Convention de code Converteo équipe data science:

Cette convention n'est pas exhaustive et pourra être modifiée,
pour plus de détail sur les différentes best practices de développement aller sur les liens ci dessous :

- https://docs.python-guide.org/writing/structure/
- https://docs.python-guide.org/writing/style/
- https://www.python.org/dev/peps/pep-0008/

Essayer autant que possible de respecter la pep 20 :

The Zen of Python

```
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!
```

- Exemple format pdf

Essayer autant que possible de respecter pep 8 :

Documentation officielle
Autre

Les exemples les plus importants (à modifier si pas assez d'exemple):

1) les listes, tuples et autres séquences peuvent permettre le saut de ligne sans l'utilisation de "\" à utiliser si on veut augmenter la facilité de lecture (ligne trop grande)

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

2) si on doit faire une opération très longue en ligne utiliser l'expression dans un tuple pour sauter des lignes, faire ce saut de ligne avant chaques opérateurs

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

3) Saut de deux lignes entre chaque classe une ligne entre chaque fonction/méthode.

4) les imports doivent être fait dans des lignes séparé

```
# Correct:
import os
import sys
```

```
# Wrong:
import sys, os
```

une exception peut être faite si les sous-modules viennent du même module

```
# Correct:
from subprocess import Popen, PIPE
```

5) Évitez le surplus de commentaire, un commentaire doit être simple est concis de manière à faciliter la lecture du code, l'objectif que l'on devrait avoir en tête est d'avoir un code tellement explicite et simple (nom de fonction/méthode explicite, de même pour les variables) qu'on ne devrait même pas à avoir à le commenter quitte à perdre en performance, les tests peuvent aussi servir à éviter le surplus de commentaire en montrant comment utiliser la fonction/méthode.

6) Respecter les conventions de nommage :

- éviter les noms qui sont déjà pris par python (utilisation de _ si vous n'avez pas d'autres choix)

  exemple :
  file => file_

- Les noms de classe doivent être respectées CapWords convention c'est à dire chaque nom/mot avec la première lettre en majuscule et aucun _

  exemple :
  NomDeClasse

- les noms de package/modules doivent être en minuscule sans underscore
- les noms de variable et de fonction/méthode doivent être en minuscule est chaque mot séparé par un underscore

- lorsque que l'on définie une méthode pour une instance/object on utilise le mot cle *self* en premier argument *cls* pour une méthode de classe

- pour des constantes (variable qui ne sera pas modifiée) utiliser des majuscules et séparer les mot par des undescore

  exemple :
  NOM_DE_CONSTANTE

- pour les attributs privé en python on utilise un double underscore en début,
  pour un attribut protégé on utilise un seule
  exemple :
  __nom_dattribut (privé)
  _nom_dattribut (protégé)

  En réalité ces attributs ne sont pas réellement privés ou protégés (ils peuvent être modifiés) cependant les underscores en début avertit les autres développeurs que ces attributs ne doivent pas être modifiés durant l'utilisation de la classe/instance/object.

Il peut être difficile de retenir toutes ces règles surtout quand le software engineering n'est pas le cœur de métier de la personne qui les lit, il faut donc surtout se concentrer sur les conventions de nommage, la pep 20.
Pour la pep 8 des librairies qui reformatte votre code existe, je vous recommande fortement d'utiliser **black** très simple d'utilisation.

