Functional Programming

What? (Lisp, Scheme, Ocaml, Haskell)

- Functions treated as objects?
- Recursion´s domination
- List Processing?
- No statements.. but expressions?
- What is more important that How
- Higher order functions...
- Mathematical look?

Advantages?

Pure/Clean/Short code?

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

Lambda

On the fly functions? or simply expressions?

```python

if <cond1>:
    func1()
elif <cond2>:
    func2()
else:
    func3()


(<cond1> and func1()) or (<cond2> and func2())
                      or (func3())



>>> x = 3
>>> def pr(s): return s
>>> (x==1 and pr('one')) or (x==2 and pr('two'))
```

```
                                        or (pr('other'))
'other'
>>> x = 2
>>> (x==1 and pr('one')) or (x==2 and pr('two'))
                                        or (pr('other'))
'two'


>>> pr = lambda s:s
>>> namenum = lambda x: (x==1 and pr("one"))
                        or (x==2 and pr("two"))
                        or (pr("other"))
>>> namenum(1) 'one'
>>> namenum(2) 'two'
>>> namenum(3) 'other'
```

Map, Reduce, Filter

Refresh
(Replacing FOR loops?)

```
for e in lst: func(e)
map(func, lst)


do_it = lambda f: f()
(let f1, f2, f3 (etc) be functions)
map(do_it, (f1,f2,f3))
```

While loops

They can be replaced too. But why?
Saves trouble by not giving values to variables.
How?
Example: Print Big Products.

```

xs = (1,2,3,4)
ys = (10,15,3,22)
bigmuls = ()
...more stuff...
for x in xs:
    for y in ys:
        if x*y > 25:
            bigmuls.append((x,y))
print bigmuls
```

Functional Way?

```
bigmuls = lambda xs,ys:
  filter(lambda (x,y):x*y > 25, combine(xs,ys))
combine =
  lambda xs,ys: map(None, xs*len(ys),
                    dupelms(ys,len(xs)))
dupelms =
  lambda lst,n: reduce(lambda s,t:s+t,
                  map(lambda l,n=n: (l)*n, lst))

print bigmuls((1,2,3,4),(10,15,3,22))
```

```
print ((x,y) for x in (1,2,3,4)
             for y in (10,15,3,22)
                       if x*y > 25)
```

Functional Module

functional provides Python users with numerous tools common in functional programming, such as foldl, foldr, flip, as well as mechanisms for partial function application and function composition.
functional comes in two flavours: one is written in a combination of C and Python, focusing on performance. The second is written in pure Python and emphasises code readability and portability.

`compose(outer, inner, unpack=False)`
compose implements function composition. In other words, it returns a wrapper around the outer and inner callables, such that the return value from inner is fed directly to outer.

```
>>> def add(a, b):
...     return a + b
...
>>> def double(a):
...     return 2 * a
...
>>> compose(double, add)(5, 6)
22
```

`flip(func)`

flip wraps the callable in func, causing it to receive its non-keyword arguments in reverse order.

```
>>> def triple(a, b, c):
...     return (a, b, c)
...
>>> triple(5, 6, 7)
(5, 6, 7)
>>>
>>> flipped_triple = flip(triple)
>>> flipped_triple(5, 6, 7)
(7, 6, 5)
```

`foldl(func, start, iterable)`

foldl takes a binary function, a starting value (usually some kind of ’zero’), and an iterable. The function is applied to the starting value and the first element of the list, then the result of that and the second element of the list, then the result of that and the third element of the list, and so on.

```

foldl(f, 0, (1, 2, 3))

f(f(f(0, 1), 2), 3)


def foldl(func, start, seq):
```

```
    if len(seq) == 0:
        return start

    return foldl(func,
          func(start, seq[0]), seq[1:])
```

Documentation

Functional Programming Howto - python.org
http://docs.python.org/dev/howto/functional.htm

Chained Decorators

We saw decorators already.
No one stops us from decorating a function twice (or more)

```
@synchronized
@logging
def myfunc(arg1, arg2, ...):
        ...do something
```

decorators are equivalent to:

`myfunc = synchronized(logging(myfunc))`

Nested in that declaration order

Bad Decoration (no function/callable is returned)

```
>>> def spamdef(fn):
...         print "spam, spam, spam"
...
>>> @spamdef
... def useful(a, b):
...         print a**2 + b**2
...
spam, spam, spam
>>> useful(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'NoneType' object is not callable
```

Class Factory

Decorators do not let you modify class instantiation, but can massage the methods.
No adjustments @ instantiation, but can change the behaviour at runtime.
Now technically, a decorator applies when a class statement is run, which for top-level classes is closer to "compile time" than to "runtime."

```
def arg_sayer(what):
    def what_sayer(meth):
        def new(self, *args, **kws):
            print what
            return meth(self, *args, **kws)
        return new
    return what_sayer

def FooMaker(word):
    class Foo(object):
        @arg_sayer(word)
        def say(self): pass
    return Foo()

foo1 = FooMaker('this')
foo2 = FooMaker('that')
print type(foo1),; foo1.say()
```

```
Output : <class '__main__.Foo'> this
print type(foo2),; foo2.say()
Output : <class '__main__.Foo'> that
```

- The Foo.say() method has different behaviors for different instances.
- The undecorated Foo.say() method in this case is a simple placeholder, with the entire behavior determined by the decorator.
- As already observed, the modification of Foo.say() is determined strictly at runtime, via the use of the FooMaker() class factory.
- The decorator is parameterized. Or rather arg_sayer() itself is not really a decorator at all; rather, the function returned by arg_sayer(), namely what_sayer(), is a decorator function that uses a closure to encapsulate its data. Parameterized decorators are common, but they wind up needed functions nested three-levels deep.

Artificial MetaClass

Decorators cannot completely modify the behaviour of classes.
But they can modify the `__new__()` method.
(Will see `_metaclass_` next week.)

```
def flaz(self): return 'flaz'
def flam(self): return 'flam'

def change_methods(new):
    "Warning: Only decorate the __new__() method
    if new.__name__ != '__new__':
        return new
    def __new__(cls, *args, **kws):
        cls.flaz = flaz
        cls.flam = flam
        if hasattr(cls, 'say'): del cls.say
        return super(cls.__class__, cls).__new__(
    return __new__

class Foo(object):
    @change_methods
    def __new__(): pass
```

```
    def say(self): print "Hi me:", self

foo = Foo()
print foo.flaz()
flaz
foo.say()
'Foo' object has no attribute 'say'
```

Hmm... careful.

Some Class things

- Pass self manually
- Check for propery and method existence
- Modify classs after creation

```
class Class:
    def a_method(self):
        print 'Hey a method'

instance = Class()

instance.a_method()
'Hey a method'

Class.a_method(instance)
'Hey a method'


class Class:
    answer = 42

hasattr(Class, 'answer')
```

```
True
hasattr(Class, 'question')
False

getattr(Class, 'answer')
42
getattr(Class, 'question', 'What is six times nir
'What is six times nine?'
getattr(Class, 'question')
AttributeError

class Class:
    def method(self):
        print 'Hey a method'

instance = Class()
instance.method()
```

```
'Hey a method'

def new_method(self):
    print 'New method wins!'

Class.method = new_method
instance.method()
'New method wins!'
```

Needless to mention, modifying classes is not a great idea.

Resources

- http://www.siafoo.net/article/52
- http://wiki.python.org/moin/PythonSpeed/Perf

Eg. String Concatenation: Use "join"

```
newlist = []
for word in oldlist:
      newlist.append(word.upper())

..............
upper = str.upper
newlist = []
append = newlist.append
for word in oldlist:
      append(upper(word))



—— 40k Words ——

Version Time (seconds)
Basic loop 3.47
```

```
Eliminate dots 2.45
Using map function 0.54
```

```
wdict = {}
for word in words:
    if word not in wdict:
        wdict(word) = 0
    wdict(word) += 1


wdict = {}
for word in words:
    try:
        wdict(word) += 1
    except KeyError:
        wdict(word) = 1


wdict = {}
get = wdict.get
```

```
for word in words:
    wdict(word) = get(word, 0) + 1
```

More or less same time taken now.