Object-Oriented Code Concepts

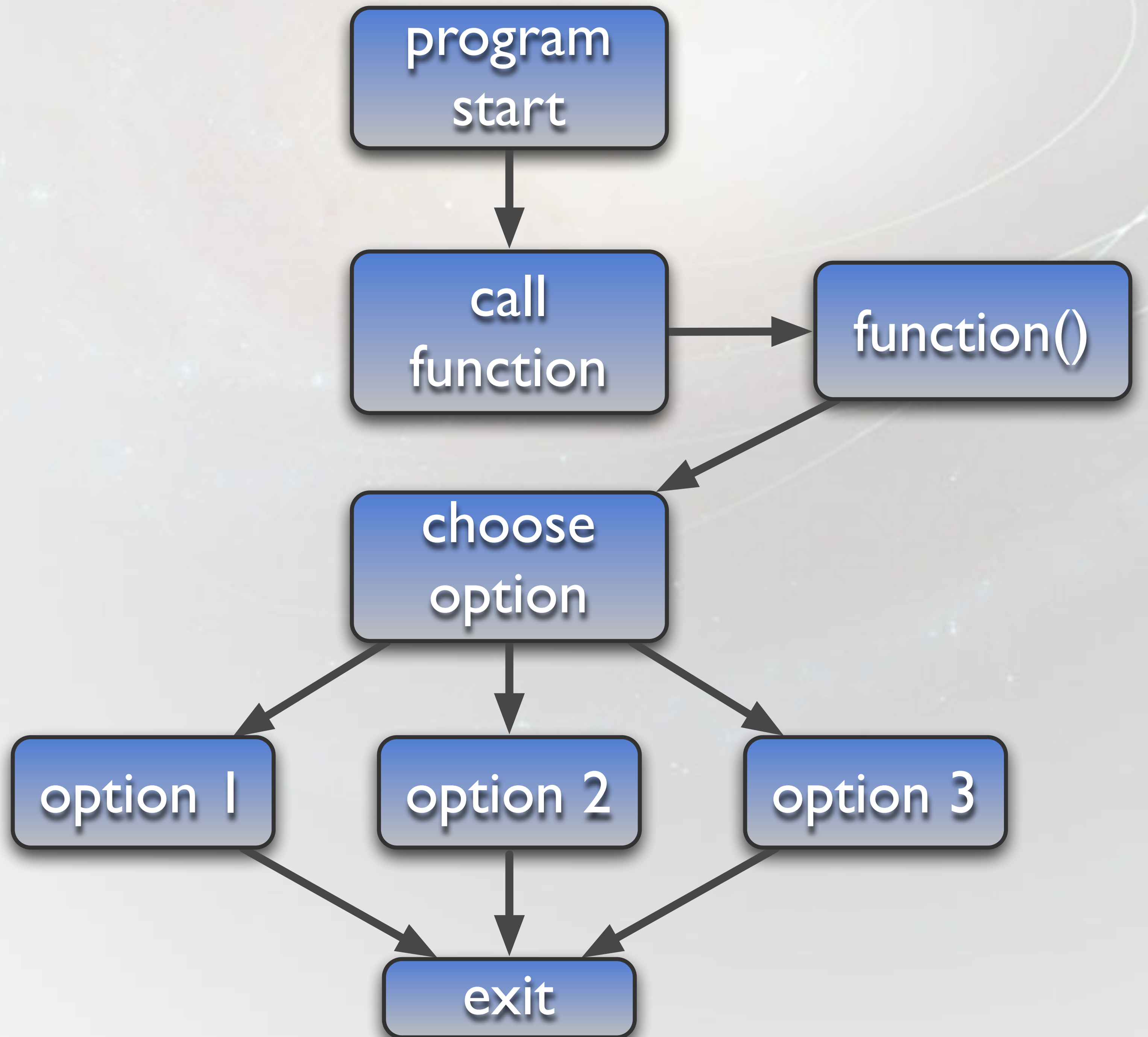
Demitri Muna
OSU

23 June 2014

Procedural Programming

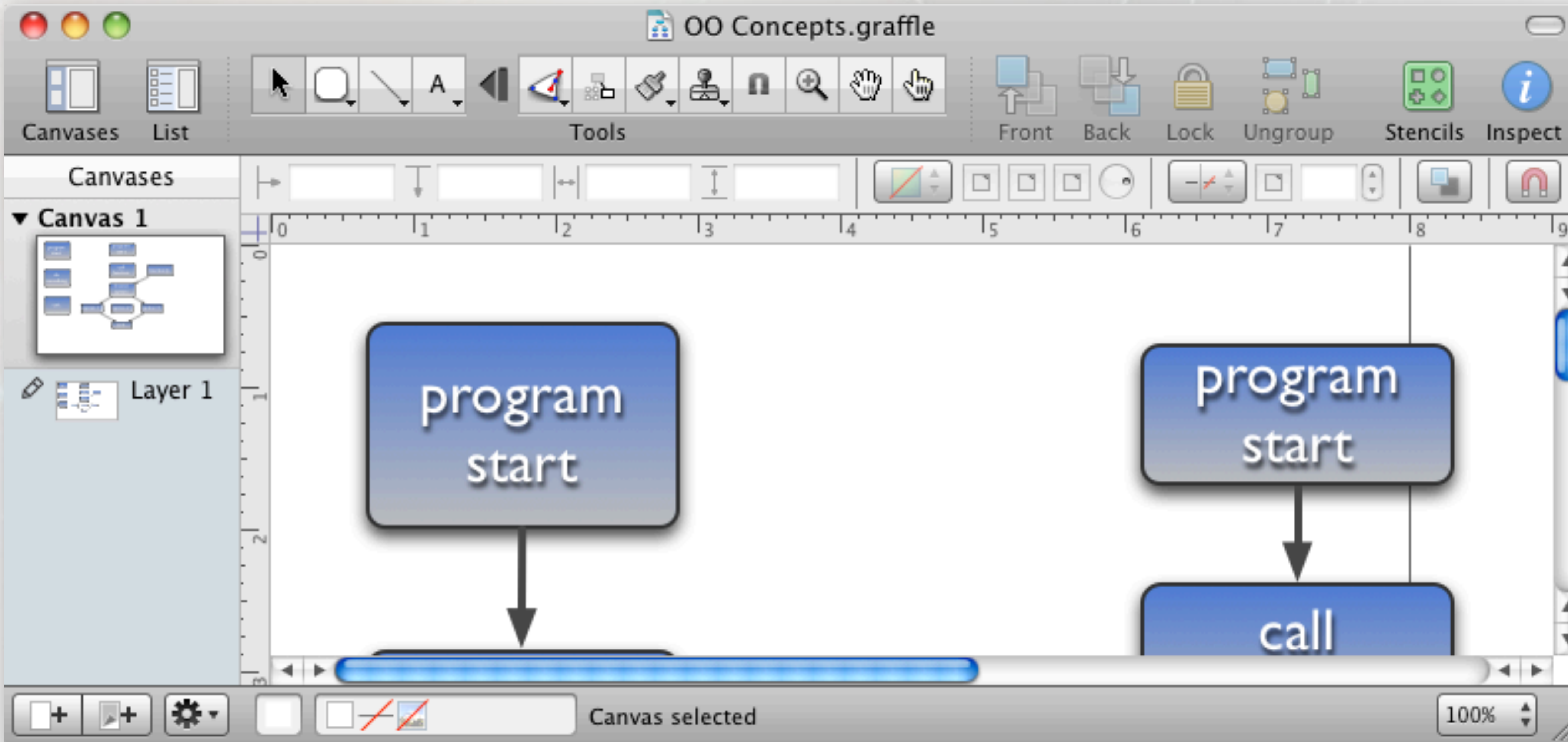
- Linear flow
- Not much opportunity to reuse code
- All data typically managed by main program

Procedural Programming



- A little more complex, can reuse functions, but still linear.

