# [Perfectionnez-vous en Python](https://openclassrooms.com/fr/courses/4425111-perfectionnez-vous-en-python)

# Table des matières

[Perfectionnez-vous en Python 1](#_Toc64280494)

[Table des matières 1](#_Toc64280495)

[Apprendre en faisant 5](#_Toc64280496)

[Organisation des chapitres 5](#_Toc64280497)

[Commençons ! 6](#_Toc64280498)

[Organisez un script 7](#_Toc64280499)

[Interpréteur et encoding 7](#_Toc64280500)

[Exécution maîtrisée 7](#_Toc64280501)

[C'est à vous ! 9](#_Toc64280502)

[Code du chapitre 9](#_Toc64280503)

[Travaillez dans un environnement virtuel 11](#_Toc64280504)

[Utiliser Virtualenv 11](#_Toc64280505)

[Un fichier de dépendances 13](#_Toc64280506)

[A vous de jouer ! 14](#_Toc64280507)

[Code du chapitre 15](#_Toc64280508)

[Organisez un projet en modules 16](#_Toc64280509)

[Organisation du projet 16](#_Toc64280510)

[Analyse d'un fichier CSV 16](#_Toc64280511)

[Créer un module 18](#_Toc64280512)

[Créer une librairie 20](#_Toc64280513)

[Lancer les fichiers en ligne de commande 22](#_Toc64280514)

[A vous de jouer ! 25](#_Toc64280515)

[Code du chapitre 25](#_Toc64280516)

[Gérez les erreurs et les bogues 26](#_Toc64280517)

[Deboguer Python avec PDB 26](#_Toc64280518)

[Erreurs et exceptions 27](#_Toc64280519)

[Logs 32](#_Toc64280520)

[A vous de jouer ! 33](#_Toc64280521)

[Code du chapitre 33](#_Toc64280522)

[Travaillez comme un·e pro ! 35](#_Toc64280523)

[Compétences évaluées 35](#_Toc64280524)

[ Question 1 35](#_Toc64280525)

[ Question 2 35](#_Toc64280526)

[ Question 3 35](#_Toc64280527)

[ Question 4 36](#_Toc64280528)

[ Question 5 36](#_Toc64280529)

[ Question 6 36](#_Toc64280530)

[ Question 7 36](#_Toc64280531)

[ Question 8 37](#_Toc64280532)

[ Question 9 37](#_Toc64280533)

[ Question 10 38](#_Toc64280534)

[Créez votre premier graphique 39](#_Toc64280535)

[Installer les trois librairies 39](#_Toc64280536)

[La librairie NumPy 39](#_Toc64280537)

[La librairie Pandas 43](#_Toc64280538)

[La librairie Matplotlib 52](#_Toc64280539)

[Code du chapitre 56](#_Toc64280540)

[Découvrez les méthodes spéciales 58](#_Toc64280541)

[Calculer le nombre de parlementaires 58](#_Toc64280542)

[Les méthodes spéciales 59](#_Toc64280543)

[Les autres méthodes spéciales 61](#_Toc64280544)

[A vous de jouer ! 62](#_Toc64280545)

[Code du chapitre 62](#_Toc64280546)

[Apprivoisez les décorateurs 64](#_Toc64280547)

[Les décorateurs 101 64](#_Toc64280548)

[Sous le capot 66](#_Toc64280549)

[Créer un décorateur plus générique 67](#_Toc64280550)

[A vous de jouer ! 70](#_Toc64280551)

[Code du chapitre 70](#_Toc64280552)

[Utilisez des expressions régulières 72](#_Toc64280553)

[Les expressions régulières 72](#_Toc64280554)

[La librairie standard re 74](#_Toc64280555)

[Code du chapitre 77](#_Toc64280556)

[Faites connaissance avec les générateurs et les itérateurs 78](#_Toc64280557)

[Les itérateurs 78](#_Toc64280558)

[Les générateurs 83](#_Toc64280559)

[Les expressions génératrices 85](#_Toc64280560)

[Aller plus loin : L'intérêt des générateurs 86](#_Toc64280561)

[Code du chapitre 91](#_Toc64280562)

[Approfondissez votre utilisation de Python 92](#_Toc64280563)

[Compétences évaluées 92](#_Toc64280564)

[ Question 1 92](#_Toc64280565)

[ Question 2 92](#_Toc64280566)

[ Question 3 92](#_Toc64280567)

[ Question 4 92](#_Toc64280568)

[ Question 5 93](#_Toc64280569)

[ Question 6 94](#_Toc64280570)

[ Question 7 94](#_Toc64280571)

[ Question 8 95](#_Toc64280572)

[ Question 9 95](#_Toc64280573)

[ Question 10 95](#_Toc64280574)

[Assimilez les bonnes pratiques de la PEP 8 96](#_Toc64280575)

[Python Extension Proposal : PEP 96](#_Toc64280576)

[La PEP 8 96](#_Toc64280577)

[Contenu de la PEP 8 96](#_Toc64280578)

[La PEP 8 en action dans le projet 99](#_Toc64280579)

[Code du chapitre 101](#_Toc64280580)

[Devenez zen avec la PEP 20 102](#_Toc64280581)

[La PEP 20 102](#_Toc64280582)

[En pratique 102](#_Toc64280583)

[Allez plus loin 107](#_Toc64280584)

[📖  Livres 107](#_Toc64280585)

[💻  Sites 107](#_Toc64280586)

[🎥  Vidéos 107](#_Toc64280587)

[🍻  Projets open source pour s'entraîner 107](#_Toc64280588)

Python est un langage bien connu à la fois pour sa **simplicité** et sa **robustesse**. Il s'appuie sur une **communauté solide** qui a construit progressivement un des écosystèmes les plus **ouverts** au monde. Les développeurs qui constituent cette communauté se basent sur de **bonnes pratiques** solidement ancrées dans leur vie professionnelle. Mais les connaissez-vous ?

Dans ce cours nous verrons ensemble comment démarrer un projet en utilisant un **environnement virtuel**. Nous aborderons également des notions avancées du langage, tels que les **décorateurs** ou les **générateurs**. Enfin, nous passerons en revue les **bonnes pratiques de la PEP 8**.

Nous vous montrerons également comment utiliser **Matplotlib** pour créer des graphiques et **Numpy** pour manipuler des données.

Vous êtes prêts à devenir un·e meilleur·e développeur·se ? **Inscrivez-vous !**

Prérequis :

Savoir créer un script en Python et l'exécuter

* Être à l'aise sur les principes fondamentaux de la programmation orientée objet en Python.

Vous doutez ? Commencez par suivre les cours [**Démarrez votre projet avec Python**](https://openclassrooms.com/courses/demarrez-votre-projet-avec-python) et [**Découvrez la programmation orientée objet avec Python**](https://openclassrooms.com/courses/decouvrez-la-programmation-orientee-objet-avec-python).

**Outils nécessaires :**

* Python
* Éditeur de code
* Librairies Pandas, Numpy, Matplotlib

À la fin de ce cours, vous aurez acquis les compétences suivantes :

* Utiliser les notions avancées de Python.
* Respecter de bonnes pratiques de développement.
* Manipuler les librairies Pandas, Numpy et Matplotlib.

Ce cours a été écrit en partenariat avec [**Régis Behmo**](https://twitter.com/regisb). Merci ! 🖖

Bienvenue dans ce cours !

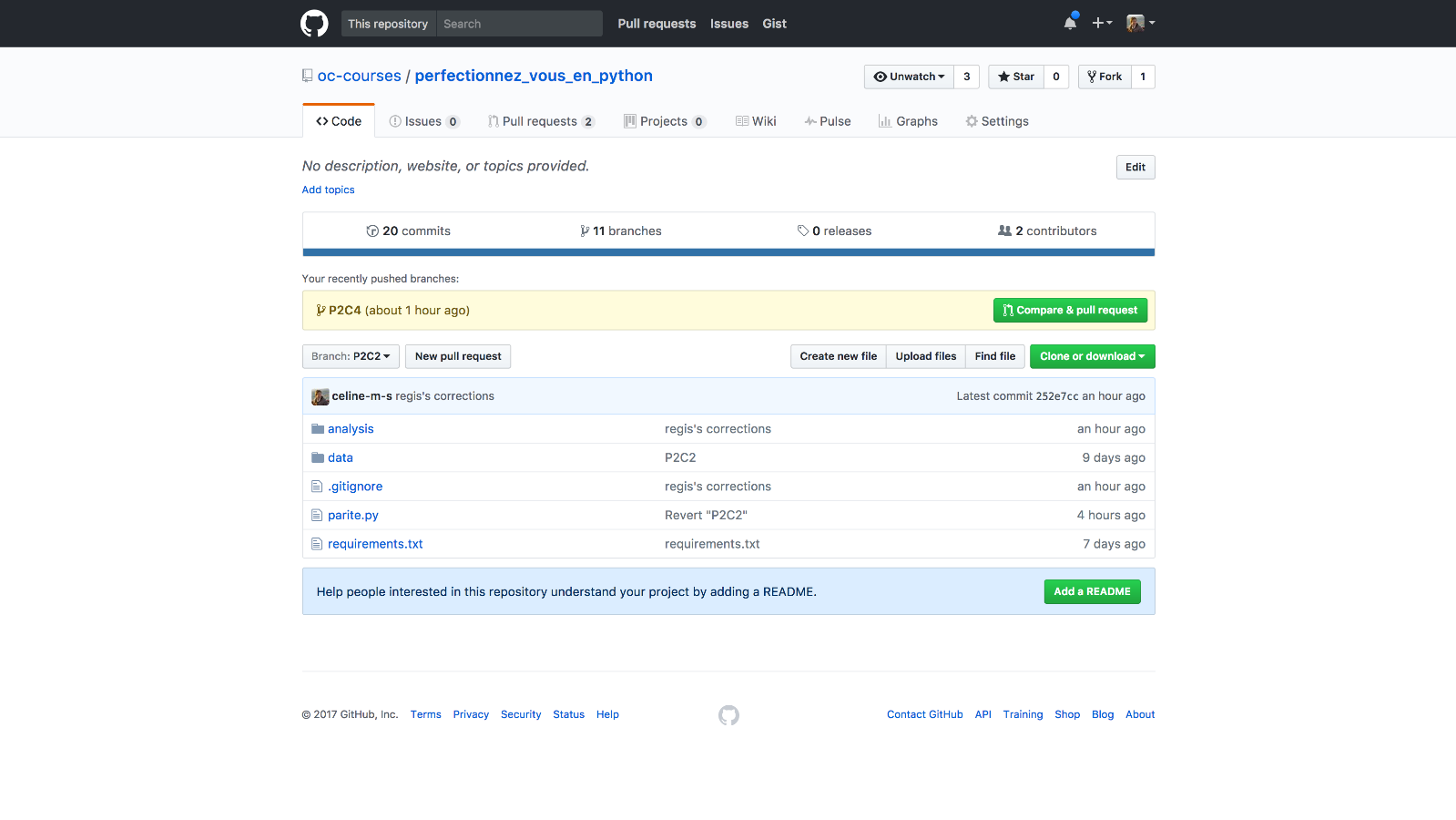
Nous allons parcourir de nombreuses notions extrêmement intéressantes ! Vous y découvrirez des bonnes pratiques qui vous permettront d'améliorer votre pratique du langage. Je vous promets que ce sera passionnant !

### Apprendre en faisant

La meilleure manière d'apprendre un langage est de pratiquer ! Nous avons d'ailleurs développé un projet qui nous servira de fil rouge dans ce cours : l'analyse de données provenant de l'Assemblée nationale. Hé oui, nous allons analyser nos chers politiques !

Nous nous intéresserons plus particulièrement aux parlementaires et analyserons la parité de chaque parti politique. Vous apprendrez à trouver les informations pertinentes dans un fichier conséquent, à les trier et à créer des graphiques.

Vous pouvez d'ailleurs [accéder au projet final sur Github](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python) dès maintenant.



### Organisation des chapitres

Chaque chapitre commence par un aperçu global des notions abordées par la suite. Puis nous ajoutons une fonctionnalité à notre programme afin d'illustrer les différents concepts. Enfin, nous vous invitons à lire l'intégralité du code source du chapitre.

Ce dernier est [librement accessible sur Github](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python). Chaque chapitre correspond à une branche différente.

Nous vous invitons fortement à **réaliser le projet en même temps** que nous afin de bien assimiler les différents concepts de ce cours.

Nous vous expliquerons tout, pas à pas, mais parfois le code source d'un chapitre intègre des lignes supplémentaires qui sont nécessaires au bon fonctionnement du programme. Regardez donc, à la fin du chapitre, que le code que vous avez écrit correspond au nôtre !

### Commençons !

Avant de continuer, téléchargez le fichier avec lequel nous travaillerons :

* [La liste détaillée de nos députés](https://www.nosdeputes.fr/deputes/enmandat/csv) : renommez-le en  current\_mps.csv  pour faciliter la suite de ce cours.
* [Les comptes rendus des débats publics depuis 2013](http://data.assemblee-nationale.fr/travaux-parlementaires/debats) comprenant les jours, dates, numéros de séance, thèmes de discussion, l'ensemble des orateurs et les textes des débats. **Dézippez ce fichier**.

N'ouvrez pas les documents pour le moment. Nous verrons plus tard comment les utiliser.

Tout est téléchargé ? Commençons !

## Organisez un script

Dans ce chapitre nous aborderons les notions suivantes :

* Organisation d'un script Python
* Utilisation de  if \_\_name\_\_ == "\_\_main\_\_"

Dans ce chapitre, commencez votre tout nouveau projet dans les règles de l'art : en suivant de bonnes pratiques qui vous serviront dans bien des cas.

Commencez par créer un dossier pour votre programme. À l'intérieur, créez un fichier que vous appellerez parite.py.

Ouvrez-le avec votre éditeur de texte favori.

Vous souhaitez utiliser le terminal mais ne vous souvenez plus des commandes ? Consultez les récapitulatifs suivants : [**Windows**](https://ss64.com/nt/), [**Linux**](https://ss64.com/bash/) et [**Mac**](https://ss64.com/osx/).

### Interpréteur et encoding

Ajoutez deux lignes en haut de votre script.

#! /usr/bin/env python3

# coding: utf-8

* #! /usr/bin/env python3 : ce commentaire conditionnel indique que ce script doit être exécuté à l'aide de Python 3. Cela permet au système d'exploitation de connaître le**chemin d'accès** vers l'interpréteur Python. Sans cette ligne, vous pouvez rencontrer des problèmes lors de l'exécution du script.
* # coding: utf-8 : cette ligne spécifie **l'encodage** du code source de notre script. Afin de prendre en compte les accents de notre chère langue fraaaançaise, nous utilisons le très commun utf-8. C'est souvent inutile si vous utilisez Python 3, car cette version utilise par défaut UTF 8, mais **nécessaire** avec Python 2.

### Exécution maîtrisée

Créez ensuite une fonction main() qui aura pour objectif de contenir les différentes instructions lancées lors de l'exécution de votre script.

Il s'agit d'une convention qui rend votre code plus lisible. Ainsi, lorsque vous ouvrirez votre projet dans six mois vous saurez exactement où regarder pour comprendre comment votre programme est exécuté !

def main():

pass

main()

[Spoiler] Nous y intègrerons certainement des instructions pour afficher les graphiques ou bien pour imprimer, sur le terminal, le nombre de députés contenu dans notre fichier de données.

D'ailleurs, laissez-moi vous parler d'une subtilité très importante. Que se passe-t-il si vous importez ce fichier dans un autre script en tant que module ? Actuellement, cela va déclencher la fonction main() et donc, par conséquent, exécuter des instructions qui devraient être lancées uniquement lors de l'exécution du fichier en ligne de commande.

Je n'ai pas bien compris pourquoi...

C'est assez simple. Lorsque vous importez un module, Python lit et exécute toutes les instructions de ce fichier. Vous pouvez ainsi utiliser tous les objets dans votre script. Si vous terminez par l'exécution d'une fonction, Python vous obéira : il exécutera bien la fonction main() !

Alors, comment faire ?

Encapsulez la fonction main() dans une structure conditionnelle :

if \_\_name\_\_ == "\_\_main\_\_":

main()

Vous ne comprenez pas ? C'est normal, moi non plus je n'y comprenais rien quand j'ai découvert cela !

Je vous ai dit que l'interpréteur Python exécutait tout le code présent dans un fichier lors de sa lecture. En vérité, j'ai omis une partie de l'histoire : Python définit plusieurs variables spéciales **avant** la lecture du fichier. Par exemple, lorsque l'interpréteur Python lit votre fichier en tant que programme principal (donc si vous l'exécutez en ligne de commande), il crée une variable spéciale \_\_name\_\_ et l'associe à la valeur "\_\_main\_\_". Si votre fichier est importé en tant que module, la variable \_\_name\_\_ aura comme valeur le nom de votre module.

Prenons un autre exemple.

Imaginons le fichier suivant :

karadoc.py

#! /usr/bin/env python3

# coding: utf-8

if \_\_name\_\_ == "\_\_main\_\_":

print("Les chicots, c'est sacré ! Parce que si j'les lave pas maintenant, dans dix ans, c'est tout à la soupe. Et l'mec qui me fera manger de la soupe il est pas né !")

else:

print('Du passé faisons table en marbre.')

Exécutons le script :

$ python karadoc.py

Les chicots, c'est sacré ! Parce que si j'les lave pas maintenant, dans dix ans, c'est tout à la soupe. Et l'mec qui me fera manger de la soupe il est pas né !

Si par contre vous l'importez, vous pouvez constater que le résultat est différent :

$ python

Python 3.6.1 (default, Mar 23 2017, 16:49:06)

[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import karadoc

Du passé faisons table en marbre.

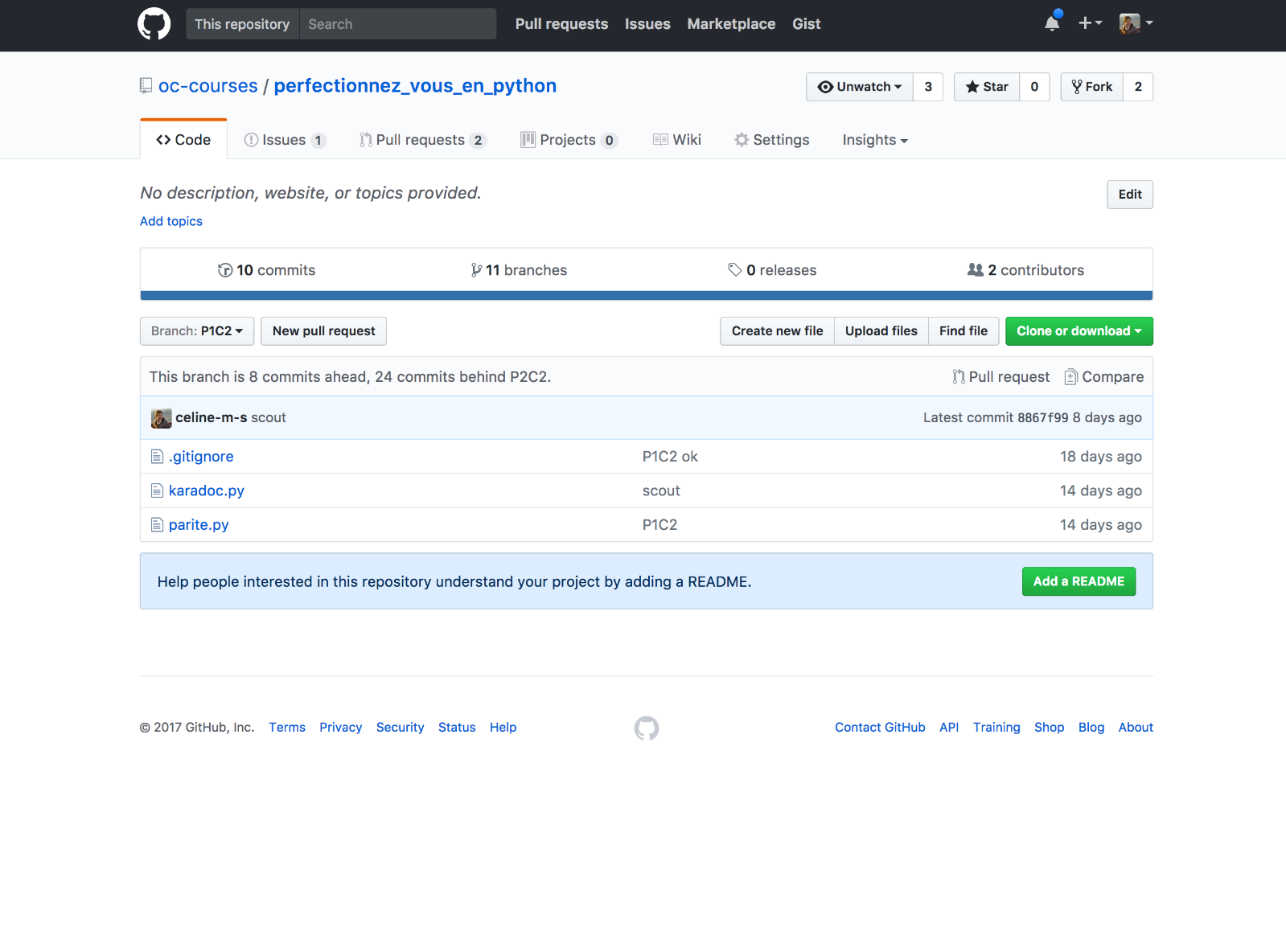
Bien ! Votre fichier est prêt à contenir le reste du projet. Avant de passer à la suite, j'aimerais vous présenter le concept d'environnement virtuel. À tout de suite !

### C'est à vous !

[Cliquez sur ce lien](https://www.codevolve.com/api/v1/publishable_key/2A9CAA3419124E3E8C3F5AFCE5306292?content_id=5180d874-c0e2-4829-9a40-25cfb1e90c77)

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P1C2)



## Travaillez dans un environnement virtuel

 Dans ce chapitre nous aborderons les notions suivantes :

* Travailler dans un environnement virtuel
* Fichier  .gitignore
* Installer des librairies via pip en se basant sur un fichier de dépendances

Vous utiliserez plusieurs librairies dans ce cours : Numpy, Scipy... Peut-être avez-vous déjà travaillé avec ces librairies ?

Si c'est votre première fois, il vous faudra les installer. Par défaut, vous les installerez dans votre ordinateur. Mais lorsque vous aurez développé plusieurs projets vous aurez peut-être la désagréable surprise de constater des erreurs dues à des versions différentes.

Imaginons que vous travailliez avec la version 1.0 de Librairie1 dans un projet X. Six mois plus tard, vous avez de nouveau besoin de cette librairie mais la version a changé, il s'agit désormais de la 1.1. Vous installez donc la nouvelle version qui, elle-même, dépend d'autres modules ! Très vite, un château de cartes instable s'installe dans votre ordinateur.

L'idéal, me direz-vous, serait de pouvoir isoler les librairies installées par projet. Ou même mieux : de pouvoir décider quel environnement de développement nous souhaitons utiliser.

C'est exactement l'objectif des environnements virtuels : créer un environnement comprenant une certaine version de Python et les librairies que vous souhaitez. Si votre ordinateur était un immeuble, nous pourrions imaginer qu'un environnement virtuel serait une pièce de cet immeuble dans laquelle vous installez ce que vous souhaitez. Lorsque vous entrez dans la pièce, les librairies installées sont accessibles. Mais lorsque vous la quittez, plus rien n'existe !

### Utiliser Virtualenv

Il existe plusieurs solutions d'environnement virtuel. Dans notre cas nous allons utiliser [VirtualEnv](https://virtualenv.pypa.io/en/stable/).

#### Installation

Commencez donc par installer VirtualEnv :

pip install virtualenv

Puis déplacez-vous dans votre dossier de travail.

#### Créer et activer un environnement virtuel

Concrètement, comment fonctionne un environnement virtuel ?

Commencez par créer un environnement virtuel en lançant la commande suivante :

virtualenv -p python3 env

Si vous travaillez avec Windows, exécutez la commande suivante dans [**PowerShell**](https://fr.wikipedia.org/wiki/Windows_PowerShell) :

virtualenv -p $env:python3 env

Elle crée un dossier env utilisant Python3 dans le répertoire courant.

