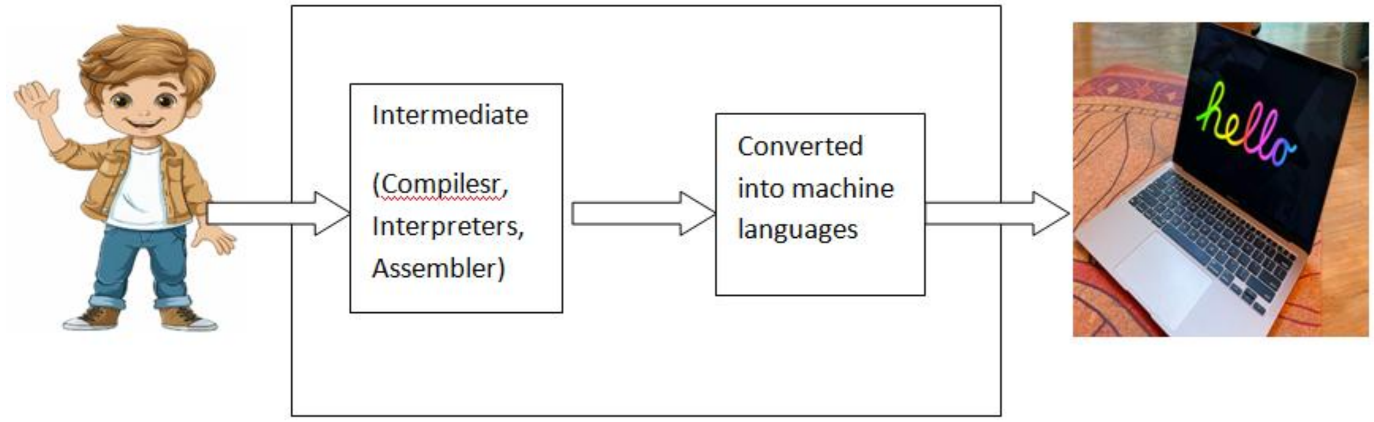
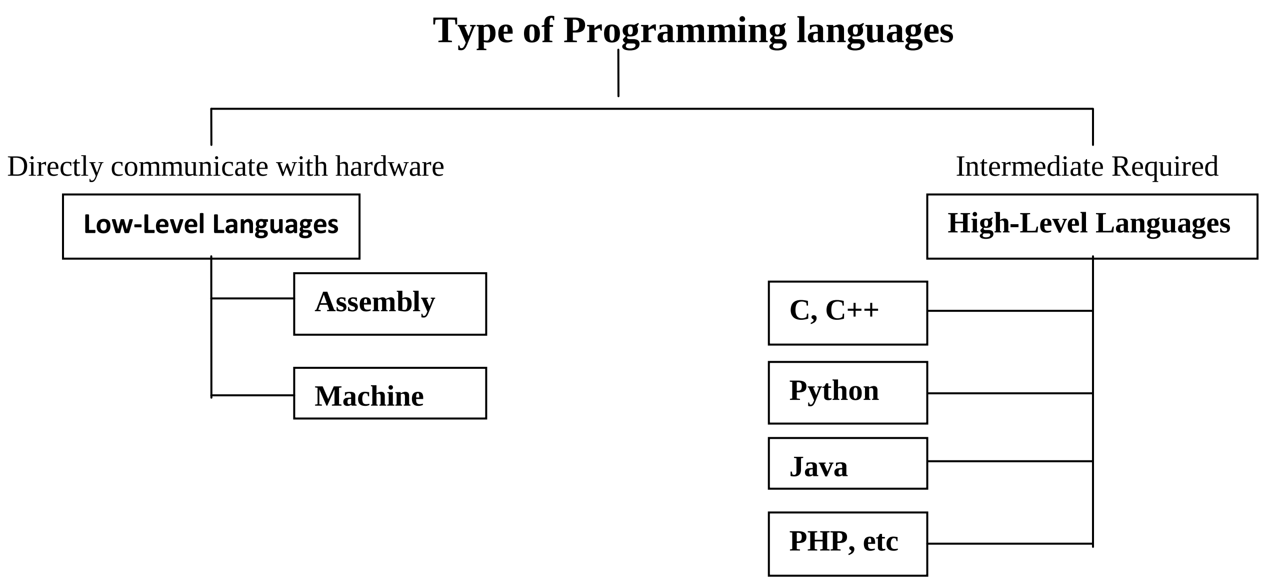
Language

Basics of languages

What is language ?
Why it is required ?
Type of languages ?
What is Low level languages (Machine & assembly language)?
What is high level languages ()?


Intermediate
(Compilesr, Interpreters, Assembler)
Converted into machine languages
hello

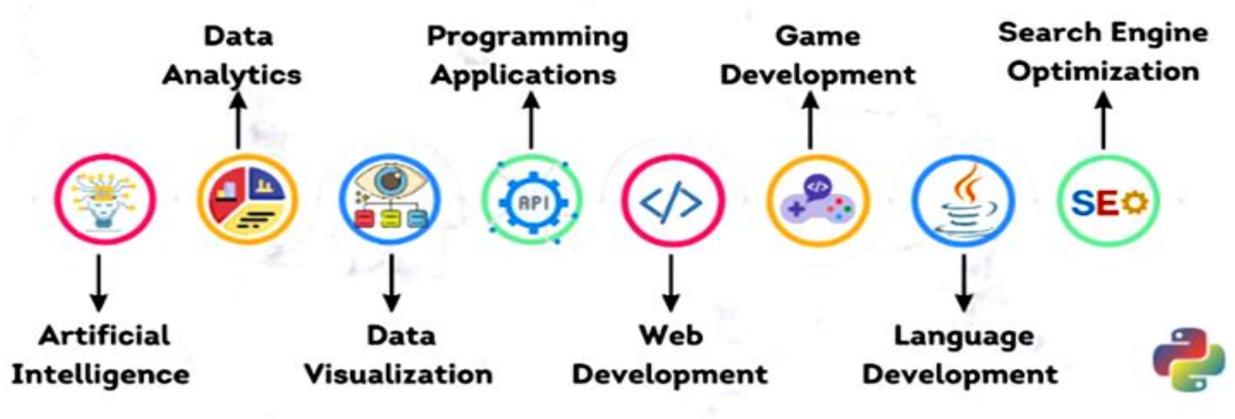


Python Introduction

Python is a :

1. Free and Open Source
2. General-purpose
3. High Level Programming language

That can be used for:



Features/Advantages of Python:

1. Simple and easy to learn
2. Procedure and object oriented

3. Platform Independent
4. Portable
5. Dynamically Typed
6. Both Procedure Oriented and Object Oriented
7. Interpreted
8. Vast Library Support

Syntex:----

Example 1:-

C:

```
#include<stdio.h>
void main()
{
        print("Hello world");
 }
```

Python:

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

Example 2:- To print the sum of 2 numbers

C:

```
 #include <stdio.h>
 void main()
 {
        int a,b;
         a =10;
         b=20;
        printf("The Sum:%d",(a+b));
 }
```

Python:

```
A,b=10,20
print("The Sum:",(a+b))
```

Limitations of Python:

1. **Performance and Speed:** Python is an interpreted language, which means that it is slower than compiled languages like C or Java. This can be a problem for certain types of applications that require high performance, such as real-time systems or heavy computation.

2. **Support for Concurrency and Parallelism:** Python does not have built-in support for concurrency and parallelism. This can make it difficult to write programs that take advantage of multiple cores or processors.
3. **Static Typing:** Python is a dynamically typed language, which means that the type of a variable is not checked at compile time. This can lead to errors at runtime.
4. **Web Support:** Python does not have built-in support for web development. This means that programmers need to use third-party frameworks and libraries to develop web applications in Python
5. **Runtime Errors**

Python can take almost all programming features from different languages:--

1. Functional Programming Features from C
2. Object Oriented Programming Features from C++
3. Scripting Language Features from Perl and Shell Script
4. Modular Programming Features from Modula-3(Programming Language)

Flavors of Python or types of python interpretors:

1. CPython:
It is the standard flavor of Python. It can be used to work with C lanugage Applications

2. Jython or JPython:
It is for Java Applications. It can run on JVM

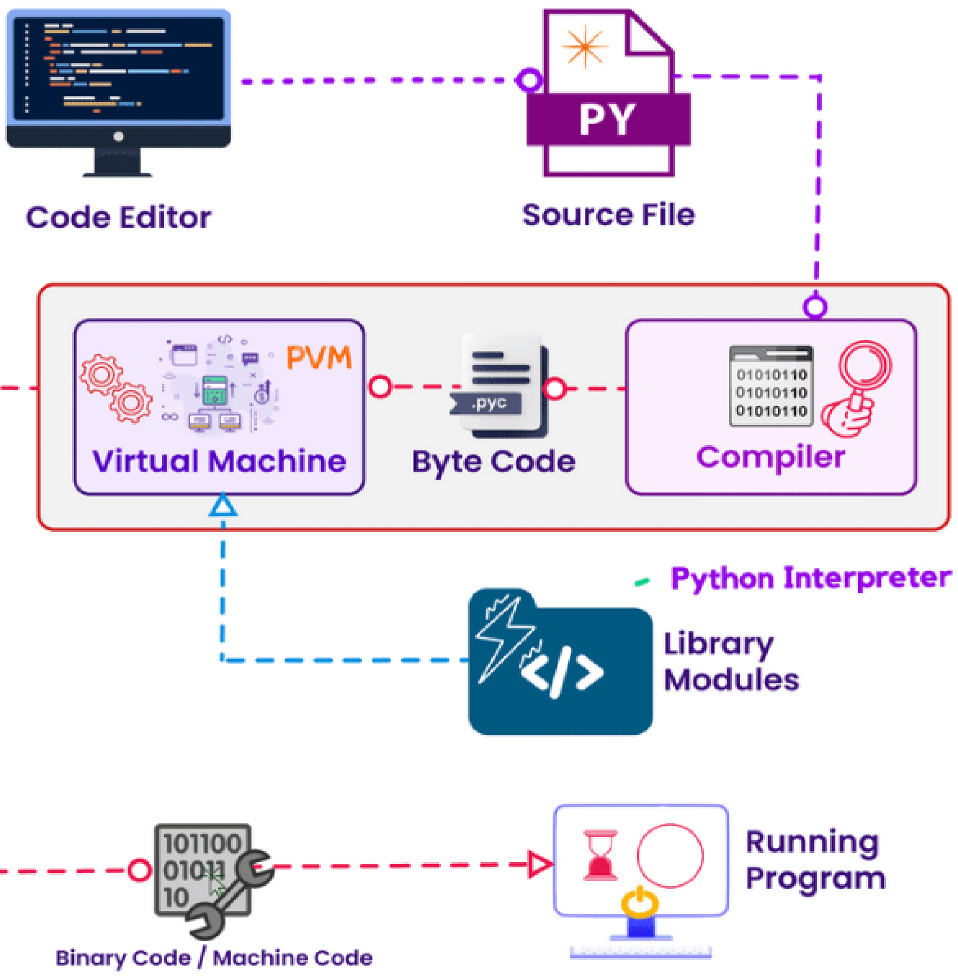
3. IronPython:
It is for C#.Net platform

4. PyPy:
The main advantage of PyPy is performance will be improved because JIT (just in time)compiler is available inside PVM.

5. RubyPython
For Ruby Platforms

6. AnacondaPython
It is specially designed for handling large volume of data processing.

Python Internal working



Python is a high-level, interpreted programming language with a clear syntax, making it user-friendly and widely used in many domains. Here's a breakdown of how Python works internally:

1. Source Code:

You write your Python code using plain text. This code can be written in .py files or interactively using environments like Jupyter Notebooks.

2. Interpreter:

Python is an interpreted language, meaning it doesn't need to be compiled to machine code before execution. The interpreter takes the code and processes it line-by-line. Python has several interpreters, such as CPython (the standard one), PyPy (JIT-compiled), and others.

Python code execution consists of several stages:

3. Tokenization (Lexical Analysis):

- The interpreter reads the source code and breaks it into tokens, which are small meaningful components like keywords, operators, identifiers, etc.
- For example, in the line x = 10, the tokens would be:
  - x (identifier)
  - = (assignment operator)
  - 10 (integer literal)

4. Parsing (Syntax Analysis):

- The tokens are analyzed according to Python's grammar rules to create an abstract syntax tree (AST).
- The AST represents the logical structure of the program, showing the relationships between tokens (e.g., variable assignments, expressions).

5. Compilation:

- Python compiles the AST into intermediate bytecode. This bytecode is a lower-level, platform-independent representation of your program, which Python can execute.
- Python stores bytecode in .pyc files in the __pycache__ folder, which helps with faster execution on subsequent runs.

6. Execution (Python Virtual Machine - PVM):

- The bytecode is executed by the Python Virtual Machine (PVM), which is part of the interpreter. The PVM reads and executes bytecode instructions one by one.
- The PVM manages memory, variables, and control flow during execution.

7. Memory Management:

- Python has a built-in garbage collector that automatically manages memory. It tracks object references and de-allocates memory when objects are no longer needed (based on reference counting and cyclic garbage collection).

8. Modules and Libraries:

- Python's vast standard library provides modules and packages that extend its functionality. External libraries can be added using package managers like pip.

9. Exception Handling:

- If errors occur during execution, Python raises exceptions. The interpreter has built-in mechanisms for handling exceptions, such as try/except blocks.

10. Final Output:

After execution, Python either returns results to the user or modifies the system (e.g., by creating files, interacting with databases, etc.).

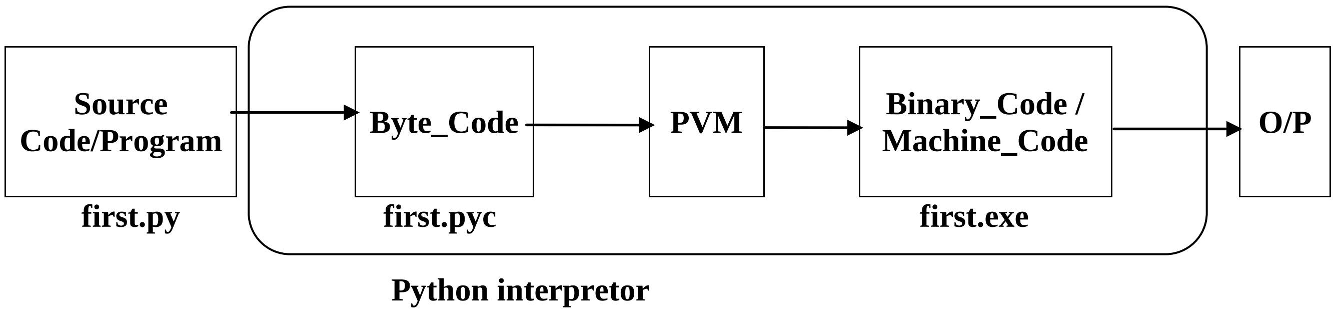
Example Flow:

For a simple code like:

```
x = 5
y = 10
z = x + y
print(z)
```

- **Lexical Analysis**: Tokens are generated (x, =, 5, y, 10, etc.).
- **Parsing**: The AST is constructed.
- **Compilation**: The code is compiled into bytecode.
- **Execution**: The PVM executes the bytecode, computes the sum, and prints 15.

Each of these steps occurs behind the scenes, making Python a powerful and flexible language.



Examples:---
first.py:---

```
a = 10
b = 10
print("Sum ", (a+b))
```

The execution of the Python program involves 2 Steps:

- Compilation
- Interpreter

Compilation

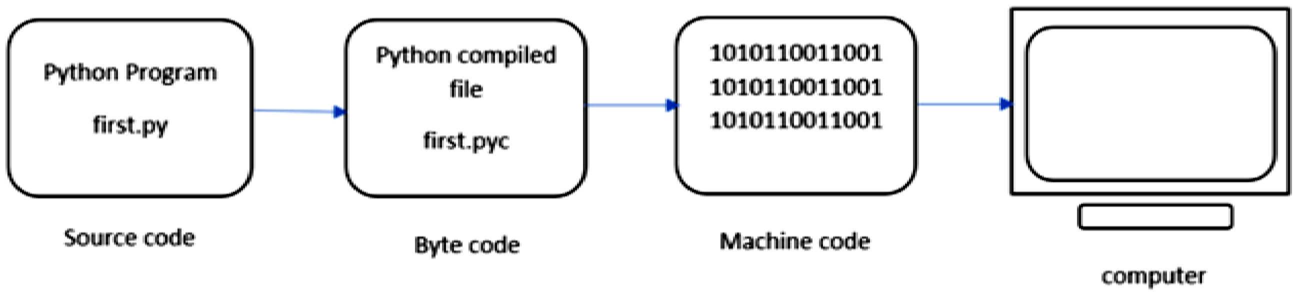
The program is converted into **byte code.** Byte code is a fixed set of instructions that represent arithmetic, comparison, memory operations, etc. It can run on any operating system and hardware. The byte code instructions are created in the **.pyc** file. The .pyc file is not explicitly created as Python handles it internally but it can be viewed with the following command:

```
PS E:\Python_data> python -m py_compile first.py
```

-m and py_compile represent module and module name respectively. This module is responsible to generate .pyc file. The compiler creates a directory named __pycache__ where it stores the first.cpython-310.pyc file.

Interpreter

The next step involves converting the byte code (.pyc file) into machine code. This step is necessary as the computer can understand only machine code (binary code). Python Virtual Machine (PVM) first understands the operating system and processor in the computer and then converts it into machine code. Further, these machine code instructions are executed by processor and the results are displayed.



However, the interpreter inside the PVM translates the program line by line thereby consuming a lot of time. To overcome this, a compiler known as Just In Time (JIT) is added to PVM. JIT compiler improves the execution speed of the Python program. This compiler is not used in all Python environments like CPython which is standard Python software.

To execute the first.cpython-310.pyc we can use the following command:

```
PS E:\Python_data\__pycache__> python first1.cpython-310.pyc
```

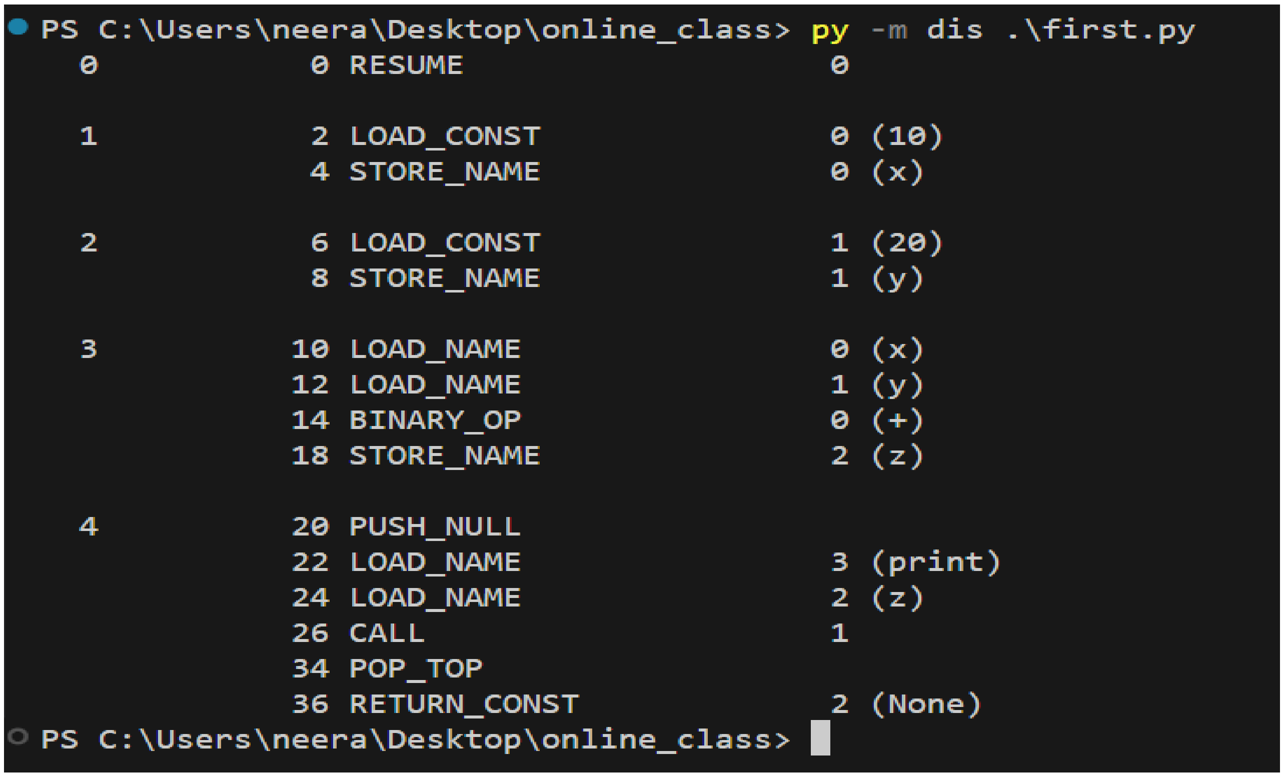
view the byte code of the file – first.py we can type the following command as : first.py:

```
x = 10
y = 10
z=x+y
print(z)
```

The command **python -m dis first.py** disassembles the Python bytecode generated from the source code in the file first.py.

- **python**: This is the command to invoke the Python interpreter.
- **-m dis**: This uses Python's built-in dis module to disassemble the Python bytecode.
    - dis stands for **disassembler**. It translates Python bytecode back into a more readable form, showing the low-level instructions that the Python Virtual Machine (PVM) executes.
- **first.py**: This is the Python script file whose bytecode will be disassembled.

When you run this command, Python compiles first.py into bytecode (if not already compiled), and the dis module disassembles it. This helps you understand the internal bytecode instructions that Python generates from your source code.



```
PS C:\Users\neera\Desktop\online_class> py -m dis .\first.py
  0           0 RESUME                   0

  1           2 LOAD_CONST               0 (10)
              4 STORE_NAME               0 (x)

  2           6 LOAD_CONST               1 (20)
              8 STORE_NAME               1 (y)

  3          10 LOAD_NAME                0 (x)
             12 LOAD_NAME                1 (y)
             14 BINARY_OP                0 (+)
             18 STORE_NAME               2 (z)

  4          20 PUSH_NULL
             22 LOAD_NAME                3 (print)
             24 LOAD_NAME                2 (z)
             26 CALL                     1
             34 POP_TOP
             36 RETURN_CONST             2 (None)
PS C:\Users\neera\Desktop\online_class>
```

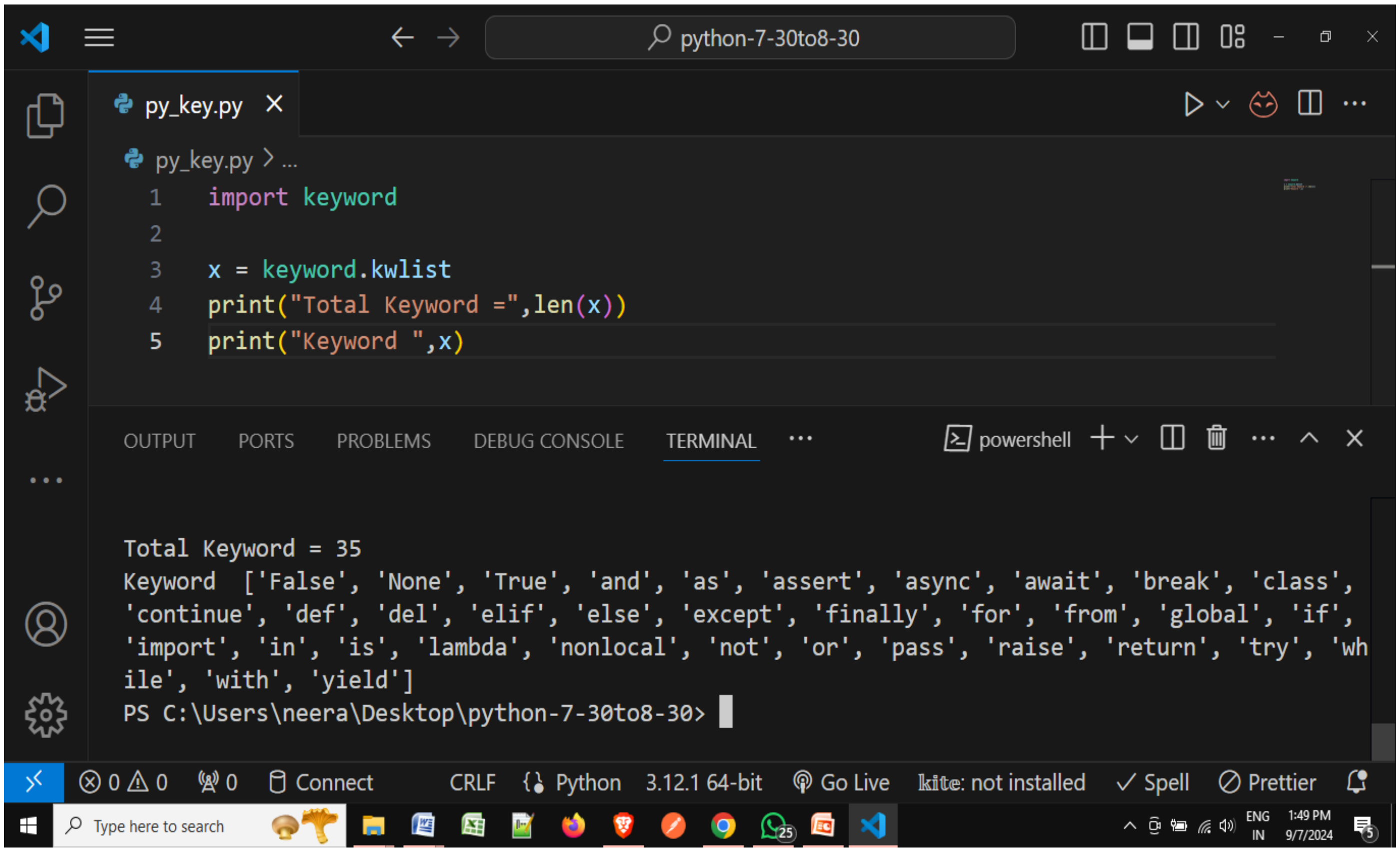
- **LOAD_CONST**: Loads a constant value (like numbers 10 and 20).
- **STORE_NAME**: Stores the value in a variable (like x, y, or z).
- **LOAD_NAME**: Loads the value of a variable from memory.
- **BINARY_ADD**: Adds two values (in this case, the values of x and y).
- **CALL_FUNCTION**: Calls a function (like print).
- **RETURN_VALUE**: Returns from a function (in this case, the main program).