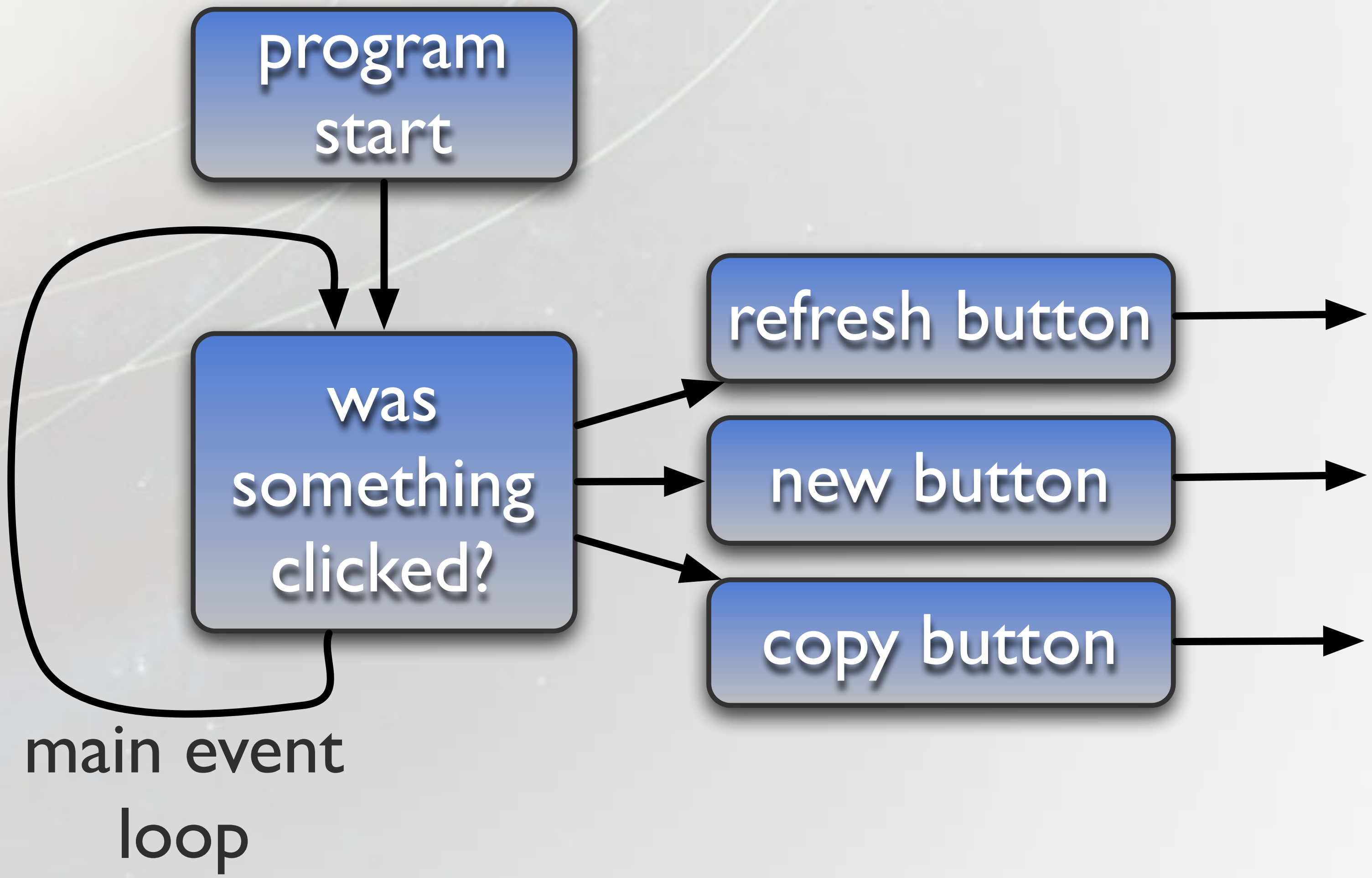
How Would You Program This??




```
if mouseClick in (30,30,60,60) then...
if mouseClick in (60,30,120,60) then...
if mouseClick in (90,30,150,60) then...
...
```

Main Event Loop

- Instead of a linear progression, the program continually asks "did something happen"?
- When something happens, (e.g. a mouse click), the thing that's clicked "knows" what to do.
- This only works when every "thing" is broken out into a separate unit of code, such as buttons, windows, etc.
- Rather than a *huge* list of "if" statements, each thing just waits for a message that says it was clicked.



What is an Object?

- If you are familiar with C, consider a struct.
- Custom-made structure to reflect your data model.
- Only stores data - doesn't know how to do anything.
- Allows you to create your own data type.
- Much easier to deal with conceptually (custom data types).

```
struct Rectangle {
    float height;
    float width;
    char color[20];
};
```

```
typedef struct {
    float height;
    float width;
    char color[20];
} Rectangle;

Rectangle r;
r.height = 9.0
r.width = 16.0;
r.color = 'red'
```

What is an Object?

- Problems with structs:
  - You must always initialize values (some values will never change).
  - The struct doesn't know how to do anything (calculate values, etc.).
  - If you need to change the definition of a struct, it could easily break existing code.

```
struct Rectangle {
    float height;
    float width;
    char color[20];
};
```

```
typedef struct {
    float height;
    float width;
    char color[20];
} Rectangle;

Rectangle r;
r.height = 9.0
r.width = 16.0;
r.color = 'red'
```

What is an Object?

- How to define an object in Python.
- Can initialize all variables that the object contains, once.

```
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 objects - 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)
claculate_triangle_area(tri.width, tri.height)
calculate_tatrahedron_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.

```
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.width = 0.0
        self.height = 0.0
        self.color = "black"
```

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 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()

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

abstract superclass

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

```
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

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()

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

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
```

ABC = Abstract Base Class

magic that tells Python that this is an abstract class

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

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

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.

C++ Diversion

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 `int`s, `float`s, and arrays, are not objects.

So actually quite similar, but Python required 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:

```
s = "A string."
s = s + " 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 exiting object

Constructor (Python)

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*

can perform any initializations needed here

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

p = Point(4,2)
print p.x # 4
```

set x, y at object creation

specifying x,y now optional - default values provided

```
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.y, p.y # 4, 0

p = Point(y=2)
print p.x, p.y # 0.0, 4
```

can set any value by name

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•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.

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

```
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]
```

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

```
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
```

Thus endeth the lesson. You now know everything about OO that you need to.

Questions?

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.