"pip install black" puis "black code_a_reformatter.py"
ou si ça ne marche pas "python black code_a_reformatter.py"

Autre recommandation :

- Une fonction/méthode ne doit faire qu'une est une seule chose, si ce n'est pas le cas décomposer la fonction en plusieurs sous fonctions
- les fonctions/méthodes doivent être courtes si possible, sinon décomposer en sous fonctions/méthodes
- privilégiez les listcomps/dictcomp/etc.. si le code n'est pas trop long

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

- pour documenter une fonction/méthode utilisation de docstring avec le style numpy/google (choix de l'équipe à venir)

Google style:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value

    """
    return True
```

NumPy style:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

- Exemple complet style numpy
- Exemple complet style Google

- Privilégiez l'annotation de type pour aider à la lecture du code, le code ne bloquera pas si elle ne sont pas respecté

```python
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Pour plusieurs type utiliser la librairie typing :

```python
import typing as t
```

```python
def __init__(
    self,
    mu_biais: t.Union[int, float] = 0,
    std_biais: t.Union[int, float] = 10,
    mu_betas: t.Union[int, float] = 0,
    std_betas: t.Union[int, float] = 1,
    sigma_std: t.Union[int, float] = 10,
    size_sample: t.Union[int, float] = 2000,
):
```

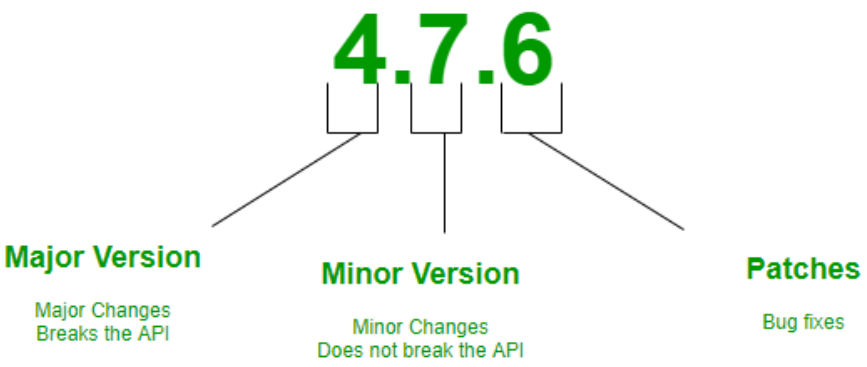
Pour des typages non natif utilisé ceux proposés par la librairie typing ou collections.abc :

```python
from collections.abc import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None:
```

```python
def example(lol: t.Dict, lolilol: t.List[float], gg: t.Any) -> None:
    pass
```

recommande l'utilisation de typing si connaisse de python moyenne, cependant collection.abc permet un typing plus générique

- Lors de l'utilisation de requirements.txt (gestion de dépendance) faire en sorte que la version de librairie utilisé soit entre deux minors (Le minor change peuvent “cassé” le code contrairement à l’image indiqué ci-dessous)



Exemple :

```
pandas>=1.1.4,<1.2.0
numpy>=1.19.5,<1.21.0
matplotlib>=3.2.2,<3.3.0
seaborn>=0.11.0,<0.12.0
statsmodels>=0.12.1,<0.13.0
scikit-learn>=0.24.0,<0.25.0
Sphinx>=1.8.5,<1.9.0
prince>=0.7.1,<0.8.0
pytest>=6.2.3,<6.3.0
tqdm>=4.55.0,<4.56.0
hyperopt>=0.2.5,<0.3.0
pymc3>=3.11.2,<3.12.0
arviz>=0.11.2,<0.12.0
google-api-core>=1.30.0,<1.31.0
google-auth>=1.32.1,<1.33.0
google-cloud-bigquery>=2.20.0,<2.21.0
google-cloud-core>=1.7.1,<1.8.0
google-cloud-storage>=1.40.0,<1.49.0
google-crc32c>=1.1.2,<1.2.0
google-resumable-media>=1.3.1<1.4.0
googleapis-common-protos>=1.53.0,<1.54.0
pyarrow>=4.0.1,<4.1.0
pydantic>=1.8.0,<1.9.0
tqdm>=4.55.0, <4.56.0
click>=7.1.0,<7.2.0
```

- création d'un environnement virtuel pour chaque nouveau projet pour ne pas avoir de problème de dépendance à venir, si développement d'un project présentant un requirements.txt =>

  création de nouvel environnement
  +

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

  - Création d'environnement avec Virtualenv
  - Création d'environnement avec Conda

  privilégiez Conda.

