Charte qualité

Codage - Nommage

Cette charte qualité regroupe des règles, conventions de codage et de nommage qui permettent une lisibilité claire et rapide du code. L'ensemble des groupes doit respecter les différents éléments décrits dans ce document afin d'assurer l'homogénéité des codes.

Nommage fichiers :

- Nom de fichier : significatif, en anglais (donc sans accents) et en minuscules.
- Version : spécifier le numéro de version
- Nom composé : éléments séparés par des underscores

Exemple : **g1_file_name_v0.extension**

Python :

Les règles définies ci-après sont largement inspirées de PEP 8 (*Python Enhancement Proposals - Style Guide for Python Code* : https://www.python.org/dev/peps/pep-0008/).

1- Version :

Nous travaillerons tous sur une version de Python **supérieure ou égale à Python 3.5** afin d'éviter les erreurs lors de la fusion des codes.
Il est possible de créer un environnement spécialement dédié au projet avec Anaconda :
https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/20/conda/

2- Import modules :

- Une ligne par import.
- Importer chaque module en une seule fois. L'utilisation de **from** est tolérée si on importe quelques fonctions spécifiques.

Ne pas écrire :

```
from math import *
import os, sys
```

Ecrire :

```
import os
import sys
import math
from sklearn.model_selection import train_test_split, cross_val_score
```

Les alias suivants seront systématiquement utilisés :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Les alias pourront également être utilisés pour les noms de modules trop longs.

3- Nommage variables, fonctions, classes et exceptions :

- Nom de variable : court (15 caractères maximum), significatif, en anglais et en minuscules.
- Nom de constante : court, significatif, en anglais et en majuscules (pour la première lettre).
- Nom composé : underscores pour séparer les éléments.
- Les noms de variables / constantes ne doivent pas être trop semblables :
  - Ne pas nommer une variable ou fonction « O », « l », ou « I » : ces noms ressemblent trop aux chiffres 0 et 1.
  - Un nom de classe doit être significatif, en anglais et doit respecter le CamelCase : une majuscule à chaque mot sans underscores séparateurs, exemple : **MyClass**.
  - Un nom d'exception suit les mêmes conventions qu'un nom de classe. Si l'exception est une erreur, utiliser le suffixe « Error », exemple : **ValueError**.

4- Type hints :

Un type hints indique le type d'une variable directement dans le code (au lieu de le préciser en commentaire).

- Utiliser les types hints dans les déclarations de variables :

Ne pas écrire :

```
age = 20  # type: int
```

Ecrire :

```
age: int = 20
```

- Utiliser les types hints dans les déclarations de fonctions :

```
def my_function(nb: int, to_add: float=3.5) -> float:
    return nb + my_float
```

5- Commentaires :

Les commentaires donnent des explications claires sur l'utilité du code. Ils doivent être en anglais. Il existe deux manières de commenter le code : les docstrings et les #.

- Chaque script **.py** doit commencer par une docstring au format suivant :

```
"""
Created on Fri Dec 6 17:03:35 2019
Group 1
@authors: A.B., Z.Y.
"""
```

- Notebooks Jupyter **.ipynb** : renseigner les mêmes informations dans une cellule Markdown.
- Les docstrings doivent être écrites avec des triples doubles guillemets. Toutes les classes, méthodes et fonctions doivent avoir une docstring, même si le commentaire est trivial :

```
class MyClass:
    """Documentation"""
    def my_method(self):
        """Documentation"""
```

- Dans le cas d'une fonction nécessitant des explications détaillées, utiliser la structure suivante :

```
def my_function(param1: type, param2: type, att1: type, att2: type) -> type:
    """Documentation

    Parameters:
        param1: description
        param2: description

    Attributes:
        att1: description
        att2: description

    Out (if exists):
        out1: description

    References:
        1. http://myfirstreference.com
        2. http://mysecondreference.com

    """
```

Note : étant donné que les types des paramètres, attributs et valeur retournée sont précisés par *type hints*, il n'est pas nécessaire de les mentionner dans la docstring.

