Intro Python Course Notes

Release 0.9

Raymond Hettinger

April 06, 2012

CONTENTS

Location: Cisco – San Jose

Date: April 2, 2012

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

This file: http://dl.dropbox.com/u/3967849/sj9/links.txt

Download tool:

- http://dl.dropbox.com/u/3967849/sj9/download_class_files.py
- http://tinyurl.com/python-sj9

Course notes: http://dl.dropbox.com/u/3967849/sj9/IntroPython.pdf

CHAPTER

ONE

DOWNLOADING PYTHON2.7

- Windows:
  - http://www.python.org/ftp/python/2.7.2/python-2.7.2.msi
- Mac:
  - http://www.python.org/ftp/python/2.7.2/python-2.7.2-macosx10.6.dmg
  - http://www.activestate.com/activetcl/downloads

CHAPTER

TWO

WHAT TO REVIEW

The most important files to review after class are:

- **kap.py** Demonstrates string theory and effective uses of dicts, sets, and string iterpolation. It shows the technique of using Python to generate the input to another program, Graphviz, to make beautiful informative diagrams.
- **getflow.py** Demonstrates ways to scrape web-pages using regular expressions and with BeautifulSoup.
- **iterator_school.py** Shows *everything* you need to know about iterators, generators, and the most important built-in functions such as map(), filter(), sorted(), sum(), range(), xrange(), zip(), itertools.count(), and itertools.repeat().
- **dict_school.py** Covers test-driven development, how classes work, the most important special methods such as __iter__, __getitem__, __len__, __init__, etc. Also, covers dictionary methods in-depth.
- **client.py server.py server1.py server2.py** Shows the fundamentals of network programming including multi-threaded and forking servers.

CHAPTER

THREE

PRESENTATIONS AND SLIDES

The slides presented in class can be found at:

- http://dl.dropbox.com/u/3967849/sj9/PythonAwesome.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://effbot.org/zone/call-by-object.htm
- http://docs.python.org/library/functions.html
- http://greenteapress.com/semaphores/downey08semaphores.pdf
- http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- http://golubenco.org/2009/09/19/understanding-the-code-inside-tornado-the-asynchronous-web-server-powering-friendfeed/

CHAPTER

FIVE

RESOURCES

These are some resources mentioned in class:

- http://txt2re.com/
- http://interactive.blockdiag.com/graphviz/
- http://sms411.net/how-to-send-email-to-a-phone/
- http://www.doughellmann.com/PyMOTW/py-modindex.html
- http://sphinx.pocoo.org/
- https://www.mturk.com/mturk/welcome

CHAPTER

SIX

FAVORITE THIRD-PARTY MODULES

- http://www.noah.org/wiki/pexpect
- http://pyserial.sourceforge.net/shortintro.html
- http://docs.python-requests.org/en/latest/index.html
- https://github.com/toastdriven/itty
- http://www.crummy.com/software/BeautifulSoup/
- http://pypi.python.org/pypi/selenium
- http://docs.fabfile.org/en/1.4.1/index.html

CHAPTER

SEVEN

DAY ONE TOPICS

7.1 Access the Twitter REST API

Demonstrate:

- Basic python syntax including *def*, docstrings, comments, assignments, loops, *import*, and function calls.
- Show how to harness a third-party API
- Discuss the speed of Python (a glue language connecting together high-speed components such as a socket library and a JSON converter.

tweet.py:

```
import urllib, json

def twitter_search(topic, maxnumtweets=10):
    'Search Twitter for a given topic'
    # API described at https://dev.twitter.com/docs/api/1/get/search
    tw_url = "http://search.twitter.com/search.json?q=%s&rpp=%d" % \
             (topic, maxnumtweets)
    u = urllib.urlopen(tw_url)
    data = u.read()
    t = json.loads(data)
    for msg in t['results']:
        print msg['text']

if __name__ == '__main__':
    twitter_search('obama')
    twitter_search(maxnumtweets=20, topic='raymondh')
```

7.2 Python Numbers

In Python, numbers are immutable objects. Once created, they never change value. Math operations on numbers with create *new* number objects, but not change existing ones.

There are four number types:

Type	Examples
bool	True, False
int	10, -25, 134134582485134513515 28L
float	1.25, 3.14159265353
complex	3.2 + 4.5j

Python's *pow()* function has a three argument form that efficiently calculates large powers in a given modulus. This makes it trivially easy to experiment with RSA style public key encryption:

```
>>> private_key, public_key =  109182490673, 5551201688147

>>> msg = 9140002345
>>> code = pow(msg, 65537, public_key)
>>> print code
1485861488457
>>> plaintext = pow(code, private_key, public_key)
>>> print plaintext
9140002345
```

The public and private keypair can be created from *keygen* on Unix, *PuTTY* on Windows, or with a Python script such as the one at: http://code.activestate.com/recipes/577737-public-key-encryption-rsa/

7.3 String Theory

We reimplement *str.replace()* showing how it could be done from scratch. This demonstrates the basics on "string theory" including searching and slicing.

```
def myreplace(s, t, r):
    i = s.index(t)
    n = len(t)
    return s[:i] + r + s[i+n:]
```

7.4 Introduction to Python Classes

Show how Python classes are created using the *class* keyword and demonstrate how new capabilities are added one method at a time. In particular, show how Python syntax and top-level functions are linked to so-called *magic* methods or *special* methods.

