INTRODUCTION À PYTHON : FONCTIONS, LISTES (AVANCÉ)

1 Les instructions `print` et `return`

- La fonction `print(...)` permet d'afficher son résultat dans la console. Elle s'utilise toujours avec des parenthèses. Elle n'a d'intérêt que si on souhaite afficher un résultat.
- L'instruction `return` permet de déterminer ce que retourne une fonction, i.e. elle définit la valeur en sortie. Cela permet d'utiliser cette valeur dans une autre partie du code (par exemple la stocker dans une variable).

Recopier et compiler les scripts suivants :

```
1 def test(a) :
2     return a+1
```

```
1 def test2(a) :
2     print(a+1)
```

Puis, dans la console, taper les instructions `b = test(3)` puis `c = test2(3)`.

Question 1. Regarder le Workspace. Que contiennent les variables `b` et `c` ? Essayez de prédire ce que va afficher l'instruction `test(3)==test2(3)`, puis vérifier.

Comme vu en cours, **le mot-clé `return` met immédiatement fin à la fonction**. Cela peut être utilisé à notre avantage.

Exercice 1. Recopier, compléter et tester le script suivant.

```
1 def contientPair(L) :
2     # retourne True si la liste L contient un nombre pair, False sinon
3
4     for ... :         # on boucle sur chaque élément de L
5         if ... :      # on teste si cet élément est pair
6             return True
7     return False
```

Dans le script ci-dessus, dès qu'on trouve un élément pair, on retourne `True` et toute la fonction s'arrête. Si toutefois on arrive au bout de la boucle `for`, c'est qu'aucun élément pair n'a été trouvé. Ainsi, on retourne `False`. Cette utilisation du mot-clé `return` est typique d'un algorithme de **first-match** (trouver le premier élément dans un conteneur qui vérifie une condition).

Exercice 2. Écrire une fonction `premierNegatif` qui prend en argument une liste de nombres et qui retourne le premier élément négatif de cette liste. S'il n'y en a pas, on affiche « Pas d'élément négatif » et on ne retourne rien (`None`).

Question 2 (Une erreur fréquente)**.** On suppose qu'un élève a défini la fonction `contientPair` avec le script ci-dessous. *Sans le tester*, que va renvoyer `contientPair([1,2])` ?

```
1 def contientPair(L) :
2
3     for ... :         # on suppose le script complété comme ci-dessus
4         if ... :
5             return True
6         else :
7             return False
```

2 Slicing

On rappelle que l'instruction `L[d:f:p]` permet d'obtenir une sous-liste des éléments de `L`, qu'on obtient en partant de l'indice `d` inclus puis par pas de `p` jusqu'à atteindre ou dépasser l'indice `f` (qui est exclu). On rappelle que :

- `p` doit être non nul mais peut être négatif : dans ce cas, il faut avoir `d` $>$ `f` sinon `L[d:f:p]` sera une liste vide.
- Chacun des paramètres `d`, `f`, `p` peut être omis :
  - La valeur par défaut de `p` est un : `L[d:f:]` ou `L[d:f]` équivalent tous les deux à `L[d:f:1]`.
  - Si `p` >0, alors `L[::p]` équivaut à `L[0:n:p]` avec `n` la longueur de la liste : on va ainsi du début de la liste (indice `0`) à la fin de la liste (indice `n-1`) par pas de `p`.
  - Si `p` <0, alors `L[::p]` équivaut à `L[-1:-n-1:p]` avec `n` la longueur de la liste : on va ainsi de la *fin* de la liste (indice `-1`) au *début* de la liste (indice `-n`) par pas de `p`.

Question 3. Si on a `L=[0,1,2,3,4,5,6]`, que retournent les instructions suivantes ?

1. `L[2:4]`
2. `L[0:6:2]`
3. `L[::2]`
4. `L[-2] + L[-1]`
5. `L[-2:-5]`
6. `L[-4::-1]`

Exercice 3. La fonction `decoupage` suivante permet de découper une liste `L` en deux morceaux selon un indice `n` choisi. Compléter et tester.

```
def decoupage(L,n): # retourne deux listes "découpées" à partir de L

    L1 = ...  # les n premières valeurs de L
    L2 = ...  # les valeurs restantes de L

    return L1, L2   # sorties multiples
```

La fonction `decoupage` retourne deux objets séparés par une virgule : cela revient à écrire « `return (L1,L2)` ». On retourne donc un couple (type *tuple*). Tester dans la console les commandes suivantes :

```
L = [0,1,2,3]
n = 2
Tuple = decoupage(L,2)
L1,L2 = decoupage(L,2)
```

La dernière ligne est ce qu'on appelle une *affectation multiple* : on a affecté à `L1` la première composante du couple et à `L2` la seconde. De manière plus générale, les affectations multiples fonctionnent avec tout conteneur ordonné :

```
a,b,c = 1,2,3      a,b,c = (1,2,3)      a,b,c = [1,2,3]
```

Exercice 4. Écrire une fonction `inversion` qui à une liste donnée retourne cette même liste où l'ordre des éléments a été inversé. Ainsi, `inversion([1,2,3])` doit retourner la liste `[3,2,1]`.