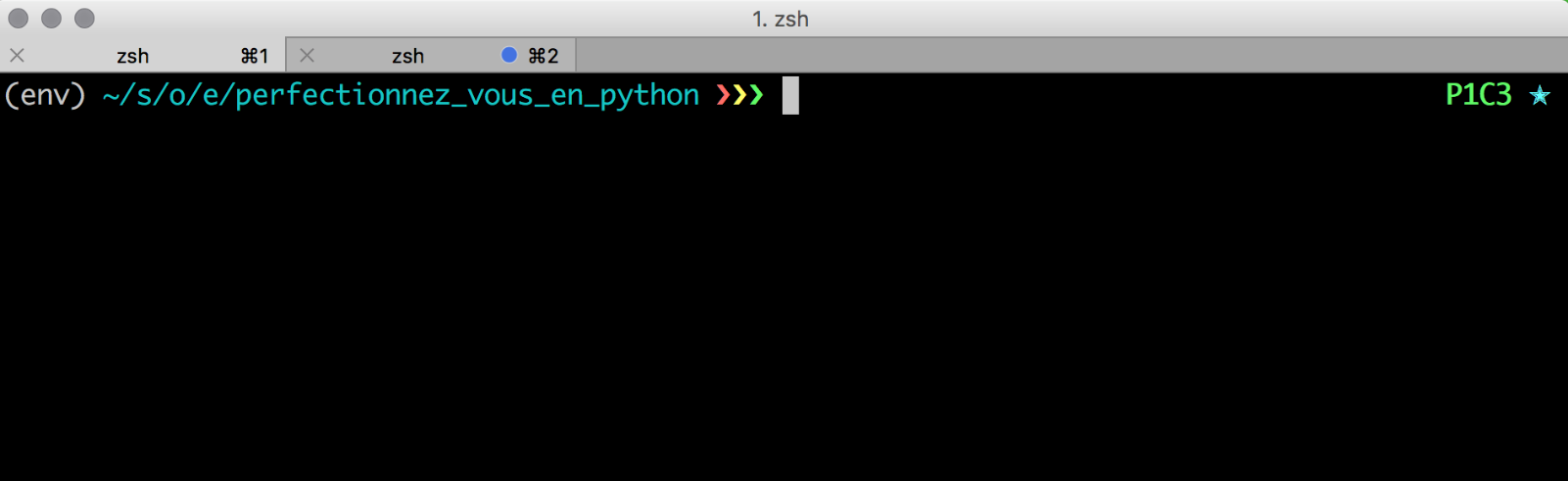
Pour activer l'environnement, écrivez la commande suivante :

source env/bin/activate

Si vous travaillez avec PowerShell, exécutez la commande suivante :

./env/scripts/activate.ps1

Vous constatez alors que l'environnement virtuel est indiqué entre parenthèses sur la gauche de votre console.



À partir de maintenant, l'interpréteur Python utilisé n'est plus celui de votre système mais bien celui de l'environnement virtuel. Vous pouvez le voir très facilement en utilisant la commande which   (UNIX) ou  where (Windows) qui affiche le chemin vers l'exécutable :

$ which python

/chemin/vers/votre/projet/env/bin/python

Si vous utilisez PowerShell, lancez la commande suivante pour afficher l'endroit duquel est exécuté pip:

pip -V

Les packages installés sont également différents :

$ pip freeze

appdirs==1.4.3

astroid==1.5.2

cycler==0.10.0

isort==4.2.5

lazy-object-proxy==1.3.1

matplotlib==2.0.2

mccabe==0.6.1

numpy==1.12.1

packaging==16.8

pandas==0.20.1

pylint==1.7.1

pyparsing==2.2.0

python-dateutil==2.6.0

pytz==2017.2

scipy==0.19.0

seaborn==0.7.1

six==1.10.0

wrapt==1.10.10

Si vous ouvrez un autre terminal, vous constatez que l'environnement virtuel n'est pas activé :

$ which python

/usr/local/bin/python

$ pip freeze

Could not find an activated virtualenv (required).

Il faut le réactiver à chaque fois que vous souhaitez l'utiliser.

#### Désactiver un environnement virtuel

Pour désactiver un environnement virtuel, tapez simplement deactivate.

Avant d'installer une librairie, vérifiez que votre environnement virtuel est bien activé et qu'il s'agit bien de celui de votre projet !

#### Supprimer un environnement virtuel

Pour supprimer un environnement virtuel, rien de plus simple ! Commencez par le désactiver puis supprimez son répertoire.

(env) $ deactivate

$ rm -rf env

L'utilisation d'un environnement virtuel est très courante. Elle permet de ne pas avoir à installer les dépendances d'un projet sur le système d'exploitation et facilite la collaboration.

#### Désactiver le suivi Git

Si vous utilisez Git (et j'espère bien que oui ! :) ), il faut impérativement désactiver le suivi de modifications de votre environnement virtuel. En effet, vous ne voulez pas envoyer sur Github tout votre environnement virtuel !

Pour cela, créez un fichier .gitignore et ajoutez à l'intérieur la ligne suivante :

env/

Elle indique à Git d'ignorer le dossier env.

Si votre environnement virtuel porte un autre nom, vous devez également le changer à cet endroit-là !

### Un fichier de dépendances

Imaginons à présent que le programme soit fini. Vous le publiez sur Github afin que toute personne intéressée puisse y accéder.

Comment ces dernières peuvent-elles connaître les différentes librairies utilisées dans le projet ? Elles n'auront pas accès à votre environnement virtuel. Bien sûr, elles peuvent regarder en haut des différents fichiers et chercher les imports. Mais cela est fastidieux et, surtout, il sera impossible de connaître les versions utilisées.

Les librairies qui ne sont pas standards, qui doivent par conséquent être installées, sont ce que nous appelons des **dépendances**. En effet, le projet est dépendant de leur installation et ne peut pas fonctionner sans.

Afin de remédier à ce souci, il existe une convention : créer un fichier requirements.txt qui liste les différentes librairies utilisées ainsi que leur version.

#### Créer un fichier de dépendances

Pypi peut le faire automatiquement pour vous ! Pour cela, exécutez la commande suivante.

pip freeze > requirements.txt

Cette commande "gèle" les librairies utilisées ainsi que leur version en les listant dans un document.

Si vous utilisez PowerShell, exécutez la commande suivante :

pip list > requirements.txt

Voici ce qui a été généré :

requirements.txt

appdirs==1.4.3

packaging==16.8

pyparsing==2.2.0

six==1.10.0

Cette commande est très utile mais peut s'avérer restrictive. Pip indique en effet toutes les librairies utilisées, y compris des librairies fondamentales telles que packaging ou six que vous n'avez pas besoin d'installer. C'est pourquoi ce que nous préconisons est d'ajouter à la main les dépendances explicitement requises par un projet.

requirements.txt

requests

pandas

#### Installer des dépendances

Si un fichier requirements.txt existe déjà, vous pouvez facilement en installer toutes les librairies en exécutant la commande suivante :

pip install -r requirements.txt

Parfait ! Nous avons vu dans ce chapitre comment utiliser un environnement virtuel et installer un fichier de dépendances. Dans le prochain chapitre nous commencerons l'analyse ! A tout de suite !

### A vous de jouer !

**Console de code**

##### À votre tour de coder

**Mettez immédiatement en application ce que vous avez appris.**  
Vous êtes libre de vous tromper et de réessayer autant que vous le souhaitez.

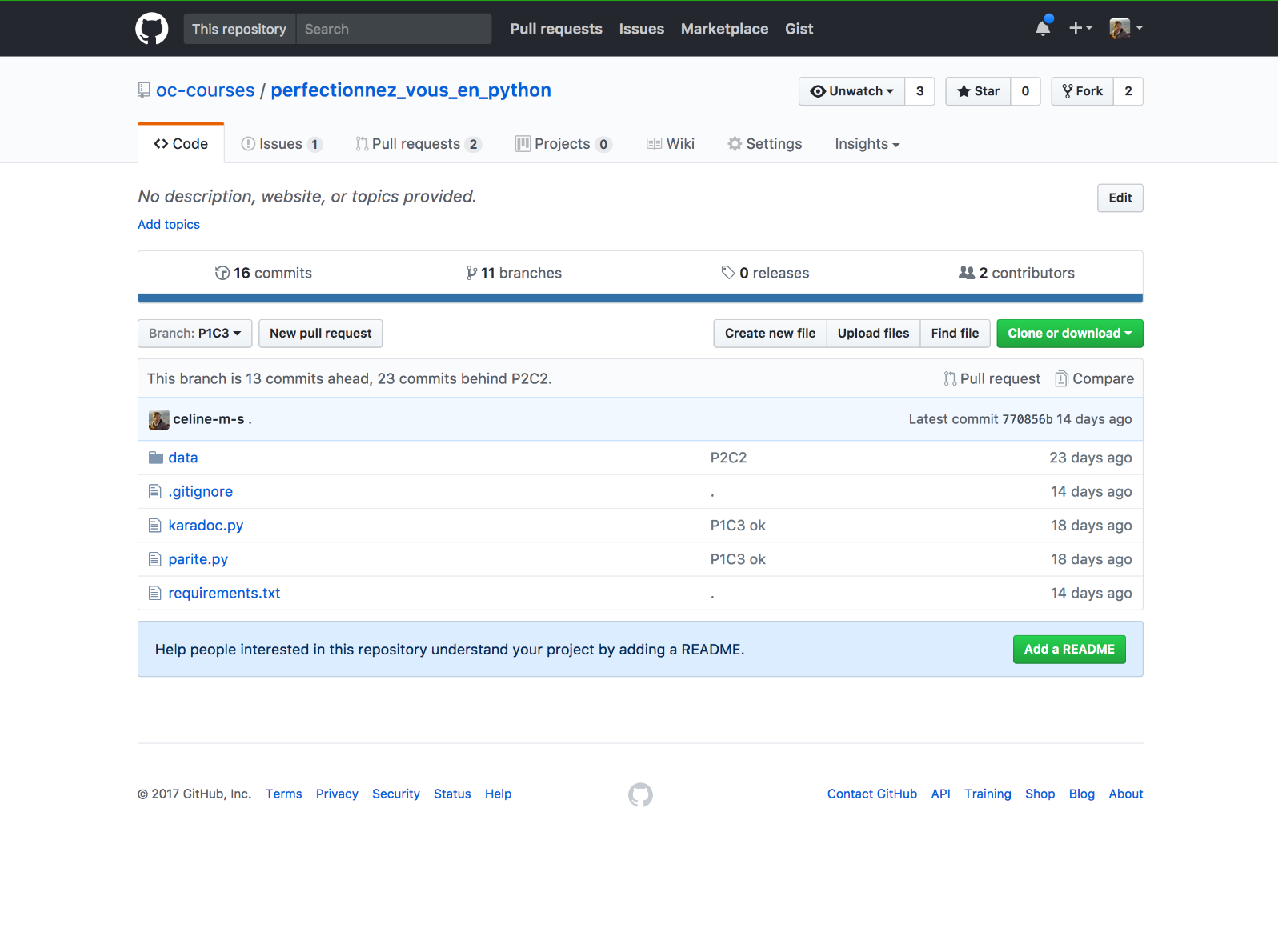
JE CODE MAINTENANT !

JE NE PEUX PAS, J'AI PISCINE

Pensez à vous entraîner avant de terminer ce chapitre.

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P1C3)



## Organisez un projet en modules

Dans ce chapitre nous aborderons les notions suivantes :

* Création et import d'un module
* Création et import d'une librairie
* Lecture d'arguments passés en ligne de commande

Dans ce chapitre vous apprendrez à lire un fichier avec Python. Mais ce n'est pas tout ! Afin de bien commencer votre projet, vous apprendrez également à passer le nom de ce fichier en argument et à organiser votre projet de manière professionnelle.

Mais commençons par lire le CSV des membres !

Créez un dossier data dans le dossier courant. Déplacez-y les fichiers que vous avez téléchargés au premier chapitre de ce cours.

* current\_mps.csv ([le télécharger de nouveau](https://www.nosdeputes.fr/deputes/enmandat/csv))
* SyceronBrut.xml ([le télécharger de nouveau](http://data.assemblee-nationale.fr/travaux-parlementaires/debats))

### Organisation du projet

Les deux fichiers ont des fomats différents : CSV et XML. Afin de différencier les analyses, je vous propose de créer un script d'analyse par fichier :

* csv\_analysis.py : analyse du fichier current\_mps.csv
* xml\_analysis.py : analyse du fichier SyceronBrut.xml

Le fichier parite.py, qui existe déjà, servira à lancer les analyses.

En récapitulant, voici l'organisation actuelle :

* csv\_analysis.py
* parite.py
* xml\_analysis.py
* data
  + current\_mps.csv
  + SyceronBrut.xml

Parfait ! Ouvrez csv\_analysis.py et commençons à analyser le fichier CSV.

### Analyse d'un fichier CSV

Nous allons utiliser [la librairie standard  os](https://docs.python.org/3.6/library/os.html)  afin d'interagir avec le système. Elle vous permet d'utiliser des fonctionnalités qui dépendent du système d'exploitation, par exemple écrire ou retrouver le chemin d'un fichier.

Importez-la en haut de votre fichier :

import os

Puis créez une fonction main() , comme précédemment :

def main():

pass

if \_\_name\_\_ == "\_\_main\_\_":

main()

#### Lire un fichier

C'est très simple ! Utilisez la méthode open() qui prend deux paramètres :

* le chemin vers le fichier à lire
* le mode d'ouverture (optionnel) : r ouvre un fichier en lecture, w  vous donne également le droit de le modifier, etc. [En savoir plus](https://docs.python.org/3/library/functions.html#open)

Le chemin correspond à l'endroit d'où la commande est exécutée. Par exemple, si vous exécutez python csv\_analysis.py , le chemin sera csv\_analysis.py.

Pour l'instant, les scripts sont tous au même niveau. Mais que se passe-t-il si vous créez un dossier analysis qui contient analysis\_cvs.py?

Si vous exécutez la commande python analysis/analysis\_csv.py, le chemin sera analysis/analysis\_csv.py. Mais si vous exécutez la commande directement dans le dossier analysis, le chemin sera alors de nouveau analysis\_csv.py.

Comment indiquer le chemin vers un fichier à l'intérieur d'un script ? Encore une fois, Python vient à notre rescousse ! Vous pouvez utiliser la variable \_\_file\_\_ dont la valeur correspond au chemin vers un fichier. Puis nous pourrons retrouver très facilement le chemin du fichier que nous souhaitons utiliser. Voici comment faire :

csv\_analysis.py

import os

def launch\_analysis(data\_file):

directory = os.path.dirname(\_\_file\_\_) # we get the right path.

path\_to\_file = os.path.join(directory, "data", data\_file) # with this path, we go inside the folder `data` and get the file.

with open(path\_to\_file,"r") as f:

pass # read first line

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('current\_mps.csv')

Vous pouvez ensuite utiliser la méthode readline() pour lire la première ligne du fichier. Démonstration :

def launch\_analysis(data\_file):

directory = os.path.dirname(\_\_file\_\_) # we get the right path.

path\_to\_file = os.path.join(directory, "data", data\_file) # with this path, we go inside the folder `data` and get the file.

with open(path\_to\_file,"r") as f:

preview = f.readline()

print("Yeah! We managed to read the file. Here is a preview: {}".format(preview))

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('current\_mps.csv')

Voici ce qui apparait au lancement du fichier :

$ python csv\_analysis.py

id;nom;nom\_de\_famille;prenom;sexe;date\_naissance;lieu\_naissance;num\_deptmt;nom\_circo;num\_circo;mandat\_debut;groupe\_sigle;parti\_ratt\_financier;sites\_web;emails;adresses;anciens\_mandats;autres\_mandats;anciens\_autres\_mandats;profession;place\_en\_hemicycle;url\_an;id\_an;slug;url\_nosdeputes;url\_nosdeputes\_api;nb\_mandats;twitter;

Bravo ! Une première étape de franchie !

Vous pouvez à présent lire le fichier. Mais il serait plus intéressant de lancer l'analyse non pas à partir de ce fichier mais à partir de parite.py afin de pouvoir choisir, plus tard, le format de fichier adéquat.

"Transformons" le fichier csv\_analysis.py en module.

### Créer un module

Nous avons parlé des modules dans les quelques chapitres précédents mais nous ne les avons pas encore définis. Qu'est-ce qu'un module ?

Un **module** est un fichier contenant des instructions écrites en Python. Le nom du fichier est le nom du module. Par exemple, le fichier csv\_analysis.py devient le module csv\_analysis.

#### Importer un module

Un module peut ensuite être importé de plusieurs manières. Celle que je préconise, et que j'utilise, est la suivante :

# Import du module

import csv\_analysis as c\_an

# Exécution de la fonction main() du module csv\_analysis

c\_an.main()

Modifiez donc les fichiers :

parite.py

import csv\_analysis as c\_an

if \_\_name\_\_ == '\_\_main\_\_':

c\_an.launch\_analysis('current\_mps.csv')

csv\_analysis.py

import os

def launch\_analysis(data\_file):

directory = os.path.dirname(\_\_file\_\_) # we get the right path.

path\_to\_file = os.path.join(directory, "data", data\_file) # with this path, we go inside the folder `data` and get the file.

with open(path\_to\_file,"r") as f:

preview = f.readline()

print("Yeah! We managed to read the file. Here is a preview: {}".format(preview))

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('current\_mps.csv')

Vous avez réussi à lire le fichier csv. Bien joué !

Procédez de la même manière pour le fichier xml\_analysis.py même si nous le modifierons plus tard dans ce cours :

xml\_analysis.py.

import os

def launch\_analysis(data\_file):

directory = os.path.dirname(\_\_file\_\_) # we get the right path.

path\_to\_file = os.path.join(directory, "data", data\_file) # with this path, we go inside the folder `data` and get the file.

with open(path\_to\_file,"r") as f:

preview = f.readline()

print("Yeah! We managed to read the file. Here is a preview: {}".format(preview))

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('SyceronBrut.xml')

parite.py

#! /usr/bin/env python3

# coding: utf-8

import csv\_analysis as c\_an

import xml\_analysis as x\_an

def main():

c\_an.launch\_analysis('current\_mps.csv')

x\_an.launch\_analysis('SyceronBrut.xml')

if \_\_name\_\_ == "\_\_main\_\_":

main()

#### Importer une partie d'un module

Selon la taille d'un module, son import peut s'avérer très consommateur en mémoire. Lorsque vous travaillez avec des librairies conséquentes, telles que Pandas ou Numpy, il est d'usage de ne pas importer toute la librairie mais uniquement les objets dont vous aurez besoin.

Les imports personnalisés ont deux avantages majeurs :

* **Optimisation de la mémoire vive** nécessaire à l'exécution du script.
* **Amélioration de la sécurité** : plus vous importez de code, plus vous augmentez le risque que ce dernier soit frauduleux ou mal conçu.

csv\_analysis.py

# remplace import os

from os import path

Pour importer plusieurs objets, séparez-les par des virgules :

from os import path, name

Les objets importés sont à utiliser sans le nom de la librairie !

Import de toute une librairie :

# import de toute la librairie

import os

# On peut utiliser toutes les methodes

os.path()

os.name

Import de certains éléments :

# import de certains elements

from os import path, name

# On ne peut utiliser que path() et name

path()

name

### Créer une librairie

Simplifions notre projet en le transformant en librairie.

Qu'est-ce qu'une librairie ?

Une librairie, également appelée **paquet** (package en anglais), est un **ensemble de modules**. Cette organisation vous permet de les importer en utilisant une autre syntaxe en "point nomdumodule". Par exemple, un module qui s'appelle A.B désigne un sous-module appelé B dans un paquet s'appelant A.

Un paquet contient impérativement un fichier \_\_init\_\_.py par module. Un dossier sans ce fichier ne sera pas reconnu comme étant un module du paquet.

Afin d'illustrer ce propos, voici ce que vous pourriez écrire dans le projet si vous le transformez en librairie :

import analysis.xml as x\_an

import analysis.csv as c\_an

Comment faire ?

Une librairie minimale est organisée de la façon suivante :

* nom\_librairie
  + setup.py
  + nom\_sous\_module
    - sous\_module.py
    - \_\_init\_\_.py

Créez simplement un nouveau dossier, à la racine du projet, qui s'appelle analysis. A l'intérieur, déplacez les fichiers xml\_analysis.py et csv\_analysis.py. Renommez-les xml.py et csv.py. Ajoutez également un fichier vide nommé \_\_init\_\_.py dans analysis .

rappel de l'organisation :

* analysis
  + \_\_init\_\_.py
  + csv.py
  + xml.py
* data
  + current\_mps.csv
  + SyceronGlobal.xml
* parite.py
* setup.py

Mettez à jour parite.py et relancez le programme. Vous devriez voir apparaître une erreur. C'est normal car les chemins d'accès aux fichiers ont changé ! Modifiez le fichier csv.py :

def launch\_analysis(data\_file):

directory = os.path.dirname(os.path.dirname(\_\_file\_\_)) # we get the right path.

...

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('SyceronBrut.xml')

os.path.dirname(os.path.dirname(\_\_file\_\_)) : La méthode os.path.dirname() renvoie le répertoire contenant le fichier passé en argument. os.path.dirname(\_\_file\_\_) renvoie donc vers le dossier analysis. Mais nos données se trouvent dans le dossier data qui est dans le dossier parent ! Nous devons remonter d'un niveau pour nous situer à la racine de notre projet. Puis nous reconstituons le chemin avec la méthode join.

Tout fonctionne comme prévu ! Faites-en de même pour xml.py.

Pour en savoir plus sur la création d'une librairie, rien de mieux que de [**lire la documentation**](https://python-packaging.readthedocs.io/en/latest/minimal.html) !

### Lancer les fichiers en ligne de commande

A présent, concentrons-nous sur la résolution d'un souci : l'exécution de parite.py lance deux analyses (xml et csv). Ce serait plutôt intéressant de choisir le type d'analyse.

Je vous propose de fournir cette indication en tant qu'argument lorsque vous lancez le script en ligne de commande. Ainsi :

$ python parite.py --extension xml

Utilisez pour cela la librairie standard **Argparse**. Elle contient de nombreuses méthodes très utiles pour récupérer des arguments.

Importez la librairie : import argparse.

Nous allons l'utiliser en trois temps :

* Création d'un objet ArgumentParser()
* Lecture des arguments
* Renvoi des arguments

parite.py

import argparse

import analysis.csv as c\_an

import analysis.xml as x\_an

def parse\_arguments():

parser = argparse.ArgumentParser()

parser.add\_argument("-e", "--extension", help="""Type of file to analyse. Is it a CSV or an XML?""")

return parser.parse\_args()

if \_\_name\_\_ == '\_\_main\_\_':

args = parse\_arguments()

if args.extension == 'xml':

x\_an.launch\_analysis('SyceronBrut.xml')

elif args.extension == 'csv':

c\_an.launch\_analysis('current\_mps.csv')

Exécutez la commande :

$ python parite.py -e csv

Opening data file current\_mps.csv from directory 'data'

Yeah! We managed to read the file. Here is a preview: id;nom;nom\_de\_famille;prenom;sexe;date\_naissance;lieu\_naissance;num\_deptmt;nom\_circo;num\_circo;mandat\_debut;groupe\_sigle;parti\_ratt\_financier;sites\_web;emails;adresses;anciens\_mandats;autres\_mandats;anciens\_autres\_mandats;profession;place\_en\_hemicycle;url\_an;id\_an;slug;url\_nosdeputes;url\_nosdeputes\_api;nb\_mandats;twitter;

Vous pouvez utiliser la même technique pour passer le nom d'un fichier en argument ! Voici ce que cela pourrait donner :

#! /usr/bin/env python3

# coding: utf-8

import argparse

import analysis.csv as c\_an

import analysis.xml as x\_an

def parse\_arguments():

parser = argparse.ArgumentParser()

parser.add\_argument("-e", "--extension", help="""Type of file to analyse. Is it a CSV or an XML?""")

parser.add\_argument("-d","--datafile",help="""CSV file containing pieces of information about the members of parliament""")

return parser.parse\_args()

def main():

args = parse\_arguments()

datafile = args.datafile

if args.extension == 'xml':

x\_an.launch\_analysis(datafile)

elif args.extension == 'csv':

c\_an.launch\_analysis(datafile)

if \_\_name\_\_ == "\_\_main\_\_":

main()

A quoi sert le paramètre   help   de la méthode   add\_argument()  ?

Argparse  vous donne la possibilité d'afficher un message expliquant les différents arguments en ajoutant  -h  ou  --help.

Par exemple :

$ python parite.py --help

usage: parite.py [-h] [-d DATAFILE] [-e EXTENSION]

optional arguments:

-h, --help show this help message and exit

-d DATAFILE, --datafile DATAFILE

CSV file containing pieces of information about the

members of parliament

-e EXTENSION, --extension EXTENSION

Kind of file to analyse. Is it a CSV or an XML?

