Python
Machine Learning
Data Science

Data Science
AI
Machine Learning
R
TensorFlow
Big Data
Hadoop
Spark

Programming

C
C++
Data Structures
Java
Python
Django
Android

Projects

perform operations in Python

There are many different types of operators. When evaluating complex expressions like **5+2*4%6-1 and 13 or 3** one might easily get confused about in which order the operations will be performed.

This Python operator precedence article will help you in understanding how these expressions are **evaluated** and the **order of precedence** Python follows.

Python Operators Precedence Table

- ✦ Python Objects
- ✦ Python Polymorphism
- ✦ Python Inheritance
- ✦ Python Multiple Inheritance
- ✦ Python Constructors
- ✦ Python Iterators
- ✦ Python Iterables
- ✦ Python Decorators
- ✦ Python Generators
- ✦ Python Counter
- ✦ Python Exception Handling
- ✦ Python Errors & Exceptions
- ✦ Python Comments, Statements
- ✦ Python Installation
- ✦ Python Interpreter
- ✦ Python Function Arguments
- ✦ Python Recursion
- ✦ Python Methods
- ✦ Python Methods vs Functions
- ✦ Python Exec()
- ✦ Python Eval()

Here we have a table that is arranged in the ascending order of precedence of operators.

The new **Assignment expression (:=) operator** from Python **3.8** onwards has the **lowest precedence** while **parentheses()** have the **highest precedence**.

Operator	Description
:=	Assignment expression (Lowest precedence)
lambda	Lambda expression
if-else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT

- Python Repr()
- Python Modules
- Python pprint Module
- Python Sys Module
- Python Defaultdict Module
- Python OrderedDict Module
- Python Calendar Module
- Python Itertools Module
- Packages in Python
- Modules vs Packages
- Property Class in Python
- Python Date and Time
- Python Iterator vs Generator
- Python Assertion
- Python Deep Copy & Shallow C…
- Python Terminologies Part 1
- Python Terminologies Part 2
- Python Facts
- Python Career
- Become a Python Developer
- Python Demand in Market

<, <=, >, >=,	Comparison operators
!=, ==	Equality operators
in, not in, is, is not,	Identity operators, membership operators
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Left and right Shifts
+, –	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+x, -x, ~x	Unary plus, Unary minus, bitwise NOT
**	Exponentiation
await x	Await expression
x[index], x[index:index],	Subscription, slicing, call,

- ✦ Python Resume
- ✦ Python Libraries for Data Scientist
- ✦ Python Web Frameworks
- ✦ Data Visualization in Python
- ✦ Matplotlib Introduction
- ✦ Python Lambda Expression
- ✦ Python Read Write File
- ✦ Modules in Python
- ✦ Python Compilers & Interpreters
- ✦ Closures in Python
- ✦ Python NumPy Introduction
- ✦ Python for Machine Learning
- ✦ Python SciPy Introduction
- ✦ SciPy Installation

Python Projects

- ✦ Python Projects with Source Code

Python Interview Questions

- ✦ Python Interview Ques Part-1
- ✦ Python Interview Ques Part-2
- ✦ Python Interview Ques Part-3

x(arguments…), x.attribute	attribute reference
(expressions…), [expressions…], {key: value…}, {expressions…}	Binding or parenthesized expression, list display, dictionary display, set display
()	Parentheses (Highest precedence)

What are Expressions in Python?

Before we move further, let’s first understand what are expressions.

An expression is made with **combinations of variables**, **values**, **operators** and **function calls.** The Python interpreter

evaluates the valid expression.

Have a look at a very simple expression.

```
5 - 2
```

Output:

```
3
```

5-2 is an expression that contains a **single operator**.

However, an expression can also contain **multiple operators** and **operands**.

```
10-5/5
```

Output:

```
9.0
```

In this expression, it first

divided the 5/5 and then **subtracted** the result from 10 because, in Python, the division operator has **higher precedence** than subtraction.

Let’s look at this example:

```
(10-5)/5
```

Output:

```
1.0
```

Here, with the use of **parentheses**, we force the interpreter to first evaluate the expression **inside** the parentheses and then continue the overall evaluation.

Follow TechVidvan on

***Google* & Stay updated with latest technology trends**

Python Operators Precedence Rule – PEMDAS

You might have heard about the **BODMAS rule** in your school's mathematics class. Python also uses a similar type of rule known as **PEMDAS**.

P – Parentheses
E – Exponentiation
M – Multiplication
D – Division
A – Addition
S – Subtraction

The precedence of operators is

listed from **High to low**. To remember the abbreviations, we have a funny mnemonic **“Please Excuse My Dear Aunt Sally”**.

Now we apply the PEMDAS rule and evaluate the following expression –

```
((((6+4)*
2)-10)//2
)-4*2
```

Output:

```
-3.0
```

How did we get -3?

Let’s break down the evaluation:

(6+4) = 10

(10*2) = 20

(20-10) = 10

(10//2) = 5

4*2 = 8

5-8 = -3

Associativity of Operators in Python

If you observed the precedence table, you may have noticed that many cells had **more than one operator** which means that they have the **same precedence**.

