Advanced Object Oriented Programming

In the regular section on Object Oriented Programming (OOP) we covered:

- Using the *class* keyword to define object classes
- Creating class attributes
- Creating class methods
- Inheritance - where derived classes can inherit attributes and methods from a base class
- Polymorphism - where different object classes that share the same method can be called from the same place
- Special Methods for classes like `__init__`, `__str__`, `__len__` and `__del__`

In this section we'll dive deeper into

- Multiple Inheritance
- The `self` keyword
- Method Resolution Order (MRO)
- Python's built-in `super()` function

Inheritance Revisited

Recall that with Inheritance, one or more derived classes can inherit attributes and methods from a base class. This reduces duplication, and means that any changes made to the base class will automatically translate to derived classes. As a review:

In [1]:

```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):
    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):
    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

In this example, the derived classes did not need their own `__init__` methods because the base class `__init__` gets called automatically. However, if you do define an `__init__` in the derived class, this will override the base:

In [2]:

```
class Animal:
    def __init__(self,name,legs):
        self.name = name
        self.legs = legs

class Bear(Animal):
    def __init__(self,name,legs=4,hibernate='yes'):
        self.name = name
        self.legs = legs
        self.hibernate = hibernate
```

This is inefficient - why inherit from Animal if we can't use its constructor? The answer is to call the Animal `__init__` inside our own `__init__` .

In [3]:

```
class Animal:
    def __init__(self,name,legs):
        self.name = name
        self.legs = legs

class Bear(Animal):
    def __init__(self,name,legs=4,hibernate='yes'):
        Animal.__init__(self,name,legs)
        self.hibernate = hibernate

yogi = Bear('Yogi')
print(yogi.name)
print(yogi.legs)
print(yogi.hibernate)
```

```
Yogi
4
yes
```

Multiple Inheritance

Sometimes it makes sense for a derived class to inherit qualities from two or more base classes. Python allows for this with multiple inheritance.

In [4]:

```
class Car:
    def __init__(self,wheels=4):
        self.wheels = wheels
        # We'll say that all cars, no matter their engine, have four wheels by d
efault.

class Gasoline(Car):
    def __init__(self,engine='Gasoline',tank_cap=20):
        Car.__init__(self)
        self.engine = engine
        self.tank_cap = tank_cap # represents fuel tank capacity in gallons
        self.tank = 0

    def refuel(self):
        self.tank = self.tank_cap


class Electric(Car):
    def __init__(self,engine='Electric',kWh_cap=60):
        Car.__init__(self)
        self.engine = engine
        self.kWh_cap = kWh_cap # represents battery capacity in kilowatt-hours
        self.kWh = 0

    def recharge(self):
        self.kWh = self.kWh_cap
```

So what happens if we have an object that shares properties of both Gasolines and Electrics? We can create a derived class that inherits from both!

In [5]:

```
class Hybrid(Gasoline, Electric):
    def __init__(self,engine='Hybrid',tank_cap=11,kWh_cap=5):
        Gasoline.__init__(self,engine,tank_cap)
        Electric.__init__(self,engine,kWh_cap)


prius = Hybrid()
print(prius.tank)
print(prius.kWh)
```

```
0
0
```

In [6]:

```
prius.recharge()
print(prius.kWh)
```

```
5
```

Why do we use `self`?

We've seen the word "self" show up in almost every example. What's the deal? The answer is, Python uses `self` to find the right set of attributes and methods to apply to an object. When we say:

```
prius.recharge()
```

What really happens is that Python first looks up the class belonging to `prius` (Hybrid), and then passes `prius` to the `Hybrid.recharge()` method.

It's the same as running:

```
Hybrid.recharge(prius)
```

but shorter and more intuitive!

Method Resolution Order (MRO)

Things get complicated when you have several base classes and levels of inheritance. This is resolved using Method Resolution Order - a formal plan that Python follows when running object methods.

To illustrate, if classes B and C each derive from A, and class D derives from both B and C, which class is "first in line" when a method is called on D?
Consider the following:

In [7]:

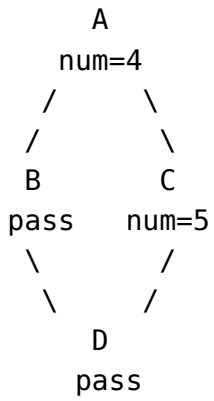
```
class A:
    num = 4

class B(A):
    pass

class C(A):
    num = 5

class D(B,C):
    pass
```

Schematically, the relationship looks like this:



Here `num` is a class attribute belonging to all four classes. So what happens if we call `D.num` ?

In [8]:

```
D.num
```

Out[8]:

```
5
```

You would think that `D.num` would follow `B` up to `A` and return **4**. Instead, Python obeys the first method in the chain that *defines* num. The order followed is `[D, B, C, A, object]` where *object* is Python's base object class.

In our example, the first class to define and/or override a previously defined `num` is `C` .

super()

Python's built-in `super()` function provides a shortcut for calling base classes, because it automatically follows Method Resolution Order.

In its simplest form with single inheritance, `super()` can be used in place of the base class name :

In [9]:

```
class MyBaseClass:
    def __init__(self,x,y):
        self.x = x
        self.y = y

class MyDerivedClass(MyBaseClass):
    def __init__(self,x,y,z):
        super().__init__(x,y)
        self.z = z
```

Note that we don't pass `self` to `super().__init__()` as `super()` handles this automatically.

In a more dynamic form, with multiple inheritance like the "diamond diagram" shown above, `super()` can be used to properly manage method definitions:

In [10]:

```
class A:
    def truth(self):
        return 'All numbers are even'

class B(A):
    pass

class C(A):
    def truth(self):
        return 'Some numbers are even'
```

In [11]:

```
class D(B,C):
    def truth(self,num):
        if num%2 == 0:
            return A.truth(self)
        else:
            return super().truth()

d = D()
d.truth(6)
```

Out[11]:

```
'All numbers are even'
```

In [12]:

```
d.truth(5)
```

Out[12]:

```
'Some numbers are even'
```

In the above example, if we pass an even number to `d.truth()`, we'll believe the `A` version of `.truth()` and run with it. Otherwise, follow the MRO and return the more general case.

For more information on `super()` visit https://docs.python.org/3/library/functions.html#super (https://docs.python.org/3/library/functions.html#super)
and https://rhettinger.wordpress.com/2011/05/26/super-considered-super/ (https://rhettinger.wordpress.com/2011/05/26/super-considered-super/)

Great! Now you should have a much deeper understanding of Object Oriented Programming!