PYTHON LAB BOOK

Python For Programmers

UCSC Extension Online

Lab 14 OOP

Topics

- Module: `shelve`
- Classes
- Inheritance
- Class variable

v.4.0

lab13_1.py

```
#!/usr/bin/env python
"""lab13_1.py -- a Printf function"""
import sys

def Printf(format, *args):
    """Emulates C-like printf function."""
    sys.stdout.write(format % args)

def Printfx(format, *args):
    """This one almost always works."""
    print format % args,

def main(function):
    print 'Testing', function
    function('%d black cats drank %d plates of milk.\n', 4, 2)
    function('%d', 3)
    function('%d\n', 3)
    function('Hello World\n')

if __name__ == '__main__':
    main(Printf)
    main(Printfx)

"""
$ lab13_1.py
Testing <function Printf at 0xb7f23f44>
4 black cats drank 2 plates of milk.
33
Hello World
Testing <function Printfx at 0xb7f23f7c>
4 black cats drank 2 plates of milk.
3 3
Hello World
$
"""
```

lab13_2.py

```
 1 #!/usr/bin/env python
 2 """
 3 lab13_2.py
 4
 5 MadLibs from dictionary.
 6 """
 7
 8 MADLIB =  """After trying to %(verb)s around the %(noun)s %(number)s times,
 9 we finally %(past_tense_verb)s the %(plural_noun)s instead."""
10
11 WORDS_NEEDED = "verb", "noun", "number", "past_tense_verb", "plural_noun"
12
13 def CollectUniqueResponses(words_needed):
14     """This solution is good if parsed out what you needed by hand, and
15     there are no duplicate parts of speech in the madlib.
16     """
17     replacers_d = {}
18     for part_speech in words_needed:
19         answer = raw_input("Give me a %s: " % part_speech)
20         if not answer:
21             return None
22         replacers_d[part_speech] = answer
23     return replacers_d
24
25 def Madlibx(madlib = MADLIB, words_needed = WORDS_NEEDED):
26     replacers_d = CollectUniqueResponses(words_needed)
27     return madlib % replacers_d
28
29 def Madlib(madlib):
30     """This is a more flexible solution."""
31
32     replacers = {}
33     all_parts = madlib.split('%')
34     new_strings = [all_parts[0]]
35     for replacer in madlib.split('%')[1:]:
36         part_speech, rest = replacer[1:].split(')')
37         answer = raw_input("Give me a %s: " % part_speech)
38         if not answer:
39             return None
40         while part_speech in replacers:
41             part_speech += '_'
42         replacers[part_speech] = answer
43         new_strings += [')'.join((part_speech, rest))]
44     return '%('.join(new_strings) % replacers
```

```
45
46 def main():
47     madlib_str = Madlibx()
48     if madlib_str:
49         print madlib_str
50     madlib = "All %(plural_animal)s, %(plural_animals)s, and %(plural_animal)s %(past_ter
51              " until %(number)s were %(past_tense_verb)s."
52     madlib_str = Madlib(madlib)
53     if madlib_str:
54         print madlib_str
55
56 if __name__ == '__main__':
57     main()
58
59 """
60 $ lab13_2.py
61 Give me a verb: drip
62 Give me a noun: ball
63 Give me a number: 3
64 Give me a past_tense_verb: hopped
65 Give me a plural_noun: toes
66 After trying to drip around the ball 3 times,
67 we finally hopped the toes instead.
68 Give me a plural_animal: sloths
69 Give me a plural_animals: ants
70 Give me a plural_animal: jellyfish
71 Give me a past_tense_verb: tied
72 Give me a number: 18
73 Give me a past_tense_verb: rolled
74 All sloths, ants, and jellyfish tied until 18 were rolled.
75 $"""
```

