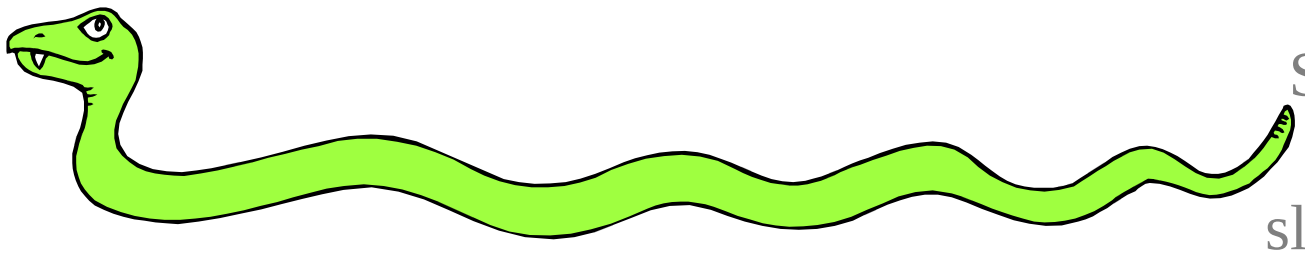
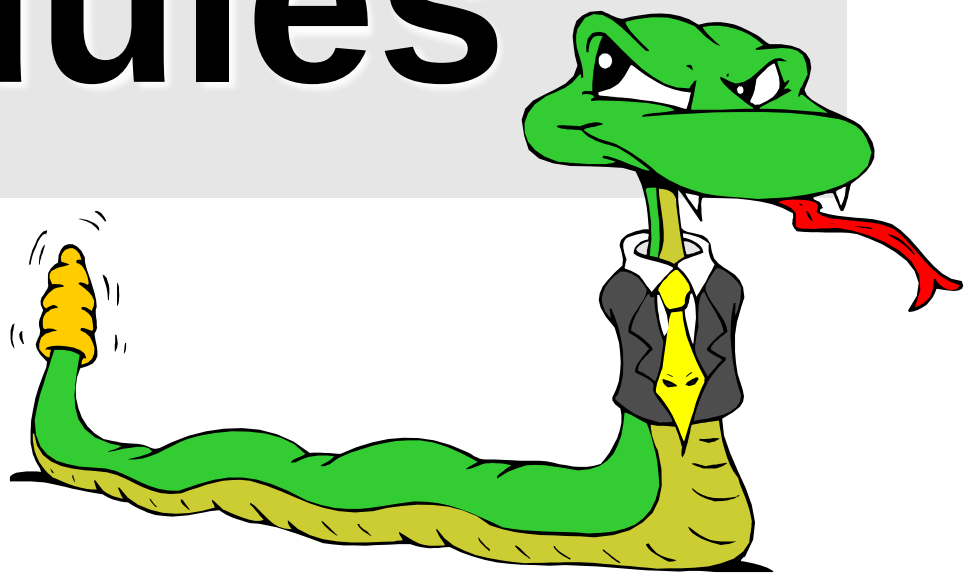
# Python 3



Some material adapted
from Upenn cis391
slides and other sources

# Importing and Modules



# Importing and Modules

- Use classes & functions defined in another file
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, C++ *include*
- Three formats of the command:

```
import somefile
from somefile import *
from somefile import className
```

- The difference? What gets imported from the file and what name refers to it after importing

# *import …*

```
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text “somefile.” to the front of its name:

```
somefile.className.method(“abc”)
somefile.myFunction(34)
```

# *from … import **

```
from somefile import *
```

- *Everything* in somefile.py gets imported
- To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.
- *Take care!* Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method("abc")

myFunction(34)
```

# *from … import …*

```
from somefile import className
```

- Only the item *className* in somefile.py gets imported.
- After importing *className*, you can just use it without a module prefix. It's brought into the current namespace.
- *Take care*! Overwrites the definition of this name if already defined in the current namespace!

