lec09_classes

November 2, 2020

0.1 Python Classes

This notebook contains the examples from lecture, plus some additional examples. It seeks to remind us about basics, and some fine points, related to object oriented mechanisms in Python:

- instance and class attributes
- inheritance
- static and class methods
- properties

Please feel free to try these things out, and if anything is confusing/surprising, try usingan environment diagram and/or asking for clarification on the forum!

0.1.1 Instance and Class Attributes

[ ]:
```
class A:
    x = "dog"

a = A()
print("a.x:", a.x)
```

[ ]:
```
x = "cat"

class B(A):
    x = "ferret"
    def __init__(self):
        #self.x = x
        self.x = "tomato"

b = B()
print("b.x:", b.x)

# What will be printed?
#  1 - b.x: dog
#  2 - b.x: ferret
#  3 - b.x: cat
#  4 - b.x: tomato
#  5 - Other or error
```

[ ]:
```
class C(B):
    x = "fish"
    def __init__(self):
        pass
        #B.__init__(self)

c = C()
print("c.x:", c.x)

# What will be printed?
#  1 - c.x: dog
#  2 - c.x: ferret
#  3 - c.x: fish
#  4 - c.x: tomato
#  5 - Other or error
```

[ ]:
```
# A glimpse behind the curtain: instance attributes are stored in a dict␣
 ↪associated with each instance
print(a.__dict__)
print(b.__dict__)
print(c.__dict__)
print()
print(A.__dict__)
print(B.__dict__)
print(C.__dict__)
print()
print("c.__class__ =", c.__class__)
print("c.__class__.__name__ =", c.__class__.__name__)
print("type(c) =", type(c))
print("isinstance(c, A):", isinstance(c, A))
```

Some attribute accessors (fine points)

[ ]:
```
print(getattr(c, 'x', "default val"))
print(getattr(c, 'y', "default val"))
print(c.x)
print(C.x)
print(type(c).x)
```

0.1.2 Method inheritance

Methods can be thought of as class (or instance) attributes that happen to be functions: they are resolved similarly, then called on their arguments. There are various special syntactic forms and protocols that govern instance creation, initialization, destruction, as well as method invocation syntax to make it convenient to pass along the instance object itself.

[ ]:
```
class Bar():
    def __init__(self, val):
        self.x = val

class Foo(Bar):
    x = 100
    def increment(this):  # conventionally 'self' rather than 'this' or other␣
 ↪variable names
        this.x += 1

f = Foo(33)
print("f.x:", f.x)
f.increment()
print("f.x:", f.x)
```

Invoking a superclass method:

[ ]:
```
class Bar():
    def __init__(self, val):
        self.x = val

class Foo(Bar):
    x = 100
    def __init__(self, val):
        Bar.__init__(self, val)
        self.x = self.x * Foo.x

    def increment(self):
        self.x += 1

f = Foo(33)
print("f.x:", f.x)
f.increment()
print("f.x:", f.x)
```

Invoking a subclass method from a superclass:

[ ]:
```
class Bar():
    def __init__(self, val):
        self.x = val

    def double_increment(self):
        self.increment()
        self.increment()
        #type(self).increment(self)

class Foo(Bar):
```

```
    def increment(self):
        self.x += 1

class Gorp(Bar):
    delta = 100
    def increment(self):
        self.x += self.delta

f = Foo(0)
print("f.x:", f.x)
f.double_increment()
print("f.x:", f.x)

g = Gorp(0)
print("g.x:", g.x)
g.double_increment()
print("g.x:", g.x)
```

[ ]:
```
class Gorp(Bar):
    delta = 100
    def increment(self):
        self.x += self.delta

    def set_delta(self, d): #note -- conventional method, takes self argument
        Gorp.delta = d
        return Gorp.delta

    @staticmethod
    def set_del(d):  #note -- staticmethod, does not take self argument
        Gorp.delta = d
        return Gorp.delta

g = Gorp(0)
print("g.set_delta(200):", g.set_delta(200))  # but feels wrong to change a␣
 ↪class attribute through g.something...
print("g.set_del(300):", g.set_del(300))      # still feels wrong
print("Gorp.set_del(400):", Gorp.set_del(400))  # cleaner/clearer
```

@classmethod – passes class of target rather than target Sometimes we want to have the class of the object as the lead argument, not the object itself (particularly useful for methods that create new instances of class, rather than working with a particular pre-existing instance):

[ ]:
```
class Polygon:
    color = "white"
    @classmethod
    def shade(cls, color):
        cls.color = color

class Rectangle(Polygon):
    color = "green"

class Square(Rectangle):
    color = "blue"

print("Square.color:", Square.color)
Square.shade("red")
print("Square.color:", Square.color)

r1 = Rectangle()
r2 = Rectangle()
print("Rectangle.color:", Rectangle.color)
print("r2.color:", r1.color)
r1.shade("burnt orange") # UGLY -- changes color of ALL Rectangles, not just r1
print("after r1.shade('burnt orange'), r2.color:", r1.color)
```

0.3 Properties

Suppose we want a simple "object.x" syntax for getting or setting an attribute, but we want/need computation beyond just looking up or setting an instance variable? We have `@property` and `@<var>.setter` for this:

[ ]:
```
class Bounded():
    """ Keep track of variable x, but clipped to xmin and xmax """
    xmin, xmax = 50, 100
    def __init__(self, x):
        self.__x = x  # __ means a private attribute not to be accessed outside␣
 ↪class

    # maintain invariant when on variable get
    @property
    def x(self):
        return max(min(self.xmax, self.__x), self.xmin)

a = Bounded(231)
print("a.x:", a.x)
#a.x = -33
```

[ ]:
```
class Bounded():
    """ Keep track of variable x, but clipped to xmin and xmax """
    xmin, xmax = 50, 100
    def __init__(self, x):
        self.x = x   # calls x.setter

    @property
    def x(self):
        return self.__x

    # maintain invariant on variable set
    @x.setter
    def x(self, val):
        self.__x = max(min(self.xmax, val), self.xmin)

a = Bounded(231)
print("a.x:", a.x)
a.x = -33
print("a.x:", a.x)
```