PYTHON LAB BOOK

Python For Programmers
UCSC Extension Online

Lab 13 Function Fancies

Topics

- Function protocols: variable length argument lists
- Formatted printing using a dictionary for replacement
- Unpacking sequences and dictionaries
- Generators (Optional)
- Decorators (Optional)

v.4.0

```
Forwards:
7 Bruce Penge
1 Maureen Mezzabo
8 Samantha Smith
6 Juvenal Ramirez
Midfielders:
4 Xavier Perra
2 Laura Dot
5 Malcolm Diamond
9 Mary Bart
Defenders:
3 Linda Jarvis
Goalies:
11 Jose Acosta
10 Tracy Lowe
```

lab12_1.py

```
 1 #!/usr/bin/env python
 2 """soccer_team.py with getattr and setattr, as much as possible."""
 3 import sys
 4 from soccer_team import *
 5
 6 this_module = sys.modules[__name__]
 7
 8 def ProcessTeam(stream):
 9     global positions
10     positions = []
11     for line in stream:
12         line = line.strip()
13         if not line:
14             continue
15         if line.endswith(':'):
16             position = line[:-1]
17             setattr(this_module, position, [])
18             positions += [position]
19             continue
20         details = line.split(' ', 1)
21         setattr(this_module, position,
22                 getattr(this_module, position) + [details])
23         print 'Yeh %s #%s ' % (details[1], details[0]) +\
24               eval("Notify%s()" % position)
25     return stream.name, positions
26
27 def PrintTeam(team_name, positions):
28     print team_name + ':'
29     for each in positions:
30         print '  %s:' % each
31         for player in sorted(getattr(this_module, each)):
32             print '    ' + ': '.join(player)
33
34 if __name__ == '__main__':
35     main()
36
37 """ Same output as soccer_team.py"""
```

lab12_2.py

```
#!/usr/bin/env python
"""lab12_1.py -- Adding up the file sizes in the current directory,
three ways, and comparing them."""
import os
import subprocess
__pychecker__ = 'no-local'

def AccuracyTest():
    print "os.listdir:", AddFilesOsListdir()
    print "os.popen:  ", AddFilesOsPopen()
    print "subprocess:", AddFilesSubprocess()

def AddFilesOsListdir():
    total = 0
    files = os.listdir('.')
    for f in files:
        if os.path.isdir('./' + f):
            continue
        total += os.path.getsize('./' + f)
    return total

def AddFilesOsPopen():
    return TotalLsSize(os.popen("ls -al"))

def AddFilesSubprocess():
    return TotalLsSize(subprocess.Popen(["ls", "-al"],
                                        stdout=subprocess.PIPE).stdout)
def ProfileTest():
    for i in range(100):
        AddFilesOsListdir()
        AddFilesOsPopen()
        AddFilesSubprocess()

def TotalLsSize(file_obj):
    total = 0
    for line in file_obj:
        if line[0] == 'd':
            continue
        parts = line.split()
        if len(parts) != 9:
            continue
        total += int(parts[4])
    return total

