Intermediate Python Course Notes

Release 0.8

Raymond Hettinger

May 11, 2012

CONTENTS

Location: Cisco – San Jose

Date: April 16, 2012

Taught by: Raymond Hettinger python@rcn.com @raymondh

This file: http://dl.dropbox.com/u/3967849/lw2/links.txt

Download tool: http://dl.dropbox.com/u/3967849/lw2/download_class_files.py http://tinyurl.com/python-lw2

CHAPTER

ONE

BACKGROUND KNOWLEDGE (PREREQUISITES)

- Proficient at creating and running Python modules
- Generators and the Iterator Protocol: - map, filter, reduce, sorted, enumerate - zip, sum, range, xrange, min, max
- Intro to Unittest and Doctest
- Conversions between str/list/tuple/int/float/dict
- Most of the builtin functions
- Understand how to make classes and use special methods
- Deep understanding of dictionaries and dict methods
- List comprehensions and generator expressions
- Regular Expressions

CHAPTER

TWO

WHAT TO REVIEW

The most important files to review after class are:

- **decorator_school.py** Shows how decorators work from the simplest case of decorator for registering functions (just like itty does) to a triply-nested decorator for checking pre-conditions. Has working code for an unbounded cache.
- **docfinder.py** Demonstrates how a small but complete application is built. Makes effective use of SQLite3, collections.Counter, list comprehensions, and subprocess.check_output. Uses a module level docstring to document the API and the data architecture.
- **dictionaries.py** Creates a dictionary-like object based on a list of tuples. Employees the MutableMapping abstract base class to build-up a full dictionary API with minimum effort. Demonstrates an effective optimization using a self-organizing table.

CHAPTER

THREE

PRESENTATIONS AND SLIDES

The slides presented in class can be found at:

- http://dl.dropbox.com/u/3967849/lw2/PythonTips.pdf
- http://dl.dropbox.com/u/3967849/lw2/PythonAwesome.pdf
- http://dl.dropbox.com/u/3967849/lw2/descriptors.pdf

Please keep in mind that these are proprietary. Please don't use my slides to give presentations. They are provided exclusively to class participants to review the course material.

CHAPTER

FOUR

RECOMMENDED READING

The following links are recommended as a way to extend the knowledge covered in class:

- http://en.wikipedia.org/wiki/Dependency_injection
- http://rhettinger.wordpress.com/2011/05/26/super-considered-super/
- http://www.clemesha.org/blog/modern-python-hacker-tools-virtualenv-fabric-pip/
- http://en.wikipedia.org/wiki/Zipf's_law
- http://docs.python.org/library/collections.html#collections-abstract-base-classes
- http://en.wikipedia.org/wiki/Self-organizing_list
- http://xkcd.com/327/
- http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- http://dl.dropbox.com/u/3967849/lw2/downey08semaphores.pdf

CHAPTER

FIVE

RESOURCES

These are some resources mentioned in class:

- http://www.w3schools.com/sql/ Notes on SQL
- http://www.pypy.org/ Optimized version of Python
- http://pypi.python.org/pypi/requests Make HTTP requests with full control over headers
- http://pyvideo.org/category/17/pycon-us-2012 Videos from PyCon 2012
- http://txt2re.com/ Generate regexes from examples
- http://sphinx.pocoo.org/ Python tool for PDF generation

CHAPTER

SIX

DAY ONE TOPICS

6.1 Important language features

- sorting with key-functions sorted(s, key=str.lower)
- slicing and invariants s[:i] + s[i:] == s[:]
- lambda
  - f = lambda x: 3*x+1
  - f = lambda : 3**10
  - s = [(lambda a: x + a) for x in range(5)]
  - s = [(lambda a, x=x: x + a) for x in range(5)]
- operator module
  - itemgetter(1,0) same as lambda r: (r[1], r[0])
  - attrgettr('name') same as lambda r: r.name
- unbound method um = int.__add__; um(10, 20)
- bound method bm = (10).__add__; bm(20)
- partial function evaluation f = partial(pow, 2) same as lambda x: pow(2,x)
- two argument form of iter()
  - for block in iter(partial(f.read, 20), ''): ...
  - # purpose is to transform functions into iterators
- else clauses on for and while
- super() and dependency injection class OrderedCounter(Counter, OrderedDict)

6.2 Motivating use cases for Python's Indentation

In C, the indentation can misrepresent the actual program structure.

