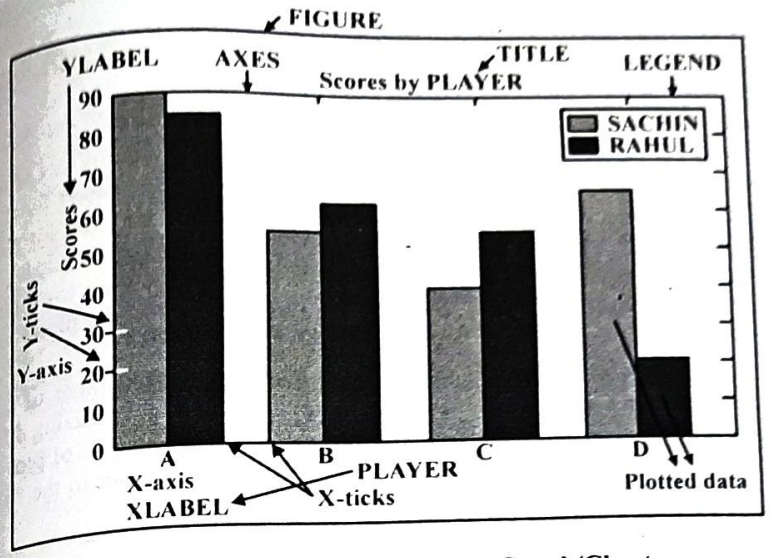


Fig. 1: Basic Components of a Graph/Chart

A Matplotlib figure can be categorized into several parts as below:

(i) **Figure:** It is the outermost area which may contain one or more than one axes (plots/subplots). You can think of a Figure as a canvas which contains plots, legend, axis labels, ticks, title, etc.

(ii) **Axes:** It is what we generally think of as a plot. A figure can contain many axes. It contains two or three (in the case of 3D) axis objects. Each axis has a title, an x-label and a y-label.

(iii) **Axis:** They are number line-like objects and take care of generating the graph limits.

(iv) **Artist:** Everything which one can see on the figure is an artist like Text objects, Line2D objects, collection objects. Most artists are tied to axes.

(v) **Labels:** To manage the axes dimensions of a plot, another important piece of information to add to a plot is the axes labels, since they usually specify what kind of data we are plotting.

(vi) **Title:** Just like in a book or a paper, the title of a graph describes what it is. Matplotlib provides a simple function, plt.title(), to add a title to an image.

(vii) **Legend:** Legends are used to explain different sets of data plotted in different colors or marks in the chart.

(viii) **xticks([ticks, labels]):** It sets the current tick locations and labels of the X-axis.

(ix) **yticks([ticks, labels]):** It sets the current tick locations and labels of the Y-axis.

Visualization in Pandas-Plot() Function

As Pandas is Python's popular data analysis library, it provides several different functions to visualize our data with the help of the .plot() function. This function directly creates the plot for the data set. Here, we plot a line using DataFrame's values as coordinates. However, starting from version 0.17.0, Pandas objects Series and DataFrame come equipped with their own .plot() methods. This plot() method is just a simple wrapper around the plot() function of pyplot.

The plot() method of Pandas accepts a considerable number of arguments that can be used to plot a variety of graphs. It allows customizing different plot types by supplying the kind keyword arguments. The type of plot you want to render can be specified by passing the "kind" parameter to the "plot" function. The "kind" parameter takes in the name of the plot.

Its syntax is:

```
df.plot(kind)
```

where kind accepts a string indicating the type of plot.

Therefore, for a line plot as an example, df.plot(kind='line') will be same as df.plot.line().

Plotting methods allow a handful of plot styles other than the default line plot. These methods can be provided as the kind keyword argument to plot(), Table 1 lists the various types of plots that can be passed as an argument:

Table 1: Different Arguments provided with 'kind' for different plots

kind =	Plot type
line	Line plot (By default)
bar	Vertical bar plot
barh	Horizontal bar plot
hist	Histogram
box	Boxplot
pie	Pie plot
scatter	Scatter plot
area	Area plot

We shall be using this plot() method in the subsequent sections for plotting various types of charts/plots using Matplotlib and Pandas DataFrame.

Line Plot/Chart

Line plot/chart is a type of plot which displays information as a series of data points called "markers" connected by straight lines. In this type of plot, we need the measurement points to be ordered (typically by their X-axis values). This type of plot is often used to visualize a trend in data over intervals of time - a time series.

The line chart is represented by a series of data points connected by a straight line. Generally, line charts are used to display trends over time. A line chart or line graph can be created using the plot() function available in pyplot module. We can not only just plot a line but also explicitly define the grid, the X-axis and Y-axis scale and labels, title and display options.

Its syntax is:

```
<matplotlib.pyplot>.plot (x-axis, y-axis, color, linewidth or lw,
linestyle or ls, marker, markersize, markeredgecolor)
```

Here,

X-axis, Y-axis	:	Points on X & Y axis in a form of sequence such as list, Series, etc.
color	:	Color of line. Should be a valid color identified by pyplot.
linewidth or lw	:	Width of line in points (float value)
linestyle or ls	:	Style of the line (Solid line, dashed line, etc.)
marker	:	A symbol denoting the marker style such as 'o', '*', '+', etc.
markersize	:	Marker size in points (float value)
markeredgecolor	:	Color or marker. Should be a valid color identified by pyplot.

To make a line plot with Matplotlib, we call plt.plot(). The first argument is used for the data on the horizontal axis and the second is used for the data on the vertical axis. This function generates your plot but it does not display it. To display the plot, we need to call the plt.show() function.

Markers and Line Styles

In the Practical Implementation examples that we are going to implement in successive sub-topics, all the plots are made of points with lines joining them. The points are the pairs (x, y) from the X and Y input lists we pass to plot(); lines are the straight segments connecting any two adjacent points.

