Object-Oriented Programming I:
Classes, Attributes, Methods, and Instances

Brief Outline

Brief Outline

- What is object-oriented programming?

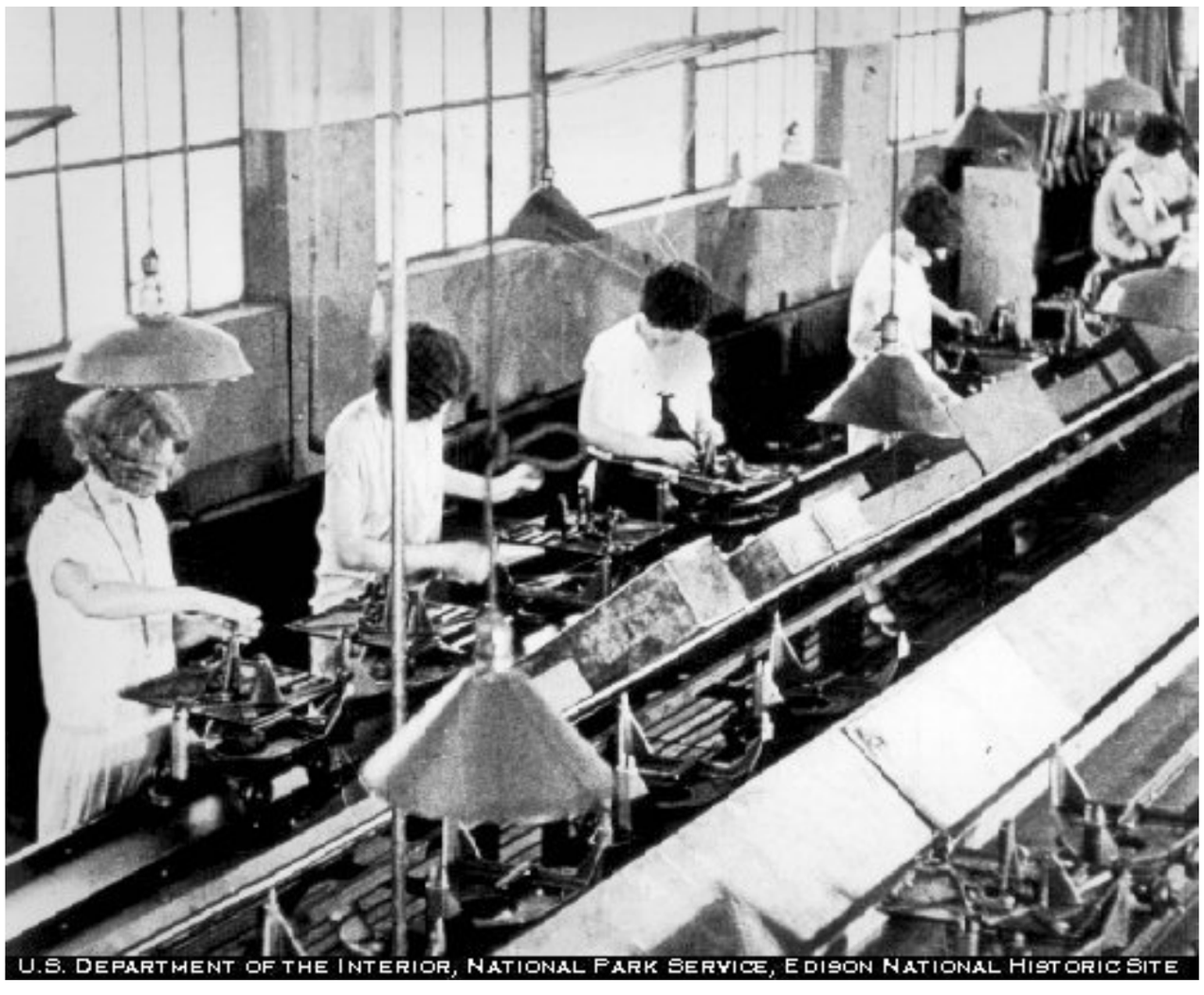
Brief Outline

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

Brief Outline

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

Procedural Programming


U.S. Department of the Interior, National Park Service, Edison National Historic Site

function1(var1, var2, etc.)

⇓

function2(var3,var4,etc.)

⇓

function3(var5,var6,etc.)

⇓

...

⇓

Final Product

Procedural Programming

- This has been the mainstay of much scientific programming, and it works well.
- But it can get very messy when you have a complex program with lots of interacting parts
- Particularly when data has to be shared and modified between many functions

What is Object-Oriented Programming?