Découvrez toutes les fonctionnalités de la librairie [**dans la documentation**](https://docs.python.org/3/library/argparse.html).

Vous pouvez désormais lancer le programme en utilisant la commande suivante :  python parite.py -d current\_mps.csv -e csv .

$ python parite.py -d current\_mps.csv -e csv

Yeah! We managed to read the file. Here is a preview: id;nom;nom\_de\_famille;prenom;sexe;date\_naissance;lieu\_naissance;num\_deptmt;nom\_circo;num\_circo;mandat\_debut;groupe\_sigle;parti\_ratt\_financier;sites\_web;emails;adresses;anciens\_mandats;autres\_mandats;anciens\_autres\_mandats;profession;place\_en\_hemicycle;url\_an;id\_an;slug;url\_nosdeputes;url\_nosdeputes\_api;nb\_mandats;twitter;

Bravo ! Vous connaissez désormais la différence entre un module et une librairie. Vous savez également utiliser la librairie ArgParse pour ajouter des arguments. Mais que se passe-t-il si vous indiquez un fichier qui n'existe pas ? Le programme renverra une erreur. Voyons dans le chapitre suivant comment les gérer.

### A vous de jouer !

**Console de code**

##### À votre tour de coder

**Mettez immédiatement en application ce que vous avez appris.**  
Vous êtes libre de vous tromper et de réessayer autant que vous le souhaitez.

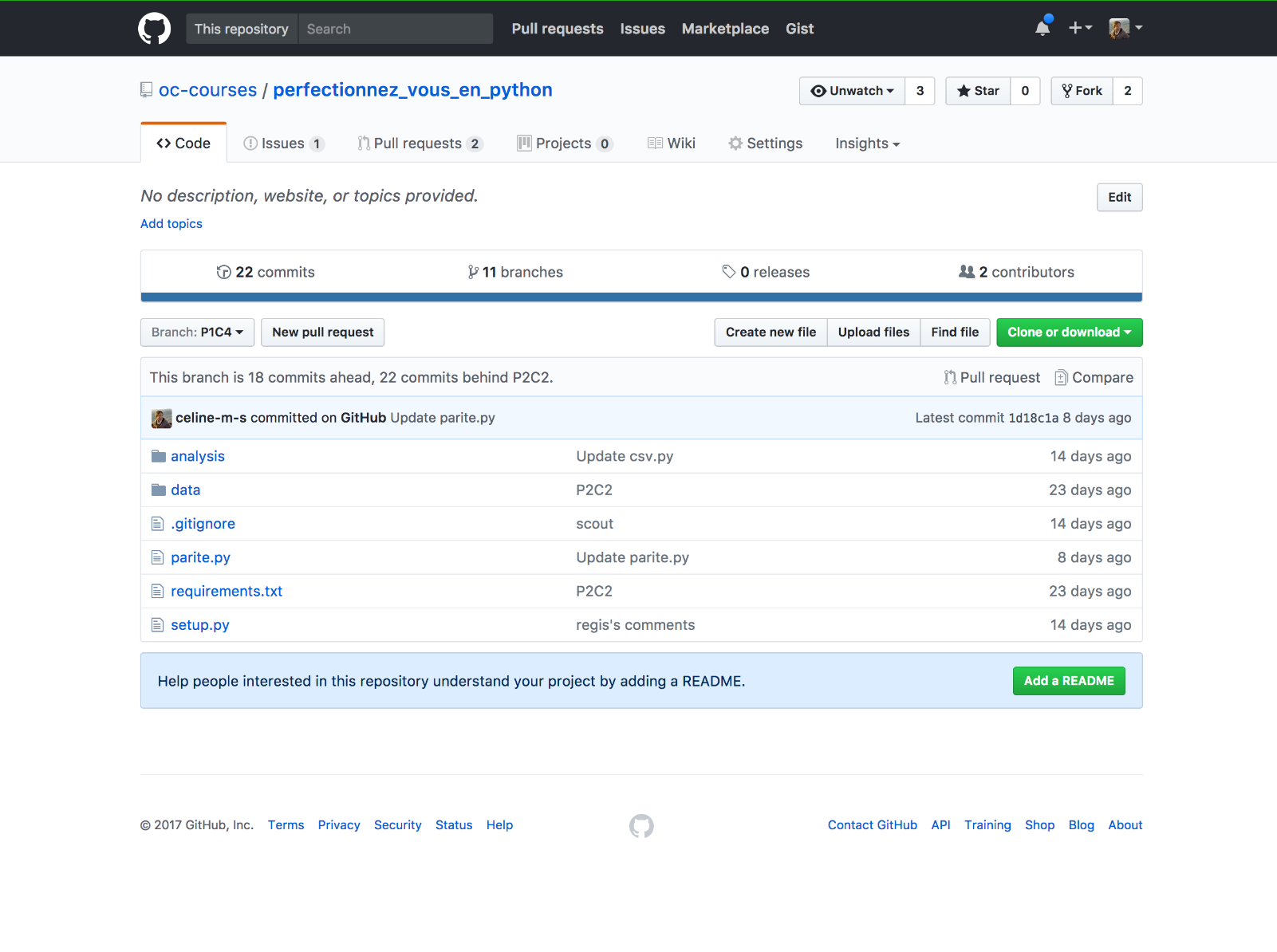
JE CODE MAINTENANT !

JE NE PEUX PAS, J'AI PISCINE

Pensez à vous entraîner avant de terminer ce chapitre.

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P1C4)



## Gérez les erreurs et les bogues

Dans ce chapitre nous aborderons les notions suivantes :

* Déboguer avec Python en utilisant Python Debugger
* Gérer les erreurs en utilisant les exceptions
* Comprendre les logs

Dans le chapitre précédent nous avons vu comment ajouter des arguments en ligne de commande à un programme et comment les utiliser. Mais que se passe-t-il si un argument nécessaire au bon fonctionnement du programme est absent ? Python affichera une erreur. Dans ce chapitre nous verrons comment gérer les erreurs que nous pouvons anticiper !

### Deboguer Python avec PDB

Jusqu'à maintenant nous avons écrit beaucoup de print() dans le programme pour vérifier quelles étaient les informations renvoyées. Et si vous pouviez "geler" le programme à un certain endroit, interagir avec lui puis le quitter ? En résumé : ouvrir une console interactive à l'endroit que vous le désirez ! Ce serait plus facile, non ?  😉  La librairie standard de Python intègre justement cette fonctionnalité-là ! Le module s'appelle PDB pour [Python Debugger](https://docs.python.org/3/library/pdb.html).

Importez Python Debugger:

parite.py

...

import pdb

Vous pouvez désormais utiliser [toutes les méthodes du module](https://docs.python.org/3/library/pdb.html). Entre toutes, j'aimerais surtout vous parler de set\_trace() qui vous permet d'entrer dans le debugger pendant que votre programme tourne.

Définissez un point d'arrêt (breakpoint en anglais) à l'endroit où vous souhaitez interrompre le programme :

parite.py

if \_\_name\_\_ == '\_\_main\_\_':

args = parse\_arguments()

import pdb; pdb.set\_trace()

if args.extension == 'xml':

x\_an.launch\_analysis(args.datafile)

elif args.extension == 'csv':

c\_an.launch\_analysis(args.datafile)

Au lancement du programme, celui-ci s'arrête pour vous laisser interagir :

$ python parite.py -e csv -d current\_mps.csv

> /chemin/vers/votre/projet/parite.py(16)<module>()

-> if args.extension == 'xml':

(Pdb) args

(Pdb) print(args)

Namespace(datafile='current\_mps.csv', extension='csv')

(Pdb) print(args.extension)

Notez que la console affiche désormais   (Pdb)   à gauche, indiquant que vous interagissez ici avec le débogueur.

Pour quitter PDB, tapez exit().

Pourquoi ne pas importer   *pdb*   en haut du fichier, comme les autres librairies ?

Afin de ne pas oublier de l'enlever lors de la mise en ligne ! D'ailleurs, pensez à supprimer l'appel à PDB avant de partager votre code à un de vos collègues.

PDB est un outil assez puissant et **couramment** utilisé. De très bons développeurs le maîtrisent sur le bout des doigts. Je vous invite fortement à découvrir toutes ses fonctionnalités dans la [**documentation officielle**](https://docs.python.org/3/library/pdb.html).

### Erreurs et exceptions

Nous avons vu comment déboguer plus efficacement, mais qu'en est-il des messages d'erreur ? Actuellement, si vous indiquez un fichier qui n'existe pas en argument, le programme s'interrompt.

$ python parite.py -e csv -d dontexist

Opening data file dontexist from directory 'data'

Traceback (most recent call last):

File "parite.py", line 19, in <module>

c\_an.launch\_analysis(args.datafile)

File "/path\_to\_project/analysis/csv.py", line 14, in launch\_analysis

with open(path\_to\_file,"r") as f:

FileNotFoundError: [Errno 2] No such file or directory: 'data/dontexist'

Il s'agit d'une erreur assez facilement prévisible. Vous allez dans cette partie :

* Afficher un message d'erreur personnalisé si le fichier n'est pas trouvé.
* Afficher un message d'erreur si aucun fichier n'est passé en argument.
* Afficher un message final indiquant que l'analyse a bien eu lieu.

#### Le bloc try / except

Lorsque nous savons qu'une erreur peut survenir, il est intéressant d'intégrer une instruction dans un bloc try / except. Que fait ce bloc ? Il **essaie** d'exécuter l'instruction mais, si une exception est levée, il exécute une autre instruction.

Voici un bloc très basique :

try:

0 / "la tête à toto"

except:

print("0 + 0")

Il est évidemment impossible de diviser un nombre par une chaîne de caractères. Python rencontrant une erreur, il passe dans le bloc except et exécute la commande print("0 + 0").

Vous pouvez également spécifier le type d'erreur attendue et afficher le message d'erreur :

try:

0 / "la tête à Toto"

except TypeError as e:

print("0 + 0 -- Message d'erreur plus sérieux : ", e)

Python écrit alors :

0 + 0 -- Message d'erreur plus sérieux : unsupported operand type(s) for /: 'int' and 'str'

De manière générale, il vaut mieux spécifier le type d'erreur à attraper. En effet, vous concevez votre code dans l'intention de le préparer à une erreur que vous avez décelée mais pas à toutes. Que se passe-t-il si une erreur inattendue survient ? Elle risque d'être confondue avec une autre.

Une autre fonctionnalité intéressante est la possibilité d'exécuter une instruction **dans tous les cas**, quoi qu'il se passe, en utilisant finally :

try:

0 / "la tête à Toto"

except TypeError as e:

print("0 + 0 -- Message d'erreur plus sérieux : ", e)

finally:

print("La tête à Toto")

Enfin, vous pouvez séparer la ligne qui risque de lever une erreur des autres instructions en utilisant else. Ce bloc sera déclenché uniquement si l'instruction contenue dans try se vérifie.

#### Récapitulatif

En résumé, voici un bloc try très complet :

try:

# instruction qui risque de lever une erreur

except Exception as e:

# Instruction exécutée quand l'exception Exception est lancée

except:

# Instruction exécutée dans le cas d'une autre erreur

else:

# Instruction exécutée si l'instruction dans try est vraie

finally:

# Instruction exécutée dans tous les cas

En d'autres termes :

try:

print('Je suis une instruction problématique')

except Exception as e:

print('Je suis déclenchée quand Exception est levée')

except:

print('Une autre erreur est survenue')

else:

print("Tout s'est bien passé ! Je passe donc à la suite")

finally:

print("Je suis déterminé : erreur ou pas, je m'affiche dans tous les cas")

#### Application concrète

Voici ce que cela donne dans le projet :

csv.py

try:

with open(path\_to\_file,"r") as f:

preview = f.readline()

print("Yeah! We managed to read the file. Here is a preview:")

print(preview)

except FileNotFoundError as e:

print("Ow :( The file was not found. Here is the original message of the exception :", e)

except:

print('Destination unknown')

Voici ce qui sera imprimé :

$ python parite.py -e csv -d dontexist

Opening data file dontexist from directory 'data'

Ow :( The file was not found. Here is the original message of the exception : [Errno 2] No such file or directory: 'data/dontexist'

Comment trouver le type d'exception qui sera levé ? Je ne les connais pas tous par coeur !

Je vous rassure : moi non plus ! En général vous lancez votre programme et vous notez le nom de l'erreur qui s'affiche. Dans l'exemple suivant, le type d'erreur est   FileNotFoundError:

$ python parite.py -e csv -d dontexist

Opening data file dontexist from directory 'data'

Traceback (most recent call last):

File "parite.py", line 19, in <module>

c\_an.launch\_analysis(args.datafile)

File "/path\_to\_project/analysis/csv.py", line 14, in launch\_analysis

with open(path\_to\_file,"r") as f:

FileNotFoundError: [Errno 2] No such file or directory: 'data/dontexist'

La documentation de Python intègre également une liste de toutes les exceptions génériques. [Lire la documentation](https://docs.python.org/3.6/library/exceptions.html#exceptions.TypeError).

#### Lever une exception

Il peut arriver que vous souhaitiez lever une exception **avant** que le problème ne se produise. Notre programme en est un parfait exemple.

Que se passe-t-il si aucun nom de fichier n'est entré ? Une erreur est certes levée mais pas à l'endroit espéré :

$ python parite.py -e csv

Traceback (most recent call last):

File "parite.py", line 19, in <module>

c\_an.launch\_analysis(args.datafile)

File "path\_to\_project/analysis/csv.py", line 7, in launch\_analysis

path\_to\_file = os.path.join("data", data\_file)

File "path\_to\_project/env/bin/../lib/python3.6/posixpath.py", line 92, in join

genericpath.\_check\_arg\_types('join', a, \*p)

File "path\_to\_project/env/bin/../lib/python3.6/genericpath.py", line 149, in \_check\_arg\_types

(funcname, s.\_\_class\_\_.\_\_name\_\_)) from None

TypeError: join() argument must be str or bytes, not 'NoneType'

Ici, Python indique qu'il utilise la méthode join() avec des arguments de type 'string' ou 'bytes' **uniquement**, et non 'NoneType'. Cette méthode se trouve dans le fichier csv.py à la ligne 7. Vous pourriez bien sûr englober cette instruction dans un bloc   try   mais, alors, il vous faudrait procéder de la même manière pour le fichier xml.py. Cela dupliquera votre code et ce n'est pas une bonne idée.

Comment faire ?

Prenez le problème à la racine et levez une exception si aucun fichier n'est passé en argument.

parite.py

if \_\_name\_\_ == '\_\_main\_\_':

args = parse\_arguments()

try:

datafile = args.datafile

if datafile == None:

raise Warning('You must indicate a datafile!')

else:

try:

if args.extension == 'xml':

x\_an.launch\_analysis(datafile)

elif args.extension == 'csv':

c\_an.launch\_analysis(datafile)

except FileNotFoundError as e:

print("Ow :( The file was not found. Here is the original message of the exception :", e)

finally:

print('#################### Analysis is over ######################')

except Warning as e:

print(e)

Bien souvent, nous utilisons raise dans un bloc except pour relancer l'erreur et déléguer la gestion de l'erreur à quelqu'un d'autre. Cela peut être utile pour, par exemple, afficher des messages complémentaires ou nettoyer des objets.

Comment choisir l'exception à lever ?

Une excellente pratique qui vous aidera à choisir l'exception à lever est d'écrire des [tests unitaires](https://openclassrooms.com/courses/testez-un-projet-python/decouvrez-les-tests), voire de coder en [Test-Driven Development](https://openclassrooms.com/courses/testez-un-projet-python/decouvrez-le-test-driven-development) ! Sinon, inspirez-vous de la documentation de Python qui est très bien faite. Regardez également [la hiérarchie des exceptions ici](https://docs.python.org/3.6/library/exceptions.html#exception-hierarchy).

### Logs

Cela commence à prendre forme ! Mais le projet contient beaucoup de print() qui ont des objectifs  différents. Parfois le message est crucial car il correspond à une exception qui a été levée. D'autres fois, il s'agit simplement d'un message informatif ('Analysis is over').

Comment instaurer des niveaux différents ? Utilisez la librairie logging qui permet justement d'afficher des messages en fonction de leur niveau d'importance.

#### Votre premier log

Logging fait partie de la librairie standard de Python. Tout comme PDB, vous n'avez pas besoin de l'installer.

Importez-la en haut de votre fichier :

import logging as lg

Puis remplacez le print() du bloc except Warning par l'instruction suivante :

lg.warning()

Voici désormais ce qui s'affiche :

$ python parite.py -e csv

WARNING:root:You must indicate a datafile!

#################### Analysis is over ######################

Le niveau du message est indiqué en majuscules sur la gauche, suivi du message.

#### Niveaux de log

Logging offre 5 niveaux de log différents :

* DEBUG : Informations détaillées dans le but d'en savoir plus sur l'exécution d'une instruction.
* INFO : Information sur le déroulement d'un programme.
* WARNING : Quelque chose d'inattendu s'est produit mais le programme continue de fonctionner.
* ERROR : A cause d'un problème important, le programme n'a pu réaliser une tâche.
* CRITICAL : Problème très sérieux qui a pu causer l'arrêt du programme.

Ces cinq niveaux de log sont indicatifs car les méthodes auront le même impact : elles afficheront un message dans la console. Simplement, le niveau ne sera pas le même. C'est à vous de choisir celui que vous désirez.

Par défaut, logging affiche dans la console les messages dont le niveau est supérieur à WARNING. Les logs dont le niveau est DEBUG ou INFO ne seront pas affichés.

Pour les afficher tout de même, indiquez en haut de votre script le niveau minimal attendu :

lg.basicConfig(level=lg.DEBUG)

Vous pouvez même aller plus loin en indiquant le niveau de debug souhaité en paramètre !

def parse\_arguments():

parser = argparse.ArgumentParser()

...

parser.add\_argument("-v", "--verbose", action='store\_true', help="""Make the application talk!""")

return parser.parse\_args()

if \_\_name\_\_ == '\_\_main\_\_':

args = parse\_arguments()

if args.verbose:

lg.basicConfig(level=lg.DEBUG)

...

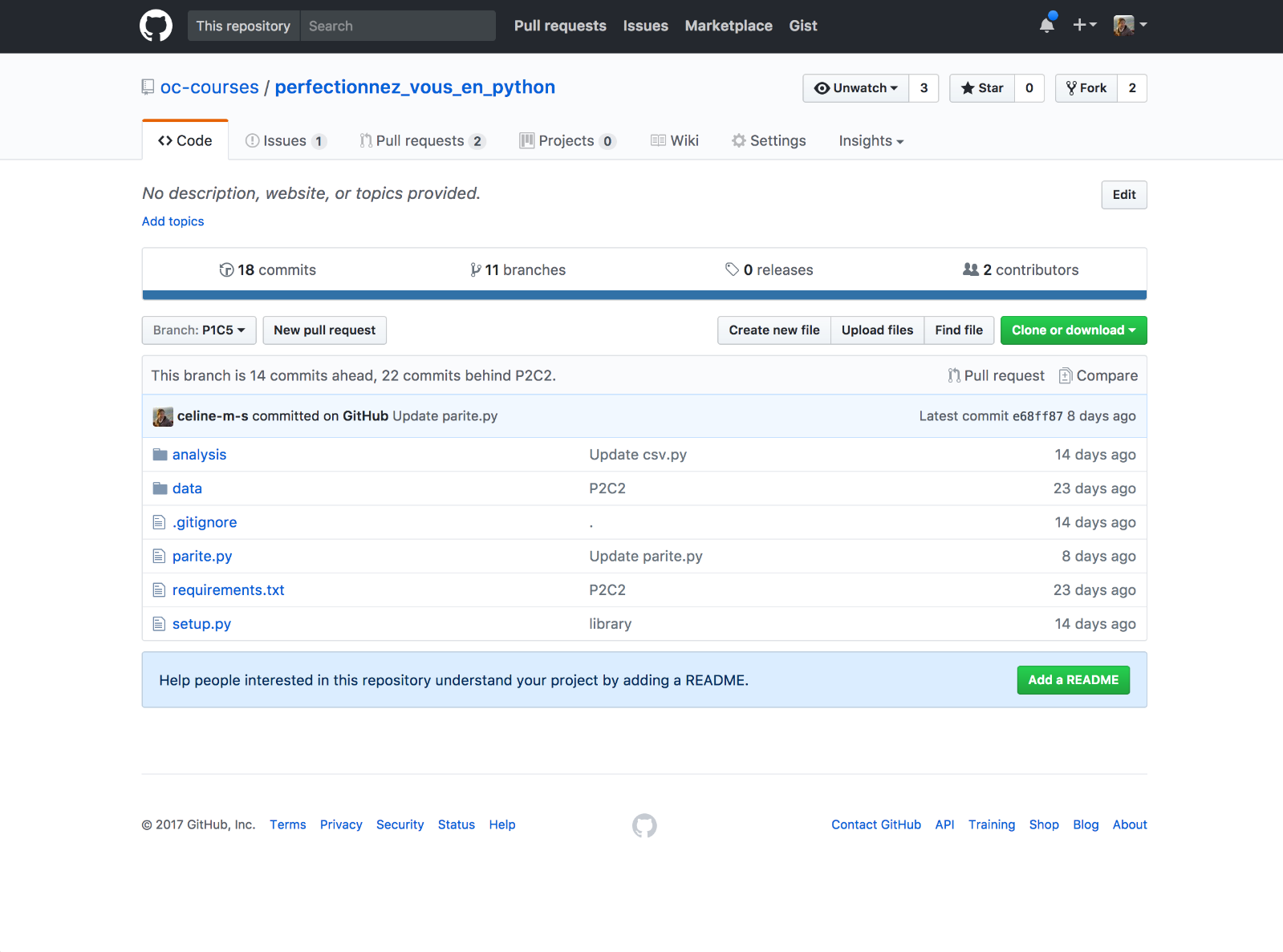
Je vous laisse remplacer les différents print() du programme par les niveaux de log qui vous paraissent les plus opportuns. Je vous retrouve dans la partie suivante pour créer notre premier graphique !

### A vous de jouer !

[Cliquez sur ce lien](https://www.codevolve.com/api/v1/publishable_key/2A9CAA3419124E3E8C3F5AFCE5306292?content_id=0c89979c-1a1e-47b8-a829-b6f63c23ef98)

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P1C5)



# Travaillez comme un·e pro !

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

### Question 1

**Les lignes suivantes sont à placer en haut d'un script Python :**

* + 
  + #! /usr/bin/env python3
  + # coding: utf-8
  + 
  + #! /usr/bin/env/path/to/my/project scala
  + # coding: latin-1
  + 
  + /usr/bin/env python3
  + coding: utf-8

### Question 2

**\_\_name\_\_  est une variable spéciale définie avant la lecture d'un script en Python.**

* + 

Vrai

* + 

Faux

### Question 3

**if \_\_name\_\_ == "\_\_main\_\_"   permet d'exécuter toujours les mêmes instructions au chargement d'un fichier, que ce dernier soit lancé via la commande  python  ou importé dans un autre script.**

* + 

Vrai

* + 

Faux

### Question 4

**Un environnement virtuel est :**

* + 

une autre galaxie. Vers l'infini et au-delà !

* + 

un dossier contenant certaines librairies et une version de Python que l'on peut activer et désactiver à l'envi.

* + 

un dossier contenant un lien symbolique vers la version de Python installée sur votre ordinateur.

### Question 5

**Activer un environnement virtuel dont le dossier s'appelle  env  s'effectue ainsi :**

* + 

$ deactivate env

* + 

$ activate env

* + 

$ source env/bin/activate

### Question 6

**Pour supprimer un environnement virtuel vous :**

* + 
    - Désactivez l'environnement.
    - Supprimez le dossier en question.
  + 

Désactivez l'environnement. Le dossier est supprimé automatiquement.

### Question 7

**PDB est :**

* + 

un débogger très puissant pour Python (Python Debugger)

* + 

un outil de base de données écrit en Python (Python Database)

* + 

un outil qui vous est inconnu.

### Question 8

**La commande suivante ouvrira une console interactive :**

* + 
  + p.r\_trace()
  + 
  + import pdb; pdb.set\_trace()
  + 
  + import pdb; p.get\_trace()

### Question 9

**Quel bloc affichera "Vous ne pouvez couper un fruit en deux" dans la console ?**

* + 
  + fruit = 2
  + try:
  + fruit / 2
  + except TypeError:
  + print('Vous ne pouvez couper un fruit en deux')
  + 
  + fruit = 'fraise'
  + try:
  + fruit / 2
  + except TypeError:
  + print('Vous ne pouvez couper un fruit en deux')

### Question 10

**Le niveau par défaut de la librairie   logging   est  WARNING .**

* + 

Vrai

* + 

Faux

## Créez votre premier graphique

Dans ce chapitre nous aborderons les notions suivantes :

* NumPy
* Pandas
* Matplotlib

Dans ce chapitre, nous allons aborder trois librairies très utilisées pour manipuler des données en python : NumPy, Pandas et Matplotlib.

