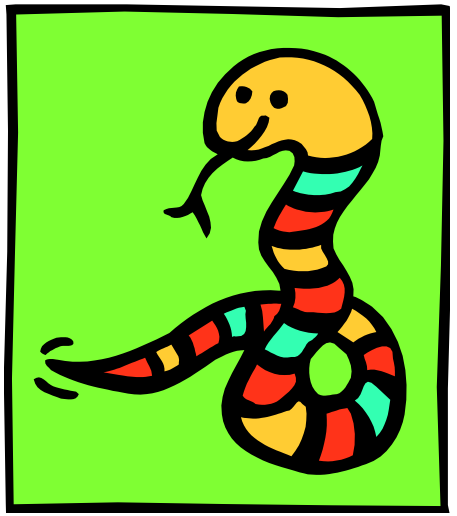
Attributes



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
```

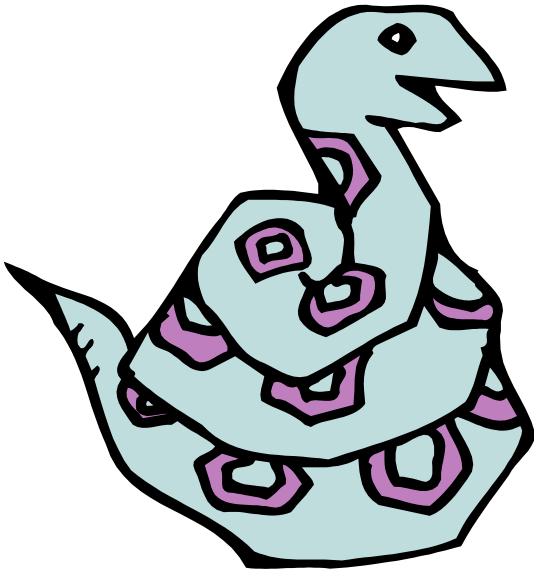
```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

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
```

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 ai_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.**
  - The old code won't get executed.

- **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.**

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 ai_student (student):
   "A class extending student."

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

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

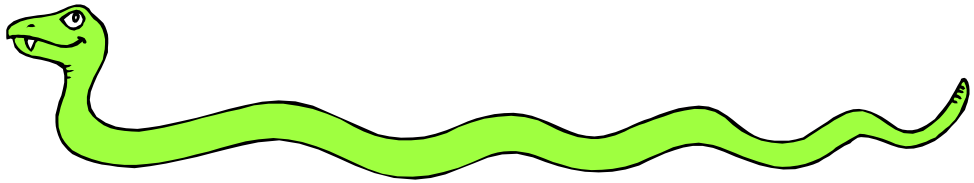
Extending __init__

- **Same as for redefining any other method…**
  - Commonly, the ancestor's `__init__` method is executed in addition to new commands.
  - You'll often see something like this in the `__init__` method of subclasses:

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

where parentClass is the name of the parent's class.

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

- **For example, the method `__repr__` exists for all classes, and you can always redefine it.**
- **The definition of this method specifies how to turn an instance of the class into a string.**
  - `print(f)` sometimes calls `f.__repr__()` to produce a string for object f.
  - If you type `f` at the prompt and hit ENTER, then you are also calling `__repr__` to determine what to display to the user as output.

Special Methods – Example

```
class student:
    ...
     def __repr__(self):
       return "I'm named " + self.full_name
    ...

>>> f = student("Bob Smith", 23)
>>> print(f)
I'm named Bob Smith
>>> 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.

  `__copy__` : Define how to copy a class.

- **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**

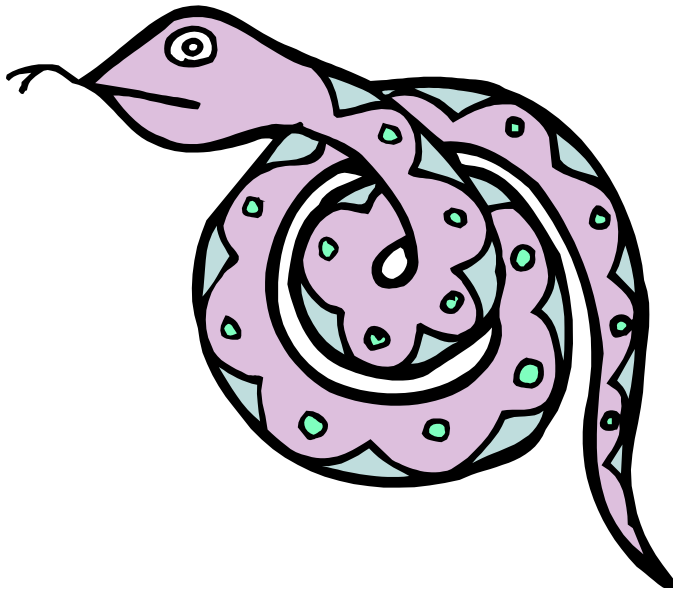
Special Data Items – Example

```
>>> 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)
```

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.**
  - 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.

File Processing and Error Handling: Learning on your own…



File Processing with Python

This is a good way to play with the error handing capabilities of Python. Try accessing files without permissions or with non-existent names, etc.

You'll get plenty of errors to look at and play with!

```
fileptr = open('filename')
somestring = fileptr.read()
for line in fileptr:
   print line
fileptr.close()
```

Exception Handling

- **Errors are a kind of object in Python.**
  - More specific kinds of errors are subclasses of the general Error class.

- **You use the following commands to interact with them:**

  try

  except

  finally

  catch

Exceptions and handlers

```
while True:
   try:
        x = int(raw_input("Number, please! "))
        print "The number was: ", x
   except ValueError:
        print "Oops! That was not a valid number."
        print "Try again."
        print
```

Exceptions and handlers

```
def loud_kaboom():
   x = 1/0;

def fireworks_factory():
   raise ZeroDivisionError("Gasoline near bone-dry Christmas trees!")

def playing_with_fire():
   try:
        loud_kaboom()
   except ZeroDivisionError as exc:
        print "Handling run-time error: ", exc

   try:
        fireworks_factory()
   except ZeroDivisionError:
        print "Gotta stop this from happening..."
```

Some folk's favorite statement in Python

- `yield(a,b,c)`
  - Turns a loop into a *generator function* that can be used for

    Lazy evaluation

    Creating potentially infinite lists in a usable way…
- **See Section 6.8 of the Python reference manual**

Finally…

- **`pass`**
  - It does absolutely nothing.

- **Just holds the place of where something should go syntactically. Programmers like to use it to waste time in some code, or to hold the place where they would like put some real code at a later time.**

```
for i in range(1000):
    pass
```

Like a "no-op" in assembly code, or a set of empty braces {} in C++ or Java.

Regular Expressions and Match Objects

- Python provides a very rich set of tools for pattern matching against strings in module *re* (for regular expression)
- For a gentle introduction to regular expressions in Python see the HOWTO regex tutorial on the course website
- More to come later …