

Language

Basics of languages

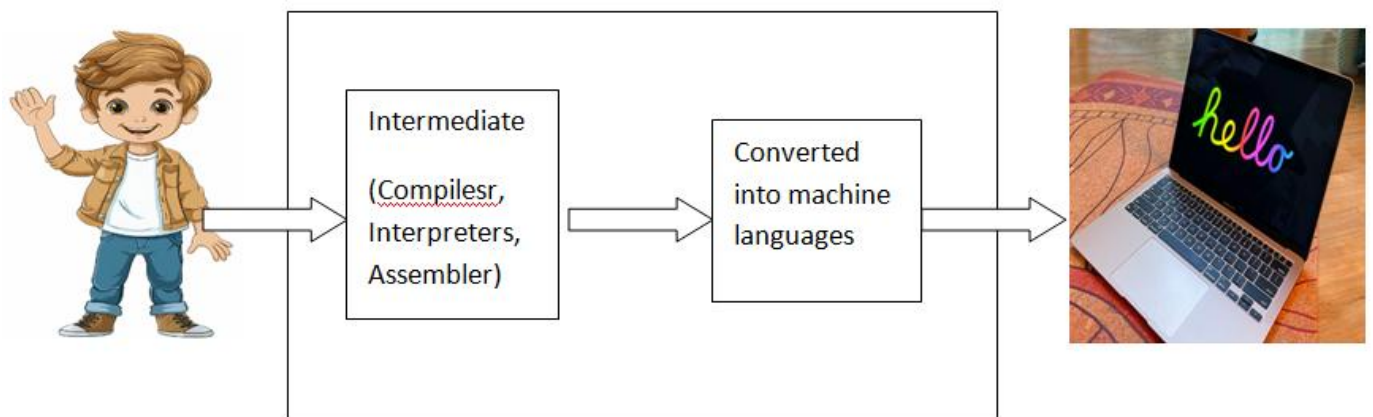
What is language ?

Why it is required ?

Type of languages ?

What is Low level languages (Machine & assembly language)?

What is high level languages ()?

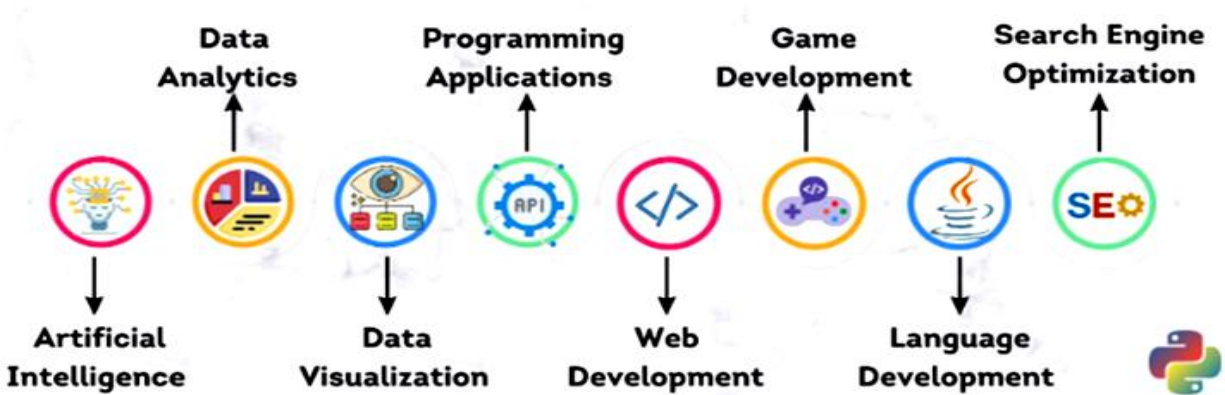


Python Introduction

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

Syntax:----

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 interpreters:

1. CPython:

It is the standard flavor of Python. It can be used to work with C language 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.

