Object Oriented Programming in Python

Richard P. Muller

Materials and Process Simulations Center

California Institute of Technology

June 1, 2000

Introduction

- We've seen Python useful for
  - Simple Scripts
  - Numerical Programming
- This lecture discusses Object Oriented Programming
  - Better program design
  - Better modularization

What is an Object?

- A software item that contains variables and methods
- Object Oriented Design focuses on
  - Encapsulation:

    dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:

    the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:

    the ability to create subclasses that contain specializations of their parents

Namespaces

- At the simplest level, classes are simply namespaces

```
class myfunctions:
  def exp():
        return 0

>>> math.exp(1)
2.71828...
>>> myfunctions.exp(1)
0
```

- It can sometimes be useful to put groups of functions in their own namespace to differentiate these functions from other similarly named ones.

Python Classes

- Python contains classes that define objects
  - Objects are instances of classes

__init__ is the default constructor

```
class atom:
  def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
```

self refers to the object itself,
like *this* in Java.

Example: Atom class

```
class atom:
  def __init__(self,atno,x,y,z):
       self.atno = atno
       self.position = (x,y,z)
  def symbol(self):   # a class method
       return Atno_to_Symbol[atno]
  def __repr__(self): # overloads printing
       return '%d %10.4f %10.4f %10.4f' %
              (self.atno, self.position[0],
               self.position[1],self.position[2])

>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```

Atom class

- Overloaded the default constructor
- Defined class variables (atno,position) that are persistent and local to the atom object
- Good way to manage shared memory:
  - instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
  - much cleaner programs result
- Overloaded the print operator

- We now want to use the atom class to build molecules...

Molecule Class

```
class molecule:
  def __init__(self,name='Generic'):
      self.name = name
      self.atomlist = []
  def addatom(self,atom):
      self.atomlist.append(atom)
  def __repr__(self):
      str = 'This is a molecule named %s\n' % self.name
      str = str+'It has %d atoms\n' % len(self.atomlist)
      for atom in self.atomlist:
          str = str + `atom` + '\n'
      return str
```

Using Molecule Class

```
>>> mol = molecule('Water')
>>> at = atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom(atom(1,0.,0.,1.))
>>> mol.addatom(atom(1,0.,1.,0.))
>>> print mol
This is a molecule named Water
It has 3 atoms
8  0.000 0.000 0.000
1  0.000 0.000 1.000
1  0.000 1.000 0.000
```

- Note that the print function calls the atoms print function
  - Code reuse: only have to type the code that prints an atom once; this means that if you change the atom specification, you only have one place to update.

Inheritance

```
class qm_molecule(molecule):
  def addbasis(self):
        self.basis = []
        for atom in self.atomlist:
              self.basis = add_bf(atom,self.basis)
```

- __init__, __repr__, and __addatom__ are taken from the parent class (molecule)
- Added a new function addbasis() to add a basis set
- Another example of code reuse
  - Basic functions don't have to be retyped, just inherited
  - Less to rewrite when specifications change

Overloading parent functions

```
class qm_molecule(molecule):
  def __repr__(self):
        str = 'QM Rules!\n'
        for atom in self.atomlist:
              str = str + `atom` + '\n'
        return str
```

- Now we only inherit __init__ and addatom from the parent
- We define a new version of __repr__ specially for QM

Adding to parent functions

- Sometimes you want to extend, rather than replace, the parent functions.

```
class qm_molecule(molecule):
  def __init__(self,name="Generic",basis="6-31G**"):
        self.basis = basis
        molecule.__init__(self,name)
```

add additional functionality
to the constructor

call the constructor
for the parent function

Public and Private Data

- Currently everything in atom/molecule is public, thus we could do something really stupid like

```
>>> at = atom(6,0.,0.,0.)
>>> at.position = 'Grape Jelly'
```

  that would break any function that used at.poisition

- We therefore need to protect the at.position and provide accessors to this data
  - Encapsulation or Data Hiding
  - accessors are "gettors" and "settors"
- Encapsulation is particularly important when other people use your class

Public and Private Data, Cont.

- In Python anything with two leading underscores is private

```
__a, __my_variable
```

- Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.

```
_b
```

  - Sometimes useful as an intermediate step to making data private

Encapsulated Atom

```
class atom:
  def __init__(self,atno,x,y,z):
       self.atno = atno
       self.__position = (x,y,z) #position is private
  def getposition(self):
       return self.__position
  def setposition(self,x,y,z):
       self.__position = (x,y,z) #typecheck first!
  def translate(self,x,y,z):
       x0,y0,z0 = self.__position
       self.__position = (x0+x,y0+y,z0+z)
```

Why Encapsulate?

- By defining a specific interface you can keep other modules from doing anything incorrect to your data
- By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users
  – Write to the Interface, not the the Implementation
  – Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions

Classes that look like arrays

- Overload __getitem__(self,index) to make a class act like an array

```
class molecule:
  def __getitem__(self,index):
       return self.atomlist[index]

>>> mol = molecule('Water') #defined as before
>>> for atom in mol:         #use like a list!
       print atom
>>> mol[0].translate(1.,1.,1.)
```

- Previous lectures defined molecules to be arrays of atoms.
- This allows us to use the same routines, but using the molecule class instead of the old arrays.
- An example of focusing on the interface!

Classes that look like functions

- Overload __call__(self,arg) to make a class behave like a function

```
class gaussian:
  def __init__(self,exponent):
       self.exponent = exponent
  def __call__(self,arg):
       return math.exp(-self.exponent*arg*arg)

>>> func = gaussian(1.)
>>> func(3.)
0.0001234
```

Other things to overload

- __setitem__(self,index,value)
  - Another function for making a class look like an array/dictionary
  - a[index] = value
- __add__(self,other)
  - Overload the "+" operator
  - molecule = molecule + atom
- __mul__(self,number)
  - Overload the "*" operator
  - zeros = 3*[0]
- __getattr__(self,name)
  - Overload attribute calls
  - We could have done atom.symbol() this way

Other things to overload, cont.

- __del__(self)
  - Overload the default destructor
  - del temp_atom
- __len__(self)
  - Overload the len() command
  - natoms = len(mol)
- __getslice__(self,low,high)
  - Overload slicing
  - glycine = protein[0:9]
- __cmp__(self,other):
  - On comparisons (<, ==, etc.) returns -1, 0, or 1, like C's strcmp

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (The Gang of Four) (Addison Wesley, 1994)
- *Refactoring: Improving the Design of Existing Code*, Martin Fowler (Addison Wesley, 1999)
- *Programming Python*, Mark Lutz (ORA, 1996).