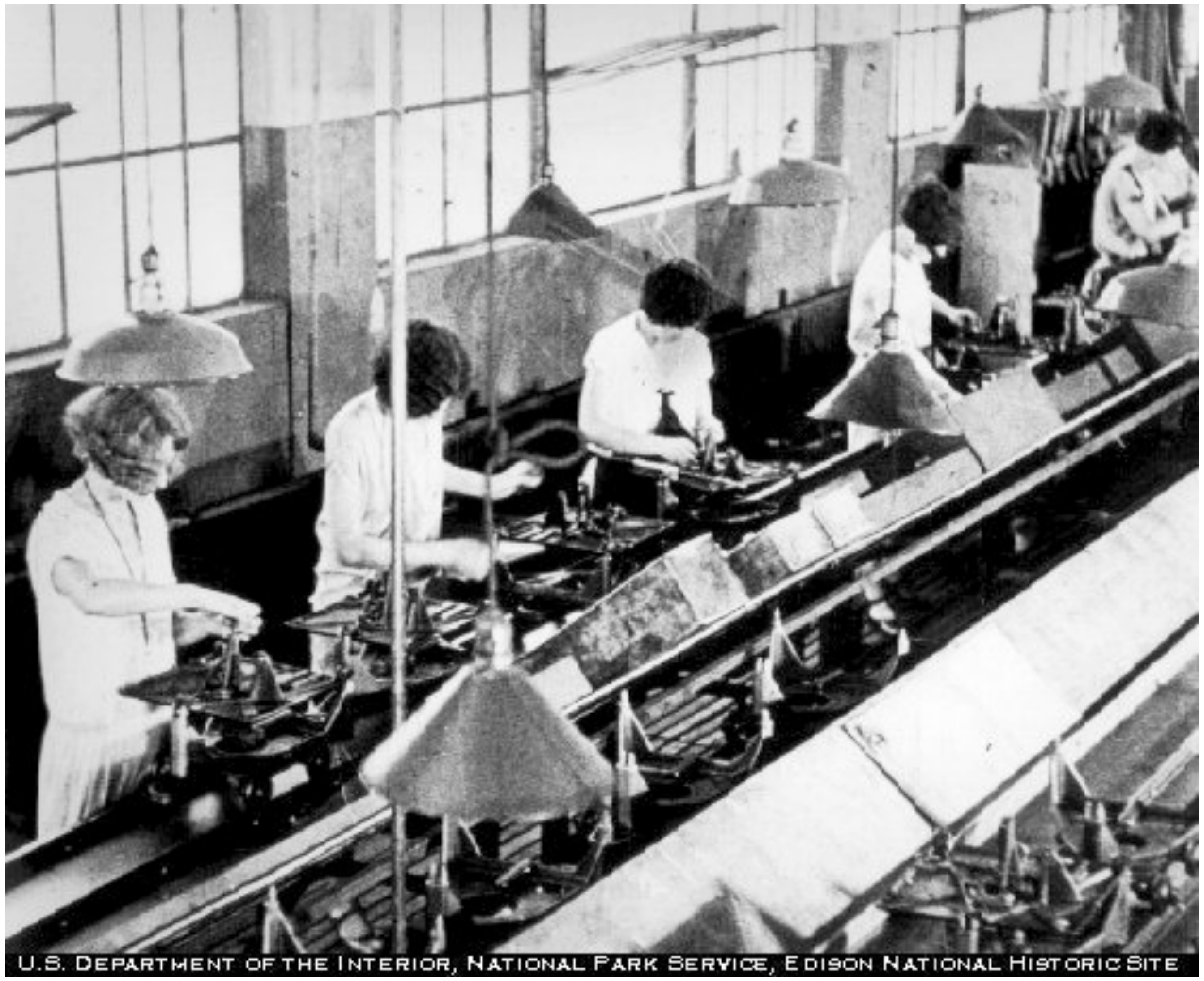
# Object-Oriented Programming I:

## Classes, Attributes, Methods, and Instances

# Brief Outline

- What is object-oriented programming?
- How do I implement it in Python?
- Basic examples

# Procedural Programming




function1(var1, var2, etc.)

⇓

function2(var3,var4,etc.)

⇓

function3(var5,var6,etc.)

⇓

...

⇓

Final Product

#1

# What is Object-Oriented Programming?


Answer 1a: Ask an expert

#2

# What is Object-Oriented Programming?

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance.