Token:-

In Python, a **token** is the smallest unit of the source code that the Python interpreter recognizes during the process of **lexical analysis** (the first step in code compilation or interpretation). Each token represents a meaningful element in Python, such as

1. Keywords.
2. Punctuation/delimiters.
3. Identifiers.
4. Operators.
5. Literals

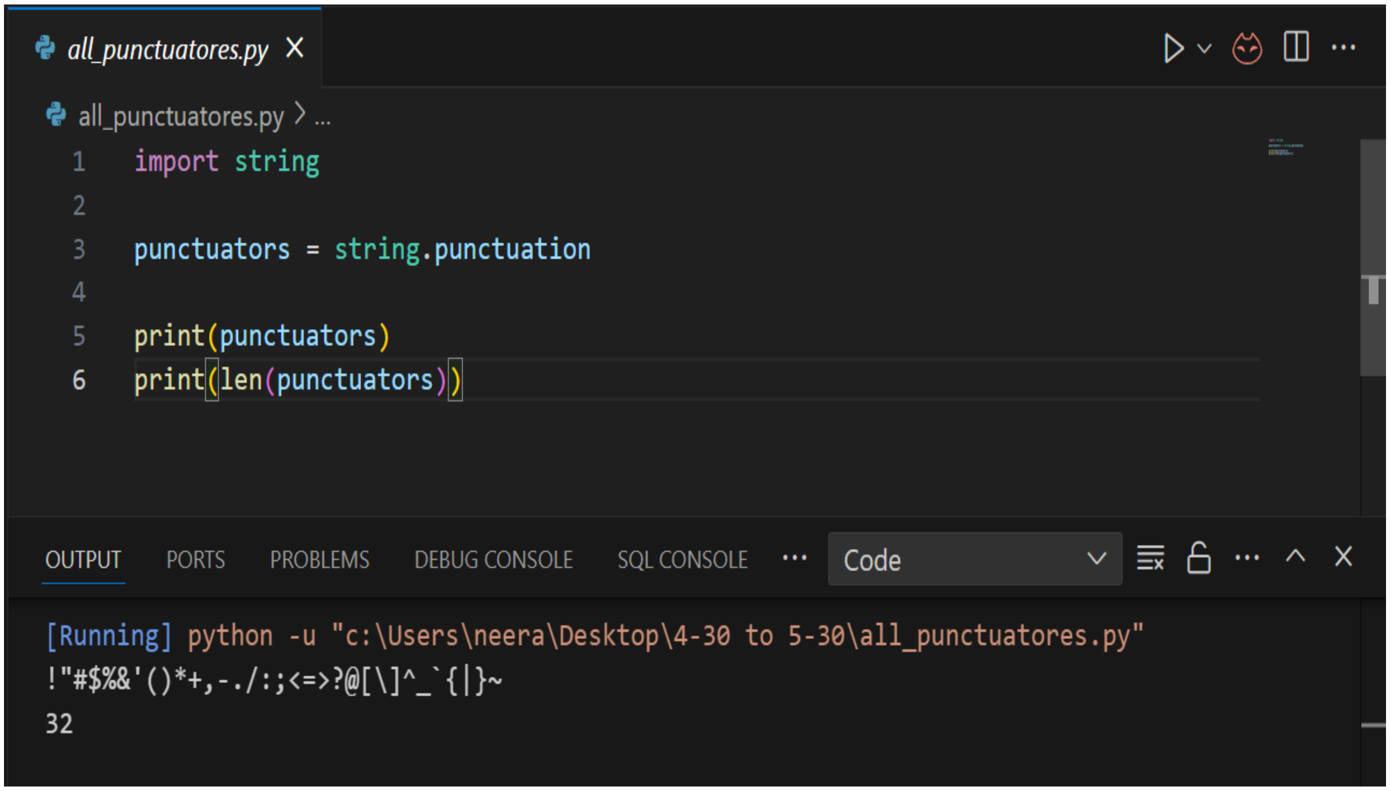
Keywords:



```python
import keyword

x = keyword.kwlist
print("Total Keyword =",len(x))
print("Keyword ",x)
```

```
Total Keyword = 35
Keyword  ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'wh
ile', 'with', 'yield']
PS C:\Users\neera\Desktop\python-7-30to8-30>
```

Punctuations:-



```python
import string

punctuators = string.punctuation

print(punctuators)
print(len(punctuators))
```

```
[Running] python -u "c:\Users\neera\Desktop\4-30 to 5-30\all_punctuatores.py"
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
32
```

Identifiers in Python are names used to identify variables, functions, classes, modules, and other objects. An identifier is a sequence of one or more characters that may consist of letters (both uppercase and lowercase), digits (0-9), and underscores (_).

Rules for Naming Identifiers in Python

1. **Start with a Letter or Underscore**: An identifier must begin with a letter (a-z, A-Z) or an underscore (_). It cannot start with a digit.
2. **Subsequent Characters**: The characters following the initial letter or underscore can be letters, digits, or underscores.
3. **Case Sensitivity**: Identifiers in Python are case-sensitive. This means myVariable, MyVariable, and myvariable are considered three different identifiers.
4. **No Spaces or Special Characters**: Identifiers cannot contain spaces or special characters like !, @, #, $, %, etc., except for the underscore (_).
5. **No Keywords**: Identifiers cannot be the same as Python keywords. Keywords are reserved words in Python that have predefined meanings, such

as if, else, while, for, def, class, etc. You can check the list of keywords using the keyword module.

6. **No Built-in Function Names**: It is not advisable (though technically possible) to use the names of built-in Python functions and modules (like print, list, str, int, etc.) as identifiers, as this can lead to confusion and bugs.

Operators:--

1. **Arithmetic Operators:** The operators which are used to perform arithmetic operations like addition, subtraction, division etc. **(+, -, *, /, %, **, //)**

2. Relational Operators/Comparison Operators: The operators which are used to check for some relation like greater than or less than etc.. between operands are called relational operators. **(<, >, <=, >=, ==, !=)**

3. **Logical Operators:** The operators which do some logical operation on the operands and return True or False are called logical operators. The logical operations can be **'and', 'or', 'not'** etc.
4. **Assignment Operators:** The operators which are used for assigning values to variables. **'='** is the assignment operator.
5. **Unary Operator:** The operator '-' is called the Unary minus operator. It is used to negate the number.
6. Membership Operators: The operators that are used to check whether a particular variable is part of another variable or not. **('in' and 'not in')**
7. **Identity Operators:** The operators which are used to check for identity.
('is' and 'is not')

8. Python Bitwise Operators: Bitwise operators are used to compare (binary) numbers:

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Literals in Python

Literals in Python are constant values that are assigned to variables or used directly in code. Python supports several types of literals:

1. **String Literals**: Enclosed in single ('...'), double ("..."), triple single ('''...'''), or triple double quotes ("""...""").
2. **Numeric Literals**:
    1. **Integer Literals**: Whole numbers, which can be written in decimal, binary (0b...), octal (0o...), or hexadecimal (0x...) form.
    2. **Float Literals**: Numbers with a decimal point or in exponential (scientific) notation.
    3. **Complex Literals**: Numbers with a real and imaginary part, defined by a number followed by a j or J.
3. **Boolean Literals**: True and False, which represent the two truth values of Boolean logic.
4. **Special Literal**: None, which represents the absence of a value or a null value.
5. **Collection Literals**: Literals for creating collections like lists, tuples, dictionaries, and sets.

- **List Literals**: Defined using square brackets [].
- **Tuple Literals**: Defined using parentheses ().
- **Dictionary Literals**: Defined using curly braces {} with key-value pairs.
- **Set Literals**: Defined using curly braces {} with comma-separated values.

Variable in Python?

All the data which we create in the program will be saved in some memory location on the system. The data can be anything, an integer, a complex number, a set of mixed values, etc. A Python variable is a symbolic name that is a reference or pointer to an object. Once an object is assigned to a variable, you can refer to the object by that name.

Python Variable – call by reference

Assign Multiple Values in multiple variables in single line:-

1. Many Values to Multiple Variables

   Example:-
   ```
   x, y, z = "Neeraj", "Ravi", "Rahul"
   print(x)
   print(y)
   print(z)
   ```

2. **One Value to Multiple Variables in single line:**

   Example:-
   ```
   x = y = z = "Neeraj Kumar"
   print(x)
   print(y)
   print(z)
   ```

3. **Advance examples:-**

   Example:-
   ```
   city = ["Bhopal", "Indore", "Jabalpur"]
   x, y, z = city
   print(x)
   print(y)
   print(z)
   ```

Python Comments:-

1. single line comments:--- (# ---------------)   ctrl+/

2. Multi-line comments:---(‘‘‘ ------------
   -----------’’’)

Operator in Python

In programming languages, an operator is a symbol that is applied to some operands (usually variables), to perform certain actions or operations. For example,

```
x = 1
y = 2
z = x+y
print(z)
```

O/P:-

```
3
```

In the above example, first, we assigned values to two variables 'a' and 'b' and then we added both the variables and stored the result in variable 'c'. The operations we performed are:

1. Assignment Operation using '=' operator. We assigned the values to variables 'a', 'b', 'c' using the operator '=' which is called the assignment operator.
2. Addition Operation using '+' operator. The values in variables are added using the '+' operator.

Note: The symbols + and = are called operators and the variables a,b,c on which operation is being performed are called operands.

BASIC CLASSIFICATION OF OPERATORS IN PYTHON

Unary Operator –A Unary Operator is a computational operator that takes any action on one operand and produces only one result. For example, the "-" binary operator in Python turns the operand negative (if positive) and positive (if negative).

So, if the operator acts on a single operand then it's called the unary operator. For example, in '-5' the operator '-' is used to make the (single) operand '5' a negative value hence called the unary operator. **Ex:-1**

```
# Unary Operator.
x=10
y=-(x)
print("For - unary operator:",y)
x=10
y=+(x)
print("For + unary operator:",y)
```

O/P:

```
-10
10
```

Binary Operator – If the operator acts on two operands then it's called a binary operator. For example, + operators need two variables (operands) to perform some operation, hence called binary operator.

```
# Binary Operator
# for + operator
x = 5
y = 4
z = x+y
print("For + binary operator:",z)
# for - operator
x = 5
y = 4
z = x-y
print("For - binary operator:",z)
# for * operator
x = 5
y = 4
z = x*y
print("For * binary operator:",z)
# for / operator
x = 5
y = 4
z = x/y
print("For / binary operator:",z)
# for %(Modulus) operator
x = 5
y = 4
z = x%y
print("For % binary operator:",z)
# for //(floor division) operator
x = 5
y = 4
z = x//y
print("For - binary operator:",z)
```

O/P:-

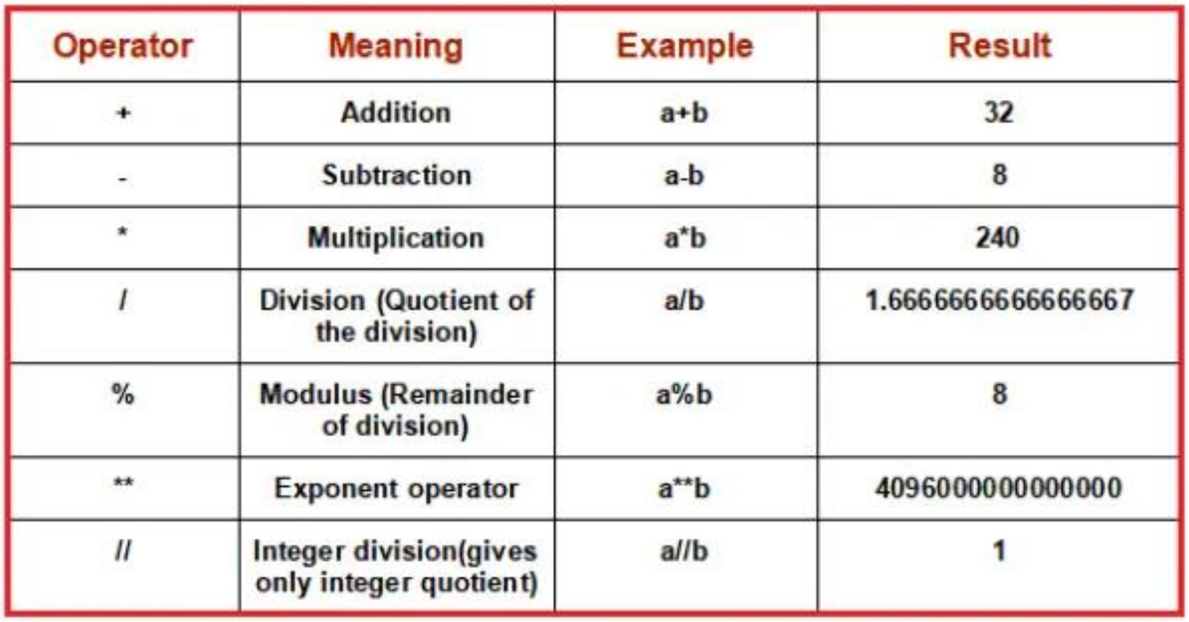
```
For + binary operator: 9
For - binary operator: 1
For * binary operator: 20
For / binary operator: 1.25
For % binary operator: 1
```

For - binary operator: 1

Ternary Operator – If the operator acts on three operands then it’s called a ternary operator. In general, the ternary operator is a precise way of writing conditional statements.
Syntax:
x = true_value if **condition** else **false_value**
The three operands here are:
1. condition
2. true_value
3. False_value

```
# Ternary Operator
a, b = 10, 20
min = a if a < b else b
print("Ternary Operator(min): ",min)
O/P:- Ternary Operator(min): 10
```

ARITHMETIC OPERATORS IN PYTHON:
As stated above, these are used to perform that basic mathematical stuff as done in every programming language. Let’s understand them with some examples. Let’s assume, a = 20 and b = 12



Operator	Meaning	Example	Result
+	Addition	a+b	32
-	Subtraction	a-b	8
*	Multiplication	a*b	240
/	Division (Quotient of the division)	a/b	1.6666666666666667
%	Modulus (Remainder of division)	a%b	8
**	Exponent operator	a**b	4096000000000000
//	Integer division(gives only integer quotient)	a//b	1

Note: Division operator / always performs floating-point arithmetic, so it returns a float value. Floor division (//) can perform both floating-point and integral as well,

1. If values are int type, the result is int type.
2. If at least one value is float type, then the result is of float type.

Example: Arithmetic Operators in Python:

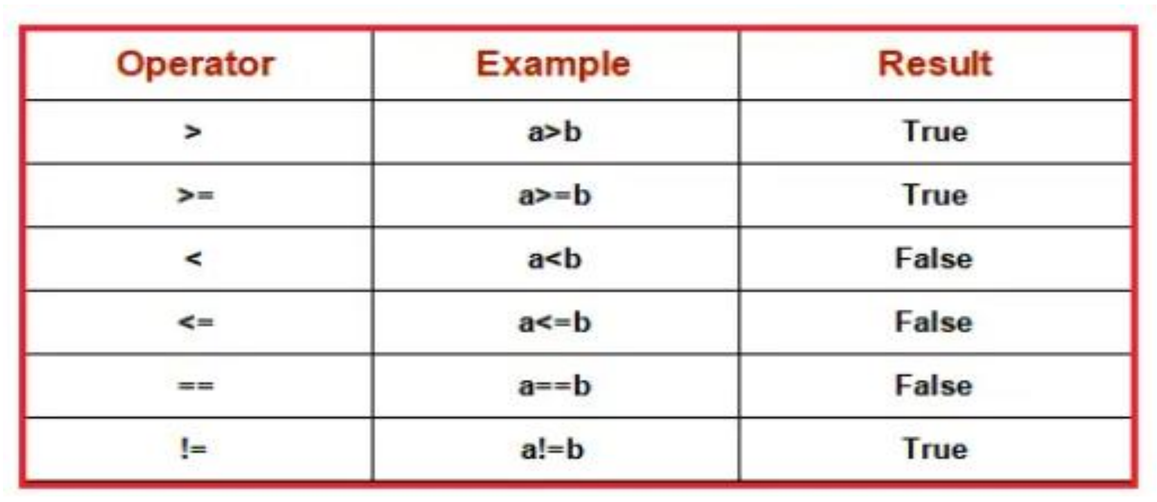
```
a = 20
b = 12
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a%b)
print(ab)
print(a//b)


O/P:-
32
8
240
1.6666666666666667
8
4096000000000000
1
```

Example: Floor division

```
print(12//5)
print(12.0//5)

O/P:-
2
2.0
```

Relational Operators in Python:-

These are used to compare two values for some relation and return True or False depending on the relation. Let’s assume, a = 13 and b = 5.



Operator	Example	Result
>	a>b	True
>=	a>=b	True
<	a<b	False
<=	a<=b	False
==	a==b	False
!=	a!=b	True

Example: Relational Operators in Python

```
a = 13
b = 5
print(a>b)
print(a>=b)
print(a<b)
print(a<=b)
print(a==b)
print(a!=b)

O/P:-
True
True
False
False
False
True
```

LOGICAL OPERATORS IN PYTHON:-

In python, there are three types of logical operators. They are and, or, not. These operators are used to construct compound conditions, combinations of more than one simple condition. Each simple condition gives a boolean value which is evaluated, to return the final boolean value.

Note: In logical operators, False indicates 0(zero) and True indicates non-zero value. Logical operators on boolean types

1. **and**: If both the arguments are True then only the result is True
2. **or**: If at least one argument is True then the result is True
3. **not**: the complement of the boolean value

Example: Logical operators on boolean types in Python

```
a = True
b = False
print(a and b)
print(a or b)
print(not a)
print(a and a)

O/P:-
False
True
False
True
```

and operator:

‘A and B’ returns A if A is False
‘A and B’ returns B if A is not False

Or Operator in Python:

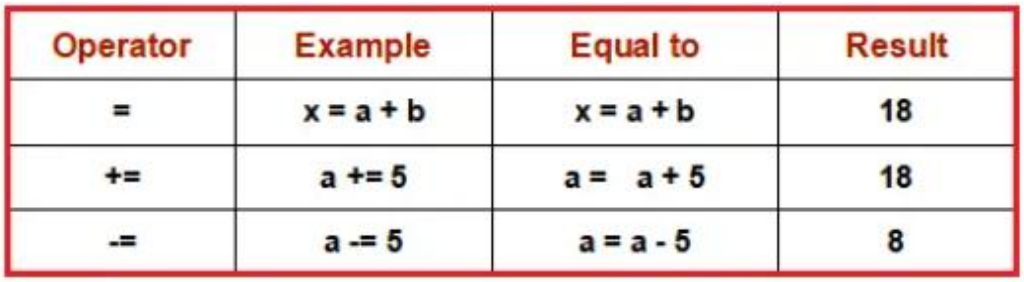
‘A or B’ returns A if A is True
‘A or B’ returns B if A is not True

Not Operator in Python:

not A returns False if A is True
not B returns True if A is False

ASSIGNMENT OPERATORS IN PYTHON

By using these operators, we can assign values to variables. ‘=’ is the assignment operator used in python. There are some compound operators which are the combination of some arithmetic and assignment operators (+=, -=, *=, /=, %=, **=, //=). Assume that, a = 13 and b = 5



Operator	Example	Equal to	Result
=	x = a + b	x = a + b	18
+=	a += 5	a = a + 5	18
-=	a -= 5	a = a - 5	8

Example: Assignment Operators in Python

```
a=13
print(a)
a+=5
print(a)

O/P:-
13
18
```

UNARY MINUS OPERATOR(-) IN PYTHON:

This operator operates on a single operand, hence unary operator. This is used to change a positive number to a negative number and vice-versa.

Example: Unary Minus Operators in Python

```
a=10
print(a)
print(-a)

O/P:-
10
-10
```

MEMBERSHIP OPERATORS IN PYTHON

Membership operators are used to checking whether an element is present in a sequence of elements are not. Here, the sequence means strings, list, tuple, dictionaries, etc which will be discussed in later chapters. There are two membership operators available in python i.e. in and not in.

1. **in operator:** The in operators returns True if element is found in the collection of sequences. returns False if not found
2. **not in operator:** The not-in operator returns True if the element is not found in the collection of sequence. returns False in found

Example: Membership Operators in Python

```
text = "Welcome to python programming"
print("Welcome" in text)
print("welcome" in text)
print("nireekshan" in text)
print("Hari" not in text)

O/P:-
True
False
False
True
```

Example: Membership Operators in Python

```
names = ["Ramesh", "Nireekshan", "Arjun", "Prasad"]
print("Nireekshan" in names)
print("Hari" in names)
print("Hema" not in names)

O/P:-
True
False
True
```

IDENTITY OPERATOR IN PYTHON

This operator compares the memory location(address) to two elements or variables or objects. With these operators, we will be able to know whether the two objects are pointing to the same location or not. The memory location of the object can be seen using the id() function.

Example: Identity Operators in Python

```
a = 25
b = 25
print(id(a))
print(id(b))

O/P:-
1487788114928
1487788114928
```

Types of Identity Operators in Python:

There are two identity operators in python, is and is not.

is:

1. A is B returns True, if both A and B are pointing to the same address.
2. A is B returns False, if both A and B are not pointing to the same address.

is not:

1. A is not B returns True, if both A and B are not pointing to the same object.
2. A is not B returns False, if both A and B are pointing to the same object.

Example: Identity Operators in Python

```
a = 25
b = 25
print(a is b)
print(id(a))
print(id(b))

O/P:-
True
2873693373424
2873693373424
```

Example: Identity Operators

```
a = 25
b = 30
print(a is b)
print(id(a))
print(id(b))
```

```
O/P:-
False
1997786711024
1997786711184
```

Note: The ‘is’ and ‘is not’ operators are not comparing the values of the objects. They compare the memory locations (address) of the objects. If we want to compare the value of the objects. we should use the relational operator ‘==’.

Example: Identity Operators

```
a = 25
b = 25
print(a == b)

O/P:-
True
```

Input and Output in Python
Why it is required?

Example: Hardcoded values to a variable:

```
age = 18
if age>=18:
    print("eligible for vote")
else:
    print("Not eligible for vote")


age = 17
if age>=18:
    print("eligible for vote")
else:
    print("Not eligible for vote")

O/P:-
eligible for vote
Not eligible for vote
```

A predefined function input() is available in python to take input from the keyboard during runtime. This function takes a value from the keyboard and returns it as a string type. Based on the requirement we can convert from string to other types.

Now, if we want to take value of age at runtime, then we use python-inbuilt function **input().**

```
age=input("Enter Your age: ")
if age>=18:
    print("eligible for vote")
else:
    print("Not eligible for vote")

O/P:-
Enter Your age: 15
Traceback (most recent call last):
  File "E:\DataSciencePythonBatch\input_output.py", line 16, in <module>
    if age>=18:
TypeError: '>=' not supported between instances of 'str' and 'int'
```

We can convert the string to other data types using some inbuilt functions. We shall discuss all the type conversion types in the later chapters. As far as this chapter is concerned, it’s good to know the below conversion functions.

1. **string to int – int() function**
2. **string to float – float() function**

```
age=input("Enter Your age: ")
print(type(age))
age=int(age)
print(type(age))
if age>=18:
    print("eligible for vote")
else:
    print("Not eligible for vote")

O/P:-
Enter Your age: 15
<class 'str'>
<class 'int'>
```

```
Not eligible for vote
```

```
# perform operations through input function.
x=int(input("Enter first No: "))
y=int(input("Enter second No: "))

z= x+y
print("Addition of x and y :",z)

O/P:-
Enter first No: 5
Enter second No: 4
Addition of x and y : 9
```

Eval() function in python:---

This is an in-built function available in python, which takes the strings as an input. The strings which we pass to it should, generally, be expressions. The eval() function takes the expression in the form of a string and evaluates it and returns the result.

Examples,

```
print(eval('10+5'))
print(eval('10-5'))
print(eval('10*5'))
print(eval('10/5'))
print(eval('10//5'))
print(eval('10%5'))

O/P:-
15
5
50
2.0
2
0
```

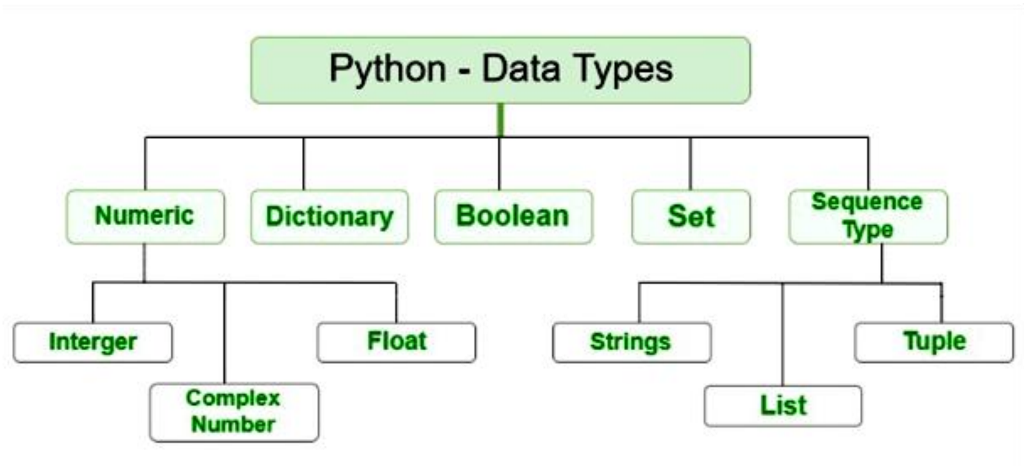
```
value = eval(input("Enter expression: "))
print(value)
```

```
O/P:
Enter expression: 5+10
15
```

```
value = eval(input("Enter expression: "))
print(value)

O/P:
Enter expression: 12-2
10
```

Data Types:-

Data Type represent the type of data present inside a variable.
In Python we are not required to specify the type explicitly. Based on value provided, the type will be assigned automatically. Hence Python is Dynamically Typed Language.



Fundamental Data Types in Python:
In Python, the following data types are considered as Fundamental Data types,

1. **Int**
2. **Float**
3. **Complex**
4. **Bool**
5. **Str**

Note: Python contains several inbuilt functions

1. type():- type() is an in-built or pre-defined function in python that is used to check the data type of the variables. The following example depicts the usage.
Example:-

```
emp_id = 11
name = 'Neeraj'
salary = 50000.40
print("emp_id type is: ", type(emp_id))
print("name type is: ", type(name))
print("salary type is: ", type(salary))

O/P---
```

```
emp_id type is:  <class 'int'>
name type is:  <class 'str'>
salary type is:  <class 'float'>
```

2. id():- to get address of object

```
emp_id = 11
name = 'Neeraj'
salary = 50000.40
print("emp_id id is: ", id(emp_id))
print("name id is: ", id(name))
print("salary id is: ", id(salary))

O/P---
emp_id id is:  3146509648432
name id is:  3146515054320
salary id is:  3146510689840
```

3. print():- to print the value

```
emp_id = 11
name = 'Neeraj'
salary = 50000.40
print("My employee id is: ", emp_id)
print("My name is: ", name)
print("My salary is: ", salary)

O/P---
My employee id is:  11
My name is:  Neeraj
My salary is:  50000.4
```

int data type:
The int data type represents values or numbers without decimal values. In python, there is no limit for the int data type. It can store very large values conveniently.

```
a=10
type(a) O/P:- <class 'int'>
```

Note: In Python 2nd version long data type was existing but in python 3rd version long data type was removed.

We can represent int values in the following ways
1. Decimal form **(bydefault)**
2. Binary form
3. Octal form
4. Hexa decimal form

1. Decimal form(base-10):
It is the default number system in Python. The allowed digits are: 0 to 9
Ex: a =10

2. Binary form(Base-2):
The allowed digits are : 0 & 1
Literal value should be prefixed with 0b or 0B
Eg: a = 0B1111
a =0B123
a=b111

3. Octal Form(Base-8):
The allowed digits are : 0 to 7
Literal value should be prefixed with 0o or 0O.

```
Ex:   a=0o123
      a=0o786
```

4. Hexa Decimal Form(Base-16):
The allowed digits are : 0 to 9, a-f (both lower and upper cases are allowed)
Literal value should be prefixed with 0x or 0X.

```
Ex:  a=0X9FcE
     a=0x9aDF
```

Note: Being a programmer we can specify literal values in decimal, binary, octal and hexa decimal forms. But PVM will always provide values only in decimal form.

```
# binary data type(Base-2)
x= 0b1111
y= 0B1010

# by default converted into decimal
print(x) # O/P-15
print(y) # O/P-10


# octal data type(Base-8)
x=0o765
y=0O542
# by default converted into decimal
print(x) # O/P-501
print(y) # O/P-354

# decimal data type(Base-10)---default
x=10
y=50
# by default converted into decimal
print(x) # O/P-10
print(y) # O/P-50


# Hexadecimal data type(Base-16)
x=0X8EA
z=0x5eA
# by default converted into decimal
print(x) # O/P-2282
print(y) # O/P-50
```

Base Conversions:--- Python provide the following in-built functions for base conversions

```

# Base Conversions
# bin()
```

```
print(bin(15)) # o/p- 0b1111
print(bin(0o11)) # o/p- 0b1001
print(bin(0X10)) # o/p- 0b10000


# oct()
print(oct(10)) # o/p-0o12
print(oct(0B1111)) # o/p-0o17
print(oct(0X123)) # o/p-0o443

# hex()
print(hex(100)) # o/p-0x64
print(hex(0B111111)) # o/p-0x3f
print(hex(0o12345)) # o/p- 0x14e5
```

Float Data Type in Python:

The float data type represents a number with decimal values. floating-point numbers can also be written in scientific notation. e and E represent exponentiation. where e and E represent the power of 10. For example, the number 2 * 10pow2 is written as 2E2, such numbers are also treated as floating-point numbers.

```
salary = 50.5
print(salary)
print(type(salary))

O/P---
50.5
<class 'float'>
```

Example: Print float values

```
a = 2e2 # 2*10^2  e stands for 10 to the power
b = 2E2 # 2*10^2
c = 2e3 # 2*10^3
d = 2e1
```

```
print(a)
print(b)
print(c)
print(d)
print(type(a))

O/P----
200.0
200.0
2000.0
20.0
<class 'float'>
```

Complex Data Type in python:

The complex data type represents the numbers that are written in the form of a+bj or a-bj, here a is representing a real part of the number and b is representing an imaginary part of the number. The suffix small j or upper J after b indicates the square root of -1. The part "a" and "b" may contain integers or floats.

```
a = 3+5j
b = 2-5.5j
c = 3+10.5j
print(a)
print(b)
print(c)
print()
print("A+B=",a+b)
print("B+C=",b+c)
print("C+A=",c+a)
print("A*B=",a*b)
print("B*C=",b*c)
print("C*A=",c*a)
print("A+B+C=", a+b+c)
print("A/B=",a/b)

O/P---
(3+5j)
(2-5.5j)
(3+10.5j)
```

```
A+B= (5-0.5j)
B+C= (5+5j)
C+A= (6+15.5j)
A*B= (33.5-6.5j)
B*C= (63.75+4.5j)
C*A= (-43.5+46.5j)
A+B+C= (8+10j)
A/B= (-0.6277372262773723+0.7737226277372262j)
```

Boolean data type in Python:

The bool data type represents Boolean values in python. bool data type having only two values are, True and False. Python internally represents, True as 1(one) and False as 0(zero). An empty string (" ") represented as False.