`className.method("abc")` ← imported

`myFunction(34)` ← Not imported

# Directories for module files

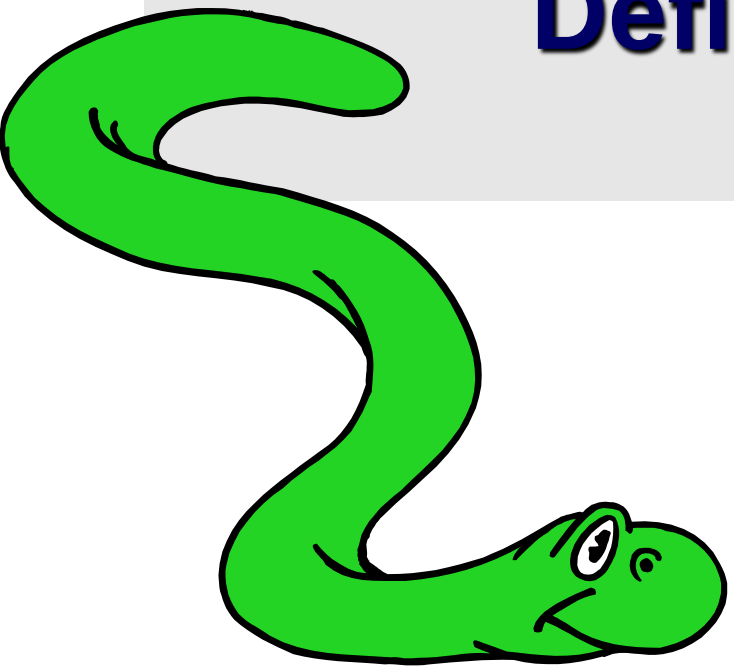
- *Where does Python look for module files?*
- The list of directories where Python will look for the files to be imported is  sys.path
- This is just a variable named ‘path’ stored inside the ‘sys’ module

```
>>> import sys
>>> sys.path
['',
    '/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages/setuptools-0.6c5-py2.5.egg', …]
```

- To add a directory of your own to this list, append it to this list

```
sys.path.append('/my/new/path')
```

# Object Oriented Programming in Python: Defining Classes



# It's all objects…

- Everything in Python is really an object.
  - We've seen hints of this already…

```
"hello".upper()
list3.append('a')
dict2.keys()
```

  - These look like Java or C++ method calls.
  - New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

# 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
- Python doesn’t use separate class interface definitions as in some languages
- You just define the class and then use it

# 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* of method definitions which gets bound to the calling instance
- There is usually a special method called ***`__init__`*** in most classes
- We'll talk about both later…

# A simple class def: *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
```

# Creating and Deleting Instances

# Instantiating Objects

- There is no “new” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = student(“Bob”, 21)
```

# 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

Defining a method:

*(this code inside a class definition.)*

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

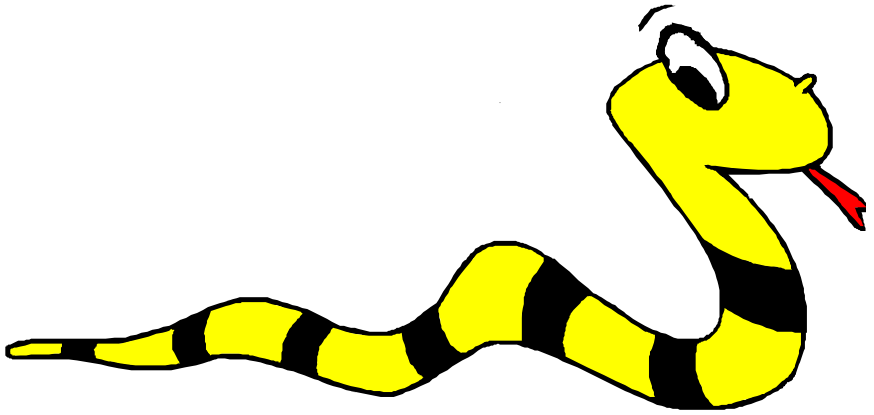
Calling a method:

```
>>> x.set_age(23)
```

