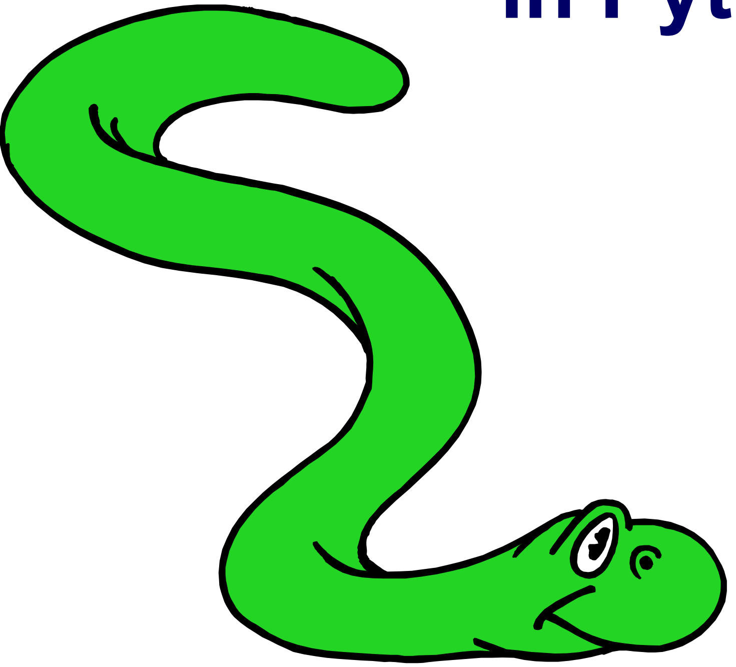
Object Oriented Programming in Python



Contents

Defining a Class

- **A *class* is a special data type which defines how to build a certain kind of object.**
  - The *class* also stores some data items that are shared by all the instances of this class.
  - *Instances* are objects that are created which follow the definition given inside of the class.

A simple class definition: *student*

```
class student:
  """A class representing a student."""
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age


>>> s = student('Bob', 21)
```

Methods in Classes

- **Define a *method* in a *class* by including function definitions within the scope of the class block.**
  - There must be a special first argument ***`self`*** in *all* method definitions which gets bound to the calling instance
  - There are some special methods (like `__init__`)

Constructor: __init__

- **An `__init__` method can take any number of arguments.**
  - Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.

- **However, the first argument `self` in the definition of __init__ is special…**

Self

- **The first argument of every method is a reference to the current instance of the class.**
  - By convention, we name this argument ***`self`***.

- **In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.**
  - Similar to the keyword *this* in Java or C++.
  - But Python uses *self* more often than Java uses *this*.

Self

- Although you must specify `self` explicitly when ***defining*** the method, you don't include it when ***calling*** the method.
- Python passes it for you automatically.

```
def __init__(self, n, a):
    self.name = n
    self.age = a

def set_age(self, num):
    self.age = num
```

```
>>> bob = student("Bob", 21)


>>> student.set_age(bob, 23)
Or:
>>> bob.set_age(23)
```

Traditional Syntax for Access

```
>>> f = student ("Bob Smith", 23)

>>> f.full_name     # Access an attribute.
"Bob Smith"

>>> f.get_age()      # Access a method.
23
```

```
class student:
  """A class representing a student."""
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

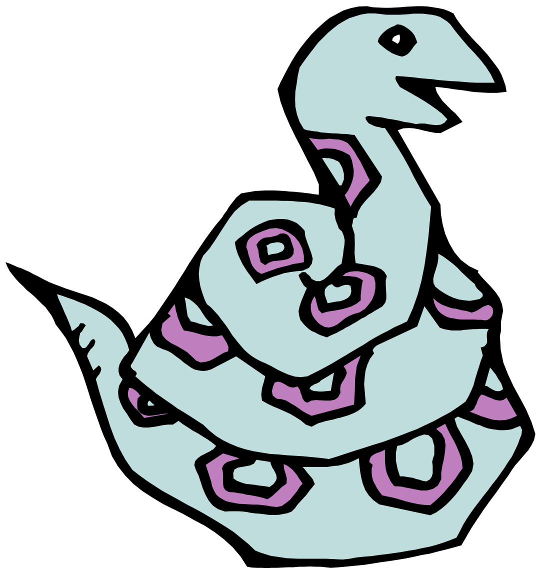
Private Data and Methods

- **Any attribute or method with two leading underscores in its name (but none at the end) is private. It cannot be accessed outside of that class.**
  - Actually Python is not that strict: if you know how, you can can access “private” attributes.
  - Note:
    Names with two underscores at the beginning ***and the end*** are for built-in methods or attributes for the class.
  - Note:
    There is no ‘protected’ status in Python; so, subclasses would be unable to access these private data either.

Deleting instances: No Need to “free”

- **When you are done with an object, you don’t have to delete or free it explicitly.**
  - Python has automatic garbage collection.
  - Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
  - Generally works well, few memory leaks.
  - There’s also no “destructor” method for classes.

Inheritance



Subclasses

- **A class can *extend* the definition of another class**
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- **To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.**

```
class phd_student(student):
```

  - Python has no 'extends' keyword like Java.
  - Multiple inheritance is supported.

Redefining Methods

To *redefine a method* of the parent class, include a new definition using the same name in the subclass.

To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

- The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.

Very common for constructor:

```
parentClass.__init__(self, x, y)
```

Definition of a class extending student

```
class student:
   "A class representing a student."