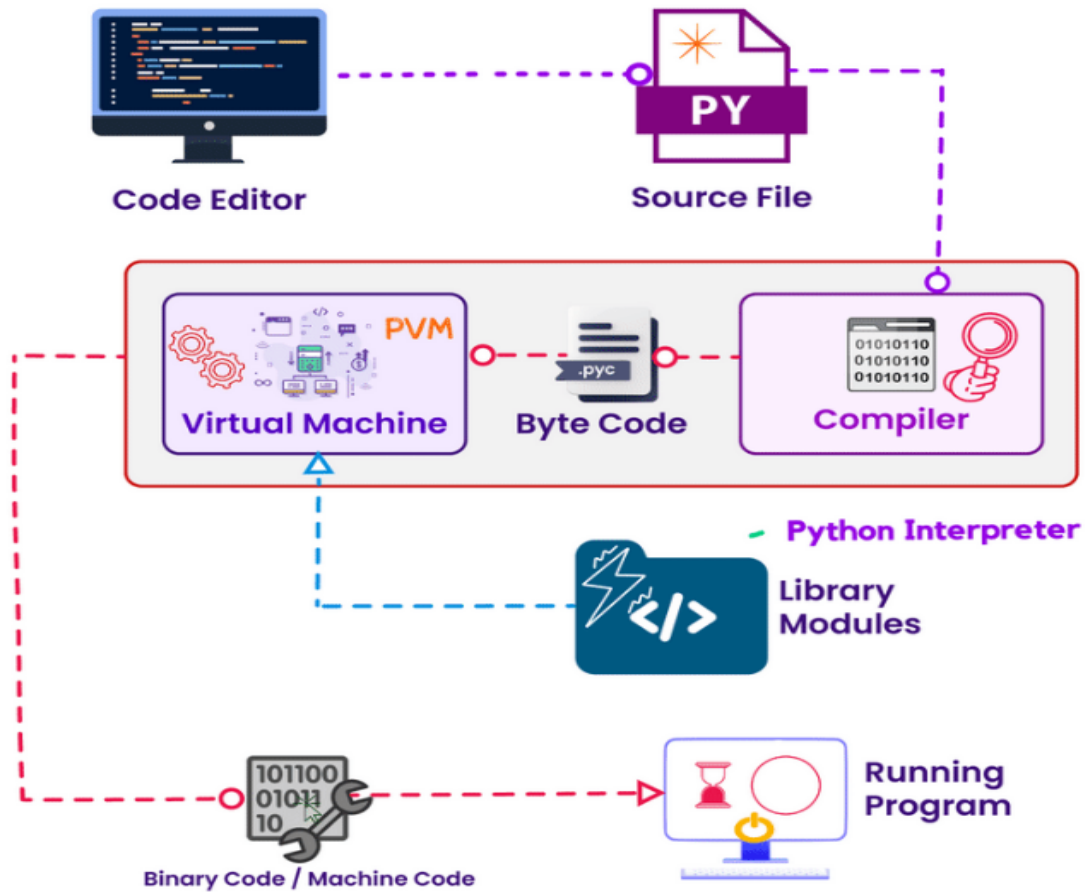
5. RubyPython

For Ruby Platforms

6. AnacondaPython

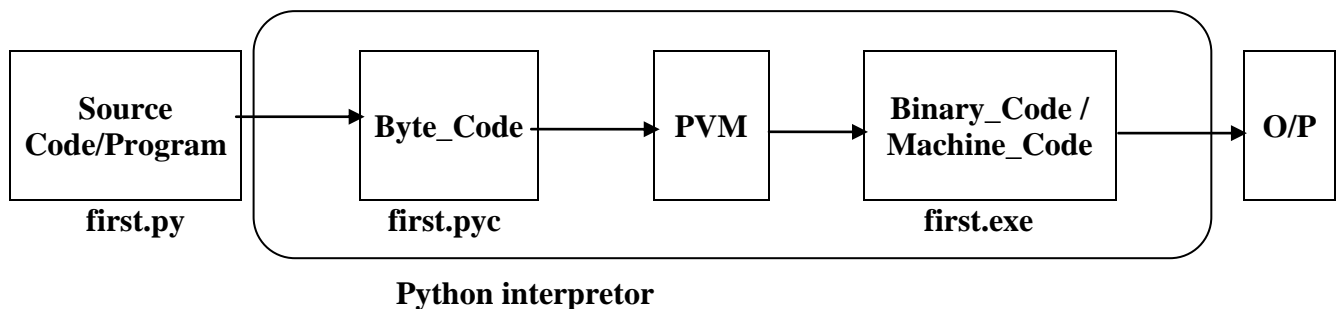
It is specially designed for handling large volume of data processing.

