1. History of Python

Python is a widely-used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code

The inspiration for the name came from BBC's TV Show – 'Monty Python's Flying Circus'

2. Features in Python

- Free and Open Source
- Easy to code
- Easy to Read
- Object-Oriented Language
- GUI Programming Support
- High-Level Language
- Extensible feature
- Easy to Debug
- Python is a Portable language
- Python is an integrated language
- Interpreted Language
- Allocating Memory Dynamically

3. Print():

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

The message can be a string, or any other object, the object will be converted into a string before written to the screen.

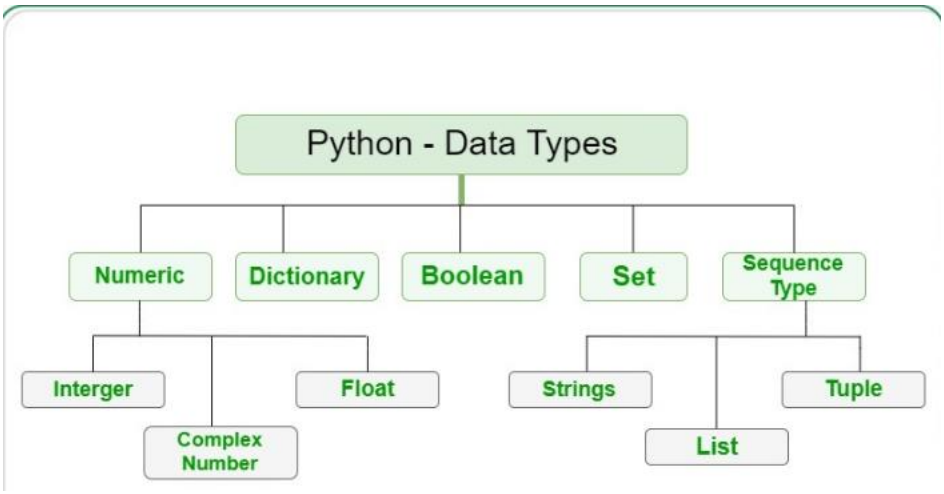
- print("Hello World")
- print("Hello", "how are you?")
- x=("apple", "banana", "cherry")
  print(x)
- print("Hello", "how are you?", sep="---")

4. Python Data Types

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are classes and variables are instance (object) of these classes.

Following are the standard or built-in data type of Python:

- Numeric
- Sequence Type
- Boolean
- Set
- Dictionary



4.1. Numeric

In Python, numeric data type represents the data which has numeric value. Numeric value can be integer, floating number or even complex numbers.
These values are defined as int, float and complex class in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.
- **Float** – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

  Note – type() function is used to determine the type of data type.

```
a = 5
print("Type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2 + 4j
print("\nType of c: ", type(c))
```

4.2. Sequence Type

In Python, sequence is the ordered collection of similar or different data types.
Sequences allows to store multiple values in an organized and efficient fashion.
There are several sequence types in Python

- String
- List
- Tuple

4.2.1 List

Python Lists are just like the arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type. The costly operation is inserting or deleting the element from the beginning of the List as all the elements are needed to be shifted. Insertion and deletion at the end of the list can also become costly in the case where the pre allocated memory becomes full.
We can create a list in python as shown below.

Example: Creating Python List
```
List = [1, 2, 3, "GFG", 2.3]
print(List)
```
Output:
[1, 2, 3, 'GFG', 2.3]

1. How to Declare Python List?
To use a list, you must declare it first. Do this using square brackets and separate values with commas.
```
>>> languages=['C++','Python','Scratch']
```
You can put any kind of value in a list. This can be a string, a Tuple, a Boolean, or even a list itself.
```
>>> list1=[1,[2,3],(4,5),False,'No']
```
Note that here, we put different kinds of values in the list. Hence, a list is (or can be) heterogeneous.

2. How to Access Python List?
a. Accessing an entire list
To access an entire list, all you need to do is to type its name in the shell.
```
>>> list1
```
Output
[1, [2, 3], (4, 5), False, 'No']
b. Accessing a single item from the list
To get just one item from the list, you must use its index. However, remember that indexing begins at 0. Let us first look at the two kinds of indexing.

