Study Material

Brief History of Python:

Python is one of the most popular programming languages today, known for its simplicity and versatility. Here is a concise timeline and background of Python's history:

1. Origins:

Creator: Python was created by Guido van Rossum in the late 1980s.

Development: It was developed during his time at Centrum Wiskunde & Informatica (CWI) in the Netherlands.

Inspiration: Guido wanted to create a language that was easy to learn and use.

Capable of handling various use cases, from scripting to software development.

Inspired by the ABC language (a teaching language) and C, with additional features.

2. Naming:

Contrary to popular belief, Python was not named after the snake.

Guido named it after the British comedy show **"Monty Python's Flying Circus"**, reflecting his desire for the language to be fun and approachable.

3. Key Milestones:

Year	Milestone
1991	Python 0.9.0 released. Included features like functions, exception handling, and core data types (str, list, dict).
1994	Python 1.0 released. Added modules, classes, and basic system calls.
2000	Python 2.0 released. Introduced list comprehensions and garbage collection based on reference counting.
2008	Python 3.0 released. A major overhaul aimed at improving readability and consistency, but not backward compatible with Python 2.
2010	Python overtook several languages in popularity due to its use in web development, data science, AI, and education.
2020	Official end of support for Python 2.x. The focus shifted entirely to Python 3.x.

4. Philosophy:

Python’s design philosophy emphasizes:

Readability: Code should be easy to read and understand.

Simplicity: "There should be one and preferably only one obvious way to do it."

Versatility: Python supports multiple programming paradigms (procedural, object-oriented, functional).

This philosophy is summarized in The Zen of Python, accessible by typing:

import this

5. Growth and Ecosystem:

Python's popularity surged due to:

- Web Development: Frameworks like Django and Flask.
- Data Science: Libraries like NumPy, pandas, and matplotlib.
- AI and Machine Learning: Tools like TensorFlow, PyTorch, and scikit-learn.
- Community Support: A large, active community of developers and contributors.

6. Current Status:

- Python is one of the most widely used programming languages globally.
- It is a staple in:
  1) Data Science
  2) Artificial Intelligence
  3) Web Development
  4) Education and more.
- Continues to evolve with new features and updates in the Python 3.x series.

Study Material: Data Types in Python

Python provides a variety of built-in data types that allow developers to handle different kinds of data effectively. Here's a brief overview:

1. Numeric Types

a. int (Integer)
Used to represent whole numbers.
Example:

```
x = 10
print(type(x))
# Output: <class 'int'>
```

b. float (Floating Point)

Used to represent decimal numbers.
Example:

```
y = 3.14
print(type(y))
# Output: <class 'float'>
```

c. complex

Represents complex numbers with a real and imaginary part.
Example:

```
z = 2 + 3j
```

```
print(type(z))
# Output: <class 'complex'>
```

d. eval()

The **eval()** function in Python is used to evaluate expressions dynamically, meaning it can execute Python expressions passed to it as a string. It interprets and executes the code or expression in real-time.

Examples

Basic Evaluation

```
result = eval("2 + 3")
print(result)
# Output: 5
```

Using Variables

```
x = 10
y = 20
result = eval("x + y")
print(result)
# Output: 30
```

Using Functions

```
result = eval("len('Hello')")
print(result)
# Output: 5
```

2. Sequence Types

a. str (String)

A collection of characters enclosed in single, double, or triple quotes.
Example:

```
s = "Hello, Python!"
print(type(s))  # Output: <class 'str'>
```

b. list

An ordered collection of items, mutable and enclosed in square brackets.
Example:

```
my_list = [1, 2, 3, "Python"]
print(type(my_list))
# Output: <class 'list'>
```

c. tuple

An ordered collection of items, immutable and enclosed in parentheses.
Example:
```
my_tuple = (1, 2, 3, "Python")
print(type(my_tuple))
# Output: <class 'tuple'>
```

3. Mapping Type

dict (Dictionary)

An unordered collection of key-value pairs.
Example:
```
my_dict = {"name": "Python", "version": 3.9}
print(type(my_dict))
# Output: <class 'dict'>
```

4. Set Types

a. set

An unordered collection of unique items.
Example:
```
my_set = {1, 2, 3, 3}
print(type(my_set))
# Output: <class 'set'>
print(my_set)
# Output: {1, 2, 3}
```

b. frozenset

An immutable version of a set.
Example:
```
frozen = frozenset([1, 2, 3])
print(type(frozen))
# Output: <class 'frozenset'>
```

5. Boolean Type

Represents one of two values: `True` or `False`.
Example:
```
is_active = True
print(type(is_active))
# Output: <class 'bool'>
```

6. Binary Types

a. bytes

Immutable sequence of bytes.

Example:

```
b = b"Hello"
print(type(b))
# Output: <class 'bytes'>
```

```
b = b"Hello"  # Prefix 'b' indicates a bytes object
print(type(b))  # Output: <class 'bytes'>

# Access elements
print(b[0])  # Output: 72 (ASCII value of 'H')

# Slicing
print(b[1:4])  # Output: b'ell'

# Immutable behavior
b[0] = 74  # TypeError: 'bytes' object does not support item assignment
```

b. bytearray

Mutable sequence of bytes.

Example:

```
ba = bytearray(5)
print(type(ba))
# Output: <class 'bytearray'>
```

```
ba = bytearray(5)  # Creates a bytearray of size 5 initialized with zeros
print(type(ba))  # Output: <class 'bytearray'>
print(ba)  # Output: bytearray(b'\x00\x00\x00\x00\x00')

# Modify an element
ba[0] = 65  # Set first element to ASCII value of 'A'
print(ba)  # Output: bytearray(b'A\x00\x00\x00\x00')

# Append more bytes
ba.append(66)  # Add ASCII value of 'B'
print(ba)  # Output: bytearray(b'A\x00\x00\x00\x00B')
```

c. memoryview

Allows byte-level manipulation of binary data.

Example:

```
mv = memoryview(bytes(5))
print(type(mv))
# Output: <class 'memoryview'>
```

```
# Create a memoryview from bytes
mv = memoryview(bytes(5))
print(type(mv))  # Output: <class 'memoryview'>

# Access elements
print(mv[0])  # Output: 0 (default value)

# Slice
print(mv[:3])  # Output: <memory at 0x...>

# Modify with a mutable buffer
ba = bytearray(b"Hello")
mv = memoryview(ba)
mv[0] = 74  # Modify first byte (ASCII for 'H' -> 'J')
print(ba)  # Output: bytearray(b'Jello')
```

7. None Type

Represents the absence of a value or a null value.

Example:

```
x = None
print(type(x))  # Output: <class 'NoneType'>
```

Type Conversion

Python allows conversion between types using built-in functions like int(), float(), str(), etc.

Example:

```
a = "123"
b = int(a)
print(type(b))
# Output: <class 'int'>
```

OPERATORS

Python Operators: A Comprehensive Guide

Python operators are symbols or keywords used to perform operations on variables and values. Operators are broadly classified into several categories based on their functionality.

1. Arithmetic Operators

Used to perform mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2.5
//	Floor Division	5 // 2	2
%	Modulus (remainder)	5 % 2	1
**	Exponentiation	5 ** 3	125

2. Comparison (Relational) Operators

Used to compare values. These return a Boolean result (True or False).

Operator	Description	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	3 <= 5	True

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example	Result
and	Logical AND	True and False	False
or	Logical OR	True or False	True
not	Logical NOT (negation)	not True	False

4. Bitwise Operators

Perform operations at the bit level.

Operator	Description	Example	Result
&	Bitwise AND	5 & 3	1 (0101 & 0011)
`	`	Bitwise OR	`5
^	Bitwise XOR	5 ^ 3	6 (0101 ^ 0011)
~	Bitwise NOT	~5	-6 (inverts bits)

<<	Left shift	5 << 1	10 (0101 → 1010)
>>	Right shift	5 >> 1	2 (0101 → 0010)

5. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Equivalent To
=	Assign	x = 5	x = 5
+=	Add and assign	x += 3	x = x + 3
-=	Subtract and assign	x -= 3	x = x - 3
*=	Multiply and assign	x *= 3	x = x * 3
/=	Divide and assign	x /= 3	x = x / 3
//=	Floor divide and assign	x //= 3	x = x // 3
%=	Modulus and assign	x %= 3	x = x % 3
**=	Exponentiate and assign	x **= 3	x = x ** 3
&=	Bitwise AND and assign	x &= 3	x = x & 3
`	=`	Bitwise OR and assign	`x
^=	Bitwise XOR and assign	x ^= 3	x = x ^ 3
<<=	Left shift and assign	x <<= 3	x = x << 3
>>=	Right shift and assign	x >>= 3	x = x >> 3

6. Identity Operators

Used to check whether two variables point to the same object in memory.

Operator	Description	Example	Result
is	Returns True if objects are the same	x is y	True/False
is not	Returns True if objects are different	x is not y	True/False

7. Membership Operators

Used to check for membership in sequences like lists, strings, or tuples.

Operator	Description	Example	Result
in	Returns True if found	"a" in "apple"	True
not in	Returns True if not found	"x" not in "apple"	True

8. Special Operators

Ternary Operator

Used for inline conditional assignments.

```
x = 10
y = 20
result = "x is greater" if x > y else "y is greater"
print(result)
# Output: "y is greater"
```

Operator Precedence Table (Highest to Lowest)

Precedence	Operator(s)	Description	Associativity
1	()	Parentheses (used to group expressions)	-
2	**	Exponentiation	Right-to-left
3	+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT	Right-to-left
4	*, /, //, %	Multiplication, Division, Floor Division, Modulus	Left-to-right
5	+, -	Addition, Subtraction	Left-to-right
6	<<, >>	Bitwise shift operators	Left-to-right
7	&	Bitwise AND	Left-to-right
8	^	Bitwise XOR	Left-to-right
9	`	`	Bitwise OR
10	in, not in, is, is not, <, <=, >, >=, ==, !=	Comparisons, Identity, Membership operators	Left-to-right
11	not	Logical NOT	Right-to-left
12	and	Logical AND	Left-to-right
13	or	Logical OR	Left-to-right
14	if-else	Conditional (ternary) operator	Right-to-left
15	=, +=, -=, *=, etc.	Assignment operators	Right-to-left
16	lambda	Lambda expression	Right-to-left

if-else in Python

The if-else statement in Python is used for conditional execution of code based on whether a given condition is True or False.