```

```
45 def main():
46     AccuracyTest()
47     import profile
48     profile.run('ProfileTest()')
49
50 if __name__ == '__main__':
51     main()
52 """
53 $ lab12_1.py
54 os.listdir: 26298
55 os.popen:   26298
56 subprocess: 26298
57          30376 function calls in 1.872 CPU seconds
58
59    Ordered by: standard name
60
61    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
62       101    0.004    0.000    0.004    0.000 :0(WEXITSTATUS)
63       101    0.012    0.000    0.012    0.000 :0(WIFEXITED)
64       101    0.000    0.000    0.000    0.000 :0(WIFSIGNALED)
65        77    0.004    0.000    0.004    0.000 :0(append)
66       300    0.016    0.000    0.016    0.000 :0(close)
67       200    0.012    0.000    0.012    0.000 :0(fcntl)
68       100    0.004    0.000    0.004    0.000 :0(fdopen)
69       100    0.068    0.001    0.068    0.001 :0(fork)
70       200    0.012    0.000    0.012    0.000 :0(isinstance)
71      5400    0.100    0.000    0.100    0.000 :0(len)
72       100    0.032    0.000    0.032    0.000 :0(listdir)
73       200    0.000    0.000    0.000    0.000 :0(pipe)
74       100    0.108    0.001    0.108    0.001 :0(popen)
75         1    0.000    0.000    0.000    0.000 :0(range)
76       100    0.016    0.000    0.016    0.000 :0(read)
77        78    0.004    0.000    0.004    0.000 :0(remove)
78         1    0.004    0.004    0.004    0.004 :0(setprofile)
79      5400    0.104    0.000    0.104    0.000 :0(split)
80      5300    0.236    0.000    0.236    0.000 :0(stat)
81       178    0.004    0.000    0.004    0.000 :0(waitpid)
82         1    0.000    0.000    1.868    1.868 <string>:1(<module>)
83       100    0.156    0.002    0.872    0.009 lab12_1.py:12(AddFilesOsListdir)
84       100    0.024    0.000    0.440    0.004 lab12_1.py:21(AddFilesOsPopen)
85       100    0.036    0.000    0.540    0.005 lab12_1.py:24(AddFilesSubprocess)
86 ... The rest of the output is irrelevant.
87 It seems fishy that os.listdir() takes longer than both subprocess.Popen()
88 and os.popen().  Maybe somehow we are comparing apples and oranges?
89 $ """
```

lab12_3.py

```
#!/usr/bin/env python
"""lab12_2.py  Reports the ip address of this machine.
Linux only."""
import subprocess

def GetIP():
    output = subprocess.Popen("/sbin/ifconfig",
                              stdout=subprocess.PIPE).stdout
    for line in output:
        address_at = line.find("inet addr:")
        if address_at == -1:
            continue
        return line[address_at + 10:].split()[0]

if __name__ == '__main__':
    print GetIP()

"""
$ lab12_2.py
10.0.0.153
$
"""
```

more.py

```
 1 #!/usr/bin/env python
 2 """Variable length argument lists are supported big time."""
 3
 4 es = {}
 5
 6 def NewWords(name, language='Spanish', *more, **and_more):
 7     print name, 'translating to', language + ':'
 8     for word in more:
 9         es[word] = raw_input('%s: ' % word)
10     if and_more:
11         print "New words:"
12         for word in and_more:
13             print "%s: %s" % (word, and_more[word])
14         es.update(and_more)
15     print 'Thank you', name
16
17 def PrintEs():
18     for word in sorted(es):
19         print '%s:%s' % (word, es[word])
20
21 def main():
22     NewWords('Emeliano')
23     NewWords('Pancho', 'espa~ol', 'carrot', 'peanut')
24     NewWords('Joaquin', 'carrot', 'grapefruit')
25     NewWords('Luis', 'spanish', 'orange', 'butter', bread='pan',
26               cheese='queso')
27     NewWords('Maria', strawberry='fresa')
28     PrintEs()
29
30 if __name__ == '__main__':
31     main()
32
33
34
35
36
37
38
39
40
41
42
43
44
```

```
45
46
47
48 """
49 $ more.py
50 Emeliano translating to Spanish:
51 Thank you Emeliano
52 Pancho translating to espa~ol:
53 carrot: zanahoria
54 peanut: cacajuate
55 Thank you Pancho
56 Joaquin translating to carrot:
57 grapefruit: toronja
58 Thank you Joaquin
59 Luis translating to spanish:
60 orange: naranja
61 butter: montequilla
62 New words:
63 cheese: queso
64 bread: pan
65 Thank you Luis
66 Maria translating to Spanish:
67 New words:
68 strawberry: fresa
69 Thank you Maria
70 bread:pan
71 butter:montequilla
72 carrot:zanahoria
73 cheese:queso
74 grapefruit:toronja
75 orange:naranja
76 peanut:cacajuate
77 strawberry:fresa
78 $
79 """
```

