Python’s Standard Library

Josh Lawrence

Dennis Tran

Ramesh S

Operating System Interface

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

```
import os   #imports the os interface module
dir(os)     #returns a list of module's functions
help(os)    #returns an extensive manual page
```

Some os module functions:

```
os.getcwd()    # get current working directory
os.chdir('/usr/cs265') # change current working directory
os.system('mkdir lab1') # perform a mkdir in the system shell
```

Shutil module

- Use for daily file/directory management tasks

```
import shutil
#copies data.db to archive.db
shutil.copyfile('data.db', 'archive.db')
#move(source, destination)
shutil.move('/build/executables', 'installdir')
```

File 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']
```

Command Line Arguments

- Command line arguments are stored in the *sys* module’s *argv* attribute as a list:
- If at command line python demo.py one two three is executed:

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

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:

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

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

String Pattern Matching

- The re module provides regular expression tools for advanced string processing.
- r - Python’s raw string notation for regular expression patterns

```
re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
 ['foot', 'fell', 'fastest']
```

Mathematics

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

```
import math
math.cos(math.pi / 4.0)
0.70710678118654757
math.log(1024, 2)
10.0
```

Random

- The random module provides tools for making random selections:

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

Internet Access

- There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib2 for retrieving data from urls and smtplib for sending mail:

```
import urllib2
for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
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()
```

Dates and Times

- The datetime module supplies classes for manipulating dates and times in both simple and complex ways. 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
```

Data Compression

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

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

Performance Measurement

```
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
```

Quality Control

- The doctest module provides a tool for scanning a module and validating tests embedded in a program’s docstrings.

```
def average(values):
    """Computes the arithmetic mean of a list of
    numbers.
    >>> print average([20, 30, 70])
    40.0 """
    return sum(values, 0.0) / len(values)
import doctest
# automatically validate the embedded tests
doctest.testmod()
```

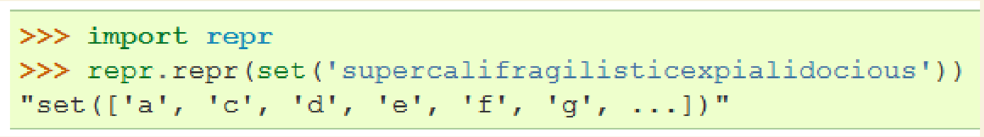
Quality Control

- The unittest module 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)
    self.assertRaises(ZeroDivisionError, average, [])
    self.assertRaises(TypeError, average, 20, 30, 70)

