PyLearn_PyTutorial_11_BriefTourStandardLibPartII

May 4, 2020

#

Learning Python

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

Link: https://docs.python.org/3/tutorial/stdlib2.html

1.1 Output Formatting

The **reprlib** module provides a version of **repr()** customized for abbreviated displays of large or deeply nested containers:

[1]:
```
import reprlib
reprlib.repr(set('supercalifragilisticexpialidocious'))
```

[1]:
```
"{'a', 'c', 'd', 'e', 'f', 'g', …}"
```

The **pprint** ("pretty printer") module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the "pretty printer" adds line breaks and indentation to more clearly reveal data structure:

[2]:
```
import pprint
t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
    'yellow'], 'blue']]]

pprint.pprint(t, width=30)
```

```
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

The **textwrap** module formats paragraphs of text to fit a given screen width:

[5]:
```
import textwrap
doc = """The wrap() method is just like fill() except that it returns
a list of strings instead of one big string with newlines to separate
the wrapped lines."""

print(textwrap.fill(doc, width=40))
```

```
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The **locale** module accesses a database of culture specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

[11]:
```
import locale

locale.setlocale(locale.LC_ALL, '')

conv = locale.localeconv()          # get a mapping of conventions

x = 1234567.8

s = locale.format_string("%d", x, grouping=True)
print(s)

s = locale.format_string("%s%.*f", (conv['currency_symbol'],
                      conv['frac_digits'], x), grouping=True)
print(s)
```

```
1,234,567
$1,234,567.80
```

1.2 Templating

The **string** module includes a versatile Template class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$$` creates a single escaped `$`:

[25]:
```
from string import Template

t = Template('${village}folk send $$10 to $cause.')
s = t.substitute(village='Nottingham', cause='the ditch fund')
```

```
print(s)
```

```
<string.Template object at 0x7fa0a410b3a0>
Nottinghamfolk send $10 to the ditch fund.
```

The **substitute()** method raises a **KeyError** when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the **safe_substitute()** method may be more appropriate — it will leave placeholders unchanged if data is missing:

[27]:
```
from string import Template

t = Template('Return the $item to $owner.')
d = dict(item='unladen swallow')

s = t.safe_substitute(d)   # ok
print(s)

s = t.substitute(d)        # will raise a KeyError
print(s)
```

```
Return the unladen swallow to $owner.


␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-27-cc974bb91f7b> in <module>
  7 print(s)
  8
----> 9 s = t.substitute(d)
 10 print(s)


/opt/jupyterhub/lib/python3.8/string.py in substitute(self, mapping,␣
↪**kws)
124             raise ValueError('Unrecognized named group in pattern',
125                              self.pattern)
--> 126         return self.pattern.sub(convert, self.template)
127
128     def safe_substitute(self, mapping=_sentinel_dict, /, **kws):


/opt/jupyterhub/lib/python3.8/string.py in convert(mo)
117             named = mo.group('named') or mo.group('braced')
```

```
        118             if named is not None:
    --> 119                 return str(mapping[named])
        120             if mo.group('escaped') is not None:
        121                 return self.delimiter


        KeyError: 'owner'
```

Template subclasses can specify a **custom delimiter**. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

[28]:
```
import time, os.path

photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']

class BatchRename(Template):
    delimiter = '%'

# fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
fmt = "Ashley_%n%f"

t = BatchRename(fmt)

date = time.strftime('%d%b%y')

for i, filename in enumerate(photofiles):
    base, ext = os.path.splitext(filename)
    newname = t.substitute(d=date, n=i, f=ext)
    print('{0} --> {1}'.format(filename, newname))
```

```
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

1.3 Working with Binary Data Record Layouts

The `struct` module provides **`pack()`** and **`unpack()`** functions for working with variable length binary record formats.

The following example shows how to loop through header information in a ZIP file without using the `zipfile` module.

Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively.

The "<" indicates that they are standard size and in little-endian byte order:

[33]:
```
import struct

with open('SampleZipFile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

```
b'PyLearn_PyTutorial_10_BriefTourStandardLib.md' 0x57ed207d 5231 12753
b'PyLearn_PyTutorial_7_IO.md' 0x3ea86198 5818 22134
b'PyLearn_PyTutorial_8_ErrorsExceptions.md' 0xc6014d16 3731 13737
```

