Best practices Python

TUXAE - 02/02/2022

Vue d'ensemble

Environnement virtuel

Requirements / gestion des dépendances

Structure d’un projet

PEP8 Style Guide pour Python

- naming (module name, function/method name, variable name, class name, constant name)
- white space

Documentation

Liste de conseils généraux

POO: Introduction to classes

Environnement virtuel

Environnement virtuel

Pourquoi ?

- Python, par défaut, installe les librairies en global (partagées entre tous les projets) et ne peut avoir qu'une seule version de chaque librairie possible
- Problème: si on a des projets qui ont besoin d'une librairie donnée mais pas de la même version de cette lib
- Solution: le virtual env
- Kézako ? Dossier spécifique à un projet contenant une copie de Python et les libraires dont vous avez besoin pour votre projet
- Idée: chaque projet est totalement indépendant

Environnement virtuel

Comment ça marche ?

- Depuis Python3.3, on peut créer des environnements virtuels directement avec Python: https://docs.python.org/fr/3/library/venv.html

```
python3 -m venv /path/to/new/virtualenv
```

- En général “/path/to/new/virtualenv” = “venv” ou “env”
- Ensuite il faut “activer” son environnement virtuel (cela va vous permettre d’y entrer et donc utiliser les lib que vous allez y installer, c’est un step **obligatoire, à faire à chaque fois que vous voulez travailler sur le projet**)

```
source env/bin/activate # si votre virtualenv s’appelle env
```

- Maintenant vous pouvez installer des lib:

```
pip install ...
```

Requirements / Gestion des dépendances

Requirements / Gestion des dépendances

Pourquoi ?

- Lib qui se mettent à jour avec des breaking changes => impact sur ton code
- Si travail à plusieurs: si vous avez des versions différentes des lib, chacun peut faire face à des comportements différents
- Solution: faire un fichier `requirements.txt` en figeant les versions utilisées

```
package-one==1.9.4
package-two==3.7.1
package-three==1.0.1
...
```

```
package-one~=1.9.4
package-two~=3.7.1
package-three~=1.0.1
...
```

```
package-one<=1.9.4
package-two<=3.7.1
package-three<=1.0.1
...
```

Requirements / Gestion des dépendances

Utilisation du requirements.txt

- Fichier très pratique pour installer plusieurs librairies en même temps
- Gestion automatique des conflits par pip (qui n'est pas faite quand on pip install à la main chaque lib)

```
pip install -r requirements.txt
```

Structure d’un projet

Structure d’un projet

Exemple d’une structure simple

```
mon_projet/ # Dossier racine, qui contient tout votre projet
│
├── mon_projet/ # fichier contenant votre code, généralement porte le même nom que le dossier racine
│   ├── __init__.py # Nécessaire pour faire un package Python
│   ├── script.py   # Fichier contenant des fonctions
│   └── utils.py    # Fichier contenant des fonctions
│
├── tests/
│   ├── __init__.py    # Nécessaire pour faire un package Python
│   ├── test_script.py # Tests unitaires pour les fonctions de script.py
│   └── test_utils.py  # Tests unitaires pour les fonctions de utils.py
│
├── .gitignore         # Fichier contenant les règles d’exclusion de fichiers/dossiers contenus dans votre projet
└── requirements.txt   # Le fameux fichiers de requirements
```

Plus d’info: https://realpython.com/python-application-layouts/

PEP8: convention de style

PEP8

Kézako ?

- Guide pour coder en Python selon des conventions usuelles, très adoptées: https://www.python.org/dev/peps/pep-0008/
- Code layout: indentation, whitespace, imports, …
- Naming conventions:
  - module name: short, snake_case `preprocessing.py`
  - function/method name: snake_case `get_data()`
  - variable names: snake_case (no verb) `number_iterations`
  - class name: UpperCamelCase `FeatureExtractor`
  - constant name: caps `THRESHOLD`
- Comments
- ...

Documentation

Documentation

Pourquoi?

- Pour que des personnes extérieures à votre projet puissent l'utiliser, comprendre ce qu'il s'y passe, etc
- Pour soi, si on arrête de travailler sur le projet pendant longtemps puis on y revient

Documentation

Qu’est-ce qu’on documente ?

- La structure générale du projet
- A quoi sert le projet ? comment l’utiliser ? comment développer dessus ?

