Operators

Simple definition : Operators can be symbols or keywords by using which we can do operation on the data.

In python there are around seven types of operators :

- Arithmetic operator
- Assignment operator
- Comparision operator
- logical operator
- bitwise operator
- membership operator
- identity operator

Arithmetic operator

Symbols by using which we can perform Arithmetic operations on the data.

Operator	Name	Example
+	Addition	x+y
-	Subtraction	x-y
*	Multiplication	x*y
/	Division	x/y
%	Modulus	x%y
**	Exponentiation	x**y

// Floor Division x//y

here x and y are operands (operation is done on them)

% → modulus will return the remainder value.

/ → division will return the quotient value.

// → floor division will only return the integer loosing the decimal value.

** → exponentiation is to the power of.

* '+' when used between strings is same as concatenation.

ex:- print("jessica" + " " + "cambell")

is jessica cambell

* '*' when used between strings is same as repetition.

ex:- print(jess * 3)

is jessjessjess

print(2 * (1,2)) or print((1,2) * 2)

↳ (1, 2, 1, 2)

Assignment

Create a simple calculator taking the two dynamic inputs and getting the Arithmetic operations.

```
x = int(input("Type the first number: "))
y = int(input("Type the second number: "))
"{} + {} = {}".format(x, y, x+y)
"{} - {} = {}".format(x, y, x-y)
"{} * {} = {}".format(x, y, x*y)
"{} / {} = {}".format(x, y, x/y)
"{} % {} = {}".format(x, y, x%y)
"{} ** {} = {}".format(x, y, x**y)
"{} // = {}".format x x// y
```

Assignment operator

Assignment operators are symbols by using which we can do Assignment operation on data.

"=" Assign data to the variable so Assignment operator.

ex:- $x = 10$

if we want to write $x = x + 1$

↳ 11

we can reduce this code in python as

$x += 1$ (this means $x = x + 1$)

$x -= 1$ $x = x - 1$

$x *= 1$ $x = x * 1$)

so first arithmetic operation is done and then the assignment operator.

Here $+=, -=, *=, /=, \%=, **=, //=$ are all Assignment operators.

Comparision operator

Comparision operators are symbols by using which we can compare the data.

>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

Output will be boolean so either True or False.

Logical operator

Logical operators are the operators where we use keywords by using which we perform logical operation on the data.

and → True if all conditions are True

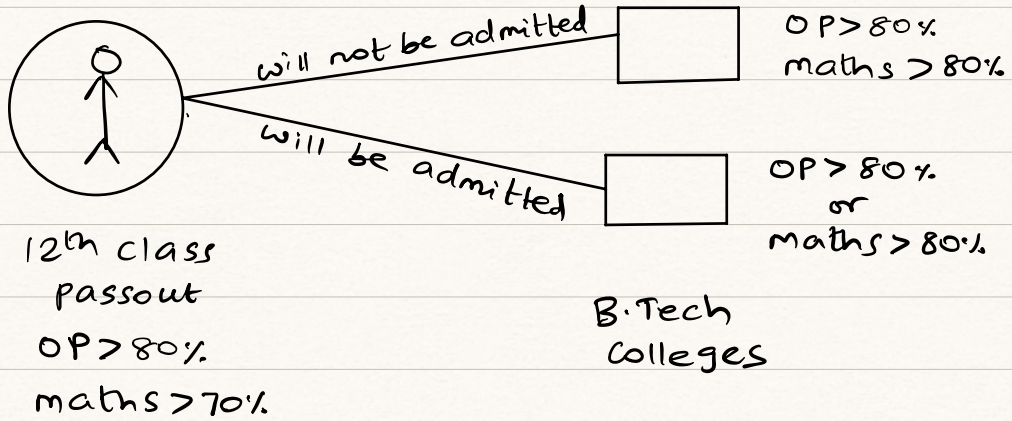
or → True if atleast one condition is True.

not → will reverse the original answer.

output will be boolean data type

- True
- or
- False

ex:-



* For and or or they should always be used between the conditions.

ex:- cond1 and cond2

Cond1 or cond2

* For 'and' if the first condition is False it will not go to the second condition.

* For or if the first condition is True it will stop and give the output as True.
* Not operator should be used at the start and not in between the conditions.

not (C_1 and C_2)

* Here if C_1 and C_2 is True then not output will be False and vice versa.

