Previous: Python Integer

Next: Python Tuple ›

Home » Python Data Types » Python Float: Working With Floating-Point Numbers

Python Float: Working With Floating-Point Numbers

July 11, 2023

Besides integers, consisting of whole numbers without fractions, Python also offers us the float type. Perhaps you've already seen floats without realizing it. After all, when you divide two integers in Python, the result of that division is often of type float.

In this article, I'll explain what floats are and how to use them. We'll also look at an important limitation of floats that you need to be aware of.

Table of Contents [hide]

Beginners Python Course (2023) **Modules, Packages, And Virtual Environments (2023)**

What is a Python float?

a Python float is a numerical data type that represents a floating-point number. A floating-point number is a number with a decimal point or exponent notation, indicating that a number is a certain number of digits before and after the decimal point or exponent. For example, the number 1.23 is a floating-point number with one digit before the decimal point and two digits after the decimal point.

Creating floating point numbers

There are multiple ways to create floating-point numbers in Python. Most of the time, you will simply enter the number as-is:

```
# You can simply enter the number
f = 1.45
```

But you may want to convert a number or a string to a float using the float() function. The variable f will become 2.0 in all cases below:

```
# f will become 2.0 in all cases below
f = float(2.0)
f = float(2)
f = float("2")
```

You can also use scientific notation:

```
f = 1.45e3
# f is now 1450.0
```

Working with Python floats

Working with floats is not much different from working with integers. You can do the usual addition, subtraction, multiplication, and division.

```
x = 1.45
y = 4.51
# Add x and y
z = x + y
# Subtract x and y
z = x - y
# Multiply x and y
z = x * y
# Divide x and y
z = x / y
```

Round, floor, and upper

With floats, we often need to round numbers. For this, Python has the round function. The round function accepts two arguments: the number itself and an optional precision (the number of decimals). The default for this last number is 0, meaning that it will round to whole integers. Some examples:

```
>>> f = 1.4567
>>> round(f)
1
>>> round(f, 2)
1.46
>>> f = 1.54
>>> round(f)
2
>>> round(f, 1)
1.5
```

While round is available without importing, we need to do an import from the `math` library to `floor` and `ceil` a number.

The `floor` function rounds a float *down* to the nearest integer while the `ceil` function rounds a float *up* to the nearest integer. Here's an example of how you can import and use these functions:

```
from math import floor, ceil
# Round 1.23 down to the nearest integer
x = floor(1.23)  # x will be 1
# Round 1.23 up to the nearest integer
y = ceil(1.23)  # y will be 2
```

Comparisons

Just like other types of numbers, floats can be compared using comparison operators such as ==, !=, >, <, >=, and <=. For example, the following code uses the == operator to check if the float `1.23` is equal to the float `4.56`:

```
x = 1.23
y = 4.56
if x == y:
    print("x and y are equal")
else:
    print("x and y are not equal")
```

The Python float has limited precision

Let's do a little experiment. For extra dramatic effect, try this for yourself first:

```
>>> 0.3 + 0.1
0.4
>>> 0.1 + 3.8
3.9
>>> 0.1 + 0.2
# What is the output of this last statement?
```

Did you expect the output to be `0.3` like any normal person would? I don't blame you, but you're wrong! The output is actually `0.30000000000000004`. As absurd as this might seem, there's an explanation.

First of all, some fractional numbers are endless. E.g., when you divide 1 by 3, you can round the result to 0.33, 0.333333333, or 0.333333333333333333. No matter what precision you choose, there will always be more 3's. Since there's a limited number of bits to store a float number, many numbers are a best-effort approximation. So floating-point numbers have limited precision because they are stored in a finite amount of memory, sometimes leading to unexpected results when performing arithmetic operations on floating-point numbers.

Solving the limited precision

The limited precision of floats is a fundamental property of how floats are represented in computer memory. Therefore, it is not possible to completely eliminate the issue of limited precision when working with floats.

However, there are a few ways that you can mitigate the effects of limited precision when working with floats in Python:

1. Use the `decimal` module: The `decimal` module in the Python standard library provides support for decimal floating-point arithmetic. This allows you to work with decimal numbers that have fixed precision, which can be useful for applications that require more precise calculations than what is possible with regular floats. The downside: floats are much faster than using the decimal pacakge.
2. Use NumPy: The numpy module is a popular scientific computing library for Python that supports working with arrays of numbers. The numpy module provides a data type called numpy.float128 that allows you to work with higher-precision floating-point numbers than regular floats.
3. Round your floats: As mentioned earlier, you can use the `round` function to round a float to a specified number of decimal places. Limiting the number of decimal places used in your calculations can help reduce the strange effects of limited precision, but this method makes your calculations even more imprecise.

Converting floats to other data types

You can convert a float to other data types, such as `int` or `str`, using the built-in functions `int()` and `str()`. For example, the following code converts the float `1.23` to an integer and a string:

```
# Convert float to integer
my_int = int(1.22)  # my_int will be 1
# Convert float to string
my_str = str(2.23)  # my_str will be "2.23"
```

‹ Previous: Python Integer

Next: Python Tuple