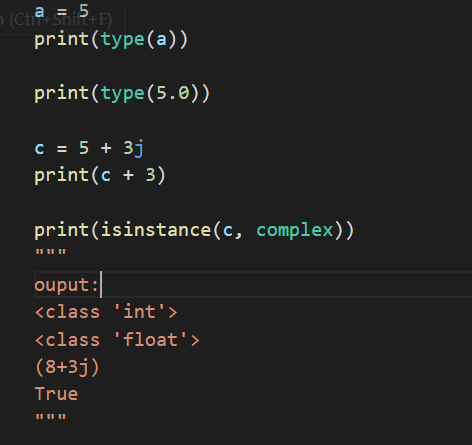
Number Data Type in Python

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.

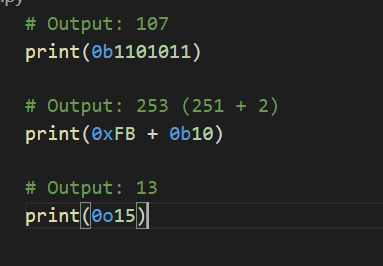


While integers can be of any length, a floating-point number is accurate only up to 15 decimal places (the 16th place is inaccurate).

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.

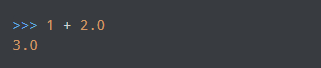
|  |  |
| --- | --- |
| Number System | Prefix |
| Binary | '0b' or '0B' |
| Octal | '0o' or '0O' |
| Hexadecimal | '0x' or '0X' |



## Type Conversion

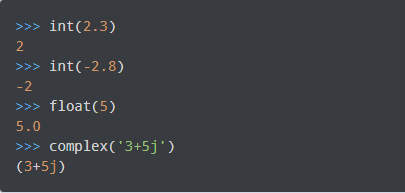
We can convert one type of number into another. This is also known as coercion.

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



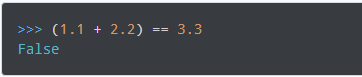
We can see above that 1 (integer) is coerced into 1.0 (float) for addition and the result is also a floating point number.

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



## 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.



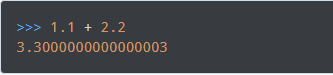
What is going on?

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.

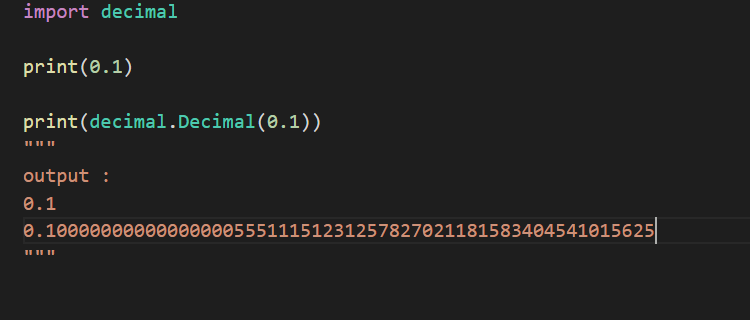
Let's take an example. We cannot represent the fraction 1/3 as a decimal number. This will give 0.33333333... which is infinitely long, and we can only approximate it.

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.



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.



### **When to use Decimal instead of float?**

We generally use Decimal in the following cases.

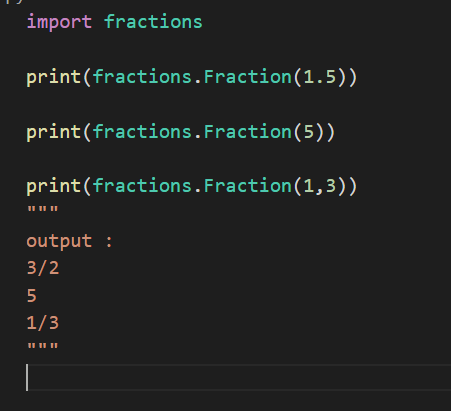
* When we are making financial applications that need exact decimal representation.
* When we want to control the level of precision required.
* When we want to implement the notion of significant decimal places.

## Python Fractions

Python provides operations involving fractional numbers through its fractions module.

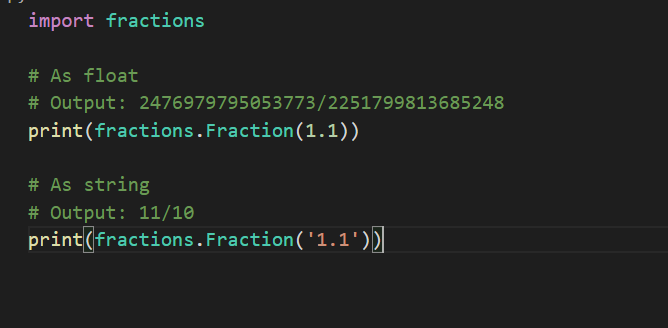
A fraction has a numerator and a denominator, both of which are integers. This module has support for rational number arithmetic.

We can create Fraction objects in various ways.



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.



To work more with math python also provided us with inbuilt module that has enumerous functions

Explore more of them here :