- **Positive Indexing**– As you can guess, positive indexing begins at 0 for the leftmost/first item, and then traverses right.
  ```
  >>> list1[3]
  ```
  Output
  False
- **Negative Indexing**– Contrary to positive indexing, negative indexing begins at -1 for the rightmost/last item, and then traverses left. To get the same item form list1 by negative indexing, we use the index -2.
  ```
  >>> type(list1[-2])
  ```
  Output
  <class 'bool'>

It is also worth noting that the index cannot be a float, it must be an integer.

3. Slicing a Python List
Sometimes, you may not want an entire list or a single item, but several items from it. Here, the slicing operator [:] comes into play.
Suppose we want items second through fourth from list 'list1'. We write the following code for this.
```
>>> list1[1:4]
```
Output
[[2, 3], (4, 5), False]

If the ending index is n, then it prints items till index n-1. Hence, we gave it an ending index of 4 here.

We can use negative indexing in the slicing operator too. Let us see how.

```
>>> list1[:-2]
```

Output

[1, [2, 3], (4, 5)]

Here, -2 is the index for the tuple (4,5).

4. A list is mutable

Mutability is the ability to be mutated, to be changed. A list is mutable, so it is possible to reassign and delete individual items as well.

```
>>> languages
```

Output

['C++', 'Python', 'Scratch']

```
>>> languages[2]='Java'
>>> languages
```

Output

['C++', 'Python', 'Java']

5. How to Delete a Python List?

Like anything else in Python, it is possible to delete a list. To delete an entire list, use the del keyword with the name of the list. But to delete a single item or a slice, you need its index/indices.

```
>>> del languages[2]
>>> languages
```

Output

['C++', 'Python']

Let us delete a slice now.

```
>>> del languages[1:]
>>> languages
```

Output

['C++']

6. Reassigning a List in Python

You can either reassign a single item, a slice, or an entire list. Let us take a new list and then reassign on it.

```
>>> list1=[1,2,3,4,5,6,7,8]
```

a. Reassigning a single item

```
>>> list1[0]=0
>>> list1
```

Output

[0, 2, 3, 4, 5, 6, 7, 8]

b. Reassigning a slice

Now let's attempt reassigning a slice.

```
>>> list1[1:3]=[9,10,11]
>>> list1
```

Output

[0, 9, 10, 11, 4, 5, 6, 7, 8]

c. Reassigning the entire list

Finally, let's reassign the entire list.

```
>>> list1=[0,0,0]
>>> list1
```

Output

[0, 0, 0]

c. Concatenation of Python List

The concatenation operator works for lists as well. It lets us join two lists, with their orders preserved.

```
>>> a,b=[3,1,2],[5,4,6]
>>> a+b
```

Output

[3, 1, 2, 5, 4, 6]

Python List Operations

a. Multiplication

This is an arithmetic operation. Multiplying a Python list by an integer makes copies of its items that a number of times while preserving the order.

```
>>> a*=3
>>> a
```

Output

[3, 1, 2, 3, 1, 2, 3, 1, 2]

b. Membership

You can apply the 'in' and 'not in' operators on a Python list.

```
>>> 1 in a
```

Output

True

```
>>> 2 not in a
```

Output

False

Iterating on a list

Python list can be traversed with a for loop in python.

```
>>> for i in [1,2,3]:
if i%2==0:
print(f"{i} is composite\n")
```

Output

2 is composite

Built-in List Functions

There are some built-in functions in Python that you can use on python lists.

a**. len():** It calculates the length of the list.

```
>>> len(even)
```

Output

3

b. **max()**: It returns the item from the list with the highest value.

```
>>> max(even)
```

Output

18

If all the items in your list are strings, it will compare.

```
>>> max(['1','2','3'])
```

Output

'3'

But it fails when some are numeric, and some are strings in python.

```
>>> max([2,'1','2'])
```

Output

Traceback (most recent call last):File "<pyshell#116>", line 1, in <module>
max([2,'1 ,'2]) ′ ′
TypeError: '>' not supported between instances of 'str' and 'int'

c. **min()**: It returns the item from the Python list with the lowest value.

```
>>> min(even)
```

Output

6

d. **sum()**: It returns the sum of all the elements in the list.

```
>>> sum(even)
```

Output

36

However, for this, the Python list must hold all numeric values.

```
>>> a=['1','2','3']
>>> sum(a)
```

Output

Traceback (most recent call last):File "<pyshell#112>", line 1, in <module>
sum(a)
TypeError: unsupported operand type(s) for +: 'int' and 'str'

