Day 04 – Piscine Python for Data Science

Intro to Python: Efficient code practices

Summary: This day will help you write the code that works faster.

Contents

Chapter I

Preamble

There are two words in English that are commonly confused: “efficiency” “and effectiveness”.

To highlight the difference, let us tell you a short joke.

My motto is “Efficiency. Efficiency. Efficiency.” Oops. I guess I only need to say it once.

Or as Peter Drucker once said: “Efficiency is doing things right; effectiveness is doing the right things”.

Your code should not only be effective, but efficient as well. And vice versa.

One of the best games that teach you how to be efficient is Factorio. Google it. Download it. And try to get back to Day 04. Not everyone will be able to do this.

Chapter II

Instructions

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- Here and further we use `Python 3` as the only correct version of Python.
- The python files for python exercises (d01, d02, d03) must have a block in the end: `if __name__ == '__main__'`.
- Pay attention to the permissions of your files and directories.
- To be assessed your solution must be in your GIT repository.
- Your solutions will be evaluated by your piscine mates.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your .gitignore to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google.
- Remember to discuss on the Intra Piscine forum.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- And may the Force be with you!

Chapter III

Specific instructions of the day

- No code in the global scope. Use functions!
- Each file must be ended by a function call in a condition similar to:

```
if __name__ == '__main__':
    # your tests and your error handling
```

- Any exception not caught will invalidate the work, even in the event of an error that was asked you to test.
- No imports allowed, except those explicitly mentioned in the section “Authorized functions” of the title block of each exercise.
- You can use any built-in function, if it is not prohibited in an exercise.
- The interpreter to use is `Python 3`.

Chapter IV

Exercise 00

Exercise 00 : List comprehensions
Directory to store your solution : `ex00/`
Files to be in the directory : `benchmark.py`
Authorized functions : `import timeit`

Imagine that you have the task to get all the emails from the list that are Gmails. The usual approach is to create a loop, and iterating from the initial list append the needed values to a new list.

But that can be less efficient if we talk about big amounts of data. There is a more efficient and pythonic way to do the task – `list comprehensions`.

In this exercise, you need to:

- write two functions:
  - in the first – you need to implement the usual approach with a loop and an append,
  - in the second – you use a list comprehension instead,
- use timeit to measure the time required to run those functions `90 000 000` times and compare them,
- put this into a script that prints "`it is better to use a list comprehension`" if the corresponding time is less or equal than the loop's, and "`it is better to use a loop`" otherwise,
- also, at the end, add the time values after the print described above, the first position must be the lesser one.

Please, use the following list of emails:

```
emails = ['john@gmail.com', 'james@gmail.com', 'alice@yahoo.com',
'anna@live.com', 'philipp@gmail.com']
```

The example:

```
$ ./benchmark.py

it is better to use a list comprehension
55.71611063099999 vs 58.849982983
```

Chapter V

Exercise 01

Exercise 01 : Map
Directory to store your solution : `ex01/`
Files to be in the directory : `benchmark.py`
Authorized functions : `import timeit`

Ok, the chances are that you saw the difference: list comprehensions are slightly more efficient than loops and more readable as well. But that is not the only available option. There is `map()`!

Map comes from functional programming. You do not have to iterate through a list. You can apply a function to an iterable. That is what you are going to do in the exercise!

Modify the script from the previous exercise:

- Write a function that does the same thing: creates a list with gmails taken from the initial list of emails, but using a map. Try `map()` and `list(map())`. Check the difference in speed.
- You still need to compare which function is faster, but now you have three options, add one more "`it is better to use a map`" and at the end, you need to display all three time values with the same condition: they should be in the ascending order.

The example:

```
$ ./benchmark.py

it is better to use a map
29.32016281 vs 54.620376492999995 vs 55.99120069
```

Check the results of all the functions. Are they all identical? They do not have to be.

Chapter VI

Exercise 02

Exercise 02 : Filter
Directory to store your solution : `ex02/`
Files to be in the directory : `benchmark.py`
Authorized functions : `import timeit, import sys`

Did you notice that what you did in the previous exercises was filtering? Why not to use the corresponding function `filter()` instead of those list comprehensions and maps? It works almost the same as `map()`. You will love it!