lab13_3.py

```
 1 #!/usr/bin/env python
 2 """lab13_3.py
 3
 4 A generator-based program to deal card games.
 5
 6 """
 7 import sys
 8 import os
 9 if __name__ == '__main__':
10     sys.path.insert(0, "..")
11 else:
12     sys.path.insert(0, os.path.join(os.path.split(__file__)[0], '..'))
13 import lab_08_Comprehensions.lab08_2 as cards
14 import random
15
16 __pychecker__ = "no-local"    # Ask me!
17
18 def DealCard():
19     """Generator to yield one card at a time from a deck."""
20     deck = cards.Cards()
21     random.shuffle(deck)
22     for card in deck:
23         yield card
24
25 def DealHand(no_cards):
26     """Generator to yield a hand of no_cards cards."""
27     card_generator = DealCard()
28     while True:
29         the_hand = []
30         for i in range(no_cards):
31             try:
32                 the_hand += [card_generator.next()]
33             except StopIteration:
34                 the_hand += ['None']
35         yield the_hand
36
37 def DealGame(no_players=4, no_cards=5):
38     """Delivers a list of card hands, one for each player."""
39     hand_generator = DealHand(int(no_cards))
40     the_hands = []
41     for i in range(int(no_players)):
42         the_hands += [hand_generator.next()]
43     return the_hands
44
```

```
45 def PrintGame(game_list):
46     for player in game_list:
47         print ', '.join([card for card in player])
48
49 def main():
50     print "DealGame():"
51     PrintGame(DealGame())
52     print "DealGame(6, 3):"
53     PrintGame(DealGame(6, 3))
54     print "DealGame(11):"
55     PrintGame(DealGame(11))
56
57 if __name__ == '__main__':
58     main()
59 """
60 $ lab13_3.py
61 DealGame():
62 Ace of Spades, 5 of Diamonds, 2 of Hearts, 8 of Clubs, 6 of Clubs
63 Joker, 9 of Spades, Joker, Jack of Diamonds, 3 of Spades
64 6 of Diamonds, Queen of Spades, 6 of Spades, Ace of Diamonds, 5 of Spades
65 7 of Diamonds, 9 of Clubs, 7 of Spades, 8 of Spades, 4 of Spades
66 DealGame(6, 3):
67 Jack of Clubs, 5 of Spades, 6 of Hearts
68 6 of Clubs, Queen of Clubs, Joker
69 Joker, 2 of Clubs, Queen of Spades
70 7 of Diamonds, 8 of Hearts, Queen of Hearts
71 Jack of Hearts, 9 of Diamonds, 5 of Hearts
72 2 of Hearts, 8 of Diamonds, 3 of Diamonds
73 DealGame(11):
74 9 of Diamonds, Queen of Diamonds, 8 of Diamonds, 6 of Spades, Jack of Diamonds
75 King of Diamonds, 3 of Spades, Joker, 10 of Diamonds, 10 of Spades
76 3 of Diamonds, Ace of Spades, 8 of Spades, 5 of Clubs, Ace of Clubs
77 4 of Clubs, 3 of Hearts, 9 of Hearts, King of Spades, 6 of Hearts
78 3 of Clubs, 4 of Hearts, Jack of Clubs, Queen of Clubs, 9 of Spades
79 King of Clubs, Ace of Diamonds, 6 of Clubs, 4 of Spades, 5 of Diamonds
80 Queen of Hearts, 5 of Hearts, King of Hearts, 2 of Clubs, 7 of Clubs
81 Queen of Spades, 2 of Spades, Jack of Hearts, 8 of Hearts, 2 of Diamonds
82 2 of Hearts, 6 of Diamonds, 7 of Hearts, 7 of Spades, 9 of Clubs
83 10 of Clubs, 10 of Hearts, Ace of Hearts, Joker, 7 of Diamonds
84 8 of Clubs, 5 of Spades, Jack of Spades, 4 of Diamonds, None
85 $ """
```

lab13_4.py

```
 1 #!/usr/bin/env python
 2 """lab13_4.py A logging lotto facility."""
 3 import time
 4 import random
 5
 6 LOG_FILE = 'lotto.log'
 7
 8 def LogIt(func):
 9     """Decorator function for logging output from the func."""
10     def LoggedFunction(*t_args, **kw_args):
11         f = open(LOG_FILE, "a")
12         got = func(*t_args, **kw_args)
13         f.write("%s  ->%s\n" % (time.ctime(), got))
14         f.close()
15         return got
16     return LoggedFunction
17
18 @LogIt
19 def Lotto():
20     return ', '.join([str(x) for x in random.sample(xrange(1, 53), 6)])
21
22 def main():
23     print Lotto()
24     print Lotto()
25
26 if __name__ == '__main__':
27     main()
28
29 """$ lab13_4.py
30 41, 26, 7, 16, 21, 5
31 2, 36, 49, 51, 16, 11
32 $ cat lotto.log
33 Wed Mar 28 12:39:58 2007  -> 41, 26, 7, 16, 21, 5
34 Wed Mar 28 12:39:58 2007  -> 2, 36, 49, 51, 16, 11
35 $"""
```

