==========

CSCI 5300: Software Design
Spring 2015: Phil Pfeiffer
Assignment 4: Python – class constructs
Terms: As individuals
Due date: Sunday night, 8 March, close of business
Value: 8.25 points –points as marked

==========

Background. This assignment continues assignment 1, the Python overview assignment, focusing on Python classes.

Requirements. This assignment assumes the use of a platform that supports

- a Python v3.4 interpreter;
- a text editor, like vi, notepad++, or PFE32, with line numbers;
- a CLI environment that has been configured for use with Python--i.e., by updating the command line environment's path variable to reference the Python directory by
  - using the Python installer to add support for Python 3.4 to the host system's PATH variable (available as an installation option; disabled by default)
  - using a command like
    set PATH=%PATH%;C:\Python33
    (DOSshell; assumes that Python is installed in C:\Python33)
- GUI environment that supports
  - the ability to do screen captures of individual windows—e.g., Windows with the Windows snipping tool; and
  - the ability to store captures in standard formats—e.g., as .gifs or .pngs or in Word for Windows files

Overview. The assignment consists of short exercises that illustrate the operation and use of Python 3's key features. My starting point for these exercises is sections 7-9 of an overview document, *a tour of python.txt*, that I created for a Python special topics course. The tutorial was compiled from a variety of sources at python.org, including

- the Python tutorial (http://docs.python.org/py3k/tutorial/index.html)
- the Python language reference (http://docs.python.org/py3k/reference/index.html)
- the Python library reference (http://docs.python.org/py3k/library/index.html)
- the Python Setup and Usage reference (http://docs.python.org/py3k/using/index.html)
- the Python enhancement proposals (PEP's) (http://www.python.org/dev/peps/)

Once you've downloaded and installed Python, you can access these materials locally, in the Python installation directory, in the (lone) .chm file in Python34/Doc (file's name dependent on which Python implementation you've downloaded)

For this assignment, I'll ask you to

- Try variations on selected exercises from the tutorial's various sections: i.e., "customize" the various examples so as to make it clear that you're doing and evaluating them
- Send me screen shots of your work.

Terms. I'll allow people to consult with one another as they do this assignment. Because the assignment is meant to teach a language that we'll be using in this class, I'll ask people submit their own work, without copying it from others.

For this assignment, I will also require ***on-time*** work: expect no credit for work submitted after close of business.

Throughout these exercises, I ask for screen shots as .pngs, since this is how Microsoft's Windows Snipping Tool formats screen images. To capture a screen shot in the Windows environment, do the following:

- select (i.e., set focus to) the window you wish to capture.
- enter "ALT-Print Screen" to saves the window's image to the clipboard.
- use a utility, like Paint, to
- import the screen shot (Ctrl-V) to the utility's work area, and
- save the image as a .PNG or a .GIF

I will, however, accept screen shots in any widely used image format, including .gif, .bmp, and .jpg. If you use a lossy format like .jpg, make sure that your images are readable. Do choose just one format and stick with it. Also, please keep all window sizes small, so as to make the images a little easier to manage.

The assignment: Starting with *a tour of python.txt*, work the indicated examples. Observe the following conventions:

- When I ask you to answer a question, do so in a way that a junior computing major can understand
- When I ask you to "personalize" an example or ask you to use data of "your choice", modify the code in some way so that the output differs slightly, but not materially, from what's requested in the tutorial.
- Deliverables for each problem are indicated after the problem. Each deliverable will either be
  - a screen shot (indicated by the phrase ***screen shot*** after the problem) or
  - a written response

  For all screen shots, make all of what you entered visible!
- For some of these questions, Google may prove useful. If you do Google an answer, please cite the website in your response.
- Submit your work as a Word file or a .zip of individual files, as you will. If you choose to submit your work as individual files, be sure to title each file so that the problems you've worked are clear from the title: e.g., 2.1.png would be a reasonable title for the assignment's first problem.

The problems:

1) (0.25 points) Use the code for lines 2402-2445 to answer the following questions:

a) What attributes are in class Trivial that aren't in class object?

Answer: __weakref__, __module__ **and** __dict__ .

b) Define the two instances of Trivial as shown in the notes. What's the result of comparing the instances proper? Of comparing their IDs?

Answer: Both results to False.

```
>>> class Trival: pass
>>> id(Trival)
140576303786840
>>> inst1 = Trival()
>>> inst2 = Trival()
>>> inst1 == inst2
False
>>> id(inst1) == id(inst2)
    False
```

c) What does what you learned from b) tell you about built-in Python notions of equality?

Answer: Each objects create their own unique ids (hash). The equality can be obtained by overriding __eq__ method.

```
class Trival:
  def __init__(self, id):
    self.id = id
  def __eq__(self, other):
    return self.id == other.id

inst1 = Trival(1)
inst2 = Trival(1)
print(inst1 == inst2)
True
```

2) (0.25 points) Use the definitions of myclass_instance_1, myclass_instance_2, and mysubclass_instance set out in 2519-2557 to answer the following questions:

a) Which of the three assignments in lines 2567-9, if any, affect a common variable?

Answer: None, because each instance gets its own copy of instance_value.

b) Which of the three assignments in lines 2585-7, if any, affect a common variable?

Answer: All three. However, the value set would be the one that was executed last. A static variable is common to all instance of the class (even inherited classes), therefore changes made from any one of the methods effect the value for all the members that access the variable.

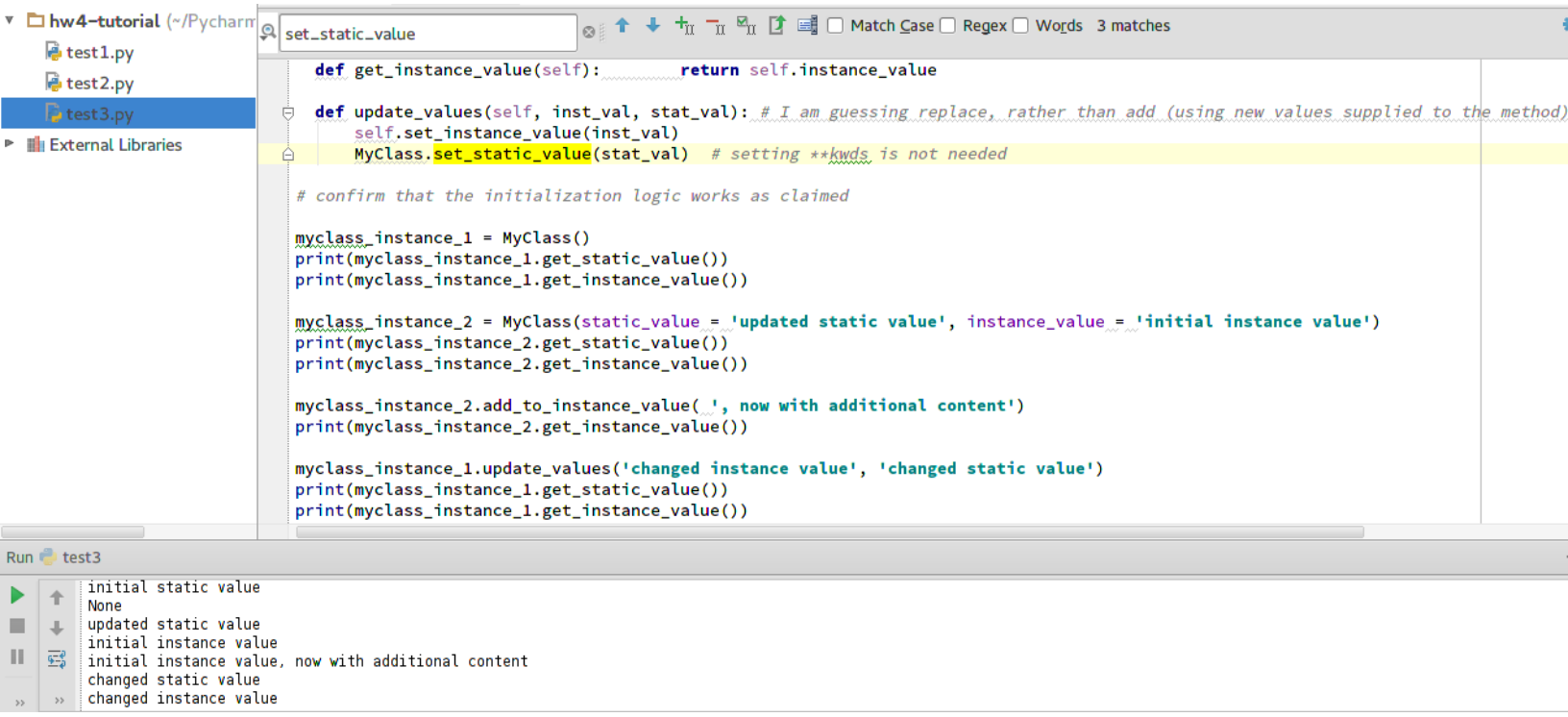
c) Which of the three assignments in lines 2601-3, if any, affect a common variable?

