L'objectif du TP est de présenter les éléments de syntaxe à connaître obligatoirement sur les chaînes de caractères et les listes, puis de le mettre en application sur quelques exemples.

Table des matières

1 Chaînes de caractères **1**
1.1 Syntaxe exigible pour les chaînes de caractères 1
1.2 Écriture de quelques fonctions avec des chaînes de caractères 3

2 Listes **3**
2.1 Syntaxe exigible pour les listes 3
2.2 Écriture de quelques fonctions avec des listes 6

3 Exercices **6**
3.1 Quelques fonctions complémentaires avec des listes 6
3.2 Recherche du second maximum d'une liste 6
3.3 Découpage d'une chaîne de caractères 7
3.4 Crible d'Ératosthène 7
3.5 Suite de Collatz 7
3.6 Suite de Conway 8

4 Complément : Conséquences pratique du caractère muable des listes en python **8**

1 Chaînes de caractères

1.1 Syntaxe exigible pour les chaînes de caractères

Cette section donne les différents éléments du langage python dont la connaissance est exigible car explicitement au programme d'informatique. Vous devez les connaître par cœur.

1.1.1 Création d'une chaîne de caractères

Une chaîne de caractères peut être écrite entre guillemets simples, doubles, ou triples. Les quatre instructions suivantes correspondent à la même chaîne de caractères.

```
>>> 'bonjour'
'bonjour'
>>> "bonjour"
'bonjour'
>>> '''bonjour'''
'bonjour'
>>> """bonjour"""
'bonjour'
>>> '' # Cas particulier : la chaîne vide
''
```

L'utilisation des différents types de guillemets permet de créer des chaînes de caractères contenant des guillemets :

```
>>> "aujourd'hui"
"aujourd'hui"
>>> 'je cite "du texte"'
'je cite "du texte"'
```

1.1.2 Concaténation et répétition

On peut **concaténer** des chaînes de caractères avec l'opérateur + et **répéter** des chaînes de caractères avec l'opérateur *.

```
>>> 'bon'+'jour'
'bonjour'
>>> 2*'bon'
'bonbon'
```

1.1.3 Accès à une chaîne de caractères

La fonction **len()** renvoie la longueur d'une chaîne :

```
>>> len('abcdefghijklmnopqrstuvwxyz')
26
```

On peut accéder aux éléments d'une chaîne de caractères par leurs indices, **le premier caractère a pour indice 0.**

```
>>> mot = 'Python'
>>> mot[0]
'P'
>>> mot[1]
'y'
>>> mot[5] # sixième lettre du mot
'n'
>>> len(mot)
6
```

On peut utiliser des indices négatifs pour compter en partant de la droite, le dernier caractère de la chaîne a pour indice -1.

```
>>> mot[-1]
'n'
>>> mot[-2]
'o'
```

On peut extraire des tranches d'une chaîne de caractères avec la syntaxe suivante (on parle de *slicing*) :

```
>>> mot[0:2] # Les caractères de l'indice 0 inclus à l'indice 2 exclu
'Py'
>>> mot[2:5] # Les caractères de l'indice 2 inclus à l'indice 5 exclu
'tho'
```

Les valeurs par défaut du premier et du deuxième indice valent respectivement 0 et la longueur de la chaîne de caractères :

```
>>> mot[:2] # Les caractères de l'indice 0 inclus à l'indice 2 exclu
'Py'
>>> mot[2:] # Les caractères de l'indice 2 inclus à la fin de la chaîne
'thon'
```

On peut aussi utiliser un troisième indice, qui correspond à un pas d'itération :

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> alphabet[0:15:2] # Les caractères de l'indice 0 inclus à l'indice 15 exclu, de 2 en 2.
'acegikmo'
>>> alphabet[0:15:3] # Les caractères de l'indice 0 inclus à l'indice 15 exclu, de 3 en 3.
'adgjm'
```

1.1.4 Itérer sur les caractères d'une chaîne de caractères

On peut itérer sur les caractères d'une chaîne de caractères de deux manières. Soit avec les indices (ci-dessous à gauche), soit avec une syntaxe plus légère (ci-dessous à droite). Les deux codes font la même chose : ils affichent successivement tous les caractères de la chaîne de caractères `chaine`.