* In the case of arithmetic values
  - logical 'and' always returns second value.
  - logical 'or' always returns first value
  - logical 'not' always returns 'False' for non-zero value.

ex:- print (10 and 20)
↳ 20

print (10 or 20)
↳ 10

print (not 10)
↳ False

print (not 0)
↳ True.

Bitwise operator

In Python, bitwise operators are used to perform bitwise calculations on integers.
The integers are first converted in to binary and then operations are performed on bit by bit, hence the name bitwise operators.
Then the result is returned in decimal format.

* Python bitwise operators work only on integers once they are converted in to binary.

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x >>
<<	Bitwise left shift	x <<

Bitwise AND operator : Returns 1 if both the bits are 1 else 0.

ex:-

a = 10 = 1010 (Binary)

b = 4 = 0100 (Binary)

a & b = 0000

= 0 (Decimal)

Bitwise OR operator: Returns 1 if either of the bits is 1 else 0.

ex:-

$$a|b = 1010$$
$$\underline{0100}$$
$$1110$$
$$= 14 \text{ (Decimal)}$$

Bitwise NOT operator: Returns 1's complement of the number.

formula: 1's complement of a is $-(a+1)$

ex:-

a = 10 then ~a = -11

0000 1010 → 1's complement

1111 0101

$= -128 + 64 + 32 + 16 + 4 + 1$

$= -11$

Bitwise XOR operator: Returns 1 if one of the bits is 1 and other 0 else 0.

ex:-

$$a = 10 = 1010$$
$$b = 4 = \underline{0100}$$
$$a \wedge b = 1110$$

= 14 (Decimal)

Code

```
# Python Program to show bitwise operators.
a = 10
b = 4
# Print bitwise AND operation
Print ("a & b = ", a & b)
# Print bitwise OR operation
Print ("a | b = ", a | b)
# Print bitwise NOT operation
Print ("~a = ", ~a)
# Print bitwise XOR operation
print ("a ^ b = ", a ^ b)
```

Output

```
a & b = 0
a | b = 14
~a = -11
a ^ b = 14
```

Shift operators

These operators are used to shift the bits of a number left or right there by multiplying or dividing the number by 2 respectively. They can be used when we have to multiply or divide the number by two.

Bitwise right shift :

Shifts the bits of a number right and fills 0 on voids left (fills 1 in case of negative number) as a result.

* similar effect as dividing the number with some power of two.

ex:- a = 10 = 0000 1010 (Binary)

a>>1 = 0000 0101

= 5 (i.e 10/2) (Decimal)

a>>2 = 0000 0010

= 2

exi-

a = -10 = 1111 0110

a>>1 = 1111 1011

= -128 + 64 + 32 + 16 + 4 + 2

= -5

a>>2 = 1111 1101

$= -128+64+32+16+8+4+1$

$= -3$

Bitwise left shift : shifts the bits of the number to the left and fills 0 on voids right as a result. Simillar effect as of multiplying the number with some power of two.

ex:-

$a = 5 = 0000\ 0101$

$a<<1 = 0000\ 1010 = 10$

$a<<2 = 0001\ 0100 = 20$

ex:-

$b = -10 = 1111\ 0110$

$b<<1 = 1110\ 1100 = -20$

$b<<2 = 1101\ 1000 = -40$

Decimal to Binary Conversion

In binary ex:- 1101 1011 (Nibble, bit, Byte)

ex:- 57 → -128 64 32 16 8 4 2 1

0 0 1 1 1 0 0 1

= 0011 1001

1's Complement

~57 = 1100 0110

~57+1 = 1100 0110

0000 0001

1100 0111 = -57

So 2's complement of a number is the negative decimal number

So -57 = 1100 0111

= -128+64+4+2+1

= -57

So 2's complement method of binary methods is used here.

