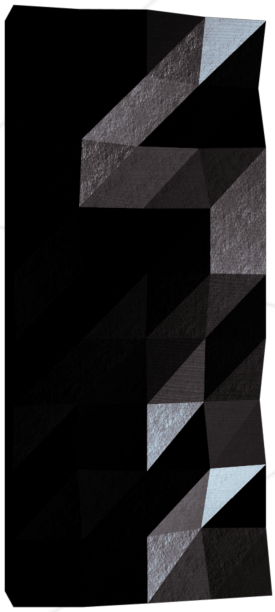

D01 - Python-Django training

Bases Python 1

Summary: Today, we'll embark on a journey to discover basics of the syntactics and semantics of `Python`.

Contents

Chapter I

Preamble

The Zen of Python, by Tim Peters

> 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!

```
import this
```

Chapter II

Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter III

Today's specific rules

- No code in the global scope. We want functions!
- Each turned-in file must end with a function call in a condition identical to:

```
if __name__ == '__main__':
    your_function( whatever, parameter, is, required )
```

- You can set an error management in this condition.
- No import will be authorized, except the ones explicitly mentioned in the 'Autorized functions' in each exercise's description.
- You won't have to manage the exceptions raised by the open function.
- You'll have to use the `python3` interpreter.

Chapter IV

Exercise 00 : my first variables

	Exercise 00
Exercise 00 : my first variables	
Turn-in directory : $ex00/$	
Files to turn in : `var.py`	
Allowed functions : `n/a`	

Create a script named `var.py` in which you will define a `my_var` function. In this function, you will declare 9 variables of different types and print them on the standard output. You will reproduce this output exactly:

```
$> python3 var.py
42 has a type <class 'int'>
42 has a type <class 'str'>
quarante-deux has a type <class 'str'>
42.0 has a type <class 'float'>
True has a type <class 'bool'>
[42] has a type <class 'list'>
{42: 42} has a type <class 'dict'>
(42,) has a type <class 'tuple'>
set() has a type <class 'set'>
$>
```

Of course, explicitly writing your variable types in the prints of you code is strictly **prohibited**. Don't forget to call your function at the end of your script as required by the instructions:

```
if __name__ == '__main__':
    my_var()
```

Chapter V

Exercise 01 : Numbers

	Exercise 01
Exercise 01 : Numbers	
Turn-in directory : $ex01/$	
Files to turn in : `numbers.py`	
Allowed functions : `n/a`	

For this exercise, you're free to define as many function as you like and name them as you like also.

The `d01.tar.gz` tarball in the appendix of this subject contains a `ex01/` sub-folder that holds a `numbers.txt` file containing the numbers 1 to 100 separated by a coma.

Design a `Python` script named `numbers.py` which role is to open a `numbers.txt` file, read the numbers it contains and display them on the standard output, one per line, without any coma.

Chapter VI

Exercise 02: My first dictionary

	Exercise 02
Exercise 02: My first dictionary	
Turn-in directory : $ex02/$	
Files to turn in : `var_to_dict.py`	
Allowed functions : `N.A.`	

Once again, you're free to define as many functions as you like and name them as you like. We won't repeat this instruction, except if it has to be explicitly contradicted.

Create a script named `var_to_dict.py` in which you will copy the following list of `d` couples as is in one of your functions:

```
d = [
  ('Hendrix'  , '1942'),
  ('Allman'   , '1946'),
  ('King'     , '1925'),
  ('Clapton'  , '1945'),
  ('Johnson'  , '1911'),
  ('Berry'    , '1926'),
  ('Vaughan'  , '1954'),
  ('Cooder'   , '1947'),
  ('Page'     , '1944'),
  ('Richards' , '1943'),
  ('Hammett'  , '1962'),
  ('Cobain'   , '1967'),
  ('Garcia'   , '1942'),
  ('Beck'     , '1944'),
  ('Santana'  , '1947'),
  ('Ramone'   , '1948'),
  ('White'    , '1975'),
  ('Frusciante', '1970'),
  ('Thompson' , '1949'),
  ('Burton'   , '1939')
]
```

Your script must turn this variable into a dictionary. The year will be the key, the name of the musician the value. It must then display this dictionary on the standard output following a clear format:

```
1970 : Frusciante
1954 : Vaughan
1948 : Ramone
1944 : Page Beck
1911 : Johnson
...
```

The final order will not have to be the same as the example. This is a regular behavior. Do you know why?