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

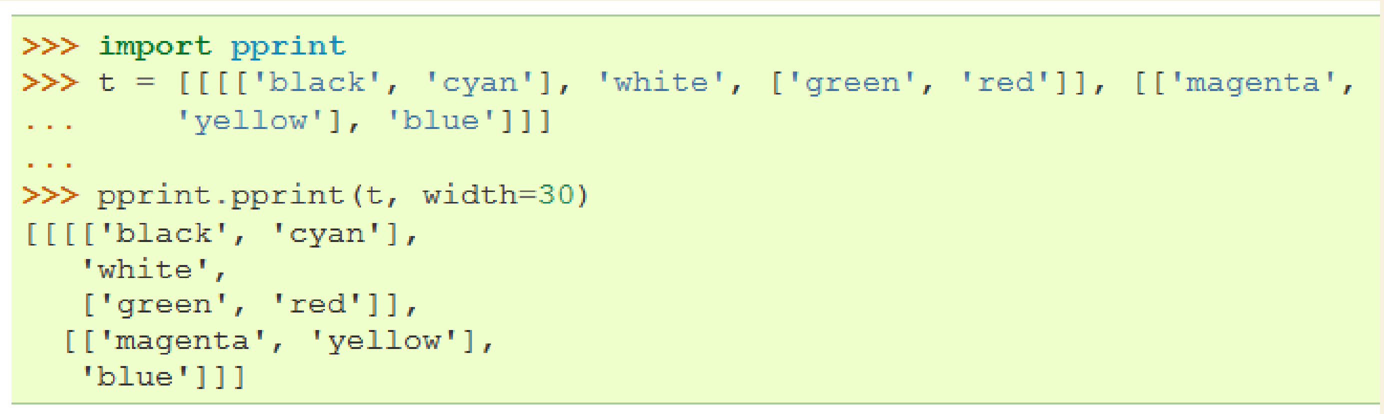
Output Formatting

- The **repr** module provides a version of repr() customized for abbreviated displays of large or deeply nested contai



```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

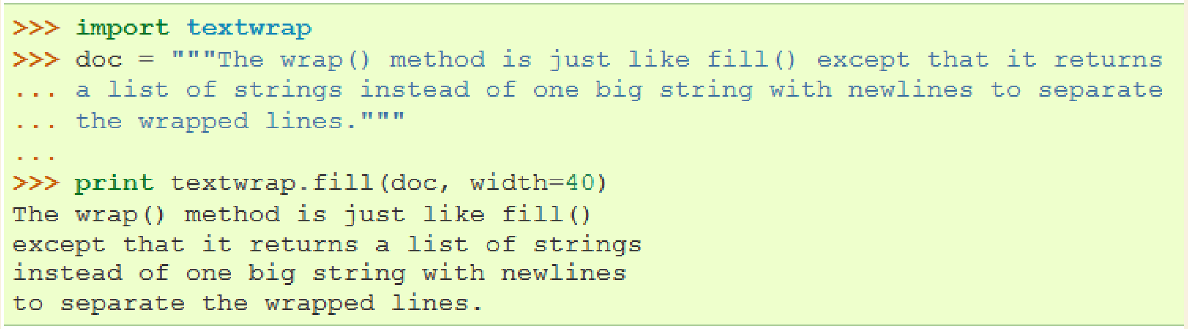
- The **pprint** module offers more sophisticated control over printing objects in a way readable by the interpreter
  - “pretty printer” adds appropriate line breaks and indentation when the result is more than one line



```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

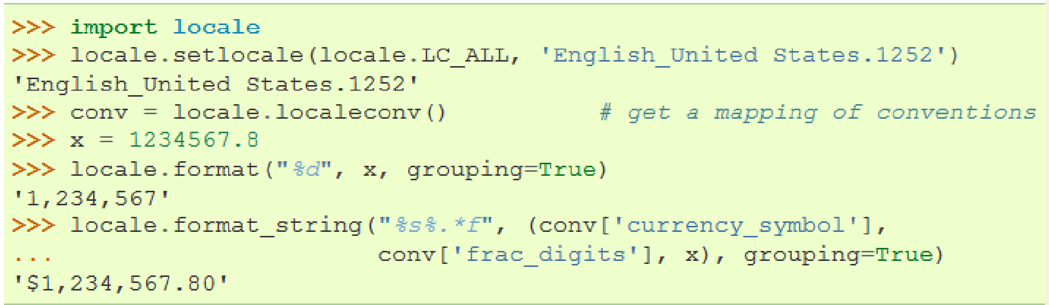
Output formatting

- Can use the **textwrap** module to format paragraphs of text to fit a given screen width



```
>>> 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.
```