indentation.txt:

```
for (i=0 ; i<n ; i++);
    printf("hello\n");

if (x<10)
    if (y == 0)
       f(x,y);
    else if (x<20)
        g(x,y);
    else:
        h(x,y);
```

6.3 Class design

The following code demonstrates the use of class variables, *property*, *staticmethod*, *classmethod*, and slots:

class_demo.py

```
''' Main product:  A toolkit for analyzing round things.
Circuitous, Inc.
'''

import math

class Circle(object):
    'A tool for circle analytics'
    version = 0.8
    __slots__ = ['diameter']

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        'Perform quadrature of the circle'
        p = self.__perimeter()     # _Circle__perimeter
        r = p / 2.0 / math.pi
        return math.pi * r ** 2.0

    def perimeter(self):
        'Return the circumference'
        return math.pi * self.radius * 2.0

    __perimeter = perimeter        # _Circle__perimeter

    @classmethod
    def from_bbd(cls, bbd):
        'Create a circle from the bounding box diagonal'
        return cls(bbd / math.sqrt(2.0) / 2.0)

    @staticmethod
```

```
    def angle_to_slope(angle_in_degrees):
        return math.tan(math.radians(angle_in_degrees))

    def set_radius(self, radius):
        self.diameter = radius * 2.0

    def get_radius(self):
        return self.diameter / 2.0

    radius = property(get_radius, set_radius)
```

6.4 Demonstration of Property

The following code is a simple example of using property for a computed attribute.

property_demo.py

```
class Range(object):
    #__slots__ = 'low', 'high'
    def __init__(self, low, high):
        self.low = low
        self.high = high

    @property
    def mid(self):
        return (self.low + self.high) / 2.0
```

6.5 Truthiness

This code shows the logic used to evaluate which objects are true in Python.

```
def mybool(obj):
        'Compute the truthiness of an object'
        try:
                size = obj.__len__()
                if size != 0:
                        return True
                else:
                        return False
        except AttributeError:
                pass
        try:
                nz = obj.__nonzero__()
                return nz
        except AttributeError:
                pass
        if obj is None:
                return False
        return True
```

6.6 Else-clauses

Searching for values in a sequence needs separate paths for the found and not-found cases. The else-clause differentiates the two outcomes.

```
def find(target, sequence):
    found = False
    for i, x in enumerate(sequence):
        if x == target:
            found = True
            # case where x is found
            break
    if found:
        return i
    return -1
```

CHAPTER

SEVEN

DAY TWO TOPICS

How to generate HTML documentation automatically:

```
$ python -m pydoc -w class_demo
```

Syntax for a list comprehension:

```
[<expr> for <var> in <iterable> if <cond>]
```

EAFP – Easier to ask forgiveness than permission:

```
def worker():
    while True:
        try:
            item = d.popitem()
        except KeyError:
            break
        do_some_work(item)
```

LBYL – Look before you leap:

```
while d:
    item = d.popitem()
    do_some_work(item)
```

7.1 Threading

All shared resources need to be run in their own thread and communicate solely with *Queue* objects.

threading_demo.py

```
''' Rule for eliminate race conditions:

    EVERY shared resource (be it print, logging, files, globals, dicts, etc),
    should have its own thread with EXCLUSIVE access to that resource.
    ALL communication should be done through a Queue object.
'''

import threading
```

```
import Queue

############ Counter resource ########

counter = 0

count_queue = Queue.Queue()

def count_manager():
    'I have exclusive write access to the global counter variable'
    global counter

    while True:
        increment = count_queue.get()
        counter += increment
        print_queue.put([
            'The current value of the counter is:',
            str(counter)
            ])

t = threading.Thread(target=count_manager)
t.daemon = True
t.start()

############### Print resource ######

print_queue = Queue.Queue()

def print_manager(print_number):
    'I have exclusive access to the printer'
    while True:
        job = print_queue.get()
        for line in job:
            print line
        print '----------------------'

t = threading.Thread(target=print_manager)
t.daemon = True
t.start()

############ Main program ###########

def worker():
    count_queue.put(1)

for i in range(10):
    t = threading.Thread(target=worker)
    t.start()
```

7.2 SQLite3 command-line tool

The SQLite3 command line tool (get this from sqlite.org):

```
$ sqlite3 pepsearch.db
sqlite> .help
sqlite> .schema documents
sqlite> SELECT * FROM documents;
```