What is Object-Oriented Programming?

Answer 1a: Ask an expert

What is Object-Oriented Programming?

Answer 1a: Ask an expert

What is Object-Oriented Programming?

What is Object-Oriented Programming?

Answer Ib: Ask ~~an expert~~ Wikipedia

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?

What is Object-Oriented Programming?

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?

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?

Characteristics → Color, Height, Weight

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?

Characteristics → Color, Height, Weight

Does Things → Eat, Sleep, Growl, Cheer

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?

Characteristics → Color, Height, Weight

Does Things → Eat, Sleep, Growl, Cheer

Interaction → Parents, siblings, friends

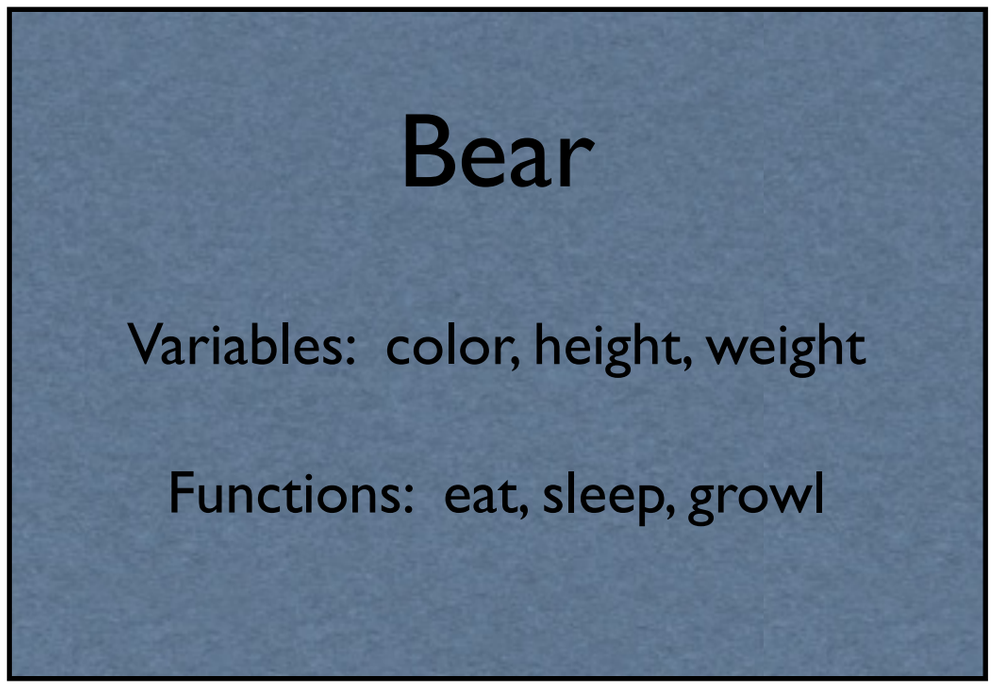
Objects are like animals: they know how to do stuff (like eat and sleep), they know how to interact with others (like make children), and they have characteristics (like height, weight).

What is Object-Oriented Programming?

What is Object-Oriented Programming?

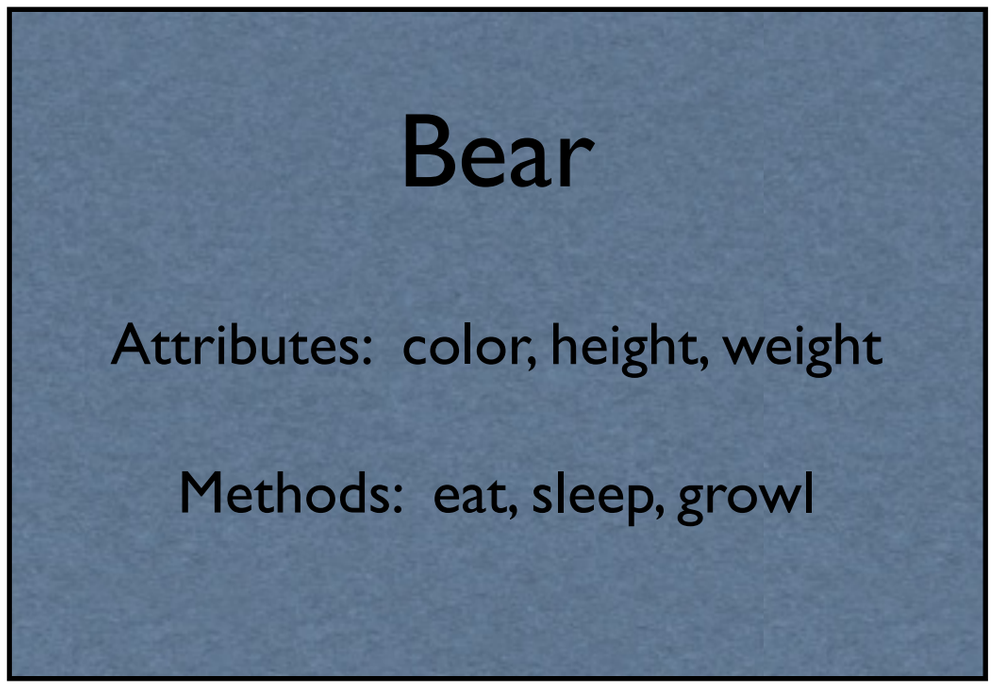
An object is a programming structure that allows you to group together variables (characteristics) and functions (doing things) in one nice, tidy package. In Python, the blueprint for an object is referred to as a class.

