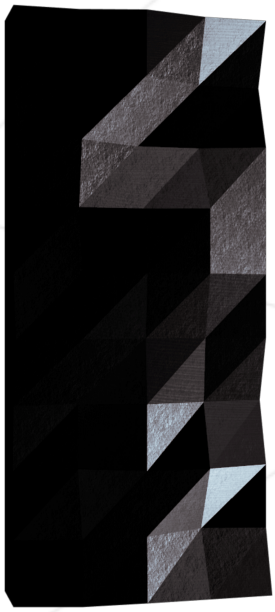

MODULE 04 – Piscine Python for Data Science

Intro to Python: Efficient code practices

Summary: Today we will help you write code that works faster.

Contents

I	Foreword	2
II	Instructions	3
III	Specific instructions of the day	4
IV	Exercise 00 : List comprehensions	5
V	Exercise 01 : Map	7
VI	Exercise 02 : Filter	8
VII	Exercise 03 : Reduce	9
VIII	Exercise 04 : Counter	10
IX	Exercise 05 : Generator	11

Chapter I

Foreword

- 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 for learning how to be efficient is Factorio. Google it. Download it. And try to get back to MODULE 04. Not everyone will be able to.

Chapter II

Instructions

- Use this page as your only reference. Do not listen to any rumors or speculations about how to prepare your solution.
- Here and further on we use Python 3 as the only correct version of Python.
- The python files for python exercises (module01, module02, module03) must have the following 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 any 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. If that fails, try your neighbor on the left.
- Your reference material: peers / Internet / Google.
- You can ask questions in Slack.
- 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 end with 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 you were asked to test.
- No imports are 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 the exercise.

Chapter IV

Exercise 00 : List comprehensions

	Exercise 00
List comprehensions	
Turn-in directory : $ex00/$	
Files to turn in : `benchmark.py`	
Allowed functions : `import timeit`	

Imagine that your task is to get all the Gmail addresses from a list of email addresses from the list that are Gmails. The usual approach is to create a loop, and iterating from the initial list append the required values to a new list.

But that can be inefficient if we are talking about large 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 that of the loop, and “it is better to use a loop” if not,
- also, add the time values at the end, after the print described above. Order them from shortest to longest.

Please, use the following list of email addresses: emails = [’john@gmail.com’, ’james@gmail.com’, ’alice@yahoo.com’, ’anna@live.com’, ’philipp@gmail.com’]

Duplicate the values 5 times. As a result, the list will contain 25 elements, but only 5 unique ones.

An example of the script being launched:

```
$ ./benchmark.py
it is better to use a list comprehension
55.71611063099999 vs 58.849982983
```

Chapter V

Exercise 01 : Map

	Exercise 01
Map	
Turn-in directory : $ex01/$	
Files to turn in : `benchmark.py`	
Allowed functions : `import timeit`	

Ok, 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 option available. There is also 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 this exercise!

Modify the script from the previous exercise:

- Write a function that does the same thing: creates a list with Gmail addresses taken from the initial list of emails (25 elements), but using a map. Try map() and list(map()). Note the difference in speed
- You still need to compare which function is faster, but now you have three options: loop, list comprehension, and map, and add one more phrase according to this in your code "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 by length.

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 : Filter

	Exercise 02
Filter	
Turn-in directory : $ex02/$	
Files to turn in : `benchmark.py`	
Allowed functions : `import timeit, import sys`	

- Did you notice that what you did in the previous exercises was filtering? Why not use the corresponding function filter() instead of those list comprehensions and maps? It works almost the same as map(). You will love it!
- Add a new function to your benchmark that uses filter(). But this time let us refactor the code. Let us create a script that takes the name of the function (loop, list comprehension, map, filter) to your benchmark and the number of calls it should perform for the benchmark. In return, it should give 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
```

Chapter VII

Exercise 03 : Reduce

	Exercise 03
Reduce	
Turn-in directory : $ex03/$	
Files to turn in : `benchmark.py`	
Allowed 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 also use it 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 the calculation of squares. In return, it should give 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 : Counter

	Exercise 04
Counter	
Turn-in directory : $ex04/$	
Files to turn in : `benchmark.py`	
Allowed functions : `import timeit, import random, from collections import Counter`	

"Know the built-in functions" is one of the most vital 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, for example, when you need to count unique values in a list. And it is faster than any function that you can write by yourself. But don't take our word for it, check it out for yourself!

- generate a list with 1 000 000 random values from 0 to 100 (remember list comprehensions?)
- 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
- write a function that returns the top 10 most common numbers where the keys are the numbers and the values are the counts, the input is the list
- solve 2 and 3 using Counter
- make a comparison: your script should display the time spent for 2 and 3 with Counter and without it

  Example:

```
$ ./benchmark.py
my function: 0.4501532
Counter: 0.0432341
my top: 0.1032348
Counter's top: 0.017573
```

Chapter IX

Exercise 05 : Generator

	Exercise 05
Generator	
Turn-in directory : $ex05/$	
Files to turn in : `ordinary.py, generator.py`	
Allowed 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. This is quite important if you work with big data. Or maybe smaller-scale data can also cause you trouble? You have already got used to making experiments. Let us do yet another one.

- Download the MovieLens dataset.
- Unzip it. You will need the file ratings.csv (678.3 MB is not that big, right?).
- Create the first script, ordinary.py. It should have only one 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
- Create the second script, generator.py. It does exactly the same thing, but in your function, you must use a generator. It uses the keyword yield 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.
- Both scripts should display Peak memory usage in GB and User mode time + System mode time in seconds. If you have Windows OS, use the corresponding functions to get the same metrics.

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
```