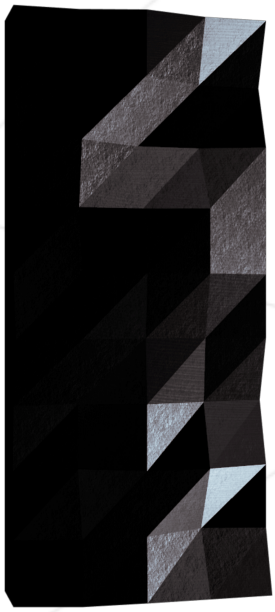

D03 - Python-Django training

Python - librairies

Summary: Today, we're gonna learn how to handle some librairies that might come handy in `Python`.

Contents

Chapter I

Preamble

Geohashing From Wikipedia, the free encyclopedia

Geohashing is an outdoor recreational activity inspired by the xkcd webcomic, in which participants have to reach a random location (chosen by a computer algorithm), prove their achievement by taking a picture of a Global Positioning System (GPS) receiver or another mobile device and then tell the story of their trip online. Proof based on non-electronic navigation is also acceptable.

Whereas other outdoor recreational activities like geocaching have a precise goal, geohashing is mainly fueled by its pointlessness, which is deemed funny by its players. The resulting geohashing community and culture is thus extremely tongue-in-cheek, supporting any kind of humorous behavior during the practice of geohashing and resulting in a parody of traditional outdoor activities. Navigating to a random point need not be pointless. Some geohashers document new mapping features they find on the OpenStreetMap project.

Source Wikipedia

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. Make 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 up an error management in this condition.
- Imports are prohibited, except for the ones specified in the "Authorized Functions" section in each exercise's header.
- You will use the `python3` interpreter.

Chapter IV

Exercise 00

	Exercise 00
Exercice 00: Antigravity	
Turn-in directory : $ex00/$	
Files to turn in : `geohashing.py`	
Allowed functions : `module sys et antigravity`	

This is a little warm up exercise echoing today's preamble. Nothing too complicated.

Create a little program named `geohashing.py` that will take as many parameters as necessary to calculate a typical geohash and should obviously calculate this geohash before displaying it on the standard output.

In case of an error, the program must show a relevant message that you'll have chosen before quitting properly.

This schematics might help: Geohashing algorithm.

Chapter V

Exercice 01

	Exercise 01
Exercice 01: Pip	
Turn-in directory : $ex01/$	
Files to turn in : `my_script.sh my_program.py`	
Allowed functions : `module path.py`	

`path.py` is a library that implements a Path object around the Python's `os.path` module, making its use very intuitive.

In this exercise, you will create a `bash` script that installs this library, as well as a `Python` program that uses it.

The Shell Script must fit this description:

- Its name must have a `.sh` extension because it's a `Shell` script.
- It must display which pip version it uses.
- It must install the `path.py` development version from its `GitHub` repo, in a folder that must be named `local_lib`, located in the repo folder. If the library has already been installed in the folder, the install must crush it.
- It must write `path.py` install logs in a file with a `.log` extension.
- If the library has been properly installed, it must execute the small program you have created.

The `Python` program you must create is a composition of your choice that must however observe these constrainsts:

- Its extension must be `.py` because it's a `Python`.
- It must import the `path.py` module from the location where this library has been installed thanks to the previous script.
- It must create a folder and a file inside this folder, write something in this file and then read and display its content.
- It must respect the specific rules of the day.

Chapter VI

Exercise 02

	Exercise 02
Exercice 02: request an API	
Turn-in directory : $ex02/$	
Files to turn in : `request_wikipedia.py requirement.txt`	
Allowed functions : `modules requests, json, dewiki et sys`	

Wikipedia is an amazing shared tool you necessarily known. It's available in your favorite browser and as a mobile application. You're invited to create a tool that will allow your to request this now essential website, straight from your terminal.

To do so, you must design a program named `request_wikipedia.py` which takes a `string` in parameter and makes a search via the Wikipedia's API before writing the result in the file. You can request the French or English API.

- The program must write a result, even if the request is misspelled. Take the original website for an example: if it finds a result for a given request, you program also should.
- The result must not be formatted in `JSON` or `Wiki Markup` before being written in the file.
- The name of the file must be formatted like this: `name_of_the_search.wiki` and must not contain any space.
- In case of parameter absence, wrong parameter, invalid request, information not found, server problem or any other problem: no file should be created and a relevant error message must be displayed on the console.
- Include `requirement.txt` file in your repo. It will be used during your evaluation to install the libraries necessary to your `VirtualEnv` program or on the system.

The dewiki library is not perfect. We're not looking for the best result, this is not the goal of this exercise.

Carefully read the API documentation. Watch the structure that is sent back to you.

