Fichiers à joindre : ip_log.csv.zip / Rep_Exercice.zip

Les notions abordées

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

- La gestion des erreurs avec le mécanisme « **`try`** » … « **`except`** ».
- La manipulation de fichiers textes et CSV.
- 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).

Exercice 1 : try_except.py

Codez le script « `try_except.py` » qui :

1. Place dans une liste un nombre aléatoire (entre 10 et 20) d'entiers qui ont eux-mêmes des valeurs aléatoires entre 0 et 100.
2. Dans une boucle « *folle* », demande à l'utilisateur de donner une valeur d'index de la liste pour afficher ensuite la valeur correspondante dans la liste.

Lors du test de votre script, donnez des valeurs d'index « *en-dedans* » et « *en-dehors* » de la liste. Que se passe-t-il ?

3. Modifiez votre code pour intégrer dans la boucle « *folle* » une structure « **`try`** » … « **`except`** » qui va gérer une éventuelle « **`IndexError`** » de manière à ce que la boucle « *folle* » ne se termine jamais… Le message d'erreur doit donner l'intervalle des valeurs d'index possibles…

Toujours en testant votre code, que se passe-t-il si vous répondez à la question par un caractère non numérique ?

4. Apportez une nouvelle modification à votre code pour que la boucle « *folle* » reste « *folle* »…

Indication *: Si l'on a des doutes sur la nature de l'erreur (**IndexError**, **ValueError**, **TypeError**, **IOError**, etc…), on peut utiliser le code suivant qui affiche le type d'erreur :*

```
try:
        ......
except Exception as e:          # "Exception" générique
      print(type(e))
```

Y a bien un « **E** » majuscule…

5. Complétez votre code pour que la bonne « ***exception*** » y figure (en plus de l'« ***exception*** » générique)…

Pour mettre en œuvre une troisième « ***exception*** », on va demander à l'utilisateur un second nombre et afficher le calcul suivant :

6. Modifiez votre code…

$$\frac{\text{liste(index)}}{\text{second\_nombre}}$$

En testant votre code, donnez la valeur « 0 » au second nombre… Que se passe-t-il ?

7. Ajoutez la bonne « ***exception*** » et un message d'erreur plus explicite à votre code !

Exercice 2 : ip_log.py

L'objectif est de chercher dans un fichier « ***log*** » le nombre d'occurrence d'une adresse IP.
Il est demandé de travailler avec un très gros fichier « ***log*** » (1,5 million de lignes) au format CSV qui se trouve en format compressé sur Moodle : « `ip_log.csv.zip` ».
Téléchargez ce fichier ZIP dans le répertoire où vous stocker vos scripts Python. Puis, décompresser-le (7-Zip ou autre) dans ce même répertoire.

> ***Note*** *: N'essayer pas de la visualiser ou de l'ouvrir avec Excel… Vous allez perdre du temps !!!*

Pour ménager votre curiosité, voici un petit extrait qui illustre son format :

```
ip,timestamp,method,url,status,size
54.36.149.41,22/Jan/2019:03:56:14 +0330,GET,/filter/27,200,30577
31.56.96.51,22/Jan/2019:03:56:16 +0330,GET,/image/60844/productModel/200x200,200,5667
31.56.96.51,22/Jan/2019:03:56:16 +0330,GET,/image/61474/productModel/200x200,200,5379
40.77.167.129,22/Jan/2019:03:56:17 +0330,GET,/image/14925/productModel/100x100,200,1696
91.99.72.15,22/Jan/2019:03:56:17 +0330,GET,/product/31893/62100PR257AT,200,41483
40.77.167.129,22/Jan/2019:03:56:17 +0330,GET,/image/23488/productModel/150x150,200,2654
40.77.167.129,22/Jan/2019:03:56:18 +0330,GET,/image/45437/productModel/150x150,200,3688
40.77.167.129,22/Jan/2019:03:56:18 +0330,GET,/image/576/article/100x100,200,14776
```