Matplotlib, c'est pour les graphiques. Numpy, c'est pour manipuler des données, et plus précisément des séries de nombres. Enfin, la librairie Pandas introduit un objet très pratique appelé le Dataframe. Ce dernier s'apparente beaucoup à ce qu'il est possible de faire dans des logiciels de type tableur : des tableaux avec plein de nombres dedans !

Qu'allons-nous faire dans ce chapitre ?

Voici notre objectif : afficher un graphique de type "camembert". Bon... soyons un peu plus "pros" : le terme officiel est le Diagramme circulaire. Nous irons ici droit au but, sans étudier ces librairies en profondeur.

Allez, c'est parti !

### Installer les trois librairies

Commençons par installer les trois librairies Numpy, Pandas et Matplotlib.

Comme elles sont assez "lourdes", cela peut prendre un certain temps.

Dans votre console, saisissez successivement ces commandes :

pip install numpy

pip install matplotlib

pip install pandas

### La librairie NumPy

La force de la librairie Numpy, c'est son objet ndarray, qui permet de représenter des listes multidimensionnelles. Le terme est un peu pompeux, mais nous allons voir ensemble de quoi il s'agit.

Imaginons que nous souhaitions représenter des animaux. Disons... des pandas par exemple !

Nous pouvons caractériser un panda par sa taille ; ou plutôt, par ses tailles : disons par exemple, la taille de ses pattes, la longueur moyenne des poils de sa fourrure, la taille de sa queue, et le diamètre de son ventre.

En python, on peut "créer" un panda de cette manière, avec les 4 mesures que nous venons de choisir :

un\_panda = [100, 5, 20, 80]

Ici, notre panda a des pattes de 100cm, des poils de 5cm en moyenne, une queue de 20cm et un ventre de 80cm de diamètre.

Mais NumPy ça sert à quoi ici?

J'y arrive. Ici un panda peut très bien se représenter avec une simple liste Python basique. Mais cette liste, nous pouvons la transformer en un objet ndarray, comme ceci :

>>> import numpy as np

>>> un\_panda\_numpy = np.array(un\_panda)

>>> un\_panda\_numpy

array([100, 5, 20, 80])

Ce qui est pratique avec les objets ndarray, c'est qu'en mathématiques, nous avons souvent besoin de manipuler des listes de données, et donc de faire subir à ces listes toutes sortes d'opérations.

Par exemple, disons que notre panda mette au monde un bébé panda. Supposons (même si c'est faux) que ce bébé panda ait des dimensions exactement proportionnelles à sa celles de sa mère, c'est à dire que la taille de ses pattes soit égale à la taille des pattes de sa mère divisée par 2. Idem pour la longueur de ses poils : 2 fois moins que la longueur des poils de sa mère, etc.

Comment ferions nous sans NumPy ?

import numpy as np

un\_panda = [100, 5, 20, 80]

k = 2

# on initialise le bebe panda avec 0 partout

un\_bebe\_panda = [0,0,0,0]

# une boucle qui parcourt chacune des 4 mesures

for index in range(4):

# on divise chaque mesure par le coefficient k

un\_bebe\_panda[index] = un\_panda[index] / k

>>> un\_bebe\_panda

[50.0, 2.5, 10.0, 40.0]

C'est plutôt long et fastidieux non? Et bien c'est ici que NumPy vient à notre rescousse ! Regardez-moi un peu cela :

import numpy as np

un\_panda\_numpy = np.array([100, 5, 20, 80])

k = 2

un\_bebe\_panda\_numpy = un\_panda\_numpy / k

>>> un\_bebe\_panda\_numpy

array([ 50. , 2.5, 10. , 40. ])

Fabuleux n'est-ce pas ? En une ligne de code, nous avons divisé par 2 tous les éléments de la liste !

On appelle cela une opération vectorielle. Ici, notre panda est représenté par un vecteur : ce vecteur, c'est simplement la liste de nombres qui le caractérise.

Peut-être que le terme de vecteur, au sens mathématique, vous évoque vaguement quelque chose. Si si, vous savez, c'est ce que l'on représentait par des flèches, soit en cours de physique pour représenter des forces, soit en cours de géométrie pour représenter des translations de figures ! Sur un graphique en 2 dimensions dessiné sur la page de votre cahier de géométrie, un vecteur avait 2 coordonnées : une sur l'axe des abcisses et une sur l'axe des ordonnées. Pour multiplier un vecteur par 3 par exemple (c'est-à-dire multiplier la longueur de la "flèche" par 3, tout en gardant la même direction), il suffisait de multiplier chacune des 2 coordonnées par 3 : c'est exactement ce que l'on vient de faire avec notre vecteur un\_panda\_numpy.

Et si je veux représenter plusieurs pandas, je crée une liste de ndarray?

C'est une bonne idée (bravo!). Il est effectivement possible de faire cela :

famille\_panda = [

np.array([100, 5 , 20, 80]), # maman panda

np.array([50 , 2.5, 10, 40]), # bébé panda

np.array([110, 6 , 22, 80]), # papa panda

]

Notez l'apparition du papa panda (Bonjour monsieur!).

Mais nous pouvons faire mieux :

famille\_panda = [

[100, 5 , 20, 80], # maman panda

[50 , 2.5, 10, 40], # bébé panda

[110, 6 , 22, 80], # papa panda

]

famille\_panda\_numpy = np.array(famille\_panda)

>>> famille\_panda\_numpy

array([[ 100. , 5. , 20. , 80. ],

[ 50. , 2.5, 10. , 40. ],

[ 110. , 6. , 22. , 80. ]])

Ici, nous avons créé une liste de pandas, c'est à dire une liste de listes ; des listes imbriquées en fait ! En langage Numpy, on dit que le rang de famille\_panda\_numpy est de 2, car nous avons 2 niveaux d'imbrication.

Quel avantage?

Par exemple, supposons que je veuille obtenir la taille des pattes (situé en position 0 dans nos vecteurs) du panda situé en position 2 dans ma liste (c'est-à-dire papa panda). Numpy offre la possibilité de le faire élégamment :

>>> famille\_panda[2] # papa panda

[110, 6, 22, 80]

>>> famille\_panda[2][0] # taille des pattes de papa panda

110

>>> famille\_panda\_numpy[2, 0] # taille des pattes de papa panda, à la manière numpy

110.0

Et si je veux connaître les tailles des pattes de toute ma famille panda ?

Facile ! Il suffit de supprimer le 2 (qui correspondait au papa panda), et de le remplacer par le caractère:, signifiant que je veux TOUS les pandas !

>>> pattes = famille\_panda\_numpy[:, 0]

>>> pattes

array([ 100., 50., 110.])

Voilà ! La taille des pattes de la maman est de 100cm, celle du bébé 50cm, et celle du papa 110cm.

Un petit dernier pour la route ?

Imaginons que je veuille calculer la somme des tailles de ces pattes (ce qui n'a aucun sens, je vous l'accorde) :

>>> pattes.sum()

260.0

### La librairie Pandas

#### Tout est une histoire de tableau !

Au fait, la manière dont j'ai écrit la variablefamille\_panda:

famille\_panda = [

[100, 5 , 20, 80],

[50 , 2.5, 10, 40],

[110, 6 , 22, 80],

]

... ça ressemble un peu à un tableau avec des lignes et des colonnes non ? Vous ne trouvez pas que c'est très similaire à cela?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **pattes** | **poil** | **queue** | **ventre** |
| **maman\_panda** | 100 | 5 | 20 | 80 |
| **bébé\_panda** | 50 | 2.5 | 10 | 40 |
| **papa\_panda** | 110 | 6 | 22 | 80 |

(La bonne réponse est : "Oui, c'est très similaire")

Dans ce cas, pourquoi ne pas utiliser une librairie Python qui manipulerait des tableaux comme celui-ci ? Cette librairie existe, et elle s'appelle Pandas.

C'est parti, représentons nos pandas avec Pandas !

>>> import pandas as pd

>>> famille\_panda\_df = pd.DataFrame(famille\_panda)

>>> famille\_panda\_df

0 1 2 3

0 100 5.0 20 80

1 50 2.5 10 40

2 110 6.0 22 80

L'objet qui permet de représenter des "tableaux" est l'objet DataFrame. Pour instancier un tel objet (et pour lui donner de la donnée), on lui transmet une liste de rang 2, c'est à dire une liste de listes.

Nous pouvons même faire mieux, en indiquant les noms de colonnes et les noms des lignes. Et encore mieux : la librairie Pandas se base en grande partie sur la librairie Numpy dans son fonctionnement interne. Comme pandas connaît intimement Numpy, on peut très bien transmettre à l'objet DataFrame de la donnée au format ndarray :

>>> import pandas as pd

>>> famille\_panda\_df = pd.DataFrame(famille\_panda\_numpy,

... index = ['maman', 'bebe', 'papa'],

... columns = ['pattes', 'poil', 'queue', 'ventre'])

>>> famille\_panda\_df

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

bebe 50.0 2.5 10.0 40.0

papa 110.0 6.0 22.0 80.0

Le nom des lignes est appelé index. Un index peut être une chaîne de caractères (un label) ou un nombre entier. Quand aucun index n'est spécifié à la création du dataframe, il est initialisé par défaut avec une suite continue d'entiers commençant par 0.

Vous allez encore me demander quel est l'intérêt de cette librairie n'est-ce pas ?

En fait, l'objet DataFrame est très similaire à certains concepts que l'on trouve en dehors du cadre du langage Python. Il est similaire :

* aux tables des bases de données relationnelles (type MySQL, PostgreSQL, etc.) que l'on manipule grâce au [langage SQL](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql)
* à l'objet dataframe sur lequel se base tout le [langage R](https://openclassrooms.com/courses/initiez-vous-au-langage-de-statistiques-r), langage destiné aux statisticiens et aux [Data Analysts](https://openclassrooms.com/paths/data-analyst)

Du coup, si vous connaissez déjà le langage SQL ou le langage R, vous aurez beaucoup de facilité à utiliser le DataFrame de Pandas !

#### Quelques fonctionnalités des DataFrames

Voici quelques petites fonctionnalités des Dataframes.

Tout d'abord, accédons à la colonne ventre de notre table. Il y a deux syntaxes possibles, qui renvoient exactement le même résultat :

>>> famille\_panda\_df.ventre

maman 80.0

bebe 40.0

papa 80.0

Name: ventre, dtype: float64

>>> famille\_panda\_df["ventre"]

maman 80.0

bebe 40.0

papa 80.0

Name: ventre, dtype: float64

L'objet que renvoie famille\_panda\_df["ventre"] est de type pandas.Series, que nous ne connaissons pas encore. Pour obtenir les valeurs de la colonne ventre au format numpy.ndarray, il faut saisir famille\_panda\_df["ventre"].values

 Parcourons maintenant tous les pandas un à un, grâce à la méthode iterrows qui renvoie (à chaque itération de la boucle for) un tuple dont le premier élément est l'index de la ligne, et le second le contenu de la ligne en question :

for ligne in famille\_panda\_df.iterrows():

index\_ligne = ligne[0]

contenu\_ligne = ligne[1]

print("Voici le panda %s :" % index\_ligne)

print(contenu\_ligne)

print("--------------------")

Voici le panda maman :

pattes 100.0

poil 5.0

queue 20.0

ventre 80.0

Name: maman, dtype: float64

--------------------

Voici le panda bebe :

pattes 50.0

poil 2.5

queue 10.0

ventre 40.0

Name: bebe, dtype: float64

--------------------

Voici le panda papa :

pattes 110.0

poil 6.0

queue 22.0

ventre 80.0

Name: papa, dtype: float64

--------------------

Accédons maintenant au papa panda : d'abord par sa position (position 2), puis par son nom "papa" :

>>> famille\_panda\_df.iloc[2] # Avec iloc(), indexation positionnelle

pattes 110.0

poil 6.0

queue 22.0

ventre 80.0

Name: papa, dtype: float64

>>> famille\_panda\_df.loc["papa"] # Avec loc(), indexation par label

pattes 110.0

poil 6.0

queue 22.0

ventre 80.0

Name: papa, dtype: float64

Déterminons les pandas dont le diamètre du ventre est de 80cm :

>>> famille\_panda\_df["ventre"] == 80

maman True

bebe False

papa True

Name: ventre, dtype: bool

Il s'agit ici aussi d'une opération vectorielle. En effet, on teste **chaque** élément de la colonne "ventre" en demandant s'il est égal à 80. On obtient la réponse suivante : "maman" : True, "bebe" : False, "papa" : True.

Le résultat de cette opération est très pratique pour filtrer des lignes ! Par exemple, pour sélectionner **uniquement** les pandas dont le ventre est de 80cm, il suffit d'intégrer ce précédent résultat en tant que masque, comme ceci :

>>> masque = famille\_panda\_df["ventre"] == 80

>>> pandas\_80 = famille\_panda\_df[masque]

>>>

>>> # On écrit plus souvent cela de cette manière :

>>> # pandas\_80 = famille\_panda\_df[famille\_panda\_df["ventre"] == 80]

>>>

>>> pandas\_80

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

papa 110.0 6.0 22.0 80.0

Qu'est-ce qu'un masque ?

Lorsque vous mettez un masque sur votre visage, celui-ci cache certaines parties de votre tête, mais en laisse visible d'autres : vos yeux, votre nez et votre bouche. Ici, c'est un peu la même chose : comme nous souhaitons ne garder que certaines lignes du dataframe, et en cacher d'autres, nous appliquons un masque. Un masque est ici une liste de variables booléennes (TrueouFalse) dans laquelle chaque élément est associé à une ligne du dataframe. Si cet élément estTrue, cela signfie que nous souhaitons garder la ligne en question. Si nous souhaitons ne pas garder la ligne, cet élément estFalse.

Pour inverser le masque, il suffit d'utiliser l'opérateur~, et nous sélectionnons les pandas qui n'ont pas un ventre de 80cm :

>>> famille\_panda\_df[~masque]

pattes poil queue ventre

bebe 50.0 2.5 10.0 40.0

Maintenant, ajoutons des lignes à notre dataframe. Il y a plusieurs méthodes pour cela, mais voyons ici la plus simple : assemblons ensemble deux dataframes.

>>> quelques\_pandas = pd.DataFrame([[105,4,19,80],[100,5,20,80]], # deux nouveaux pandas

... columns = famille\_panda\_df.columns)

... # même colonnes que famille\_panda\_df

>>> tous\_les\_pandas = famille\_panda\_df.append(quelques\_pandas)

>>> tous\_les\_pandas

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

bebe 50.0 2.5 10.0 40.0

papa 110.0 6.0 22.0 80.0

0 105.0 4.0 19.0 80.0

1 100.0 5.0 20.0 80.0

Dans le dataframe tous\_les\_pandas, il y a des doublons. En effet, le premier panda (maman) et le dernier panda (dont l'index est 1) ont exactement les mêmes mesures. Si nous souhaitions dédoublonner, nous ferions ceci :

>>> tous\_les\_pandas.drop\_duplicates()

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

bebe 50.0 2.5 10.0 40.0

papa 110.0 6.0 22.0 80.0

0 105.0 4.0 19.0 80.0

Le résultat de cette ligne ne modifie pas le dataframe  tous\_les\_pandas  . Il ne fait que renvoyer un autre dataframe. Dans l'état actuel, ce nouveau dataframe est juste affiché et n'est pas enregistré en mémoire : il sera donc perdu. Pour le garder, on peut soit l'enregistrer dans une nouvelle variable, ...

pandas\_uniques = tous\_les\_pandas.drop\_duplicates()

 ... soit remplacer la variable tous\_les\_pandas par ce nouveau dataframe dédoublonné :

tous\_les\_pandas = tous\_les\_pandas.drop\_duplicates()

Allez, encore un petit effort : voici quelques autres fonctionnalités en vrac, que nous utiliserons dans le projet :

# accéder aux noms des colonnes

famille\_panda\_df.columns

# créer une nouvelle colonne, composée de chaînes de caractères

famille\_panda\_df["sexe"] = ["f", "f", "m"]

# la maman et le bébé sont des femelles, le papa est un mâle

# obtenir le nombre de lignes

len(famille\_panda\_df)

>>> # obtenir les valeurs distinctes d'une colonne :

>>> # pour la colonne ventre, il y a deux valeurs distinctes : 40 et 80

>>> famille\_panda\_df.ventre.unique()

array([ 80., 40.])

#### Lire un fichier CSV avec Pandas

Revenons à nos moutons : notre analyse des députés de l'assemblée nationale, et laissons de côté notre famille panda.

Un fichier [CSV](https://fr.wikipedia.org/wiki/Comma-separated_values) (comma separated values), c'est un fichier permettant de représenter des données sous forme de tableau. D'ailleurs, si vous utilisez un logiciel de type tableur, il y a fort à parier qu'il puisse lire et exporter au format CSV !

Vous me voyez venir ? La librairie Pandas est justement spécialisée dans la manipulation de tableaux ! Lire un fichier CSV avec Pandas est donc un vrai jeu d'enfant : il ne suffit que d'une ligne pour créer un dataframe à partir d'un CSV :

mps = pd.read\_csv("data/current\_mps.csv", sep=";")

C'est fait ! La variable mps (signifiant "Members of Parliament" en anglais) contient maintenant un dataframe dont chaque ligne correspond à un député. Petite particularité ici : dans notre fichier CSV, les valeurs sont séparées par le caractère;. Par défaut, [pd.read\_csv](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html) attend des valeurs séparées par une virgule. Nous sommes donc obligés de spécifier sep=";".

#### Revenons à notre projet

Maintenant que nous avons lu notre fichier CSV, nous allons créer une classe python qui permet de manipuler et d'analyser des groupes de députés. Nous appellerons cette classe ensemble de députés, ou en anglais SetOfParliamentMembers. Par exemple, un parti politique représenté à l'assemblée nationale pourra être considéré comme un ensemble de députés.

Chacun de ces ensembles aura un nom, qui sera stocké dans la variable name de chaque objet SetOfParliamentMembers. La liste de députés y sera également stockée sous forme de dataframe, dans la variable nommée dataframe. Nous aurons la possibilité d'ajouter des députés soit à partir d'un fichier CSV via la méthode data\_from\_csv, ou soit via un dataframe déjà existant via la méthode data\_from\_dataframe.

Nous créerons aussi deux méthodes, l'une appelée display\_chart, qui affichera un diagramme circulaire (camembert pour les intimes) représentant la proportion de femmes et d'hommes dans le groupe de députés, et une autre méthode split\_by\_political\_party, qui séparera le groupe de députés selon leur parti politique en renvoyant un objet de type dictionnaire.

Nous définirons également la fonction launch\_analysis, destinée à lancer nos analyses futures selon nos envies. Pour le moment, elle affichera le diagramme de la proportion femme/homme pour toute l'assemblée nationale, et affichera (si nous lui spécifions l'argument by\_party = True) ces diagrammes pour chacun des partis politiques.

Voici le code correspondant :

import os

import pandas as pd

class SetOfParliamentMembers:

def \_\_init\_\_(self, name):

self.name = name

def data\_from\_csv(self, csv\_file):

self.dataframe = pd.read\_csv(csv\_file, sep=";")

def data\_from\_dataframe(self, dataframe):

self.dataframe = dataframe

def display\_chart(self):

# à venir, patience !

pass

def split\_by\_political\_party(self):

result = {}

data = self.dataframe

all\_parties = data["parti\_ratt\_financier"].dropna().unique()

for party in all\_parties:

data\_subset = data[data.parti\_ratt\_financier == party]

subset = SetOfParliamentMembers('MPs from party "{}"'.format(party))

subset.data\_from\_dataframe(data\_subset)

result[party] = subset

return result

def launch\_analysis(data\_file, by\_party = False):

sopm = SetOfParliamentMembers("All MPs")

sopm.data\_from\_csv(os.path.join("data",data\_file))

sopm.display\_chart()

if by\_party:

for party, s in sopm.split\_by\_political\_party().items():

s.display\_chart()

Une petite explication à propos de la méthode split\_by\_political\_party s'impose n'est-ce-pas ?

Nous souhaitons ici renvoyer un dictionnaire (que l'on appelle ici result) dans lequel les clés sont les noms des partis politiques, et les valeurs sont des objets de type SetOfParliamentMembers contenant les députés du parti en question.

Il nous faut tout d'abord établir la liste des partis politiques. Celle-ci est présente dans la colonne parti\_ratt\_financier du dataframe. Cependant, certains députés ne sont rattachés à aucun parti. Il arrive donc d'avoir des valeurs nulles dans cette colonne. Nous ne nous attarderons pas ici sur les valeurs nulles dans les dataframes, mais retenez juste que pour supprimer celles-ci, nous employons la méthode dropna(). Ensuite, comme nous ne souhaitons **que** les valeurs **distinctes** de la colonne parti\_ratt\_financier, nous appelons la méthode unique().

Ensuite, dans la boucle for, nous créons à chaque itération un nouveau dataframe nommé subset, obtenu par filtrage du dataframe data. La suite de la construction du dictionnaire result ne devrait pas

vous poser de problème.

### La librairie Matplotlib

Je sens qu'à ce stade, vous mourrez d'envie d'afficher un graphique !

Si votre système d'exploitation est MacOS, il est probable que vous soyez obligé d'intégrer ces 2 lignes de code en haut de votre programme afin d'utiliser correctement Matplotlib :

import matplotlib

matplotlib.use('TkAgg')

Parmi les types de graphiques que propose la librairie matplolib, celui qui nécessite le moins de ligne de code tout en étant très simple à comprendre est probablement l'histogramme. Voici sans plus attendre un exemple dans lequel on cherche à représenter graphiquement les âges d'un groupe de personnes :

import matplotlib.pyplot as plt

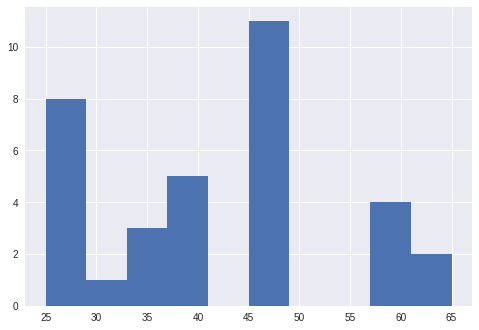
ages = [25, 65, 26, 26, 46, 37, 36, 36, 28, 28, 57, 37, 48, 48, 37, 28, 60,

25, 65, 46, 26, 46, 37, 36, 37, 29, 58, 47, 47, 48, 48, 47, 48, 60]

fig, ax = plt.subplots()

ax.hist(ages)

plt.show()



L'histogramme des âges

Avec Matplotlib, nous créons des graphiques grâce à l'objet matplotlib.pyplot, que l'on nomme souvent plt comme ici.

Lorsque vous exécuterez ce code, une nouvelle fenêtre s'ouvrira pour afficher l'histogramme. Avec Matplotlib, le terme figure correspond à **une** fenêtre.

A l'intérieur d'une fenêtre, il est possible d'afficher plusieurs graphiques. Ici, nous ne souhaitons qu'un seul graphique. Créer un unique graphique par fenêtre, c'est justement le comportement par défaut de la méthode plt.subplot(). Cette dernière renvoie un tuple contenant deux objets, que l'on nomme communément fig et ax. Nous ne verrons pas la signification de fig dans ce chapitre. Quant à ax, il représente notre unique graphique.

À cet unique graphique, nous spécifions que nous souhaitons afficher un histogramme, grâce à la méthode ax.hist() à laquelle nous transmettons la liste des âges à afficher.

A ce stade, le graphique n'est pas encore visible, car il était toujours en cours de création. Pour afficher la fenêtre, nous appelons plt.show().

Vous aimez ? Alors continuons sans plus tarder avec le **diagramme circulaire** tant attendu ! Représentons un groupe de personnes composé de 24 femmes et 18 hommes :

import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.pie([24, 18],

labels = ["Femmes", "Hommes"])

plt.show()

Nous pouvons améliorer quelques petites choses. Tout d'abord, ajoutons un titre à notre graphique, grâce à le méthode plt.title().

