Classes and Objects

Basics

Copyright © Software Carpentry 2010

This work is licensed under the Creative Commons Attribution License

See http://software-carpentry.org/license.html for more information.

Two basic concepts in OOP are *class* and *object*

Two basic concepts in OOP are *class* and *object*

A class defines the behavior of a new kind of thing

Two basic concepts in OOP are *class* and *object*

A class defines the behavior of a new kind of thing

An object is a thing with particular properties

Two basic concepts in OOP are *class* and *object*

A class defines the behavior of a new kind of thing

An object is a thing with particular properties

	Biology	Programming
General		
Specific		

Two basic concepts in OOP are *class* and *object*

A class defines the behavior of a new kind of thing

An object is a thing with particular properties

	Biology	Programming
General	Species *canis lupus*	
Specific	Organism Waya	

Two basic concepts in OOP are *class* and *object*

A class defines the behavior of a new kind of thing

An object is a thing with particular properties

	Biology	Programming
General	Species *canis lupus*	Class `Vector`
Specific	Organism Waya	Object `velocity`

Define a new class with no behavior

software carpentry

Define a new class with no behavior

```
>>> class Empty(object):
...     pass
```

Define a new class with no behavior

```
>>> class Empty(object):
...     pass
```

Create two objects of that class

Define a new class with no behavior

```
>>> class Empty(object):
...     pass
```

Create two objects of that class

```
>>> first = Empty()
>>> second = Empty()
```

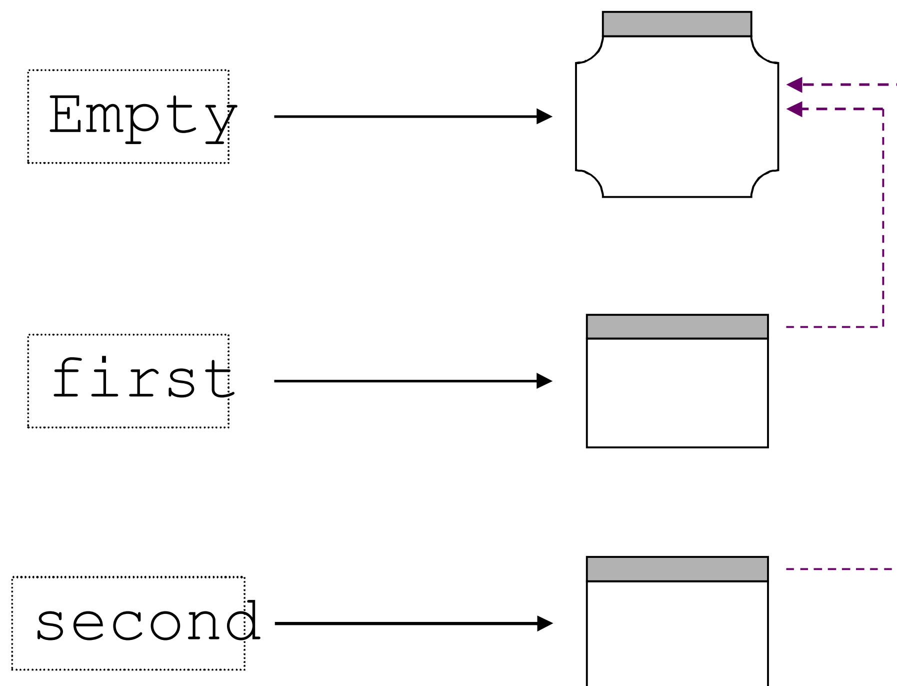
Define a new class with no behavior

```
>>> class Empty(object):
...     pass
```