class_demo.py:

```
class Dog:
    def __init__(self, name):
        self.name = name
    def __len__(self):
        return len(self.name)
    def __add__(self, other):
        newname = self.name + '~' + other.name
        return Dog(newname)
    def __repr__(self):
```

```
        return 'Dog(%r)' % self.name
    def __str__(self):
        return 'Hello, I am a dog named %s' % self.name
    def __getitem__(self, index):
        return index * 111
    def __call__(self, action):
        if action == 'fetch':
            print 'I am fetching'
        elif action == 'owner':
            print 'My master is Raymond'
        else:
            raise AttributeError('I do not know that one')


d = Dog('Fido')
e = Dog('Max')
```

CHAPTER

EIGHT

DAY TWO TOPICS

8.1 Truthiness

By default, all objects in Python are *True*. Containers and numbers are *False* if the container is empty or if the number is equal to zero. In addition, there is an object, *None*, that is always *False*:

Containers:

```
>>> bool([])                # Empty list
False
>>> bool([10, 20, 30])      # Non-empty list
True
>>> bool(())                # Empty tuple
False
>>> bool((10, 20, 30))      # Non-empty tuple
True
>>> bool('')                # Empty string
False
>>> bool('abc')             # Non-empty string
True
```

Numbers:

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> bool(0)
False
>>> bool(10)
True
>>> bool(0.0)
False
>>> bool(12.3)
True
>>> bool(Decimal(0))
False
>>> bool(Decimal('12.3'))
True
>>> bool(Fraction(0, 1))
False
```

```
>>> bool(Fraction(12, 34))
True
```

None:

```
>>> bool(None)
False
```

We know container is empty if its length is zero. We know number is zero if it is equal to zero. So, we can make classes that have truthiness:

```
>>> class A:
    'Number-like class'
    def __init__(self, s):
        self.s = s
    def __nonzero__(self):
        return True if self.s == 'T' else False

>>> bool(A('F'))
False
>>> bool(A('T'))
True

>>> class A:
    'Container-like class'
    def __init__(self, n):
        self.n = n
    def __len__(self):
        return self.n

>>> bool(A(0))
False
>>> bool(A(123))
True
```

8.2 Stock Portfolio Example

The objective of this section to learn to read real-time information from the internet, to parse data stored in files, and to do analysis on that data.

The portfolio reader shows to parse a delimited file. It demonstrates common string manipulations and type conversions. This code can be made into reusable functions:

stock.py:

```
'Collection of tools for analyzing a stock portfolio'

import urllib

url = 'http://download.finance.yahoo.com/d/quotes.csv?s=%s&f=sl1d1t1c1ohgv&e=.csv'

def getquote(symbol):
```

```
        'Get a current market quote from finance.yahoo.com'
        line = urllib.urlopen(url % symbol).read()
        return float(line.split(',')[1])

    def getquotes(*symbols):
        '''Return a dictionary with the prices for each symbol

            >>> getquotes('CSCO', 'IBM')
            {'CSCO': 23.99, 'IBM': 0.97}

        '''
        prices = {}
        for symbol in symbols:
            prices[symbol] = getquote(symbol)
        return prices

    def read_portfolio(filename, delimiter=','):
        'Read portfolio from a delimited file with the symbol,price, and shares'
        result = []
        for line in open(filename):
            symbol, shares, price = line.rstrip().split(delimiter)
            shares = int(shares)
            price = float(price)
            holding = symbol, shares, price
            result.append(holding)
        return result

    if __name__ == '__main__':
        print getquote('CSCO')
        print getquotes('CSCO', 'IBM')
        print read_portfolio('intro/stocks.txt')
```

The sample data for the example can be found here: http://dl.dropbox.com/u/3967849/sj9/stocks.txt

stocks.txt:

```
CSCO,100,18.04
WLP,200,45.03
CSCO,150,19.05
MSFT,250,80.56
IBM,500,22.01
WLP,250,44.23
GOOG,200,501.45
CSCO,175,19.56
MSFT,75,80.81
GOOG,300,502.65
IBM,150,25.01
```

List comprehensions are useful for doing data analysis. Here is the syntax for list comprehensions:

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

Here is how they can be used for data analysis:

analysis.py:

```
'''Show how to analyze data using list comprehensions,
  generator expressions, and dictionaries'''

import stock

port = stock.read_portfolio('stocks.txt')
prices = stock.getquotes('WLP', 'CSCO', 'IBM', 'GOOG', 'MSFT')

print '\nSymbols in the portfolio:'
print sorted(set(symbol for symbol, shares, price in port))
# SELECT DISTINCT symbol FROM port ORDER BY symbol;

print '\nList the quantities of CSCO purchases:'
print [shares for symbol, shares, price in port if symbol == 'CSCO']
# SELECT shares FROM port WHERE symbol = "CSCO";

print '\nList the prices of CSCO purchases:'
print [price for symbol, shares, price in port if symbol == 'CSCO']
# SELECT price FROM port WHERE symbol = "CSCO";

print '\nTotal cost of CSCO purchases:'
print sum(shares * price for symbol, shares, price in port if symbol == 'CSCO')
# SELECT SUM(shares * price) FROM port WHERE symbol = "CSCO";

print '\nCurrent value of CSCO purchases:'
print sum(shares * prices[symbol] for symbol, shares, price in port if symbol == 'CSCO
# SELECT SUM(p.shares * c.price) FROM port p, current_prices c
#        WHERE p.symbol = c.symbol AND p.symbol = "CSCO";

print '\nTotal cost basis for the whole portfolio:'
print sum(shares * price for symbol, shares, price in port )
# SELECT SUM(shares * price) FROM port;

print '\nCurrent value of the whole portfolio:'
print sum(shares * prices[symbol] for symbol, shares, price in port)
# SELECT SUM(p.shares * c.price) FROM port p, current_prices c
#        WHERE p.symbol = c.symbol;
```

8.3 Example using Kaprekar's constant

The code for *kap.py* shows:

- how to build sophisticated functions
- use automatically generated help from docstrings
- use automatic tests in the docstring
- effective use of sets, dicts, and lists
- how to generate output for a graphical language
- introduce the *graphviz* tool and the *dot* language

Here is the code for *kap.py*:

```
'Utilities for a complete analysis of the Kaprekar function'

def kap(n):
    ''' Compute on step of the Kaprekar process

        >>> kap(3524)
        3087
        >>> kap(3087)
        8352
        >>> kap(8352)
        6174
        >>> kap(6174)
        6174

    '''
    # http://en.wikipedia.org/wiki/6174_(number)
    s = '%04d' % n
    little = int(''.join(sorted(s)))
    big = int(''.join(sorted(s, reverse=True)))
    return big - little

def gen_graph():
    'Analyze the kap function by creating a graphviz diagram'
    # http://graphviz.org/
    # http://interactive.blockdiag.com/graphviz/
    # Run this with:  dot -Tpng kap.dot -o kap.png

    kapdict = {}
    for i in range(10000):
        source = '%04d' % i
        target = '%04d' % kap(i)
        kapdict[source] = target

    firsts = set(kapdict.keys()) - set(kapdict.values())
    rest = set(kapdict.keys()) - firsts
    assert len(firsts) + len(rest) == 10000

    # Loop over the 9945 firsts and group them based on their target
    # Given that 48, 84, and 158 all map to 8352
    # and the 39, 93, and 149 map to 9261
    # create a dictionary:
    # { 8352: [48, 84, 158], 9261: [39, 93, 149] }
    group = {}
    for i in sorted(firsts):
        target = kapdict[i]
        group.setdefault(target, []).append(i)

    print 'digraph {'
    print 'rankdir = "LR";'
    print 'edge [color="blue", fontsize=10];'

    for i in sorted(rest):
```

```
            print '%s -> %s;' % (i, kapdict[i])

        print 'node [shape="box"];'

        for target, inputs in sorted(group.items()):
            block = ', '.join(inputs[:3])
            print '"%s" -> %s [label="%d"];' % \
                  (block, target, len(inputs))

        print '}'

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

The output of *kap.gen_graph()* is a dot file suitable for input to graphviz:

```
digraph {
rankdir = "LR";
edge [color="blue", fontsize=10];
0000 -> 0000;
0999 -> 8991;
1089 -> 9621;
1998 -> 8082;
2088 -> 8532;
2178 -> 7443;
2997 -> 7173;
3087 -> 8352;
3177 -> 6354;
3267 -> 5265;
3996 -> 6264;
4086 -> 8172;
4176 -> 6174;
4266 -> 4176;
4356 -> 3087;
4995 -> 5355;
5085 -> 7992;
5175 -> 5994;
5265 -> 3996;
5355 -> 1998;
5445 -> 1089;
5994 -> 5355;
6084 -> 8172;
6174 -> 6174;
6264 -> 4176;
6354 -> 3087;
6444 -> 1998;
6534 -> 3087;
6993 -> 6264;
7083 -> 8352;
7173 -> 6354;
7263 -> 5265;
7353 -> 4176;
```

