**Random, Math & Decimals**

lunes, 16 de enero de 2023

7:47 p. m.

Relevant Modules while working with Numbers

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

* Statistics

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

Support: [The Python Math Module Explained..](https://www.youtube.com/watch?v=ZxJs4M0qPqA)

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.