



Les notions abordées

Au cours de ce TD vous allez revoir des éléments de base mais surtout, aborder des notions nouvelles :

- Les arguments de ligne de commandes.
- La manipulation de fichiers textes et TXT.
- Des appels systèmes qui feront appel à la notion de chemin relatif et absolu
- L'exécution des scripts dans un IDE et dans une fenêtre CLI (« *cmd.exe* » ou même « *Powershell* »).

Introduction : les arguments de ligne de commandes

Une ligne de commande commence par le nom du script et peut être suivi par des arguments.

```
$> script.py 553 -v -f fichier.txt
```

Dans l'exemple donné ci-dessus, vous avez 5 arguments :

- **script.py** (argument 0 : le nom du script)
- **553** (argument 1)
- **-v** (argument 2)
- **-f** (argument 3)
- **fichier.txt** (argument 4)

Vous retrouvez ce type d'appel souvent sous Linux mais aussi sous « *Powershell* » rendant l'appel des scripts beaucoup plus simple. Ils rendent l'appel à des programmes et des scripts beaucoup plus rapides et plus ergonomique. Ils sont doublés généralement avec des options et d'une aide. Les programmes peuvent être ensuite appelés par des scripts ou d'autres programmes.

En python, la mise en œuvre du contenu de la ligne de commande se fait avec le module « **sys** » qui devra être importer préalablement.

Ce module dispose d'un attribut « **argv** » de type liste qui contient l'ensemble des arguments de la ligne de commande.

L'analyse de la longueur de la liste (« **len (...)** ») permet de connaître le nombre d'arguments présents après le nom du script...

Note : Cette liste contient au minimum toujours un élément : le nom du script. Autrement dit, si le script est suivi d'un argument, ce dernier est le second élément de la liste...

Exemples :

```
import sys

print(sys.argv[0])

for i in range(len(sys.argv)):      # Boucle avec index...
    print(sys.argv[i])

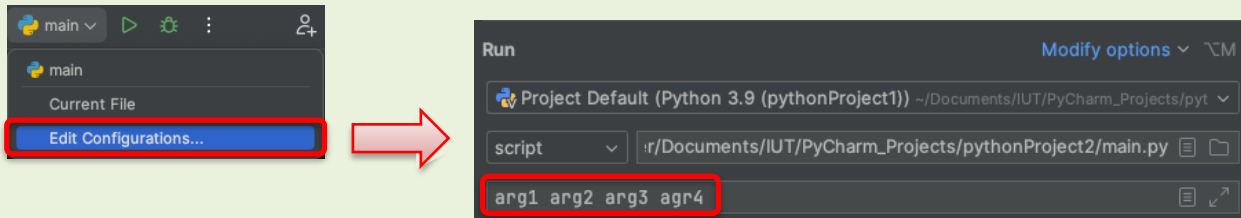
for argument in sys.argv:          # Boucle avec élément de la liste...
    print(argument)
```




Exercice 1 : arguments.py

Codez le script « `arguments.py` » qui affiche tous les arguments (autres que le nom du script) qui sont passés sur la ligne de commande. Si aucun argument ne figure après le nom du script, un message doit être affiché.

Note : Vous pouvez ajouter des arguments à la ligne de commande dans « PyCharm » avec :



Note : La solution donnée ci-dessus n'est pas très pratique lorsque l'on veut rapidement changer les arguments. Il est alors préférable de travailler en CLI.

Vous pouvez aussi lancer une interface de CLI sous « Pycharm » dans la fenêtre du « Terminal » avec l'icône : 

Dans ce cas, il faut que le répertoire courant du « Terminal » soit celui où vous avez sauvegardé votre script. Dans le cas contraire, rejoindez-le avec les commandes « DOS » usuelles (`cd.. / cd nom_rep`)



Remarque : Vous pouvez aussi faire cela dans l'interface CLI de Powershell (tapez : `pwsh` ou `powershell`)...

Le résultat souhaité est :

```
(base) -->python ./arguments.py

Aucun argument après le nom du script : ./arguments.py
(base) -->python ./arguments.py arg1 arg2 arg3 arg4

--> arg1
--> arg2
--> arg3
--> arg4
```

Le module « **pathlib** » : rappels...

