Python Operators

Python operators can be classified into several categories.

- Assignment Operators
- Arithmetic Operators
- Logical Operators
- Comparison Operators
- Bitwise Operators

Python Assignment Operators

Assignment operators include the basic assignment operator equal to sign (=).

But to simplify code, and reduce redundancy, Python also includes arithmetic assignment operators.

This includes the += operator in Python used for addition assignment, **//= floor division assignment operator**, and others.

List of all the arithmetic assignment operators in Python.

Operator	Description
+=	a+=b is equivalent to a=a+b
*=	a*=b is equivalent to a=a*b
/=	a/=b is equivalent to a=a/b
%=	a%=b is equivalent to a=a%b
**=	a**=b is equivalent to a=a**b (exponent operator)
//=	a//=b is equivalent to a=a//b (floor division)

Using assignment operators

Example 1:

```
# Take two variables, assign values with assignment operators (Observation and Record)

a=3

b=4

print("a: "+str(a))

print("b: "+str(b))

# it is equivalent to a=a+b

a+=b

print("a: "+str(a))

print("b: "+str(b))

# it is equivalent to a=a*b

a*=b

print("a: "+str(a))

print("b: "+str(b))

# it is equivalent to a=a/b

a/=b

print("a: "+str(a))

print("b: "+str(b))

# it is equivalent to a=a%b

a%=b

print("a: "+str(a))

print("b: "+str(b))

# it is equivalent to a=a**b ( exponent operator)

a**=b
```

```
print("a: "+str(a))

print("b: "+str(b))

# it is equivalent to a=a//b (floor division)

a//=b

print("a: "+str(a))

print("b: "+str(b))
```

Example:2

```
# Assignment Operator
a = 10

# Addition Assignment
a += 5
print ("a += 5 : ", a)

# Subtraction Assignment
a -= 5
print ("a -= 5 : ", a)

# Multiplication Assignment
a *= 5
print ("a *= 5 : ", a)

# Division Assignment
a /= 5
print ("a /= 5 : ",a)

# Remainder Assignment
a %= 3
print ("a %= 3 : ", a)

# Exponent Assignment
a **= 2
print ("a **= 2 : ", a)

# Floor Division Assignment
a //= 3
print ("a //= 3: ", a)
```

Arithmetic operators

The following table lists the arithmetic operators supported by Python:

Operator	Example	Meaning	Result
+ (unary)	+a	Unary Positive	a In other words, it doesn’t really do anything. It mostly exists for the sake of completeness, to complement Unary Negation.
+ (binary)	a + b	Addition	Sum of a and b
- (unary)	-a	Unary Negation	Value equal to a but opposite in sign
- (binary)	a - b	Subtraction	b subtracted from a
*	a * b	Multiplication	Product of a and b
/	a / b	Division	Quotient when a is divided by b. The result always has type float.
%	a % b	Modulo	Remainder when a is divided by b
//	a // b	Floor Division (also called Integer Division)	Quotient when a is divided by b, rounded to the next smallest whole number
**	a ** b	Exponentiation	a raised to the power of b

Examples of these operators in use:

```
>>> a = 4
>>> b = 3
>>> +a
4
>>> -b
-3
>>> a + b
7
>>> a - b
1
>>> a * b
12
>>> a / b
1.3333333333333333
>>> a % b
```

```
1
>>> a ** b
64
```

The result of standard division (/) is always a float, even if the dividend is evenly divisible by the divisor:

```
>>> 10 / 5
2.0
>>> type(10 / 5)
<class 'float'>
```

When the result of floor division (//) is positive, it is as though the fractional portion is truncated off, leaving only the integer portion. When the result is negative, the result is rounded down to the next smallest (greater negative) integer:

```
>>> 10 / 4
2.5
>>> 10 // 4
2
>>> 10 // -4
-3
>>> -10 // 4
-3
>>> -10 // -4
2
```

Examples: **(Record and Observation)**

```
a = 21
b = 10
# Addition
print ("a + b : ", a + b)
# Subtraction
print ("a - b : ", a - b)

# Multiplication
```

```
print ("a * b : ", a * b)
# Division
print ("a / b : ", a / b)
# Modulus
print ("a % b : ", a % b)
# Exponent
print ("a ** b : ", a ** b)
# Floor Division
print ("a // b : ", a // b)
```