7.3 Abstract Base Classes

The following two dictionaries are build using the *MutableMapping* abstract base class:

dictionaries.py

```
'Create dictionary like classes from scratch'

import collections

class TupleDictionary(collections.MutableMapping):
    'Dictlike object built on a list of tuples for space efficiency'

    def __init__(self):
        self.lot = []

    def __setitem__(self, key, value):    # d['raymond']='red'
        if key in self:
            del self[key]
        item = key, value
        self.lot.append(item)

    def __getitem__(self, key):           # d['raymond'] --> 'red'
        for i, (k, v) in enumerate(self.lot):
            if k == key:
                if i:
                    self.lot[i], self.lot[i-1] = self.lot[i-1], self.lot[i]
                return v
        raise KeyError(key)

    def __delitem__(self, key):
        for i, k in enumerate(self):
            if k == key:
                del self.lot[i]
                return
        raise KeyError(key)

    def __len__(self):
        return len(self.lot)

    def __iter__(self):
        for k, v in self.lot:
```

```
            yield k

import sqlite3

class PersistentDict(collections.MutableMapping):
    '''Implement a disk based dictionary that remembers
       values between sessions.  It also supports concurrent
       access from multiple processes, can be queried during
       live updates, and can afford access to other programming
       languages.
    '''

    def __init__(self, dbname):
        self.dbname = dbname
        self.c = sqlite3.connect(dbname)
        self.c.text_factory = str
        try:
            self.c.execute(
                'CREATE TABLE mydict (key text PRIMARY KEY, value text)')
        except sqlite3.OperationalError:
            pass
        self.c.commit()

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        self.c.execute('INSERT INTO mydict VALUES (?, ?)', (key, value))
        self.c.commit()

    def __len__(self):
        rows = list(self.c.execute('SELECT COUNT(*) FROM mydict'))
        return rows[0][0]

    def __getitem__(self, key):
        rows = list(self.c.execute('SELECT value FROM mydict WHERE key = ?',
                                   (key,)))
        if not rows:
            raise KeyError(key)
        return rows[0][0]

    def __delitem__(self, key):
        if key not in self:
            raise KeyError(key)
        self.c.execute('DELETE FROM mydict WHERE key = ?', (key,))
        self.c.commit()

    def __iter__(self):
        cursor = self.c.cursor()
        for key, in cursor.execute('SELECT key FROM mydict'):
            yield key

    def close(self):
        self.c.close()
```

```
if __name__ == '__main__':
    d = PersistentDict(':memory:')
    d['raymond'] = 'red'
    d['rachel'] = 'blue'
    d['matthew'] = 'green'
    print d['rachel']
    print list(d)
    print len(d)
    del d['rachel']
    print list(d)
    print d.items()
    print d.get('roger', 'black')
```

Here is an alternate implementation of the dictionary based on SQLite3:

sqlite_dict.py

```
import sqlite3
import collections
import pickle
import json

class PersistentDict(collections.MutableMapping):
    '''Implement a disk based dictionary that remembers
       values between sessions.  It also supports concurrent
       access from multiple processes, can be queried during
       live updates, and can afford access to other programming
       languages.
    '''

    def __init__(self, dbname):
        self.c = sqlite3.connect(dbname)
        try:
            self.c.execute('CREATE TABLE mydict (key text PRIMARY KEY, value blob)')
        except sqlite3.OperationalError:
            pass
        self.c.commit()

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        value = sqlite3.Binary(pickle.dumps(value))
        item = key, value
        self.c.execute("INSERT INTO mydict VALUES (?, ?)", item)
        self.c.commit()

    def __getitem__(self, key):
        rows = list(self.c.execute("SELECT value FROM mydict WHERE key = ?",
                                   (key,)))
        if rows:
            value = rows[0][0]
            return pickle.loads(value)
```

```
            raise KeyError(key)

        def __delitem__(self, key):
            if key not in self:
                raise KeyError(key)
            self.c.execute("DELETE FROM mydict WHERE key = ?", (key,))
            self.c.commit()

        def __len__(self):
            rows = list(self.c.execute("SELECT COUNT(*) FROM mydict"))
            return rows[0][0]

        def __iter__(self):
            cursor = self.c.cursor()
            for row in cursor.execute("SELECT key FROM mydict"):
                key = row[0]
                yield key

    d = PersistentDict('namecolor.db')
    #d['raymond'] = 'red'
```

