Algorithms for Graphical Models (AGM)

Python: Object-orientation

$Date: 2008/10/15 15:30:11 $

AGM-03

Everything is an object

- Builtin types: string, dictionary, set, file, function etc.
- Much of their functionality accessed via *methods*.
- User-defined types: classes

But first: modules

- Every Python object lives in some *module*.
- Even objects created with the interpreter live in a special module called `__main__` (this is also the module for objects in the top-level script)
- Normal modules are just Python source files.
- The module `foo` will be in the file `foo.py`

Importing modules

- The statement `import math` imports the builtin module `math` into your *namespace*.

- You can then get to the objects defined within that module:

```
>>> math.sqrt(2)
1.4142135623730951
```

- `sqrt`, a function, is an *attribute* of the `math` module.

- Use `dir(math)` to get all its attributes.

Another way to import

- Can also do `from math import sqrt` to put the `sqrt` function (but not the module) directly into your namespace.

- `from math import *` grabs everything.

Rolling your own: creating new datatypes

```
class Point(object):          # An object of class Point is an object
    'Simple class to define points' # Documentation
    def __init__(self,xval,yval):   # __init__ called when a new ...
        self.x = float(xval)        # ...Point object (self) is created
        self.y = float(yval)        # x and y are attributes for self
```

Creating instances

```
>>> from pt import Point
>>> type(Point)
<type 'type'>
>>> p = Point(3.2,3.4)  # __init__ is called here
>>> p.x
3.2000000000000002
>>> type(p)
<class 'pt.Point'>
>>> type(p.x)
<type 'float'>
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__doc__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'__weakref__', 'x', 'y']
```

Special method names

Can get user-defined classes to behave like builtins by defining methods with special names.

```
class Point(object)
    ....
    def __add__(self,other):
        return Point(self.x+other.x,self.y+other.y)

    def __str__(self):
        return '(%f,%f)' % (self.x,self.y)
```

The effect of special method names

```
>>> from pt import Point
>>> p1=Point(2,3)
>>> p2=Point(1,4)
>>> p1+p2
<pt.Point object at 0x403f8a8c>
>>> print p1+p2
(3.000000,7.000000)
```

Normal methods

```
    def norm(self):
        from math import sqrt
        return sqrt(self.x**2 + self.y**2)
```

```
>>> from pt import Point
>>> p1=Point(2,3)
>>> p1.norm()
3.605551275463989
```

Adding attributes on the fly

```
>>> class Foo(object):
...  pass
...
>>> x=Foo()
>>> x.onthefly = 'blah'
>>> x.onthefly
'blah'
>>> del x.onthefly
>>> x.onthefly
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Foo' object has no attribute 'onthefly'
```

Objects you can see!

```
>>> from Tkinter import *
>>> root=Tk()
>>> root.title('example')
''
>>> lab = Label(root,text='hello')
>>> lab.pack()
>>> but = Button(root,text='die',command=x.destroy)
>>> but.pack()
```

Inheritance

- A class can inherit all the methods, attributes from some parent class

- And then override some of them, and add new ones.

Here's the syntax:

```
class GraphCanvas(Canvas):
    << methods etc>> go here
```

More on inheritance

- Can have *class attributes* inherited by all instances of a class.

- An instance can override class attributes to have its own private attribute.

- Can call parent methods directly if needed.

```
def __init__(self, parent=None, **config):
    # parent class initialiser ...
    Canvas.__init__(self,parent,config)
    # key bindings
    self.bind('<ButtonPress-1>', self.sel_or_new)
```

Inheritance by example

Let's have a look at `colouring.py`