Python Numbers, Type Conversion and Mathematics

Python supports integers, floating point numbers and complex numbers. They are defined as `int`, `float` and `complex` class in Python. Integers and floating points are separated by the presence or absence of a decimal point. 5 is 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.

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> type(5.0)
<class 'float'>
>>> c = 5 + 3j
>>> c + 3
(8+3j)
>>> isinstance(c, complex)
True
>>> 1.1234567890123456789
1.1234567890123457
```

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

Numbers we deal with everyday are decimal (base 10) number system. But computer programmers (generally embedded programmer) 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. Following table lists these prefix.

Number system prefix for Python numbers	
Number System	**Prefix**
Binary	'0b' or '0B'
Octal	'0o' or '0O'
Hexadecimal	'0x' or '0X'

Here are some examples

```
>>> 0b1101011     # 107
107
>>> 0xFB + 0b10   # 251 + 2
253
>>> 0o15          # 13
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 operand 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 use built-in functions like `int()`, `float()` and `complex()` to convert between types explicitly. These function 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 (integer that is closer to zero).

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 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. Turns out decimal fraction 0.1 will result into an infinitely long binary fraction of 0.0001100110011001100110011... 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 `decimal` module that comes with Python. While floating point numbers have precision up to 15 decimal places, the decimal module has user settable precision.

```
>>> import decimal
>>> 0.1
0.1
>>> decimal.Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

This module is used when we want to carry out decimal calculations like 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
>>> D('1.1') + D('2.2')
Decimal('3.3')
>>> D('1.2') * D('2.50')
Decimal('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. 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.
- When we want the operations to be carried out like we did at school

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.

```
>>> import fractions
>>> fractions.Fraction(1.5)
Fraction(3, 2)
>>> fractions.Fraction(5)
Fraction(5, 1)
>>> fractions.Fraction(1,3)
Fraction(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, this class allows us to instantiate with string as well. This is the preferred options when using decimal numbers.

```
>>> fractions.Fraction(1.1)   # as float
Fraction(2476979795053773, 2251799813685248)
>>> fractions.Fraction('1.1') # as string
Fraction(11, 10)
```

This datatype supports all basic operations. Here are few examples.

```
>>> from fractions import Fraction as F
>>> F(1,3) + F(1,3)
Fraction(2, 3)
>>> 1 / F(5,6)
Fraction(6, 5)
>>> F(-3,10) > 0
False
>>> F(-3,10) < 0
True
```

Python Mathematics

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

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
>>> math.exp(10)
22026.465794806718
```

```
>>> math.log10(1000)
3.0
>>> math.sinh(1)
1.1752011936438014
>>> math.factorial(6)
720
```

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

```
>>> import random
>>> random.randrange(10,20)
16
>>> x = ['a', 'b', 'c', 'd', 'e']
>>> random.choice(x)
'd'
>>> random.shuffle(x)
>>> x
['a', 'b', 'd', 'c', 'e']
>>> random.random()
0.8025729372178537
```

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