Python Operators

Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

Operator	Description	Syntax
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when first operand is divided by the second	x % y
**	Power : Returns first raised to power second	x ** y

Arithmetic operators Examples

Input: a = 9, b = 4

- # Addition of numbers
  add = a + b
- # Subtraction of numbers
  sub = a - b
- # Multiplication of number
  mul = a * b
- # Division(float) of number
  div1 = a / b
- # Division(floor) of number
  div2 = a // b
- # Modulo of both number
  mod = a % b
- # Power
  p = a ** b

Output:

- 13
- 5
- 36
- 2.25
- 2
- 1
- 6561

Logical operators

Logical operators 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

Logical operators Examples

- **Input:**
  a = True
  b = False

- # Print a and b is False
- print(a and b)

Output:

False

- # Print a or b is True
- print(a or b)

True

- # Print not a is False
- print(not a)

False

Bitwise operators

Bitwise operators acts on bits and performs bit by bit operation.

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<<

Examples of Bitwise operators

Input: a = 10, b = 4

- # Print bitwise AND operation
  print(a & b)
- # Print bitwise OR operation
  print(a | b)
- # Print bitwise NOT operation
  print(~a)
- # print bitwise XOR operation
  print(a ^ b)
- # print bitwise right shift operation
  print(a >> 2)
- # print bitwise left shift operation
  print(a << 2)

Output:

- 0
- 14
- -11
- 14
- 2
- 40

Keywords

yield

- yield is used inside a function like a return statement. But yield returns a generator.
- Generator is an iterator that generates one item at a time. A large list of values will take up a lot of memory. Generators are useful in this situation as it generates only one value at a time instead of storing all the values in memory. For example,

g = (2**x for x in range(100))

will create a generator g which generates powers of 2 up to the number two raised to the power 99. We can generate the numbers using the next() function as shown below.

next(g) **Output:1**

next(g) **Output:2**

next(g) **Output:4**

next(g) **Output:8**

next(g) **Output:16**

And so on… This type of generator is returned by the yield statement from a function.

Example of Yield

```
def generator():
    for i in range(6):
        yield i*i
g = generator()
for i in g:
    print(i)
```

Output

0

1

4

9

16

25

Here, the function generator() returns a generator that generates square of numbers from 0 to 5. This is printed in the for loop.

lambda

- lambda is used to create an anonymous function (function with no name). It is an inline function that does not contain a return statement. It consists of an expression that is evaluated and returned. For example:

```
a = lambda x: x*2
for i in range(1,6):
    print(a(i))
```

Output

2

4

6

8

10

async, await

- The async and await keywords are provided by the asyncio library in Python. They are used to write concurrent code in Python.
- For example:

```
import asyncio
async def main():
    print('Hello')
    await asyncio.sleep(1)
    print('world')
```

To run the program, we use:

- asyncio.run(main())

finally

- finally is used with try…except block to close up resources or file streams.
- Using finally ensures that the block of code inside it gets executed even if there is an unhandled exception. For example:
- try:
  Try-block
except exception1:
  Exception1-block
except exception2:
  Exception2-block
else: Else-block
finally: Finally-block

- Here if there is an exception in the Try-block, it is handled in the except or else block. But no matter in what order the execution flows, we can rest assured that the Finally-block is executed even if there is an error. This is useful in cleaning up the resources.

assert

- assert is used for debugging purposes.
- While programming, sometimes we wish to know the internal state or check if our assumptions are true. assert helps us do this and find bugs more conveniently. assert is followed by a condition.
- If the condition is true, nothing happens. But if the condition is false, AssertionError is raised. For example:

```
a = 4
assert a < 5
assert a > 5
Traceback (most recent call last): File "<string>", line 301, in runcode File "<interactive input>", line 1, in <module> AssertionError
```

Break and continue

- break and continue are used inside for and while loops to alter their normal behavior.
- break will end the smallest loop it is in and control flows to the statement immediately below the loop. continue causes to end the current iteration of the loop, but not the whole loop.

Continue

- for i in range(1,11):
  ```
  if i == 5:
      continue
  print(I)
  ```

Output

1
2
3
4
6
7
8
9
10

Break

- for i in range(1,11):
  ```
  if i == 5:
      break
  print(i)
  ```

Output

1
2
3
4