Syntax:

```
if condition:
    # Code block executed if the condition is True
else:
    # Code block executed if the condition is False
```

Key Points:

Condition:

It must evaluate to a Boolean value (True or False).

Python treats any non-zero number, non-empty string, or non-empty collection as True.

Indentation:

The body of if and else must be indented.

Consistent indentation is crucial in Python to define code blocks.

Single Statement:

For simple conditions, the body of if or else can be written in a single line using the conditional expression format.

Examples

Basic Example:

```
age = 20
if age >= 18:
   print("You are an adult.")
else:
   print("You are a minor.")
```

Output:

You are an adult.

Using Multiple Conditions:

```
score = 75
if score >= 90:
   print("Grade: A")
elif score >= 75:
   print("Grade: B")
else:
   print("Grade: C")
```

Output:

Grade: B

Nested if-else:

```
num = 5
if num > 0:
   if num % 2 == 0:
      print("Positive Even Number")
   else:
      print("Positive Odd Number")
else:
   print("Not a Positive Number")
```

Output:

Positive Odd Number

Inline if-else (Ternary Operator):

```
x = 10
result = "Even" if x % 2 == 0 else "Odd"
print(result)
```

Output:

Even

Handling Empty Collections:

```
my_list = []
if my_list:
    print("The list is not empty.")
else:
    print("The list is empty.")
```
Output:
The list is empty.

Practical Application:
Example: Checking Eligibility
```
age = 17
citizen = True

if age >= 18 and citizen:
    print("Eligible to vote.")
else:
    print("Not eligible to vote.")
```
Output:
Not eligible to vote.

Example: Categorizing a Number
```
number = -3
if number > 0:
    print("Positive")
elif number < 0:
    print("Negative")
else:
    print("Zero")
```
Output:
Negative

Conclusion:

- The if-else statement is a fundamental building block for decision-making in Python programs.
- Use it to direct the flow of your program based on conditions.

For Loop and While Loop in Python: A Brief Overview

Loops in Python allow you to execute a block of code repeatedly. The two primary types of loops in Python are the **for loop** and the **while loop**.

1. For Loop:
A for loop in Python is used to iterate over a sequence (such as a list, tuple, string, or range) or any iterable object.
Syntax:

```
for variable in sequence:
    # Code block to execute
```

Key Features:

- Iterates over each element in the sequence.
- Stops when the sequence is exhausted.
- The loop variable takes the value of each element in the sequence.

2. While Loop:

A while loop in Python repeatedly executes a block of code as long as the given condition evaluates to True.

Syntax:

```
while condition:
    # Code block to execute
```

Key Features:

- Runs until the condition becomes False.
- Requires a condition to be explicitly updated (to avoid infinite loops).

Differences Between For and While Loops

Feature	for Loop	while Loop
Use Case	Iterating over a sequence or range	Repeated execution based on a condition
Iteration Control	Handles iteration automatically	Requires manual condition management
Common Use	Known number of iterations	Unknown or conditional iterations

Examples of for **and** while **Loops**

1. For Loop Examples

Iterating Over a List

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

Using Range

```
for i in range(1, 6):
    print(i)
```

Output:

```
1
2
3
4
5
```

Iterating Over a String

```
word = "Python"
for char in word:
```

```
    print(char)
```
Output:
```
P
y
t
h
o
n
```

2. While Loop Examples

Basic Example
```
count = 1
while count <= 5:
    print(count)
    count += 1
```
Output:
```
1
2
3
4
5
```

Using a Break Statement
```
number = 1
while True:
    print(number)
    if number == 3:
        break
    number += 1
```
Output:
```
1
2
3
```
Using a Continue Statement
```
num = 0
while num < 5:
    num += 1
    if num == 3:
        continue
    print(num)
```
Output:
```
1
2
4
5
```
Summary:

- Use for loops when iterating over a sequence or when the number of iterations is predetermined.
- Use while loops when you need to repeat a task based on a condition, especially if the number of iterations is uncertain.
- Both loops can use **break** to exit early or **continue** to skip the current iteration.

Range Function in Python

The range() function in Python is used to generate a sequence of numbers. It is often used in loops for iterating over a range of values.

Syntax
range(start, stop, step)

start: The starting number of the sequence (inclusive). Defaults to 0 if not specified.
stop: The ending number of the sequence (exclusive). Required.
step: The increment (or decrement) between numbers. Defaults to 1.

Key Features:
- **Lazy Evaluation**: range() does not generate the sequence all at once. It produces numbers on demand, making it memory efficient.
- **Immutable**: The sequence generated by range() cannot be modified.
- **Versatility**: Works with positive and negative steps.

Examples
1. Basic Usage
```
for i in range(5):  (start=0, stop=5, step=1)
   print(i)
```
Output:
0
1
2
3
4
2. Specifying Start and Stop
```
for i in range(2, 7):  (start=2, stop=7, step=1)
   print(i)
```
Output:
2
3
4
5
6
3. Using a Step
```
for i in range(0, 10, 2):  (start=0, stop=10, step=2)
   print(i)
```

Output:
0
2
4
6
8

4. Using Negative Step

```
for i in range(5, 0, -1):  (start=5, stop=0, step=-1)
   print(i)
```

Output:
5
4
3
2
1

Converting range() to a List:

Although range() itself doesn't generate a list, you can convert it into one:

```
numbers = list(range(1, 6))
print(numbers)
```

Output:
[1, 2, 3, 4, 5]

Using range() with len():

Often used to iterate over the indices of a sequence:

```
fruits = ["apple", "banana", "cherry"]
for i in range(len(fruits)):
   print(f"Index {i}: {fruits[i]}")
```

Output:
Index 0: apple
Index 1: banana
Index 2: cherry

Summary:

- The range() function is versatile and efficient for generating sequences of numbers.
- It is commonly used in loops, often with for.
- By customizing the start, stop, and step arguments, you can generate a wide variety of sequences.

FUNCTIONS

Mathematical FUNCTIONS

$y=f(x)=x$^2

$z=f(x, y)=x$^2+y^2

What is a FUNCTION in Programming?

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Functions are the real building blocks of any programming language.

The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Benefits of Using Functions

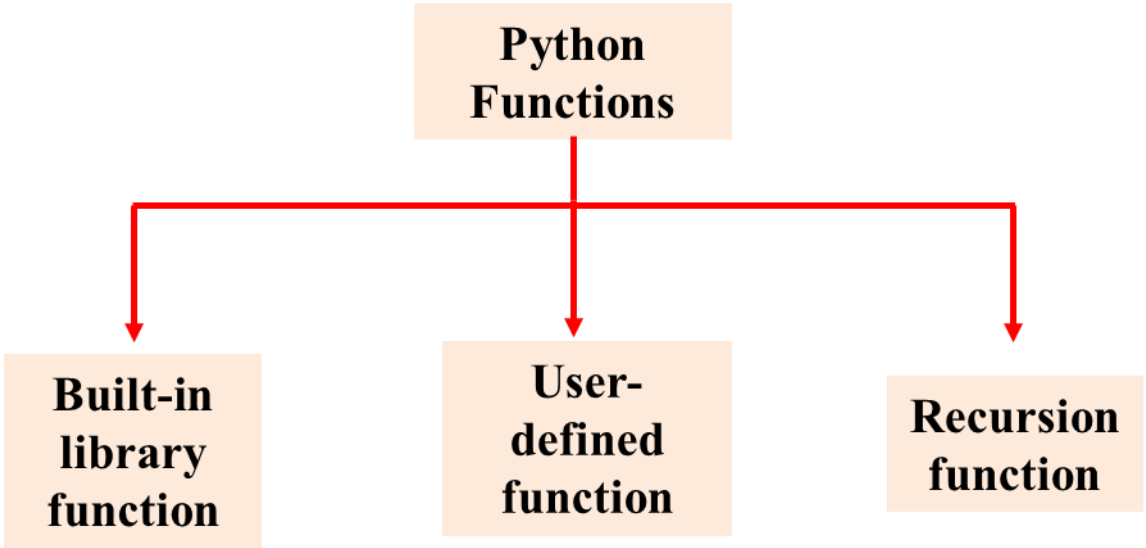
Increase Code Readability

Increase Code Reusability

Types of Functions in Python

Built-in library function: These are Standard functions in Python that are available to use.

Example: abs(), max(), bin(), print(), sqrt()



Built-in Library Function

A built-in function in Python is a pre-defined function that is readily available for use without the need for importing any additional modules, such as print(), len(), or sum().

Prob: 1: Absolute value of integer

An integer

var = -94

print('Absolute value of integer is:', abs(var))

Output: Absolute value of integer is: 94

Prob: 2: Square root of different numbers

Import math Library

import math

Return the function

print (math.sqrt(9))

print (math.sqrt(25))

print (math.sqrt(16))

Output:

3.0

5.0

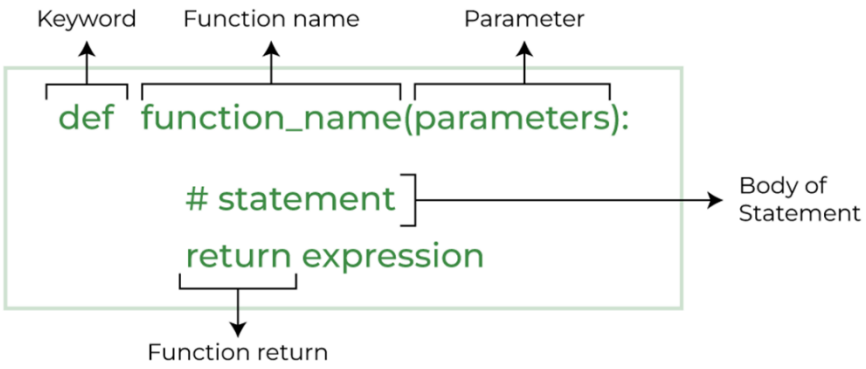
4.0

User Defined Function

We can create our own functions based on our requirements.

