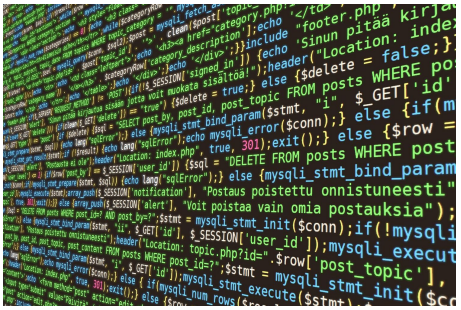
Programmation fonctionnelle 1/2



- Il existe plus de 2500 langages de programmation créés depuis les années 50 jusqu'à aujourd'hui ! Certains n'existaient pas il y a quelques années et d'autres émergeront probablement dans un futur proche.
- Certains langages sont adaptés à des tâches précises (SQL our les bases de données par exemple), d'autres plus généralistes comme python.
- On peut regrouper ces langages en familles, que l'on appelle des paradigmes, certains langages pouvant appartenir à plusieurs paradigmes, en voici quelques exemples :
  - Impératif : y appartiennent les langages qui permettent de manipuler des structures de données modifiables (variables, tableaux, etc...) en utilisant notamment des boucles (`while` , `for` , ...)
  - Orienté objet : On y définit des objets à l'aide de de classes, d'attributs et de méthodes.
  - Fonctionnel : On y trouve les langages qui manipule des fonctions et pas autre chose.
- Python appartient à ces trois paradigmes, mais ce n'est pas le seul. En revanche, certains langages sont par exemple exclusivement fonctionnels et ne laissent pas la possibilité d'utiliser un autre paradigme.
- Depuis le début de la première, vous avez appris à écrire beaucoup de fonctions. Nous allons nous intéresser un peu plus au paradigme fonctionnel et par exemple écrire des fonctions qui prennent en argument d'autres fonctions ou qui renvoient des fonctions en sortie. Nous verrons aussi que ce paradigme est lié aux structures de données *immuables* (qui ne peuvent plus être modifiées une fois construites).
- Dans le paradigme fonctionnel, le concept de fonction est celui des fonctions mathématiques : une ou des valeurs en entrée, et une valeur en sortie. On est sûr que pour des entrées données, on obtient toujours le même résultat.

1. Fonction pure

- Un programme écrit en style fonctionnel se caractérise essentiellement par une chose : l'absence *d'effets de bord*. Le code ne dépend pas de données se trouvant à l'extérieur de la fonction courante et il ne modifie pas des données à l'extérieur de cette fonction.
- Ainsi, une fonction pure est une fonction:
  - Dont le résultat dépend uniquement des entrées.
  - Qui n'a pas d'effet de bord.
  - Qui doit renvoyer quelque chose.

Exemple 1 :

In [2]:
```
#1. Fonction non fonctionnelle
a = 3
def increment(n):
    return n+a

increment(2)
```

Out[2]: 5

- Dans cet exemple, la fonction fait appel à une variable `a` qui est définie en dehors du corps de la fonction, il s'agit d'un effet de bord. Ainsi, le résultat de cette fonction ne dépend pas que de ces entrées, ce n'est pas une fonction pure.

In [13]:
```
#2. Fonction fonctionnelle
def increment(n,a):
    return n+a

increment(2,3)
```

Out[13]: 5

- Ici, le résultat de la fonction ne dépend que de ce qui lui aura été passé en argument. On pourrait imaginer utiliser le code de cette fonction dans un autre programme avec un copier-coller, sans aucune modification.

Remarques :

- En limitant les effets de bords, on rend les fonctions autonomes ce qui présente plusieurs avantages :
  - Propreté : le style de programmation fonctionnel réduit la proportion de défauts en s'imposant des règles telles que l'interdiction de modifier l'état d'un objet ou l'obligation de garantir qu'une fonction ne doit avoir aucun impact sur le reste du programme.
  - Test : le style de programmation fonctionnel favorise des fonctions pures qui manipulent de simples valeurs et qui sont donc extrêmement faciles à tester les unes indépendemment des autres.
  - Economie : le style fonctionnel permet de réduire le code.
  - Modularité :
    - Le découpage d'un code en fonctions rend le code plus simple à comprendre, et plus évolutif.
    - On peut utiliser ces fonctions, regroupées par exemple en bibiothèque dans plusieurs programmes.

Exercice 1 :

1. que fait le code suivant ?

