TP3 : premières manipulations de boucles et de listes

1 Un savoir faire essentiel : manipuler les boucles `for`

a) Ecrire un code qui affiche dix fois le mot `python` de deux manières : avec une boucle `for` et *sans* boucle for (en pensant aux opérateurs vus sur les chaînes de caractères).

b) Faire afficher tous les entiers pairs de 0 à 20.

c) Faire calculer $\sum_{i \text{ impair}, 0 \leq i \leq 100} i$.

d) Faire de même le calcul de $\prod_{i=1}^{100} i$.

2 Fabriquer une liste à l'aide d'une boucle

Supposons par exemple qu'on veuille fabriquer une liste `L` qui contient tous les i^2 pour $i \in [\![0, 100]\!]$ comment faire ?

a) Idée 0 (qui ne marche pas !) L'idée serait d'utiliser le code suivant :

```
for i in range(100):
    L[i]= i**2
```

Que pensez-vous du résultat ? Eh oui, c'est triste, mais c'est ainsi : les commandes `L[i]` ne peuvent que *modifier* l'entrée d'indice `i` d'une liste qui a *déjà* une entrée d'indice `i`. On ne peut pas *créer* une liste comme cela en Python [1] !

Ne jamais commencer la création d'une liste avec des `L[i]=` qui ne sont pas que des *modifications* d'une liste L qui doit déjà exister.

b) Idée 1 : la commande `+`, en partant d'une liste vide.

On rappelle que `[a,b,c]+[d]` renvoie `[a,b,c,d]`. Ainsi `[]+[a]` renvoie...

Quand on a calculé des sommes au § 0, on est parti d'une somme vide. De même ici, on peut partir d'une liste vide pour fabriquer une liste. Fabriquer de cette manière la liste `L` qui contient tous les i^2 pour $i \in [\![0, 100]\!]$ avec un `+` à chaque étape.

c) Idée 2 : où l'on rend possible la méthode qui ne marchait pas à l'idée 0, grâce au préformatage

On rappelle qu'on a un opérateur $*$ sur les listes. Par exemple `[0]*3` donne ...

Commencer par fabriquer une liste de `0` de la bonne taille, puis *une fois que cette liste existe* vous pouvez en modifier les entrées.

d) Idée 3 : la méthode `append`.

La syntaxe d'une *méthode* en PYTHON est très différente de celle des fonctions que nous avons vues pour l'instant. Par exemple si `L=[2]`, pour passer à `L=[2,3]`, on tapera :

```
L=[2]
L.append(3)
```

A vous de jouer.

Certains penseront : à quoi bon ce `append` alors que le `+` va bien ? On va voir plus loin... mais noter la syntaxe particulière des méthodes. Noter aussi ici que l'objet `L` (qui est une liste) est modifié par la méthode sans avoir à faire une affectation. Ne pas taper `L=L.append(3)`

e) Idée 4 : La fabrication de *listes par compréhension* nous verrons cela plus tard.

1. Cela marche dans d'autres langages comme Javascript...

3 Introduction à l'écriture de *fonctions*

a) Utiliser une fonction ?

On sait déjà utiliser des fonctions de PYTHON : par exemple, puisque ce T.P. parle de *listes*, si on a une liste `L=[1,4,5,2,16,3]`, on a, en PYTHON, une fonction `max` qui renvoie la valeur maximale des entrées de `L`. Essayez de l'utiliser.

b) Fabriquer une fonction ?

Un des buts de ce T.P. est de programmer des fonctions à vous. Pour cela, il faut introduire la syntaxe nécessaire à la *définition* d'une fonction. Nous reprendrons tout cela en cours ensuite !

On se contente ici d'un exemple. On a vu en cours comment faire calculer la somme des entiers de 1 à n si n est une valeur en mémoire.

```
n=12
S=0
for i in range(n+1):
    S=S+i
print(S)
```

Considérons alors le code suivant :

```
def Somme(n):
    S=0
    for i in range(n+1):
        S=S+i
    return S
```

Le premier code a été incorporé après la première ligne, mais indenté. Une fois ce nouveau code exécuté, vous pouvez taper `Somme(14)` dans le shell et vous aurez la somme des entiers de 1 à 14 comme valeur de retour. Nous reviendrons sur le `return` en cours : ce qui suit le `return` est la valeur (ou les valeurs) renvoyée(s) par la fonction. Le mot clef `return` n'est à utiliser qu'à l'intérieur de la définition d'une fonction.

c) Exercice : Reprendre le script que vous avez fait pour les années bissextiles et fabriquer une fonction `bissextile` qui prend comme argument une variable que vous appellerez `annee` et renvoie `True` ou `False` suivant que l'année est bissextile ou pas. Autrement dit ; une fois votre code exécuté, vous devrez pouvoir taper `bissextile(2000)` dans le shell, et il devra vous répondre `True`.....