Answer: First two. Class attribute is common to a class. However if the derived class changes the value, it becomes common to all instances of the derived class. If the derived class does not modify the value of the base class's attribute it can access value set to the base class's attribute.

3) (0.25 points) Starting with the definition of MyClass as given in lines 2711-32,

a) Extend the definition with a method, update_values, that updates instance_value as well as static_value, using new values supplied to the method. Show your updated class, along with logic that illustrates your method's operation. (***screen shot***)

Answer:



b) Should this method be defined as an instance method or as a static method, or can it be defined as either? Explain your answer.

Answer: This method should be an instance method. Instead of calling the instance method set_instance_value to set instance_value the "self.instance_value = inst_val" could have been used to directly set the value to the variable. In any case to set the variable instance of the class MyClass is required. Static members (attributes/methods) can be accessed from inside instance members, but a static member operates on a single variable across all the instance of the classes, therefore the update could not have been achieved using static method.

4) (0.25 points) Starting the definition of MyClass as given in lines 2783-99, show the effect of executing lines 2803-2805 (***screen shot***)

Answer:

```
class MyClass:
    def __init__(self, iv_1, iv_2):
        self.set_instance_value_1( iv_1 )              # - - - a virtual method call - - -
        MyClass.set_instance_value_2( self, iv_2 )     # - - - a non-virtual method call - - -
    #
    def set_instance_value_1(self, v):     self.instance_value_1 = 'set from MyClass: ' + v
    def get_instance_value_1(self):        return self.instance_value_1
    #
    def set_instance_value_2(self, v):     self.instance_value_2 = 'set from MyClass: ' + v
    def get_instance_value_2(self):        return self.instance_value_2

class MySubclass(MyClass):
    def set_instance_value_1(self, v):     self.instance_value_1 = 'set from MySubclass: ' + v
    def get_instance_value_1(self):        return self.instance_value_1
    #
    def set_instance_value_2(self, v):     self.instance_value_2 = 'set from MySubclass: ' + v
    def get_instance_value_2(self):        return self.instance_value_2

# confirm that the initialization logic works as claimed

mysubclass_instance = MySubclass( 'first', 'second' )
print(mysubclass_instance.get_instance_value_1())
print(mysubclass_instance.get_instance_value_2())
```

```
Run  virtual_nonvirtual
set from MySubclass: first
set from MyClass: second
```

5) (0.25 points) Starting again with the definition of MyClass as given in lines 2783-99, first confirm that replacing the second line of MyClass.__init__, which currently reads

MyClass.set_instance_value_2(self, iv_2)

with

super(type(self), self).set_instance_value_2(iv_2)

leaves the output of lines 2803-2805 unchanged. Then, construct an example where the effects of this change to MyClass.__init__ would indeed be observable.

Answer: The difference is noted, when we try to create an object of MyClass, because the base class of MyClass i.e., object does not have set_instance_value_2 method.

```
class MyClass:
  def __init__(self, iv_1, iv_2):
    self.set_instance_value_1( iv_1 )              # - - - a virtual method call -
    MyClass.set_instance_value_2( self, iv_2 )     # - - - a non-virtual method ca
    #super(type(self), self).set_instance_value_2( iv_2)
  #
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MyClass:
  def get_instance_value_1(self):       return self.instance_value_1
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MyClass:
  def get_instance_value_2(self):       return self.instance_value_2

class MySubclass(MyClass):
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MySubcla
  def get_instance_value_1(self):       return self.instance_value_1
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MySubcla
  def get_instance_value_2(self):       return self.instance_value_2

# confirm that the initialization logic works as claimed

```

```
>>> ================================ RESTART ================================
>>>
>>> mysubclass_instance = MySubclass( 'first', 'second' )
>>> myclass_instance = MyClass('first', 'second')
>>> mysubclass_instance.get_instance_value_1()
'set from MySubclass: first'
>>> mysubclass_instance.get_instance_value_2()
'set from MyClass: second'
>>> myclass_instance.get_instance_value_1()
'set from MyClass: first'
>>> myclass_instance.get_instance_value_2()
'set from MyClass: second'
>>>
```

```
class MyClass:
  def __init__(self, iv_1, iv_2):
    self.set_instance_value_1( iv_1 )              # - - - a virtual method call -
    #MyClass.set_instance_value_2( self, iv_2 )     # - - - a non-virtual method c
    super(type(self), self).set_instance_value_2( iv_2)
  #
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MyClass:
  def get_instance_value_1(self):       return self.instance_value_1
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MyClass:
  def get_instance_value_2(self):       return self.instance_value_2

class MySubclass(MyClass):
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MySubcla
  def get_instance_value_1(self):       return self.instance_value_1
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MySubcla
  def get_instance_value_2(self):       return self.instance_value_2

# confirm that the initialization logic works as claimed

```

```
>>> ================================ RESTART ================================
>>>
>>> mysubclass_instance = MySubclass( 'first', 'second' )
>>> myclass_instance = MyClass('first', 'second')
>>> mysubclass_instance.get_instance_value_1()
'set from MySubclass: first'
>>> mysubclass_instance.get_instance_value_2()
'set from MyClass: second'
>>> myclass_instance.get_instance_value_1()
'set from MyClass: first'
>>> myclass_instance.get_instance_value_2()
'set from MyClass: second'
>>> ================================ RESTART ================================
>>>
>>> mysubclass_instance = MySubclass( 'first', 'second' )
>>> mysubclass_instance.get_instance_value_1()
'set from MySubclass: first'
>>> mysubclass_instance.get_instance_value_2()
'set from MyClass: second'
>>> myclass_instance = MyClass('first', 'second')
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    myclass_instance = MyClass('first', 'second')
  File "/home/dell/PycharmProjects/hw4-tutorial/5-1.py", line 5, in __init__
    super(type(self), self).set_instance_value_2( iv_2)
AttributeError: 'super' object has no attribute 'set_instance_value_2'
>>>
```

6) (0.25 points) Starting with the definition of MyClass as given in lines 2836-96, change just enough of the example to get line 2896 to display

set from MyMixinClass_2: four

What line(s) did you change? Why?

