**Numbers**

martes, 27 de septiembre de 2022

6:18 p. m.

Corey Schafer tutorial

Source: [Python Tutorial for Beginners 3: Integers and Floats - Working with Numeric Data](https://www.youtube.com/watch?v=khKv-8q7YmY&list=PL-osiE80TeTskrapNbzXhwoFUiLCjGgY7&index=3)

**PEP**: Python Enhancement Proposals

hierarchy of Number : > Complex : > Real : > Rational : > Integral where A :> B means “A is a supertype of B”.

Taken from PEP 3141 - A Type Hierarchy for Numbers

With numbers we could work the basic arithmetic operations ( + , - , \* , / ), but there are also some others not so familiar in the common language:

* Floor division ( // ): it drops the decimal part of the result and only displays the integer part

* Floor exponent ( \*\* ): it's actually the same to use ^ in common math.

* Floor modulo ( % ): it returns the remainder of a division. Let's say 6 divided by 4, it will only be 1 time in 6 and what's left will be 2, so 6 % 4 = 2.

Any number modulus itself = 0 & any number modulus 1 = 0.

The Modulo operator has a very useful functionality and is to check whether a number is even or odd. The way to do it is to apply the Modulo operation by 2, if there is any remainder after divide it means the number is odd, but if the reaminder is 0, the result is that the number is even.

Modulo division by two has only 2 results: 1 or 0 (Odd or Even).

Similar to += operator that sums up a variable with the other part of the operation, it could be used with the rest of the operators:

* -=
* \*=
* /=
* //=
* \*\*=
* %=

**round()** function: rounds up a float to the nearest integer, and can take up to 2 arguments, the first would be the floating number to round, and the second would be the number of decimals are needed on the result.

It's possible to turn a string into a integer by **casting** it, but it would only work if the string contains strictly whitespaces and numbers, the way to do it is to precede the variable or the string with the int( \_ )

Binary, Octal and Hexa

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' |

E.g.:

print(0b1101011)

# Output: 107

print(0xFB + 0b10)

# Output: 253 (251 + 2)

print(0o15)

# Output: 13

Expanding the work with number in python

Source: <https://www.programiz.com/python-programming/numbers>

There are three types of numeric data in python, int, float, and complex, which are actually classes in the language.

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

Is possible to access to the real and imaginary part of a complex number, accessing the attributes of the object.

Let's say:

a = 3 +5j

b = a.imag

c = a.real

If printed, b and c would return a floating point number of the a representing its imaginary and real part respectively

print(b) = 3.0

print(c) = 5.0

Float numbers are only accurate up to the 15th place.

Scientific notation

Is possible to use scientific notation (1x1012), by adding 'eX' where e represents the exponent and X represents the power of 10 for the notation. Note: the resulting number is coerced by float numbers.

l = 4e7

print(f'{l:,}') = 40,000,000.0

Or

l = 4e-3

print(f'{l:,}') = 0.004

Type conversion

Coercion is when we force a type of data into another. For example, when adding a integer with a floating point number, the result would be a floating point number and this happens because the language automatically coerce to float implicitly.

Python Decimals

Since there is fractions resulting in numbers infinitely long, like 1/3, due to hardware limitations the language automatically approximate, but to avoid this kinds of problems, to reach the top precision available at the moment is possible to work with the built-in module import decimal.

It's tempting to use the module in all cases, but the trade-off with floating operations is the efficiency. As float is a built-in class in Python it will run much faster.

There's three main cases to know when to use the module:

* 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 significance, with decimal places.

Python Fractions

There's also a module to work with fractional number import fractions. This module will as well handle the arithmetics of fractions.

Printing fractions directly from floats could raise some issue due to the imprecision discussed above, but this can be overcome printing the fraction from a string.

# As float

print(fractions.Fraction(1.1))

# Output: 2476979795053773/2251799813685248

# As string

print(fractions.Fraction('1.1'))

# Output: 11/10

Python Mathematics

Python has also built-in modules to handle math, trigonometry, logarithms, probability and statistics, etc. This is from the math module. And there's another more module to handle random numbers and operations, called random.