Points are almost invisible, if not for the edges in the graph. However, they are the real generators of the plot because points mark positions. As a

result, they are called markers in Matplotlib terminology. A marker is any symbol that represents a data value in a line chart or a scatter plot.

By default, Matplotlib draws markers as a single dot and lines as straight thin segments; there are situations where we would like to change either the marker style (to clearly identify them in the plot) or the line style. Table 2 shows a list of markers along with their corresponding symbols and description.

Table 2: Matplotlib Markers with Symbols

Marker	Symbol	Description
"."	•	point
","	.	pixel
"o"	●	circle
"v"	▼	triangle_down
"^"	▲	triangle_up
"<"	◀	triangle_left
">"	▶	triangle_right
"1"	Y	tri_down
"2"	⅄	tri_up
"3"	<	tri_left
"4"	>	tri_right
"8"	●	octagon
"s"	■	square
"p"	⬟	pentagon
"P"	✚	plus (filled)
"*"	★	start
"h"	⬢	hexagon 1
"H"	⬢	hexagon 2
"+"	+	plus
"x"	×	x
"X"	✖	x (filled)
"D"	◆	diamond
"d"	♦	thin_diamond

"|"	|	vline
"_"	—	hline

CTM: A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.

In order to draw a line plot, the steps to be followed are as under:

Steps:

1. Importing pyplot module from Matplotlib library.
2. Call plt.plot(x, y, color, others) function to plot y versus x as lines and/or markers.
3. Call plt.xlabel("Your Text") to set the X-axis label of the current axes.
4. Call plt.ylabel("Your Text") to set the Y-axis label of the current axes.
5. Call plt.set_title("Your Title") to set a title of the current axes.
6. Call plt.show() to display a figure.

Interactive Navigation Toolbar

When using matplotlib.pyplot, the toolbar is enabled by default for every figure. It provides basic elaboration and manipulation functions for interactive plotting. At the bottom of the window, we can find the navigation toolbar. A description of each of its buttons (from left to right) follows:

Apart from the plot area, we have a few options displayed at the bottom-left corner of the window.



Reset Original Window: We can resize the plot window as per our requirement. It can be reset to the original window size using this option.

Back to Previous View: It takes us to the previous view from the current view.

Forward to Next View: It takes us to the next view from the current view.

Pan Axis with Left Mouse, Zoom with Right: Used to zoom any section of the plotted figure.

(i) **Pan:** Click on the left mouse button and hold it to pan the figure, dragging it to a new position. Once you are happy with the position, release the mouse button. While panning, if we press (or hold) the x or y key, then the panning is limited to the selected axis.

(ii) **Zoom:** Click on the right mouse button and hold it to zoom the figure, dragging it to a new position. Movement to the right or to the left generates a proportional zoom in or out of the X-axis of the figure. The same holds true for the up or down movement of the Y-axis. The point where we click the mouse remains still so that we are able to zoom around a given point in the figure. The x and y keys work in the same way, but now we can press the Ctrl key to preserve the aspect ratio.



Zoom to Rectangle: Enabling this mode, we can draw a rectangle on the figure (hold the left mouse button while drawing it) and the view will be zoomed to that rectangle.



Configure Subplots: When we click on this button, a window pops up that allows us to configure the various spaces that surround the figure (left, right, up, button, between).



Save the Figure: This option is used to save the figure drawn to the hard disk by giving a proper name to it. Click on this button and a save file dialog box will pop up that allows us to save the current figure.

This is useful because we might want to add some additional customizations to our plot. For example, we might want to add labels to the axis and title for the plot.

Multiple Plots

If we want to plot multiple lines in one chart, we can simply call the plot() function multiple times.

Multiple Views

In case we want to display multiple charts in the same window, we can use the subplot() function.

arange()

One more function that we have used in the above code is arange(). The NumPy arange() function (sometimes called np.arange) is a tool for creating numeric sequences in Python. It returns evenly spaced numeric values within an interval, stored as a NumPy array or we can say a list (i.e., an ndarray object).

The syntax for arange() is:

```
                                        The data type
The function name                       (optional)
      ↓                                     ↓
np.arange ( [ start, ] stop [, step, ] [, dtype ] )
            ↑       ↑       ↑
      The start of  The end  The "step"
      the interval  of the   between values
      (optional)    interval (optional)
```

Here,

(i) start (optional)

The start parameter indicates the beginning value of the range.

This parameter is optional, so if you omit it, it will automatically default to 0.

(ii) stop (required)

The stop parameter indicates the end of the range. Keep in mind that like all Python indexing, this value will not be included in the resulting range.

(iii) step (optional)

The step parameter specifies the spacing between values in the sequence.

This parameter is optional. If you don't specify a step value, by default, the step value will be 1.

(iv) dtype (optional)

The dtype parameter specifies the data type.

For example, to create a range of values from 0 to 8, in increments of 2, we will use the start position of 0 and a stop position of 8. To increment in steps of 2, we will set the step parameter to 2. Hence, the statement shall be:

```
np.arange(start = 0, stop = 8, step = 2)
```

This command shall create a sequence of numbers as shown:

0	2	4	6

The last value should be 8 but as per the syntax it is to be excluded and, hence, 6 will be taken as the last number to be displayed for the given range.

Changing Line Width, Colors and Styles

We have already seen that in a multiline plot, Matplotlib automatically chooses different colors for different lines. We are also free to choose them by ourselves. Matplotlib provides different styles, width and colors for line(s) we are plotting.

Q.3 Write about Data Visualization with Matplotlib in Python.

Ans. Data Visualization is the process of presenting data in the form of graphs or charts. It helps to understand large and complex amounts of data very easily. It allows the decision-makers to make decisions very efficiently and also allows them in identifying new trends and patterns very easily. It is also used in high-level data analysis for Machine Learning and Exploratory Data Analysis (EDA). Data visualization can be done with various tools like Tableau, Power BI, Python.