Python propose la bibliothèque « **pathlib** » qui va permettre de manipuler les chemins de fichiers sur les plateformes Windows et MacOS/Linux. Elle s'importe avec :

```
from pathlib import Path
```

L'idée consiste d'associer ensuite un objet de type « **Path** » à toute chaîne de caractères qui représente un nom de fichier, un chemin (relatif ou absolu) dans une arborescence d'OS... Tout le travail qui doit s'opérer sur les noms de chemins et de fichiers se fait ensuite sur ces objets « **Path** ».

Lorsqu'un affichage sous forme de texte d'un chemin ou/et d'un nom de fichier doit se faire, un « **casting** » (conversion) en string sera nécessaire. Et suivant l'OS sur lequel est interprété le code Python, l'affichage se fera automatiquement au bon format...



Méthode	Description	Exemples
<code>resolve()</code>	Retourne un objet « Path » où le chemin complet a été ajouté au nom du fichier si ce n'était pas déjà le cas...	<code>f_path = f_path.resolve()</code> <code>--> /users/steve_jobs/nom_fichier.txt</code> <code>--> C:\users\bill_gates\nom_fichier.txt</code>
<code>name</code>	Nom du fichier.	<code>f_path.name</code> <code>--> nom_fichier.txt</code>
<code>suffix</code>	Suffixe du fichier.	<code>f_path.suffix</code> <code>--> .txt</code>
<code>parent</code>	Chemin absolu du fichier.	<code>f_path = f_path.resolve()</code> <code>f_path.parent</code> <code>--> /users/steve_jobs</code> <code>--> C:\users\bill_gates</code>
<code>exists()</code>	Retourne un booléen indiquant si le fichier ou dossier existe ?	<code>f_path.exists()</code> <code>--> True</code>
<code>is_file()</code>	Retourne un booléen indiquant si c'est un fichier ?	<code>f_path.is_file()</code> <code>--> True</code>
<code>is_dir()</code>	Retourne un booléen indiquant si c'est un dossier ?	<code>f_path.is_dir()</code> <code>--> False</code>

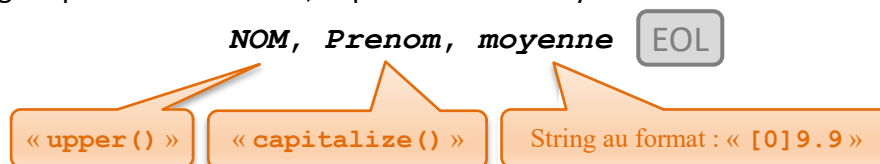
Exemples :

```
from pathlib import Path

fichier = "nom_fichier.txt" # Chaîne de caractère du nom du fichier...
f_path = Path(fichier)      # Objet Path associé à ce fichier
f_path = f_path.resolve()   # Ajout du chemin complet au nom de fichier
print(str(f_path))          # Affichage de la string de l'objet Path.
                             --> /users/steve_jobs/nom_fichier.txt      # MacOS/Linux
                             --> C:\users\bill_gates\nom_fichier.txt    # Windows
```

Exercice 2 : db_moy.py

L'objectif est de créer un script qui va ajouter à un fichier texte « notes.txt » éventuellement existant, une ligne qui contient le nom, le prénom et sa moyenne de notes avec le format suivant :



L'ajout de la ligne se fait par passage d'arguments à un script dont la liste est la suivante :

- Argument 1 : le nom du fichier avec un chemin relatif ou absolu.
- Argument 2 : le nom de l'étudiant.
- Argument 3 : le prénom de l'étudiant.
- Argument 4 : une note.
- Argument 5 (facultatif) : une seconde note.
- Argument 6 (facultatif) : une troisième note.
- Etc...



La syntaxe de la ligne de commande est la suivante :

--> `python db_moy.py nom_db.txt nom prenom n1 [n2] [n3]...`

Codez le script « `db_moy.py` » qui effectue les tâches suivantes :

1. Vérifiez qu'au minimum quatre arguments sont fournis (après le nom du script). Sinon, message d'erreur...
2. Convertissez le 1^{er} argument en objet « **Path** » et ajoutez-lui le chemin complet.
3. Convertissez les 2^{ème} et 3^{ème} arguments au bon format...
4. Convertissez le 4^{ème} argument ainsi que tous les arguments qui suivent en flottant pour calculer la moyenne.