Comparison Operators

Operator	Example	Meaning	Result
==	a == b	Equal to	True if the value of a is equal to the value of b False otherwise
!=	a != b	Not equal to	True if a is not equal to b False otherwise
<	a < b	Less than	True if a is less than b False otherwise
<=	a <= b	Less than or equal to	True if a is less than or equal to b False otherwise
>	a > b	Greater than	True if a is greater than b False otherwise
>=	a >= b	Greater than or equal to	True if a is greater than or equal to b False otherwise

Examples of the comparison operators in use:

```
>>> a = 10
>>> b = 20
>>> a == b
False
>>> a != b
True
>>> a <= b
True
>>> a >= b
False
```

```
>>> a = 30
>>> b = 30
>>> a == b
True
>>> a <= b
True
>>> a >= b
True
```

```
a = 5


b = 2


# equal to operator
print('a == b =', a == b)


# not equal to operator
print('a != b =', a != b)


# greater than operator
print('a > b =', a > b)


# less than operator
print('a < b =', a < b)


# greater than or equal to operator
print('a >= b =', a >= b)


# less than or equal to operator
print('a <= b =', a <= b)
```

Example: **(Record and Observation)**

```
a = 4
b = 5

# Equal
print ("a == b : ", a == b)

# Not Equal
print ("a != b : ", a != b)

# Greater Than
print ("a > b : ", a > b)

# Less Than
print ("a < b : ", a < b)

# Greater Than or Equal to
print ("a >= b : ", a >= b)

# Less Than or Equal to
print ("a <= b : ", a <= b)
```

Python Logical Operators

Operator	Example	Meaning
and	a and b	Logical AND: True only if both the operands are True
or	a or b	Logical OR: True if at least one of the operands is True
not	not a	Logical NOT: True if the operand is False and vice-versa.

Example: **(Record and Observation)**

```
# logical AND
```

```
print(True and True)     # True
print(True and False)    # False
# logical OR
print(True or False)     # True
# logical NOT
print(not True)          # False
```

Python Bitwise operators

For example, **2** is 10 in binary and **7** is 111.
In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

It performs 1's complement operation. It invert each bit of binary value and returns the bitwise negation of a value as a result.

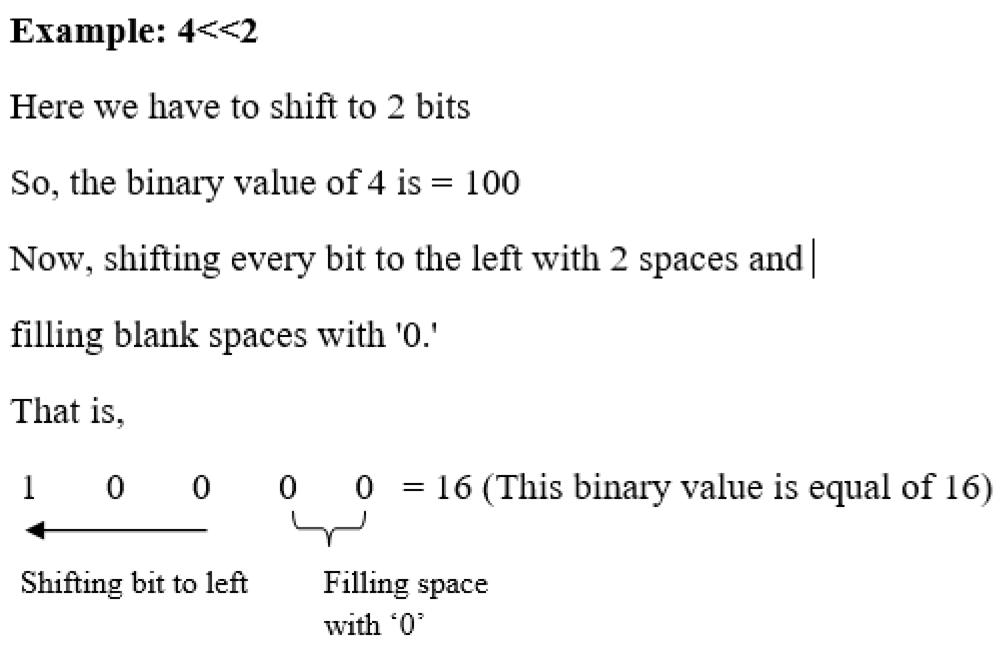
```
a = 7
b = 4
c = 3
```

```
print(~a, ~b, ~c)
# Output -8 -5 -4
```