breakfast.py

```
#!/usr/bin/env python
""" Remember this lab solution:
---
def Breakfast(meat="bacon", eggs="over easy",
              potatoes="hash browns", toast="white",
              beverage="coffee"):
    print "Here is your %s and %s eggs with %s and %s toast."\
          % (meat, eggs, potatoes, toast)
    print "Can I bring you more %s?" % beverage

Breakfast()
Breakfast("ham", "basted", "cottage cheese", "cinnamon", "orange juice")
Breakfast("sausage", toast="white", beverage="chai")
----
Here's an even more flexible way, using the '%' operator for strings
again, but this time with a dictionary -- as well as variable length
keyword calls.
"""

def Breakfast(**substitutions):
    order = {'meat':'bacon','eggs':'over easy',
             'potatoes':'hash browns',
             'toast':'white','beverage':'coffee'}
    # updating values in order from substitutions
    order.update(substitutions)
    # string replacement from a dictionary
    print "Here is your %(meat)s and %(eggs)s eggs with %(potatoes)s "\
          "and %(toast)s toast." % order
    print "Can I bring you more %(beverage)s?" % order

def main():
    Breakfast()
    Breakfast(meat="sausage", toast="wheat", beverage="chai")

if __name__ == '__main__':
    main()

"""
$ breakfast.py
Here is your bacon and over easy eggs with hash browns and white toast.
Can I bring you more coffee?
Here is your sausage and over easy eggs with hash browns and wheat toast.
Can I bring you more chai?
$ """
```

unpack.py

```
#!/usr/bin/env python

""" unpack.py - sequences and dictionaries can be unpacked into
a argument list with the * and ** operators. """

def PrintThings(a, b, c, *tup_args, **dict_args):
    print 'First three args are required:', a, b, c,
    if tup_args:
        print '\n  From tup_args:',
        for stuff in tup_args:
            print stuff,
    if dict_args:
        print '\n  From dict_args:',
        for k in dict_args.keys():
            print k, '->', dict_args[k],
    print

def PrintDict(uno=8, dos=10, **rest):
    print uno, dos, rest

def main():
    tup = ('Eat','chocolate','candy')
    PrintThings(*tup)
    PrintThings('this', 'that', 'other')
    L = ['live','life','in','the', 'fast', 'lane']
    PrintThings(*L)
    D = {'uno':1, 'dos':2, 'tres':3, 'cuatro':4}
    PrintThings(*tup, **D)
    S = 'Too fast'
    PrintThings(*S)
    print "PrintDict(**D)"
    PrintDict(**D)
    print "PrintThings(*D)"
    PrintThings(*D)

if __name__ == '__main__':
    main()
```

```
45
46
47
48 """
49 $ unpack.py
50 First three args are required: Eat chocolate candy
51 First three args are required: this that other
52 First three args are required: live life in
53   From tup_args: the fast lane
54 First three args are required: Eat chocolate candy
55   From dict_args: cuatro -> 4 dos -> 2 tres -> 3 uno -> 1
56 First three args are required: T o o
57   From tup_args:   f a s t
58 PrintDict(**D)
59 1 2 {'cuatro': 4, 'tres': 3}
60 PrintThings(*D)
61 First three args are required: cuatro dos tres
62   From tup_args: uno
63 $"""
```

