Introduction à Python

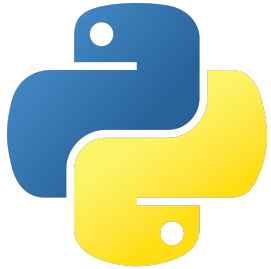
CSC 4538

Julien Romero

Introduction

Python

- Un langage (très) populaire dans la communauté Data Science et machine learning
- Actuellement en version 3.12 (évitez Python 2.X)
- Flexible et facile d'utilisation
  - Mais il faut bien en connaître les rouages pour ne pas faire n'importe quoi
- Langage interprété
  - Différent de Java et C
  - Conséquences :
    - Les erreurs apparaissent souvent à l'exécution
    - Python est lent car pas d'optimisation à la compilation



Programmer en Python

Python est un programme que l'on peut lancer en ligne de commande

```
python mon_programme.py
# Avec un numéro de version
python3 mon_programme.py
```

Sans argument, nous avons le mode interactif de Python

En général, on utilise un IDE qui facilite les choses :

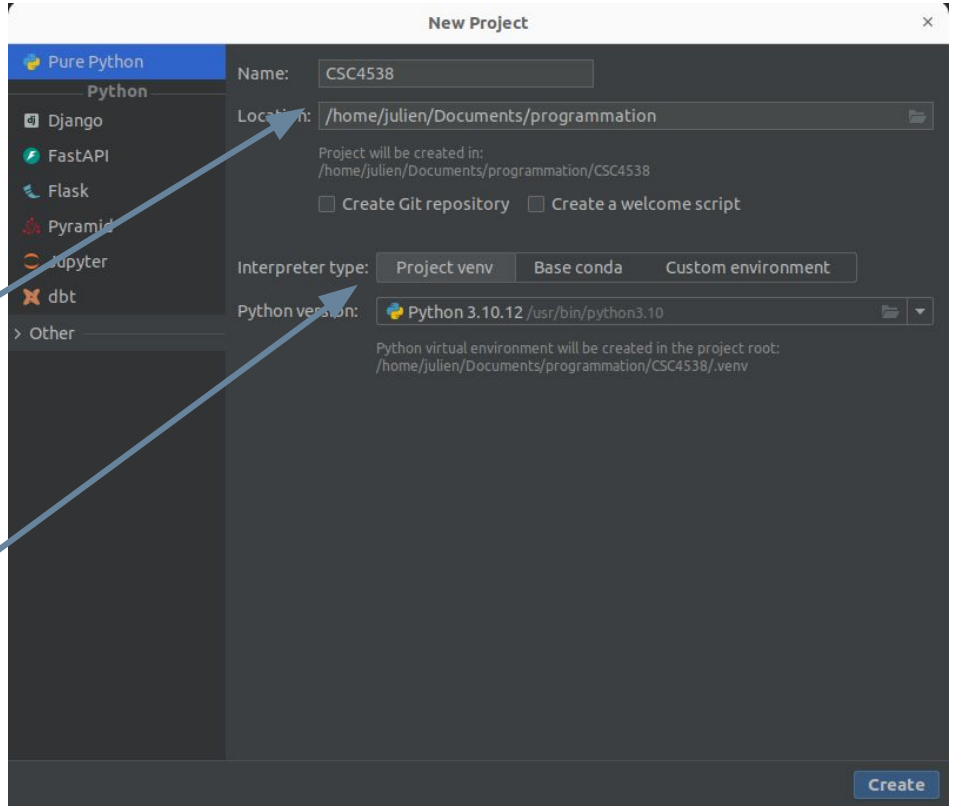
- Pycharm (recommandé dans ce cours)
- VSCode

Pycharm

Créer nouveau projet :
File>New Project

Le nom du projet
(sans espace !)

