Slides from INF3331 lectures - Python functions and classes revisited

Joakim Sundnes, Ola Skavhaug and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

October 2013

Contents

- Function programming, focusing on the “special features” of Python functions;
  - Functions are regular objects; passed as arguments, returned from functions, modified inside functions etc...
- Classes and object oriented programming in Python vs other OOP languages;
  - Classes are also objects
  - No declaration of variables; attributes can be added on the fly, `self` used to distinguish between local and instance variables, ...

Python functions revisited

Contents

More info

- Chapter 6 in *Python; essential reference*, by David M Beazley
- Python language reference (`docs.python.org/2/reference`)
- Online tutorials and examples...

Python functions are objects

- Instances of class `function`;

```
In [1]: def f():
    pass
   ...:
In [2]: f.__class__
Out[2]: function
In [3]: issubclass(f.__class__,object)
Out[3]: True
```

- Pyton functions have attributes

Functions as arguments

- Like all objects, functions can be arguments to functions

```
In [4]: def add(x,y):
   ...:     return x+y
   ...:
In [5]: def sub(x,y):
   ...:     return x-y
   ...:
In [6]: def apply(func,x,y):
   ...:     return func(x,y)
   ...:
In [7]: apply(add,2,2)
Out[7]: 4
In [8]: apply(sub,7,3)
Out[8]: 4
```

Functions inside functions (1)

- Like many other languages (but unlike standard C/C++) Python allows nested function definitions

```
In [30]: def f(x,y):
   ....:     def cube(x):
   ....:         return x*x*x
   ....:     return y*cube(x)
In [31]: f(4,6)
Out[31]: 384
```

Functions inside functions (2)

- A function can also return a function;

```
In [33]: def f():
   ....:     def inner_f():
   ....:         print "Inside inner_f"
   ....:     return inner_f
In [34]: foo = f()

In [35]: foo
Out[35]: <function __main__.inner_f>
In [36]: foo()
Inside inner_f
```

- This shows the close relation of Python functions to standard classes and OOP;
  - `foo` is a standard object
  - `foo()` invokes the object's call operator
  - Try `dir(foo)` on any function `foo` to see its methods and attributes.

Closure

- Consider the previous example, slightly tweaked:

```
In [44]: def f():
    ....:     x = 3
    ....:     def inner():
    ....:         print "x = ", x
    ....:     return inner
In [45]: foo = f()
In [46]: x = 10
In [47]: foo()
x =  3
```

- When a function is returned, it remembers the variables declared in the namespace surrounding its definition. This is called the function's *closure*.
- The closure of a function is an attribute:

```
In [54]: foo.func_closure
Out[54]: (<cell at 0x101ca72f0: int object at 0x100313788>,)
In [55]: [cell.cell_contents for cell in foo.func_closure]
Out[55]: [3]
```

An old example revisited

- Suppose we need a function of `x` and `y` with three additional parameters `a`, `b`, and `c`:

  ```
  def f(x, y, a, b, c):
      return a + b*x + c*y*y
  ```

- Suppose we need to send this function to another function

  ```
  def gridvalues(func, xcoor, ycoor, file):
      for i in range(len(xcoor)):
          for j in range(len(ycoor)):
              f = func(xcoor[i], ycoor[j])
              file.write('%g %g %g\n' % (xcoor[i], ycoor[j], f)
  ```

  `func` is expected to be a function of `x` and `y` only (many libraries need to make such assumptions!)

- How can we send our `f` function to `gridvalues`?

Solution 1: class with call operator

- Make a class with function behavior instead of a pure function
- The parameters are class attributes
- Class instances can be called as ordinary functions, now with `x` and `y` as the only formal arguments

```
class F:
    def __init__(self, a=1, b=1, c=1):
        self.a = a;  self.b = b;  self.c = c

    def __call__(self, x, y):     # special method!
        return self.a + self.b*x + self.c*y*y

f = F(a=0.5, c=0.01)
# can now call f as
v = f(0.1, 2)
...
gridvalues(f, xcoor, ycoor, somefile)
```

Solution 2: closure

- Make a function that locks the namespace and returns the function we need;

```
In [57]: def F(a=1,b=1,c=1):
    ....:     def inner(x,y):
    ....:         return a+b*x+c*y*y
    ....:     return inner
    ....:
In [58]: f = F(a=0.5,c=0.01)
```

- We can now call this function with two arguments; `v = f(0.1,2)`
- Class or closure is approximately equivalent for this simple case; closure slightly more efficient, class more flexible.

