

Object-Oriented Code Concepts

Demitri Muna
OSU

1 August 2016

What is an Object?

Consider an array in IDL:

```
spectrum = [10, 15, 5, 35, 39, 41, 36, 30]
```

This is literally nothing more than a block of memory containing these values, a list of numbers. The array doesn't know how to do anything; any operation must be performed by an external function.

```
IDL> print, total(spectrum)
      211.000
IDL> print, mean(spectrum)
      26.3750
```

Any meaning is up to you to keep track of – basically only through the variable's name. While to you it might represent masses, velocities, magnitudes, observations... no array is really any different from any other.

What is an Object?

Consider a Numpy array in Python:

```
import numpy as np
spectrum = np.array([10, 15, 5, 35, 39, 41, 36, 30])
```

This is more than a list of numbers – the variable `spectrum` “knows” how to perform certain tasks by accessing methods (functions) or properties with the “dot” syntax:

```
>>> spectrum.size
```

```
8
```

```
>>> spectrum.max()
```

```
41
```

```
>>> spectrum.mean()
```

```
26.375
```

```
>>> spectrum.sum()
```

```
211
```

← knows how many elements are in the array

← can find the maximum value

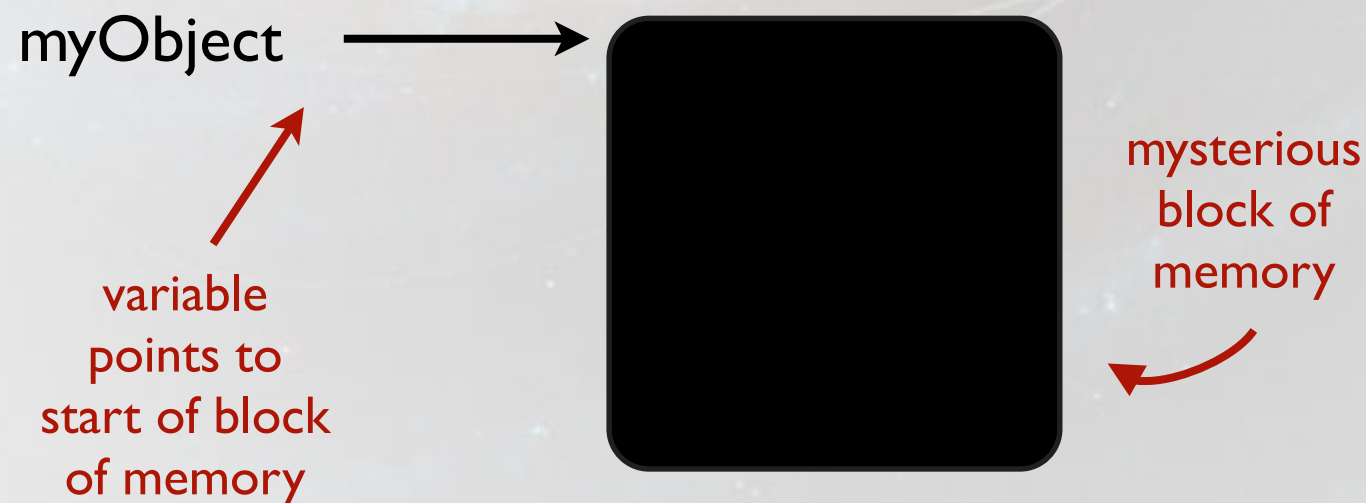
← can calculate the mean

← can calculate the sum

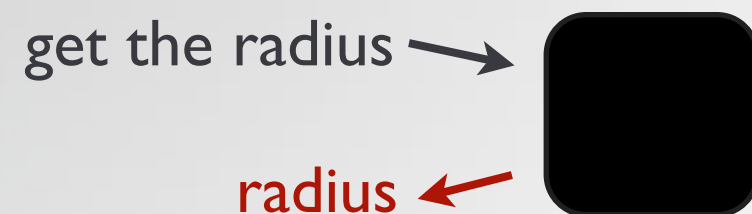
What is an Object?

An *object* is a structure that contains the data *and* knows how to perform common operations on them without requiring external functions.

An object is defined to represent an idea that you are trying to model, grouping together related data and functionality into a single unit. This idea called *encapsulation*, which is the heart of object-oriented programming.



The object is a black box – you do not know *and should not know* how the internals are organized. Values are named, and set and retrieved through *accessors*.