It works on floats.

```
>>> sum([1.1,2.2,3.3])
```

Output

6.6

e. **sorted()**: It returns a sorted version of the list, but does not change the original one.

```
>>> a=[3,1,2]
>>> sorted(a)
```

Output

[1, 2, 3]

```
>>> a
```

Output

[3, 1, 2]

If the Python list members are strings, it sorts them according to their ASCII values.

```
>>> sorted(['hello','hell','Hello'])
```

Output

['Hello', 'hell', 'hello']

Here, since H has an ASCII value of 72, it appears first.

f. **list()**: It converts a different data type into a list.

```
>>> list("abc")
```

Output

['a', 'b', 'c']

It can't convert a single int into a list, though, it only converts iterables.

```
>>> list(2)
```

Output

Traceback (most recent call last):File "<pyshell#122>", line 1, in <module>
list(2) TypeError: 'int' object is not iterable

g**. any()**: It returns True if even one item in the Python list has a True value.

```
>>> any(["','1'])
```

Output

True

It returns False for an empty iterable.

```
>>> any([])
```

Output

False

h. **all():** It returns True if all items in the list have a True value.

```
>>> all(["','1'])
```

Output
False

It returns True for an empty iterable.

>>> all([])
Output
True

Built-in Methods

While a function is what you can apply on a construct and get a result, a method is what you can do to it and change it. To call a method on a construct, you use the dot-operator(.).

Python supports some built-in methods to alter a Python list.

a. **append()**: It adds an item to the end of the list.

>>> a
Output
[2, 1, 3]
>>> a.append(4)
>>> a
Output
[2, 1, 3, 4]

b. **insert():** It inserts an item at a specified position.

>>> a.insert(3,5)
>>> a
Output
[2, 1, 3, 5, 4]

This inserted the element 5 at index 3.

c. **remove()**: It removes the first instance of an item from the Python list.

>>> a=[2,1,3,5,2,4]
>>> a.remove(2)
>>> a
Output
[1, 3, 5, 2, 4]

Notice how there were two 2s, but it removed only the first one.

d. **pop()**: It removes the element at the specified index, and prints it to the screen.

>>> a.pop(3)
Output
2
>>> a
Output
[1, 3, 5, 4]

e. **clear()**: It empties the Python list.

>>> a.clear()
>>> a
Output
[]

It now has a False value.

>>> bool(a)
Output
False

f**. index():** It returns the first matching index of the item specified.

>>> a=[1,3,5,3,4]
>>> a.index(3)
Output
1

g. **count()**: It returns the count of the item specified.

```
>>> a.count(3)
```

Output

2

h. **sort()**: It sorts the list in an ascending order.

```
>>> a.sort()
>>> a
```

Output

[1, 3, 3, 4, 5]

i. **reverse()**: It reverses the order of elements in the Python lists.

```
>>> a.reverse()
>>> a
```

Output

[5, 4, 3, 3, 1]

This was all about the Python lists

4.2.2 Python String

In Python, **Strings** are arrays of bytes representing Unicode characters.

Example:

"KiranmaieP" or 'Kiranmaiep'

Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

a) Creating a String in Python

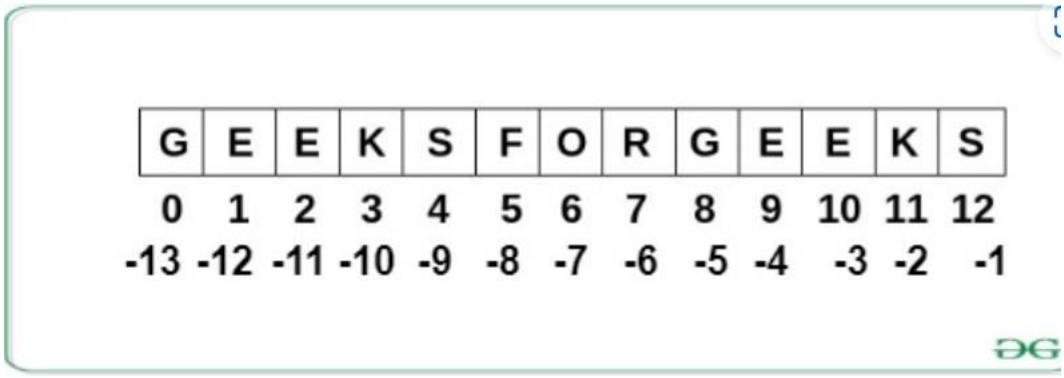
```
# Creating a String with single Quotes
String1 = 'Welcome to the ghost World'
print("String with the use of Single Quotes: ")
print(String1)
# Creating a String with double Quotes
String1 = "I'm a ghost"
print("\nString with the use of Double Quotes: ")
print(String1)
# Creating a String with triple Quotes
String1 = '''I'm a ghost and I live in a world of "ghosts"'''
print("\nString with the use of Triple Quotes: ")
print(String1)
#Creating String with triple Quotes allows multiple lines
String1 = '''ghosts
For
Life'''
print("\nCreating a multiline String: ")
print(String1)
```

