Python → Data types and operations → Basic data types and operations → Integer arithmetic

22579 users solved this topic. Latest completion was 13 minutes ago.

Theory: Integer arithmetic

14 minutes 14 / 14 problems solved

Start practicing

In real life, we often perform arithmetic operations. They help us to calculate the change from a purchase, determine the area of a room, count the number of people in a line, and so on. The same operations are used in programs.

Current topic:

✓ Integer arithmetic Stage 1 17★ ···

Table of contents:

§1. Basic operations

Python supports basic arithmetic operations:

- addition `+`
- subtraction `-`
- multiplication `*`
- division `/`
- integer division `//`

The examples below show how it works for numbers.

```
print(10 + 10)   # 20
print(100 - 10)  # 90
print(10 * 10)   # 100
print(77 / 10)   # 7.7
print(77 // 10)  # 7
```

There is a difference between division `/` and integer division `//`. The first produces a floating-point number (like `7.7`), while the second one produces an integer value (like `7`) ignoring the decimal part.

Python raises an error if you try to divide by zero.

```
ZeroDivisionError: division by zero
```

§2. Writing complex expressions

Arithmetic operations can be combined to write more complex expressions:

```
print(2 + 2 * 2)  # 6
```

The calculation order coincides with the rules of arithmetic operations. Multiplication has a higher priority level than addition and subtraction, so the operation `2 * 2` is calculated first.

To specify an order of execution, you can use **parentheses**, as in the following:

```
print((2 + 2) * 2)  # 8
```

Like in arithmetic, parentheses can be nested inside each other. You can also use them for clarity.

The minus operator has a unary form that negates the value or expression. A positive number becomes negative, and a negative number becomes positive.

```
print(-10)  # -10
print(-(100 + 200))  # -300
print(-(-20))  # 20
```

§3. Other operations

The remainder of a division. Python modulo operator `%` is used to get the remainder of a division. It may come in handy when you want to check if a number is even. Applied to `2`, it returns `1` for odd numbers and `0` for the even ones.

```
print(7 % 2)  # 1, because 7 is an odd number
print(8 % 2)  # 0, because 8 is an even number
```

Here are some more examples:

```
# Divide the number by itself
print(4 % 4)     # 0
# At least one number is a float
print(11 % 6.0)  # 5.0
# The first number is less than the divisor
print(55 % 77)   # 55
# With negative numbers, it preserves the divisor sign
print(-11 % 5)    # 4
print(11 % -5)    # -4
```

Taking the remainder of the division by `0` also leads to `ZeroDivisionError`.

The behavior of the mod function in Python might seem unexpected at first glance. While `11 % 5 = 1` and `-11 % -5 = -1` when both numbers on the left are of the same sign, `11 % -5 = -4` and `-11 % 5 = 4` if we have one negative number. The thing is, in Python, the remainder always has the same sign as the divisor.

In the first case, `11 % -5 = -4`, as the remainder should be negative, we need to compare 15 and 11, not 10 and 11: `11 = (-5) * (-3) + (-4)`. In the second case, `-11 % 5 = 4`, the remainder is supposed to be positive: `-11 = 5 * (-3) + 4`.

Exponentiation. Here is a way to raise a number to a power:

```
print(10 ** 2)  # 100
```

This operation has a higher priority over multiplication.

§4. Operation priority

To sum up, there is a list of priorities for all considered operations:

1. parentheses
2. power
3. unary minus
4. multiplication, division, and remainder
5. addition and subtraction

As mentioned above, the unary minus changes the sign of its argument.

Sometimes operations have the same priority:

```
print(10 / 5 / 2)  # 1.0
print(8 / 2 * 5)   # 20.0
```

The expressions above may seem ambiguous to you, since they have alternative solutions depending on the operation order: either `1.0` or `4.0` in the first example, and either `20.0` or `0.8` in the second one. In such cases, Python follows a left-to-right operation convention from mathematics. It's a good thing to know, so try to keep that in mind, too!

§5. PEP time!

There are a few things to mention about the use of binary operators, that is, operators that influence both operands. As you know, readability matters in Python. So, first, remember to surround a binary operator with a single space on both sides:

```
number=30+12      # No!

number = 30 + 12  # It's better this way
```

Also, sometimes people use the break **after** binary operators. But this can hurt readability in two ways:

- the operators are not in one column,
- each operator has moved away from its operand and onto the previous line:

```
# No: operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

Mathematicians and their publishers follow the opposite convention in order to solve the readability problem. Donald Knuth explains this in his ***Computers and Typesetting*** series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations". Following this tradition makes the code more readable:

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

In Python code, it is **permissible** to break before or after a binary operator, as long as the convention is consistent locally. For new code, **Knuth's style** is suggested, according to PEP 8.

Report a typo

1870 users liked this theory. **63** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (33) Hints (0) Useful links (5) Show discussion