On utilise un
environnement
virtuel



Installation de bibliothèques en Python

Les bibliothèques sont essentielles en programmation pour facilement avoir accès à des fonctionnalités avancées

En Python, deux gestionnaires de paquets :

- Pip (utilisé dans ce cours)
- Conda

Installation de bibliothèque très simple :

```
pip install nom_bibliothèque
pip install nom_bibliothèque==numéro_version
```

Gestion des bibliothèques dans un projet

- Un projet définit toutes les bibliothèques nécessaires dans un fichier `requirements.txt`
  - Un nom de bibliothèque par ligne, avec potentiellement un numéro de version

```
# Liste de bibliothèque simple
pytest
Torch
# Bibliothèques avec une version
pyformlang == 1.0.9
transformers >= 4.1.1
```

- Installation avec pip :

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

Environnements virtuels

- Comment faire si nous avons plusieurs projets avec des bibliothèques différentes et des versions différentes ?
- On crée un environnement de travail local séparé de l'environnement global : un environnement virtuel
- En Python, la bibliothèque `venv` permet de créer un environnement virtuel

```
python -m venv /path/to/new/virtual/environment
# En général, on utilise :
python -m venv venv
```

- Création d'un nouveau dossier qui contient les bibliothèques de notre projet

Utilisation d'un environnement virtuel

- Pour dire à notre terminal d'utiliser l'environnement virtuel :

```
source venv/bin/activate
# On vérifie le Python utilisé
which python
# Le chemin doit pointer vers notre venv
# Désactivation
deactivate
```

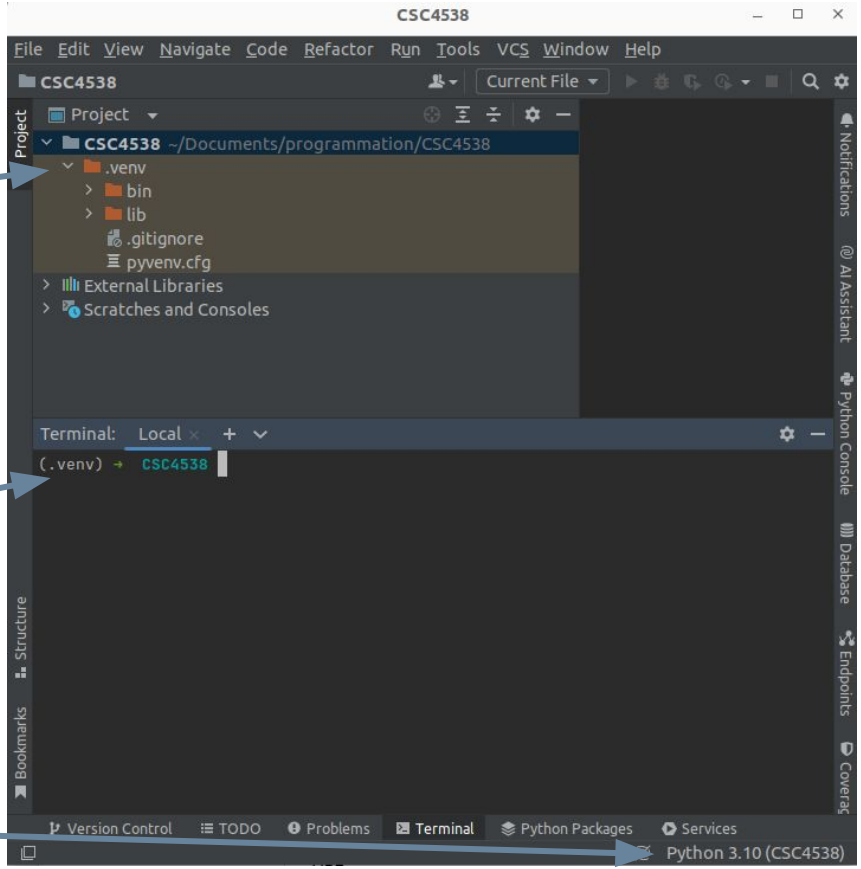
- Automatique si notre IDE est bien configuré

Environnement virtuel dans Pycharm

Le dossier de notre environnement virtuel

Notre terminal est par défaut avec le bon venv

On peut changer l'environnement utilisé



Fondamentaux

Préambule

- Python est un **langage typé** dynamiquement
  - Les types sont vérifiés à l'exécution
- Attention ! Python a des types, même s'ils sont un peu cachés !
- En Python, un retour à la ligne signifie la fin d'une instruction
  - Sauf dans des case précis, en particulier en utilisant `\`
  - Bien qu'il soit possible de finir une instruction avec un `;` comme en Java, on ne le fait jamais

Déclaration de variables

```
nom_variable = valeur
```

Par rapport à Java, on ne déclare pas le type de la variable. Par contre, on peut y accéder:

```
type(nom_variable)
```