Python Function Declaration

The syntax to declare a function is:



Creating a Function in Python

Define a function in Python, using the ***def*** keyword.

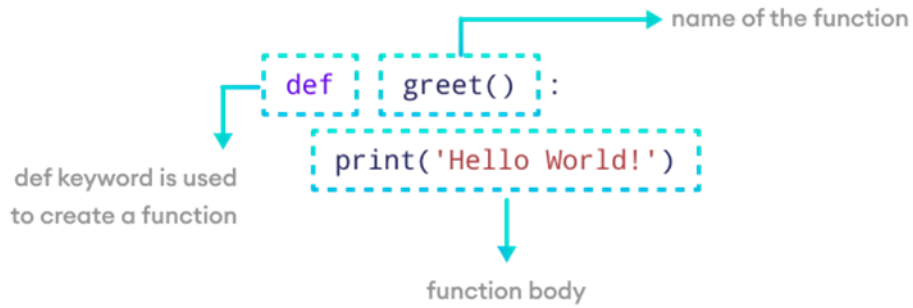
Can add any type of functionalities and properties to it as we require.

Example

def greet():

print(“Hellow World”)

Control of the program flows:



Calling a Function in Python

After creating a function in Python, we can call it by using the name of the function followed by parentheses containing parameters of that particular function.

```
def greet():
    print('Hello World!')
```

If we run the above code, we won't get an output.

It's because creating a function doesn't mean we are executing the code inside it. It means the code is there for us to use if we want to.

To use this function, we need to call the function.

```
greet()
```

Example: Python Function Call

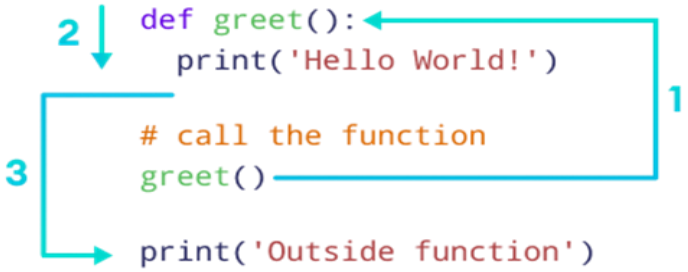
```
def greet():
    print('Hello World!')
# call the function
greet()
print('Outside function')
```

Output

Hello World!

Outside function

Control of the program flows:



Python Function Arguments

Arguments are inputs given to the function.

```
def greet(name):
    print("Hello,", name)
# pass argument
greet("John")
```

Sample Output 1

Hello John

We can pass different arguments in each call, making the function re-usable and dynamic.

Let's call the function with a different argument.

```
greet("David")
```

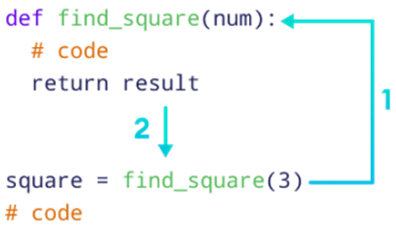
Sample Output 2

Hello David

Return Statement

We return a value from the function using the return statement.

Control of the program flows:



Recursion Function

Recursion functions are functions that calls itself.

It is always made of two portions:

The base case is the condition to stop the recursion.

The recursive case is the part where the function calls itself.

```
## Write a Python program to calculate the factorial of any user
##given input number using recursion function.

def factorial(N):
    fact = 1
    if N == 0:
        fact = 1
    elif N == 1:
        fact = 1
    else:
        fact = N*factorial(N-1)
    return fact
```

fact(5)

Write a program to calculate a to the power b using recursion function.

###

```
def power(a,b):
        if b == 0:
                return  1
        elif a==0:
                return 0
        elif b ==1:
                return a
        else:
                return a*power(a, b -1)
power(2,5)
```

32

Define a function to calculate the harmonic sum up to N terms.

1 + ½ + 1/3 + ¼ + 1/5 + 1/6 + …

##

```
def harmonic_sum(n):
        # check if n is less than 2
        if n == 1:
                return 1
        else:
                return (1/n) + harmonic_sum(n-1)

Print(harmonic_sum(7))
```

```
## Write a Python program to generate first N
## Fibonacci series using recursion function.
def fibo(N):
    if N<=0:
        return 0
    elif N==1:
        return 1
    else:
        return fibo(N-1)+fibo(N-2)
## Now call the above function to create Fibonacci series:

A = int(input('Enter N:'))
for i in range(A):
    print(fibo(i), end=' ')
```

```
Enter N:12
0 1 1 2 3 5 8 13 21 34 55 89
```

1. Introduction to Strings

A string in Python is a sequence of characters enclosed within single (' '), double (" "), or triple quotes (''' ''' or """ """). Strings are immutable, meaning once created, their content cannot be modified.

Example:

```
single_quote_string = 'Hello'
double_quote_string = "World"
triple_quote_string = '''Hello, Python!'''
```

2. String Operations

a. Concatenation

Strings can be concatenated using the + operator.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: Hello World
```

b. Repetition

The * operator repeats a string multiple times.

```
str1 = "Hi"
print(str1 * 3)  # Output: HiHiHi
```

c. Accessing Characters

You can access individual characters using indexing. Python uses zero-based indexing.

```
str1 = "Python"
print(str1[0])  # Output: P
print(str1[-1]) # Output: n (last character)
```

d. Slicing

Slicing allows you to obtain a substring by specifying a start and end index.

```
str1 = "Python Programming"
print(str1[0:6])  # Output: Python
print(str1[7:])   # Output: Programming
print(str1[:6])   # Output: Python
```

3. String Methods

Python provides many built-in methods to manipulate strings.

a. lower() and upper()

Convert the string to lowercase or uppercase.

```
str1 = "Hello"
print(str1.lower())  # Output: hello
print(str1.upper())  # Output: HELLO
```

b. strip()

Removes leading and trailing whitespace.

```
str1 = "   Hello   "
print(str1.strip())  # Output: Hello
```

c. replace()

Replaces a substring with another substring.

```
str1 = "Hello World"
print(str1.replace("World", "Python"))  # Output: Hello Python
```

d. find()

Returns the index of the first occurrence of a substring. Returns `-1` if the substring is not found.

```
str1 = "Python Programming"
print(str1.find("Pro"))  # Output: 7
```

e. split() and join()

- `split()` splits a string into a list based on a delimiter.
- `join()` joins elements of a list into a string using a specified delimiter.

```
str1 = "Hello,World,Python"
list1 = str1.split(",")
print(list1)  # Output: ['Hello', 'World', 'Python']

str2 = "-".join(list1)
print(str2)  # Output: Hello-World-Python
```

4. String Formatting

a. Using format()

```
name = "Alice"
age = 25
print("My name is {} and I am {} years old.".format(name, age))
# Output: My name is Alice and I am 25 years old.
```

b. f-strings (Python 3.6+)

```
name = "Bob"
age = 30
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Bob and I am 30 years old.
```

5. Escape Characters

Escape characters are used to include special characters in strings.

- \n - New line
- \t - Tab
- \" - Double quote
- \' - Single quote

Example:

```
print("Hello\nWorld")  # Output:
# Hello
# World
print("Hello\tWorld")  # Output: Hello    World
```

6. Immutable Nature of Strings

Since strings are immutable, operations like concatenation and slicing return a new string, leaving the original string unchanged.

```
str1 = "Hello"
str2 = str1 + " World"
print(str1)  # Output: Hello
print(str2)  # Output: Hello World
```

Data Structure in Python
List

A list is an object that contains multiple data items. The items in a list are ordered, and each item has an index indicating its position in the list. The first item in a list is at index 0, the second at index 1, and so on.

The data items are also called elements. The elements of a list can be any type. Lists are dynamic data structures, meaning that items may be added to them or removed from them. Lists are mutable, which means that their contents can be changed during a program's execution.

A list of integers

numbers = [1, 2, 3, 4]

A list of strings

cities = ['Delhi', 'Mumbai', 'Chennai', 'Kolkata']

A list holding different types

add = [8, 'Deepak', 'India', 9210458408]

A list holding another list

x = ['CBSE', 2020, [11, 12]]

An empty list

There is a special list that contains no elements. It is called the empty list, and it is denoted [].

empty = []

Accessing list element

>>> num = [10, 20, 30, 40]

>>> print(num[0])

10

If you try to read or write an element that does not exist, you get a runtime error:

>>> num[4] = 50

Traceback (most recent call last):

File "", line 1, in

num[4] = 50

IndexError: list index out of range

If an index has a negative value, it counts backward from the end of the list:

>>> print(num[-1])

40

Lists Are Mutable

Lists in Python are mutable, which means their elements can be changed.

>>> col= ['red','green','yellow']

>>> col[1] = 'blue'

>>> print(col)

['red', 'blue', 'yellow']

List slicing

A slice is a span of items that are taken from a sequence. To get a slice of a list, you write an expression in the following general format:

list_name[start : end]

In the general format, start is the index of the first element in the slice, and end is the index marking the end (but not including) of the slice. If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So, if you omit both, the slice is a copy of the whole list.

>>> t = [10, 20, 30, 40, 50, 60]

>>> t[1:3]

[20, 30]

>>> t[:4]

[10, 20, 30, 40]

>>> t[3:]

[40, 50, 60]

>>> t[:]

[10, 20, 30, 40, 50, 60]

Extended slicing

You can also use the step value in slicing. Syntax is given below:
list[start:end:step]

>>> t = [10, 20, 30, 40, 50, 60]

>>> t[::2]

[10, 30, 50]

>>> t[1::2]

[20, 40, 60]

>>> t[:4:2]

[10, 30]

#To reverse list use negative step

>>> t[::-1]

[60, 50, 40, 30, 20, 10]

Concatenating Lists

The + operator concatenates lists:

>>> list1 = [10,20,30]

>>> list2 = [7,8,60]

>>> list3 = list1 + list2

>>> print(list3)

[10, 20, 30, 7, 8, 60]

The Repetition Operator

The * operator repeats a list a given number of times:

>>> a = [20,30,40]

>>> b = a*2

>>> print(b)

[20, 30, 40, 20, 30, 40]

del operator

del removes an element from a list:

>>> a = ['one', 'two', 'three']

>>> del a[1]

>>> a

['one', 'three']

>>> b = ['Sun', 'Mon', 'Tue', ‘Wed’]

>>> del b[-1]

>>> b

['Sun', 'Mon', 'Tue']

Useful Built-in Functions

len()

The function len() returns the length of a list.