Here some examples:

import math

print(math.pi) = 3.141592653589793

print(math.cos(math.pi)) = -1.0

print(math.exp(10)) = 22026.46579480718

print(math.log10(1000)) = 3.0

print(math.sinh(1)) = 1.1752011936438014

print(math.factorial(6)) = 720

import random

print(random.randrange(10, 20)) = 12

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

# Get random choice

print(random.choice(x)) = a

# Shuffle x

random.shuffle(x)

# Print the shuffled x

print(x) = ['b', 'e', 'c', 'a', 'd']

# Print random element

print(random.random()) = 0.9876052270729684

Math built-in functions

**sum**(*iter*) function:

This function sums up all the elements within an iterable and returns the result

sum(range (30000000))= 449,999,985,000,000

**sum**(*iter*) vs for loop:

As the sum function is native from Python it will run significantly faster than a for loop, resulting in capacity efficiency.

c = 0

start = timer()

for i in range (30000000):

    c += i

stop = timer()

print(f'Result: {c:,}\nTime: {stop-start}') =

Result: 449,999,985,000,000

Time: 4.04999019999741

start = timer()

s = sum(range (30000000))

stop = timer()

print(f'Result: {s:,}\nTime: {stop-start}') =

Result: 449,999,985,000,000

Time: 0.6879388999986986

Up to 5x more efficient in this example.

**complex**() function:

This function receive a couple of parameters and return the complex number representation of it. It could take a tuple of numbers representing the real and imaginary part, or it could take as well a string with the literal of a complex number and return it as a complex number instance.

l = complex(3,5)

print(l) = 3 + 5j

or

l = complex('3+5j')

print(l) = 3 + 5j

**divmod**() function:

This function receive two parameters and return the quotient and the remainder of the division. It supports int and floats.

l = divmod(5,3)

print(l) = (1, 2)

or

l = divmod(5.1,3.7)

print(l) = (1.0, 1.3999)

Numbers built-in methods

Integers built-in methods

**print**(**dir**(**int**)) = [

'\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_bool\_\_', '\_\_ceil\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_float\_\_', '\_\_floor\_\_', '\_\_floordiv\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_index\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_int\_\_', '\_\_invert\_\_', '\_\_le\_\_', '\_\_lshift\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_or\_\_', '\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rand\_\_', '\_\_rdivmod\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rlshift\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_ror\_\_', '\_\_round\_\_', '\_\_rpow\_\_', '\_\_rrshift\_\_', '\_\_rshift\_\_', '\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_trunc\_\_', '\_\_xor\_\_',

'as\_integer\_ratio', 'bit\_count', 'bit\_length', 'conjugate', 'denominator', 'from\_bytes', 'imag', 'numerator', 'real', 'to\_bytes'

]

Floats built-in methods

**print**(**dir**(**float**)) = [

'\_\_abs\_\_', '\_\_add\_\_', '\_\_bool\_\_', '\_\_ceil\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_float\_\_', '\_\_floor\_\_', '\_\_floordiv\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getformat\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_int\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rdivmod\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_round\_\_', '\_\_rpow\_\_', '\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_setattr\_\_', '\_\_setformat\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_trunc\_\_',

'as\_integer\_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is\_integer', 'real'

]

]

Integer methods

.as\_integer\_ratio( ) method:

Note: This method is more intended to work with floats but, the 3.8 version included the functionality to work with integers as well. The method returns a tuple of integers whose ration is exactly equal to the original integer and with a positive denominator. For integers ratio the pair is always the numerator the same parameter integer and the denominator 1. *It's practically useless.*

.bit\_count( ) method:

This method will count the number of "1"s in the binary representation of an integer.