Bitwise left-shift <<

The left-shift << operator performs a shifting bit of value by a given number of the place and fills 0’s to new positions.

```
print(4 << 2)
# Output 16
print(5 << 3)
# Output 40
```



Example: 4<<2

Here we have to shift to 2 bits

So, the binary value of 4 is = 100

Now, shifting every bit to the left with 2 spaces and

filling blank spaces with '0.'

That is,

$\underset{\longleftarrow}{1 \quad 0 \quad 0} \quad \underbrace{0 \quad 0} = 16$ (This binary value is equal of 16)

Shifting bit to left
Filling space with ‘0’

Bitwise right-shift >>

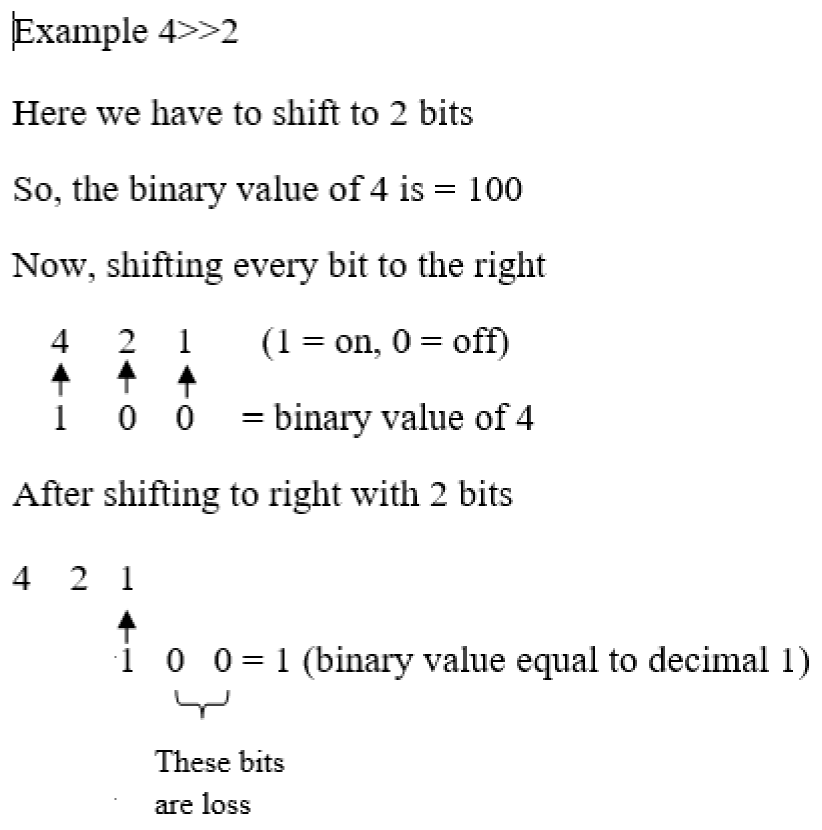
The left-shift >> operator performs shifting a bit of value to the right by a given number of places. Here some bits are lost.

```
print(4 >> 2)
```

# Output

print(5 >> 2)

# Output



Example 4>>2

Here we have to shift to 2 bits

So, the binary value of 4 is = 100

Now, shifting every bit to the right

```
4   2   1      (1 = on, 0 = off)
↑   ↑   ↑
1   0   0    = binary value of 4
```

After shifting to right with 2 bits

```
4   2   1
        ↑
        1   0   0 = 1 (binary value equal to decimal 1)
```

$\underbrace{0\ \ 0}$

These bits
are loss

Example: **(Record and Observation)**

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

# Binary AND
c = a & b       # 12 = 0000 1100
print ("a & b : ", c)
```

```
# Binary OR
c = a | b        # 61 = 0011 1101
print ("a | b : ", c)