1.4 Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level **threading** module can run tasks in background while the main program continues to run:

[35]:
```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)
```

```
background = AsyncZip('myfile.html', 'myfile.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

```
The main program continues to run in foreground.
Finished background zip of: myfile.html
Main program waited until background was done.
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `queue` module to feed that thread with requests from other threads. Applications using `Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

1.5 Logging

The **`logging`** module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

[36]:
```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: DEBUG, INFO, WARNING, ERROR, and CRITICAL.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

1.6 Weak References

Python does automatic memory management (reference counting for most objects and **garbage collection** to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The **weakref** module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

[50]:
```
import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return str(self.value)

a = A(10)                   # create a reference
d = weakref.WeakValueDictionary()
d['primary'] = a            # does not create a reference
print( d['primary'] )       # fetch the object if it is still alive

del a                       # remove the one reference
gc.collect()                # run garbage collection right away

print( d['primary'] )       # entry was automatically removed, will raise␣
 ↪KeyError
```

```
10


      ␣
↪---------------------------------------------------------------------------

       KeyError                                  Traceback (most recent call␣
↪last)

       <ipython-input-50-eab9016091e7> in <module>
        15 gc.collect()                # run garbage collection right away
        16
   ---> 17 print( d['primary'] )       # entry was automatically removed, will␣
↪raise KeyError


       /opt/jupyterhub/lib/python3.8/weakref.py in __getitem__(self, key)
```

```
    129         if self._pending_removals:
    130             self._commit_removals()
--> 131         o = self.data[key]()
    132         if o is None:
    133             raise KeyError(key)


    KeyError: 'primary'
```

1.7 Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The **array** module provides an **array()** object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

[51]:
```
from array import array
a = array('H', [4000, 10, 700, 22222])
print( sum(a) )

print( a[1:3] )
```

```
26932
array('H', [10, 700])
```

The **collections** module provides a **deque()** object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

[52]:
```
from collections import deque
d = deque(["task1", "task2", "task3"])
d.append("task4")
print("Handling", d.popleft())
```

```
Handling task1
```

Sample breadth first tree searches code:

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the **bisect** module with functions for manipulating sorted lists:

[53]:
```
import bisect
scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
bisect.insort(scores, (300, 'ruby'))
print(scores)
```

```
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The **heapq** module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

[55]:
```
from heapq import heapify, heappop, heappush
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapify(data)                      # rearrange the list into heap order
heappush(data, -5)                 # add a new entry
list = [heappop(data) for i in range(3)]  # fetch the three smallest entries
print(list)
```

```
[-5, 0, 1]
```

1.8 Decimal Floating Point Arithmetic

The **decimal** module offers a **Decimal** datatype for decimal floating point arithmetic. Compared to the built-in float implementation of binary floating point, the class is especially helpful for - financial applications and other uses which require exact decimal representation, - control over precision, - control over rounding to meet legal or regulatory requirements, - tracking of significant decimal places, or - applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

[56]:
```
from decimal import *

d1 = round(Decimal('0.70') * Decimal('1.05'), 2)
print(d1)

d2 = round(.70 * 1.05, 2)
print(d2)
```

```
0.74
0.73
```

The **Decimal** result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the **Decimal** class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

[57]:
```
d = Decimal('1.00') % Decimal('.10')
print(d)

d = 1.00 % 0.10
print(d)

check = sum([Decimal('0.1')]*10) == Decimal('1.0')
print(check)

check = sum([0.1]*10) == 1.0
print(check)
```

```
0.00
0.09999999999999995
True
False
```

The **decimal** module provides arithmetic with as much precision as needed:

[58]:
```
getcontext().prec = 36
d = Decimal(1) / Decimal(7)
print(d)
```

```
0.142857142857142857142857142857142857
```

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