Answer Ib: Ask ~~an expert~~ Wikipedia

# What is Object-Oriented Programming?

*Characteristics* → Color, Height, Weight

*Does Things* → Eat, Sleep, Growl, Cheer

*Interaction* → Parents, siblings, friends

Objects are like animals: they know how to do stuff (like eat and sleep), they know how to interact with others (like make children), and they have characteristics (like height, weight).

# What is Object-Oriented Programming?

**Bear**

Variables: color, height, weight

Functions: eat, sleep, growl

An object is a programming structure that allows you to group together variables (characteristics) and functions (doing things) in one nice, tidy package. In Python, the blueprint for an object is referred to as a class.

#5

# What is Object-Oriented Programming?

**Bear**

Attributes: color, height, weight

Methods: eat, sleep, growl

Within a class, the variables are referred to as attributes and the functions are referred to as methods.

# What is Object-Oriented Programming?

| Yogi | Winnie |
|---|---|
| Attributes: brown, 1.8 m, 80 kg | Attributes: yellow, 1.2 m, 100 kg |
| Methods: eat, sleep, growl | Methods: eat, sleep, growl |

Instances are specific realizations of a class

# Object Syntax in Python

```
class ClassName[(BaseClasses)]:

   """[Documentation String]"""

   [Statement1] # Executed only when class is defined
   [Statement2]
   ...
   [Variable1] # "Global" class variables can be defined here

   def Method1(self, args, kwargs={}):
      # Performs task 1

   def Method2(self, args, kwargs={}):
      # Performs task 2
   ...
```

# Bear: Our first Python class

```
>>> class Bear:
```

We are defining a new class named *Bear*. Note the lack of parentheses. These are only used if the class is derived from other classes (more on this next lecture).

# Bear: Our first Python class

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
```

This print statement is executed only when the class is defined.

# Bear: Our first Python class

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
>>> a = Bear
>>> a
<class __main__.Bear at 0x10041d9b0>
```

This statement equates the object *a* to the class *Bear*. This is typically not very useful.

# Bear: Our first Python class

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
>>> a = Bear
>>> a
<class __main__.Bear at 0x10041d9b0>
>>> a = Bear()
>>> a
<__main__.Bear instance at 0x100433cb0>
```

By adding parenthesis, we are creating a new *instance* of the class *Bear.*

# Attributes: Access, Creation, Deletion

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
>>> a = Bear()
>>> a.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Bear instance has no attribute 'name'
>>> a.name = "Oski"
>>> a.color = "Brown"
>>> del(a.name)
>>> a.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Bear instance has no attribute 'name'
```

Object attributes are accessed with the "." (period) operator

# Attributes: Access, Creation, Deletion

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
>>> a = Bear()
>>> a.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Bear instance has no attribute 'name'
>>> a.name = "Oski"
>>> a.color = "Brown"
>>> del(a.name)
>>> a.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Bear instance has no attribute 'name'
```

(Instance-specific) attributes can be created and deleted outside of the class definition

# Methods: Access, Creation, and (not) Deletion

```
>>> class Bear:
...     print "The bear class is now defined."
...     def say_hello(self):
...        print "Hello, world!  I am a bear."
...
The bear class is now defined.
>>> a = Bear()
>>> a.say_hello
<bound method Bear.say_hello of <__main__.Bear
instance at 0x100433e18>>
>>> a.say_hello()
Hello, world!  I am a bear.
```

Methods are defined in the same way normal functions are (note that we will return to the *self* object in a few slides)

# Methods: Access, Creation, and (not) Deletion

```
>>> class Bear:
...     print "The bear class is now defined."
...     def say_hello(self):
...        print "Hello, world!  I am a bear."
...
The bear class is now defined.
>>> a = Bear()
>>> a.say_hello
<bound method Bear.say_hello of <__main__.Bear
instance at 0x100433e18>>
>>> a.say_hello()
Hello, world!  I am a bear.
```

Like attributes, methods are also accessed via the "." operator. Parentheses indicate the method should be executed.

# The __init__ method

```
>>> class Bear:
...     def __init__(self, name):
...         self.name = name
...     def say_hello(self):
...         print "Hello, world!  I am a bear."
...         print “My name is %s.” % self.name
...
>>> a = Bear()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2
arguments (1 given)
>>> a = Bear(“Yogi”)
>>> a.name
'Yogi'
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
```

__init__ is a special Python method. It is always run when a new instance of a class is created.

# The __init__ method

```
>>> class Bear:
...     def __init__(self, name):
...         self.name = name
...     def say_hello(self):
...         print "Hello, world!  I am a bear."
...         print “My name is %s.” % self.name
...
>>> a = Bear()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2
arguments (1 given)
>>> a = Bear(“Yogi”)
>>> a.name
'Yogi'
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
```

Arguments specified by *__init__* must be provided when creating a new instance of a class (else an Exception will be thrown)

# The __init__ method

