Rethink programming: a functional approach

Francesco Bruni

PyConSette - Florence, April '16

Notebook @ http://ern.is/fp

@brunifrancesco

Who I am

- MSc in Telecommunication Engineering @ Poliba
- Despite a Java background, I prefer Python whenever possible
- I'm not a computer scientist :)

Agenda

- Why functional programming
    - Everything's object
    - Laziness: why evaluation matters
    - Immutable data
    - Recursion and/or cycles
    - Pattern matching
- Mixing OOP with FP
- FP "patterns"
- Conclusions

Disclaimer

I'm not a vacuum cleaner salesman.

Why functional programming

- Think in terms of **functions**
- *Function evaluation* instead of *state change* and/or *mutable objects*
- Testing is easy
- A new viewpoint is required

(MATH) What is a function?

A **relation** from a set of inputs (X) to a set of possible outputs (Y) where **each** input is related to **exactly** one output.

In [1]:

```python
import random
def sum_imperative():
    res = 0
    for n in [random.random() for _ in range(100)]:
        res += n
    return res

def sum_functional():
    return sum(random.random() for _ in range(100))

print(sum_imperative())
print(sum_functional())
assert True
```

```
42.90290739224947
54.64527198535234
```

Functional features in Python

- Not all functional patterns apply to Python
- Hybrid approach is often requested
- Other libraries needs to be (eventually) integrated

(MATH) Function composition

A pointwise application of one function to the result of another to produce a third function.

Given:

- $f : X \to Y$
- $g : Y \to Z$

$$g \circ f : X \to Z$$

First class (or high order) functions

- Since everything's object
- Functions are objects too (with fields and methods)

In [2]:

```python
class Fn:
    pass

def fn():
    """
    Dummy function
    """
    return 1+2

print(fn.__doc__.strip())
print(Fn.__name__)
```

```
Dummy function
Fn
```

High order functions

- Functions accepting functions as params
- Functions returning other functions
- **filter**, **map**, **reduce** (now in *functools* module)

In [3]:
```python
def mapper_func(lst, reduce_func):
    """
    Apply a function to each member of <lst>
    """
    return map(reduce_func, lst)

def check_if_neg(value):
        """
        Check if values is neg
        """
        return value > 0


assert list(mapper_func([1,2,3], str))  == ["1", "2", "3"]
assert list(mapper_func(["1","2","3"], int))  == [1, 2, 3]
```

In [4]:

```python
def v3(v):
    return v**3

def v2(v):
    return v**2

def my_awesome_function(v,h):
    return(h(v))

assert my_awesome_function(3,v2) == 9
assert my_awesome_function(3,v3) == 27
```

(MATH) λ calculus

- A formal system to analyze functions and their calculus
- It deals with rewriting functions with simplified terms
- A formal definition of *λ term*:

$$\Lambda ::= X \mid (\Lambda\Lambda) \mid \lambda X.\Lambda$$

In [5]:

```python
from functools import reduce

def filter_negative_numbers(lst):
    """
    Filter negative numbers from <lst>
    """
    return filter(check_if_neg, lst)

def reduce_sum(lst):
    """
    Reduce list summing up consecutives elements
    """
    return reduce(lambda x, y: x+y, lst)

assert list(filter_negative_numbers([-3, 4, 5, -10, 20])) == [4, 5, 20]
assert reduce_sum([-3, 4, 5, -10, 20]) == 16
```

More about function composition

- Use class like syntax to compose complex functions

In [6]:

```
from collections.abc import Callable
from random import random
from statistics import mean

class MapReduceFunction(Callable):
    """
    'Chain' two functions to perform map reduce;
    the class returns a new callable object
    """
    def __init__(self, map_function, reduce_function):
        self.map_function = map_function
        self.reduce_function = reduce_function

    def __call__(self, value):
        return map(lambda item: self.reduce_function(item), self.map_function(valu
e))

data = [round(random()*10, 3) for _ in range(0, 23)]
mr = MapReduceFunction(
    map_function=lambda item: zip(*[iter(data)] * 7),
    reduce_function=lambda item: max(item)
)
```

Pure functions

- Functions that cannot include any assignement statement
- No side effects
- What about default param values?
- Is a IO based function pure by default?

