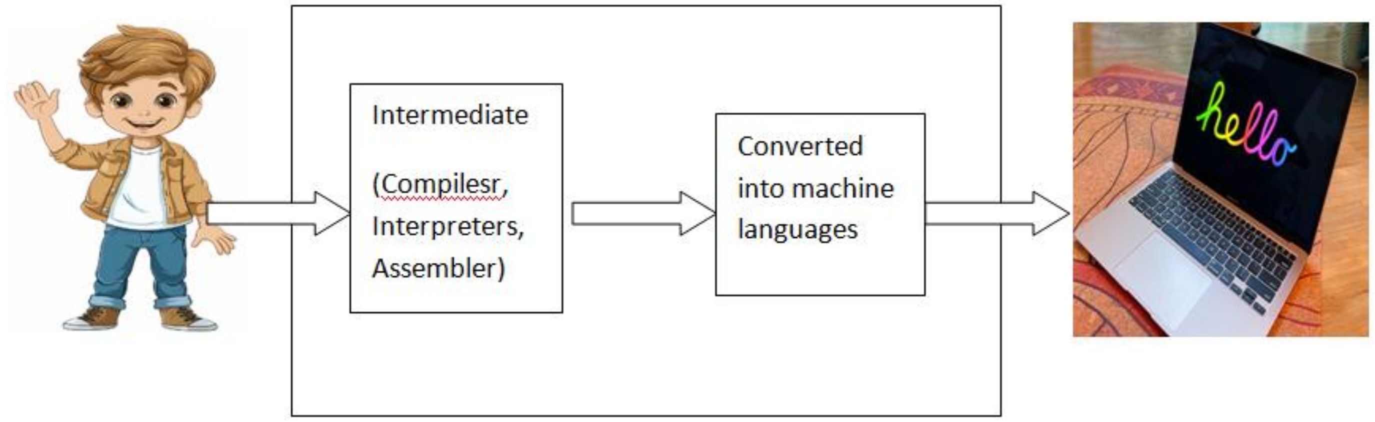
Language

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 ()?

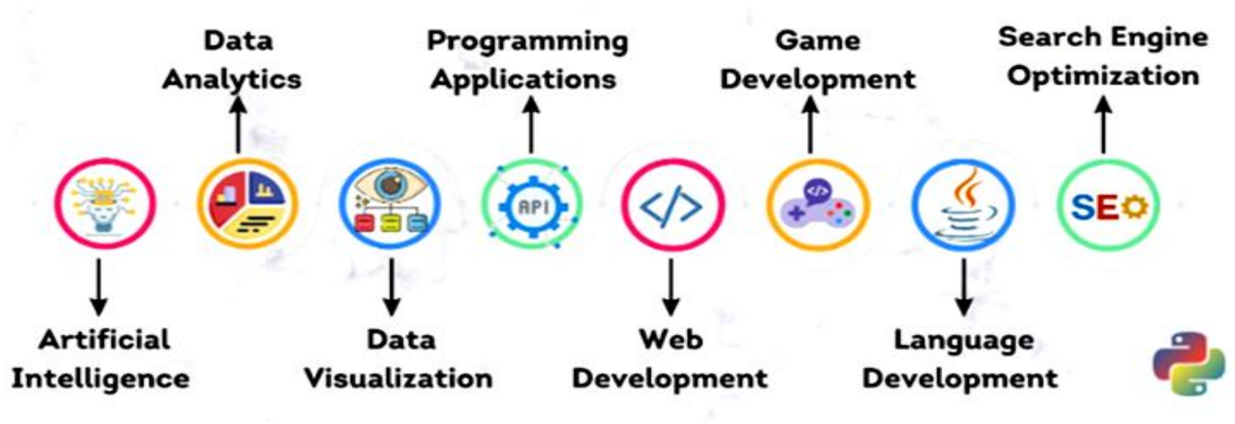

Intermediate
(Compilesr, Interpreters, Assembler)
Converted into machine languages
hello

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

<u>Syntex:----</u>

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. **Done Not have 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