```
>>> class Bear:
...     def __init__(self, name):
...         self.name = name
...     def say_hello(self):
...         print "Hello, world!  I am a bear."
...         print “My name is %s.” % self.name
...
>>> a = Bear()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2
arguments (1 given)
>>> a = Bear(“Yogi”)
>>> a.name
'Yogi'
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
```

Attributes and methods are accessed with the “.” operator. Methods require a parentheses to invoke action.

# Scope: self and “class” variables

```
>>> class Bear:
...    population = 0
...    def __init__(self, name):
...       self.name = name
...       Bear.population += 1
...    def say_hello(self):
...       print "Hello, world!  I am a bear."
...       print “My name is %s.” % self.name
...       print “I am number %i.” % Bear.population
...
>>> a = Bear(“Yogi”)
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
I am number 1.
>>> b = Bear("Winnie")
>>> b.say_hello()
Hello, I am a bear.
My name is Winnie.
I am number 2.
```

Class-wide (“global”) attributes can be declared. It is good style to do this before the __init__ method.

# Scope: self and "class" variables

```
>>> class Bear:
...    population = 0
...    def __init__(self, name):
...       self.name = name
...       Bear.population += 1
...    def say_hello(self):
...       print "Hello, world!  I am a bear."
...       print “My name is %s.” % self.name
...       print “I am number %i.” % Bear.population
...
>>> a = Bear(“Yogi”)
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
I am number 1.
>>> b = Bear("Winnie")
>>> b.say_hello()
Hello, I am a bear.
My name is Winnie.
I am number 2.
```

They are accessed in the same way as "instance-specific" attributes, but using the class name instead of the instance name.

# Scope: self and “class” variables

```
>>> class Bear:
...    population = 0
...    def __init__(self, name):
...       self.name = name
...       Bear.population += 1
...    def say_hello(self):
...       print "Hello, world!  I am a bear."
...       print “My name is %s.” % self.name
...       print “I am number %i.” % Bear.population
...
```

The *self* variable is a placeholder for the specific instance of a class. Attributes referenced to *self* are known as “object” attributes.

# Scope: self and “class” variables

```
>>> class Bear:
...     population = 0
...     def __init__(self, name):
...        self.name = name
...        Bear.population += 1
...     def say_hello(self):
...        print "Hello, world!  I am a bear."
...        print “My name is %s.” % self.name
...        print “I am number %i.” % Bear.population
...
```

It should be listed as a required argument in all class methods (even if it is not explicitly used by the method).

# Scope: self and “class” variables

```
>>> class Bear:
...     population = 0
...     def __init__(self, name):
...        self.name = name
...        Bear.population += 1
...     def say_hello(self):
...        print "Hello, world!  I am a bear."
...        print “My name is %s.” % self.name
...        print “I am number %i.” % Bear.population
...
>>> a = Bear(“Yogi”)
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
I am number 1.
>>> b = Bear("Winnie")
>>> b.say_hello()
Hello, world!  I am a bear.
My name is Winnie.
I am number 2.
```

When calling a method directly from a specific instance of a class, the *self* variable is NOT passed (Python handles this for you)

# Scope: self and “class” variables

```
>>> class Bear:
...    population = 0
...    def __init__(self, name):
...       self.name = name
...       Bear.population += 1
...    def say_hello(self):
...       print "Hello, world!  I am a bear."
...       print “My name is %s.” % self.name
...       print “I am number %i.” % Bear.population
...
>>> a = Bear(“Yogi”)
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
I am number 1.
>>> b = Bear("Winnie")
>>> b.say_hello()
Hello, world!  I am a bear.
My name is Winnie.
I am number 2.
```

Here the *population* variable is incremented each time a new instance of the *Bear* class is created.

# Scope: self and “class” variables

```
>>> class Bear:
...     population = 0
...     def __init__(self, name):
...        self.name = name
...        Bear.population += 1
...     def say_hello(self):
...        print "Hello, world!  I am a bear."
...        print “My name is %s.” % self.name
...        print “I am number %i.” % Bear.population
...
>>> a = Bear(“Yogi”)
>>> a.say_hello()
Hello, world!  I am a bear.
My name is Yogi.
I am number 1.
>>> b = Bear("Winnie")
>>> b.say_hello()
Hello, world!  I am a bear.
My name is Winnie.
I am number 2.
```