shelve_dictionary.py

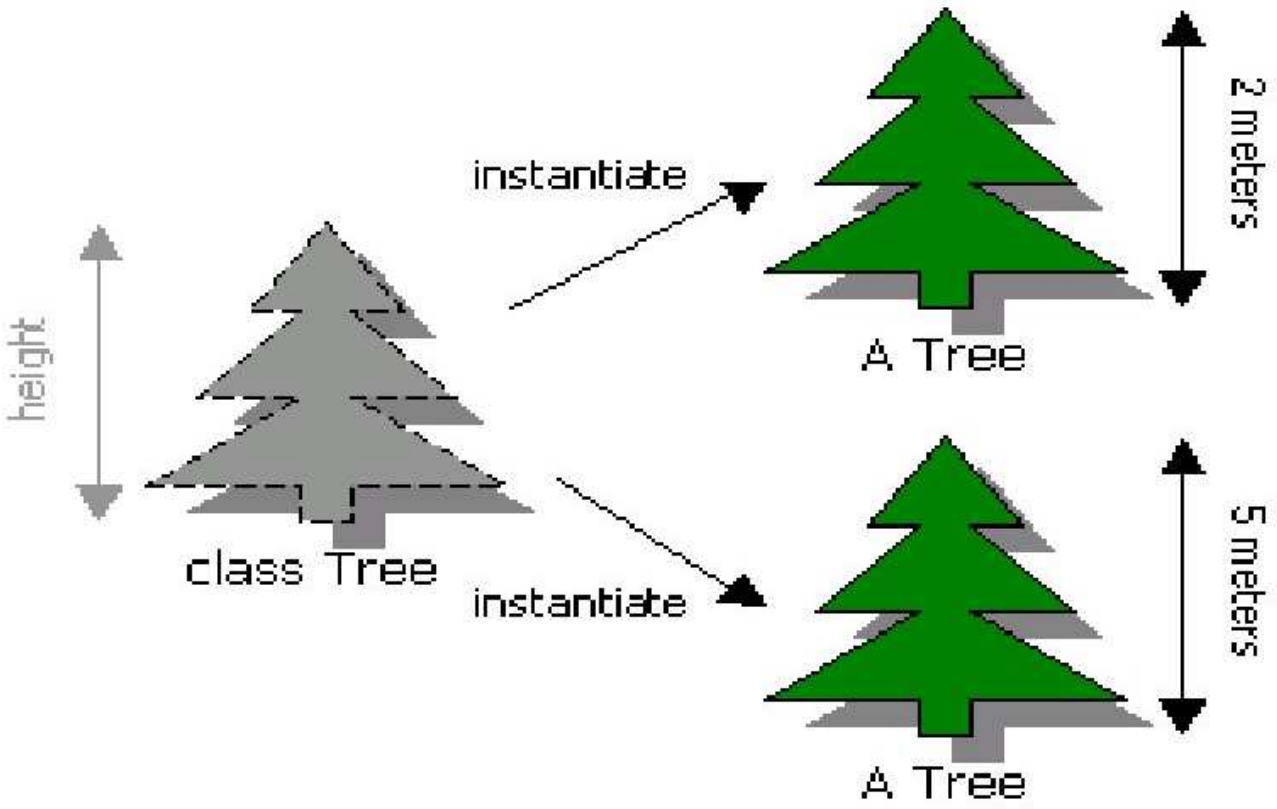
```
#!/usr/bin/env python
"""shelve_dictionary.py
Here again is our dictionary of Python keywords.

Now, we are making the dictionary persistent by 'shelving' it.
"""
import shelve

# a surgical import
import sys
if __name__ == '__main__':
    sys.path.insert(0, "..")
else:
    sys.path.insert(0, os.path.join(os.path.split(__file__)[0], '..'))
from lab_09_Dictionaries.py_dict \
     import CollectEntries, FindDefinitions, MakePrompt, PrintEntries

def main():
    """Runs the user interface for dictionary manipulation."""
    global py_dict
    choices = {'add': CollectEntries, 'find': FindDefinitions,
               'print': PrintEntries}
    prompt = MakePrompt(choices)

    try:
        py_dict = shelve.open("py_dict.dat")
    except IOError, msg:
        print 'File could not be opened.'
        return

    while True:
        raw_choice = raw_input(prompt)
        if raw_choice == '':
            break
        given_choice = raw_choice[0].lower()
        for maybe_choice in choices:
            if maybe_choice[0] == given_choice:
                choices[maybe_choice]()
                break
        else:
            print '%s is not an acceptible choice.' % raw_choice

    py_dict.close()