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

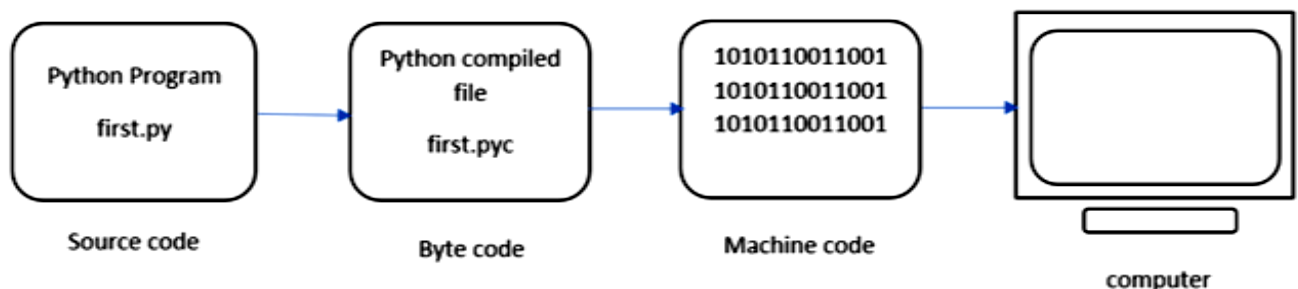
The program is converted into **byte code**. Byte code is a fixed set of instructions that represent arithmetic, comparison, memory operations, etc. It can run on any operating system and hardware. The byte code instructions are created in the **.pyc** file. The .pyc file is not explicitly created as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

```

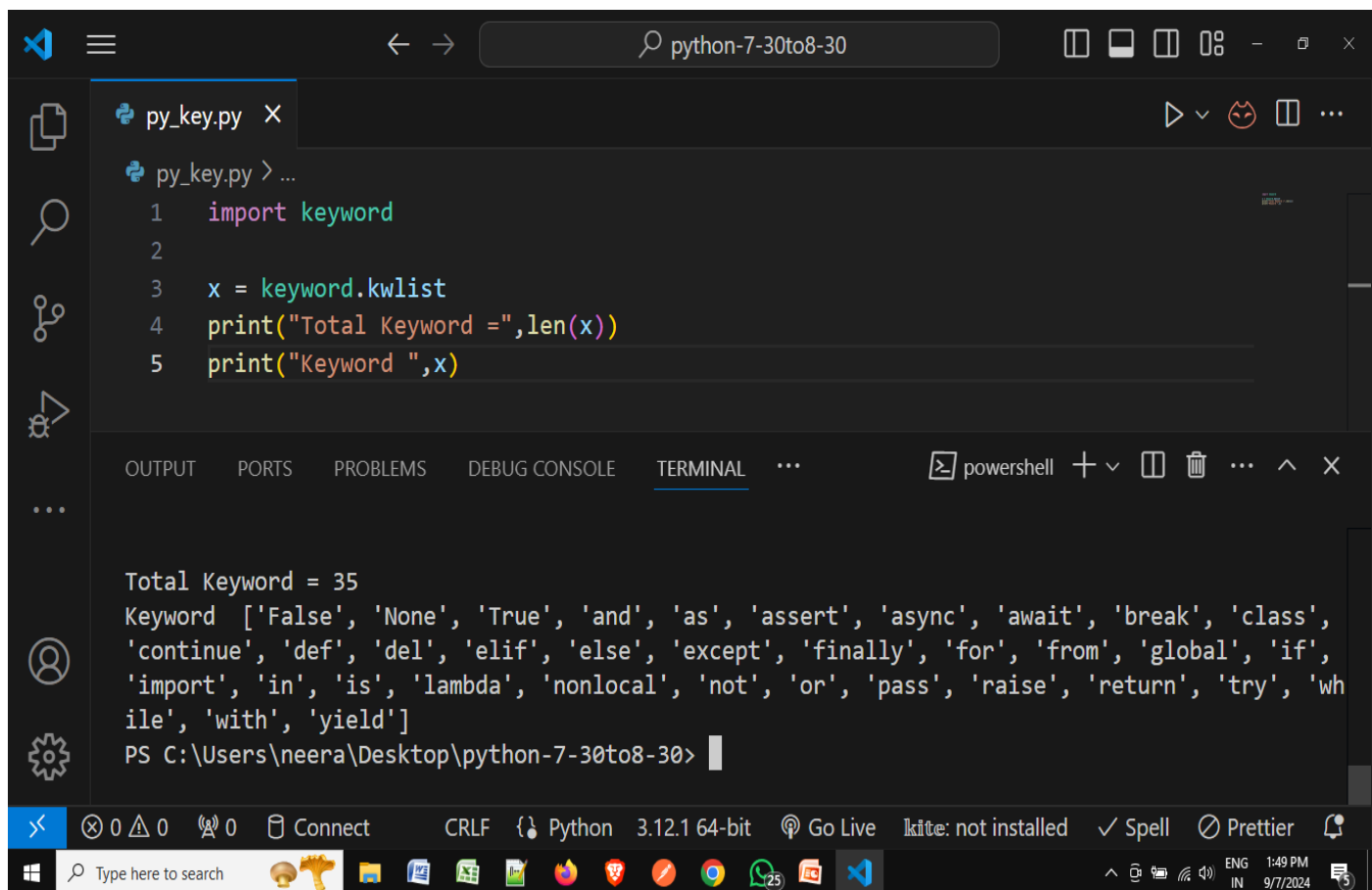
- **LOAD_CONST:** Loads a constant value (like numbers 10 and 20).
- **STORE_NAME:** Stores the value in a variable (like x, y, or z).