Output:

```
String with the use of Single Quotes:
Welcome to the ghosts World
String with the use of Double Quotes:
I'm a ghost
String with the use of Triple Quotes:
I'm a ghost and I live in a world of "ghosts"
Creating a multiline String:
        ghosts
        For
        Life
```

b) Accessing characters in Python String

In Python, individual characters of a String can be accessed by using the method of Indexing.
Indexing allows negative address references to access characters from the back of the String,
e.g. -1 refers to the last character, -2 refers to the second last character, and so on.
While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types that will cause a **TypeError**.




```
String1 = "kiranmaiep"
print("Initial String: ")
print(String1)
# Printing First character
print("\nFirst character of String is: ")
print(String1[0])
# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
```

c) Reversing a Python String

With Accessing Characters from a string, we can also reverse them. We can Reverse a string by writing [::-1] and the string will be reversed.

```
kp = "kiranmaiep"
print(kp[::-1])
```

Output:

Skeegrofkeeg

We can also reverse a string by using built-in join and reversed function.

```
kp = "Kiranmaiep"
# Reverse the string using reversed and join function
kp = "".join(reversed(kp))
print(kp)
```

Output:

peiamnarik

d) String Slicing To access a range of characters in the String, the method of slicing is used. Slicing in a String is done by using a Slicing operator (colon).

```
# Creating a String
String1 = "helloeveryone"
print("Initial String: ")
print(String1)
# Printing 3rd to 12th character
print("\nSlicing characters from 3-12: ")
print(String1[3:12])
# Printing characters between 3rd and 2nd last character
print("\nSlicing characters between " +"3rd and 2nd last character: ")
print(String1[3:-2])
```

e) Deleting/Updating from a String

In Python, Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. Although deletion of the entire String is possible with the use of a built-in del keyword. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name.

Updation of a character:

```
# Python Program to Update character of a String
String1 = "Hello, I'm a Geek"
print("Initial String: ")
print(String1)

# Updating a character of the String as python strings are immutable, they don't support item updation directly there are following two ways

list1 = list(String1)
list1[2] = 'p'
String2 = ''.join(list1)
print("\nUpdating character at 2nd Index: ")
print(String2)
#2
String3 = String1[0:2] + 'p' + String1[3:]
print(String3)
```

Updation of a whole string:

```
String1 = "Hello, I'm kiranmaie"
print("Initial String: ")
print(String1)
# Updating a String
String1 = "Welcome to my World"
print("\nUpdated String: ")
print(String1)
```

Output:

Initial String:
Hello, I'm kiranmaie
Updated String:
Welcome to my world

Deletion of a character:

```
String1 = "Hello, I'm kiranmaie"
print("Initial String: ")
print(String1)
# Deleting a character of the String
String2 = String1[0:2] + String1[3:]
print("\nDeleting character at 2nd Index: ")
print(String2)
```

Output:

Initial String:
Hello, I'm kiranmaie
Deleting character at 2nd Index:
Helo, I'm kiranmaie

Deletion of entire string:
Deletion of the entire string is possible with the use of del keyword. Further, if we try to print the string, this will produce an error because String is deleted and is unavailable to be printed.

```
String1 = "Hello, I'm kiranmaie"
print("Initial String: ")
print(String1)
# Deleting a String with the use of del
del String1
print("\nDeleting entire String: ")
print(String1)
```

Output:

```
Error:
Traceback (most recent call last):
File "/home/e4b8f2170f140da99d2fe57d9d8c6a94.py", line 12, in
print(String1)
NameError: name 'String1' is not defined
```

f) Formatting of Strings

Strings in Python can be formatted with the use of format() method which is a very versatile and powerful tool for formatting Strings. Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.

```
String1 = "{} {} {}".format('love', 'For', 'Life')
print("Print String in default order: ")
print(String1)
# Positional Formatting
String1 = "{1} {0} {2}".format('love', 'For', 'Life')
print("\nPrint String in Positional order: ")
print(String1)
# Keyword Formatting
String1 = "{l} {f} {g}".format(g='love', f='For', l='Life')
print("\nPrint String in order of Keywords: ")
print(String1)
```