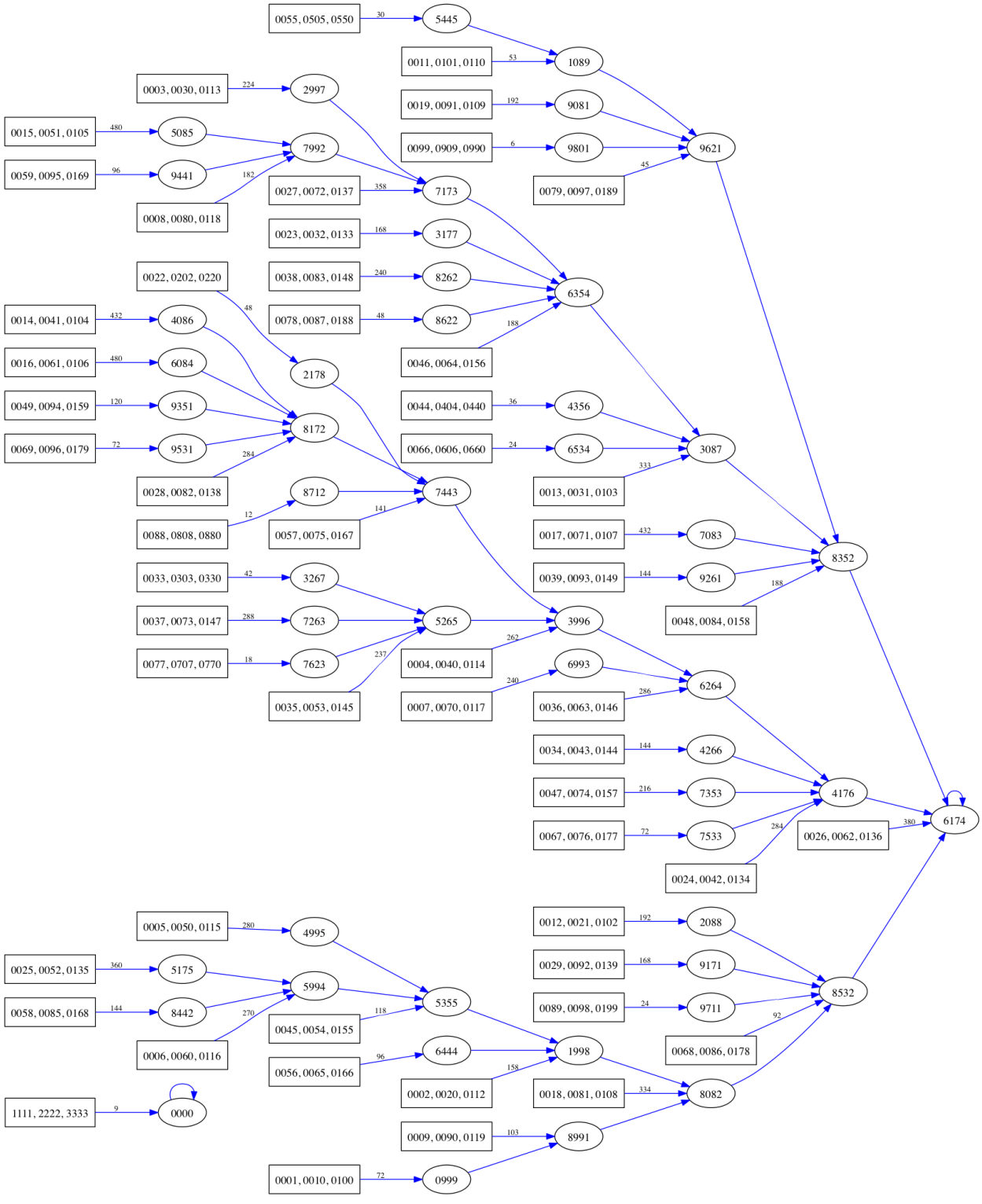
```
7443 -> 3996;
7533 -> 4176;
7623 -> 5265;
7992 -> 7173;
8082 -> 8532;
8172 -> 7443;
8262 -> 6354;
8352 -> 6174;
8442 -> 5994;
8532 -> 6174;
8622 -> 6354;
8712 -> 7443;
8991 -> 8082;
9081 -> 9621;
9171 -> 8532;
9261 -> 8352;
9351 -> 8172;
9441 -> 7992;
9531 -> 8172;
9621 -> 8352;
9711 -> 8532;
9801 -> 9621;
node [shape="box"];
"1111, 2222, 3333" -> 0000 [label="9"];
"0001, 0010, 0100" -> 0999 [label="72"];
"0011, 0101, 0110" -> 1089 [label="53"];
"0002, 0020, 0112" -> 1998 [label="158"];
"0012, 0021, 0102" -> 2088 [label="192"];
"0022, 0202, 0220" -> 2178 [label="48"];
"0003, 0030, 0113" -> 2997 [label="224"];
"0013, 0031, 0103" -> 3087 [label="333"];
"0023, 0032, 0133" -> 3177 [label="168"];
"0033, 0303, 0330" -> 3267 [label="42"];
"0004, 0040, 0114" -> 3996 [label="262"];
"0014, 0041, 0104" -> 4086 [label="432"];
"0024, 0042, 0134" -> 4176 [label="284"];
"0034, 0043, 0144" -> 4266 [label="144"];
"0044, 0404, 0440" -> 4356 [label="36"];
"0005, 0050, 0115" -> 4995 [label="280"];
"0015, 0051, 0105" -> 5085 [label="480"];
"0025, 0052, 0135" -> 5175 [label="360"];
"0035, 0053, 0145" -> 5265 [label="237"];
"0045, 0054, 0155" -> 5355 [label="118"];
"0055, 0505, 0550" -> 5445 [label="30"];
"0006, 0060, 0116" -> 5994 [label="270"];
"0016, 0061, 0106" -> 6084 [label="480"];
"0026, 0062, 0136" -> 6174 [label="380"];
"0036, 0063, 0146" -> 6264 [label="286"];
"0046, 0064, 0156" -> 6354 [label="188"];
"0056, 0065, 0166" -> 6444 [label="96"];
"0066, 0606, 0660" -> 6534 [label="24"];
"0007, 0070, 0117" -> 6993 [label="240"];
"0017, 0071, 0107" -> 7083 [label="432"];
```

```
"0027, 0072, 0137" -> 7173 [label="358"];
"0037, 0073, 0147" -> 7263 [label="288"];
"0047, 0074, 0157" -> 7353 [label="216"];
"0057, 0075, 0167" -> 7443 [label="141"];
"0067, 0076, 0177" -> 7533 [label="72"];
"0077, 0707, 0770" -> 7623 [label="18"];
"0008, 0080, 0118" -> 7992 [label="182"];
"0018, 0081, 0108" -> 8082 [label="334"];
"0028, 0082, 0138" -> 8172 [label="284"];
"0038, 0083, 0148" -> 8262 [label="240"];
"0048, 0084, 0158" -> 8352 [label="188"];
"0058, 0085, 0168" -> 8442 [label="144"];
"0068, 0086, 0178" -> 8532 [label="92"];
"0078, 0087, 0188" -> 8622 [label="48"];
"0088, 0808, 0880" -> 8712 [label="12"];
"0009, 0090, 0119" -> 8991 [label="103"];
"0019, 0091, 0109" -> 9081 [label="192"];
"0029, 0092, 0139" -> 9171 [label="168"];
"0039, 0093, 0149" -> 9261 [label="144"];
"0049, 0094, 0159" -> 9351 [label="120"];
"0059, 0095, 0169" -> 9441 [label="96"];
"0069, 0096, 0179" -> 9531 [label="72"];
"0079, 0097, 0189" -> 9621 [label="45"];
"0089, 0098, 0199" -> 9711 [label="24"];
"0099, 0909, 0990" -> 9801 [label="6"];
}
```

The dot file can be converted to an image using:

```
$ dot -Tsvg kap.dot -o kap.svg
```

The resulting image is shown on the next page.


