Python - TP

Clément LACAÏLE - DI5 SI2 - 21200141t

1. TP1 : Prise en main

Ce TP a pour but de se familiariser avec le langage Python. Puisque c'est un langage avec lequel j'ai déjà de bonnes bases (je l'ai étudié lors d'un semestre en mobilité internationale), j'ai décidé d'aller un peu plus loin et d'ajouter les deux particularités : l'utilisation de la convention `PEP8` et l'outil de compilation de documentation `Sphinx`. De fait, outre l'explication des fonctions liées aux questions du TP, j'entrerai plus en détail sur ces deux aspects supplémentaires.

1.1. La convention `PEP8`

L'acronyme `PEP` signifie *Python Enhancement Proposal*. Il est utilisé pour décrire des documents écrits par et pour la communauté des développeurs Python, à la manière des RFC (*Requests For Comments*, documents décrivant les aspects techniques et fonctionnels d'internet). Il en existe plusieurs, mais celle qui est utilisée dans ce TP est la convention `PEP8`. C'est un document écrit en 2001 par *Guido Van Rossum*, *Barry Warsaw* et *Nick Coghlan* dans le but de donner aux développeurs un guide afin d'améliorer la lisibilité et l'harmonie des scripts Python et est aujourd'hui tellement répandu qu'il est presque devenu une norme.

1.1.1. Quelques exemples

Les fonctions et les variables doivent être écrites en minuscules, et si elles sont composées de plusieurs mots, il faut les séparer par un caractère *underscore* `'_'` :

```
def ma_fonction(ma_variable):
    print(ma_variable)
```

Les classes doivent respecter le `CamelCase`, elles doivent débuter par une majuscule et les mots qui la décrivent ne doivent pas être séparés par un caractère *underscore* :

```
class MaClasse:
    pass
```

Il faut minimiser autant que possible l'utilisation de variables avec un seul caractère. De la même manière, les abbréviations sont à éviter. Il est recommandé de donner des noms clairs et concis.

```
# à ne pas faire:
def db(x):
    return x*2
# à faire:
def multiplie_par_deux(nombre):
    return nombre*2
```

L'indentation est composée de caractères d'échappement et non de tabulations ! De plus, la longueur maximum d'une ligne doit être de 79 caractères.

Il existe de nombreuses autres règles à connaitre. Mais il est inutile de les apprendre par coeur ! Il existe un outil bien pratique pour pouvoir vérifier automatiquement le respect de la convention `PEP8`

1.1.2. L'outil `pycodestyle`

Précédemment appelé `pep8`, `pycodestyle` est un outil en ligne de commande qui permet de contrôler directement si les règles ont été respectées. Pour l'installer, il faut utiliser l'installateur de paquets fourni par Python `pip` :

```
$ pip install -U pycodestyle
```

Pour l'utiliser, il suffit de l'appeler en lui passant comme argument le fichier à tester :

```
$ pycodestyle digest.py
digest.py:24:3: E111 indentation is not a multiple of four
digest.py:25:3: E111 indentation is not a multiple of four
digest.py:26:3: E111 indentation is not a multiple of four
digest.py:50:18: E225 missing whitespace around operator
digest.py:54:80: E501 line too long (83 > 79 characters)
```

1.2. L'outil de documentation `Sphinx`

`Sphinx` est un outil de documentation qui permet d'éditer des documentations à partir de `docstrings`.

1.2.1. Les `docstrings`

Les `docstrings` sont des commentaires dans le code Python qui permettent de décrire des classes, des fonctions, ... Par défaut, ils peuvent prendre n'importe quelle forme du moment qu'ils sont entourés de trois guillemets:

```
def ma_fonction(parametre_1, parametre_2):
    """
    Ma fonction utilise deux paramètres pour en faire quelque chose
    """
    return parametre_1, parametre_2
```

De cette manière, dans l'interpréteur Python, la description peut être appelée:

```
>>> help(ma_fonction)
Ma fonction utilise deux paramètres pour en faire quelque chose
```

Dans ce TP, je vais utiliser un certain formattage de la `docstring` : le *Google Docstring*. Il a l'avantage d'être lisible dans le code et d'être reconnu par `Sphinx`, outil dont je parlerai dans la sous-partie suivante. Comme pour le `PEP8`, il possède de nombreux cas d'utilisations. Dans ce TP, je me cantonnerai à deux utilisations : la documentation de fonctions et la documentation de classe. Voici un exemple :

```
class MaClasse:
    """ Classe de gestion d'une classe MaClasse

    Cette classe est une classe permettant de gérer une classe classique

    Attributes:
        __attribut_1 (str): attribut de type chaine de caractère
        __attribut_2 (int): attribut de type entier
    """
    def methode_de_ma_classe(self, parametre_1, parametre_2):
        """ Methode appartenant à ma classe de gestion d'une classe MaClasse

        Cette méthode permet d'assigner des valeurs aux attributs de la classe
MaClasse

        Args:
            parametre_1 (str): paramètre de type chaine de caractères
            parametre_2 (int): paramètre de type entier

        Returns:
            boolean: True si l'affectation s'est correctement déroulée, False
sinon

        """
        #...
```

1.2.2 L'outil `Sphinx`

`Sphinx est un outil qui permet de générer une documentation à partir de docstrings défini dans le code. Pour l'installer il faut :

```
$ pip install -U Sphinx```
```

Puis, pour l'utiliser :

```
$ sphinx-quickstart
```

Cette commande permet de préparer un "projet" Sphinx complet. Elle pose quelques questions relatives au projet (nom, auteur, version, ...) puis crée deux dossiers : `sources` et `build`, ainsi qu'un fichier de

configuration `conf.py`, un fichier `index.rst` et un fichier `makefile` qui permettra de lancer la compilation.

Avant de pouvoir lancer la compilation de la documentation, il faut paramétrer Sphinx afin de pouvoir créer une documentation à partir du code Python (sinon, il faudrait l'écrire `index.rst`). Dans le fichier `conf.py`, il faut décommenter ou mettre à jour les lignes suivantes:

```
# conf.py
import os
import sys
sys.path.insert(0, os.path.abspath(r'..\.')) # Permet de spécifier le chemin du
code Python à analyser
...
extensions = ["sphinx.ext.autodoc"]
```

Dans le fichier `index.rst`, on ajoute le nom des modules (fichiers) Python à analyser:

```
# index.rst
.. automodule:: ex1_hello_world
   :members:
.. automodule:: ex2_editeur_fichier
   :members:
.. automodule:: ex3_class_Date
   :members:
```

Suite à ces modifications et la bonne implémentation des docstrings dans le code, il faut lancer la commande `make` accompagnée d'un argument qui permet de spécifier le format de destination. Ainsi, la commande `make html` permet, dans le dossier `build` de créer un ensemble de fichier dont le fichier `index.html`.

Dans le code fourni avec le TP, le dossier TP1 suit l'arborescence suivante :

```
TP
+- TP1
    +- build
        +- doctrees
        +- html
            +- index.html
            +- ...
    +- source
        +- conf.py
        +- index.rst
    +- ex1_hello_world.py       # fichier relatif à la question 1
    +- ex2_editeur_fichier.py   # fichier relatif à la question 2
    +- ex3_class_Date.py        # fichier relatif à la question 3
    +- make.bat
    +- Makefile
```

Pour afficher la documentation complète des modules Python du TP, il faut ouvrir le fichier `index.html` dans un navigateur.