- **LOAD_NAME:** Loads the value of a variable from memory.
- **BINARY_ADD:** Adds two values (in this case, the values of x and y).
- **CALL_FUNCTION:** Calls a function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



The screenshot shows a Visual Studio Code editor window with a file named `py_key.py` open. The code in the editor is as follows:

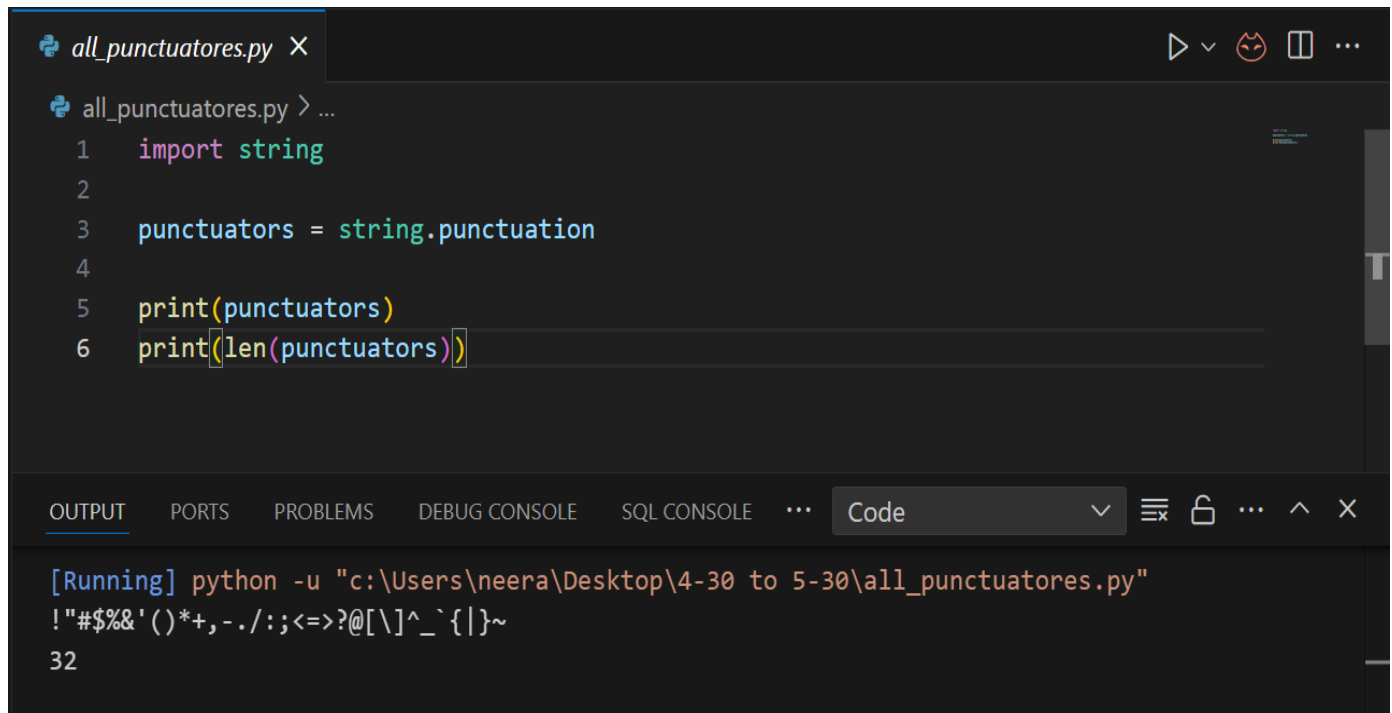
```
1 import keyword
2
3 x = keyword.kwlist
4 print("Total Keyword =", len(x))
5 print("Keyword ", x)
```

Below the editor, the **TERMINAL** panel is active, showing the output of running the script. The output is:

```
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>
```

The status bar at the bottom of the editor indicates the current configuration: Python 3.12.1 64-bit, with various extensions like Go Live, Kite, Spell, and Prettier.

Punctuations:-



```
all_punctuatores.py X
all_punctuatores.py > ...
1 import string
2
3 punctuators = string.punctuation
4
5 print(punctuators)
6 print(len(punctuators))
```

OUTPUT PORTS PROBLEMS DEBUG CONSOLE SQL CONSOLE ... Code

```
[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

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

String:---

```
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))
```

1

1

(2485072560864 2485072845888)

Different memory address

1

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))
```

1

1

(2485072560864 2485072845888)

Different memory address due to unordered collection

1

Immutable object

Boolean:---

```
my_bool1 = True
```

```
my_bool2 = True
```

```
print(id(my_bool1),id(my_bool2))
```

1

1

(140707224715696 140707224715696)

(Same memory address)

1

Immutable object

Python Objects

Mutable

1. list
2. dictionary
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)	10000000000	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:

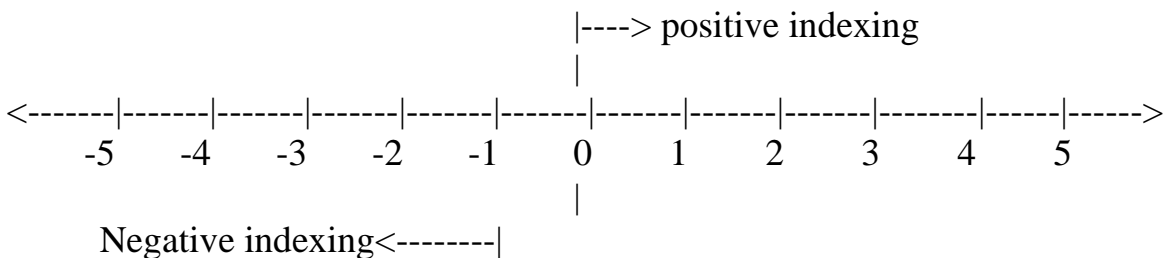
```
Enter expression: 12-2  
10
```


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

Index is a stored position of an object from ordered collections like string,list,tuple.

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

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]`



=> -ve indexing start with -1

=> +ve index start with 0

=> -ve indexing read from R to L

=> +ve index read from L to R

=> -ve indexing write from L to R

=> +ve indexing write from L to R

=> +ve indexing stop point in (stop+1)

=> +ve indexing stop point in (stop-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:-

collection.index(element, start, stop)

collection.index(element)

collection.index(element, start)

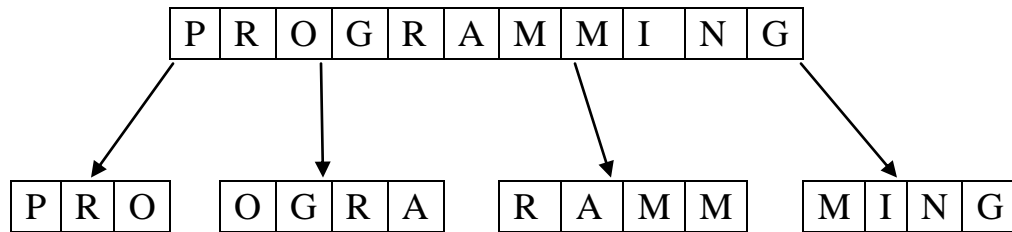
Example:- index() provides you an option to give it hints to where the value searched for might lie.