Here is an example of what's expected:

```
$>python3 request_wikipedia.py "chocolatine"
$>cat chocolatine.wiki




   Une chocolatine designe :
   * une viennoiserie au chocolat, aussi appelee pain au chocolat ou couque au chocolat ;
   * une viennoiserie a la creme patissiere et au chocolat, aussi appelee suisse ;
   * une sorte de bonbon au chocolat ;
   * un ouvrage d'Anna Rozen


   Malgre son usage ancien, le mot n'est entre dans le dictionnaire Petit Robert qu'en 2007 et dans le
        Petit Larousse qu'en 2011.

   L'utilisation du terme "Chocolatine" se retrouve egalement au Quebec, dont la langue a evolue a partir
         du vieux francais differemment du francais employe en Europe, mais cet usage ne prouve ni n'
        infirme aucune anteriorite, dependant du hasard de l'usage du premier commercant l'ayant introduit
         au Quebec.

    References




   Categorie:Patisserie
   Categorie:Chocolat
```

Chapter VII

exercise 03

	Exercise 03
Exercise 03: HTML parser	
Turn-in directory : $ex03/$	
Files to turn in : `roads_to_philosophy.py, requirement.txt`	
Allowed functions : `modules sys, requests et BeautifulSoup`	

Legend has it that if you start from any Wikipedia article, that you click on the first link in this article introduction that neither in italic nor between brackets, and repeat the process ad lib, you will always end up on the article about Philosopy.

Well, this is not a legend! (please, look astounded). As can testify this ... Wikipedia article.

But since you only believe what you see with your own eyes, you **must** create a program that tests this phenomenon listing and counting all the articles between your request and the wikipedia article: the **roads to philosophy**.

This program must be named `roads_to_philosophy.py` and behave as follows:

- The program must take a string in parameter that is a word or a group of words matching only one Wikipedia search.
- The program must request an **English** Wikipedia URL identical to a standard search on a browser. In other words: you **cannot** use the site's API.

- It must parse the `html` page thanks to the `BeautifulSoup` library to:
  - Find the eventual redirection and take it into account in the **roads to philosophy**. Beware, it's not a URL redirection.
  - Find the main title of the page and add it to the **roads to philosophy**.
  - Find (if it exists) **the first link of the introduction's paragraph** leading to another Wikipedia article. Rather than ignoring what's in italic or between brackets, the program must carefully **ignore** the links that don't direct to a new article, like the ones leading to the help section of `Wikipédia`.
- The program must start again from **step 2** starting from the link obtained during the previous step until getting to one of those occurrences:
  - The link leads to the `philosophy` page. If so, it must print the visited articles as well as the total count of these articles in the following format `<number> roads from <request> to philosophy` on the standard output.
  - The page didn't include any valid link. The program must display: `It leads to a dead end !`.
  - The link leads to a page that's already been visited, which means the program is about to loop indefinitely. If so, the displayed message must be: `It leads to an infinite loop !`
- At this stage, after displaying the necessary messages on the standard output, the program must quit properly.

`If, anytime during the execution of the program, an error like a a connection, server, parameter, request error or any other kind of error occurs, the program must quit properly with a relevant error message.`

Like the previous exercise, you must provide a `requirement.txt` file with your program to facilitate the libraries install.

Your program's output must look like this:

```
$> python3 roads_to_philosophy.py "42 (number)"
42 (number)
Natural number
Mathematics
Ancient Greek
Greek language
Modern Greek
Colloquialism
Word
Linguistics
Science
Knowledge
Awareness
Conscious
Consciousness
Quality (philosophy)
Philosophy
17 roads from 42 (number) to philosophy !
$> python3 roads_to_philosophy.py Accuvio
It's a dead end !
$>
```

The Wikipedia community often updates the articles. It is highly probable that between the creation of this subject and the day you take it, the roads to philosophy have changed and the accuvio example is not a dead end anymore.

Chapter VIII

Exercise 04

	Exercise 04
Exercice 04: Virtualenv	
Turn-in directory : $ex04/$	
Files to turn in : `requirement.txt my_script.sh`	
Allowed functions : `everything`	

Tomorrow, you'll start your `Django` framework training. You must lay the foundation configuring an easy little installation.

TO do so, you will create two elements:

- A `requirement.txt` file that must include the latest stable versions of `django` and `psycopg2`.
- A script with the following behavior:
  - Have the `.sh` extension.
  - Create a `virtualenv` on `python3` named `django_venv`.
  - Install the `requirement.txt` file that you've created in the `VirtualEnv`.
  - The `virtualenv` must be **activated** when quitting.

Chapter IX

Exercise 05

	Exercise 05
Exercise 05: Hello World	
Turn-in directory : $ex05/$	
Files to turn in : `any necessary file`	
Allowed functions : `everything`	

It must be frustrating to simply install `Django`. We feel you.

This is why you will end this day about libraries with a bang, designing your first `Hello World` with `Django`.

In this final exercise, you must follow and adapt the official tutorial to make a web page that simply displays the text `Hello World !` in the browser at the following address: `http://localhost:8000/helloworld`.

Your repo must be a folder containing the `Django` project.