Python's standard library

By standard library, we usually mean most used built-in modules and functions in Python. There are many of them, but we will focus only on few of them. Those who are interested can check the references at the bottom of this notebook.

1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system. This is a link to the module webpage.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

In [ ]:
```
import os
dir(os) # returns a list of all module functions
```

In [3]:
```
help(os.getcwd) # path to current working directory
```

```
Help on built-in function getcwd in module nt:

getcwd()
    Return a unicode string representing the current working directory.
```

In [4]:
```
help(os.listdir)
```

```
Help on built-in function listdir in module nt:

listdir(path=None)
    Return a list containing the names of the files in the directory.

    path can be specified as either str, bytes, or a path-like object.  If path is bytes,
      the filenames returned will also be bytes; in all other circumstances
      the filenames returned will be str.
    If path is None, uses the path='.'.
    On some platforms, path may also be specified as an open file descriptor;\
      the file descriptor must refer to a directory.
      If this functionality is unavailable, using it raises NotImplementedError.

    The list is in arbitrary order.  It does not include the special
    entries '.' and '..' even if they are present in the directory.
```

In [3]:
```
os.listdir()
```

Out[3]:
```
['.ipynb_checkpoints',
 'section 1.ipynb',
 'section 2.ipynb',
 'section 3.ipynb',
 'section 4.ipynb']
```

In [4]:
```
help(os.mkdir) # creates a directory/folder
```

```
Help on built-in function mkdir in module nt:

mkdir(path, mode=511, *, dir_fd=None)
    Create a directory.

    If dir_fd is not None, it should be a file descriptor open to a directory,
      and path should be relative; path will then be relative to that directory.
    dir_fd may not be implemented on your platform.
      If it is unavailable, using it will raise a NotImplementedError.

    The mode argument is ignored on Windows. Where it is used, the current umask
    value is first masked out.
```

In [5]:
```
os.mkdir("new_folder")
```

now let's check if new folder was created

In [6]:
```
os.listdir()
```

Out[6]:
```
['.ipynb_checkpoints',
 'new_folder',
 'section 1.ipynb',
 'section 2.ipynb',
 'section 3.ipynb',
 'section 4.ipynb']
```

we can check if there's smth in new folder, now we need to pass the path to the new folder

In [7]:
```
os.listdir("new_folder")
```

Out[7]:
```
[]
```

Let's create a new folder called `very_new_folder` inside the `new_folder`

In [8]:
```
os.mkdir("new_folder/very_new_folder")
```

In [9]:
```
os.listdir("new_folder")
```

Out[9]:
```
['very_new_folder']
```

2. Mathematics

The `math` module

The `math` module gives access to the underlying `C` library functions for floating-point math. This is a link to the module's webpage.

In [ ]:
```
import math
dir(math)
```

In addition to functions, `math` module includes some constants such as PI

In [12]:
```
math.pi
```

Out[12]:
```
3.141592653589793
```

Let's check the cosine function. We need to pass radians instead of degrees. Therefore, constant PI is useful here

In [13]:
```
math.cos(math.pi)
```

Out[13]:
```
-1.0
```

In [14]:
```
math.cos(0)
```

Out[14]:
```
1.0
```

In [17]:
```
math.cos(math.pi/4) # cosine of 45 degrees
```

Out[17]:
```
0.7071067811865476
```

There is square root function, using which we can verify above result

In [18]:
```
math.sqrt(2)/2
```

Out[18]:
```
0.7071067811865476
```

In [19]:
```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base is not specified, returns the natural logarithm (base e) of x.
```

In [21]:
```
math.log(1024, 10)
```

Out[21]:
```
3.0102999566398116
```

In [20]:
```
math.log(1024, 2)
```

Out[20]:
```
10.0
```

In the following case, we don't provide `base`, therefore default base of `e` is used making it natural logarithm

In [23]:
```
math.log(10)
```

Out[23]:
```
2.302585092994046
```

The `random` module

It is used to generate pseud-random numbers. Link to the module's page.

In [25]:
```
import random
```

In [26]:
```
help(random.choice)
```

```
Help on method choice in module random:

choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

In [27]:
```python
random.choice([1,2,3,4])
```

Out[27]: 3

In [28]:
```python
help(random.random)
```

```
Help on built-in function random:

random() method of random.Random instance
    random() -> x in the interval [0, 1).
```

In [29]:
```python
random.random()
```

Out[29]: 0.6737220523956396

In [30]:
```python
help(random.randint)
```

```
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

In [31]:
```python
random.randint(1,10)
```

Out[31]: 7

In [32]:
```python
for i in range(5):
    print(random.randint(1,100))
```

```
58
90
4
38
57
```

If I run the same code as above, I'm going to get different outputs

In [33]:
```python
for i in range(5):
    print(random.randint(1,100))
```

```
80
36
35
35
35
```

But sometimes we want to control the series of random numbers being generated.

`seed` is used for this purpose. It initializes the internal state of the module with given seed value.