Indication : On peut placer dans une liste plusieurs arguments contigus simultanément à partir de la liste « `argv` » en faisant du « **slicing** »... Exemple : `notes = sys.argv[4:]`

5. Calculez la moyenne des notes avec le bon format de nombre.
6. Ouvrez ce fichier en écriture et ajoutez-y les données de l'étudiant sous forme d'une ligne en utilisant le code suivant :

```
with open(f_path, mode='a', newline='') as file:
    file.writelines(f"{nom}, {prenom}, {round(moyenne, 1)}\n")
    print(f"La moyenne de {prenom} {nom} a été mise dans {str(f_path)}")
```

7. Testez votre code en ajoutant plusieurs étudiants. Visualisez le fichier texte obtenu...

Note : Avec cette approche, on ne gère pas d'éventuels doublons au niveau des noms et prénoms.

Exercice 3 : `extract_log.py`

L'objectif est de créer un script qui va rechercher un mot clé dans chaque ligne du fichier « **log** » (journal d'évènement) qui se nomme « `connections.log` » et se trouve sur Moodle. Un extrait de ce « **log** » système est donné ci-dessous :

```
] INFO - Données invalides
[2025-01-20 22:33:15] INFO - Nouvelle connexion détectée
[2025-01-20 23:16:15] WARNING - Ressources limitées
[2025-01-21 00:13:15] INFO - Nouvelle connexion détectée
[2025-01-21 00:19:15] INFO - Mise à jour effectuée
[2025-01-21 00:46:15] WARNING - Réseau instable
[2025-01-21 01:43:15] INFO - Nouvelle connexion détectée
[ [2025-01-21 05:09:15] INFO - Reconnexion réussie
[2025-01-21 05:42:15] ERROR - Échec de l'authentification
[2025-01-21 06:27:15] INFO - Mise à jour effectuée
[2025-01-21 07:01:15] WARNING - Réseau instable
```

L'objectif de cet exercice est de créer un script susceptible d'extraire les évènements de ce journal qui contiennent un mot clé donné. Exemple : « **ERROR** ».

On souhaite paramétrer l'exécution de ce script avec :

- Argument 1 : le nom du fichier avec un chemin relatif ou absolu.
- Argument 2 : le mot clé à rechercher.



Codez le script « `extract_log.py` » dans le code principal en partant du squelette suivant :

```
import sys
from pathlib import Path

# Code du programme principal
```

Ce code doit réaliser les tâches suivantes :

1. Vérifiez que 2 arguments sont fournis (après le nom du script). Sinon, message d'erreur du type :
--> **usage : `extract_log.py nom_fichier.log mot_cle`**
2. Convertissez le 1^{er} argument en objet « **Path** » et ajoutez-lui le chemin complet.
3. Vérifiez que le fichier existe bien. Sinon, message d'erreur...
4. Vérifiez que le fichier est réellement un fichier et non un répertoire. Sinon, message d'erreur...
5. Vérifiez que le 2^{ème} argument est bien dans la liste des 3 catégories de messages possibles : `["ERROR", "WARNING", "INFO"]`. Sinon, message d'erreur...
6. Ouvrez le fichier en lecture.
7. Parcourez-le ligne par ligne et chercher le mot clé... Comptez les lignes où figurent le mot clé.

Note : L'affichage des lignes n'est pas demandé !!!

8. Affichez le compteur de ligne à la fin et comme ceci :

Le nombre de lignes contenant XXXXX est : nnnnn

On souhaite à présent améliorer ce code de manière à ce qu'il dispose d'un argument supplémentaire **mais facultatif**. Lorsque cet argument est présent, on souhaite afficher toutes les lignes où figurent le mot clé. Ce nouvel argument est :

- Argument 3 : « **-v** » (verbose = détaillé).

ATTENTION : Lorsque l'on a des arguments facultatifs, cela complique très vite l'organisation et l'imbrication des tests de vérifications des arguments. Il est possible de se simplifier un peu cette tâche à chaque fois que le test aboutit à une erreur en utilisant l'instruction « **exit(1)** »...