How this value is determined or where it is in memory is opaque to you.

Classes

A *class* is the definition of an object. It is where the variables (data) and functions associated with the model are defined.

This is how to define a class in Python.

The function `__init__` is a special one that is automatically called when a new object is created. We use this to automatically define (and initialize) variables within the object.

creates a new object →
`__init__()` is implicitly called,
declaring (and initializing) variables

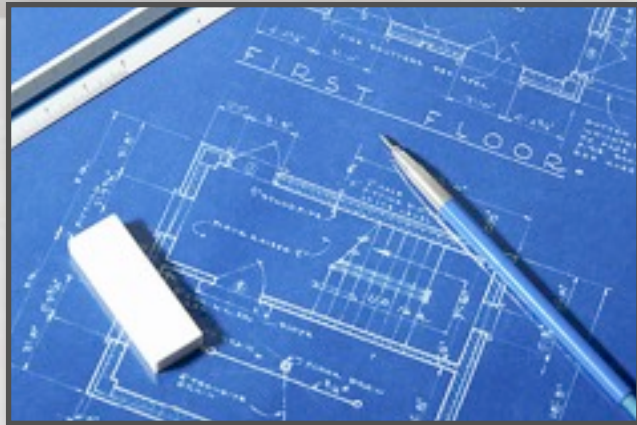
```
class Rectangle(object):  
    def __init__(self):  
        self.height = 0  
        self.width = 0  
        self.color = "burnt sienna"
```

```
class Square(object):  
    def __init__(self):  
        self.height = 0  
        self.width = 0  
        self.color = "aquamarine"
```

```
class Circle(object):  
    def __init__(self):  
        self.radius = 0  
        self.color = "periwinkle"
```

```
r = Rectangle()  
s = Square()  
c = Circle()  
  
print(c.radius)
```

Class vs Object



Class

- *Definition* of an object (the blueprint).
- Does not allocate any memory when defined.
- Rule of thumb: define a class for nouns, e.g. 'detector', 'particle', 'star', 'observation'.

Class

```
class Rectangle(object):  
    def __init__(self):  
        self.height = 0  
        self.width = 0  
        self.color = "blue"
```

Object

```
r = Rectangle()
```



Object

- An instance of the blueprint (a built house).
- Allocates memory when created.
- Can create as many as memory allows.

Methods

To perform calculations on data, typically you define a function:

```
float calculate_rectangle_area(float height, float width)
{
    return height * width;
}
```

This becomes messy...

```
calculate_square_area(s.height)
calculate_circle_area(c.radius)
calculate_triangle_area(tri.width, tri.height)
calculate_tetrahedron_area(t.edge_length)
calculate_dodecahedron_area(d.edge_length)
...
```

Each function name is different, the function parameters are different...

Methods

Here, we “ask” the Circle for its area – we are not concerned with the details outside of the class definition. The Circle should know how to calculate its own area.

These functions that are part of the class definition are known as *methods*.

The object itself should know how to perform certain calculations, and no other code should be aware of (or *need* to know) the implementation details. All of the “knowledge” (data, methods) of the object is contained in the class. This concept is known as *encapsulation*.

Note the parentheses, needed as a method is being called.

```
class Rectangle(object):
    def __init__(self):
        self.height = 0
        self.width = 0
        self.color = "burnt sienna"

    def area(self):
        return self.height * self.width

class Square(object):
    def __init__(self):
        self.height = 0
        self.color = "aquamarine"

    def area(self):
        return self.height * self.height

class Circle(object):
    def __init__(self):
        self.radius = 0
        self.color = "periwinkle"

    def area(self):
        return math.pi * self.radius * self.radius

c = Circle()
c.radius = 5
print(c.area()) # Output: 78.5398163397
```

these are called
properties of the class

method

Methods

Note the common features - can we take advantage of this?