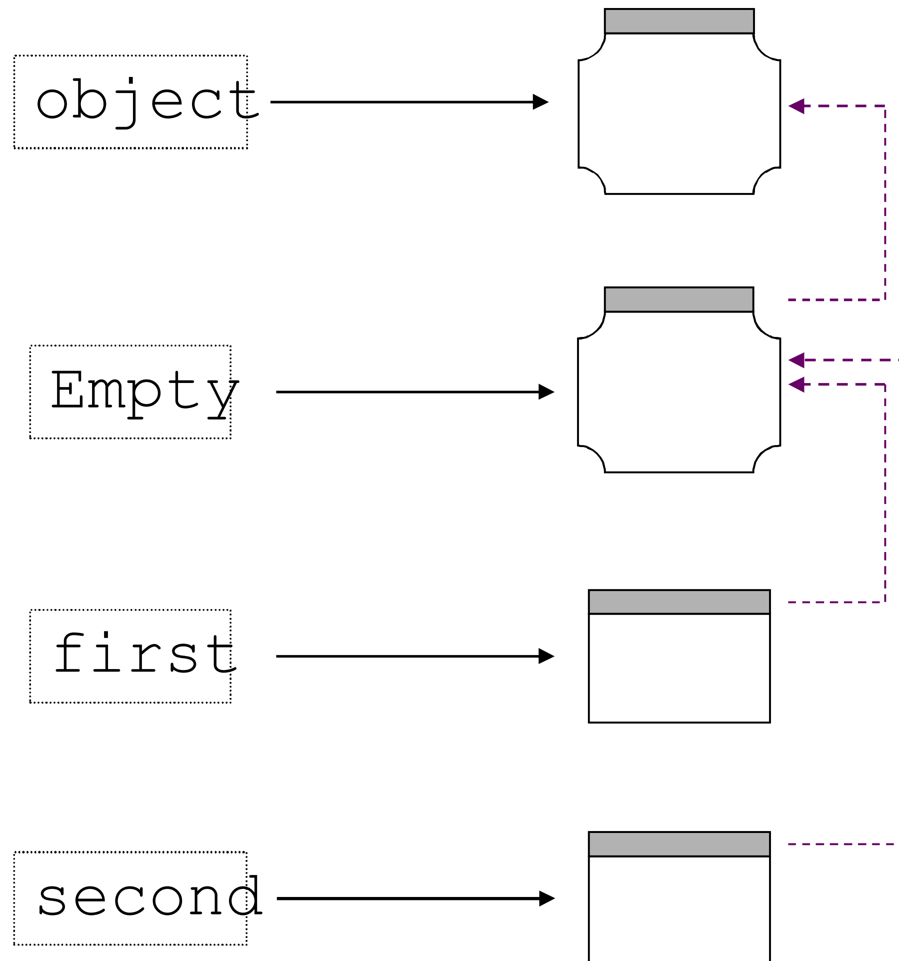
Create two objects of that class

```
>>> first = Empty()
>>> second = Empty()
>>> print 'first is', id(first)
35855140
>>> print 'second is', id(second)
35855152
```

Contents of memory



Contents of memory



Define the class's behavior with *methods*

Define the class's behavior with *methods*

A function defined inside a classõ

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'
```

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'
```

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'
```

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'

g = Greeter()
g.greet('Waya')
hello Waya !
```

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'

g = Greeter()
g.greet('Waya')
hello Waya !
```

software carpentry

Define the class's behavior with *methods*

A function defined inside a classõ

õ that is called for an object of that class

```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'