0055, 0505, 0550
30
5445
0011, 0101, 0110
53
1089
0003, 0030, 0113
224
2997
0019, 0091, 0109
192
9081
0015, 0051, 0105
480
5085
0099, 0909, 0990
6
9801
9621
0059, 0095, 0169
96
9441
182
7992
45
0027, 0072, 0137
358
7173
0079, 0097, 0189
0008, 0080, 0118
0023, 0032, 0133
168
3177
0022, 0202, 0220
0038, 0083, 0148
240
8262
6354
48
0014, 0041, 0104
432
4086
0078, 0087, 0188
48
8622
188
0016, 0061, 0106
480
6084
2178
0046, 0064, 0156
0049, 0094, 0159
120
9351
0044, 0404, 0440
36
4356
8172
0069, 0096, 0179
72
9531
0066, 0606, 0660
24
6534
3087
284
333
0028, 0082, 0138
8712
7443
0013, 0031, 0103
12
141
0088, 0808, 0880
0057, 0075, 0167
0017, 0071, 0107
432
7083
8352
0033, 0303, 0330
42
3267
0039, 0093, 0149
144
9261
188
0037, 0073, 0147
288
7263
5265
3996
0048, 0084, 0158
262
0077, 0707, 0770
18
7623
237
0004, 0040, 0114
6993
240
6264
0035, 0053, 0145
0007, 0070, 0117
0036, 0063, 0146
286
0034, 0043, 0144
144
4266
0047, 0074, 0157
216
7353
4176
6174
0067, 0076, 0177
72
7533
284
0026, 0062, 0136
380
0024, 0042, 0134
0005, 0050, 0115
280
4995
0012, 0021, 0102
192
2088
0025, 0052, 0135
360
5175
0029, 0092, 0139
168
9171
5994
0058, 0085, 0168
144
8442
270
0089, 0098, 0199
24
9711
8532
92
118
5355
0045, 0054, 0155
0006, 0060, 0116
0068, 0086, 0178
96
6444
1998
158
0056, 0065, 0166
0002, 0020, 0112
0018, 0081, 0108
334
8082
1111, 2222, 3333
9
0000
0009, 0090, 0119
103
8991
0001, 0010, 0100
72
0999

8.4 Docstrings

The following code hints at how the built-in *help* function works. It also shows how to generate HTML templates and how to use docstrings can also be used for automated testing:

docstring.py:

```
'Collection of utilities that act on docstrings'

def mytest(func):
    'Conduct automated testing of a docstring'
    s = func.__doc__
    k = 0
    attempted = 0
    failed = 0
    while True:
        i = s.find('>>> ', k+1)
        if i == -1:
            return attempted, failed
        attempted += 1
        j = s.find('\n', i+1)
        k = s.find('\n', j+1)
        command = s[i+4:j]
        expected = s[j:k].strip()
        actual = str(eval(command))
        if expected == actual:
            pass
        else:
            failed += 1
            print 'Sad!'
            print 'Ran the test: %r' % command
            print 'Expected: %r' % expected
            print 'But got: %r' % actual
            print '=' * 20

def myhelp(func):
    'Emulate the built-in help() function'
    name = func.__name__
    doc = func.__doc__
    print 'My custom help'
    print '--------------'
    print 'Function name: ', name
    print 'Docstring:'
    print doc
    print '=============='
    print

template = '''
My custom help with a template
------------------------------
Function name: %s
Docstring:
%s
```

```
==============================
'''

def myhelp2(func):
    'Same help but using a template'
    print template % (func.__name__, func.__doc__)

html_template = '''
<html>
<head><title>Custom Help</title></head>
<body>
    <h1> <em>%s</em>
    <hr>
    <pre>
%s
    </pre>
</body>
</html>
'''

def myhtmlhelp(func):
    'Emit function help in HTML'
    print html_template % (func.__name__, func.__doc__)


if __name__ == '__main__':
    from kap import kap
    print mytest(kap)
```

8.5 REPL and Eval

Here, we create a new REPL loop as a front-end for Python:

repl.py:

```
#! /usr/bin/env python2.7
'Create a custom REPL for Python'

# http://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop


print '=' * 50
print 'Welcome to the powerful custom Python REPL'
print '.' * 50
print

prompt = '$ '

In = []
Out = []
while True:
    prompt = 'In [%d]: ' % len(In)
```

```
        request = raw_input(prompt)

        # convenience feature:
        # transform 'pow 2, 5'  into 'pow(2, 5)'
        if request.startswith(('pow', 'bin', 'oct', 'divmod')) \
           and '(' not in request:
            fields = request.split()
            request = '%s(%s)' % (fields[0], ' '.join(fields[1:]))
            print '--> ', request

        result = eval(request)
        print 'Out [%d]: %r' % (len(Out), result)
        print
        In.append(request)
        Out.append(result)
```

And the following code suggests hints the *eval()* function works. Essentially, its job is to translate Python syntax into a series of method calls.

myeval.py:

```
'Simple version of eval() implemented manually'

def myeval(s):
    ''' Evaluate a string expression by dispatching to a special method

        >>> myeval('30 + 40')
        70
        >>> myeval('315 - 25')
        290
        >>> myeval('42 * 5')
        210
        >>> myeval('2 ^ 5')
        32

    '''
    lop, operator, rop = s.split()
    lop = int(lop)
    rop = int(rop)
    if operator == '+':
        return lop.__add__(rop)
    elif operator == '-':
        return lop.__sub__(rop)
    elif operator == '*':
        return lop.__mul__(rop)
    elif operator == '^':
        # implement MS Excel style exponentation
        return lop.__pow__(rop)
    else:
        raise SyntaxError('I do not know the operator: %r' % operator)

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

8.6 Sorted

The built-in *sorted* function works like this:

```
def mysorted(iterable, reverse=False):
    'Emulate the built-in sorted() function'
    s = list(iterable)
    s.sort(reverse=reverse)
    return s
```

CHAPTER

NINE

DAY THREE TOPICS

9.1 Syntax for making lists and tuples

How to make tuples and lists