Example: Printing bool values:

```
a = True
b = False
print(a)
print(b)
print(a+a)
print(a+b)

O/P---
True
False
2
1
```

None data type in Python:

None data type represents an object that does not contain any value. If any object has no value, then we can assign that object with None type.

Example: Printing None data type:

```
a = None
print(a)
print(type(a))

O/P------
None
```

```
<class 'NoneType'>
```

Sequences in python:

Sequences in Python are objects that can store a group of values. The below data types are called sequences.

1. Str---Immutable
2. Bytes (as a list but in range of o to 256 (256 not included))--Immutable
3. Bytearray---- mutable
4. List-----mutable
5. Tuple----Immutable
6. Range

str data type in python:

A string is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it. A group of characters enclosed within single quotes or double quotes or triple quotes is called a string.

```
# string data type
name1 = 'Neeraj'
name2 = "Neeraj"
name3 = """Neeraj"""

O/P---
Neeraj
Neeraj
Neeraj
```

Bytes Data Type in Python:

Bytes data type represents a group of numbers just like an array. It can store values that are from 0 to 256. The bytes data type cannot store negative numbers. To create a byte data type. We need to create a list. The created list should be passed as a parameter to the bytes() function.

Note: The bytes data type is immutable means we cannot modify or change the bytes object. We can iterate bytes values by using for loop.

Example: creating a bytes data type:

```
# creating a bytes data type
x = [15, 25, 150, 4, 15,19]
y = bytes(x)
print(type(y))

O/P---<class 'bytes'>
```

Example: Accessing bytes data type elements using index

```
# Accessing data by using index
x = [15, 25, 150, 4, 15]
y = bytes(x)
print(y[0])
print(y[1])
print(y[2])
print(y[3])
print(y[4])

O/P---
25
150
4
15
```

Example: Printing the byte data type values using for loop

```
# Bytes data type
x = [15, 25, 150, 4, 15]
y = bytes(x)
for i in y:
    print(i)


O/P---
15
25
```

```
150
4
15
```

Example: To check Values must be in range 0,256

```
x = [10, 20, 300, 40, 15]

y = bytes(x)
```

Output: ValueError: bytes must be in range(0, 256)

Example: To check Byte data type is immutable

```
x = [10, 20, 30, 40, 15]

y = bytes(x)

y[0] = 30
```

Output: TypeError: ‘bytes’ object does not support item assignment

The bytearray data type is the same as the bytes data type, but bytearray is mutable means we can modify the content of bytearray data type. To create a bytearray

1. We need to create a list
2. Then pass the list to the function bytearrray().
3. We can iterate bytearray values by using for loop.

Example: Creating bytearray data type

```
x = [10, 20, 30, 40, 15]
y = bytearray(x)
print(type(y))
```

Example: Accessing bytearray data type elements using index

```
x = [10, 20, 30, 40, 15]

y = bytearray(x)
```

```
print(y[0])

print(y[1])

print(y[2])

print(y[3])

print(y[4])
```

Example: Printing the byte data type values using for loop

```
x = [10, 20, 00, 40, 15]

y = bytearray(x)

for a in y:

print(a)
```

Example: Values must be in the range 0, 256

```
x = [10, 20, 300, 40, 15]

y = bytearray(x)
```

Output: ValueError: bytes must be in range(0, 256)

Example: Bytearray data type is mutable

```
x = [10, 20, 30, 40, 15]

y = bytearray(x)

print("Before modifying y[0] value: ", y[0])

y[0] = 30

print("After modifying y[0] value: ", y[0])
```

List Data Type in Python:

We can create a list data structure by using square brackets []. A list can store different data types. **The list is mutable.**

Tuple Data Type in Python:

We can create a tuple data structure by using parenthesis (). A tuple can store different data types. **A tuple is immutable.**

Set Data Type:

We can create a set data structure by using parentheses symbols (). The set can store the same type and different types of elements.

Dictionary Data Type in Python:

We can create dictionary types by using curly braces {}. The dict represents a group of elements in the form of key-value pairs like a map.

----: Indexing in Python :---

Indexing is the process of accessing an element in a sequence using its position in the sequence (its index).In Python, indexing starts from 0, which means the first element in a sequence is at position 0, the second element is at position 1, and so on. To access an element in a sequence, you can use square brackets [] with the index of the element you want to access.

In Python, indexing refers to the process of accessing a specific element in a sequence, such as a string or list, using its position or index number. Indexing in Python starts at 0, which means that the first element in a sequence has an index of 0, the second element has an index of 1, and so on.

For example, if we have a string "HELLO", we can access the first letter "H" using its index 0 by using the square bracket notation: `string[0]`

Positive Index	**0**	**1**	**2**	**3**	**4**
	H	**E**	**L**	**L**	**O**
Negative Index	**-5**	**-4**	**-3**	**-2**	**-1**

Python's built-in `index()` function is a useful tool for finding the index of a specific element in a sequence. This function takes an argument representing the value to search for and returns the index of the first occurrence of that value in the sequence.

If the value is not found in the sequence, the function raises a ValueError. For example, if we have a list `[1, 2, 3, 4, 5]`, we can find the index of the value `3` by calling `list.index(3)`, which will return the value `2` (since `3` is the third element in the list, and indexing starts at 0).

Python Index Examples

The method index() returns the lowest index in the list where the element searched for appears. If any element which is not present is searched, it returns a **ValueError**.

Example:--

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
element = 3
print(list.index(element))

O/P:-
2
```

Example:--(Throws a ValueError)

```
list = [4, 5, 6, 7, 8, 9, 10]
element = 3 # Not in the list
print(list.index(element))

O/P:-
Traceback (most recent call last):
  File "e:\DataSciencePythonBatch\index.py", line 7, in <module>
    print(list.index(element))
ValueError: 3 is not in list
```

Example:--(Index of a string element)

```
list = [1, 'two', 3, 4, 5, 6, 7, 8, 9, 10]
element = 'two'
print(list.index(element))

O/P:-
1
```

What does it mean to return the lowest index?

```
list = [3, 1, 2, 3, 3, 4, 5, 6, 3, 7, 8, 9, 10]
element = 3
print(list.index(element))
O/P:-
0
```

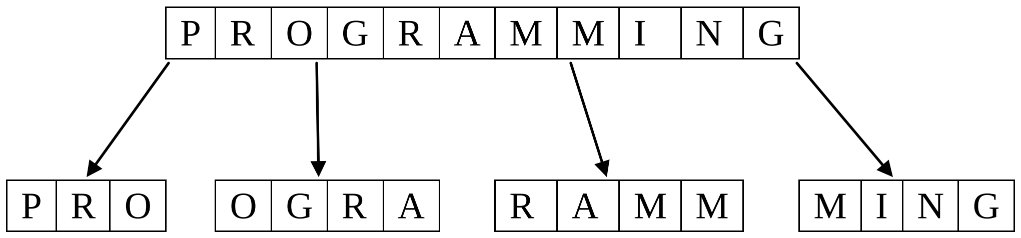
Find element with particular start and end point:--

Syntax:-list_name.index(element, start, stop)

Example:- index() provides you an option to give it hints to where the value searched for might lie.

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
element = 7
print(list.index(element, 5, 8))

O/P:-
6
```

-----: Slicing in Python:-----



Slicing is the extraction of a part of a string, list, or tuple. It enables users to access the specific range of elements by mentioning their indices.

Syntax: Object [start : stop : step]
Object [start : stop]

- **start:** The start parameter in the slice function is used to set the starting position or index of the slicing. The default value of the start is 0.
- **stop:** The stop parameter in the slice function is used to set the end position or index of the slicing[(n-1) for positive value and (n+1) for negative value].
- **step:** The step parameter in the slice function is used to set the number of steps to jump. The default value of the step is 1.

Rules for working :---

Step1:**--** Need to check step direction by default it's goes to positive direction.

Setp2:- Need to check start-point and end-point direction.

Step3:-If both directions are matched, then working fine.

Step4:- Otherwise it gives empty subsequence.

-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
I		L	O	V	E		P	Y	T	H	O	N
0	1	2	3	4	5	6	7	8	9	10	11	12

Ex:-1

```
var = "I love python"
print(var[::])

O/P:-
I love python
```

Ex:2

```
var = "I love python"
print(var[::-1])

O/P:-
nohtyp evol I
```

Ex:-3

```
var = "I love python"
print(var[-2:-5:])

O/P:-
```

Ex:-4

```
var = "I love python"
print(var[2:5:-1])

O/P:-

```

Ex:-5

```
var = "I love python"
print(var[::2])

O/P:-
Ilv yhn
```

Ex:-6

```
var = "I love python"
print(var[::-2])
O/P:-
nhy vlI
```

-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
W	E	L	C	O	M	E		T	O		M	Y		B	L	O	G
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Ex:-7,8,9,10,11,12

```
var = "WELCOME TO MY BLOG"
print(var[3:18]) O/P:- COME TO MY BLOG

print(var[2:14:2]) O/P:- LOET Y

print(var[:7]) O/P:- WELCOME

print(var[8:-1:1]) O/P:- TO MY BLO

print(var[-6:-9:-3]) O/P:- Y

print(var[-9:-9:-1]) O/P:-
```

----:String in Python:----

A group of characters enclosed within single or double or triple quotes is called a string. We can say the string is a sequential collection of characters.

```
s1 = "Welcome to 'python' learning"
s2 = 'Welcome to "python" learning'
s3 = """Welcome "to" 'python' learning"""
print(s1)
print(s2)
print(s3)

O/P:--
Welcome to 'python' learning
Welcome to "python" learning
Welcome "to" 'python' learning
```

In-built functions:--

1. len()	- To check how many objects/characters present in string.
2. max()	-To check which object/character may have maximum ASCII value.
3. min()	-To check which object/character may have minimum ASCII value.
4. type()	-To check data-type
5. str()	-For type casting
6. ord()	-Character to ASCII value
7. chr()	-ASCII value to character

```
str1="Neeraj"
print(max(str1))
print(min(str1))
print(len(str1))
print(type(str1))

# find ACII value of any charactor(ord() take only one argument)
for i in str1:
   print(ord(i))
```

```
# find ACII value of any special symbol(ord() take only one argument)
x='#'
print("ASCII value of X=",ord(x))
```

```

O/P:--
r
N
6
<class 'str'>
78
101
101
114
97
106
ASCII value of X= 35
```

Accessing string characters in python:

We can access string characters in python by using,

1. Indexing
2. Slicing

Indexing:

Indexing means a position of string's characters where it stores. We need to use square brackets [] to access the string index. String indexing result is string type. String indices should be integer otherwise we will get an error. We can access the index within the index range otherwise we will get an error.
Python supports two types of indexing

1. **Positive indexing:** The position of string characters can be a positive index from left to right direction (we can say forward direction). In this way, the starting position is 0 (zero).

2. **Negative indexing:** The position of string characters can be negative indexes from right to left direction (we can say backward direction). In this way, the starting position is -1 (minus one).

Slicing:--

A substring of a string is called a slice. A slice can represent a part of a string from a string or a piece of string. The string slicing result is string type. We need to use square brackets [] in slicing. In slicing, we will not get any Index out-of-range exception. In slicing indices should be integer or None or __index__ method otherwise we will get errors.

Different Cases:

wish = “Hello World”

1. wish [::] => accessing from 0th to last
2. wish [:] => accessing from 0th to last
3. wish [0:9:1] => accessing string from 0th to 8th means (9-1) element.
4. wish [0:9:2] => accessing string from 0th to 8th means (9-1) element.
5. wish [2:4:1] => accessing from 2nd to 3rd characters.
6. wish [:: 2] => accessing entire in steps of 2
7. wish [2 ::] => accessing from str[2] to ending
8. wish [:4:] => accessing from 0th to 3 in steps of 1
9. wish [-4: -1] => access -4 to -1

Note: If you are not specifying the beginning index, then it will consider the beginning of the string. If you are not specifying the end index, then it will consider the end of the string. The default value for step is 1

```
wish = "Hello World"
print(wish[::])
print(wish[:])
print(wish[0:9:1])
print(wish[0:9:2])
print(wish[2:4:1])
print(wish[::2])
print(wish[2::])
print(wish[:4:])
print(wish[-4:-1])

O/P:--
Hello World
Hello World
Hello Wor
HloWr
ll
HloWrd
```

```
llo World
Hell
orl
```

Strings are immutable in Python:

Once we create an object then the state of the existing object cannot be changed or modified. This behavior is called immutability. Once we create an object then the state of the existing object can be changed or modified. This behavior is called mutability. A string having immutable nature. Once we create a string object then we cannot change or modify the existing object.

```
name = "Python"
print(name)
print(name[0])
name[0]="X"
O/P:-

Python
P
Traceback (most recent call last):
  File "e:\DataSciencePythonBatch\string.py", line 15, in <module>
    name[0]="X"
TypeError: 'str' object does not support item assignment
```

Mathematical operators on string objects in Python

We can perform two mathematical operators on a string. Those operators are:

1. Addition (+) operator.
2. Multiplication (*) operator.

Addition operator on strings in Python:

The + operator works like concatenation or joins the strings. While using the + operator on the string then compulsory both arguments should be string type, otherwise, we will get an error.

```
a = "Python"
b = "Programming"
```

```
print(a+b)

O/P:--

PythonProgramming
```

```
a = "Python"
b = "Programming"
print(a+" "+b)

O/P:--

Python Programming
```

Multiplication operator on Strings in Python:

This is used for string repetition. While using the * operator on the string then the compulsory one argument should be a string and other arguments should be int type.

```
a = "Python"
b = 3
print(a*b)

O/P:--
PythonPythonPython
```

Length of a string in Python:--

We can find the length of the string by using the len() function. By using the len() function we can find groups of characters present in a string. The len() function returns int as result.

```
course = "Python"
print("Length of string is:",len(course))

O/P:--
Length of string is: 6
```

Membership Operators in Python:

We can check, if a string or character is a member/substring of string or not by using the below operators:

1. In
2. not in

in operator:---

in operator returns True, if the string or character found in the main string.

```
print('p' in 'python')
print('z' in 'python')
print('on' in 'python')
print('pa' in 'python')

O/P:--
True
False
True
False
```

```
main=input("Enter main string:")
s=input("Enter substring:")
if s in main:
   print(s, "is found in main string")
else:
   print(s, "is not found in main string")

O/P:--
Enter main string:Neeraj
Enter substring:raj
raj is found in main string
```

Pre-define methods:--

1. **upper()** – This method converts all characters into upper case

```
str1 = 'python programming language'
print('converted to using title():', str1.upper())
```

```
str2 = 'JAVA proGramming laNGuage'
print('converted to using upper():', str2.upper())

str3 = 'WE ARE SOFTWARE DEVELOPER'
print('converted to using upper ():', str3.upper())

O/P:--
converted to using upper(): PYTHON PROGRAMMING LANGUAGE
converted to using upper (): JAVA PROGRAMMING LANGUAGE
converted to using upper (): WE ARE SOFTWARE DEVELOPER
```

2. **lower()** – This method converts all characters into lower case

```
str1 = 'python programming language'
print('converted to using lover():', str1.lower())

str2 = 'JAVA proGramming laNGuage'
print('converted to using lover ():', str2.lower())

str3 = 'WE ARE SOFTWARE DEVELOPER'
print('converted to using lover ():', str3.lower())

O/P:--
converted to using lover(): python programming language
converted to using lover (): java programming language
converted to using lover (): we are software developer
```

3. **swapcase()** – This method converts all lower-case characters to uppercase and all upper-case characters to lowercase

```
str1 = 'python programming language'
print('converted to using swapcase():', str1.swapcase())

str2 = 'JAVA proGramming laNGuage'
print('converted to using swapcase ():', str2.swapcase())

str3 = 'WE ARE SOFTWARE DEVELOPER'
print('converted to using swapcase ():', str3.swapcase())

O/P:--
converted to using title(): PYTHON PROGRAMMING LANGUAGE
converted to using title(): java PROgRAMMING LAngUAGE
converted to using title(): we are software developer
```

4. **title()** – This method converts all character to title case (The first character in every word will be in upper case and all remaining characters will be in lower case)

```
str1 = 'python programming language'
print('converted to using title():', str1.title())

str2 = 'JAVA proGramming laNGuage'.title()
print('converted to using title():', str2.title())

str3 = 'WE ARE SOFTWARE DEVELOPER'.title()
print('converted to using title():', str3.title())

O/P:--
converted to using title(): Python Programming Language
converted to using title(): Java Programming Language
converted to using title(): We Are Software Developer
```

5. **capitalize()** – Only the first character will be converted to upper case and all remaining characters can be converted to lowercase.

```
str1 = 'python programming language'
print('converted to using capitalize():', str1.capitalize())

str2 = 'JAVA proGramming laNGuage'
print('converted to using capitalize ():', str2.capitalize())

str3 = 'WE ARE SOFTWARE DEVELOPER'
print('converted to using capitalize ():', str3.capitalize())

O/P:--
converted to using capitalize(): Python programming language
converted to using capitalize (): Java programming language
converted to using capitalize (): We are software developer
```

6. **center():-**Python String center() Method tries to keep the new string length equal to the given length value and fills the extra characters using the default character (space in this case).

```
str = "python programming language"

new_str = str.center(40)
# here fillchar not provided so takes space by default.
print("After padding String is: ", new_str)

O/P:--
       python programming language      .
```

```
str = "python programming language"

new_str = str.center(40,'#')
# here fillchar not provided so takes space by default.
print("After padding String is: ", new_str)

O/P:--
######python programming language#######
```

```
str = "python programming language"
new_str = str.center(15,'#')
# here fillchar not provided so takes space by default.
print("After padding String is: ", new_str)

O/P:--
python programming language
```

7. **count():--** count() function is an inbuilt function in Python programming language that returns the number of occurrences of a substring in the given string.

Syntax: string. Count(substring, start= …., end= ….)
Parameters:

The count() function has one compulsory and two optional parameters.
Mandatory parameter:
substring – string whose count is to be found.
Optional Parameters:
start (Optional) – starting index within the string where the search starts.
end (Optional) – ending index within the string where the search ends.

```
str = "python programming language"
count = str.count('o')
# here fillchar not provided so takes space by default.
print("count of given charactor is: ", count)

O/P:--
ount of given charactor is:  2
```

```
str = "python programming language"
count = str.count('o',5,9)
# here fillchar not provided so takes space by default.
print("count of given charactor is: ", count)

O/P:--
count of given charactor is:  0
```

8. **Join():---** The string join() method returns a string by joining all the elements of an iterable (list, string, tuple), separated by the given separator.

The join() method takes an iterable (objects capable of returning its members one at a time) as its parameter.Some of the example of iterables are: Native data types - List, Tuple, String, Dictionary and Set.

```
str = ['Python', 'is', 'a', 'programming', 'language']
# join elements of text with space
print(' '.join(str))

O/P:--
Python is a programming language
```

```
str = ['Python', 'is', 'a', 'programming', 'language']
# join elements of text with space
print('_'.join(str))

O/P:-
Python_is_a_programming_language
```

```
# .join() with lists
numList = ['1', '2', '3', '4']
separator = ', '
print(separator.join(numList))

O/P:--
1, 2, 3, 4
# .join() with tuples
numTuple = ('1', '2', '3', '4')
print(separator.join(numTuple))
O/P:--
1, 2, 3, 4
s1 = 'abc'
s2 = '123'
# each element of s2 is separated by s1
# '1'+ 'abc'+ '2'+ 'abc'+ '3'
print('s1.join(s2):', s1.join(s2))

O/P:--
s1.join(s2): 1abc2abc3

# each element of s1 is separated by s2
# 'a'+ '123'+ 'b'+ '123'+ 'b'
print('s2.join(s1):', s2.join(s1))

O/P:--
s2.join(s1): a123b123c
```

```
# .join() with sets
test = {'2', '1', '3'}
s = ', '
```

```
print(s.join(test))

O/P:--
2, 3, 1

test = {'Python', 'Java', 'Ruby'}
s = '->->'
print(s.join(test))

O/P:--
Ruby->->Java->->Python
```

```
# .join() with dictionaries
test = {'mat': 1, 'that': 2}
s = '->'

# joins the keys only
print(s.join(test))

O/P:--
mat->that
```

9. **split():--** The split() method splits a string at the specified separator and returns a list of substrings.

```
str = "Python is a programming language"
print(str.split(" "))
str = "Python is a programming language"
print(str.split(",",2))
print(str.split(":",4))
print(str.split(" ",1))
print(str.split(" ",0))

O/P:--
['Python', 'is', 'a', 'programming', 'language']
['Python is a programming language']
['Python is a programming language']
['Python', 'is a programming language']
['Python is a programming language']
```

---: List :---

Whenever we want to create a group of objects where we want below mention properties, then we are using list sequence.

1. Duplicates are allowed.
2. Order is preserved.
3. Objects are mutable.
4. Indexing are allowed.
5. Slicing are allowed.
6. Represented in square bracket with comma separated objects.
7. Homogeneous and Heterogeneous both objects are allowed.

1. Duplicates are allowed.

```
List=['neeraj', 10,20,30,10,20]
print(List)

O/P:--
['neeraj', 10, 20, 30, 10, 20]
```

2. Order is preserved:

```
List=['neeraj', 10,20,30,10,20]
x=0
for i in List:
    print('List[{}] = '.format(x),i)
    x=x+1

O/P:--
List[0] =  neeraj
List[1] =  10
List[2] =  20
List[3] =  30
List[4] =  10
List[5] =  20
```

3. Objects are mutable.

```
List=['neeraj', 10,20,30,10,20]
x=0
for i in List:
    print('List[{}] = '.format(x),i)
    x=x+1
List[0]="Arvind"
print(List)

O/P:--
List[0] =  neeraj
List[1] =  10
List[2] =  20
List[3] =  30
List[4] =  10
List[5] =  20
['Arvind', 10, 20, 30, 10, 20]
```

4. Indexing are allowed.

```
List=['neeraj', 10,20,30,10,20]
print(List[0])
print(List[1])
print(List[2])
print(List[3])
print(List[4])
print(List[5])

O/P:--
neeraj
10
20
30
10
20
```

5. Slicing are allowed:

```
List=['neeraj', 10,20,30,10,20]
print(List[:5])

O/P:--
['neeraj', 10, 20, 30, 10]
```

```
List=['neeraj', 10,20,30,10,20]
print(List[::-1])

O/P:--
[20, 10, 30, 20, 10, 'neeraj']
```

Inbuilt functions in list:

1.len(list)
2.max(list) - homogeneous collection required
3.min(list) - homogeneous collection required
4.sum(list) - integer homogeneous collection required
5.list(tuple)
6.type(list)
7.id()
8.list()

Methos:--

1. **list.append(obj/list/str)-** add object in last

```
animals = ['cat', 'dog', 'rabbit']
# Add 'rat' to the list
animals.append('rat')
print('Updated animals list: ', animals)

O/P:--
Updated animals list:  ['cat', 'dog', 'rabbit', 'rat']

animals = ['cat', 'dog', 'rabbit']
wild_animals = ['tiger', 'fox']
animals.append(wild_animals)
print('Updated animals list: ', animals)

O/P:--
Updated animals list:  ['cat', 'dog', 'rabbit', ['tiger', 'fox']]
```

2. **list.count(obj)** – count how many times given-object are present in list

```
numbers = [2, 3, 5, 2, 11, 2, 7]
count = numbers.count(2)
```

```
print('Count of 2:', count)

O/P:--
Count of 2: 3

# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
count = vowels.count('i')
print('The count of i is:', count)
count = vowels.count('p')
print('The count of p is:', count)

O/P:--
The count of i is: 2
The count of p is: 0
# random list
random = ['a', ('a', 'b'), ('a', 'b'), [3, 4]]
count = random.count(('a', 'b'))
print("The count of ('a', 'b') is:", count)
count = random.count([3, 4])
print("The count of [3, 4] is:", count)

O/P:--
The count of ('a', 'b') is: 2
The count of [3, 4] is: 1
```

3. **list.extend(list1)** – add list1 in last of list.

```
# create a list
list1 = [2, 3, 5]
list2 = [1, 4]
list1.extend(list2)
print('List after extend():', list1)

O/P:--
List after extend(): [2, 3, 5, 1, 4]



list = ['Hindi']
tuple = ('Spanish', 'English')
```

```
set = {'Chinese', 'Japanese'}
list.extend(tuple)
print('New Language List:', list)
list.extend(set)
print('Newer Languages List:', list)

O/P:--

New Language List: ['Hindi', 'Spanish', 'English']
Newer Languages List: ['Hindi', 'Spanish', 'English', 'Japanese', 'Chinese']
```

4. **list.insert(index,obj)** – insert given object in given index.
5. **list.pop()** – delete bydefault last object from given list.
6. **list.remove(obj)** – Remove given object from given list.
7. **list.reverse() –**

```
Example:---
numbers = ['Neeraj',2, 3, 5, 7]
numbers.reverse()
print('Reversed List:', numbers)

O/P:--
Reversed List: [7, 5, 3, 2, 'Neeraj']

Example:----
numbers = ['Neeraj',2, 3, 5, 7]
print(numbers[::-1])

O/P:--
[7, 5, 3, 2, 'Neeraj']

Example:----
numbers = ['Neeraj',2, 3, 5, 7]
# print(numbers[::-1])
list=[]
for i in reversed(numbers):
   list.append(i)
print(list)

O/P:--
```

```
[7, 5, 3, 2, 'Neeraj']
```

8. **list.sort**(reverse=True/False) default-False

```
Example:---
numbers = [2, 3, 7, 5, 4]
numbers.sort()
print('Sort_List:', numbers)

O/P:--
Sort_List: [2, 3, 4, 5, 7]
Example:---
numbers = [2, 3, 7, 5, 4]
numbers.sort(reverse=True)
print('Sort_List:', numbers)

O/P:--
Sort_List: [7, 5, 4, 3, 2]
```

---:Tuple :---

In Python, tuples are immutables. Meaning, you cannot change items of a tuple once it is assigned. There are only two tuple methods count() and index() that a tuple object can call.

1. **Duplicates are allowed.**
2. **Order is preserved.**
3. **Objects are immutable.**
4. **Indexing is allowed.**
5. **Slicing is allowed.**
6. **Represented in parenthesis () with comma separated objects.**
7. **Homogeneous and Heterogeneous both objects are allowed.**

Tuple occupies less memory as compare to list, that's why tuple is more faster as compare to list**.**

Example:--

```
list = [10,20,30,40,50,60,70]
tuple = (10,20,30,40,50,60,70)
print(sys.getsizeof('Size of list = ',list))
print(sys.getsizeof('Size of tuple',tuple))

O/P-   64
       62
```

Built-in functions:-

1. **Len(tuple)   # tuple variable must be a iterable.**
2. **Max(tuple)**
3. **Min(tuple)**
4. **Sum(tuple)**
5. **Tuple(list)**
6. **Type(tuple)**

Methods:--

1. **Count(obj). (How many occurrences)**

```
# Creating tuples
Tuple = (0, 1, (2, 3), (2, 3), 1, [3, 2],'Neeraj', (0), (0,))
res = Tuple.count((2, 3))
print('Count of (2, 3) in Tuple is:', res)

res = Tuple.count(0)
print('Count of 0 in Tuple is:', res)

res = Tuple.count((0,))
print('Count of (0,) in Tuple is:', res)

O/P:--
Count of (2, 3) in Tuple is: 2
Count of 0 in Tuple is: 2
Count of (0,) in Tuple is: 1
```

2. **Index(obj,start,stop)(obj is compulsory argument but rest are optional)**

```
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)
# getting the index of 3
res = Tuple.index(3)
print(res)
O/P:--
3
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)
# getting the index of 3
print(Tuple.index(3,4))
O/P:--
5
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)
# getting the index of 3
print(Tuple.index(3,0,4))
o/p:--
3
```

---: Dictionary :---

If we want to represent a group of objects as key-value pairs then we should go for dictionaries.