Answer: Changed MySubclass's signature from

```
class MySubclass(MyMainClass, MyMixinClass_1, MyMixinClass_2): pass
```

to

```
class MySubclass(MyMainClass, MyMixinClass_2, MyMixinClass_1): pass
```

Classes that appear to the left in list take priority (for name resolution over classes that appear to the right in case of name conflicts (if any) involving superclass methods.

7) (0.5 points) When I first attempted to define the Circle class shown in lines 2965-3014, I coded the radius property as follows:

```
def getradius(self):
    if not 'radius' in dir(self): raise UnboundLocalError("radius undefined")
    return self.radius
def setradius(self, r): self.radius = r
def delradius(self):
    if 'radius' in dir(self):  del self.radius
radius = property(getradius, setradius, delradius, 'circle radius')
```

Run the example in lines 3009-14, replacing the code for the radius property as shown in the *tour* with the code shown above. What goes wrong? Why?

Answer: It throws RuntimeError (maximum recursion depth exceeded) error , because python's recursion is limited to 999. Using same name 'radius' as an instance variable and as a property confuses Python. It attempts to use the variable as a property rather than as a name of 'self', leading ultimately to stack overflow and program crash.

8) (0.25 points) For each of the two definitions of the Circle class – the one at lines 2965-3007 and the one at lines 3021-74,

- confirm that my logic for defining docstrings for the radius, area, circumference, and diameter properties for Circle is consistent with the syntax prescribed in section 2 of the Python Library documentation
- for each of these definitions of Circle,
  - display the docstring for a Circle object's radius, area, circumference, and diameter properties
  - describe and account for what you see

Answer: First circle example's docstring inspection shows documentation for methods like delarea, delcircumference, deldiameter, delradius while the second circle example does not show that. In terms of code first code defines the getarea, setarea, delarea and the docstring using property method. Second example uses decorator such as @property, @circumference.setter to decorate methods. Once the accessor, mutator, del method and the docstring are defined the variables can be used based on context (set, get, del etc). This can be displayed using help method (as defined in the section 2 of the Python Library documentation). Since the code displays docstring for both cases, the logic confirms with the documentation.

```python
class Circle(object):
    pi = 3.14159
    #
    def __init__(self, **kwargs):
        print(set(kwargs.keys()))
        if len(set(['radius', 'diameter', 'circumference', 'area']) & set(kwargs.keys(
            raise KeyError("constructor must be called with exactly one keyword from 'r
        try:  self.radius = kwargs['radius']
        except KeyError:
            try:  self.diameter = kwargs['diameter']
            except KeyError:
                try:  self.circumference = kwargs['circumference']
                except KeyError:
                    self.area = kwargs['area']
    #
    def getradius(self):
        if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
        return self.r
    def setradius(self, r):  self.r = r
    def delradius(self):
        if 'r' in dir(self):  del self.r
    radius = property(getradius, setradius, delradius, 'circle radius')
    #
    def getdiameter(self):
        try:        return self.radius * 2
        except:        raise UnboundLocalError("diameter undefined")
    def setdiameter(self, d):  self.radius = d / 2
    def deldiameter(self):      del self.radius
    diameter = property(getdiameter, setdiameter, deldiameter, 'circle diameter')
    #
    def getcircumference(self):
        try:        return 2 * Circle.pi * self.radius
        except:        raise UnboundLocalError("circumference undefined")
    def setcircumference(self, c):  self.radius = c / (2 * Circle.pi)
    def delcircumference(self):      del self.radius
    circumference = property(getcircumference, setcircumference, delcircumference, '
    #
```

```
>>> ================================ RESTART ================================
>>>
>>> c = Circle(radius=3)
{'radius'}
>>> c.radius, c.diameter, c.circumference, c.area
(3, 6, 18.849539999999998, 28.274309999999996)
>>> c.area = 3
>>> c.radius, c.diameter, c.circumference, c.area
(0.9772054365109875, 1.954410873021975, 6.139957654577106, 3.0)
>>> del c.circumference
>>> c.radius, c.diameter, c.circumference, c.area
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    c.radius, c.diameter, c.circumference, c.area
  File "/home/dell/PycharmProjects/hw4-tutorial/8/circle-1.py", line 17, in getra
dius
    if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
UnboundLocalError: radius undefined
>>> c = Circle(radius=3)
{'radius'}
>>> help(c)
Help on Circle in module __main__ object:

class Circle(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, **kwargs)
 |
 |  delarea(self)
 |
 |  delcircumference(self)
 |
 |  deldiameter(self)
 |
 |  delradius(self)
 |
```

```python
class Circle(object):
    pi = 3.14159
    #
    def __init__(self, **kwargs):
        print(set(kwargs.keys()))
        if len(set(['radius', 'diameter', 'circumference', 'area']) & set(kwargs.keys(
            raise KeyError("constructor must be called with exactly one keyword from 'r
        try:  self.radius = kwargs['radius']
        except KeyError:
            try:  self.diameter = kwargs['diameter']
            except KeyError:
                try:  self.circumference = kwargs['circumference']
                except KeyError:
                    self.area = kwargs['area']
    #
    def getradius(self):
        if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
        return self.r
    def setradius(self, r):  self.r = r
    def delradius(self):
        if 'r' in dir(self):  del self.r
    radius = property(getradius, setradius, delradius, 'circle radius')
    #
    def getdiameter(self):
        try:        return self.radius * 2
        except:        raise UnboundLocalError("diameter undefined")
    def setdiameter(self, d):  self.radius = d / 2
    def deldiameter(self):      del self.radius
    diameter = property(getdiameter, setdiameter, deldiameter, 'circle diameter')
    #
    def getcircumference(self):
        try:        return 2 * Circle.pi * self.radius
        except:        raise UnboundLocalError("circumference undefined")
    def setcircumference(self, c):  self.radius = c / (2 * Circle.pi)
    def delcircumference(self):      del self.radius
    circumference = property(getcircumference, setcircumference, delcircumference, '
    #
```

```
 |  getradius(self)
 |
 |  setarea(self, a)
 |
 |  setcircumference(self, c)
 |
 |  setdiameter(self, d)
 |
 |  setradius(self, r)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  area
 |      circle area
 |
 |  circumference
 |      circle circumference
 |
 |  diameter
 |      circle diameter
 |
 |  radius
 |      circle radius
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  pi = 3.14159

>>>
```

```python
class Circle(object):
    pi = 3.14159
    #
    def __init__(self, **kwargs):
        if len(set(['radius', 'diameter', 'circumference', 'area']) & set(kwargs.keys(
            raise KeyError("constructor must be called with exactly one keyword from 'r
        try:  self.radius = kwargs['radius']
        except KeyError:
            try:  self.diameter = kwargs['diameter']
            except KeyError:
                try:  self.circumference = kwargs['circumference']
                except KeyError:
                    self.area = kwargs['area']
    #
    @property
    def radius(self):
        """circle radius"""
        if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
        return self.r
    @radius.setter
    def radius(self, r):   self.r = r
    @radius.deleter
    def radius(self):
        if 'r' in dir(self):  del self.r
    #
    @property
    def diameter(self):
        """circle diameter"""
        try:        return self.radius * 2
        except:         raise UnboundLocalError("diameter undefined")
    @diameter.setter
    def diameter(self, d):  self.radius = d / 2
    @diameter.deleter
    def diameter(self):     del self.radius
    #
    @property
```

```
>>> ================================ RESTART ================================
>>>
>>> c = Circle(radius=3)
>>> c.radius, c.diameter, c.circumference, c.area
(3, 6, 18.849539999999998, 28.274309999999996)
>>> c.area = 3
>>> c.radius, c.diameter, c.circumference, c.area
(0.9772054365109875, 1.954410873021975, 6.139957654577106, 3.0)
>>> del c.area
>>> c.radius, c.diameter, c.circumference, c.area
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    c.radius, c.diameter, c.circumference, c.area
  File "/home/dell/PycharmProjects/hw4-tutorial/8/circle-2.py", line 18, in radiu
s
    if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
UnboundLocalError: radius undefined
>>> c = Circle(radius=3)
>>> help(c)
Help on Circle in module __main__ object:

class Circle(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, **kwargs)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  area
```

```
  return self.r
@radius.setter
def radius(self, r):   self.r = r
@radius.deleter
def radius(self):
  if 'r' in dir(self):  del self.r
#
@property
def diameter(self):
  """circle diameter"""
  try:       return self.radius * 2
  except:       raise UnboundLocalError("diameter undefined")
@diameter.setter
def diameter(self, d):  self.radius = d / 2
@diameter.deleter
def diameter(self):     del self.radius
#
@property
def circumference(self):
  """circle circumference"""
  try:       return 2 * Circle.pi * self.radius
  except:       raise UnboundLocalError("circumference undefined")
@circumference.setter
def circumference(self, c):  self.radius = c / (2 * Circle.pi)
@circumference.deleter
def circumference(self):     del self.radius
#
@property
def area(self):
  """circle area"""
  try:       return Circle.pi * self.radius * self.radius
  except:       raise UnboundLocalError("circumference undefined")
@area.setter
def area(self, a):   self.radius = pow(a / Circle.pi, 0.5)
@area.deleter
def area(self):      del self.radius
```

```
UnboundLocalError: radius undefined
>>> c = Circle(radius=3)
>>> help(c)
Help on Circle in module __main__ object:

class Circle(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, **kwargs)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  area
 |      circle area
 |
 |  circumference
 |      circle circumference
 |
 |  diameter
 |      circle diameter
 |
 |  radius
 |      circle radius
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  pi = 3.14159

>>>
```