```
>>> c = Bear("Fozzie")
>>> Bear.say_hello(c)
Hello, I am a bear.
My name is Fozzie.
I am number 3.
```

When calling methods from a class, a specific instance DOES need to be passed.

# A Zookeeper's Travails I

Suppose you are a zookeeper. You have three bears in your care (Yogi, Winnie, and Fozzie), and you need to take them to a shiny new habitat in a different part of the zoo. However, your bear truck can only support 300 lbs. Can you transfer the bears in just one trip?

#14

# A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...
>>> a = Bear("Yogi", 80)
>>> b = Bear("Winnie", 100)
>>> c = Bear("Fozzie", 115)
>>> my_bears = [a, b, c]
>>> total_weight = 0
>>> for z in my_bears:
...     total_weight += z.weight
...
>>> total_weight < 300
True
```

Class instances in Python can be treated like any other data type: they can be assigned to other variables, put in lists, iterated over, etc.

# A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...
>>> a = Bear("Yogi", 80)
>>> b = Bear("Winnie", 100)
>>> c = Bear("Fozzie", 115)
>>> my_bears = [a, b, c]
>>> total_weight = 0
>>> for z in my_bears:
...     total_weight += z.weight
...
>>> total_weight < 300
True
```

In iterating over *my_bears*, we are assigning the temporary variable *z* to *Bear* instances *a*, *b*, and *c*. The *weight* method is accessed again with the "." operator.

#15

# A Zookeeper's Travails II

Consider now a (marginally) more realistic scenario, where a bear's weight changes when he/she eats and hibernates

# A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...     def eat(self, amount):
...         self.weight += amount
...     def hibernate(self):
...         self.weight /= 1.20
...
>>> a = Bear("Yogi", 80)
>>> b = Bear("Winnie", 100)
>>> c = Bear("Fozzie", 115)
>>> my_bears=[a, b, c]
```

Object methods can alter other properties of the object

#17

# A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...     def eat(self, amount):
...         self.weight += amount
...     def hibernate(self):
...         self.weight /= 1.20
...
>>> a = Bear("Yogi", 80)
>>> b = Bear("Winnie", 100)
>>> c = Bear("Fozzie", 115)
>>> my_bears=[a, b, c]
```

```
>>> a.weight
80
>>> a.eat(20)
>>> a.weight
100
>>> b.eat(10)
>>> c.hibernate()
>>> total_weight = 0
>>> for z in my_bears:
...     total_weight += z.weight
...
>>> total_weight < 300
False
```

Yogi finds several picnic baskets to snack on.

# A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...     def eat(self, amount):
...         self.weight += amount
...     def hibernate(self):
...         self.weight /= 1.20
...
>>> a = Bear("Yogi", 80)
>>> b = Bear("Winnie", 100)
>>> c = Bear("Fozzie", 115)
>>> my_bears=[a, b, c]
```

```
>>> a.weight
80
>>> a.eat(20)
>>> a.weight
100
>>> b.eat(10)
>>> c.hibernate()
>>> total_weight = 0
>>> for z in my_bears:
...     total_weight += z.weight
...
>>> total_weight < 300
False
```

Winnie eats a large pot of honey, while Fozzie hibernates

#17

# A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...     def eat(self, amount):
...         self.weight += amount
...     def hibernate(self):
...         self.weight /= 1.20
...
>>> a = Bear("Yogi", 80)
>>> b = Bear("Winnie", 100)
>>> c = Bear("Fozzie", 115)
>>> my_bears=[a, b, c]
```

```
>>> a.weight
80
>>> a.eat(20)
>>> a.weight
100
>>> b.eat(10)
>>> c.hibernate()
>>> total_weight = 0
>>> for z in my_bears:
...     total_weight += z.weight
...
>>> total_weight < 300
False
```

As a result, they are too heavy for the truck

#17

# A Zookeeper's Travails III

Bears are social creatures. In spending time together at the zoo, they can become friends and even change their behavior.

#18

# A Zookeeper's Travails III

```
>>> class Bear:
...     def __init__(self, name, fav_food, friends=[]):
...         self.name = name
...         self.fav_food = fav_food
...         self.friends = friends
...     def same_food(self):
...         for friend in self.friends:
...             if (friend.fav_food == self.fav_food):
...                 print "%s and %s both like %s" % \
...                  (self.name, friend.name, self.fav_food)
...
>>> a = Bear("Yogi", "Picnic baskets")
>>> b = Bear("Winnie", "Honey")
>>> c = Bear("Fozzie", "Frog legs")
```

Class instances (in this case in list format) can be passed as arguments, just like any other Python object

#19

# A Zookeeper's Travails III