Here, we will discuss how to visualize data with the help of the Matplotlib library of Python.

Matplotlib

Matplotlib is a low-level library of Python which is used for data visualization. It is easy to use and emulates MATLAB like graphs and visualization. This library is built on the top of NumPy arrays and consist of several plots like line chart, bar chart, histogram, etc. It provides a lot of flexibility but at the cost of writing more code.

Pyplot

Pyplot is a Matplotlib module that provides a MATLAB-like interface. Matplotlib is designed to be as usable as MATLAB, with the ability to use Python and the advantage of being free and open-source. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. The various plots we can utilize using Pyplot are Line Plot, Histogram, Scatter, 3D Plot, Image, Contour, and Polar.

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the pit alias:

```
import matplotlib.pyplot as plt
```

Now the Pyplot package can be referred to as pit.

After knowing a brief about Matplotlib and pyplot let's see how to create a simple plot.

```
import matplotlib.pyplot as pit

# initializing the data
x = [10, 20, 30, 40]
```

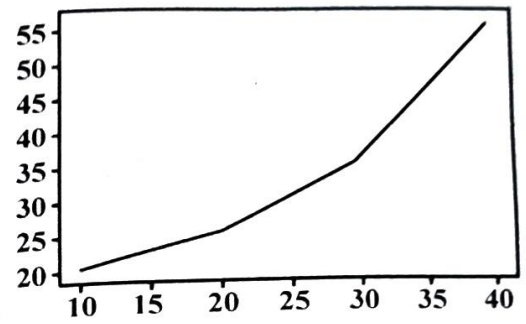


```
y = [20, 25, 35, 55]

# plotting the data
plt.plot(x, y)

plt.show()
```

Output:



Now let see how to ad0d some basic elements like title, legends, labels to the graph.

(1) Adding Title

The title() method in matplotlib module is used to specify the title of the visualization depicted and displays the title using various attributes.

Syntax:

```
matplotlib.pyplot.title(label, fontdict=None, loc= 'center',
pad=None, **kwargs)
```

Ex.:

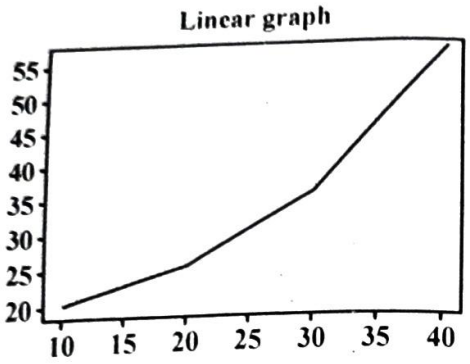
```
import matplotlib.pyplot as pit

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# plotting the data
plt.plot(x y)
```

```
# Adding title to the plot
plt.title("Linear graph")
plt.show()
```

Output:



We can also change the appearance of the title by using the parameters of this function.

(2) Adding X label and Y Label

In layman's terms, the X label and the Y label are the titles given to X-axis and Y-axis respectively. These can be added to the graph by using the xlabel() and ylabel() methods.

Syntax:

```
matplotlib.pyplot.xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)
matplotlib.pyplot.ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)
```

Ex.:

```
import matplotlib.pyplot as plt
# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# plotting the data
plt.plot(x, y)

# Adding title to the plot
```



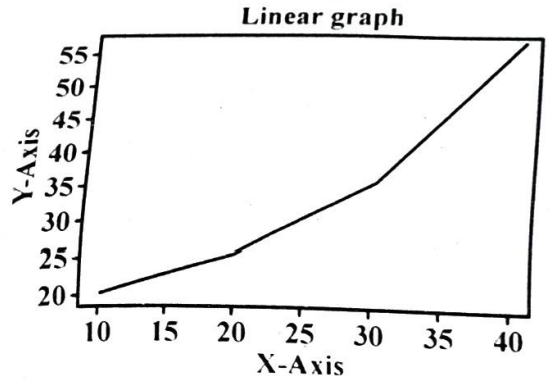
```
plt.title("Linear graph", fontsize=25, color="green")

# Adding label on the y-axis
pit.ylabel('Y-Axis')

# Adding label on the x-axis
plt.xlabel('X-Axis')

plt.show()
```

Output:



(3) Setting Limits and Tick Labels

You might have seen that Matplotlib automatically sets the values and the markers(points) of the X and Y axis, however, it is possible to set the limit and markers manually. xlim() and ylim() functions are used to set the limits of the X-axis and Y-axis respectively. Similarly, xticks() and yticks() functions are used to set tick labels.

Ex.: In this example, we will be changing the limit of Y-axis and will be setting the labels for X-axis.

```
import matplotlib.pyplot as plt
# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# plotting the data
```

```
plt.plot(x, y)

# Adding title to the plot
plt.title("Linear graph", fontsize=25, color="green")

# Adding label on the y-axis
plt.ylabel('Y-Axis')

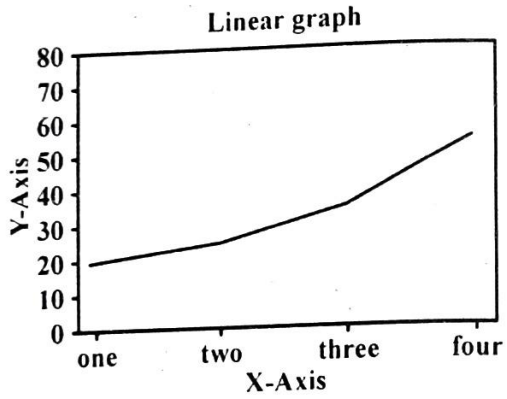
# Adding label on the x-axis
plt.xlabel('X-Axis')

# Setting the limit of y-axis
plt.ylim(0, 80)

# setting the labels of x-axis
plt.xticks(x, labels=["one", "two", "three", "four"])

plt.show()
```

Output:



(4) Adding Legends

A legend is an area describing the elements of the graph. In simple terms, it reflects the data displayed in the graph's Y-axis. It generally appears as the box containing a small sample of each color on the graph and a small description of what this data means.