g = Greeter()
g.greet('Waya')
hello Waya !
```

Define the class's behavior with *methods*

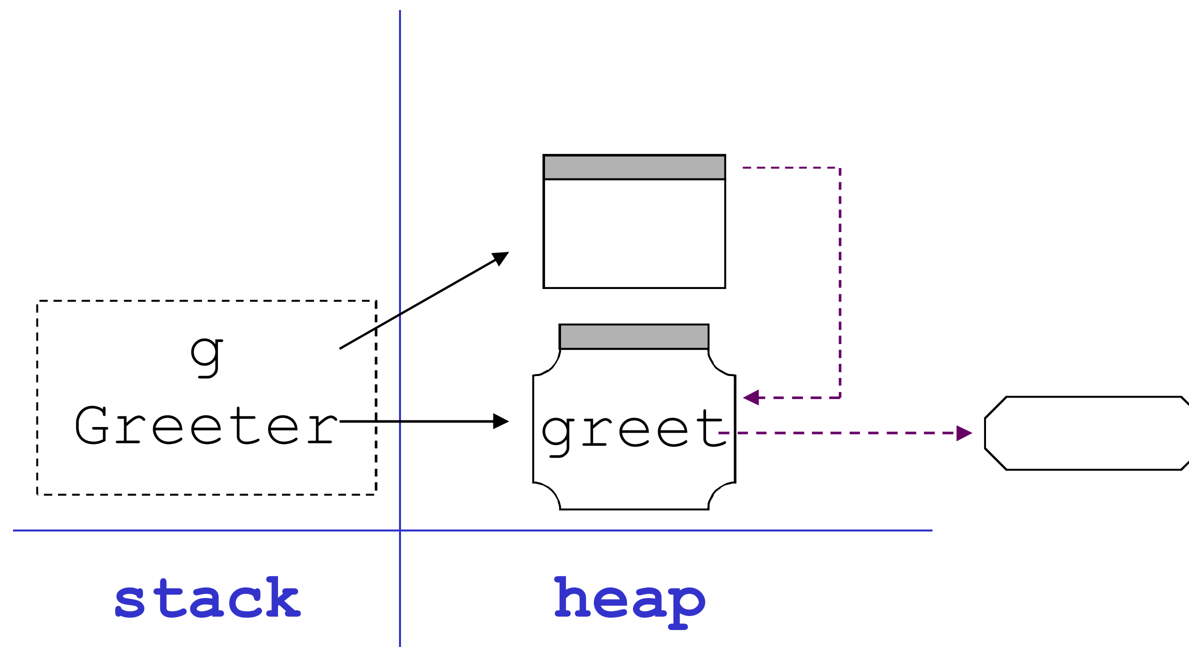
A function defined inside a classõ

õ that is called for an object of that class

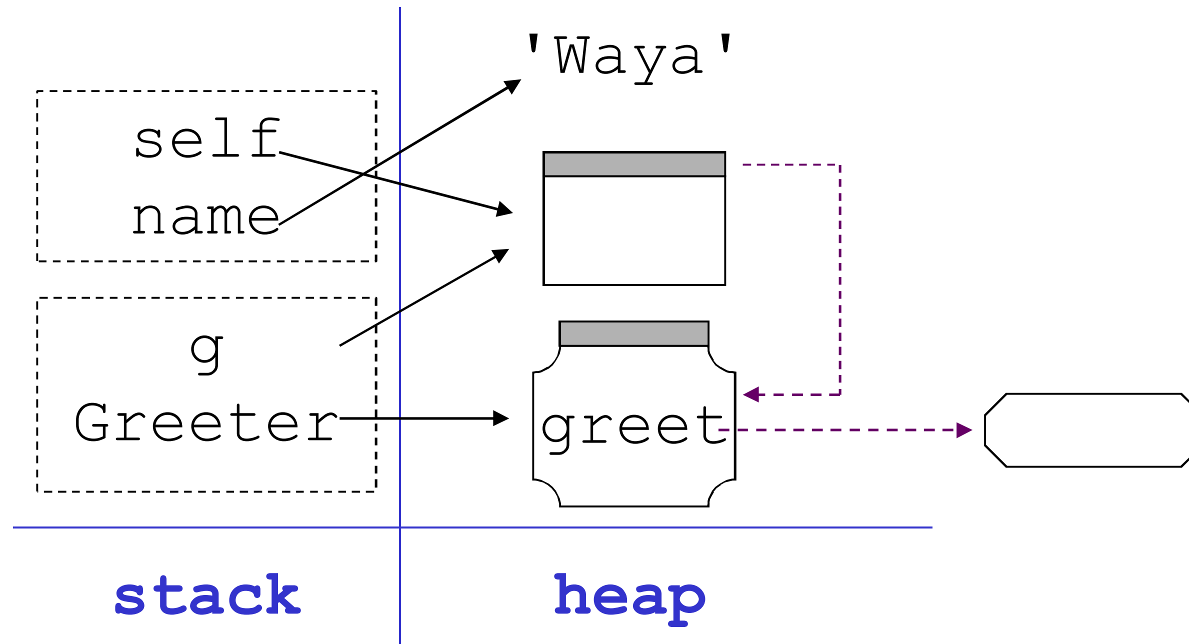
```
class Greeter(object):
  def greet(self, name):
    print 'hello', name, '!'