```
>>> class Bear:
...     def __init__(self, name, fav_food, friends=[]):
...        self.name = name
...        self.fav_food = fav_food
...        self.friends = friends
...     def same_food(self):
...        for friend in self.friends:
...           if (friend.fav_food == self.fav_food):
...              print "%s and %s both like %s" % \
...               (self.name, friend.name, self.fav_food)
...
>>> a = Bear("Yogi", "Picnic baskets")
>>> b = Bear("Winnie", "Honey")
>>> c = Bear("Fozzie", "Frog legs")
```

The *same_food* method determines if a bear has the same favorite food as any of his friends

#19

# A Zookeeper’s Travails III

```
>>> class Bear:
...     def __init__(self, name, fav_food, friends=[]):
...         self.name = name
...         self.fav_food = fav_food
...         self.friends = friends
...     def same_food(self):
...         for friend in self.friends:
...             if (friend.fav_food == self.fav_food):
...                 print "%s and %s both like %s" % \
...                  (self.name, friend.name, self.fav_food)
...
>>> a = Bear("Yogi", "Picnic baskets")
>>> b = Bear("Winnie", "Honey")
>>> c = Bear("Fozzie", "Frog legs")
```

Initially, none of the bears have friends and they all have different favorite foods

# A Zookeeper's Travails III

```
>>> c.friends
[]
>>> c.fav_food
'Frog legs'
>>> c.same_food()
>>> c.friends= [a, b]
>>> c.same_food()
>>> c.fav_food = "Honey"
>>> c.same_food()
Fozzie and Winnie both like Honey
```

Fozzie's *friends* method is therefore initially defined to be an empty list. His favorite food is frog legs.

# A Zookeeper’s Travails III

```
>>> c.friends
[]
>>> c.fav_food
'Frog legs'
>>> c.same_food()
>>> c.friends= [a, b]
>>> c.same_food()
>>> c.fav_food = "Honey"
>>> c.same_food()
Fozzie and Winnie both like Honey
```

Without any friends, Fozzie can’t share the same favorite food with anyone.

# A Zookeeper's Travails III

```
>>> c.friends
[]
>>> c.fav_food
'Frog legs'
>>> c.same_food()
>>> c.friends= [a, b]
>>> c.same_food()
>>> c.fav_food = "Honey"
>>> c.same_food()
Fozzie and Winnie both like Honey
```

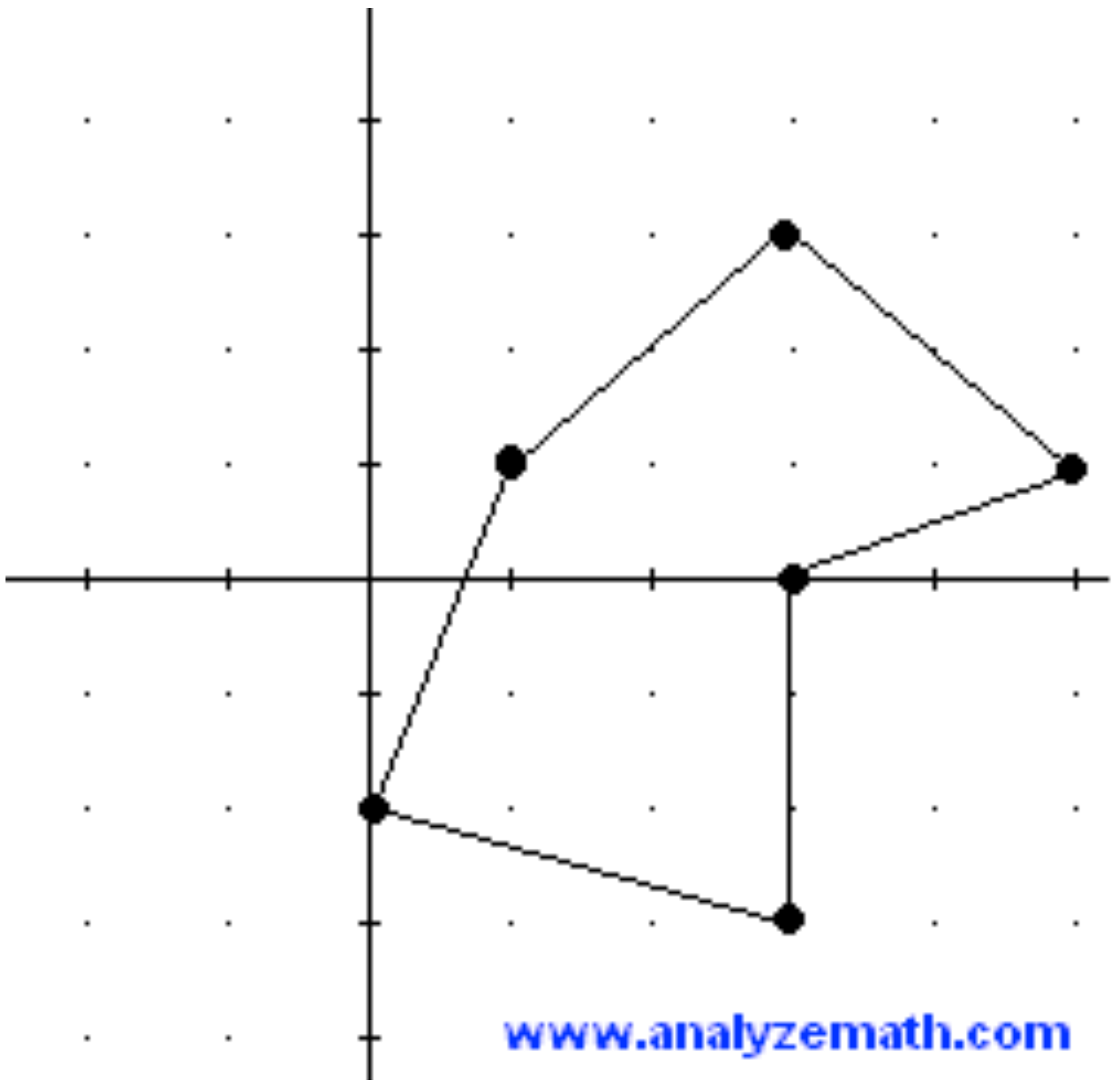
After some time together, Fozzie makes friends with the other bears. But they still don't share a common favorite food.

# A Zookeeper's Travails III