So range is -128 till +127

-128	64	32	16	8	4	2	1	
0	1	1	1	1	1	1	1	+127
1	0	0	0	0	0	0	0	-128

Binary addition

		carry	value
1 + 1	=	1	0
0 + 1	=		1
1 + 0	=		1
0 + 0	=		0
1 + 1 + 1	=	1	1

Binary substraction

1 − 1 = 0

1 − 0 = 1

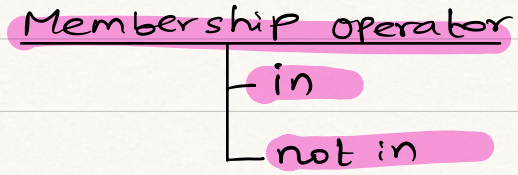
0 − 1 = Need to carry over

Ex:-

$$\begin{array}{r} {}^{0}\overset{\curvearrowright 2}{+}\,0 \\ -\,1 \\ \hline 0\;1 \end{array} \qquad \begin{array}{r} {}^{0}\overset{\curvearrowright 2}{+}\,0\;0 \\ -\,1\;0 \\ \hline 0\;1\;0 \end{array} \qquad \begin{array}{r} {}^{\overset{1}{\curvearrowright}\,2}\overset{\curvearrowright 2}{+}\,0\;0 \\ -\,0\;1 \\ \hline 0\;1\;1 \end{array}$$

Membership operator

They are the keywords by using which we can do membership operation on the data.



here 'in' and 'not' are keywords.

Membership operator checks whether a particular element is present in the data or not.

Output will be Boolean data type "True" or "False".

ex:- "a" in "abcd"

↳ True

"a" not in "abcd"

↳ False

'in' and 'not in' can be used for any data type.

ex:- 10 in [1, 'a', 'c', 2.2]

↳ False

They check whether 10 is a member of data or not.

* 'not in' returns 'True' if the object is not present in the given sequence, otherwise it returns 'False'.

Identity operators

Identity operator is used to check whether the value of two variables is same or not.
This operator is known as reference equality operator because the identity operator compares values according to two variables memory addresses.

Python has 2 identity operators 'is' and 'is not'.

'is' operator: The 'is' operator returns boolean 'True' or 'False'.
It returns 'True' if the memory address of the first value is equal to second value. otherwise it returns 'False'.

ex:- x = 10
y = 11
z = 10
print(x is y) → False
print(x is z) → True.

* Here we can use is() function to check whether both variables are pointing to the same object or not

'is not' operator: The 'is not' operator returns boolean 'True' or 'False'.

It returns 'True' if the memory address of the first value is not equal to second value. otherwise it returns 'False'.

ex:- x = 10

y = 11

z = 10

print(x is not y) → True

print(x is not z) → False

* Here we can use is() function to check whether both variables are pointing to the same object or not

Python Operator Precedence:

An expression is the combination of variables and operators that evaluate based on operator Precedence.

The following table shows operator precedence.

Highest to lowest

Precedence level	Operator	Meaning
1 (Highest)	( )	Parenthesis
2	**	Exponent
3	+x, -x, ~x	Unary addition, subtraction, 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	==, !=, >, >=, <, <=	Comparision
11	is, isnot, in, notin	Identity, Membership
12	not	Logical NOT
13	and	Logical AND
14	or	Logical OR

List, set, Dictionary Comprehension :

These comprehensions provide a much more short syntax for creating a new sequential data on the values of an existing sequential data.

List comprehension

Advantages :-

* more time-effecient and space-efficient than loops.
* require fewer lines of code.
* Transforms iterative statement in to a formula.

Syntax

```
newlist = [expression (element) for element in oldlist
                                    if condition]
```

ex:-

original

```
l = [ ]
for y in range (1,10)
    l.append (y)
```

list comprehension

```
[y for y in range(1,10)]
```

* we do not have to explicitly append anything to the list. It automatically creates a new list.

ex:- Even list using list comprehension

```
list = [y for y in range(11) if y%2==0]
```

↳ [2, 4, 6, 8, 10]

Squares list using list comprehension

```
list = [y**2 for y in range(11)]
```

↳ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

* Since tuples are immutable we will not have tuple comprehension.

* This same concept can be used for set comprehension.

Set Comprehension

ex:- `set = {y**2 for y in range(1,10)}`

Dictionary comprehension:

Dictionary comprehension is also similar to list comprehension but since dictionaries are pairs of keys and values we will have slight change in syntax.

```
dict = { key : value for (key, value) in iterable
                                    if condition }
```

ex:- create a dictionary for key in range(1, 11) and values are square of keys.

Normal code

```
d = { }
for y in range(1, 11)
    d.update ({y, y**2})
```

dictionary comprehension

```
{ y : y**2 for y in range(1, 11) }
```

↳ {1:1, 2:4, 3:9, ... 10:100}

dictionary comprehension for only even numbers

```
{ y : y**2 for y in range(1, 11) if y%2 == 0 }
```

↳ {2:4, 4:16, 6:36, 8:64, 10:100}