Le nom du fichier ainsi que l'adresse IP à chercher doit se faire par passage d'argument au script comme :

- Argument 1 : le nom du fichier avec un chemin relatif ou absolu.
- Argument 2 : « **`-ip`** ».
- Argument 3 : l'adresse IP à rechercher

La syntaxe de la ligne de commande est la suivante :

```
--> python ip_log.py nom_fichier.csv -ip adresse_ip
```

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

1. Vérifiez que 3 arguments sont fournis (après le nom du script). Sinon, message d'erreur…
2. Vérifiez que le 2ème argument est « **`-ip`** ». Sinon, message d'erreur.
3. Récupérez l'adresse IP dans le 3ème argument.
4. Convertissez le 1er argument en objet « **`Path`** ».
5. Ouvrez ce fichier en lecture avec un « ***reader CSV*** » au sein d'une structure « **`try`** » … « **`except`** ».
6. Parcourez ligne après ligne pour rechercher l'adresse IP dans la ligne et comptabiliser les occurrences trouvées.
7. Affichez le résultat final sous la forme :

```
Nombre d'occurrences de l'adresses IP xxx.xxx.xxx.xxx : xxxxxx
```

Dans le code précédent, aucune vérification n'est faite sur la validité de l'adresse IP saisie en argument. On va compléter le code pour effectuer cette vérification.
On peut utiliser pour cela la fonction « **`ip_address(`***`string`***`)`** » du module « **`ipaddress`** ».
Si l'adresse IP (IPV4 ou IPV6) est valide, cette fonction retourne un objet de type « **`ipaddress.IPv4Address`** » ou « **`ipaddress.IPv6Address`** » qui peut être converti en string au besoin.
En revanche, li l'adresse IP n'est pas valide, une erreur est générée…

8. Créez une fonction « **`est_ip_valide(…)`** » qui admet pour argument la « ***string*** » de l'adresse IP et qui retourne un booléen.

> ***Indication*** *: Utilisez fonction « **`ip_address(string)`** » du module « **`ipaddress`** » au sein d'une structure « **`try`** » … « **`except`** »…*

9. Modifiez votre code principal pour qu'il prenne en compte cette fonction de validation.

Pour garder un historique de toutes les recherches d'adresses IP faites dans le fichier « ***log*** », on souhaite ajouter à un fichier JSON chaque nouvelle recherche.
Il est demandé d'avoir dans le fichier JSON une liste de dictionnaires comme cela :

```
[
  {"40.77.167.129": 427},
  {"40.77.167.128": 520},
  {"5.78.190.181": 27979}
]
```

10. Apportez les modifications nécessaires à votre code.

> ***Indications*** *: Ouvrir le fichier JSON en mode « `'a'` » n'est pas suffisant. Il faut lire le fichier JSON (mode « `'r'` »), modifier en Python la liste des dictionnaires et réécrire la nouvelle liste (mode « `'w'` »)…*

Le module « `pathlib` » : rappels…

La gestion des chemins et noms de fichiers doit se faire avec les outils du module « **`pathlib`** ». Le tableau qui suit fait un petit rappel :

Méthode	Description	Exemples

`resolve()`	Retourne un objet « Path » où le chemin complet a été ajouté au nom du fichier si ce n'est pas déjà le cas…	`f_path = f_path.resolve()`
		`--> /users/steve_jobs/nom_fichier.txt`
		`--> C:\users\bill_gates\nom_fichier.txt`

`name`	Nom du fichier.	`f_path.name --> nom_fichier.txt`
`suffix`	Suffixe du fichier.	`f_path.suffixe --> .txt`

`parent`	Chemin absolu du fichier.	`f_path = f_path.resolve()`
		`f_path.parent`
		`--> /users/steve_jobs`
		`--> C:\users\bill_gates`