```
chaine = 'bonjour'
for i in range(len(chaine)):
    caractere = chaine[i]
    print(caractere)
```

```
chaine = 'bonjour'
for caractere in chaine:
    print(caractere)
```

La syntaxe plus légère a l'inconvénient de ne pas mettre en évidence l'indice des caractères, qui est parfois utile.

1.2 Écriture de quelques fonctions avec des chaînes de caractères

Q.1 Écrivez une fonction `voyelles(s)` prenant en argument une chaîne de caractères `s` et renvoyant le nombre de voyelles dans cette chaîne de caractères. (Pour déterminer si un caractère est une voyelle, on peut utiliser des expressions booléennes comme `caractere == 'a'` ou bien `caractere in 'aeiouy'`.) Les appels `voyelles('astrophysique')`, `voyelles('abracadabra')` et `voyelles('grrr')` doivent respectivement renvoyer `6`, `5` et `0`.

Q.2 Écrivez une fonction `inverse_chaine(s)` prenant en argument une chaîne de caractères `s` et renvoyant la même chaîne dans l'ordre inverse. Par exemple, `inverse_chaine('retartiner')` doit renvoyer `'renitrater'`.

2 Listes

2.1 Syntaxe exigible pour les listes

Comme pour les chaînes de caractères, cette section donne les différents éléments du langage python dont la connaissance est exigible : vous devez les connaître par cœur.

2.1.1 Création de listes

Une liste est une collection d'objets python. On peut les créer en les écrivant explicitement. Remarquez qu'on peut mélanger différents types (entiers, flottants, chaînes de caractères, etc.) dans une liste.

```
>>> [2, 4, 6, 8]
[2, 4, 6, 8]
>>> [2, 4, 'bonjour']
[2, 4, 'bonjour']
```

On peut aussi créer des listes par compréhension :

```
>>> [2*x for x in range(1, 5)]
[2, 4, 6, 8]
```

2.1.2 Concaténation et répétition

Comme pour les chaînes de caractères, on peut **concaténer** des chaînes de caractères avec l'opérateur `+` et **répéter** des chaînes de caractères avec l'opérateur `*`. Remarquez que la répétition peut être utile pour créer une liste remplie de zéros.

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

2.1.3 Utilisation de append

Les listes sont ce qu'on appelle des types *muables* : on peut modifier leur contenu, par exemple en leur ajoutant un élément avec la méthode `append` :

```
>>> L = [1, 2, 3]
>>> L.append(4)
>>> L
[1, 2, 3, 4]
```

La méthode `append` peut être utilisée pour initialiser des listes à partir d'une liste vide. Le code ci-dessous :

```
L = [] # Liste vide
for i in range(10):
    L.append(i) # On ajoute les entiers strictement inférieurs à 10.
print(L)
```

affiche `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`.

Il faut préférer l'utilisation de append par rapport à l'opérateur + lorsque c'est possible. En effet, pour ajouter un élément à une liste, on peut envisager deux syntaxes :

```
>>> L = [1, 2, 3]
>>> L.append(4) # Avec append
>>> L
[1, 2, 3, 4]
>>> L = L + [5] # Avec +
>>> L
[1, 2, 3, 4, 5]
```

L'utilisation de `+` est moins efficace, car la ligne `L = L + [5]` est interprétée de la manière suivante :

- on évalue le membre de droite `L + [5]`,
- on stocke le résultat en mémoire, ce qui crée une copie temporaire de `L` à laquelle on a ajouté un élément,
- on affecte le résultat à la variable `L` : la nouvelle liste écrase l'ancienne.

Ces opérations ont demandé de créer une copie de la liste `L` en mémoire, alors que `append` ajoute un élément à une liste sans en faire de copie : `append` est plus efficace.

Remarque : `append` n'est pas définie pour les chaînes de caractères.

2.1.4 Accès à une liste

La fonction `len()` renvoie la longueur d'une liste :