In [7]:
```
import random
def filter_out(result=[random.random() for _ in range(0, 5)]):

    exclude = map(lambda item: item ** 2, range(30))

    result = filter(lambda item: item not in exclude, result)

    sorted_result = sorted(result, key=lambda item: str(item)[1])

    return map(lambda item: round(item, 2), sorted_result)

filter_out()
```

Out[7]:
```
<map at 0x108dd5630>
```

Practical considerations of λ functions

- Inline functions
- Concise
- No need of defining *one time* functions
- Overusing them is not a solution
- λ function assignment is discouraged (PEP8)

Immutable vs mutable data structure

- Can a variable don't **vary** anymore?

In [8]:

```
value = 100

def change_f_value(new_f_value=5):
    value = new_f_value
    print("Value in function %s" %value)

print("Initialized value %s "%value)
change_f_value()
print("Final value %s "%value)
```

```
Initialized value 100
Value in function 5
Final value 100
```

Function scopes and closures

- "If a name is bound anywhere within a code block, all uses of the name within the block are treated as references to the current block." (PEP 227)
- What if we wanted change the *value* variable?

```
In [9]: class Foo:
            def __init__(self, value):
                self.value = value

        foo_obj = Foo(value=10)

        def func(obj):
            obj.value = 3

        print("Object ID: %i" %id(foo_obj))
        print("Object 'value' field before applying function: %i" %foo_obj.value)
        func(foo_obj)
        print("Object 'value' field after applying function: %i" %foo_obj.value)
        print("Object ID: %i" %id(foo_obj))
```

```
Object ID: 4443530520
Object 'value' field before applying function: 10
Object 'value' field after applying function: 3
Object ID: 4443530520
```

Data mutation

- *foo_obj* didn't change
- **foo_obj.value** changed!
- So, foo_obj changed or not? If so, can you always determine who changed it?

Immutability

- Don't change existing objects, use new ones :)

In [10]:

```
import random
import pprint
from collections import namedtuple

data = str(random.random() + 4)
MyObj = namedtuple("MyClassReplacement",("some_string", "my_smart_function",))
o = MyObj(
    some_string=data,
    my_smart_function=lambda item: float(item)*3)
some_string, some_function = o

o2 = o._replace(some_string="a new dummy string")
assert(o.my_smart_function(o.some_string) == float(o.some_string) * 3)
assert (some_string == data)
assert not id(o) == id(o2)
```

Strict vs not strict evaluation

- Strict evaluation requires that all operators needs to be evaluated
- Non strict (or lazy) evaluation, evaluates expression if and when requested
- Careful with lazy evaluated structures

(MATH) A dummy truth table

p	**q**	**(p & q) | !q**
T	T	?
T	F	?
F	T	?
F	F	?

In [11]:

```
import random

generate_random_list = lambda size: [random.choice([True, False]) for _ in
range(0, size)]

def all_true_values(lst):
    print("evaluating ALL true values")
    return all(lst)

def any_true_value(lst):
    print("evaluating ANY true values")
    return any(lst)

all_true_values(generate_random_list(size=10)) and any_true_value(generate_random_
list(size=10))
print("+++++++++++")
all_true_values(generate_random_list(size=10)) or any_true_value(generate_random_l
ist(size=10))
```

```
evaluating ALL true values
+++++++++++
evaluating ALL true values
evaluating ANY true values
```

Out[11]: True

Use case: Python iterables structures

- Creating lists requires time/space
- What if you don't need it anymore?
- Be lazy: use functions to **generate** the *next* element you need
- **asyncio** was inspired by the *generator approach* :)

In [12]:
```
import random

def lc():
    return [random.random() for _ in range(0, 10)]

def _iter():
    return iter([random.random() for _ in range(0, 10)])

def lazy_1():
     for item in range(10, size):
        if not item % 2 and not str(item).endswith("4"):
            yield item

def lazy_2():
    yield from (r for r in range(0, 10) if not r % 2 and not str(r).endswith("4"))

print(lc())
print(_iter())
print(lazy_1())
print(lazy_2())
```

```
[0.3715365641589854, 0.21968153174666838, 0.5694550450864405, 0.67849617189266
75, 0.12265891948697638, 0.9803208951269902, 0.9661576370333822, 0.69911857951
80963, 0.9940147002373155, 0.4647425397290714]
<list_iterator object at 0x108e2fdd8>
<generator object lazy_1 at 0x108dd36c0>
<generator object lazy_2 at 0x108dd36c0>
```