Characteristics of Dictionary

1. Dictionary will contain data in the form of key, value pairs.
2. Key and values are separated by a colon ":" symbol
3. One key-value pair can be represented as an item.
4. Duplicate keys are not allowed.
5. Duplicate values can be allowed.
6. Heterogeneous objects are allowed for both keys and values.
7. Insertion order is not preserved.
8. Dictionary object having mutable nature.
9. Dictionary objects are dynamic.
10. Indexing and slicing concepts are not applicable

syntax for creating dictionaries with key,value pairs is: **d = { key1:value1, key2:value2, …., keyN:valueN }**
Creating an Empty dictionary in Python:

```
d = {}
print(d)
print(type(d))

O/P:--
{}
<class 'dict'>
```

Adding the items in empty dictionary:--

```
d = {}
d[1] = "Neeraj"
d[2] = "Rahul"
d[3] = "Ravi"
print(d)

O/P:--
{1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
```

Accessing dictionary values by using keys:--

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}

print(d[1])
print(d[2])
print(d[3])

O/P:--
Neeraj
Rahul
Ravi
```

Note:--- While accessing, if the specified key is not available then we will get **KeyError**

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}

print(d[1])
print(d[2])
print(d[3])
print(d[10])

O/P:--
Neeraj
Rahul
Ravi
Traceback (most recent call last):
  File "E:\DataSciencePythonBatch\dict.py", line 16, in <module>
    print(d[10])
KeyError: 10
```

handle this KeyError by using in operator:

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}

if 10 in d:
    print(d[10])
else:
    print('Key Not found')

O/P:--
Key Not found
```

Getting student information's in the form of dictionaries:--

```
d={}
n=int(input("Enter how many student detail you want: "))
i=1
while i <=n:
   name=input("Enter Employee Name: ")
   email=input("Enter Employee salary: ")
   d[name]=email
   i=i+1
print(d)

O/P:--
Enter how many student detail you want: 3
Enter Employee Name: Neeraj
Enter Employee salary: neeraj@gmail.com
Enter Employee Name: Rahul
Enter Employee salary: rahul@gmail.com
Enter Employee Name: Ravi
Enter Employee salary: ravi@gmail.com
{'Neeraj': 'neeraj@gmail.com', 'Rahul': 'rahul@gmail.com', 'Ravi':
'ravi@gmail.com'}
```

Updating dictionary elements:
We can update the value for a particular key in a dictionary. The syntax is:

d[key] = value

Case1: While updating the key in the dictionary, if the key is not available then a new key will be added at the end of the dictionary with the specified value.

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print("Old dict data",d)
```

```
d[10]="Arvind"
print("Nwe dict data",d)

O/P:--
Old dict data {1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
Nwe dict data {1: 'Neeraj', 2: 'Rahul', 3: 'Ravi', 10: 'Arvind'}
```

Case2: If the key already exists in the dictionary, then the old value will be replaced with a new value.

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print("Old dict data",d)
d[2]="Arvind"
print("New dict data",d)

O/P:--
Old dict data {1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
New dict data {1: 'Neeraj', 2: 'Arvind', 3: 'Ravi'}
```

Removing or deleting elements from the dictionary:

1. By using the del keyword, we can remove the keys
2. By using clear() we can clear the objects in the dictionary

By using the del keyword

Syntax: del d[key]

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
del d[3]
print("New dict is",d)

O/P:--
New dict is {1: 'Neeraj', 2: 'Rahul'}
```

By using clear() keyword

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
d.clear()
print("New dict is",d)
```

```
O/P:--
New dict is {}
```

Delete entire dictionary object:- We can also use the del keyword to delete the total dictionary object. Before deleting we just have to note that once it is deleted then we cannot access the dictionary.

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
del d
print("New dict is",d) O/P:--

Traceback (most recent call last):
  File "E:\DataSciencePythonBatch\dict.py", line 51, in <module>
    print("New dict is",d)
NameError: name 'd' is not defined. Did you mean: 'id'?
```

Functions of dictionary in Python

1. dict()
2. len() – total object length
3. max() – on the basis of key
4. min() – on the basis of key
5. id()
6. type()

Methods of dictionary in Python

1. setdefault()   # x.setdefault('name','Neeraj')

```
x = {'age': 25}
x.setdefault('name', 'Neeraj')
print(x)
```

- Since 'name' is not in x, setdefault adds 'name': 'Neeraj' to the dictionary.
- If 'name' were already a key in the dictionary, setdefault would leave it unchanged and just return the existing value.

2. fromkeys() # dict.fromkeys(keys, value) **Initializing multiple keys with the same value.**

keys = ['a', 'b', 'c']
new_dict = dict.fromkeys(keys, 0)
print(new_dict) # **{'a': 0, 'b': 0, 'c': 0}**

4. update() # x.update(collection) **Updating a dictionary with another dictionary**
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict1.update(dict2)
print(dict1)   # **{'a': 1, 'b': 3, 'c': 4}**

5. copy() # x.copy()
6. get() # x.get('key')
7. clear() # x.clear()
8. pop() # x.pop('key')
9. popitem() # x.popitem()
10. key() # x.keys()
11. values() # x.values()
12. items() # x.items()

dict() function:

This can be used to create an empty dictionary.

```
d=dict()
print(d)
print(type(d))

O/P:--
{}
<class 'dict'>
```

len() function: This function returns the number of items in the dictionary.

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print(len(d))

O/P:--
3
```

clear() method: This method can remove all elements from the dictionary.

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print(d.clear())O/P:--
```

```
O/P:--
None
```

get() method:
This method used to get the value associated with the key. This is another way to get the values of the dictionary based on the key. The biggest advantage it gives over the normal way of accessing a dictionary is, this doesn't give any error if the key is not present. Let's see through some examples:

Case1: If the key is available, then it returns the corresponding value otherwise returns None. It won't raise any errors.
Syntax: d.get(key)

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print(d.get(1))
print(d.get(2))
print(d.get(3))

O/P:--
Neeraj
Rahul
Ravi
```

Case 2: If the key is available, then returns the corresponding value otherwise returns the default value that we give.
Syntax: d.get(key, defaultvalue)

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print(d.get(7,"Neeraj"))
print(d.get(6,"Neeraj"))
print(d.get(5,"Neeraj"))

O/P:--
Neeraj
Neeraj
Neeraj
```

pop() method: This method 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.
Syntax: d.pop(key)

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
d.pop(3)
print(d)

O/P:
{1: 'Neeraj', 2: 'Rahul'}
```

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi'}
print(d.pop(3)) O/P:-- Ravi
```

popitem() method: This method removes an arbitrary item(key-value) from the dictionary and returns it.

```
d={1: 'Neeraj', 2: 'Rahul', 3: 'Ravi',4:'Jai',5:'Santosh'}
print(d.popitem())
print(d)

O/P:--
(5, 'Santosh')
{1: 'Neeraj', 2: 'Rahul', 3: 'Ravi', 4: 'Jai'}
```

keys() method: This method returns all keys associated with the dictionary

```
d = {1: 'Ramesh', 2: 'Suresh', 3: 'Mahesh'}
print(d)
for k in d.keys():
  print(k)

O/P:--
1
2
3
```

values() method: This method returns all values associated with the dictionary

```
d = {1: 'Ramesh', 2: 'Suresh', 3: 'Mahesh'}
print(d)
for k in d.values():
   print(k)


O/P:--
Ramesh
Suresh
Mahesh
```

items() method: A key-value pair in a dictionary is called an item. items() method returns the list of tuples representing key-value pairs.

```
d = {1: 'Ramesh', 2: 'Suresh', 3: 'Mahesh'}
for k, v in d.items():
   print(k, "---", v)

O/P:--
1 --- Ramesh
2 --- Suresh
3 --- Mahesh
```

---: Set :---

If we want to represent a group of unique elements then we can go for sets. Set cannot store duplicate elements.

1. **Duplicates are not allowed.**
2. **Order is not preserved.**
3. **Objects are mutable.**
4. **Indexing is not allowed.**
5. **Slicing is not allowed.**
6. **Represented in { } with comma separated objects.**
7. **Homogeneous and Heterogeneous both objects are allowed.**

```
# Creating a set
s = {10,20,30,40}
print(s)
print(type(s))

O/P:--
{40, 10, 20, 30}
<class 'set'>

# Creating a set with different elements
s = {10,'20','Rahul', 234.56, True}
print(s)
print(type(s))

O/P:--
{'20', True, 234.56, 10, 'Rahul'}
<class 'set'>

# Creating a set using range function
s=set(range(5))
print(s)

O/P:--
{0, 1, 2, 3, 4}
```

```
# Duplicates not allowed
s = {10, 20, 30, 40, 10, 10}
print(s)
print(type(s))
O/P:--
{40, 10, 20, 30}
<class 'set'>

# Creating an empty set
s=set()
print(s)
print(type(s))

O/P:--
set()
<class 'set'>
```

# Methods in set:----

1. **add(only_one_argument not iterable)**

```
s={10,20,30,50}
s.add(40)
print(s)

O/P:--
{40, 10, 50, 20, 30}
```

2. **update(iterable_obj1,iterable_obj2)**

```
s = {10,20,30}
l = [40,50,60,10]
s.update(l)
print(s)

O/P:--
{40, 10, 50, 20, 60, 30}



s = {10,20,30}
```

```
l = [40,50,60,10]
s.update(l, range(5))
print(s)

O/P:--
{0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}
```

Difference between add() and update() methods in set:

3. We can use add() to add individual items to the set, whereas we can use update() method to add multiple items to the set.
4. The add() method can take one argument whereas the update() method can take any number of arguments but the only point is all of them should be iterable objects.

3. **copy() --Clone of set**

```
s={10,20,30}
s1=s.copy()
print(s1)

O/P:--
{10, 20, 30}
```

4. **pop()---** This method removes and returns some random element from the set.

```
s = {40,10,30,20}
print(s)
print(s.pop())
print(s)

O/P:--
{40, 10, 20, 30}
40
{10, 20, 30}
```

5. **remove(element) ---** This method removes specific elements from the set. If the specified element is not present in the set then we will get KeyError.

```
s={40,10,30,20}
s.remove(30)
```

```
print(s)

O/P:--
{40, 10, 20}

s={40,10,30,20}
s.remove(50)
print(s)

O/P:--
Traceback (most recent call last):
  File "E:\DataSciencePythonBatch\sets.py", line 65, in <module>
    s.remove(50)
KeyError: 50
```

6. **discard(element) ---** This method removes the specified element from the set. If the specified element is not present in the set, then we won't get any error.

```
s={10,20,30}
s.discard(10)
print(s)

O/P:--
{20, 30}

s={10,20,30}
s.discard(40)
print(s)

O/P:--
{10, 20, 30}
```

7. **clear() --- removes all elements from the set.**

```
s={10,20,30}
print(s)
s.clear()
print(s)
```

```
O/P:--
{10, 20, 30}
set()
```

MATHEMATICAL OPERATIONS ON SETS

1. **union() ---** This method return all elements present in both sets.

```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y))

O/P:--
{40, 10, 50, 20, 60, 30}
```

2. **intersection() ---** This method returns common elements present in both x and y.

```
x = {10,20,30,40}
y = {30,40,50,60}
print(x.intersection(y))
print(x&y)
print(y.intersection(x))
print(y&x)

O/P:--
{40, 30}
{40, 30}
{40, 30}
{40, 30}
```

3. **difference() ---** This method returns the elements present in x but not in y

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.difference(y)
print(z)
```

O/P:--- {‘banana’, ‘cherry’}

Control Flow Statements

In programming languages, flow control means the order in which the statements or instructions, that we write, get executed. In order to understand a program, we should be aware of what statements are being executed and in which order. So, understanding the flow of the program is very important. There are, generally, three ways in which the statements will be executed. They are,

1. **Sequential**
2. **Conditional**
3. **Looping**

Sequential: In this type of execution flow, the statements are executed one after the other sequentially. By using sequential statements, we can develop simple programs

Example: Sequential statements:-

```
print("Welcome")
print("to")
print("python class")

O/P:-
Welcome
to
python class
```

Conditional: Statements are executed based on the condition. As shown above in the flow graph, if the condition is true then one set of statements are executed, and if false then the other set. Conditional statements are used much in complex programs.

Conditional statements are also called decision-making statements. Let's discuss some conditions making statements in detail. There are three types of conditional statements in python. They are as follows:

1. **if statement**
2. **if-else statement**
3. **nested-if (if-elif-elif-else)**

if-statement:-
syntax:-

```
if condition:
    print("Block statement")
print("Out of block statement")
```

Example:-

```

num=int(input("Enter any no: "))
if num>=18:
    print("if block statment executed")
print("out of if block statements")

O/P:
Enter any no: 10
out of if block statements
PS E:\DataSciencePythonBatch> python control.py
Enter any no: 18
if block statment executed
out of if block statements
```

Example:---

```
# Example:---Checking if a number is positive, negative, or zero.

num = float(input("Enter a number: "))
if num > 0:
    print("The number is positive.")
elif num < 0:

    print("The number is negative.")
else:
    print("The number is zero.")
```

if-else condition:-
syntax:-

```
if condition:
    print("if block statement executed")
else:
    print("else block statement executed ")
```

Example:---

```
num=int(input("Enter any no: "))
if num>=18:
    print("if block statment executed")
else:
    print("else block statement executed")

O/P:-
PS E:\DataSciencePythonBatch> python control.py
Enter any no: 18
if block statment executed
PS E:\DataSciencePythonBatch> python control.py
Enter any no: 15
else block statement executed
```

Example:---

```
# Example:---Find grater no.

x=int(input("Enter first no."))
y=int(input("Enter second no."))
z=int(input("Enter third no."))
if x>y:
    if x>z:
        print("Greater ni is(x): ",x)
    else:
        print("Greater no is(z): ",z)
else:
    if y>z:
        print("Greater ni is(y): ",y)
    else:
        print("Greater no is(z): ",z)

O/P:-

Enter first no.10
Enter second no.10
Enter third no.20
Greater no is(z):  20
```

Example:---

```
# Example:-- Checking if a person is eligible to vote

age = int(input("Enter your age: "))
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")

O/P:-
Enter your age: 35
You are eligible to vote.
```

Example:---

```
# Example:-- Checking if a year is a leap year
year = int(input("Enter a year: "))
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0 and year %100==0):
    print("It's a leap year.")
else:
    print("It's not a leap year.")

O/P:-
Enter a year: 2000
It's a leap year.
```

Nested-If else:--

```
# Example:-- Check your gread based on your own score

score = int(input("Enter your score: "))

if score >= 90:
    print("You got an A.")
else:
    if score >= 80:
        print("You got a B.")
    else:
        if score >= 70:
            print("You got a C.")
        else:
            if score >= 60:
                print("You got a D.")
            else:
                print("You got an F.")

O/P:-
Enter your score: 90
You got an A.
```

Example:--

```
# Example:-- Check given year is leep year or not.

year = int(input("Enter a year: "))
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print("Leap year")
        else:
            print("Not a leap year")
    else:
        print("Leap year")
else:
    print("Not a leap year")
```

if elif else statement in python:
Syntex:

```
if (condition1):
    statement of if Block
elif(condition2):
    statment of elif Block
elif(condition3):
    statement if elif block
else:
    ststement of else block
```

Example:---

```
# example:-- Please choose value within range of o to 4.

print("Please enter the values from 0 to 4")
x=int(input("Enter a number: "))
if x==0:
      print("You entered:", x)
elif x==1:
      print("You entered:", x)
elif x==2:
      print("You entered:", x)
elif x==3:
      print("You entered:", x)
elif x==4:
      print("You entered:", x)
else:
      print("Beyond the range than specified")

O/P:-
Enter a number: 5
Beyond the range than specified
PS E:\DataSciencePythonBatch> python control.py
```

```
Please enter the values from 0 to 4
Enter a number: 4
You entered: 4
```

Example:- Python Program to calculate the square root.

```
# Example:-Python

num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of Num :', num_sqrt)

O/P:-
Enter a number: 4
The square root of 4.000 is 2.000
PS E:\DataSciencePythonBatch> python control.py
Enter a number: 8
The square root of 8.000 is 2.828
```

Example:-- Python Program to find the area of triangle.

```
# Python Program to find the area of triangle
# s = (a+b+c)/2
# area = √(s(s-a)*(s-b)*(s-c))
a = float(input('Enter first side: '))
b = float(input('Enter second side: '))
c = float(input('Enter third side: '))
s = (a + b + c) / 2
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is :', area)

O/P:---
Enter first side: 5
Enter second side: 6
Enter third side: 7
The area of the triangle is : 14.696938456699069
```

Example:-- Python program to swap two variables.

```
# Python program to swap two variables
x = input('Enter value of x: ')
y = input('Enter value of y: ')

# create a temporary variable and swap the values
temp = x
x = y
y = temp
print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))

O/P:---
Enter value of x: 5
Enter value of y: 8
The value of x after swapping: 8
The value of y after swapping: 5


# without using third variable
x = input('Enter value of x: ')
y = input('Enter value of y: ')
x, y = y, x
print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))

O/P:---
Enter value of x: 4
Enter value of y: 6
The value of x after swapping: 6
The value of y after swapping: 4

# By-using Addition and Subtraction.
x = int(input('Enter value of x: '))
y = int(input('Enter value of y: '))
x = x + y
y = x - y
x = x - y
print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))

O/P:---
Enter value of x: 4
Enter value of y: 6
The value of x after swapping: 6
The value of y after swapping: 4
# By-using Multiplication and division.
x = int(input('Enter value of x: '))
y = int(input('Enter value of y: '))
x = x * y
y = x / y
x = x / y
print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))

O/P:---
Enter value of x: 2
Enter value of y: 5
The value of x after swapping: 5.0
The value of y after swapping: 2.0


# By-using x-or(^) operator.
x = int(input('Enter value of x: '))
y = int(input('Enter value of y: '))
x = x ^ y
y = x ^ y
x = x ^ y
print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))

O/P:--
Enter value of x: 10
Enter value of y: 20
The value of x after swapping: 20
The value of y after swapping: 10
```

Range()

Range() function is used to generate collection in python.

Syntax:

range(start,stop/end,step/direction)

Note :-

1. **for** +ve direction collection step must be +ve.
2. **for** -ve direction collection step must be –ve.
3. for +ve direction collection stop/end point must be (required+1).
4. for -ve direction collection stop/end point must be (required-1).
5. Start point is always what we require.

```
my_range = range(1,11)
print(list(my_range))

my_range = range(1,11,-1)
print(list(my_range))

my_range = range(-1,-11,-1)
print(list(my_range))

my_range = range(-1,-11,1)
print(list(my_range))

my_range = range(11)
print(list(my_range))


O/P:--
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[]
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
[]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
my_range = range(2,11,2)
print(list(my_range))

my_range = range(1,10,2)
print(list(my_range))

my_range = range(-2,-11,-2)
print(list(my_range))

my_range = range(-1,-10,-2)
print(list(my_range))

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

```
my_range = range(5,2,-1)
print(list(my_range))

my_range = range(-5,-2,1)
print(list(my_range))

my_range = range(5,6,1)
print(list(my_range))

my_range = range(-5,-6,-1)
print(list(my_range))

O/P:--
[5, 4, 3]

[-5, -4, -3]

[5]

[-5]
```

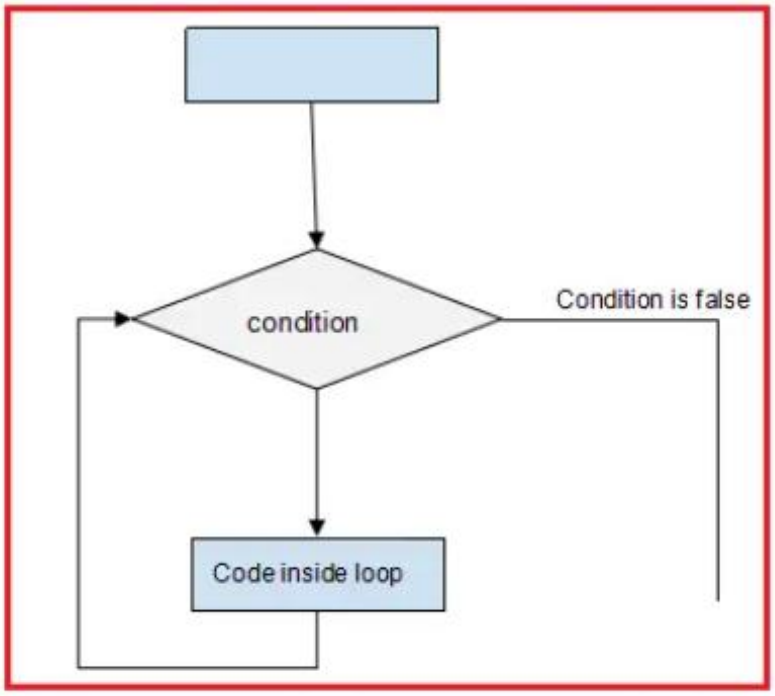
----:LOOPING Statement in Python (Iterations):----

If we want to execute a group of statements multiple times, then we should go for a looping kind of execution. There are two types of loops available in python.
They are:

1. **while loop**
2. **for loop**

1. while loop:- The while loop contains an expression/condition. As per the syntax colon (:) is mandatory otherwise it throws a syntax error. The condition gives the result as bool type, either True or False. The loop keeps on executing the statements until the condition becomes False. i.e. With the while loop we can execute a set of statements as long as a condition is true.

Flowchart for the while-loop:



Parts of while loop in Python:
Initialization:

This is the first part of the while loop. Before entering the condition section, some initialization is required.

Condition:
Once the initializations are done, then it will go for condition checking which is the heart of the while loop. The condition is checked and if it returns True, then execution enters the loop for executing the statements inside.

After executing the statements, the execution goes to the increment/decrement section to increment the iterator. Mostly, the condition will be based on this iterator value, which keeps on changing for each iteration. This completes the first iteration. In the second iteration, if the condition is False, then the execution comes out of the loop else it will proceed as explained in the above point.

Increment/Decrement section: This is the section where the iterator is increased or decreased. Generally, we use arithmetic operators in the loop for this section.

Example: Printing numbers from 1 to 5 by using while loop

1. The program is to print the number from 1 to 5
2. Before starting the loop, we have made some assignments(x = 1). This is called the Initialization section.
3. After initialization, we started the while loop with a condition x<=5. This condition returns True until x is less than 5.
4. Inside the loop, we are printing the value of x.
5. After printing the x value, we are incrementing it using the operator x+=1. This is called the increment/decrement section.
6. For each iteration, the value of x will increase and when the x value reaches 6, then the condition x<=5 returns False. At this iteration, the execution comes out of the loop without executing the statements inside. Hence in the output '6' is not printed.

```
x=1
while x<=5:
   print(x)
   x+=1
O/P:--
1
2
3
4
5
```

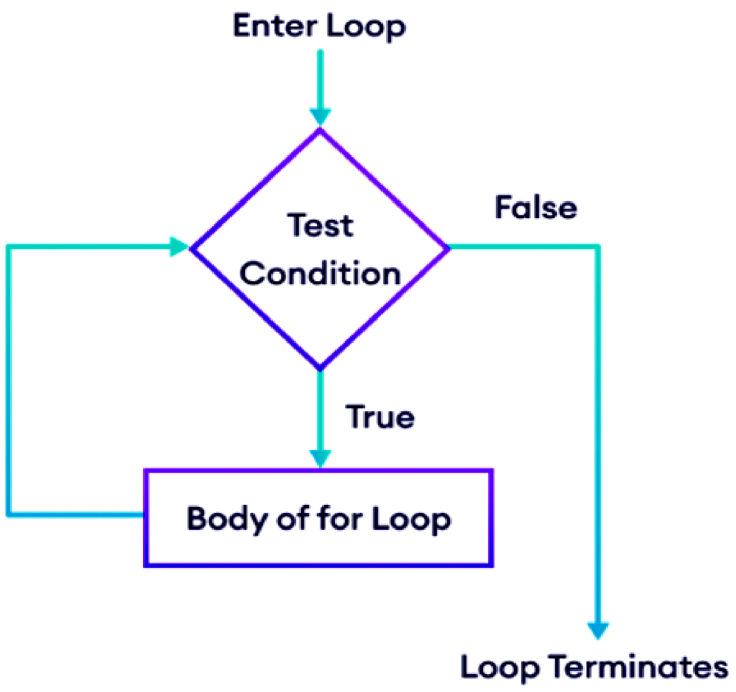
```
# Printing numbers from 1 to 5 by using while loop.
x=1
while x<=5:
    print(x)
    x+=1


# Printing numbers from 1 to 5 by using while loop.
x=1
while x<=5:
    if x<5:
        print(x,end=",")
    else:
        print(x,end="")
    x+=1


# Printing even numbers from 10 to 20 by using while loop.
x=10
while (x>=10) and (x<=20):
   print(x)
   x+=2

print("End")


# print sun of given n netural no
x=int(input("Enter any no : "))
sum=0
i=1
while i<=x:
    sum=sum+i
    if i<x:
        print(i,end="+")
    else:
        print(i,end="=")
    i=i+1

print(sum)
```

```
# print n even numbers
x= int(input("Enter how many even number you want :"))
n=1
while n<=x:
    print(2*n)
    n=n+1

# print n even numbers(1,2,3,4,5,6,--------)
x= int(input("Enter how many even numbers you want :"))
n=1
while n<=x:
    if n<x:
        print(2*n,end=",")
    else:
        print(2*n,end="")
    n=n+1

# Print sum of given even numbers.
x= int(input("Enter how many even numbers sum you want :"))
n=1
sum=0
while n<=x:
    sum=sum+2*n
    if n<x:
        print(2*n,end="+")
    else:
        print(2*n,end="=")
    n=n+1
print(sum)

# Print n odd numbers
x= int(input("Enter how many odd number you want :"))
n=1
while n<=x:
    if n<x:
        print((2*n-1),end=",")
    else:
        print((2*n-1),end="")
    n=n+1
```

```python
# Print sum of n odd numbers
x= int(input("Enter how many odd number you want :"))
n=1
sum = 0
while n<=x:
    sum=sum+(2*n-1)
    if n<x:
        print((2*n-1),end="+")
    else:
        print((2*n-1),end="=")
    n=n+1
print(sum)

# WAP to print even no upto n netural no.
n=int(input("Enter no : "))
i=2
while i<=n:
    if i<=n-2:
        print(i,end=",")
    else:
        print(i)
    i+=2

# WAP to print odd no upto n netural no.
n=int(input("Enter no : "))
i=1
while i<=n:
    if i<=n-2:
        print(i,end=",")
    else:
        print(i)
    i+=2
```

<u>**for-loop**</u>

Basically, a for loop is used to iterate elements one by one from sequences like string, list, tuple, etc. This loop can be easily understood when compared to the while loop. While iterating elements from the sequence we can perform operations on every element.
The Python For Loop is used to repeat a block of statements until there are no items in the Object may be String, List, Tuple, or any other object.



1. Initialization: We initialize the variable(s) here. Example i=1.
2. Items in Sequence / Object: It will check the items in Objects. For example, individual letters in String word. If there are items in sequence (True), then it will execute the statements within it or inside. If no item is in sequence (False), it will exit.
3. After completing the current iteration, the controller will traverse to the next item.
4. Again it will check the new items in sequence. The statements inside it will be executed as long as the items are in sequence.

```python
# Upto n natural no.
n = int(input("Enter a number: "))
for i in range(1, n + 1):
    if i < n:
        print(i, end=",")  # Print numbers with commas
    else:
        print(i)


# for upto n even natural no.
n=int(input("enter a no"))
for i in range(2,n+1,2):
    if(n%2==0):
        if i<n:
         print(i,end=',')
        else:
          print(i)
    else:
        if(i<n-1):
            print(i,end=',')
        else:
            print(i)

# for upto n odd natural no.
n=int(input("enter a no"))
for i in range(1,n+1,2):
    if(n%2!=0):
        if i<n:
         print(i,end=',')
        else:
          print(i)
    else:
        if(i<n-1):
            print(i,end=',')
        else:
            print(i)


# Odd no upto n natural no.
n = int(input("Enter a number: "))
```

```
for i in range(1, n + 1,2):
    if n%2 !=0:
        if i<n:
            print(i,end=',')
        else:
            print(i)
    else:
        if i<n-2:
            print(i,end=",")
        else:
            print(i)

# Even no upto n natural no.
n = int(input("Enter a number: "))
for i in range(2, n + 1,2):
    if n%2 ==0:
        if i<n:
            print(i,end=',')
        else:
            print(i)
    else:
        if i<n-2:
            print(i,end=",")
        else:
            print(i)


# n natural even no.
n = int(input("Enter a number: "))
for i in range(2, 2*n + 1,2):
    if n%2 ==0:
        if i<2*n:
            print(i,end=',')
        else:
            print(i)
    else:
        if i<2*n-1:
            print(i,end=",")
        else:
            print(i)
```

```
# n natural odd no.
n = int(input("Enter a number: "))
for i in range(1, 2*n + 1,2):
    if n%2 ==0:
        if i<2*n-1:
            print(i,end=',')
        else:
            print(i)
    else:
        if i<2*n-1:
            print(i,end=",")
        else:
            print(i)

# n even natural no.
n=int(input("enter a no"))
for i in range(1,n+1):
    if i<n:
     print(2*i,end=',')
    else:
        print(2*i)

# n odd natural no.
n=int(input("enter a no"))
for i in range(1,n+1):
    if i<n:
     print(2*i-1,end=',')
    else:
        print(2*i-1)
```

Transfer Statements:

1. Break
2. continue
3. pass

Break statement:--- We can use break statement inside loops to break loop execution based on some condition.

```
for i in range(10):
    if i==7:
        print("processing is enough..  plz break !!!!!!! ")
        break
    print(i)

O/P:---
0
1
2
3
4
5
6
processing is enough..  plz break !!!!!!!

list=[10,20,600,60,70]
for i in list:
    if i>500:
        print("no need to check next object of list")
        break
    print(i)

O/P:--
10
20
no need to check next object of list
```

continue statement:-- We can use continue statement to skip current iteration and continue next iteration.

```
for i in range(10):
    if i%2==0:
        continue
    print(i)

O/P:--
1
3
5
7
9

list=[10,20,600,60,70]
for i in list:
    if i>500:
        print("no need to print this object")
        continue
    print(i)

O/P:--
10
20
no need to print this object
60
70

list=[10,20,600,60,70]
for i in list:
    if i>500:
        continue
    print(i)

O/P:--
10
20
60
70
```

pass statement:--
pass is a keyword in Python. In our programming syntactically if block is required which won't do anything then we can define that empty block with pass keyword.