```
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/direction]

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:-
nh y vll
```

-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:-
```

-----: 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]
```


---: 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.

1. Numeric
 1. Integer
 2. Complex
 3. Float
2. Mapped – Dictionary
3. Ordered
 1. String
 2. List
 3. Tuple
4. Unordered
 1. Set
 2. Frozenset
5. Boolean

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 (**by default**)
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 $2 * 10^2$ 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 0 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 bytearray().
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])
```

----: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 ASCII 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
HloWrld
```

```
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 lower():', str1.lower())
```

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

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

O/P:--

```
converted to using lower(): python programming language  
converted to using lower (): java programming language  
converted to using lower (): 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:

6.**len(list)**

7.**max(list)** - homogeneous collection required

8.**min(list)** - homogeneous collection required

9.**sum(list)** - integer homogeneous collection required

10.**list(tuple)**

11.**type(list)**

12.**id()**

13.**list()**

Methods:--

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 by default 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 immutable. 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:--

```
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'}
```

MATHEMATICAL OPERATIONS ON SETS

1. Union



$A = \{1, 2, 3, 4, 5, 6, 7\}$
 $B = \{5, 6, 7, 8, 9, 10\}$
`print(A.union(B))`
O/P:--
 $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

2 Intersection



$A = \{1, 2, 3, 4, 5, 6, 7\}$
 $B = \{5, 6, 7, 8, 9, 10\}$
`print(A.intersection(B))`
O/P:--
 $\{5, 6, 7\}$

3. Difference



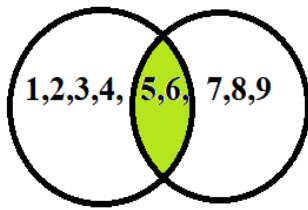
$A = \{1, 2, 3, 4, 5, 6, 7\}$
 $B = \{5, 6, 7, 8, 9, 10\}$
`print(A.difference(B))`
O/P:--
 $\{1, 2, 3, 4\}$

4. Symmetric_difference



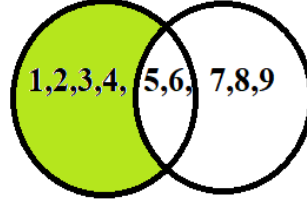
$A = \{1, 2, 3, 4, 5, 6, 7\}$
 $B = \{5, 6, 7, 8, 9, 10\}$
`print(A.symmetric_difference)`
O/P:-- $\{1, 2, 3, 4, 8, 9, 10\}$

5. Intersection_update



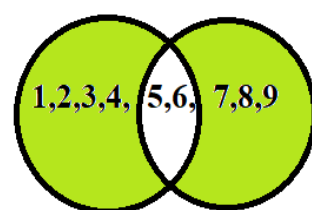
A = {1,2,3,4,5,6}
B = {5,6,7,8,9}
A.intersection_update(B)
print(A)
O/P:-- {5,6}

6. difference_update



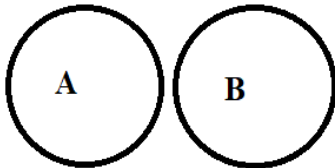
A = {1,2,3,4,5,6}
B = {5,6,7,8,9}
A.difference_update(B)
print(A)
O/P:--
{1,2,3,4}

7. symmetric_difference_update



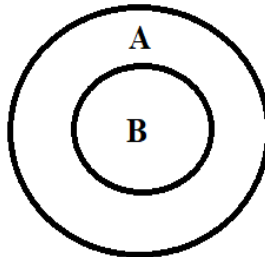
A = {1,2,3,4,5,6}
B = {5,6,7,8,9}
A.symmetric_difference_update(B)
print(A)
O/P:--
{1,2,3,4,7,8,9}

8. isdisjoint



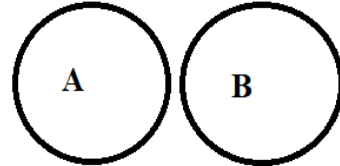
print(A.isdisjoint(B))
O/P:-
True

9. issuperset

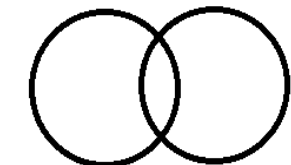


print(A.issuperset(B))
O/P:-- True

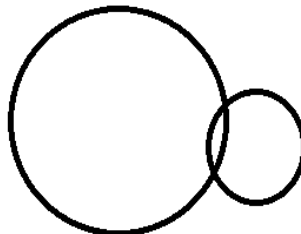
10. issubset



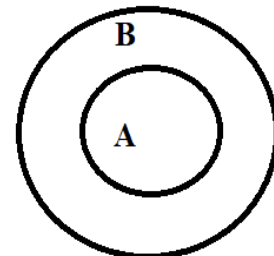
print(A.subset(B))
O/P:-- False



print(A.isdisjoint(B))
O/P:- False



print(A.issuperset(B))
O/P:-- False



print(A.issubset(B))
O/P:-- True

---: Frozenset :---

If we want to represent a group of unique elements with freezing nature then we can go for frozensets. frozenset cannot store duplicate elements.