Recursion vs loop

- Functional programming relies on recursion instead of iteration
- Python suffers by recursion limit
- Python doesn't offer any tail call optimization
- Use iterations :)

In [13]:

```
def facti(n):
    if n == 0: return 1
    f= 1
    for i in range(2,n):
        f *= i
    return f

def fact_nt(n):
    if n == 0: return 1
    else: return n*fact(n-1)

def fact(n, acc=1):
    if n == 0:
        return acc
    return fact(n-1, acc*n)
```

Currying

- Multiple arguments functions mapped to single arguments functions

In [14]:
```
def mult(a):
    def wrapper_1(b):
        def wrapper_2(c):
            return a*b*c
        return wrapper_2
    return wrapper_1

def mult2(a, b, c):
    return a*b*c

assert(mult(2)(3)(4) == mult2(2,3,4))
```

In [15]:
```
mult1 = mult(2)
mult12 = mult1(3)
mult12(4)
```

Out[15]:
```
24
```

Partials

Python provides "partial" functions for manual currying

In [16]:
```
from functools import reduce, partial
import operator

def sum_random_numbers(size):
    return reduce(operator.add, (random.random())*size)

def mul_random_numbers(size):
    return reduce(operator.mul, (random.random())*size)

def handle_random_numbers(size, function):
    return reduce(function, (random.random())*size)

two_random_sum = partial(sum_random_numbers, size=2)
three_random_sum = partial(sum_random_numbers, size=3)

two_random_pow = partial(mul_random_numbers, size=2)
five_random_product = partial(mul_random_numbers, size=5)

three_random_sum = partial(handle_random_numbers, function=operator.add, size=3)
three_random_mod = partial(handle_random_numbers, function=operator.mod, size=3)
```

Decorators

- Return a modified version of a decorated function
- Add properties at runtime, before using the actual decorated function

In [17]:
```
from functools import wraps
from functools import partial

def get_ned_data(n):
    def get_doubled_data(func, *args, **kwargs):
        @wraps(func)
        def _inner(*args, **kwargs):
            kwargs["multiplied_by_n_param"] = kwargs["initial_param"]*n
            return func(*args, **kwargs)
        return _inner
    return get_doubled_data
```

In [18]:

```
@get_ned_data(n=2)
def double_func(*args, **kwargs):
    assert(kwargs["multiplied_by_n_param"] == kwargs["initial_param"]*2)

@get_ned_data(n=3)
def triple_func(*args, **kwargs):
    assert(kwargs["multiplied_by_n_param"] == kwargs["initial_param"]*3)

double_func(initial_param=3)
triple_func(initial_param=5)
```

FP patterns

- OOP and FP
- Monads
- Memoization
- Actor model
- Pattern matching

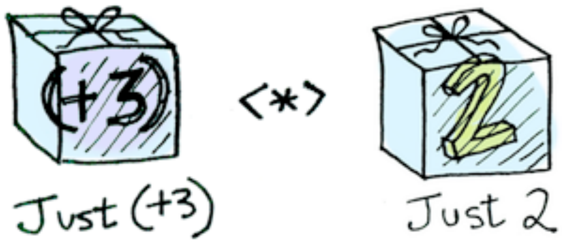
OOP and FP

- Use functions to set the strategy
- Use decorators to change function behaviour

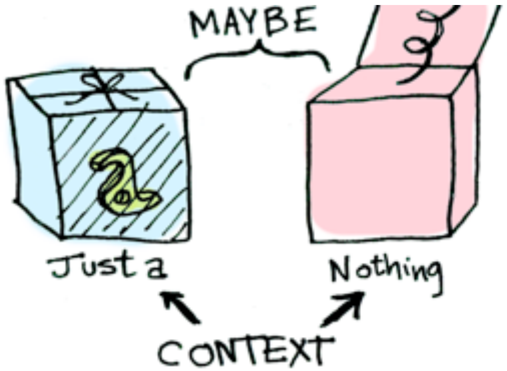
(MATH) Category theory

- Formalize mathematical structures
- Category *C* is characterized by:
  - $ob(C)$, as a set of objects;
  - $\forall (A, B) \in ob(C), f : A \rightarrow B$. defines a *morphism*
- Set/Functions is a category