- La compréhension du code doit être facilitée par des #commentaires.
- Un commentaire doit être situé sur le même niveau d'indentation que le code qu'il commente, de préférence avant celui-ci. Un commentaire sur la même ligne que le code commenté est toléré s'il est très court (20 caractères maximum) et pertinent.

```
#  Step 1: calculate and convert distances
my_list = list()
for v in vals:
    # convert from miles to kilometers, then round distance =
    np.round(v * 1.60934, 2)
```

- Suivre la règle "*D.R.Y.*" : *don't repeat yourself*. Par exemple, ne pas écrire :

```
#  for each n
for n in range(N):
```

- Ne pas oublier de mettre à jour les commentaires en cas de mise à jour du code.

6- Règles de codage :

- Encodage : UTF-8.
- Indenter le code (obligatoire en Python) avec 4 espaces.
- Longueur maximale d'une ligne : 79 caractères (72 pour les docstrings et commentaires).
- Utiliser autant que possible des fonctions.
- Comparaison avec un singleton (comme **None**) : utiliser **is** et **is not**, exemple :

Ne pas écrire :

```
if my_obj == None:
```

Ecrire :

```
if my_obj is None:
```

- Privilégier les affectations directes des booléens, exemple :

Ne pas écrire :

```
if x > 1000:
    accepted = True
else:
    accepted = False
```

Ecrire :

```
accepted = x > 1000
```

- Privilégier les utilisations directes des booléens, exemple :

Ne pas écrire :

```
if accepted == False:
```

Ecrire :

```
if not accepted:
```

Espaces :

- Un espace avant et après +, -, =, /, //, *, **, %, ==, +=, -=, !=, >, <, >=, <=, is, not, in, and, or. Exception : pas d'espace autour du =d'un paramètre d'une fonction.
- Pas d'espace à l'intérieur de [],{}ou ().
- Un espace après : et , mais pas avant. Exception : les tranches de liste.

```
my_variable = 1 + 2
my_text = 'hello'
my_text == str(my_variable)
my_list = [1, 2, 3]
my_list[1:2]
my_dict = {'key1': 'value1', 'key2': 'value2'}
my_function(param1=10, param2=15)
```

Sauts de lignes :

- Si nécessaire, sauter une ligne après un opérateur, et non avant :

```
my_total = (var1 +
            var2 +
            (var3 – var4) -
            var6)
```

- Laisser 2 lignes vides avant et après : définitions de classes.
- Laisser une ligne vide avant et après : définitions de méthodes et fonctions.
- Sauter des lignes si nécessaire pour marquer des séparations logiques entre sections.
- Listes et dictionnaires : à créer sur plusieurs lignes, en allant à la ligne après une virgule :

```
my_list = [
    'el1',
    'el2',
    'el3'
]
```

```
my_dict = {
    'key1': 1,
    'key2': 2,
    'key3': 3
}
```

NoSQL/SQL :

Le langage SQL n'est pas sensible à la casse mais pour une meilleure lecture, des règles sont définies.

1- Nommage entités, attributs :

- Nom d'une entité : court, significatif, en anglais et en majuscules.
- Nom d'un attribut : court, significatif, en anglais, en minuscules, underscore pour séparer les mots.
- Nom d'un attribut qui est un identifiant : id_attribut.

2- Commentaires :

Il est obligatoire de mettre un commentaire au début d'un bloc de code pour expliquer sa fonction générale.

- Commentaire multiligne : /* commentaire */, et laisser une ligne vide après le commentaire.
- Commentaire d'une ligne : # commentaire.
- Commentaire pour NoSQL : //commentaire.

3- Règles de codage :

- Mots clefs et instructions en majuscules.
- Retour à la ligne à chaque nouvelle instruction.
- Déclaration des attributs : retour à la ligne et indentation pour chaque attribut.
- Sélection : 3 sélections d'attributs par ligne maximum, si plus : retour à la ligne et indentation.
- Espace avant et après chaque opérateur.
- Laisser 2 lignes vides entre 2 blocs de lignes de code.