What is Object-Oriented Programming?



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.

What is Object-Oriented Programming?

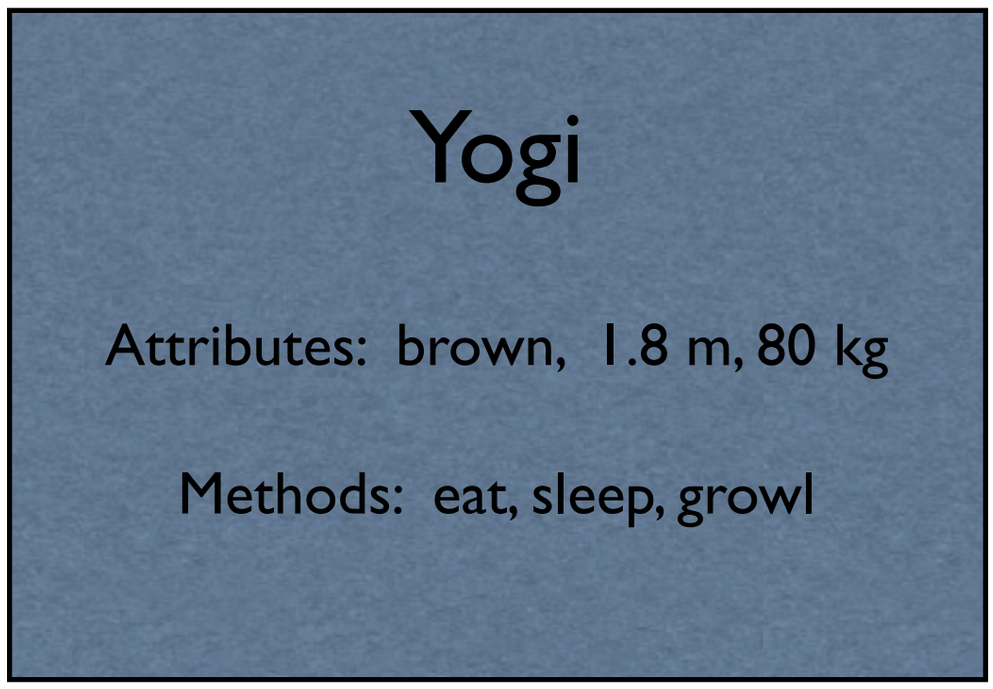


Within a class, the variables are referred to as attributes and the functions are referred to as methods.

What is Object-Oriented Programming?

Instances are specific realizations of a class

What is Object-Oriented Programming?



Instances are specific realizations of a class

What is Object-Oriented Programming?

Yogi

Attributes: brown, 1.8 m, 80 kg

Methods: eat, sleep, growl

Winnie

Attributes: yellow, 1.2 m, 100 kg

Methods: eat, sleep, growl

Instances are specific realizations of a class

Object Syntax in Python

Object Syntax in Python

```
class ClassName[(BaseClasses)]:
```

Object Syntax in Python

```
class ClassName[(BaseClasses)]:
```

Object Syntax in Python

```
class ClassName[(BaseClasses)]:

   """[Documentation String]"""
```

Object Syntax in Python

```
class ClassName[(BaseClasses)]:

   """[Documentation String]"""
```

Object Syntax in Python

```
class ClassName[(BaseClasses)]:

   """[Documentation String]"""

   [Statement1] # Executed only when class is defined
```

Object Syntax in Python

```
class ClassName[(BaseClasses)]:

   """[Documentation String]"""

   [Statement1] # Executed only when class is defined
   [Statement2]
```