```
>>> c.friends
[]
>>> c.fav_food
'Frog legs'
>>> c.same_food()
>>> c.friends= [a, b]
>>> c.same_food()
>>> c.fav_food = "Honey"
>>> c.same_food()
Fozzie and Winnie both like Honey
```

Finally, Fozzie tries honey and realizes he loves it. Now he and Winnie share a common favorite food. Kermit is very happy.

# Breakout problem: Polygon Perimeter

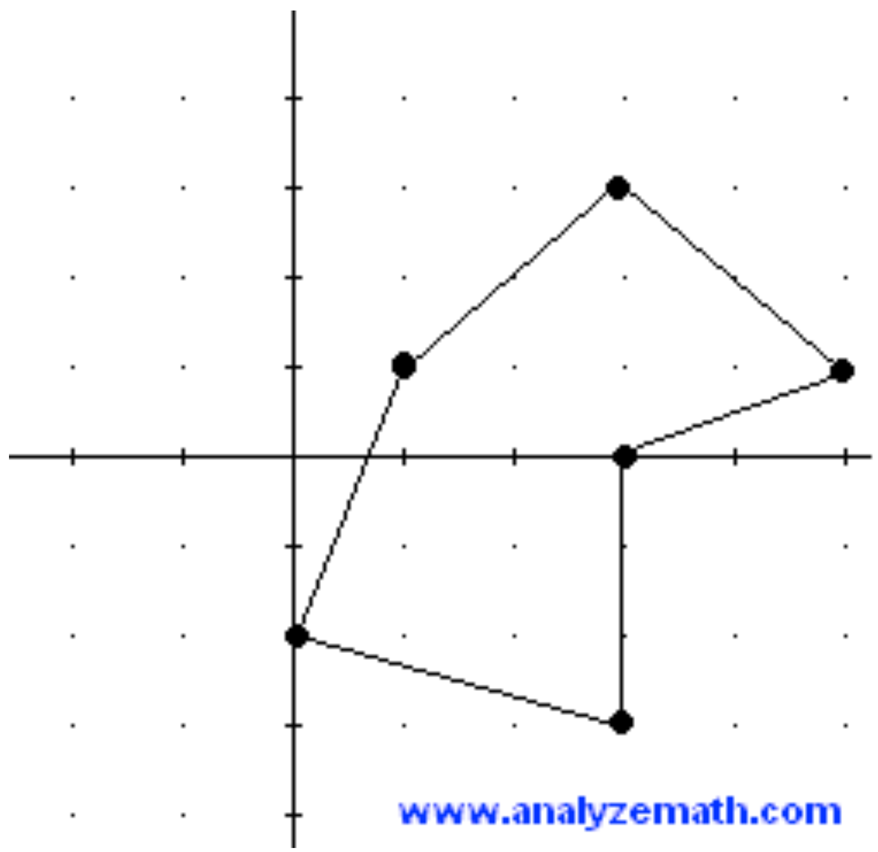



*Calculate the perimeter (and, if you are up for it, the area) of a polygon provided the vector coordinates (in order) of its N vertices.*

*Hint: Sum over distance between adjacent points, where d = math.sqrt($\delta x^2 + \delta y^2$) .*

#21

# In other words ...




*Calculate the perimeter (and, if you are up for it, the area) of a polygon provided the vector coordinates (in order) of its N vertices.*