unique.py

```
 1 #!/usr/bin/env python
 2 """(Optional) Function to generate random numbers without repetition.
 3 Demonstrates generators and the 'yield' keyword."""
 4
 5 import random
 6
 7 def Unique(bot, over_top):
 8     """Generator to deliver randomly chosen values from bot
 9     to over_top - 1, delivering each value once only."""
10     answers = range(bot, over_top)
11     random.shuffle(answers)
12     for each in answers:
13         yield each
14
15 def ListUnique(bot, over_top):
16     """Returns a list of the generated numbers"""
17     gen = Unique(bot, over_top)
18     while True:
19         try:
20             print gen.next(),
21         except StopIteration:
22             return
23
24 if __name__ == '__main__':
25     print '(0, 5) = ',
26     ListUnique(0, 5)
27     print
28     print '(10, 21) = ',
29     ListUnique(10, 21)
30     print
31
32 """
33 $ unique.py
34 (0, 5) =  1 2 3 0 4
35 (10, 21) =  14 15 20 11 19 10 18 13 17 12 16
36 $
37 """
```

decorator.py

```
 1 #!/usr/bin/env python
 2 """Optional: A decorator is a function for wrapping another function,
 3 or many other functions.  Here we are timestamping the function
 4 calls."""
 5
 6 import time
 7
 8 def TimeDecorator(func):
 9     """Decorator function for reporting when the function was called."""
10     def WrappedFunction(*args, **kw_args):
11         print "It's %s, time for %s:" % (time.ctime(), func.__name__)
12         return func(*args, **kw_args)
13     return WrappedFunction
14
15 @TimeDecorator  # syntax available in 2.5
16 def Breakfast(meat='bacon', eggs='scrambled'):
17     print "Here's your %s and %s eggs.  Enjoy!" % (meat, eggs)
18
19 def Lunch(**substitutions):
20     menu = {'meat':'ham', 'cheese':'american', 'bread':'white'}
21     menu.update(substitutions)
22     print "Here's your %(meat)s and %(cheese)s on %(bread)s bread.  Enjoy!"\
23           % menu
24 Lunch = TimeDecorator(Lunch)  # older syntax
25
26 @TimeDecorator
27 def Tea():
28     print "Tea time!"
29
30 @TimeDecorator
31 def Dinner(menu):
32     print "%s for dinner tonight." % (menu.title())
33
34 def main():
35     Breakfast(meat='sausage', eggs='basted')
36     time.sleep(1)
37     Lunch(cheese='swiss', bread='rye')
38     time.sleep(1)
39     Tea()
40     time.sleep(1)
41     Dinner('roast beef')
42
43 if __name__ == '__main__':
44     main()
```

```
45 """
46 $ decorator.py
47 It's Thu Mar  1 11:45:28 2007, time for Breakfast:
48 Here's your sausage and basted eggs.  Enjoy!
49 It's Thu Mar  1 11:45:29 2007, time for Lunch:
50 Here's your ham and swiss on rye bread.  Enjoy!
51 It's Thu Mar  1 11:45:30 2007, time for Tea:
52 Tea time!
53 It's Thu Mar  1 11:45:31 2007, time for Dinner:
54 Roast Beef for dinner tonight.
55 $ """
```

timeout_decorator_0.py

```
#! /usr/bin/env python
'''(Optional and esoteric, but useful).Decorator to time out
after one second.  Simplified from the example by Chris Wright
from the online the ASPN Cookbook.  I found it by Googling.'''

import signal, time

TIME_OUT = 1
def TimeOut(Func):

    def FunctionWrapper(*args, **kwargs):
        def AlarmHandler(signum, frame):
            all_args = ', '.join([str(a) for a in args] \
                                 + ["%s=%s" % (k,v) \
                                    for (k, v) in kwargs.items()])
            print "%s(%s) timed out at %d seconds." \
                  % (Func.__name__, all_args, TIME_OUT)
            raise RuntimeError

        old = signal.signal(signal.SIGALRM, AlarmHandler)
        signal.alarm(TIME_OUT)
        try:
            result = Func(*args, **kwargs)
        finally:
            signal.signal(signal.SIGALRM, old)
            signal.alarm(0)
        return result
    return FunctionWrapper

@TimeOut
def main():
    try:
        time.sleep(2)
    except RuntimeError:
        pass

if __name__ == '__main__':
    main()

"""
$ timeout_decorator_0.py
main() timed out at 1 seconds.
"""