1. It is an empty statement
2. It is null statement
3. It won't do anything

```
for i in range(100):
    if i%9==0:
        print(i)
    else:
        pass

O/P:--
0
9
18
27
36
45
54
63
72
81
90
99
```

----: Some Important Pattern examples :---

```
# O/P:--
# 10
# 10 8
# 10 8 6
# 10 8 6 4
# 10 8 6 4 2
# code for above output
k=[]
for i in range(10,1,-2):
    k.append(i)
    for j in k:
        print(j,end=" ")
    print()


# O/P:-- {1: 2, 2: 3}
# code for above output
t=(1,1,2,2,4,2)
dict={}
for i in t:
    count=0
    for j in t:
        if j==i:
            count=count+1
    if count>=2:
        dict[i]=count
print(dict)

#  O/P --- {1: 2, 2: 3, 4: 1}
# code for above output
t=(1,1,2,2,4,2)
dict={}
for i in t:
    count=0
    for j in t:
        if j==i:
            count=count+1
    dict[i]=count
print(dict)
```

```
# *
# **
# ***
# ****
# *****
n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
    print("*"*i)


# *
# * *
# * * *
# * * * *
# * * * * *
n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
    print("* "*i)


#      *
#     **
#    ***
#   ****
#  *****

n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
    print(" "*(n-i),"*"*i)


#      *
#     ***
#    *****
#   *******
#  *********

n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
    print(" "*(n-i),"*"*(2*i-1))
#       *
#      * *
#     * * *
#    * * * *
#   * * * * *
n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
    print(" "*(n-i)," *"*i)


# 1
# 1 2
# 1 2 3
# 1 2 3 4
# 1 2 3 4 5

n=int(input("Enter the number of rows:"))
for i in range(1,n+1):
    for j in range(1,i+1):
        print(j,end=" ")
    print()


#  *****
#   ****
#    ***
#     **
#      *
n=int(input("Enter the number of rows: "))
for i in range(n,0,-1):
    print(" "*(n-i),"*"*i)

# *****
# ****
# ***
# **
# *
n=int(input("Enter the number of rows: "))
for i in range(n,0,-1):
    print("*"*i)
```

```
#   * * * * *
#    * * * *
#     * * *
#      * *
#       *
n=int(input("Enter the number of rows: "))
for i in range(n,0,-1):
    print(" "*(n-i)," *"*i)



#       *
#      * *
#     * * *
#    * * * *
#   * * * * *
#   * * * * *
#    * * * *
#     * * *
#      * *
#       *
n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
    print(" "*(n-i),"* "*i)
m=n-1
for i in range(m,0,-1):
    print(" "*(m-i)," *"*i)



#      *
#     **
#    ***
#   ****
#  *****
# *****
# ****
# ***
# **
# *
```

```
n=int(input("Enter the number of rows: "))
for i in range(0,n+1):
    print(" "*(n-i),"*"*i)
for i in range(n,0,-1):
    print("*"*i," "*(n-i))


# *
# * *
# * * *
# * * * *
# * * * * *
# * * * *
# * * *
# * *
# *
n=int(input("Enter the number of rows: "))
for i in range(0,n+1):
    print("* "*i)
m=n-1
for i in range(m,0,-1):
    print("* "*i)

#      *
#     **
#    ***
#   ****
#  *****
#  *****
#   ****
#    ***
#     **
#      *
n=int(input("Enter the number of rows: "))
for i in range(0,n+1):
    print(" "*(n-i),"*"*i)
for i in range(n,0,-1):
    print(" "*(n-i),"*"*i)
```

IF-ELIF-ELSE STATEMENT EXAMPLES

Example 1: Write a program to check given no is positive. (Only if-statement)

Example 2: Write a program to check given no is positive or negative. (Only if-else statement)

Example 3: Write a program to check given no is positive, negative or Zero.(Only if-elif-else statement)

Example 4: Write a program to swap two variables without using third variable.

Example 5: Write a program to swap two variables using third variable.

Example 6: Write a program to swap two variables using using Addition and Subtraction.

Example 9: Write a program to find squre root of given no.

Example 10: Write a program to find largest no among the three inputs numbers.

Example 11: Write a program to find area of triangle. (1/2* hight*base)

Example 12: Write a program to find area of square.

Example 13: Write a program to find given year is leep year or not.

While-Loop EXAMPLES

Example 1: Write a program to display n natural numbers. (In Horizontal- 1,2,3,4,5……..)

Example 2: Write a program to calculate the sum of numbers.

Example 3: Write a program to find even no. (2,4,6,8,….)

Example 4: Write a program find odd no.(1,3,5,7,9,……)

Example 5: Write a program to find factorial of given no.

Example 6: Write a program to print your names ten times.

Example 7: Write a program to find how many vowels and consonants are present in strings.

Example 8: Write a program to add 5 in each elements in given list. [10,20,30,40,50]

Example 9: Write a program to add 5 in each elements in given tuple. (10,20,30,40,50)

Example 10: Write a program to create a list from given string.

Examples for FOR_LOOPs

Example 1: Print the first 10 natural numbers using for loop.

Example 2: Python program to print all the even numbers within the given range.

Example 3: Python program to calculate the sum of all numbers from 1 to a given number.

Example 4: Python program to calculate the sum of all the odd numbers within the given range.

Example 5: Python program to print a multiplication table of a given number

Example 6: Python program to display numbers from a list using a for loop.

Example 7: Python program to count the total number of digits in a number.

Example 8: .(madam=madam)

Example 9: Python program that accepts a word from the user and reverses it.

Example 10: Python program to check if a given number is an Armstrong number**. (153=1**3+5**3+3**3)**

Example 11: Python program to count the number of even and odd numbers from a series of numbers.

Example 12: Python program to display all numbers within a range except the prime numbers.

Example 13: Python program to get the Fibonacci series. (0,1,1,2,3,5,8,13,21……………..)

Example 14: Python program to find the factorial of a given number.

Example 15: Python program that accepts a string and calculates the number of digits and letters.

Example 16: Write a Python program that iterates the integers from 1 to 25.

Example 17: Python program to check the validity of password input by users.

Example 18: Python program to convert the month name to a number of days.

----:Functions:----

If a group of statements is repeatedly required then it is not recommended to write these statements every time separately. We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting.This unit is nothing but function.

```
x=10
y=20
print("Addition of x & y =",x+y)
print("Addition of x & y =",x-y)
print("Addition of x & y =",x*y)

x=200
y=100
print("Addition of x & y =",x+y)
print("Addition of x & y =",x-y)
print("Addition of x & y =",x*y)

x=10
y=5
print("Addition of x & y =",x+y)
print("Addition of x & y =",x-y)
print("Addition of x & y =",x*y)

O/P:--
Addition of x & y = 30
Addition of x & y = -10
Addition of x & y = 200
Addition of x & y = 300
Addition of x & y = 100
Addition of x & y = 20000
Addition of x & y = 15
Addition of x & y = 5
Addition of x & y = 50
```

Now, repeated code can be bound into single unit that is called function.

The advantages of function:

1. **Maintaining the code is an easy way.**
2. **Code re-usability.**

Now,

```
def calculate(x, y):
    print("Addition of x & y =",x+y)
     print("Addition of x & y =",x-y)
     print("Addition of x & y =",x*y)

calculate(10,20)
calculate(200,100)
calculate(10,5)

O/P:--
Addition of x & y = 30
Addition of x & y = -10
Addition of x & y = 200
Addition of x & y = 300
Addition of x & y = 100
Addition of x & y = 20000
Addition of x & y = 15
Addition of x & y = 5
Addition of x & y = 50
```

Types of function:---

1. **In-built function :--** The functions which are coming along with Python software automatically,are called built-in functions or pre defined functions.
   Examples:--
   1. print()
   2. id()
   3. type()
   4. len()
   5. eval()
   6. sorted()
   7. count() etc…….

2. User define function:--- The functions which are defined by the developer as per the requirement are called user-defined functions.

Syntax:---

```
def fun_name(parameters….):
  ‘‘‘ doc string….’’’

  Statment1…….
  Statment2…….
  Statment3…….

  return (anything)
# call function
fun_name(arguments....)
```

Important terminology	
def-keyword	mandatory
return-keyword	optional
arguments	optional
parameters	optional
fun_name	mandatory

1. **def keyword –** Every function in python should start with the keyword ‘def’. In other words, python can understand the code as part of a function if it contains the ‘def’ keyword only.
2. **Name of the function –** Every function should be given a name, which can later be used to call it.
3. **Parenthesis** – After the name ‘()’ parentheses are required
4. **Parameters** – The parameters, if any, should be included within the parenthesis.
5. **Colon symbol ‘:’** should be mandatorily placed immediately after closing the parentheses.
6. **Body –** All the code that does some operation should go into the body of the function. The body of the function should have an indentation of one level with respect to the line containing the ‘def’ keyword.
7. **Return statement** – Return statement should be in the body of the function. It’s not mandatory to have a return statement. If we are not writing return statement then default return value is **None**
8. **Arguments**:-- At the time of calling any function, in between the parentheses we passes arguments.

Relation between parameters and arguments:--

When we are creating a function, if we are using parameters in between parenthesis, then it is compulsory to at the time of calling this function, you need to pass correspond arguments.

Parameters are inputs to the function. If a function contains parameters, then at the time of calling, compulsory we should provide values as a arguments, otherwise we will get error.

```
def calculate(x, y):
    print("Addition of x & y =",x+y)
    print("Addition of x & y =",x-y)
    print("Addition of x & y =",x*y)

calculate(10,20)
calculate(200,100)
calculate(10,5)

O/P:--
Addition of x & y = 30
Addition of x & y = -10
Addition of x & y = 200
Addition of x & y = 300
Addition of x & y = 100
Addition of x & y = 20000
Addition of x & y = 15
Addition of x & y = 5
Addition of x & y = 50
```

```
# Write a function to take number as input and print its square value
def square(x):
    print("The Square of",x,"is", x*x)
square(4)
square(5)

O/P:--
The Square of 4 is 16
The Square of 5 is 25
```

```
# Write a function to check whether the given number is even or odd?
def even_odd(num):
    if num%2==0:
        print(num,"is Even Number")
    else:
        print(num,"is Odd Number")

even_odd(10)
even_odd(15)

O/P:--
10 is Even Number
15 is Odd Number
```

```
# Write a function to find factorial of given number?
def fact(num):
    result=1
    while num>=1:
        result=result*num
        num=num-1
    return result
i=int(input("Enter any no "))
print("The Factorial of",i,"is :",fact(i))

O/P:--
Enter any no 5
The Factorial of 5 is : 120
```

Returning multiple values from a function: In other languages like C, C++ and Java, function can return almost one value. But in Python, a function can return any number of values.

```
def add_sub(a,b):
    add=a+b
    sub=a-b
    return add,sub

x,y=add_sub(100,50)
```

```
print("The Addition is :",x)
print("The Subtraction is :",y)

O/P:--
The Addition is : 150
The Subtraction is : 50

          Or

def add_sub(a,b):
    add=a+b
    sub=a-b
    return add,sub
x,y=int(input("Enter first value:")),int(input("Enter second value: "))
print("The Addition is :",x)
print("The Subtraction is :",y)

O/P:--
The Addition is : 100
The Subtraction is : 50
```

```
def calc(a,b):
    add=a+b
    sub=a-b
    mul=a*b
    div=a/b
    return add,sub,mul,div

x,y,z,p=calc(int(input("Enter first value:")),int(input("Enter second value: ")))
print("The Addition is",x)
print("The Substraction is",y)
print("The Multip is",z)
print("The Division is",p)
O/P:--
Enter first value:100
Enter second value: 10
The Addition is 110
The Substraction is 90
The Multip is 1000
The Division is 10.0
```

Types of arguments:

```
def f1(a,b):
    ------
    ------
f1(10,20)
```

There are 4 types are actual arguments are allowed in Python.

1. **positional arguments:**

```
def f1(a,b):
    ------
    ------
f1(10,20)
```

```
def square(x):
   print("The Square of",x,"is", x*x)
square(4)
square(5)

O/P:-
The Square of 4 is 16
The Square of 5 is 25
```

2. **keyword arguments:**

```
def f1(a,b):
    ------
    ------
f1(a=10,b=20)
```

```
def square(x):
   print("The Square of",x,"is", x*x)
square(x=4)
square(x=5)

O/P:--
The Square of 4 is 16
The Square of 5 is 25
```

3. **default arguments:**

```
def f1(a=0,b=0):
    ------
    ------
f1(10,20)
f1()
```

```
def square(x=0):
    print("The Square of",x,"is", x*x)
square(x=4)
square()

O/P:--
The Square of 4 is 16
The Square of 0 is 0
```

4. **Variable length arguments:**

```
def f1(*n):
    ------
    ------
f1(10)
f1(10,20)
f1(10,20,30)
```

```
def sum(*n):
    total=0
    for i in n:
        total=total+i
    print("The Sum=",total)
sum()
sum(10)
sum(10,20)
sum(10,20,30,40)

O/P:--
The Sum= 0
The Sum= 10
The Sum= 30
The Sum= 100
```

5. **key word variable length arguments:**

```
def f1(**n):
        ------
        ------
f1(n1=10, n2=20)
```

```
def display(**kwargs):
    for k,v in kwargs.items():
        print(k,"=",v)
display(n1=10,n2=20,n3=30)
print("-----------")
display(rno=100, name="Neeraj", marks=70, subject="Java")

O/P:--
n1 = 10
n2 = 20
n3 = 30
-----------
rno = 100
name = Neeraj
marks = 70
subject = Java

# ==================================================
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(Name="Neeraj",age=37)

O/P:---
Name: Neeraj
age: 37
```

```
def args_and_kwargs(*args, **kwargs):
    print("Positional arguments:")
    for arg in args:
        print(arg)
    print("Keyword arguments:")
    for key, value in kwargs.items():
        print(f"{key}: {value}")
args_and_kwargs(1, 2, 3, name="Neeraj",age=37, quali="M.Tech")

O/P:--
Positional arguments:
1
2
3
Keyword arguments:
name: Neeraj
age: 37
quali: M.Tech
```

Types of Variables in Python

The variables based on their scope can be classified into two types:

1. **Local variables**
2. **Global variables**

Local Variables in Python:

The variables which are declared inside of the function are called local variables. Their scope is limited to the function i.e we can access local variables within the function only. If we are trying to access local variables outside of the function, then we will get an error.

```
def a():
    x=10
    return "value of Local variable is:",x
def b():
    return "value of Local variable is:",x
p=a()
print(p)
y=b()
print(y)
```

```
O/P:-
('value of Local variable is:', 10)
Traceback (most recent call last):
  File "E:\Python Core_Advance\local.py", line 10, in <module>
    y=b()
  File "E:\Python Core_Advance\local.py", line 6, in b
    return "value of Local variable is:",x
NameError: name 'x' is not defined
```

We Can’t access local variable outside the function:

```
def a():
    x=10
    return "value of Local variable is:",x

def b():
    return "value of Local variable is:",x

p=a()
print(p)
print(x)

O/P:--
('value of Local variable is:', 10)
Traceback (most recent call last):
  File "E:\Python Core_Advance\local.py", line 12, in <module>
    print(x)
NameError: name 'x' is not defined
```

Global variables in Python:

The variables which are declared outside of the function are called global variables. Global variables can be accessed in all functions of that module.

```
a=11
b=12
def m():
   print("a from function m(): ",a)
   print("b from function m(): ",b)
def n():
   print("a from function n(): ",a)
   print("b from function n(): ",b)
m()
```

```
n()


O/P:--
a from function m():  11
b from function m():  12
a from function n():  11
b from function n():  12
```

GLOBAL KEYWORD IN PYTHON

The keyword global can be used for the following 2 purposes:

1. To declare a global variable inside a function
2. To make global variables available to the function.

```
def m1():
  global a
  a=2
  print("a value from m1() function: ", a)
def m2():
  print("a value from m2() function:", a)
m1()
m2()

O/P:--
a value from m1() function: 2
a value from m2() function: 2
```

global and local variables having the same name in Python

```
a=1
def m1():
  global a
  a=2
  print("a value from m1() function:", a)
def m2():
  print("a value from m2() function:", a)
m1()
m2()
```

```
O/P:--
a value from m1() function: 2
a value from m2() function: 2
```

If we use the global keyword inside the function, then the function is able to read-only global variables.
PROBLEM: This would make the local variable no more available.

globals() built-in function in python:

The problem of local variables not available, due to the use of global keywords can be overcome by using the Python built-in function called globals(). The globals() is a built-in function that returns a table of current global variables in the form of a dictionary. Using this function, we can refer to the global variable “a” as: global()[“a”].

```
a=1
def m1():
   a=2
   print("a value from m1() function:", a)
   print("a value from m1() function:", globals()['a'])

m1()

O/P:--
a value from m1() function: 2
a value from m1() function: 1
```

---: Higher order function :---

A function in Python with another function as an argument or returns a function as an output is called the High order function. A function that is having another function as an argument or a function that returns another function as a return in the output

- The function can be stored in a variable.
- The function can be passed as a parameter to another function.
- The high order functions can be stored in the form of lists, hash tables, etc.
- Function can be returned from a function.

1. **Map()**
2. **Filter()**
3. **Lambda()**
4. **Reduce()**
5. **Decorators()**
6. **Generators()**

---: Map :----

Python's map() is a built-in function that enables the processing and transformation of all items in an iterable without the need for an explicit for loop, a technique referred to as mapping. This function is particularly useful when you want to apply a transformation function to each element in an iterable, producing a new iterable as a result. map() is one of the tools that facilitate a functional programming approach in Python.

map() Syntax

map(function, iterable, ...)

map() Arguments

The map() function takes two arguments:

1. function - a function
2. iterable - an iterable like sets, lists, tuples, etc

The map() function returns an object of map class. The returned value can be passed to functions like list() - to convert to list, set() - to convert to a set, and so on.

Example:-1

```
# Map() higher order function---------------

my_list=[10,20,30,40]

def sqr(n):
   return n*n

x=map(sqr,my_list)
print(x)
print(list(x))

O/P:--
<map object at 0x000001EA310E3490>
[100, 400, 900, 1600]
```

Example:-2

```
my_tuple=(10,20,30,40)

def sqr(n):
    return n*n

x=map(sqr,my_tuple)
print(x)
print(tuple(x))

O/P:-
<map object at 0x0000019833A83490>
(100, 400, 900, 1600)
```

Example:-3

```
my_str="Neeraj"
def add(n):
    x=ord(n)
    return x
x=map(add,my_str)
print(x)
print(list(x))

O/P:-
<map object at 0x000001D03A4E3490>
[78, 101, 101, 114, 97, 106]
```

Example:-4

```
my_str="Neeraj"
def add(n):
    x=ord(n)
    return chr(x+5)
x=map(add,my_str)
print(x)
print(list(x))

O/P:-
<map object at 0x0000026D8F1634C0>
['S', 'j', 'j', 'w', 'f', 'o']
```

---: Filter :---

The filter function extracts elements from an iterable (such as a list or tuple) based on the results of a specified function. This function is applied to each element of the iterable, and if it returns True that element is included in the output of the filter() function.

filter() Syntax

The syntax of filter() is: **filter(function, iterable)**

filter() Arguments

The filter() function takes two arguments:

1. **function** - a function
2. **iterable** - an iterable like sets, lists, tuples etc.

The filter() function returns an iterator.

Example 1:-

```
# filter() higher order function ----------------
my_list=[60,10,70,90,55,75,10,20,40]

def fun(n):
    if n>=60:
        return True

x=filter(fun , my_list)
print(list(x))

O/P:--
[60, 70, 90, 75]
```

Example 2:-

```
def check_even(number):
    if number % 2 == 0:
        return True
    return False

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers_iterator = filter(check_even, numbers)
even_numbers = list(even_numbers_iterator)
print(even_numbers)

O/P:--
[2, 4, 6, 8, 10]
```

Example 3:-

```
def check_odd(number):
    if number % 2 != 0:
        return True
    return False

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers_iterator = filter(check_odd, numbers)
odd_numbers = list(odd_numbers_iterator)
print(odd_numbers)

O/P:--
[1, 3, 5, 7, 9]
```

---: Lambda :---

a lambda function is a special type of function without the function name. For example, lambda : print('Hello World').

A lambda function can take any number of arguments, but can only have one expression.

Here, we have created a lambda function that prints 'Hello World'.

lambda Function Declaration: We use the **lambda keyword** instead of def to create a lambda function. Here's the syntax to declare the lambda function:

Syntex:----

lambda argument(s) : expression

argument(s) - any value passed to the lambda function

expression - expression is executed and returned

Let's see an example,

```
# without argument
greet = lambda : print('Hello World')
greet()

O/P:--
Hello World
```

greet = lambda : print('Hello World'). Here, we have defined a lambda function and assigned it to the variable named greet. In the above example, we have defined a lambda function and assigned it to the greet variable. When we call the lambda function, the print() statement inside the lambda function is executed.

```
# with argument
x=lambda p,q,r:3*p+4*q+5*r+5
print(x(10,20,30))
O/P:-
265
```

```
# with argument
user = lambda name : print('Hello', name)
user('Neeraj')

O/P:--
Hello Neeraj
```

---: Reduce :---

The reduce() function in Python is part of the functools module, which needs to be imported before it can be used.

This function performs functional computation by taking a function and an iterable (such as a list, tuple, or dictionary) as arguments. It applies the function cumulatively to the elements of the iterable, reducing it to a single value. Unlike other functions that may return multiple values or iterators, reduce() returns a single value, which is the result of the entire iterable being condensed into a single integer, string, or boolean.

Steps of how to reduce function works:

1. The function passed as an argument is applied to the first two elements of the iterable.
2. After this, the function is applied to the previously generated result and the next element in the iterable.
3. This process continues until the whole iterable is processed.
4. The single value is returned as a result of applying the reduce function on the iterable.

```
from functools import reduce

def product(x,y):
   return x*y

ans = reduce(product, [2, 5, 3, 7])
print(ans)

O/P:--
210
```

```
import functools

my_list=(10,20,60,30,40)
def greater(a,b):
    if a>b:
        return a
    else:
        return b
x=functools.reduce(greater,my_list)
print(x)

O/P:-
60

my_list=(10,20,60,30,40)
def lowest_digit(a,b):
    if a<b:
        return a
    else:
        return b
x=functools.reduce(lowest_digit,my_list)
print(x)

O/P:-
10

my_str="Neeraj"
def greater(a,b):
    if a>b:
        return a
    else:
        return b
x=functools.reduce(greater,my_str)
print("This char have greater asci value:",x)

O/P:-
This char have greater asci value: r
```

---: Decorators :---

Decorators are the most common use of higher-order functions in Python. They enable programmers to modify the behavior of a function or class. By wrapping one function with another, decorators allow us to extend the behavior of the wrapped function without permanently changing it. In this process, functions are passed as arguments to another function and then called within the wrapper function.

```
# defining a decorator
def decorator(func):
   def inner1():
      print("Hello, this is before function execution")
      func()
      print("This is after function execution")
   return inner1
def function():
    print("This is inside the function !!")

function_used = decorator(function)
function_used()

O/P:--
Hello, this is before function execution
This is inside the function !!
This is after function execution

def decorator(func):
   def inner1():
      print("Hello, this is before function execution")
      func()
      print("This is after function execution")
   return inner1

@decorator # second method for calling
def function():
   print("This is inside the function !!")

function() # second method for calling
```

```
O/P:--
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

Examples:---

```
def greet(fun):
    def inner():
        print("Good morning")
        fun()
        print("Thanks for using")
    return inner

def hello():
    print("Hello world")

var=greet(hello)
var()

O/P:--
Good morning
Hello world
Thanks for using
```

With @decorator :------

```
def greet(fun):
    def inner():
        print("Good morning")
        fun()
        print("Thanks for using")
    return inner
@greet
def hello():
    print("Hello world")

hello()
```

```
O/P:--
Good morning
Hello world
Thanks for using
```

Nested decorators :---

```
def decorator1(fun):
    def inner():
        a=fun()
        add = a+5
        return add
    return inner

def decorator2(fun):
    def inner():
        b=fun()
        add = b+5
        return add
    return inner

def fun():
    return 100

fun = decorator2(decorator1(fun))
print(fun())

O/P:--
110
```

With @ decorators :-----

```
def decorator1(fun):
    def inner():
        a=fun()
        add = a+5
        return add
    return inner

def decorator2(fun):
    def inner():
        b=fun()
        add = b+5
        return add
    return inner

@decorator2
@decorator1
def fun():
    return 100

print(fun())

O/P:--
110
```

--- : Generators :---

Generators are similar to functions but produce a sequence of values that can be iterated over using loops. Instead of using return statements, generators use yield statements to return values one at a time.

In Python, a generator is a special type of function that allows you to create an iterator. It's a way to produce a sequence of values on demand, rather than generating them all at once and storing them in memory.

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
for y in var:
    print(y)

O/P:--
5
6
7
8
9
10
```

Next() function in generators:

If we want to retrieve elements from a generator, we can use the next function on the iterator returned by the generator. This is the other way of getting the elements from the generator. (The first way is looping in through it as in the examples above).

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
print("first object from generator :",next(var))
print("Second object from generator :",next(var))
for y in var:
    print(y)

O/P:--
first object from generator : 5
Second object from generator : 6
7
8
9
10
```

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))

O/P:--
object from generator : 5
object from generator : 6
object from generator : 7
object from generator : 8
object from generator : 9
object from generator : 10
```

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
O/P:--
object from generator : 5
object from generator : 6
object from generator : 7
object from generator : 8
object from generator : 9
object from generator : 10
Traceback (most recent call last):
  File "E:\Python Core_Advance\generators.py", line 25, in <module>
    print("object from generator :",next(var))
StopIteration
```

Modules

In Python, a **module** is defined by simply creating a .py file containing Python code. The name of the file (without the .py extension) becomes the name of the module. This file can include functions, classes, variables, and executable statements.

File name is first.py

```
x = 10
def sum(a, b):
  print("Sum of two values: " , (a+b))
def multiplication(a, b):
  print("Multiplication of two values: " , (a*b))
```

Now **first.py** file is a module. **first.py** module contains one variable and two functions.

If we want to use other members (variable, function, etc) of a module in your program, then you should import that module by using the **import keyword**. After importing you can access members by using the name of that module.

Syntax for importing:

Import module_name

Note: Whenever we are using a module in our program, that module's compiled file will be generated and stored in the hard disk permanently.

Renaming or Aliasing a module:

Syntax:-

Import module_name as alias_name

Example:---

```
# file name is first.py

x = 10
def sum(a, b):
   print("Sum of two values: " , (a+b))
def multiplication(a, b):
   print("Multiplication of two values: " , (a*b))


# file name is first01.py
# In this file we are importing first.py file as a module
---------------------------------------------------
import first as mod
mod.sum(10,20)

O/P:--
Sum of two values:  30

----------------------------------------------------
import module as cal
print(10+cal.x)

O/P:--
20
```

From and import * keyword:

We can import some specific members of the module by using the from keyword. The main advantage of the from keyword is we can access members directly without using module names.

Syntax:-

from module_name import fun_name, variable_name

or

from module_name import *(for all funtions & variables)

```
# file name is first.py
```

```
x = 10
def sum(a, b):
   print("Sum of two values: " , (a+b))
def multiplication(a, b):
   print("Multiplication of two values: " , (a*b))


# file name is first01.py
# In this file we are importing first.py file as a module

# from keyword-------------------
from module import sum
sum(10,20)

O/P:--
Sum of two values:  30

# import * keyword-------------------
from module import *
sum(10,20)

O/P:--
Sum of two values:  30
```

<u>Aliasing members with from keyword:</u>

```
# file name is first.py

x = 10
def sum(a, b):
   print("Sum of two values: " , (a+b))
def multiplication(a, b):
   print("Multiplication of two values: " , (a*b))


# file name is first01.py
# In this file we are importing first.py file as a module

# Aliasing members with from keyword--------------
from module import multiplication as multi, sum as add, x as y
```

```
multi(5,10)
add(10,20)
print("Value of x=",y)

O/P:--
Multiplication of two values:  50
Sum of two values:  30
Value of x= 10
```

Note:- Once an alias name is given, we should use the alias name only and not the original name.

Reloading a module in Python:

By default, a module will be loaded only once even though we are importing multiple times. Let's consider a module with name module1.