```
>>> # Length 0:
>>> []
>>> ()

>>> # Length 1:
>>> [10]  [10,]
>>> 10,   (10,)
```

9.2 Automatic Documentation

How to generate documentation automatically

```
>>> python -m pydoc -w kap stock iterator_school docstring myeval
```

This creates the following documentation:

- http://dl.dropbox.com/u/3967849/sj9/kap.html
- http://dl.dropbox.com/u/3967849/sj9/stock.html
- http://dl.dropbox.com/u/3967849/sj9/iterator_school.html
- http://dl.dropbox.com/u/3967849/sj9/docstring.html
- http://dl.dropbox.com/u/3967849/sj9/myeval.html

9.3 Iterator School

Here is code that develops the iterator protocol from scratch:

iterator_school.py:

```
''' Lessons on the iterator protocol

ITERABLE:
   * That which can be used on the right side of a for-loop
   * Anything that can be looped-over
   * Any object that implement a __iter__ method
     whose responsibility is to return an iterator.
   Examples:  str, list, tuple, set, dict, file, urls

ITERATOR:
   * An object with state responsible for giving you values
     one-at-a-time
   * Any object that implements a next() method responsible for:
     - give you a value
     - update the state
     - raise StopIteration when its done
   * Must be self-iterable
     - Need to have an __iter__ that returns self

Convenience functions:
   iter(obj)      -->   obj.__iter__()
   next(obj)      -->   obj.next()

'''

def mylist(iterable):
    '''Emulate the built-in list() function
       whose job is to loop over an iterable
       and put its values in a new list.

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

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

def mylist2(iterable):
    ''' Second version of list() using a for-loop

        >>> mylist2('cat')
        ['c', 'a', 't']

    '''
    result = []
    for c in iterable:
```

```
            result.append(c)
        return result

    def mysum(iterable, start=0):
        ''' Emulate the built-in sum() function
            Adds all the values in an iterable.

            >>> mysum([10, 20, 30])
            60
            >>> mysum([10, 20, 30], 1)
            61

        '''
        total = start
        for x in iterable:
            total += x
        return total

    def myrange(a, b=None, step=1):
        ''' Emulate the built-in range() function.
            That produces a list on consecutive values.

            >>> myrange(10)
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
            >>> myrange(995, 1000)
            [995, 996, 997, 998, 999]
            >>> myrange(0, 30, 5)
            [0, 5, 10, 15, 20, 25]

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

    def mymin(iterable):
        '''Return the lowest value in the iterable

            >>> mymin([10, 5, 30])
            5
            >>> mymin([5, 10, 30])
            5

        '''
        it = iter(iterable)
```

```
    lowest = next(it)
    for x in it:
        if x < lowest:
            lowest = x
    return lowest

class MyXRange:
    'Emulate xrange() using the traditional iterator protocol'
    # ITERABLE
    def __init__(self, stop):
        self.stop = stop
    def __iter__(self):
        return XrangeIterator(self.stop)

class XrangeIterator:
    'Stateful object for looping over an xrange object'
    # ITERATOR
    def __init__(self, stop):
        self.i = 0
        self.stop = stop
    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):
    ''' Emulate the built-in 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 myzip(iterable1, iterable2):
    ''' Emulate the itertools.izip() function
        for lock-step iteration.

        >>> s = 'manny mo jack'.split()
        >>> t = 'red green blue'.split()
        >>> list(myzip(s, t))
        [('manny', 'red'), ('mo', 'green'), ('jack', 'blue')]

    '''
    it1 = iter(iterable1)
    it2 = iter(iterable2)
```

```
        while True:
            try:
                v1 = next(it1)
                v2 = next(it2)
            except StopIteration:
                break
            yield v1, v2

    def count(start=0, step=1):
        'Count to infinity'
        i = start
        while True:
            yield i
            i += step

    import time

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

    def repeat(value):
        'Repeat the same value ad-infinitum'
        while True:
            yield value

    def imap(func, iterable):
        'Apply a function to every element of the iterable'
        for x in iterable:
            yield func(x)

    def ifilter(predicate, iterable):
        'Emit values from the iterable where the predicate is True.'
        for x in iterable:
            if predicate(x):
                yield x

    def myenumerate(iterable, start=0):
        '''Emulate the built-in enumerate() function
           that returns tuples with the values from the iterable
           preceded by their corresponding position index.

           >>> list(enumerate('abc', 1))
           [(1, 'a'), (2, 'b'), (3, 'c')]

        '''
        i = start
        for x in iterable:
            yield i, x
            i += 1

    if __name__ == '__main__':
```

```
import doctest
print doctest.testmod()
```

9.4 Dictionary Example

In his example shows how to build dictionaries from scratch. The objectives are to demonstrate test driven development, show how to use the *unittest* module, show how Python classes are built-in, learn the most common special (magic) methods, and to learn how dictionaries are implemented.

dict_school.py:

```
'Create a dictionary from scratch'

class Dict:
    'Dictionary-like object based on lists'

    def _set_up_the_desks(self, n):
        self.n = n
        self.desks = [[] for i in range(self.n)]
        self.folders = [[] for i in range(self.n)]

    def _find_desk_and_folder(self, key):
        i = abs(hash(key)) % self.n
        desk = self.desks[i]
        folder = self.folders[i]
        return desk, folder

    def _find_position_at_a_desk(self, desk, key):
        try:
            j = desk.index(key)
        except ValueError:
            raise KeyError(key)
        return j

    def __init__(self):
        self._set_up_the_desks(8)