```

timeout_decorator.py

```
#! /usr/bin/env python
"""Sooo optional, advanced and esteric!
A TimeOut decorator with a variable argument in the
sugar requires nested decorators! """

import signal, time

def TimeOut(timeout):
    """@TimeOut(3) before any function will cause it to
    time out in 3 seconds.  Then, when you call it, there
    will be a "RuntimeError" if it times out.
    """
    def DecoratorWrapper(Func):
        def WrappedFunction(*args, **kwargs):
            def AlarmHandler(signum, frame):
                all_args = \
                         ', '.join([str(a) for a in args] \
                                   + ["%s=%s" % (k,v) \
                                      for (k, v) in kwargs.items()])
                report =  "%s(%s) timed out at %d seconds." \
                      % (Func.__name__, all_args, timeout)
                raise RuntimeError, report
            old = signal.signal(signal.SIGALRM, AlarmHandler)
            signal.alarm(timeout)
            try:
                result = Func(*args, **kwargs)
            finally:
                signal.signal(signal.SIGALRM, old)
                signal.alarm(0)
            return result
        return WrappedFunction
    return DecoratorWrapper

@TimeOut(2)                          # Output
def main():                          #
    try:                             # $ timeout_decorator.py
        time.sleep(3)                # main() timed out at 2 seconds.
    except RuntimeError, msg:        # $
        print msg

if __name__ == '__main__':
    main()
```

Lab 13

1. Write a `Printf()` function for Python that behaves like `printf` in C. The function receives a format string for the first argument (just like Python's print statement) and then it receives any number of additional arguments as, one for each `%d` or other `%-conversion-character` sequence appearing in the format string. A call to your function might be:

   ```
   Printf("%s slid %d times.", "John", 3)
   ```

   or

   ```
   Printf("%s and %s ate %d %s.", "Lynn", "Mary", 22, "grapes")
   ```

   Hint, the implementation is *very* easy. This is just an exercise in setting up a function with a variable number of arguments.

2. Collect values from the user for the following keywords and make a dictionary of the results:

   - verb
   - noun
   - number
   - past_tense_verb
   - plural_noun

   Use your dictionary to print out the following MadLib:

   > After trying to <verb> around the <noun> <number> times, we finally <past_tense_verb> the <plural_noun> instead.

   That's good practice for string replacement using a dictionary, but it's not a good solution. Note that, for this particular madlib, the same part of speech was never used more than once, i.e., we didn't need two verbs.

   If you have time and interest, just for the string practice, make a function that takes in any random madlib, maybe:

   > All <plural_animal>, <plural_animal>, and <plural_animal> <past_tense_verb> until <number> were <past_tense_verb>.

   and returns the madlib with the responses filled in.

3. (Optional – Generators)

   Collect and extract `labs.zip` from WebCT. Find `labs/lab_08_comprehensions/lab08_2.py` so that you can use the `Cards()` function to make a generator-based function to deal card games:

`DealGame()` – deals 4 hands of 5 cards each, the default.

`DealGame(6, 3)` – deals 6 hands (for 6 players) of 3 cards each

If you like, make the program interactive so that it prompts for the number of hands and cards to deal.

4. (Optional – Decorators)The random module provides a `sample` function. The help facility starts:

```
sample(self, population, k) method of random.Random instance
    Chooses k unique random elements from a population sequence.
```

How would you call this to deliver a list of 6 numbers between 1 and 52, for a lotto pick?

Make a `Lotto` function that returns the result of this call.

Make a decorator that logs the time and the output of the function it decorates. Decorate your Lotto function so that each call is logged, complete with the numbers generated. My log entries look like:

```
Wed Mar 28 12:39:58 2007  -> 41, 26, 7, 16, 21, 5
```