```
>>> len([1, 2, 3, 4])
4
```

On peut accéder aux éléments d'une liste de caractères par leurs indices, exactement comme pour les chaînes de caractères, la technique du slicing est la même :

```
>>> L = [2, 4, 6, 8, 10]
>>> L[0]
2
>>> L[2]
6
>>> L[-1]
10
>>> L[1:3]
[4, 6]
>>> L[:2]
[2, 4]
```

```
>>> L[::3]
[2, 8] # etc.
```

2.1.5 Modifier les éléments d'une liste, caractère muable des listes

On peut modifier les éléments d'une liste :

```
>>> L = [1, 2, 4, 5]
>>> L[1] = 17
>>> L
[1, 17, 4, 5]
```

On remarque ici que les listes sont *muables* : on peut modifier leur contenu, par opposition aux chaînes de caractères qui sont immuables. La commande ci-dessous affiche un message d'erreur et `s` n'est pas modifiée :

```
>>> s = 'abcd'
>>> s[0] = 'A'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s
'abcd'
```

Si on veut changer le contenu d'une chaîne de caractères, il faut en créer une nouvelle, par exemple de cette manière :

```
>>> s1 = 'abcd'
>>> s2 = 'A' + s[1:]
>>> s1
'abcd'
>>> s2
'Abcd'
```

2.1.6 Utilisation de pop

La méthode `pop()` supprime le dernier élément d'une liste et le renvoie en valeur de retour :

```
>>> L
[1, 17, 4, 5]
>>> L = [1, 2, 3, 4]
>>> a = L.pop()
>>> L
[1, 2, 3]
>>> a
4
```

Cette méthode nous servira principalement à implémenter une structure de données appelée *pile* plus tard dans l'année. Elle n'est pas particulièrement utile ici.

Il faut préférer l'utilisation de pop par rapport au slicing. En effet, pour supprimer le dernier élément d'une liste, on peut envisager deux syntaxes :

```
>>> L = [1, 2, 3, 4]
>>> L.pop() # Avec pop
>>> L
[1, 2, 3]
>>> L = L[:-1] # Avec du slicing
>>> L
[1, 2]
```

L'utilisation du slicing est moins efficace, car la ligne `L = L[:-1]` est interprétée de la manière suivante :

— on évalue le membre de droite `L[:-1]`,

— on stocke le résultat en mémoire, ce qui crée une copie temporaire de `L` à laquelle on a retiré un élément,

— on affecte le résultat à la variable `L` : la nouvelle liste écrase l'ancienne.

Ces opérations ont demandé de créer une copie de la liste `L` en mémoire, alors que `pop` supprime le dernier élément d'une liste sans en faire de copie : `pop` est plus efficace.

Remarque : `pop` n'est pas définie pour les chaînes de caractères.

2.1.7 Itérer sur les éléments d'une liste

On peut itérer sur les éléments d'une liste de deux manières. Soit avec les indices (ci-dessous à gauche), soit avec une syntaxe plus légère (ci-dessous à droite). Les deux codes font la même chose : ils affichent successivement tous les éléments de la liste `L`.

```
L = [1, 2, 3, 4, 5, 6]
for i in range(len(L)):
    element = L[i]
    print(element)
```