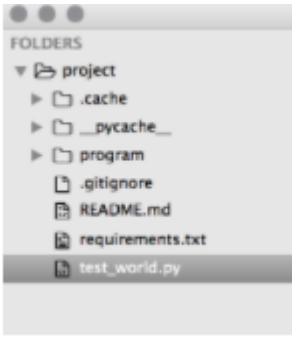
- pour chaque module ne pas oubliez d'intégrer le __init__.py

  - nom_librairie
    - setup.py
    - nom_sous_module
      - sous_module.py
      - __init__.py

- Pour les parties critiques de code définir à minima un test avec pytest
  - installation :

```
pip install -U pytest
```

  - tous les codes contenant les tests doivent respecter la syntaxe par test_*.py



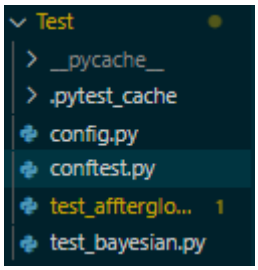
  - définir une assertion pour la vérif

```python
def hello(name):
    return 'Hello ' + 'name'

def test_hello():
    assert hello('Celine') == 'Hello Celine'
```

- Toutes les fonctions de test doivent posséder le préfixe“test_”

```python
from Core.Contributions import Bayesian_Modelisation
import numpy as np
import pandas as pd


def test_bayesian_base(data_bayesian_inference):
    X, y = data_bayesian_inference
    objinference = Bayesian_Modelisation.InferenceBayesianRegression()
    objinference.inference(X, y)
    r2_square = objinference.get_rsquarred(y).r2

    assert r2_square > 0.8


def test_bayesian_robust(data_bayesian_inference):
    X, y = data_bayesian_inference
    objinference = Bayesian_Modelisation.RobustInferenceBayesianRegression()
    objinference.inference(X, y)
    r2_square = objinference.get_rsquarred(y).r2

    assert r2_square > 0.8

def test_bayesian_ridge(data_bayesian_inference):
    X, y = data_bayesian_inference
    objinference = Bayesian_Modelisation.RidgeInferenceBayesianRegression()
    objinference.inference(X, y)
    r2_square = objinference.get_rsquarred(y).r2

    assert r2_square > 0.8

def test_bayesian_robust_ridge(data_bayesian_inference):
    X, y = data_bayesian_inference
    objinference = Bayesian_Modelisation.RobustRidgeInferenceBayesianRegression()
    objinference.inference(X, y)
    r2_square = objinference.get_rsquarred(y).r2

    assert r2_square > 0.8
```

- Si le test à besoin de données externe utilisé un conftest.py + pytest.fixture()




```python
@pytest.fixture
def data_bayesian_inference():
    np.random.seed(314)
    N = 100
    alpha_real = 2.5
    beta_real = [0.9, 1.5]
    eps_real = np.random.normal(0, 0.5, size=N)
    X = np.array([np.random.normal(i, j, N) for i, j in zip([10, 2], [1, 1.5])]).T
    X_mean = X.mean(axis=0, keepdims=True)
    X_centered = X - X_mean
    y = alpha_real + np.dot(X, beta_real) + eps_real

    return X_centered, y
```

vas permettre d'appeler les données de sortie dans les codes test par le nom de la fonction

```python
def test_bayesian_ridge(data_bayesian_inference):
    X, y = data_bayesian_inference
    objinference = Bayesian_Modelisation.RidgeInferenceBayesianRegression()
    objinference.inference(X, y)
    r2_square = objinference.get_rsquarred(y).r2

    assert r2_square > 0.8
```

Adresse si incompréhension : jonathan.ndamba@converteo.com

Pour aller plus loin :

- Environnement virtuel
- Code maintenable
- Débuggage
- Orientez Objet
- S'améliorer
- Comment écrire un package
- Top niveau (nécessite un bon niveau de python pour être lu)