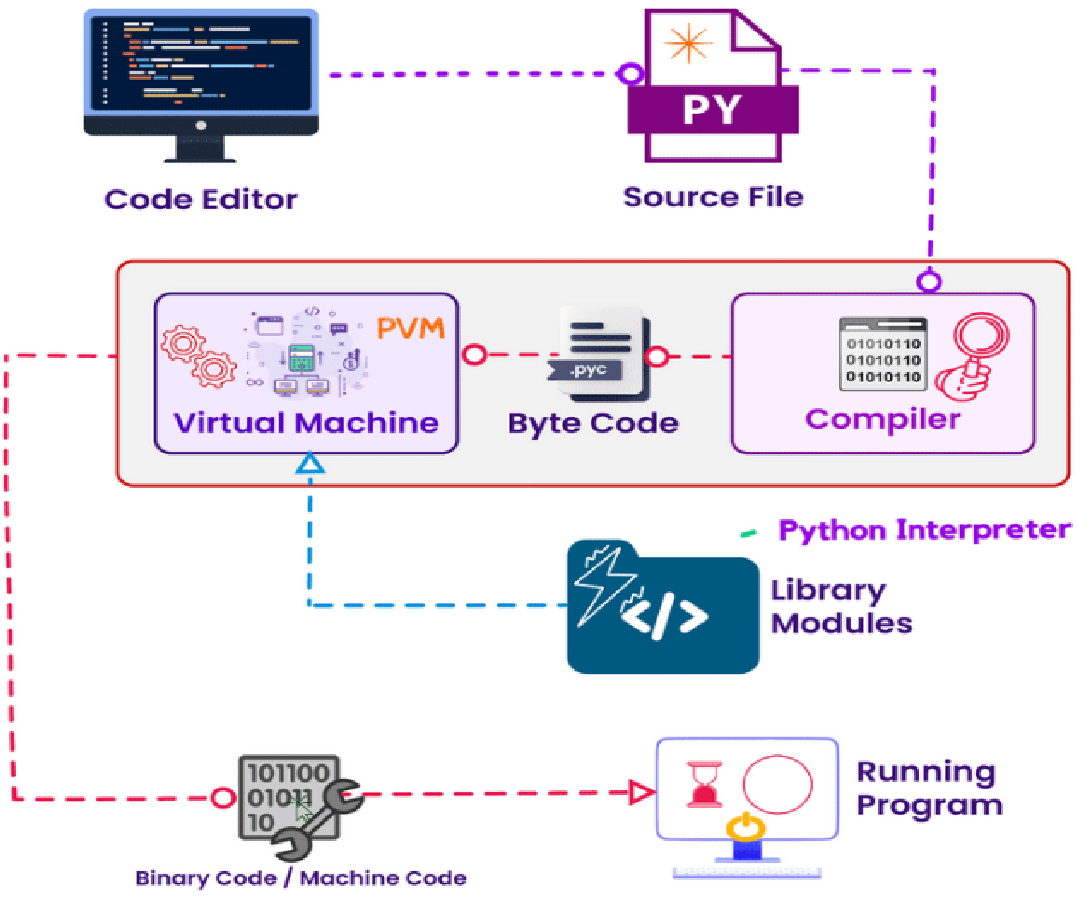
3. IronPython:
It is for C#.Net platform

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

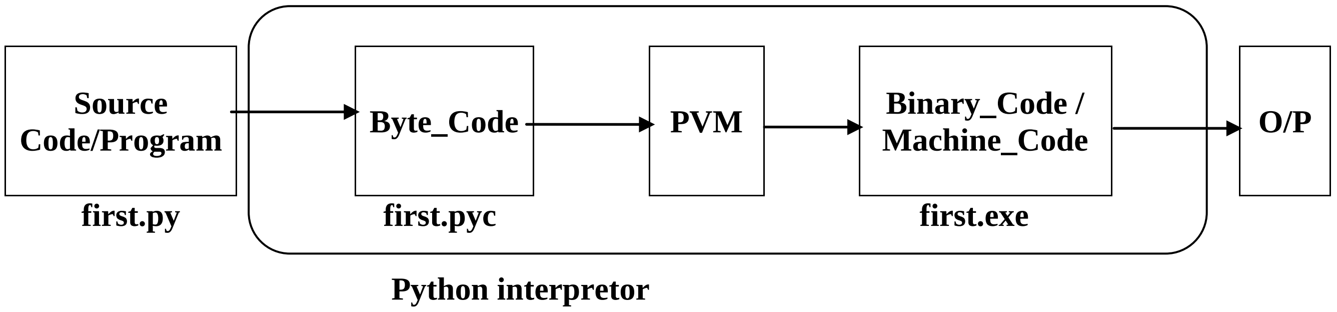
6. AnacondaPython
It is specially designed for handling large volume of data processing.