So then, which will be evaluated first is managed by the **associativity of operators**.

1. Associative Operators

The associative operators are **division**, **multiplication**, **remainder**, etc. and the expressions will be evaluated from **left to right**. Almost all operators except the **exponentiation(**) operator** support left-to-right associativity.

Example 1:

Suppose **modulus(%)** and **division(/) operators** have the **same precedence**. So, if both operators are present in an expression, then the **left one** is evaluated **first**.

```
45 % 10 /
2
```

Output:

```
2.5
```

First, 45%10 gives 5 and then 5/2 gives us 2.5 as output. If this was evaluated from right to left, we would get a **different** output.

```
45%
(10/2)
```

Output:

```
0.0
```

Here, we **forced** the expression to evaluate from right to left.

Example 2:

The **exponentiation operator** evaluates from right to left.

```
2**2**3
```

Output:

```
256
```

If we want to see the output of left to right, we can use **parentheses**.

```
(2**2)**3
```

Output:

```
64
```

2. Non-Associative Operators

The comparison operator and the assignment operators do not support **associativity** which means that an expression like **10<20<30** doesn’t mean **(10<20)<30** or **10<(20<30)**. They both mean the same thing as they are evaluated from **left to right**.

The statement **10<20<30** means **10<20** and **20<30**. You can also chain the assignment operators in any order and they will behave the same way.

a = b = c = d will be same as **b = a = d = c** or **d = c = b = a**.

Short-Circuiting in Python Operators Precedence

As we saw how Python mostly evaluates the expression from **left-to-right**.

In expression with **‘and’**, **‘or’ operators**, Python uses **Short-Circuiting** which means that it will evaluate the right side only when it is **needed**.

You’ll understand this better with examples.

1. Short-circuiting with and/or

The **boolean operation** will stop executing when we arrive at the truth value of the expression.

- x or y: Evaluates y only when x is false.
- x and y: Evaluates y only when x is true.

```
0 or
“Hey” and
1
```

Output:

```
1
```

0 or “Hey” returns “Hey” “Hey” and 1 returns 1

2. Short-circuiting with all()/any()

The inbuilt functions **all()** and **any()** also **supports short-circuiting**.

- **all()** function checks that all statements should be ‘True’.
- So when the first ‘False’ statement occurs, it **stops** further

executing and returns **False**.

```
def short_circuit(i):
  print(“Executing”)
  return i

print(all(short_circuit(i) for i in [1,2,3,0,5,6] ) )
```

Output:

```
Executing
Executing
Executing
Executing
False
```

- **any()** function returns “True” if one of the statements is true.
- So, when the first ‘True’ statement occurs, we

don’t need to **execute** any further and simply return **“True”**.

```
def short_ciruit(i):
  print(“Executing”)
  return i

print(any(short_circuit(i) for i in [0,0,3,0,5,6] ) )
```

Output:

```
Executing
Executing
Executing
True
```

3. Short-circuiting with conditional operators

Conditional operators also follow short-circuiting. Let’s see it with an example.

```
def
check(i)

"Watch
how this
unfurls
with
condition
al
operators
like >
and <.
     Have
a look at
Python
Bitwise
Operator"

return i

print(5>2
0>check(5
0))
```

Output:

```
False
```

The statement stopped executing when the statement becomes false and it was no longer needed to execute it further so the **check(50) method** didn’t run.

Summary

In this article, we studied the important topic of Python **operators precedence table**. We understood the rules of operator precedence and how Python evaluates complex expressions. Some operators are **associative** while some are **non-associative**.

Later on, we saw more on **short-circuiting** that Python stops executing when it is sure of the result and thus it doesn’t need to execute code any further.

We work very hard to provide you quality material
Could you take 15 seconds and share your happy experience on **Google** | **Facebook**

Tags:

- associativity of operators in python
- operator precedence in python
- python operator precedence
- python operator precedence table
- python rules
- What are expressions in python

3 RESPONSES

Comments 3

Pingbacks 0

Aaj Kaal

🕓 September 16, 2020 at 8:12 pm

seems you have a typo in Short-circuiting with conditional operators

Reply

foobar

February 4, 2022 at 2:13 am

This is wrong! Left/right bitwise shifts are higher precedence that bitwise AND/OR/XOR.

Reply

uche agu

October 20, 2022 at 4:52 pm

how do you express 7*9+5*7/6 in python

Reply

LEAVE A REPLY

Comment *

Name * Email *

Post Comment

Data Science Tutorials

- Data Science Tutorial
- Machine Learning Tutorial
- AI Tutorial
- R Tutorial
- Python Tutorial
- TensorFlow Tutorial
- Big Data Tutorial
- Hadoop Tutorial
- Spark Tutorial

Programming Tutorials

- C Tutorial
- C++ Tutorial
- Data Structures Tutorial
- Java Tutorial
- Python Tutorial
- Django Tutorial

Trending Tutorials

- Android Tutorial
- Blockchain Tutorial
- Cloud Tutorial
- IoT Tutorial
- Projects [with source code]

About Us
Contact Us
Terms and Conditions
Privacy Policy
Disclaimer