The attribute bbox_to_anchor=(x, y) of legend() function is used to specify the coordinates of the legend, and the attribute ncol represents the number of columns that the legend has. Its default value is 1.

Syntax:

```
matplotlib.pyplot.legend(["name 1", "name2"], bbox_to_anchor=
(x, y), ncol = 1)
```

Ex.:

```
import matplotlib.pyplot as plt
# initializing the data
x [10, 20, 30, 40]
y = [20, 25, 35, 55]

# plotting the data
plt.plot(x, y)

# Adding title to the plot
plt.title("Linear graph", fontsize=25, color="green")

# Adding label on the y-axis
plt.ylabel('Y-Axis')

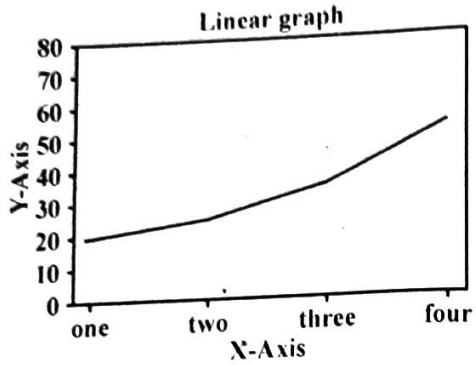
# Adding label on the x-axis
plt.xlabel('X-Axis')

# Setting the limit of y-axis
plt.ylim(0, 80)

# setting the labels of x-axis
plt.xticks(x, labels=["one", "two", "three", "four"])

# Adding legends
plt.legend(["GFG"])

plt.show()
```



Before moving any further with Matplotlib let's discuss some important classes. These classes are :

(i) **Figure class:** Consider the figure class as the overall window or page on which everything is drawn. It is a top-level container that contains one or more axes. A figure can be created using the figure() method.

Syntax:

```
class matplotlib.figure.Figure(figsize=None, dpi=None,
facecolor=None, edgecolor=None, linewidth=0.0, frameon =
None, subplotpars= None, tight_layout=None,
constrained_layout= None)
```

Ex.:

```
# Python program to show pyplot module
import matplotlib.pyplot as plt
from matplotlib.figure import Figure
# initializing the data
x       [10, 20, 30, 40]
y       [20, 25, 35, 55]
# Creating a new figure with width = 7 inches
# and height = 5 inches with face color as
# green, edgecolor as red and the line width
# of the edge as 7
fig = plt.figure(figsize =(7, 5), facecolor='g',
      edgecolor='b', linewidth=7)
# Creating a new axes for the figure
ax = fig.add_axes([1, 1, 1, 1])
```



```
# Adding the data to be plotted
ax.plot(x, y)

# Adding title to the plot
plt.title("Linear graph", fontsize= 25, color="yellow")

# Adding label on the y-axis
plt.ylabel('Y-Axis')

# Adding label on the x-axis
plt.xlabel('X-Axis')

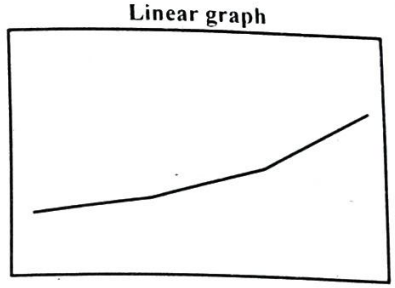
# Setting the limit of y-axis
plt.ylim(0, 80)

# setting the labels of x-axis
plt.xticks(x, labels= ["one", "two", "three", "four"])

# Adding legends
plt.legend(["GFG"])

plt.show()
```

Output:



(ii) **Axes Class:** Axes class is the most basic and flexible unit for creating sub-plots. A given figure may contain many axes, but a given axes can only be present in one figure. The axes() function creates the axes object.

Syntax:

axes([left, bottom, width, height])

Just like pyplot class, axes class also provides methods for adding titles, legends, limits, labels, etc. Let's see a few of them :

(i) **Adding Title :** ax.set_ title()

(ii) **Adding X Label and Y label :** ax.set_xlabel(), ax.set_ylabel()

(iii) **Setting Limits :** ax.set_xlim(), ax.set_ylim()

(iv) **Tick labels :** ax.set_xticklabels(), ax.set_yticklabels()

v) **Adding Legends :** ax.legend()

Ex.:

```
# Python program to show pyplot module.
import matplotlib.pyplot as plt
from matplotlib.figure import Figure

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

fig = plt.figure(figsize = (5, 4))

# Adding the axes to the figure
ax = fig.add_axes([1, 1, 1, 1])

# plotting 1st dataset to the figure
ax1 = ax.plot(x, y)

# plotting 2nd dataset to the figure
ax2 = ax.plot(y, x)

# Setting Title
ax.set_title("Linear Graph")

# Setting Label
ax.set_xlabel("x-Axis")
ax.set_ylabel("Y-Axis")

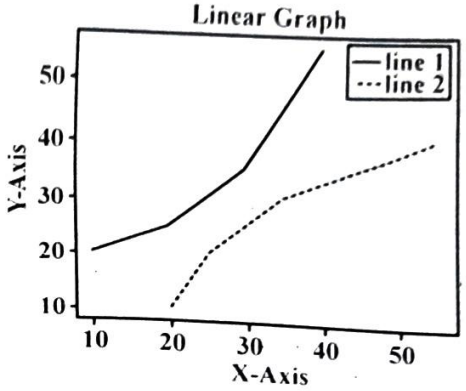
# Adding Legend
```



```
ax.legend(labels = ('line 1', 'line 2'))

plt.show()
```

Output:



Multiple Plots

We have learned about the basic components of a graph that can be added so that it can convey more information. One method can be by calling the plot function again and again with a different set of values as shown in the above example. Now let's see how to plot multiple graphs using some functions and also how to plot sub-plots.

