# Initiation au développement

## 7 — Fichiers

Anthony Labarre

Université Gustave Eiffel

# Plan d'aujourd'hui

1. Lecture / écriture dans un fichier
2. `enumerate` et arguments en ligne de commande

# Python pas à pas

# Lecture / écriture dans un fichier

# Rappel : la méthode `join()`

- `join()` est une méthode des chaînes de caractères ;

# Rappel : la méthode `join()`

- `join()` est une méthode des chaînes de caractères ;
- On l'utilise de la façon suivante :

  ```
  s.join(it)
  ```

  où `s` est une chaîne de caractères et `it` est un itérable contenant des chaînes de caractères ;

# Rappel : la méthode `join()`

- `join()` est une méthode des chaînes de caractères ;
- On l'utilise de la façon suivante :

  ```
  s.join(it)
  ```

  où `s` est une chaîne de caractères et `it` est un itérable contenant des chaînes de caractères ;
- Le résultat est une chaîne qui contient les mots de `it` reliés par `s` ;

# Rappel : la méthode `join()`

- `join()` est une méthode des chaînes de caractères ;
- On l'utilise de la façon suivante :

  ```
  s.join(it)
  ```

  où `s` est une chaîne de caractères et `it` est un itérable contenant des chaînes de caractères ;
- Le résultat est une chaîne qui contient les mots de `it` reliés par `s` ;
- `':'.join(['ab','cd','efg'])` → `'ab:cd:efg'` ;

# Construction de chemins vers des fichiers

On peut s'en servir par exemple pour construire des chemins vers des fichiers :

## Exemple

```
>>> "/".join(["chemin", "vers", "mon_fichier.py"])
'chemin/vers/mon_fichier.py'
>>> "\\".join(["chemin", "vers", "mon_fichier.py"])
'chemin\\vers\\mon_fichier.py'
```

# Construction de chemins vers des fichiers

On peut s'en servir par exemple pour construire des chemins vers des fichiers :

## Exemple

```
>>> "/".join(["chemin", "vers", "mon_fichier.py"])
'chemin/vers/mon_fichier.py'
>>> "\\".join(["chemin", "vers", "mon_fichier.py"])
'chemin\\vers\\mon_fichier.py'
```

Ou de manière plus portable, avec le module **os** :

## Exemple

```
>>> import os
>>> os.sep.join(["chemin", "vers", "mon_fichier.py"])
'chemin/vers/mon_fichier.py'  # sous Unix
>>> os.path.join("chemin", "vers", "mon_fichier.py")
'chemin/vers/mon_fichier.py'  # sous Unix
```

# Ouverture et fermeture d'un fichier (1)

- La **fonction** `open(...)` renvoie une variable de type `file` associée au fichier à ouvrir, qui permet ensuite de le manipuler ;

# Ouverture et fermeture d'un fichier (1)

- La **fonction** `open(...)` renvoie une variable de type `file` associée au fichier à ouvrir, qui permet ensuite de le manipuler ;
- La **méthode** `file.close()` ferme le fichier quand on n'en a plus besoin ;

# Ouverture et fermeture d'un fichier (1)

- La **fonction** `open(...)` renvoie une variable de type `file` associée au fichier à ouvrir, qui permet ensuite de le manipuler ;
- La **méthode** `file.close()` ferme le fichier quand on n'en a plus besoin ;

## Manipulation basique d'un fichier

```
fichier = open(chemin, mode)      # ouverture du fichier
# diverses manipulations
fichier.close()                   # fermeture du fichier
```

# Ouverture et fermeture d'un fichier (1)

- La **fonction** `open(...)` renvoie une variable de type `file` associée au fichier à ouvrir, qui permet ensuite de le manipuler ;
- La **méthode** `file.close()` ferme le fichier quand on n'en a plus besoin ;

## Manipulation basique d'un fichier

```
fichier = open(chemin, mode)      # ouverture du fichier
# diverses manipulations
fichier.close()                   # fermeture du fichier
```

Il faut **toujours** fermer les fichiers qu'on ouvre ! En cas d'oubli, on risque de perdre des données. Le mot-clé **`with`** nous permettra d'éviter ces oublis (voir plus loin).