Create a new class that contains as many common features as possible. New classes will automatically “inherit” everything from this common class.

superclass



```
class Shape(object):
    def __init__(self):
        self.color = "black"
    def area(self):
        pass
```

This is an *abstract* class, meaning you would never directly create it.

```
class Rectangle(object):
    def __init__(self):
        self.height = 0
        self.width = 0
        self.color = "burnt sienna"

    def area(self):
        return self.height * self.width

class Square(object):
    def __init__(self):
        self.height = 0
        self.color = "aquamarine"

    def area(self):
        return self.height * self.height

class Circle(object):
    def __init__(self):
        self.radius = 0
        self.color = "periwinkle"

    def area(self):
        return math.pi * self.radius * self.radius

c = Circle()
c.radius = 5
print(c.area()) # Output: 78.5398163397
```

Subclasses

```
class Rectangle(object):
    def __init__(self):
        self.height = 0
        self.width = 0
        self.color = "burnt sienna"

    def area(self):
        return self.height * self.width

class Square(object):
    def __init__(self):
        self.height = 0
        self.color = "aquamarine"

    def area(self):
        return self.height * self.height

class Circle(object):
    def __init__(self):
        self.radius = 0
        self.color = "periwinkle"

    def area(self):
        return math.pi * self.radius * self.radius

c = Circle()
c.radius = 5
print(c.area()) # Output: 78.5398163397
```

```
class Point(object):
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
```

```
class Shape(object):
    def __init__(self):
        self.color = "black"
        self.origin = Point()
    def area(self):
        pass
```

abstract superclass

```
class Square(Shape):
    def __init__(self):
        super(Square, self).__init__()
        self.height = 0
```

Note! Python 2.x inheritance!

```
    def area(self):
        return self.height * self.height

class Rectangle(Square):
    def __init__(self):
        super(Rectangle, self).__init__()
        self.width = 0.0

    def area(self):
        return self.height * self.width
```

Rectangle inherits from Square (i.e. is a subclass of Shape)

```
class Circle(Shape):
    def __init__(self):
        super(Circle, self).__init__()
        self.radius = 0.0

    def area(self):
        return math.pi * self.radius * self.radius
```

Subclasses

We need to address “custom” behavior (or exceptions) and possible problems.

What if you forget to define “area()” for an object?

```
class Point(object):
    def __init__(self):
        self.x = 0.0
        self.y = 0.0

class Shape(object):
    def __init__(self):
        self.color = "black"
        self.origin = Point()
    def area(self):
        pass

class Square(Shape):
    def __init__(self):
        super(Square, self).__init__()
        self.height = 0.0

    def area(self):
        return self.height * self.height

class Rectangle(Square):
    def __init__(self):
        super(Rectangle, self).__init__()
        self.width = 0.0

    def area(self):
        return self.height * self.width

class Circle(Shape):
    def __init__(self):
        super(Circle, self).__init__()
        self.radius = 0.0

    def area(self):
        return math.pi * self.radius * self.radius
```

Subclasses

Although we can't define every possible thing in **Shape**, we can require that certain things be defined to be valid.

This way, we can look at the **Shape** class and know that any class that is derived from it will have an area method without having to make sure ourselves.

(We're starting to get more into the Python dialect, but all concepts here are applicable to any other object oriented language, e.g. C++. Only the syntax changes.)

```
from abc import ABCMeta
from abc import abstractproperty
import math
```

ABC = Abstract Base Class
(Python 2.6+)

```
class Shape(object):
    __metaclass__ = ABCMeta
```

magic that tells Python that
this is an abstract class

```
    def __init__(self):
        self.color = "black"
```

```
    @abstractproperty
    def area(self):
        pass
```

magic that tells Python that
any subclass of Shape *must*
define an "area" method

```
class Circle(Shape):
    def __init__(self):
        super(Circle, self).__init__()
        self.radius = 0.0
```

```
c = Circle()
print(c.area())
```

This error points out that "area"
was not defined - note the error
was thrown when the object was
created, not when area() was
called.

```
# Output:
# Traceback (most recent call last):
#   File "untitled text 33", line 23, in <module>
#     c = Circle()
# TypeError: Can't instantiate abstract class Circle with
#         abstract methods area
```


Abstract Classes, 2.6+ vs 3.x

This is one of the incompatible differences between Python 2 and Python 3.

Python 2

```
from abc import ABCMeta

class MyAbstractClass(object):
    __metaclass__ = ABCMeta

    def __init__(self):
        ...
```

Python 3

```
from abc import ABCMeta

class MyAbstractClass(metaclass=ABCMeta):

    def __init__(self):
        ...
```

Calling Methods From the Superclass

The **Shape** class defines and initializes values in `__init__()`. The same initializer in **Square** doesn't automatically call `__init__()` in **Shape**; we have to call it explicitly (meaning we have the option not to).