```

```
45 if __name__ == '__main__':
46     main()
47 """
48 $ shelve_dictionary.py
49 Choose (a)dd, (d)efinitions, (f)ind, (p)rint (enter to quit) a
50 Word: key
51 Meaning: an object used to access a value in a dictionary
52 Word: break
53 Meaning: break out of a loop and skip the else
54 Word:
55 Choose (a)dd, (d)efinitions, (f)ind, (p)rint (enter to quit) p
56 break : break out of a loop and skip the else
57 key : an object used to access a value in a dictionary
58 Choose (a)dd, (d)efinitions, (f)ind, (p)rint (enter to quit)
59 $ shelve_dictionary.py
60 Choose (a)dd, (d)efinitions, (f)ind, (p)rint (enter to quit) p
61 break : break out of a loop and skip the else
62 key : an object used to access a value in a dictionary
63 Choose (a)dd, (d)efinitions, (f)ind, (p)rint (enter to quit)
64 $
65 """
```

Pythonic Thinking About namespaces: Python supports namespace nesting:

```
menu.lunch.salad = 'caesar'
```

`object.attribute.`attribute.attribute ...



Object Oriented Programming

A **class** is:

a blueprint for a namespace.

An **object** is:

a namespace constructed from the blueprint.

greeter1_def.py

```
 1 #!/usr/bin/env python
 2 """Here is a very simple object-oriented python program.
 3 Note that we make 3 namespaces: the Greeter class, and two
 4 instances of the Greeter class, fred and alma."""
 5
 6 class Greeter:
 7     """The Greeter class makes Greeter objects that can greet you."""
 8
 9     def Greet(self):           # 'self' is always the first argument
10         print 'Hello World'    # of every method in every class.
11
12 def main():                   # A call:  fred.Greet() is interpreted
13     fred = Greeter()           # as Greeter.Greet(fred). So fred is
14     print 'fred.Greet():'      # the self in this call, and fred is
15     fred.Greet()               # the namespace we are referring to.
16     alma = Greeter()
17     print 'alma.Greet():'
18     alma.Greet()
19     print 'Fred is', id(fred), 'Alma is', id(alma), \
20           'Greeter class is', id(Greeter)
21
22 if __name__ == '__main__':
23     main()
24
25 """
26 $ greeter1_def.py
27 fred.Greet():
28 Hello World
29 alma.Greet():
30 Hello World
31 Fred is 135591268 Alma is 135590980 Greeter class is 135413484
32 $
33 """
```

greeter2_def.py

```
 1 #!/usr/bin/env python
 2 """Here we add a name attribute to our class and a method
 3 to set a value into the name.
 4
 5 The value of an attribute belongs solely to the object;
 6 the methods belong both to the class and the objects."""
 7
 8 class Greeter:
 9     """The Greeter class makes potentially named Greeter
10     objects that can greet you."""
11
12     def SetName(self, name_in):
13         self.name = name_in
14
15     def Greet(self):
16         try:
17             print "Hello World. I'm %s" % self.name
18         except AttributeError:
19             print "Hello World."
20
21 def main():
22     gracy = Greeter()
23     gracy.Greet()
24     gracy.SetName('Gracy')
25     gracy.Greet()
26     george = Greeter()
27     george.SetName('George')
28     george.Greet()
29     gracy.Greet()
30
31 if __name__ == '__main__':
32     main()
33 """
34 $ greeter2_def.py
35 Hello World.
36 Hello World. I'm Gracy
37 Hello World. I'm George
38 Hello World. I'm Gracy
39 $
40 """
```