On peut afficher une variable avec :

```
print(nom_variable)
```

Déclaration simultanée de variables

- On peut attribuer d'un seul coup plusieurs variables en affectant les valeurs d'une séquence (voir plus bas). Par contre, la séquence doit avoir exactement le bon nombre d'éléments

```
x, y, z = [1, 2, 3]
# x = 1, y = 2, z = 3
x, y = "ab"
# x = "a", y = "b"
x, y = "ABC"  # Error
```

Les types - Booléens

- Valeurs : `True` et `False` (attention à la majuscule !)
- Opérateurs classiques en toutes lettres : `and`, `or`, `not`
- Certains objets et structures ont une valeur booléenne associée
  - 0, et structures vides sont `False`

```
# Litéraux
True
False
# Opérations
True and False  # False
True or False  # True
not True  # False
# Transformations
not []  # Negation of the empty list is True!
```

Les types - Nombres

- Un type entier (`int`) et un type flottant (`float`)
- Notation exposant (float) : 1e10
- Opérations : `+`, `-`, `/`, `//` (division entière), `*`, `%`, `**` (puissance)
- Fonctions prédéfinies : `abs(x)` (valeur absolue), `int(x)` (conversion en entier), `float(x)` (conversion en flottant)
- Fonctions avancées dans la bibliothèque `math` (installée par défaut)
  - `floor(x),ceil(x)`

```
1 + 5.0    # 6.0

int(5.5)   # 5

int("5")   # 5
```

Chaînes de caractères

- Les strings en Python sont de type `str`. On les définit :
  - Avec des simples guillets ('mon texte'). Attention ! Différent de Java
  - Avec des guillemets doubles ("my text")
  - Avec des triple guillemets simples ('''my text'''). Possibilité de revenir à la ligne dans les guillemets.
    - Souvent utilisé pour faire des commentaires sur plusieurs lignes
  - Avec des triple guillemets doubles ("""my text"""). Idem
- Opérations à connaître : `s.find` (trouve une sous-string), `s.lower` (tout en minuscule), `s.upper` (tout en majuscule), `s.strip` (enlève des caractères au début et à la fin de la string), `s.split` (coupe la string en un tableau de sous-string suivant un délimiteur), `s.replace(ancien, nouveau)` (remplace `ancien` par `nouveau` dans `s`)

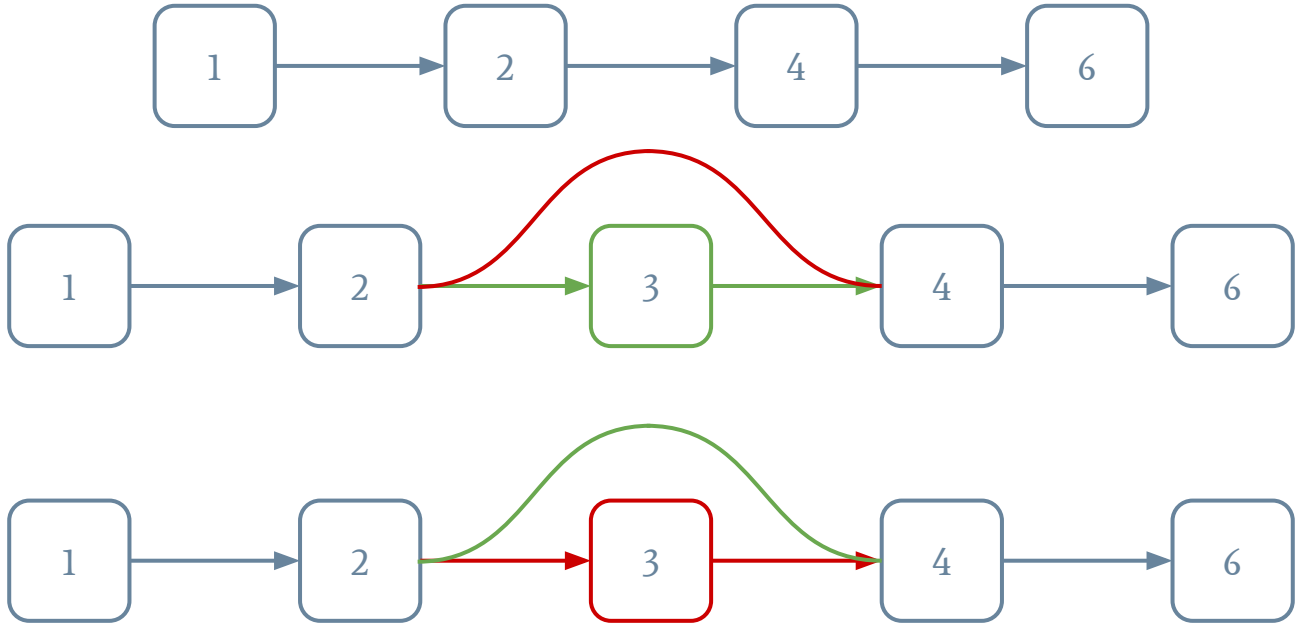
Chaînes de caractères - Exemple

```
"This is " + "Sparta!"  # Concaténation

"WolOLO".lower()  # "wololo"

"Where is Wally?".find("Wally")  # 9, l'index

"This is a sentence".split(" ")

# ['This', 'is', 'a', 'sentence']

"    White    ".strip()  # "White"

"Hello".replace("H", "h")

# 'hello'
```