```
>>> a = Polygon([[0,0], [0,1], [1,1], [1,0]])
>>> a.perimeter()
4.0
>>> a.area()
1.0
>>> b = Polygon([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
>>> b.perimeter()
17.356451097651515
```

#21

# Solutions

# For the remaining skeptics ...

| | | |
|---|---|---|
| Instantiation | >>> a = Polygon("Polly")<br>(Creating an instance of the class Polygon) | >>> b = “Polygon” |
| Types | >>> type(a)<br><type 'instance'><br>>>> type(type(a))<br><type 'type'> | >>> type(b)<br><type 'str'><br>>>> type(type(b))<br><type 'type'> |
| Methods | >>> a.print_name()<br>Hi, my name is Polly.<br>>>> a.perimeter()<br>0 | >>> b.upper()<br>POLYGON<br>>>> b.replace(“gon”, “wog”)<br>Polywog |

Because of the way Python is set up, you have been using object-oriented techniques this entire time!

# A "Procedural" Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Define a function that takes vertices as input (i.e., variable) and returns perimeter

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    """A new class named Polygon."""
...    def __init__(self, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
...
>>> a = Polygon([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
>>> a.perimeter()
17.356451097651515
```

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Imports math module and associated routines

# A "Procedural" Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Define a new **function** named *perimeter* that requires a single **argument** named *polygon*

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

A documentation string describes (in English) the purpose of the function

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Initializes the variable *sum*

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Loop over each individual vertex in the variable *polygon*

# A "Procedural" Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Grab adjacent vertices. Modulo operator (%) avoids list index exception

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Calculate the distance between adjacent vertices

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Increment the *distance* variable

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Return the value of *sum*

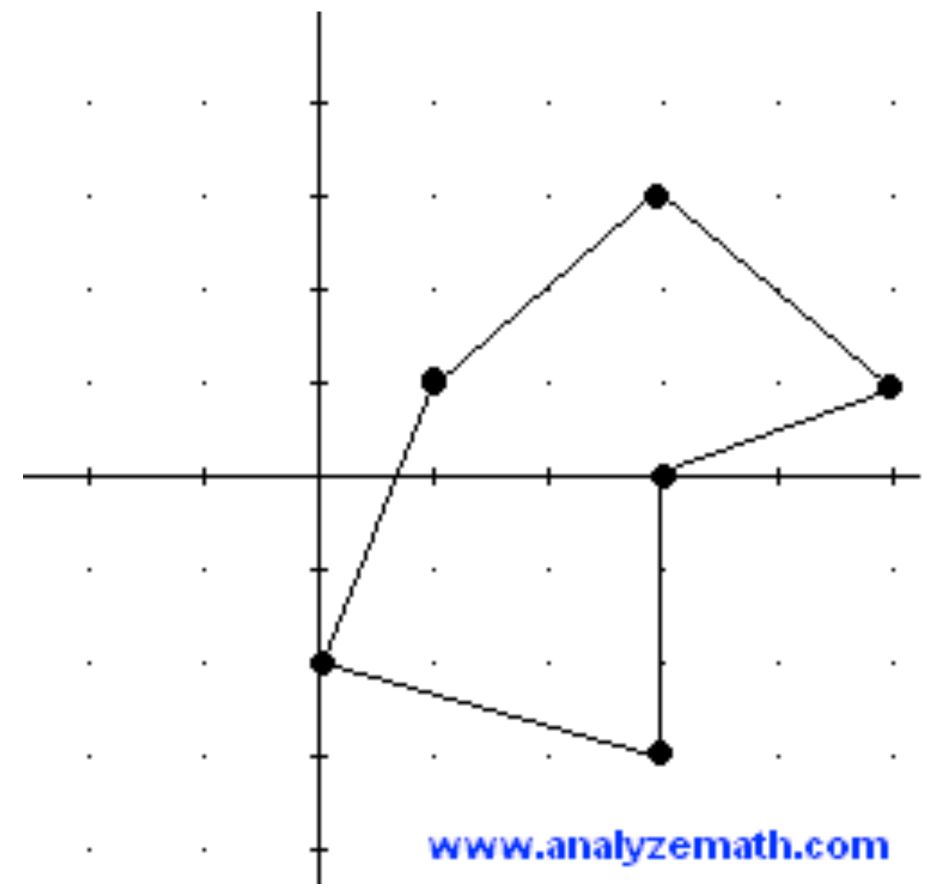
# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Unit square