```
L = [1, 2, 3, 4, 5, 6]
for element in L:
    print(element)
```

2.2 Écriture de quelques fonctions avec des listes

Q.3 Écrivez une fonction `moyenne(L)` prenant en argument une liste de nombres et renvoyant leur moyenne. Vous n'avez pas le droit d'utiliser la fonction `sum` intrinsèque de python. `moyenne([1, 2, 3, 4])` doit renvoyer `2.5`.

Q.4 On veut écrire une fonction `maximum(L)` prenant en argument une liste de nombres et renvoyant la valeur de leur maximum. Vous n'avez pas le droit d'utiliser la fonction `max` intrinsèque de python. Pour cela :

— Repérez le plus grand élément de la liste suivante : [1, 2, 5, 13, 14, 9, 8, 9, 15, 4, 3, 6].

— Écrivez, **obligatoirement sur une feuille de papier**, un pseudo-code correspondant à l'algorithme que vous avez mis en œuvre.

— Écrivez enfin la fonction python demandée. Vérifiez que l'appel `maximum([1, 9, 2, 3, 6])` renvoie bien `9`.

3 Exercices

3.1 Quelques fonctions complémentaires avec des listes

Q.5 Écrivez une fonction `indices_minimum(L)` prenant en argument une liste de nombres et renvoyant une liste contenant les indices `i` pour lesquels `L[i]` est minimum. Par exemple, `indices_minimum([5, 2, 3])` doit renvoyer `[1]` et `indices_minimum([4, -1, 2, -1])` doit renvoyer `[1, 3]`.

Q.6 Écrivez une fonction `pairs(L)` prenant en argument une liste `L` d'entiers et retournant la liste des éléments pairs de `L`. Par exemple, `pairs([1, 2, 3, 4, 2])` doit renvoyer `[2, 4, 2]`.

3.2 Recherche du second maximum d'une liste

Le second maximum d'une liste est sa deuxième plus grande valeur. Par exemple, le second maximum de `[1, 2, 14, 8, 20, 7]` est `14`.

Pour simplifier, on suppose que tous les éléments de la liste sont distincts.

Pour déterminer le second maximum d'une liste, on propose l'algorithme suivant :

— Initialiser deux variables m_1 et m_2 égales aux deux premiers éléments et telles que $m_1 > m_2$.

— Parcourir les éléments ℓ_i de la liste à partir du troisième élément :

  — Si $\ell_i > m_1$, alors ℓ_i est le nouveau maximum : mettre à jour $m_2 = m_1$ et $m_1 = \ell_i$.

— Si $m_2 < \ell_i < m_1$, alors ℓ_i est le nouveau second maximum : mettre à jour $m_2 = \ell_i$.

— Une fois qu'on a parcouru toute la liste m_2 est le second maximum.

Q.7 Écrivez une fonction `second_max(L)` renvoyant le second maximum d'une liste en s'appuyant sur l'algorithme donné ci-dessus. On supposera que le tableau contient au moins deux éléments.

3.3 Découpage d'une chaîne de caractères

Les chaînes de caractères en python disposent d'une méthode `split()` qui renvoie une liste des mots de la chaîne en utilisant l'espace (`' '`) comme séparateur de mot :

```
>>> s = "Une phrase dans une chaîne de caractères"
>>> s.split()
['Une', 'phrase', 'dans', 'une', 'chaîne', 'de', 'caractères']
```

Q.8 Écrivez une fonction `decoupe(s)` prenant en argument une chaîne de caractères et renvoyant la même liste que `s.split()` (sans utiliser `s.split()` évidemment !).

3.4 Crible d'Ératosthène

Q.9 Écrivez un programme qui établit la liste de tous les nombres premiers compris entre 1 et 1000 en utilisant la méthode du crible d'Ératosthène :

— Créer une liste de 1000 éléments, tous initialisés à la valeur 1

— Parcourir cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettre à zéro tous les autres éléments de la liste dont les indices sont des multiples entiers de l'indice de l'élément analysé (par exemple, avec l'indice 3, éliminer les éléments d'indice 6, 9, 12...)

— Afficher la liste des nombres premiers inférieurs à 1000.

3.5 Suite de Collatz

La suite de Collatz du nombre entier N est définie comme suit :

— $u_0 = N$

— si u_n est pair alors $u_{n+1} = \dfrac{u_n}{2}$

— si u_n est impair alors $u_{n+1} = 3u_n + 1$

— on arrête le calcul lorsque $u_n = 1$

Par exemple, pour $N = 13$, on obtient la suite : 13, 40, 20, 10, 5, 16, 8, 4, 2,1.
On conjecture que toutes les suites ainsi définies finissent par 1.

Q.10 Écrivez une fonction `collatz(N)` qui prend en argument un entier `N` et qui retourne une liste `L` contenant tous les éléments de la suite.

Par exemple `collatz(13)` retournera `[13,40,20,10,5,16,8,4,2,1]`

On peut tracer l'allure de cette suite à l'aide des commandes suivantes :