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        desk, folder = self._find_desk_and_folder(key)
        desk.append(key)
        folder.append(value)
        if len(desk) > 4:
            # we need a bigger boat
            items = self.items()
            self._set_up_the_desks(self.n * 4)
            for key, value in items:
                self[key] = value

    def __getitem__(self, key):
```

```
        desk, folder = self._find_desk_and_folder(key)
        j = self._find_position_at_a_desk(desk, key)
        return folder[j]

    def __delitem__(self, key):
        desk, folder = self._find_desk_and_folder(key)
        j = self._find_position_at_a_desk(desk, key)
        del desk[j]
        del folder[j]

    def __iter__(self):
        for desk in self.desks:
            for key in desk:
                yield key

    def __len__(self):
        return sum(len(desk) for desk in self.desks)

    ### Code below here does not know about the internals ###

    def __contains__(self, key):
        try:
            value = self[key]
        except KeyError:
            return False
        return True

    def keys(self):
        'Return a list of the keys in the dictionary'
        return list(self)

    def values(self):
        'Return a list of the values in the dictionary'
        return [self[k] for k in self]

    def items(self):
        'Return a list of the key and value pairs'
        return [(k, self[k]) for k in self]

    def get(self, key, default=None):
        'Return value for a key or the default if not found'
        return self[key] if key in self else default

    def setdefault(self, key, default):
        'Return value for a key. If not found set and return the default'
        if key in self:
            return self[key]
        self[key] = default
        return default

    def copy(self):
        'Copy all the keys and values to a new dictionary'
        c = Dict()
```

```
            for key, value in self.items():
                c[key] = value
            return c

        def clear(self):
            'Remove all the keys from a dictionary'
            for key in self.keys():
                del self[key]
```

Here is the corresponding test code:

test_dict.py:

```
import unittest
from dict_school import Dict

class TestSequenceFunctions(unittest.TestCase):

    def test_basics(self):
        d = Dict()                  # test instantiation

        d['raymond'] = 'red'        # sets a key

        # tests retrieval
        self.assertEqual(d['raymond'], 'red')
        with self.assertRaises(KeyError):
            d['roger']

        # test changing a value
        d['raymond'] = 'crimson'
        self.assertEqual(d['raymond'], 'crimson')

        # tests deletion
        del d['raymond']
        with self.assertRaises(KeyError):
            d['raymond']

    def test_iteration(self):
        d = Dict()
        d['raymond'] = 'red'
        d['rachel'] = 'blue'
        d['matthew'] = 'green'
        self.assertEqual(set(d), {'raymond', 'rachel', 'matthew'})

    def test_length(self):
        d = Dict()
        self.assertEqual(len(d), 0)
        d['raymond'] = 'red'
        self.assertEqual(len(d), 1)
        d['rachel'] = 'blue'
        self.assertEqual(len(d), 2)
        d['matthew'] = 'green'
        self.assertEqual(len(d), 3)
```

```
    def test_contains(self):
        d = Dict()
        d['raymond'] = 'red'
        self.assertTrue('raymond' in d)
        d['rachel'] = 'blue'
        self.assertTrue('rachel' in d)
        d['matthew'] = 'green'
        self.assertTrue('matthew' in d)
        self.assertTrue('roger' not in d)

if __name__ == '__main__':
    unittest.main(exit=False)
```

9.5 Looping Idioms

The primary looping idioms in Python are

```
for x in s:  ...

for x, y in zip(s, t): ...

for x in reversed(s): ...

for i, x in enumerate(s): ...

for x in sorted(s): ...
```

9.6 Evolution of Python Looping Techniques

The following shows the evolution of techniques for summing squares on *n* integers. In the beginning, it took multiple lines of code and was memory intensive. In the end, the expression was succinct and it ran in very little memory:

squares.py:

```
'Functions for computing the sum of squares'

# Sugar from 1990
def sos1(n):
    'Compute the sum of squares upto n but not including n'
    result = []
    for i in range(n):
        result.append(i * i)
    return sum(result)

# Chocolate from 2000
def sos2(n):
    'List comprehension version'        # LISTCOMP
    return sum([i*i for i in range(n)])
```

```
# Peanut butter from 2003
def squares(n):
    'Generator example'
    for i in xrange(n):
        yield i * i

def sos3(n):
    return sum(squares(n))

# KitKat bars from 2006
def sos4(n):
    'Generator expressions version'      # GENEXP
    return sum(i*i for i in xrange(n))

print sos1(1000)
print sos2(1000)
print sos3(1000)
print sos4(1000)
```

CHAPTER

TEN

DAY FOUR TOPICS

10.1 Handy functions in the OS module

Popular commands in the OS module

```
>>>  os.getcwd()         # show the current working directory
>>>  os.listdir('.')     # list the files in a given directory
>>>  os.chdir(somedir)   # change to another directory
```

10.2 Using the PDB debugger

To run the debugger from the command-line (not the interactive prompt):

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

The most common debugger commands are:

```
c   - continue until an exception or normal finish
n   - next (run to the next line)
l   - list the source and show where you are
b   - set a breakpoint
p   - print a variable
pp  - pretty print a variable
q   - quit
```

10.3 Logging

To setup a logger:

```
>>> import logging
>>> logging.basicConfig(filename='example.log',level=logging.DEBUG)
```

To make log entries:

```
>>> logging.info(msg)
>>> logging.debug(msg)
>>> logging.error(msg)
>>> logging.critical(msg)
>>> logging.warn(msg)
```

10.4 Scraping Web Pages

Here we develop the skills for extracting data from webpages by using two techniques:

- Extract a single datum using regular expressions
- Use *BeautifulSoup* to parse mal-formed HTML and extract data from tables

This following code serves as a good model for the two techniques:

getflow.py:

```
url = 'http://www.woodlands-junior.kent.sch.uk/Homework/rivers/longest.htm'

import re
import urllib
import BeautifulSoup


def amazon_flow():
    ''' Get the real-time flow rate of the Amazon river
        in cubic meters per second.