We call **Shape**'s (the superclass) method like this, i.e. get the superclass of the **Square** object and call its `__init__()` method. (Yes, the syntax is awkward.)

```
class Shape(object):
    def __init__(self):
        self.color = "black"
        self.origin = Point()

class Square(Shape):
    def __init__(self):
        super(Square, self).__init__()
        self.height = 0.0

    def area(self):
        return self.height * self.height
```

Python 2 & Python 3

```
class Square(Shape):
    def __init__(self):
        super(Square, self).__init__()
        self.height = 0.0
```

Python 3

```
class Square(Shape):
    def __init__(self):
        super().__init__()
        self.height = 0.0
```

Cleaner syntax, incompatible with Python 2.

Subclasses

Note the special keyword “self”. If you are inside a class definition, this word refers to the object itself.

Use

`self.property`
to refer properties that belong to the class.

Methods always have `self` as the first parameter since Python always passes the object itself as the first term.

```
from abc import ABCMeta
from abc import abstractproperty
import math
```

```
class Shape(object):
    __metaclass__ = ABCMeta

    def __init__(self):
        self.color = "black"

    @abstractproperty
    def area(self):
        pass
```

```
class Circle(Shape):
    def __init__(self):
        super(Circle, self).__init__()
        self.radius = 0.0
```

```
c = Circle()
print(c.area())
```

```
# Output:
# Traceback (most recent call last):
#   File "untitled text 33", line 23, in <module>
#     c = Circle()
# TypeError: Can't instantiate abstract class Circle with
#         abstract methods area
```

Note the “object” superclass -- this is a special Python class. Use it as the superclass of any object you don't have a superclass for.

The first parameter of any class method is “self”. This is a special keyword that allows you to refer to the object itself.

OO Terminology

Instantiation

Act of creating an object from a class.

```
c = Circle()
```

Instance

An object created from a class (above “e” is the instance).

Attribute (or Property, or Instance Variable)

A variable defined inside a class (or object).

```
class Rectangle(object):  
    def __init__(self):  
        self.height = 0  
        self.width = 0  
        self.color = "burnt sienna"
```

← 3 attributes

Method

A function defined inside a class (or object).

Encapsulation

The implementation details of the class should be hidden from any code that accesses or uses a given object. Data and functions are merged together.



Abstract Class

A class that is designed to be subclassed and cannot be instantiated itself. It defines methods and properties common to many classes.

Subclass

A class that is defined to have all of the methods and properties of another class, but adds its own methods and/or properties (a specialization of the superclass).

Superclass

A class that is used as the base definition of another class.

Comparison to Other Languages

Python

- All methods, properties are public. Python philosophy - “we’re all adults here.” Don’t modify properties that are described as private.
- Getter/setter methods automatically generated.
- Garbage collection (i.e. computer throws things away when you’re done with them).
- EVERYTHING in Python is an object: strings, numbers, arrays, etc. Everything. This is good... but it can bite you.

C++

- public/protected/private strictly enforced.
- Must write getter/setter methods for every property.
- You must manage memory yourself -- delete everything you create.
- You define your own classes (or use them from external libraries). Data types such as `ints`, `floats`, and arrays, are not objects.

Similar, but Python requires far fewer lines of code and is cleaner.

```
Square s = Square()  
s.height = 10.0  
print(s.area())
```

Python generated the method that sets “height”.

You wrote this. As if you have nothing better to do.

```
Square *s = new Square();  
s->SetHeight(10.0);  
s->Area();  
delete s;
```

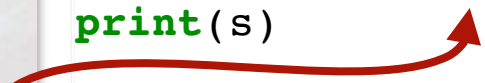
Mutable vs Immutable

Objects are either *mutable* or *immutable*:

- Mutable: the object can be changed or modified. Examples: dictionaries, lists (arrays).
- Immutable: the value of the object, once defined, cannot be changed.

Consider this code:

```
a = "A string."  
s = a + " A second string."  
print(s)  
  
# Output  
# A string. A second string.
```



Although not assigned to a variable, this is also an immutable string object.

A string 's' is defined (and since strings are immutable, it cannot be changed). The string is then "added" (concatenated) to a second immutable string, creating a third object. The first two are thrown away.

Typically, you don't need to worry about this (no premature optimization!). But there are cases where being aware of this can improve the speed of your code, e.g. large simulations, iterating over a large number of objects.

Rule of thumb: try to minimize the number of objects you create inside a large loop.