Avec cette approche, il n'est plus nécessaire de coder le « **else** »...

Exemple :

```
import sys

if len(sys.argv) < 3:    # Arguments insuffisants
    print("message d'erreur...")
    exit(1)              # Arrêt exécution

flag_verbose = False
if len(sys.argv) == 4:    # Si argument facultatif présent
    if sys.argv[3] != "-v":    # Si argument facultatif != -v
        print("message d'erreur...")
        exit(1)              # Arrêt exécution
    flag_verbose = True
mot_cle = sys.argv[2]      # Traitement des arg. non facultatifs
.....
```



Remarque : Il est de bon usage de toujours envisager un « **else** » après un « **if** ». Dans ce cas précis, comme le « **if** » aboutit un arrêt de l'exécution avec « **exit(1)** » le « **else** » devient donc implicite...



Codez un nouveau script « `extract_log_v.py` » qui s'appuie sur le code « `extract_log.py` » avec les modifications suivantes :

9. Modifiez l'analyse des arguments.
10. Modifiez les messages d'erreur qui indique l'usage du script.
11. Modifiez l'affichage suivant la présence ou non de l'argument facultatif.

Exercice 4 : `calc.py`

L'objectif est de créer une mini-calculatrice en CLI qui propose les 6 opérations de base : « + », « - », « x », « / », « // » et « % ». La syntaxe de la ligne de commande est la suivante :

--> `python calc.py nombre operateur nombre`

Codez le script « `calc.py` » qui effectue les tâches suivantes :

1. Vérifiez que trois arguments sont fournis (après le nom du script). Sinon, message d'erreur...
2. Convertissez le 1^{er} et le 3^{ème} argument en flottant.
3. Vérifiez que le 2^{ème} argument est bien dans la liste des 6 opérateurs : `["+","-","x","/","//","%"]`.
4. Effectuez le calcul.
5. Affichez le résultat.
6. Réorganisez votre code de manière à placer les étapes 1 et 2 dans la fonction « `test_io(...)` » et les étapes 3 à 5 dans la fonction « `calculer(...)` ». A vous de choisir les arguments et retours de ces 2 fonctions... Ces deux fonctions doivent être appelées dans le programme principal qui se trouve forcément en dehors de ces deux fonctions...

Indication : Lorsqu'une fonction doit retourner plusieurs éléments, il est possible de faire en Python :

```
def ma_fonction(...) :  
    .....  
    return a, b, c
```



Remarque : Les éléments sont retournés dans un « tuple »...

```
resu_a, resu_b, resu_c = ma_fonction(...)
```

On peut aussi retourner une liste, mais dans ce cas, il faut la créer explicitement dans le « `return` »...

7. Créez une fonction « `aide(msg_err)` » qui centralise l'affichage des erreurs et donne des indications à l'utilisateur avec notamment l'usage du script comme :

--> `usage : calcul.py nombre operateur nombre`

8. Ajoutez un « **docstring** » à chaque fonction et testez-le avec la fonction « **help** »...



Activité complémentaire

Suite de l'exercice 4 : calcul.py

On souhaite à présent enrichir le script « `calcul.py` » avec 2 options qui vont mettre en œuvre 3 arguments optionnels dans la CLI. La nouvelle syntaxe de la ligne de commande est la suivante :

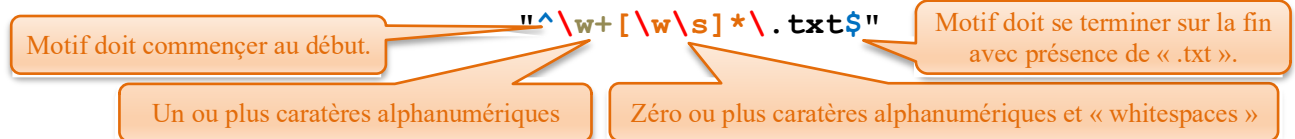
```
--> python calcul.py nb op nb [-v] [-f nom_fichier.txt]
```

Si l'argument « `-v` » (verbose = détaillé) est présent, l'affichage ne doit pas seulement donner le résultat mais afficher le détail du calcul. Exemple : `3 + 4 = 7`

Si l'argument « `-f` » (fichier) est présent, l'argument suivant doit exister et donner le nom d'un fichier dans lequel le calcul détaillé doit être ajouté (à d'éventuels calculs antérieurs). La présence de l'argument « `-f` » sans le nom de fichier doit générer un message d'erreur !