# Binary XOR
c = a ^ b        # 49 = 0011 0001
print ("a ^ b : ", c)

# Binary Ones Complement
c = ~a;          # -61 = 1100 0011
print ("~a : ", c)

# Binary Left Shift
c = a << 2;      # 240 = 1111 0000
print ("a << 2 : ", c)

# Binary Right Shift
c = a >> 2;      # 15 = 0000 1111
print ("a >> 2 : ", c)
```

Python Special operators

Special types of operators like the **identity** operator and the **membership** operator

Identity operators

In Python, is and is not are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

The is operator returns Boolean True or False. It Return True if the memory address first value is equal to the second value. Otherwise, it returns False.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not	x is not

	refer to the same object)	True

Example:(Record and Observation)

```
x = 10
y = 11
z = 10
print(x is y) # it compare memory address of x and y
print(x is z) # it compare memory address of x and z
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]
print(x1 is not y1)  # prints False
print(x2 is y2)  # prints True
print(x3 is y3)  # prints False
x = 10
y = 11
z = 10
print(x is not y) # it campare memory address of x and y
print(x is not z) # it campare memory address of x and z
```

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).
But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

Membership operators

In Python, in and not in are the membership operators. They are used to test whether a value or variable is found in a sequence
(string, list, tuple, set and dictionary).
In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Membership operators in Python

Example:(Record and Observation)

```
x = 'Hello world'
y = {1:'a', 2:'b'}
# check if 'H' is present in x string
print('H' in x)  # prints True
# check if 'hello' is present in x string
print('hello' not in x)  # prints True
# check if '1' key is present in y
print(1 in y)  # prints True

# check if 'a' key is present in y
print('a' in y)  # prints False
```

Python Operators Precedence

Precedence level	Operator	Meaning
1 (Highest)	()	Parenthesis
2	**	Exponent
3	+x, -x ,~x	Unary plus, Unary Minus, Bitwise negation
4	*, /, //, %	Multiplication, Division, Floor division, Modulus
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise shift operator
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	==, !=, >, >=, <, <=	Comparison
11	is, is not, in, not in	Identity, Membership

12	not	Logical NOT
13	and	Logical AND
14 (Lowest)	or	Logical OR

Python Built-in Functions

1. abs(x)

Returns the absolute value of a number. In case a complex number is passed, the magnitude of that number is returned. It is the same as the distance from the origin of a point on an x-y graph.

Example:(Record and Observation)

Abs(-3) =3

abs(3+4i) = 5

```
a = 12
b = -4
c = 3+4j
d = 7.90
print(abs(a))
print(abs(b))
print(abs(c))
print(abs(d))
```

2. all(x)

Same as logical 'and' operator. That means it will return true if all variables in the iterator are true. Here iterable objects are referred to as tuple, lists, dictionary.

A variable is said to be true in python if it is non-zero and not NONE. Here NONE is a keyword defined in python that is considered null.

For eg=, if the iterable 'item' contains value '2,4,5,6,1' – The result will be true.

If item1= '2,0,4,5' – The result will be false

```
tuple = (0, True, False)
x = all(tuple)
print(x)
```

```
sampledict = {0 : "Apple", 1 : "Orange"}
x = all(sampledict)
print(x)
```

Explanation- Here, all() returns False because one of the keys is false, and in the case of dictionaries, only the keys are checked, not the values.

3. any(x)

This function is the same as the logical ‘OR’ operator that returns False only if all the variables present in an iterable are false. Here iterable refers to the tuple, dictionary, and lists.

Note- For an empty iterable object, any() returns False.

For example- any(2,3,4,5,9) – True

Any(2,0,9,1,8) – Returns False

Code:

```
myset = {0, 1, 0}
x = any(myset)
print(x)
```

Output:

Explanation- In the above program, any function returns True, and the set is given is having at least one True value.

4. bin()
This function returns the binary value of a number.

Code:

```
a=5
print(bin(a))
```

5. round()

It gives a roundoff value for a number, i.e. gives, the nearest integer value for a number. This function accepts one argument, either decimal, float, or integer, and gives roundoff output.

Code:

```
print(round(4.5))
print(round(-7.7))
```

6. bin()

It returns the binary value for the number passed in the argument. The only integer can be passed as an argument to the function.

Code:

```
print(bin(4))
print(bin(9))
```

7. bool()

This function returns the Boolean value of an object.

Code:

```
print(bool(0))

print(bool(-4.5))

print(bool(None))
print(bool("False"))
```

8. bytearray()

This function returns a new array of bytes, i.e. a mutable sequence of integers from range 0 to 256.

Syntax –

```
bytearray(source,encoding,errors)
```

Note-

1. The values to function are optional.
2. If any non-ascii value is given to the function, it gives the error - TypeError: string argument without an encoding.

Code:

```
print(bytearray())
print(bytearray('Python','utf-8'))
```

Output:

9. compile()

It is used to generate a Python code object from a string or an AST object.

Following is the syntax for the function –

```
Compile(source,filename,mode,flags=0,dont_inherit=False,optimize=-1)
```

This function’s output is given as an argument to evaluate the () and exec() functions.

Code:

```
myCode = 'a = 7\nb=9\nresult=a*b\nprint("result =",result)'

codeObject = compile(myCode, 'resultstring', 'exec')
exec(codeObject)
```

10. list()
This function is used to convert an object to a list object.

Syntax –

```
list([iterable])
```

Here iterable refers to any sequence such as string, tuples, and iterable object or collection object such as a set or dictionary.

A mutable sequence of the list of elements is returned as an output of this function.

Code:

```
print(list()) #returns empty list

stringobj = 'PALINDROME'

print(list(stringobj))

tupleobj = ('a', 'e', 'i', 'o', 'u')

print(list(tupleobj))
listobj = ['1', '2', '3', 'o', '10u']
print(list(listobj))
```

11. len()
This function gives the length of the object as an output.

Syntax –

```
len([object])
```

Here objects must be either a sequence or a collection.

Note- The interpreter throws an error in case it encounters no argument given to the function.

Code:

```
stringobj = 'PALINDROME'

print(len(stringobj))

tupleobj = ('a', 'e', 'i', 'o', 'u')

print(len(tupleobj))
```

```
listobj = ['1', '2', '3', 'o', '10u']
print(len(listobj))
```

Output:

12. max()

This function returns the largest number in the given iterable object or the maximum of two or more numbers given as arguments.

Syntax –

```
max(iterable) or max(num1,num2…)
```

Here iterable can be a list, tuple, dictionary, or any sequence or collection.

Code:

```
num = [11, 13, 12, 15, 14]
print('Maximum is:', max(num))
```

Output:

Note- In case no arguments are given to the function, then ValueError is thrown by the interpreter.

13. min()

This function returns the minimum value from the collection object or the values defined as arguments.

Syntax –

```
min([iterable])
```

Code:

```
print(min(2,5,3,1,0,99))

sampleObj = ['B','a','t','A']
print(min(sampleObj))
```

Output:

Note– In case no arguments are given to the function, then ValueError is thrown by the interpreter.

14. map()

This function helps to debug as it provides the result after an operation is applied to each item in an iterable object.

Syntax –

```
map(fun,[Iterable])
```

where iterable can be a list, tuple, etc…

Code:

```
numList = (11, 21, 13, 41)

res = map(lambda x: x + x, numList)
print(list(res))
```

15. open()

After opening a particular file, this function returns a file object that helps to read or write into that file.

Syntax –

```
open(file, mode)
```

file -refers to the name with the complete path of the file to read or written into.\

mode- refers to the manner or the work we need to perform with the file. It can value like ‘r’,’ a’,’x’ etc.

Code:

```
f = open("myFile.txt", "r")#read mode
print(f.read())
```

16. pow()

This function returns the value of the power of 1 number to another number.

Syntax –

```
pow(num1,num2)
```

where num1, and num2 must be an integer, float, or double.

Code:

```
print(pow(2,-3))
print(pow(2,4.5))
print(pow(3,0))
```

17. oct()

This function helps to generate the octal representation of a number.

Syntax –

```
oct(number)
```

where the number can be an integer, hexadecimal, or binary number.

Example:(Record and Observation)

```
print("The octal representation of 32 is " + oct(32))
print("The octal representation of the"
    " ascii value of 'A' is " + oct(ord('A')))
print("The octal representation of the binary" " of 32 is " + oct(100000))
```

```
print("The octal representation of the binary"
                " of 23 is " + oct(0x17))
```

18. sorted()
This function has made the sorting of numbers very easy.

Syntax –