Plusieurs façons de le faire

- README.md : le projet au global
- Chaque fichier: les classes, les méthodes, les fonctions (docstring)
- Pour les parties très complexes du code: possibilité de mettre un commentaire au dessus du bloc de code concerné

Documentation

Important

- Il ne faut pas tout documenter non plus: si ce que vous faites est trivial, il n'y a pas besoin de commentaires
- Un **code clair est la meilleure documentation**
- Le README.md est crucial, il permet de comprendre à quoi sert votre projet, comment l'installer et comment se l'approprier

Liste de conseils généraux

Liste de conseils généraux

Vrac:

- Privilégier le code simple au code alambiqué
- Découper le code en petits morceaux (une fonction = une task, une classe = une task, ...) -> permet de se repérer plus facilement et de débugger plus facilement, permet plus de modularité également
- Ne pas se répéter
- Ne pas réinventer la roue
- ...

POO: introduction aux classes

POO: introduction aux classes

Pourquoi ?

- Si un seul script de 10000 lignes: on s'y perd et on se répète beaucoup
- Solution: faire des fonctions -> permet de rendre des parties réutilisables et plus claires
- Si que des fonctions: on va passer des args entre toutes nos fonctions, donc stocker tout l'état du code en permanence -> augmente les risques d'erreurs et complexifie la gestion des arguments
- Solution: utiliser des objets qui vont partager de la mémoire entre des méthodes -> les classes

POO: introduction aux classes

Une classe c'est:

- Un ensemble de fonctions que l'on appelle alors méthodes qui partagent une mémoire i.e. des arguments communs (que l'on appelle des attributs)
- On remarque que dans le "main" on n'a plus aucun paramètre qui apparaît et qui doit être échangé, tout est stocké dans la classe

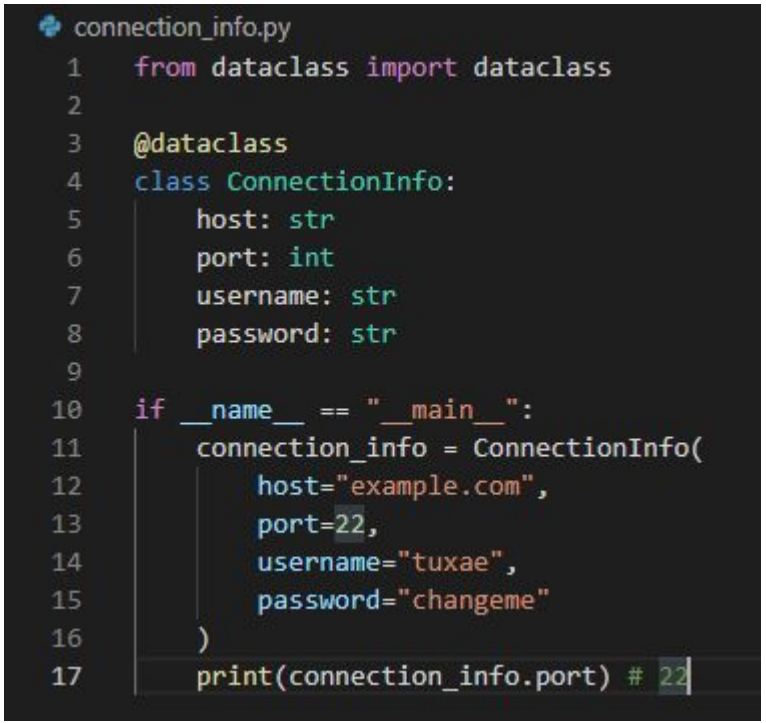
robot.py

```python
class Robot:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def go_up(self):
        self.y += 1

    def go_down(self):
        self.y -= 1

    def go_right(self):
        self.x += 1

    def go_left(self):
        self.x -= 1

    @property
    def position(self):
        return self.x, self.y


if __name__ == "__main__":
    robot = Robot(x=0, y=0)
    robot.go_right()
    robot.go_right()
    robot.go_up()
    print(robot.position) # (2, 1)
```

POO: introduction aux classes

Les dataclasses:

- Une classe qui ne sert qu'à contenir des données (sorte de dictionnaire)
- Elle n'a pas de méthodes en général
- Python va générer automatiquement des méthodes spéciales (__init__, __eq__, __repr__ et autres) sur la dataclasse créée
- L'avantage c'est que ça nous évite d'avoir à écrire du code "bateau", on a de l'autocomplétion et du typage dans notre IDE (ce qu'on n'a pas avec un dico)



La suite au prochain épisode ;)