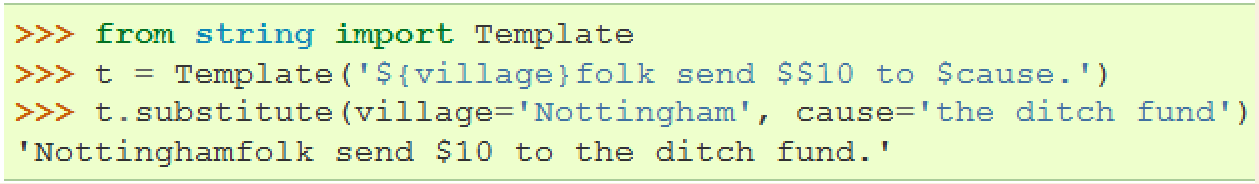
- The **locale** module access a database of cultural specific data formats
  - Allows programmers to deal with certain cultural issues in an application w/out knowing all the specifics of where the SW is executed



```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

Templating

- The `string` module includes a Template class with a simplified syntax suitable for editing by end users
  - Format uses placeholder names formed by $
    - substitute() method performs template substitution, returning a new string
    - “$$” creates a single escaped $
    - Surround placeholder with “{ }” to follow it with more alphanumeric characters



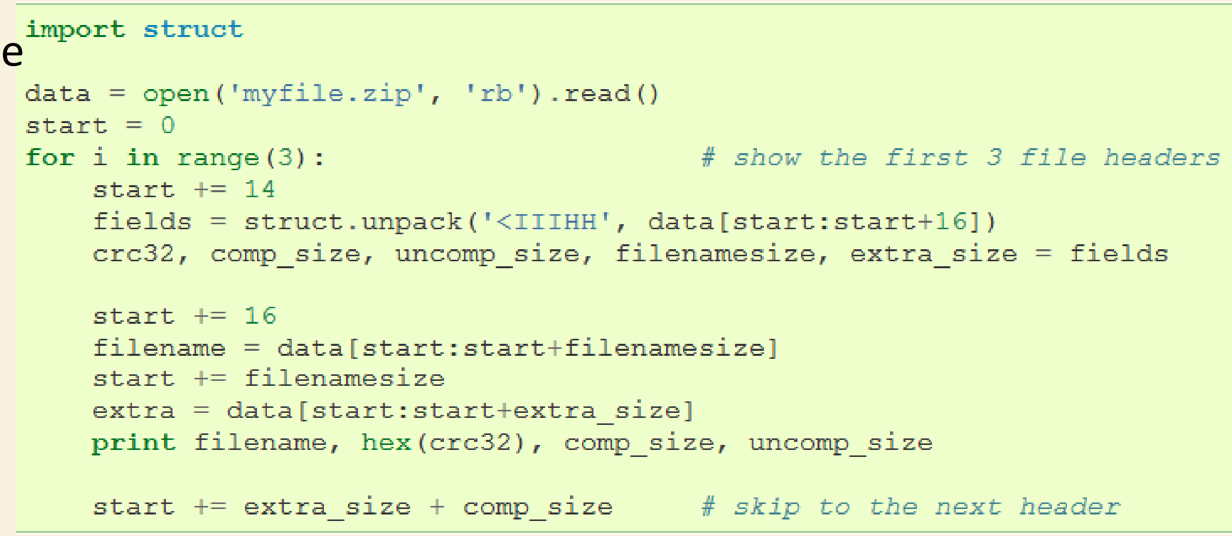
```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

Working with Binary Data Record Layouts

- The `struct` module provides pack() and unpack() functions for working with variable length binary record formats
  - Example below shows how to loop through header info in a ZIP file w/out using the zipfile module