Codez un nouveau script « `calcul.py` » qui s'appuie sur le code précédent avec les modifications qui sont demandées :

1. Vérifiez que s'il y a exactement 4 arguments, le dernier est bien « `-v` ».
2. Vérifiez que s'il y a exactement 5 arguments, le 4^{ème} est bien « `-f` » et le cinquième est bien une chaîne de caractère représentant un nom de fichier texte avec le motif REGEX suivant :



3. Vérifiez que s'il y a exactement 6 arguments, on a bien « `-v` » en 4^{ème} position, « `-f` » en 5^{ème} position et un nom de fichier texte en dernière position...
4. Modifiez l'affichage du résultat.
5. Ajoutez une fonction qui écrit le calcul dans le fichier (historique des calculs...).

Variante de la suite exercice 4 : calcul_pars.py

Dans l'amélioration de l'exercice 4 que vous venez de faire, les arguments facultatifs devaient forcément être dans un ordre précis et leurs présences pouvaient se détecter grâce au nombre d'arguments... On n'a pas toujours cette chance.

Pour que le script fonctionne correctement tout en autorisant la saisie dans le désordre des arguments facultatifs, il faut deux choses :

- Les arguments facultatifs doivent toujours être étiquetés.

Note : Une étiquette d'argument commence par un « `-` » ou un « `--` »...

- Utiliser un « **parser** » (interpréteur) qui analyse les arguments. Sans cela, le code devient très vite une « usine à gaz » !!!

Remarque : L'usage veut que les arguments OBLIGATOIRES soient placés AVANT les arguments FACULTATIFS.

On peut éventuellement étiqueter les arguments obligatoires mais cela alourdit la syntaxe et la saisie de la ligne de commande...



Au final, on souhaite que toutes les lignes de commandes suivantes soient valides :

```
--> python calcul.py nb op nb
--> python calcul.py nb op nb [-v]
--> python calcul.py nb op nb [-f nom_fichier.txt]
--> python calcul.py nb op nb [-v] [-f nom_fichier.txt]
--> python calcul.py nb op nb [-f nom_fichier.txt] [-v]
```

Le code qui suit donne un exemple de mise en œuvre du « **parser** » du module « **argparse** » de Python. Étudiez en détails cet exemple pour le mettre en œuvre dans le code du script « `calcul_pars.py` », évolution du code : « `calcul.py` »...

Exemple :

```
import argparse

# Création du parseur
parser = argparse.ArgumentParser(usage = "python calcul_pars nombre1 operateur nombre2 [-v] [-f filename.txt]")

# Ajouter les arguments positionnels non facultatifs
parser.add_argument("nb1", type=float, metavar="Nombre 1", help="1er nombre (obligatoire)")
parser.add_argument("op", type=str, metavar="Opérateur", help="String (obligatoire)")
parser.add_argument("nb2", type=float, metavar="Nombre 2", help="2ème nombre (obligatoire)")

# Ajouter les arguments facultatifs
parser.add_argument("-f", type=str, metavar="Fichier", help="Nom du fichier (facultatif)")
parser.add_argument("-v", action="store_true", help="(facultatif)")

arguments = parser.parse_args() # Analyse des arguments

# Utilisation des arguments positionnels
print(f"Premier nombre : {arguments.nb1}")
print(f"Chaîne de caractères : {arguments.op}")
print(f"Deuxième nombre : {arguments.nb2}")

# Utilisation des arguments facultatifs
if arguments.f:
    print(f"Fichier source : {arguments.f}")
else:
    print("Aucun fichier source spécifié.")

if arguments.v:
    print("Mode verbeux activé.")
```

3 arguments (obligatoires) SANS étiquette (pas de « - »)...

2 arguments (facultatifs) AVEC étiquette (présence du « - »)...

On récupère directement les valeurs des arguments car il ne disposent pas d'étiquette...

On récupère l'argument qui est juste après l'étiquette « -f »...

L'étiquette « -v » se suffit à elle-même et représente un booléen. Elle n'a pas d'argument qui la complète...