Première séance

Dans cette séance, nous allons présenter les outils que nous utiliserons cette année, et referont une

I. Prise en main

Pour les TPs de Python, vous aurez besoin en général de deux choses, en plus du sujet : le ou les fichiers sur lesquels vous travaillez, et le ou les programmes que vous utilisez.

1. Les fichiers

Vous allez utiliser une machine utilisant un système d'exploitation de type Unix (ici, Linux) conçu pour avoir plusieurs utilisateurs en toute sécurité. Chaque utilisateur a un "chez-lui" où il peut contrôler ce qui s'y passe, on parle de *répertoire principal* ou *home directory*. Cela se traduit principalement par la possibilité de décider des *droits* des fichiers d'un utilisateur, avec en particulier le droit à la lecture (on peut vouloir garder des choses secrètes) et le droit à l'écriture et à la modification (pour empêcher que l'on modifie ses fichiers).

2. Le programme

Dans l'immense majorité des cas, nous allons utiliser un interpréteur Python nommé IDLE (pour *Integrated DeveLopment Environment*). Un *interpréteur* est un programme qui lit et exécute au fur et à mesure du code Python[1].

Il existe deux versions de Python, l'une ancienne, avec des versions numérotées 2… et une autre, moderne et à jour, numérotée 3…

Pour être sûr d'utiliser l'IDLE avec la bonne version de Python (celle à jour), vous devrez **commencer** par démarrer le bon programme (avec un nom qui ressemble à quelque chose comme `IDLE 3.2`) que vous trouverez dans le menu *Démarrer*. Ensuite, depuis l'IDLE, nous ouvrirez le fichier avec lequel vous voulez travailler en utilisant la commande `Open` du menu `File`.

3. Pour aujourd'hui

Vous allez, pour débuter ce TP :

1. chercher le répertoire des fichiers à utiliser chez vous (l'endroit où les trouver sera indiqué au tableau),

[1] Par opposition à un *compilateur* qui transforme en une seule fois un code Python en programme exécutable.

2. le copier chez vous (le répertoire en entier, pas simplement les fichiers),
3. ouvrir *le bon* IDLE,
4. depuis l'IDLE, ouvrir le fichier dans lequel travailler.

Le fichier, après une entête à ne pas modifier (et ne pas essayer de comprendre), vous aurez une suite d'exercice dont voici les deux premiers :

```
### Exercice 1

def factorielle(t):
    ...

test_tp1.exercice1(factorielle)

### Exercice 2

def occurences(t, e):
    ...

# test_tp1.exercice2(occurences)
```

À chaque fois, après le numéro de l'exercice, un début de fonction à remplir et une ligne de test. La ligne de test du premier exercice n'est pas commentée, alors que les suivantes le sont.

Pour chaque exercice, vous veillerez à ce que tous les tests sont réussis avant de passer à l'exercice suivant. Ensuite, vous commenterez la ligne de test de l'exercice fait, et décommenterez celle de l'exercice suivant.

II. Rappels

1. Syntaxe de base

On rappelle les éléments principaux de la syntaxe de Python avec l'exemple suivant :

```
def mystere(n):
    s = 0
    for i in range(n):
        if i % 3 == 0:
            s = s + i
    return s
```

En particulier, tout ce qui est suivi par un ensemble (ou *bloc*) d'instructions se termine par " : " et l'ensemble du bloc est *indenté* vers la droite. Assurez-vous que vous comprenez parfaitement ce code, et n'hésitez pas à demander des explications.

Exercice 1 Écrire une fonction `factorielle` qui, étant donné un entier $n \geq 1$, renvoie le produit

$$n! = 1 \times 2 \times \cdots \times n.$$

Par exemple, on veut :

```
>>> factorielle(4)
24
```

2. Petit point technique : `return` ou `print` ?

L'instruction `print` effectue un affichage à l'écran, à destination de l'utilisateur. La commande **`return`**, quand à elle, s'utilise dans une fonction et renvoie une valeur qui est disponible pour une utilisation ultérieure (et interrompt au besoin l'exécution de la fonction).

```
>>> def f1(x):
...     return x + 1
...
>>> def f2(x):
...     print(x + 1)
...
>>> a = f1(4)
>>> a
5
>>> b = f2(4)
5
>>> b
>>> 2 * a
10
>>> 2 * b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: "int" and "NoneType"
```

Ici, même si `f2` ne contient pas de **`return`**, Python considère néanmoins qu'elle renvoie une valeur, nommée **`None`** et qui traduit... l'absence de valeur, d'où l'erreur finale où l'on tente de faire un produit entre l'entier `2` et **`None`**.