g = Greeter()
g.greet('Waya')
hello Waya !
```

software carpentry

Contents of memory



Contents of memory



Every object has its own variables

software carpentry

Every object has its own variables ← members

Every object has its own variables

Create new ones by assigning them values

Every object has its own variables

Create new ones by assigning them values

```
class Empty(object):
  pass

e = Empty()
e.value = 123
print e.value
123
```

software carpentry

Every object has its own variables

Create new ones by assigning them values

```
class Empty(object):
  pass

e = Empty()
e.value = 123
print e.value
123

e2 = Empty()
print e2.value
AttributeError: 'Empty' object has no attribute 'value'
```

The values of member variables customize objects

The values of member variables customize objects

Use them in methods

The values of member variables customize objects

Use them in methods

```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'
```

The values of member variables customize objects

Use them in methods

```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'
```

Every object has its own variables

Create new ones by assigning them values

```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'

g = Greeter()
```

Every object has its own variables

Create new ones by assigning them values

```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'

g = Greeter()
g.hello = 'Bonjour'
```

Every object has its own variables

Create new ones by assigning them values

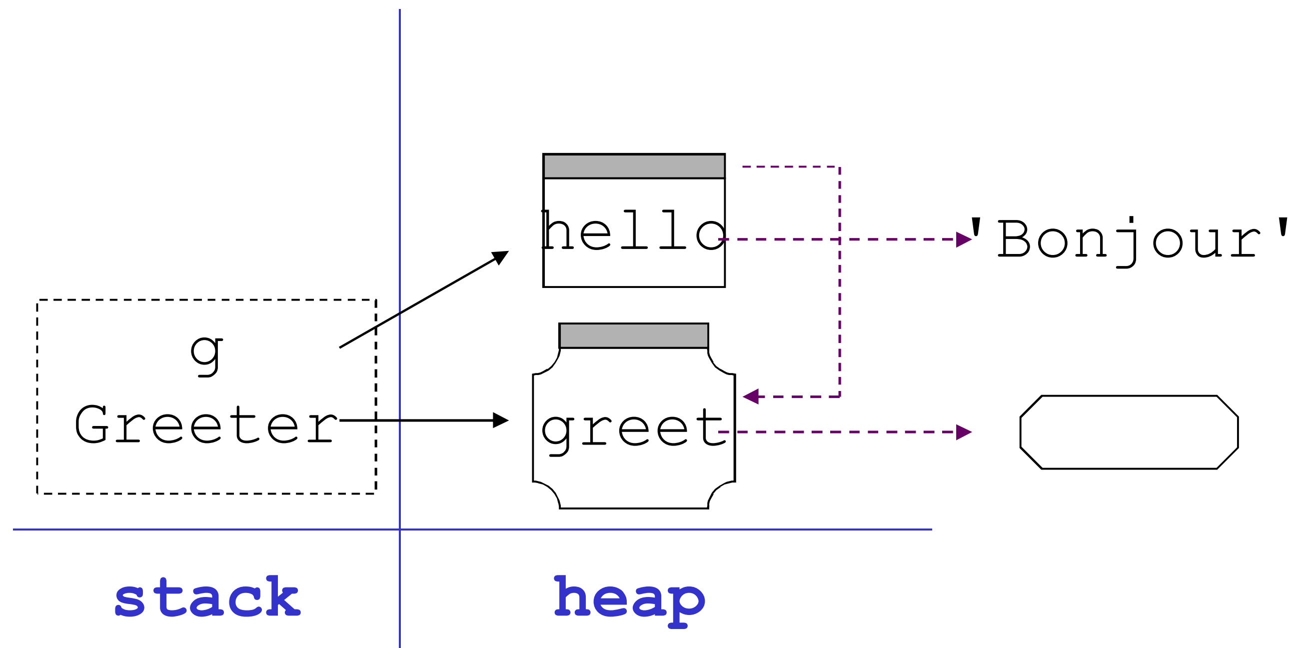
```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'