# La fonction `open`

- La fonction `open` prend deux chaînes en paramètres :
  - `chemin` est le chemin vers le fichier à ouvrir (par défaut dans le répertoire courant) ;
  - `mode` décrit le mode d'utilisation du fichier ;

# La fonction `open`

- La fonction `open` prend deux chaînes en paramètres :
  - `chemin` est le chemin vers le fichier à ouvrir (par défaut dans le répertoire courant) ;
  - `mode` décrit le mode d'utilisation du fichier ;
- Les trois modes d'accès les plus communs aux fichiers sont :

# La fonction `open`

- La fonction `open` prend deux chaînes en paramètres :
  - `chemin` est le chemin vers le fichier à ouvrir (par défaut dans le répertoire courant) ;
  - `mode` décrit le mode d'utilisation du fichier ;
- Les trois modes d'accès les plus communs aux fichiers sont :
  - `'r'` : mode lecture seule (par défaut)

# La fonction `open`

- La fonction `open` prend deux chaînes en paramètres :
  - `chemin` est le chemin vers le fichier à ouvrir (par défaut dans le répertoire courant) ;
  - `mode` décrit le mode d'utilisation du fichier ;
- Les trois modes d'accès les plus communs aux fichiers sont :
  - `'r'` : mode lecture seule (par défaut)
  - `'w'` : mode écriture, le fichier est créé s'il n'existe pas, sinon il est effacé pour pouvoir y écrire

# La fonction `open`

- La fonction `open` prend deux chaînes en paramètres :
  - `chemin` est le chemin vers le fichier à ouvrir (par défaut dans le répertoire courant) ;
  - `mode` décrit le mode d'utilisation du fichier ;
- Les trois modes d'accès les plus communs aux fichiers sont :
  - `'r'` : mode lecture seule (par défaut)
  - `'w'` : mode écriture, le fichier est créé s'il n'existe pas, sinon il est effacé pour pouvoir y écrire
  - `'a'` : mode ajout : on écrit à partir de la fin du fichier

# Ouverture et fermeture d'un fichier (2)

- On peut également utiliser un bloc `with`, à la fin duquel le fichier est automatiquement fermé ;

# Ouverture et fermeture d'un fichier (2)

- On peut également utiliser un bloc **`with`**, à la fin duquel le fichier est automatiquement fermé ;

## Le bloc `with`

```
with open(chemin, mode) as fichier:          # ouverture
    # diverses manipulations
    # fermeture automatique du fichier à la fin du bloc
```

# Ouverture et fermeture d'un fichier (2)

- On peut également utiliser un bloc **`with`**, à la fin duquel le fichier est automatiquement fermé ;

## Le bloc `with`

```
with open(chemin, mode) as fichier:          # ouverture
    # diverses manipulations
    # fermeture automatique du fichier à la fin du bloc
```

- Ce code fait exactement la même chose que ce qu'on a vu avant :

# Ouverture et fermeture d'un fichier (2)

- On peut également utiliser un bloc **`with`**, à la fin duquel le fichier est automatiquement fermé ;

## Le bloc `with`

```
with open(chemin, mode) as fichier:          # ouverture
    # diverses manipulations
    # fermeture automatique du fichier à la fin du bloc
```

- Ce code fait exactement la même chose que ce qu'on a vu avant :

## Manipulation basique d'un fichier

```
fichier = open(chemin, mode)      # ouverture du fichier
# diverses manipulations
fichier.close()                   # fermeture du fichier
```

# Les objets `file`

- Un objet de type `file` :

# Les objets `file`

- Un objet de type `file` :
  - est **modifiable** : si on le fait évoluer dans une fonction, il évolue globalement ;

# Les objets `file`

- Un objet de type `file` :
  - est **modifiable** : si on le fait évoluer dans une fonction, il évolue globalement ;
  - possède une **position courante** dans le fichier, modifiée au fur et à mesure des lectures et des écritures ;

# Les objets `file`

- Un objet de type `file` :
  - est **modifiable** : si on le fait évoluer dans une fonction, il évolue globalement ;
  - possède une **position courante** dans le fichier, modifiée au fur et à mesure des lectures et des écritures ;

Le format des chemins dépend de votre OS ; par exemple, Unix ne comprendra pas `C:\\Program\ Files\\`, et Windows ne comprendra pas `/home/pierre/Documents/`