Method 1: Using the add_axes() method

The add_ axes() method is used to add axes to the figure. This is a method of figure class

Syntax:

add_axes(self, *args, **kwargs)

Ex.:

```
# Python program to show pyplot module
import matplotlib.pyplot as plt
from matplotlib.figure import Figure

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# Creating a new figure with width = 5 inches
```

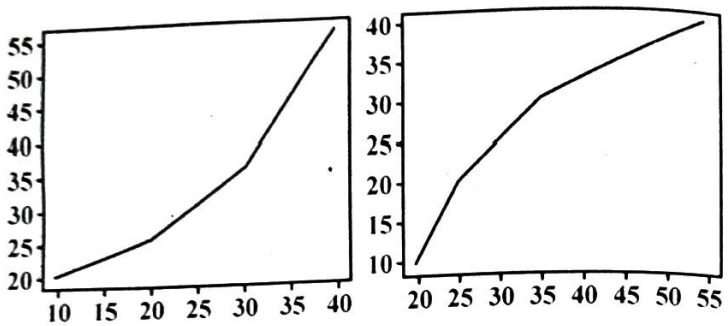
```
# and height = 4 inches
fig = plt.figure(figsize =(5, 4))

# Creating first axes for the figure
ax1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])

# Creating second axes for the figure
ax2 = fig.add_axes([1, 0.1, 0.8, 0.8])

# Adding the data to be plotted
ax1.plot(x, y)
ax2.plot(y, x)
plt.show()
```

Output:



Method 2: Using subplot() method.

This method adds another plot at the specified grid position in the current figure.

Syntax:

```
subplot(nrows, ncols, index, **kwargs)
subplot(pos, * *kwargs)
subplot(ax)
```

Ex.:

```
import matplotlib.pyplot as plt
# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]
# Creating figure object
```

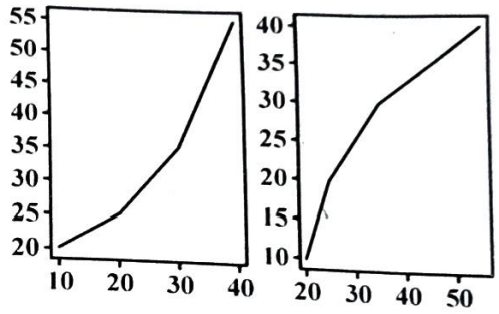


```
plt.figure()

# adding first subplot
plt.subplot(121)
plt.plot(x, y)

# adding second subplot
plt.subplot(122)
plt.plot(y, x)
```

Output:



Method 3: Using subplots() method

This function is used to create figures and multiple sub-plots at the same time.

Syntax:

```
matplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False,
sharey=False, squeeze=True, subplot_kw=None,
gridspec_kw=None, **fig_kw)
```

Ex.:

```
import matplotlib.pyplot as plt

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# Creating the figure and subplots
```

```
# according the argument passed
fig, axes = plt.subplots(1, 2)

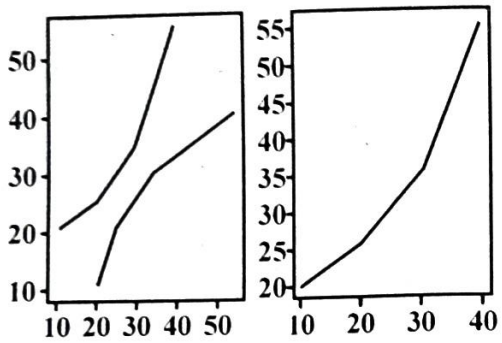
# plotting the data in the
# 1st subplot

axes[0].plot(x, y)

# plotting the data in the 1st
# subplot only
axes[0].plot(y, x)

# plotting the data in the 2nd
# subplot only
axes[1].plot(x, y)
```

Output:



Method 4: Using subplot2grid() method

This function creates axes object at a specified location inside a grid and also helps in spanning the axes object across multiple rows or columns. In simpler words, this function is used to create multiple charts within the same figure.

Syntax:

Plt.subplot2grid(shape, location, rowspan, colspan)

Ex.:

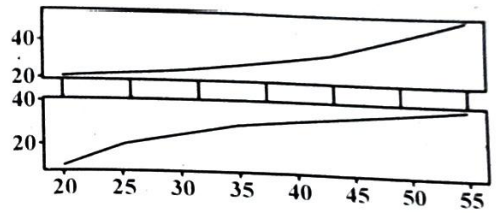
```
import matplotlib.pyplot as plt

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# adding the subplots
axes1 = plt.subplot2grid (
(7, 1), (0, 0), rowspan = 2, colspan = 1)

axes2 = plt.subplot2grid
((7, 1), (2, 0), rowspan = 2, colspan = 1)
# plotting the data
axes1.plot(x, y)
axes2.plot(y, x)
```

Output:



Different Types of Matplotlib Plots

Matplotlib supports a variety of plots including line charts, bar charts, histograms, scatter plots, etc. We will discuss the most commonly used charts here :

Note: Some elements like axis, color are common to each plot whereas some elements are pot specific.

(1) Line Chart

Line chart is one of the basic plots and can be created using the plot() function. It is used to represent a relationship between two data X and Y on a different axis.

