Ch14-OOP

August 7, 2020

1 Object Oriented Programming (OOP)

http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html
http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_II.html

- we've been using procedural programming paradigm; focus on functions/procedures
- OOP paradigm is best used in large and complex modern software systems
  - OOD (Object Oriented Design) makes it easy to maintain and improve software over time
- focus is on creation of objects which contain both data and functionality together under one name
- typically, each class definition corresponds to some object or concept in the real world with some attributes/properties that maintain its state; and the functions/methods correspond to the ways real-world objects interact

1.1 class

- we've used classes like str, int, float, dict, tuple, etc.
- class keyword lets programmer define their own compound data types
- class is a collection of relevant attributes and methods like real world objects
- syntax:

```
class className:
    [statement-1]
    .
    .
    [statement-N]
```

1.1.1 a simple Point class

- a class that represents a point in 2-D coordinates

[1]:
```
# OK but NOT best practice!
class Point:
    pass
```

[2]:
```
# instantiate an object a of type Point
a = Point()
a.x = 0
a.y = 0
```

```
print(a.x, a.y)
```

```
0 0
```

1.1.2 better class example

[3]:
```
class Point:
    """
    Point class to represent and manipulate x and y in 2D coordinates
    """
    count = 0 # class variable/attribute

    # constructor to customize the initial state of an object
    # first argument refers to the instance being manipulated;
    # it is customary to name this parameter self; but can be anything
    def __init__(self, xx=0, yy=0):
        """Create a new point with given x and y coords"""
        # x and y are object variables/attributes
        self.x = xx
        self.y = yy
        Point.count += 1 # increment class variable

    # destructor gets called
    def __del__(self):
        Point.count -= 1
```

1.2 class members

- like real world objects, object instances can have both attributes and methods
  - attributes are properties that store data/values
  - methods are operations that operate on or use data/values
- use . dot notation to access members
- x and y are attributes of Point class
- *__init__*() (constructor) and *__del__*() (destructor) are sepcial methods
  - more on speical methods later
- can have as many relevant attributes and methods that help mimic real-world objects

[14]:
```
# instantiate an object
p = Point()
# what is the access specifier for attributes?
print('p: x = {} and y = {}'.format(p.x, p.y))
print("Total point objects = {}".format(Point.count)) # access class variable␣
 ↪outside class
# p.__del__() # call destructor explictly
p1 = Point(10, 100)
print("p1: x = {} and y = {}".format(p1.x, p1.y))
print("Total point objects = {}".format(Point.count))
```

```
# Run this cell few times and see the value of Point.count
# How do you fix this problem? Use __del__ destructor method.
```

```
p: x = 0 and y = 0
Total point objects = 2
p1: x = 10 and y = 100
Total point objects = 1
```

[8]:
```
print("Total point objects = {}".format(Point.count))
```

```
Total point objects = 2
```

1.2.1 visualizing class and instance attributes using pythontutor.com

- https://goo.gl/aGuc4r

1.2.2 exercise: add a method dist_from_origin() to Point class

- computes and returns the distance from the origin
- test the methods

[56]:
```
class Point:
    """
    Point class represents and manipulates x,y coords
    """
    count = 0

    def __init__(self, xx=0, yy=0):
        """Create a new point with given x and y coords"""
        self.x = xx
        self.y = yy
        Point.count += 1

    def dist_from_origin(self):
        import math
        dist = math.sqrt(self.x**2+self.y**2)
        return dist

    def __str__(self):
        return "({}, {})".format(self.x, self.y)
```

[37]:
```
p1 = Point(2, 2)
print(p1.dist_from_origin())
```

```
2.8284271247461903
```

1.3 objects are mutable

- can change the state or attributes of an object

[38]:
```
p2 = Point(3, 2)
print(p2)
p2.x = 4
p2.y = 10
print(p2)
```

```
(3, 2)
(4, 10)
```

1.3.1 better approach to change state/attribute is via methods

- move(xx, yy) method is added to class to set new x and y values for a point objects