Functors apply a function to a wrapped value



Applicatives apply a wrapped function to a wrapped value



Monads apply a function that returns a wrapped value to a wrapped value

Images by adit.io

In [19]:

```
class Container:
    def __init__(self, value=None):
        self.value = value

    def map(self, function):
        try:
            return Full(function(self.value))
        except Exception as e:
            return Empty()


class Empty(Container):
    def map(self, value):
        return Empty()

    def __str__(self):
        return "Container's empty"

class Full(Container):
    def __str__(self):
        return self.value

    def get_or(self, none_value=None):
        return self.value or none_value
```

In [20]:

```
from fn.monad import optionable
from collections import namedtuple

def get(request, *args, **kwargs):
    @optionable
    def _get_values(data):
        return data.get("values", None)
    _split = lambda item: item.split(",")
    _strip = lambda item: item.replace(" ", "")
    _filter = lambda item: list(filter(lambda i: i, item))
    return _get_values(request.body)\
        .map(_strip)\
        .map(_split)\
        .map(_filter)\
        .get_or(["v1,v2"])


req_class = namedtuple("Request", ("body",))

request_1 = req_class(dict(values="v1, v2,v3"))
request_2 = req_class(dict(values="v1,v2,v3 "))
request_3 = req_class(dict(values="v1, v2,v3, "))

assert(get(request_1) == ['v1', 'v2', 'v3'])
assert(get(request_2) == ['v1', 'v2', 'v3'])
assert(get(request_3) == ['v1', 'v2', 'v3'])
```

In [21]:
```python
from pymonad import List

_strip = lambda item: List(item.replace(" ", ""))
_slice = lambda item: List(item[:-1] if item[-1] =="," else item)
_split = lambda item: List(*item.split(","))


List("v1, v2,v3, ") >> _strip >> _slice >> _split
```

Out[21]:
```
['v1', 'v2', 'v3']
```

Memoization

- Enjoy *referential transparency*: do not compute functions for the same input

In [22]:
```
import functools

def memoize(obj):
     cache = obj.cache = {}

     @functools.wraps(obj)
     def memoizer(*args, **kwargs):
         if args not in cache:
             cache[args] = obj(*args, **kwargs)
         return cache[args]
     return memoizer

@memoize
def fact_m(n, acc=1):
    if n == 0:
        return acc
    return fact(n-1, acc*n)
```

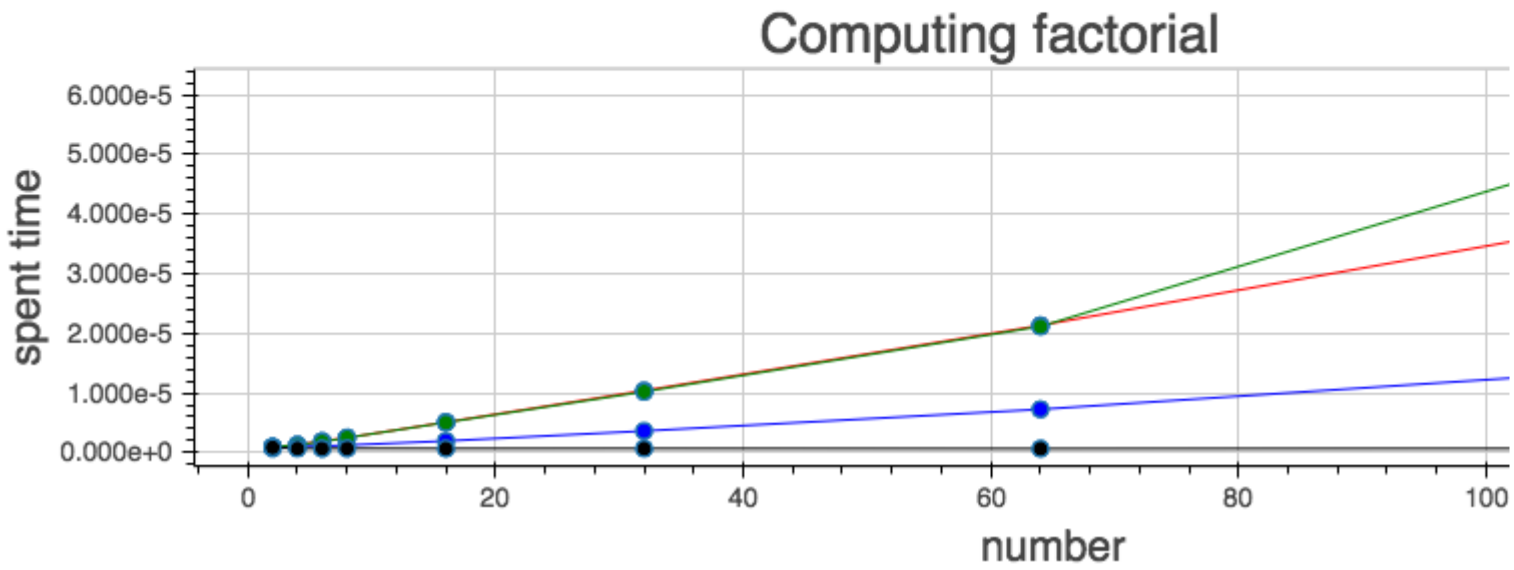
In [28]:

```
from bokeh.plotting import figure, output_file, show, output_notebook
from cmp import get_data, DEF_VALUES

y = get_data()
p = figure(title="Computing factorial", x_axis_label='number', y_axis_label='spent
 time' ,tools="pan,box_zoom,reset, save",plot_width=1000, plot_height=300)
p.line(DEF_VALUES, y[0][1], legend=y[0][0], line_width=1,line_color="blue")
p.circle(DEF_VALUES, y[0][1],fill_color="blue", size=8)
p.line(DEF_VALUES, y[1][1], legend=y[1][0], line_width=1,line_color="red")
p.circle(DEF_VALUES, y[1][1],fill_color="red", size=8)
p.line(DEF_VALUES, y[2][1], legend=y[2][0], line_width=1,line_color="green")
p.circle(DEF_VALUES, y[2][1],fill_color="green", size=8)
p.line(DEF_VALUES, y[3][1], legend=y[3][0], line_width=1,line_color="black")
p.circle(DEF_VALUES, y[3][1],fill_color="black", size=8)

output_notebook(hide_banner=True)
show(p)
```