CHAPTER

EIGHT

DAY THREE TOPICS

8.1 Iterator school

The following code show the iterator protocol and how it is expressed in Python using the *for* and *yield* keywords. It also shows how two-way generators work using the *send* method.

iterator_school.py

```
''' Teach the universal iterator protocol and how Python made it easy to use.
    Learn the most common premade iterators in the language.

ITERABLE:
    * Anything that can looped over
    * Anything on the right side of a for-loop
    * Anything that has an __iter__ method (or equivalent)
        responsible for returning an ITERATOR

ITERATOR:
    * Object with mutable state responsible for giving values one at a time.
    * It must have a next() method responsible for:
        1. returns the next value
        2. updates the state
        3. tell us when it's done -- raise StopIteration
    * It must have an __iter__ method that returns self
        -- an iterator is self-iterable

'''

def myiter(obj):
    'Emulate the builtin iter() function'
    return obj.__iter__()

def mynext(obj):
    'Emulate the builtin next() function'
    return obj.next()

def mylist(iterable):
    ''' Emulate the built-in list()
        Return a new list composed of elements from an iterable
```

```
        >>> mylist('cat')
        ['c', 'a', 't']

    '''
    result = []
    it = iter(iterable)
    while True:
        try:
            element = next(it)
        except StopIteration:
            break
        result.append(element)
    return result

def mylist_for(iterable):
    'Same as list() but implemented with a for-loop'
    result = []
    for element in iterable:
        result.append(element)
    return result

def my_sum(iterable, start=0):
    ''' Emulate the built-in sum() function.
        Add-up the values taken from an iterable.

        >>> my_sum([25, 7, 2])
        34
        >>> my_sum([25, 7, 2], 100)
        134
    '''
    total = start
    for x in iterable:
        total += x
    return total

def mymin(iterable):
    ''' Emulate the built-in min() function.
        Return the lowest value in an iterable.

        >>> mymin([10, 30, 20])
        10
        >>> mymin([30, 20, 10])
        10

    '''
    it = iter(iterable)
    lowest = next(it)
    for x in it:
        if x < lowest:
            lowest = x
    return lowest

def myrange(a, b=None, step=1):
```

```
        ''' Emulate the builtin range() function.
            Produce a list of consecutive values.

            >>> myrange(10)
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
            >>> myrange(2, 10)
            [2, 3, 4, 5, 6, 7, 8, 9]
            >>> myrange(2, 10, 3)
            [2, 5, 8]

        '''
        if b is None:
            start = 0
            stop = a
        else:
            start = a
            stop = b
        result = []
        i = start
        while i < stop:
            result.append(i)
            i += step
        return result

    class MyXrange:
        ''' Emulate the built-in xrange() function.
            This iterable returns an iterator that gives
            consecutive values.

            >>> list(MyXrange(10))
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
            >>> set(MyXrange(10))
            set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
            >>> sorted(MyXrange(10))
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
            >>> sum(MyXrange(10))
            45
            >>> min(MyXrange(10))
            0
            >>> max(MyXrange(10))
            9

        '''
        def __init__(self, stop):
            self.stop = stop
        def __iter__(self):
            return MyXrangeIterator(self.stop)

    class MyXrangeIterator:
        def __init__(self, stop):
            self.stop = stop
            self.i = 0        # <-- the mutable state is here
        def next(self):
```

```
            value = self.i
            if value >= self.stop:
                raise StopIteration
            self.i += 1
            return value
        def __iter__(self):
            return self


    def myxrange(stop):
        ''' Alternate version of xrange() using yield.

            >>> list(myxrange(10))
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        '''
        i = 0
        while i < stop:
            yield i
            i += 1

    def timestamp():
        'Timestamp generator'
        while True:
            yield time.ctime()

    def count(start=0, step=1):
        'Infinite iterator of counts'
        i = start
        while True:
            yield i
            i += step

    def repeat(obj):
        'Return the same value over and over'
        while True:
            yield obj

    def myizip(iterable1, iterable2):
        ''' Emulate the itertools.izip() function.
            Iterate over successive pairs taken from two iterables.

            >>> list(myizip('ABC', 'wxyz'))
            [('A', 'w'), ('B', 'x'), ('C', 'y')]

        '''
        it1 = iter(iterable1)
        it2 = iter(iterable2)
        while True:
            a = next(it1)
            b = next(it2)
            t = a, b
            yield t
```

```
if __name__ == '__main__':
    import doctest
    print doctest.testmod()
```

8.2 Docfinder Python Application

Here is the code for docfinder application.

docfinder.py

```
''' Keyword searchable document database

API:

    create_db()
    add_document(uri, text)  'pep-0308'
    get_document(uri) --> text
    document_search(keyword0, keyword1, ...) --> [uri0, ur1, ...]

Database schema:

    characters                documents
    -------------             --------------
    index by word             index by uri
    -------------             --------------
    uri      text             uri       text
    word     text             document  blob
    relfreq  real

'''

from __future__ import division
import os, re, collections, sqlite3, glob, subprocess, pprint, bz2
from decorator_school import bounded_cache

__all__ = ['create_db', 'add_document', 'get_document', 'document_search']

database = 'pepsearch.db'

class NotFound(Exception):
    pass

def normalize(words):
    '''Improve comparability by stripping plurals and lowercasing

        >>> normalize(['Hettinger', 'Enumerates'])
        ['hettinger', 'enumerate']

    '''
    return [word.lower().rstrip('s') for word in words]
```

```
def characterize(uri, text, n=200):
    'Scan text and return relative frequencies of the n most common words'
    # return list of tuples in the form: (uri, word, relative_frequency)
    words = re.findall(r'\b[A-Za-z]{3,}\b', text)
    words = normalize(words)
    count = collections.Counter(words).most_common(n)
    total = sum([cnt for word, cnt in count])
    return [(uri, word, cnt/total) for word, cnt in count]

def add_document(uri, text):
    'Add a document with a given identifier to the search database'
    c = sqlite3.connect(database)
    btext = sqlite3.Binary(bz2.compress(text))
    c.execute('INSERT INTO documents VALUES (?, ?)', (uri, btext))
    lot = characterize(uri, text)
    c.executemany('INSERT INTO characters VALUES (?,?,?)', lot)
    c.commit()

def create_db():
    'Set-up a new characterized document database, eliminating an old one if it exists
    try:
        os.remove(database)
    except OSError:
        pass
    c = sqlite3.connect(database)
    c.text_factory = str
    c.execute('CREATE TABLE documents (uri text PRIMARY KEY, document blob)')
    c.execute('CREATE TABLE characters (uri text, word text, relfreq real)')
    c.execute('CREATE INDEX Wordindex ON characters (word)')
    c.commit()

@bounded_cache(20)
def get_document(uri):
    'Retrieve a full document by name'
    c = sqlite3.connect(database)
    rows = list(c.execute('SELECT document FROM documents WHERE uri = ?',
                          (uri,)))
    if not rows:
        raise NotFound(uri)
    return bz2.decompress(rows[0][0])

query_template = '''
    SELECT uri, SUM(relfreq) AS relevance
    FROM characters
    WHERE word IN (%s)
    GROUP BY uri
    ORDER BY relevance DESC;
'''

def document_search(*keywords):
    'Find ranked list of best matched URIs for a given keyword'
    c = sqlite3.connect(database)
    c.text_factory = str
```

```
    keywords = tuple(normalize(keywords))
    questions = ','.join('?' * len(keywords))
    query = query_template % questions
    rows = c.execute(query, keywords)
    return [uri for uri, relevance in rows]


##############################################################
###  Test harness code follows  ##############################

if __name__ == '__main__':
    import os

    docdir = 'peps'

    if 0:
        print normalize(['Hettinger', 'enumerates'])

    if 0:
        filename = 'pep-0238.txt'
        fullname = os.path.join(docdir, filename)
        with open(fullname) as f:
            text = f.read()
        uri = os.path.splitext(filename)[0]
        c = characterize(uri, text)
        pprint.pprint(c)

    if 0:
        create_db()

    if 0:
        #for filename in ['pep-0237.txt', 'pep-0236.txt', 'pep-0235.txt']:
        for filename in  os.listdir(docdir):
            fullname = os.path.join(docdir, filename)
            with open(fullname) as f:
                text = f.read()
            uri = os.path.splitext(filename)[0]
            print uri, len(text)
            add_document(uri, text)

    if 0:
        print get_document('pep-0237')[:100]

    if 0:
        print document_search('Hettinger', 'enumerates')[:100]
```

