Functional Programming with Python

Why It's Good To Be Lazy?

Adam Byrtek
adambyrtek@gmail.com

CodeSprinters
Quality Software Fast

EuroPython, Vilnius, July 9th 2008

Agenda

1. Different programming paradigms
2. Functional programming in general
3. Functional features in Python

Algorithm

- How to explain your granny what is programming?
- Algorithm is a recipe how to cook a program
- Actually computers work this way (machine language)
- Called *imperative programming*

Functional programming

- *Functional programming* is a more abstract approach
- Program seen as evaluations of mathematical functions
- More focused on **what** to compute than **how** to compute

Features of functional languages

- Functions as *first-class* objects
- Support for *high-order* functions
- Recursion used instead of loop constructs (*tail recursion* often optimized)
- Lists as basic data structures (see Lisp)
- **Avoiding side effects** (no shared state)

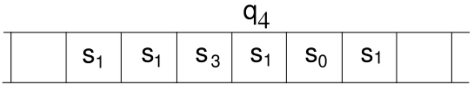
Benefits of stateless programs

- Idempotent (*pure*) functions
- Order of evaluation not defined
- Lazy evaluation possible
- Optimizations
- Concurrent processing
- Easier to test and debug

Side effects can't be eliminated, but can be isolated (*monads*)

Theoretical models

Turing machine (Alan Turing)



Lambda λ calculus (Alonzo Church)

$$\text{TRUE} := \lambda xy.x$$
$$\text{FALSE} := \lambda xy.y$$

Computationally equivalent (Church-Turing thesis)

What about the real world?

- Functional programming is not mainstream
- But it widens your perspective on programming
- Pure functional programming is difficult
- Languages borrow concepts from the functional world
- Recent revival due to a need for concurrency (Erlang)

Functional Python

- Python is **not** a functional language
- But has some functional features...

First-class functions

`Lambda` defines an anonymous function

```
def square(x):
    return x**2
```

equivalent to

```
square = lambda x: x**2
```

Unfortunately no multi-line lambdas (like blocks in Ruby)

First-class functions

Lambda defines an anonymous function

```
def square(x):
    return x**2
```

equivalent to

```
square = lambda x: x**2
```

Unfortunately no multi-line lambdas (like blocks in Ruby)

Closures

Closure is a function with bound variables

```
def build_taxer(rate):
    def taxer(amount):
        return amount * (float(rate) / 100)
    return taxer

vat1 = build_taxer(22)
vat2 = build_taxer(7)
```

Closure can be seen as a “functional object”

Closures

Closure is a function with bound variables

```
def build_taxer(rate):
    def taxer(amount):
        return amount * (float(rate) / 100)
    return taxer

vat1 = build_taxer(22)
vat2 = build_taxer(7)
```

Closure can be seen as a “functional object”

Prime numbers

Definition

Natural number n is *prime* **iff**

$$\neg \exists k \in [2, n) : n \equiv 0 \mod k$$

How to translate this into code?

Imperative primes

```
def is_prime(n):
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

- List of statements to execute one after another
- Not obvious if and when the loop ends
- Local side effects

Imperative primes

```
def is_prime(n):
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

- List of statements to execute one after another
- Not obvious if and when the loop ends
- Local side effects

Map, filter and reduce

High-order functions operating on lists (sequences)

- Apply a function to every element

  ```
  map(lambda x: x**2, range(1,5))
  -> [1, 4, 9, 16]
  ```

Map, filter and reduce

High-order functions operating on lists (sequences)

- Apply a function to every element

```
map(lambda x: x**2, range(1,5))
-> [1, 4, 9, 16]
```

- Select elements matching the predicate

```
filter(lambda x: x%2==0, range(10))
-> [0, 2, 4, 6, 8]
```

Map, filter and reduce

High-order functions operating on lists (sequences)

- Apply a function to every element
  ```
  map(lambda x: x**2, range(1,5))
  -> [1, 4, 9, 16]
  ```
- Select elements matching the predicate
  ```
  filter(lambda x: x%2==0, range(10))
  -> [0, 2, 4, 6, 8]
  ```
- Cumulatively reduce elements to a single value
  ```
  reduce(lambda x,y: x+y, [7, 3, 12])
  -> 22
  ```

Why `map` and `reduce` are so useful?

- Can simplify complex loops
- Can be chained

Why `map` and `reduce` are so useful?

- Can simplify complex loops
- Can be chained
- Many computations can be reduced to those (not only numeric ones)

Why `map` and `reduce` are so useful?

- Can simplify complex loops
- Can be chained
- Many computations can be reduced to those (not only numeric ones)
- Can be easily distributed (see Google's MapReduce)

Primes, second approach

```
def is_prime(n):
    len(filter(lambda k: n%k==0, range(2,n))) == 0

def primes(m):
    filter(is_prime, range(1,m))
```

- Clear intention: “Is the list of non-trivial divisors empty?”
- High-order functions can be composed
- No side effects
- `return` omitted for readability

Primes, second approach

```
def is_prime(n):
    len(filter(lambda k: n%k==0, range(2,n))) == 0

def primes(m):
    filter(is_prime, range(1,m))
```

- Clear intention: “Is the list of non-trivial divisors empty?”
- High-order functions can be composed
- No side effects
- `return` omitted for readability

Primes, second approach

```
def is_prime(n):
    len(filter(lambda k: n%k==0, range(2,n))) == 0

def primes(m):
    filter(is_prime, range(1,m))
```

- Clear intention: "Is the list of non-trivial divisors empty?"
- High-order functions can be composed
- No side effects
- `return` omitted for readability

Primes, second approach

```
def is_prime(n):
    len(filter(lambda k: n%k==0, range(2,n))) == 0

def primes(m):
    filter(is_prime, range(1,m))
```

- Clear intention: “Is the list of non-trivial divisors empty?”
- High-order functions can be composed
- No side effects
- `return` omitted for readability

Primes, second approach

```
def is_prime(n):
    len(filter(lambda k: n%k==0, range(2,n))) == 0

def primes(m):
    filter(is_prime, range(1,m))
```

- Clear intention: “Is the list of non-trivial divisors empty?”
- High-order functions can be composed
- No side effects
- `return` omitted for readability

List comprehensions

But we can do better!

- *List comprehensions* borrowed from Haskell

  `[i**2 for i in range(1,10) if i%2==0]`
  `-> [4, 16, 36, 64]`
- Inspired by mathematical notation (slight difference)
  $\{i^2 | \ i \in \mathbf{N}, i \in [1, 10) : i \equiv 0 \bmod 2\}$
- Can replace `map` and `filter` (even `lambda`)
- Simplifies complex chains (more dimensions)

List comprehensions

But we can do better!

- *List comprehensions* borrowed from Haskell
  `[i**2 for i in range(1,10) if i%2==0]`
  `-> [4, 16, 36, 64]`
- Inspired by mathematical notation (slight difference)
  $\{i^2 |\ i \in \mathbf{N}, i \in [1, 10) : i \equiv 0 \text{ mod } 2\}$
- Can replace `map` and `filter` (even `lambda`)
- Simplifies complex chains (more dimensions)

List comprehensions

But we can do better!

- *List comprehensions* borrowed from Haskell
  `[i**2 for i in range(1,10) if i%2==0]`
  `-> [4, 16, 36, 64]`
- Inspired by mathematical notation (slight difference)
  $\{i^2 |\ i \in \mathbf{N}, i \in [1, 10) : i \equiv 0 \bmod 2\}$
- Can replace `map` and `filter` (even `lambda`)
- Simplifies complex chains (more dimensions)

Primes, third approach

```
def is_prime(n):
    True not in [n%k==0 for k in range(2,n)]

def primes(m):
    [n for n in range(1,m) if is_prime(n)]
```

- Is there any problem with the last two versions?

Primes, third approach

```
def is_prime(n):
    True not in [n%k==0 for k in range(2,n)]

def primes(m):
    [n for n in range(1,m) if is_prime(n)]
```

- Is there any problem with the last two versions?
- Do we have to go through the whole list?

Generators, iterators and streams

It is said that good programmers are lazy...

- *Iterators* are lazy sequences
- *Generator expressions* help building iterators

  ```
  (i**2 for i in xrange(1,10) if i%2==0)
  -> <generator object at 0x12c4850>
  ```
- `Map` and `filter` will be lazy in Python 3000
- Called *streams* in the functional world

Prime numbers, fourth approach

```
def is_prime(n):
    True not in (n%k==0 for k in xrange(2,n))

is_prime(100000000)
-> False
```

- Lazy evaluation

Prime numbers, fourth approach

```
def is_prime(n):
    True not in (n%k==0 for k in xrange(2,n))

is_prime(100000000)
-> False
```

- Lazy evaluation

Quantification

Can do even better with Python 2.5!

- `any(seq)` returns true if at least one element of the sequence is true ($\exists$, *exists*)
- `all(seq)` returns true if all elements of the sequence are true ($\forall$, *for all*)

Short-circuit lazy evaluation, like with logical operators

Primes, grand finale

```
def is_prime(n):
    not any(n%k==0 for k in xrange(2,n))
```

Does this look familiar?

Primes, grand finale

```
def is_prime(n):
    not any(n%k==0 for k in xrange(2,n))
```

Does this look familiar?

Definition

Natural number *n* is *prime* **iff**

$$\neg \exists k \in [2, n) : n \equiv 0 \mod k$$

Primes, grand finale

```
def is_prime(n):
    not any(n%k==0 for k in xrange(2,n))
```

Does this look familiar?

Definition

Natural number n is *prime* **iff**

$$\neg\exists k \in [2, n) : n \equiv 0 \mod k$$

Tadam!

Summary

- Functional programming allows you to describe the problem in a more abstract way
- Learning functional approach widens your perspective on programming
- It's worth applying when it makes sense
- Python has some useful functional features
- Python 3000 is getting more lazy

The wizard book

http://mitpress.mit.edu/sicp/

Thank you

May the λ be with You!

`adambyrtek@gmail.com`

`http://www.adambyrtek.net`

`http://del.icio.us/alpha.pl/functional-programming`