`exists()`	Retourne un booléen indiquant si le fichier ou dossier existe ?	`f_path.exist() --> True`
`is_file()`	Retourne un booléen indiquant si c'est un fichier ?	`f_path.is_file() --> True`
`is_dir()`	Retourne un booléen indiquant si c'est un dossier ?	`f_path.is_dir() --> False`
`iterdir()`	Retourne une liste de tous les fichiers que contient le répertoire de départ…	`liste = f_path.iterdir()`

`glob("*.txt")`	Retourne une liste de tous les fichiers que qui vérifient une condition dans le répertoire de départ…	`liste_txt = f_path.glob("*.txt")`
		`liste_py = f_path.glob("*.py")`
		`liste_all = f_path.glob("*")`

`rglob("*")`	Méthode similaire à la précédente mais avec un parcours « récursif » de toute l'arborescence à partir du répertoire de départ (sous-répertoire, sous-sous-répertoire, etc…)	`liste_txt = f_path.rglob("*.txt")`
		`liste_py = f_path.rglob("*.py")`
		`liste_all = f_path.rglob("*")`

Exemple :

```
# Exemple : Recherche récursive de tous les fichiers et dossier .txt
for fichier in Path("documents").rglob("*.txt"):
    print(fichier)
```

Exercice 3 : find.py

L'objectif est de trouver tous les fichiers ayant le même suffixe dans toute une arborescence. Le résultat de cette recherche doit être placé dans un fichier CSV avec le format :

nom_fichier, chemin_absolu, taille EOL

Le script doit fonctionner avec les arguments suivants :

- Argument 1 : « **-d** » pour indiquer que l'argument suivant est le nom du répertoire de recherche.
- Argument 2 : le nom répertoire de base pour la recherche (chemin relatif ou absolu).
- Argument 3 : « **-s** » pour indiquer que l'argument suivant est le suffixe des fichiers à chercher.
- Argument 4 : le suffixe des fichiers à chercher. Exemple : « **.txt** ».
- Argument 5 : « **-o** » pour indiquer que l'argument suivant est le nom du fichier CSV.
- Argument 6 : Le nom du fichier CSV à utiliser pour stocker le résultat de la recherche.

Lorsqu'un script attend un nombre important d'arguments (c'est un peu le cas de cet exercice), l'analyse en Python de ces derniers est très vite complexe. Il est alors beaucoup plus pratique de mettre en œuvre un « ***parser*** » via le module « **argparse** » de Python.

Le « ***parser*** » se met en œuvre avec le code suivant :

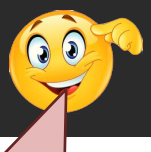
```
import argparse

parser = argparse.ArgumentParser(description="Recherche de fichiers avec un suffixe donné et export en CSV.",
                                 usage = "find_file.py [-h] -d repertoire -s .suffixe -o fichier.csv")

parser.add_argument("-d", type=Path, required=True, metavar="Dossier", help="Dossier de recherche.")
parser.add_argument("-s", type=str, required=True, metavar="Suffixe", help="Suffixe des fichiers recherchés.")
parser.add_argument("-o", type=Path, required=True, metavar="Fichier", help="Nom du fichier CSV.")

arguments = parser.parse_args()

print(f"Argument après '-d' : {arguments.d}")
print(f"Argument après '-s' : {arguments.s}")
print(f"Argument après '-o' : {arguments.o}")
```



*Remarque : L'avantage du « **parser** » est que l'on peut mettre dans le désordre les couples d'arguments…*

Si on utilise un « ***parser*** », toutes les commandes en ligne suivantes sont équivalentes :

```
--> python find.py -d Temp -s .txt -o resu.csv
--> python find.py -d Temp -o resu.csv -s .txt
--> python find.py -s .txt -o resu.csv -d Temp
--> python find.py -s .txt -d Temp -o resu.csv
--> python find.py -o resu.csv -d Temp -s .txt
--> python find.py -o resu.csv -s .txt -d Temp
```