# Les objets `file`

- Un objet de type `file` :
  - est **modifiable** : si on le fait évoluer dans une fonction, il évolue globalement ;
  - possède une **position courante** dans le fichier, modifiée au fur et à mesure des lectures et des écritures ;

Le format des chemins dépend de votre OS ; par exemple, Unix ne comprendra pas `C:\\Program\ Files\\`, et Windows ne comprendra pas `/home/pierre/Documents/`

- Voir la documentation du module standard **`os`**, à utiliser pour assurer la portabilité de vos programmes de ce point de vue ;

# Lecture dans un fichier

- Le fichier doit être **ouvert en lecture** (`open("nom.txt", "r")`);

# Lecture dans un fichier

- Le fichier doit être **ouvert en lecture** (`open("nom.txt", "r")`);
- On peut lire une ligne de `fichier` avec la méthode `fichier.readline()`, qui :

# Lecture dans un fichier

- Le fichier doit être **ouvert en lecture** (`open("nom.txt", "r")`);
- On peut lire une ligne de `fichier` avec la méthode `fichier.readline()`, qui :
  1. renvoie la ligne actuelle du fichier sous forme de chaîne (vide s'il n'y a plus de ligne à lire) ;

# Lecture dans un fichier

- Le fichier doit être **ouvert en lecture** (`open("nom.txt", "r")`);
- On peut lire une ligne de `fichier` avec la méthode `fichier.readline()`, qui :
  1. renvoie la ligne actuelle du fichier sous forme de chaîne (vide s'il n'y a plus de ligne à lire) ;
  2. déplace la position courante à la ligne suivante ;

# Lecture dans un fichier

- Le fichier doit être **ouvert en lecture** (`open("nom.txt", "r")`);
- On peut lire une ligne de `fichier` avec la méthode `fichier.readline()`, qui :
  1. renvoie la ligne actuelle du fichier sous forme de chaîne (vide s'il n'y a plus de ligne à lire) ;
  2. déplace la position courante à la ligne suivante ;
- On peut répéter l'appel à `fichier.readline()` pour lire toutes les lignes une à une ;

# Lecture dans un fichier

- Le fichier doit être **ouvert en lecture** (`open("nom.txt", "r")`);
- On peut lire une ligne de `fichier` avec la méthode `fichier.readline()`, qui :
  1. renvoie la ligne actuelle du fichier sous forme de chaîne (vide s'il n'y a plus de ligne à lire) ;
  2. déplace la position courante à la ligne suivante ;
- On peut répéter l'appel à `fichier.readline()` pour lire toutes les lignes une à une ;

## Exemple (`read_line.py`)

```
if __name__ == "__main__":
    with open('filtre.py') as fichier:
        ligne = None
        while ligne != '':
            ligne = fichier.readline()
            print(ligne)
```

# Solution alternative

- Un fichier peut aussi être vu comme un **itérable** de lignes ;

# Solution alternative

- Un fichier peut aussi être vu comme un **itérable** de lignes ;
- Cela permet de très facilement lire les lignes de `fichier` ;

# Solution alternative

- Un fichier peut aussi être vu comme un **itérable** de lignes ;
- Cela permet de très facilement lire les lignes de `fichier` ;

## Exemple (`iterable.py`)

```
if __name__ == "__main__":
    with open('iterable.py') as fichier:
        for ligne in fichier:
            print(ligne)
```

# Solution alternative

- Un fichier peut aussi être vu comme un **itérable** de lignes ;
- Cela permet de très facilement lire les lignes de `fichier` ;

## Exemple (`iterable.py`)

```
if __name__ == "__main__":
    with open('iterable.py') as fichier:
        for ligne in fichier:
            print(ligne)
```

Que l'on utilise `readline()` ou le format d'itérable, les lignes renvoyées conservent le caractère `'\n'` à la fin. On peut l'enlever avec l'instruction `ligne = ligne[:-1]` (cf prochain cours)

# Les fichiers sont des itérables particuliers

On peut itérer sur les fichiers... mais une seule fois !