```
>>> num = [10,4,90]
>>> len(num)
3
```

sum()

Returns the sum of a list.

```
>>> num = [10,4,90]
>>> sum(num)
104
```

max()

Returns the largest item in a list.

```
>>> num = [10,4,90]
>>> max(num)
90
```

min()

Returns the smallest item in a list.

```
>>> num = [10,4,90]
>>> min(num)
4
```

append()

Append function adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
>>> t = ['a', 'b', 'c','d']
>>> y =['m', 'n']
>>> t.append(y)
>>> print(t)
['a', 'b', 'c', 'd', ['m', 'n']]
```

extend()

extend takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

sort()

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

reverse parameter is used to arrange list elements in descending order

```
>>> L = [34,66,12,89,28,99]
>>> L.sort(reverse = True)
>>> print(L)
[99,89,66,34,28,12]
```

pop()

Delete an element from list at the specified position. If you don't provide an index, it deletes and returns the last element.

```
>>> n = [10,20,30]
>>> m = n.pop()
>>> print(n)
[10, 20]
>>> print(m)
30
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

index()

Returns the index of the first element with the specified value a

```
>>> t = ['a', 'b', 'c']
>>> t.index('b')
1
```

reverse()

Reverses the order of the list

```
>>> t = ['a', 'b', 'c']
>>> t.reverse()
>>> print(t)
['c', 'b', 'a']
```

remove()

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

insert()

Adds an element at the specified position

```
>>> t = ['a', 'b', 'c']
>>> t.insert(2,'x')
>>> print(t)
['a', 'b', 'x', 'c']
```

list()

The range() function can generate a sequence. This sequence can be converted to list by using list() function.

```
>>> n = list(range(10))
>>> print(n)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Traversing a List

We can access each element of the list or traverse a list using a for loop or a while loop.

```
# Using FOR loop
colors = ['orange', 'green', 'blue']
for color in colors:
    print(color)
# Using WHILE loop
i = 0
colors = ['red','green','yellow']
while i < len(colors):
    print(colors[i])
    i = i+1
```

Some problems

1. **Write a program that accepts a list from user and print the alternate element of list.**

```
mylist = []
size = int(input('How many elements you want to enter? '))

print('Enter',str(size),'elements')

for i in range(size):
    data = int(input())
    mylist.append(data)

print('Alternate elements are:')

for i in range(0,size,2):
    print(mylist[i])
```

2. **Find and display the largest and least numbers of a list without using built-in functions max() and min(). The list elements are user given inputs.**

```
mylist = []
size = int(input(‘Number of elements in the list:’))

print('Enter',str(size),' numbers one by one:')

for i in range(size):
    data = int(input())
    mylist.append(data)

max = mylist [0]
min = mylist [0]
for data in mylist:
    if data > max:
        max = data
    if data<min:
        min = data
print('The largest number in list is', max)
print('The least number in list is', min)
```

3. Write a program that input a string and ask user to delete a given word from a string.

```
text = input('Enter a string: ')
words = text.split()

data = input('Enter a word to delete: ')
status = False

for word in words:
    if word == data:
        words.remove(word)
        status = True

if status:
    text = ' '.join(words)
    print('String after deletion:',text)
else:
    print('Word not present in string.')
```

4. Write a program to count the total word length in a user given string.

```
text = input('Enter a sentence without punctuation: ')
words = text.split()

sum=0
for item in words:
      sum = sum + len(item)
print('Total word length of the string:',sum)
```

Tuple

A **tuple** is a sequence of values much like a list. The values stored in a tuple can be any type, and each item has an index indicating its position in the list. The first item in a tuple is at index 0, the second at index 1, and so on. The important difference between list and tuple is that tuples are **immutable**.

A tuple is a comma-separated list of values inside parentheses (). The parentheses are optional.
Examples:
>>> t1 = 'a', 'b', 'c', 'd', 'e'
>>> t2 = ('a', 'b', 'c', 'd', 'e')
Here, t1 and t2 both are tuples.

Construction of tuple:

1. Empty tuple:

>>> t = ()
>>> print(t)
()
Or, we can construct tuple by using **tuple()** function.

>>> t = tuple()
>>> print(t)
()

2. Tuple with a single element:

To create a tuple with a single element, you have to include the final comma. Without final comma the single element within parenthesis will not be a tuple.

>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
Tuple can be created using tuple() function.
>>> t2 = tuple('a')
>>> print(t2)
('a',)

Conversion Between Lists and Tuples:

You can use the built-in list() function to convert a tuple to a list and the built-in tuple() function to convert a list to a tuple.

```
>>> n1 = (1, 2, 3)
>>> n2 = list(n1)
>>> print(n2)
[1, 2, 3]

>>> x1 = ['red', 'green', 'blue']
>>> x2 = tuple(x1)
>>> print(x2)
('red', 'green', 'blue')
```

Operations on tuples

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Concatenating tuples

The + operator concatenates tuples:

```
>>> t1 = (10,20,30)
>>> t2 = (7,8,60)
>>> t3 = t1 + t2
>>> print(t3)
(10, 20, 30, 7, 8, 60)
```

The Repetition Operator

The * operator repeats a tuple a given number of times:

```
>>> a = (20,30,40)
>>> b = a*2
>>> print(b)
(20, 30, 40, 20, 30, 40)
```

Tuple Methods and Useful Built-in Functions

len()

The function len returns the length of a tuple.

```
>>> num = (10,4,90)
>>> len(num)
3
```

max()

Returns the largest item in a tuple.

```
>>> num = (10,4,90)
>>> max(num)
90
```

min()

Returns the smallest item in a tuple.

```
>>> num = (10,4,90)
>>> min(num)
4
```

sorted()

Returns a sorted list of the specified tuple.

```
>>> t = (10,4,1,55,3)
>>> sorted(t)
[1, 3, 4, 10, 55]                    # Output is a list.
```

count()

Returns the number of times a specified value occurs in a tuple

```
>>> num = (10,4,90,4)
>>> num.count(4)
2
```

index()

Searches the tuple for a specified value and returns the position of where it was found

```
>>> num = (10,4,90,4)
>>> num.index(90)
2
```

Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable*, and *unindexed*.

Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable means you cannot alter any element. But you can remove and add any element. Thus, set is mutable.

Sets are written with curly brackets.

Set1 = {"apple", "banana", "cherry"}
print(Set1)

{"apple", "banana", "cherry"}

Sets cannot have two items with the same value i.e. **duplicates are not allowed** in set.

set2 = {"apple", "banana", "cherry", "apple"}
print(set2)

{"apple", "banana", "cherry"}

A set can contain different data types:

A set with strings, integers and Boolean values:

set1 = {"abc", 34, True, 40, "male"}

- To determine how many items a set has, use the **len()** function.
- Use **set()** function to construct set from list or tuple.

Mathematical operations using sets:

set1 = {1,3,5,9}

set2 = {1,11,7,5}

intersection of sets

print(set1&set2) OUTPUT: {1, 5}

print(set1.intersection(set2)) OUTPUT: {1, 5}

print(set1) OUTPUT: {1,11,7,5}

union of sets

print(set1|set2) OUTPUT: {1, 3, 5, 7, 9, 11}

print(set1.union(set2)) OUTPUT: {1, 3, 5, 7, 9, 11}

difference of sets

print(set1-set2) OUTPUT: {9, 3}

print(set1.difference(set2)) OUTPUT: {9, 3}

Symmetric difference of sets

print(set1.symmetric_difference(set2)) OUTPUT: {3, 7, 9, 11}

add(x):----Adds item x to the set

s={10, 20, 30}

s.add(40);

print(s) OUTPUT: {40, 10, 20, 30}

update(x, y, z):

To add multiple items to the set.

Arguments are not individual elements and these are Iterable objects like List, range etc.

All elements present in the given Iterable objects will be added to the set.

s={10, 20, 30}

L=[40,50,60,10]

s.update(L, range(5))

print(s) OUTPUT: {0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}

copy():

Returns copy of the set.

s={10,20,30}

s1=s.copy()

print(s1)

pop():

It removes and returns some random element from the set.

s={40,10,30,20}

print(s) **OUTPUT: {40, 10, 20, 30}**

print(s.pop()) **OUTPUT: 40**

print(s) **OUTPUT: {10, 20, 30}**

remove(x):

It removes specified element from the set.

s={40,10,30,20}

s.remove(30)

print(s) **OUTPUT: {40, 10, 20}**

s.remove(50) **OUTPUT: KeyError: 50**

discard(x):

It removes the specified element from the set.

If the specified element not present in the set then we won't get any error.

s={10,20,30}

s.discard(10)

print(s) **OUTPUT: {20, 30}**

s.discard(50)

print(s) **OUTPUT: {20, 30}**

clear():

To remove all elements from the Set.

s={10,20,30}

print(s) **OUTPUT: {10, 20, 30}**

s.clear()

print(s)

Dictionary Data Structure

We can use List, Tuple and Set to represent a group of individual objects as a single entity. If we want to represent a group of objects as key-value pairs then we should go for Dictionary.

A dictionary in Python is an unordered set of key: value pairs. Each KEY must be unique, but the VALUES may be the same for two or more keys. You can use a KEY to locate a specific VALUE in dictionary.