Object Syntax in Python

```
class ClassName[(BaseClasses)]:

   """[Documentation String]"""

   [Statement1] # Executed only when class is defined
   [Statement2]
   ...
```

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
```

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
```

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={}):
```

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
```

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
```

Bear: Our first Python class

Bear: Our first Python class

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

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:
```

Bear: Our first Python class

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

Bear: Our first Python class

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

Bear: Our first Python class

```
>>> class Bear:
...     print "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.
```

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.
```

Bear: Our first Python class

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

Bear: Our first Python class

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

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
```

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>
```

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>
```

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()
```

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
```

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>
```

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()
```

Attributes: Access, Creation, Deletion

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
>>> a = Bear()
>>> a.name
```

Attributes: Access, Creation, Deletion

```
>>> class Bear:
...     print "The bear class is now defined."
...
The bear class is now defined.
>>> a = Bear()
>>> a.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):
```

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>
```

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'
```

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'
```

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'
```

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

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"
```

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

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"
```

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

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)
```

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

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
```

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

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):
```

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

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>
```

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

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."
```

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):
```

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."
```

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."
...
```

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.
```

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()
```

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

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
```

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

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>>
```

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

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()
```

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

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:
```

__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):
```

__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
```

__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):
```

__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."
```

__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
```

__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
...
```

__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()
```

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):
```

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>
```

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)
```

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”)
```

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

Scope: self and “class” variables

Scope: self and “class” variables

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

Scope: self and “class” variables

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

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
```

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
```

Scope: self and “class” variables

```
>>> class Bear:
...     population = 0
```

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):
```

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
```

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
```

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):
```

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."
```

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
```

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
```

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
...
```

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
...
```

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”)
```

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()
```

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.
```

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.
```

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.
```

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")
```

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()
```

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.
```

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.
```

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.
```

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.
```

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

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")
```

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

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)
```

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

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.
```

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

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.
```

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

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?

A Zookeeper's Travails I

A Zookeeper's Travails I

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

A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
```

A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
```

A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
```

A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...
```

A Zookeeper's Travails I

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...
>>> a = Bear("Yogi", 80)
```

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)
```

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)
```

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]
```

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]
```

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]
```

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.

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
```

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.

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:
```

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.

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
```

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.

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
...
```

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.

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
```

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.

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.

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

Object methods can alter other properties of the object

A Zookeeper's Travails II

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

Object methods can alter other properties of the object

A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
```

Object methods can alter other properties of the object

A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
```

Object methods can alter other properties of the object

A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
```

Object methods can alter other properties of the object

A Zookeeper's Travails II

```
>>> class Bear:
...     def __init__(self, name, weight):
...         self.name = name
...         self.weight = weight
...     def eat(self, amount):
```

Object methods can alter other properties of the object

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
```

Object methods can alter other properties of the object

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):
```

Object methods can alter other properties of the object

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
```

Object methods can alter other properties of the object

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
...
```

Object methods can alter other properties of the object

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)
```

Object methods can alter other properties of the object

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)
```

Object methods can alter other properties of the object

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)
```

Object methods can alter other properties of the object

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

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 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]
```

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
```

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
```

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)
```

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
```

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
```

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
```

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
```

Winnie eats a large pot of honey, while Fozzie 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]
```

```
>>> a.weight
80
>>> a.eat(20)
>>> a.weight
100
>>> b.eat(10)
```

Winnie eats a large pot of honey, while Fozzie 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]
```

```
>>> a.weight
80
>>> a.eat(20)
>>> a.weight
100
>>> b.eat(10)
>>> c.hibernate()
```

Winnie eats a large pot of honey, while Fozzie 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]
```

```
>>> a.weight
80
>>> a.eat(20)
>>> a.weight
100
>>> b.eat(10)
>>> c.hibernate()
```

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
```

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:
```

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
```

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
...
```

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
```

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
```

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

For the remaining skeptics ...

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

For the remaining skeptics ...

Instantiation	`>>> a = Polygon("Polly")` `(Creating an instance of the class Polygon)`	`>>> b = “Polygon”`

Types	`>>> type(a)`
	`<type 'instance'>`
	`>>> type(type(a))`
	`<type 'type'>`
		`>>> type(b)`
		`<type 'str'>`
		`>>> type(type(b))`
		`<type 'type'>`

Methods	`>>> a.print_name()`
	`Hi, my name is Polly.`
	`>>> a.perimeter()`
	`0`
		`>>> b.upper()`
		`POLYGON`
		`>>> b.replace(“gon”, “wog”)`
		`Polywog`

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

A More Relevant Example: Simple N-body Code

OOPI-nbody.ipynb