E.g.:

n = 25

print(bin(n)) = 0b11001

print(n.bit\_count()) = 3

.bit\_lenght( ) method:

This method will return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

E.g.:

n = 25

print(bin(n)) = 0b11001

print(n.bit\_lenght()) = 5

.to\_bytes( ) method:

This method will return an array of bytes representing the integer. It will take two arguments, one, the length of the resulting array and the second, the byteorder (either "big" or "little")

E.g.:

n = 25

print(n.to\_bytes(5,byteorder="big")) = b'\x00\x00\x00\x00\x19'

Integer attributes

To finish, there is also four attributes that can be accessed from an integer object, and these are: int.denominator, int.numerator, int.imag and int.real, and it will return the denominator (for int are always "1"), numerator (the same int accessed), the imaginary and real part respectively

Floats methods

.as\_integer\_ratio( ) method:

The method returns a tuple of integers whose ration is exactly equal to the original float and with a positive denominator.

f = 2.5

print(f.as\_integer\_ratio()) = (5, 2)

.hex( ) method:

Instance method - The method returns a string of the floating number hexadecimal representation with a leading 0x which represents the Hex numbers and a trailing p and exponent

f = 2.5

print(f.hex()) = 0x1.4000000000000p+1

.fromhex(s) method:

Class method - The method returns a float represented by a string of the number in hexadecimal representation. The string (s) must have a leading and trailing whitespace.

s = float.fromhex(' 0x1.4000000000000p+1 ')

print(s) = 2.5

Integer attributes

To finish, there is also two attributes that can be accessed from an float object, and these are int.imag and int.real, the imaginary and real part respectively.

Relevant Modules while working with Numbers

Documentation: <https://docs.python.org/3/library/numeric.html>

numbers Module

Besides the concept of ABCs (Abstract Base Classes) and the 'Numeric Tower': Number :> Complex :> Real :> Rational :> Integral  I didn't found anything worth writing to my current python knowledge extent

math Module

This module works with number excepting complex numbers, for this type exists the cmath module.

This module contains some functions that already exist in python standard library, but since Python were implemented in C, there are some operations that would raise inconsistencies in rounding results, therefore, in this module the precision is secured.

Next is a quick list of some of the functions within the module. This was taken from [programiz article](https://www.programiz.com/python-programming/modules/math) and left some functions and constants out, but to have a general idea of what it contains, a copy is left below.

|  |  |
| --- | --- |
| **Function** | **Description** |
| ceil(x) | Returns the smallest integer greater than or equal to x. |
| copysign(x, y) | Returns x with the sign of y |
| fabs(x) | Returns the absolute value of x |
| factorial(x) | Returns the factorial of x |
| floor(x) | Returns the largest integer less than or equal to x |
| fmod(x, y) | Returns the remainder when x is divided by y |
| frexp(x) | Returns the mantissa and exponent of x as the pair (m, e) |
| fsum(iterable) | Returns an accurate floating point sum of values in the iterable |
| isfinite(x) | Returns True if x is neither an infinity nor a NaN (Not a Number) |
| isinf(x) | Returns True if x is a positive or negative infinity |
| isnan(x) | Returns True if x is a NaN |
| ldexp(x, i) | Returns x \* (2\*\*i) |
| modf(x) | Returns the fractional and integer parts of x |
| trunc(x) | Returns the truncated integer value of x |
| exp(x) | Returns e\*\*x |
| expm1(x) | Returns e\*\*x - 1 |
| log(x[, b]) | Returns the logarithm of x to the base b (defaults to e) |
| log1p(x) | Returns the natural logarithm of 1+x |
| log2(x) | Returns the base-2 logarithm of x |
| log10(x) | Returns the base-10 logarithm of x |
| pow(x, y) | Returns x raised to the power y |
| sqrt(x) | Returns the square root of x |
| acos(x) | Returns the arc cosine of x |
| asin(x) | Returns the arc sine of x |
| atan(x) | Returns the arc tangent of x |
| atan2(y, x) | Returns atan(y / x) |
| cos(x) | Returns the cosine of x |
| hypot(x, y) | Returns the Euclidean norm, sqrt(x\*x + y\*y) |
| sin(x) | Returns the sine of x |
| tan(x) | Returns the tangent of x |
| degrees(x) | Converts angle x from radians to degrees |
| radians(x) | Converts angle x from degrees to radians |
| acosh(x) | Returns the inverse hyperbolic cosine of x |
| asinh(x) | Returns the inverse hyperbolic sine of x |
| atanh(x) | Returns the inverse hyperbolic tangent of x |
| cosh(x) | Returns the hyperbolic cosine of x |
| sinh(x) | Returns the hyperbolic cosine of x |
| tanh(x) | Returns the hyperbolic tangent of x |
| erf(x) | Returns the error function at x |
| erfc(x) | Returns the complementary error function at x |
| gamma(x) | Returns the Gamma function at x |
| lgamma(x) | Returns the natural logarithm of the absolute value of the Gamma function at x |
| pi | Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...) |
| e | mathematical constant e (2.71828...) |

decimal Module

Source: [Decimal Module in Python For Accurate Floats](https://www.youtube.com/watch?v=-4XI4B39k_U)

This module works to track with desired precision calculations, like moneywise.

The decimal module contains the Decimal class, this is something to remember, since is often used to call the generator to create a decimal type object. *The difference between the module and the class is that the module starts with low-cap d and the Class Generator with capitalized D.*

To avoid any confusions is better (for now) to unpack the whole module, like this:

from decimal import \*

Is also possible to set how precise the calculations will be, with this attribute assignment "getcontext().prec = X" where X is the number of decimal places desired

n = Decimal(20.45)

k = Decimal(4.2)

getcontext().prec = 5

print(n/k) = 4.8690

getcontext().prec = 10

print(n/k) = 4.869047619

There is a difference on how decimals are built, and one could pass a string as argument to do so or could also pass a float, and the difference between the two would be the innate precision due to how floating point numbers are store in computers

n = Decimal("0.1")

k = Decimal(0.1)

print(k==n) = False

fractions Module

Source: [Fractions Module In Python](https://www.youtube.com/watch?v=k5DixGC54fk)

This module works to specifically work with fractions but I didn't feel like it adds some value to my development learning code.

random Module

Source: <https://www.programiz.com/python-programming/modules/random>

[Python Tutorial: Generate Random Numbers and Data Using the random Module](https://www.youtube.com/watch?v=KzqSDvzOFNA&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=28)

Documentation: <https://docs.python.org/3/library/random.html>

This module works to generate pseudo-random number but since is sort of unsecure to encrypt and securing info and is recommended, by he Python documentation, that to use the module secrets for security purposes.

*"Python uses the Mersenne Twister as the core generator … The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes" .*

Module functions

random.getstate( ): Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state.

random.getstate(*state*): state should have been obtained from a previous call to getstate(), and setstate() restores the internal state of the generator to what it was at the time getstate() was called.

The module natively with the base method random() return a floating point number between 0 and 0,999…

For integers, there's a uniform selection from a range.

* The random.randrange(*stop*) or random.randrange(*start, stop, step*) produce the desired result:

print(random.randrange(10)) = 4

print(random.randrange(15,26)) = 19

* There's also the random.randint(*a, b*) which is similar to random.randrange(*a, b+1*), meaning the randint includes the upper limit in the selection:

print(random.randint(1,3)) = 3

For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

The random.choice(*seq*) returns a random element of a non-empty sequence, if the sequence is empty it'd raise an IndexError:

l = ['apple', 'orange', 'pineapple']

print(random.choice(l)) = apple

The random.choices(*population, weights=None, \*, cum\_weights=None, k=1*) returns a k sized list of elements chosen from the population with replacement. If the population is empty it'd raise an IndexError.

l = ['apple', 'orange', 'pineapple']

print(random.choices(l, cum\_weights=None, k=10)) = ['pineapple', 'pineapple', 'orange', 'pineapple', 'orange', 'orange', 'orange', 'pineapple', 'orange', 'pineapple']

