SD314 Outils pour le Big Data

Functional programming in Python

Christophe Garion

DISC – ISAE

License CC BY-NC-SA 3.0

Outline

What is Python?

Python is a widely-used programming language whose objective is **simplicity**: you can write concisely and efficiently programs. It has a good standard library and numerous librairies offer good API.

Some of its features (that can also be drawbacks...):

- multi-paradigm: procedural, object-oriented, functional (+ more via extension)
- extensible
- dynamically typed
- automatic memory management
- mainly intepreted

Python Software Foundation (2015a).
Python.
http://www.python.org.

Python's philosophy: the Zen of Python

Python Software Foundation (2015b).
PEP 20 – The Zen of Python.
https://www.python.org/dev/peps/pep-0020/.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

An simple example

Let us look at one of the most ancien algorithm:

```
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".format(I1, I2, gcd(I1, I2)))
```

Python is dynamically typed

Python is **dynamically typed**:

- types are attached to values, not variables
- but Python is strongly typed: you cannot add an integer to a string for instance

```
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".
      format(I1, I2, gcd(I1, I2)))
```

Python syntax: blocks using tabulations

Blocks in Python are represented using tabulations (Tab key).

As they are **mandatory**, the source code is (should be?) easy to read.

```
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".
      format(I1, I2, gcd(I1, I2)))
```

Python syntax: imperative kernel

The instructions for the imperative kernel have (more or less) the same syntax than C/Java.

Beware of the **:** syntax!

```
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".
     format(I1, I2, gcd(I1, I2)))
```

Python syntax: functions

You can define functions using the **def** keyword. Function call has a classical syntax.

You can define functions inside functions if you want...

```python
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".
      format(I1, I2, gcd(I1, I2)))
```

Python syntax: program

Python is historically a scripting language, you do not need to define a `main` function to define a program.

To execute your program, you have to **interpret** it:

```
python3 filename.py
```

or

```
ipython3 filename.py
```

```python
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".
      format(I1, I2, gcd(I1, I2)))
```

Python syntax: objects

In Python, everything is an object: integers, lists, even functions!

You can call a method using the classical . notation.

For instance:

```
l = [1, 2, 3]
l.append(4)
print(l)
```

```
def gcd(i1, i2):
    a = i1
    b = i2

    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a

    return a

I1 = int(input("First integer? "))
I2 = int(input("Second integer? "))

print("The GCD of {0} and {1} is {2}".
      format(I1, I2, gcd(I1, I2)))
```

Lists, tuples and dictionaries

Lists are mutable sequences:

```
l1 = []
l2 = list('abc')
l3 = [1, 2, 3]
l1.append(l2[0])
```

Lists, tuples and dictionaries

Lists are mutable sequences:

```
l1 = []
l2 = list('abc')
l3 = [1, 2, 3]
l1.append(l2[0])
```

You can extract sublists with slices using the `[start:end:step]` notation:

```
[1, 2, 3, 4][1:3]
[1, 2, 3, 4][-1:0:-1]
[1, 2, 3, 4][-1::-1]
```

Lists, tuples and dictionaries

Lists are mutable sequences:

```
l1 = []
l2 = list('abc')
l3 = [1, 2, 3]
l1.append(l2[0])
```

You can extract sublists with slices using the `[start:end:step]` notation:

```
[1, 2, 3, 4][1:3]
[1, 2, 3, 4][-1:0:-1]
[1, 2, 3, 4][-1::-1]
```

Tuples are immutable sequences (often needed as keys for instance):

```
t = (1, 2, 3)
t[0] = 2
```

Lists, tuples and dictionaries

Lists are mutable sequences:

```
l1 = []
l2 = list('abc')
l3 = [1, 2, 3]
l1.append(l2[0])
```

You can extract sublists with slices using the `[start:end:step]` notation:

```
[1, 2, 3, 4][1:3]
[1, 2, 3, 4][-1:0:-1]
[1, 2, 3, 4][-1::-1]
```

Tuples are immutable sequences (often needed as keys for instance):

```
t = (1, 2, 3)
t[0] = 2
```

Dictionaries are mapping objects:

```
d = dict()
d['a1'] = 4
d = { 'a1': 4, 'a2': 5}
```

The for loop

The **for** loop instruction works with everything that is **iterable**.

From lists...

```
for x in [1, 2, 3, 4]:
    print(x)
```

... ranges...

```
for x in range(2, 8):
    print(x)
```

... to lines of files:

```
data = open("/etc/passwd")

for line in data:
    print(line)
```

Outline

Functional programming: the big picture

Functional programming is a programming **style** or **paradigm** based on evaluation of **mathematical** functions.

As in mathematics, there are **no side-effects** and **no mutable state**.

Functional programming: the big picture

Functional programming is a programming **style** or **paradigm** based on evaluation of **mathematical** functions.

As in mathematics, there are **no side-effects** and **no mutable state**.

The foundations of functional programming are nested in lambda calculus, a mathematical formalism developed in the 30's to answer Hilbert's *Entscheidungsproblem*.

Features of functional programming

Main characteristics of functional programming:

- **higher-order functions**: functions that can take other functions as parameters or return them as results

  Question: can you think of a simple mathematical higher-order function that you know for a long time?

Features of functional programming

Main characteristics of functional programming:

- **higher-order functions**: functions that can take other functions as parameters or return them as results

  Question: can you think of a simple mathematical higher-order function that you know for a long time?

- **pure functions**: functions that does not have side effects.

  An important associated property is **referentiel transparency**: you can call a function with the same arguments multiple times, it will return the same result.

Features of functional programming

Main characteristics of functional programming:

- **higher-order functions**: functions that can take other functions as parameters or return them as results

  Question: can you think of a simple mathematical higher-order function that you know for a long time?

- **pure functions**: functions that does not have side effects.

  An important associated property is **referentiel transparency**: you can call a function with the same arguments multiple times, it will return the same result.

- **recursion**: functions that call themselves for iteration.

  Can be optimized via tail-recursion

Advantages of functional programming

- higher-order functions allow to write more compact code.

  For instance, take a sorting function written in Java (without lambda expressions). How can you pass the comparison method as parameter? Is it easy to implement?

- referential transparency is powerful and allows to:
  - remove some code if unused
  - optimize code via memoization for instance
  - **parallelize** code
  - use whatever evaluation strategy you want

Functional programming languages

There are lots of functional programming languages that have been developped since the beginning of CS: Lisp, OCaml, Haskell, Erlang, Clojure...

They are powerful, but may be a little bit cryptic for "average" programmer. For instance, Fibonacci's function in Haskell:

```haskell
fibonacci = 0:1:zipWith (+) fibonacci (tail fibonacci)
```

Functional programming languages

There are lots of functional programming languages that have been developped since the beginning of CS: Lisp, OCaml, Haskell, Erlang, Clojure. . .

They are powerful, but may be a little bit cryptic for "average" programmer. For instance, Fibonacci's function in Haskell:

```haskell
fibonacci = 0:1:zipWith (+) fibonacci (tail fibonacci)
```

The main drawback that was claimed against FP languages was efficiency, as many FP languages use **immutable data**.
But languages like OCaml or K have excellent execution performances.

Industrial uses of FP

FP was not really popular in industry, but some successful stories exist:

- Erlang has been developed and used at Ericsson for telecommunications
- OCaml is used in finance, compiler implementation and static verification of programs

Nowadays, FP has regained interest with Big Data problematics.

Outline

FP in Python: possible or not?

We will examine FP features and see if it is possible to implement/use them in Python.

feature	in Python
higher-order functions	□
pure functions	□
recursion	□
immutable data	□

FP in Python: possible or not?

We will examine FP features and see if it is possible to implement/use them in Python.

feature	in Python
higher-order functions	☐
pure functions	☐
recursion	☑
immutable data	☐

First, let us remark that recursion is possible in Python, so that's one feature checked ☺

Let us look at the other features.

Higher-order functions

Remember that higher-order functions are functions that accept functions as parameters or return functions as value.

Higher-order functions

Remember that higher-order functions are functions that accept functions as parameters or return functions as value.

As everything is an object in Python, even functions, you can pass a function as argument of another function:

```
def higher_order(value, function):
    return function(value)

def inc_int(i):
    return i + 1

def first_char(s):
    return s[0]

print(higher_order(2, inc_int))
print(higher_order("hello", first_char))
```

Sorting, again and again. . .

✎

Exercise

Write a `selection_sort` function that takes a list to be sorted and a comparison function.
Test it on several examples.

Trying partial function application

Let us consider the following code:

```
def higher_order(value, function):
    return function(value)

def multiply_int(i, j):
    return i * j

print(higher_order(3, multiply_int))        # ouch
```

Trying partial function application

Let us consider the following code:

```
def higher_order(value, function):
    return function(value)

def multiply_int(i, j):
    return i * j

print(higher_order(3, multiply_int))        # ouch
```

In functional programming languages, this should be possible: the return of `higher_order(3, multiply_int)` should be a function that takes **one** argument and multiply this argument by 3.

This is call partial function application.

The partial function

The functools module offers a partial function that allows to create partial functions:

```python
from functools import partial

def higher_order(value, function):
    return partial(function, value)

def multiply_int(i, j):
    return i * j

print(higher_order(3, multiply_int)(4))
```

Lambdas

Using the `inc_int` function in the first example was a little bit fastidious.

We can use instead **lambdas**, which are anonymous functions:

```
def higher_order(value, function):
    return function(value)

print(higher_order(2,
                   lambda x: x + 1))
print(higher_order("hello",
                   lambda x: x[0]))
```

Lambdas

Using the `inc_int` function in the first example was a little bit fastidious.

We can use instead **lambdas**, which are anonymous functions:

```python
def higher_order(value, function):
    return function(value)

print(higher_order(2,
                   lambda x: x + 1))
print(higher_order("hello",
                   lambda x: x[0]))
```

Lambdas are related to the lambda-calculus, the mathematical foundation for functional languages.

Lambdas

Using the `inc_int` function in the first example was a little bit fastidious.

We can use instead **lambdas**, which are anonymous functions:

```python
def higher_order(value, function):
    return function(value)

print(higher_order(2,
                   lambda x: x + 1))
print(higher_order("hello",
                   lambda x: x[0]))
```

Lambdas are related to the lambda-calculus, the mathematical foundation for functional languages.

Beware

Lambdas in Python are limited to an expression.

Lambdification of your sort ✎

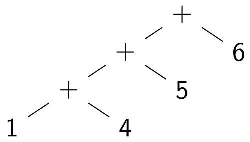
Exercise

Call your sort function with some lambdas.

Map, filter, reduce

There are three "famous" higher-order functions in FP (can be generalized to other structures than lists):

- **map** is a function that applies a function on each element of a list and returns the resulting list
- **filter** is a function that takes a function returning a boolean value (condition) and a list as parameters and returns a list consisting of the element of the initial list satisfying the condition
- **reduce** or **fold** (more specifically `fold_left` here) is a function that takes a combining function, a list and an initial element and computes (reduces) the list to a value using the function.
  For instance, calling `reduce(operator.add, [4, 5, 6], 1)` should produce the following computation:



Exercise

Using only recursion (no loops!), write your own version of those three functions.

Hint: `reduce` can be used in the other functions...

Use `reduce` to verify if all elements of a list verify a given property.

Use all functions to create a function that returns the sum of the squared values of each even element of a list.

The map function

Of course, Python has a built-in **map** function ☺

Beware

What follows on iterators apply on Python 3, not Python 2!

The **map** function of Python3 returns an **iterator**, i.e. an object that implements the **next** function. This allows to yield elements of the result when needed and can be used in a loop or to build a list:

```
obj = map(lambda x: x + 1, [1, 2, 3, 4])

print(type(obj))

for i in obj:
    print(i)

print(list(obj))          # what happen here?
```

Exercise

Create a function that takes a list of integers and returns a list of partial functions such that each partial function adds the corresponding integer to its argument.

List comprehension

Python recommends to use **generators** and **list comprehensions** instead of **map**.

For instance, the following **map** call:

```
l = [1, 2, 3, 4]
print(map(lambda x: x + 1, l))
```

is equivalent to

```
l = [1, 2, 3, 4]
print([(lambda x: x + 1)(i) for i in l])
```

List comprehension

Python recommends to use **generators** and **list comprehensions** instead of **map**.

For instance, the following **map** call:

```
l = [1, 2, 3, 4]
print(map(lambda x: x + 1, l))
```

is equivalent to

```
l = [1, 2, 3, 4]
print([(lambda x: x + 1)(i) for i in l])
```

You can also constraint the comprehension:

```
l = [1, 2, 3, 4]
print([(lambda x: x + 1)(i) for i in l if i % 2 == 0])
```

The filter and reduce functions

Of course (again), Python has a built-in **filter** function ☺

```
l = [1, 2, 3, 4]

for i in filter(lambda x: x % 2 == 0, l):
    print(i)
```

The filter and reduce functions

Of course (again), Python has a built-in **filter** function ☺

```
l = [1, 2, 3, 4]

for i in filter(lambda x: x % 2 == 0, l):
    print(i)
```

And the poor **reduce** function has been exiled in the `functools` module:

```
from functools import reduce

l = [1, 2, 3, 4]
print(reduce(lambda x, y: x + y, l, 0))
```

OK, let's check

feature	in Python
higher-order functions	☑
pure functions	☐
recursion	☑
immutable data	☐

So, you have to verify if we can write pure functions and use immutable data.

Pure functions and immutable data

For pure functions, use only local code. Do not use global variables and assignements.

Pure functions and immutable data

For pure functions, use only local code. Do not use global variables and assignements.

If you respect this rule, everything's gonna be allright.

Pure functions and immutable data

For pure functions, use only local code. Do not use global variables and assignements.

If you respect this rule, everything's gonna be allright.

For immutable data, you may use tuples, but a simple solution would be to return new lists for instance.