L'instruction `iterable(fichier)` place les lignes d'un fichier dans un itérable, mais ne fonctionne qu'une seule fois :

## Exemple

```
>>> fichier = open("trois_mots.txt")
>>> set(fichier)  # stocke les mots du fichier dans un ensemble
{"oui\n", "non\n", "peut-être\n"}
>>> len(set(fichier))  # ne marche plus!
0
>>> fichier.close()
>>> fichier = open("trois_mots.txt")
>>> set(fichier)  # remarche
{"oui\n", "non\n", "peut-être\n"}
```

Attention, `len(fichier)` ne fonctionne pas !

# Les fichiers sont des itérables particuliers

On peut itérer sur les fichiers... mais une seule fois !

De la même manière, `chaine in fichier` teste l'appartenance, mais ne fonctionne qu'une seule fois :

## Exemple

```
>>> fichier = open("trois_mots.txt")
>>> "oui\n" in fichier  # vérifie l'appartenance d'une LIGNE
True
>>> "oui\n" in fichier  # ne marche plus
False
>>> fichier.close()
>>> fichier = open("trois_mots.txt")
>>> "oui\n" in fichier  # remarche
True
```

# Les fichiers sont des itérables particuliers

- Ces remarques sont aussi valables pour `file.readline()` ;
- Une fois le parcours du fichier fini, il faut le fermer puis le rouvrir pour recommencer ;
- Ceci est une autre bonne raison pour éviter les lectures multiples et préférer le chargement du fichier en mémoire ;

# Méthodes de lecture

Vous rencontrerez peut-être en pratique d'autres méthodes de lecture. Voici un récapitulatif :

- `file.read()` : lit et renvoie tout le fichier (chaîne) ;

Dans tous les cas, chaque lecture fait avancer le curseur !

# Méthodes de lecture

Vous rencontrerez peut-être en pratique d'autres méthodes de lecture. Voici un récapitulatif :

- `file.read()` : lit et renvoie tout le fichier (chaîne) ;
- `file.read(n)` : lit et renvoie les `n` premiers caractères (chaîne) ;

Dans tous les cas, chaque lecture fait avancer le curseur !

# Méthodes de lecture

Vous rencontrerez peut-être en pratique d'autres méthodes de lecture. Voici un récapitulatif :

- `file.read()` : lit et renvoie tout le fichier (chaîne) ;
- `file.read(n)` : lit et renvoie les `n` premiers caractères (chaîne) ;
- `file.readline()` : lit et renvoie une ligne (chaîne) ;

Dans tous les cas, chaque lecture fait avancer le curseur !

# Méthodes de lecture

Vous rencontrerez peut-être en pratique d'autres méthodes de lecture. Voici un récapitulatif :

- `file.read()` : lit et renvoie tout le fichier (chaîne) ;
- `file.read(n)` : lit et renvoie les `n` premiers caractères (chaîne) ;
- `file.readline()` : lit et renvoie une ligne (chaîne) ;
- `file.readlines()` : lit et renvoie toutes les lignes (liste de chaînes) ;

Dans tous les cas, chaque lecture fait avancer le curseur !

# Écriture dans un fichier

- Pour écrire dans un fichier, il faut l'ouvrir en **écriture** `'w'` ou en **ajout** `'a'` ;

# Écriture dans un fichier

- Pour écrire dans un fichier, il faut l'ouvrir en **écriture** `'w'` ou en **ajout** `'a'` ;
- Pour écrire la chaîne `s` dans le fichier `f`, on utilise la méthode `f.write(s)` ;

# Écriture dans un fichier

- Pour écrire dans un fichier, il faut l'ouvrir en **écriture** `'w'` ou en **ajout** `'a'` ;
- Pour écrire la chaîne `s` dans le fichier `f`, on utilise la méthode `f.write(s)` ;
- Attention, contrairement à `print()`, cela ne rajoute pas un saut de ligne à la fin ;

# Écriture dans un fichier

- Pour écrire dans un fichier, il faut l'ouvrir en **écriture** `'w'` ou en **ajout** `'a'` ;
- Pour écrire la chaîne `s` dans le fichier `f`, on utilise la méthode `f.write(s)` ;
- Attention, contrairement à `print()`, cela ne rajoute pas un saut de ligne à la fin ;

