PyLearn_PyTutorial_10_BriefTourStandardLib

May 3, 2020

#

Learning Python

1 The Python Tutorial -> Brief Tour of the Standard Library

Link: https://docs.python.org/3/tutorial/stdlib.html

1.1 Operating System Interface

[22]:
```
import os

msg = os.getcwd()                    # Return the current working directory
print(msg)

os.system('mkdir testing')    # Run the command mkdir in the system shell

os.chdir('testing')           # Change current working directory

msg = os.getcwd()             # Return the current working directory
print(msg)

os.chdir('..')

os.system('rmdir testing')

msg = os.getcwd()
print(msg)
```

```
/home/yp/dev/python/py-learning/PyLearn_PyTutorial_10_BriefTourStandardLib
/home/yp/dev/python/py-
learning/PyLearn_PyTutorial_10_BriefTourStandardLib/testing
/home/yp/dev/python/py-learning/PyLearn_PyTutorial_10_BriefTourStandardLib
```

> Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

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

```
import os
dir(os)
help(os)
```

For daily **file and directory management** tasks, the `shutil` module provides a higher level interface that is easier to use:

```
import shutil
shutil.copyfile('data.db', 'archive.db')
shutil.move('/build/executables', 'installdir')
```

1.2 File mWildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

[34]:
```
import glob
mylist = glob.glob('*.txt')
print(mylist)
mylist = glob.glob('*.*')
print(mylist)
```

```
['test_file.txt']
['test_file.txt', 'PyLearn_PyTutorial_10_BriefTourStandardLib.ipynb']
```

1.3 Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the **`sys`** module's **`argv`** attribute as a list. For instance the following output results from running `python3 test.py one two 3` at the command line:

Given `test1.py`

```
import sys
print(sys.argv)
```

[42]:
```
!python3 test1.py one two 3
```

```
['test1.py', 'one', 'two', '3']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

Given test2.py

```
import argparse

parser = argparse.ArgumentParser(prog = 'test2',
```

```
    description = 'Test2 program to test argparse')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

[45]:
```
!python3 test2.py --lines=5 alpha.txt beta.txt
```

```
Namespace(filenames=['alpha.txt', 'beta.txt'], lines=5)
```

1.4 Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when stdout has been redirected:

[48]:
```
import sys
sys.stderr.write('Warning, log file not found starting a new one\n')
```

```
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use :

```
sys.exit()
```

1.5 String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

[54]:
```
import re
mylist = re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
print(mylist)
mylist = re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
print(mylist)
```

```
['foot', 'fell', 'fastest']
cat in the hat
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

[56]:
```
s = 'tea for too'.replace('too', 'two')
print(s)
```

```
tea for two
```

1.6 Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

[58]:
```
import math
c = math.cos(math.pi / 4)
print(c)

lg = math.log(1024, 2)
print(lg)
```

```
0.7071067811865476
10.0
```

The `random` module provides tools for making random selections:

[67]:
```
import random

r = random.choice(['apple', 'pear', 'banana'])
print(r)

r = random.sample(range(100), 10)   # sampling without replacement
print(r)

r = random.random()         # random float
print(r)

r = random.randrange(6)     # random integer chosen from range(6)
print(r)
```

```
pear
[16, 91, 10, 3, 39, 82, 60, 85, 46, 52]
0.9204193334618603
3
```

The `statistics` module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

[69]:
```
import statistics
data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]

s = statistics.mean(data)
print(s)

s = statistics.median(data)
print(s)

s = statistics.variance(data)
print(s)
```

```
1.6071428571428572
1.25
1.3720238095238095
```

The SciPy project https://scipy.org has many other modules for numerical computations.