Creating Dictionary:

d={} or **d=dict()**

d[100]="durga"

d[200]="ravi"

d[300]="shiva"

print(d) OUTPUT:{100: 'durga', 200: 'ravi', 300: 'shiva'}

Accessing Items

d={100:'durga' , 200:'ravi', 300:'shiva'}

print(d[100]) OUTPUT: durga

print(d[300]) OUTPUT: shiva

Adding an entry:

d={100:'durga' ,200:'ravi', 300:'shiva'}

d[400] = 'rama'

print(d) OUTPUT: {100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'rama'}

Important functions of dictionary:

dict():

d=dict() ===>It creates empty dictionary
d=dict({100:"durga",200:"ravi"}) ==>It creates dictionary with specified elements
d=dict([(100,"durga"),(200,"shiva"),(300,"ravi")])==>It creates dictionary with the given list of tuple elements

len()

Returns no of items

clear():

To remove all elements

get():

To get value associated with key

d.get(key)

d={100:"durga",200:"ravi",300:"shiva"}

print(d[100]) ==>durga

print(d[400]) ==>KeyError:400

print(d.get(100)) ==durga

print(d.get(400)) ==>None

print(d.get(100,"Guest")) ==durga

print(d.get(400,"Guest")) ==>Guest

pop():

d.pop(key)

It removes the entry associated with the specified key and returns the corresponding value

If the specified key is not available then we will get KeyError

d={100:"durga",200:"ravi",300:"shiva"}

2) print(d.pop(100))-->durga

3) print(d)--->{200: 'ravi', 300: 'shiva'}

4) print(d.pop(400))--> KeyError: 400

popitem():

It removes an arbitrary item(key-value) from the dictionary and returns it.

d={100:"durga",200:"ravi",300:"shiva"}

2) print(d)--> {100: 'durga', 200: 'ravi', 300: 'shiva'}

3) print(d.popitem())--> (300, 'shiva')

4) print(d)--> {100: 'durga', 200: 'ravi'}

keys():

It returns all keys associated with dictionary

d={100:"durga",200:"ravi",300:"shiva"}

print(d.keys()) --> dict_keys([100, 200, 300])

for k in d.keys():

print(k)

--> 100

200

300

values():

returns all values associated with dictionary

d={100:"durga",200:"ravi",300:"shiva"}

print(d.values())--> dict_values(['durga', 'ravi', 'shiva'])

for v in d.values():

print(v)

--> durga

ravi

shiva

1. **Write a program to enter student names and percentage marks in a dictionary and display information on the screen. Then search marks of a student by giving name as input.**

```
rec={}
n=int(input("Enter no. of students: "))
i=1
while i <=n:
   name=input("Enter St. Name: ")
   marks=input("Enter %Marks of St: ")
   rec[name]=marks
   i=i+1
# dictionary is created
# To display the content of dictionary
print("Name of St","\t","% of marks")
for x in rec:
   print("\t", x,"\t\t", rec[x])
        # search marks by student name

        Search_Name = input('Name of the student to search marks:')

        MARK= rec.get(Search_Name)

        if MARK is not None:

        print("Marks for {}: {}".format(Search_Name, MARK))

        else:

        print("Student {} not found in the records.".format(Search_Name))
```

2. **Write a program to find number of occurrences of each letter present in the given string?**

```
word=input("Enter any word: ")

d={}

for x in word:

        d[x]=d.get(x,0)+1

for k, v in d.items():

        print(k,"occurred ",v," times")
```

OUTPUT:

```
Enter any word: mississippi

m occurred 1 times
```

i occurred 4 times

s occurred 4 times

p occurred 2 times

Update dictionaries

d[key]=value

- If the key is not available then a new entry will be added to the dictionary with the specified key-value pair
- If the key is already available then old value will be replaced with new value.

d={100:"durga",200:"ravi",300:"shiva"}

print(d)--> {100: 'durga', 200: 'ravi', 300: 'shiva'}

d[400]="pavan"

print(d)--> {100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}

d[100]="sunny"

print(d)---> {100: 'sunny', 200: 'ravi', 300: 'shiva', 400: 'pavan'}

Delete elements from dictionary

del d[key]

- It deletes entry associated with the specified key.
- If the key is not available then we will get KeyError

d={100:"durga",200:"ravi",300:"shiva"}

print(d)--> {100: 'durga', 200: 'ravi', 300: 'shiva'}

del d[100]

print(d)- {200: 'ravi', 300: 'shiva'}

del d[400]--> Error

d.clear()

To remove all entries from the dictionary

d={100:"durga",200:"ravi",300:"shiva"}

print(d)--> {100: 'durga', 200: 'ravi', 300: 'shiva'}

d.clear()

print(d)--> {}

del d

d ={100:"durga", 200:"ravi", 300:"shiva"}

print(d)--> {100: 'durga', 200: 'ravi', 300: 'shiva'}

del d

print(d)--> Error

Files

Need for a data file

To Store data in organized manner

• To store data permanently

• To access data faster

• To Search data faster

• To easily modify data later on

File

A file is the collection of data stored on a disk in one unit identified by filename.

Types of File

Text File: Text file usually we use to store character data. For example, test.txt

Binary File: The binary files are used to store binary data such as images, video files, audio files, etc.

Steps to process file Input/output in Python

Step 1. Open the file — Opening a file creates a connection between the file and the program.

Step 2. Process the file — In this step data is either written to the file (if it is an output file) or read from the file (if it is an input file).

Step 3. Close the file — When the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.

The open function is used in Python to open a file. The open function returns a file object and associates it with a file on the disk.

variable = open(filename, mode)

File Mode	Description
'r'	Opens the file in read-only mode.
'rb'	Opens the file in binary and read-only mode.
'r+'	Opens the file in both read and write mode.
'w'	Opens the file in write mode. If the file already exists, all the contents will be overwritten. If the file doesn't exist, then a new file will be created.
'wb+'	Opens the file in read, write and binary mode. If the file already exists, the contents will be overwritten. If the file doesn't exist, then a new file will be created.
'a'	Opens the file in append mode. If the file doesn't exist, then a new file will be created.

Writing Data to a Text File

write() method: takes a string as an argument and writes it to text file.

Program (oceans.py)

```
OutFile = open('Python.txt', 'w') #Step 1
OutFile.write('Kiran is a CSE Student\n') #Step 2
OutFile.write('This is a Python class\n')
OutFile.write('Topic is Files\n')
OutFile.write('This is Theory Class\n')
OutFile.close() #Step 3
```

writelines() method: It takes an iterable as argument (an iterable object can be a tuple, a list). Each item contained in the iterator is expected to be a string.

```
OutFile = open('Python.txt', 'w')
text = 'Atlantic\n','Pacific\n','Indian\n','Arctic\n'
OutFile.writelines(text)
OutFile.close()
```

File Path

A file path defines the location of a file or folder in the computer system.

There are two ways to specify a file path.

Absolute path: which always begins with the root folder

Relative path: which is relative to the program's current working directory

The absolute path includes the complete directory list required to locate the file.

For example, /user/Pynative/data/sales.txt is an absolute path to discover the sales.txt. All of the information needed to find the file is contained in the path string.

A relative path is the path that is relative to the working directory location on your computer. In Program (oceans.py) we have used relative path of ocean.txt in open function. Therefore, text file 'oceans.txt' is created and stored in same folder (current working directory) where we have saved 'oceans.py'

Program (absolute.py)

```
OutFile = open(r'C:\Users\91981\Desktop\Python.txt', 'w')

outFile.write('Atlantic\n')

OutFile.write('Pacific\n')

OutFile.write('Indian\n')

outFile.write('Arctic\n')

OutFile.close()
```

Append Data to a File

Program (appendocean.py)

```
Outfile = open('Python.txt', 'a') #'a' For Append

Outfile.write('Southern\n')

Outfile.close()
```

Reading Data From a File

To open a file in read mode use the 'r' mode in open() function. You can use read(), readline() or readlines() methods for reading data from file.

read([n]) method: The read() method reads and returns a string of n characters, or the entire file as a single string if n is not provided.

Program (readfile1.py)

```
infile = open('Python.txt', 'r')

data = infile.read()
```

```
print(data)
infile.close()
```

Output

```
Atlantic
Pacific
Indian
Arctic
Southern
```

Reading File Using readline() method

In Python you can use the readline() method to read a line from a file. A line is simply a string of characters that are terminated with a \n. The method returns the line as a string, including the \n.

Program (readfile2.py)

```
infile = open(‘Python.txt', 'r')
line1 = infile.readline()
line2 = infile.readline()
line3 = infile.readline()
print(line1)
print(line2)
print(line3)
infile.close()
```

Output

```
Atlantic
Pacific
Indian
```

rstrip(): Output of above program shows extra spaces between lines. This is because readline() method reads the string including the \n. However, in many cases you want to remove the \n from a string after it is read from a file. In Python rstrip() method removes specific characters from the end of a string.

Program (readfile3.py)

```
infile = open(‘Python.txt', 'r')
line1 = infile.readline().rstrip('\n')
line2 = infile.readline().rstrip('\n')
line3 = infile.readline().rstrip('\n')
print(line1)
print(line2)
print(line3)
infile.close()
```

Output

```
Atlantic
Pacific
Indian
```

readlines() method: The readlines() returns a list of strings, each representing a single line of the file. If n is not provided then all lines of the file are returned.

for Loop to Read All Lines from file

Program (readfile4.py)

```
file = open(‘Python.txt', 'r')
for line in file:
    line = line.rstrip()
    print(line)
```

```
file.close()
```

Output

Atlantic

Pacific

Indian

Arctic

Southern

Q1. Write a program that writes 10 random numbers to a file 'numbers.txt'. Each random number should be in the range of 1 through 100.

```
import random

# Open the file in write mode
file = open("numbers.txt", "w")

# Generate and write random numbers to the file
for i in range(10):
    number = random.randint(1, 100)
    file.write(str(number) + "\n")

# Close the file after writing
file.close()
```

NumPy

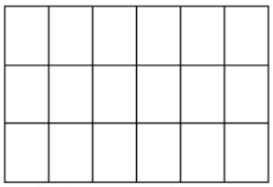
NumPy is 'Numerical Python'.

What is NumPy?

NumPy is an open-source Python library that is used in almost every field of science and engineering. It contains a powerful N-dimensional array object. An N-dimensional array is simply an array with any number of dimensions.