Peut-être avez-vous remarqué, en fonction de la dimension de votre fenêtre, le camembert est peut-être aplati. C'est parce que les axes du graphique (l'axe vertical et l'axe horizontal) s'adaptent à la dimension de la fenêtre. Ici, pour que le camembert soit un cercle et non pas une ellipse, nous devons spécifier que l'axe vertical doit être de même taille que l'axe horizontal, pour éviter un aplatissement. Cette opération est rendue possible en appelant ax.axis("equal").

Petite amélioration d'ordre cosmétique. Il existe une ligne un peu magique, qui modifie les propriétés graphiques par défaut de matplotlib, afin de rendre les graphiques un peu plus beaux. Cette ligne consiste juste en l'importation de la librairie seaborn.

Enfin, il est possible d'afficher pour chaque part du camembert la proportion représentée. Ici, on a  57.1% de femmes. Nous le faisons en transmettant le paramètre autopct="%1.1f pourcents" à la méthode ax.pie().

La syntaxe transmise en argumentautopct="%1.1f pourcents"semble un peu complexe, mais elle signifie que nous souhaitons afficher la proportion avec une précision d'un chiffre après la virgule. Vous connaissez peut-être cette manière de formater une chaîne de caractères ; si c'est le cas, sachez que l'affichage final de la proportion 51.3% est équivalent à exécuter cette ligne de code :"%1.1f pourcents" % 51.3

import matplotlib.pyplot as plt

import seaborn as sns

fig, ax = plt.subplots()

ax.axis("equal")

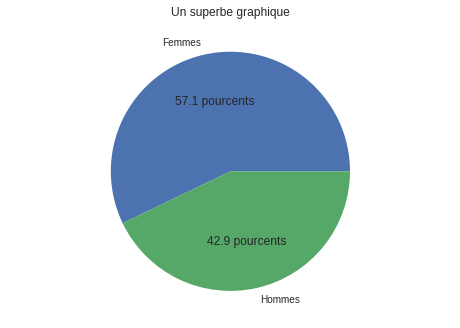
ax.pie([24, 18],

labels = ["Femmes", "Hommes"],

autopct="%1.1f pourcents")

plt.title("Un superbe graphique")

plt.show()



Un beau diagramme circulaire avec matplotlib

Félicitation ! Vous avez maintenant toutes les connaissances requises pour écrire la méthode display\_chart de notre objet SetOfParliamentMembers.

L'objectif ici est de calculer la proportion de femmes et d'hommes à partir de la colonne sexe du dataframe contenant les députés.

Commençons donc par créer deux dataframes, l'un contenant les députés femmes : female\_mps et l'autre contenant les députés hommes male\_mps. Ensuite, stockons le nombre de députés de chaque sexe dans la variable counts. Il ne nous reste plus qu'à diviser ces deux nombres par le nombre total de députés nb\_mps, grâce à une opération vectorielle, et nous obtenons la variable proportions.

Voici le code :

class SetOfParliamentMembers:

#[...]

def display\_chart(self):

data = self.dataframe

female\_mps = data[data.sexe == "F"]

male\_mps = data[data.sexe == "H"]

counts = [len(female\_mps), len(male\_mps)]

counts = np.array(counts)

nb\_mps = counts.sum()

proportions = counts / nb\_mps

labels = ["Female ({})".format(counts[0]), "Male ({})".format(counts[1])]

fig, ax = plt.subplots()

ax.axis("equal")

ax.pie(

proportions,

labels=labels,

autopct="%1.1f pourcents"

)

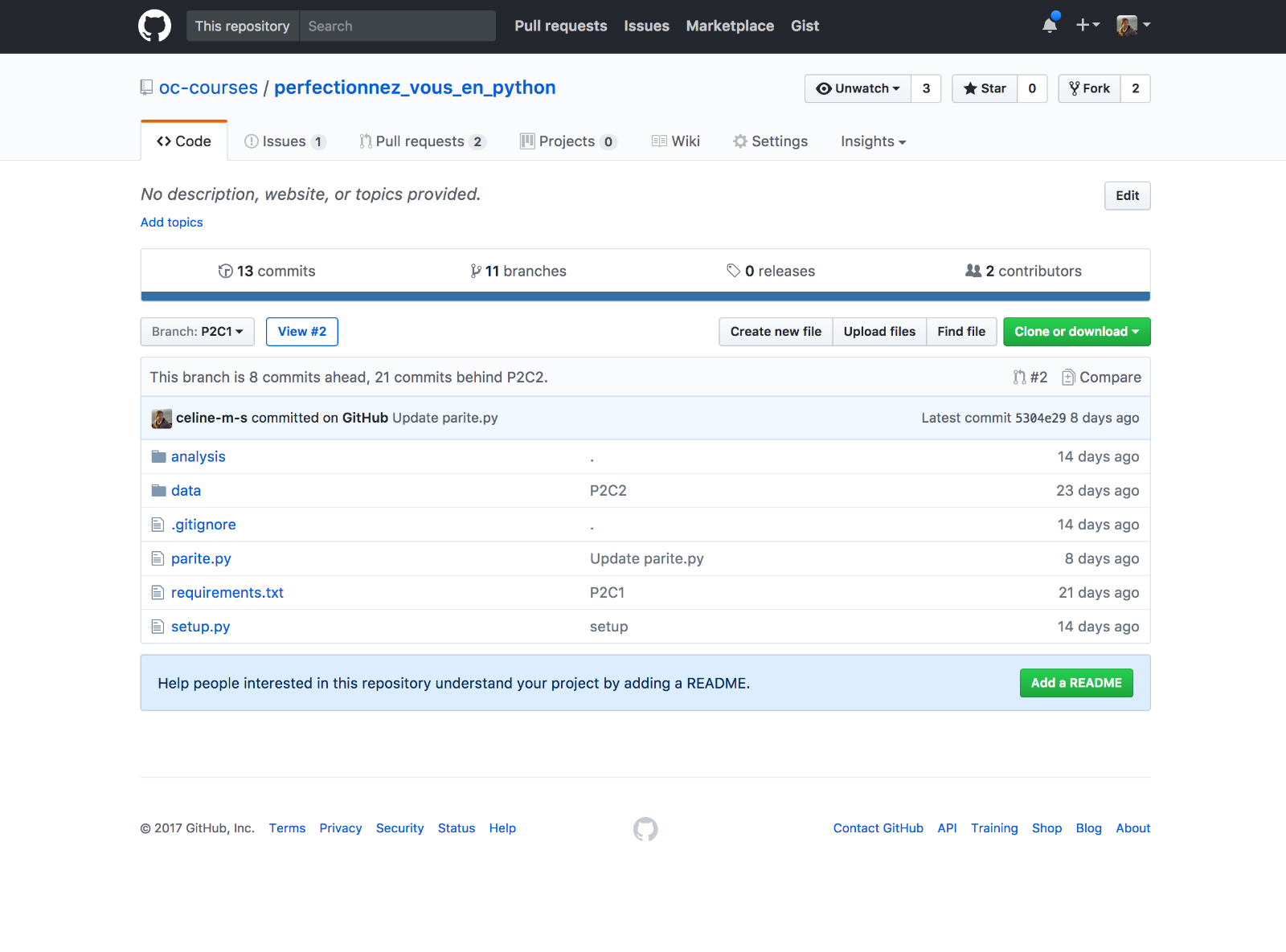
plt.title("{} ({} MPs)".format(self.name, nb\_mps))

plt.show()

Voilà, vous savez maintenant utiliser les librairies les plus populaires de manipulation de données mathématiques en python, et surtout, vous savez afficher un graphique !

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P2C1)



## Découvrez les méthodes spéciales

Dans ce chapitre nous aborderons les notions suivantes :

* Les méthodes spéciales
* Surcharger des méthodes spéciales

Dans ce chapitre vous ajouterez une nouvelle fonctionnalité au programme ! Vous serez en mesure d'afficher le nombre de parlementaires que contient votre fichier CSV en ajoutant l'argument  --info . Comme ceci :

$ python parite.py -d current\_mps.csv -e csv --info

Allons-y !

### Calculer le nombre de parlementaires

Calculer le nombre de parlementaires est assez simple. Puisque dataframe est un tableau, vous pouvez utiliser la fonction len() pour en calculer le nombre d'items. Comme ceci :

csv.py

class SetOfParliamentMember:

...

def total\_mps(self):

return len(self.dataframe)

def launch\_analysis(data\_file, info=False):

sopm = SetOfParliamentMember("All MPs")

...

if info:

print(sopm.total\_mps())

parite.py

def parse\_arguments():

...

parser.add\_argument("-i","--info", action='store\_true', help="""information about

the file""")

...

if \_\_name\_\_ == '\_\_main\_\_':

...

else:

if args.extension == 'xml':

x\_an.launch\_analysis(datafile)

elif args.extension == 'csv':

c\_an.launch\_analysis(datafile, args.info)

finally:

lg.info('#################### Analysis is over ######################')

Vous pourriez bien sûr vous contenter de créer une nouvelle méthode et de l'imprimer... mais non. Vous utiliserez les méthodes spéciales car, comme leur nom l'indique, elles sont [assez magiques](https://www.youtube.com/watch?v=NLxN0wpFoP8) ! Comme vous ! 🙂

### Les méthodes spéciales

Chaque méthode du langage peut être vue comme un ensemble de pièces Légo qui forment un composant important : une échelle, le toit d'une maison, un mur en briques, ... Python met à votre disposition ces méthodes afin que vous puissiez aller plus vite en ne réinventant pas la roue. Après tout, nous avons très souvent besoin de calculer le nombre d'éléments que contient une liste par exemple. Mais que faire lorsque vous voulez **adapter** cette méthode à vos besoins ? Et si vous pouviez la **changer** ?

😱  Es-tu en train de me dire que je peux recoder Python si je le souhaite ?

D'une certaine manière... Vous pouvez modifier très simplement un ensemble de méthodes du langage en utilisant les fameuses **méthodes spéciales**.

**A retenir :** L'interpréteur de Python invoque des méthodes spéciales pour réaliser des opérations basiques sur des objets. Les méthodes spéciales commencent et finissent toujours par deux tirets du bas (\_\_).

Par exemple, l'instruction  obj[key]  est réalisée grâce à la méthode spéciale \_\_getitem\_\_(). Les méthodes spéciales permettent aux autres objets d'implémenter ou d'interagir avec des constructions basiques du langage telles que les itérateurs, les collections, les accesseurs, l'appel de fonctions, la création et la destruction d'objets, ...

#### Comment utiliser les méthodes spéciales ?

Que se passe-t-il lorsque vous utilisez la fonction len() ? L'interpréteur Python utilise la méthode spéciale \_\_len\_\_() pour renvoyer le résultat attendu. C'est cette méthode que vous pouvez modifier en la réécrivant dans votre script. Démonstration :

>>> class Hack:

... def \_\_len\_\_(self):

... print('Wow, I just hacked Python!')

... return 5

...

>>> a = Hack()

>>> print(len(a))

Wow, I just hacked Python!

5

Assez génial, n'est-ce pas ?!

Avant tout, il faut savoir que les méthodes spéciales sont pensées pour être exécutées par l'interpréteur Python et non par vous. D'ailleurs, vous aurez certainement remarqué que je n'ai pas écrit a.\_\_len\_\_() dans l'exemple ci-dessus. J'ai écrit len(a) et l'interpréteur a trouvé la méthode \_\_len\_\_() tout seul. Vous n'écrivez pas objet.\_\_len\_\_() ! Vous écrivez len(objet) et si objet est une instance d'une classe que vous avez définie, alors Python appelle la méthode \_\_len\_\_() que vous avez implémentée.

Dans tous les cas, vous devez définir ces méthodes spéciales dans une classe ! 😉

#### La méthode spéciale \_\_repr\_\_()

Quelle méthode spéciale peut-on utiliser pour remplacer la méthode total\_mps()? Je vous propose d'utiliser \_\_repr\_\_() qui est appelée lorsque vous souhaitez imprimer les informations d'un objet. Comme ceci :

>>> class Babar:

... pass

...

>>> a = Babar()

>>> print(a) # appelle la méthode \_\_repr\_\_ sur l'instance de la classe Babar

<\_\_main\_\_.Babar object at 0x109fa9198>

En définissant la méthode spéciale   \_\_repr()\_\_ , voici ce qui s'affiche :

>>> class Babar:

... def \_\_repr\_\_(self):

... return "Je suis un éléphant très spécial"

...

>>> a = Babar()

>>> print(a)

Je suis un éléphant très spécial

Voici comment vous pouvez l'utiliser dans le projet :

csv.py

class SetOfParliamentMember:

...

def \_\_repr\_\_(self):

return "SetOfParliamentMember: {} members".format(len(self.dataframe))

def launch\_analysis(data\_file, info=False):

sopm = SetOfParliamentMember("All MPs")

sopm.data\_from\_csv(os.path.join("data", data\_file))

sopm.display\_chart()

if info:

print(sopm)

Si vous lancez le script avec l'argument --info, vous verrez dans la console que le fichier contient 570 parlementaires.

### Les autres méthodes spéciales

Python intègre de nombreuses autres méthodes spéciales qui vous faciliteront la vie !

|  |  |
| --- | --- |
| **Catégorie** | **Nom des méthodes spéciales** |
| Représentation | \_\_repr\_\_, \_\_str\_\_, \_\_format\_\_, \_\_bytes\_\_ |
| Conversion en nombre | \_\_abs\_\_, \_\_bool\_\_, \_\_complex\_\_, \_\_int\_\_, \_\_float\_\_, \_\_hash\_\_, \_\_index\_\_ |
| Collections | \_\_len\_\_, \_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_, \_\_contains\_\_ |
| Itérateurs | \_\_iter\_\_, \_\_reversed\_\_, \_\_next\_\_ |
| Création et destruction d'instances | \_ \_new\_\_, \_\_init\_\_, \_\_del\_\_ |
| Gestion des attributs | \_\_getattr\_\_, \_\_getattribute\_\_, \_\_setattr\_\_, \_\_delattr\_\_, \_\_dir\_\_ |
| Comparaison | \_\_lt\_\_<, \_\_le\_\_<=, \_\_eq\_\_==, \_\_ne\_\_!=, \_\_gt\_\_>=, \_\_ge\_\_>= |
| Opérateurs arithmétiques | \_\_add\_\_+, \_\_sub\_\_-, \_\_mul\_\_\*, \_\_truediv\_\_/, \_\_floordiv\_\_//, \_\_mod\_\_%, \_\_round\_\_ |

Nous avons ajouté de nombreux exemples dans le code source de ce chapitre afin que vous puissiez voir les méthodes spéciales suivantes en action :

* \_\_len\_\_
* \_\_contains\_\_
* \_\_getitem\_\_
* \_\_add\_\_
* \_\_radd\_\_
* \_\_lt\_\_
* \_\_gt\_\_
* \_\_getattr\_\_
* \_\_setattr\_\_

Elles nous permettent de réaliser les opérations suivantes :

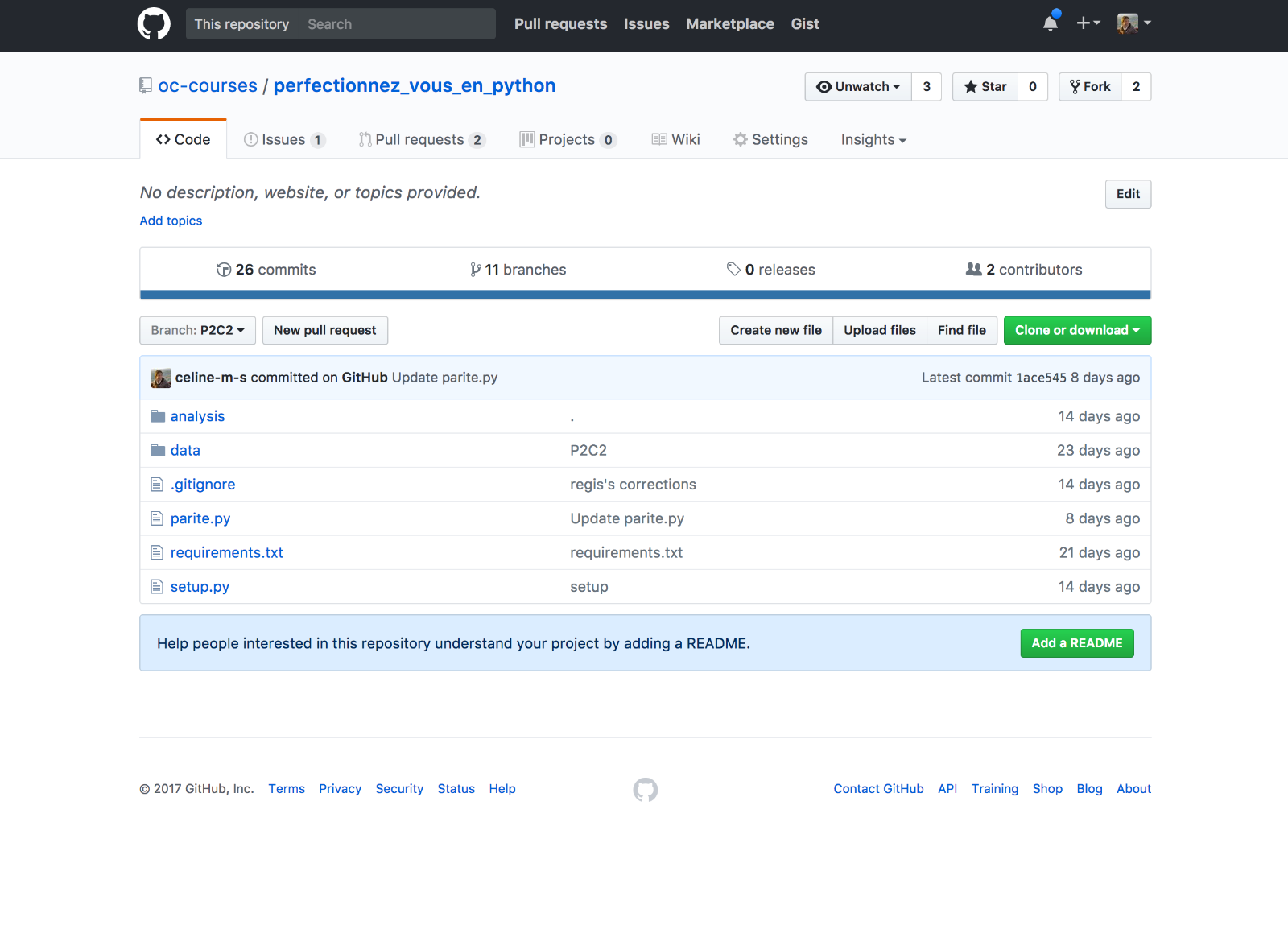
* Afficher un graphique par parti politique avec l'argument --byparty
* Chercher un député par son nom avec l'argument --searchname
* Trouver un député grâce à un index avec l'argument --index
* Afficher les n plus grands groupes politiques avec l'argument --groupfirst n

### A vous de jouer !

[Cliquez sur ce lien](https://www.codevolve.com/api/v1/publishable_key/2A9CAA3419124E3E8C3F5AFCE5306292?content_id=91990850-7264-4fa8-9f26-aab09597d94c)

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P2C2)



Allez plus loin en lisant [**cet article de Dive into Python**](http://www.diveintopython3.net/special-method-names.html).

## Apprivoisez les décorateurs

Dans ce chapitre nous aborderons les notions suivantes :

* Les décorateurs
* Créer un décorateur

Dans ce chapitre nous parlons des décorateurs. Mais qu'est-ce que cela veut dire ?

Vous en avez déjà probablement croisé ! Un décorateur ressemble à ceci : @decorator. Si vous avez suivi mon précédent cours sur la [Programmation Orientée Objet](https://openclassrooms.com/courses/decouvrez-la-programmation-orientee-objet-avec-python), vous reconnaissez aisément @classmethod et @property dans cette syntaxe. Nous couvrons tout cela dans ce chapitre !

### Les décorateurs 101

C'est bien joli, mais à quoi ça sert ?

Un décorateur permet de modifier le comportement d'une fonction. Il commence par un @ suivi de lettres ou de chiffres. Il se place sur la ligne précédant la définition d'une fonction. Comme ceci :

@decorator

def karadoc():

print("Le gras, c'est la vie.")

Python intègre de nombreux décorateurs standards mais vous pouvez également en définir vous-même. Pourquoi ? Car un décorateur est simplement une fonction qui prend en paramètre une fonction et renvoie une (autre) fonction.

def decorator(func):

def inner():

# do something with the function

return func()

return inner

Dans cet exemple très générique, nous voyons que le décorateur définit une nouvelle fonction qui va renvoyer la fonction décorée. En quoi est-ce intéressant ? Cela permet de réaliser des opérations avec la fonction décorée : la modifier, l'étendre, ...

En voici une illustration :

def e\_t(func):

def inner():

print("Maison... Téléphone... Maison...")

return func()

return inner

@e\_t

def gertie():

print("Je lui ai appris à parler ! Ecoute !")

Cela permet de faire le dialogue complet...

def e\_t(func):

def inner():

print("Maison... Téléphone... Maison...")

return func()

return inner

@e\_t

def gertie():

print("Je lui ai appris à parler ! Ecoute !")

@e\_t

def elliott():

print("Il veut rentrer chez lui !")

>>> def e\_t(func):

... def inner():

... print("Maison... Téléphone... Maison...")

... return func()

... return inner

...

>>> @e\_t

... def gertie():

... print("Je lui ai appris à parler ! Ecoute !")

...

Maison... Téléphone... Maison...

Je lui ai appris à parler ! Ecoute !

>>>

... @e\_t

... def elliott():

... print("Il veut rentrer chez lui !")

...

Maison... Téléphone... Maison...

Il veut rentrer chez lui !

### Sous le capot

Il n'y a pas de magie. Ce @ peut vous sembler abscond mais il s'agit d'un raccourci vers une autre syntaxe que vous connaissez très bien.

def e\_t(func):

def inner():

print("Maison... Téléphone... Maison...")

return func()

return inner

@e\_t

def gertie():

print("Je lui ai appris à parler ! Ecoute !")

Est la même chose que :

def e\_t(func):

def inner():

print("Maison... Téléphone... Maison...")

return func()

return inner

def gertie():

print("Je lui ai appris à parler ! Ecoute !")

gertie = e\_t(gertie)

Un décorateur est du **sucre syntaxique**. Autrement dit, il s'agit simplement d'une notation différente. Mais elle est si utile que vous la verrez souvent !

Vous pourriez également créer un décorateur qui affiche les erreurs en les loguant grâce à la librairie   logging. Il existe de nombreuses utilisations différentes des décorateurs !

### Créer un décorateur plus générique

Prenons un autre exemple concret que certains d'entre vous connaissent déjà.

class CoffeeMachine():

water\_level = 100

def \_start\_machine(self):

# Start the machine

if self.water\_level > 20:

return True

else:

print("Please add water!")

return False

def \_\_boil\_water(self):

return "boiling..."

def make\_coffee(self):

# Make a new coffee!

if self.\_start\_machine():

self.water\_level -= 20

print(self.\_\_boil\_water())

print("Coffee is ready!")

machine = CoffeeMachine()

for i in range(0, 5):

machine.make\_coffee()

print("Make Coffee: Public", machine.make\_coffee())

print("Start Machine: Protected", machine.\_start\_machine())

print("Boil Water: Private", machine.\_CoffeeMachine\_\_boil\_water())

Les trois print() en fin de code ont pour objectif d'afficher un message et d'exécuter les méthodes. Vous pourriez créer un décorateur qui aurait pour simple mission d'imprimer le nom de la méthode avant son exécution, comme ceci :

def name(func):

def inner(\*args, \*\*kwargs):

print('Running this method:', func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

return inner

class CoffeeMachine():

water\_level = 100

@name

def \_start\_machine(self):

# Start the machine

if self.water\_level > 20:

return True

else:

print("Please add water!")

return False

@name

def \_\_boil\_water(self):

return "boiling..."

@name

def make\_coffee(self):

# Make a new coffee!

if self.\_start\_machine():

self.water\_level -= 20

print(self.\_\_boil\_water())

print("Coffee is ready!")

machine = CoffeeMachine()

for i in range(0, 5):

machine.make\_coffee()

machine.make\_coffee()

machine.\_start\_machine()

machine.\_CoffeeMachine\_\_boil\_water()

Que se passe-t-il ?

* def name(func): Définition d'une nouvelle fonction qui prend la fonction décorée en paramètre.
* def inner(\*args, \*\*kwargs): A l'intérieur, définition d'une nouvelle fonction.
* print('Running this method:', func.\_\_name\_\_) : Première instruction : imprimer un message ainsi que le nom de la fonction.
* return func(\*args, \*\*kwargs) : Deuxième instruction : exécuter la fonction décorée.