9) (0.5 points) As I was writing this assignment, I hit upon a strategy for using Python's object.__class__.mro function (see ll. 2865, 2868, 2871, and 2874) to manage superclass invocation in the presence of multiple inheritance. The following exercise walks you through this strategy:

- Modify MyMainClass's __init__ method so that it
  - takes one parameter besides self: a keyword parameter;
  - throws an exception if this keyword parameter lacks a keyword named v1
  - otherwise, passes v1's value to self.set_instance_value_1
- Similarly, modify MyMixinClass's __init__ method so that it
  - takes one parameter besides self: a keyword parameter;
  - throws an exception of this keyword parameter lacks a keyword named v2
  - otherwise, passes v2's value to self.set_instance_value_2
- Modify MySubclass's __init__ method so that it
  - takes one parameter besides self: a keyword parameter;
  - invokes the __init__ method for each class in MySubclass.__bases__ on this parameter
- Modify l. 3192 so that MySubclass is invoked with two keyword arguments:
  - v1, with a value of 'one'
  - v2, with a value of 'two'

Submit a screen shot of your work, along with the execution of lines 3192-3194 for the new code. (***screen shot***)

Answer:

mro_example-1.py

```
class MyMainClass:
    def __init__(self, **kwargs):
        if 'v1' in kwargs:
            self.set_instance_value_1(kwargs['v1'])
        else:
            raise KeyError("v1 not found")
    #
    def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MyMainClass: ' + v
    def get_instance_value_1(self):       return self.instance_value_1

class MyMixinClass:
    def __init__(self, **kwargs):
        if 'v2' in kwargs:
            self.set_instance_value_2(kwargs['v2'])
        else:
            raise KeyError("v2 not found")
    #
    def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MyMixinClass: ' + v
    def get_instance_value_2(self):       return self.instance_value_2

# - - - simpler here, I think, to name the classes then to refer to super(),
# - - - which would change if the order of inheritance is varied

class MySubclass(MyMainClass, MyMixinClass):
    def __init__(self, **kwargs ):
        MyMainClass.__init__( self, **kwargs )
        MyMixinClass.__init__( self, **kwargs)

my_subclass_instance = MySubclass(v1='one', v2= 'two')
print(my_subclass_instance.get_instance_value_1())    # from MyClass
print(my_subclass_instance.get_instance_value_2())    # from MyMixinClass
```

Run mro_example-1

```
set from MyMainClass: one
set from MyMixinClass: two
```

10) (0.5 points) The code at lines 3265-3277 violates the DRY principle, in that the any expressions at the end of each return clause are identical, up to the use of different operators for comparisons. Clean this code up by

- Adding another instance function to Elist that
  - accepts three parameters: self, other, and a comparison operator
  - returns the result of the any expression, but with the comparison operator applied to a and b
- Rewriting the other six functions, by
  - replacing the expressions in any with
  - calls to your new function, using operators from the Python library's operator module (see §10.3)

Submit a screen shot of your work, along with a few sample tests. (***screen shot***)

Answer:

```
import operator

class Elist(list):
  def __lt__(self, other):
    return NotImplemented if not isinstance(other, Elist) else self.custom_func(other, operator.lt)
  def __le__(self, other):
    return NotImplemented if not isinstance(other, Elist) else self.custom_func(other, operator.le)
  def __eq__(self, other):
    return isinstance(other, Elist) and self.custom_func(other, operator.eq)
  def __ne__(self, other):
    return not isinstance(other, Elist) or self.custom_func(other, operator.ne)
  def __ge__(self, other):
    return NotImplemented if not isinstance(other, Elist) else self.custom_func(other, operator.ge)
  def __gt__(self, other):
    return NotImplemented if not isinstance(other, Elist) else self.custom_func(other, operator.gt)
  def custom_func(self, other, comp_oper):
      return any([comp_oper (x , y ) for x in self for y in other])

e = Elist([1,2])
f = Elist([3,4])
g = Elist([1,5])
print(f.__lt__(e))
print(e.__lt__(f))
print(g.__lt__(f))
print(g.__gt__(f))
```

```
Run  list_dry
True
True
True
```

11) (0.5 points) Modify your comparison function from your answer to the previous problem so that the various comparisons are true if half or more of the items in self satisfy the comparison with at least one of the items in other. Submit your updated class—call it Hlist, for "half list"—along with a few sample tests. (***screen shot***)

Answer:

```
class Hlist(list):
  def __lt__(self, other):
    return NotImplemented if not isinstance(other, Hlist) else self.custom_func(other, operator.lt)
  def __le__(self, other):
    return NotImplemented if not isinstance(other, Hlist) else self.custom_func(other, operator.le)
  def __eq__(self, other):
    return isinstance(other, Hlist) and self.custom_func(other, operator.eq)
  def __ne__(self, other):
    return not isinstance(other, Hlist) or self.custom_func(other, operator.ne)
  def __ge__(self, other):
    return NotImplemented if not isinstance(other, Hlist) else self.custom_func(other, operator.ge)
  def __gt__(self, other):
    return NotImplemented if not isinstance(other, Hlist) else self.custom_func(other, operator.gt)
  def custom_func(self, other, comp_oper):
      max_count_size = 0
      for x in other:
          local_count_size = 0
          for y in self:
              if comp_oper(y, x):
                  local_count_size = local_count_size + 1
          if (local_count_size > max_count_size):
              max_count_size = local_count_size

      if (max_count_size >= len(self)/2):
          return True
      return False

e = Hlist([1,2])
f = Hlist([3,4])
g = Hlist([1,5])
print(f.__lt__(e))
print(e.__lt__(f))
print(g.__lt__(f))
print(g.__gt__(f))
```

```
Run  list_dry
False
True
True
True
```