   def __init__(self,n,a):
       self.full_name = n
       self.age = a

   def get_age(self):
       return self.age
```

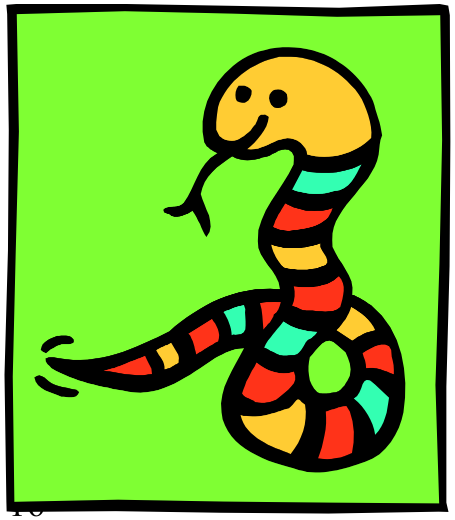
```
-----------------------------------------------------
```

```
class phd_student (student):
   "A class extending student."

   def __init__(self,n,a,s):
       student.__init__(self,n,a) #Call __init__ for student
       self.graduate_school = s

   def get_age(self):   #Redefines get_age method entirely
       print "Age: " + str(self.age)
```

Accessing unknown attributes



Accessing unknown members

- **Problem: Occasionally the name of an attribute or method of a class is only given at run time…**

- **Solution: `getattr(object_instance, string)`**
  - `string` is a string which contains the name of an attribute or method of a class
  - `getattr(object_instance, string)` returns a reference to that attribute or method

getattr(object_instance, string)

```
>>> f = student(“Bob Smith”, 23)

>>> getattr(f, “full_name”)    #same as f.full_name
“Bob Smith”

>>> getattr(f, “get_age”)
 <method get_age of class studentClass at 010B3C2>

>>> getattr(f, “get_age”)()    # We can call this.
23

>>> getattr(f, “get_birthday”)
      # Raises AttributeError – No method exists.
```

hasattr(object_instance,string)

```
>>> f = student(“Bob Smith”, 23)

>>> hasattr(f, “full_name”)
True

>>> hasattr(f, “get_age”)
True

>>> hasattr(f, “get_birthday”)
False
```

Two Kinds of Attributes

- **The non-method data stored by objects are called attributes.**

- *Data* attributes
  - Variable owned by a *particular instance* of a class.
  - Each instance has its own value for it.
  - These are the most common kind of attribute.

- *Class* attributes
  - Owned by the *class as a whole*.
  - *All instances of the class share the same value for it.*
  - Called "static" variables in some languages.
  - Good for
    - —class-wide constants
    - —building counter of how many instances of the class have been made

Data Attributes

- **Data attributes are created and initialized by an `__init__()` method.**
  - Simply assigning to a name creates the attribute.
  - Inside the class, refer to data attributes using **`self`**
    - —for example, **`self.full_name`**

```
class teacher:
   “A class representing teachers.”
   def __init__(self,n):
        self.full_name = n
   def print_name(self):
        print self.full_name
```

Class Attributes

- **Because all instances of a class share one copy of a class attribute:**
  - when *any* instance changes it, the value is changed for *all* instances.
- **Class attributes are defined**
  - *within* a class definition
  - *outside* of any method

- **Since there is one of these attributes *per class* and not one *per instance*, they are accessed using a different notation:**
  - Access class attributes using `self.__class__.name` notation.

```
class sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

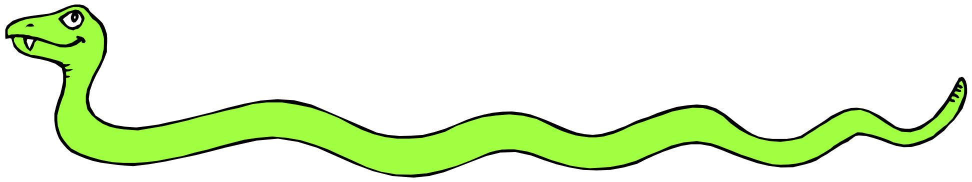
Data vs. Class Attributes

```
class counter:
   overall_total = 0
         # class attribute
   def __init__(self):
       self.my_total = 0
          # data attribute
   def increment(self):
      counter.overall_total = \

      counter.overall_total + 1
      self.my_total = \
      self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Special Built-In Methods and Attributes



Built-In Members of Classes

- **Classes contain many methods and attributes that are included by Python even if you don't define them explicitly.**
  - Most of these methods define automatic functionality triggered by special operators or usage of that class.
  - The built-in attributes define information that must be stored for all classes.
- **All built-in members have double underscores around their names: `__init__` `__doc__`**

Special Methods – Example

```
class student:
    ...
     def __str__(self):
       return “I’m named ” + self.full_name
    ...

>>> f = student(“Bob Smith”, 23)
>>> print f
I’m named Bob Smith
```

Special Methods

- You can redefine these as well:

  `__init__` : The constructor for the class.

  `__cmp__` : Define how == works for class.

  `__len__` : Define how `len( obj )` works.

  `__add__` : Define how to add objects

  ...

- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call.

Special Data Items

- These attributes exist for all classes.

  `__doc__` : Variable storing the documentation string for that class.

  `__class__` : Variable which gives you a reference to the class from any instance of it.

  `__module__` : Variable which gives you a reference to the module in which the particular class is defined.

- Useful:
  - `dir(x)` returns a list of all methods and attributes defined for object x

Special Data Items – Example

```
>>> f = student(“Bob Smith”, 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >

>>> g = f.__class__(“Tom Jones”, 34)
```

Acknowledgements

The talk relied on a course given in spring 2008 at the University of Pennsylvania

www.cis.upenn.edu/~cis530/slides-2008/Python-complete-tutorial-2008-spring.ppt

Thank you for your attention.

Questions?