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

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

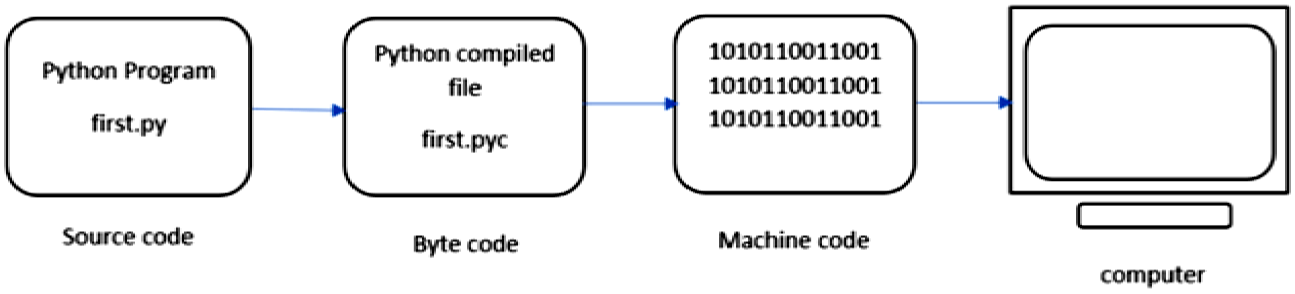
The program is converted into **byte code.** Byte code is a fixed set of instructions that represent arithmetic, comparison, memory operations, etc. It can run on any operating system and hardware. The byte code instructions are created in the **.pyc** file. The .pyc file is not explicitly created as 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
```

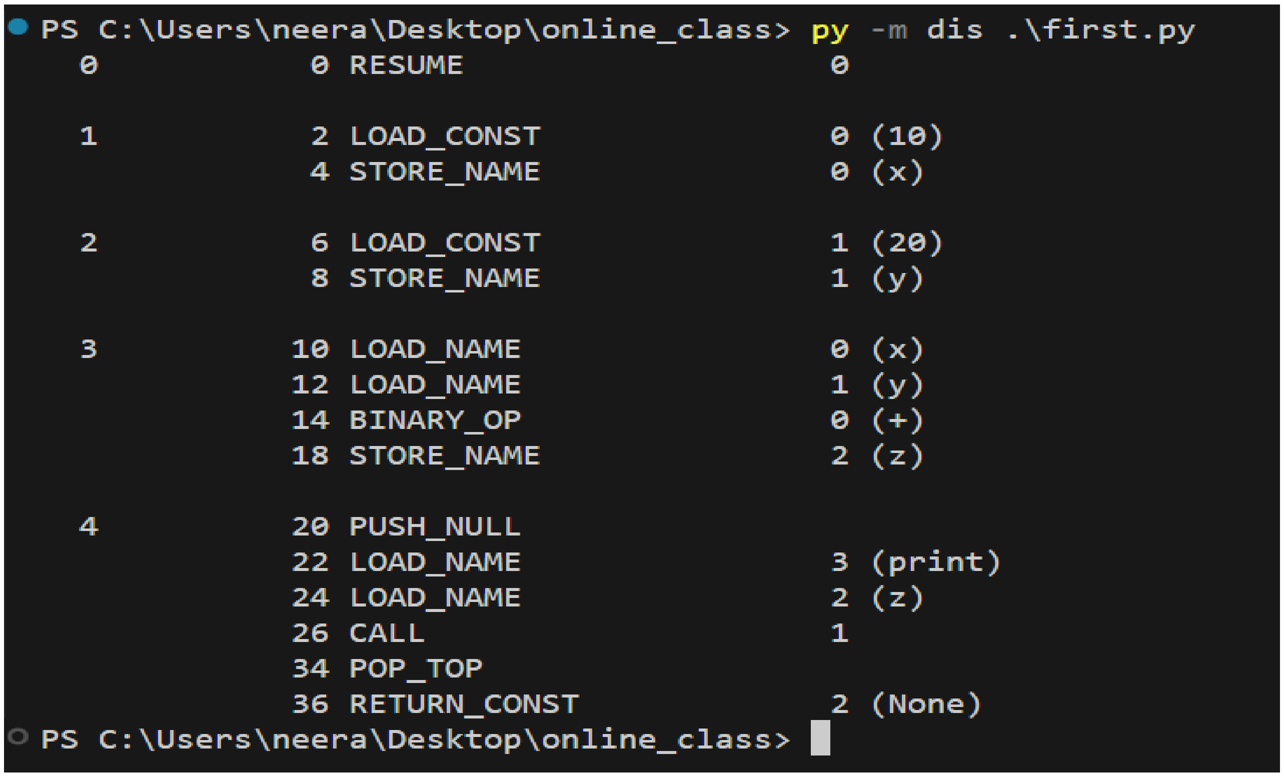
view the byte code of the file – first.py we can type the following command as : first.py:

```
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> 
```

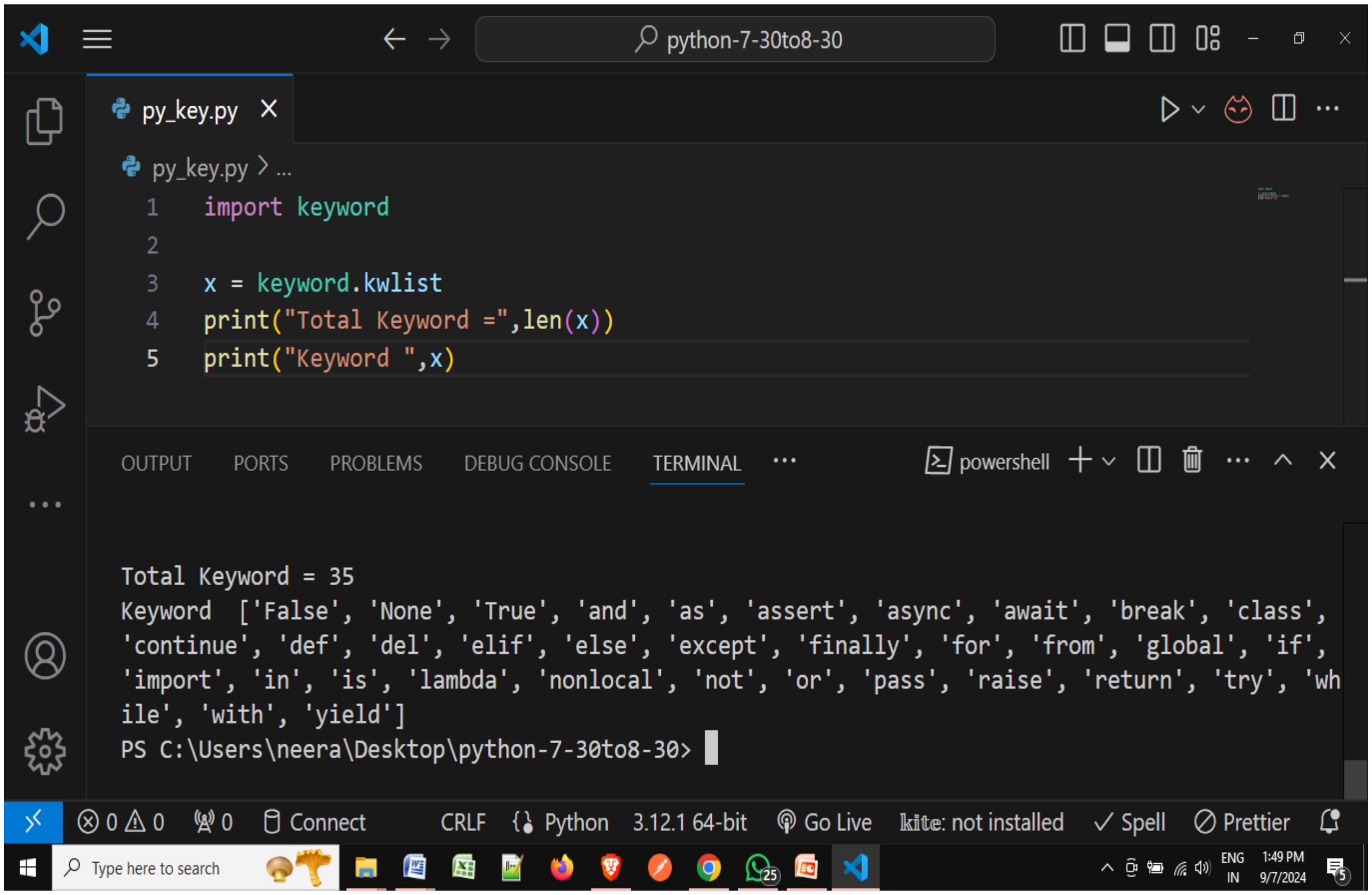
- **LOAD_CONST**: Loads a constant value (like numbers 10 and 20).
- **STORE_NAME**: Stores the value in a variable (like x, y, or z).