Pourquoi passer en paramètres *\*args* et *\*\*kwargs*?

Un décorateur doit être le plus générique possible. Il ne connaît pas la signature de l'objet à décorer, autrement dit il ne connaît pas son comportement. Est-ce que l'objet décoré prend un ou plusieurs paramètres ? Le décorateur ne le sait pas et n'a, d'ailleurs, aucun intérêt à le savoir. Sa seule responsabilité est de réaliser une action avec cet objet et de le renvoyer si nécessaire. C'est pourquoi nous indiquons   \*args   et   \*\*kwargs.

Voici ce qui est alors affiché dans la console :

>>> machine = CoffeeMachine()

>>> for i in range(0, 5):

... machine.make\_coffee()

...

Running this method: make\_coffee

Running this method: \_start\_machine

Running this method: \_\_boil\_water

boiling...

Coffee is ready!

Running this method: make\_coffee

Running this method: \_start\_machine

Running this method: \_\_boil\_water

boiling...

Coffee is ready!

Running this method: make\_coffee

Running this method: \_start\_machine

Running this method: \_\_boil\_water

boiling...

Coffee is ready!

Running this method: make\_coffee

Running this method: \_start\_machine

Running this method: \_\_boil\_water

boiling...

Coffee is ready!

Running this method: make\_coffee

Running this method: \_start\_machine

Please add water!

>>> machine.make\_coffee()

Running this method: make\_coffee

Running this method: \_start\_machine

Please add water!

>>> machine.\_start\_machine()

Running this method: \_start\_machine

Please add water!

False

>>> machine.\_CoffeeMachine\_\_boil\_water()

Running this method: \_\_boil\_water

'boiling...'

>>>

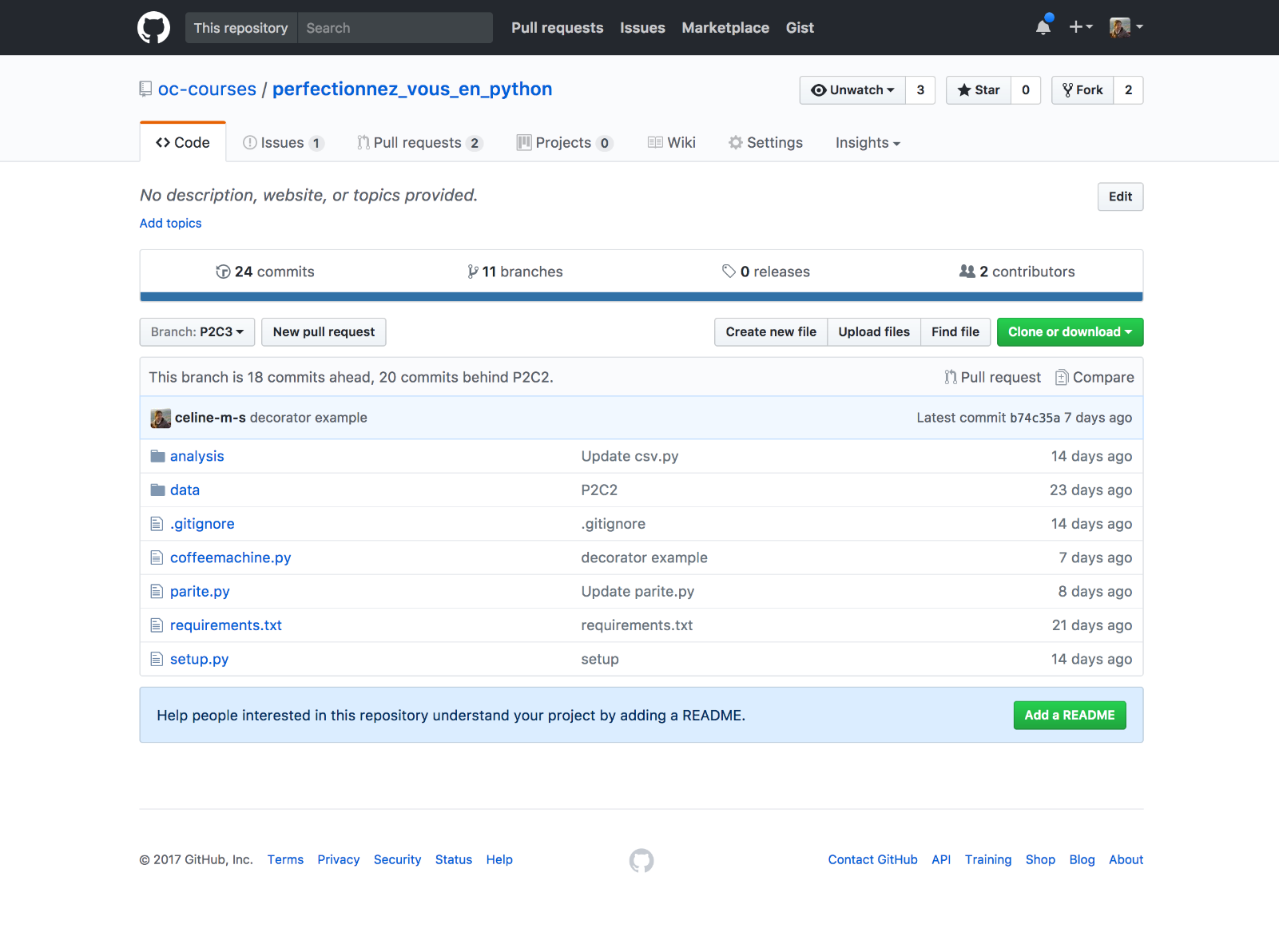
La librairie standard de Python intègre de nombreux décorateurs. Découvrez-les [**dans cette page**](https://wiki.python.org/moin/PythonDecoratorLibrary).

### A vous de jouer !

[Cliquez sur ce lien](https://www.codevolve.com/api/v1/publishable_key/2A9CAA3419124E3E8C3F5AFCE5306292?content_id=befdab67-fc61-4e63-9656-39ce3717f7f1)

### Code du chapitre

Nous avons intégré, dans le code de ce chapitre, de nombreux exemples de décorateurs. [Découvrez-les tout de suite !](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P2C3)



## Utilisez des expressions régulières

Dans ce chapitre nous aborderons les notions suivantes :

* Les expressions régulières
* Syntaxe des expressions régulières
* Utiliser une méthode avec une expression régulière

Dans ce chapitre vous découvrirez le pouvoir extraordinaire des expressions régulières ainsi que leur utilisation ! D'ailleurs, vous les utiliserez pour améliorer le projet.

Actuellement, notre projet ne reconnait pas le format de fichier que nous passons en argument. Ce serait assez agréable de ne plus spécifier le format et que notre programme le détermine par lui-même, n'est-ce pas ? C'est un parfait cas d'utilisation d'une expression régulière.

### Les expressions régulières

Nous utilisons les expressions régulières, sans nous en rendre compte, dans le langage courant. Est-ce que la discussion suivante vous semble familière ?

- (téléphone) Pouvez-vous me donner votre nom s'il vous plait ?  
- Oui bien sûr. Céline.  
- Comment ? Cécile ?  
- Non non. Cé***l***ine. Avec un '***l'***.  
- Ah. Heu... Cécilia ?  
- Non. Céline. La première lettre est ***C*** comme dans Crabe, puis vous avez 4 lettres : ***é*** comme Echelle, ***l*** comme Litre, ***i*** comme Internet, ***n*** comme Noisette. Et ça finit par e comme de l'Eau.  
- Ah oui. Céline ! Comme Céline Dion.  
- (soupir)

Que s'est-il passé ? Afin de me faire comprendre, j'ai épelé mon prénom en prenant des exemples que mon interlocuteur connaissait déjà. A chaque fois, j'ai porté son attention sur les parties similaires en occultant les autres.

Une expression régulière réalise la même opération mais **de manière plus abstraite**. Il s'agit d'une **chaîne de caractères**, souvent appelée motif, **qui décrit selon une syntaxe particulière un ensemble de chaînes de caractères possible**.

Par exemple, voici la représentation de Céline : ^C.{4}e$

[Spoiler] Vous découvrirez tout à l'heure que cette expression régulière est volontairement incomplète !

#### Syntaxe

Les expressions régulières sont très puissantes lorsque vous cherchez des variantes d'un même motif. Par exemple, si nous souhaitions trouver dans une liste de prénoms tous ceux qui sont de la forme suivante : Céline, Cécile, Célia, Cécilia. Comment faire ?

Regardons en premier les éléments en commun :

* Les prénoms commencent par un 'C',
* La seconde lettre est un 'é',
* La quatrième lettre est un 'i'.

Nous avons, par extension, le modèle suivant :

* Ils commencent par un 'C',
* La seconde lettre est un 'é',
* La troisième lettre est variable,
* La quatrième lettre est un 'i',
* Les lettres suivantes sont des n, e, l, a ou i. Il peut y en avoir plusieurs.

Comment représenter cela par une expression régulière ? De la manière suivante :

* Ils commencent par un 'C' : ^C
* La seconde lettre est un 'é' : é=> ^Cé
* La troisième lettre est variable : .=> ^Cé.
* La quatrième lettre est un 'i' : i=> ^Cé.i
* Les lettres suivantes sont des n, e, l, a ou i. Il peut y en avoir plusieurs : [nelai]+=> ^Cé.i[nelai]+

L'expression régulière est alors la suivante : ^Cé.i[nelai]+

Cette syntaxe m'est complètement inconnue. Pourquoi commencer par un accent circonflexe alors qu'il n'est présent nulle part ?

Très bonne question ! La syntaxe des expressions régulières est assez particulière. Commençons par les signes les plus connus !

**Représentation :**

* . : représente tout caractère (chiffre, lettre ou symbole)
* ^ : représente le début d'une ligne. Tout signe le suivant sera considéré comme étant le début d'une ligne.
* $ : représente la fin d'une ligne. Tout signe le précédant sera considéré comme étant à la fin de la ligne.
* \w : représente un caractère venant d'un mot (a-z, A-Z, 0-9 ou tout caractère unicode comme 爱)
* \d : représente un chiffre (0 - 9)

**Occurrences :**

* \* : aucune ou plusieurs occurrences
* + : au moins une occurrence
* [] : au moins un des caractères présents entre crochets

#### Sélectionner un groupe : les parenthèses

Une dernière utilisation assez commune concerne les parenthèses. Elles permettent de sélectionner uniquement ce qui est entre parenthèses dans un ensemble plus large.

Par exemple :

'^C(.{4})e$' # renvoie les quatre caractères après le 'C' du début et avant le 'e' de la fin.

'(^C)é.i[nelai]+' # renvoie le c du début

Cela permet de trouver les occurrences d'une expression régulière en ne renvoyant pas l'intégralité de la chaîne de caractères mais uniquement une partie.

#### Entrainez-vous !

Parfait ! Entrainez-vous quelques instants sur [Pythex](http://pythex.org/) à trouver l'expression régulière qui correspond aux modèles suivants :

* Numéro de téléphone (10 chiffres entre 0 et 10, commençant par 06 ou 07). [Découvrez une proposition](https://bit.ly/2ru96Y9).
* Adresse e-mail (caractères alphanumériques ou chiffres, @, caractères alphanumériques ou chiffres, point et caractères alphanumériques ou chiffres).

A vous de jouer !

Lisez bien l'aide-mémoire situé sous le formulaire !

L'expression régulière correspondant à une adresse e-mail est un cas extrêmement complexe mais c'est toujours bien de s'entraîner ! Si le sujet vous intéresse, découvrez les différentes réponses sur [**cette question posée sur StackOverFlow**](https://stackoverflow.com/questions/201323/using-a-regular-expression-to-validate-an-email-address). Regardez également [**cette expression régulière**](http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html) (attention les yeux !).

### La librairie standard re

Les expressions régulières ne sont pas spécifiques à Python. Par exemple, vous pouvez les utiliser pour filtrer le trafic dans Google Analytics ! Néanmoins, il faut utiliser une librairie qui vous donnera des méthodes permettant d'utiliser les expressions régulières.

Voyons tout de suite comment utiliser les expressions régulières dans notre projet. Nous allons réaliser cela en plusieurs étapes :

* import de la librairie re
* écriture de l'expression régulière
* utilisation de la méthode appropriée

#### Import de la librairie re

Commencez par importer [la librairie standard re](https://docs.python.org/2/library/re.html) (pour Regular Expression, Expression Régulière en anglais).

parite.py

import re

#### Ecriture de l'expression régulière

Nous allons chercher ce qui ressemble à un format de fichier dans la chaîne de caractères renvoyée par args.datafile. Pour rappel, voici notre code actuel :

def parse\_arguments():

parser = argparse.ArgumentParser()

parser.add\_argument("-d","--datafile",help="""CSV file containing pieces of information about the members of parliament""")

...

return parser.parse\_args()

if \_\_name\_\_ == '\_\_main\_\_':

args = parse\_arguments()

try:

datafile = args.datafile

if datafile == None:

raise Warning('You must indicate a datafile!')

except Warning as e:

lg.warning(e)

else:

if args.extension == 'xml':

x\_an.launch\_analysis(datafile)

elif args.extension == 'csv':

c\_an.launch\_analysis(datafile, args.byparty, args.info, args.displaynames,

args.searchname, args.index, args.groupfirst)

finally:

lg.info('#################### Analysis is over ######################')

Un format de fichier suit le modèle suivant :

* un ou plusieurs caractères (dont des espaces)
* un point
* trois caractères (nous nous limiterons à trois caractères pour le moment puisque nous ne prenons en charge que le csv et le xml)

Je vous laisse essayer avant de lire ma proposition !!

La voici : ^.+\.(\D{3})$

#### Utilisation de la méthode appropriée

La librairie offre de nombreuses méthodes, dont les deux suivantes :

* search : trouve dans une chaîne de caractères la première occurrence correspond à l'expression régulière,
* findall : trouve dans une chaîne de caractères toutes les occurrences correspondant à une expression régulière.

Lisez [**la documentation**](https://docs.python.org/3.6/library/re.html) pour en savoir plus !

Utilisez la première, search, car nous ne souhaitons obtenir qu'un seul résultat.

if \_\_name\_\_ == '\_\_main\_\_':

args = parse\_arguments()

try:

datafile = args.datafile

if datafile == None:

raise Warning('You must indicate a datafile!')

except Warning as e:

lg.warning(e)

else:

e = re.search(r'^.+\.(\D{3})$', args.datafile)

extension = e.group(1)

if extension == 'xml':

x\_an.launch\_analysis(datafile)

elif extension == 'csv':

c\_an.launch\_analysis(datafile, args.byparty, args.info, args.displaynames,

args.searchname, args.index, args.groupfirst, args.byage)

finally:

lg.info('#################### Analysis is over ######################')

La méthode group() renvoie les valeurs d'un groupe. Puisque vous souhaitez travailler avec la **fin** de la chaîne de caractères, sans le point, utilisez des parenthèses dans l'expression régulière et récupérez les valeurs grâce à group. [Lire la documentation](https://docs.python.org/3.6/library/re.html#re.match.group).

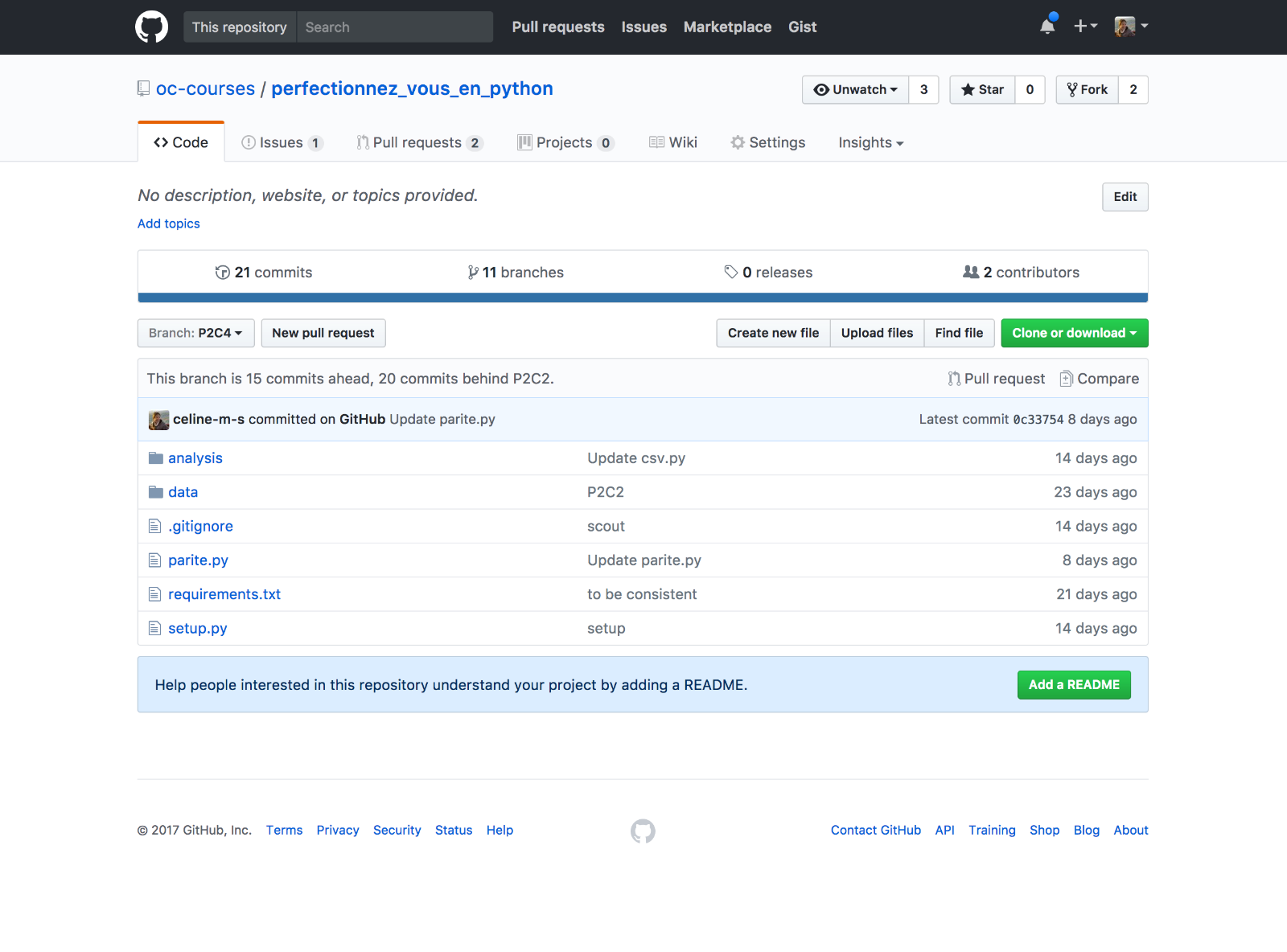
Il est également d'usage d'utiliser des **raw strings** pour écrire une expression régulière.

Une chaîne de caractères normale contient des caractères pouvant être interprétés. Afin de les échapper, nous utilisons un \. Mais bien souvent nous ne souhaitons pas que Python interprète la chaîne de caractères passée dans une expression régulière.

C'est pour cela que nous utilisons des raw strings : les caractères spéciaux n'y sont pas interprétés. Pour cela, ajoutez r avant le guillemet ouvrant la chaîne. Par exemple : re.search(r'C')

### Code du chapitre

[Le code du chapitre est à retrouver ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P2C4)



## Faites connaissance avec les générateurs et les itérateurs

Dans ce chapitre nous aborderons les notions suivantes :

* itérateurs
* générateurs
* expressions génératrices

Vous êtes-vous déjà demandé ce qu'il se passait lorsque vous écriviez une boucle for?

>>> for i in [0, 1, 2, 3]:

... print(i)

C'est assez intuitif, nous avons une liste avec 4 éléments, et l'instruction for assigne à la variable i chacun de ses éléments un à un. Ensuite, nous faisons ce que nous souhaitons avec i : ici, on se contente de l'afficher.

Oui mais, l'instruction produit le même résultat que ce code :

>>> for i in range(4):

... print(i)

Cependant, l'objet renvoyé par range(4) n'est pas une liste, c'est un objet de type... range ! A vrai dire, ce type d'objet est plutôt énigmatique : nous ne savons ni ce que c'est, ni ce qu'il y a dedans.

>>> range(4)

range(0, 4)

Si vous utilisez la version 2 de Python, alors  range()  renvoie **quand même** une liste, ce qui fait de l'exemple ci dessus un mauvais exemple ! Ici, nous utilisons Python 3. Écrire  range(4)  en Python 3 est équivalent à écrire  xrange(4)  en Python 2 (notons l'apparition de la lettre **x**).

Comment donc l'instruction for  peut-elle déterminer les différents éléments qui composent un objet de type range?

La réponse tient en un mot : les **itérables** !

### Les itérateurs

En fait, tous les objets de type itérables peuvent être itérés par un   for. Les listes, mais aussi les chaînes de caractères, les dictionnaires, etc.

#### Comment for interagit-il avec les itérateurs?

Que se passe-t-il lorsque j'écris cela?

for element in iterable\_object:

[...]

Comme   iterable\_object  se trouve dans un for, Python va tout d'abord appeler cette instruction :

iter(iterable\_object)

Celle-ci aura pour effet de retourner un objet de type **itérateur**, mais à condition que iterable\_object soit un objet **itérable**. Bah oui, on n'itère pas sur un objet non itérable, c'est comme ça !

Par exemple, en écrivant for element in [1, 2, 3, 4], Python appellera l'instruction iter([1, 2, 3, 4]), qui renvoie ceci :

>>> it = iter([1, 2, 3, 4])

>>> it

<list\_iterator at 0x7f712a7e3cc0>

Ensuite, Python va appeler l'instruction suivante autant de fois que possible :

next(it)

On tente ?

>>> iterable\_object = [1, 2, 3, 4]

>>> iterator = iter(iterable\_object)

>>> next(iterator)

1

>>> next(iterator)

2

>>> next(iterator)

3

>>> next(iterator)

4

>>> next(iterator)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

Ouh là ! Mais ce code a levé une exception ! Qu'a-ton fait de mal?

Pas de panique, nous n'avons rien fait de mal.

Pour la boucle for, il n'est pas toujours possible de déterminer combien d'éléments contient un objet itérable. Alors elle nous demande de la mettre au courant, et c'est bien normal ! Pour cela, l'itérateur lève une exception de type StopIteration lorsqu'il n'a plus d'objets à renvoyer. Lorsque for attrape cette exception, il sait qu'il faut s’arrêter.

#### Créer son propre itérateur

Puis-je créer mon propre itérateur à moi tout seul ?

Oui ! Il suffit de créer un objet qui implémente les 2 méthodes spéciales appelées par Python lorsqu'il rencontre une boucle for : \_\_iter\_\_ et \_\_next\_\_.

Par exemple :

class MyIterator:

def \_\_init\_\_(self):

print("Je m'initialise à 40")

self.i = 40

def \_\_iter\_\_(self):

print("On a appelé \_\_iter\_\_")

return self

def \_\_next\_\_(self):

print("On a appelé \_\_next\_\_")

self.i += 2

if self.i > 56 :

raise StopIteration()

return self.i

Ici, nous avons créé un itérateur qui renverra successivement les nombres de 40 (non compris) à 56, en comptant de deux en deux.MyIteratorinitialise son attribut self.i à 40. A chaque appel de \_\_next\_\_, il incrémente son attribut self.i de 2, et il **garde** intérieurement son "état" (représenté par self.i). Testons !

>>> for i in MyIterator():

... print(i)

Je m'initialise à 40

On a appelé \_\_iter\_\_

On a appelé \_\_next\_\_

42

On a appelé \_\_next\_\_

44

On a appelé \_\_next\_\_

46

On a appelé \_\_next\_\_

48

On a appelé \_\_next\_\_

50

On a appelé \_\_next\_\_

52

On a appelé \_\_next\_\_

54

On a appelé \_\_next\_\_

56

On a appelé \_\_next\_\_

Ici, un objet de type MyIteratorn'est utilisable qu'une seule fois. En effet, le "compteur"   self.i  n'est initialisé à 40 qu'à la création de l'objet, et pas à l'appel de   \_\_iter\_\_. Si on appelle   \_\_iter\_\_  une seconde fois, par exemple dans une seconde boucle   for, l'attribut self.i sera égal à 58, car il n'aura pas été réinitialisé à 40. Pour utiliserMyIteratorplusieurs fois, il suffit de déplacer l'instruction  self.i = 40  de la méthode  \_\_init\_\_  vers la méthode  \_\_iter\_\_.