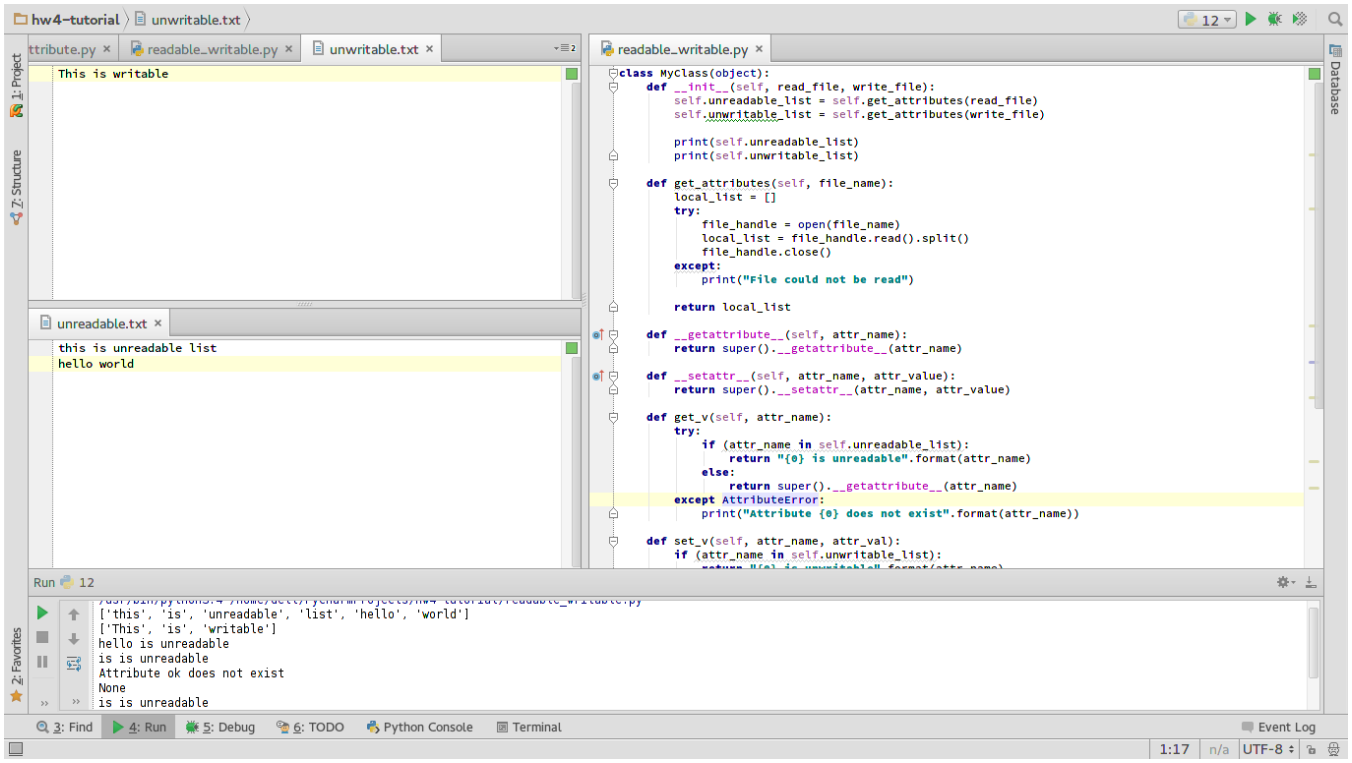
12) (2 points) Modify the version of MyClass from lines 3421-3444 as follows:

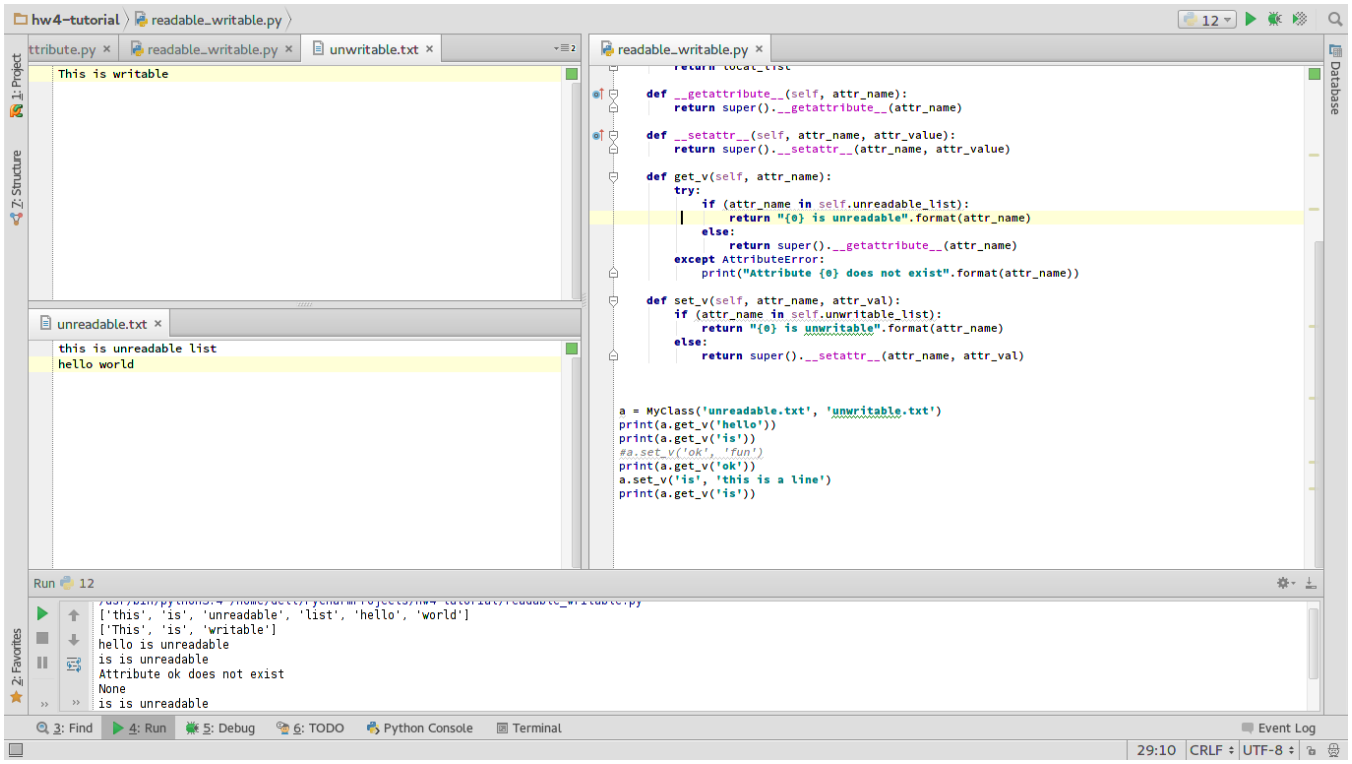
- change the __init__ method so that it accepts two parameters (other than self):
  - the name of a file from which to load the names of attributes that the class is to treat as unreadable
  - the name of a file from which to load the names of attributes that the class is to treat as unwriteable
- for each of these files, the __init__ method should do the following:
  - if the file can be opened for reading as a text file (see Python library, section 2, open())
    - for all words in the file — i.e., maximal sequences of non-whitespace characters that begin with a letter or an underscore — treat those words as the names of unreadable or unwriteable attributes, according to the file being opened
    - be sure to close the file when done reading it
  - otherwise
    - print an appropriate and fully descriptive error message
    - treat any attribute as readable or writeable, according to the file being opened
- change the get_v and set_v methods to accept an additional parameter: the name of the attribute to access:

Produce a screenshot of your code, with a few representative test cases (***screen shot***)
Hints:

- Review the Python library section 16.2.3.1 doc on text I/O and section 6.2 doc on regular expressions. The re library \w and \W shorthands will save a fair amount of typing, as will the judicious use of comprehensions to filter words from non-words.
- If a class defines __getattribute__, ***all*** of that class's instance methods ***must*** use super().__getattribute__('foo') to read that instance's foo attribute. Expressions like self.foo will produce errors. This is true for ***all*** instance attributes.
- If a class defines __setattr__, ***all*** of that class's instance methods ***must*** use super().__setattr__('foo', 3) to set that instance's foo attribute to 3. Expressions like self.foo=3 will produce errors. This is true for ***all*** instance attributes.