En conclusion, dans l'immense majorité des cas, l'exécution d'une fonction se termine par un **`return`**, et on n'utilise `print` que lorsque l'on doit explicitement afficher quelque chose à l'écran.

III. Tableaux

1. Quelques rappels supplémentaires

Vous les avez peut-être croisés sous le nom de liste, mais nous utiliser la bonne terminologie[2].

Si t est un tableau, alors sa longueur ℓ s'obtient à l'aide de la commande `len(`t`)`. Dans ce cas, les éléments du tableaux sont accessibles et modifiables sous la forme $t[i]$ pour des indices i allant de 0 inclus à $\ell - 1$ inclus (et ℓ exclu).

```
>>> t = [213, 45, 78, 319]
>>> len(t)
4
>>> t[2]
78
>>> t[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> t[3] = 12
>>> t
[213, 45, 78, 12]
>>> for i in range(len(t)):
...     print(t[i])
...
213
45
78
12
```

On peut ajouter un argument supplémentaire à `range` pour indiquer la valeur de départ.

```
>>> for i in range(5, 10):
...     print(i)
...
5
6
7
8
9
```

[2] En informatique, les tableaux et listes sont des structures de données distinctes dans l'immense majorité des langages de programmation... sauf en Python, ce qui crée des confusions, puisque les deux structures s'utilisent différemment.

Pour information, on peut ajouter un troisième argument indiquant le pas de l'incrément.

2. Parcours complet

Une première utilisation typique d'un tableau est de le parcourir en entier, et de visiter chacun de ses éléments dans l'ordre et leur appliquement un traitement. Cela s'effectue avec une boucle **for**.

La structure générale d'un tel parcours est le suivant :

```
def canevas_de_parcours_complet(tab):
    ... # initialisation
    for i in range(len(tab)):
        ... # on traite tab[i]
    ... # traitement final
```

Par exemple, voici une fonction qui calcule la somme des éléments d'un tableau :

```
def somme(tab):
    s = 0
    for i in range(len(tab)):
        s = s + tab[i]
    return s
```

Exercice 2 Écrire une fonction `occurences` qui, étant donné un tableau t et un élément e, renvoie le nombre d'occurences de e dans t.

```
>>> occurences([5, 2, 4, 8, 7, 4, 1, 2, 5, 4], 4)
3
>>> occurences([5, 2, 4, 8, 7, 4, 1, 2, 5, 4], 6)
0
```

Exercice 3 Écrire une fonction `maximum` qui renvoie la plus grande valeur contenue dans le tableau supposé non vide passé en argument. **On n'utilisera pas la fonction `max`. À la place, on fera des comparaisons "à la main".**

Exercice 4 Écrire une fonction `position_maximum` qui renvoie la position du plus grand élément du tableau supposé non vide passé en argument. À nouveau, on n'utilisera pas la fonction `max`. Si le maximum apparaît plusieurs fois, on retournera le plus petit indice.