More functions returning functions; *decorators*

- A toy example;

```
In [61]: def foo():
   ....:     return 1
In [62]: def outer(func):
   ....:     def inner():
   ....:         print "before calling func"
   ....:         return func()
   ....:     return inner
In [63]: decorated = outer(foo)
In [64]: decorated()
before calling func
Out[64]: 1
```

- The function `decorated` is a decorated version of function `foo`; it is `foo` plus something more
- To simplify, we could just write

```
In [65]: foo = outer(foo)
```

to replace `foo` with its decorated version each time it is called

A (slightly) more useful decorator

- Suppose we want to limit the range of values sent to a mathematical formula:

```
In [74]: def f(x):
   ....:     return x**3 - 2
In [75]: def checkrange(func):
    def inner(x):
        if x <0:
            print "out of range"
        else:
            return func(x)
    return inner
In [77]: f(5)
Out[77]: 123
In [78]: f(-1)
Out[78]: -3
In [79]: f = checkrange(f)
In [80]: f(5)
Out[80]: 123
In [81]: f(-1)
out of range
```

The `@decorator` syntax

- Python provides a short notation for decorating a function with another function:

```
In [82]: @checkrange
   ....: def g(x):
   ....:     return x**3-2
In [83]: g(2)
Out[83]: 6
In [84]: g(-2)
out of range
```

- This is exactly the same as writing `g=checkrange(g)`.
- A decorator is simply a function taking another function as input and returning another function. The syntax `@decorator` is a short-cut for the more explicit `f=decorator(f)`

Python classes revisited

Contents

- Short recap of Python classes and OOP
- Use of `self` in instance methods
- Instance attributes vs class attributes
- Instance methods, class methods, static methods
- Special attributes and special methods

More info

- Ch. 8.6 in *Python scripting for computational science*
- Ch. 2 in *Illustrating Python via bioinformatics examples*
- Chapter 7 in *Python essential reference* by D. M. Beazley
- Python Reference Manual (special methods in 3.3)

Python classes (1)

- Similar class concept as in Java and C++
- All functions are virtual
- No private/protected variables
  (the effect can be "simulated")
- Single and multiple inheritance
- Remember; everything in Python is an object (even a class)

Python classes (2)

- Classes work as usual, containing methods and attributes;

```
class MyBase:

    def __init__(self,i,j):   # constructor
        self.i = i; self.j = j

    def write(self):          # member function
        print 'MyBase: i=',self.i,'j=',self.j
```

- `self` is a reference to the instance
- Instance attributes are prefixed by self:
  `self.i`, `self.j`
- All methods take `self` as first argument in the declaration, but not in the call

```
inst1 = MyBase(6,9); inst1.write()
```

Subclasses work as we are used to

- Class `MySub` is a subclass of `MyBase`:

```
class MySub(MyBase):

    def __init__(self,i,j,k):  # constructor
        MyBase.__init__(self,i,j)
        self.k = k;

   def write(self):
        print 'MySub: i=',self.i,'j=',self.j,'k=',self.k
```

- Example:

```
# this function works with any object that has a write func:
def write(v): v.write()

# make a MySub instance
i = MySub(7,8,9)

write(i)   # will call MySub's write
```

The magical `self`

- The explicit `self` argument in instance methods is one of the most debated details of the Python language
- Why is it there?
  - Distinguish between local (method) variables and instance variables
  - The definition matches how instance methods are actually called, with the instance passed as the first argument
- For an instance `obj` of class `class`, these calls are equivalent:

```
obj.someMethod()
class.someMethod(obj)
```

New-style vs classic classes

- The class concept was rewritten in Python 2.2, but the old style was retained for backward compatibility.
- New-style classes are sub-classes of `object`:

```
class Bar(object): pass    #defines a new-style class
class Bar: pass               #defines a classic class
```

- The difference is small, but new-style classes are recommended
- From Python 3.0, all classes are new-style, no need for the explicit `object` base class