The sample data can be found at: http://dl.dropbox.com/u/3967849/lw2/peps.zip

CHAPTER

NINE

DAY FOUR TOPICS

9.1 Decorator school

The following code shows how decorators are made, from a simple registration example to a tool for caching:

decorator_school.py

```
from functools import wraps

functions = []

def register(func):
    'Register a function on the functions list'
    name = func.__name__
    if name not in functions:
        functions.append(name)
    return func

def add_docstring(func):
    'Fill-in missing docstrings with a default docstring'
    doc = func.__doc__
    if doc is None:
        func.__doc__ = '<missing docstring>'
    return func

def logcall(func):
    @wraps(func)      #newfunc.__name__ = func.__name__; newfunc.__doc__ = func.__doc__
    def newfunc(x):
        print 'I was called with', x
        y = func(x)
        print 'The result is', y
        return y
    return newfunc

def cache(func):
    'Improve speed by caching results'
    result_dict = {}
    @wraps(func)
    def newfunc(x):
        if x in result_dict:
```

```
            return result_dict[x]
        y = func(x)
        result_dict[x] = y
        return y
    return newfunc

def bounded_cache(maxsize):
    'Limited size cache for long running processes'
    def cache(func):
        'Improve speed by caching results'
        result_dict = {}
        @wraps(func)
        def newfunc(x):
            if x in result_dict:
                return result_dict[x]
            if len(result_dict) >= maxsize:
                result_dict.clear()
            y = func(x)
            result_dict[x] = y
            return y
        return newfunc
    return cache

########## Sample functions ##########
# collatz square big_calc

@logcall          # square = logcall(square)
@cache
def square(x):
    'Return value times itself'
    return x * x

@logcall          # collatz = logcall(collatz)
@register         # collatz = register(collatz)
@cache
def collatz(x):
    'One step in verifying the Collatz conjecture'
    if x % 2 == 0:
        return x // 2
    return 3 * x + 1


@add_docstring     # big_calc = add_docstring(big_calc)
@register          # big_calc = register(big_calc)
@bounded_cache(4)            # big_calc = cache(big_calc)
def big_calc(x):
    print 'Doing a big calculation'
    return x + 1


##cache = {}
##
##def faster_big_calc(x):
```

```
##     if x in cache:
##         return cache[x]
##     print 'Doing a big calculation'
##     result = x + 1
##     cache[x] = result
##     return result
```

The caching decorate is used to dramatically speed-up a *fibonacci* example:

fibo.py

```
'Compute the second most important function ever'

from decorator_school import cache

def fibofast(n):
    a, b = 0, 1
    for i in xrange(n):
        a, b = b, a+b
    return a

@cache
def fibonacci(n):
    '''Return the n-th value in the recurrence:

           F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)
    '''
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print fibonacci(200)
```

It is easy to do tracing and line counts from the command-line:

```
# python -m trace --count fibo.py    # creates fibo.cover
```

The *itty* webframe can be found at http://dl.dropbox.com/u/3967849/lw2/itty.py . It uses decorators to good effect. For example, here is a simple set of webservices implemented using itty:

itty_webservice.py

```
from itty import get, post, run_itty
import os, subprocess, json, functools, shlex

def jsonify(origfunc):
    @functools.wraps(origfunc)
    def wrapper(*args, **kwds):
        result = origfunc(*args, **kwds)
        return json.dumps(result, indent=4)
    return wrapper

@get('/env')
@jsonify
```

```
def lookup_environ(request):
    return dict(os.environ)

@get('/env/(?P<name>\w+)')
def lookup_environ_variable(request, name):
    return os.environ[name]

@get('/freespace')
def compute_free_disk_space(request):
    return subprocess.check_output('df')

@get('/processes')
def show_currently_running_processes(request):
    return subprocess.check_output(shlex.split('ps aux'))

@post('/restart')
def test_post(request):
    os.system('restart')

run_itty(host='localhost', port=8080)
```