g = Greeter()
g.hello = 'Bonjour'
g.greet('Waya')
Bonjour Waya !
```

Every object has its own variables

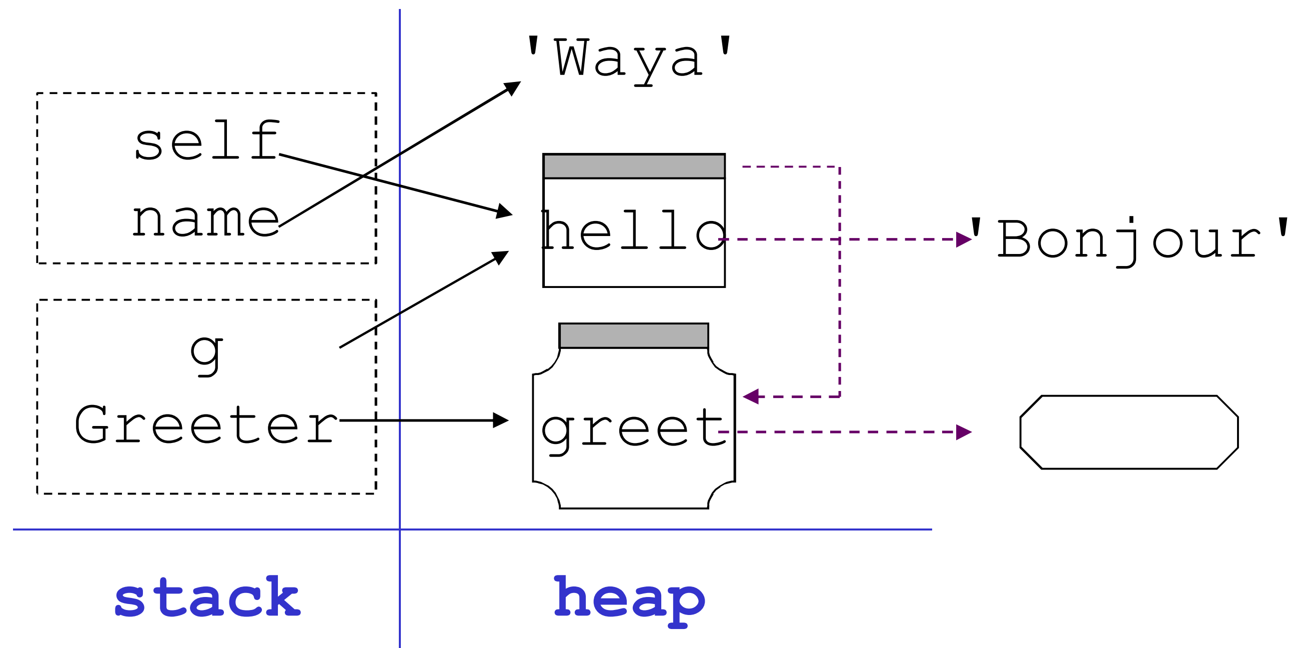
Create new ones by assigning them values

```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'

g = Greeter()                g2 = Greeter()
g.hello = 'Bonjour'          g2.hello =
  'Salut'
g.greet('Waya')              g2.greet('Waya')
Bonjour Waya !               Salut Waya !
```

Contents of memory



Contents of memory



Every object's names are separate

Every object's names are separate

```
class Greeter(object):
  def greet(self, name):
    print self.hello, name, '!'

hello = 'Hola'
g = Greeter()
g.hello = 'Bonjour'
g.greet('Waya')
Bonjour Waya !
```

Creating objects and *then* giving them members is error-prone

Creating objects and *then* giving them members is error-prone

Might forget some (especially when making changes)

Creating objects and *then* giving them members is error-prone

Might forget some (especially when making changes)

Any code repeated in two or more placesõ

Creating objects and *then* giving them members is error-prone

Might forget some (especially when making changes)

Any code repeated in two or more placesõ

Define a *constructor* for the class

Creating objects and *then* giving them members is error-prone

Might forget some (especially when making changes)

Any code repeated in two or more placesõ

Define a *constructor* for the class

Automatically called as new object is being created

Creating objects and *then* giving them members is error-prone

Might forget some (especially when making changes)

Any code repeated in two or more placesõ

Define a *constructor* for the class

Automatically called as new object is being created

A natural place to customize individual objects

software carpentry

Creating objects and *then* giving them members is error-prone

Might forget some (especially when making changes)

Any code repeated in two or more placesõ

Define a *constructor* for the class

Automatically called as new object is being created

A natural place to customize individual objects

Python uses the special name `__init__(self, ...)`

A better `Greeter`