Syntax:

matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, ***kwargs)

Ex.:

```
import matplotlib.pyplot as plt

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# plotting the data
plt.plot(x, y)

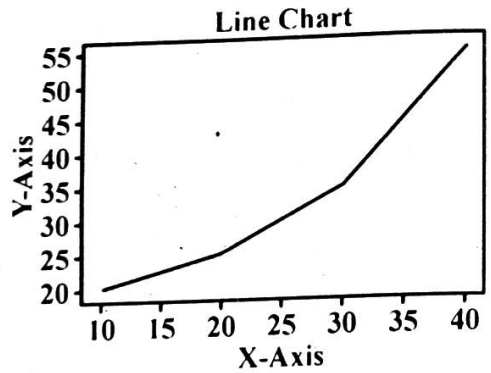
# Adding title to the plot
plt.title("Line Chart")

# Adding label on the y-axis
plt.ylabel('Y-Axis')

# Adding label on the x-axis
plt.xlabel('X-Axis')

plt.show()
```

Output:



Let's see how to customize the above-created line chart. We will be using the following properties :

(i) **color:** Changing the color of the line

(ii) **linewidth:** Customizing the width of the line

(iii) **marker:** For changing the style of actual plotted point

(iv) **markersize:** For changing the size of the markers

(v) **linestyle:** For defining the style of the plotted line

Different Linestyle Available

Character	Definition
-	Solid line
-	Dashed line
-.	dash-dot line
:	Dotted line
.	Point marker
O	Circle marker
,	Pixel marker
V	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
S	square marker
P	pentagon marker
*	star marker
H	hexagon1 marker
H	hexagon2 marker

+	Plus marker
X	X marker
D	Diamond marker
D	thin_diamond marker
|	vline marker
_	hline marker

Ex.: Python3

```
import matplotlib.pyplot as plt

# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

# plotting the data
plt.plot(x, y, color='green', linewidth=3, marker='o',
        markersize=15, linestyle='—')

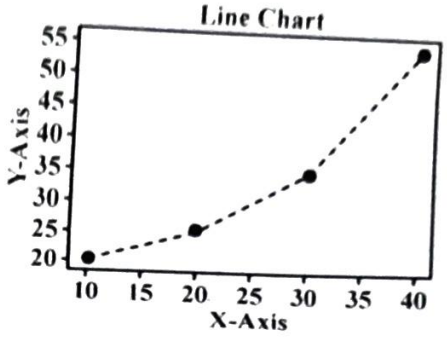
# Adding title to the plot
plt.title("Line Chart")

# Adding label on the y-axis
plt.ylabel('Y-Axis')

# Adding label on the x-axis
plt.xlabel('X-Axis')

plt.show()
```


Output:



(2) Bar Chart

A bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories. It can be created using the bar() method.

In the below example, we will use the tips dataset. Tips database is the record of the tip given by the customers in a restaurant for two and a half months in the early 1990s. It contains 6 columns as total_bill, tip, sex, smoker, day, time, size.

Ex.:

```
import matplotlib.pyplot as plt
import pandas as pd

# Reading the tips.csv file
data = pd.read_csv('tips.csv')

# initializing the data
x = data['day']
y = data['total_bill']

# plotting the data
```

```
plt.bar(x, y)

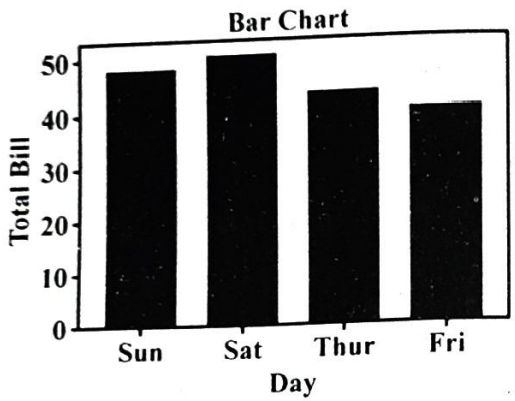
# Adding title to the plot
plt.title("Tips Dataset")

#Adding label on the y-axis
plt.ylabel('Total Bill')

# Adding label on the x-axis
plt.xlabel('Day')

plt.show()
```

Output:



Customization that is available for the Bar Chart:

(i) **color:** For the bar faces

(ii) **edgecolor:** Color of edges of the bar

(iii) **linewidth:** Width of the bar edges

(iv) **width:** Width of the bar

(3) Histogram

A histogram is basically used to represent data provided in a form of some groups. It is a type of bar plot where the X-axis represents the bin ranges

while the Y-axis gives information about frequency. The hist() function is used to compute and create histogram of x.

Syntax:

```
matplotlib.pyplot.hist(x, bins=None, range=None, density=False,
weights=None, cumulative = False, bottom = None, histtype =
'bar', align ='mid', orientation='vertical', rwidth=None,
log=False, color= None, label=None, stacked=False, \*,
data=None, \*\*kwargs)
```

Ex.:

```
import matplotlib.pyplot as plt
import pandas as pd

# Reading the tips.csv file
data = pd.read_csv('tips.csv')

# initializing the data
x = data['total_bill']

# plotting the data
plt.hist(x)

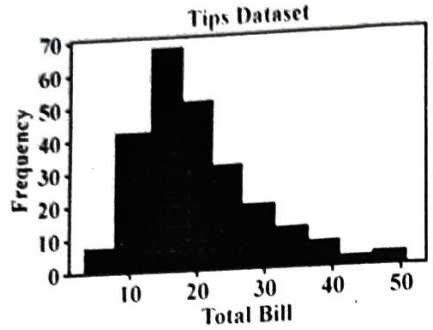
# Adding title to the plot
plt.title("Tips Dataset")

# Adding label on the y-axis
plt.ylabel('Frequency')

# Adding label on the x-axis
plt.xlabel('Total Bill')

plt.show()
```

Output:



Customization that is available for the Histogram:

(i) **bins:** Number of equal-width bins

(ii) **color:** For changing the face color

(iii) **edgecolor:** Color of the edges

(iv) **linestyle:** For the edgelines

(v) **alpha:** blending value, between 0 (transparent) and 1 (opaque)

(4) Scatter Plot

Scatter plots are used to observe relationships between variables. The scatter() method in the matplotlib library is used to draw a scatter plot.