[1]:
```
class Point:
    """
    Point class represents and manipulates x and y coordinates
    """
    count = 0

    def __init__(self, xx=0, yy=0):
        """Create a new point with given x and y coords"""
        self.x = xx
        self.y = yy
        Point.count += 1

    def dist_from_origin(self):
        import math
        dist = math.sqrt(self.x**2+self.y**2)
        return dist

    def __str__(self):
        return "({}, {})".format(self.x, self.y)

    # use setters to set attributes
    def setX(self, xx):
        if isinstance(x, int) or isinstance(x, float):
            self.x = int(xx)
        elif isinstance(xx, str):
            if xx.isnumeric():
                self.x = int(xx)

    def setY(self, yy):
        if isinstance(x, int) or isinstance(x, float):
            self.y = int(y)
        elif isinstance(yy, str):
```

```
            if yy.isnumeric():
                self.y = int(yy)

    # use getters to get attributes
    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def move(self, xx, yy):
        self.x = xx
        self.y = yy
```

[2]:
```
p3 = Point()
print(p3)
p3.move(10, 20)
print(p3)
```

```
(0, 0)
(10, 20)
```

1.4 sameness - alias or deep copy

[3]:
```
import copy
p2 = Point(3, 4)
p3 = p2 # alias or deepcopy?
print(p2 is p3) # checks if two references refer to the same object
p4 = copy.deepcopy(p2)
print(p2 is p4)
```

```
True
False
```

1.5 passing objects as arguments to functions

[4]:
```
def print_point(pt):
    #pt.x = 100
    #pt.y = 100
    print('({0}, {1})'.format(pt.getX(), pt.getY()))
```

[5]:
```
p = Point(10, 10)
print_point(p)
#print(p)
print(p.getX(), p.getY())
```

```
(10, 10)
10 10
```

1.6 are objects passed by value or reference?

- how can you tell?
- write a simple program to test.

1.7 returning object instances from functions

- object(s) can be returned from functions

[6]:
```
def midpoint(p1, p2):
    """Returns the midpoint of points p1 and p2"""
    mx = (p1.getX() + p2.getY())/2
    my = (p2.getX() + p2.getY())/2
    return Point(mx, my)
```

[7]:
```
p = Point(4, 6)
q = Point(6, 4)
r = midpoint(p, q)
print_point(r) # better way to do this: use __str__() special method
print(r)
```

```
(4.0, 5.0)
(4.0, 5.0)
```

exercise 1: In-class demo: Design a class to represent a triangle and implement methods to calculate area and perimeter.

1.8 composition

- class can include another class as a member
- let's say we want to represent a rectangle in a 2-D coordinates (XY plane)
- corner represents the top left point on a XY plane

[46]:
```
class Rectangle:
    """ A class to manufacture rectangle objects """

    def __init__(self, posn, w, h):
        """ Initialize rectangle at posn, with width w, height h """
        self.corner = posn
        self.width = w
        self.height = h

    def __str__(self):
        return  "({0}, {1}, {2})".format(self.corner, self.width, self.height)

box = Rectangle(Point(0, 0), 100, 200)
```

```
bomb = Rectangle(Point(100, 80), 5, 10)    # In my video game
print("box: ", box)
print("bomb: ", bomb)
```

```
box:  ((0, 0), 100, 200)
bomb:  ((100, 80), 5, 10)
```

1.9 copying objects

- can be challenging as assigning one object to anohter simply creates alias

[47]:
```
r1 = Rectangle(Point(1, 1), 10, 5)
r2 = copy.copy(r1)
```

[48]:
```
# r1 is not r2
r1 is r2
```

[48]:
```
False
```

[49]:
```
# but two corners are same
r1.corner is r2.corner
```

[49]:
```
True
```

[50]:
```
# let's test alias by moving r1 to a different location
r1.corner.move(10, 10)
```

[51]:
```
# you can see r2 is moved to that location as well
print(r1)
print(r2)
```

```
((10, 10), 10, 5)
((10, 10), 10, 5)
```

[52]:
```
# fix: use deepcopy from copy module
r3 = copy.deepcopy(r1)
```

[53]:
```
r1 is r3
```

[53]:
```
False
```

[54]:
```
print(r1, r3)
```

```
((10, 10), 10, 5) ((10, 10), 10, 5)
```

[55]:
```
r1.corner.move(20, 20)
# r1 is moved but not r3
print(r1, r3)
```

```
((20, 20), 10, 5) ((10, 10), 10, 5)
```

[ ]: