Classes, Instances, and Methods

Outline

- User-defined types
  - defining classes
  - creating instances
  - invoking methods on instances
- Polymorphism
  - __str__ and __cmp__
- Inheritance

User-defined types

- You do not have to stick with only the built-in Python data types (strings, lists, dictionaries)
- You can define your own types!
  - Can compose from other types
  - Can define both data and operations on that data
  - Can represent more abstract concepts
- These type definitions are called **classes**
- Objects of these types are called **instances**

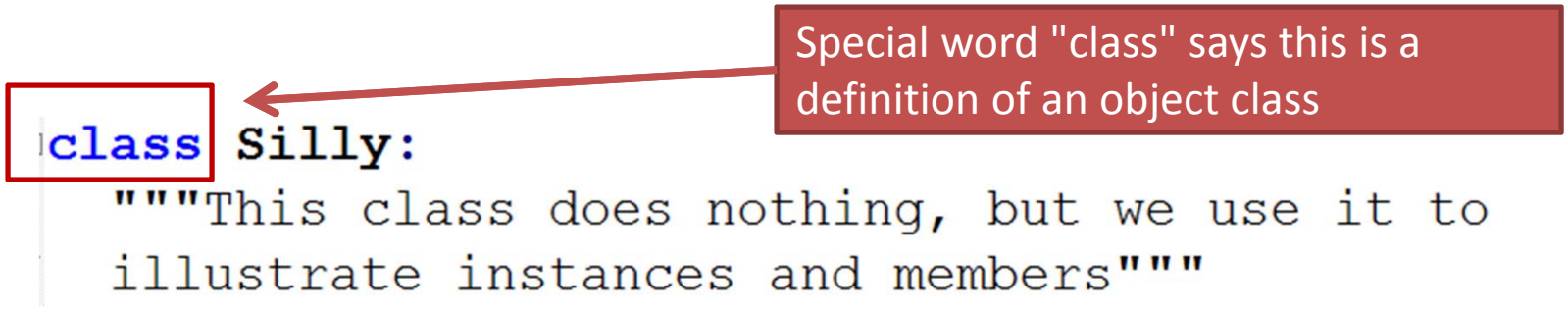
Why define your own types?

- Represent higher-level concepts
  - E.g., student, lecture, university, department
- Modular design
  - Encapsulate data and operations on data together
  - You don't have to know how a type works to use it
    - E.g, you can use dictionaries without knowing hashing
  - Makes it easier to write code in groups
  - Makes it easier to change implementations

A (Minimal) Class Definition

```
class Silly:
    """This class does nothing, but we use it to
    illustrate instances and members"""
```

A (Minimal) Class Definition

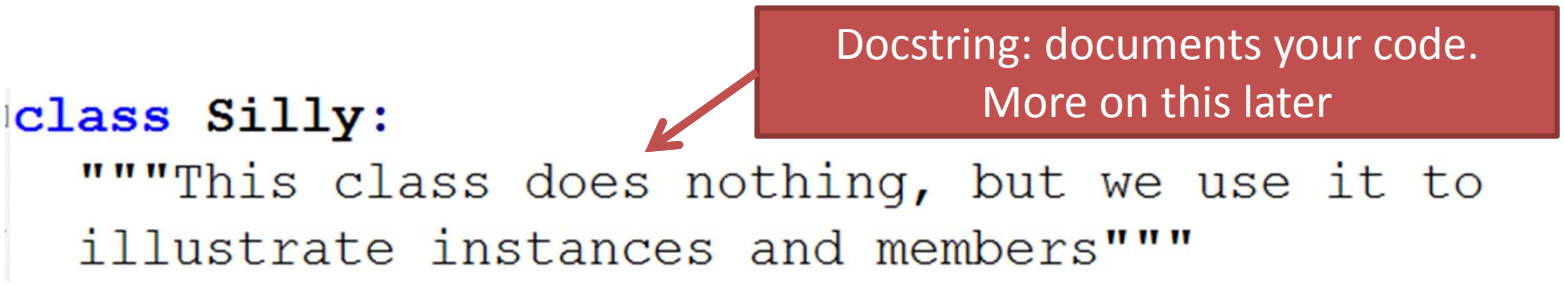


A (Minimal) Class Definition

The class name. Just a variable name, to which the class object will be bound.

```
class Silly:
    """This class does nothing, but we use it to
    illustrate instances and members"""
```

A (Minimal) Class Definition

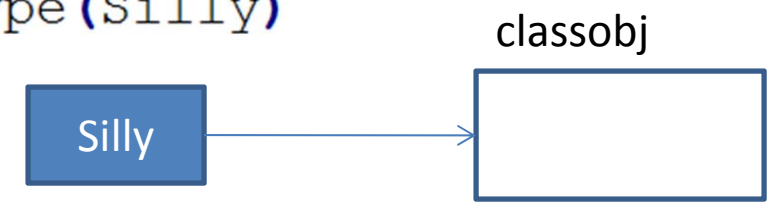


A (Minimal) Class Definition

```
class Silly:
    """This class does nothing, but we use it to
    illustrate instances and members"""
```

A (Minimal) Class Definition

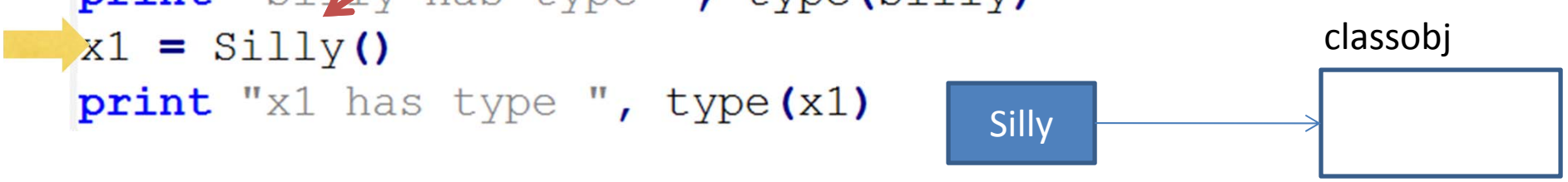
```
class Silly:
    """This class does nothing, but we use it to
    illustrate instances and members"""

print "Silly has type ", type(Silly)
```



An Instance

```
class Silly:
    """This class does
    illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
```

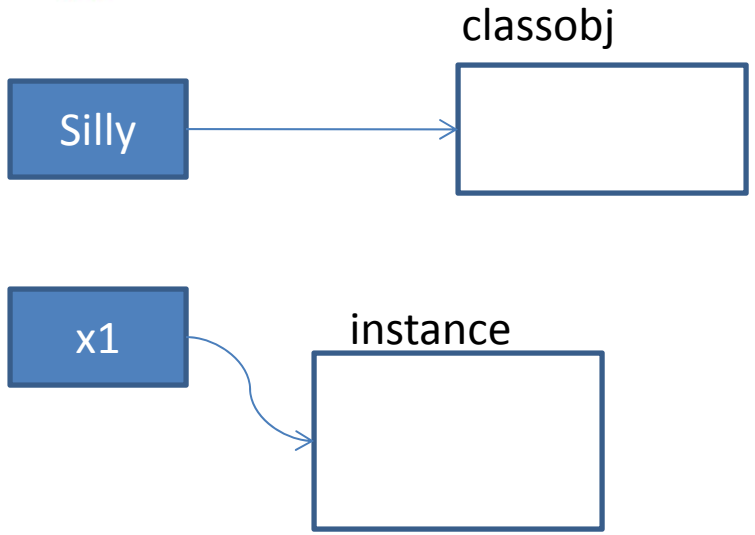
Creates a new instance of the class




```
Silly has type  <type 'classobj'>
```

An Instance

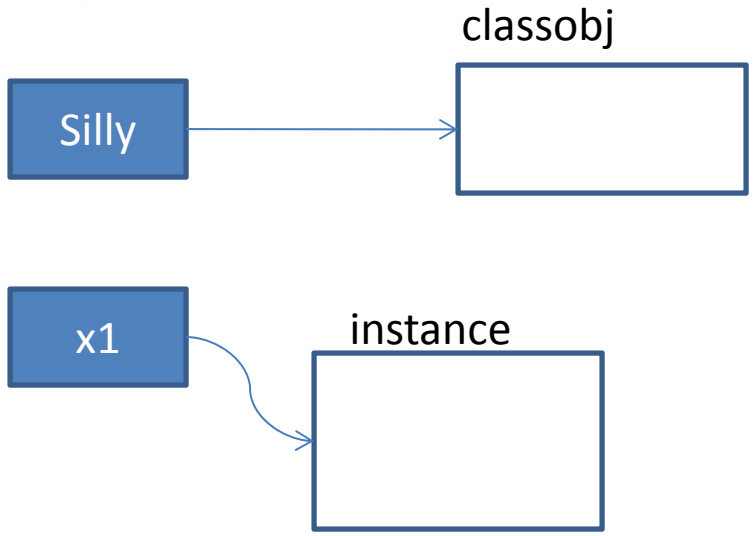
```
class Silly:
    """This class does nothing, but we use it to
    illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
```




```
Silly has type  <type 'classobj'>
```

Setting/Getting Members

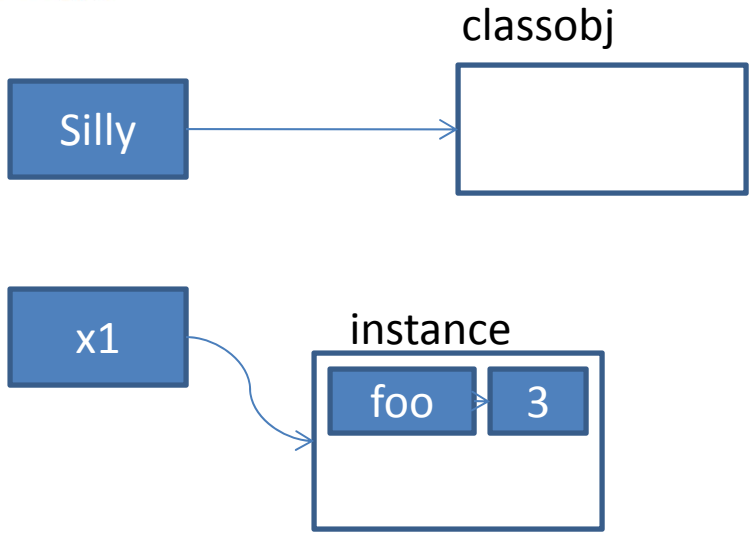
```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
```




```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
```

Setting/Getting Members

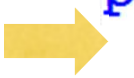
```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
```

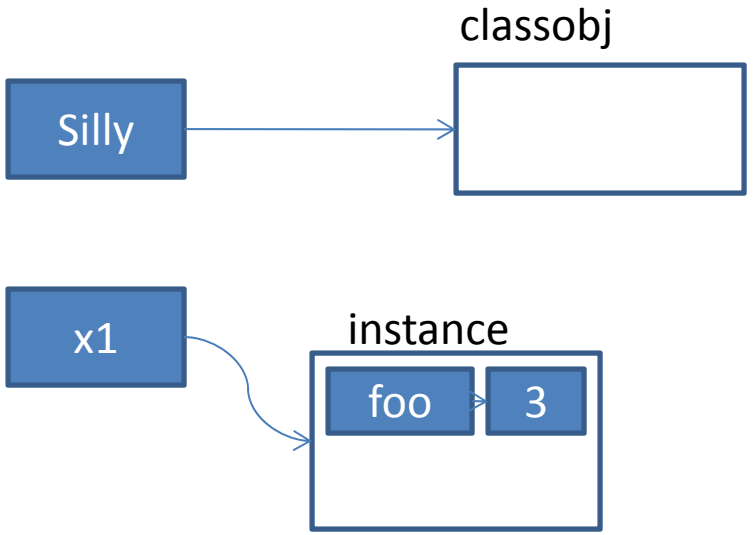



```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
```

Setting/Getting Members

```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
```



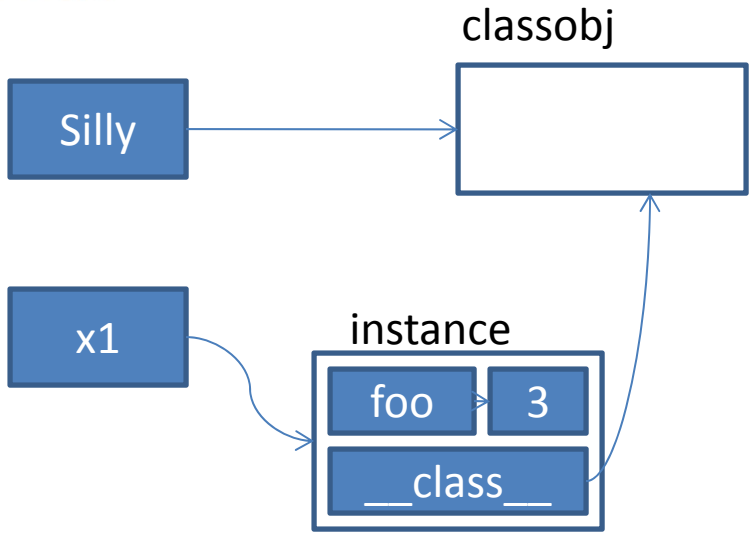



```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
```

Finding the Class

```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
```

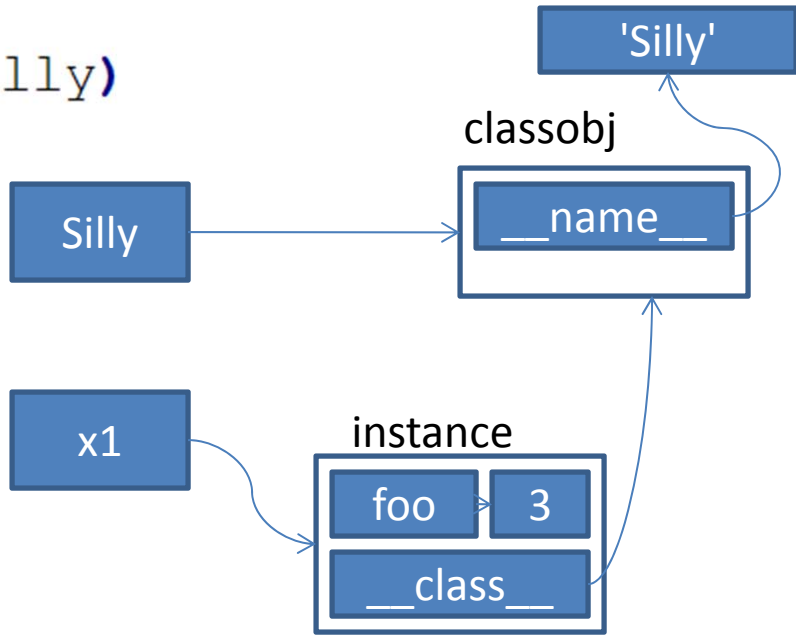
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
```



Finding the Class

```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
```

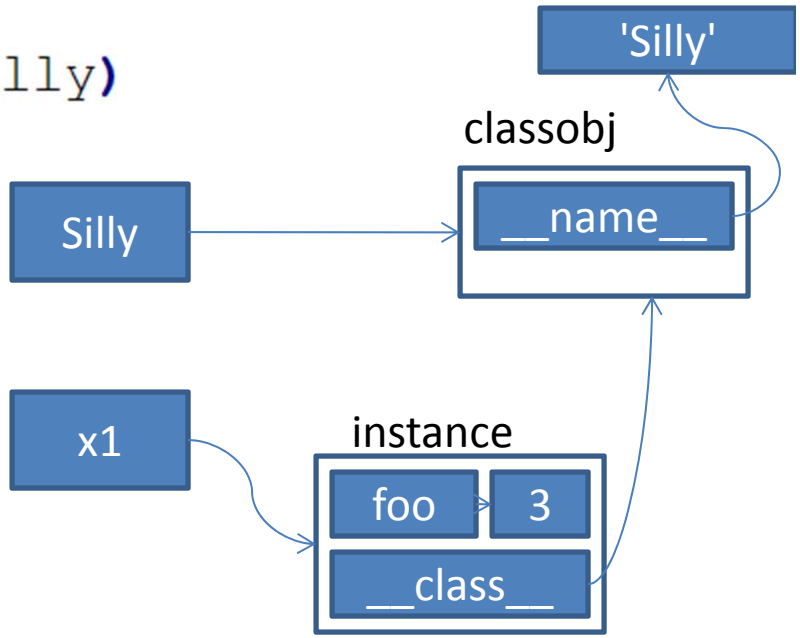
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
```



Finding the Class

```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
```

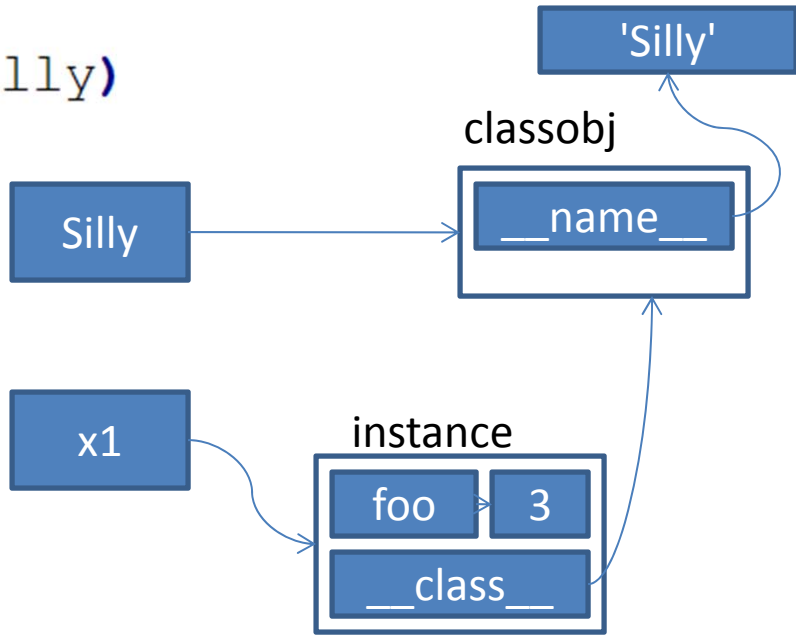
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
```



Finding the Class

```
class Silly:
  """This class does nothing, but we use it to
  illustrate instances and members"""

print "Silly has type ", type(Silly)
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
```

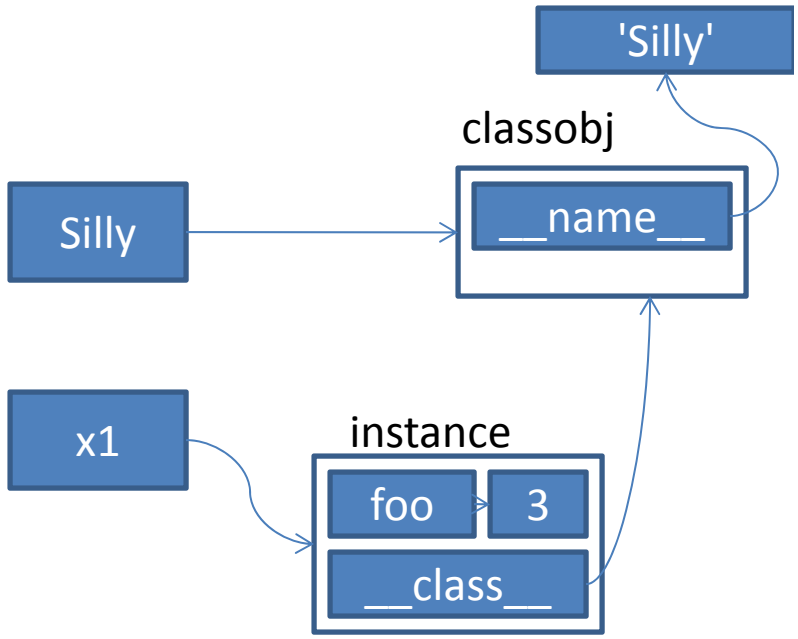
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
True
```



Another Instance

```
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
x2 = Silly()
x2.foo = 4
print x2.foo
print x1.foo
```

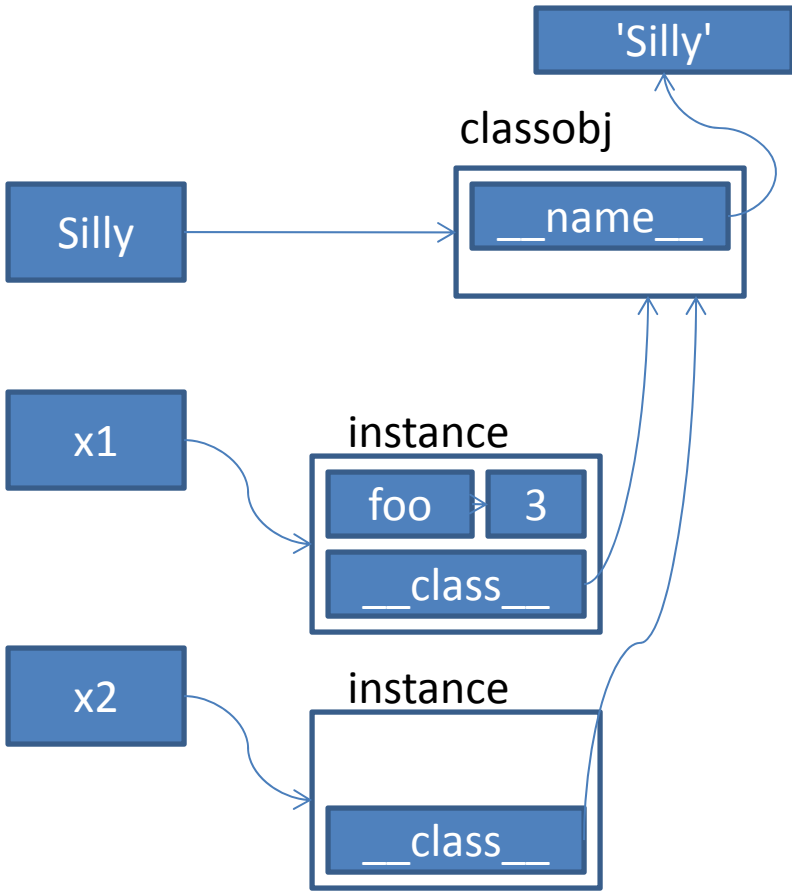
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
True
```



Another Instance

```
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
x2 = Silly()
x2.foo = 4
print x2.foo
print x1.foo
```

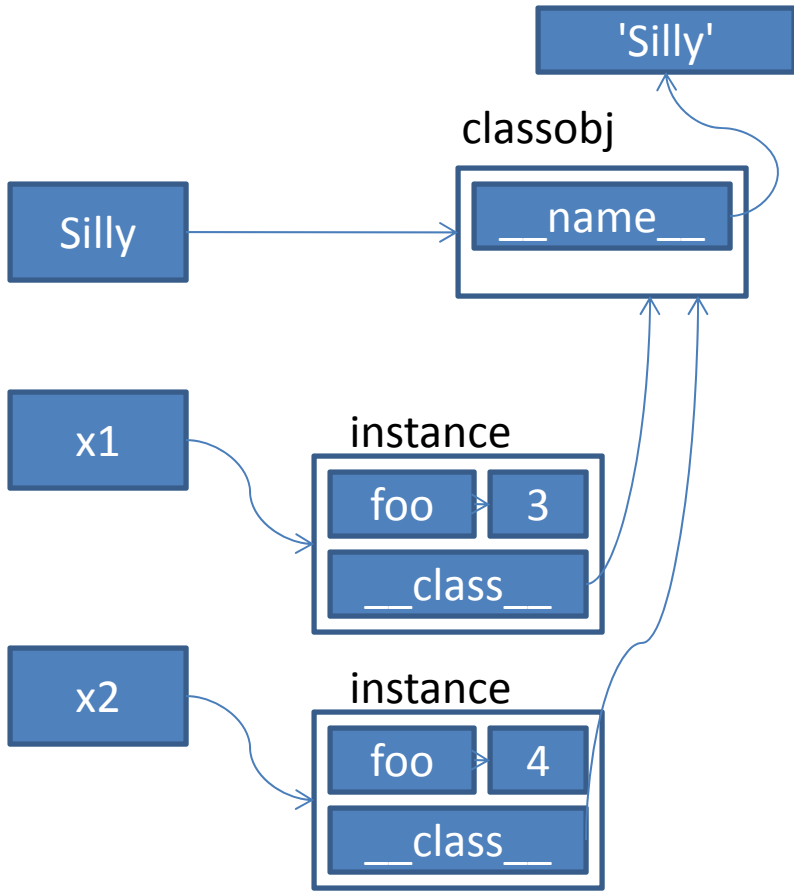
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
True
```



Another Instance

```
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
x2 = Silly()
x2.foo = 4
print x2.foo
print x1.foo
```

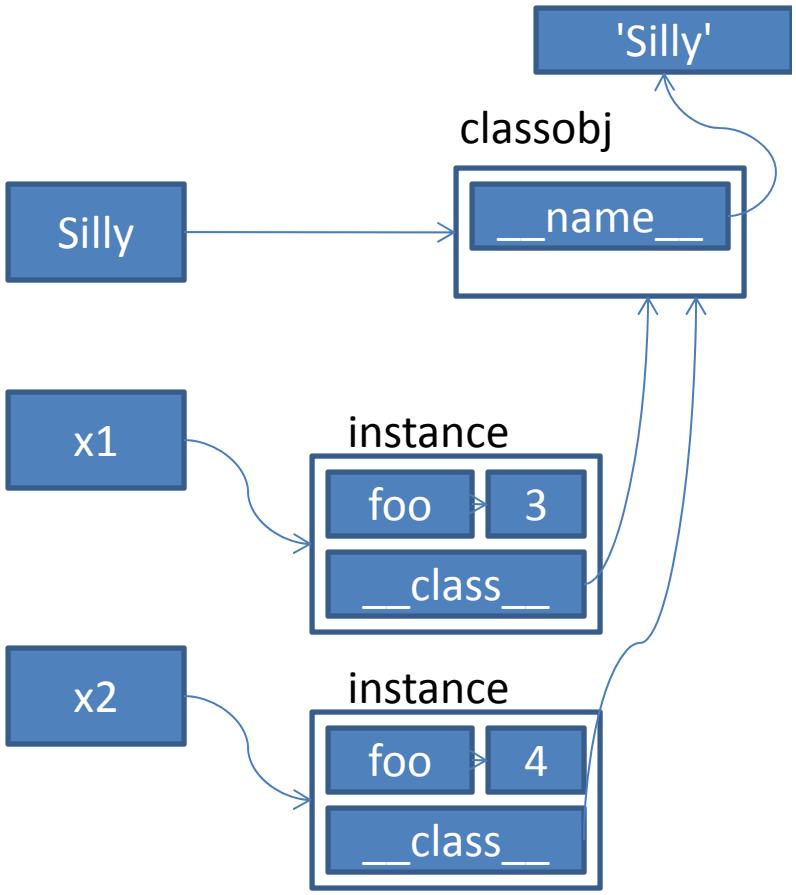
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
True
```



Another Instance

```
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
x2 = Silly()
x2.foo = 4
print x2.foo
print x1.foo
```

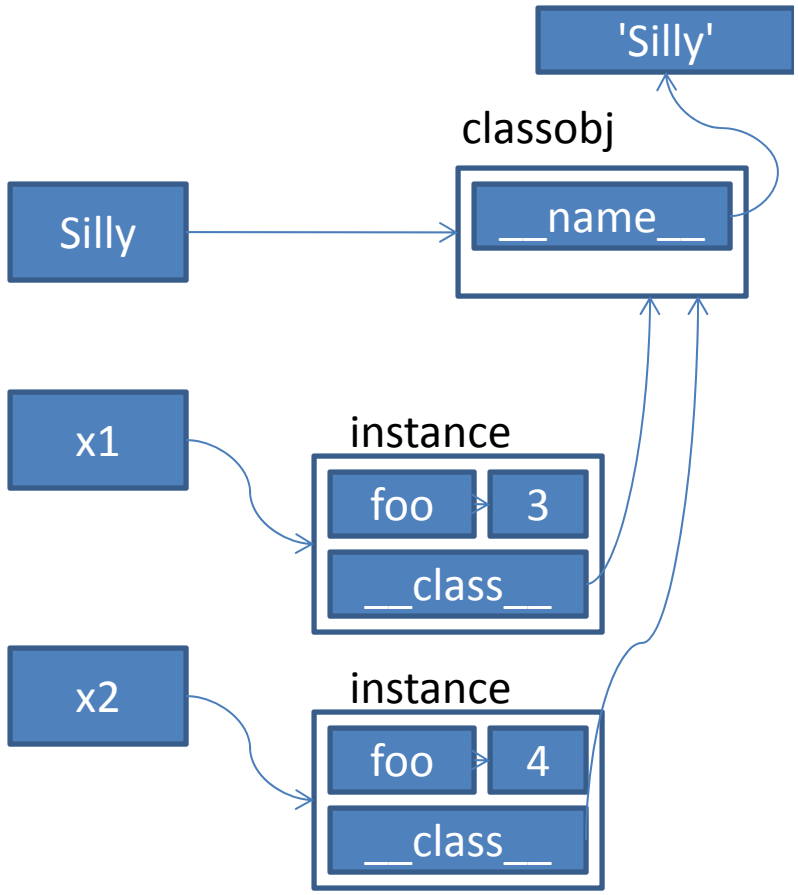
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
True
4
```



Another Instance

```
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
x2 = Silly()
x2.foo = 4
print x2.foo
print x1.foo
```

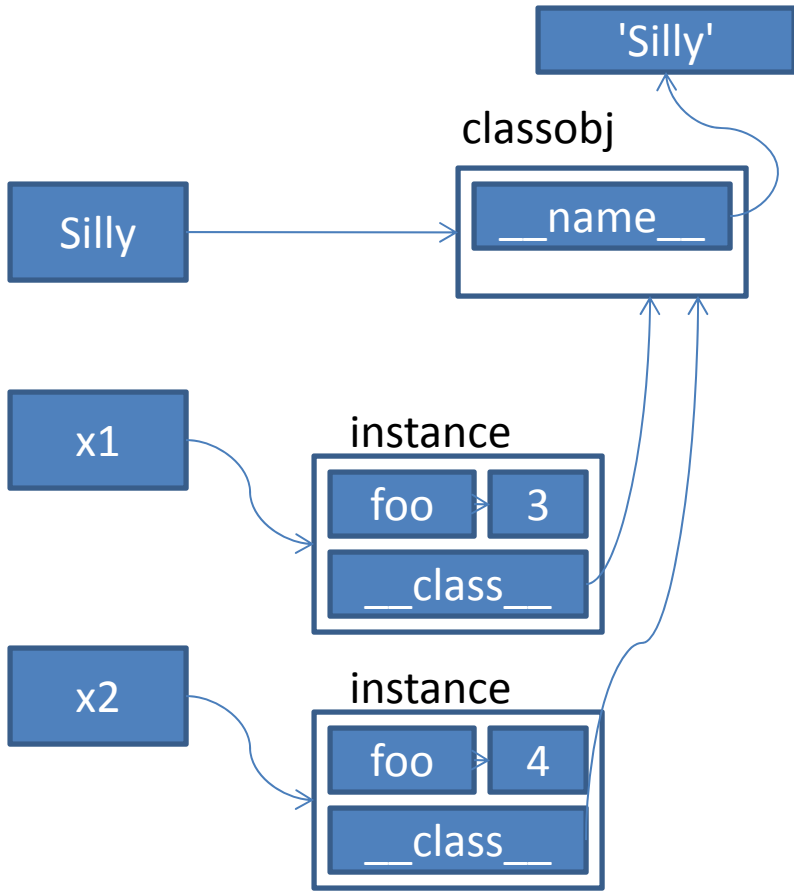
```
Silly has type  <type 'classobj'>
x1 has type  <type 'instance'>
3
Silly
True
4
3
```



CYU: What will print?

```
x1 = Silly()
print "x1 has type ", type(x1)
x1.foo = 3
print x1.foo
print x1.__class__.__name__
print x1.__class__ == Silly
x2 = Silly()
x2.foo = 4
print x2.foo
print x1.foo
```

```
16 print Silly.__name__
17 print Silly.foo
```



Summary So Far

- class <Classname>: statement creates a class object
  - By convention we capitalize class names
- <Classname>() creates an instance object
- Instances are mutable objects
- Each instance has its own local namespace
  - Can get/set "members" of the instance

Classes vs. Dictionaries

- Name for the class
- Inst. vars (members)
  - Variable names only
  - <inst>.varname
- Shared class variables
  - __name__
  - …
- Methods
- Inheritance between classes

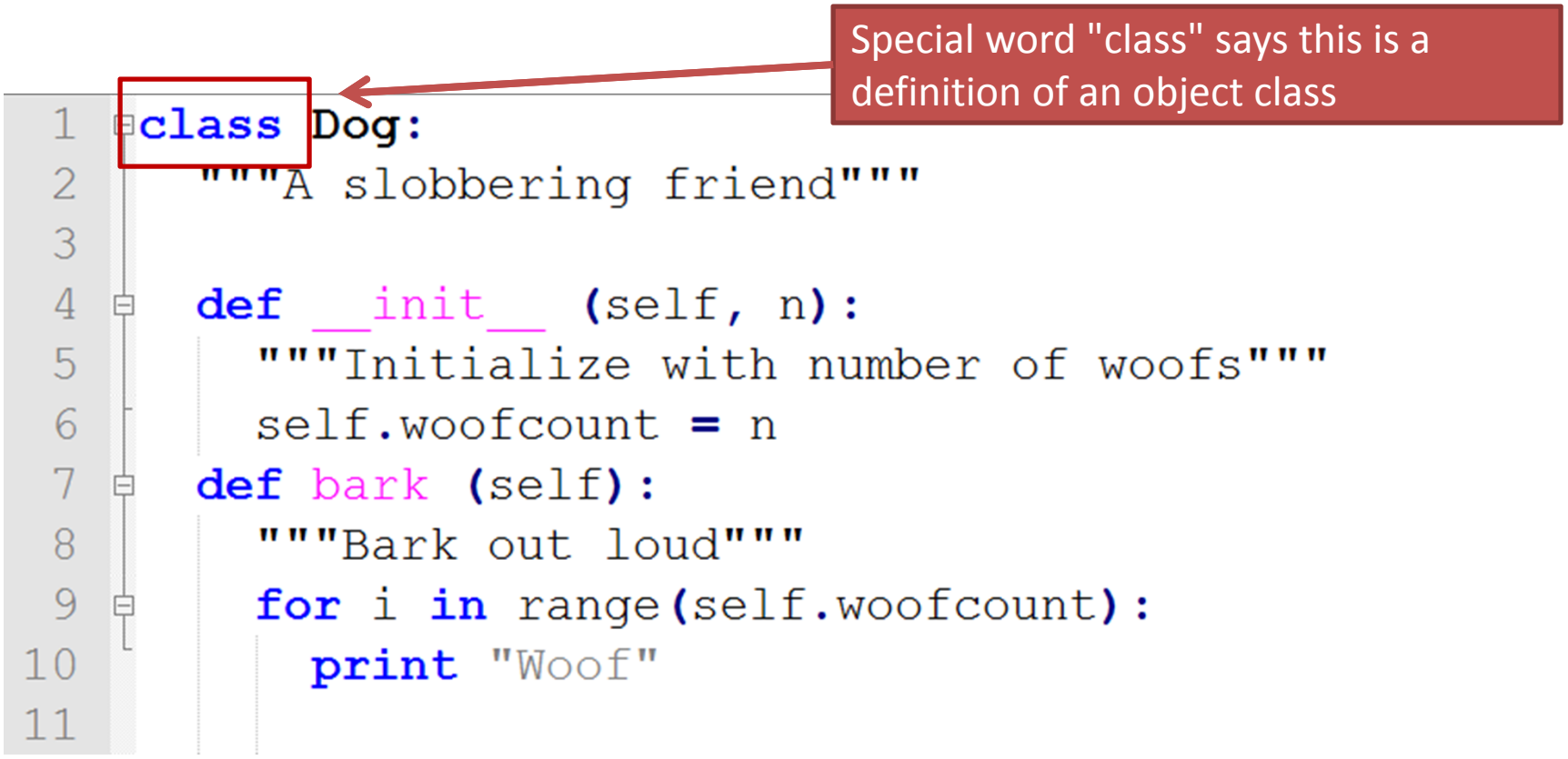
- No name
- Keys
  - Any immutable object
  - <diction>[<key>]
- Nothing shared

Methods

- Definition for the user-defined type **Dog:**

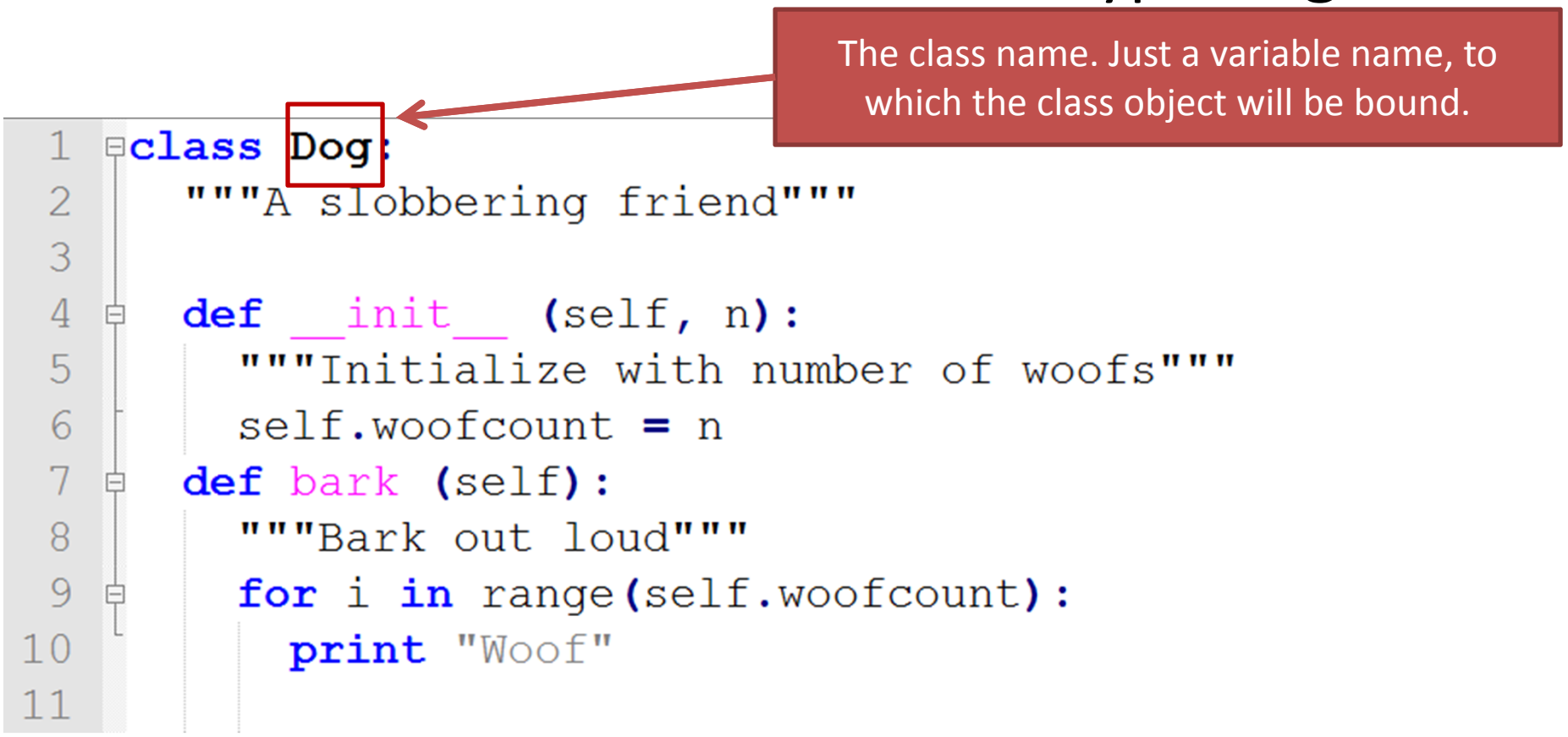
Methods

- Definition for the user-defined type **Dog:**



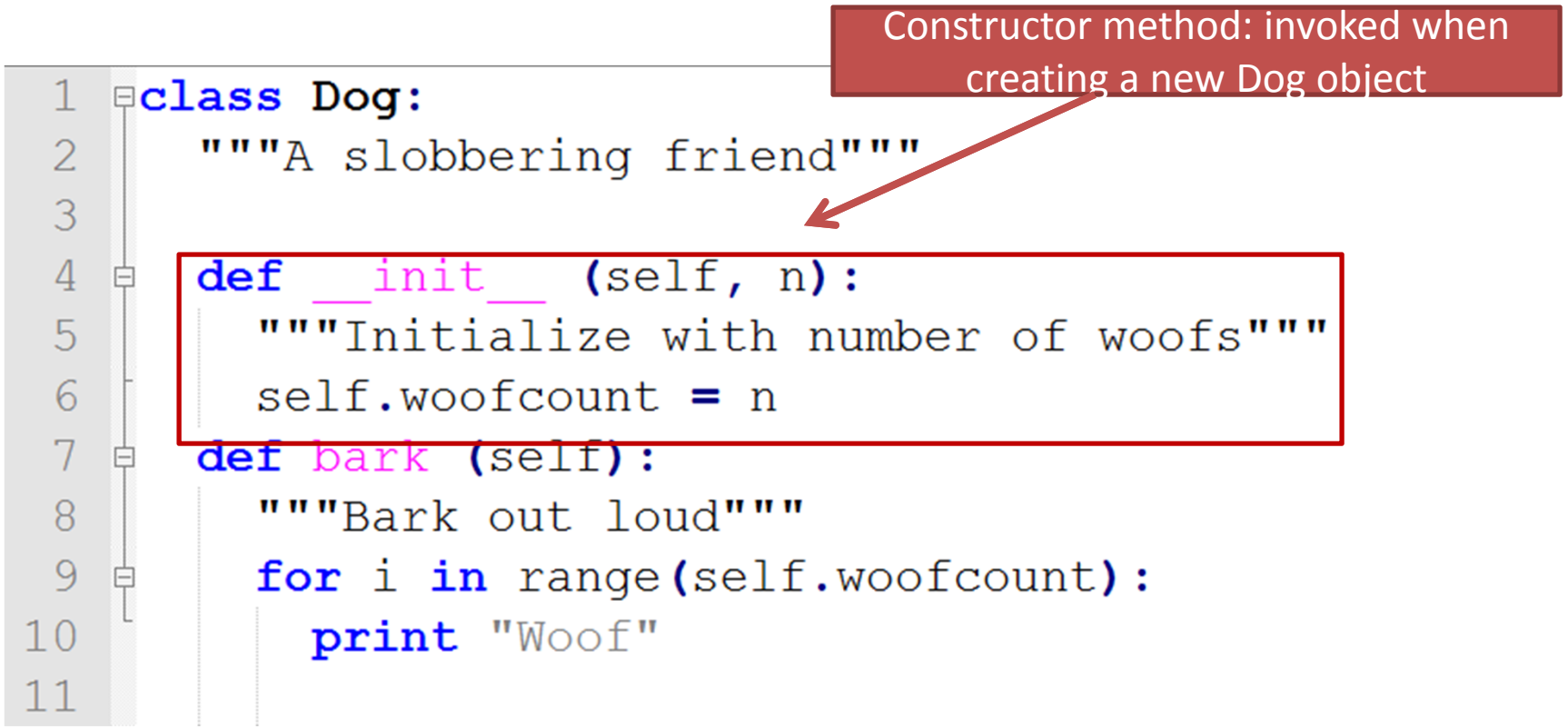
Methods

- Definition for the user-defined type **Dog:**



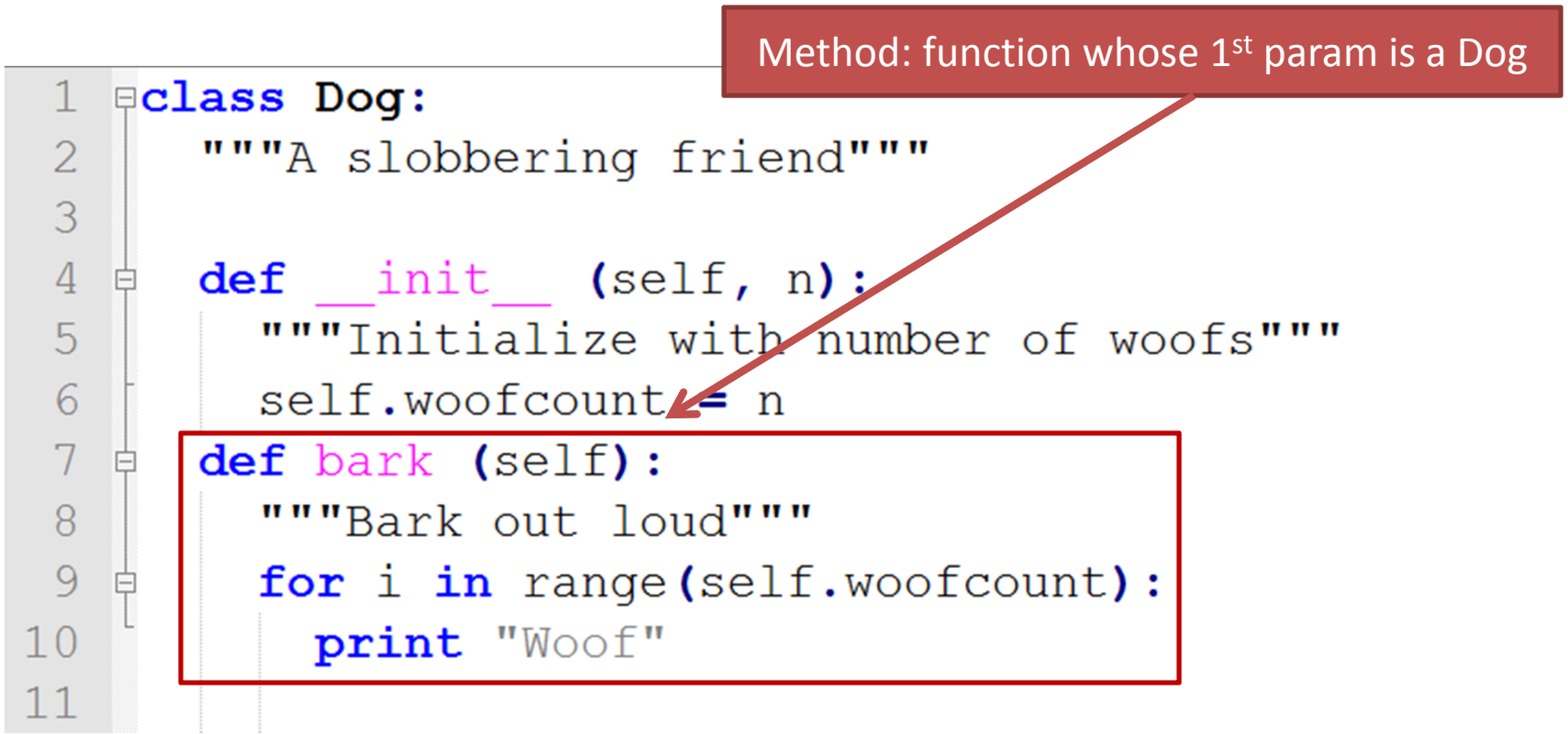
Methods

- Definition for the user-defined type **Dog:**



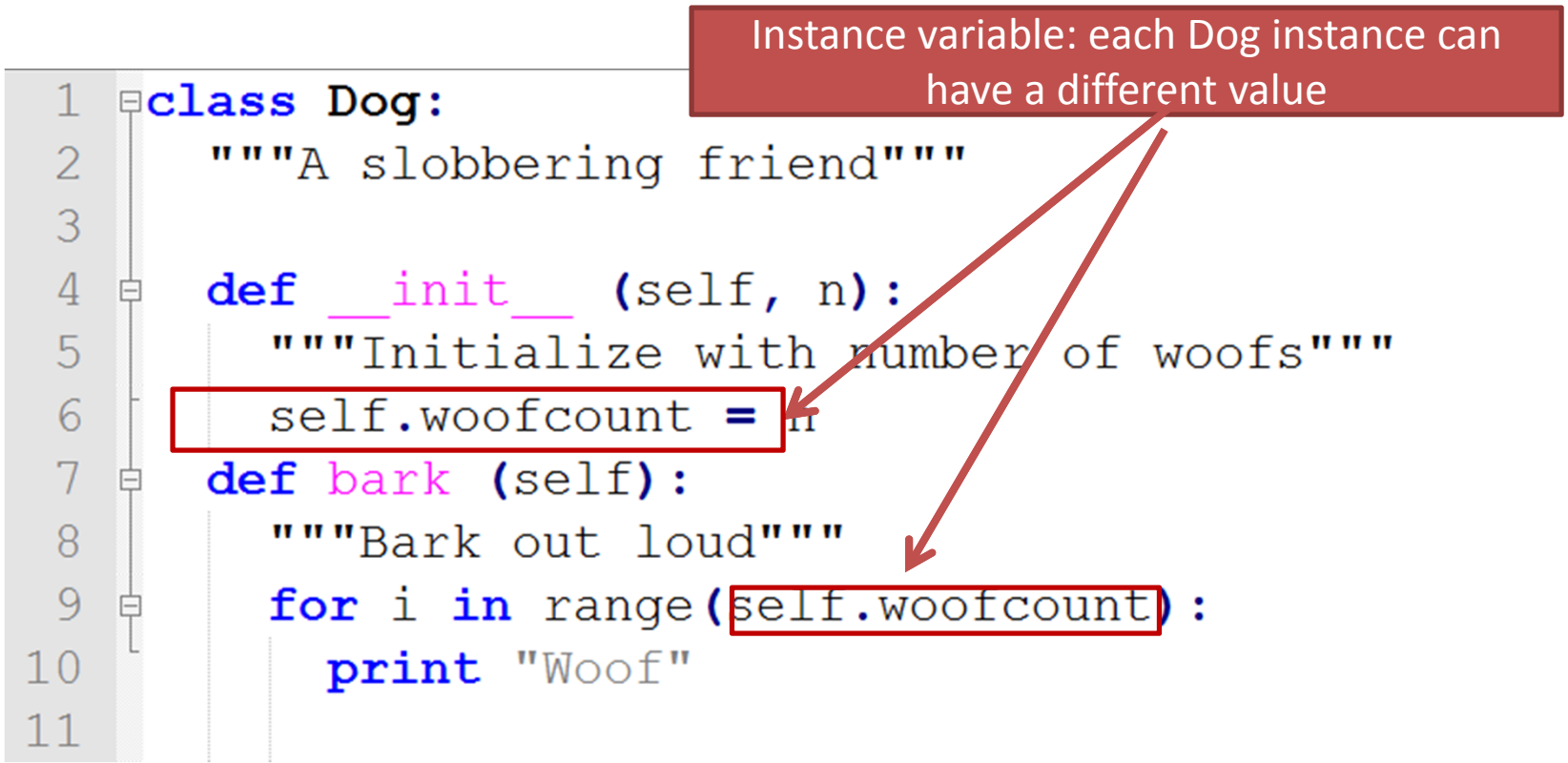
Methods

- Definition for the user-defined type **Dog:**



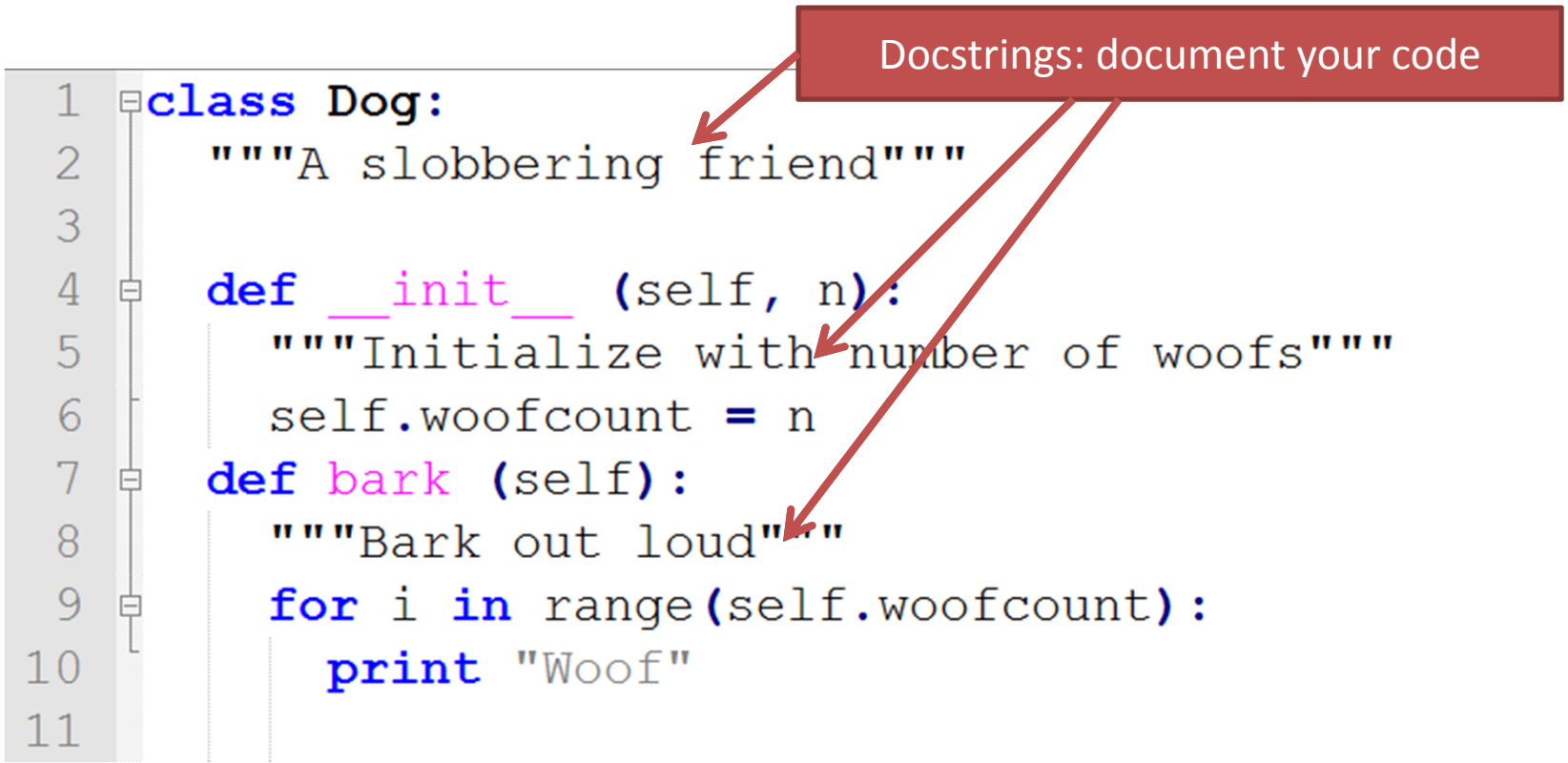
Methods

- Definition for the user-defined type **Dog:**



Methods

- Definition for the user-defined type **Dog:**



Docstrings

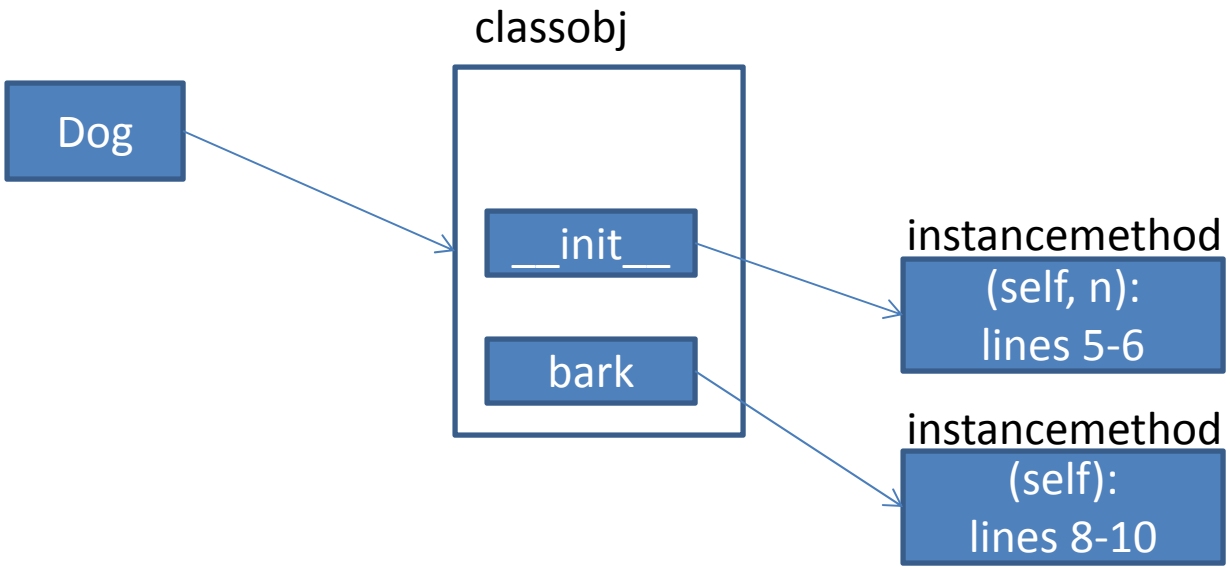
- At the top of a class (or a method)
  - A string is treated as documentation
  - """Text in triple quotes
    delimits a multiline string"""
    - Could use regular quotes for docstrings, if no line breaks
    - Could use triple quotes for multi-line strings elsewhere
- Can access these docstrings in programs
- More usefully, pydoc.py module
  - Go to directory where module is, and invoke pydoc
    - May require finding the path to pydoc.py

```
python /c/Python27/Lib/pydoc.py session18
```

The Dog Class Object

```
38 print "Dog has type", type(Dog)
39 print "Dog.bark has type", type(Dog.bark)
```

```
Dog has type <type 'classobj'>
Dog.bark has type <type 'instancemethod'>
```



Creating an Instance

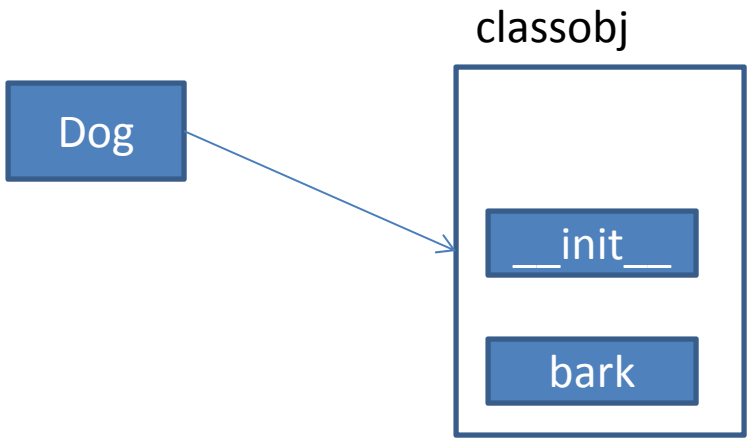
- Dog(expr)
  - Create a new instance of class Dog
  - Evaluate expr
  - Call the __init__ method
    - bind first parameter (self) to the new instance
    - bind second parameter to value of expr

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n
```

  - Value of the whole expression, Dog(expr), is the new instance object

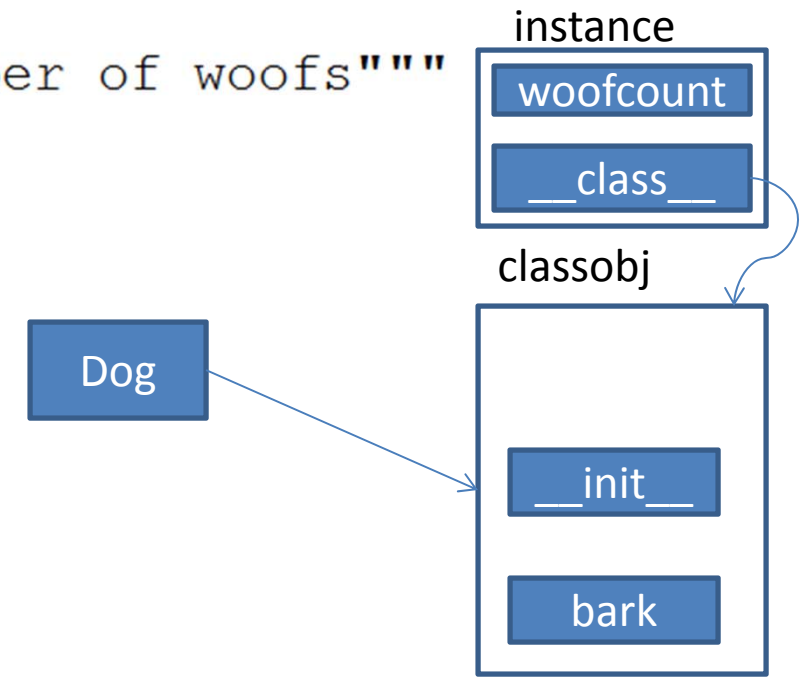
Creating an Instance

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n

Iorek = Dog(23)
```



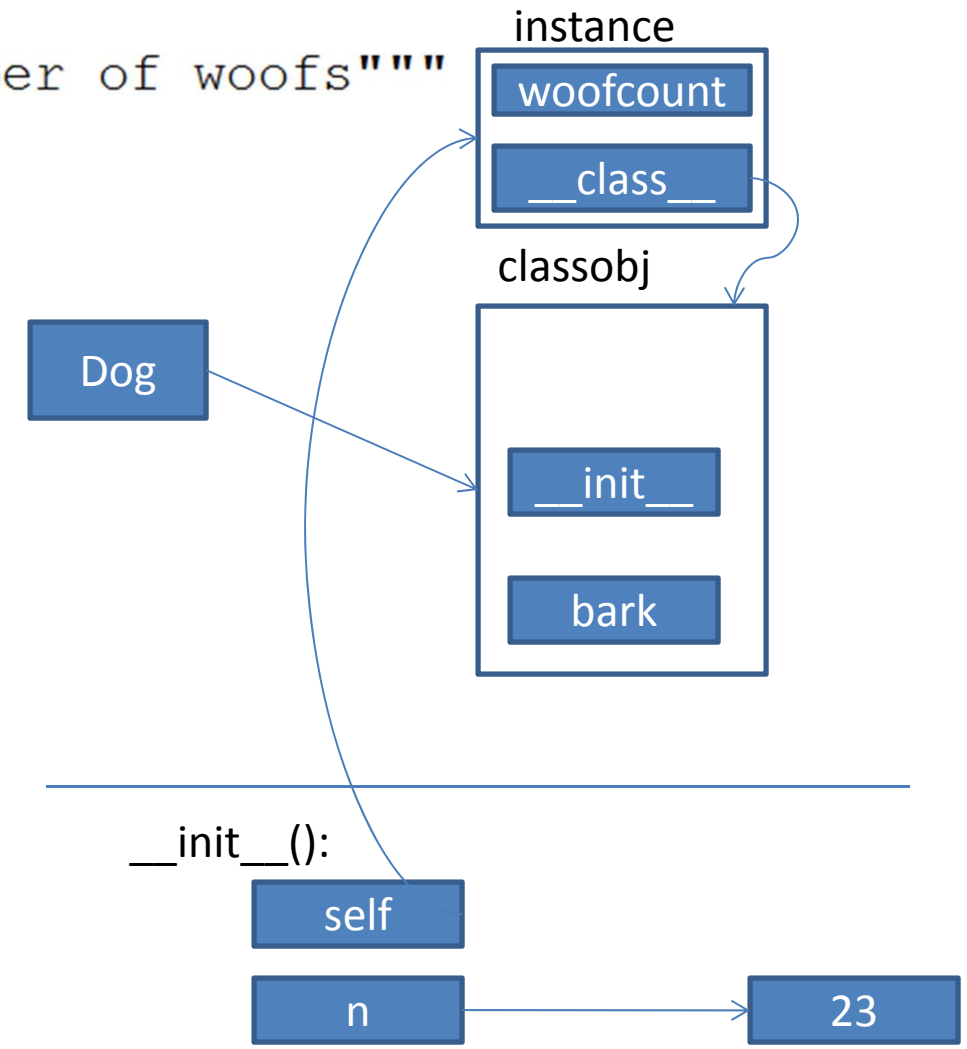
Creating an Instance

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n

Iorek = Dog(23)
```



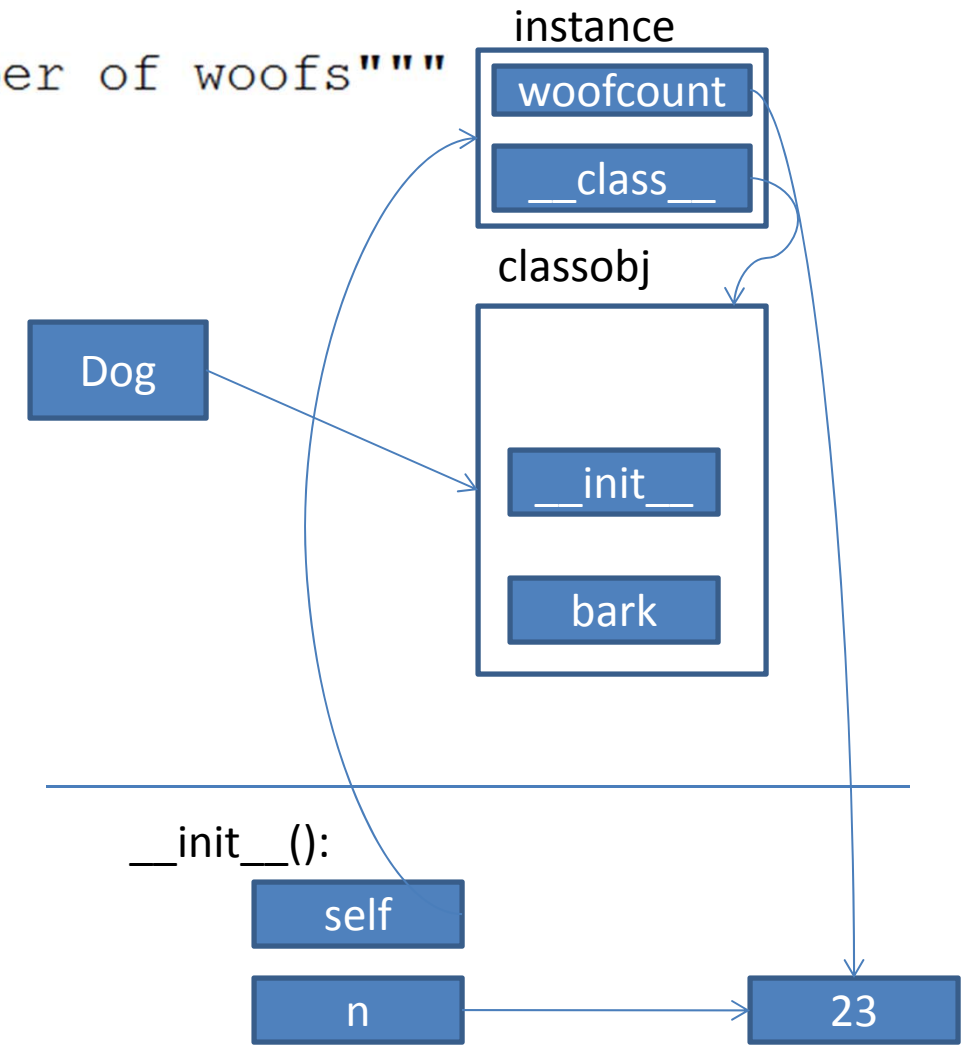
Creating an Instance

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n

Iorek = Dog(23)
```



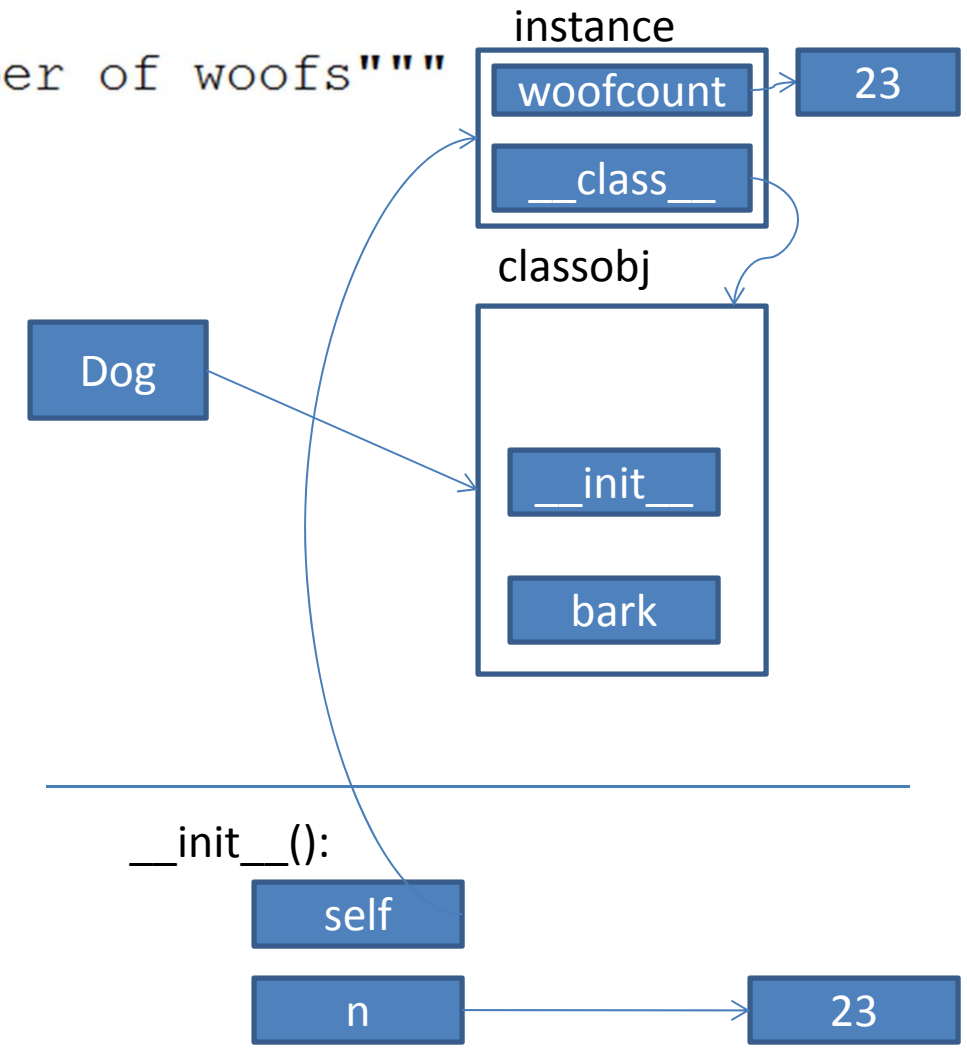
Creating an Instance

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n

Iorek = Dog(23)
```



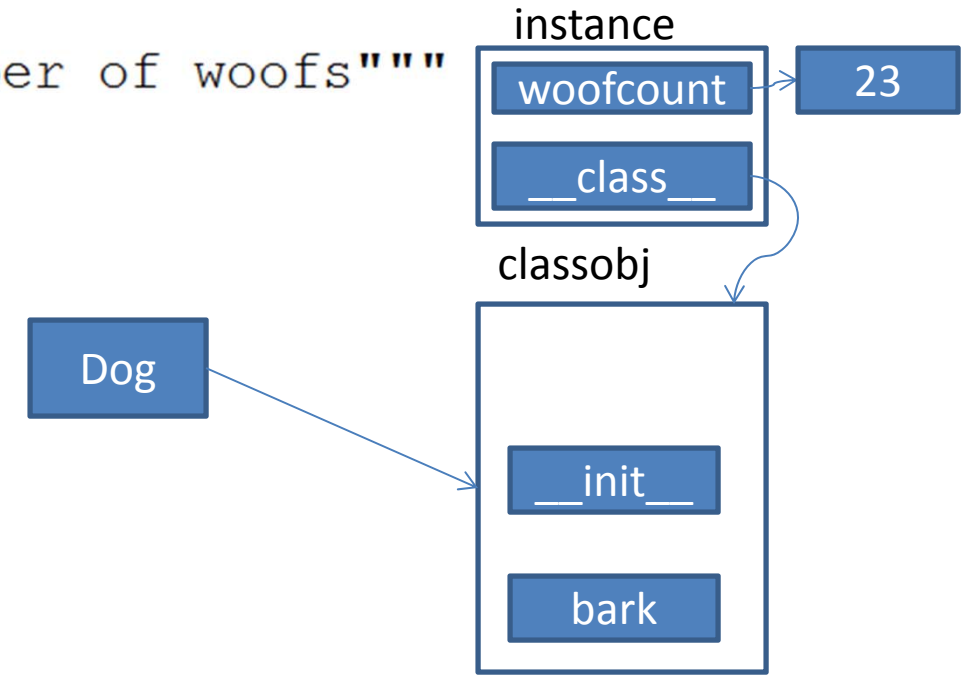
Creating an Instance

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n

Iorek = Dog(23)
```



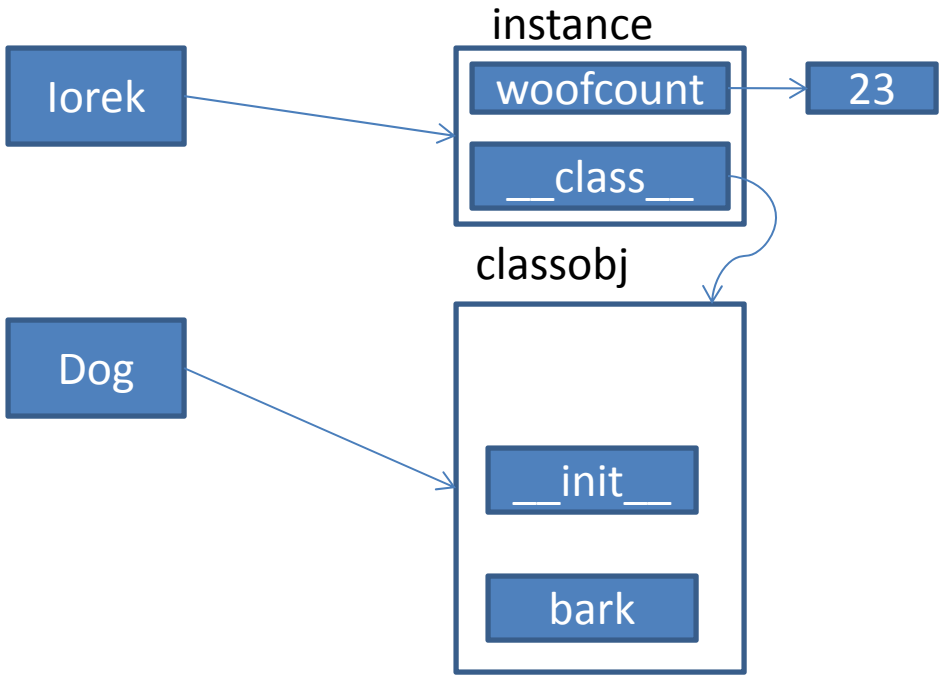
Creating an Instance

```
def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n

Iorek = Dog(23)
```



Creating an Instance

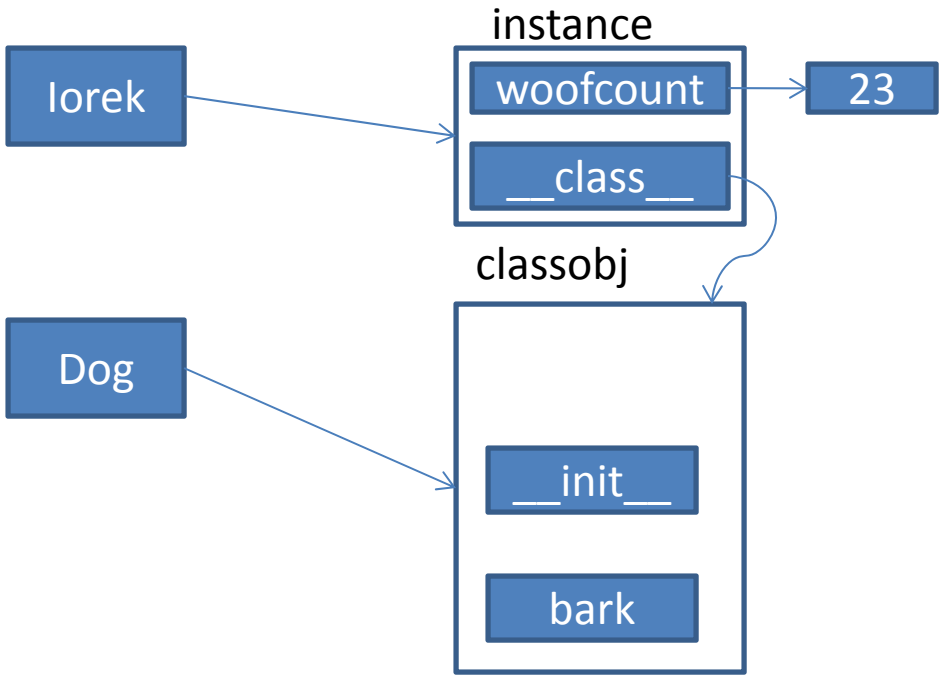
```
Iorek = Dog(23)
```



Creating an Instance

```
Iorek = Dog(23)
print type(Iorek)
print Iorek.__class__.__name__
print Iorek.__class__ == Dog
print Iorek.woofcount
```

```
<type 'instance'>
Dog
True
23
```



Calling Methods Directly

- obj.methodname(params)
  - 1st parameter is bound to obj itself
  - rest of parameters are assigned positionally (or with keywords), like regular functions

```
25      def bark (self):
26          """Bark out loud"""
27          for i in range(self.woofcount):
28              print "Woof"
```

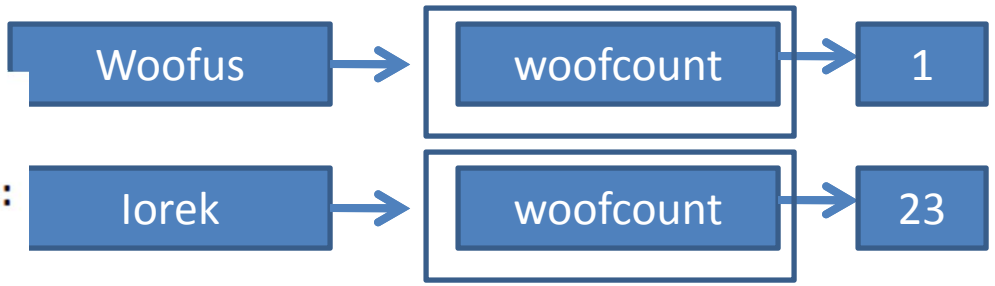
```
Woofus = Dog(1)
print Woofus.woofcount
Woofus.bark()
```

```
1
Woof
```

Accessing instance variables

- Each instance has its own set of instance variables:

```
25    def bark (self):
26        """Bark out loud"""
27        for i in range(self.woofcount):
28            print "Woof"
```



- Can use dot notation to get/set them:

```
47 Woofus = Dog(1)
48 print Woofus.woofcount
49 Woofus.bark()
50 print "-----"
51 Woofus.woofcount = 2
52 print Woofus.woofcount
53 Woofus.bark()
```

```
1
Woof
-----
2
Woof
Woof
```

Iorek.bark()

```
25      def bark (self):
26          """Bark out loud"""
27          for i in range(self.woofcount):
28              print "Woof"
```

```
Iorek.bark()
```

```
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
Woof
```

Overloading/Polymorphism

- Poly = many
- morph = form
- Many types (forms)
  - all implement the same method
- Some examples:
  - __init__
  - __repr__
  - __str__
  - __cmp__

.__str__(self)

`object.__str__(self)`

Called by the `str()` built-in function and by the `print` statement to compute the "informal" string representation of an object. This differs from `__repr__()` in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

```
class Dog:
  """A slobbering friend"""

  def __init__ (self, n):
    """Initialize with number of woofs"""
    self.woofcount = n
  def bark (self):
    """Bark out loud"""
    for i in range(self.woofcount):
      print "Woof"
  def __str__(self):
    """Called whenever a printed representation of self is needed"""
    return "Barks %d times" % self.woofcount

Iorek = Dog(23)
print Iorek
```

```
Barks 23 times
```

In-Class Exercise

- Modify the dog class to add a new instance variable, called barktype.
  1. Have the constructor method initialize this variable to "Woof" by default.
  2. Change the **bark** method to print this variable's value instead of "Woof".
  3. Create dogs with different barktypes and make them bark

```
Fluffy = Dog(1, "Ruff")
Woofus = Dog(3)
Rover = Dog(4, "Arf")

for d in [Fluffy, Woofus, Rover]:
  d.bark()
```

```
Ruff
Woof
Woof
Woof
Arf
Arf
Arf
Arf
```