Chapter VII

Exercise 03: Key search

	Exercise 03
Exercise 03: Key search	
Turn-in directory : *ex*03/	
Files to turn in : `capital_city.py`	
Allowed functions : `import sys`	

Here are dictionaries you have to copy unaltered in one of your script's functions:

```
states = {
  "Oregon"   : "OR",
  "Alabama"  : "AL",
  "New Jersey": "NJ",
  "Colorado" : "CO"
}

capital_cities = {
  "OR": "Salem",
  "AL": "Montgomery",
  "NJ": "Trenton",
  "CO": "Denver"
}
```

Write a program that takes a state as an argument (ex: Oregon) and displays its capital city (ex: Salem) on the standard output. If the argument doesn't give any result, your script must display: `Unknown state`. If there is no argument - or too many - your script must no do anything and quit.

```
$> python3 capital_city.py Oregon
Salem
$> python3 capital_city.py Ile-De-France
Unknown state
$> python3 capital_city.py
$> python3 capital_city.py Oregon Alabama
$> python3 capital_city.py Oregon Alabama Ile-De-France
$>
```

Chapter VIII

Exercise 04: Search by value

	Exercise 04
Exercise 04: Search by value	
Turn-in directory : $ex04/$	
Files to turn in : `state.py`	
Allowed functions : `import sys`	

You get the same dictionaries as in the previous exercise. You have to copy them unaltered again in one of the functions of your script.

Create a program that takes the capital city for argument and displays the matching state this time. The rest of your program's behaviors must remain the same as in the previous exercise.

```
$> python3 state.py Salem
Oregon
$> python3 state.py Paris
Unknown capital city
$> python3 state.py
$>
```

Chapter IX

Exercise 05: Search by key or value

	Exercise 05
Exercise 05: Search by key or value	
Turn-in directory : $ex05/$	
Files to turn in : `all_in.py`	
Allowed functions : `import sys`	

Starting with the same dictionaries, you must copy then unaltered again in one of your script functions, and write a program that behaves as follows:

- The program must take for argument a string containing as many expressions as we search for, separated by a coma.
- For each expression in this string, the program must detect if it's a capital, a state or none of them.
- The program must not be case-sensitive. It must not take multiple spaces in consideration either.
- If there is no parameter or too many parameters, the program doesn't display anything.
- When there are two successive comas in the string, the program doesn't display anything.
- The program must display results separated by a carriage return and strictly use the following format:

```
$> python3 all_in.py "New jersey, Tren ton, NewJersey, Trenton, toto,   ,     sAlem"
Trenton is the capital of New Jersey
Tren ton is neither a capital city nor a state
NewJersey is neither a capital city nor a state
Trenton is the capital of New Jersey
toto is neither a capital city nor a state
Salem is the capital of Oregon
$>
```

Chapter X

Exercise 06: Dictionary sorting

	Exercise 06
Exercise 06: Dictionary sorting	
Turn-in directory : $ex06/$	
Files to turn in : `my_sort.py`	
Allowed functions : `N.A.`	

Integrate this dictionary in either function of yours as:

```
d = {
    'Hendrix'  :  '1942',
    'Allman'   :  '1946',
    'King'     :  '1925',
    'Clapton'  :  '1945',
    'Johnson'  :  '1911',
    'Berry'    :  '1926',
    'Vaughan'  :  '1954',
    'Cooder'   :  '1947',
    'Page'     :  '1944',
    'Richards' :  '1943',
    'Hammett'  :  '1962',
    'Cobain'   :  '1967',
    'Garcia'   :  '1942',
    'Beck'     :  '1944',
    'Santana'  :  '1947',
    'Ramone'   :  '1948',
    'White'    :  '1975',
    'Frusciante': '1970',
    'Thompson' :  '1949',
    'Burton'   :  '1939',
}
```

Write a program that displays the names of the musicians sorted by year in ascending order, then in alphabetic order for similar years. One per line, without showing the year.

Chapter XI

Exercise 07: Periodic table of the elements

	Exercise 07
Exercise 07: Periodic table of the elements	
Turn-in directory : *ex*07/	
Files to turn in : `periodic_table.py`	
Allowed functions : `import sys`	

The `d01.tar.gz` tarball in the appendix of this subject contains the `ex07/` sub-folder in which you'll find the `periodic_table.txt` file, that describes a periodic table of the elements in a format made for programmers.

Create a program that uses the file to write a `HTML` page representing the periodic table of the elements in a proper format.

- Each element must be in a 'box' of a `HTML` table.
- The name of an element must be in a level 4 title tag.
- The attributes of an element must appear as a list. The lists must state at least the atomic numbers, the symbol and the atomic mass.
- You must at least abide to the layout of the Mendeleiev's Table as it appears on Google. There must be empty boxes where there should, as well as carriage return where it should.

Your program must create the result file `periodic_table.html`. Of course, this `HTML` file must be readable in any browser and must be `W3C` valid.

You're free to design you program as you like. Don't hesitate to fragment your code in specific functionalities you may reuse. You can customize the tags with a CSS "inline" style to make your turn-in prettier (think of the table's borders' colors). You can even

generate `periodic_table.css` file if you prefer.

Here is an excerpt of an output example that will give you an idea:

```
[...]
 <table>
   <tr>
     <td style="border: 1px solid black; padding:10px">
       <h4>Hydrogen</h4>
           <ul>
              <li>No 42</li>
              <li>H</li>
              <li>1.00794</li>
              <li>1 electron</li>
           <ul>
     </td>
     [...]
```