Answer:





13) 0.5 points) Since I created the virtual method example in the tour (ll. 3462-3501), I've discovered a cleaner way of getting the effect of virtual attributes, using __getattr__, a method that does lookups for only unknown attributes. I've also figured out a strategy for managing two-level attribute lookups. Both are illustrated by the following sample code:

```
class person_set(object):
    class partially_qualified_person_set(object):
        def __init__(self, people, qualifier):
            valid_qualifiers = ['First', 'Last']
            assert qualifier in valid_qualifiers, "invalid qualifier: {} (must be in {})".format(valid_qualifiers)
            self.people = people
            self.qualifier = valid_qualifiers.index(qualifier)
        def __getattr__(self, attr_name):
            return [person[1-self.qualifier] for person in self.people if person[self.qualifier] == attr_name]
    #
    def __init__(self, person_set):      self.person_set = person_set
    def __getattr__(self, attr_name):
        return person_set.partially_qualified_person_set(self.person_set, attr_name)

all_students = \
  set([ ('Evan', 'Blankenship'), ('Jordan', 'Brown'), ('Mark', 'Buckner'), ('Jason', 'Bunn'), ('Ben', 'Burton'),
      ('Yan', 'Cao'), ('Brad', 'Cross'), ('Joseph', 'Elliott'), ('Dale', 'Giblin'), ('Kamrul', 'Hasan'),
      ('Elijah', 'Laws'), ('Jalaj', 'Nautiyal'), ('Pramod', 'Nepal'), ('Cindy', 'Taylor')] )

x=person_set(all_students)
x.First.Evan
```

x.Last.Taylor

Extend this example to support

- names as triples rather than names as pairs (i.e., first, middle, last)
- a middle name qualifier
- any combination of upper and lower case letters for all qualifiers
- a list of pairs as a return value, where the qualified name (first, middle, last) is omitted

Show your code and a couple of representative test cases (***screen shot***)

Hints:

- Python 3.4's str class has a useful method, casefold, for blurring case differences between strings.
- If you use casefold, however, do be careful, to preserve the cases of people's names in your output

```
class person_set(object):
    class partially_qualified_person_set(object):
        def __init__(self, people, qualifier):
            valid_qualifiers = ['first', 'middle', 'last']
            assert qualifier.casefold() in valid_qualifiers, "invalid qualifier: {} (must be in {})".format(valid_qualifiers)
            self.people = people
            self.qualifier = valid_qualifiers.index(qualifier.casefold())

        def __getattr__(self, attr_name):
            list_counter = [0,1,2]
            list_counter.remove(self.qualifier)

            return [(person[list_counter[0]], person[list_counter[1]]) for person in self.people if person[self.qualifier] == attr_name]
    #
    def __init__(self, person_set): self.person_set = person_set
    def __getattr__(self, attr_name):
        return person_set.partially_qualified_person_set(self.person_set, attr_name)

all_students = \
     set([ ('Evan', '', 'Blankenship'), ('Jordan', '', 'Brown'), ('Mark', '', 'Buckner'), ('Jason','', 'Bunn'), ('Ben','', 'Burton'),
           ('Yan','', 'Cao'), ('Brad','', 'Cross'), ('Joseph','', 'Elliott'), ('Dale','', 'Giblin'), ('Kamrul', 'Md', 'Hasan'),
           ('Elijah','', 'Laws'), ('Jalaj', 'Anil', 'Nautiyal'), ('Pramod','', 'Nepal'), ('Cindy', '', 'Taylor')] )

x=person_set(all_students)
print(x.fiRst.Evan)
print(x.Last.Taylor)
print(x.First.Jalaj)
```

```
Run 13-1
[('', 'Blankenship')]
[('Cindy', '')]
[('Anil', 'Nautiyal')]
```

14) (0.75 points) Dependency inversion exercise, part 1. Consider the following code:

```
def provolone_topping_name():  return 'provolone'

def provolone_topping_type():  return 'cheese'

def chicken_topping_name():  return 'chicken'

def chicken_topping_type():  return 'meat'

def red_pepper_topping_name():  return 'red pepper'

def red_pepper_topping_type():  return 'vegetable'

class Pizza(object):
    def __init__(self, toppings):
        self.toppings = toppings
    def show_topping_names(self):
        def get_name(t):
            if   t == "provolone":   return provolone_topping_name()
```

```
            elif  t == "chicken":    return chicken_topping_name()
            elif  t == "red pepper":  return red_pepper_topping_name()
            else:                return "??"
        print("toppings = ", end='');
        for t in self.toppings[:-1]:  print(get_name(t), ",", sep="", end="")
        print(get_name(self.toppings[-1]))
    def show_topping_types(self):
        def get_type(t):
            if   t == "provolone":   return provolone_topping_type()
            elif  t == "chicken":    return chicken_topping_type()
            elif  t == "red pepper":  return red_pepper_topping_type()
            else:                return "??"
        print("topping types = {}".format(set([get_type(t) for t in self.toppings])))

Pizza(['provolone', 'chicken', 'horsefeathers']).show_topping_names()
Pizza(['provolone', 'chicken', 'horsefeathers']).show_topping_types()
```

This use of repeated if clauses to take action based on an entity's type is known as ***run-time type inference***, or ***RTTI*** for short. While RTTI-based code compiles and runs, it's typically regarded as awful; as shown here, it creates maintenance headaches by violating DRY:

- In order to add or remove an ingredient to the set of toppings, a developer has to change Pizza.show_topping_names.get_name and Pizza.show_topping_types.get_type in addition to adding two functions.
- If Pizza.__init__ included a check for valid toppings, another redundant change would be needed as well.

For this problem, use the principle of dependency inversion to redo this code, as follows:

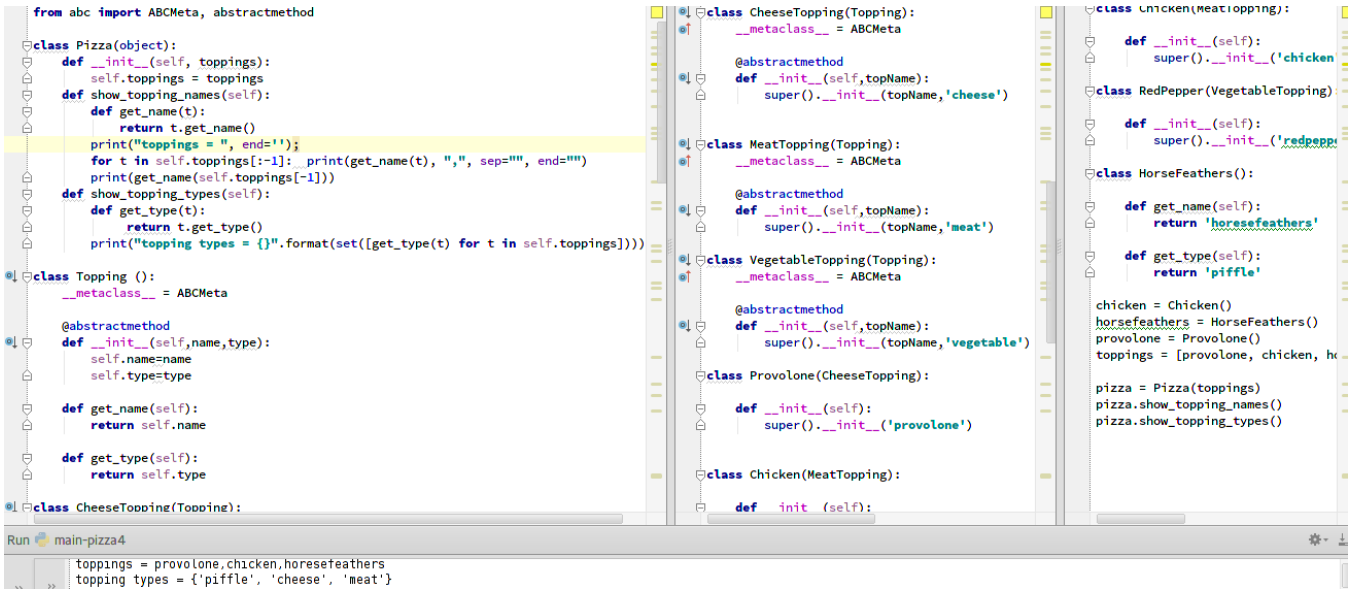
- Create an abstract base class, Topping, with
  - Two instance attributes: name and type
  - An __init__ method that takes two arguments, name and type, and sets its instance attributes accordingly
  - Two instance methods that return the name and type instance attributes, respectively.
- Create three abstract base classes, CheeseTopping, MeatTopping, and VegetableTopping,
  - each of which inherits from Topping
  - each of which has one instance method: an __init__method that
    - takes one argument, a topping name
    - invokes super().__init__ on its name argument and either 'cheese' (for CheeseTopping), 'meat' (for MeatTopping), or 'vegetable' (for VegetableTopping)
- Create three concrete classes, Provolone, Chicken, and RedPepper,
  - each of which inherits from one superclass: i.e., CheeseTopping (for Provolone), MeatTopping (for Chicken), and VegetableTopping (for RedPepper)
  - each of which has one instance method: an __init__method that
    - takes no arguments
    - invokes super().__init__ on either 'provolone' (for Provolone), 'chicken' (for Chicken), or 'red pepper' (forRedPepper)
- Create a fourth concrete class, Horsefeathers, with two instance methods:
  - a first method, get_name(), that returns 'horsefeathers'
  - a second method, get_type(), that returns 'piffle'

Show your code and a couple of representative test cases (***screen shot***)

Hints:

- For examples of abstract base class creation in Python, see *tour*, ll. 3650, 3652. Use multiple inheritance to support creation for classes that need to inherit from a user-defined class as well as ABCmeta.

Answer:



15) (0.25 points) *Dependency inversion exercise, part 2.* In the previous example, Pizza's methods treat Horsefeathers() as a topping, even though Horsefeathers isn't a subclass of Topping. This use of names as a substitute for typed attributes is known as ***latent typing*** or, more informally, as ***duck typing***. The latter name comes from an American proverb: "If it walks like a duck and quacks like a duck, it's a duck."

The standard rationale for latent typing, from what I can tell, is threefold:

- Type declarations and type checking create syntactic clutter.
- For the most part, latent typing is good enough, because
    - "Real" programmers are intelligent, and
    - anyone who's stupid enough to create name collisions like the ones shown above and who's too stupid to debug them deserves what they get.
- If you really need the type checks, you can add them by hand.

While I sympathize with the argument about syntactic clutter, I've made enough stupid mistakes to appreciate the value of explicit type-checking. So, for this exercise, to get a feel for what such a check might entail, add a loop to Pizza.__init__ that

- Checks that each topping is an instance of class Topping, and
- Throws an assertion with a descriptive error message upon encountering a "bogus" topping.

Show your code and a couple of representative test cases (***screen shot***)

```
from abc import ABCMeta, abstractmethod

class Pizza(object):
    def __init__(self, toppings):
        for t in toppings:
            assert isinstance(t, Topping), "bogus"

        self.toppings = toppings

    def show_topping_names(self):
        def get_name(t):
            return t.get_name()

        print("toppings = ", end='');
        for t in self.toppings[:-1]: print(get_name(t), ",", sep="", end="")
        print(get_name(self.toppings[-1]))

    def show_topping_types(self):
        def get_type(t):
            return t.get_type()

        print("topping types = {}".format(set([get_type(t) for t in self.toppings])))

class Topping ():
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,name,type):
        self.name=name
        self.type=type

    def get_name(self):
        return self.name

    def get_type(self):
        return self.type
```

```
        return self.name

    def get_type(self):
        return self.type

class CheeseTopping(Topping):
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,topName):
        super().__init__(topName,'cheese')


class MeatTopping(Topping):
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,topName):
        super().__init__(topName,'meat')

class VegetableTopping(Topping):
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,topName):
        super().__init__(topName,'vegetable')

class Provolone(CheeseTopping):

    def __init__(self):
        super().__init__('provolone')


class Chicken(MeatTopping):

    def __init__(self):
```

```
class Chicken(MeatTopping):

    def __init__(self):
        super().__init__('chicken')

class RedPepper(VegetableTopping):

    def __init__(self):
        super().__init__('redpepper')

class HorseFeathers():

    def get_name(self):
        return 'horsefeathers'

    def get_type(self):
        return 'piffle'

chicken = Chicken()
horsefeathers = HorseFeathers()
provolone = Provolone()
#toppings = [provolone, chicken, horsef
toppings = [provolone, chicken]
pizza = Pizza(toppings)

pizza.show_topping_names()
pizza.show_topping_types()
```

```
class Chicken(MeatTopping):

    def __init__(self):
        super().__init__('chicken')

class RedPepper(VegetableTopping):

    def __init__(self):
        super().__init__('redpepper')

class HorseFeathers():

    def get_name(self):
        return 'horsefeathers'

    def get_type(self):
        return 'piffle'

chicken = Chicken()
horsefeathers = HorseFeathers()
provolone = Provolone()
#toppings = [provolone, chicken, horsefeathers]
toppings = [provolone, chicken]
pizza = Pizza(toppings)

pizza.show_topping_names()
pizza.show_topping_types()
```

main-pizza5

```
toppings = provolone,chicken
topping types = {'cheese', 'meat'}
```

```
class HorseFeathers():

    def get_name(self):
        return 'horsefeathers'

    def get_type(self):
        return 'piffle'

chicken = Chicken()
horsefeathers = HorseFeathers()
provolone = Provolone()
toppings = [provolone, chicken, horsefeathers]
#toppings = [provolone, chicken]
pizza = Pizza(toppings)

pizza.show_topping_names()
pizza.show_topping_types()
```

```
n main-pizza5
Traceback (most recent call last):
  File "/home/dell/PycharmProjects/hw4-tutorial/pizza_example-15/main-pizza5.py", line 89, in <module>
    pizza = Pizza(toppings)
  File "/home/dell/PycharmProjects/hw4-tutorial/pizza_example-15/main-pizza5.py", line 6, in __init__
    assert isinstance(t, Topping), "bogus"
AssertionError: bogus
```

16) (0.5 points) *Serialization*. Starting with the Pizza class from the last two problems, add two methods in support of serialization:

- an __eq__ method that treats two Pizza objects as equal if
  - the other object is an instance of Pizza and
  - both have the same set of toppings.
- a repr method that meets the requirement for repr methods given on line 3930.