Formellement, si le tableau t est de longueur ℓ et que `position_maximum`(t) ren-

voie p, alors

$$\forall i \in [\![0, \ell-1]\!], \quad \begin{cases} t[i] < t[p] & \text{si } i < p \\ t[i] \leq t[p] & \text{si } i \geq p \end{cases}$$

```
>>> position_maximum([5, 2, 4, 8, 7, 4, 1, 2, 5, 4])
3
>>> position_maximum([5, 7, 2, 6, 4, 7, 3, 1, 6])
1
```

Exercice 5 (un peu plus difficile, vous pouvez y revenir plus tard) Écrire une fonction `position_2_maximums` qui, étant donné un tableau d'entiers de taille au moins 2, renvoie un couple d'entiers distincts (p, q) tels que p soit l'entier retourné par `position_maximum` et q est tel que $t[q] \leq t[p]$ et

$$\forall i \in [\![0, \ell-1]\!], \quad i \neq p \implies \begin{cases} t[i] < t[q] & \text{si } i < q \\ t[i] \leq t[q] & \text{si } i \geq q \end{cases}$$

Par exemple, on veut :

```
>>> position_2_maximums([3, 5, 1, 2, 5, 4, 5])
(1, 4)
>>> position_2_maximums([3, 2, 5, 2, 6, 4, 5])
(4, 2)
```

Vous essaierez, dans la mesure du possible, de programmer cette fonction avec une **unique** boucle **`for`**.

3. Parcours partiel

Il n'est parfois pas nécessaire de parcourir un tableau en entier pour avoir le résultat désiré. Dans ce cas, on peut avoir une condition d'arrêt anticipé qui indique si l'on peut interrompre le parcours. Pour en tenir compte, on a besoin d'une boucle **`while`** et non une boucle **`for`**.

Pour cela, quelques précautions sont nécessaires. Tout d'abord, il faut gêrer un indice de parcours "à la main" (ce qui apparaît dans la programme suivant avec les lignes `i = 0` et `i = i + 1`). Il faut de plus s'assurer que l'on ne sort pas du tableau, ce qui apparaît avec la partie du test `i < len(tab)`. Notons aussi qu'une fois ce test passé, on est sûr que `tab[i]` est bien défini.

```
def canevas_de_parcours_partiel(tab):
    ... # initialisation
    i = 0
    while i < len(tab) and ...: # condition d'arrêt anticipé
```

```
        ... # on fait quelque chose avec tab[i]
        i = i + 1
    ... # traitement final
```

On portera une attention particulière à la condition qui a déclenché la sortie de la boucle **while**. En effet, la condition d'arrêt anticipé (notons la $P(\texttt{i})$) peut dépendre de la valeur `tab[i]`, or cette dernière n'est pas nécessairement définie. On a donc *a priori* deux cas :

- ⋆ soit `i` = `len(tab)`,
- ⋆ soit `i` < `len(tab)` et $P(\texttt{i})$ est fausse.

Une autre façon de sortir de façon anticipée... de la fonction cette fois, est d'utiliser un **return** à l'intérieur de la boucle.

Exercice 6 Écrire une fonction `est_présent` qui, étant donné un tableau t et une valeur e, renvoie **True** si e apparaît dans t, et **False** sinon.

```
>>> est_présent([4, 1, 2, 5], 3)
False
>>> est_présent([4, 1, 2, 5], 1)
True
```

Exercice 7 Écrire une fonction `est_croissante` qui indique si le tableau passé en argument est croissant (au sens large).

```
>>> est_croissant([2, 4, 4, 5, 7])
True
>>> est_croissant([5, 7, 3, 4, 8])
False
```

Exercice 8 Écrire une fonction `est_modal` qui indique si le tableau passé en argument est *modal*, c'est-à-dire s'il peut être décomposé en un sous-tableau croissant suivi d'un sous-tableau décroissant (les sous-tableaux peuvent être de longueur nulle, mais leur concaténation doit correspondre au tableau entier). On a, bien sûr, le droit de faire plusieurs boucles **while** à la suite.

```
>>> est_modal([2, 3, 3, 4, 5, 5, 3, 3, 1])
True
>>> est_modal([1, 2, 3, 2, 3, 4])
False
```

Exercice 9 Écrire une fonction `somme_croissante` qui prends en entrée un tableau non vide de nombres et renvoie la somme des premiers éléments en s'arrêtant dès qu'un élément est strictement plus petit que le précédent.

```
>>> somme_croissante([2, 4, 4, 5, 7])
22
>>> somme_croissante([5, 7, 3, 4, 8])
12
>>> somme_croissante([4, 3, 2, 1])
4
```