WHAT WE HAVE LEARNED

Playing with **graphics objects**
- Canvas
- Drawable objects
- Operations on drawable objects
- Layer
- Animation
Object Oriented Programming in python

- Class
- Inheritance
- Polymorphism
- Encapsulation
An object has **state** and can perform **actions**

**Object Oriented Design focuses on**

- **Encapsulation**: A interface is released to users (public interface) and the implementation of that interface is hidden (private implementation)
- **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 specialization of their parents
Python contains classes that define objects

- Objects are *instances* of classes
- E.g) `sun = Circle(30)`
**ATOM CLASS**

`__init__` is the default constructor

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

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

def symbol(self):
    return Atno_to_Symbol[atno]

def __repr__(self):
    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'
```

Creation of the object : `__init__` is called automatically

If you print an object, `__repr__` is called defined in the class

You can call a class method by . notation
WHAT WE DO FOR ATOM CLASS

Overloaded the default constructor
Define class variables (atno, position) that are persistent and local to the atom object
Overloaded the print operator
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
' % self.name
        str = str+'It has %d atoms
' % len(self.atomlist)
        for atom in self.atomlist:
            str = str + atom + '\n'
        return str
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  0.0000 0.0000 0.0000
1  0.0000 0.0000 1.0000
1  0.0000 1.0000 0.0000

Note that the print function calls the atom’s print function
- Code reuse: only have to type the code that print an atom once,
which means that if you change the atom specification you only
have on place to update
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
Now, we only inherit \texttt{\_\_init\_} and \texttt{addatom} from the parent.

We define a new version of \texttt{\_\_repr\_} specially for QM.
Sometimes you want to extend, rather than replace, the parent functions.

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

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

    def __repr__(self):
        str = 'QM Rules!
        for atom in self.atomlist:
            str = str + atom + '
        return str
```
PUBLIC AND PRIVATE DATA

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

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

, which would break any function that used at.position

We therefore need to protect the at.position and provide accessors to this data

- Encapsulation or Data hiding
- Accessors are “getters” and “settors”

Encapsulation is particularly important when other people use your class
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 for 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)

You cannot access to the variable __position any more, since it’s private.
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 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

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
' % self.name
        str = str+'It has %d atoms
' % len(self.atomlist)
        for atom in self.atomlist:
            str = str + atom + '
'
        return str
    def __getitem__(self, index):
        return self.atomlist[index]

Overload __getitem__(self, index) to make a class act like an array
CLASSES THAT LOOK LIKE ARRAYS

An example of focusing on the interface

```python
>>> mol = molecule('Water')  # defined as before

>>> for atom in mol:        # use like a list!
    print atom

>>> mol[0].translate(1.,1.,1.)
```
CLASSES THAT LOOK LIKE FUNCTIONS

```python
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.)
#use like a function
0.0001234
```

Overload `__call__(self, arg)` to make a class act like a function.
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), ___radd__(self, number)
- Overload the "+" operator
- molecule = molecule + atom

___mul__(self, number), ___rmul__(self, number)
- Overload the "*" operator
- zeros = 3*[0]
OTHER THINGS TO OVERLOAD

```python
__del__(self)
• Overload the default destructor
• del temp_atom

__len__(self)
• Overload the len() command
• natoms = len(mol)

__cmp__(self, other):
• On comparisons (<, ==, etc.) returns -1, 0, or 1
```
class MyClass:
    i = 12345
    def f(self):
        return 'hello world'
    def g(self):
        self.i = 2

x = MyClass()
y = MyClass()
print MyClass.i, x.i
y.i = 67890
print MyClass.i, x.i, y.i
MyClass.i = 9999
print MyClass.i, x.i, y.i
x.i = 00000
print MyClass.i, x.i, y.i
class K:
    i = 'foo'
    def get_static_i(self):
        return K.i
    def get_instance_i(self):
        return self.i
    def __init__(self):
        self.i = 'not foo'

Kobj=K()
print K.i
print Kobj.get_static_i()
print Kobj.get_instance_i()

Outputs:
foo
foo
not foo
Class foo(object):
    name = 'foo'

    @staticmethod
    def get_name_static():
        print foo.name

    @classmethod
    def get_name_class(cls):
        print cls.name

Class bar(foo):
    name = 'bar'

>>> foo.get_name_static()  # foo is a class name, not an object name
foo
>>> foo.get_name_class()
foo

>>> bar.get_name_static()  # foo.name is printed exactly
foo
>>> bar.get_name_class()   # cls.name is printed (cls == bar)
bar