Add to your benchmark the new function that uses `filter()`. But this time let us refactor the code. Let us create a script that takes as an argument the name of the function (loop, list comprehension, map, filter) and the number of calls it should perform for the benchmark. It should give in return the time spent to make that number of calls of the function.

The examples:

```
$ ./benchmark.py loop 10000000
6.230267604

$ ./benchmark.py list_comprehension 10000000
6.214286791

$ ./benchmark.py map 10000000
3.063598874

$ ./benchmark.py filter 10000000
4.076410670000001
```

Chapter VII

Exercise 03

Exercise 03 : Reduce
Directory to store your solution : `ex03/`
Files to be in the directory : `benchmark.py`
Authorized functions : `import timeit, import sys, from functools import reduce`

Besides `map()` and `filter()` there is another function that might be useful for you in the future – `reduce()`. You can use it again instead of loops and, in most cases, it will be more efficient when you need to calculate a sum. In this exercise, you need to calculate the sum of squares up to the number given as an argument. For example, if 5 was given, the sum will be 1 + 4 + 9 + 16 + 25 = 55.

In your script create two functions:

- in the first – you need to implement the usual approach with a loop and `sum = sum + i*i`,
- in the second – you use a `reduce()` instead,

Let us create a script that takes as an argument the name of the function (loop or reduce), the number of calls it should perform for the benchmark, and the number for the sum of squares calculations. It should give in return the time spent to make that number of calls of the function.

The example:

```
$ ./benchmark.py loop 10000000 5
6.230267604

$ ./benchmark.py reduce 10000000 5
3.063598874
```

Chapter VIII

Exercise 04

Exercise 04 : Counter
Directory to store your solution : `ex04/`
Files to be in the directory : `benchmark.py`
Authorized functions : `import timeit, import random, from collections import Counter`

"Know the built-in functions" should be one of the most important commandments for a Python coder. Here we are going to use the `collections` module that is shipped with Python. It contains a number of container data types. We will use `Counter`. It is very handy when you need to count unique values in a list, for example. And it is faster than any function that you can write by yourself. But we do not want you to trust us in this. Check it yourself!

1. generate a list with `1 000 000` random values from 0 to 100 (remember about list comprehensions?),
2. write a function that creates a dict out of the list where the keys are the numbers from 0 to 100 and the values are their counts,
3. write a function that returns the top 10 most common numbers where the keys are the numbers and the values are the counts,
4. solve 2 and 3 using `Counter`,
5. make a comparison: your script should display the time spent for 2 and 3 with Counter and without it.

   The example:

```
$ ./benchmark.py
my function: 0.4501532
Counter: 0.0432341

my top: 0.1032348
Counter's top: 0.017573
```

Chapter IX

Exercise 05

Exercise 05 : Generator
Directory to store your solution : `ex05/`
Files to be in the directory : `ordinary.py`, `generator.py`
Authorized functions : `import sys, import resource` for Windows: `import sys, import os, import psutil`

Code efficiency is not only about the time spent, but also about the RAM used. It is quite important if you work with big data. Or maybe the data can be smaller to cause you trouble? You have already got used to making experiments. Let us do yet another one.

1. Download the MovieLens dataset.
2. Unzip it. You will need the file `ratings.csv` (678.3 MB is not that big, huh?).
3. Create the first script `ordinary.py`. It should have the only function: it reads all the file lines into a list and then returns it. In the main program, write a loop that iterates through the list and calls `pass`. You should give the path to the file as an argument to the script.
4. Create the second script `generator.py`. It does exactly the same thing, but in your function, you must use a generator. It uses the `yield` keyword to read one line at a time and returns it to the caller. In the main program, write a loop that iterates through the generator and calls `pass`. You should give the path to the file as an argument to the script.
5. Both scripts should display Peak memory usage (`ru_maxrss`) in GB and User mode time (`ru_utime`) + System mode time (`ru_stime`) in seconds. If you have Windows OS, use the corresponding functions to get the same metrics.

   The example:

```
$ ./ordinary.py ratings.csv
Peak Memory Usage = 2.114 GB
User Mode Time + System Mode Time = 5.77s

$ ./generator.py ratings.csv
Peak Memory Usage = 0.005 GB
User Mode Time + System Mode Time = 9.04s
```