An array with a single dimension is known as vector, while a matrix refers to an array with two dimensions. For 3-D or higher dimensional arrays, the term tensor is also commonly used. In NumPy, dimensions are also called axes.

1- dimensional array



2- dimensional array

Differences between NumPy array and the standard Python list

- NumPy arrays have fixed size, unlike Python lists which can grow dynamically.
- All elements in a NumPy array are required to be of the same data type whereas the Python list can contain any type of element.
- NumPy arrays are faster than lists.

How to install?

- In windows write ***pip install numpy*** in command prompt.
- In Ubuntu write ***sudo apt install python3-numpy*** in terminal.

How to import NumPy?

In order to start using NumPy and all of the functions available in NumPy, you'll need to import it. This can be easily done with this import statement:

>>> import numpy

Or you can shorten NumPy to an alias version **np**.

>>> import numpy as np

1- Dimensional Array:

```
>>> x = np.array([10, 20, 30])
>>> type(x)  # Type of object
```

<class 'numpy.ndarray'>
>>> len(x) # Length: No. of elements # 3
>>> sum(x) # Sum of elements #60

2- Dimensional Array:

>>> M = np.array([[1, 2, 3], [4, 5, 6]])
>>> M
array([[1, 2, 3],
[4, 5, 6]])
[The display is like a Matrix with 2 rows and 3 columns.]
>>> M.ndim
2
>>> M.size
6
>>> M.shape # [Shape of the 2D array as a tuple]
(2, 3)

Shape, Reshape, Size and Resize of arrays:

>>> A = np.array([2, 4, 6, 8, 10, 12])
>>> **A.shape**
(6,)
[1D array with 6 elements.]
# Converted to a 2D array (2 × 3)
>>> **B =A.reshape(2, 3)**
>>> B
array([[2, 4, 6],
[8, 10, 12]])
>>> **A.reshape(3, 2)**
array([[2, 4],
[6, 8],
[10, 12]])
>>> **A**
array([2, 4, 6, 8, 10, 12])
Thus, In case of reshape, the original array remains unchanged.
>>> A.size
6
>>> B.size
6
>>> A.resize(2, 3)

>>> A
array([[2, 4, 6],
[8, 10, 12]])
Thus, in case of using resize, the original array becomes modified.

1 Range of numbers

np.arange(start, stop, step size)

```
# Examples
>>> np.arange(2, 10, 3)
array([2, 5, 8])
# Default step = 1
>>> np.arange(2, 10)
array([2, 3, 4, 5, 6, 7, 8, 9])
# Default start = 0
>>> np.arange(5)
array([0, 1, 2, 3, 4])
>>> np.arange(0.2, 2, 0.4)
array([0.2, 0.6, 1. , 1.4, 1.8])
>>> np.arange(5,10, 2, dtype= ‘f’)
array([5., 7., 9.], dtype=float32)
```

2 Linear space

np.linspace(start, end, number of elements)

```
# Examples
>>> np.linspace(10, 20, 5)
array([10., 12.5, 15., 17.5, 20.])
Note: Step size = (20 – 10)/4 = 2.5 [4 gaps]
>>> np.linspace(10, 20, 5, endpoint = True)
array([10., 12.5, 15., 17.5, 20.])
>>> np.linspace(10, 20, 5, endpoint = False)
array([10., 12., 14., 16., 18.])
```

Concatenating arrays:

1D:

```
>>> a = np.array([10, 20, 30])
>>> b =np.array([40, 50, 60])
>>> np.concatenate((a, b))
array([10, 20, 30, 40, 50, 60])
```

2D:

```
>>> x
array([[1, 2],
[3, 4]])

>>> y
array([[5, 6],
[7, 8]])

>>> np.concatenate((x, y), axis = 1)
array([[1, 2, 5, 6 ],
[3, 4, 7, 8 ]])
```

```
>>> np.concatenate((x, y), axis = 0)
array([[1, 2],
[3, 4],
[5, 6],
[7, 8] ])
# By default, axis = 0
>>> np.concatenate((x, y))
array([[1, 2],
[3, 4],
[5, 6] ,
[7, 8] ])
```

Some other numpy fuctions:

```
>>> x = np.array([10, 20, 30, 40, 50,60, 70, 80, 90])
>>> np.sum(x)
450
>>>x.sum()
450
# Minimum and Maximum
 >>> x.min() # Minimum
10
>>> x.max() # Maximum
90
>>> y = x.reshape(3, 3)
>>> y
array([[10, 20, 30],
[40, 50, 60],
[70, 80, 90]])
# Sum of each row
>>> np.sum(y, axis = 1)  #row sum
array([ 60, 150, 240])
# Methods as attribute
>>> y.sum(0) # Along axis = 0, #column sum
array([120, 150, 180])
>>> y.sum(1) # Along axis = 1
array([ 60, 150, 240])
>>> y.sum() # No axis ref. # 450 [Sum of all elements.]
```

Mean, Median

```
>>> np.mean(y, axis = 0)
 array([40., 50., 60.])
```

[Also, **y.mean(0)** as attribute.]

```
>>> np.mean(y, 1)
array([20., 50., 80.])
>>> np.mean(y)
50.0
>>> np.median(y, 0)
array([40., 50., 60.])
```

```
>>> np.median(y, 1)
array([20., 50., 80.])
>>>np.median(y)
50.0
```

Sorting

```
>>> x = [-2, 0, 3, 1, 10, 5, -7]
# Ascending order
>>> np.sort(x)
array([-7, -2, 0, 1, 3, 5, 10])
# Descending order
>>> np.sort(x)[::-1]
array([10, 5, 3, 1, 0, -2, -7])
```

Matrix operations in NumPy:

```
>>>a1 = np.arange(6).reshape(2, 3)
print(a1)
# [[0 1 2]
[3 4 5]]

>>>a2 = np.arange(6, 18, 2).reshape(2, 3)
print(a2)
# [[ 6  8 10]
 [12 14 16]]
```

Transpose:

```
>>>a1.transpose()
#[[0 3]
[1 4]
[5 2]]

>>>a1.T
#[[0 3]
[1 4]
[5 2]]
```

Element wise operations:

```
>>>print(a1 + a2)
# [[ 6  9 12]
 [15 18 21]]

>>>print(a1 - a2)
# [[ -6  -7  -8]
```

```
 [ -9 -10 -11]]

>>>print(a1 * a2)
# [[ 0  8 20]
[36 56 80]]

>>>print(a1 / a2)
# [[0.         0.125      0.2]
 [0.25       0.28571429 0.3125]]

>>>print(a1**a2)
# [[         0          1       1024]
[    531441  268435456 152587890625]]
```

Calculations with scalar values are also possible:

```
>>>print(a1 * 100)
# [[  0 100 200]
 [300 400 500]]
```

Matrix multiplication: np.matmul(), np.dot() or @ operator

```
>>>a1 = np.arange(4).reshape((2, 2))
print(a1)
# [[0 1]
 [2 3]]

>>>a2 = np.arange(6).reshape((2, 3))
print(a2)
# [[0 1 2]
  [3 4 5]]
>>>print(np.matmul(a1, a2))
# [[ 3  4  5]
  [ 9 14 19]]

>>>print(np.dot(a1, a2))
# [[ 3  4  5]
  [ 9 14 19]]

>>>print(a1.dot(a2))
# [[ 3  4  5]
  [ 9 14 19]]
>>>print(a1@ a2)
# [[ 3  4  5]
  [ 9 14 19]]
```

Determinant and inverse of matrix: np.linalg.det() and np.linalg.inv()

```
>>>a = np.array([[0, 1], [2, 3]])
print(a)
# [[0 1]
 [2 3]]

>>>print(np.linalg.det(a))
# -2.0

>>>a = np.array([[2, 5], [1, 3]])
print(a)
# [[2 5]
  [1 3]]

>>>print(np.linalg.inv(a))
# [[ 3. -5.]
  [-1.  2.]]
# For singular matrix it will produce error.
>>>a_singular = np.array([[0, 0], [1, 3]])
>>>print(a_singular)
# [[0 0]
  [1 3]]

 >>>print(np.linalg.inv(a_singular))
# LinAlgError: Singular matrix
```

Some NumPy arrays:

np.empty((3,3))
It return a new empty array of given shape and type, without any entries.
np.identity(4)
It returns identity matrix of given order
np.eye(4)
#It returns a 2-D array with ones on the diagonal and zeros elsewhere.
np.ones((3, 4))
#It returns an array of the given shape with all elements 1.
np.zeros((3, 3))
It returns an array with given shape with all elements zero.
np.diag((1, 2, 3))
It returns an array with given diagonal elements and all other elements are zeroes.

```
##
>>>A=np.array([[1,2],[2,3],[3,4]])
>>>print(A.flatten())
#[1 2 2 3 3 4]
>>> print(A)
#[[1 2]
[2 3]
[3 4]]
>>>print(A.ravel())
#[1 2 2 3 3 4]
>>> print(A)
```

#[1 2 2 3 3 4]

Thus, ravel() modifies the original array but flatten() does not modify original array.