Mutable vs Immutable

```
import time
```

```
# Method 1
```

```
# -----
```

```
start = time.time()
```

```
long_string = ''
```

```
for i in range(100000):  
    long_string += str(i)
```

```
end = time.time()
```

```
elapsed = end - start
```

```
print elapsed, "seconds"
```

```
# Method 2
```

```
# -----
```

```
start = time.time()
```

```
string_list = list()
```

```
for i in range(100000):  
    string_list.append(str(i))  
long_string = ''.join(string_list)
```

```
print time.time() - start, "seconds"
```

```
# Method 3
```

```
# -----
```

```
start = time.time()
```

```
long_string = ''.join([str(x) for x in  
range(100000)])
```

```
print time.time() - start, "seconds"
```

```
# Output:
```

```
# 0.0746450424194 seconds
```

```
# 0.07293009758 seconds
```

```
# 0.056009054184 seconds <-- 25% faster
```

strings are
concatenated, creating
two new objects each
for iteration

strings are added to a
mutable list, then
concatenated all at
once

same, but with list
comprehension!

What this shows is that
unless your code is
running for *hours*, this
won't bite you!

The point is that there
is an overhead to
creating objects, so do
your best to avoid
doing that in time-
critical parts of your
code. Otherwise, go
nuts.

```
c = Circle()  
c.radius = 10.0
```

```
start = time.time()
```

```
for i in range(10000):  
    p = Point()  
    p.x = 1  
    p.y =  
    x.point = p
```

```
elapsed = time.time() - start
```

```
print elapsed
```

```
start = time.time()
```

```
for i in range(10000):  
    c.point.x = i  
    c.point.y = i
```

```
elapsed = time.time() - start
```

```
print elapsed
```

```
# Output
```

```
# 0.0100679397583
```

```
# 0.00519800186157
```

Rather than create a
new Point, just access
the existing object

Initializing the Object

We can require that certain information be provided before an object is created. This also provides a shorter form to create an object.

If default values are specified, then all, some, or none of the parameters can be specified. Anything not given defaults to the value given in `__init__`.

The preferred method is to name the parameters (then, order doesn't matter):

```
p = Point(x=4, y=2)
```

I strongly encourage this for better readability!!

x, y now required

```
class Point(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
p = Point(4, 2)  
print p.x # 4
```

can perform any initializations needed here

set x, y at object creation

```
class Point(object):  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
  
p = Point()  
print p.x # 0  
  
p = Point(4) # same as Point(x=4)  
print p.x, p.y # 4, 0  
  
p = Point(y=2) # can set any value by name  
print p.x, p.y # 0.0, 2
```

specifying x, y now optional - default values provided

Static Methods

So we want to create an object for everything. What if we need a very simple function? Doesn't this just create more work?

Let's say we want to convert hours to radians. Here, the task doesn't really refer to a "noun" – it's more of a utility.

By defining a method as *static*, it means that you don't have to first create an object to use it – you can use it directly.

Caveat: the method must stand alone, i.e. it can't use any properties of the class (because we're not creating that class).

```
import math
class ConvertHoursToRad(object):
    """Convert hours (1 hr = 15 radians)
    to radians."""
```

```
# no properties needed
```

```
def convert(self, hours):
    return hours * math.pi/12.0
```

```
h2r_object = ConvertHoursToRad()
h2r_object.convert(14.5)
```

create object for
one-off calculation,
then perform task

```
class Convert(object):
    def hours2radians(self, hours):
        return hours * math.pi/12.0
    hours2radians = staticmethod(hours2radians)
    can add many more utility methods...
radians = Convert.hours2radians(14.5)
```

super

Let's say we want to create a class to represent a quarter* using Circle as the the super class. Here we'll define the area as twice that of a circle.

We have to implement a new `area` method of course. We could copy the code from the Circle method, but that's not a good idea. We want to return $2 \cdot \text{area}$, but what if the definition of area ever changes?** We'd have to change the same code in two places. We could forget. Or we may not have control over the subclass. Or know how Circle calculates its area.

Instead, we can get access to the superclass's method through the keyword *super*, then use the result of the superclass' method to calculate our own result.

```
class Quarter(Circle):  
  
    def area(self):  
        return 2.0 * math.pi * \  
            self.radius * self.radius
```

This copies the same code from circle –
if it changes, then you have to change it
in two places!!

```
class Quarter(Circle):  
  
    def area(self):  
        return 2.0 * super.area( )
```

returns the value from
Circle's (the superclass)
method

* No, I don't know why either.