Syntax:

matplotlib.pyplot.scatter(x_axis_data, y_axis_data, s=None, c=None, marker=None, cmap=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors=None

Ex.:

```
import matplotlib.pyplot as plt
import pandas as pd

# Reading the tips.csv file
data = pd.read_csv('tips.csv')

# initializing the data
x = data['day']
y = data['total_bill']
```



```
# plotting the data
plt.scatter(x, y)

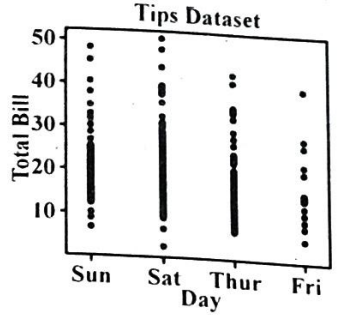
# Adding title to the plot
plt.title("Tips Dataset")

# Adding label on the y-axis
plt.ylabel('Total Bill')

# Adding label on the x-axis
plt.xlabel('Day')

plt.show()
```

Output:



Customizations that are available for the scatter plot are :

(i) **s:** marker size (can be scalar or array of size equal to size of x or y)

(ii) **c:** color of sequence of colors for markers

(iii) **marker:** marker style

(iv) **linewidths:** width of marker border

(v) **edgecolor:** marker border color

(vi) **alpha:** blending value, between 0 (transparent) and 1 (opaque)

(5) Saving a Plot

For saving a plot in a file on storage disk, savefig() method is used. A file can be saved in many formats like .png, .jpg, .pdf, etc.

Syntax:

```
pyplot.savefig(fname, dpi=None, facecolor='w', edgecolor='w',
orientation='portrait', papertype=None, format=None,
transparent= False, bbox_inches=None, pad_inches=0.1,
frameon=None, metadata= None)
```

Ex.:

```
import matplotlib.pyplot as plt

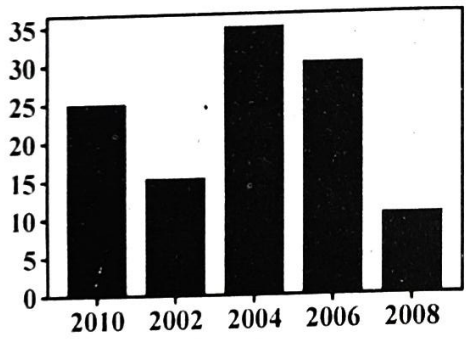
# Creating data
year = ['2010', '2002', '2004', '2006', '2008']
production = [25, 15, 35, 30, 10]

# Plotting barchart
plt.bar(year, production)

# Saving the figure.
plt.savefig("output.jpg")

# Saving figure by changing parameter values
plt.savefig("output1", facecolor= 'y', bbox _inches= "tight",
        pad_inches = 0.3, transparent=True)
```

Output:



(6) Adding Text to Plots

To add text inside a plot in Matplotlib, we can take the following steps:

(i) Set the figure size and adjust the padding between and around the subplots.

(ii) Create x and y data points using numpy.

(iii) Place text with some text properties.

(iv) Plot x and y using plot() method.

(v) Turn off the axes.

(vi) To display the figure, use show() method.

Ex.:

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

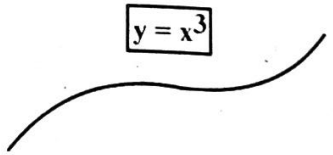
plt.rcParams["figure.figsize"] = [7.50, 3.50]
plt.rcParams["figure.autolayout"] = True

x = np.linspace(-5, 5, 100)
y = x ** 3

plt.text(0, 100, '$y=x ^ 3$', fontsize=22,
bbox=dict(facecolor='red', alpha=0.5))
plt.plot(x, y, c='g')
plt.axis('off')

plt.show()
```

Output:



(7) Pie Chart

Pie chart is a circular chart used to display only one series of data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges. It can be created using the pie() method.

Syntax:

```
matplotlib.pyplot.pie(data, explode=None, labels=None,
colors=None, autopct=None, shadow=False)
```

Ex.:

```
import matplotlib.pyplot as plt
import pandas as pd

# Reading the tips.csv file
data = pd.read_csv('tips.csv')

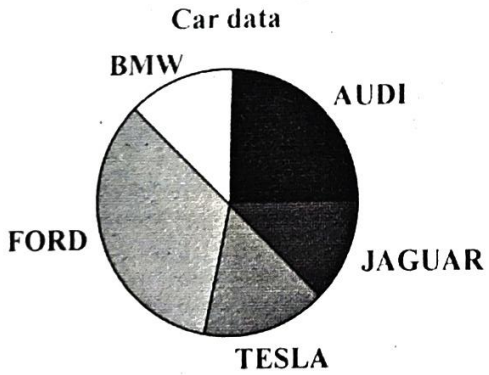
# initializing the data
cars = ['AUDI', 'BMW', 'FORD',
        'TESLA', 'JAGUAR',]
data = [23, 10, 35, 15, 12]