## Exemple

```
if __name__ == "__main__":
    with open('tmp', 'w') as fichier:
        for i in range(10):
            fichier.write('ligne ' + str(i) + '\n')
```

# Manipulation de plusieurs fichiers

- On peut bien sûr ouvrir plusieurs fichiers à la fois :

# Manipulation de plusieurs fichiers

- On peut bien sûr ouvrir plusieurs fichiers à la fois :

## Exemple

```
fichier_1 = open("truc_à_lire")
fichier_2 = open("truc_à_écrire", "w")
# ...
fichier_1.close()
fichier_2.close()
# ...
```

# Manipulation de plusieurs fichiers

- On peut bien sûr ouvrir plusieurs fichiers à la fois :

## Exemple

```
fichier_1 = open("truc_à_lire")
fichier_2 = open("truc_à_écrire", "w")
# ...
fichier_1.close()
fichier_2.close()
# ...
```

- Avec **`with`**, on doit tout déclarer après le **`with`** :

# Manipulation de plusieurs fichiers

- On peut bien sûr ouvrir plusieurs fichiers à la fois :

## Exemple

```
fichier_1 = open("truc_à_lire")
fichier_2 = open("truc_à_écrire", "w")
# ...
fichier_1.close()
fichier_2.close()
# ...
```

- Avec **with**, on doit tout déclarer après le **with** :

## Exemple

```
with open("truc_à_lire") as fichier_1, \
     open("truc_à_écrire", "w") as fichier_2, ... :
    # ...
```

# Manipulation de plusieurs fichiers

- Si on a beaucoup de fichiers à ouvrir, une solution plus ergonomique et plus lisible peut consister à stocker ces fichiers dans une liste, ce qui facilite les fermetures :

# Manipulation de plusieurs fichiers

- Si on a beaucoup de fichiers à ouvrir, une solution plus ergonomique et plus lisible peut consister à stocker ces fichiers dans une liste, ce qui facilite les fermetures :

## Exemple

```
mes_fichiers = [
    open("truc_à_lire"), open("truc_à_écrire", "w"), ...
]
# ...
for fichier in mes_fichiers:
    fichier.close()
```

# Méthodes d'écriture

Vous rencontrerez peut-être en pratique d'autres méthodes d'écriture. Voici un récapitulatif (attention, les `\n` ne sont jamais inclus !) :

- `file.write(chaine)` : écrit le contenu de la chaîne ;

Dans tous les cas, chaque écriture fait avancer le curseur !

# Méthodes d'écriture

Vous rencontrerez peut-être en pratique d'autres méthodes d'écriture. Voici un récapitulatif (attention, les `\n` ne sont jamais inclus !) :

- `file.write(chaine)` : écrit le contenu de la chaîne ;
- `file.writelines(liste)` : écrit chaque chaîne de la liste ;

Dans tous les cas, chaque écriture fait avancer le curseur !

# Méthodes d'écriture

Vous rencontrerez peut-être en pratique d'autres méthodes d'écriture. Voici un récapitulatif (attention, les `\n` ne sont jamais inclus !) :

- `file.write(chaine)` : écrit le contenu de la chaîne ;
- `file.writelines(liste)` : écrit chaque chaîne de la liste ;
- `file.flush()` : vide le buffer d'écriture ($\Rightarrow$ écrit "réellement" les données dans le fichier) ;

Dans tous les cas, chaque écriture fait avancer le curseur !

# Accès aux fichiers

L'accès aux fichiers est lent ! Évitez les parcours inutiles, et chargez leur contenu en mémoire.

## Version 1 (environ 1s)

```
for perso in personnages:
    with open("dumas-3-mousquetaires.txt") as donnees:
        for ligne in donnees:
            for mot in ligne.split():
                nom = mot.strip().lower()
                if nom == perso:
                    frequences[nom] += 1
```

## Version 2 (environ 0.2s)

```
with open("dumas-3-mousquetaires.txt") as donnees:
    for ligne in donnees:
        for mot in ligne.split():
            nom = mot.strip().lower()
            for perso in personnages:
                if nom == perso:
                    frequences[nom] += 1
```

# Transtypage

Les fichiers étant des itérables, on peut facilement les transtyper en d'autres itérables.