        >>> 100000 < amazon_flow() < 200000
        True

    '''
    u = urllib.urlopen(url)
    data = u.read()
    data.replace('\n', ' ')    # ignore line endings
    mo = re.search(r'average ([0-9,]+) cubic', data)
    return int(mo.group(1).replace(',', ''))

def get_river_table():
    'Return a list of tuples for river data'
    u = urllib.urlopen(url)
    data = u.read()
    soup = BeautifulSoup.BeautifulSoup(data)
    t = soup.find('table', border=1)

    result = []
    for row in t.findAll('tr'):
        rowlist = []
        for col in row.findAll('td'):
            rowlist.append(str(col.text))
        result.append(tuple(rowlist))
```

```
    return result


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

The sample for this exercise is at: http://bit.ly/py_rivers_ex

The source for BeautifulSoup can be found at: http://dl.dropbox.com/u/3967849/sj6/BeautifulSoup.py

The technique for extracting groups using regular expressions is

```
>>> import re
>>> s = 'Today is 3/5/2012 and there are 200 day left.'
>>> month, day, year = re.search(r'(\d+)/(\d+)/(\d+)', s).groups()
>>> int(month)
3
>>> int(day)
5
>>> int(year)
2012
```

10.5 Networking

In TCP communication, the *client* is the one that initiates the call and the *server* the one that answers. Once the call is connected, the distinction between the two disappears. Either side, can then send, receive, or do both at the same time.

The code for a basic client is:

client.py:

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.connect(('www.python.org', 80))
request = 'GET /index.html HTTP/1.1\nHost: www.python.org\nConnection: close\n\n'
s.send(request)
d = s.recv(20000)
s.close()
print d.replace('\r\n', '\n')
```

The code for a basic server is:

server.py:

```
from socket import *
import time
import os

def serve(handler):
    s = socket(AF_INET,SOCK_STREAM)
```

```
    s.bind(('', 9000))
    s.listen(5)
    print 'Server up, running, and waiting for call'
    try:
        while True:
            c,a = s.accept()
            handler(c, a)
    finally:
        s.close()

def handler(c, a):
    print "Received connection from", a
    c.send("Hello %s\r\n" % a[0])
    c.send('The current time is %s\r\n' % time.ctime())
    c.send('\r\n'.join(os.listdir('.')) + '\r\n\r\n')
    for i in range(10):
        time.sleep(1)
        c.send('Counting: %d\r\n' % i)
    c.close()

if __name__ == '__main__':
    serve(handler)
```

Only a minor modification is necessary to make the server multi-threaded:

server1.py:

```
from socket import *
import time
import os
import threading

def serve(handler):
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(('', 9000))
    s.listen(5)
    print 'Server up, running, and waiting for call'
    try:
        while True:
            c,a = s.accept()
            threading.Thread(target=handler, args=(c, a)).start()
    finally:
        s.close()

def handler(c, a):
    print "Received connection from", a
    c.send("Hello %s\r\n" % a[0])
    c.send('The current time is %s\r\n' % time.ctime())
    c.send('\r\n'.join(os.listdir('.')) + '\r\n\r\n')
    for i in range(10):
        time.sleep(1)
        c.send('Counting: %d\r\n' % i)
    c.close()
```

```
if __name__ == '__main__':
    serve(handler)
```

Likewise, on minor modifications are necessary to make the server use multiple processes:

server2.py:

```
from socket import *
import time
import os

def serve(handler):
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(('', 9000))
    s.listen(5)
    print 'Server up, running, and waiting for call'
    try:
        while True:
            c,a = s.accept()
            pid = os.fork()
            if pid == 0:
                # We must be the clone
                handler(c, a)
                os._exit(0)
    finally:
        s.close()

def handler(c, a):
    print "Received connection from", a
    c.send("Hello %s\r\n" % a[0])
    c.send('The current time is %s\r\n' % time.ctime())
    c.send('\r\n'.join(os.listdir('.')) + '\r\n\r\n')
    for i in range(10):
        time.sleep(1)
        c.send('Counting: %d\r\n' % i)
    c.close()

if __name__ == '__main__':
    serve(handler)
```

10.6 Threading

The most common bugs in threaded programs are race conditions. There is a straight forward solution: EVERY shared resource (including global variables, files, print, input, etc) should be put in a single thread that has EXCLUSIVE access to the resource. ALL inter-thread communication should done through a *Queue*.

Here's an example that shows how access to the *print* statement can be isolated in a thread and accessed only through a print queue:

threading_demo.py:

```
''' Raymond's threading advice:
        * Every shared resource should be in its own thread
        * The thread should only communicate via Queue objects
        * In this way, lies happiness and no race conditions
'''

import threading
import Queue

######## Printer Resource ################################

print_queue = Queue.Queue()

def printer():
    'I have EXCLUSIVE access to the printer'
    while True:
        job = print_queue.get()
        print '=' * 20, ' New print job ', '=' * 20
        for line in job:
            print line

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


######### Counter Resource ###################################

counter_queue = Queue.Queue()

counter = 0

def count_updater():
    'I have exclusive access to the counter'
    global counter

    while True:
        increment = counter_queue.get()
        counter += increment

u = threading.Thread(target=count_updater)
u.daemon = True
u.start()


####### Main thread for launching workers ##############################

def worker():
    counter_queue.put(1)
    print_queue.put(['The current value of the counter is', str(counter)])

print_queue.put(['Starting up all the worker threads'])
for i in range(10):
```

```
t = threading.Thread(target=worker)
t.start()
```