# Computational Graphics: Lecture 3

Francesco Furiani
(fra.uni@furio.me)

Università degli Studi Roma TRE

Tue, Mar 4, 2014

# What is it?

In a functional program, input flows through a set of functions: each function operates on its input and produces some output.
Functional style discourages functions with side effects that modify internal state or make other changes that arent visible in the functions return value.
Functions that have no side effects at all are called purely functional.
Avoiding side effects means not using data structures that get updated as a program runs; every functions output must only depend on its input.

# But i miss OOP

Objects are little capsules containing some internal state along with a collection of method calls that let you modify this state, and programs consist of making the right set of state changes.
Functional programming wants to avoid state changes as much as possible and works with data flowing between functions.
We're then using a paradigm that is the opposite of object-oriented programming.

# Advantages

There are theoretical and practical advantages to the odd constraint that the functional style imposes:

- Formal provability
- Modularity
- Composability
- Ease of debugging and testing

# Examples

## Iterative

```
def post_process(data):
    newdict = {}
    for k, v in data.iteritems():
        if k == 'df_use_percent':
            v = v.rstrip('%')
        newdict[k] = v
    return newdict
```

## Functional

```
def post_process(data):
    def remove_percent(k, v):
        if k == 'df_use_percent':
            v = v.rstrip('%')
        return (k, v)
    return dict([remove_percent(k, v)
                 for k, v in data.iteritems()])
```

# Partial application

In the context of a functional style, is useful to have functions that accept only a part of the arguments returning a partial state that could be completed on another call.

```
def somma(x,y): return x+y
```

The function somma needs all of its arguments to work, how can we redesign it?

```
def somma(x): return lambda y: x+y
>>> somma(2)(4)
6
>>> a = somma(2)
>>> type(a)
<type 'function'>
>>> a(7)
9
>>> a(15)
17
```

# Partial application

This is a typical approach of functional languages that is supported in Python (although being multi-paradigm Python cannot considered a fully functional language because of the lack of functional interfaces in some of its standard library).

```
def S(n):
  return lambda x: x[int(n)-1]
```

What the function `S` does once called?

```
def AA(f):
  def AA0(args): return map(f, args)
  return AA0
```

And the function `AA`?

# Closures

A function or reference to a function together with a referencing environment. Closures are used to implement continuation-passing style, and in this manner, hide state.
Closures are typically implemented with a special data structure that contains a pointer to the function code, plus a representation of the set of available variables at the time when the closure was created.

# Closures

Using partial function, is then an application of closures. Is possible then to build any number of these functions to use them at any time.

```
>>> somme = [lambda x: x+i for i in range(1,4)]
>>> somme
[<function <lambda> at 0x25ffaa0>, <function <lambda> at 0x25ff9b0
    >, <function <lambda> at 0x25ffc80>]
>>> somme[0](10)
13
>>> somme[1](10)
13
>>> somme[2](10)
13
>>> i
3
```

Wait what happened there?

# Closures

Let's go back to closure and functional definitions:

- *functions that have no side effects at all are called purely functional*
- *reference to a function together with a referencing environment*

`lambda x: x+i` is indeed a function, but it has side-effects... why? The linked environment is the global one!
In the global environment the `i` is mutating!
How to fix this?

# Closures

Let's try to change the referenced environment!

```
>>> somme = [lambda x,z=i: x+z for i in range(1,4)]
>>> somme[0](10)
11
>>> somme[1](10)
12
>>> somme[2](10)
13
```

Using the default assignment of the function parameter we changed the binding environment!
The produced closure contains an environment in which the *i* value is bind to the *z* of the function.

# Modules

A module allows you to logically organize your Python code. A module is a Python object with arbitrarily named attributes that you can bind and reference.
A module is, simply, a file consisting of Python code that can contains functions, classes and variables. A module can also include runnable code. You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax:

```
import module1[, module2[,... moduleN]
```

# Modules

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. For example, to import the module `support.py`, you need to put the following command at the top of the script:

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

# Modules

Python's from statement lets you import specific attributes from a module into the current namespace. The `from...import` has the following syntax:

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

# Modules

It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

# ... and classes?

Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data.

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

# ... and classes?

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self,other):
        return Vector(self.a + other.a, self.b + other.b)

>> v1 = Vector(2,10)
>> v1.a
2
>> v2 = Vector(5,-2)
>> print v1 + v2
Vector(7,8)
```

# ... and classes?

The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name BaseClassName must be defined in a scope containing the derived class definition but other arbitrary expressions are also allowed:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered: this is used for resolving attribute references.

# ... and classes?

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.
New-style classes, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method (*C3 algorithm*).

# ... and classes?

*Private* instance variables that cannot be accessed except from inside an object don't exist in Python.
Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

# ... and ~~classes~~ structs?

Sometimes it is useful to have a data type similar to the Pascal record or C struct, bundling together a few named data items. An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

# Real life applications

# References

Official

- Site
- Docs

Books

- Think Python (FREE)
- Python in a Nutshell

Tutorial

- Quick tutorial
- Learn Python the hard way


Programming cat
Is not amused by your HTML structure.