```
sorted(iterable,key,reverse)
```

Here, iterable – refers to the list, tuple or any collection of objects that needs to be sorted.

Key – refers to the key on which the values must be sorted.

Reverse- this can be set to true to generate the list in descending order.

The output of this function is always a list.

Code:

```
sampleObj = (3,6,8,2,5,8,10)
print(sorted(sampleObj,reverse=True))
sampledict = {'a':'sss','g':'wq','t':2}
print(sorted(sampledict,key= len))
```

19. sum()
This function helps to sum the member of an iterable object.

Syntax –

```
sum([iterable],start)
```

Iterable refers to any iterable object list, tuple, dictionary, or sequence of numbers.

Start – this marks the initialization of the sum result that needs to be added to the final result. It is an optional argument.

Code:

```
num = [2.5, 3, 4, -5]
numSum = sum(num)
print(numSum)
numSum = sum(num, 20)
print(numSum)
```

20. str()

This function helps to generate the printable representation of an Object.

Syntax –

```
str(object,encoding,errors)
```

where encoding and errors are optional.

Code:

```
print(str('A1323'))
```

```
b = bytes('pythön', encoding='utf-8')

print(str(b, encoding='ascii', errors='ignore'))
#errors='ignore' helps interpreter to ignore when it found a non Ascii character
```

21. type()

This function is used to print the type or the class the object passed as an argument belongs to. This function is used for debugging purposes.

Syntax –

```
type(Object)
```

It is also sometimes used to create a new object

Syntax-

```
type(name,bases,dict)
```

Code :

```
tupleObj=(3,4,6,7,9)

print(type(tupleObj))

new1 = type('New', (object, ),

dict(var1 ='LetsLearn', b = 2029))
print(type(new1))
```

22. callable()

This function returns True if the object passed as an argument is callable.

Syntax –

```
callable(Object)
```

Code:

```
def myFun():
   return 5
res = myFun
print(callable(res)) #function is called to get this value
num1 = 15 * 5
print(callable(num1))#no function is called
```

23. input()

This function helps python to take input from the user. In python 2.7, Its name is raw_input() which has been changed to input(). Once enter or esc is pressed system is resumed again.

Syntax –

```
input()
```

24. range()

This function returns the series of numbers between 2 specific numbers. This is very useful while dealing with a loop in a program as it helps to run a loop in a specific number of times. In python 3.6 – xrange() has been renamed as the range().

```
range(start,stop,step)
```

Here, start- an Integer that marks the start of the series.

A stop-an integer that marks the last number of the series. The last number in the range is stop-1.

Step – an integer that lets to increment the loop with a specific number. By default, it is +1.

Code:

```
res = 1
for i in range(1, 10,2):
    res = res * i
print("multiplication of first 10 natural number :", res)
```

Note- Float numbers as an argument throws an error.

25. reversed()

This function returns an iterator to access the collection in reverse order.

Syntax–

```
reversed([sequence] or [collection])
```

Code:

```
tupleObj=(3,4,6,7,9)
```

```
for i in reversed(tupleObj):
    print(i,end=' ')
```

Most used Functions and Methods(LIST):

1. List.len()	To find the length of the list
2. List.append()	Add an element at the end of the list.
3. List.insert()	To add an element at a specified index.
4. List.remove()	To remove a specified element from the list.
5. List.reverse()	Returns the reverse ordered list.
6. List.index()	Returns the first-found index of the specified element.
7. List.extend()	Adds the elements of another list as elements to the specified list.
8. List.pop()	Removes the returns of the last element of the specified list.
9. String.split()	Converts the given string into a list

Example:(Record and Observation)

```
#Creating a list
emptylist = []
mylist = [20, 20, 'H', "Hello"]
nestedlist = [[1, 2], mylist]
print("Created lists:")
print("emptylist:", emptylist)
print("mylist:", mylist)
print("nestedlist:", nestedlist)
print("Concatenating mylist and nestedlist:", mylist + nestedlist)
print("Repeating the elements of a list:", mylist*3)
#List as an input
print("\nList as an input:",end="")
```