- Pack codes “H” and “I” represent two and four byte unsigned integers

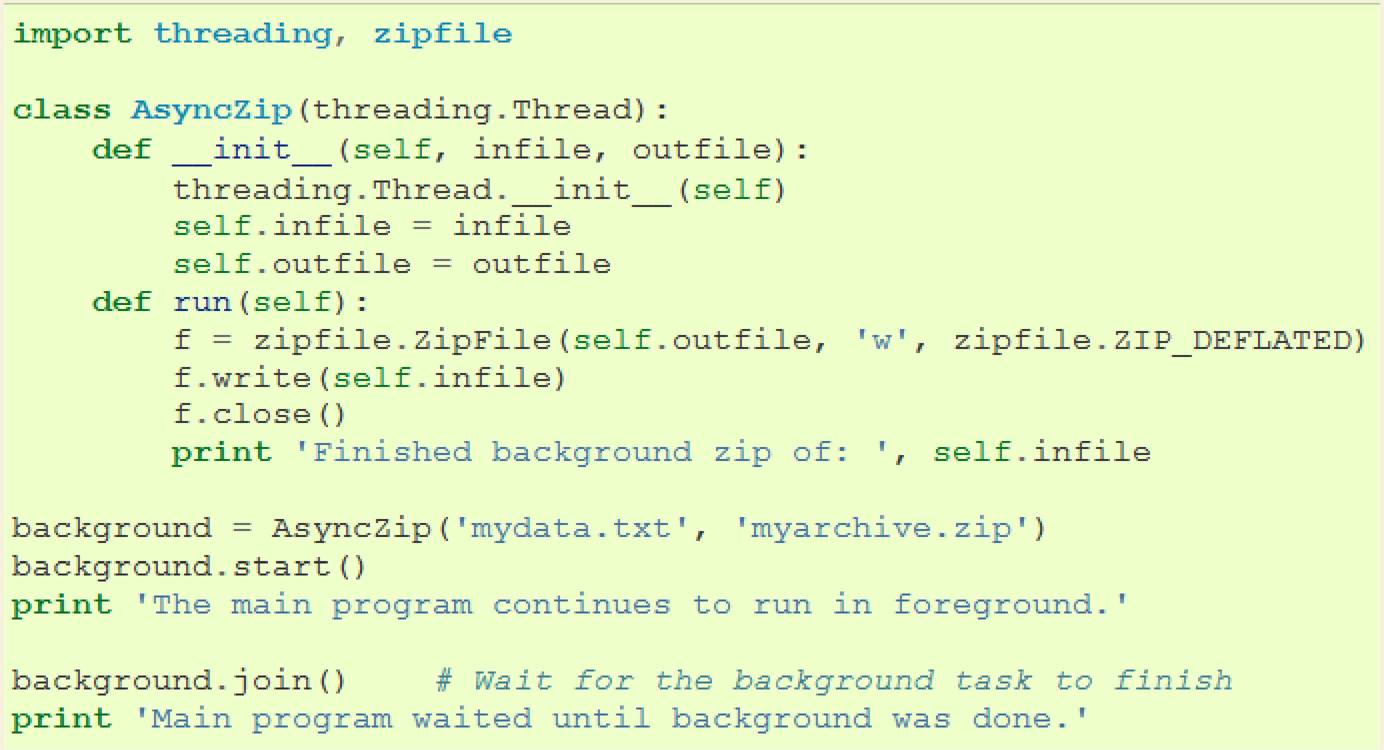
- “<“ indicates pack codes are standard size and in little-endian byte order



```python
import struct

data = open('myfile.zip', 'rb').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
```

Multi-threading

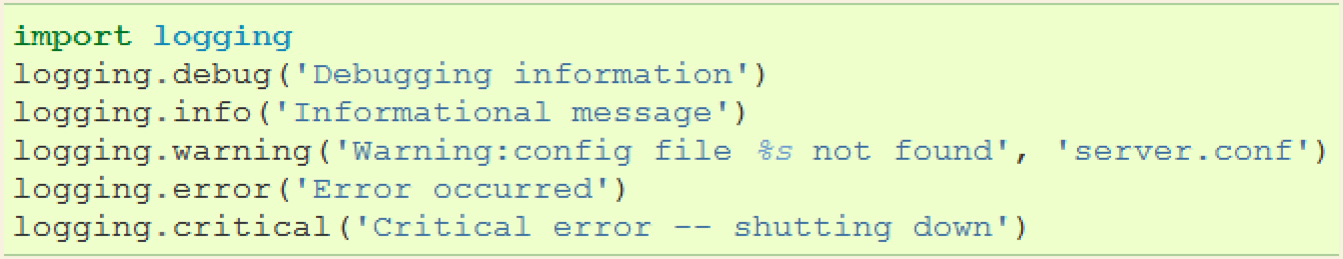
- Use the `threading` module to run tasks in the background while the main program runs
  - After creating a thread object, call the start() method to invoke the run() method in a separate thread of control
  - join() method waits until a thread terminates



```
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('mydata.txt', 'myarchive.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.'
```

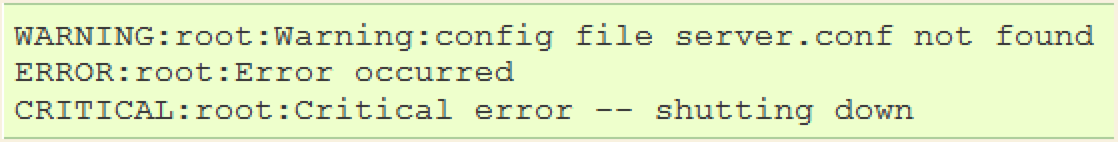
Logging

- The `logging` module offers a full featured and flexible logging system for applications
  - Log messages usually sent to a file or sys.stderr
- Example:



```
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')
```

  - Produces the following output:



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

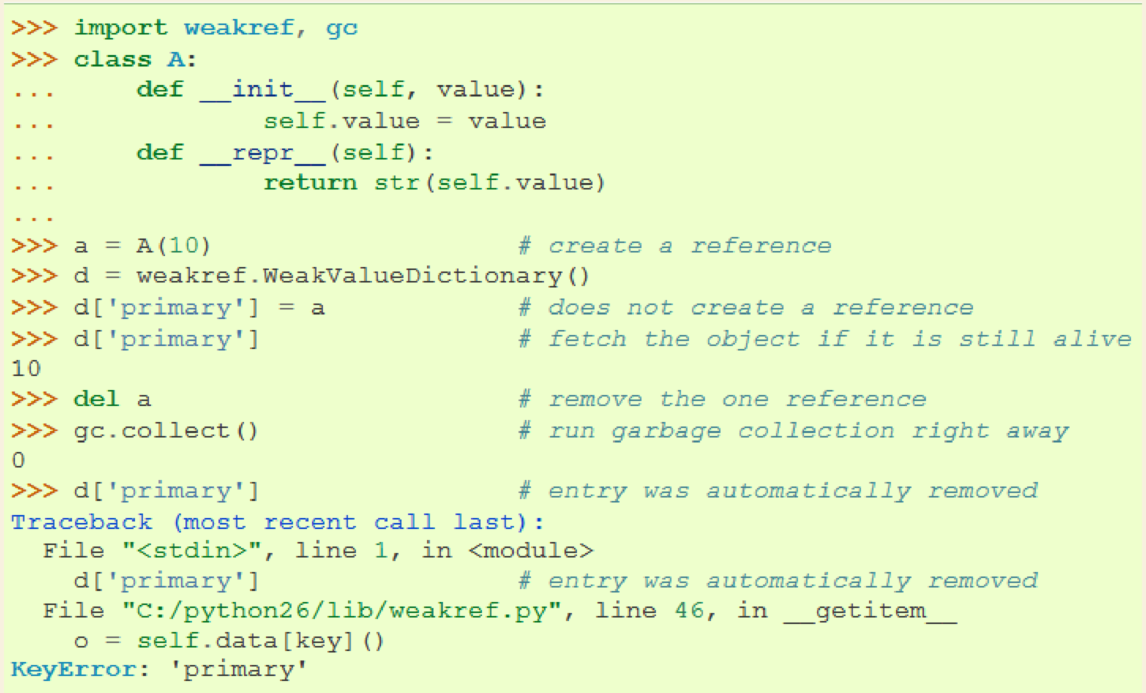
- Other output options include routing messages through email, datagrams, sockets, or an HTTP server

Weak References

- The `weakref` module provides tools to track objects without creating references
  - When an object is no longer needed, it is automatically removed from the weakref table and a callback is triggered for weakref objects

- WeakKeyDictionary is a mapping class that references keys weakly

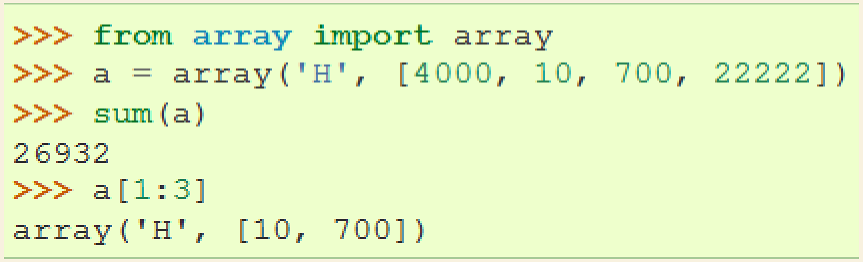
- Entries in dictionary are destroyed when there is no longer a strong reference to the key



```
>>> 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
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

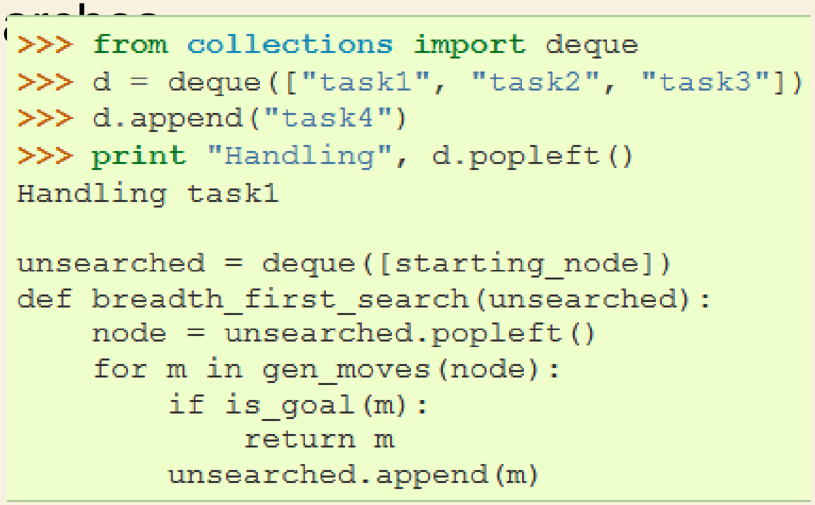
Tools for Working with Lists

- The **array** module provides an array() object that stores data of a single type more compactly
  - Typecode “H” represents two byte unsigned binary numbers



```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

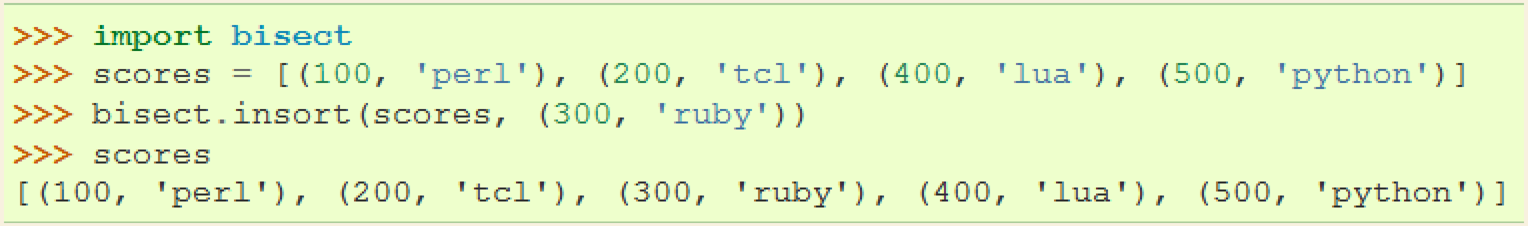
- The **collections** module provides a deque() object with fast appends and O(1) pops from either side; good for implementing queues and breadth first tree searches