And here is a complete LRU cache decorator backported from Python 3.3:

lru_cache.py

```
## {{{ http://code.activestate.com/recipes/578078/ (r5)
from collections import namedtuple
from functools import update_wrapper
from threading import Lock

_CacheInfo = namedtuple("CacheInfo", ["hits", "misses", "maxsize", "currsize"])

def lru_cache(maxsize=100, typed=False):
    """Least-recently-used cache decorator.

    If *maxsize* is set to None, the LRU features are disabled and the cache
    can grow without bound.

    If *typed* is True, arguments of different types will be cached separately.
    For example, f(3.0) and f(3) will be treated as distinct calls with
    distinct results.

    Arguments to the cached function must be hashable.

    View the cache statistics named tuple (hits, misses, maxsize, currsize) with
    f.cache_info().  Clear the cache and statistics with f.cache_clear().
    Access the underlying function with f.__wrapped__.

    See:  http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

    """

    # Users should only access the lru_cache through its public API:
    #       cache_info, cache_clear, and f.__wrapped__
```

```
        # The internals of the lru_cache are encapsulated for thread safety and
        # to allow the implementation to change (including a possible C version).

        def decorating_function(user_function):

            cache = dict()
            stats = [0, 0]                  # make statistics updateable non-locally
            HITS, MISSES = 0, 1             # names for the stats fields
            kwd_mark = (object(),)          # separate positional and keyword args
            cache_get = cache.get           # bound method to lookup key or return None
            _len = len                      # localize the global len() function
            lock = Lock()                   # because linkedlist updates aren't threadsafe
            root = []                       # root of the circular doubly linked list
            nonlocal_root = [root]                  # make updateable non-locally
            root[:] = [root, root, None, None]      # initialize by pointing to self
            PREV, NEXT, KEY, RESULT = 0, 1, 2, 3    # names for the link fields

            def make_key(args, kwds, typed, tuple=tuple, sorted=sorted, type=type):
                # helper function to build a cache key from positional and keyword args
                key = args
                if kwds:
                    sorted_items = tuple(sorted(kwds.items()))
                    key += kwd_mark + sorted_items
                if typed:
                    key += tuple(type(v) for v in args)
                    if kwds:
                        key += tuple(type(v) for k, v in sorted_items)
                return key

            if maxsize == 0:

                def wrapper(*args, **kwds):
                    # no caching, just do a statistics update after a successful call
                    result = user_function(*args, **kwds)
                    stats[MISSES] += 1
                    return result

            elif maxsize is None:

                def wrapper(*args, **kwds):
                    # simple caching without ordering or size limit
                    key = make_key(args, kwds, typed) if kwds or typed else args
                    result = cache_get(key, root)   # root used here as a unique not-found
                    if result is not root:
                        stats[HITS] += 1
                        return result
                    result = user_function(*args, **kwds)
                    cache[key] = result
                    stats[MISSES] += 1
                    return result

            else:
```

```
        def wrapper(*args, **kwds):
            # size limited caching that tracks accesses by recency
            key = make_key(args, kwds, typed) if kwds or typed else args
            with lock:
                link = cache_get(key)
                if link is not None:
                    # record recent use of the key by moving it to the front of th
                    root, = nonlocal_root
                    link_prev, link_next, key, result = link
                    link_prev[NEXT] = link_next
                    link_next[PREV] = link_prev
                    last = root[PREV]
                    last[NEXT] = root[PREV] = link
                    link[PREV] = last
                    link[NEXT] = root
                    stats[HITS] += 1
                    return result
            result = user_function(*args, **kwds)
            with lock:
                root = nonlocal_root[0]
                if _len(cache) < maxsize:
                    # put result in a new link at the front of the list
                    last = root[PREV]
                    link = [last, root, key, result]
                    cache[key] = last[NEXT] = root[PREV] = link
                else:
                    # use root to store the new key and result
                    root[KEY] = key
                    root[RESULT] = result
                    cache[key] = root
                    # empty the oldest link and make it the new root
                    root = nonlocal_root[0] = root[NEXT]
                    del cache[root[KEY]]
                    root[KEY] = None
                    root[RESULT] = None
                stats[MISSES] += 1
            return result

    def cache_info():
        """Report cache statistics"""
        with lock:
            return _CacheInfo(stats[HITS], stats[MISSES], maxsize, len(cache))

    def cache_clear():
        """Clear the cache and cache statistics"""
        with lock:
            cache.clear()
            root = nonlocal_root[0]
            root[:] = [root, root, None, None]
            stats[:] = [0, 0]

    wrapper.__wrapped__ = user_function
    wrapper.cache_info = cache_info
```

```
            wrapper.cache_clear = cache_clear
            return update_wrapper(wrapper, user_function)

        return decorating_function
    ## end of http://code.activestate.com/recipes/578078/ }}}
```