```
class Greeter(object):

  def __init__(self, what_to_say):
    self.hello = what_to_say

  def greet(self, name):
    print self.hello, name, '!'
```

software carpentry

Why it's better

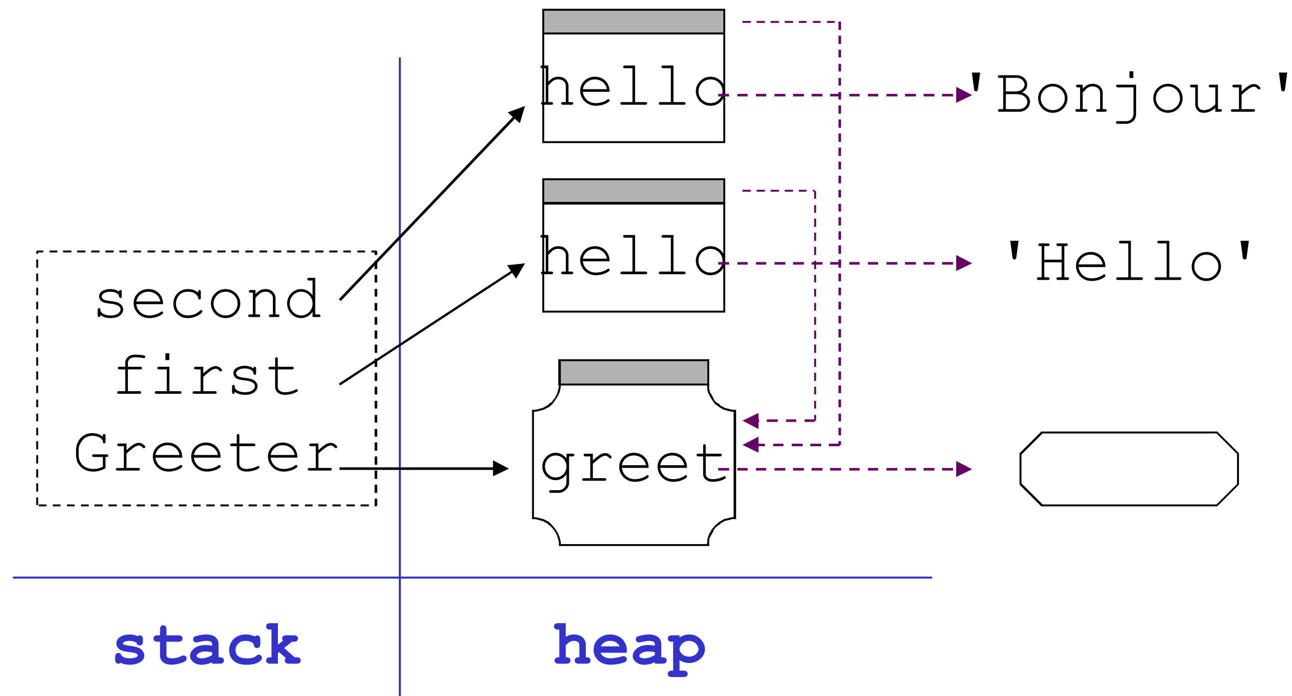
```
first = Greeter('Hello')
first.greet('Waya')
Hello Waya !
```

software carpentry

Why it's better

```
first = Greeter('Hello')
first.greet('Waya')
Hello Waya !
second = Greeter('Bonjour')
second.greet('Waya')
Bonjour Waya !
```

Contents of memory



A comon mistake

```
class Greeter(object):

  def __init__(self, what_to_say):
    hello = what_to_say

  def greet(self, name):
    print self.hello, name, '!'
```

What goes wrong

```
first = Greeter('Hello')
```

What goes wrong

```
first = Greeter('Hello')
first.greet('Waya')
```

Attribute Error: 'Greeter' object has no attribute 'hello'

What goes wrong