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

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)
```

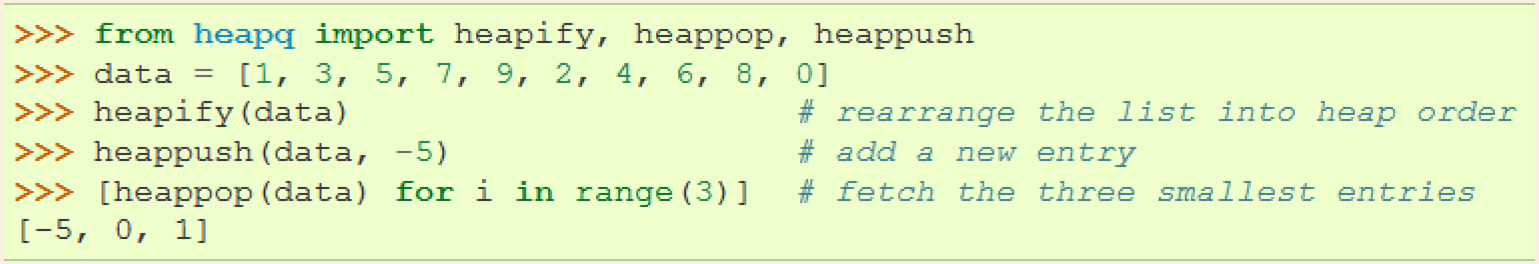
Tools for Working with Lists

- The `bisect` module contains functions for manipulating sorted lists
  - Implements bisection algorithm



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

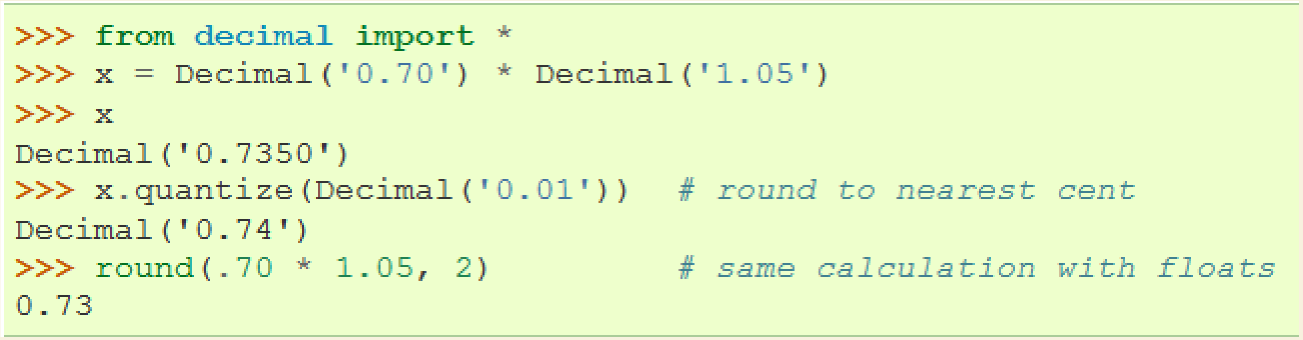
- The heapq module provides functions for implementing heaps based on regular lists
  - Implements the heap queue algorithm
  - Smallest valued entry always kept at position 0



```
>>> 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
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

Decimal Floating Point Arithmetic

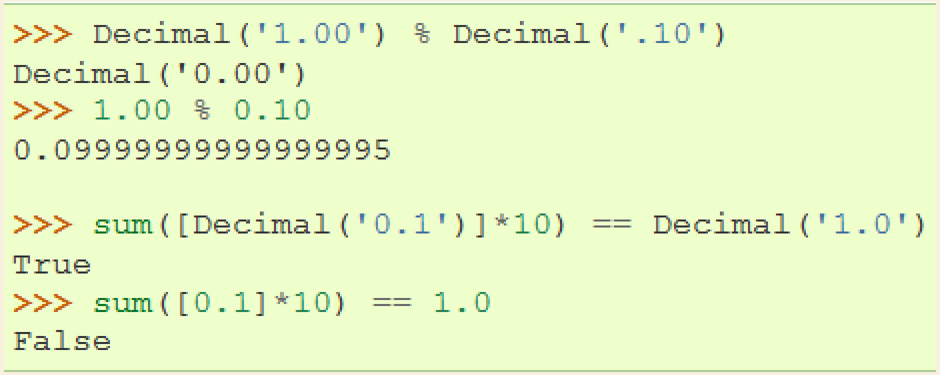
- The `decimal` module can be used for decimal floating point arithmetic
- Useful for:
  - Financial applications that require exact decimal representation
  - Control over precision
  - Control over rounding to meet legal or regulatory requirements
  - Tracking of significant figures
  - Applications where the results should match hand calculations
- Difference of results between decimal floating point and binary floating point:



```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

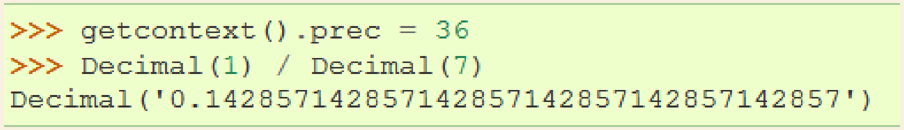
Decimal Floating Point Arithmetic

- Exact representation enables the Decimal class to perform modulo calculations and equality tests where binary floating point would fail:



```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

- `decimal` module also provides arithmetic with as much precision as needed
  - Using getcontext().prec method to set # of decimal places:



```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

References

- Python v2.7.1 documentation
  - “11. Brief Tour of the Standard Library Part II”
    - http://docs.python.org/tutorial/stdlib2.html
  - “The Python Standard Library”
    - http://docs.python.org/library/