In [35]:
```python
help(random.seed)
```

```
Help on method seed in module random:

seed(a=None, version=2) method of random.Random instance
    Initialize internal state from a seed.

    The only supported seed types are None, int, float,
    str, bytes, and bytearray.

    None or no argument seeds from current time or from an operating
    system specific randomness source if available.

    If *a* is an int, all bits are used.

    For version 2 (the default), all of the bits are used if *a* is a str,
    bytes, or bytearray.  For version 1 (provided for reproducing random
    sequences from older versions of Python), the algorithm for str and
    bytes generates a narrower range of seeds.
```

In [36]:

```
random.seed(1) # we can pass any number or even a string
for i in range(5):
    print(random.randint(1,100))
```

```
18
73
98
9
33
```

In [37]:

```
random.seed(1) # we can pass any number or even a string
for i in range(5):
    print(random.randint(1,100))
```

```
18
73
98
9
33
```

As you see now, we have the same sequence of random number generations from both cells.

If we pass another seed, the sequence will be different, but still repeatable with the same seed

In [38]:

```
random.seed(10) # we can pass any number or even a string
for i in range(5):
    print(random.randint(1,100))
```

```
74
5
55
62
74
```

In [39]:

```
random.seed(10) # we can pass any number or even a string
for i in range(5):
    print(random.randint(1,100))
```

```
74
5
55
62
74
```

The `statistics` module

It includes mathematical statistics functions and calculates basic statistical properties (the mean, median, variance, etc.) of numeric data. Link to the module's webpage.

In [ ]:
```
import statistics
dir(statistics)
```

In [61]:
```
help(statistics.mean)
```

```
Help on function mean in module statistics:

mean(data)
    Return the sample arithmetic mean of data.

    >>> mean([1, 2, 3, 4, 4])
    2.8

    >>> from fractions import Fraction as F
    >>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
    Fraction(13, 21)

    >>> from decimal import Decimal as D
    >>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
    Decimal('0.5625')

    If ``data`` is empty, StatisticsError will be raised.
```

In [62]:
```
random_nums = [random.random() for i in range(10)]
print(random_nums)
```

```
[0.07047152530229883, 0.44208190421490245, 0.03937679671032379, 0.9443919130253583, 0.067655358
62690975, 0.04014410806336721, 0.8896963679382558, 0.21609999659449042, 0.782361401688433, 0.72
87821037931089]
```

In [65]:
```
statistics.variance(random_nums)
```

Out[65]:
```
0.14441706836812923
```

3. Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware. Link to the module's page.

We can create a new date object by calling `date()` and passing year, month, and day information

In [66]:
```
import datetime
dir(datetime)
```

Out[66]:
```
['MAXYEAR',
 'MINYEAR',
 'UTC',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'date',
 'datetime',
 'datetime_CAPI',
 'time',
 'timedelta',
 'timezone',
 'tzinfo']
```

In [ ]:
```python
from datetime import date
help(date)
```

In [68]:
```python
birthday = date(2006, 1, 1)
birthday
```

Out[68]:
```
datetime.date(2006, 1, 1)
```

In [70]:
```python
today = date.today()
today
```

Out[70]:
```
datetime.date(2024, 10, 9)
```

In [71]:
```python
today - birthday
```

Out[71]:
```
datetime.timedelta(days=6856)
```

The `time` module

It provides access to several types of clocks, each useful for different purposes. The standard system calls such as `time()` report the system “wall clock” time. Link to the module's page.

Wall Clock Time

One of the core functions of the `time` module is `time()`, which returns the number of seconds since the start of the “epoch” as a floating-point value. The epoch is the start of measurement for time, which for Unix systems is 0:00 on January 1, 1970.

The float representation is highly useful when storing or comparing dates, but less useful for producing human-readable representations. For logging or printing times, `ctime()` can be a better choice.

In [72]:
```python
import time
time.time()
```

Out[72]:
```
1728470693.8903344
```

In [73]:
```
time.ctime()
```

Out[73]:
```
'Wed Oct  9 15:45:15 2024'
```

Monotonic clocks

Because `time()` looks at the system clock, and because the system clock can be changed by the user or system services for synchronizing clocks across multiple computers, calling `time()` repeatedly may produce values that go forward and backward. This can result in unexpected behavior when trying to measure durations or otherwise use those times for computation. To avoid those situations, use `monotonic()`, which always returns values that go forward. It returns number seconds since last boot.

In [74]:
```
time.monotonic()
```

Out[74]:
```
170299.968
```

4. Performance Measurement

Sometimes you want to know the relative performance of different approaches to the same problem. You can benchmark Python code using the `monotonic()` function.

But if we need more accurate timing with the highest possible resolution, we have to use `perf_counter()` function

In [75]:
```
def fibonacci_loop(n):
    f_0, f_1 = 0, 1
    if n == f_0 or n == f_1:
        return n
    else:
        for i in range(2, n+1):
            f_i = f_0 + f_1
            f_0 = f_1
            f_1 = f_i
        return f_i

start = time.perf_counter()
fibonacci_loop(100)
end = time.perf_counter()
print(end-start)
```

```
7.419998291879892e-05
```

References

1. Doug Hellmann, "The Python 3 Standard Library by Example," 2nd edition, O'reilly, Available online
2. Doug Hellmann, "Python 3 Module of the Week," Online blog
3. Python standard library: https://docs.python.org/3/library/index.html
4. Brief tour of the standard library: https://docs.python.org/3/tutorial/stdlib.html