Python Numbers, Type Conversion and Mathematics

programiz.com/python-programming/numbers

In this article, you'll learn about the different numbers used in Python, how to convert from one data type to the other, and the mathematical operations supported in Python.

Number Data Type in Python

Python supports integers, floating-point numbers and complex numbers. They are defined as `int` , `float` , and `complex` classes in Python.

Integers and floating points are separated by the presence or absence of a decimal point. For instance, 5 is an integer whereas 5.0 is a floating-point number.

Complex numbers are written in the form, `x + yj` , where *x* is the real part and *y* is the imaginary part.

We can use the `type()` function to know which class a variable or a value belongs to and `isinstance()` function to check if it belongs to a particular class.

Let's look at an example:

```
a = 5

print(type(a))

print(type(5.0))

c = 5 + 3j
print(c + 3)

print(isinstance(c, complex))
```

When we run the above program, we get the following output:

```
<class 'int'>
<class 'float'>
(8+3j)
True
```

While integers can be of any length, a floating-point number is accurate only up to 15 decimal places (the 16th place is inaccurate).

The numbers we deal with every day are of the decimal (base 10) number system. But computer programmers (generally embedded programmers) need to work with binary (base 2), hexadecimal (base 16) and octal (base 8) number systems.

In Python, we can represent these numbers by appropriately placing a prefix before that number. The following table lists these prefixes.

Number System	Prefix
Binary	'0b' or '0B'
Octal	'0o' or '0O'
Hexadecimal	'0x' or '0X'

Here are some examples

```
# Output: 107
print(0b1101011)

# Output: 253 (251 + 2)
print(0xFB + 0b10)

# Output: 13
print(0o15)
```

When you run the program, the output will be:

```
107
253
13
```

Type Conversion

We can convert one type of number into another. This is also known as coercion.

Operations like addition, subtraction coerce integer to float implicitly (automatically), if one of the operands is float.

```
>>> 1 + 2.0
3.0
```

We can see above that 1 (integer) is coerced into 1.0 (float) for addition and the result is also a floating point number.

We can also use built-in functions like `int()`, `float()` and `complex()` to convert between types explicitly. These functions can even convert from strings.

```
>>> int(2.3)
2
>>> int(-2.8)
-2
>>> float(5)
5.0
>>> complex('3+5j')
(3+5j)
```

When converting from float to integer, the number gets truncated (decimal parts are removed).

Python Decimal

Python built-in class float performs some calculations that might amaze us. We all know that the sum of 1.1 and 2.2 is 3.3, but Python seems to disagree.

```
>>> (1.1 + 2.2) == 3.3
False
```

What is going on?

It turns out that floating-point numbers are implemented in computer hardware as binary fractions as the computer only understands binary (0 and 1). Due to this reason, most of the decimal fractions we know, cannot be accurately stored in our computer.

Let's take an example. We cannot represent the fraction 1/3 as a decimal number. This will give 0.33333333... which is infinitely long, and we can only approximate it.

It turns out that the decimal fraction 0.1 will result in an infinitely long binary fraction of 0.000110011001100110011... and our computer only stores a finite number of it.

This will only approximate 0.1 but never be equal. Hence, it is the limitation of our computer hardware and not an error in Python.

```
>>> 1.1 + 2.2
3.3000000000000003
```

To overcome this issue, we can use the decimal module that comes with Python. While floating-point numbers have precision up to 15 decimal places, the decimal module has user-settable precision.

Let's see the difference:

```
import decimal

print(0.1)

print(decimal.Decimal(0.1))
```

Output

```
0.1
0.1000000000000000055511151231257827021181583404541015625
```

This module is used when we want to carry out decimal calculations as we learned in school.

It also preserves significance. We know 25.50 kg is more accurate than 25.5 kg as it has two significant decimal places compared to one.

```
from decimal import Decimal as D

print(D('1.1') + D('2.2'))

print(D('1.2') * D('2.50'))
```

Output

```
3.3
3.000
```

Notice the trailing zeroes in the above example.

We might ask, why not implement `Decimal` every time, instead of float? The main reason is efficiency. Floating point operations are carried out must faster than `Decimal` operations.

When to use Decimal instead of float?

We generally use Decimal in the following cases.

- When we are making financial applications that need exact decimal representation.
- When we want to control the level of precision required.
- When we want to implement the notion of significant decimal places.

Python Fractions

Python provides operations involving fractional numbers through its `fractions` module.

A fraction has a numerator and a denominator, both of which are integers. This module has support for rational number arithmetic.

We can create Fraction objects in various ways. Let's have a look at them.

```
import fractions

print(fractions.Fraction(1.5))

print(fractions.Fraction(5))

print(fractions.Fraction(1,3))
```

Output

```
3/2
5
1/3
```

While creating `Fraction` from `float`, we might get some unusual results. This is due to the imperfect binary floating point number representation as discussed in the previous section.

Fortunately, `Fraction` allows us to instantiate with string as well. This is the preferred option when using decimal numbers.

```
import fractions

# As float
# Output: 2476979795053773/2251799813685248
print(fractions.Fraction(1.1))

# As string
# Output: 11/10
print(fractions.Fraction('1.1'))
```

Output

```
2476979795053773/2251799813685248
11/10
```

This data type supports all basic operations. Here are a few examples.

```
from fractions import Fraction as F

print(F(1, 3) + F(1, 3))

print(1 / F(5, 6))

print(F(-3, 10) > 0)

print(F(-3, 10) < 0)
```

Output

```
2/3
6/5
False
True
```

Python Mathematics

Python offers modules like `math` and `random` to carry out different mathematics like trigonometry, logarithms, probability and statistics, etc.

```
import math

print(math.pi)

print(math.cos(math.pi))

print(math.exp(10))

print(math.log10(1000))

print(math.sinh(1))

print(math.factorial(6))
```

Output

```
3.141592653589793
-1.0
22026.465794806718
3.0
1.1752011936438014
720
```

Here is the full list of functions and attributes available in the Python math module.

```
import random

print(random.randrange(10, 20))

x = ['a', 'b', 'c', 'd', 'e']

# Get random choice
print(random.choice(x))

# Shuffle x
random.shuffle(x)

# Print the shuffled x
print(x)

# Print random element
print(random.random())
```

When we run the above program we get the output as follows.(Values may be different due to the random behavior)

```
18
e
['c', 'e', 'd', 'b', 'a']
0.5682821194654443
```

Here is the full list of functions and attributes available in the Python random module.