In order to get repr and __eq__ to work properly, you'll also need to add two methods to Topping:

- an __eq__ method that treats two Topping objects as equal if
  - the other object is an instance of Topping and
  - both have the same name and type attributes.
- a repr method that returns self's class name with an empty constructor (e.g., 'Chicken()' for a Chicken topping object)

Show your code and a couple of representative test cases (***screen shot***)

Answer:

```
from abc import ABCMeta, abstractmethod
class Pizza(object):
    def __init__(self, toppings):
        for t in toppings:
            assert isinstance(t, Topping), "bogus"

        self.toppings = toppings

    def __eq__(self, other):
        pizza_equal = False
        toppings_equal = True

        if isinstance(other, Pizza):
            pizza_equal = True

        for x in other.toppings:

            local_equal_toppings = False
            for y in self.toppings:

                if (x.__eq__(y)):
                    local_equal_toppings = True
                    break
            if (local_equal_toppings == False):
                toppings_equal = False
                break
        if (toppings_equal == True and pizza_equal == True):
            return True
        else:
            return False
        else:
            return False

    def __repr__(self):
        return "{0}({1!r})".format(self.__class__.__name__, self.toppings)

    def show_topping_names(self):
        def get_name(t):
            return t.get_name()

        print("toppings = ", end='');
        for t in self.toppings[:-1]: print(get_name(t), ",", sep="", end="")
        print(get_name(self.toppings[-1]))

    def show_topping_types(self):
        def get_type(t):
            return t.get_type()

        print("topping types = {}".format(set([get_type(t) for t in self.toppings])))

class Topping ():
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,name,type):
        self.name=name
        self.type=type

    def get_name(self):
        return self.name
```

Run main-pizza9

```
True
False
```

```
    @abstractmethod
    def __init__(self,name,type):
        self.name=name
        self.type=type

    def get_name(self):
        return self.name

    def get_type(self):
        return self.type

    def __eq__(self, other):
        if (self.name == other.get_name() and self.type == other.get_type()):
            return True
        else:
            return False

    def __repr__(self):
        return "{0}()".format(self.__class__.__name__)

class CheeseTopping(Topping):
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,topName):
        super().__init__(topName,'cheese')


class MeatTopping(Topping):
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,topName):
        super().__init__(topName,'meat')

class VegetableTopping(Topping):
    __metaclass__ = ABCMeta

    @abstractmethod
    def __init__(self,topName):
        super().__init__(topName,'vegetable')

class Provolone(CheeseTopping):

    def __init__(self):
        super().__init__('provolone')


class Chicken(MeatTopping):

    def __init__(self):
        super().__init__('chicken')

class RedPepper(VegetableTopping):

    def __init__(self):
        super().__init__('redpepper')

class Chicken(MeatTopping):

    def __init__(self):
        super().__init__('chicken')

class RedPepper(VegetableTopping):

    def __init__(self):
        super().__init__('redpepper')

class HorseFeathers():

    def get_name(self):
        return 'horsefeathers'

    def get_type(self):
        return 'piffle'

pizza = Pizza([Chicken(), Provolone()])
print(eval(repr(pizza)) == pizza)

pizza1 = Pizza([RedPepper(), RedPepper()])
print(eval(repr(pizza1)) == pizza)
```

Run main-pizza9

```
True
False
```

17) (0.5 points) Create a doctest-based (see ll. 3797-3841) function for testing class Fib's methods (see ll. 4083-4096). Your function should

- Accept one argument: a value—say, k—to use as an argument for Fib's constructor
- Use Fib(k) to return a list of the first k values of the Fibonacci series, in order
- Include a docstring that correctly tests the function's operation for at least five arguments: 0, 1, 2, 10, and a non-numeric value, like a string

Show your code and the result of doctesting your test function with the verbose=True option (***screen shot***)

Answer:

```
import doctest

class Fib:
  def __init__(self, val_count = float('inf')):
    self.initial_val_count = val_count
  #
  def __iter__(self):
    self.val_count, self.result_queue = self.initial_val_count, [0, 1]
    return self
  #
  def __next__(self):
    if self.val_count <= 0:  raise StopIteration
    self.val_count -= 1
    self.next_result = self.result_queue[0]
    self.result_queue = [self.result_queue[1], self.result_queue[0] + self.result_
    return self.next_result


def docstring_fib(k):
    """  print fibonacii series for few numbers
    >>> docstring_fib(0)
    []
    >>> docstring_fib(1)
    [0]
    >>> docstring_fib(2)
    [0, 1]
    >>> docstring_fib(10)
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    >>> docstring_fib('a string')
    Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
    docstring_fib('a string')
    File "/tmp/fibmy.py", line 36, in docstring_fib
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 36, in <listcomp>
    return [i for i in Fib(k)]
```

```
>>> doctest.run_docstring_examples(docstring_fib, None, optionflags=doctest.ELLIP
SIS, verbose=True)
Finding tests in NoName
Trying:
    docstring_fib(0)
Expecting:
    []
ok
Trying:
    docstring_fib(1)
Expecting:
    [0]
ok
Trying:
    docstring_fib(2)
Expecting:
    [0, 1]
ok
Trying:
    docstring_fib(10)
Expecting:
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
ok
Trying:
    docstring_fib('a string')
Expecting:
    Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
    docstring_fib('a string')
    File "/tmp/fibmy.py", line 36, in docstring_fib
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 36, in <listcomp>
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 12, in __next__
    if self.val_count <= 0:  raise StopIteration
    TypeError: unorderable types: str() <= int()
```

```
  #
  def __iter__(self):
    self.val_count, self.result_queue = self.initial_val_count, [0, 1]
    return self
  #
  def __next__(self):
    if self.val_count <= 0:  raise StopIteration
    self.val_count -= 1
    self.next_result = self.result_queue[0]
    self.result_queue = [self.result_queue[1], self.result_queue[0] + self.result_
    return self.next_result


def docstring_fib(k):
    """  print fibonacii series for few numbers
    >>> docstring_fib(0)
    []
    >>> docstring_fib(1)
    [0]
    >>> docstring_fib(2)
    [0, 1]
    >>> docstring_fib(10)
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    >>> docstring_fib('a string')
    Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
    docstring_fib('a string')
    File "/tmp/fibmy.py", line 36, in docstring_fib
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 36, in <listcomp>
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 12, in __next__
    if self.val_count <= 0:  raise StopIteration
    TypeError: unorderable types: str() <= int()
    """
    return [i for i in Fib(k)]
```

```
    File "/tmp/fibmy.py", line 36, in docstring_fib
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 36, in <listcomp>
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 12, in __next__
    if self.val_count <= 0:  raise StopIteration
    TypeError: unorderable types: str() <= int()
**********************************************************************
File "/tmp/fibmy.py", line 29, in NoName
Failed example:
    docstring_fib('a string')
Expected:
    Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
    docstring_fib('a string')
    File "/tmp/fibmy.py", line 36, in docstring_fib
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 36, in <listcomp>
    return [i for i in Fib(k)]
    File "/tmp/fibmy.py", line 12, in __next__
    if self.val_count <= 0:  raise StopIteration
    TypeError: unorderable types: str() <= int()
Got:
    Traceback (most recent call last):
      File "/usr/lib/python3.4/doctest.py", line 1318, in __run
        compileflags, 1), test.globs)
      File "<doctest NoName[4]>", line 1, in <module>
        docstring_fib('a string')
      File "/tmp/fibmy.py", line 41, in docstring_fib
        return [i for i in Fib(k)]
      File "/tmp/fibmy.py", line 41, in <listcomp>
        return [i for i in Fib(k)]
      File "/tmp/fibmy.py", line 12, in __next__
        if self.val_count <= 0:  raise StopIteration
    TypeError: unorderable types: str() <= int()
>>>
```