Avec la ligne de commande : **--> python find.py -d Temp -s .txt -o resu.csv**
On obtient :

```
Argument après '-d' : Temp
Argument après '-s' : .txt
Argument après '-o' : resu.csv
```

Avec la ligne de commande : **--> python find.py -h**
On obtient :

```
usage: find.py [-h] -d repertoire -s .suffixe -o fichier.csv

Recherche de fichiers avec un suffixe donné et export en CSV.

optional arguments:
  -h, --help  show this help message and exit
  -d Dossier  Dossier de recherche.
  -s Suffixe  Suffixe des fichiers recherchés.
  -o Fichier  Nom du fichier CSV.
```

Note importante : « `arguments.d` » est l'équivalent de « **`sys.argv[2]`** » de même que « `arguments.s` » est l'équivalent de « **`sys.argv[4]`** » et « `arguments.o` » est comme « **`sys.argv[6]`** »…

*Et on fait exactement les mêmes choses avec les « `arguments.X` » qu'avec les « **`sys.argv[N]`** » !!!*

Pour avoir une arborescence de travail significative et pas trop grande, il est demandé d'utiliser celle qui se trouve en format compressé sur Moodle : « `Rep_Exercice.zip` ».
Récupérez ce fichier ZIP et décompressez-le dans votre répertoire de travail…

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

1. Mettez en œuvre le « ***parser*** » pour récupérer tous les arguments du script.
2. Effectuez quelques tests sur les valeurs des arguments :
   - Existence du répertoire.
   - Vérification que le répertoire est bien un répertoire.
   - Suffixe différent de "" (string vide).
   - Suffixe commence par un « . ».
3. Générez la liste des fichiers qui vérifient les critères de recherche.
4. Exportez cette liste vers un fichier CSV.

Activité complémentaire

Suite de l'exercice 2 : ip_log_top10.py

On souhaite à présent améliorer le code de l'exercice 2 pour qu'il fournisse les 10 adresses IP les plus fréquentes. Cela doit se faire lorsque l'argument « **-top10** » est passé au script. La nouvelle syntaxe de la commande en ligne est donc :

```
--> python ip_log_top10.py fichier.csv (-ip adresse_ip | -top10)
```

Soit les uns, soit l'autre…

Codez le script « `ip_log_top10.py` » à partir du code de « `ip_log.py` » et qui effectue les tâches suivantes :

1. Placez tout le code de recherche de l'adresse IP la plus fréquente dans la fonction « **`affiche_recherche(…)`** ». Il pourrait être judicieux de passer en arguments à cette fonction le nom du fichier CSV ainsi que l'adresse IP valide qu'il faut rechercher…
2. Créez une nouvelle fonction « **`affiche_top10(…)`** » qui va être complétée par la suite et dont l'objectif est de rechercher le top10 des adresses IP.
3. Revoyez la gestion des arguments fournis au script…

Pour obtenir le top10 des adresses IP les plus fréquentes, la solution la plus simple en termes de codage mais couteuse en termes d'occupation mémoire, consiste à :

- Répertorier dans un dictionnaire toutes les adresses IP présentes dans un fichier « ***log*** ». Les adresses IP sont les « ***clés*** » du dictionnaire. La valeur associée à chaque « ***clé*** » donne le nombre d'occurrences de cette adresse IP dans le « ***log*** »…
- D'extraire ensuite du dictionnaire précédent, les 10 adresses IP les plus fréquentes dans une liste.

4. Ouvrez le fichier « ***log*** » en lecture avec un « ***reader CSV*** » dans une structure « **`try`** » … « **`except`** ».
5. Ajoutez au dictionnaire l'adresse IP trouvée dans chaque ligne CSV.

