Exercise Sheet 3

Exercise 1 - Exceptions

Write a function called oops that explicitly raises an IndexError exception when called. Then write another function that calls oops inside a try/except statement to catch the error. What happens if you change oops to raise KeyError instead of IndexError? Also try to print a Python stack trace by using the print_exc() function in the standard traceback module (see the Python library reference for details)

```
import traceback

def oops():
    raise IndexError

try:
    oops()
except IndexError:
    print('got␣an␣IndexError')
    traceback.print_exc()
```

Exercise 2 - Operator Overloading

Write a class called Mylist that "wraps" a Python list: it should overload most list operator and operations: '+', indexing, iteration, slicing, and list methods such as append and sort. Also provide a constructor for your class that takes an existing list (or a Mylist instance) and copies its components into an instance member. Experiment with your class interactively. Things to explore:

- Why is copying the intitial value important here?
- Can you use an empty slice (e.g. start [:] to copy the initial value if its a Mylist instance?
- Is there a general way to route list method calls to the wrapped list?
- Can you add a Mylist and a regular list? How about a list and a Mylist instance?
- What type of object should operations like + and slicing return; how about indexing?

```
class MyList:
  def __init__(self, start):
    #self.wrapped = start[:]
    self.wrapped = []
    for x in start: self.wrapped.append(x)
  def __add__(self, other):
    return MyList(self.wrapped + other)
  def __mul__(self, time):
    return MyList(self.wrapped * time)
  def __getitem__(self, offset):
    return self.wrapped[offset]
  def __len__(self):
    return len(self.wrapped)
  def __getslice__(self, low, high):
    return MyList(self.wrapped[low:high])
  def append(self, node):
    self.wrapped.append(node)
  def __getattr__(self, name):
    return getattr(self.wrapped, name)
  def __repr__(self):
    return repr(self.wrapped)

if __name__ == '__main__':
  x = MyList('spam')
  print(x)
  print(x[2])
  print(x[1:])
  print(x + ['eggs'])
  print(x * 3)
  x.append('a')
  x.sort()
  for c in x: print c
```

Exercise 3 - Kangaroo classes

This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named Kangaroo with the following methods:

a) An ___init___ method that initializes an attribute named pouch_contents to an empty list.

b) A method named put_in_pouch that takes an object of any type and adds it to pouch_contents.

c) A ___str___ method that returns a string representation of the Kangaroo object and the contents of the pouch.

Test your code by creating two Kangaroo objects, assigning them to variables named kanga and roo, and then adding roo to the contents of kanga's pouch.
Open the file BadKangaroo.py in the course repository. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug.
Hint: you can use pylint to highlight the bug for you

```
"""This module contains a code example related to

Think Python, 2nd Edition
by Allen Downey
http://thinkpython2.com

Copyright 2015 Allen Downey

License: http://creativecommons.org/licenses/by/4.0/
"""

from __future__ import print_function, division

"""

WARNING: this program contains a NASTY bug.  I put
it there on purpose as a debugging exercise, but
you DO NOT want to emulate this example!

"""

class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        # The problem is the default value for contents.
        # Default values get evaluated ONCE, when the function
        # is defined; they don't get evaluated again when the
        # function is called.

        # In this case that means that when __init__ is defined,
        # [] gets evaluated and contents gets a reference to
        # an empty list.

        # After that, every Kangaroo that gets the default
        # value gets a reference to THE SAME list.  If any
        # Kangaroo modifies this shared list, they all see
        # the change.

        # The next version of __init__ shows an idiomatic way
        # to avoid this problem.
        self.name = name
        self.pouch_contents = contents

    def __init__(self, name, contents=None):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        # In this version, the default value is None.  When
        # __init__ runs, it checks the value of contents and,
        # if necessary, creates a new empty list.  That way,
        # every Kangaroo that gets the default value gets a
        # reference to a different list.

        # As a general rule, you should avoid using a mutable
        # object as a default value, unless you really know
        # what you are doing.
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents

    def __str__(self):
        """Return a string representaion of this Kangaroo.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.pouch_contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
```

```
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Adds a new item to the pouch contents.

        item: object to be added
        """
        self.pouch_contents.append(item)


kanga = Kangaroo('Kanga')
roo = Kangaroo('Roo')
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
kanga.put_in_pouch(roo)

print(kanga)
print(roo)

# If you run this program as is, it seems to work.
# To see the problem, trying printing roo.
```

Exercise 4 - Composition

Simulate a fast-food ordering scenario by defining four classes:

1. Lunch: a container and controller class
2. Customer: the actor that buys food
3. Employee: the actor that a customer orders from
4. Food: what the customer buys

To get you started, here are the classes and methods you'll be defining:

```
class Lunch:
  def __init__(self)                 # make/embed Customer and Employee
  def order(self, foodName)          # start a Customer order simulation
  def result(self)                   # as the Customer what kind of Food it has

class Customer:
  def __init__(self)                   # initialize my food to None
  def placeOrder(self, foodName, employee) # place order with an Employee
  def printFood(self)                # print the name of my food

class Employee:
  def takeOrder(self, foodName)      # return a Food, with requested name

class Food:
  def __init__(self, name)  # store food name
```

The order simulation works as follows:

1. The Lunch class's constructor should make and embed an instance of Customer and Employee, and export a method called order. When called, this order method should ask the Customer to place an order, by calling its placeOrder method. The Customer's placeOrder method should in turn ask the Employee object for a new Food object, by calling the Employee's takeOrder method
2. Food objects should store a food name string (e.g. "burritos"), passed down from Lunch.order to Customer.placeOrder, to Employee.takeOrder, and finally to Food's constructor. The top-level Lunch class should also export a method called result, which asks the customer to print the name of the food it received from the Employee (this can be used to test your simulation)
3. Note that Lunch needs to either pass the Employee to the Customer, or pass itself to the Customer, in order to allow the Customer to call Employee methods.