Les listes - Rappels

- Une liste est une structure de donnée permettant de facilement parcourir les éléments et de rajouter des éléments
- La liste chaînée est constituée de nœuds contenants une valeur et une référence vers le nœud suivant

Les listes - Rappels



Les listes

- Les tableaux ne sont pas présents par défaut. À la place, on utilise des listes (de type `list`)
- Une liste peut contenir des éléments de plusieurs types (différent de Java)
- Déclaration avec une crochets, et les éléments sont séparés par des virgules
- Opérations : `len(l)` (nombre d'éléments), `l1 + l2` (concaténation), `l.append(element)` (ajout d'un élément à la fin de la liste), `l[i]` (obtient l'élément à la position i), `l[i:j]` (tous les éléments entre i inclu et j exclu), `x in l` (vérifie si x est dans l), `x not in l` (vérifie si x n'est pas dans l), `list(x)` (conversion en liste si possible)

Les types séquences

- La plupart des opérations présentes pour les listes fonctionnent aussi pour d'autres types "séquence" :
  - Les strings
  - Les tuples : structure immuable déclarée avec des éléments entre parenthèses
  - Les ranges : représentation d'entiers entre deux limites
    - `range(x)` : tous les éléments entre 0 et x (exclu)
    - `range(x, y)` : tous les éléments entre x (inclu) et y (exclu)
    - `range(x, y, z)` : tous les éléments entre x (inclu) et y (exclu) avec un pas de z

Exemple

```
l = [1, "2", 3.3]
l[2]  # 3.3
l[0:2]  # [1, "2"]
l[-1]  # 3.3, les indices négatifs commencent de la fin
l[-1] = 3  # On peut changer les valeurs
l.append(4)
print(l)  # [1, "2", 3.3, 4]
len(l)  # 3
"2" in l  # True
2 not in l  # True
"2" in "123456"  # True
t = (1, 2, "3")
t[0]  # 1
t[0] = 0  # Erreur, les tuples sont immuables
range(3)  # 0, 1, 2
range(1, 3)  # 1, 2
list(range(1, 5, 2))  # 1, 3
```

Les sets

- Les sets permettent de représenter un ensemble de valeur (pas de duplication) et de pouvoir rapidement ajouter un élément et vérifier la présence d'un élément
- Définit avec des accolades `{ }`
- Opérations : `x in s`, `x not in s`, `len(s)`, `s1.issubset(s2)`, `s1.union(s2)`, `s1.difference(s2)`, `s1.intersection(s2)`, `s.remove(x)`, `set(x)` (conversion en set)

Les sets - Example

```
s0 = {1, 2, "3", 2}
# is equivalent to
s0 = set([1, 2, "3", 2])


len(s0)  # 3, the element 2 is saved only once
2 in s  # True


s1 = {2, 5}
s1.union(s0)  # {1, 2, "3", 5}
s1.intersection(s0)  # {2}
s1.difference(s0)  # {5}
```

Les tables d'associations

- Les tables d'associations permettent d'associer à des clefs une valeur
- En Python, on les appelle les dictionnaires. On utilise les accolades `{}` comme les sets, mais les éléments sont de la forme `key: value`
- Opérations : `d[clef]` (obtient la valeur associée à la clef), `len(d)` (nombre de clefs), `d.keys()` (retourne les clefs), `d.values()` (retourne les valeurs), `d.items()` (retourne une collection de tuples (clef, valeur)), `clef in d` (vérifie si la clef est dans d), `d.get(key, valeur_défaut)` (obtient la valeur associée à une clef si elle existe, sinon un valeur par défaut)