4 Comment fabriquer une liste de nombres aléatoires pour nos tests

A l'aide de `from random import randint`, on dispose de la fonction `randint` qui fabrique un *entier aléatoire*.

Précisément `randint(a,b)` renvoie un entier aléatoire dans $[\![a, b]\!]$.

Fabriquer alors une fonction `liste_alea` qui prend un entier `n` comme entrée et qui fabrique une liste de `n` nombres entiers aléatoires entre 1 et 1000.

5 Maximum dans une liste ou un tuple :

Un algorithme fondamental à comprendre

a) Ecrire une fonction `mon_max` qui prend en entrée une liste ou un tuple, qu'on notera `L`, dont on suppose que les entrées sont des nombres, et qui retourne la plus grande de ces entrées.

Idée : créer une variable `M` qu'on initialise avec la valeur de la première entrée de la liste. Ensuite, on parcourt la liste en comparant chaque entrée à `M` et si l'entrée `L[i]` qu'on examine est plus grande que `M`, on met la valeur de `L[i]` dans `M`.

A la fin `M` doit contenir la valeur maximum des entrées de la liste.

b) Améliorer la fonction du a) pour qu'elle renvoie deux valeurs : la valeur du max. et un indice où ce max. est atteint.

6 Programmer le `del`

a) On a parlé en cours de la commande `del` (avec sa syntaxe un peu bizarre en python) : Si `L=[4,3,5,4]`, que fait `del(L[2])` ?

b) On a parlé en cours des commandes de `slicing` : par exemple pour `L=[4,3,5,4]` que donne `L[1:3]` ?

Mais on doit aussi dire qu'on peut *modifier non seulement une tranche d'une liste, même en changeant sa taille*. Par exemple pour la liste précédente, que donne `L[1:3]=[2]` ?

c) En déduire l'écriture d'une fonction `MonDelAMoi` qui prend comme argument une liste `L` et un numéro d'entrée `i`, donc qu'on utilisera avec la commande `MonDelAMoi(L,i)` et qui modifie `L` pour avoir le même résultat que `del(L[i])`.

7 Davantage de méthodes sur les listes :

L'aide de python (quand on l'a, notamment dans le terminal) donne les détails suivants sur les méthodes qui s'appliquent aux listes :

```
|  index(...)
|      L.index(value, [start, [stop]]) -> integer -- return first index of value.
|      Raises ValueError if the value is not present.
|
|  insert(...)
|      L.insert(index, object) -- insert object before index
|
|  pop(...)
|      L.pop([index]) -> item -- remove and return item at index (default last).
|      Raises IndexError if list is empty or index is out of range.
|
|  remove(...)
|      L.remove(value) -> None -- remove first occurrence of value.
|      Raises ValueError if the value is not present.
|
|  reverse(...)
|      L.reverse() -- reverse *IN PLACE*
|
|  sort(...)
|      L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

Travail à faire : Programmer des fonctions à vous qui font la même chose que ces méthodes. Il est utile de commencer par celle qui remplace `index`, que vous pouvez appeler `mon_indice` par exemple.

La règle du jeu est que vous pouvez à chaque étape utiliser toutes les fonctions maisons que vous avez déjà programmées. Il est intéressant aussi de savoir ce qu'on fait par exemple pour `mon_indice`, si l'entrée cherchée n'apparaît pas.

8 Comparaison de la rapidité de deux fonctions : le module `time`

a) Que fait le petit script suivant ?

```
from time import clock
def duree(fonction,n=100):
    debut=clock()
    fonction(n)
    fin=clock()
    return fin-debut
```

Remarque : le second argument de `duree` est un *argument optionnel* : si on ne l'entre pas i.e. qu'on entre seulement `duree(f)` où `f` est une fonction, alors `n=100` par défaut. En revanche `duree(f,1000)` donnera `n=1000`.

b) Retour sur le 1) : y-a-t-il une différence entre les techniques du 1.b), du 1.c) et du 1.d).

Tester la durée des différentes méthodes vues au 1), pour constituer la liste des `sin(i)` pour $i \in [\![0, 1000]\!]$. Pour appliquer la fonction durée, on pourra fabriquer des fonctions renvoyant cette liste des `sin(i)` pour $i \in [\![0, 1000$.