Output:

```
Print String in default order:
love For Life
Print String in Positional order:
For love Life
Print String in order of Keywords:
Life For love
```

```
# String alignment
String1 = "|{:<10}|{:^10}|{:>10}|".format('love', 'for', 'life')
print("\nLeft, center and right alignment with Formatting: ")
print(String1)
# To demonstrate aligning of spaces
String1 = "\n{0:^16} in {1:<4}!".format("loveforlife",2009)
print(String1)
```

Output:

```
Left, center and right alignment with Formatting:
|love    |   for   |    life|
```

Loveforlife in 2009 !

```
Integer1 = 12.3456789
print("Formatting in 3.2f format: ")
print('The value of Integer1 is %3.2f' % Integer1)
print("\n Formatting in 3.4f format: ")
print('The value of Integer1 is %3.4f' % Integer1)
```

Output:

```
Formatting in 3.2f format:
The value of Integer1 is 12.35
Formatting in 3.4f format:
The value of Integer1 is 12.3457
```

4.2.3 Tuple

This Python Data Structure is like a, like a list in Python, is a heterogeneous container for items. But the major difference between the two (tuple and list) is that a list is mutable, but a tuple is immutable. This means that while you can reassign or delete an entire tuple, you cannot do the same to a single item or a slice.

To declare a tuple, we use parentheses.

```
>>> colors=('Red','Green','Blue')
```

- **Python Tuple Packing**

Python Tuple packing is the term for packing a sequence of values into a tuple without using parentheses.

```
>>> mytuple=1,2,3
>>> mytuple
```

Output

```
(1, 2, 3)
```

- **Python Tuple Unpacking**

The opposite of tuple packing, unpacking allots the values from a tuple into a sequence of variables.

```
>>> a,b,c=mytuple
>>> print(a,b,c)
```

Output

```
1 2 3
```

- **Creating a tuple with a single item**

Let’s do this once again. Create a tuple and assign a 1 to it.

```
>>> a=(1)
```

N

ow, let’s call the type() function on it.

```
>>> type(a)
```

Output

```
<class ‘int’>
```

As you can see, this declared an integer, not a tuple. To get around this, you need to append a comma to the end of the first item 1. This tells the interpreter that it’s a tuple.

```
>>> a=(1,)
>>> type(a)
```

Output

```
<class ‘tuple’>
```

- **Accessing, Reassigning, and Deleting Items**

We can perform these operations on a tuple just like we can on a list. The only differences that exist are because a tuple is immutable, so you can’t mess with a single item or a slice.

```
>>> del a[0]
```

Output

Traceback (most recent call last):File “”, line 1, in del a[0] TypeError: ‘tuple’ object doesn’t support item deletion

Even though this tuple has only one item, we couldn’t delete it because we used its index to delete.

4.3. Python Set

This is one of the important Python Data Structures. A Python set is a slightly different concept from a list or a tuple. A set, in Python, is just like the mathematical set. It does not hold duplicate values and is unordered. However, it is not immutable, unlike a tuple.

Let’s first declare a set. Use curly braces for the same.

>>> myset={3,1,2}

>>> myset

Output {1, 2, 3}

As you can see, it rearranged the elements in an ascending order.

Since a set is unordered, there is no way we can use indexing to access or delete its elements. Then, to perform operations on it, Python provides us with a list of functions and methods like discard(), pop(), clear(), remove(), add(), and more. Functions like len() and max() also apply on sets.

- **To initialize an empty set**: sample_set = set()
- **Add elements to the set:**

sample_set.add(item) This adds a single item to the set

sample_set.update(items) This can add multiple items via a list, tuple, or another set

- **Remove elements from the set:**

sample_set.discard(item) Removes element without warning if element not present

sample_set.remove(item) Raises an error if the element to be removed is not present.

- **Set operations** (Assume two sets initialized: A and B):

A | B or A. union(B): Union operation

A & B or A. intersection(B): Intersection operation

A – B or A. difference(B): Difference of two sets

A ^ B or A. symmetric_difference(B) : Symmetric difference of sets

5. Operators:

Operators are used to perform operations on values and variables. These are the special symbols that carry out arithmetic and logical computations. The value the operator operates on is known as *Operand*.