9.2 HTTP Front-end

Today, we extend yesterday's work, by adding an HTTP front-end using *itty*:

http_finder.py

```
from itty import get, post, run_itty
import os, subprocess, json, functools, shlex
import docfinder, cgi

#### Define some presentation logic decorators  ######################

html_pre_template = '''
<html><body><pre>
%s
</pre></body></html>
'''

def pre(func):
    'Decorator that encases a result in HTML pre-tags'
    def newfunc(*args, **kwds):
        result = func(*args, **kwds)
        return html_pre_template % cgi.escape(result)
    return newfunc

def jsonify(origfunc):
    @functools.wraps(origfunc)
    def wrapper(*args, **kwds):
        result = origfunc(*args, **kwds)
        return json.dumps(result, indent=4)
    return wrapper

##### The web services ################################################

@get('/doc/(?P<uri>.+)')            # localhost:8080/doc/pep-0289
def retrieve_document(request, uri):
    return docfinder.get_document(uri)

@get('/viewdoc/(?P<uri>.+)')        # localhost:8080/viewdoc/pep-0289
@pre
def retrieve_document_in_browser(request, uri):
    try:
        return docfinder.get_document(uri)
    except docfinder.NotFound:
        return 'Sorry, I do not have a record for: %s' % uri
```

```
search_template = '''
<html>
<head>
<title> Custom document search utility for PEPS </title>
</head>
<body>
<h2> Search results for: <em> %s </em>
<hr>
<ol>
%s
</ol>
</body>
</html>
'''

link_template = '<li> <a href="../viewdoc/%s"> %s </a>'

@get('/search/(?P<terms>.+)')         # localhost:8080/search/Hettinger&enumerates
def search_documents_in_browser(request, terms):
    keywords = terms.split('&')
    uris = docfinder.document_search(*keywords)[:100]
    links = '\n'.join([link_template % (uri, uri) for uri in uris])
    return search_template % (','.join(keywords), links)


run_itty(host='localhost', port=8080)
```

9.3 Command Shell Front-end

We also use the *cmd* module to provide a user-friendly interactive shell:

search_shell.py

```
import cmd
import docfinder

class FinderShell(cmd.Cmd):
    intro = 'Welcome to the document finder utility'
    prompt = '(doc) '

    def do_wave(self, arg):
        'Wave to papa'
        print 'Waving with', arg

    def do_get(self, uri):
        'Retrieve a document based on its URI'
        print docfinder.get_document(uri)[:200]



FinderShell().cmdloop()
```

9.4 With Statement

The following code shows typical uses of the with-statement including file-closing and releasing locks. It also shows how to build a context manager from scratch.

with_demo.py

```
# Old-way
f = open('stocks.txt')
try:
    d = f.read()
    print d, len(d)
finally:
    f.close()

# New-way
with open('stocks.txt') as f:
    print 'I have %d stocks' % len(list(f))

# Old-way
lock.acquire()
try:
    do_some_work_critical()
finally:
    lock.release()

# New-way
with lock:
    do_some_work_critical()


class CM(object):
    def __enter__(self):
        print 'Now entering'
        return 50
    def __exit__(self, exctype, excinst, exctb):
        print 'Now exiting'
        print 'Exeception type:', exctype
        if exctype == KeyError:
            print "I know how to handle key errors"
            return True
        print 'I do not know about that one'


cm = CM()
with cm as x:
    print 'I got', x
    raise IndexError
    print 'Never gets here'

class File(object):
    def __init__(self, filename):
        pass
```

```
        def read(self):
            pass
        def write(self):
            pass
        def close(self):
            pass
        def __iter__(self):
            pass
        def __enter__(self):
            return self
        def __exit__(self, exctype, excinst, exctb):
            self.close()

    with File('stocks.txt') as f:
        print f.read()


    class Connection:
        def __exit__(self, exctype, excinst, exctb):
            if exctype is not None:
                self.rollback()
                return None
            self.commit()
```

9.5 Python's Command-line Tools

How to run the debugger:

```
$ python -m pdb tweet_bug.py
```

How to run the code tracer:

```
$ python -m trace --count fibo.py
```

How to make HTML documentation:

```
$ python -m pydoc -w iterator_school
```