Some problems:

Write a program to add two 2-dimensional NumPy arrays where the elements of the arrays are user given.

```
import numpy as np

# Taking input for the dimensions of the arrays
rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))

# Taking input for the elements of the first array
print("Enter the elements of the first array (one row at a time):")
array1_elements = []
for i in range(rows):
    row = [float(x) for x in input().split()]
    array1_elements.append(row)

# Taking input for the elements of the second array
print("Enter the elements of the second array (one row at a time):")
array2_elements = []
for i in range(rows):
    row = [float(x) for x in input().split()]
    array2_elements.append(row)

# Creating NumPy arrays from the input elements
array1 = np.array(array1_elements)
array2 = np.array(array2_elements)

# Adding the arrays
result_array = array1 + array2

# Displaying the result
print("Resultant array after addition:")
print(result_array)
```

Output sample:

```
Enter the number of rows: 2
Enter the number of columns: 2
Enter the elements of the first array (one row at a time):
1 2
3 4
Enter the elements of the second array (one row at a time):
5 6
7 8
Resultant array after addition:
[[ 6.  8.]
 [10. 12.]]
```

Write a program to find the matrix-product of two NumPy arrays.

```
import numpy as np

# Taking input for the dimensions of the arrays
rows1 = int(input("Enter the number of rows for array 1: "))
```

```
cols1 = int(input("Enter the number of columns for array 1: "))

rows2 = int(input("Enter the number of rows for array 2: "))
cols2 = int(input("Enter the number of columns for array 2: "))
# Taking input for the elements of the arrays
print("Enter the elements of array 1 (one row at a time):")
array1_elements = []
for i in range(rows1):
    row = [float(x) for x in input().split()]
    array1_elements.append(row)
print("Enter the elements of array 2 (one row at a time):")
array2_elements = []
for i in range(rows2):
    row = [float(x) for x in input().split()]
    array2_elements.append(row)

# Creating NumPy arrays from the input elements
array1 = np.array(array1_elements)
array2 = np.array(array2_elements)

# Multiplying the arrays
result_array =np.matmul( array1, array2)

# Displaying the result
print("Resultant array after addition:")
print(result_array)
```

Pandas

What is Pandas?

Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Why Pandas?

The beauty of Pandas is that it simplifies the task related to data frames and makes it simple to do many of the time-consuming, repetitive tasks involved in working with data frames, such as:

Import datasets - available in the form of spreadsheets, comma-separated values (CSV) files, and more.

Data cleansing - dealing with missing values and representing them as NaN, NA, or NaT.

Size mutability - columns can be added and removed from DataFrame and higher-dimensional objects.

Data normalization – normalize the data into a suitable format for analysis.

Reshaping and pivoting of datasets – datasets can be reshaped and pivoted as per the need.

Efficient manipulation and extraction - manipulation and extraction of specific parts of extensive datasets using intelligent label-based slicing, indexing, and subsetting techniques.

Statistical analysis - to perform statistical operations on datasets.

Data visualization - Visualize datasets and uncover insights.

Data structures in Pandas:

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

How to install?

- In windows write ***pip install pandas*** in command prompt.
- In Ubuntu write ***sudo apt install python3-pandas*** in terminal.
-

pandas.Series

pandas.Series(data, index, dtype, copy)

 # Series from list

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array([1,2,3,4])
s = pd.Series(data, index=[0,2,5,8], dtype=int, copy=False)
print(s)
s.iloc[0] = 6
print(s)
print(data)
```

```
OUTPUT:
0 1
2 2
5 3
8 4
dtype: int64

0 6
2 2
5 3
8 4
dtype: int64
[6 2 3 4]
```

#Series from dictionary

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print(s)
```

a 0.0

b 1.0

c 2.0

dtype: float64

#Series from scalar

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
```

0 5
1 5
2 5
3 5
dtype: int64

print(s)

How to retrieve elements of series?

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s[[‘b’]])  # b    2
```

```
#Retrieve multiple elements
print(s[['a','c','d’]])
print(s[[‘f’]])
```

```
a  1
c  3
d  4
dtype: int64
KeyError: 'f'
```

DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

A pandas DataFrame can be created using the following constructor −

pandas.DataFrame(data, index, columns, dtype, copy)

A pandas DataFrame can be created using various inputs like −

Lists

dictionary

Series

Numpy ndarrays

Another DataFrame

Creating DataFrame

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

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

 # Data frame from list

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

```
   0
0  1
1  2
2  3
3  4
4  5
```

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],index=['a','b','c'])
print(df)
```

```
      Name  Age
a    Alex   10
b     Bob   12
c  Clarke   13
```

 # Data frame from dictionary

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)
```

```
    Name  Age
0    Tom   28
1   Jack   34
2  Steve   29
3  Ricky   42
```

Read a CSV/Excel file as a pandas DataFrame:

```
import pandas as pd
# Replace 'your_file.csv' with the actual path to your CSV file
file_path = 'your_file.csv'
# Read the CSV file into a DataFrame
df = pd.read_csv(file_path)
# Read the Excel file into a DataFrame
File_path2 = 'your_file.xlsx'
df1 = pd.read_excel(file_path2)
# Display the first few rows of the DataFrame
print(df.head())
```

Export pandas DataFrame to a CSV/Excel file:

```
import pandas as pd
# Replace 'your_file.csv' with the actual path to your CSV file
file_path = 'your_file.csv'
# Export DataFrame to CSV file
df.to_csv(file_path)
# Export DataFrame to Excel file
File_path2 = 'your_file.xlsx'
df.to_excel(file_path2)
# Export DataFrame to JSON file
File_path3 = 'your_file.json'
df.to_json(file_path3)
```

Fetch rows from the DataFrame based on a specific attribute

```
import pandas as pd

# Load the data and create a DataFrame (replace with your data loading mechanism)

# For demonstration, let's create a sample DataFrame

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],

        'Age': [30, 25, 35, 28, 32]}
```

```
df = pd.DataFrame(data)
# Print the original DataFrame
print("Original DataFrame:")
print(df)
# Example: Fetch rows where 'Age' is greater than or equal to 30
target_age = 30
filtered_df = df[df['Age'] >= target_age]
# Print the filtered DataFrame
print(f"\nFiltered DataFrame (Age >= {target_age}):")
print(filtered_df)
```

Original DataFrame:

	Name	Age
0	Alice	30
1	Bob	25
2	Charlie	35
3	David	28
4	Emily	32

Filtered DataFrame (Age >= 30):

	Name	Age
0	Alice	30
2	Charlie	35
4	Emily	32

Find the mean and standard deviation of a specific column containing numeric data.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
        'Age': [30, 25, 35, 28, 32],
        'Score': [85, 90, 78, 92, 88]}
df = pd.DataFrame(data)
# Print the original DataFrame
print("Original DataFrame:")
print(df)
# Specify the column for which calculate mean and standard deviation
target_column = 'Score'

# Calculate mean and standard deviation for the specified column
mean_value = df[target_column].mean()
std_value = df[target_column].std()
# Display the mean and standard deviation
print(f"\nMean of '{target_column}': {mean_value}")
print(f"Standard Deviation of '{target_column}': {std_value}")
```

Matplotlib

1. Introduction

When we want to convey some information to others, there are several ways to do so. The process of conveying the information with the help of plots and graphics is called **Data Visualization**.

- In python, we will use the basic data visualisation tool **Matplotlib.**
- It can create popular visualization types – line plot, scatter plot, histogram, bar chart, error charts, pie chart, box plot, and many more types of plots.
- It can export visualizations to all of the common formats like PDF, SVG, JPG, PNG, BMP etc.

How to install?

- In windows write ***pip install matplotlib*** in command prompt.
- In Ubuntu write ***sudo apt install python3-matplotlib*** in terminal.

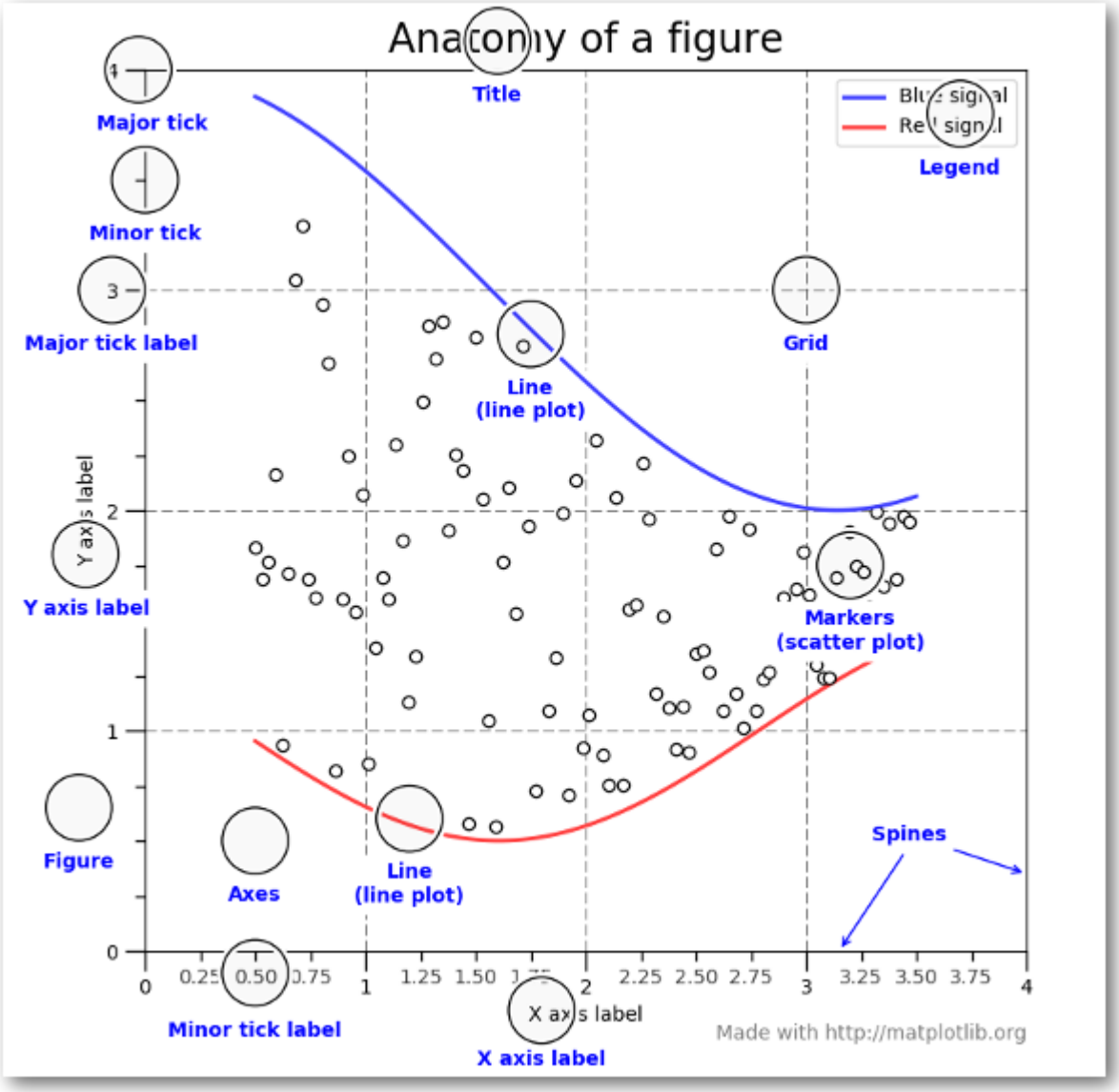
How to import Matplotlib?

We can import Matplotlib as follows:-

import matplotlib

Most of the time, we have to work with **pyplot** interface of Matplotlib. So, I will import pyplot interface of Matplotlib as follows:-

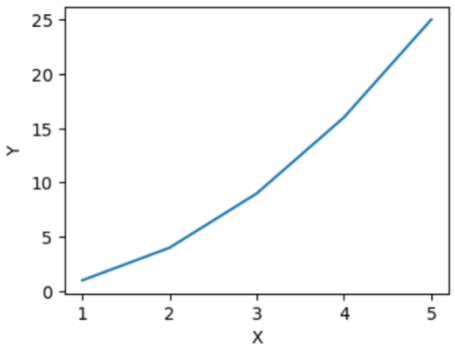
import matplotlib.pyplot as plt



Line Plot:

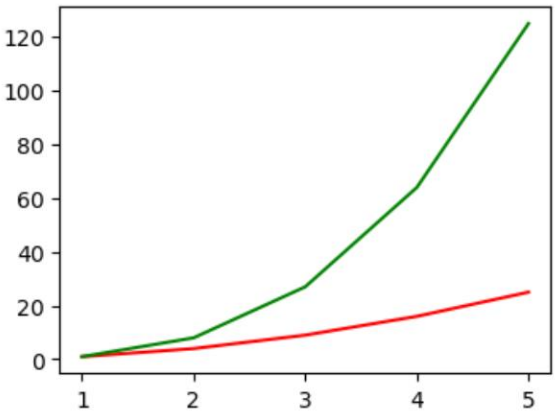
Code for line plot using matplotlib