Instance attributes

- Instance attributes are prefixed with self, and normally added in the constructor;

```
>>> class Point(object):
               def __init__(self,x,y):
               self.x = x
               self.y = y
```

- Can also be added to a specific instance;

```
>>> origo = point(0.0,0.0)
>>> origo.z = 0.0
```

Class attributes (1)

- Class variables are common to the class;

```
>>>class Point(object):
              counter = 0
              def __init__(self,x,y):
                     self.x = x
                     self.y = y
                     Point.counter +=1
>>> p0 = Point(0.0,0.0)
>>> p1 = Point(1.0,1.0)
>>> Point.counter
2
>>> p0.counter
2
```

Class attributes (2)

- Warning; class attributes can be accessed through the instance (as above), but assigning or modifying creates an *instance variable with the same name*;

```
>>> p0 = Point(0,0)
>>> p0.counter
1
>>> p0.__dict__
{'y': 0, 'x': 0}
>>> p0.__class__.__dict__
{'__module__': '__main__', 'counter': 1, '__doc__': None, '__ini
>>> p0.counter +=1
>>> p0.__dict__
{'y': 0, 'x': 0, 'counter': 2}
>>> p0.__class__.__dict__
{'__module__': '__main__', 'counter': 1, '__doc__': None, '__ini
```

Static methods and class methods (1)

- New-style classes support *static methods* and *class methods*
- Both can be called without having an instance of the class
- Static method;
  - No knowledge of the class it belongs to
  - Declared as a regular function, without `self` or other class or instance related arguments
  - No implicit passing of instance or class when called
  - Defined using the decorator `@staticmethod`

```
class A(object):
      @staticmethod
      def method1():
            pass
      #or old style; method1 = staticmethod(method1)
```

  - Not widely used in Python

Static methods and class methods (2)

- Class method;
  - The first argument is the class, by convention named `cls`
  - When calling, the class is passed implicitly (just as with `self` for instance methods
  - Defined using decorator `@classmethod`;

```
class A(object):
        instances = {}

        @classmethod
        def method1(cls):
                print cls.instances
```

  - Commonly used as alternative constructors, to enable alternative ways of constructing an instance of the class

Special attributes

`i1` is `MyBase`, `i2` is `MySub`

- Dictionary of user-defined attributes:

```
>>> i1.__dict__  # dictionary of user-defined attributes
{'i': 5, 'j': 7}
>>> i2.__dict__
{'i': 7, 'k': 9, 'j': 8}
```

- Name of class, name of method:

```
>>> i2.__class__.__name__ # name of class
'MySub'
>>> i2.write.__name__     # name of method
'write'
```

- List names of all methods and attributes:

```
>>> dir(i2)
['__doc__', '__init__', '__module__', 'i', 'j', 'k', 'write']
```

Special methods

- Special methods have leading and trailing double underscores (e.g. `__str__`)
- Here are some operations defined by special methods:

```
len(a)                 # a.__len__()
print a                #calls a.__str__()
repr(a)                #a.__repr__()
c = a*b                # c = a.__mul__(b)
a = a+b                # a = a.__add__(b)
d = a[3]               # d = a.__getitem__(3)
a[3] = 0               # a.__setitem__(3, 0)
f = a(1.2, True)       # f = a.__call__(1.2, True)
if a:                  # if a.__len__()>0: or if a.__nonzero__():
```

Python functions summary

- Most of the “unusal” features relate to functions being objects
- Closure;
  - A function remembers the surrounding namespace from when it was defined
  - Becomes a function object with some additional attributes
  - Definition and use very similar to OOP
- Decorators;
  - A function that takes a function as argument and returns a modified function
  - `@decorator` syntax simply a short cut for the standard function call `f = decorator(f)`
  - Most common decorators; `@classmethod`, `@staticmethod`

Python classes summary

- Classes and objects work similar to other languages
- A class is also an object
- Lack of declared variables leads to some unfamilar behavior;
  - Instance attributes are added in functions (normally init function) instead of in the class
  - `self`,`cls` used to distinguish instance and class variables from local and global variables
- Special methods implement common operations on instances;
  - Examples are `__init__`, `__call__`, `__getitem__`, `__setitem__`, `__str__`, `__repr`
  - Complete list at `http://docs.python.org/2/reference/datamodel.html`