Is possible to pass the relative or cumulative weight for each element of the iterable to be selected. The weights, if passed, must have the same length of the iterable, otherwise the method would raise a TypeError. *Apparently is more efficient if cumulative weight are passed, since the method would turn relative into cumulative one way or another.*

Let's say 'apple' has a 20 percent of chance to be selected and 'orange' and 'pineapple' have 30 and 40 percent respectively. Then, the argument should be passed " cum\_weights=[20, 30, 40] " if relative and " cum\_weights=[20, 50, 90] " if cumulative

Relative: The method sums up all the values and divide each item of the cum\_weights to assign its respective probability to be picked

print(random.choices(l, cum\_weights=[20, 30, 40], k=10)) = ['apple', 'apple', 'pineapple', 'apple', 'apple', 'pineapple', 'apple', 'pineapple', 'apple', 'apple']

Cumulative: The method sums up all the values and divide each item of the cum\_weights to assign its respective probability to be picked

print(random.choices(l, cum\_weights=[20, 50, 90], k=10)) = ['apple', 'apple', 'pineapple', 'orange', 'orange', 'orange', 'apple', 'pineapple', 'orange', 'orange']

The random.shuffle(*x[, random]*) shuffle the sequence x passed.

To shuffle an immutable sequence and return a new shuffled list, is recommended to use the random.sample*(x, k=len(x)*) method instead.

This method actually modify the argument, so let's suppose a deck of card numbered from 1 to 52 to illustrate this method. method instead.

deck = list(range (1,52+1))

print(deck) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52]

random.shuffle(deck)

print(deck) = [13, 52, 20, 15, 14, 4, 9, 12, 42, 21, 32, 50, 38, 43, 33, 47, 49, 31, 18, 19, 44, 51, 7, 24, 3, 28, 46, 22, 23, 34, 25, 2, 29, 41, 16, 10, 11, 27, 36, 45, 17, 6, 35, 30, 39, 1, 48, 40, 5, 26, 37, 8]

The random.sample(*population, k,\*, counts=None*) return a k length list of **unique** elements chose from *the population* sequence or set. Used for sampling without replacement.

It could be drawn either from the arguments passed or from another iterable (sequence).

From arguments

print(random.sample(['red', 'blue', 'white'], counts=[3, 2, 4], k=5)) = ['white', 'blue', 'red', 'white', 'red']

This is the same as to say random.sample(['red', 'red', 'red', 'blue', 'blue', 'white', 'white', 'white', 'white'], k=5) and if the sample size *k* is greater than the sample, it would raise a ValueError.

From Iterables

Let's say that we want to draw 5 card from the past example deck.

print(random.sample(deck, k=5)) = [22, 47, 11, 31, 41]

**Note**: To choose a sample from a range of integers, use a range() object as an argument. This is especially fast and space efficient for sampling from a large population: sample(range(10000000), k=60).

This module also offers modules to return numbers following some statistic distributions.

random.uniform(*a, b*): returns a floating point number N such that a <= N <= b. the end-point b may or may not be included in the range depending on floating-point rounding in the equation a + (b-a)\*random( ).

random.triangular(*low, high, mode*)

random.betavariate(*alpha, beta*)

random.expovariate(*lambd*)

random.gammavariate(*alpha, beta*)

random.gauss(*mu, sigma*)

random.lognormvariate(*mu, sigma*)

random.normalvariate(*mu, sigma*)

random.vonmisesvariate(*mu, kappa*)

random.paretovariate(*alpha*)

random.weibullvariate(*alpha, beta*)

And finally, is also possible to generate pseudo-random numbers with a given seed to the numbers generator. andom.random(*[seed]*) returns floating point numbers that are reproducible depending on the seed.

statistics Module

Source: [statistics module in Python | Python Modules | CBSE Class 11 and 12 Computer Science with Python](https://www.youtube.com/watch?v=xGqMSr3wtq4)

This module works to specifically work with statistic variables and parameters but I didn't feel like it adds some value to my development learning code.