What is functional programming?
- Functions are first class (objects).
- That is, everything you can do with "data" can be done with functions themselves (such as passing a function to another function).
- Recursion is used as a primary control structure.
- In some languages, no other "loop" construct exists.
- There is a focus on list processing.
- Lists are often used with recursion on sub-lists as a substitute for loops.
- "Pure" functional languages eschew side-effects.
- This excludes assignments to track the program state.
- This discourages the use of statements in favor of expression evaluations.

Whys
- All these characteristics make for more rapidly developed, shorter, and less bug-prone code.
- A lot easier to prove formal properties of functional languages and programs than of imperative languages and programs.

Functional Programming in Python
map(), reduce() & filter()
Python has functional capability since its first release
- with new releases just a few syntactical sugar has been added

Basic elements of functional programming in Python are:
- map(): it applies a function to a sequence
  ```python
  >>> import math, functools
  >>> print(list(map(math.sqrt, [x**2 for x in range(1,11)])))
  [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
  ```
- filter(): it extracts from a list those elements which verify the passed function
  ```python
  >>> def odd(x):
  ...     return (x%2 != 0)
  >>> print(list(filter(odd, range(1,30))))
  [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
  ```
- reduce(): it reduces a list to a single element according to the passed function
  ```python
  >>> def sum(x,y):
  ...     return x+y
  >>> print(functools.reduce(sum, range(1000)))
  499500
  ```
  Note, map() and filter() return an iterator rather than a list.

Eliminating Flow Control Statements: If
- Short-Circuit Conditional Calls instead of if

```python
if __name__ == '__main__':
    for i in range(3):
        print("cond({0}) :- {1}".format(i,cond(i)))
```
```python
[17:34] cazzola@ulik:~/esercizi-pa>python3 fcond.py
cond(0) :- other
cond(1) :- one
cond(2) :- two
```

Doing some abstraction
```python
block = lambda s: s
cond = \n    lambda x:
        (x==1 and block("one")) or (x==2 and block("two")) or block("other")
if __name__ == '__main__':
    print("cond({0}) :- {1}".format(3,cond(3)))
```
```python
[17:55] cazzola@ulik:~/esercizi-pa>vpython3 fcond.py
cond(3) :- other
```
Functional Programming in Python

Do Abstraction: Lambda Functions

The name lambda comes from λ-calculus which uses the Greek letter λ to represent a similar concept.

Lambda is a term used to refer to an anonymous function
- that is, a block of code which can be executed as if it were a function but without a name.

Lambdas can be defined anywhere a legal expression can occur.
A Lambda looks like this:

```
lambda «args»: «an expr on the args»
```

Thus the previous reduce() example could be rewritten as:

```
>>> import functools
>>> print(functools.reduce(lambda i,j: i+j, range(1000)))
499500
```

Alternatively the lambda can be assigned to a variable as in:

```
>>> add = lambda i,j: i+j
>>> print(functools.reduce(add, range(1000)))
499500
```

Evolving Factorial

Traditional implementation

```
def fact(n):
    return 1 if n<=1 else n*fact(n-1)
```

Short-circuit implementation

```
def ffact(n):
    return (n<=1 and 1) or n*ffact(n-1)
```

reduce()-based implementation

```
from functools import reduce
def f2fact(p):
    return reduce(lambda n,m : n*m, range(1,p+1))
```

Eliminating Flow Control Statements: Sequence

Sequential program flow is typical of imperative programming
- it basically relies on side-effects (variable assignments)

This is basically in contrast with the functional approach.

In a list processing style we have:

```
# let's create an execution utility function
do_it = lambda f: f()
# let f1, f2, f3 (etc) be functions that perform actions
map(do_it, [f1,f2,f3]) # map() based action sequence
```

- single statements of the sequence are replaced by functions
- the sequence is realized by mapping an activation function to all the function objects that should compose the sequence.

Eliminating While Statements: Echo

Statement-based echo function

```
def echo_IMP():
    while True:
        x = input("FP -- ")
        if x == 'quit':
            break
        else:
            print(x)
```

Utility function for 'identity with side-effect'

```
def monadic_print(x):
    print(x)
    return x
```

Functional version of the echo function

```
echo_FP = \
    lambda: monadic_print(input("FP -- "))=='quit' or echo_FP()
```

```
if __name__ == '__main__': echo_IMP()
```
Functional Programming in Python

Whys

- To eliminate the side-effects
  - Mostly all errors depend on variables that obtain unexpected values
  - Functional programs bypass the issue by not assigning values to variables at all.

E.g., To determine the pairs whose product is >25.

```python
def bigmuls(xs, ys):
    bigmuls = []
    for x in xs:
        for y in ys:
            if x*y > 25:
                bigmuls.append((x, y))
    return bigmuls

if __name__ == '__main__':
    print(bigmuls((1, 2, 3, 4), (10, 15, 3, 22)))
```

```ruby
from functools import reduce
import itertools
bigmuls = lambda xs, ys: [x_y for x_y in combine(xs, ys) if x_y[0]*x_y[1] > 25]
combine = lambda xs, ys: itertools.zip_longest(xs*len(ys), dupelms(ys, len(xs))

dupelms = lambda lst, n: reduce(lambda s, t: s+t, list(map(lambda l,n=n: [l]*n, lti)))
```

Guido von Rossum finds the `reduce()` too cryptic and prefers to use more ad hoc functions instead

- `sum()`, `any()` and `all()`

To have moved `reduce()` in a module in Python 3 should be a manifest this intent.

References