# 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

# Access to Attributes and Methods



# Definition of 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
```

# Traditional Syntax for Access

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

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

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

# 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

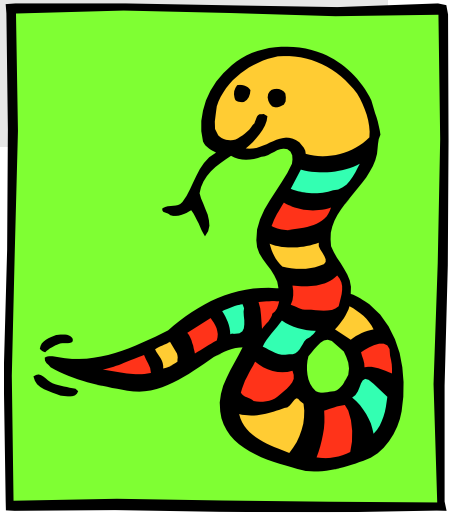
# getattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)
>>> getattr(f, "full_name")
"Bob Smith"
>>> getattr(f, "get_age")
 <method get_age of class
 studentClass at 010B3C2>
>>> getattr(f, "get_age")() # call it
23
>>> getattr(f, "get_birthday")
# Raises AttributeError - No method!
```

# hasattr(object_instance,string)

```
>>> f = student(“Bob Smith”, 23)
>>> hasattr(f, “full_name”)
True
>>> hasattr(f, “get_age”)
True
>>> hasattr(f, “get_birthday”)
False
```

# 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 class instances share the same value for it*
  - Called “static” variables in some languages
  - Good for (1) class-wide constants and (2) 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 and *outside* of any method
- Since there is one of these attributes *per class* and not one *per instance*, they're accessed via a different notation:
  - Access class attributes using **`self.__class__.name`** notation
    -- This is just one way to do this & the safest in general.

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

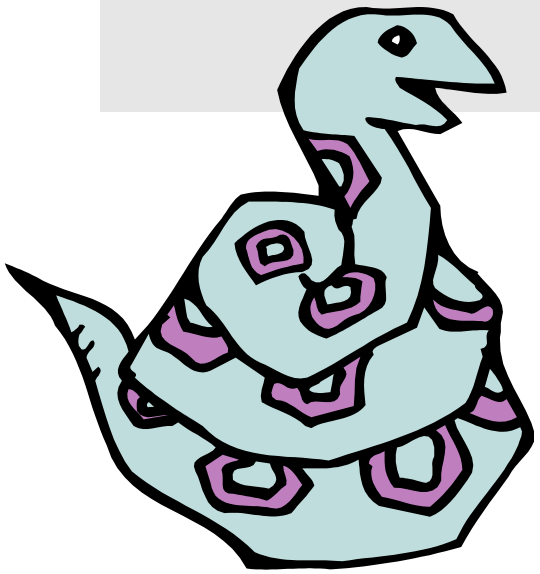
```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

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

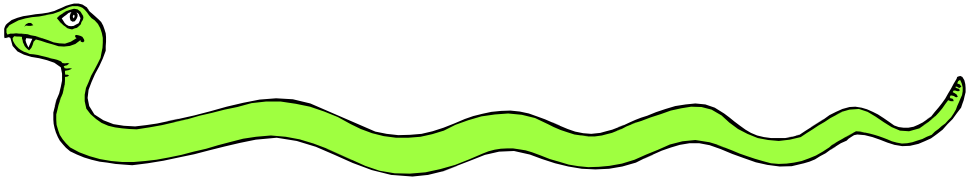
# Extending __init__

- 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 for documentation string for class

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

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

  __dict__ :The dictionary that is actually the namespace for a class (but not its superclasses)

- 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/method with 2 leading under-scores in its name (but none at the end) is **private** and can't be accessed outside of 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.