```
inputlist = list(map(int, input("Enter elements(separate by space):").split()))
print(inputlist)
#Accessing elements
print("In the nested list: ")
print("(Normal)nestedlist[0]:", nestedlist[0])
print("(Negative)nestedlist[-2]:", nestedlist[-2])
#adding elements
print()
print("Adding elements into empty list:")
emptylist.append(1)
for i in range(3, 6):
    emptylist.append(i)
print("emptylist:", emptylist)
emptylist.insert(1, 2)
print("Adding at index emptylist[1]:", emptylist)
print("Extending mylist with emptylist:", mylist.extend(emptylist))
#Using Slicing
print("Slicing mylist[:]:", mylist[:])
print("Reverse using slicing[::-1]:", mylist[::-1])
print("Slicing using indices[1:3]:", mylist[1:3])
#list comprehension
print()
print("Creating a newlist[] using list comprehension:")
newlist = [i for i in emptylist if i%2 == 0]
print("newlist with even elements in emptylist:", newlist)
#Using the built-in functions
print("\nUsing functions:")
print("Length using len():", len(mylist))
print("Removing an element using remove:", mylist.remove(20))
print("Removing the last element using pop:", mylist.pop())
print("Using index():",mylist.index(3))
print("Using reverse on",emptylist,":")
emptylist.reverse()
print(emptylist)
```

Most used Functions and Methods(Tuples):

1. Tuple.len()	To find the length of the tuple
2. sorted(Tuple)	Sorts the specified tuple
3. max(Tuple)	To find the maximum valued element in the tuple.
4. min(Tuple)	To find the minimum valued element in the tuple.
5. sum(Tuple)	To find the sum of the elements in the tuple.
6. Tuple.index()	Returns the first-found index of the specified element.
7. all(Tuple)	Returns True if all the elements in the tuple are True.
8. any(Tuple)	Returns true if atleast one element in the tuple is True.
9. Tuple.count(element)	Returns the number of occurrences of the specified element in the tuple.

Example:(Record and Observation)

```
#Creating a tuple
emptytuple = ()
mytuple = (1, 'h', "Hello")
nestedtuple = ([1, 2], mytuple)
print("Emptytuple:",emptytuple)
print("mytuple:",mytuple)
print("nestedtuple:",nestedtuple)
mytuple1 = 1,'Hi'
print("By Tuple packing:", mytuple1)
list1 = [1, 2, 3]
print("Using tuple():", tuple(list1))
tuple2 = (1,)
print("Tuple with one element:", tuple2)
print("Concatenating nestedtuple and mytuple:", nestedtuple + mytuple)
print("Repeating the elements of a tuple:", mytuple*3)
#Tuple as an input
```

```
print("\nTuple as an input:",end="")
inputtuple = tuple(map(int, input("Enter elements(separate by space):").split()))
print(inputtuple)
#Using Slicing
print("\nSlicing mytuple[:]:", mytuple[:])
print("Reverse using slicing[::-1]:", mytuple[::-1])
print("Slicing using indices[1:3]:", mytuple[1:3])
#Accessing elements
print("\nAcccessing elements:")
print("In nestedtuple[0]:", nestedtuple[0])
print("In nestedtuple[-2]:", nestedtuple[-2])
a, b, c = mytuple
print("By tuple unpacking:",a,b,c)
#Using the built-in functions
print("\nUsing the built-in functions: ")
print("Length of mytuple:", len(mytuple))
print("Sorting a tuple using sorted():", sorted((2, 1, 0)))
print("Using max():", max((8, 23, 1)))
print("Using min():", min((3, 1, 2)))
print("Using sum():", sum((3, 4, 3)))
print("Using all():", all((3, 0, 9)))
print("Using any():", any((2, 0, 9)))
print("Using count():", (3, 1, 2, 1).count(1))
```

Most used Functions and Methods on Sets:

Function/ method	Equivalent	Explanation

	operator	
1. Set.add(value)		Adds the specified element into a set.
2. Set1.union(Set2)	Set1 | Set2	Merges two given sets
3. Set1.intersection(Set2)	Set1 & Set2	Finds the common elements of the given two sets.
4. Set1.difference(Set2)	Set1 - Set2	Finds elements in Set1 that are not in Set2
5. Set.clear()		Empties the given set.
6. set()		To create a set or to convert other data types into a set.
7. Set1.isdisjoint(Set2)		To check if Set1 and Set2 have no elements in common.

Set Operators:

Equality: 1. Set1 == Set2 2. Set1 != Set2	Checks if: 1. Set1 is equal to Set2 2. Set1 is not equal to Set2
Subset: 1. Set1 <= Set2 2. Set1 < Set2	Checks if: 1. Set1 is a subset of Set2 2. Set1 is a proper subset of Set2
Superset: 1. Set1 >= Set2 2. Set1 > Set2	Checks if: 1. Set1 is a superset of Set2 2. Set1 is a proper subset of Set2
Set1 ^ Set2	Returns a set of elements in either Set1 or Set2 but not in both sets.

Example:(Record and Observation)