greeter3_def.py

```
#!/usr/bin/env python
"""Alternatively, we define an __init__ method so that the
name is given when the object is created."""

class Greeter:
    # This Greeter class needs a name when instantiated.

    def __init__(self, name):
        # __init__ is Python's constructor method
        self.name = name

    def Greet(self):
        print "Hello World. I'm", self.name

def main():
    fred = Greeter('Fred')
    print 'fred.Greet():'
    fred.Greet()

    x = Greeter()    #  error!
    print 'x.Greet():'
    x.Greet()

if __name__ == '__main__':
    main()

"""
$ greeter3_def.py
fred.Greet():
Hello World. I'm Fred
Traceback (most recent call last):
  File "./greeter3_def.py", line 25, in ?
    main()
  File "./greeter3_def.py", line 20, in main
    x = Greeter()    #  error!
TypeError: __init__() takes exactly 2 arguments (1 given)
$
"""
```

One of the greatest benefits of OOP is the ability to add functionality and complication to a class and yet leave the class alone, keeping it simple.

It's like eating cake and still having it.

This is **inheritance**.

greeter4_def.py

```
#!/usr/bin/env python
"""Here we implement 'inheritance' to leave the original
Greeter class intact and add a NamedGreeter class, that
has all the functionality of the Greeter class plus some
new things. """

class Greeter:

    def Greet(self):
        print "Hello World"

class NamedGreeter(Greeter):
    # Inherits the methods in the Greeter class, and adds some
    # of it's own.

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

    def SayMyName(self):
        print "I'm", self.name

def main():
    fred = NamedGreeter('Fred')
    print 'fred.Greet():'
    fred.Greet()
    fred.SayMyName()

    # code that depends on the Greeter class is unaffected.
    x = Greeter()
    print 'x.Greet():'
    x.Greet()

if __name__ == '__main__':
    main()
"""
$ greeter4_def.py
fred.greet():
Hello World
I'm Fred
x.Greet():
Hello World
$ """
```

greeter5_def.py

```
#!/usr/bin/env python
"""Here, both classes have a Greet method.  The lingo is:
the NamedGreeter.Greet() method 'overrides' the
Greeter.Greet() method.  When an object of the NamedGreeter
class calls Greet(), it accesses and runs NamedGreeter.Greet()
while an object of the Greeter class executes Greeter.Greet().
"""
class Greeter:

    def Greet(self):
        print "Hello World"
    def Bye(self):
        print "Bye now."

class NamedGreeter(Greeter):

    def __init__(self, name):
        self.name = name
    def Greet(self):
        # NamedGreeter.Greet() calls the Greeter.Greet() method
        # and then adds some functionality.  This is a common and
        # useful technique.
        Greeter.Greet(self)
        print "I'm", self.name

def main():
    fred = NamedGreeter('Fred')
    print 'fred.Greet():'
    fred.Greet()                # Output:
    print 'fred.Bye():'         #
    fred.Bye()                  # $ greeter5_def.py
    x = Greeter()               # fred.Greet():
    print 'x.Greet():'          # Hello World
    x.Greet()                   # I'm Fred
    print 'x.Bye():'            # fred.Bye():
    x.Bye()                     # Bye now.
                                # x.Greet()
if __name__ == '__main__':      # Hello Word
    main()                      # x.Bye():
                                # Bye now.
                                # $
```

greeter6_def.py

```
 1 #!/usr/bin/env python
 2 """Here we implement another class, a HipGreeter, and have it
 3 further down the inheritance tree."""
 4
 5 class Greeter:
 6     def Greet(self):
 7         print "Hello World"
 8     def Bye(self):
 9         print "Bye now."
10
11 class NamedGreeter(Greeter):
12     def __init__(self, name):
13         self.name = name
14     def Greet(self):
15         Greeter.Greet(self)
16         print "I'm", self.name
17
18 class HipGreeter(NamedGreeter):
19     def Greet(self):
20         NamedGreeter.Greet(self)
21         print "Wazzup."
22
23 def main():
24     rocky = HipGreeter("Rocky")
25     rocky.Greet()
26     rocky.Bye()
27
28 if __name__ == '__main__':
29     main()
30
31 """
32 $ greeter6_def.py
33 Hello World
34 I'm Rocky
35 Wazzup.
36 Bye now.
37 $
38 """
```