```
# file name is module1.py

print("This comes from module1.py")

# ----------------------------------------------
# file name is module1.py
# In this file we are importing first02.py file as a module

import module1
import module1
import module1
import module1
import module1
import module1

print("This comes from first01.py file")

O/P:--
This comes from module1.py
This comes from first01.py file
```

The problem in this approach is if a module is updated outside after loading it in our program, then the updated version of the module will not be available to our program. We can solve this problem by reloading modules explicitly based on our requirement wherever needed. We can reload by using the reload() function of the imp module.

Syntax:
import importlib
importlib.reload(module1)

Or

from importlib import reload
reload(module1)

```
# file name is module1.py

print("This comes from module1.py")

# ------------------------------------------------
# file name is module1.py
# In this file we are importing first02.py file as a module

from importlib import reload
import module1
import module1
import module1

print("This comes from first01.py file")

reload(module1)
reload(module1)
reload(module1)
reload(module1)

O/P:--
This comes from module1.py
This comes from first01.py file
This comes from module1.py
This comes from module1.py
This comes from module1.py
This comes from module1.py
```

Note:-- The main advantage of explicit module reloading is we can ensure that updated versions are always available to our program.

Recursion:-- Recursion means that a function calls itself.

```python
# Write a function to Find Factorial of user given no.

def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = int(input("Enter any no: "))
print("The factorial of", num, "is", factorial(num))

# Write a function to find sum  of all numbers from 1 to n
def summation(n):
    if n == 1:
        return 1

    else:
        return n+summation(n-1)
num = int(input("Enter any no: "))
print(summation(num))


# Write a function to find fabonicci series...
def fibonacci(n):
  if n <= 1:
      return n
  else:
      return(fibonacci(n-1) + fibonacci(n-2))
```

```
num = int(input("Enter any no: "))
if num <= 0:
  print("Invalid input ! Please input a positive value")
else:
  print("Fibonacci series:")
for i in range(num):
    print(fibonacci(i))



# Write a function to find the smallest number in a list-------
def findmin(list,n):
    if n == 1:
        return list[0]
    else:
        return min(list[n-1],findmin(list,n-1))

my_list = [1, 4, 24, 17, -5, 10, -22]
n = len(my_list)

print(findmin(my_list,n))

# write a function for Count down from a number
def count_down(start):
    print(start)
    next = start - 1
    if next > 0:
        count_down(next)
count_down(10)

# Write a function to find the greatest common divisor (GCD) of two positive
integers
def gcd(a, b):
   if b == 0:
      return a
   else:
       return gcd(b, a % b)
x=int(input("Enter first no: "))
y=int(input("Enter second no: "))
```

```
print(gcd(x, y))


# Write a function to find the sum of all digits against user given no.
def sum_digits(n):
   if n < 10:
      return n
   else:
      return n % 10 + sum_digits(n // 10)
print(sum_digits(12345))

# Write a function to find the length of a string
def str_len(s):
   if s == '':
      return 0
   else:
      return 1 + str_len(s[1:])
my_str = input("Enter any string: ")
print(str_len(my_str))

# Write a function to find given string is pelendrom or not.
def is_palindrome(str):
  if len(str) < 2:
    return ("Given string is palindrome")
  if str[0] != str[-1]:
    return ("Given string is not a palindrome")
  return is_palindrome(str[1:-1])
my_str = input("Enter any string: ")
my_str1 = is_palindrome(my_str)
print(my_str1)
```

---: File Handling :---

File handling is used to store data permanently in a file.

Basic operations:-

1. **open('file_name with extention','mode')** (default extention and mode are .txt and r)
2. Perform **read()/write()** operations.
3. **close()** the file.

Type of files:

1. **text file(.txt) -** It stores data/characters in ASCII form.
2. **binary(.dat) -** It is used to store audio, video, or image.
3. **csv(.csv) –** It is used to store data in key value format.

Mode:-

1. **'r' :- read mode**(default mode)-It open file in read mode. If file not exist then it give error of FileNotFoundError: [Errno 2] No such file or directory: filename.
2. **'w' :- write mode** – It open file in write mode and if in file previous data exist then it override with new data. If file not exist, it create new file.
3. **'a' :- append mode -** It open file in append mode and if in file, previous data exist then cursor position in last of the previous data. If file not exist, it create new file.
4. **'x' :- exclusive mode(Create mode)** – This mode is used to create a new file only.

File object attributes –

1. **closed:** It returns true if the file is closed and false when the file is open.
2. **encoding:** Encoding used for byte string conversion.
3. **mode:** Returns file opening mode
4. **name:** Returns the name of the file which file object holds.

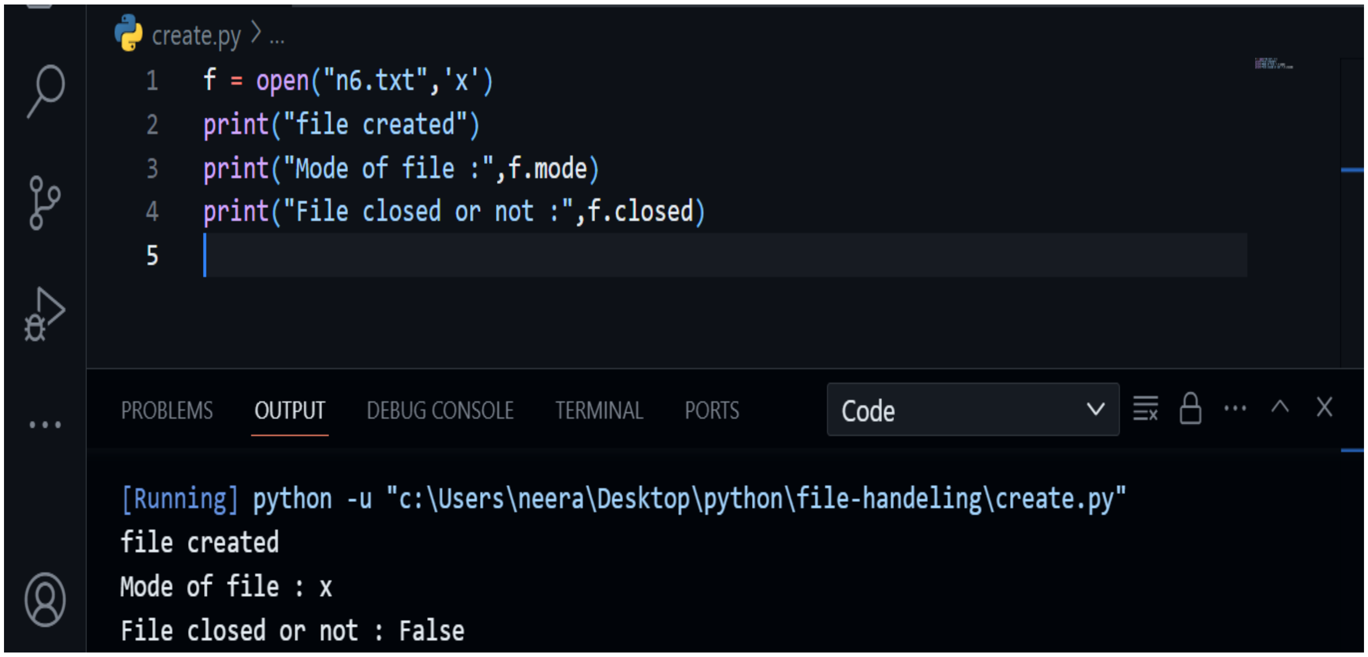
5. **newlines:** Returns “\r”, “\n”, “\r\n”, None or a tuple containing all the newline types seen.

open	**read mode**	**write mode**	**close**
open()	read(n)	write()	closed
	read()	writelines()	close()
	readline()	writable()	
	readlines()		
	readable()		

open("file_name","mode")		
If file exist	**Mode**	**If file not exis**
Give error: file already exist.	**"x"**	Created a new file with given name.
Write mode, cursor present in zero index position. That means previous data will be destroyed.	**"w"**	Created a new file with given name.
Normal as read mode.	**"r"**	Give error: file not exist.
Normal as append mode with cursor position ahead of previous data.	**"a"**	Created a new file with given name.

Create Mode examples:--

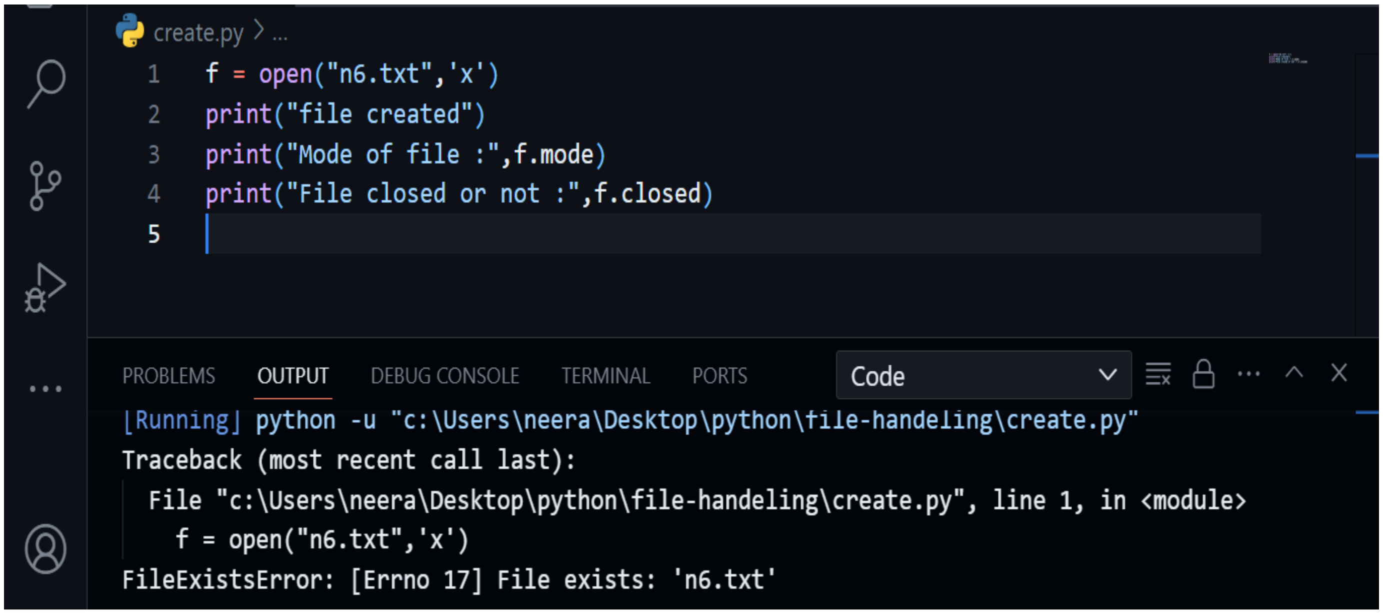
1. **if file was not exist, then it create a new file with mention name**



```python
f = open("n6.txt",'x')
print("file created")
print("Mode of file :",f.mode)
print("File closed or not :",f.closed)
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\create.py"
file created
Mode of file : x
File closed or not : False
```

2. **If file already exist, then it give error that already exist.**

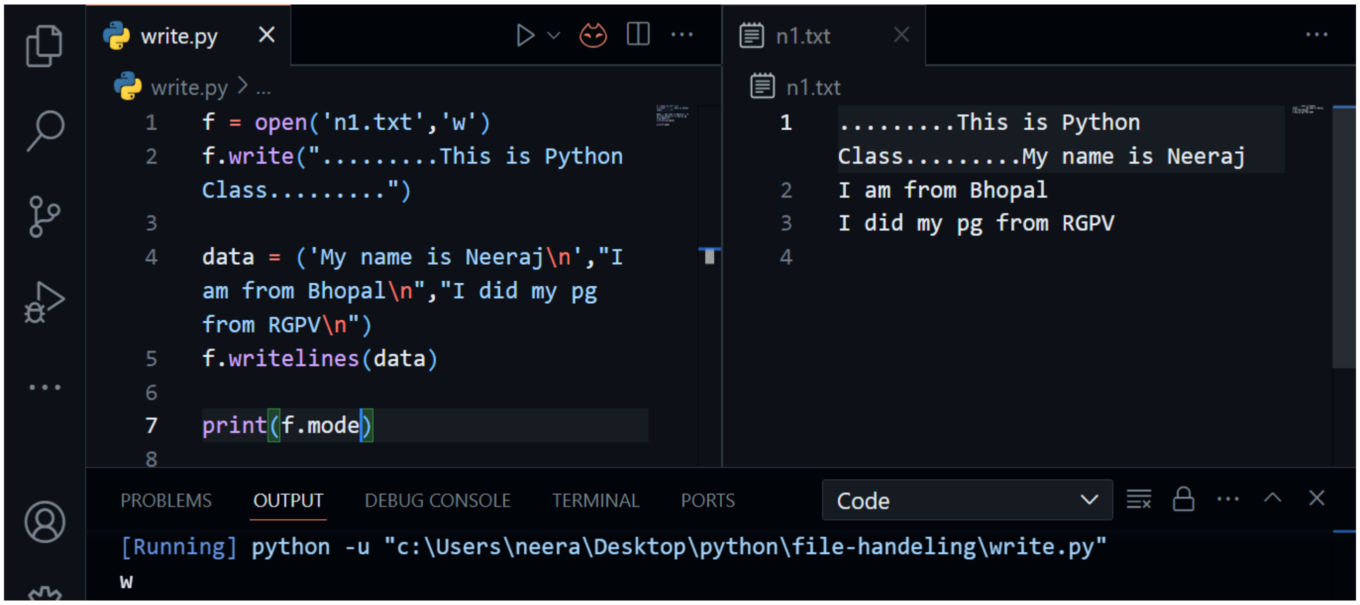


```python
f = open("n6.txt",'x')
print("file created")
print("Mode of file :",f.mode)
print("File closed or not :",f.closed)
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\create.py"
Traceback (most recent call last):
  File "c:\Users\neera\Desktop\python\file-handeling\create.py", line 1, in <module>
    f = open("n6.txt",'x')
FileExistsError: [Errno 17] File exists: 'n6.txt'
```

Write mode example:---

1. **if file was not exist, then it create a new file with mention name**



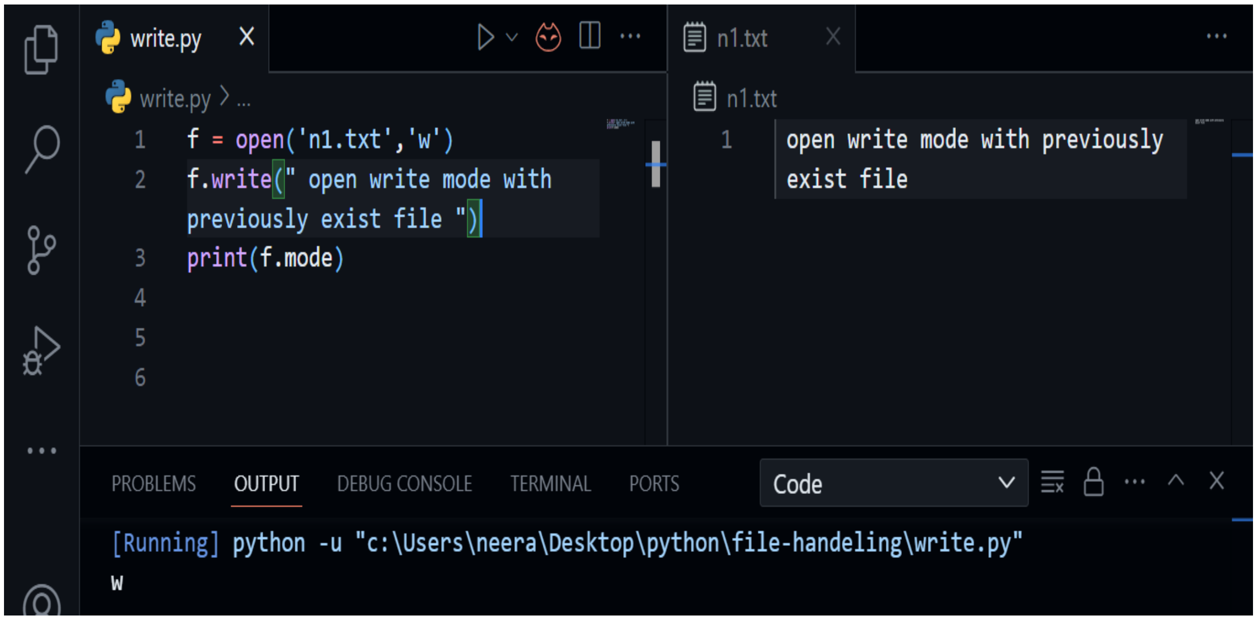
```python
f = open('n1.txt','w')
f.write(".........This is Python Class.........")

data = ('My name is Neeraj\n',"I am from Bhopal\n","I did my pg from RGPV\n")
f.writelines(data)

print(f.mode)
```

n1.txt

```
.........This is Python Class.........My name is Neeraj
I am from Bhopal
I did my pg from RGPV
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\write.py"
w
```

2. Write mode, cursor present in zero index position. That means previous data will be destroyed.



```python
f = open('n1.txt','w')
f.write(" open write mode with previously exist file ")
print(f.mode)
```

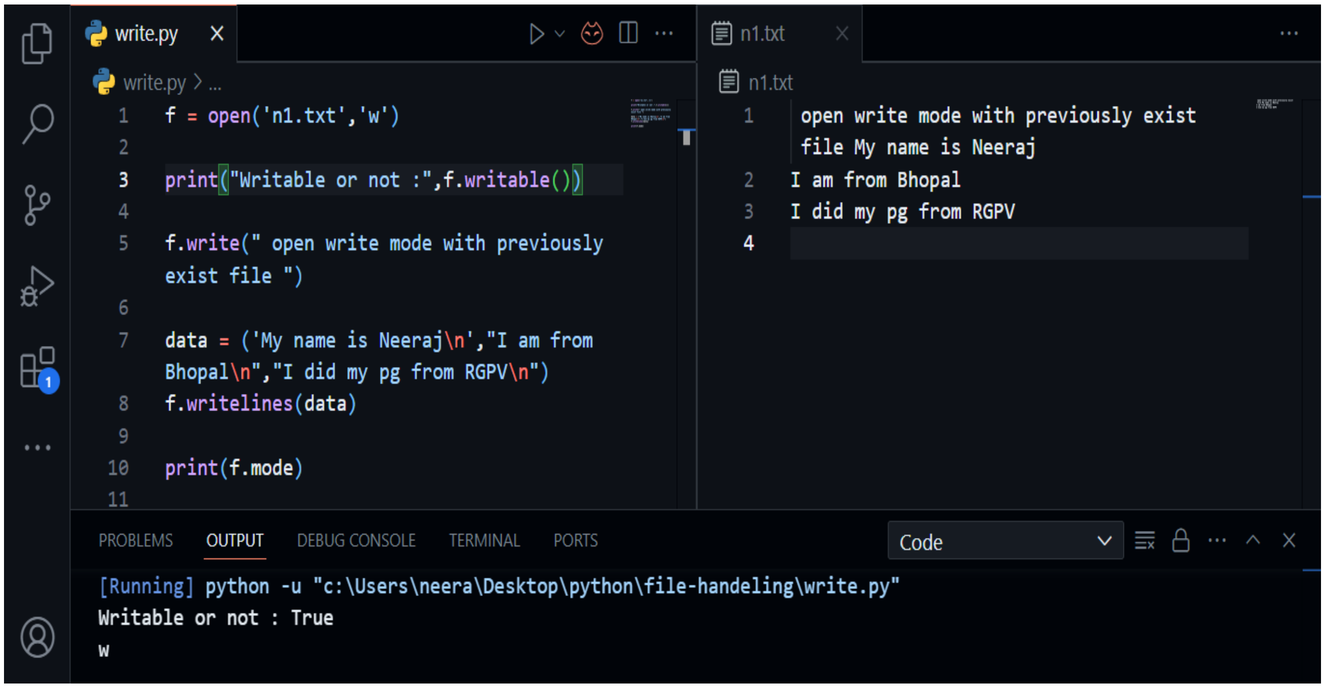
n1.txt

```
open write mode with previously exist file
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\write.py"
w
```

Write mode methods:

1. **write() – it is used to write single line of data.**
2. **writelines() – It is used for multiple lines of data**
3. **writable() – To check file is writable or not.**



```python
f = open('n1.txt','w')

print("Writable or not :",f.writable())

f.write(" open write mode with previously exist file ")

data = ('My name is Neeraj\n',"I am from Bhopal\n","I did my pg from RGPV\n")
f.writelines(data)

print(f.mode)

```

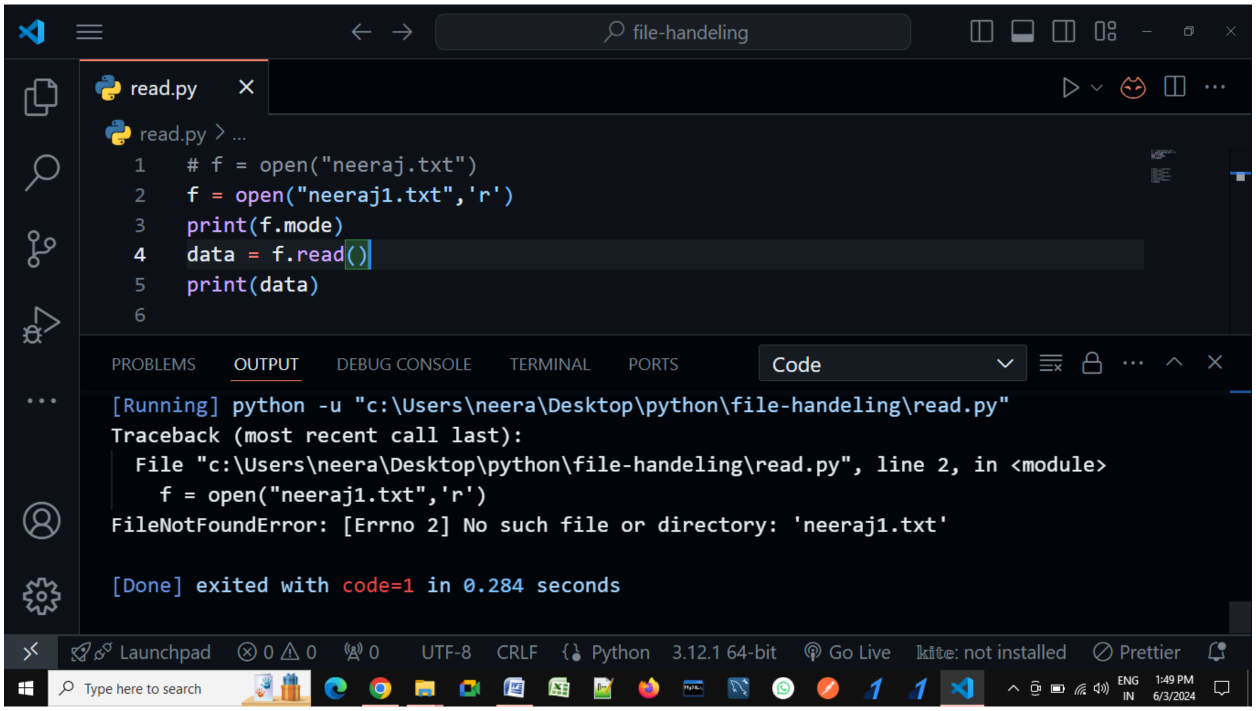
n1.txt:

```
open write mode with previously exist file My name is Neeraj
I am from Bhopal
I did my pg from RGPV

```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\write.py"
Writable or not : True
w
```

Read mode:--

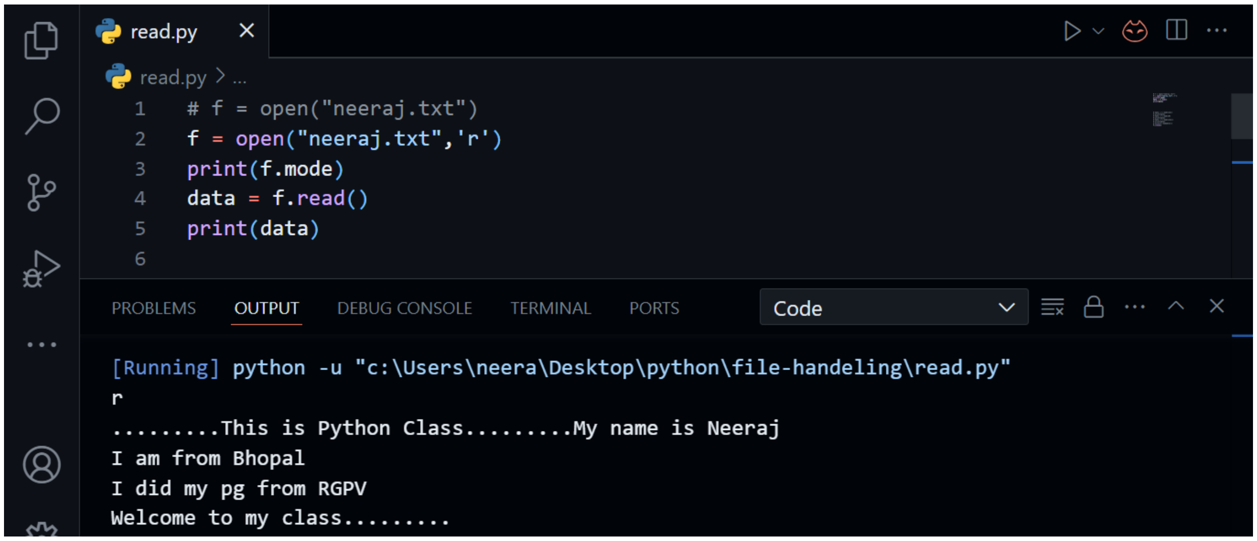
1. Give error: file not exists.



```python
# f = open("neeraj.txt")
f = open("neeraj1.txt",'r')
print(f.mode)
data = f.read()
print(data)

```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\read.py"
Traceback (most recent call last):
  File "c:\Users\neera\Desktop\python\file-handeling\read.py", line 2, in <module>
    f = open("neeraj1.txt",'r')
FileNotFoundError: [Errno 2] No such file or directory: 'neeraj1.txt'