**Petit bonus :** Sachez que beaucoup de fonctions natives de Python savent très bien interagir avec votre nouvel itérateur ! Regardez :

>>> list(MyIterator())

[42, 44, 46, 48, 50, 52, 54, 56]

>>> sum(MyIterator())

392

#### Itérons sur les députés !

Si nous revenons à notre projet, il est tout à fait possible de transformer notre objet SetOfParliamentMember en itérateur. Il suffit de faire cela :

class SetOfParliamentMembers:

...

def \_\_iter\_\_(self):

self.iterator\_state = 0

return self

def \_\_next\_\_(self):

if self.iterator\_state >= len(self):

raise StopIteration()

result = self[self.iterator\_state]

self.iterator\_state += 1

return result

Les instructions len(self) et self[self.iterator\_state] sont respectivement rendues possibles grâce à la définition (dans les précédents chapitres) des méthodes spéciales  \_\_len\_\_ et \_\_getitem\_\_. Pour connaître leur effet, reportez-vous au chapitre concernant les **Méthodes Spéciales**.

Nous pouvons ici afficher les noms et adresses mail de tous les députés un à un :

>>> sopm = SetOfParliamentMembers("All MPs")

>>> sopm.data\_from\_csv(os.path.join("data","current\_mps.csv"))

>>> for mp in sopm:

... print(mp["nom"], mp["emails"])

#### Finalement, l'objet  range, c'est quoi ?

Vous l'aurez deviné, l'objet renvoyé par l'instruction  range(4)  que nous avons vu au tout début de ce chapitre, c'est un itérable ! Vous pouvez donc très bien l'utiliser dans une boucle  for, ou via  next  et  iter!

### Les générateurs

 En Python, les générateurs ont été créés afin de simplifier la création d'itérateurs, et utilisent un mot magique : yield. Ce dernier est souvent difficile à appréhender, mais arrivé à ce stade, vous avez les neurones bien échauffés : vous allez y arriver !

Réécrivons MyIteratorsous forme de générateur. Nous l’appellerons my\_generator.

def my\_generator():

i = 40

while i <= 56:

i += 2

yield i

Mais ce que tu viens d'écrire, c'est une fonction !

Oui, c'est une fonction. Une fonction **génératrice**, qui renvoie un **générateur**. En fait, dès qu'une fonction contient le mot clé yield, elle retourne un générateur. Le terme **générateur** est à la fois employé pour désigner la fonction elle même (ici my\_generator) et pour désigner le générateur que celle-ci retourne.

Pour tester ce générateur, utilisez le même code que pour testerMyIterator, et vous verrez que le résultat est exactement le même. Sauf qu'ici, nous avons écrit beaucoup moins de lignes de code.

Que se passe t-il lors de l’exécution de my\_generator?

Quand une fonction génératrice est appelée, elle retourne uniquement un générateur, sans exécuter les instructions de la fonction. C'est seulement lorsque next est appelée que les instructions sont exécutées jusqu'à ce que le mot clé yield soit rencontré. A ce moment, la valeur spécifiée par yield est retournée. Ensuite, l'exécution est stoppée, mais l'état interne de la fonction génératrice (c'est à dire ses variables) est conservé. Lors d'un nouvel appel de next, l'exécution des instructions reprend là où elle s'était arrêtée, jusqu'à rencontrer un prochain yield. Si aucun yield n'est rencontré avant la fin de la fonction (ou si le mot clé return est rencontré), alors la fonction génératrice lève la StopIteration.

Un petit exemple : prenons un générateur qui renverra tous les nombre entiers compris entre deux nombres passés en paramètre, en alternant le résultat sous forme de float si le résultat est pair, puis sous forme de str si le résultat est impair. Enfin, il affichera quelques chaînes de caractères.

def generator(beginning, end):

print(" On commence !")

cpt = beginning

while cpt <= end:

if cpt % 2 == 0:

print(" On s'arrete au yield")

yield float(cpt)

print(" On reprend après le yield")

else:

print(" On s'arrete au yield")

yield str(cpt)

print(" On reprend après le yield")

cpt += 1

yield "C'est bientôt la fin"

yield "C'est VRAIMENT bientôt la fin"

yield "Là c'est la fin"

>>> for i in generator(4, 8):

... print(i)

On commence !

On s'arrete au yield

4.0

On reprend après le yield

On s'arrete au yield

5.0

On reprend après le yield

On s'arrete au yield

6.0

On reprend après le yield

On s'arrete au yield

7.0

On reprend après le yield

On s'arrete au yield

8.0

On reprend après le yield

C'est bientôt la fin

C'est VRAIMENT bientôt la fin

Là c'est la fin

### Les expressions génératrices

Vous vous souvenez des [**compréhensions de listes**](https://openclassrooms.com/courses/decouvrez-la-programmation-orientee-objet-avec-python/calculez-l-agreabilite-moyenne-d-une-zone#r-4313506)?

>>> [2\*x for x in range(3)]

[0, 2, 4]

Et bien je vous présente leur équivalent en générateur : les **expressions génératrices** ! C'est exactement le même principe, sauf qu'au lieu de renvoyer une liste, une expression génératrice renvoie un générateur :

>>> gen = (2\*x for x in range(3))

>>> gen

<generator object <genexpr> at 0x7fa777055e60>

>>> sum(gen)

6

Vous venez de le voir : il suffit de remplacer les crochets par des parenthèses.

### Aller plus loin : L'intérêt des générateurs

Si vous avez intuitivement compris l'intérêt des générateurs, vous pouvez passer cette partie. Je vais vous expliquer ici en quoi les générateurs peuvent nous faire gagner en mémoire et en lisibilité de code.

Prenons un exemple :

Un de vos amis a besoin de créer une fonction python qui filtre dans un texte tous les mots ayant au moins 6 caractères et qui contiennent la lettre "A". La taille du texte est potentiellement très grosse : imaginez par exemple que vous deviez exécuter cette fonction tous les jours (à heure fixe) sur l'ensemble des textes publiés sur facebook par l'ensemble des utilisateurs de ce réseau social ! Cela représente des centaines de milliers de mots chaque jour !

Vous décidez de relever le défi, et vous vous répartissez le travail : vous vous occupez de créer une fonction qui lit une très grosse chaîne de caractères (l'ensemble des posts facebook des dernières 24h) et la découpe en une liste de mots. Votre ami, de son côté, s'occupera des fonctions qui filtreront ces mots (mots contenant au moins 6 caractères avec la lettre A).

Vous créez donc la fonctionget\_wordset lui les fonctionsfilter\_by\_sizeetfilter\_by\_letters.

Voici tout d'abord la donnée à traiter. Elle est ici petite, mais c'est pour tester !

big\_data = """Le sénateur, dont il a été parlé plus haut, était un homme entendu qui

avait fait son chemin avec une rectitude inattentive à toutes ces rencontres qui font

obstacle et qu'on nomme conscience, foi jurée, justice, devoir; il avait marché droit à

son but et sans broncher une seule fois dans la ligne de son avancement et de son intérêt.

C'était un ancien procureur, attendri par le succès, pas méchant homme du tout, rendant

tous les petits services qu'il pouvait à ses fils, à ses gendres, à ses parents, même à

des amis; ayant sagement pris de la vie les bons côtés, les bonnes occasions, les bonnes

aubaines. Le reste lui semblait assez bête. Il était spirituel, et juste assez lettré

pour se croire un disciple d'Epicure en n'étant peut-être qu'un produit de Pigault-Lebrun.

[...]

(Les Misérables, Victor Hugo)

"""

Voici maintenant votre code :

import re

def is\_part\_of\_a\_word(character):

return len(re.findall('\w', character, flags = re.UNICODE))

def get\_words(text):

print("Je commence à lire le texte maintenant")

words = []

current\_word = ""

for character in text:

if not is\_part\_of\_a\_word(character):

if current\_word != "":

words += [current\_word]

current\_word = ""

else:

current\_word += character

return words

Et voici le code de votre ami :

def filter\_by\_size(words):

return [w for w in words if len(w) >= 6]

def filter\_by\_letters(words):

return [w for w in words if "a" in w]

words = get\_words(big\_data)

print("Nombre de mots: %i" % len(words))

words = filter\_by\_size(words)

print("Nombre de mots: %i" % len(words))

words = filter\_by\_letters(words)

print("Nombre de mots: %i" % len(words))

print(words)

Ce qui donne ce résultat :

Je commence à lire le texte maintenant

Nombre de mots: 146

Nombre de mots: 42

Nombre de mots: 17

['sénateur', 'inattentive', 'obstacle', 'marché', 'avancement', 'ancien', 'attendri', 'méchant', 'rendant', 'pouvait', 'parents', 'sagement', 'occasions', 'aubaines', 'semblait', 'Pigault', 'Misérables']

 Oui, mais ici il y a un problème. En effet, à la ligne 8 du code de votre ami, la variablewordscontient 146 mots, alors qu'en fait, l'ensemble de ces 146 mots ne nous sera pas utile, car ils seront encore filtrés par la suite ! 146 mots c'est peu, mais si votre texte est beaucoup plus grand, votre machine peut facilement manquer de mémoire et planter !

 Pour éviter de stocker 146 mots (dont beaucoup sont inutiles) dans une variable, vous décidez donc de ne stocker les mots dans la variablewordsqu'une fois que ceux-ci auront été **totalement** filtrés.

Mais ceci implique que votre ami modifie votre code en y implantant le sien :

def get\_words(text):

print("Je commence à lire le texte maintenant")

words = []

current\_word = ""

for character in text:

if not is\_part\_of\_a\_word(character):

if current\_word != "":

# code ajouté :

if len(current\_word) >= 6:

if "a" in current\_word:

# fin du code ajouté

words += [current\_word]

current\_word = ""

else:

current\_word += character

return words

words = get\_words(big\_data)

print(words)

C'est une bonne solution, mais qui a deux inconvénients :

1. Votre ami n'aime pas modifier ce que vous avez fait, il préfère que chacun travaille de son côté, car comme il n'est pas "dans votre tête", il a peur d'introduire des erreurs en modifiant votre code.
2. Cela ajoute de la complexité visuelle au code : ici, on a rajouté 2 niveaux d'indentation ! (lignes 12 et 13)

Voici donc la meilleure des solutions, qui utilise des générateurs et des expressions génératrices :

def get\_words(text):

print("Je commence à lire le texte maintenant")

current\_word = ""

for character in text:

if not is\_part\_of\_a\_word(character):

if current\_word != "":

yield current\_word

current\_word = ""

else:

current\_word += character

def filter\_by\_size(words):

return (w for w in words if len(w) >= 6)

def filter\_by\_letters(words):

return (w for w in words if "a" in w)

words = get\_words(big\_data)

words = filter\_by\_size(words)

words = filter\_by\_letters(words)

print("'words' est encore un générateur. Le texte n'a toujours pas été lu")

print("L'opération suivante va lancer la lecture du texte: ")

[w for w in words]

Elle est beaucoup plus lisible n'est-ce pas ?

Prenez 5 minutes pour bien comprendre l’enchaînement des opérations à l'aide desprint. Voici le résultat affiché :

'words' est encore un générateur. Le texte n'a toujours pas été lu

L'opération suivante va lancer la lecture du texte:

Je commence à lire le texte maintenant

['sénateur',

'inattentive',

'obstacle',

'marché',

'avancement',

'ancien',

'attendri',

'méchant',

'rendant',

'pouvait',

'parents',

'sagement',

'occasions',

'aubaines',

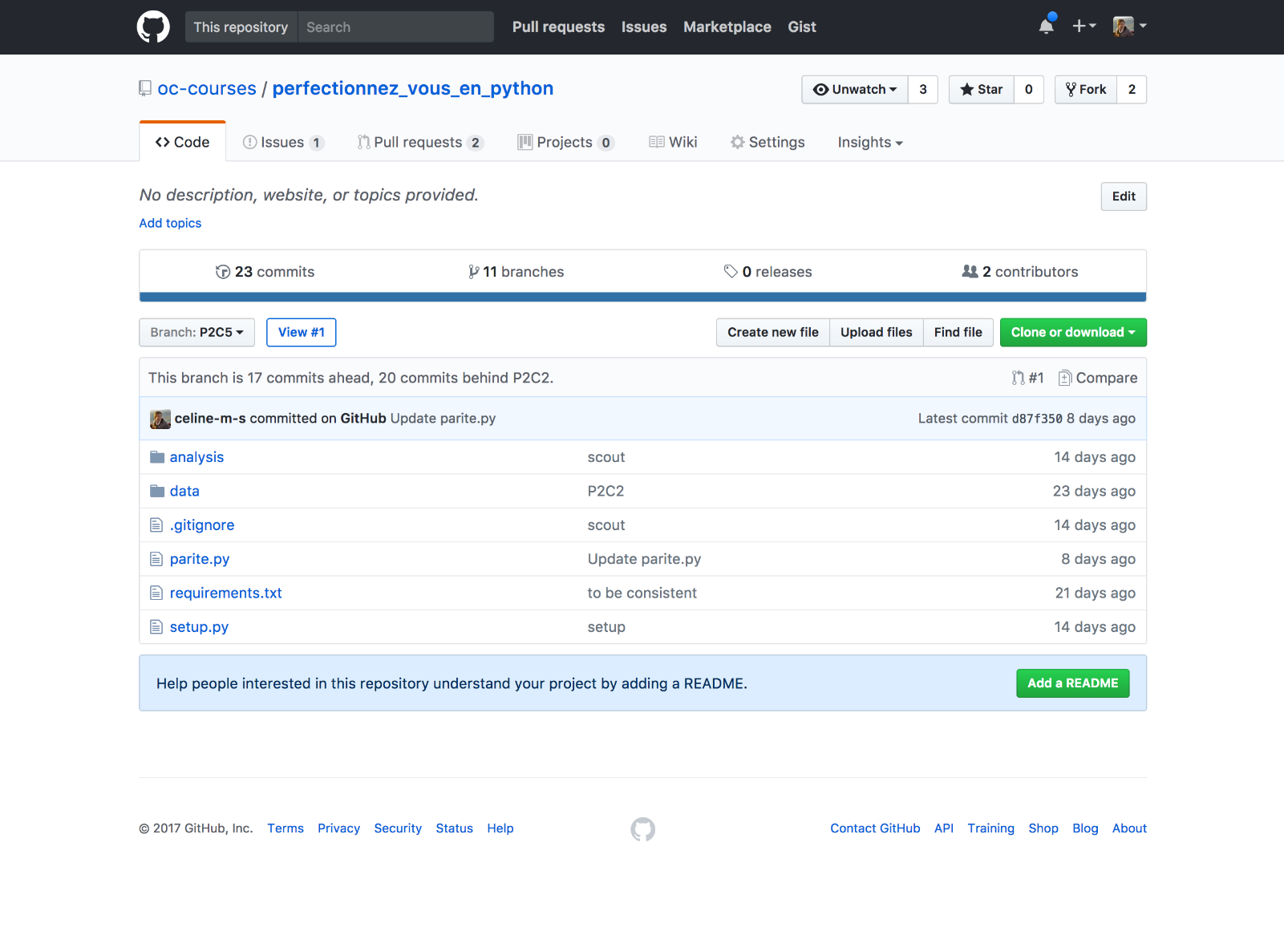
'semblait',

'Pigault',

'Misérables']

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P2C5)



# Approfondissez votre utilisation de Python

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Manipuler des objets en Python

### Question 1

**L'interpréteur de Python invoque des méthodes spéciales pour réaliser des opérations basiques sur des objets.**

* + 

Vrai

* + 

Faux

### Question 2

**La méthode spéciale correspondant à la fonction  len()  est la suivante :**

* + 

\_\_item\_\_

* + 

\_\_iter\_\_

* + 

\_\_len\_\_

### Question 3

**La méthode spéciale  \_\_repr\_\_  est appelée par la fonction  str() .**

* + 

Vrai

* + 

Faux

### Question 4

**Un décorateur permet...**

* + 

d'imprimer des informations détaillées sur une fonction. Il ne sert qu'à cela et ne peut pas être modifié.

* + 

de modifier le comportement d'une fonction.

### Question 5

**Trouvez le code qui imprimera "Moi président de la République, j'engagerai de grands débats".**

* + 
  + def chorus(func):
  + def inner():
  + print("Moi président de la République, ")
  + return func()
  + return inner
  + @chorus
  + def speech():
  + print("j'engagerai de grands débats.")

* + 
  + def chorus(func):
  + def inner():
  + print("Moi président de la République, ")
  + return inner
  + @chorus
  + def speech():
  + print("j'engagerai de grands débats.")
  + 
  + def chorus(func):
  + print("Moi président de la République, ")
  + return func
  + @chorus
  + def speech():
  + print("j'engagerai de grands débats.")

### Question 6

**La librairie standard de Python intègre de nombreux décorateurs.**

* + 

Vrai

* + 

Faux

### Question 7

**Trouvez l'expression régulière qui trouve  aiguille  , et uniquement  aiguille ,  dans la chaîne de caractères suivante :**

Botte de foin aiguille anguille aigüe

**Vous pouvez vous aider de**[***Pythex***](http://pythex.org/)**! ;-)**

* + 

^a...[ille]+$

* + 

ai..[ile]{4}

* + 

^a..[ille]+$

### Question 8

**Une expression régulière est :**

* + 

une chaîne de caractères très précise qui représente un mot à trouver dans une phrase. Par exemple, "maison" pour trouver le mot "maison" dans "c'est une maison bleue".

* + 

une chaîne de caractères, souvent appelée motif, qui décrit selon une syntaxe particulière un ensemble de chaînes de caractères possibles. Par exemple, "^m[ai]{2}son$" pour trouver le mot "maison" dans "c'est une maison bleue".

### Question 9

**Le symbole   .  représente, dans une expression régulière, tout caractère (lettre, chiffre ou symbole).**

* + 

Vrai

* + 

Faux

### Question 10

**Le symbole  $, dans une expression régulière, représente le début d'un mot.**

* + 

Vrai

* + 

Faux

## Assimilez les bonnes pratiques de la PEP 8

Dans ce chapitre nous aborderons les notions suivantes :

* Code pythonique
* PEP 8
* Pylint
* Utiliser l'extension Pylint avec un éditeur de texte

### Python Extension Proposal : PEP

Le langage Python a été créé par Guido van Rossum (dictateur bienveillant à vie) mais grandit et évolue grâce à sa communauté. Chaque proposition d'amélioration est publique et publiée sur [le site de Python](https://www.python.org/dev/peps/). Elles sont connues sous le nom de Python Enhancement Proposal (proposition d'amélioration de Python) et portent chacunes un numéro. Certaines sont acceptées, d'autres reportées mais toutes sont ouvertes au débat. Vous pouvez en consulter la liste sur [la page de l'index](https://www.python.org/dev/peps/).

### La PEP 8

[La PEP 8](http://pep8.org/) a pour objectif de définir des règles de développement communes entre développeurs. En effet, vous avez pu remarquer que le langage est assez flexible. Par exemple, il est possible d'utiliser 2 espaces ou 4 pour l'indentation. Vous pouvez également ne pas mettre d'espace entre les opérateurs. Le code fonctionnera toujours ! Mais ce n'est pas parce que vous **pouvez** que vous **devez** le faire !

Que se passe-t-il le jour où une autre personne reprend votre projet ? Elle essaiera de le comprendre avant tout. Si le code est complexe, dense, et qu'il ne suit pas des règles communes, votre interlocuteur aura besoin de plus de temps et d'énergie pour se l'approprier. Ce n'est pas ce que vous souhaitez !

Guido van Rossum, d'ailleurs, insiste sur le fait qu'un développeur passe plus de temps à lire du code qu'à l'écrire. C'est on ne peut plus juste ! Vous êtes vous-même en train de lire du code à travers ce cours... C'est pourquoi votre code doit avant tout être facile à comprendre. Autrement dit, **lisible**.

### Contenu de la PEP 8

Nous verrons quelques points clés qui sont essentiels mais nous ne détaillerons pas la PEP 8 dans son ensemble. Pourquoi ? Avant tout car elle est très bien expliquée sur le site ! Mais également car j'aimerais attirer votre attention sur quelques points clés, essentiels à connaître, avant de vous laisser la parcourir par vous-même.

#### Mise en page

Votre code suit une certaine syntaxe et une mise en page. Vous le savez déjà : l'indentation a une importance capitale dans ce langage ! Voici quelques règles importantes :

* Une ligne doit contenir 80 caractères maximum.
* L'indentation doit être de 4 espaces.
* Ajoutez deux lignes vides entre deux éléments de haut niveau, des classes par exemple, pour des questions d'ergonomie.
* Séparez chaque fonction par une ligne vide.
* Les noms (variable, fonction, classe, ...) ne doivent pas contenir d'accent. Que des lettres ou des chiffres !

print("-- I am a line made of eighty characters maximum, no more, no less! --")

if n < m:

print('indent: 4 spaces')

class FirstClass:

def \_\_init\_\_(self, luggages):

self.luggages = luggages

def travel(self):

print('Luxe, calme et volupté')

# Seperate each high level element by two lines

class SecondClass(FirstClass):

def \_\_init\_\_(self):

self.luggages = 0

# Separate each low level element by one line

def travel(self):

print('Down the road, I go...')

#### Documentation strings (Docstrings) obligatoires

Une doctring est un ensemble de mots qui documente un bout de code. Elle commence par trois guillemets ouvrants, le commentaire que vous souhaitez apporter puis trois guillements fermants.

Exemple :

""" This is a doctring. I'm here to explain what the following code will do.

Oh, I'm multiline too!

"""

Selon la PEP 8, chaque partie de votre code devrait contenir une Doctring.

* tous les modules publics
* toutes les fonctions
* toutes les classes
* toutes les méthodes de ces classes

#### Imports

L'import d'une librairie doit être rapide à déceler. Il est également important de bien différencier la source des librairies : standard, externe ou locale. Cela permet de savoir ce qu'il faut installer.

* Les imports sont à placer au début d'un script.
* Ils précèdent les Docstrings.
* Une ligne par librairie. Exemple : import os
* Une ligne peut néanmoins inclure plusieurs composantes. Exemple : from subprocess import Popen, PIPE
* L'import doit suivre l'ordre suivant : Bibliothèques standard, Bibliothèques tierces et imports locaux. Sautez une ligne entre chacun de ces blocs.

#### Espaces dans les instructions

Les espaces suivent la syntaxe anglosaxone et non française. De manière plus générale, elle s'axe sur la lisibilité tout en supprimant les espaces surperflus.

* Pas d'espace avant : mais un après. Exemple : {oeufs: 2}
* **Opérateurs** : un espace avant et un après. Exemple : i = 1 + 1
* Aucun espace avant et après un signe = lorsque vous assignez la valeur par défaut du paramètre d'une fonction. Exemple : def elephant(trompe=True, pattes=4)
* Une instruction par ligne.

#### Commentaires

Même si les développeurs ne sont pas tous d'accord sur ce point, les commentaires jouent un rôle essentiel dans la compréhension d'un code. Voici quelques bonnes pratiques :

* Ecrivez des phrases complètes, ponctuées et compréhensibles.
* Le commentaire doit être cohérent avec le code.
* Il doit suivre la même indentation que le code qu'il commente.
* Evitez d'enfoncer des portes ouvertes : ne décrivez pas le code, expliquez plutôt à quoi il sert.
* Il doit être **en anglais**.

Guido van Rossum est très clair sur ce point. Il l'explique d'ailleurs ainsi : "Aux développeurs Python de pays non anglophones : écrivez vos commentaires en anglais, s'il vous plaît, sauf si vous êtes sûr à 120% que ce code ne sera jamais lu par des personnes qui ne parlent pas votre langage.