** Yes, I know it won't in this particular example.

What Should Be A Class?

Consider your (mental) model of the data rather than arrays of numbers or single values. A class and its properties might look like this:

```
g = Galaxy()  
g.redshift  
g.morphology  
g.mass
```

```
t = Telescope()  
t.name  
t.altitude  
t.mirror_diameter
```

```
s = Star()  
s.mass  
s.T_eff  
s.surface_gravity
```

The returned values might be calculated, or else they might be returned from the initialized values.

Passing Around Data

Typically in your applications, you'll pass around disparate data – an image, coordinates, arrays, references to files, etc. In a C-style program (I'm looking at you IDL), this is a hassle and tends to involve functions with many parameters:

```
process_my_observation(*image, data[], filename1, filename2, ra, dec, radius, ...)
```

If you group things together logically, then your code should be much easier to handle, be easier to write, and far easier for other people to read:

```
process_my_observation(observation):  
    new_ra = observation.ra + 10  
    etc.
```

All of the information that is related to that object (something that easily maps to your data or logic) is contained in (or can be accessed from) that object. Maybe `process_my_observation` doesn't need to use `dec`, but if it does later, you don't have to change the calling method - you already have everything you need.

Singletons

There are times where it would only make sense to create a single instance of a class. For example, if you were working with data from an observatory, there would only be one instance of `Telescope()`, or `AtlasDetector()`. In this case, when we call the constructor:

`t = Telescope()` # first time called – create the object

`t = Telescope()` # subsequent times – return the *same* object that was first created.

This is a singleton.

Python

override the method that
actually creates the instance

The class is an object itself! We
then save the first instance in a
dictionary in the class

and return it each time thereafter

Normally, this would create two objects,
but we see here they are the same.

```
class Singleton(object):
    _singletons = dict()
    def __new__(cls, *args, **kwargs):
        if not cls._singletons.has_key(cls):
            cls._singletons[cls] = object.__new__(cls)
        return cls._singletons[cls]
```

```
s = Singleton()
s2 = Singleton()
print s
print s2

# Output:
# <__main__.Singleton object at 0x1d3970>
# <__main__.Singleton object at 0x1d3970>
```

Singletons

I've found this useful...

```
#!/usr/bin/python

'''
This function implements the singleton design pattern in Python.
To use it, simply import this file:

from singleton import singleton

and declare your class as such:

@singleton
class A(object):
    pass
'''

def singleton(cls):
    instance_container = []
    def getinstance():
        if not len(instance_container):
            instance_container.append(cls())
        return instance_container[0]
    return getinstance
```

Properties

Consider the width of a **Shape**. This is easy for the **Rectangle** shape: it's already a property.

```
r = Rectangle()
print(r.width)
```

Circle doesn't have a width property, but we can easily calculate it. It would be nice to be able to ask any **Shape** object for its width. Easy enough.

```
class Circle(object):
    ...
    def width(self):
        return self.radius * 2.0
```

But in **Rectangle** width is a property and in **Circle** it's a method, so they must be called like this. One shouldn't have to remember that. (Also, no parameter will ever be passed to **width()**).

```
r = Rectangle()
c = Circle()
r.width
c.width()
```

Even though it's a method, we can declare that width should be treated as a property with the **@property** decorator.

```
class Circle(object):
    ...
    @property
    def width(self):
        return self.radius * 2.0
```

Now this works:

```
r.width
c.width
```

Derived Properties

We can see that some properties are calculated, i.e. really methods, not stored values. What if I want to set such a “property”? For example, this should seem reasonable to a user:

```
c = Circle()
c.width = 12.0
```

But there is no width value to set! We can define what that means:

```
class Circle(object):
    ...
    @property
    def width(self):
        return self.radius * 2.0

    @width.setter
    def width(self, new_width):
        self.radius = new_width / 2.0
```

first define the method, declared as a property

Note the methods have the same name! The decorators distinguish them.

the decorator is the property name followed by “.setter”

make sure to accept “self” and the new value

changing the width actually changes the stored radius value – the value “width” here is a *derived value*

Now the code above works. This method is called a *setter*; the first one is called a *getter*. Do not name these methods `getWidth` or `setWidth`. (People will laugh at you; children will cry.)