5.1 Arithmetic operators:

Python Arithmetic operators include Addition, Subtraction, Multiplication, Division, Floor Division, Exponent (or Power), and Modulus. All these Arithmetic are binary operators, which means they operate on two operands.

```
a = 12
b = 3
addition = a+b
subtraction = a-b
multiplication = a*b
division = a / b
modulus = a % b
exponent = a**b
Floor_Division = a // b
print("Addition of two numbers 12 and 3 is : ", addition)
print("Subtracting Number 3 from 12 is : ", subtraction)
print("Multiplication of two numbers 12 and 3 is : ", multiplication)
print("Division of two numbers 12 and 3 is : ", division)
print("Modulus of two numbers 12 and 3 is : ", modulus)
print("Exponent of two numbers 12 and 3 is : ", exponent)
print("Floor Division of two numbers 12 and 3 is : ", Floor_Division)
```

Arithmetic Operations on Strings

In this Python Arithmetic operators example, we use two variables, a and b, of string data type. Next, we use these two variables to show the problems we generally face while performing arithmetic operations on String Data type.

```
a = "Tutorial"
b = "Gateway"
print(a + b)
print(a + " " + b)
print(a * 3)
print((a + " ")* 3)
print(a + 3)
print(a + str(3))
```

PRECEDENCE:

- P – Parentheses
- E – Exponentiation
- M – Multiplication (Multiplication and division have the same precedence)
- D – Division
- A – Addition (Addition and subtraction have the same precedence)
- S – Subtraction

5.2. Relational Operators

Relational operators are used for comparing the values. It either returns True or False according to the condition. These operators are also known as Comparison Operators.

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Example:

```
a = 9
b = 5
print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

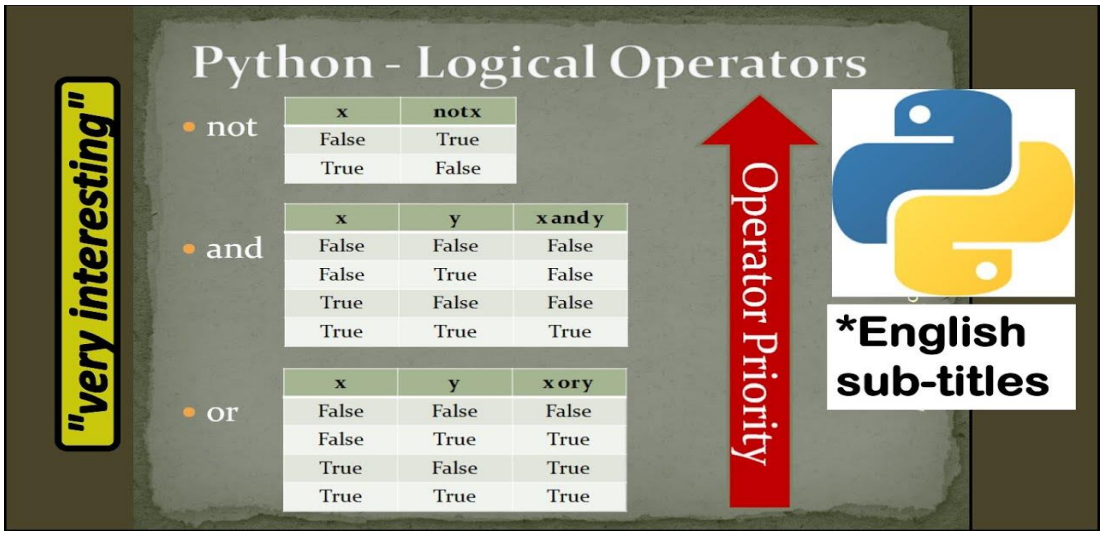
Output:

```
True
False
False
True
True
False
```

5.3. Logical operators

In Python, Logical operators are used on conditional statements when there is more than one condition to check (either True or False). They perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x