```
import matplotlib.pyplot as plt
import numpy as np
X=np.array([1,2,3,4,5])
Y =np.array( [1,4,9,16,25])
plt.figure(figsize=(4,3))
plt.plot(X,Y)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Changing line colors

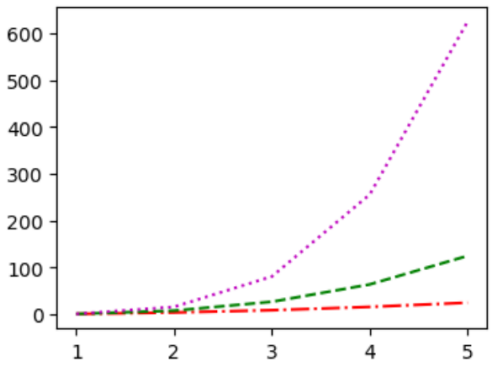
```
import matplotlib.pyplot as plt
X=[1,2,3,4,5]
Y = [1,4,9,16,25]
Z=[1,8,27,64,125]
plt.figure(figsize=(4,3))
plt.plot(X,Y,'r')
plt.plot(X,Z,'g')
plt.show()
```



Some predefined colors and its notations:

Changing line types:

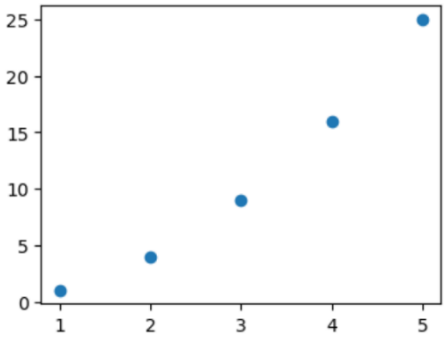
```
import matplotlib.pyplot as plt
import numpy as np
X=np.array([1,2,3,4,5])
Y = X**2
Z=X**3
A=X**4
plt.figure(figsize=(4,3))
plt.plot(X,Y,'-.r')
plt.plot(X,Z,'--g')
plt.plot(X,A,':m')
plt.show()
```



Scatter plot:

Here the points are represented individually with a dot or a circle. We cannot change markers in this type of plot.

```
import matplotlib.pyplot as plt
import numpy as np
X=np.array([1,2,3,4,5])
Y =np.array( [1,4,9,16,25])
plt.figure(figsize=(4,3))
plt.scatter(X,Y)
plt.show()
```



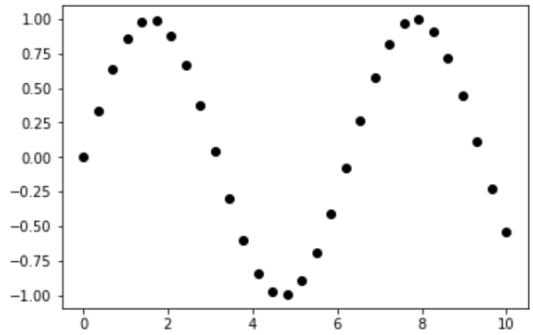
Scatter Plot with plt.plot():

We have used plt.plot() to produce line plots. We can use the same functions to produce the scatter plots as follows:-

X = np.linspace(0, 10, 30)

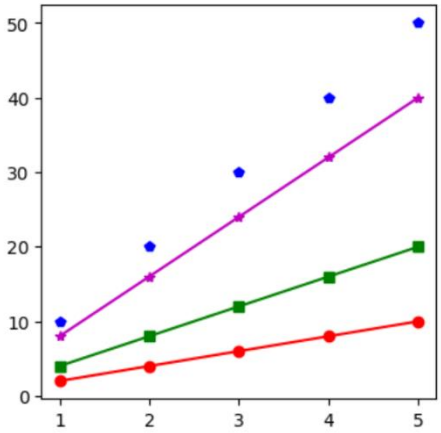
y = np.sin(x7)

plt.plot(X, y, 'o', color = 'black')



Markers:

```
import matplotlib.pyplot as plt
import numpy as np
X=np.array([1,2,3,4,5])
A =2*X;B=4*X;C=8*X;D= 10*X
plt.figure(figsize=(4,4))
#plt.plot(X,A,'o-r')
plt.plot(X,A,'r',marker='o')
plt.plot(X,B,'s-g')
plt.plot(X,C,'*-m')
plt.plot(X,D,'pb')
plt.show()
```

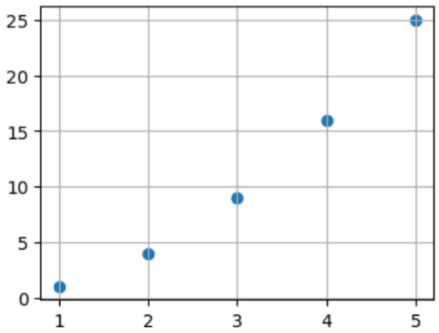


Some markers:

Syntax	Marker	Description
"."		point
","		pixel
"o"		circle
"v"		triangle_down
"^"		triangle_up
“>"		triangle_right
"s"		square
"p"		pentagon
"*"		star
"+"		plus
"d"		diamond

Adding a grid in plot:

```
import matplotlib.pyplot as plt
import numpy as np
X=np.array([1,2,3,4,5])
Y =np.array( [1,4,9,16,25])
plt.figure(figsize=(4,3))
plt.scatter(X,Y)
plt.grid()
plt.show()
```

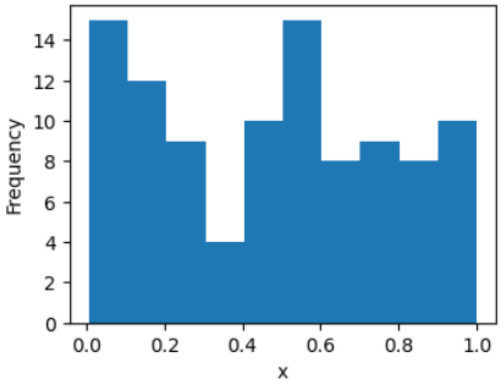


Histogram:

Histogram charts are a graphical display of frequencies. They are represented as bars. They show what portion of the dataset falls into each category, usually specified as non-overlapping intervals. These categories are called bins.

The **plt.hist()** function can be used to plot a simple histogram as follows:-

```
# Histogram
X=np.random.random(100)
plt.hist(X)
plt.xlabel('x')
plt.ylabel('Frequency')
plt.show()
```

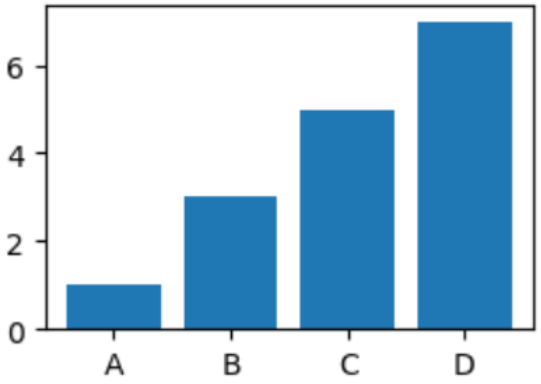


Bar Chart:

Bar charts display rectangular bars either in vertical or horizontal form. Their length is proportional to the values they represent. They are used to compare two or more values.

We can plot a bar chart using **plt.bar()** function. We can plot a bar chart as follows:-

```
X=[1,3,5,7]
Label=['A','B','C','D']
plt.bar(Label,X)
plt.show()
```



Pie chart:

Pie charts are circular representations, divided into sectors. The sectors are also called **wedges**. The arc length of each sector is proportional to the quantity we are describing. It is an effective way to represent information when we are interested mainly in comparing the wedge against the whole pie, instead of wedges against each other.

Matplotlib provides the **plt.pie()** function to plot pie charts from an array X. Wedges are created proportionally, so that each value x of array X generates a wedge proportional to x/sum(X).

```
plt.figure(figsize=(3,3))

X= [35, 25, 20, 20]

labels = ['Computer', 'Electronics', 'Mechanical', 'Chemical']

plt.pie(X, labels=labels);

plt.show()
```