# plotting the data
plt.pie(data, labels=cars)

# Adding title to the plot
plt.title("Car data")

plt.show()
```

Output:



Customizations that are available for the Pie chart are:

(i) **explode:** Moving the wedges of the plot

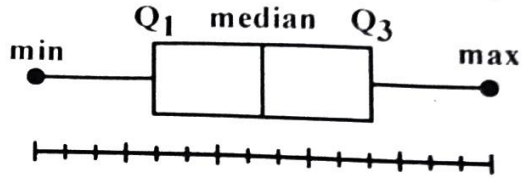
(ii) **autopct:** Label the wedge with their numerical value.

(iii) **color:** Attribute is used to provide color to the wedges

(iv) **shadow:** Used to create shadow of wedge.

(8) Box Plot

A box plot which is also known as a whisker plot displays a summary of a set of data containing the minimum, first quartile, median, third quartile, and maximum. In a box plot, we draw a box from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.



Let us create the data for the boxplots. We use the numpy.random.normal() function to create the fake data. It takes three arguments, mean and standard deviation of the normal distribution and the number of values desired.

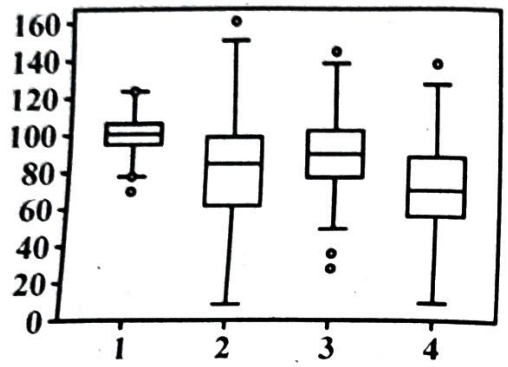
```
np.random.seed(10)
collectn_1 = np.random.normal(100, 10, 200)
collectn_2 = np.random.normal(80, 30, 200)
collectn_3 = np.random.normal(90, 20, 200)
collectn_4 = np.random.normal(70, 25, 200)
```

The list of arrays that we created above is the only required input for creating the boxplot. Using the data_to_plot line of code, we can create the boxplot with the following code :

```
fig = plt.figure()
# Create an axes instance
ax = fig.add_axes([0,0,1,1])
```

```
# Create the boxplot
bp = ax.boxplot(data_to_plot)
plt.show()
```

The above line of code will generate the following output:



Visualizing 3D Data with Matplotlib

Matplotlib was initially devèloped with only two-dimension plot. Its 1.0 release was built with some of three-dimensional plotting utilities on top of two-dimension display, and the result is a convenient set of tools for 3D data visualization.

Three-dimension plots can be created by importing the mplot3d toolkit, include with the main Matplotlib installation:

```
from mpl_toolkits import mplot3d
```

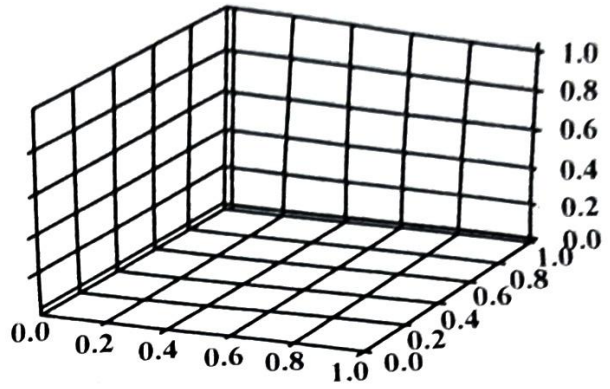
When this module is imported in the program, three-dimension axes can be created by passing the keyword projection ='3d' to any of the normal axes creation routines:

Let's see the simple 3D plot

Ex.:

```
from mpltoolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')_
```


Output:



3D Contour Plot

The ax.contour3D() function creates three-dimensional contour plot. It requires all the input data to be in the form of two-dimensional regular grids, with the Z-data evaluated at each point. Here, we will show a three-dimensional contour diagram of a three-dimensional sinusoidal function.

```
from mpl_toolkits import mplot3d
import numpy as np

import matplotlib.pyplot as plt
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

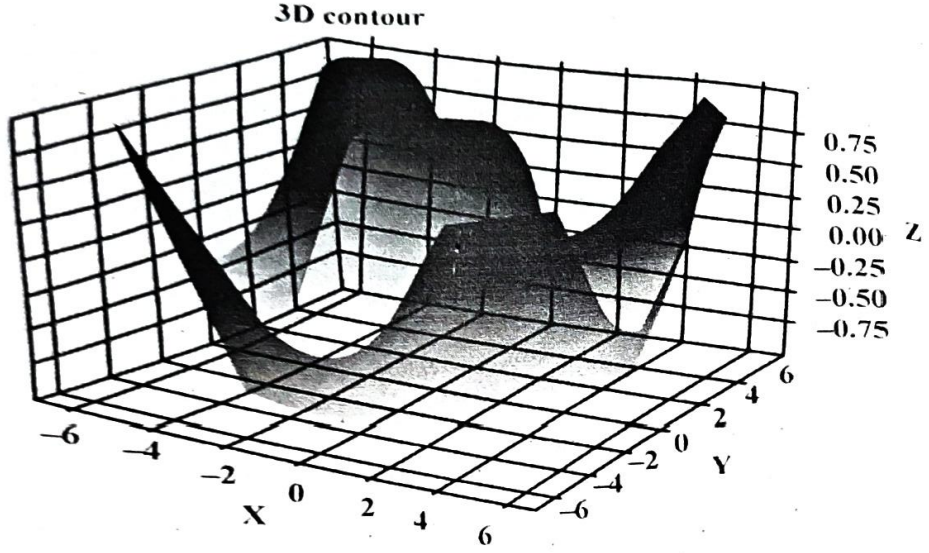
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)

fig = plt.figure()

ax = plt.axes(projection='3d')
```

```
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('3D contour')
plt.show()
```



Interactive Visualization using Widgets

Matplotlib is a popular library in Python for creating static, animated, and interactive visualizations. However, to make these visualizations more interactive, we can use a library called ipywidgets.

IPyWidget is an open-source Python library that is used to create widgets that can be helpful in controlling the graphs or data and make it interactive.