- **LOAD_NAME**: Loads the value of a variable from memory.
- **BINARY_ADD**: Adds two values (in this case, the values of x and y).
- **CALL_FUNCTION**: Calls a function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

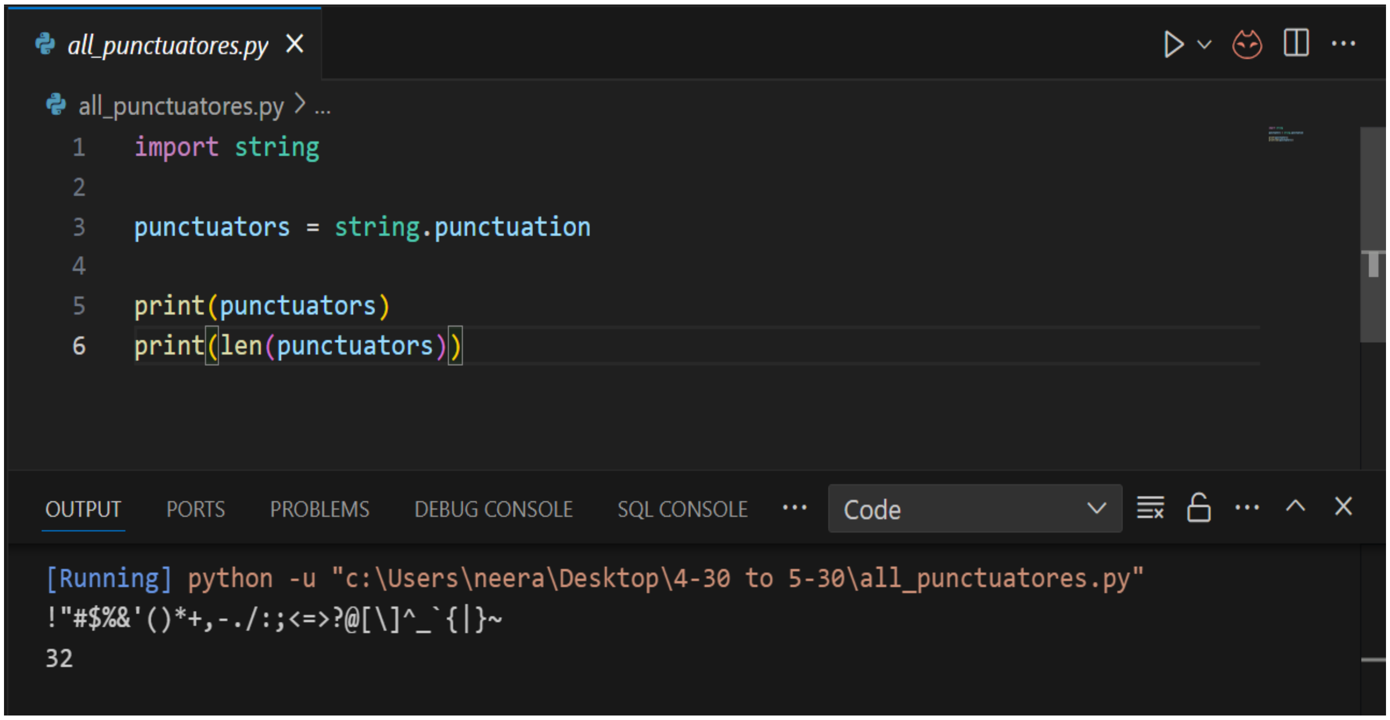
Keywords:



```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

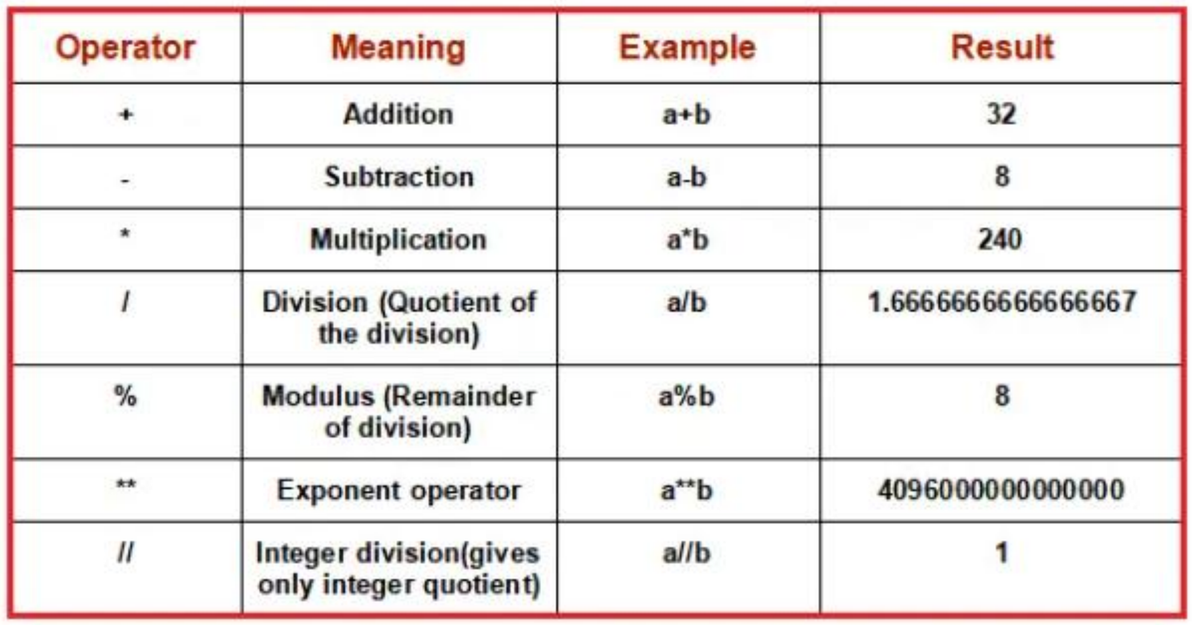
1. **Start with a Letter or Underscore**: An identifier must begin with a letter (a-z, A-Z) or an underscore (_). It cannot start with a digit.
2. **Subsequent Characters**: The characters following the initial letter or underscore can be letters, digits, or underscores.
3. **Case Sensitivity**: Identifiers 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.

Operator

ARITHMETIC OPERATORs:

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.

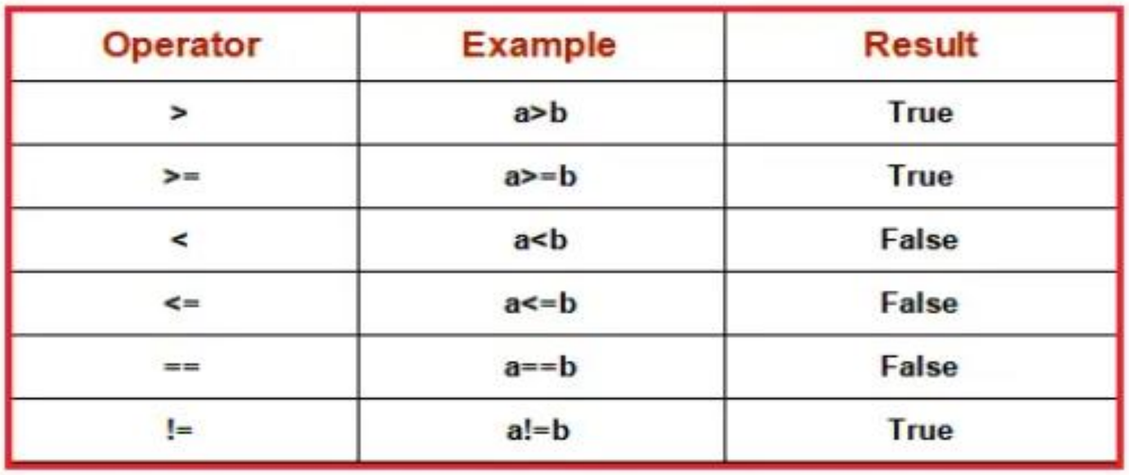
Example: Arithmetic Operators in Python:

```
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, 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:

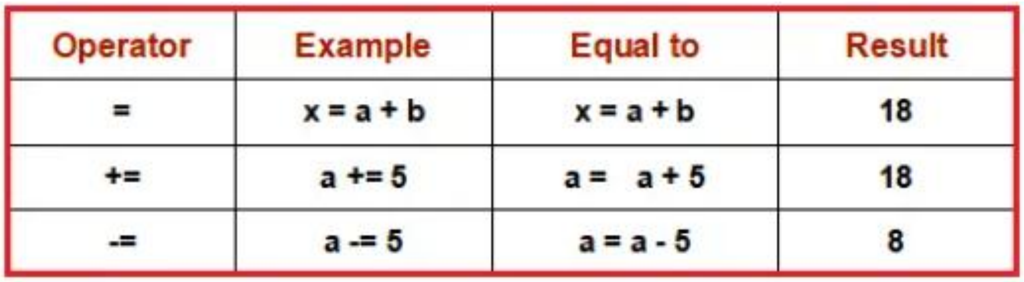
'A or B' returns A if A is True
'A or B' returns B if A is not True

Not Operator in Python:

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

ASSIGNMENT OPERATORS:

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
```