1.7 Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are: - `urllib.request` for retrieving data from URLs and - `smtplib` for sending mail:

Example test_urllib.py:

```
from urllib.request import urlopen
with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
    for line in response:
        line = line.decode('utf-8')  # Decoding the binary data to text.
        if 'Sun' in line or 'EDT' in line:  # look for Eastern Time
            print(line)
```

Example test_smtplib.py
(Note that the second example needs a mailserver running on localhost.)

```
import smtplib
server = smtplib.SMTP('localhost')
server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
"""To: jcaesar@example.org
From: soothsayer@example.org
Beware the Ides of March.
""")
server.quit()
```

1.8 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.

[82]:
```
# dates are easily constructed and formatted
from datetime import date

now = date.today()
print(now)
print(str(now))
print(repr(now))

s = now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
```

```
print(s)

# dates support calendar arithmetic
birthday = date(1964, 7, 31)
print(birthday)

age = now - birthday
print(age)
print(str(age))
print(repr(age))

print(age.days)
```

```
2020-05-03
2020-05-03
datetime.date(2020, 5, 3)
05-03-20. 03 May 2020 is a Sunday on the 03 day of May.
1964-07-31
20365 days, 0:00:00
20365 days, 0:00:00
datetime.timedelta(days=20365)
20365
```

1.9 Data Compression

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` and `tarfile`.

[89]:
```
import zlib
s = b'witch which has which witches wrist watch'
print( len(s) )

t1 = zlib.compress(s)
print( t )
print( len(t) )

s_dc = zlib.decompress(t)
print( s_dc )

t2 = zlib.crc32(s)
print( t2 )
```

```
41
b'x\x9c+\xcf,I\xceP(\xcf\xc8\x04\x92\x19\x89\xc5PV9H4\x15\xc8+\xca,.Q(O\x04\xf2\
x00D?\x0f\x89'
37
b'witch which has which witches wrist watch'
```

```
226805979
```

1.10 Performance Measurement

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The **timeit** module quickly demonstrates a modest performance advantage:
In contrast to **timeit**'s fine level of granularity, the **profile** and **pstats** modules provide tools for identifying time critical sections in larger blocks of code.

[93]:
```
from timeit import Timer
t1 = Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
print(t1)

t2 = Timer('a,b = b,a', 'a=1; b=2').timeit()
print(t2)
```

```
0.016219922341406345
0.01279264036566019
```

1.11 Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The **doctest** module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

[95]:
```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

[95]:
```
TestResults(failed=0, attempted=1)
```

The **unittest** module is not as effortless as the **doctest** module, but it allows a more comprehensive set of tests to be maintained in a separate file:

Given test_unittest.py

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.
    return sum(values) / len(values)

import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

if __name__ == '__main__':
    unittest.main()  # Calling from the command line invokes all tests
```

[105]:
```
!python3 test_unittest.py
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

1.12 Unit Testing in Python – Unittest

https://www.geeksforgeeks.org/unit-testing-python-unittest/

- **What is Unit Testing?**
  - Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed. The unittest test framework is python's xUnit style framework.
- **Method:**
  - White Box Testing method is used for Unit testing.
- **OOP concepts supported by unittest framework:**
  - **test fixture**:
    A test fixture is used as a baseline for running tests to ensure that there is a fixed environment in which tests are run so that results are repeatable. Examples :
    * creating temporary databases.
    * starting a server process.
  - **test case**:
    A test case is a set of conditions which is used to determine whether a system under test works correctly.
  - **test suite**:
    Test suite is a collection of testcases that are used to test a software program to show

that it has some specified set of behaviours by executing the aggregated tests together.
- **test runner**:
  A test runner is a component which set up the execution of tests and provides the outcome to the user.

1.13 Batteries Included

Python has a "batteries included" philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The `xmlrpc.client` and `xmlrpc.server` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other **RFC 2822**-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `json` package provides robust support for parsing this popular data interchange format. The `csv` module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the `xml.etree.ElementTree`, `xml.dom` and `xml.sax` packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The `sqlite3` module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.

2 END OF The Python Tutorial -> Brief Tour of the Standard Library

[ ]: