Informatique 2 : Travaux Pratiques

Programmes avancés

Le but de cette séance est de se familiariser avec la manipulation des entrées et sorties d'un programme ainsi que la gestion des paramètres et exceptions en Python.
Procéder par étape et tester votre code pour chaque étape. Un dossier contenant des fichiers de test vous est fourni sur Moodle pour que vous puissiez tester votre code.
Temps mentionné (🕓) à titre strictement indicatif.

De manière générale, la bonne pratique veut qu'on découple les I/O (entrées/sorties) utilisateurs et les fonctions "métier" (fonctions qui effectuent les opérations directement sur les données).

Exemple

```python
# exemple de mauvaise pratique
def square():
    x = int(input())
    print(x * x)


# exemple de bonne pratique
def square(x):
    return x * x


def read_int():
    s = input('Entrez un nombre: ')
    x = int(s)
    return x


if __name__ == '__main__':
    try:
        x = read_int()
    except ...:
        ...

    x_2 = square(x)
    print(x_2)
```

Exercices de base

Question 1: (🕓 *15 minutes*) Dans un module *module.py*, définir un bloc `main` qui ne sera exécuté que si le module est exécuté directement (i.e. en utilisant *run* dans PyCharm, ou *python3 module.py* dans le Terminal).

- Définir une fonction **echo** dans ce module, qui affiche une chaîne de caractères.
- Insérer deux instructions `print`, l'une en dehors du bloc `main` et l'autre à l'intérieur, qui affichent chacune une chaîne de caractères différente.
- Créer un autre module *test.py*, importer *module.py* et appeler dans son bloc `main` la fonction **echo** définie dans le module *module.py* (lui passer une chaîne de caractères en paramètre).
- Comparer l'exécution des deux modules.

Attention

Dans la suite des exercices, utiliser systématiquement un bloc `main`.

Question 2: (🕓 *10 minutes*) Ecrire un programme qui prend un ou plusieurs entier(s) en argument de la ligne de commande (utiliser **sys.argv**) et qui affiche le produit de ces entiers. S'il y en a un seul, le programme affiche le nombre en question. S'il y en a aucun, il affiche `None`.

Rappel

Pour rappel, les arguments de la ligne de commande fonctionnent de la manière suivante : lorsqu'on exécute un programme, on peut ajouter un certain nombre d'arguments (des chaînes de caractères) après le nom du fichier, séparés par des espaces. Ces chaînes de caractères peuvent être récupérées dans le programme via la variable **sys.argv** qui prend la forme d'une liste de chaînes de caractères. Attention, le premier élément de cette liste correspond au nom du fichier Python exécuté.

Exemple

python3 e2.py 3 4 2 2 affiche **48**, *python3 e2.py 3* affiche **3** et *python3 e2.py* affiche **None**.

Question 3: (*10 minutes*) Ecrire un programme qui affiche le quotient de deux nombres passés en argument de la ligne de commande (le premier divisé par le deuxième). Utiliser **sys.argv**. Gérer les problèmes dûs à une division par zéro **ZeroDivisionError** avec une exception.

Question 4: (*10 minutes*) Ecrire un programme qui affiche la racine carrée d'un nombre passé en argument de la ligne de commande (utiliser **sys.argv**). Avec des exceptions, gérer le cas où l'utilisateur passe autre chose qu'un nombre et le cas où il passe un nombre négatif. Afficher un message si aucune exception n'est levée. Dans tous les cas, afficher **'Au revoir!'** à la fin de l'exécution.

Question 5: (*10 minutes*) Ecrire un programme qui demande à l'utilisateur d'entrer un nombre (en utilisant la méthode **input**()) et qui affiche le carré de ce nombre. Gérer le cas où l'utilisateur entre une chaîne de caractères qui ne représente pas un entier : Afficher un message d'erreur et demander à nouveau à l'utilisateur d'entrer un nombre tant que la chaîne de caractères entrées par l'utilisateur n'est pas correcte.

Question 6: (*15 minutes*) Ecrire un programme qui récupère un entier n passé en argument depuis la ligne de commande. Le programme demande alors n fois à l'utilisateur d'entrer une chaîne de caractères puis affiche la concaténation de ces chaînes, séparées par des espaces.
Vérifier que n est bien un entier et gérer les exceptions.

Exemple

Si l'utilisateur exécute le programme en passant le nombre 3 et entre les chaînes de caractères **'un'**, **'deux'** et **'trois'** : le programme affiche **'un deux trois'**.

Question 7: (*10 minutes*) Ecrire un programme qui récupère un nombre passé en argument de la ligne de commande avec le module **argparse** et qui affiche le logarithme (népérien) de ce nombre. Spécifier au module **argparse** que l'argument de la liste de commande est un nombre (entier ou flottant, tester avec une chaîne de caractère et observer le résultat). Afficher l'aide générée par le module **argparse** en exécutant la commande *python3 logarithme.py -h*. Gérer le cas où le nombre passé en argument est plus petit ou égal à zéro) en affichant un message sur la sortie d'erreur.

Vous pouvez trouver la documentation concernant **argparse** sur le lien suivant : `https://docs.python.org/fr/3/library/argparse.html`.

Exemple

python3 logarithme.py -n 3 affiche **1.099**

Attention

— Le nombre à récupérer doit être un argument obligatoire et être spécifié avec un "flag" (*-n*)
— N'utiliser ni **input**() ni **sys.argv** !

Question 8: (*15 minutes*) Ecrire un programme qui récupère deux arguments de la ligne de commande avec le module **argparse** et qui affiche leurs valeurs. Le détail des arguments de la ligne de commande (affiché en saisissant *python3 mon_fichier.py -h* dans le Terminal) devrait être le suivant :

```
usage: mon_fichier.py [-h] -i ENTIER [-s CHAINE]

options:
  -h, --help            show this help message and exit
  -i ENTIER, --entier ENTIER
                        Un entier positif
  -s CHAINE, --chaine CHAINE
                        Un chaine de caracteres
```

Exemple

python3 print.py -s abc -i 5 affiche **s='abc', i=5**

Tester en inversant les arguments **s** et **i** (*python3 print.py -i 5 -s abc*). Quel est l'avantage d'utiliser **argparse** plutôt que **sys.argv** ?

Attention

— N'utiliser ni **input()** ni **sys.argv** !

Question 9: (*10 minutes*) projet Ecrire un programme qui récupère le nom d'un fichier passé en arguments de la ligne de commande. Chaque ligne de ce fichier représente une paire (clé,valeur) d'un dictionnaire. Lire le fichier, créer et afficher le dictionnaire correspondant. Utiliser le fichier *config.txt* (disponible sur moodle) pour tester.

Exemple

Le fichier suivant :

SEPARATOR,;
N_MAX,16

Genèrera le dictionnaire suivant : {**'SEPARATOR' : ';', 'N_MAX' : '16'**}

Attention

Le fichier de configuration devra comporter une paire **(clé, valeur)** par ligne. En particulier, il ne devra pas comporter de ligne vide.

Ajouter la possibilité de passer une clé du dictionnaire en argument de la ligne de commande (2ème argument (facultatif)), s'il y a un deuxième argument, afficher la valeur correspondant à la clé (utiliser **sys.argv**).

Exemple

python3 programme.py input_configuration.txt N_MAX affichera 16.

Question 10: (*20 minutes*) projet Lister et afficher le nom des fichiers du répertoire courant de type : *'JPG', 'JPEG', 'jpg', 'jpeg'*.

Question 11: (*20 minutes*) projet Implémenter, à l'aide du module **argparse**, la récupération des informations à partir de la ligne de commande. À titre d'indication, le message d'aide du module **sherlock** devra être le suivant :

Exemple

```
$ python3 sherlock.py -h

usage: sherlock.py [-h] [-v] -s SUSPECT -t TWITTER_API_KEY -u
                   TWITTER_API_KEY_SECRET -g GOOGLE_API_KEY -lat LATITUDE -lng
                   LONGITUDE -d DATE

Identifie les suspect.e.s les plus plausibles à partir de leurs traces de
mobilité (issues de sources multiples incluant les tweets géo-taggés, les
traces Wi-Fi et les flux de photos géo-taggées) pour un crime spécifié par une
date/heure et une localisation

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         affiche les détails de l'exécution du programme et les
                        avertissements
  -s SUSPECT, --suspect SUSPECT
                        fichier contenant la liste des suspect.e.s et les
                        sources de données de localisation
  -t TWITTER_API_KEY, --twitter-api-key TWITTER_API_KEY
                        clé pour l'accès à l'API Twitter (clé privée de
                        l'application Twitter)
  -u TWITTER_API_KEY_SECRET, --twitter-api-key-secret TWITTER_API_KEY_SECRET
                        clé secrète pour l'accès à l'API Twitter
  -g GOOGLE_API_KEY, --google-api-key GOOGLE_API_KEY
                        clé pour l'accès à l'API Google (clé privée du compte
                        développeur Google)
  -lat LATITUDE, --latitude LATITUDE
                        latitude de la scène du crime
  -lng LONGITUDE, --longitude LONGITUDE
                        longitude de la scène du crime
  -d DATE, --date DATE  date et heure du crime (au format JJ/MM/AAAA-hh:mm,
                        par exemple 19/04/2019-15:02:45)
```

Rappel

Pour déclarer un nouvel argument, utiliser la méthode **add_argument** du parser, par exemple :
parser.add_argument('-s', '--suspect', help="fichier contenant la liste des suspect.e.s et les sources de données de localisation", required=True)
ajoute un argument obligatoire avec le flag **-s** (accessible depuis l'attribut **suspect** du parser) qui représente le nom d'un fichier.