MEMBERSHIP OPERATORS:----

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

```
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

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

O/P:-
True
False
True
```

IDENTITY OPERATORS:--

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

```
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

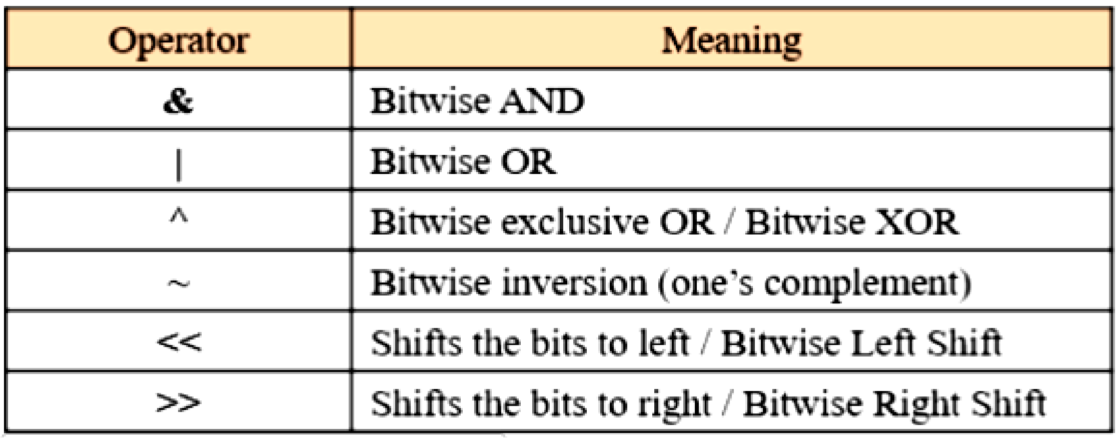
```
a = 25
b = 25
print(a is b)
print(id(a))
print(id(b))

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 ‘==’.

Bitwise wise operator :-



Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR / Bitwise XOR
~	Bitwise inversion (one’s complement)
<<	Shifts the bits to left / Bitwise Left Shift
>>	Shifts the bits to right / Bitwise Right Shift

-------------	4	3	2	1	0	**Bit position**
	2^4	2^3	2^2	2^1	2^0	**Bit weight**
	Binary number					
Decimal number	16	8	4	2	1	
0	0	0	0	0	0	
1	0	0	0	0	1	
2	0	0	0	1	0	
3	0	0	0	1	1	
4	0	0	1	0	0	

5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
A	0	1	0	1	0
B	0	1	0	1	1
C	0	1	1	0	0
D	0	1	1	0	1
E	0	1	1	1	0
F	0	1	1	1	1

Bit-wise and(&):-----

x = 10
y = 20
print(x & y)

```
        x = 10      0    1    0    1    0
                    &    &    &    &    &
        y = 20      1    0    1    0    0
        --------------------------
        o/p=0       0    0    0    0    0
```

Bit-wise or(|):-----

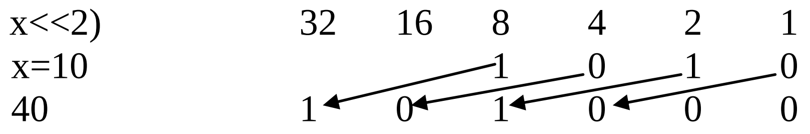
x = 10
y = 20
print(x | y)

```
        x = 10      0    1    0    1    0
                    |    |    |    |    |
        y = 20      1    0    1    0    0
      ---------------------------------------------
        o/p=30      1    1    1    1    0
```

Left Shift :--

x =10

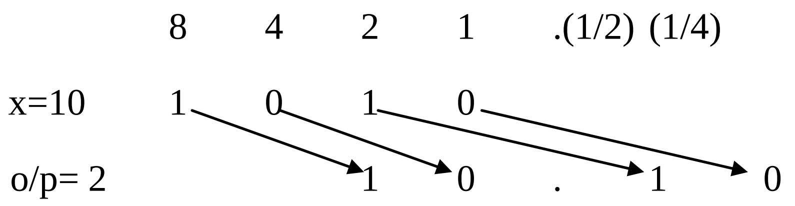
Print(x<<2)



Right Shift :--

x =10

Print(x>>2)



-----: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**:
   Integer Literals: Whole numbers, which can be written in decimal, binary (0b...), octal (0o...), or hexadecimal (0x...) form.
   Float Literals: Numbers with a decimal point or in exponential (scientific) notation.
   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.