```
a = 10
b = 10
c = -10
if a > 0 and b > 0:
        print("The numbers are greater than 0")
if a > 0 and b > 0 and c > 0:
        print("The numbers are greater than 0")
else:
        print("Atleast one number is not greater than 0")
```

Output:

The numbers are greater than
Atleast one number is not greater than 0

5.4. Bitwise operators

Operator	Description
& Binary AND	Operator copies a bit to the result if it exists in both operands
| Binary OR	It copies a bit if it exists in either operand.
^ Binary XOR	It copies the bit if it is set in one operand but not both.
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.

Bitwise Operator

✓ Applied to integer type – long, int, short, byte and char.

A	B	A | B	A & B	A^ B	~A
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0


```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print "Line 1 - Value of c is ", c

c = a | b;        # 61 = 0011 1101
print "Line 2 - Value of c is ", c

c = a ^ b;        # 49 = 0011 0001
print "Line 3 - Value of c is ", c

c = ~a;           # -61 = 1100 0011
print "Line 4 - Value of c is ", c

c = a << 2;       # 240 = 1111 0000
print "Line 5 - Value of c is ", c
```

```
c = a >> 2;       # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

When you execute the above program it produces the following result −

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

5.5. Assignment operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3

5.6. Identity operators

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

5.7. Membership operators

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

6. Expressions in Python

An expression is a combination of operators and operands that is interpreted to produce some other value. In any programming language, an expression is evaluated as per the precedence of its operators. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first. We have many different types of expressions in Python. Let's discuss all types along with some exemplar codes:

Constant Expressions: These are the expressions that have constant values only.

Example:

```
x = 15 + 1.3
print(x)
```

Arithmetic Expressions: An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis. The result of this type of expression is also a numeric value. The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

Example:

```
add = x + y
sub = x – y
pro = x * y
div = x / y
```

Integral Expressions: These are the kind of expressions that produce only integer results after all computations and type conversions.

Example:

```
a = 13
b = 12.0
c = a + int(b)
print(c)
```

Floating Expressions: These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

Example:

```
a = 13
b = 5
c = a / b
print(c)
```

Relational Expressions: In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=). Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end. These expressions are also called Boolean expressions.

Example:

```
a = 21
b = 13
c = 40
d = 37
p = (a + b) >= (c - d)
print(p)
```

Logical Expressions: These are kinds of expressions that result in either *True* or *False*. It basically specifies one or more conditions.

Example:

```
P = (10 == 9)
Q = (7 > 5)
# Logical Expressions
R = P and Q
S = P or Q
```

T = not P

Bitwise Expressions: These are the kind of expressions in which computations are performed at bit level.

Example:

```
a = 12
x = a >> 2
y = a << 1
```

Combinational Expressions: We can also use different types of expressions in a single expression, and that will be termed as combinational expressions.

Example:

```
a = 16
b = 12
c = a + (b >> 1)
print(c)
```

Precedence	Name	Operator
1	Parenthesis	( ) [ ] { }
2	Exponentiation	**
3	Unary plus or minus, complement	-a , +a , ~a
4	Multiply, Divide, Modulo	/ * // %
5	Addition & Subtraction	+ -
6	Shift Operators	>> <<

7	Bitwise AND	&
8	Bitwise XOR	^
9	Bitwise OR	|
10	Comparison Operators	>= <= > <
11	Equality Operators	== !=
12	Assignment Operators	= += -= /= *=
13	Identity and membership operators	is, is not, in, not in
14	Logical Operators	and, or, not

CHARACTER SET

The characters that can be used to form words numbers and expressions depend upon the computer on which the program runs.

characters are grouped into following categories.

Alphabets	A, B, C, D, E, F,...Y,Z
	a, b, c, d, e, f,..... y, z
Digits	0, 1, 2, 3, 4,..... 9
special characters	~ ! @ # % ^ & * ( ) _ - + = $ 1 \ { } [ ] : ; " <> , . ? /
White spaces	Blanks space
	Horizontal tab, carriage return, New line, form feed (\ f), Page Break

Identifiers:

Identifiers are the names given to the variable, functions, arrays. These are user defined names and consist of sequence of letters and digits with a letter as a first character. Both upper case and Lower case letters permitted. The underscore character is also permitted.

Rules for identifiers:

- First character must be an alphabet or an underscore.
- Must contain letters,digits and "_"only.
- Only first 31 characters are significant.
- A max 31 characters should be used.
- Some compilers allow 63 characters also key words should not be used as identifiers. White space is not allowed.

  Ex: sum sum-1, int, x, a ->valid

  int, 1var, sum.123,sum123 -> invalid

7. Sequence Mutation and accumulating patterns:

Mutating methods are ones that change the object after the method has been used. Non-mutating methods do not change the object after the method has been used. The `count` and `index` methods are both non-mutating. Count returns the number of occurrences of the argument given but does not change the original string or list. Similarly, index returns the leftmost occurrence of the argument but does not change the original string or list.

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

Append versus Concatenate:

The append method adds a new item to the end of a list. It is also possible to add a new item to the end of a list by using the concatenation operator. However, you need to be careful.

Eg:

origlist = [45,32,88]

origlist.append("cat")

We can also use concatenate

```
1  origlist = [45,32,88]
2
3  origlist = origlist + ["cat"]
```

The key difference is while using append method the id of list doesn't change but while using concatenation it will become completely new list and id will change.

ID of list unique number for each list

We can use append or concatenate repeatedly to create new objects. If we had a string and wanted to make a new list, where each element in the list is a character in the string, where do you think you should start? In both cases, you'll need to first create a variable to store the new object.

```
st = "Warmth"
a = []
b = a + [st[0]]
c = b + [st[1]]
d = c + [st[2]]
e = d + [st[3]]
f = e + [st[4]]
g = f + [st[5]]
print(g)
```

```
st = "Warmth"
a = []
a.append(st[0])
a.append(st[1])
a.append(st[2])
a.append(st[3])
a.append(st[4])
a.append(st[5])
print(a)
```

Non-mutating Methods on Strings:

ss = "Hello, World"

print(ss.upper())

tt = ss.lower()

print(tt)

print(ss)

similarly count and index are non-mutating

- **Accumulator patterns are used with iterators**

  Let us create a list of strings and named it colors. We can use a for loop to iterate the list like:

  colors= [“orange”, ”pink”, ”White”]

  for each in colors:

  Print(each)

Example:

a=[3,5,8]

Pow=[]

for w in a:

x=w**2

pow.append(x)

print(pow)

We can also user **itertools** by importing them

Itertools module is a collection of functions. We are going to explore one of these **accumulate()** function.

Example1:

```
import itertools

import operator

siri = [1, 2, 3, 4, 5]

result = itertools.accumulate(siri, operator.mul)

for each in result:

        print(each)
```

Output:

1

2

6

24

120

Example:2

```
import itertools

import operator

siri = [5, 3, 6, 2, 1, 9, 1]

result = itertools.accumulate(siri, max)

for each in result:

        print(each)
```

Output:

5

5

6

6

6

9

9

Another module of **itertools is repeat**

```
import itertools

print ("Printing the numbers repeatedly : ")

print (list(itertools.repeat(25, 4)))
```

Output:

Printing the numbers repeatedly :

[25, 25, 25, 25]

8. Global and Local Variables in Python

Global variables are those which are not defined inside any function and have a global scope whereas local variables are those which are defined inside a function and its scope is limited to that function only. In other words, we can say that local variables are accessible only inside the function in which it was initialized whereas the global variables are accessible throughout the program and inside every function. Local variables are those which are initialized inside a function and belong only to that particular function. It cannot be accessed anywhere outside the function. Let's see how to create a local variable.

Example:

```
def f():

# local variable

        s = "we love kiranmaie mam"

        print(s)

# Driver code

f()
```

Output:

We love kiranmaie mam

If we will try to use this local variable outside the function then let's see what will happen.

Example:

```
def f():

# local variable

        s = "we love kiranmaie mam "

        print("Inside Function:", s)

# Driver code

f()

print(s)
```

output:

Inside Function: we love kiranmaie mam

NameError: name 's' is not defined

Global Variables

These are those which are defined outside any function and which are accessible throughout the program, i.e., inside and outside of every function. Let's see how to create a global variable.

Example: Defining and accessing global variables

```
def f():

        print("Inside Function", s)

# Global scope

s = "we love kiranmaie mam"

f()

print("Outside Function", s)
```

Output:

Inside Function we love kiranmaie mam

Outside Function we love kiranmaie mam

Now, what if there is a variable with the same name initialized inside a function as well as globally. Now the question arises, will the local variable will have some effect on the global variable or vice versa, and what will happen if we change the value of a variable inside of the function f()? Will it affect the globals as well? We test it in the following piece of code:

```
def f():

        s = "Me too."

        print(s)

# Global scope

s = "we love kiranmaie mam"

f()

print(s)
```

Output:

```
Me too.

we love kiranmaie mam
```

We can use global keyword to make local variable global.