## Exemple

Pour convertir le contenu d'un fichier en une liste de chaînes, on peut simplement écrire ceci :

```
with open(chemin) as fichier:
    contenu = list(fichier)
```

Bien entendu, chaque ligne de la liste se terminera par un `\n`.

# Python pas à pas

## `enumerate` et arguments en ligne de commande

# `enumerate`

- La fonction `enumerate` numérote les éléments d'un itérable ;

# `enumerate`

- La fonction `enumerate` numérote les éléments d'un itérable ;
- Elle renvoie elle-même un itérable ;

# enumerate

- La fonction `enumerate` numérote les éléments d'un itérable ;
- Elle renvoie elle-même un itérable ;

## Exemple

```
>>> mots = ["bonjour", "tout", "le", "monde"]
>>> for position, chaine in enumerate(mots):
...     print("mot", position, ":", chaine)
...
mot 0 : bonjour
mot 1 : tout
mot 2 : le
mot 3 : monde
>>> # même résultat que:
>>> for position in range(len(mots)):
...     print("mot", position, ":", mots[position])
```

# enumerate

- La fonction `enumerate` numérote les éléments d'un itérable ;
- Elle renvoie elle-même un itérable ;

## Exemple

```
>>> mots = ["bonjour", "tout", "le", "monde"]
>>> for position, chaine in enumerate(mots):
...     print("mot", position, ":", chaine)
...
mot 0 : bonjour
mot 1 : tout
mot 2 : le
mot 3 : monde
>>> # même résultat que:
>>> for position in range(len(mots)):
...     print("mot", position, ":", mots[position])
```

- `enumerate(iterable, n)` commence l'énumération à la valeur `n` ;

# Arguments en ligne de commande

- On peut passer des arguments aux programmes Python, ce qui permet d'éviter les interactions avec l'utilisateur ;

# Arguments en ligne de commande

- On peut passer des arguments aux programmes Python, ce qui permet d'éviter les interactions avec l'utilisateur ;
- L'option la plus simple consiste à utiliser `sys.argv`, une liste stockant les arguments passés au programme :

# Arguments en ligne de commande

- On peut passer des arguments aux programmes Python, ce qui permet d'éviter les interactions avec l'utilisateur ;
- L'option la plus simple consiste à utiliser `sys.argv`, une liste stockant les arguments passés au programme :

## Exemple

```
import sys

if __name__ == "__main__":
    print("Je ne fais rien d'autre qu'afficher mes "
          "paramètres:\n")
    for position, parametre in enumerate(sys.argv):
        print("\tparamètre", position, ":", parametre)
```

# Arguments en ligne de commande

- On peut passer des arguments aux programmes Python, ce qui permet d'éviter les interactions avec l'utilisateur ;
- L'option la plus simple consiste à utiliser `sys.argv`, une liste stockant les arguments passés au programme :

## Exemple

```
import sys

if __name__ == "__main__":
    print("Je ne fais rien d'autre qu'afficher mes "
          "paramètres:\n")
    for position, parametre in enumerate(sys.argv):
        print("\tparamètre", position, ":", parametre)
```

- Attention :

# Arguments en ligne de commande

- On peut passer des arguments aux programmes Python, ce qui permet d'éviter les interactions avec l'utilisateur ;
- L'option la plus simple consiste à utiliser `sys.argv`, une liste stockant les arguments passés au programme :

## Exemple

```
import sys

if __name__ == "__main__":
    print("Je ne fais rien d'autre qu'afficher mes "
          "paramètres:\n")
    for position, parametre in enumerate(sys.argv):
        print("\tparamètre", position, ":", parametre)
```

- Attention :
  - en position 0, on a toujours le nom du programme ;

# Arguments en ligne de commande

- On peut passer des arguments aux programmes Python, ce qui permet d'éviter les interactions avec l'utilisateur ;
- L'option la plus simple consiste à utiliser `sys.argv`, une liste stockant les arguments passés au programme :

## Exemple

```
import sys

if __name__ == "__main__":
    print("Je ne fais rien d'autre qu'afficher mes "
          "paramètres:\n")
    for position, parametre in enumerate(sys.argv):
        print("\tparamètre", position, ":", parametre)
```

- Attention :
  - en position 0, on a toujours le nom du programme ;
  - tous les paramètres sont des **chaînes** ;