Les tables d'associations - Exemples

```
d = {"France": "Paris", "Italy": "Rome"}
"France" in d  # True
len(d)  # 2
d.keys()  # "France", "Italy"
d.values()  # "Paris", "Rome"
d.items()  # ("France", "Paris), ("Italy", "Rome")
d["France"] = "Lyon"
# On change la valeur associée à France
d.get("Spain", "Inconnu")
# 'Inconnu'
```

None

- En Python, le `null` de Java s'écrit `None`
- Retourné par défaut par une fonction sans valeur de retour
- Pour savoir si une valeur est `None`, on utilise `value is None` ou `value is not None`

Structures de contrôle

Les structures de contrôle

- On retrouve les même que dans les autres langages
- En Python, l'indentation est obligatoire car elle permet de délimiter les blocs

Conditions

```
if condition:
    instruction1
    instruction2
    ...
elif condition:
    instructions
elif condition:
    instructions
else:
    instructions
```

Parenthèses non obligatoires autour de la condition

Attention à l'indentation

elif et non else if

Conditions - Exemple

```
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

Boucles - while

```
while condition:
    instructions

l = [1, 2, 3, 4]
# Une liste est considérée vraie si elle n'est pas vide
while l:
    # pop enlève et retourne le dernier élément de la liste
    print(l.pop())
```

Boucles - For

- La boucle for diffère des autres langages. En Python, on itère toujours sur une collection
- Comme les autres langages, on peut utiliser `break` pour sortir de la boucle et `continue` pour passer directement à l'itération suivante

```
for var in collection:
    instructions
```

Boucles - For - Exemple

```
# Itère sur une liste
l = [1, 2, 3, 4]
for x in l:
    print(x)

# Itère sur des entiers
for x in range(10):
    print(x)

# Itère sur tous les items d'un dictionnaire
# (clefs et valeurs ensembles)
d = {"France": "Paris",
     "Italy": "Rome"}
for key, value in d.items():
    print("The key is ", key,
          "and the value is", value)
    # Remarquez la syntaxe de print
```

Définition de fonctions

- On définit une fonction avec le mot clef `def`

```
def function_name(arg1, arg2, ...):
    # implementation
```

```
def add(first_value, second_value):
    return first_value + second_value
# Appel à la fonction
x = add(1, 2)
```

- On peut donner une valeur par défaut à un argument avec la syntaxe : `arg=valeur`

List comprehension

- Il est possible de créer une liste à partir d'une autre liste en utilisant une syntaxe très simple :

```
[expression for x in collection]
```

Ou

```
[expression for x in collection if condition]
```

Ce qui est équivalent à :

```
res = []
for x in collection:
    if condition:
        res.append(expression)
```

- Avantages : Clair, concis, et souvent plus rapide

List comprehension - Exemple

```
# Divise tous les nombres pairs par deux
[x // 2 for x in range(10) if x % 2 == 0]
# [0, 1, 2, 3, 4]

# Fonctionne aussi avec les sets...
{x // 2 for x in range(10)}
# {0, 1, 2, 3, 4}

# et avec les dictionnaires !
{x: x // 2 for x in range(10)}
# {0: 0, 1: 0, 2: 1, 3: 1, 4: 2, 5: 2, 6: 3, 7: 3, 8: 4, 9: 4}

# Définition d'une matrice nulle
l = [[0 for _ in range(10)] for _ in range(10)]
[x for row in l for x in row]  # Aplanit la matrice

# Permet aussi de créer des générateurs
# Qui ne génère une valeur que si on y accède
(x for row in l for x in row)
```

Structure d’un projet Python

La fonction `main`

- Par défaut tout le code contenu dans un fichier Python va être exécuté
  - Bien pour écrire un script
  - Problématique pour des projets plus complets où l'on importe d'autres fichiers Python
- On peut définir une "fonction main" en Python, qui en réalité est une condition

```
if __name__ == '__main__':
    # Le contenu de notre "fonction" main
```

Les bibliothèques

- Les bibliothèques sont essentielles en Python
- On peut les importer avec le mot-clé `import` ou `from … import`
- Pour les bibliothèques composées de plusieurs sous-composantes, on utilise le `.` pour sélectionner ce qui nous intéresse

```
import math
math.floor(3.3)

from math import floor
floor(3.3) # Pas besoin de préciser math

# On suit un chemin avec .
from urllib.request import URLError
```

- On peut aussi importer ses propres fichiers en enlevant le *.py* du nom

Passer des arguments à un programme

- En Java, nous récupérions les arguments directement depuis la fonction main
- En Python, on doit importer la variable contenant les arguments

