TP n°1 - Structures de données linéaires en Python

Notions abordées

- Exécution de script Python, utilisation de l'interpréteur, notamment import
- Révision de syntaxe élémentaire en Python : `def`, `if`/`then`/`else`, `print`
- Précision des types dans la signature, commentaires, documentation
- Parcours de structures linéaires avec `for`, par indices et par valeurs
- Typage dynamique
- Passage par valeur/par référence

Toutes les fonctions doivent être commentées et testées. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments placée entres triple guillemets sous la signature de la fonction, (`""" documentation sur plusieurs lignes"""`). Les autres commentaires s'écrivent sur des lignes commençant par dièse : `# ligne de commentaire`.

Pour commencer

Exercice 0 Prise en main

Question 1

Télécharger sur `cahier de prépa` le fichier nommé `exo0.py`. Placer ce fichier dans un répertoire dont le chemin d'accès finit par `MP2I/itc/TPs/TP1`, où `itc` signifie "informatique tronc commun". Ceci permettra de ne pas écraser le dossier nommé `TP1` du premier semestre. Dans ce dossier, ouvrir un terminal pour exécuter le fichier téléchargé grâce à la commande suivante : `python3 exo0.py`.

Question 2

Ouvrir un deuxième onglet dans le terminal (avec l'icône + ou bien dans Fichier/Nouvel onglet). Dans ce nouvel onglet lancer l'interpréteur Python grâce à la commande `python3`. Taper alors `2+3` puis Entrée, et observer la réponse de l'interpréteur. Comparer ensuite les réponses obtenues en tapant `print ("ok")` ou seulement `"ok"`, ou `a="ok"` puis `a`.

Question 3

Copier/coller une à une les définitions dans l'interpréteur grâce aux raccourcis clavier Ctrl+C dans l'éditeur de texte, et Ctrl+Maj+V dans l'interpréteur, puis taper Entrée deux fois pour lancer l'interprétation. On note que l'interpréteur ne fait pas de réponse (pas de nouvelle, bonne nouvelle)., néanmoins on voudrait s'assurer que la définition copiée a été prise en compte, Dans le cas d'une définition de fonction, effectuer dans l'interpréteur un appel de fonction et observer la réponse. Dans la cas d'une définition de valeur, taper le nom de cette valeur et observer la réponse.

Exercice 1 Différence entre afficher et retourner

Question 1 💻
Définir dans un fichier nommé `exo1.py` une fonction `somme` qui prend en argument deux entiers `a` et `b`, et qui retourne leur somme. Proposer (dans le fichier `exo1.py`, à la suite de la définition) un jeu de test pour cette fonction (grâce à la fonction `assert`).

Question 2 💻
Dans le fichier `exo1.py`, définir une fonction `affiche_somme` qui prend en argument deux entiers `a` et `b`, et qui affiche leur somme. Quel est le type de retour de cette fonction ? Peut-on tester cette fonction avec `assert` ?

Question 3 💻
Essayer d'appeler `affiche_somme` dans l'interpréteur ? Cela fonctionne-t-il ? Pourquoi ?
Sortir de l'interpréteur, et le relancer. Taper alors `from exo1 import affiche_somme`, puis réessayer d'appeler la fonction `affiche_somme`. Faire plusieurs appels, pertinents et représentatifs, afin de tester la fonction.

Exercice 2 Définir des fonctions, les tester et les appeler

Question 1 💻
Définir dans un fichier nommé `exo2.py` une fonction `convertit_minutes` qui prend en argument un nombre de minutes, et qui retourne le nombre d'heures pleines et le nombre de minutes restantes auxquels il correspond. *Afin de pouvoir préciser en annotation de type que le type de retour de cette fonction est un couple, on importe en début de fichier le type à paramètres* `Tuple` *du module* `typing`.

```
from typing import Tuple
```

Il suffit ensuite de préciser entre crochets les types des argument du n-uplet souhaité, par exemple `Tuple[int, bool, bool]` *pour un triplet composé d'un entier et de deux booléens.*

Question 2 💻
Dans le fichier `exo2.py`, définir une fonction `convertit_heures` qui prend en argument un nombre d'heures, et qui retourne le nombre de jours pleins et le nombre d'heures restantes auxquels il correspond. À nouveau on rassemble ces deux nombres au sein d'un couple.

Question 3 💻
Dans le fichier `exo2.py`, définir une fonction `convertit_minutes_plus` qui prend en argument un nombre de minutes, et qui retourne cette fois le nombre de jours pleins, le nombre d'heures pleines restantes et le nombre de minutes restantes auxquels il correspond. On rassemble ces deux nombres au sein d'un triplet. Cette fonction **doit** faire appel aux fonctions précédentes, et ne faire apparaître ni `//` ni `%`.

Parcourir des listes ou des chaînes

Exercice 3 Échauffement

Question 1 💻
Définir une fonction `somme_liste` qui prend en argument une liste d'entiers et qui retourne leur somme.

Question 2 💻
Définir une fonction `somme_liste_ind_paires` qui prend en argument une liste d'entiers et qui retourne la somme de ceux placés à des indices pairs dans la liste.

Question 3 💻
Définir une fonction `somme_liste_val_paires` qui prend en argument une liste d'entiers et qui retourne la somme de ceux qui sont pairs.

Question 4 💻
Définir une fonction `moyenne_simple` qui prend en argument une liste d'entiers et qui retourne leur moyenne. *On veillera à préciser les hypothèses adéquates*

Question 5 💻
Définir une fonction `moyenne_pond` qui prend en argument une liste d'entiers et une liste de coefficients entiers de même taille et qui retourne la moyenne pondérée des entiers. *On veillera à préciser les hypothèses adéquates.*

Question 6 💻
Définir une fonction `isobarycentre` qui prend en argument une liste de points du plan et qui renvoie leur isobarycentre, en supposant que les points sont représentés par des couples de flottants. *On veillera à préciser les hypothèses adéquates.*

Question 7 💻
Définir une fonction `barycentre` qui prend en argument une liste de points du plan et une liste de coefficients de même taille et qui retourne le barycentre des points selon les coefficients. *On veillera à préciser les hypothèses adéquates.*

Exercice 4 Listes d'éléments ordonnés

- *Minimum global, minimum partiel*

Question 1 💻
Définir une fonction `min_liste` qui prend en argument une liste d'entiers non vide et qui retourne le plus petit de ces entiers.

Question 2 💻
Définir une fonction `min_liste_part` qui prend en argument une liste d'entiers non vide `L` et un indice `k`, et le plus petit des entiers placés dans la liste après l'indice `k`. Par exemple, `min_liste_part ([1,2,3,6,5,8],2)` vaut `3`, car c'est le minimum de l'ensemble $\{3, 6, 5, 8\}$ qui est l'ensemble des entiers placés dans la liste à un indice $i \geqslant 2$.

- *Tri par sélection*

Question 3 💻
Définir une fonction `tri_selec` qui prend en argument une liste d'entiers et qui trie ses éléments par ordre croissant, en plaçant à chaque étape, par échange, le minimum de la partie non triée de la liste en tête de celle-ci. Par exemple, pour trier `[6,4,8,3]`, on sélectionne le minimum 3 et on le place en tête de liste en l'échangeant avec 6, on obtient donc `[3,4,8,6]`. À ce stade les 3 derniers éléments de la liste sont non triés, on sélectionne alors leur minimum, c'est-à-dire 4, et on le place en tête de liste, par échange avec lui même (ce qui revient à ne rien faire, mais bon). On obtient encore `[3,4,8,6]` mais à ce stade on sait que les deux premiers éléments sont triés (et même bien placés). Les deux derniers ne sont pas triés, on sélectionne leur minimum, c'est-à-dire 6, et on le place en tête de cette partie non triée, en l'échangeant avec 8. On obtient alors `[3,4,6,8]`, qui est triée.

Question 4 ✍
Peut-on faire un jeu de tests pour cette fonction en suivant la syntaxe `assert (f(arg) == res )` ? Pourquoi ? Comment testez-vous votre fonction ?

Question 5 💻 puis ✍
Dans l'interpréteur définir une liste `L1`, évaluer `L1` pour vérifier que cette définition a été prise en compte, puis appeler `tri_selec` sur `L1`. Quelle réponse obtient-on ? Évaluer à nouveau `L1`. Que constate-t-on ? Expliquer ce qu'il se passe selon vous.

Question 6 💻 puis ✍
Dans l'interpréteur définir un entier `a1`, évaluer `a1` pour vérifier que cette définition a été prise en compte. Recopier la définition suivante dans l'interpréteur

```
def ajoute_1 (a:int) -> None :
        a=a+1
        print ("-->",a)
```

Appeler `ajoute_1` sur `a1`. Quelle réponse obtient-on ? Évaluer à nouveau `a1`. Que constate-t-on ? Expliquer ce qu'il se passe selon vous. M'appeler pour me soumettre vos explications.

Question 7 💻
Définir (dans le fichier `exo4.py` une fonction `est_croissante` qui prend en entrée une liste d'entiers, potentiellement vide, et qui teste si elle est triée par ordre croissant. *Vous devez savoir justifier le comportement de votre fonction sur la liste vide.* Penser à tester cette fonction.

Question 8 💻
Utiliser la fonction `est_croissante` pour tester, à l'aide d'`assert` la fonction `tri_selec`. Quelle critique peut-on émettre à propos de ce jeu de tests.

- *Second minimum*

On remarque que dans une liste L dont les éléments sont deux à deux distincts, il y a une seule occurrence du minimum. Donc si L est de taille supérieure ou égale à 2, en supprimant cette unique occurrence du minimum on obtient une liste ayant au moins un élément, qui a donc elle-même un minimum. Ce minimum est appelé **second minimum** de L.

Question 9 💻
Définir une fonction `sans_doublon` qui prend en argument une liste d'entiers et qui teste si cette liste ne contient aucun entier deux fois.

Question 10 💻
Définir une fonction `second_min_liste` qui prend en argument une liste d'entiers non vide et qui retourne le second minimum de cette liste.

- *Passer à des listes de réels ou de caractères*

Question 11 💻 puis ✍
Dans l'interpréteur définir une liste de flottants `L2`, évaluer `L2` pour vérifier que cette définition a été prise en compte. Appeler `tri_selec` sur `L2`. puis évaluer à nouveau `L2`. Que constate-t-on ? Cela vous paraît-il normal au vu de la signature de la fonction `tri_selec` ?

En fait les types que l'on a écrits jusqu'ici n'étaient que des **annotations de type**. Elles font partie intégrante du code source, au sens où l'on pourrait appliquer à ce code source un programme qui vérifie la cohérence des types indiqués, mais elles sont ignorées par Python.
En Ocaml, on a vu qu'on pouvait aussi omettre de préciser les types lors de la définition de fonction, mais la situation est bien différente. En effet Ocaml est muni d'un moteur d'inférence de types qui lui permet de déduire le type de la fonction (de ses arguments et de sa sortie), de sorte qu'il peut fixer par lui-même le type d'une fonction dès sa définition. (c'est ce qu'on appelle le typage statique). Python ne se donne pas cette pas cette peine. Il attend de voir sur quels arguments on va appeler cette fonction, et ne fixe pas à la définition le type autorisé pour chaque argument. C'est ce qui permet d'appliquer la fonction `tri_selec` à une liste de flottants, alors qu'on l'avait créée pour trier des listes d'entiers. On parle de **typage dynamique**.

Question 12 💻 ∗
Afin de donner une signature représentative d'une fonction qui s'applique en réalité à plusieurs types, on peut utiliser des variables de types grâce au module `typing`.
Après l'import `from typing import TypeVar` on déclare une variable de type comme suit.
`T = TypeVar('T')`.
On peut alors utiliser `T` pour désigner un type quelconque. Dans notre cas, `tri_selec` s'applique donc à une liste de type `List[T]` pourvu que les éléments de type `T` soient comparables entre eux par l'opérateur `<=`, ce qu'on précise en commentaire.

Question 13 💻 puis ✍
Dans l'interpréteur taper `"a"<"b"`, `"b"<"a"`, `"a"<"c"`, `"a"<"B"` et `"a"<"A"`. Observer les réponses, au besoin faire d'autres tests puis synthétiser le comportement de l'opérateur `<` sur les chaînes réduites à un caractère.