greeter7_def.py

```
 1 #!/usr/bin/env python
 2 """Here we have a NamedGreeterGuru Class, and a
 3 GuruNamedGreeter Class, overriding the Bye()
 4 method in the Guru class, demonstrating left-
 5 right, depth-first method resolution.
 6 """
 7
 8 import random
 9
10 class Guru:
11     # And here is a class variable, accessable anywhere
12     # by Guru.sayings
13     sayings = ("The great man is one who never loses his\n"
14                "child's heart.                    Mencius",
15                "There is no solution, for there is no\n"
16                "problem.                Marcel Duchamp",
17                "If you could get rid of yourself just once, \n"
18                "The secret of secrets would open to you.   \n"
19                "                              Jalaluddin Rumi",
20                "Don't be consistent, but be simply true.\n"
21                "                    Oliver Wendell Holmes",
22                "Nothing is so simple that it cannot be\n"
23                "misunderstood.          Freeman Teague",
24                "What one understands is only half true.\n"
25                "What one does not understand is the full\n"
26                "truth.                              Zen Saying",
27                "Trying to define yourself is like trying\n"
28                "to bite your own teeth.       Alan Watts")
29
30     def Bye(self):
31         print "Good Bye.  And remember:"
32         self.Pontificate()
33
34     def Pontificate(self):
35         print random.choice(Guru.sayings)
36
37 class Greeter:
38
39     def Greet(self):
40         print "Hello World"
41
42     def Bye(self):
43         print "Bye now."
44
```

```
45 class NamedGreeter(Greeter):
46
47     def __init__(self, name):
48         self.name = name
49
50     def Greet(self):
51         Greeter.Greet(self)
52         print "I'm", self.name
53
54 class GuruNamedGreeter(Guru, NamedGreeter):
55     pass
56
57 class NamedGreeterGuru(NamedGreeter, Guru):
58     pass
59
60 def main():
61     rocky = GuruNamedGreeter("Rocky")
62     rocky.Greet()
63     rocky.Pontificate()
64     rocky.Bye()
65
66     moose = NamedGreeterGuru("Moose")
67     moose.Greet()
68     moose.Pontificate()
69     moose.Bye()
70     print "\nAccessing the class variable:"
71     print Guru.sayings[random.randrange(len(Guru.sayings))]
72
73 if __name__ == '__main__':
74     main()
75
76
77 """
78 $ greeter7_def.py
79 Hello World
80 I'm Rocky
81 Nothing is so simple that it cannot be
82 misunderstood.          Freeman Teague
83 Good Bye.  And remember:
84 If you could get rid of yourself just once,
85 The secret of secrets would open to you.
86                              Jalaluddin Rumi
87 Hello World
88 I'm Moose
89 Nothing is so simple that it cannot be
```

```
misunderstood.          Freeman Teague
Bye now.
Accessing the class variable:
If you could get rid of yourself just once,
The secret of secrets would open to you.
                            Jalaluddin Rumi
$ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      Notes on discovering all about an object:


      an_object == another_object  --> True

                                       They are the same builtin type
                                       and have the same values for
                                       all nested objects.

      an_object is another_object  --> True
                |
                |                      id(an_object) == id(another_object)
        'is' is a keyword!             They occupy the same spot in memory.
                                       id() is a builtin that is rarely
                                       used.

      isinstance(an_object, class-or-type-or-tuple)  --> True
        If class-or-type-or-tuple is:

          class  --  if an_object is of the class or any subclass of it
          type   --  if an_object is the type
          tuple  --  if an_object isinstance of any of the elements in the
                     tuple.  The tuple elements can be classes and/or types.

      issubclass(C, B)             --> True
                                       C is a subclass of any of the classes in
                                       the tuple.  B can be a tuple of classes.

"""
```