In [13]:
```
T=[1,2,3]
n=4
dernier=T[-1]
for i in range(n):
    T.append(dernier+1)
    dernier=T[-1]

print(T)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Réponse:

- **2.** Modifiez ce code pour en faire une fonction pure, donc sans effet de bord. Ainsi cette fonction ne devra pas modifier la liste existante, mais en créer une nouvelle.

In [20]:
```
T=[1,2,3]









#
```

2. Fonction passée en argument.

- Dans le paradigme fonctionnnel, certaines fonctions sont emblématiques : `map` , `reduce` et `filter` . Elles permettent d'agir sur des tableaux sans avoir à écrire le code qui permet de les parcourir.
- Elles ont la particularité de prendre comme paramètre...une fonction. Dans les langages fonctionnels, une fonction est une donnée comme une autre.
- Pour comprendre le fonctionnement de ces fonctions, dans les exercices 2, 3 et 4, nous allons les construire sous un autre nom, puis les utiliserons pour résoudre quelques problèmes.
- Vocabulaire : on nomme ici *itérable* un objet sur lequel on peut itérer, c'est à dire pour simplifier parcourir ses éléments, directement ou à l'aide d'indices (liste python, chaîne de caractères, dictionnaires, tuples, etc...)

Exercice 2: map

- La fonction `applique` prend deux paramètres : une fonction et un itérable. Elle renvoie une nouvel itérable contenant les images par la fonction des éléments de l'itérable initial.

1. Que doit renvoyer l'appel de `applique(len , ['Joe','Jack','William','Averell']` ?

Réponse :

-

2. En supposant que la fonction `double(x)` renvoie le double de la valeur `x` passée en paramètre, que doit renvoyer `applique(double,[1,2,3,4])` ?

Réponse :

-

3. Compléter la fonction `applique` ci-dessous :

In [10]:
```
def double(x):
    return 2*x

def applique(fonction, liste):
    res=[]









    return res
```

In [11]:
```
#test 1
applique(len,['Joe','Jack','William','Averell'])
```

Out[11]: [3, 4, 7, 7]

In [12]:
```
#test 2
applique(double,[1,2,3,4])
```

Out[12]: [2, 4, 6, 8]

1. La fonction `map` existe sous python. Elle s'utilise presque comme celle que vous venez d'écrire, mais elle renvoie un objet que l'on peut ensuite convertir en liste avec la fonction `list()` . Utiliser `map()` pour retrouver les résultats des tests précédents.

In [13]:
```

```

Out[13]: [3, 4, 7, 7]

In [14]:
```

```

Out[14]: [2, 4, 6, 8]

Exercice 3 : filter

- La fonction `filtre` prend deux paramètres : une fonction booléènne (qui renvoie `True` ou `False`) et un itérateur. Elle renvoie une nouvel itérateur contenant les éléments filtrés à l'aide de la fonction booléènne.

1. En supposant que la fonction `pair(n)` renvoie `True` si et seulement si n est pair, que renvoie `filtre(pair,[1,2,3,4]`) ?

Réponse:

-

2. Compléter la fonction `filtre()` ci-dessous :

In [1]:
```
def pair(n):
    return n%2==0

def filtre(fonction, liste):
    res=[]




    return res
```

In [7]:
```
filtre(pair,[1,2,3,4])
```

Out[7]: [2, 4]

3. La fonction `filter` existe sous python. Elle s'utilise presque comme celle que vous venez d'écrire, mais elle renvoie un objet que l'on peut ensuite convertir en liste avec la fonction `list()`. Utiliser `filter()` pour retrouver le résultat du test précédent.

In [5]:
```

```

Out[5]: [2, 4]

Exercice 4 : reduce

La fonction `reduit` est un peut plus délicate à comprendre. Elle prend en paramètre une fonction et un itérable et renvoie une combinaison des éléments de l'itérable. Au premier appel, les deux premiers éléments de l'itérable sont passés en paramètres. Ensuite, le résultat de cet appel et l'élément suivant de l'itérable sont passés en paramètres, et ainsi de suite...

1. En supposant que la fonction `g(x,y)` renvoie la somme des éléments `x` et `y`, que renvoie `reduit(g,['a','b','c','d'])` ?

Réponse :

-

2. Que renvoie `reduit(g,[1,2,3,4])` ?

Réponse :

-

3. Compléter la fonction `reduit` ci-dessous :

In [2]:
```
def g(x,y):
    return x+y

def reduit(fonction,liste):
    i=0
    valeur=liste[i]




    return valeur
```

In [3]:
```
reduit(g,['a','b','c','d'])
```

Out[3]: 'abcd'

In [4]:
```
reduit(g,[1,2,3,4])
```

Out[4]: 10

4. La fonction `reduce` existe sous python. Elle s'utilise comme celle que vous venez d'écrire, mais elle renvoie un objet Utiliser `reduce()` pour retrouver le résultat du test précédent.

Remarque : Cette fonction n'est pas disponible par défaut sur python. Elle fait partie du module `functools` qu'il faut donc importer

In [5]:

Out[5]: 10

In [6]:

Out[6]: 'abcd'

3. Fonction anonyme

- Dans les fonctions `map` , `filter` et `reduce` , l'un des arguments est une fonction qui elle même doit être définie. Il existe une façon de faire qui permet de se passer du mot-clé `def` , et d'intégrer directement la définition de la fonction comme paramètre de `map` , `filter` ou `reduce` (ou toute autre fonction qu prend en argument une fonction).
- Sous python, l'opérateur `lambda` permet de créer des fonctions "à la volée" en une seule ligne.
- On parle de *fonction anonyme* car ces fonctions n'existent pas en dehors de la ligne à la quelle elle ont été créées.

Exemple 2:

Reprenons l'exercice 2 et l'utilisation de `map` . Plutôt que de définir la fonction carré puis de l'appeler dans `map` , on peut utiliser l'opérateur `lambda` :

In [14]:
```
res=map( lambda x:2*x , [1,2,3,4])
list(res)
```

Out[14]: [2, 4, 6, 8]

On peut lire ici `lambda x:2*x` comme "Une fonction qui prend en paramètre x et qui renvoie x^2."

Exemple 3 :

Voici quelques exemples de syntaxes utilisables pour les fonctions lambda :

- `lambda n : n//2 if n%2==0 else 3*n+1` : Une fonction qui renvoie le quotient de n dans la division par 2 s'il est pair, et qui renvoie $3n+1$ sinon.
- `lambda x,y : x+y` : Une fonction qui renvoie la somme des valeurs x et y, ou leur concaténation si x et y sont des chaînes de caractères.

Exercice 5 : lambda

On considère les variables ci-dessous

In [3]:
```
textes=['numérique','science','informatique']
nombres=[1,2,3,4]
```

En utilisant une fonction lambda et `map` , `filter` ou `reduce` , répondre à chaque question en utilisant une seule ligne code(on appelle cela un *one-liner*).

1. Renvoyer la liste des carrés des éléments de `nombres` (le résultat doit être `[1,4,9,16]`)

In [20]:
```
#1.
```

Out[20]: [1, 4, 9, 16]

2. Renvoyer les éléments impairs de `nombres` dans une liste (le résultat doit être `[1,3]`).

In [21]:
```
#2.
```

Out[21]:
```
[1, 3]
```

3. Renvoyer le produit des éléments de `nombres` (le résultat doit être 24)

In [25]:
```
#3.
```

Out[25]:
```
24
```

4. Convertir les éléments de `textes` en majuscules (on pourra utiliser `.upper()`) et les renvoyer dans une liste (le résultat doit être : `['NUMÉRIQUE', 'SCIENCE', 'INFORMATIQUE']`)

In [22]:
```
#4.
```

Out[22]:
```
['NUMÉRIQUE', 'SCIENCE', 'INFORMATIQUE']
```

5. Ne garder que le premier caractère de chaque élément de `textes` , le convertir , et renvoyer le tout dans une liste. (le résultat doit être : `['N','S','I']`)

In [23]:
```
#5.
```

Out[23]:
```
['N', 'S', 'I']
```

6. Ne garder que le premier caractère de chaque élément de `textes` , le convertir en majuscule, et concaténer le tout pour obtenir `'N.S.I'` .

In [24]:
```
#6.
```

Out[24]:
```
'N.S.I'
```

4. Fonction renvoyée comme résultat

- Comme déjà dit auparavant, une fonction est une donnée comme une autre, elle peut donc être aussi renvoyée comme sortie d'une autre fonction !

Exercice 6 :

On considère la fonction ci-dessous :

In [1]:
```
def f(n):
    def g(m):
        return m**n
    return g
```

1. En effectuant quelques tests, trouver comment utiliser la fonction ci-dessous :

In [4]:
```
#tests
```

Out[4]:
```
9
```

2. Simplfier le code de `f` en remplaçant le code de la fonction `g` par une fonction `lambda` :

In [3]:
```

```

In [4]:
```
#tests
```

Out[4]:
```
9
```

5. Structures immuables

- Un autre aspect de la programmation fonctionnelle est l'utilisation de structures de données *immuables*. Il s'agit d'une structures de données comme un tableau, une chaîne de caractères, un dictionnaire, que l'on ne modifie plus après sa création. On l'utilise bien sûr pour la consulter ou pour créer de nouvelles strucutres.
- Avec Python, on est plutôt habitué pour l'instant à une programmation ou les structures de données sont *mutables*, c'est à dire que leur contenu est modifié par des opérations(affecter de nouvelles cases au tableau, ajouter de nouvelles clés dans un dictionnaire,...)
- Le paradigme fonctionnel incite le programmeur non pas à modifier une structure existante, mais plutôt à créer une nouvelle structure à partir de celle existante, ainsi on évite les effets de bords.

Exemple 4 :

Python est langage qui permet de programmer dans différents paradigmes. Par exemple, les tableaux permettent des opérations *impératives* mais aussi *fonctionnelles*.

In [7]:
```
#1. C'est le même tableau qui change d'état au fur et à mesure
tab_depart=[1,1]
tab_depart.append(2)
tab_depart.append(3)
tab_depart
```

Out[7]:
```
[1, 1, 2, 3]
```

In [1]:
```
#2.On ne modifie pas les variables de départ. On en crée d'autres à partir de celles-ci.
tab_depart=[1,1]
tab = tab_depart+[2,3]
tab
```

Out[1]:
```
[1, 1, 2, 3]
```

Exemple 5 : Le cas des listes python

In [3]:
```
L1=[1,2,3,4]
L2=L1