#### Conventions de nommage

Il s'agit d'un point essentiel que nous avons couvert tout au long de ces cours sur Python. En voici les grandes lignes :

* **Modules** : nom court, tout en minuscules, tiret du bas si nécessaire. great\_module
* **paquets** : nom court, tout en minuscules, tirets du bas très déconseillés. paquet
* **classes** : lettres majuscules en début de mot. MyGreatClass
* **exceptions** : similaire aux classes mais avec un Error à la fin. MyGreatError
* **fonctions** : minuscules et tiret du bas : my\_function()
* **méthodes** : minuscules, tiret du bas et self en premier paramètre : my\_method(self)
* **arguments des méthodes et fonctions** : identique aux fonctions. my\_function(param=False)
* **variables** : identique aux fonctions.
* **constantes** : tout en majuscules avec des tirets si nécessaire. I\_WILL\_NEVER\_CHANGE
* **privé** : précédé de deux tirets du bas : \_\_i\_am\_private
* **protégé** : précédé d'un tiret du bas : \_i\_am\_protected

#### Comparaisons

Par convention, il est mieux d'utiliser is ou is not lors d'une comparaison avec None. Par exemple :

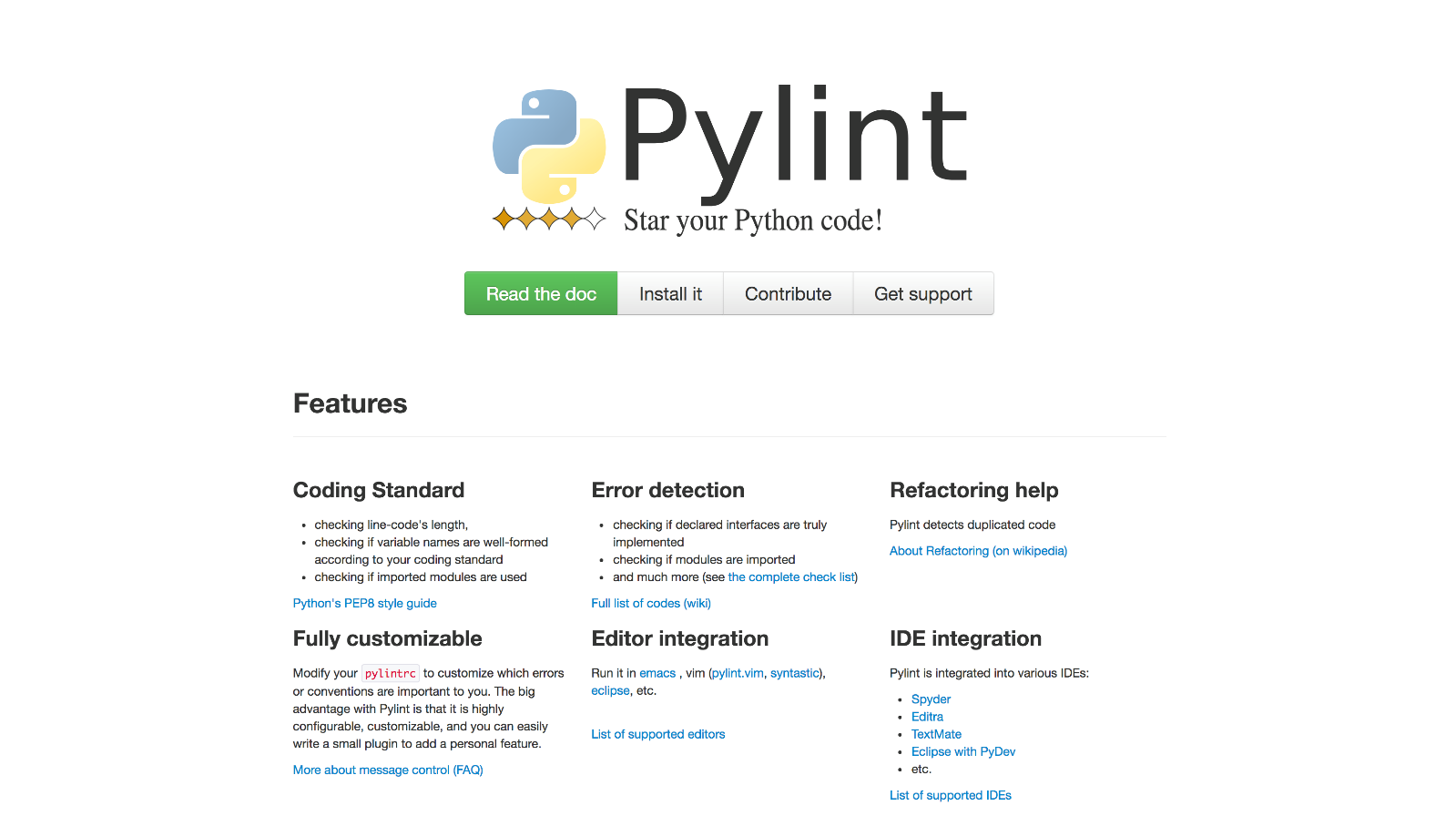
if datafile is None:

print("I'm pythonic")

### La PEP 8 en action dans le projet

Les conventions de la PEP 8 s'appliquent au fur et à mesure du développement d'un projet. Néanmoins, il est intéressant d'effectuer une recherche de temps en temps (avant un commit par exemple) pour déceler les quelques incartades qui pourraient se glisser. Plus vous coderez, plus vous la connaîtrez par coeur et donc moins vous aurez besoin de la regarder !

Bien sûr, vous pouvez le faire à la main : reprenez la liste plus haut et appliquez-la à votre code. Mais c'est fastidieux et chronophage. Plusieurs solutions d'analyse de code existent pour vous faciliter le travail ! Laissez-moi vous en présenter une : [Pylint](https://www.pylint.org/).



Pylint est un logiciel en ligne de commande qui analyse le code en fonction des standards de la PEP 8. Il détecte également les erreurs et le code dupliqué.

#### Utiliser Pylint

Commencez par installer Pylint :

pip install pylint

Quand Pylint est installé, lancez l'analyse en tapant pylint [votrescript.py].

pylint parite.py

Voici le résultat de son analyse :

$ pylint parite.py

No config file found, using default configuration

\*\*\*\*\*\*\*\*\*\*\*\*\* Module parite

C: 15, 0: Exactly one space required after comma

parser.add\_argument("-d","--datafile",help="""CSV file containing pieces of

^ (bad-whitespace)

C: 15, 0: Exactly one space required after comma

parser.add\_argument("-d","--datafile",help="""CSV file containing pieces of

^ (bad-whitespace)

C: 17, 0: Line too long (107/100) (line-too-long)

C: 18, 0: Exactly one space required after comma

parser.add\_argument("-p","--byparty", action='store\_true', help="""displays

^ (bad-whitespace)

C: 20, 0: Exactly one space required after comma

parser.add\_argument("-i","--info", action='store\_true', help="""information about

^ (bad-whitespace)

C: 22, 0: Exactly one space required after comma

parser.add\_argument("-n","--displaynames", action='store\_true', help="""displays

^ (bad-whitespace)

C: 24, 0: Exactly one space required after comma

parser.add\_argument("-s","--searchname", help="""search for a mp name""")

^ (bad-whitespace)

C: 25, 0: Exactly one space required after comma

parser.add\_argument("-I","--index", help="""displays information about the Ith mp""")

^ (bad-whitespace)

C: 26, 0: Exactly one space required after comma

parser.add\_argument("-g","--groupfirst", help="""displays a graph groupping all the 'g'

^ (bad-whitespace)

C: 45, 0: Wrong continued indentation (add 6 spaces).

args.searchname, args.index, args.groupfirst)

^ | (bad-continuation)

C: 1, 0: Missing module docstring (missing-docstring)

C: 13, 0: Missing function docstring (missing-docstring)

C: 30, 0: Missing function docstring (missing-docstring)

C: 34,11: Comparison to None should be 'expr is None' (singleton-comparison)

C: 36, 4: Invalid variable name "e" (invalid-name)

C: 39, 8: Invalid variable name "e" (invalid-name)

------------------------------------------------------------------

Your code has been rated at 4.86/10 (previous run: 8.00/10, -3.14)

Ouch ! 4.86/10... Notre code n'est pas de très bonne qualité. En effet, il n'y a aucune Docstring et de nombreux espaces sont mal placés. Corrigeons cela et relançons Pylint.

$ pylint parite.py

No config file found, using default configuration

-------------------------------------------------------------------

Your code has been rated at 10.00/10 (previous run: 8.33/10, +1.67)

Youhou ! Bravo !

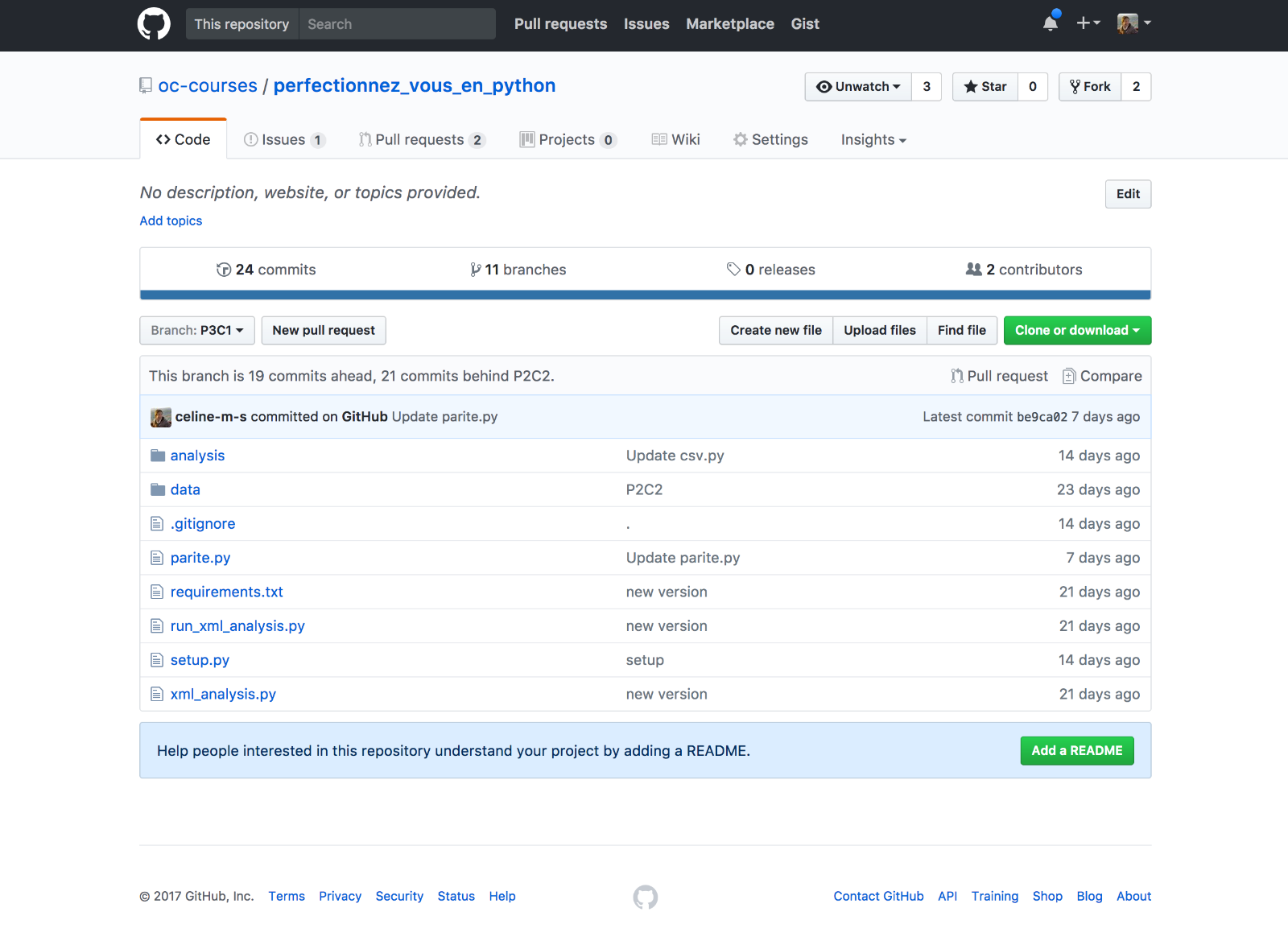
Il existe de nombreuses extensions qui permettent d'utiliser Pylint directement dans un éditeur de texte. [**Linter-pylint**](https://atom.io/packages/linter-pylint) pour Atom ou [**SublimeLinter-pylint**](https://packagecontrol.io/packages/SublimeLinter-pylint) pour Sublime Text par exemple. Dans les deux cas, vous devez lancer l'éditeur en ayant activé l'environnement virtuel ! Sinon votre éditeur ne connaîtra pas le chamin vers l'exécutable et ne pourra pas lancer Pylint.

* Installez l'extension Pylint pour l'éditeur de votre choix.
* Fermez ce dernier.
* Ouvrez votre console et déplacez-vous dans le répertoire de travail souhaité.
* Activez l'environnement virtuel : source env/bin/activate
* Ouvrez les fichiers avec l'éditeur de texte : atom . pour Atom ou subl . pour Sublime Text.

La PEP 8 est un ensemble de règles très importantes mais qui peuvent être transgressées dans certains contextes spécifiques. Retenez simplement qu'elle garantit un code lisible et de qualité qui améliore la collaboration entre développeurs.

### Code du chapitre

[Retrouvez le code de ce chapitre en cliquant ici.](https://github.com/OpenClassrooms-Student-Center/perfectionnez-vous-en-python/tree/P3C1)



## Devenez zen avec la PEP 20

Dans ce chapitre nous aborderons les notions suivantes :

* Les aphorismes de la PEP 20
* Comment les mettre en pratique

### La PEP 20

Dans le chapitre précédent vous avez découvert la PEP 8 qui est le guide de style pour écrire un code de qualité en Python. Cette PEP intimement liée à [la PEP 20](https://www.python.org/dev/peps/pep-0020/), également appelée [le "Zen de Python"](http://pep20.org/). Il s'agit de 19 [aphorismes](https://fr.wikipedia.org/wiki/Aphorisme) écrits par Tim Peters et reflétant la philosophie de Python. Vous pouvez les voir comme des guides de conduite vous aidant à écrire du meilleur code et à bien comprendre l'état d'esprit du langage.

Les voici, accompagnés d'une humble traduction :

* Le beau est préférable au laid. - Beautiful is better than ugly.
* L'explicite est préférable à l'implicite. - Explicit is better than implicit.
* Le simple est préférable au complexe. - Simple is better than complex.
* Le complexe est préférable au compliqué. - Complex is better than complicated.
* L'horizontal est préférable à l'imbriqué. - Flat is better than nested.
* L'aéré est préférable au dense. - Sparse is better than dense.
* La lisibilité compte. - Readability counts.
* Les cas spéciaux ne le sont pas assez pour transgresser les règles. - Special cases aren't special enough to break the rules.
* Sauf si le cas pratique bat le cas théorique. - Although practicality beats purity.
* Les erreurs ne devraient jamais arriver silencieusement. - Errors should never pass silently.
* Sauf si on les a explicitement rendues silencieuses. - Unless explicitly silenced.
* En cas de doute, ne tentez pas de deviner. - In the face of ambiguity, refuse the temptation to guess.
* Il devrait y avoir une, et de préférence une seule, manière évidente de le faire. - There should be one - and preferably only one - obvious way to do it
* Même si cette manière peut ne pas sembler évidente au premier abord sauf si vous êtes nééerlandais. - Although that way may not be obvious at first unless you're Dutch.
* Ce qui est fait maintenant est préférable à ce qui ne sera jamais fait. - Now is better than never.
* Même si jamais est souvent mieux que tout de suite. - Although never is often better than right now.
* Si l'implémentation est difficile à expliquer, c'est que c'est une mauvaise idée. - If the implementation is hard to explain, it's a bad idea.
* Si l'implémentation est facile à expliquer, c'est que c'est peut-être une bonne idée. - If the implementation is easy to explain, it may be a good idea.
* Les espaces de noms sont une brillante idée, créons-en plus ! - Namespaces are one honking great idea, let's do more of those!

Il est très courant que des développeurs Python s'y réfèrent pour commenter du code donc ne vous étonnez pas ! 😉

### En pratique

Rien de tel que la pratique pour bien comprendre ce que ces aphorismes impliquent ! [Lee Wei Yeong](https://www.github.com/evandrix), un développeur singapourien, a créé des exemples mettant en oeuvre les aphorismes.

Voici les trois premiers.

#### Le beau est préférable au laid

"""

Give me a function that takes a list of numbers and returns only the

even ones, divided by two.

"""

#-----------------------------------------------------------------------

# Wrong example

halve\_evens\_only = lambda nums: map(lambda i: i/2, filter(lambda i: not i%2, nums))

#-----------------------------------------------------------------------

# Beautiful is better than ugly.

def halve\_evens\_only(nums):

return [i/2 for i in nums if not i % 2]

#-----------------------------------------------------------------------

#### L'explicite est préférable à l'implicite

"""

Load the cat, dog, and mouse models so we can edit instances of them.

"""

#-----------------------------------------------------------------------

# Wrong example

def load():

from menagerie.cat.models import \*

from menagerie.dog.models import \*

from menagerie.mouse.models import \*

#-----------------------------------------------------------------------

# Explicit is better than implicit.

def load():

from menagerie.models import cat as cat\_models

from menagerie.models import dog as dog\_models

from menagerie.models import mouse as mouse\_models

#-----------------------------------------------------------------------

#### Le simple est préférable au complexe

"""

Can you write out these measurements to disk?

"""

measurements = [

{’weight’: 392.3, ’color’: ’purple’, ’temperature’: 33.4},

{’weight’: 34.0, ’color’: ’green’, ’temperature’: -3.1},

]

#-----------------------------------------------------------------------

# Wrong example

def store(measurements):

import sqlalchemy

import sqlalchemy.types as sqltypes

db = sqlalchemy.create\_engine(’sqlite:///measurements.db’)

db.echo = False

metadata = sqlalchemy.MetaData(db)

table = sqlalchemy.Table(’measurements’, metadata,

sqlalchemy.Column(’id’, sqltypes.Integer, primary\_key=True),

sqlalchemy.Column(’weight’, sqltypes.Float),

sqlalchemy.Column(’temperature’, sqltypes.Float),

sqlalchemy.Column(’color’, sqltypes.String(32)),

)

table.create(checkfirst=True)

for measurement in measurements:

i = table.insert()

i.execute(\*\*measurement)

#-----------------------------------------------------------------------

# Simple is better than complex.

def store(measurements):

import json

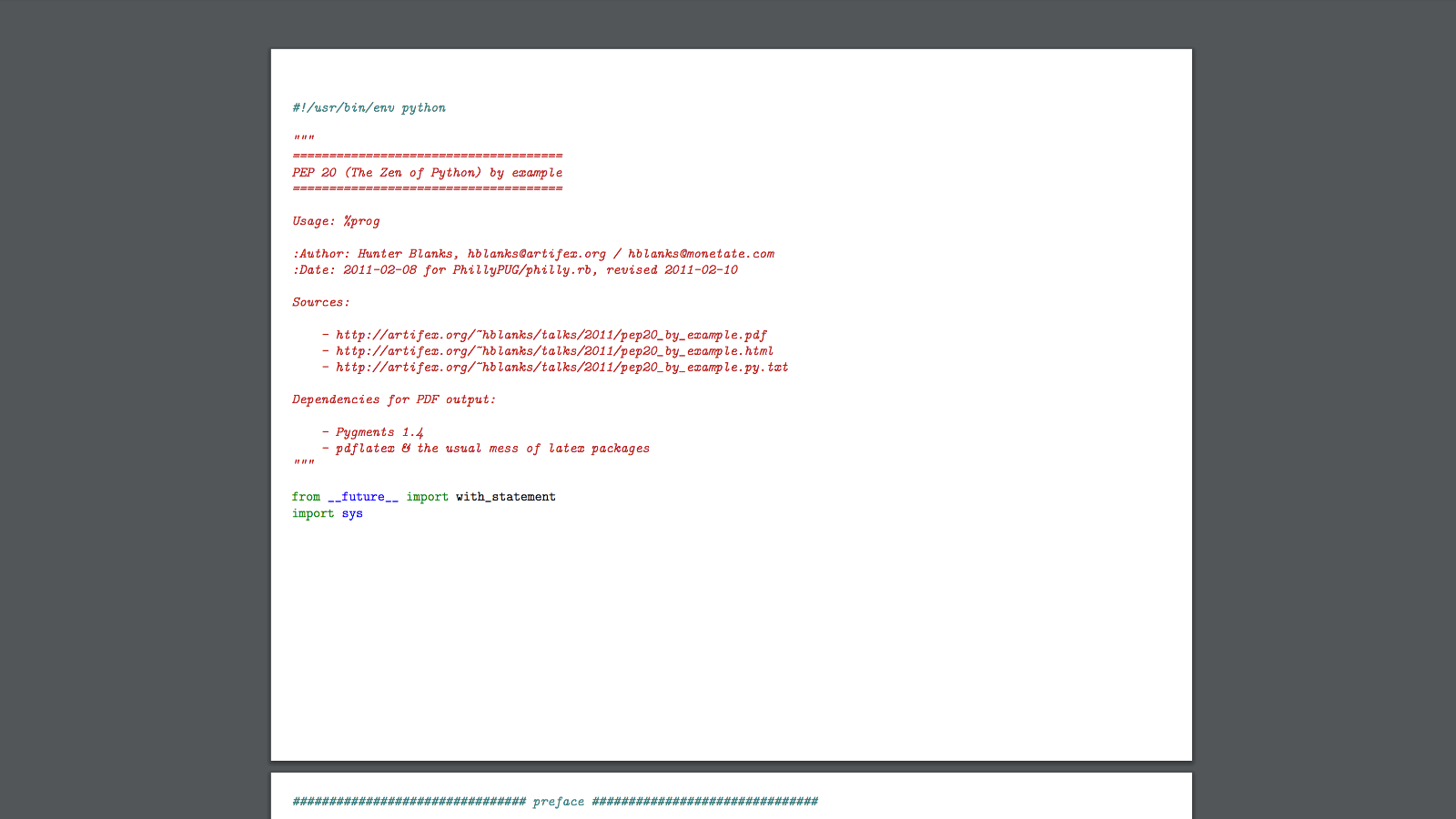
with open(’measurements.json’, ’w’) as f:

f.write(json.dumps(measurements))

#-----------------------------------------------------------------------

6

[Découvrez-les tous ici](https://gist.github.com/evandrix/2030615) ou dans [une version en pdf](http://artifex.org/~hblanks/talks/2011/pep20_by_example.pdf).



Ces 19 aphorismes sont accessibles à tout moment à partir d'un interpréteur Python. Ecrivez simplement  import this :

$ python3

Python 3.6.1 (default, Mar 23 2017, 16:49:06)

[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import this

The Zen of Python, by Tim Peters

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!

>>>

## Allez plus loin

Bravo, vous avez terminé ce cours ! Vous avez désormais une meilleure vision de Python et pouvez utiliser le langage de manière plus avancée. Mais votre apprentissage ne s'arrête pas là ! Il vous reste bien des secrets à découvrir. Voici quelques ressources qui vous aideront à aller plus loin !

### 📖  Livres

* [Fluent Python de Luciano Ramalho](https://amzn.to/2ruaY2Y) : un excellent ouvrage de référence sur l'utilisation avancée de Python. Luciano Ramalho détaille le modèle de données de Python, les décorateurs, les classes, l'héritage et ouvre sur la métaprogrammation. Si vous ne devez lire qu'un seul livre, lisez celui-là !
* [Hitchhiker's guide to Python](https://amzn.to/2pZ6AIc) : Manuel moins avancé que le précédent mais très bien pour comprendre l'écosystème de Python ainsi que les bonnes pratiques. A noter qu'il est [disponible gratuitement en ligne](http://docs.python-guide.org/en/latest/).

### 💻  Sites

* [Sam et Max](http://sametmax.com/), un blog en français à la ligne éditoriale très assumée. Les articles sont de qualité et vous y trouverez de nombreuses astuces.
* [Mouse versus Python](http://www.blog.pythonlibrary.org/)
* [Inside the head of Pydanny](https://www.pydanny.com/), par un des auteurs de Two Scoops of Django, un des livres de référence sur ce framework.
* [Doug Hellmann](https://doughellmann.com/blog/)

### 🎥  Vidéos

* Toutes les vidéos des conférences PyCon. En français, vous pouvez regarder celles de la [PyCon à Rennes en 2016](https://www.pycon.fr/2016/pages/programme.html). En anglais, [voici la chaîne de celle de 2016](https://www.youtube.com/channel/UCwTD5zJbsQGJN75MwbykYNw).

### 🍻  Projets open source pour s'entraîner

Vous souhaitez aller plus loin ? Rien de tel que de coder pour cela ! Voici quelques liens listant des projets open source :

* [Projets en Python](https://wiki.python.org/moin/PythonProjects)
* [Les projets les plus consultés sur Github en Python](https://github.com/trending/python)
* [11 ressources pour apprendre et enseigner Python](https://opensource.com/education/16/4/teaching-python-and-more-with-oer)
* [Projet open source : développer la vision par ordinateur en Python](http://www.pyimagesearch.com/)
* [Projet open source : ressources pour programmer des robots en Python avec Arduino](https://playground.arduino.cc/Interfacing/Python)