1. Duplicates are not allowed.
2. Order is not preserved.
3. Objects are immutable.
4. Indexing is not allowed.
5. Slicing is not allowed.
6. Represented in `frozenset({ collection })` with comma separated objects.
7. Homogeneous and Heterogeneous both objects are allowed.

```
d = {'name':'Neeraj','age':37,'quali':'M.Tech'}
fs = frozenset(d)
print(fs)
print(type(fs))
print(id(fs))
```

O/P:---

```
frozenset({'quali', 'age', 'name'})
<class 'frozenset'>
3031112853184
```

Python in-built functions for frozenset:-

```
max()
min()
len()
sum()
id()
type()
print()
```

```
d = {'name':'Neeraj','age':37,'quali':'M.Tech'}
fs = frozenset(d)

print(max(fs))
print(min(fs))
print(len(fs))
```



```
print(type(fs))  
print(id(fs))  
print(fs)
```

O/P:--

quali

age

3

<class 'frozenset'>

2888669832896

frozenset({'age', 'name', 'quali'})

methods in frozenet :-

union()

intersection()

difference()

symmetric difference()

issuperset()

isdisjoint()

idsubset()

Python Basic practice Questions

1. Write some benefits/Advantages of Python.
2. Write some Limitation/Disadvantages of Python.
3. When to use a tuple vs list vs dictionary in Python?
4. What is a Negative Index in Python?
5. How do I modify a string in python?
6. What is indexing in Python?
7. What is slicing in Python?
8. Write some key features of Python
9. Name some Libraries of Python Programming language and their application?
10. What is the difference between Compiled Languages and Interpreted Languages?
11. What is a module in Python with example?
12. What is the use of Floor Division in python? Explain with examples.
13. What is the use of Modulus in python? Explain with examples.
14. What is the use Range function in python ?
15. What are .py and .pyc files ?
16. What are the types of literals in Python?
17. What are some built in data types in python ?
18. List some common Python interpreters.
19. Write a program to print all keywords in python?
20. Write a program to print punctuation in python.
21. What is a token in python?
22. Write python inbuilt functions with examples.
23. Write inbuilt methods with examples in python.
24. What is the list in python? Explain list methods with examples.
25. What is the tuple in python? Explain methods with examples.
26. What is the dictionary in python? Explain methods with examples.
27. What is the set in python? Explain methods with examples.
28. What is the frozenset in python? Explain methods with examples.
29. Explain python objects and their types.
30. Write a difference between list and tuple.
31. Write a difference between set and frozen set.
32. Explain join and split methods in string with examples.
33. Write how we declared empty literal-types in python.
34. What is identifiers in python?
35. Write the difference between identifier and variable.
36. Write a program to swap two numbers without using third variable.

37. Write a program to swap two numbers using third variable.
38. Write a program to swap two numbers using addition/subtraction, multiplication/Division.
39. Write a program to take input from runtime and print type and id of that input.
40. Write a program to find area of triangle.
41. Write a program to find area of square.
42. Write a program to find area of rectangle.
43. Write a program to find square of any number(x^2).
44. Write a program to find square root of any number (\sqrt{x}).
45. Write a program to find cube root of any number ($\sqrt[3]{x}$).
46. Write a program to find cube of a number (x^3).
47. Write a program to find area of circle(πr^2).
48. Find max(),min(),len() against given dictionary.
d= {1:"Python",2:"Java",3:"Python"}
49. Find max(),min(),len() against given dictionary.
d= {1:"Python",2:"Java",'3':"Python"}
50. Find max(),sum(),len() against given dictionary.
d= {1:"Python",2:"Java",1:"Python"}
51. Find max(),min(),sum() against given dictionary.
d= {1:"Python",2:"Java",'3':"Python"}
52. Write a difference between is and ==.