Question 14 💻
Comment trier la liste de caractères `["a","b","d","e","c","c"]` par ordre alphabétique ?

Question 15 💻
Comment trier la liste de caractères `["a","B","d","e","C","c"]` par ordre alphabétique ?

Exercice 5 Listes de mots

Question 1 💻
Définir une fonction `sans_espace` qui prend en argument une chaîne de caractères et qui teste si cette chaîne ne contient aucun espace.

Question 2 💻
Définir une fonction `est_mot` qui prend en argument une chaîne de caractères et qui teste si cette chaîne est un mot, c'est-à-dire une suite de lettres uniquement. *Pour simplifier, on peut omettre le cas de mots composés et des lettres accentuées.*

Question 3 💻
Quel lien existe-t-il systématiquement entre les valeurs retournées par `est_mot` et `sans_espace` appelées sur une même chaîne ?

Question 4 💻
Définir une fonction `sont_des_mots` qui prend en argument une liste de chaînes de caractères et qui teste si toutes ces chaînes sont des mots.

On rappelle qu'un mot u est **préfixe** d'un mot v s'il existe un mot w tel que $u.w = v$, où . désigne la concaténation.

Question 5 ✍
`"a"` est-elle préfixe de `"abracadabra"` ?
`"bra"` est-elle préfixe de `"abracadabra"` ?
`""` est-elle préfixe de `"abracadabra"` ?
`"abracadabra"` est-elle préfixe de `"abracadabra"` ?

Question 6 💻
Définir une fonction `est_prefixe` qui prend en argument deux chaînes de caractères et qui teste si la première est préfixe de la deuxième.

Question 7 💻
Définir une fonction `sont_prefixes_successifs` qui prend en argument une liste de chaînes de caractères, et qui teste si chacune, hormis la dernière, est préfixe de la suivante.

Question 8 💻 puis ✍
Définir une fonction `existe` qui prend en argument une chaîne de caractères `mot` et une liste de chaînes de caractères `L` et qui teste `mot` est un élément de `L`. Quelle est la complexité de cette fonction (en fonction de la taille de `L`) ?

Question 9 💻 puis ✍
Dans l'hypothèse où la liste `L` est triée, peut-on tester la présence d'un élément de manière plus efficace. Si oui, définir une nouvelle fonction `existe_bis` et préciser sa complexité (en fonction de la taille de `L`).

On cherche dans la suite de l'exercice à modéliser le jeu suivant. Chaque joueur a son tour va proposer un caractère, le but étant que la suite des caractères donnés depuis le début du jeu forme un préfixe d'un mot (un mot au sens usuel, c'est-à-dire un mot qui existe dans la langue considérée). La première personne qui ne peut pas donner un caractère à ajouter au *mot* courant pour en faire un préfixe valable a perdu. De plus, si l'on pense que le dernier caractère ajouté ne forme pas un préfixe valable, on peut accuser la personne qui l'a proposé de bluffer. Dans ce cas, la personne accusée peut

se défendre en donnant un mot existant qui prolonge le mot courant, alors le jeu s'arrête et c'est la personne qui a accusé à tort qui a perdu. Si la personne accusée ne peut justifier le dernier caractère qu'elle a ajouté, elle a perdu.

Question 10 💻
Donner une fonction `concatene` qui prend en entrée une liste de caractères `Lc`, et qui retourne la chaîne de caractères obtenue en concaténant tous les caractères de `Lc`.

Question 11 💻
Donner une fonction `justif_valide` qui prend en entrée une liste de caractères `Lc`, un mot `temoin` et une liste de mots `Lvoc`, et qui teste si le mot `temoin` est à la fois un mot qui existe dans `Lvoc` et un mot qui prolonge la concaténation des caractères de `Lc`.

Question 12 💻
Donner une fonction `ajout_valide` qui prend en entrée une liste de caractères `Lc`, un caractère `c` et une liste de mots `Lvoc`, et qui teste si le mot obtenu en concaténant les caractères de `Lc` puis `c` peut-être prolongé en un mot de `Lvoc`.