California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Tutorial

Section 10

Brief Tour of the Standard Library

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

10.1. Operating System Interface

- The `os` module provides dozens of functions for interacting with the operating system
- 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

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python39'
>>> os.chdir('/server/accesslogs')   # Change current working
directory
>>> os.system('mkdir today')   # Run the command mkdir in the
system shell
0
```

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

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's
docstrings>
```

- 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')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2. Wildcards

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

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. 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 python `demo.py one two three` at the command line

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

- 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
- When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
    description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

10.4. Error Output Redirection and Program Termination

- The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`
- `stderr` is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected
- The most direct way to terminate a script is to use `sys.exit()`

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

10.5. 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

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

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

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6. Mathematics

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

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

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

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

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

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

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

10.7. 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

```
>>> 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 'EST' in line or 'EDT' in line:  # look for Eastern
Time
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> 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()
```

- Note: the `smtplib` module needs a mailserver running on localhost

10.8. 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

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.8. Dates and Times (cont.)

- `class datetime.date(year, month, day)`
  - An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect
  - Attributes: year, month, and day
- `class datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)`
  - An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds
  - Attributes: hour, minute, second, microsecond, and tzinfo
- `class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)`
  - A combination of a date and a time
  - Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo
- `class datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)`
  - A duration expressing the difference between two date, time, or datetime instances to microsecond resolution
- `class datetime.tzinfo`
  - An abstract base class for time zone information objects
  - These are used by the datetime and time classes to provide a customizable notion of time adjustment
- `class datetime.timezone(offset, name=None)`
  - A class that implements the tzinfo abstract base class as a fixed offset from the UTC

10.8. Dates and Times (Format Codes)

Directive	Meaning	Example
%a	Weekday as locale's abbreviated name.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)
%A	Weekday as locale's full name.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6
%d	Day of the month as a zero-padded decimal number.	01, 02, …, 31
%b	Month as locale's abbreviated name.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)
%B	Month as locale's full name.	January, February, …, December (en_US); Januar, Februar, …, Dezember (de_DE)
%m	Month as a zero-padded decimal number.	01, 02, …, 12
%y	Year without century as a zero-padded decimal number.	00, 01, …, 99
%Y	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999

Directive	Meaning	Example
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)
%M	Minute as a zero-padded decimal number.	00, 01, …, 59
%S	Second as a zero-padded decimal number.	00, 01, …, 59
%f	Microsecond as a decimal number, zero-padded on the left.	000000, 000001, …, 999999
%z	UTC offset in the form±HHMM[SS[.ffffff]] (empty string if the object is naive).	(empty), +0000, -0400, +1030, +063415, -030712.345216
%Z	Time zone name (empty string if the object is naive).	(empty), UTC, GMT
%j	Day of the year as a zero-padded decimal number.	001, 002, …, 366
%U	Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, …, 53
%W	Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, …, 53

10.8. Dates and Times (cont.)

```
>>> import datetime
>>> month1 = 10; year1 = 2021; day1 = 9; hour1 = 13; minute1 = 7; second1 = 57
>>> #create datetime object
>>> dt = datetime.datetime(year1, month1, day1, hour1, minute1, second1)
>>> #create string of the form: October 9, 2021 1:07:57pm
>>> dt_string = dt.strftime('%B %d, %Y %I:%M:%S%p')
>>> print(dt_string)
October 09, 2021 01:07:57PM

>>> #strip leading zero from day
>>> str_day = dt.strftime('%d').lstrip('0')
>>> print(str_day)
9

>>> #strip leading zero from hour
>>> str_hour = dt.strftime('%I').lstrip('0')
>>> print(str_hour)
1

>>> #lower case am/pm
>>> str_ampm = dt.strftime('%p').lower()
>>> print(str_ampm)
pm

>>> #recreate string of the form: October 9, 2021 1:07:57pm
>>> dt_string = dt.strftime(f'%B {str_day}, %Y {str_hour}:%M:%S{str_ampm}')
>>> print(dt_string)
October 9, 2021 1:07:57pm
```

10.8. Dates and Times (cont.)

```
>>> import datetime
>>> #create datetime string
>>> dt_string = "October 09, 2021 01:07:57PM"
>>> #create datetime object
>>> dt_object = datetime.datetime.strptime(dt_string, '%B %d, %Y %I:%M:%S%p')
>>> print(dt_object)
2021-10-09 13:07:57
```

10.9. Data Compression

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

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. Performance Measurement

- Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem
- Python provides a measurement tool that answers those questions immediately
- 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

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

10.11. 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

```
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
```

- 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

```
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)

unittest.main()  # Calling from the command line invokes all tests
```