Note that the user does not need to know *how* the width is stored in the object – it just works as expected. This is key: the implementation details are hidden from the user.

Dependent Properties

Sometimes, the value of one property can depend on another. If this is the case, when you update one, you want the other to be updated automatically. Let's take the **Circle** class as an example. We want the circle to turn blue when the radius is less than 10, and red when it's equal to or greater than 10.

```
class Circle(object):  
  
    def __init__(self):  
        self.radius = 10.0  
        self.color = "red"
```

Radius here is an instance (which is accessed as a property) – we need a “hook” so we know when it changes. But we don't have a setter or getter since Python provides one transparently.

Dependent Properties

```
class Circle(object):  
  
    def __init__(self):  
        self._radius = 10.0  
        self.color = "red"  
  
    @property  
    def radius(self):  
        return self._radius  
  
    @radius.setter  
    def radius(self, new_radius):  
        self._radius = new_radius  
        if new_radius < 10:  
            self.color = "blue"  
        else:  
            self.color = "red"
```

We create a *private* instance to store the actual value, but then declare the property with two methods. We use the same name but add a leading underscore to the private instance. This isn't required, but the naming style informs others of what we are doing. (This is a common pattern.)

To make the property read-only, we can omit the setter.

Before:
radius is a property.

```
c = Circle()  
c.radius = 10  
print(c.radius)
```

After:
radius is a method.

```
c = Circle()  
c.radius = 10  
print(c.radius)
```

Lazy Loading

Some properties may take a long time to initialize. For example, one might need to read a file, parse its contents, and perform some calculations to determine the value. This can be time consuming, and there's no guarantee that the person who created the object may even use the property. We again create a private instance to handle this:

```
class Thingy(object):  
  
    def __init__(self):  
        self._complexValue = None  
  
    @property  
    def complexValue(self):  
        if self._complexValue is None:  
            # do expensive thing  
            self._complexValue = <result>  
        return self._complexValue
```

private instance set to None; this indicates we haven't defined it yet

If new, create it. Only happens once (though can be reset and recalculated as needed).

Since the "expensive" part is outside of the initializer, the object can be created much more quickly.

Naming Conventions – Naming Styles

There are several styles that can be used to create names of classes or variables:

<code>soylent_green</code>	<i># words separated by underscores</i>
<code>camelCase</code>	<i># no spaces between words, each word capitalized</i>
<code>ALL_UPPERCASE</code>	<i># use for constants</i>
<code>_leading_underscore</code>	<i># use to indicate private instances (variables)</i>

Which to use? Variable names should *always* start with lower case letters. Python in general names:

- variables : words separated with underscores
- class names : first letter capitalized, camel case
- method names : words separated with underscores

The most important things are to a) be consistent and b) match the coding style of code you are working on.

Naming Conventions – Demitri Style

This is my (Python) style:

```
class Thingy(object):
```

```
    def __init__(self):  
        self._complexValue = None
```

```
@property
```

```
def complexValue(self):  
    if self._complexValue is None:  
        # do expensive thing  
        self._complexValue = <result>  
    return self._complexValue
```

```
a_thingy = Thingy()
```

```
a_thingy.doAThing(param_one=7, param_two=42)
```

```
some_function(a_thingy)
```