```
first = Greeter('Hello')
first.greet('Waya')
Attribute Error: 'Greeter' object has
   no
attribute 'hello'
```

`self.name` stores the value in the object

What goes wrong

```
first = Greeter('Hello')
first.greet('Waya')
Attribute Error: 'Greeter' object has
   no
attribute 'hello'
```

`self.name` stores the value in the object

`name` on its own is a local variable on the stack

What goes wrong

```
first = Greeter('Hello')
first.greet('Waya')
Attribute Error: 'Greeter' object has
  no
attribute 'hello'
```

`self.name` stores the value in the object

`name` on its own is a local variable on the stack

```
class Greeter(object):
  def __init__(self, what_to_say)
    hello = what_to_say
```

software carpentry

Object data is not protected or hidden in Python

Object data is not protected or hidden in Python

```
first = Greeter('Hello')
first.greet('Waya')
Hello Waya !
first.hello = 'Kaixo'
Kaixo Waya !
```

Object data is not protected or hidden in Python

```
first = Greeter('Hello')
first.greet('Waya')
Hello Waya !
first.hello = 'Kaixo'
Kaixo Waya !
```

Some languages prevent this

Object data is not protected or hidden in Python

```
first = Greeter('Hello')
first.greet('Waya')
Hello Waya !
first.hello = 'Kaixo'
Kaixo Waya !
```

Some languages prevent this

All discourage it

A more practical example

```
class Rectangle(object):

  def __init__(self, x0, y0, x1, y1):
    assert x0 < x1, 'Non-positive X 
  extent'
    assert y0 < y1, 'Non-positive Y 
  extent'
    self.x0 = x0
    self.y0 = y0
    self.x1 = x1
    self.y1 = y1
```

A more practical example

```
class Rectangle(object):

  def __init__(self, x0, y0, x1, y1):
    assert x0 < x1, 'Non-positive X extent'
    assert y0 < y1, 'Non-positive Y extent'
    self.x0 = x0
    self.y0 = y0
    self.x1 = x1
    self.y1 = y1
```

Benefit #1: fail early, fail often

Benefit #1: fail early, fail often

```
# Programmer thinks rectangles are
  written
# [[x0, x1], [y0, y1]]
>>> field = [[50, 100], [0, 200]]
```

Benefit #1: fail early, fail often

```
# Programmer thinks rectangles are
  written
# [[x0, x1], [y0, y1]]
>>> field = [[50, 100], [0, 200]]
>>>

# Class knows rectangles are (x0, y0,
  x1, y1)
>>> field = Rectangle(50, 100, 0, 200)
AssertionError: non-positive X extent
```

Benefit #2: readability

```
class Rectangle(object):
  ...

  def area(self):
    return (self.x1-self.x0)*(self.y1-
  self.y0)

  def contains(self, x, y):
    return (self.x0 <= x <= self.x1)
  and \
           (self.y0 <= y <= self.y1)
```

software carpentry

Compare

List of Lists	Object
`field = [[0, 0], [100, 100]]`	`field = Rectangle(0, 0, 100, 100)`
`rect_area(field)`	`field.area()`
`rect_contains(field, 20, 25)`	`field.contains(20, 25)`

Compare

List of Lists	Object
`field = [[0, 0], [100, 100]]`	`field = Rectangle(0, 0, 100, 100)`
`rect_area(field)`	`field.area()`
`rect_contains(field, 20, 25)`	`field.contains(20, 25)`

Make it even clearer by creating a `Point2D` class

Compare

List of Lists	Object
`field = [[0, 0], [100, 100]]`	`field = Rectangle(0, 0, 100, 100)`
`rect_area(field)`	`field.area()`
`rect_contains(field, 20, 25)`	`field.contains(20, 25)`

Make it even clearer by creating a `Point2D` class

Then re-defining `Rectangle` in terms of it

created by

Greg Wilson

January 2011

Copyright © Software Carpentry 2010

This work is licensed under the Creative Commons Attribution License

See http://software-carpentry.org/license.html for more information.