Answers for Quiz 4 (Lab 14)

1. Predict the output:

```
>>> xlist = [1, 2, ['a', 'b']]
>>> xlist_ref = xlist
>>> xlist_copy = xlist[:]
>>> import copy
>>> xlist_deepcopy = copy.deepcopy(xlist)
>>> xlist[0] = '***'
>>> xlist[2][0] = '***'
>>> print xlist
['***', 2, ['***', 'b']]
>>> print xlist_ref
['***', 2, ['***', 'b']]
>>> print xlist_copy
[1, 2, ['***', 'b']]
>>> print xlist_deepcopy
[1, 2, ['a', 'b']]
>>>
```

2. More predictions:

```
>>> xdict = {'a':[1,2,3], 'b':[4,5,6]}
>>> xdict_ref = xdict
>>> xdict_copy = xdict.copy()
>>> import copy
>>> xdict_deepcopy = copy.deepcopy(xdict)
>>> xdict['a'] = 'ok'
>>> xdict['b'][0] =  888
>>> xdict['c'] = 'watermelon'
>>> print xdict
{'a': 'ok', 'c': 'watermelon', 'b': [888, 5, 6]}
>>> print xdict_ref
{'a': 'ok', 'c': 'watermelon', 'b': [888, 5, 6]}
>>> print xdict_copy
{'a': [1, 2, 3], 'b': [888, 5, 6]}
>>> print xdict_deepcopy
{'a': [1, 2, 3], 'b': [4, 5, 6]}
>>>
```

```
Functional Programming

What are the results?

>>> [str(x) for x in range(3)]

['0', '1', '2']

>>> [[0] for x in range(3)]

[[0], [0], [0]]

>>> map(lambda x:x**2, range(5))

[0, 1, 4, 9, 16]

>>> [x for x in range(10) if not x % 2]

[0, 2, 4, 6, 8]
```

Lab 14

1. If you have a file named lab14_1.py:

```
class X:
    def __init__(self):
        self.x = 1
    def Which(self):
        print "X"

class A(X):
    def __init__(self):
        X.__init__(self)
        self.y = 2

class Y:
    def __init__(self):
        self.z = 3
    def Which(self):
        print "Y"

class B(Y):
    def __init__(self):
        Y.__init__(self)
        self.x = 4

class AB(A, B):
    pass

class BA(B, A):
    pass

ab = AB()
ba = BA()
```


Guess the results if you were to run `python -i lab14_1.py` and then ask the interpreter to evaluate each of the following:

a _____ ab.x

b _____ ab.y

c _____ ab.z

d _____ ab.Which()

e _____ ba.x

f _____ ba.y

g _____ ba.z

h _____ ba.Which()

Please don't take the time to type in the code. It's fine to get the wrong answer. Just make your best quick guess. We'll discuss it.

2. Implement a Stack class. It should have two methods: one adds things to the top of your stack (usually called push); and the other takes things from the top of your stack, (usually called `pop`).

   This is just wrapping some class definition syntax around a list.

3. (Optional) Implement this inheritance tree:

```
Employee
   name

SalariedEmployee   --> Inherits from Employee
   Has a yearly salary.

ContractEmployee   --> Inherits from Employee
   Has an hourly rate.
```

   This code:

```
joe = SalariedEmployee('Joe')
joe.SetSalary(52000)  # Joe is salaried so this is his yearly salary
joe.PrintName()
print "here's $%.2f for you. " % joe.CalculatePay(1) # 1 week
joe.GiveRaise(2)     # A 2% raise!
joe.PrintName()
print "here's $%.2f for you. " % joe.CalculatePay(2) # 2 weeks

susan = ContractEmployee('Susan')
susan.PrintName()
susan.SetRate(100) # Susan is contract so this is her hourly pay
print "here's $%.2f for you. " % susan.CalculatePay(80)
susan.GiveRaise(2)   # A 2% raise!
susan.PrintName()
print "here's $%.2f for you. " % susan.CalculatePay(80) # 80 hours
```


   Should produce this output:

```
Joe here's $1000.00 for you.
Joe here's $2040.00 for you.
Susan here's $8000.00 for you.
Susan here's $8160.00 for you.
```