Le slicing marche avec tout conteneur ordonné (liste, string, tuple notamment). Essayez les commandes suivantes :

```
inversion("Radar")     inversion("élu par cette crapule")
```

3 Convertisseurs

On connait beaucoup de types de Python :

`int, float, bool, list, str, tuple, ...`

Chacun des types ci-dessus dispose d'une fonction du même nom qui permet de convertir une valeur dans ce type. Tester dans la console :

```
int(2.5)
int(-1.3)
float(2)
float([1,3])     # la conversion liste -> flottant ne fonctionne pas
bool(0)
bool(-1)
list(8)          # list() doit prendre en argument un itérable
list(range(9))
tuple(8)         # tuple() est très similaire à liste
tuple("haha")
str(654.3)
str([4,3])
```

On notera que `int(x)` n'est PAS la partie entière de `x` mais la troncature de `x` : cela fait une différence si `x` est négatif et non entier.

Remarque. Lorsqu'on écrit `if <expr>`, Python convertit au préalable `<expr>` en booléen : cela revient donc à écrire `if bool(<expr>)`. Tester le script suivant :

```
if -1:
    print("ok")
```

Une telle utilisation n'est pas nécessaire, sauf avec une boucle `while` où on souhaite avoir une boucle infinie : on écrira alors « `while 1:` ». On pourra ensuite sortir de cette boucle avec, par exemple, une instruction « `return ...` ». Mais cela reste dangereux, il faut être sûr que cette boucle ne soit pas infinie !

La fonction suivante a une signature pour indiquer qu'elle prend en argument une liste et retourne une liste.

```
def listeEntier( L: list ) -> list : # fonction avec signature
    return ...
```

Exercice 5. Compléter le script ci-dessus : la fonction `listeEntier` prend en argument une liste et retourne la liste où chaque élément a été converti en entier. On utilisera une liste en compréhension.

Exercice 6. Écrire une fonction `arrondi` qui à un flottant positif x retourne un entier qui est la valeur arrondie de x : par exemple `arrondi(3.4)` retourne 3, tandis que `arrondi(3.5)` retourne 4.

Désormais, on pourra utiliser la fonction `round`, qui réalise l'arrondi ci-dessus. Attention aux nombres négatifs : `round(-3.5)` renverra `-4`. Enfin, `round` peut s'utiliser avec un deuxième argument pour préciser le nombre de décimales : essayer en console « `round(3.1415,2)` ».

4 L'instruction « `M = L` » avec des listes, effet de bord

Question 4. Recopier et tester le script suivant. Que remarque-t-on ?

```
L = [1,2,3]
M = L
M[0] = -4
print(M)
print(L)
```

Lorsqu'on a affaire à des listes (ou tout conteneur muable, comme les dictionnaires), l'instruction « `M = L` », ne réalise pas une copie : elle fait en sorte que `M` pointe vers la même zone mémoire que `L`.

Ainsi, toute modification de `L` modifie également `M` et réciproquement.

Pour éviter cela, on écrit « `M = L.copy()` », ou « `M = L + []` », ou encore le slicing « `M = L[:]` ». Tester la différence.

Compiler ces deux scripts. Que constate-t-on ?

```
def ajout(L,a):
    L.append(a) # effet de bord
    return L

L = [1,2,3]
M = ajout(L,4)
print(L,M)
```

```
def ajout2(L,a):
    L = L + [a] # pas d'effet ici
    return L

L = [1,2,3]
M = ajout2(L,4)
print(L,M)
```

La méthode `.append()` modifie directement la zone mémoire où est stockée `L`. Ainsi, la fonction `ajout` présente un **effet de bord** : elle modifie la valeur de la liste `L` qu'on lui donne en argument. Ainsi, même si on écrit juste « `ajout(L,4)` » sans stocker le résultat dans une variable `M`, cela va modifier la valeur de `L`. Il en va de même pour la majorité des méthodes pour les listes :

`L.reverse()` `L.remove(3)` `L.sort()` etc.

Exercice 7. La fonction `inversion` de l'exercice 4 présente-t-elle un effet de bord ? Écrire une fonction `inversionBis` de sorte que l'une des fonctions présente un effet de bord, et l'autre non.

5 Exercices pour s'entrainer

Exercice 8. Écrire une fonction `inception` qui prend en argument une *fonction* f et un flottant x et retourne $(f \circ f)(x)$.

Exercice 9. Écrire une fonction `partieEntiere` qui prend en argument un flottant et qui retourne sa partie entière. On testera notamment sur des flottants négatifs.

Exercice 10. Écrire une fonction `biggestDiv` qui prend en argument un entier $n \geq 2$ et qui retourne son plus grand diviseur, excepté lui-même. On utilisera obligatoirement une boucle `while`.

Exercice 11. Écrire une fonction `panachage` qui prend en argument une liste `L` et qui retourne la liste qui contient d'abord tous les éléments d'indices impairs de `L` (dans le même ordre) puis tous les éléments d'indices pairs de `L`.

Exercice 12 ($\star$)**.** Écrire une fonction `InvParPaquet` qui prend en argument une liste `L` et un entier $n \geq 2$. La fonction retourne ensuite une liste qui commence par les n derniers éléments de `L`, puis les n suivants (toujours en partant de la fin), etc. jusqu'aux premiers éléments de la liste `L`. Par exemple, avec les arguments

`L = [0,1,2,3,4,5,6,7,8,9]` `n = 3`

L'instruction `InvParPaquet(L,n)` doit renvoyer :

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