Numeric:------

Integer:---

```
my_int1 = 10
my_int2 = 10
print(id(my_int1),id(my_int2))
            |            |
            --------------
                  |
 (140707225601224 140707225601224)
       Same memory address
                  |
```

That means immutable object

Float:----

```
my_float1 = 10.5
my_float2 = 10.5

print(id(my_float1),id(my_float2))
                |               |
                 ---------------
                        |
  (1740399292944 1740399292944)
        (Same memory address)
                        |
```

Immutable object

Complex:---

```
my_comp1 = 10.5+3j
my_comp2 = 10.5+3j

print(id(my_comp1),id(my_comp2))
                |               |
                |               |
                 ---------------
                        |
  (3039707779024 3039707779024)
         (Same memory address)
                        |
```

Immutable object

<u>**String:---**</u>

```
my_str1 = 'Neeraj'
my_str2 = 'Neeraj'
print(id(my_str1),id(my_str2))
                |               |
                 ---------------
                        |
        Same memory address
 (2421953832800 2421953832800)
                        |
```

That means immutable object

List:----

```
my_list1 = ['Neeraj','jai']
my_list2 = ['Neeraj','jai']
print(id(my_list1),id(my_list2))
               |            |
               --------------
                     |
  (2070751895616 2070752043520)
       Different memory address
                     |
```

That means mutable object

Tuple:---

```
my_tup1 = ('Neeraj','jai')
my_tup2 = ('Neeraj','jai')

print(id(my_tup1),id(my_tup2))
               |            |
               --------------
                     |
  (1607757822976 1607757822976)
        Same memory address
                     |
```

That means immutable object

Dictionary:---

```
my_dict1 = {'name':'Neeraj','age':37}
my_dict2 = {'name':'Neeraj','age':37}

print(id(my_dict1),id(my_dict2))
               |            |
               --------------
                     |
  (2084796816704 2084797210368)
      Different memory address
                     |
```

That means mutable object

Set: ---

```
my_set1 = {'name','Neeraj','age',37}
my_set2 = {'name','Neeraj','age',37}

print(id(my_set1),id(my_set2))
             |              |
             ---------------
                    |
  (2485072560864 2485072845888)
       Different memory address
                    |
      That means mutable object
```

Frozenset:---

```
my_fset1 =frozenset({'name','Neeraj','age',37})
my_fset2 = frozenset({'name','Neeraj','age',37})

print(id(my_fset1),id(my_fset2))
             |              |
             ---------------
                    |
   (2485072560864 2485072845888)
Different memory address due to unordered collection
                    |
          Immutable object
```

Boolean:----

```
my_bool1 = True
my_bool2 = True

print(id(my_bool1),id(my_bool2))
             |              |
             ---------------
                    |
(140707224715696 140707224715696)
        (Same memory address)
                    |
           Immutable object
```

Python Objects

Mutable
1. list
2. dictinory
3. set

Immutable
1. numeric
2. tuple
3. string
4. frozenset
5. Boolean

Python Data Types Memory required

Rank	Data Type	Example	Notes
1	bool	True / False	Smallest size, just 1-bit conceptually (but actually ~28 bytes in CPython)
2	int (small)	0 to 256	Interned, small int optimization
3	int (large)	1000000000	Size grows with value size (unlike C/C++)
4	float	3.14	Typically 24 bytes
5	complex	1+2j	Stores two floats
6	str	"abc"	More characters = more memory, plus overhead
7	tuple	(1, 2, 3)	Immutable, a bit more efficient than lists
8	list	[1, 2, 3]	Dynamic, more memory due to extra flexibility
9	set / frozenset	{1, 2, 3}	Uses hash table internally
10	dict	{'a': 1}	Hash maps take more space per item

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 objects – call by object 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:---(‘‘‘ ------------
   -----------’’’)

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
```