[Done] exited with code=1 in 0.284 seconds
```

2. If File exists, then Normal as read mode

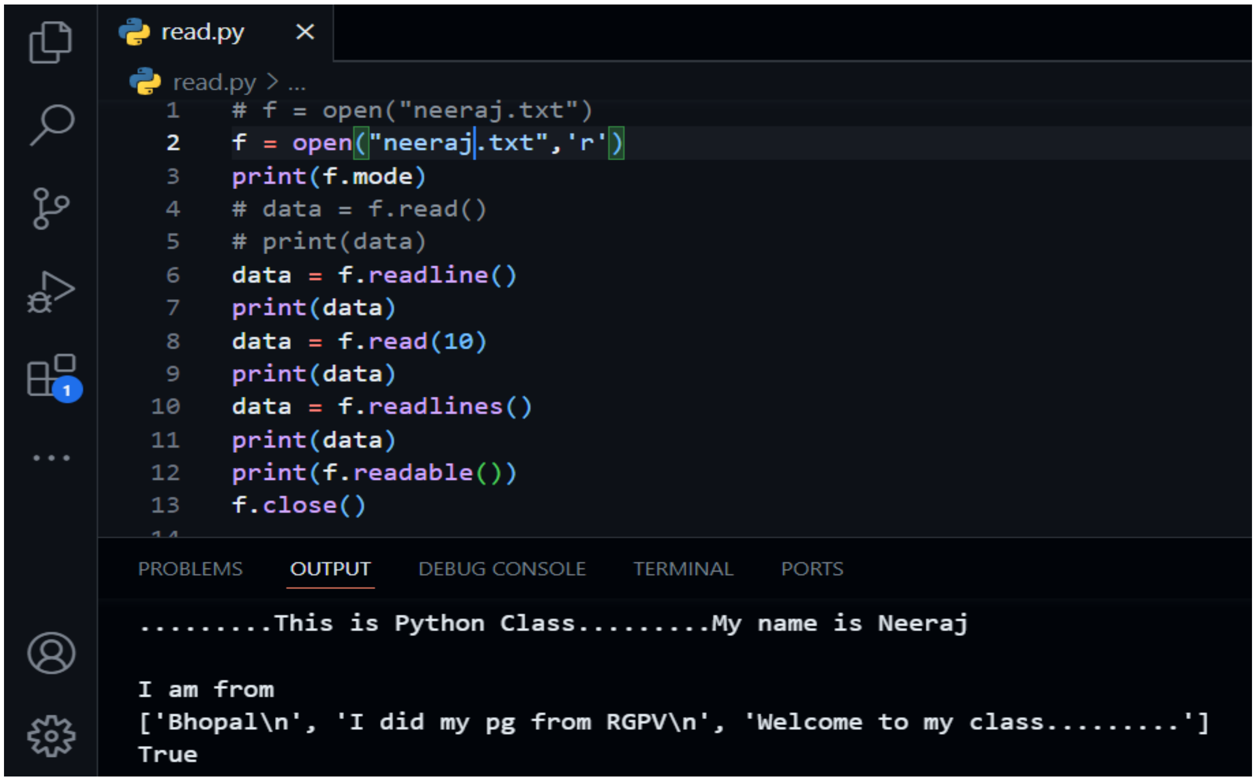


```python
# f = open("neeraj.txt")
f = open("neeraj.txt",'r')
print(f.mode)
data = f.read()
print(data)

```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\read.py"
r
.........This is Python Class.........My name is Neeraj
I am from Bhopal
I did my pg from RGPV
Welcome to my class.........
```

Read mode methods :------

1. **read(n)**
2. **read()**
3. **readline()**
4. **readlines()**
5. **readable()**



```python
# f = open("neeraj.txt")
f = open("neeraj.txt",'r')
print(f.mode)
# data = f.read()
# print(data)
data = f.readline()
print(data)
data = f.read(10)
print(data)
data = f.readlines()
print(data)
print(f.readable())
f.close()
```

```
.........This is Python Class.........My name is Neeraj

I am from
['Bhopal\n', 'I did my pg from RGPV\n', 'Welcome to my class.........']
True
```

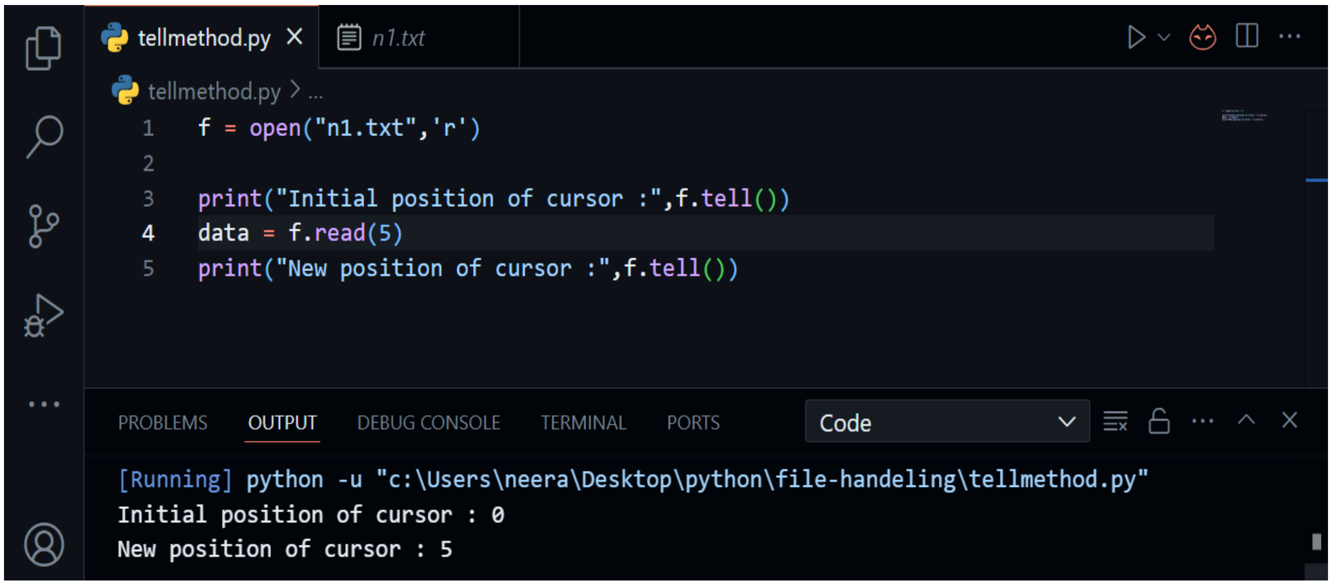
Delete data, file, or folder with python:---

```python
import os, shutil

os.remove('new1/n4.txt')
print(".........n4 file deleted .....")

os.remove('n4.txt')
print(".........n3 file deleted.....")

os.mkdir("new2")
print(".........new1 folder created......")

os.chdir("new2")
print(".........change fron one directory to another directory......")

x = os.getcwd()
print(x)

os.chdir("neeraj")
print(".........change fron one directory to another directory......")
```

```
# get current working directory.......
x = os.getcwd()
print(x)

f = open('new1/n4.txt','a')
print(".........create new files within the new1 folder......")

os.rmdir('new1')
print("...............Delete empty folder..........")

shutil.rmtree('new1')
print("...............Delete empty folder..........")

os.rename('new1',"neeraj")
print(".......Rename Folder name......")

os.rename('n1.txt',"neeraj.txt")
print(".......Rename File name......")
```

<u>tell() & seek()</u>

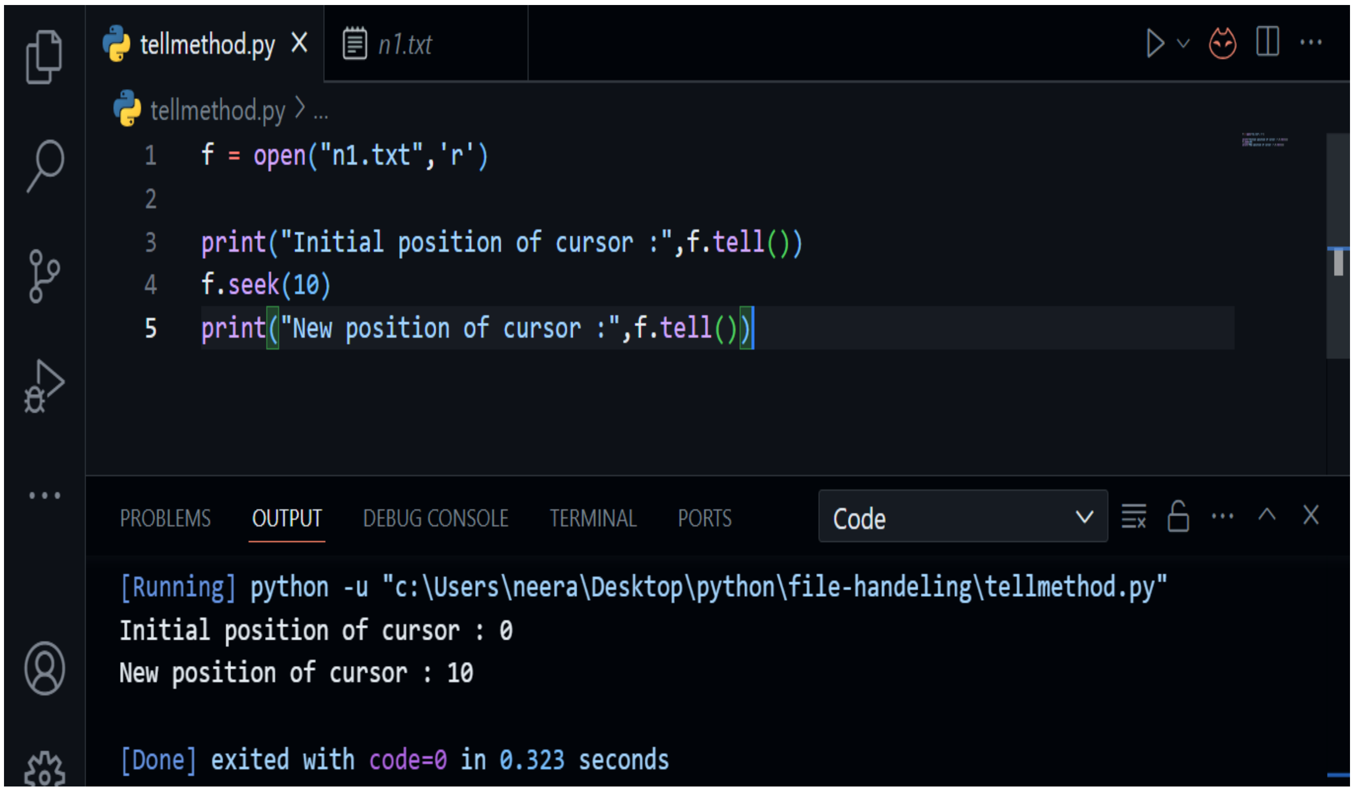
tell() : With the help of tell() we find out the current position of cursor.



```python
f = open("n1.txt",'r')

print("Initial position of cursor :",f.tell())
data = f.read(5)
print("New position of cursor :",f.tell())
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\tellmethod.py"
Initial position of cursor : 0
New position of cursor : 5
```

seek() : With the help of seek() method, we can move cursor from our required positions.



```python
f = open("n1.txt",'r')

print("Initial position of cursor :",f.tell())
f.seek(10)
print("New position of cursor :",f.tell())
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\tellmethod.py"
Initial position of cursor : 0
New position of cursor : 10

[Done] exited with code=0 in 0.323 seconds
```

Syntax:--

seek(attribute1, attribute2)

attribute1 : Where we want our cursor

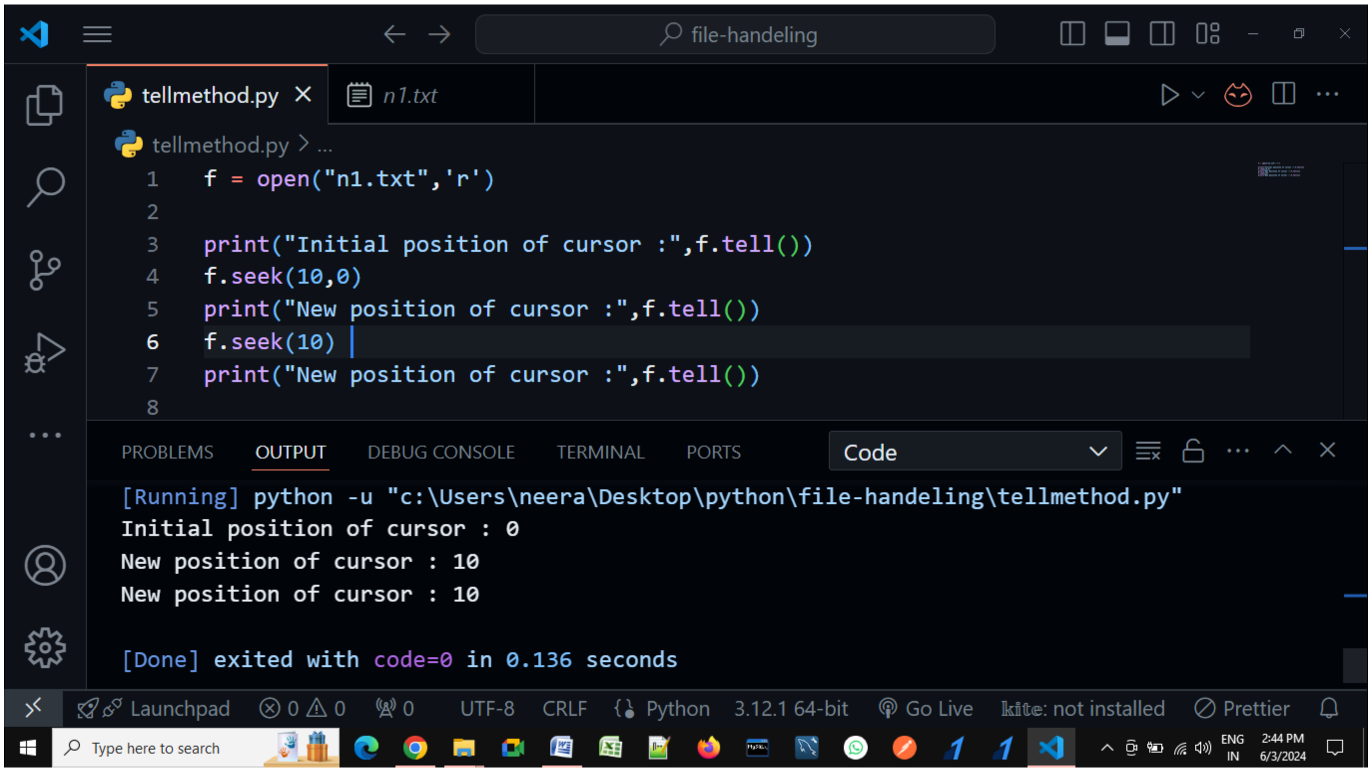
attribute2: Start from which position

1. 0 (start from beginning)- by-default that means not required to write.
2. 1 (start from current position)
3. 2 (start from last position(for negative indexing))

Note: In python 3.2, 1 and 2 both are used only in binary mode

Examples:----

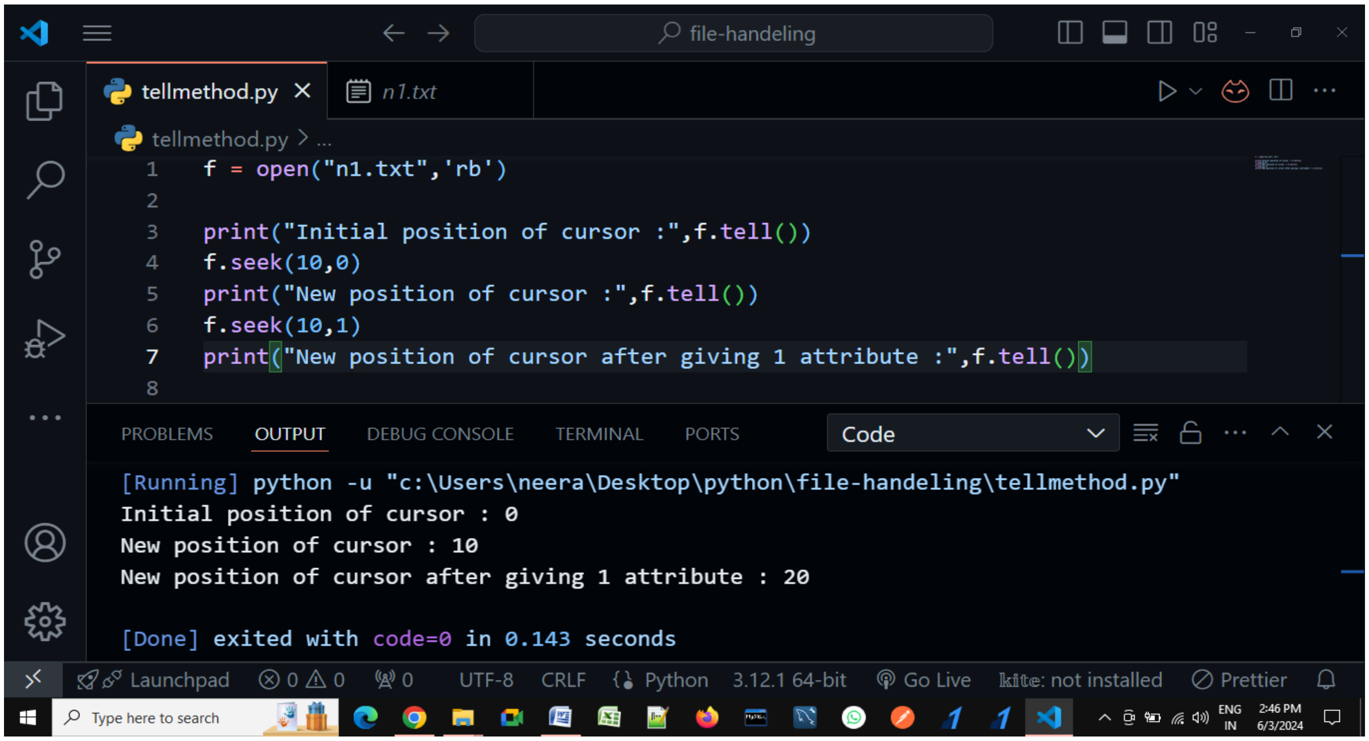
Attribute2 : 0 (start from beginning)------------



```python
f = open("n1.txt",'r')

print("Initial position of cursor :",f.tell())
f.seek(10,0)
print("New position of cursor :",f.tell())
f.seek(10)
print("New position of cursor :",f.tell())

```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\tellmethod.py"
Initial position of cursor : 0
New position of cursor : 10
New position of cursor : 10

[Done] exited with code=0 in 0.136 seconds
```

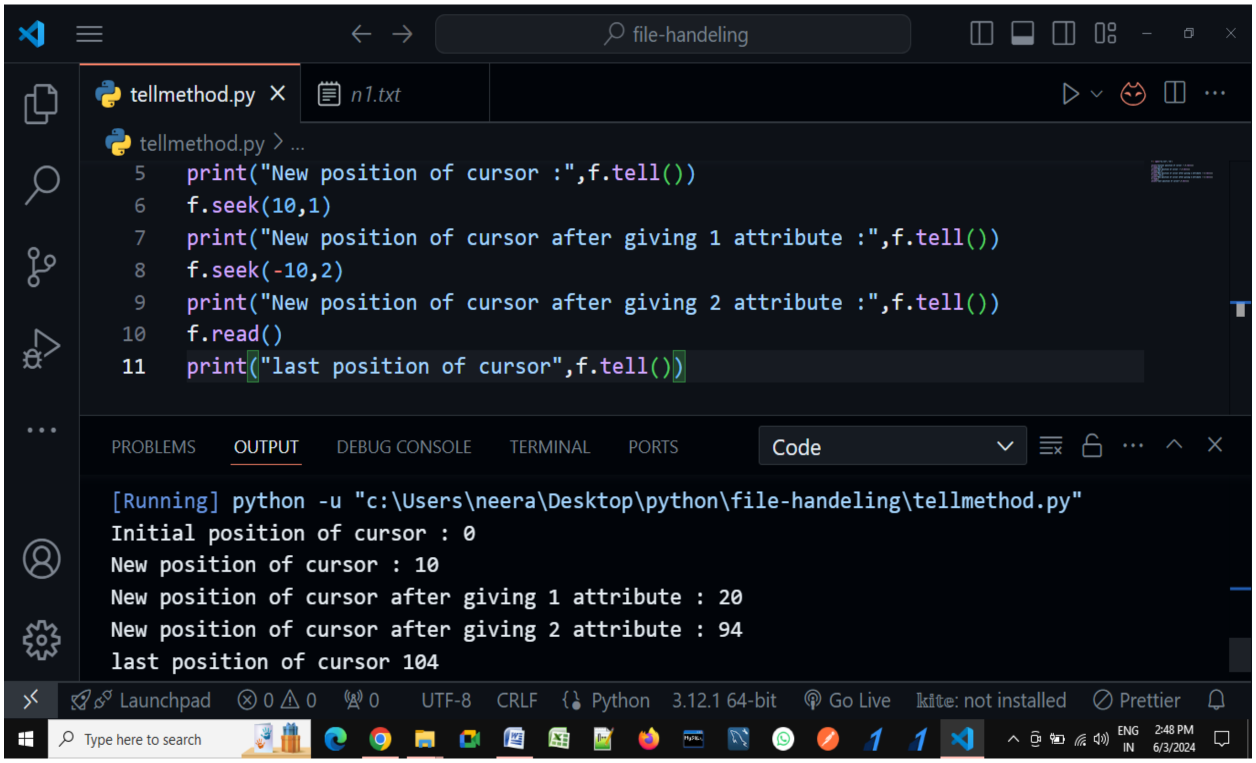
Attribute2 : 1 (start from current position)



```python
f = open("n1.txt",'rb')

print("Initial position of cursor :",f.tell())
f.seek(10,0)
print("New position of cursor :",f.tell())
f.seek(10,1)
print("New position of cursor after giving 1 attribute :",f.tell())
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\tellmethod.py"
Initial position of cursor : 0
New position of cursor : 10
New position of cursor after giving 1 attribute : 20

[Done] exited with code=0 in 0.143 seconds
```

Attribute2 : 2 (start from last position(for negative indexing))



```python
print("New position of cursor :",f.tell())
f.seek(10,1)
print("New position of cursor after giving 1 attribute :",f.tell())
f.seek(-10,2)
print("New position of cursor after giving 2 attribute :",f.tell())
f.read()
print("last position of cursor",f.tell())
```

```
[Running] python -u "c:\Users\neera\Desktop\python\file-handeling\tellmethod.py"
Initial position of cursor : 0
New position of cursor : 10
New position of cursor after giving 1 attribute : 20
New position of cursor after giving 2 attribute : 94
last position of cursor 104
```

Error Handling

In any programming language, there are 2 types of errors possible. They are:

1. **Syntax Errors**
2. **Runtime Errors**

Syntax Errors

The errors which occur because of invalid syntax are called syntax errors.

```
x=123
if x==123
print("Hello")

O/P:--
  File "E:\Python Core_Advance\ex.py", line 2
    if x==123
             ^
SyntaxError: expected ':'
```

Here we missed placing a colon in the if condition, which is violating the syntax. Hence, syntax error.

Runtime Errors in Python

While executing the program if something goes wrong then we will get Runtime Errors. They might be caused due to,

1. **End-User Input**
2. **Programming Logic**
3. **Memory Problems etc.**

Note:---Such types of errors are called exceptions.

Normal Flow of Program Execution in Python:

In a program, if all statements are executed as per the conditions successfully and if we get the output as expected then that flow is called the normal flow of the Program Execution. The below program will get executed successfully from start to end.

```
print('One')
print('Two')
print('Three')
print('Four')
print('Five')

O/P:--
PS E:\Python Core_Advance> py ex.py
One
Two
Three
Four
Five
```

Abnormal Flow of Program Execution in Python:

While executing statements in a program, if any error occurs at runtime, then immediately program flow gets terminated abnormally, without executing the other statements. This kind of termination is called an abnormal flow of execution. The following example terminated abnormally.

```
print('One')
print('Two')
print(10/0)
print('Four')
print('Five')

O/P:--
PS E:\Python Core_Advance> py ex.py
One
Two
Traceback (most recent call last):
  File "E:\Python Core_Advance\ex.py", line 14, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Above program terminated in the middle of the execution where a run time error occurred. As discussed, if a runtime error occurs it won't execute the remaining statements.

Exception

An unwanted or unexpected event that disturbs the normal flow of the program is called an exception. Whenever an exception occurs, immediately the program will terminate abnormally. In order to get our program executed normally, we need to handle those exceptions on high priority.

Exception Handling

Exception handling does not mean repairing or correcting the exception. Instead, it is the process in which we define a way so that the program doesn't terminate abnormally due to the exceptions.

Default Exception Handing in Python:

In python, for every exception type, a corresponding class is available and every exception is an object to its corresponding class. Whenever an exception occurs, Python Virtual Machine (PVM) will create the corresponding exception object and will check for handling code.

If handling code is not available, then the Python interpreter terminates the program abnormally and prints corresponding exception information to the console. The rest of the program won't be executed.

Note: Every Exception in Python is a class. The BaseException class is the root class for all exception classes in the python exception hierarchy and all the exception classes are child classes of BaseException. The Programmers need to focus and understand clearly the Exception and child classes.

Handle Exceptions

Using Try-Except statements we can handle exceptions in python.

- **Try block:** try is a keyword in python. The code which may be expected to raise an exception should be written inside the try block.
- **Except block:** except is a keyword in python. The corresponding handling code for the exception, if occurred, needs to be written inside the except block.

Syntax:

try:

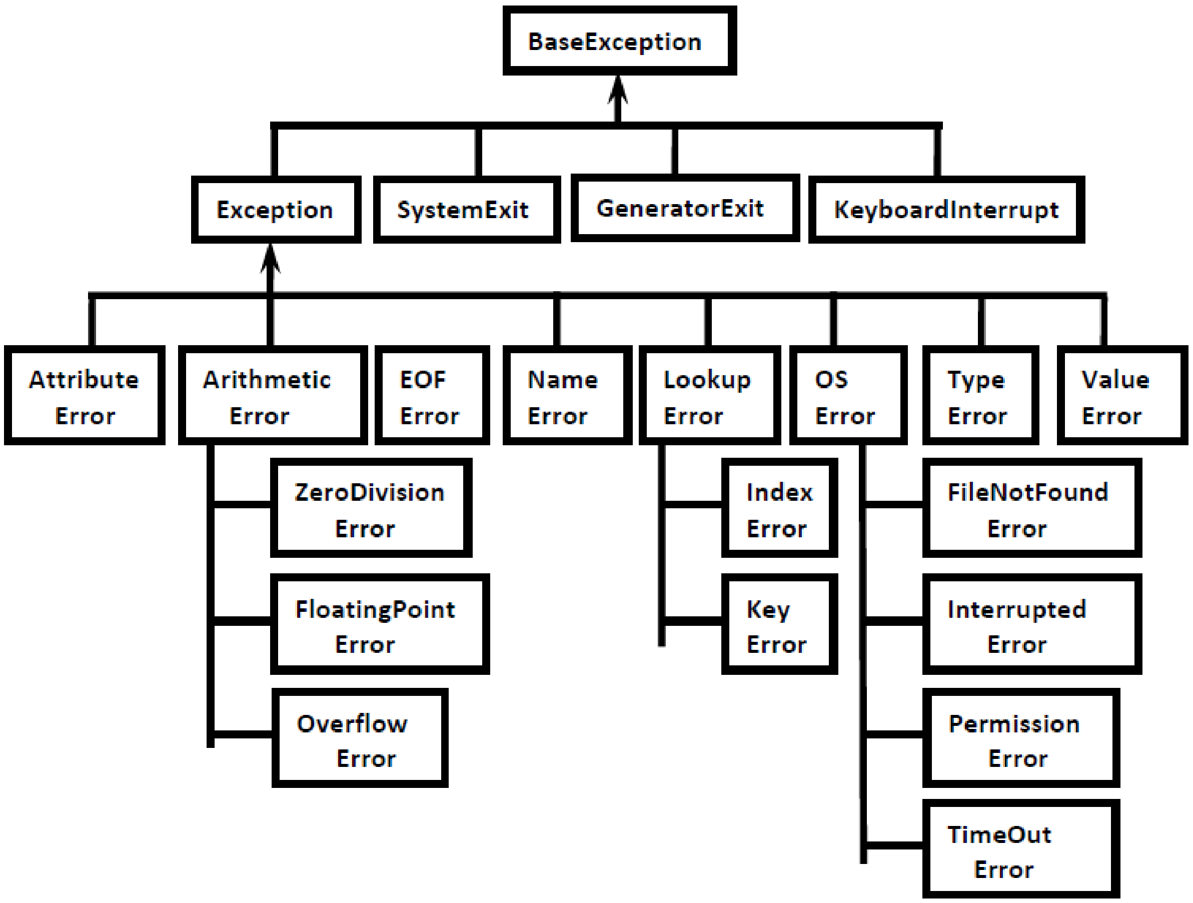
    Write those line where exception occurred

except Exception:

    Exception handling code

```
print('One')
print('Two')
try:
  print(10/0)
except ZeroDivisionError:
  print("Exception passed")
print('Four')
print('Five')

O/P:--
PS E:\Python Core_Advance> py ex.py
One
Two
Exception passed
Four
Five
```

Python's Exception Hierarchy:--



BaseException:

1. **Exception**
2. **SystemExit** ------ **exit()**
3. **GeneratorExit** ------ **generator.close()**
4. **KeyboardInterrupt** ------ **ctrl+c**

Exception:

1. **Attribute Error**
   AttributeError can be defined as an error that is raised when an attribute reference or assignment fails.