Indication *: Vous pouvez utiliser pour cela la méthode « **`get(`**`nom_cle, val_defaut`**`)`** » propre aux dictionnaires.*

Si la clé « `nom_cle` » existe déjà dans le dictionnaire, la méthode retourne la valeur associée à cette clé.

En revanche si la clé n'existe pas, la méthode retourne la valeur « `val_defaut` » utilisée pour une nouvelle clé va être ajoutée au dictionnaire…

Exemple :

```
dico[ip] = dico.get(ip, 0) + 1
```

Si la clé « **`ip`** » n'existe pas, elle est ajoutée au dictionnaire avec la valeur « **`1`** ».
Si la clé « **`ip`** » existe, sa valeur est juste incrémentée…

Le dictionnaire ainsi créé toutes les adresses IP du fichier « ***log*** » dans l'ordre d'apparition.

Exemple :

```
{'54.36.149.41': 157, '31.56.96.51': 78, '40.77.167.129': 427, …}
```

Pour trouver les 10 adresses IP les plus fréquentes, la solution la plus simple en termes de codage mais couteuse en termes d'occupation mémoire, est de trier le dictionnaire avec la méthode « **`sorted(…)`** ». Cette dernière retourne une liste de tuples.

Exemple :

```
[ ('66.249.66.194', '353483'),
  ('66.249.66.91', '314522'),
  ('151.239.241.163', '92473'),…
]
```

*Les exemples suivants illustrent la mise en œuvre de la méthode « **sorted(…)** » :*

```
dico = {"c": 3,  "a": 2,  "b": 1}


liste_tri_cle_croissant = sorted(dico.items())
     [('a', 2), ('b', 1), ('c', 3)]
liste_tri_cle_decroissant = sorted(dico.items(), reverse = True)
     [('c', 3), ('b', 1), ('a', 2)]
tri_val_crois = sorted(dico.items(), key=lambda v: v[1])
     [('b', 1), ('a', 2), ('c', 3)]
tri_val_decr = sorted(dico.items(), key=lambda v: v[1], reverse = True)
     [('c', 3), ('a', 2), ('b', 1)]
```

Ordre croissant des « ***clés*** ».

Ordre décroissant des « ***clés*** ».

Ordre croissant des « ***valeurs*** ».

Ordre décroissant des « ***valeurs*** ».

6. Créez une liste de tuples triés par valeur dans le bon ordre à partir du dictionnaire généré précédemment.
7. Générez une sous-liste des 10 adresses IP les plus fréquentes (par « ***slicing*** » par exemple).
8. Affichez le résultat sous la forme :

```
Nombre total d'adresses IP uniques : xxxxxx
Top 10 des IP les plus fréquentes :
xxxxxx fois :    xx.xxx.xx.xxx
xxxxxx fois :    xx.xxx.xx.xx
xxxxx fois :     x.xxx.xxx.xxx
xxxxx fois :
xxxxx fois :
xxxxx fois :
xxxxx fois :     xx.xx.xx.xx
xxxxx fois :     xx.xx.xx.xx
xxxxx fois :     x.xx.xxx.xxx
xxxxx fois :     xxx.xxx.xxx.xxx
```

Présence d'un « **tab** » (« **\t** »)…

Dans l'activité complémentaire qui précède, les solutions retenues (création d'un dictionnaire puis d'une liste pour trier le dictionnaire) sont simples à mettre en œuvre mais couteuses en mémoire.
En utilisant la méthode « **asizeof(***iterable***)** » du module « **pympler** », il est possible d'avoir la taille totale d'un itérable (liste ou dictionnaire).
Évaluez la quantité de mémoire occupée par le dictionnaire et la liste généré ci-dessus.
Proposez une solution qui permet de trouver les 10 adresses IP les plus fréquentes sans générer la liste résultant du tri du dictionnaire.
Essayez de trouver une solution algorithmique qui évite de créer un dictionnaire… Elle sera forcément très couteuse en temps de calcul !