L1.append(5)
print(L1)
print(L2)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Contrairement à e que l'on pourrait attendre, la copie L2 de la liste L1 aussi a été modifiée… Une liste en Python ne contient en réalité que l'adresse mémoire où sont stockés ses éléments ; c'est ce que l'on appelle un pointeur. Ici, le fait d'écrire `L2=L1` ne fait qu'ajouter un *alias* de la même liste mais c'est bien la même adresse mémoire qui est référencée.

Cette manière de procéder permet un gain de temps (copie rapide) et de mémoire (une seule plage mémoire pour deux variables). Mais elle pose un problème technique : si le contenu de la plage mémoire de la valeur de L1 est modifié cela entraine automatiquement la modification de la valeur de L2 ! Pour le programmeur Python non averti, cet effet de bord, peut provoquer une erreur de programmation difficile à élucider.

Si l'on veut réellement copier une liste, on peut utiliser utiliser la méthode `deepcopy()` du module standard `copy` qui effectue une « copie profonde »:

In [2]:
```
from copy import deepcopy

L1=[1,2,3,4]
L2=deepcopy(L1)

L1.append(5)
print(L1)
print(L2)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

Exercice 7 : immuables/mutables

En python, un type de données est soit *mutable*(on peut le mettre jour), soit *immuable*(la valeur d'une variable de ce type ne peut changer que par l'affectation d'une nouvelle valeur à cette variable).

Considérons le type de données entier ("int"), et déterminons s'il est immuable ou mutable:

- La fonction `id(x)` de python indique l'adresse mémoire(pour simplifier) à laquelle est stockée la valeur de la variable `x`.

In [18]:
```
#int
a=5
b=a
print(a,id(a))
print(b,id(b))
```

```
5 140711551637280
5 140711551637280
```

Les lignes ci-dessus indiquent le même identifiant pour `a` et `b`, ces variables pointent vers la même adresse (celle à laquelle est stockée le nombre 5).

- Modifions la valeur de `a` :

In [19]:
```
a=a+1
print(a, id(a))
print(b,id(b))
```

```
6 140711551637312
5 140711551637280
```

Cette fois l'identifiant de `a` a changé, mais pas celui de `b` qui pointe toujours vers le nombre 5.

La modification de la valeur de `a`, n'a aucun effet sur la valeur de `b`.

Ceci est vrai quelque soit l'entier. Le type entier est *immuable*.

En utilisant un raisonnement analogue, compléter le tableau ci-dessous en indiquant pour chaque type de données s'il est *immuable* ou *mutable*.

Type	Immuable	Mutable
int	X	
float		
bool		
string		
tuple		
list		
dict		

In [2]:
```
#float




#
```

In [3]:
```
#bool


#
```

In [4]:
```
#string


#
```

In [5]:
```
#tuple


#
```

In [6]:
```
#list


#
```

In [7]:
```
#dict


#
```

6. Conclusion

- Au delà de la connaissance de plusieurs langages de programmation qui sont par ailleurs en constante évolution, il est important de pouvoir penser dans plusieurs paradigmes de programmation (impératif, objet ou encore fonctionnel). En effet, il sera plus facile d'utiliser le paradigme objet dans certains cas alors que dans d'autres situations, l'utilisation du paradigme fonctionnel sera préférable.
- Être capable de Choisir le "bon" paradigme en fonction des situations fait partie du bagage de tout bon programmeur.