# A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```



# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...     def __init__(self, name, vertices=[]):
...        self.vertices = vertices
...        print "(Creating an instance of the class Polygon)"
...     def perimeter(self):
...        sum = 0
...        for i in range(len(self.vertices)):
...           vertex1 = self.vertices[i]
...           vertex2 = self.vertices[(i+1) % len(self.vertices)]
...           distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                                pow(vertex2[1]-vertex1[1],2))
...           sum += distance
...        return sum
```

Define a new class named *polygon.* Note that the class name is not followed by parentheses (unless it has a parent class - more on this in the next lecture)

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

__*init*__ is a special Python method. It is called every time a new instance of a class is created

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

*self* is also a special Python object. It is essentially a reference to the specific instance of the class

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

The arguments that follow *self* in the declaration of the *__init__* method (and all others, for that matter), are just like other Python arguments. In this case, *name* is required and *vertices* is optional

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

The initialization steps. Whenever a new instance of the *Polygon* class is created, the attribute *vertices* is set, and a message is printed to stdout

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...     def __init__(self, name, vertices=[]):
...        self.vertices = vertices
...        print "(Creating an instance of the class Polygon)"
...     def perimeter(self):
...        sum = 0
...        for i in range(len(self.vertices)):
...           vertex1 = self.vertices[i]
...           vertex2 = self.vertices[(i+1) % len(self.vertices)]
...           distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                                pow(vertex2[1]-vertex1[1],2))
...           sum += distance
...        return sum
```

The *perimeter* method takes no arguments. Using the vertices associated with the polygon instance, it calculates the perimeter exactly as before.

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

```
>>> a = Polygon(vertices=[[0,0],[0,1],\
...              [1,1],[1,0]])
(Creating an instance of the class
Polygon)
>>> a.perimeter()
4.0
```

# Extra Slides

# An Object-Oriented Approach

Polygon

Attributes: name, vertices

Functions: perimeter, area

1) Define a new class named *polygon* incorporating the relevant attributes and functions
2) Create an instance of the class *polygon* with the desired properties and determine the perimeter

# An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.name = name
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def print_name(self):
...       print "Hi, my name is %s." % self.name
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

```
>>> a = Polygon("Polly", [[0,-2],[1,1],
[3,3],[5,1],[4,0],[4,-3]])
(Creating an instance of the class
Polygon)
>>> a.print_name()
Hi, my name is Polly.
>>> a.perimeter()
17.356451097651515
```

The original polygon problem, solved using object-oriented concepts

# When/why should I use object-oriented programs?

- There is no hard and fast rule - like most programming design decisions, there are many different paths to the same goal
- General guideline:
  - Many instances of a complex data type (think *struct* in C) that is intimately connected to the primary functions you will be writing

# A Real-World Example: Robotic Telescopes

The 1.3 m PAIRITEL dome (Mt. Hopkins, Az.) at sunset

Robotic telescopes are designed to perform via software all the standard tasks associated with manual astronomical observing. In particular, this includes:

1) Scheduling
2) Data reduction

# Robotic Telescopes

- Scheduler - Given a pre-defined list of targets, each with an associated intrinsic priority, select the optimal field to observe at any given time

- Data Reduction Pipeline - Produce a set of science quality images from the raw data and relevant calibration files on a given night

# An Object-Oriented Scheduler

## AstroTarget

1) Sky coordinates (RA, Dec, proper motion)
2) Priority
3) Timing requirements
4) Weather requirements

5) Derive sky location (given time, observatory location)
6) Are timing requirements met?
7) Are weather requirements met?
8) Score / figure of merit

# A “Procedural” Image Reduction Pipeline

- Step 1 - Obtain appropriate calibrations in the afternoon
- Step 2 - Apply zero and gain corrections (bias and flat-field)
- Step 3 - Solve for sky position as a function of detector position (astrometry)
- Step 4 - Solve for sensitivity by comparing with known catalogs (photometry)