(http://bokeh.pydata.org/)



Actors

- Message passing pattern
- Actors receive a message, do something, return new messages (or None)
- They behave like humans

In [29]:
```
%run ./actors.py
```

```
21:54:55 [p=12010, t=140736514905024, INFO, pulsar.arbiter] mailbox serving on
 127.0.0.1:58006
21:54:55 [p=12010, t=140736514905024, INFO, pulsar.arbiter] started
21:54:56 [p=12047, t=140736514905024, INFO, pulsar.actor1] started
21:54:56 [p=12048, t=140736514905024, INFO, pulsar.actor2] started
Got the message
<Task finished coro=<request() done, defined at /Users/boss/git/talk3/lib/pyth
on3.4/site-packages/pulsar/async/mailbox.py:279> result=None>
Message sent
21:54:57 [p=12010, t=140736514905024, INFO, pulsar.arbiter] Stopping actor1(ia
7a6e0c).
21:54:57 [p=12010, t=140736514905024, INFO, pulsar.arbiter] Stopping actor2(if
eb7f3c).
21:54:57 [p=12047, t=140736514905024, INFO, pulsar.actor1] Bye from "actor1(ia
7a6e0c)"
21:54:57 [p=12048, t=140736514905024, INFO, pulsar.actor2] Bye from "actor2(if
eb7f3c)"
21:54:58 [p=12010, t=140736514905024, WARNING, pulsar.arbiter] Removed actor1
(ia7a6e0c)
21:54:58 [p=12010, t=140736514905024, WARNING, pulsar.arbiter] Removed actor2
(ifeb7f3c)

Bye (exit code = 0)
```

Pattern matching

- Match data over patterns and apply a function
- Scala's pattern matching involve types/expressions/object deconstruction
- Implemented via *multimethods*, a kind of method overloading

Useful libraries

- **fn.py**
- **pyMonad**
- **pyrsistent**
- **toolz**
- **pykka**
- **pulsar**
- **cleveland**
- **underscore.py** (ported from underscore_js)

Useful readings

- Functional Python Programming
- Becoming functional
- Learn Haskell at your own good
- Functional Javascript
- Functional programming in Scala

Going functional is not just about coding

- Lambda Architecture
- FAAS: AWS lambda

Summary

“The point is not that imperative programming is broken in some way, or that functional programming offers such a vastly superior technology. The point is that functional programming leads to a change in viewpoint that can—in many cases—be very helpful.”

Questions?

map(answer, questions)