```python
from sys import argv

print(argv)
# ['add.py', '1', '2']
# On doit parser les arguments manuellement
x = int(argv[1])
y = int(argv[2])

print(x + y)
# 3
```

- Pour une gestion avancée des arguments, utilisez la bibliothèque *argparse*

Programmation orientée objet

Programmation orientée objet en Python

- Python est aussi un langage orienté objet comme Java
- On déclare une classe avec le mot-clé `class`
- Pour référer à l'objet actuel, on utilise le mot-clé `self` (`this` de Java) qui est toujours donné en premier argument.
- Le constructeur de la classe se fait toujours à travers une fonction `__init__`. Cette fonction initialise les champs (pas la peine de les déclarer en dehors du constructeur).
- Les méthodes fonctionnent comme les fonctions, mais prennent `self` en premier argument

POO - Exemple

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
        # Appel interne
        self.get_fullname()

    def get_fullname(self):
        return self.name + ' ' + self.surname


if __name__ == '__main__':
    p1 = Person('John', 'Smith')
    print(p1.get_fullname())
```

Héritage

- La notion d'héritage existe aussi en Python, mais n'est pas abordée dans ce cours
- Tous les objets héritent de `object` qui définit quelques méthodes utiles :
  - `def __str__(self)` : retourne la représentation en string de l'objet
    - méthode appelée par `print(x)` ou `str(x)`
  - `def __hash__(self)` : retourne un hash de l'objet
  - `def __eq__(self, other)` : teste l'égalité entre deux objets
    - méthode appelée quand on fait `x == y`
  - `def __lt__(self, other)` : teste si `self` est inférieur à `other`
    - méthode appelée quand on fait x < y
  - `def __bool__(self)` : retourne l'interprétation booléenne de l'objet
  - `def __len__(self)` : retourne la "longueur" de l'objet
  - …

Méthodes statiques

- Python possède aussi des méthodes statiques comme Java, c'est-à-dire des méthodes ne nécessitant pas d'instance pour être appeler.
- Pour déclarer une méthode statique, il faut rajouter une annotation `@staticmethod` avant la déclaration de la méthode (qui est dans une classe), et enlever `self` des arguments.

```
class Math:
    @staticmethod
    def floor(x):
        return math.floor(x)

Math.floor(11.6)
```

Les fichiers

Lecture de fichiers

- Savoir lire et écrire des fichiers est crucial en Python car on l'utilise sous pour écrire de petits scripts qui transforment un fichier.
- Pour lire un fichier :

```
with open("mon fichier.txt") as f:
    # instructions
```

- Le bloc `open` est une syntaxe particulière en Python qui définit une zone de validité d'un objet (ici `f`) et ferme cet objet à la fin du bloc. En Java, il faut ouvrir le fichier, mais penser à le fermer plus tard (aussi possible en Python)
- `f` représente le flux de lecture du fichier. Il vient avec plusieurs fonctions, dont `f.readlines()` qui retourne une liste des lignes de `f`. Il est aussi possible d'itérer directement sur `f`.

Lecture de fichiers - Exemple

```
with open("mon_fichier.txt") as f:
    for line in f:
        # Enlève les retours à la ligne
        line = line.strip()
        print(len(line))
```

Écriture de fichier

- Pour écrire un fichier, il faut l'ouvrir en mode écriture en ajoutant le paramètre "`w`" (write) aux arguments de `open`.

```
with open("test.txt", "w") as f:
    # On peut maintenant utiliser la fonction write
    f.write("Hello")
```

- `f.write` prend un string en argument. Si l'on veut enregistrer une liste, on peut soit utiliser la fonction `join`, soit passez par un JSON.

```
"\n".join(["Hello", "world!"])
# 'Hello\nworld!'

import json
json.dumps(["Hello", "world!"])
# '["Hello", "world!"]'
```

Pour la prochaine fois...

Pour le 19 mai

- Se créer un compte sur LeetCode (https://leetcode.com/)
- Résoudre 10 problèmes en utilisant Python
  - Au moins 1 difficile (hard)
  - Au moins 4 moyens (medium) ou plus
  - Au moins 6 faciles (easy) ou plus
- M'envoyer par mail le lien de votre profil LeetCode pour que je puisse vérifier
  - https://leetcode.com/u/julien10/

En route pour le TP !