2. **Arithmetic Error--------- ZeroDivision, FloatingPoint, Overflow**
   ArithmeticError is simply an error that occurs during numeric calculations.

```
x=int(input('Enter any no :---  '))
y=int(input('Enter any no :---  '))
print(x/y)

O/P:--
PS E:\Python Core_Advance> py ex.py
Enter any no :---  10
Enter any no :---  0
Traceback (most recent call last):
  File "E:\Python Core_Advance\ex.py", line 29, in <module>
    print(x/y)
ZeroDivisionError: division by zero
```

```
print("Simple program for showing overflow error")
import math
print("The exponential value is")
print(math.exp(1000))

O/P:--
Simple program for showing overflow error
The exponential value is
Traceback (most recent call last):
  File "E:\Python Core_Advance\ex.py", line 35, in <module>
```

```
    print(math.exp(1000))
OverflowError: math range error
```

3. **EOF Error----(End of file)**
   EOF stands for "end of file," and this syntax error occurs when Python detects an unfinished statement or block of code. This can happen for many reasons, but the most likely cause is missing punctuation or an incorrectly indented block.
4. **Name Error**
   In Python, a NameError: name 'x' is not defined error is raised when the program attempts to access or use a variable that has not been defined or assigned a value

```
print(x)

O/P:--
PS E:\Python Core_Advance> py ex.py
Traceback (most recent call last):
  File "E:\Python Core_Advance\ex.py", line 37, in <module>
    print(x)
NameError: name 'x' is not defined
```

5. **Lookup Error------------- Index, Key**
   The LookupError exception in Python forms the base class for all exceptions that are raised when an index or a key is not found for a sequence or dictionary respectively.

6. **OS Error------------------- FileNotFound, Interrupted, Permission, TimeOut**
   The os. error in Python is the error class for all I/O errors and is an alias of the OSError exception

```
f = open('neeraj.txt')
data = f.read()
print(data)
f.close()
```

```
O/P:--
PS E:\Python Core_Advance> py file.py
Traceback (most recent call last):
  File "E:\Python Core_Advance\file.py", line 97, in <module>
    f = open('neeraj.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'neeraj.txt'
```

7. **Type Error**

   TypeError is raised whenever an operation is performed on an incorrect/unsupported object type. For example, using the + (addition) operator on a string and an integer value will raise a TypeError.

```
x=input('Enter any str :---  ')
y=input('Enter any str :---  ')
print(x*y)

O/P:--
PS E:\Python Core_Advance> py ex.py
Enter any str :---  neeraj
Enter any str :---  kumar
Traceback (most recent call last):
  File "E:\Python Core_Advance\ex.py", line 29, in <module>
    print(x*y)
TypeError: can't multiply sequence by non-int of type 'str'
```

8. **Value Error**

   a ValueError occurs when a correct argument type but an incorrect value is supplied to a function

```
x=int(input('Enter any no :---'))
print(x)
O/P—
PS E:\Python Core_Advance> py ex.py
Enter any no :---  Neeraj
Traceback (most recent call last):
  File "E:\Python Core_Advance\ex.py", line 27, in <module>
    x=int(input('Enter any no :---  '))
ValueError: invalid literal for int() with base 10: 'Neeraj'
```

Object Oriented Programming System

For introducing real word entities, in our programming world we need object oriented concept. In object oriented concept, we are having so many important terminologies like,

1. class
2. object

class : class may be define as a blueprint of an object, in which we are defining object properties and action/behaviors. Hear, properties can be represented by variables and action or behavior can be represented by methods.(Class may be define as a blue-print that contains attributes like variables and methods).

Syntax for defining any class:

```
    Class class_name:
        "doc string"
        Contractors-
        Variables-
            Instance variable
            Static variable
            Local variable
        Methods-
            Instance method
            Static method
            Class method
```

Object : instance of a class is known as object.

Properties of oops concept:

1. abstraction	(Data-security)
2. encapsulations	(Data-security)
3. inheritance	(Code-Reusability)
4. polymorphism	(Code-Reusability)

What are Constructors :-- In any programming language, a constructor is a method that is automatically invoked whenever an instance (object) of a class is created. There is no need to explicitly call it. Typically, the constructor is used to perform any necessary initializations when the object is being created. In Python, the constructor is a method named __init__. The first parameter of this method should be self, which refers to the instance or object of the current class.

Syntax:
def __init__(self):
        body of the constructor

In Python, Constructor is mandatory or not:--- No, it is not mandatory for a class to have a constructor. Whether a class includes a constructor depends entirely on the requirements. If any initialization is needed during object creation, then a constructor should be used. Otherwise, it is not necessary. A Python program remains valid even without a constructor.

```
class Test:
    def __init__(self):
        print("Constructor executed....!!!!!!!")
t = Test()

O/P:--
Constructor executed....!!!!!!!
```

Can constructor called explicitly? :---

Yes, we can call constructor explicitly with object name. But since the constructor gets executed automatically at the time of object creation, it is not recommended to call it explicitly.

```
class Student:
    def __init__(self):
        print("Constructor called............")

obj = Student() # Constructor called implecitilly or automatically when we are creating object...
obj.__init__() # we are calling explicitally constructor method
obj.__init__()                # we are calling explicitally constructor method
obj.__init__()                # we are calling explicitally constructor method

O/P:--
Constructor called...........
Constructor called...........
Constructor called...........
Constructor called...........
```

Note: Including a constructor is not mandatory. If we do not include a constructor, Python will internally provide an empty constructor. This can be verified using the dir(class_name) built-in method.

```
# Constructor is not mandatory for any class, it is optional on the bases of our requirement.
Class Test:
    def m1(self):
        print(“Instence method executed….!!!!!!”)
t = Test()
t.m1()
print(dir(Test))

O/P:--
Instence method executed….!!!!!!
[‘__class__’, ‘__delattr__’, ‘__dict__’, ‘__dir__’, ‘__doc__’, ‘__eq__’,
‘__format__’, ‘__ge__’, ‘__getattribute__’, ‘__gt__’, ‘__hash__’, ‘__init__’,
‘__init_subclass__’, ‘__le__’, ‘__lt__’, ‘__module__’, ‘__ne__’, ‘__new__’,
‘__reduce__’, ‘__reduce_ex__’, ‘__repr__’, ‘__setattr__’, ‘__sizeof__’, ‘__str__’,
‘__subclasshook__’, ‘__weakref__’,
‘m1’]
```

How many parameters we passed in constructor:---
Constructor can accept n number of parameters. It totally depends on our requirements. All values that need to be initialized during object creation should be passed to the constructor. The first parameter of the constructor should always refer to the current instance, which is typically denoted as self.

Without parameter (except self):

```
class Student:
    def __init__(self):
        print("Constructor called............")
        print(self) #
stu = Student()

O/P:--
Constructor called............
<__main__.Student object at 0x00000245668B3400>
```

Hear, self contains the current object address.
With parameters:

```
class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
        self.marks = marks
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)

# help(Student)
obj1= Student("Neeraj",101,84)
print(obj1.name)
print(obj1.roll)
print(obj1.marks)
print(Student.__doc__)
obj1.display()
```

```
O/P:--
Neeraj
101
84
 This class is develop by Neeraj for demo
my name is Neeraj
my roll no is 101
my marks is 84
```

Multiple constructors in class:

We can define multiple constructors (__init__()) methods in a class but always last one is executed.

```
class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
        self.marks = marks
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city

    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

# help(Student)
obj1= Student("Neeraj",101,84)
obj1= Student("Neeraj",101,84,"Bhopal")
print(obj1.name)
print(obj1.roll)
print(obj1.marks)
print(Student.__doc__)
obj1.display()
```

```
O/P:---

obj1= Student("Neeraj",101,84)
TypeError: Student.__init__() missing 1 required positional argument: 'city'


class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
        self.marks = marks
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

# obj1= Student("Neeraj",101,84)
obj1= Student("Neeraj",101,84,"Bhopal")
print(obj1.name)
print(obj1.roll)
print(obj1.marks)
print(obj1.city)
obj1.display()

O/P:--
Neeraj
101
84
Bhopal
my name is Neeraj
my roll no is 101
my marks is 84
my city is Bhopal
```

Types of Variables in a Class in Python:---

Inside a class, we can have three types of variables. They are:

1. **Instance variables (object level variables)**
2. **Static variables (class level variables)**
3. **Local variables**

1.Instance Variables in Python:
If the value of a variable is changing from object to object then such variables are called as instance variables.

```
# Instence Variable..........

class Student:
   ''' This class is develop by Neeraj for demo'''
   def __init__(self,name,roll,marks,city):
      self.name=name
      self.roll=roll
      self.marks = marks
      self.city = city
   def display(self):
      print("my name is", self.name)
      print("my roll no is", self.roll)
      print("my marks is", self.marks)
      print("my city is", self.city)

stu1 = Student("Neeraj",101,"90","Bhopal")
stu2 = Student("Rahul",102,"92","Indore")
print(stu1.name)
print(stu2.name)
stu1.display()
stu2.display()
print(stu1.__dict__)
print(stu2.__dict__)


O/P:--
Neeraj
Rahul
my name is Neeraj
my roll no is 101
my marks is 90
my city is Bhopal
my name is Rahul
my roll no is 102
my marks is 92
my city is Indore
{'name': 'Neeraj', 'roll': 101, 'marks': '90', 'city': 'Bhopal'}
{'name': 'Rahul', 'roll': 102, 'marks': '92', 'city': 'Indore'}
```

```
# Instence Variable..........(By using instence method)
class Student:
    ''' This class is develop by Neeraj for demo'''
    def display(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)
stu = Student()
stu.display("Neeraj",101,"90","Bhopal")
print(stu.name)
stu.display("Rahul",102,"92","Indore")
print(stu.name)
print(stu.__dict__)

O/P:--
my name is Neeraj
my roll no is 101
my marks is 90
my city is Bhopal
Neeraj
my name is Rahul
my roll no is 102
my marks is 92
my city is Indore
Rahul
{'name': 'Rahul', 'roll': 102, 'marks': '92', 'city': 'Indore'}
```

```
# Instence Variable..........(By using object)
class Student:
    def __init__(self):
        print("This is constructor")
    def m1(self):
        print("This is instance method")
t=Student()
t.m1()
t.a=10
t.b=20
t.c=55
print(t.a)
print(t.b)
print(t.c)
print(t.__dict__)

O/P:--
This is constructor
This is instance method
10
20
55
{'a': 10, 'b': 20, 'c': 55}
```

Accessing instance variables
The instance variable can be accessed in two ways:

1. By using self variable
2. By using object name

By using self variable:--- We can access instance variables within the class by using self variable.

```
# Access instence variable...........(by using self reference variable)
class Student:
   def __init__(self):
       self.a=10
       self.b=20
   def display(self):
       print(self.a)
       print(self.b)
s= Student()
```

```
s.display()
O/P:---
10
20
```

By using object name :--- We can access instance variables outside of the class by using object name.

```
# Access instence variable...........(by using object name)
class Student:
    def __init__(self):
        self.a=10
        self.b=20

s= Student()
print(s.a)
print(s.b)

O/p:--
10
20
```

2. Static Variables in Python

If the value of a variable is not changing from object to object, such types of variables are called static variables or class level variables. We can access static variables either by class name or by object name. Accessing static variables with class names is highly recommended than object names.

```
# Static variable...........
class Student:
    ''' This class is develop by Neeraj for demo'''
    School_name="SHSC"
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

stu = Student("Neeraj",101,"90","Bhopal")
stu.display()
```

Declaring static variables in Python:
We can declare static variable in the following ways,

1. **Inside class and outside of the method**
2. **Inside constructor**
3. **Inside instance method**
4. **Inside class method (@classmethod):**
   We can declare and initialize static variable inside class method in two ways, one is using class name, other is using cls pre-defined variable.
5. **Inside static method (@staticmethod)**

```
# 1.  Declaring static variable  Inside class and outside of the method........
class Demo:
   a=20
   def m(self):
       print("this is method")
       print(Demo.a)
       print(obj.a)
obj = Demo()
print(Demo.a)
print(obj.a)
obj.m()

O/P:--
20
20
this is method
20
20

# 2.  Declaring static variable Inside constructor

class Demo:
    def __init__(self):
        print("this is constructor")
        Demo.a=20
    def m(self):
        print("This is method")
        print(Demo.a)
        print(obj.a)
print(Demo.__dict__)
obj = Demo()
print(Demo.a)
print(obj.a)
obj.m()

O/P:--
this is constructor
20
20
This is method
20
20
# 3. Declaring static variable inside instance method
class Demo:
    def m(self):
        Demo.a=20
        print("This is method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(obj.a)
print(Demo.a)

O/P:-
This is method
20
20
20
20

# 4(1). Declaring static variable inside class method
class Demo:
    @classmethod
    def m(cls):
        Demo.a=20
        print("This is class-method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(obj.a)
print(Demo.a)

O/P:--
This is class-method
20
20
20
20

# 4(2). Declaring static variable inside class method
class Demo:
    @classmethod
    def m(cls):
        cls.a=20
        print("This is class-method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(obj.a)
print(Demo.a)

O/P:--
This is class-method
20
20
20
20

# 5. Declaring static variable inside static method
class Demo:
    @staticmethod
    def m():
        Demo.a=20
        print("This is class-method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(Demo.__dict__)
print(obj.a)
print(Demo.a)

O/P:--
This is class-method
20
20
20
20
```

Accessing static variables:-
The **static** variable can be accessed in two ways:

1. By using class name (highly recommended).
2. By using object name.

Accessing static variables either outside or inside of the class by using either class name or object name.

```
class Student:
    ''' This class is develop by Neeraj for demo'''
    School_name="SHSC"
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)
        print(Student.School_name) # Access static variable by using class name
        print(stu.School_name) # Access static variable by using object name

stu = Student("Neeraj",101,"90","Bhopal")
stu.display()
print(stu.School_name)  # Access static variable by using object name
print(Student.School_name) # Access static variable by using class name

O/P:--
my name is Neeraj
my roll no is 101
my marks is 90
my city is Bhopal
SHSC
SHSC
SHSC
SHSC
```

3.Local Variables in Python:
The variable which we declare inside of the method is called a local variable. Generally, for temporary usage we create local variables to use within the methods. The scope of these variables is limited to the method in which they are declared. They are not accessible outside of the methods.

```
# Local variable...........

class Demo:
   def m(self):
       a=10 #Local Variable
       print(a)
   def n(self):
       print(a) #'a' is local variable of m() so it raise error
d=Demo()
d.m()
d.n()

O/P:--
print(a) #'a' is local variable of m()
```

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

If you want to access local variable outside of the block then use global keyword.

```
class Demo:
  def m(self):
      global a
      a=10 #Local Variable
      print("instence methos m()")
      print(a)
  def n(self):
      print("instence methos n()")
      print(a) #'a' is local variable of m()
d=Demo()
d.m()
d.n()
print("access local variable outside of the class")
print(a)

O/P:--
instence methos m()
10
instence methos n()
10
access local variable outside of the class
10
```

Types of Methods in a Class

In python, we can classify the methods into three types from the perspective of object oriented programming.

1. Instance Methods
2. Class Methods
3. Static Methods

Instance Methods in Python:

Instance methods are methods which act upon the instance variables of the class. They are bound with instances or objects, that"s why called as instance methods. The first parameter for instance methods should be self variable which refers to instance. Along with the self variable it can contain other variables as well.

```
# instence method

class Demo:
   def __init__(self, a):
       self.a=a
   def m(self):
       print(self.a)
d=Demo(10)
d.m()

O/P:-
10
```

Class Methods in Python:

Class methods are methods which act upon the class variables or static variables of the class. We can go for class methods when we are using only class variables (static variables) within the method.

1. Class methods should be declared with @classmethod.
2. Just as instance methods have ‘self’ as the default first variable, class method should have ‘cls’ as the first variable. Along with the cls variable it can contain other variables as well.
3. We can access class methods by using class name or object reference.

```
# class method
class Test:
    x=200
    @classmethod
    def get_radius(cls):
        return cls.x
obj=Test()
print("class methos access by using class name")
print(Test.get_radius())
print("class methos access by using object name")
print(obj.get_radius())

O/P:--
class methos access by using class name
200
class methos access by using object name
200
```

Static Methods in Python: The static methods, in general, utility methods. Inside these methods we won't use any instance or class variables. No arguments like cls or self are required at the time of declaration.

1. We can declare static method explicitly by using @staticmethod decorator.
2. We can access static methods by using class name or object reference.

```
# static method
class Demo:
  @staticmethod
  def sum(x, y):
      print(x+y)
  @staticmethod
  def multiply(x, y):
      print(x*y)

obj = Demo()
print("static methos access by using class name")
Demo.sum(2, 3)
Demo.multiply(2,4)
print("static methos access by using class name")
obj.sum(2, 3)
obj.multiply(2,4)

O/P:--
static methos access by using class name
5
8
static methos access by using class name
5
8
```

FEATURES OF OOPS:

1. Inheritance (Code reusability)
2. Polymorphism (Code reusability)
3. Encapsulation (Data security)
4. Abstraction (Data security)

Inheritance is the passing of properties to someone. In programming languages, the concept of inheritance comes with classes.

1. Creating new classes from already existing classes is called inheritance.
2. The existing class is called a super class or base class or parent class.
3. The new class is called a subclass or derived class or child class.
4. Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.

Advantages of Inheritance:

1. The main advantage of inheritance is code re-usability.
2. Time taken for application development will be less.
3. Redundancy (repetition) of the code can be reduced.

Types of Inheritance in Python:

There are three types of inheritance, they are:

1. **Single inheritance**
2. **Multilevel inheritance**
3. **Multiple inheritance**

```
# 1 Single inheritance.

class A:
    def m1(self):
        print("Method m1() is called")

class B(A):
    def m2(self):
        print("Method m2() is called")

obj = B()
obj.m1()
obj.m2()
O/P:--
Method m1() is called
Method m2() is called
```

```
# 2 Multilevel inheritance

class A:
    def m1(self):
        print("Method m1() is called")

class B(A):
    def m2(self):
        print("Method m2() is called")

class C(B):
    def m3(self):
        print("Method m3() is called")

obj = C()
obj.m1()
obj.m2()
obj.m3()
```

```
# 3 Multiple inheritance.
class A:
    def m1(self):
        print("Method m1() is called")

class B:
    def m2(self):
        print("Method m2() is called")

class C(A,B):
    def m3(self):
        print("Method m3() is called")

obj = C()
obj.m1()
obj.m2()
obj.m3()

O/P :---
Method m1() is called
Method m2() is called
Method m3() is called
```

Method Resolution Order (MRO):

In situations involving multiple inheritance, a particular attribute or method is initially searched in within the current class. If it is not located in the current class, the search proceeds to the parent classes following a depth-first, left-to-right fashion. This sequence of searching is referred to as the Method Resolution Order (MRO).

```
# Method Resolution Order (MRO).
class A:
    def m1(self):
        print("Method m1() is called from class A")

class B:
    def m1(self):
        print("Method m2() is called from class B")

class C(B,A):
    def m3(self):
        print("Method m3() is called")

obj = C()
obj.m1()
obj.m3()

O/P:--
Method m2() is called from class B
Method m3() is called
```

```
# Method Resolution Order (MRO).
class A:
    def m1(self):
        print("Method m1() is called from class A")
class B:
    def m1(self):
        print("Method m2() is called from class B")
class C(A,B):
    def m3(self):
        print("Method m3() is called")
```

```
obj = C()
obj.m1()
obj.m3()

O/P :--
Method m1() is called from class A
Method m3() is called
```

```
# Method Resolution Order (MRO).
class A:
   def m1(self):
       print("m1 from A")
class B(A):
   def m1(self):
       print("m1 from B")
class C(A):
   def m1(self):
       print("m1 from C")
class D(B, C):
   def m1(self):
       print("m1 from D")

print(A.mro())
print(B.mro())
print(C.mro())
print(D.mro())

O/P:--
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>]
```

Polymorphism:-----

The word ‘Poly’ means many and ‘Morphs’ means forms. The process of representing “one form in many forms” is called a polymorphism.

1. Duck Typing Philosophy of Python:--

“If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.”

```
class Duck:
   def talk(self):
       print("Quack.. Quack")
class Dog:
   def talk(self):
       print("Bow...Bow")
class Cat:
   def talk(self):
       print("Moew...Moew ")

def my_func(obj):
   obj.talk()

duck = Duck()
my_func(duck)

cat = Cat()
my_func(cat)

O/P:--
Quack.. Quack
Moew...Moew
```

In the above program, the function ‘m’ takes an object and calls for the talk() method of it. With duck typing, the function is not worried about what object type of object it is. The only thing that matters is whether the object has a method with name ‘talk()’ supported or not.

2. Overloading:-

  1. Operator Overloading
  2. Method Overloading
  3. Constructor Overloading

Operator Overloading:

```
print(10+20)
print("Neeraj"+" "+"Kumar")

print(10*5)
print("neeraj"*5)

O/P:-
30
Neeraj Kumar
50
neerajneerajneerajneerajneeraj
```

```
class Book:
   def __init__(self, pages):
       self.pages=pages
obj1=Book(100)
obj2=Book(200)
print(type(obj1))
print(type(obj2))
print(type(obj1.pages))
print(type(obj2.pages))
print(obj1.pages + obj2.pages)
print((obj1.pages).__add__(obj2.pages))

O/P:--
<class '__main__.Book'>
<class '__main__.Book'>
<class 'int'>
<class 'int'>
300
300
class A:
    def __init__(self,x):
        self.x = x
    def __add__(self,other):
        return self.x+other.x
class B:
    def __init__(self,x):
        self.x = x


a = A(10)
b = B(20)
print(a+b) # a.__add__(10+20)

O/P:-
30

class A:
    def __init__(self,x):
        self.x = x
    def __add__(self,other):
        return self.x+other.x
class B:
    def __init__(self,x):
        self.x = x


a = A(10)
b = B(20)
print(b+a) # b.__add__(10+20) not define __add__ method in B class so it gives
error.

O/P:-
TypeError: unsupported operand type(s) for +: 'B' and 'A'
```

Method Overloading
If 2 methods have the same name but different types of arguments, then those methods are said to be overloaded methods. If we are trying to declare multiple methods with the same name and different number of arguments, then Python will always consider only the method which was last declared. **But in Python Method overloading is not possible**. If we are trying to declare multiple methods with the

same name and different number of arguments, then Python will always consider only the last method.

3. Overriding:-

1. Method overriding

2. Constructor overloading

Overriding

All the members available in the parent class, those are by-default available to the child class through inheritance. If the child class is not satisfied with parent class implementation, then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.Overriding concept applicable for both methods and constructors.

1. **Method Overriding**
2. **Constructor Overriding**

Method Overriding:-

```
class A:
   def display(self):
      print('Display fron class A')
   def show(self):
      print('Show fron class A')
class B(A):
   def display1(self):
      print("Display fron class B")
   def show(self):
      print('Show fron class B')

c=B()
c.display1()
c.show()
c.display()
```

Constructor Overriding

1. If child class does not have constructor, then parent class constructor will be executed at the time of child class object creation.

2. If child class has a constructor, then child class constructor will be executed at the time of child class object creation.

3. From child class constructor we can call parent class constructor by using super() method

```
class Person:
  def __init__(self, name, age):
     self.name=name
     self.age=age
class Employee(Person):
  def __init__(self, name, age, eno, esal):
     super().__init__(name, age)
     self.eno=eno
     self.esal=esal
  def display(self):
     print('Employee Name:', self.name)
     print('Employee Age:', self.age)
     print('Employee Number:', self.eno)
     print('Employee Salary:', self.esal)
obj1=Employee('Neeraj', 36, 101, 26000)
obj1.display()
obj2=Employee('Rahul',37,102,36000)
obj2.display()

O/P:--
Employee Name: Neeraj
Employee Age: 36
Employee Number: 101
Employee Salary: 26000
Employee Name: Rahul
Employee Age: 37
Employee Number: 102
Employee Salary: 36000
```

Encapsulation

Encapsulation is the concept of wrapping data and methods that work with data in one unit. Encapsulation is achieved through the use of access modifiers such as 'public', 'private', and 'protected'.

```
class parent:
    def __init__(self):
        self._p = 78

class child(parent):
    def __init__(self):
        parent.__init__(self)
        print ("We will call the protected member of base class: ", self._p)
        self._p = 433
        print ("we will call the modified protected member outside the class: ",self._p)


obj_1 = parent()
obj_2 = child()
print ("Access the protected member of obj_1: ", obj_1._p)
print ("Access the protected member of obj_2: ", obj_2._p)

O/P:-
We will call the protected member of base class:  78
we will call the modified protected member outside the class:  433
Access the protected member of obj_1:  78
Access the protected member of obj_2:  433
```

Access Modifier/Specifire :-

1. **Public:-**
2. **Protected**(not supported by python):-
3. **Private:-**

```
# protected variable------------
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number  # Public attribute
        self._balance = balance               # Protected attribute

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited ${amount}. New balance: ${self._balance}")
        else:
            print("Invalid deposit amount!")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f" Withdrew ${amount}. New balance: ${self._balance}")
        else:
            print("Invalid or insufficient funds!")

    def get_balance(self):  # Getter for _balance
        return self._balance

account = BankAccount("12345", 1000)
print(f"Account Number: {account.account_number}")
print(f"Balance (via getter): {account.get_balance()}")
print(f"Directly accessing balance: {account._balance}")
account.deposit(500)
account.withdraw(300)

O/P:---
Account Number: 12345
Balance (via getter): 1000
Directly accessing balance: 1000
Deposited $500. New balance: $1500
Withdrew $300. New balance: $1200
```

```
# private variable------------
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance                # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid deposit amount!")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid or insufficient funds!")

    def get_balance(self):  # Getter for __balance
        return self.__balance

account = BankAccount("12345", 1000)
print(account.__balance)

O/P:--
Traceback (most recent call last):
  File "c:\Users\neera\Desktop\DADS-30\new.py", line 56, in <module>
    print(account.__balance)
AttributeError: 'BankAccount' object has no attribute '__balance'. Did you mean:
'get_balance'?
```

```
# private variable------------
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance                # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid deposit amount!")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid or insufficient funds!")

    def get_balance(self):  # Getter for __balance
        return self.__balance

account = BankAccount("12345", 1000)
print(account._BankAccount__balance)

O/P:--
1000
```

Abstraction :---

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.

Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user. This is one of the core concepts of object-oriented programming (OOP).

Important terminologies:--

1. **Abstract class**
2. **Abstract methods**
3. **Concrete methods**

Abstract class:-- An abstract class is the class which contains one or more abstract methods. An abstract method is the one which is just defined but not implemented.

1. Every abstract class in Python should be derived from the ABC class which is present in the abc module. Abstract class can contain Constructors, Variables, abstract methods, non-abstract methods, and Subclass.
2. Abstract methods should be implemented in the subclass or child class of the abstract class.
3. If in subclass the implementation of the abstract method is not provided, then that subclass, automatically, will become an abstract class.
4. Then, if any class is inheriting this subclass, then that subclass should provide the implementation for abstract methods.
5. Object creation is not possible for abstract class.
6. We can create objects for child classes of abstract classes to access implemented methods.

Abstract methods:-- A method which has only method name and no method body, that method is called an unimplemented method. They are also called as non-concrete or abstract methods.

1. By using @abstractmethod decorator we can declare a method as an abstract method.
2. @abstractmethod decorator presents in abc module. We should import the abc module in order to use the decorator.
3. Since abstract method is an unimplemented method, we need to put a pass statement, else it will result in error.
4. Class which contains abstract methods is called an abstract class.
5. For abstract methods, implementation must be provided in the subclass of abstract class

Here, I am changing the name of abc lib to my_abc and I am doing some changes in that particular library.

my_abc:-

(Link for abc libreary:-- https://github.com/python/cpython/blob/3.12/Lib/abc.py)

```
from my_abc import ABC , abstractmethod

class Fruit(ABC):

   @abstractmethod
   def fruit_shape(self):
      pass
```

```
class Mango(Fruit):
    description = "King of fruits"
    def __init__(self, x,y,z):
        self.desc=Mango.description
        self.shape = x
        self.taste = y
        self.color = z

    def fruit_shape(self):
        print(self.desc)
        print(self.shape)


    def fruit_color(self):
        print(self.color)

    def fruit_taste(self):
        print(self.taste)

# obj = Fruit() # You can not create object for any abstract class.
obj1 = Mango("Oval","yellow","sweet")
obj1.fruit_shape()
obj1.fruit_color()
obj1.fruit_taste()

O/P:--
Welcome to home screen
King of fruits
Oval
sweet
yellow
```

Concrete methods:- A method which has a both method name and method body, that method is called an implemented method. They are also called concrete methods or non-abstract methods.