```python
#Creating a set
print("Creating a set:")
emptyset = set()
print("Empty set:", emptyset)
myset = {1, 2, 3}
print("Myset:", myset)
nestedset = {(1, 2), 3, 4}
print("Nestedset:", nestedset)
set1 = set([1, 2, 1])
print("Removing duplicacy using set():", set1, end = "")
inputset = set(map(int, input("Enter elements: ").split()))
print("inputset:", inputset)
#Frozen sets
myfrozenset = frozenset([1, 2, 3, 4, 5])
print("\nFrozen set:", myfrozenset)
#Using functions
emptyset.add('a')
print("\nAdding to emptyset:",emptyset)
print("Union of", myset,"and", nestedset,":")
print(myset.union(nestedset))
print("Intersection of", myset,"and", nestedset,":")
print(myset.intersection(nestedset))
print("Difference of", myset,"and", nestedset,":")
print(myset.difference(nestedset))
emptyset.clear()
print("Clearing emptyset:", emptyset)
#Set operators
print("\nMembership operator to check if (1, 2) is in nestedset:",(1, 2) in nestedset)
print("Equivalency on myset and nested set:", myset == nestedset)
print("Subset operator to check if {1} is a subset of nestedset:", {1} <= nestedset)
```

31. print("proper subset operator to check if {1} is a proper subset of nesteds et:", {1} < nestedset)

32. print("Superset operator to check if nestedset is a superset of {1}:", neste dset >= {1})

33. print("proper superset operator to check if nestedset is a proper superset o f {1}:", nestedset > {1})

34. print("Elements in either myset or nestedset but not in both:", myset ^ nes tedset)

Most used dict Functions and Methods(Dictionary)

1. keys()	Returns a list of keys in the dictionary
2. values()	Returns a list of values in the dictionary
3. items()	Returns a list of key, value pairs as tuples in the dictionary
4. pop()	Deletes and returns the value at the specified key
5. dict()	Creates a dictionary and converts a list of key-value tuples into a dictionary.
6. get()	Returns the value at the given key
7. update()	Extends the dictionary with the given key: value pairs.

Example:(Record and Observation)

1. #Creating a dictionary

```
emptydict = {}
mydict1 = {1: 'A', 2: 'B', 3: 'C'}
mydict2 = {4: 'D', 'E': 5}
mydict3 = dict([(6, 'F'), (7, 'G')])
nesteddict = {1: mydict1, 2: mydict2}
inputdict = {}
#Taking dict as input
keys = []
keys = map(int, input("Enter keys: ").split(" "))
for i in keys:
    inputdict[i] = input("Enter value for "+ str(i) +": ")
print("Created dictionaries:")
print("Emptydict:", emptydict)
print("mydict1:", mydict1)
print("mydict2:", mydict2)
print("mydict3:", mydict3)
print("nesteddict:", nesteddict)
print("inputdict:", inputdict)
#Operations on dictionary
mydict2['E'] = 'E'
mydict2[5] = mydict2['E']
del mydict2['E']
print("\nAltering mydict2:", mydict2)
mydict3[8] = 'H'
mydict3[9] = 'I', 'J', 'K'
print("Adding elements to mydict3:", mydict3)
#Using built-in functions
print("\nUsing get() to access: mydict1.get(1): ", mydict1.get(1))
print("Using keys() on",mydict1,":", mydict1.keys())
print("using values() on",mydict1,":", mydict1.values())
print("Using items() on",mydict1,":", mydict1.items())
print("Updating",mydict1,"with",mydict2,":")
mydict1.update(mydict2)
print(mydict1)
print("Deleting a value using pop():", mydict1.pop(2), mydict1)
```