```
1 import matplotlib.pyplot as plt
2 plt.figure() # Initialise une nouvelle figure
3 plt.plot(collatz(13))
4 plt.show()
```

Q.11 Écrivez une fonction `plus_longue(M)` qui prend en argument un nombre entier `M` et qui renvoie l'entier $N < M + 1$ dont la suite de Collatz est la plus longue.

Par exemple `plus_longue(4)` renverra le nombre `3`.

Q.12 En vous inspirant du code ci-dessous, tracez l'allure de la plus longue suite telle que $N < 1000$.

3.6 Suite de Conway

La suite de Conway est une suite dont le premier terme est 1 et dans laquelle on trouve un terme en énonçant les chiffres qui composent le terme précédent. Les premiers termes sont donc : 1, 11, 21, 1211, 111221, 312211, etc.

Q.13 Écrivez une fonction `affiche_conway(n)` qui affiche les `n` premiers termes de la suite de Conway.
Par exemple, `affiche_conway(4)` doit afficher

```
1
11
21
1211
```

4 Complément : Conséquences pratique du caractère muable des listes en python

Q.14 Définissez la liste `semaine` contenant sept chaînes de caractères correspondant aux jours de la semaine (`'lundi'`, `'mardi'`, etc.) puis posez `week = semaine`. Remplacez le premier terme de `week` par `'monday'`. Que sont devenues les deux listes `semaine` et `week` ?

L'affectation en Python est l'association entre un nom de variable et une valeur. Lorsqu'on affecte une variable à une autre variable, on crée simplement un autre nom qui partage la même valeur en mémoire.

Cela ne pose pas de problème avec les types **`int`**, **`float`**, **`bool`** et **`str`** car les valeurs elle-mêmes ne sont pas modifiables (en Python en dit que ces types **`int`**, **`float`**, **`bool`**, **`str`** sont « immuables », pour changer la valeur associée à la variable il faut réaliser une nouvelle affectation).

De la même façon, l'affectation d'une variable à une autre variable d'une valeur de type **`list`**, ne crée pas une nouvelle liste qui soit une copie de la première, mais met simplement en place un nouveau nom qui référence la même liste en mémoire. Or il est possible de modifier directement cette valeur liste en utilisant les opérateurs et méthodes vus précédemment. Si deux variables référencent la même liste, les modifications réalisées en utilisant une variable seront visibles aussi avec l'autre variable.

On peut vérifier tout cela en utilisant la fonction **`id()`**, qui donne l'« identité » d'un objet python, c'est-à-dire son adresse en mémoire donnée sous la forme d'un entier.

Q.15 Vérifiez que **`id`**`(semaine)` et **`id`**`(week)` renvoient bien la même adresse.

Pour créer une copie d'une liste en étant sûr que les modifications ne toucheront pas la liste originale, il faut utiliser une des méthodes suivantes :

- On peut créer une nouvelle liste par compréhension : `liste2 = [x for x in liste1]`
- On peut créer une nouvelle liste avec une boucle **`for`** :

```
liste2 = []
for x in liste1:
    liste2.append(x)
```

- On peut aussi utiliser l'instruction suivante : `liste2 = liste1[:]`
- Enfin, les listes disposent d'une méthode `copy` : `liste2 = liste1.copy()`

Q.16 Testez ces instructions en réalisant quatre copies de la liste `semaine`, nommées `week1`, `week2`, `week3` et `week4`, et en vérifiant qu'on peut modifier les éléments de ces copies sans modifier la liste originale. Vérifiez que les identifiants de ces quatre listes sont bien différents entre eux, et différents de l'identifiant de `semaine`.