private instances are camel case

method names are camel case

variable names are underscore separated

parameter names are
underscore separated

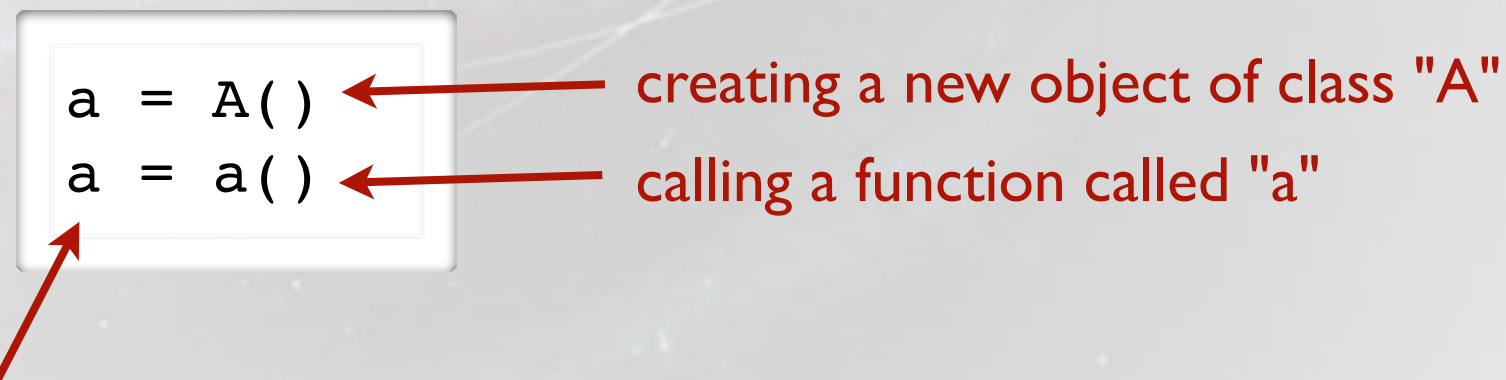
function names are underscore separated

The naming style suggests what the item is.

Naming Conventions – Class Names

Python doesn't enforce naming conventions (most or no languages do), however each language has its own style and people do stick to. Some things should be considered strict, and others are a little more flexible.

Class names should always begin with a capital letter; this signals the user that something is a class:



The diagram shows a white box containing two lines of code: `a = A()` and `a = a()`. A red arrow points from the text "creating a new object of class 'A'" to the `A()` part of the first line. Another red arrow points from the text "calling a function called 'a'" to the `a()` part of the second line. A third red arrow points from the text "starts with a lowercase letter: this is a variable" to the lowercase `a` in the first line.

```
a = A() ← creating a new object of class "A"
```

```
a = a() ← calling a function called "a"
```

starts with a lowercase letter: this is a variable

You *can* create a class whose name starts with a lowercase letter; don't.

Class names should be singular; each object represents a single item.

strict!

Documentation

```
class Sprocket(object):  
    '''  
    This class implements a sprocket.  
    Sprockets are of German origin.  
    '''  
  
    def __init__(self):  
        self._complexValue = None  
  
    @property  
    def complexValue(self):  
        '''  
        The value of the complex thing.  
        '''  
        if self._complexValue is None:  
            # do expensive thing  
            self._complexValue = <result>  
        return self._complexValue  
  
a_thingy = Thingy()  
a_thingy.doAThing(param_one=7, param_two=42)  
  
some_function(a_thingy)
```

Triple quoted (i.e. multi-line) string immediately under class definition.

One-line description, followed by as much description as needed. The one liner is used in iPython as documentation.

Another triple-quoted string underneath the method declaration.

Python uses *restructured text* (reST) to define parameters, return types, etc. This is very similar to Markdown and is used to generate full documentation from the code. A full description is beyond the scope of this lecture; google "python restructured text" for more information and usage.

Comments

Comments should appear in your code to explain to the reader the intent and/or overview of the code.

```
class Sprocket(object):  
  
    def __init__(self, thing1=False, thing2=12):  
        # both things must be present  
        # for an object to be valid, raise an  
        # exception otherwise  
        if not all(thing1, thing2):  
            raise Exception("Missing a thing!")
```

Avoid being pedantic, e.g.

```
class Sprocket(object):  
  
    def __init__(self):  
        ''' This is the init method.'''  
        # set the initial value for area  
        self.area = 10  
  
s = Sprocket() # create a new Sprocket
```

While the intent or plan of the code is obvious as you write it, it won't be six months from now (or necessarily to anyone else). Write the comments for someone who is reading the code for the first time – don't make someone guess what you are trying to do and how you are going about it.

Exercise

Identify parts of the scripts you've written that would be appropriate as classes and write them. Create a new file for each class. Look for opportunities to put common functionality in a superclass. Use properties to access data, use methods to perform calculations (i.e. return a value) or modify the data in the object.

Modify your script to use the class you created. If you defined a class in a file called `mynewclass.py`, put it in the same directory as your script. You can then use it like this:

mynewclass.py ← name of file

```
#!/usr/bin/env python

class MyNewClass(object):
    # class definition here...
```

my_script.py

name of class in file

```
from mynewclass import MyNewClass

mnc = MyNewClass()
```

Exercise

Consider this code snippet:

```
p1 = Point(x=12, y=4)
p2 = Point(x=3, y=32)

print (p1.distance_to(p2))
```

Write a Python script and a "Point" class that makes the above code work.