Exemple :

Voici une fonction Python où un effet de bord est présent :

```
l = [1, 2, 4]
def fct():
  l.append(7)
fct()
fct()
```

Quel sera le contenu du tableau l après exécution de ce programme ?

Le paradigme fonctionnel vise à éviter ces effets de bords : on va chercher au maximum à utiliser des **fonctions** qui ne modifieront pas les variables globales.

Les fonctions utilisées en programmation fonctionnelle sont parfois appelées "fonction pure" : le résultat retourné par une fonction pure doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction. Elle ne génère donc pas d'effets de bord.

❐ — À faire vous-même **1** —

Voici un programme Python contenant une fonction.

```
1 i = 4
2 def fct(a):
3   if i > 5: return True
4   else: return False
5
6 fct(10)
```

- Ce programme va-t-il fonctionner ?
  Fonctionnerait-il si la ligne i = 5 était positionnée à la ligne 7 ?
- Que va renvoyer la fonction fct() ? En déduire si la fonction fct est une fonction "pure".

❐ — À faire vous-même **2** —

Voici un programme Python contenant une fonction.

```
1 i = 4
2 def fct(i):
3   if i > 5: return True
4   else: return False
5
6 fct(10)
```

- ❖ Ce programme va-t-il fonctionner ?
  Fonctionnerait-il si la ligne i = 5 était positionnée à la ligne 7 ?
- ❖ Que va renvoyer la fonction fct(10) ? En déduire si la fonction fct est une fonction "pure".

Avec Python, on est habitué à une programmation impérative où les structures de données sont mutables (leur contenu est modifiable). Ainsi, généralement on affecte de nouveaux éléments à des dictionnaires ou à des tableaux déjà existants.
Dans la suite, nous allons travailler sur des structures de données immuables en accord avec le paradigme fonctionnel (voir exercices).

II. Paradigme fonctionnel

1) Fonction en tant qu'argument

Dans cette section, on va étudier le paradigme fonctionnel au travers d'un exemple. On considère donc le tri par insertion étudiée l'an dernier sous la forme de la fonction `tri_par_insertion(t)`. Celle-ci reçoit un tableau t en entrée et trie, **en place**, les éléments du tableau dans l'ordre croissant.

❐ — À faire vous-même **3** —

Voici un programme Python contenant deux fonctions.

```
def insere(L: list, i: int, v: float)-> list:
    j = i
    while j>0 and v < L[j-1]:
        L[j] = L[j-1]
        j = j-1
    L[j] = v

def triInsertion(t: list):
    for i in range(1, len(t)):
        insere(t, i, t[i])
```

- ❖ Rappelez le fonctionnement général du tri par insertion.
- ❖ Expliquez précisément le fonctionnement de la fonction `insere()`.
- ❖ En ajoutant quelques lignes, effectuez le tri du tableau [2,1,4,-1].
- ❖ Comment les comparaisons entre éléments sont-elles effectuées ?

On voit que les comparaisons sont ici effectuées grâce à l'opérateur < de Python. Cet opérateur est assez générique, fonctionnant sur des entiers, des flottants ou des chaines de caractères. Elle fonctionne même sur des tableaux de tableaux !

❐ — À faire vous-même **4** —

- Reprenez le code précédent et effectuez le tri du tableau de tableaux :

  `[[2,10,1,-1], [5,1,4,-1], [2,1,3,-1], [8,3,2,-1], [8,1,5,-1]]`

- Affichez puis expliquez le résultat obtenu. Comment l'opérateur < de Python fonctionne-t-il sur les tableaux de tableaux ?
- Modifiez le code précédent afin de trier le tableau de tableaux en fonction de l'indice numéro 2.

Vous avez sans doute procédé à un copier/coller du code précédent suivi d'une modification de la **relation d'ordre <.** De la même manière et pour répondre à des besoins particuliers, il est toujours possible de faire des copier/coller puis de modifier la fonction impactée. Toutefois, effectuer une telle opération est assez déplaisant. De plus, si une erreur existe dans le code initial de la fonction `insere`, chaque copier/coller va être impacté.

Pour éviter ce type de problème, nous devons changer de paradigme et accepter que la **relation d'ordre <** devienne une fonction que la fonction `insere` (et donc `triInsertion`) prenne en argument d'entrée : c'est une des émanations du paradigme fonctionnel.

❐ — À faire vous-même **5** —

Le code ci-dessous est écrit en paradigme fonctionnel :

```
def insere(inf, L: list, i: int, v: float)-> list:
    j = i
    while j>0 and inf(v, L[j-1]):
        L[j] = L[j-1]
        j = j-1
    L[j] = v

def triInsertion(inf, t: list):
    for i in range(1, len(t)):
        insere(inf, t, i, t[i])
```

- Écrire une fonction `inf1(x : int, y: int)` permettant de comparer deux entiers x et y à l'aide de la relation d'ordre <.
- En appelant la fonction `triInsertion()`, effectuez à nouveau le tri de L = [2,1,4,-1].

❐ — À faire vous-même **6** —

Reprenez la fonction `inf1` afin de trier le tableau suivant l'élément numéro 2 du tableau de tableau ci-dessous :

```
[[2,10,1,-1], [5,1,4,-1], [2,1,3,-1], [8,3,2,-1], [8,1,5,-1]]
```

Rem : L'écriture d'une fonction inf1 est un peu pénible car, finalement, elle n'utilise que l'opérateur Python < ! Pour pouvoir améliorer la lisibilité, Python a donc introduit le concept de fonction anonyme, nommée `lambda` . Ces fonctions n'ont pas de **nom**, n'ont pas le mot-clé **return** et n'ont pas de parenthèses autour des variables.

Exemple :

Nous pouvons obtenir le même résultat que dans le à faire vous-même 6 en écrivant simplement :

```
triInsertion3((lambda x, y : x[2] < y[2]), L)
```

Comme pour les fonctions, nous pouvons également donner des valeurs par défaut aux fonctions anonymes :

```
triInsertion3((lambda x, y, i=2 : x[i] < y[i]), L)
```

Exercice :

Un tableau Python contient n valeurs $x_1, x_2, \ldots, x_n$.

On souhaite calculer le résultat de $x_1 \oplus x_2 \oplus \ldots \oplus x_n$ avec $\oplus$ une opération quelconque.

1) Compléter la fonction `calcul` qui prend en argument une opération $\oplus$ ainsi qu'un tableau t et renvoie le résultat de $x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

```
def calcul(op, t):
    """ calcule t[0] op t[1] op ... t[n-1],
    le tableau t étant supposé non vide"""
    v = t[0]
    for i in ... ... ... ... ...:
        v = ... ... ... ... ...
    return v
```

2) En utilisant des fonctions lambda et le tableau t = [1,2,3,4,5] , calculez :
   a) le résultat de l'opération si $\oplus$ est l'addition
   b) le résultat de l'opération si $\oplus$ est la multiplication
   c) le résultat de l'opération si a$\oplus$b = max(a,b), où max(a,b) est le maximum de a et b

3) Application à la valeur de racine de 2.

Une fraction continue est une fraction de la forme $a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}}$.

On considère le tableau $t = [a_0, a_1, a_2, a_3 \dots]$. En utilisant la fonction `calcul` précédente, une fonction lambda ainsi que le tableau t = [2,2,2,2,1], trouvez le résultat de l'opération sachant que l'opérateur $\oplus$ doit maintenant permettre de calculer une fraction continue.

2) Fonctions comme résultats

En programmation fonctionnelle, les fonctions peuvent également renvoyer une fonction comme résultat. Nous allons illustrer cette section avec le calcul de la dérivée d'une fonction mathématique donnée.
Nous rappelons que la dérivée d'une fonction en un point x_0 est :

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

❐ — À faire vous-même **7** —

- Rappelez quelle sera la méthode numérique pour remplacer le symbole limite (h vers 0).
- Écrire une fonction `deriveImperative(f, x0)` qui calcule la dérivée d'une fonction f en un point x_0 dans le paradigme impératif.
- À partir du module math de Python, importez la fonction sinus et à l'aide de votre fonction `deriveImperative`, calculez la dérivée du sinus en $x_0 = 0$ et stockez le résultat dans une variable d. Donnez le type de cette variable.

Le problème principale de l'approche ci-dessus est que nous ne disposons pas d'un objet fonction en sortie. En effet, intuitivement, une dérivée **est** une fonction. Or, dans la paradigme impératif, nous ne disposons que d'un nombre !

❐ — À faire vous-même **8** —

- Recopiez la fonction écrite ci-dessous, écrite en paradigme fonctionnel.

```
def derive(f):
    h = 1e-7
    return lambda x0 : (f(h+x0) - f(x0))/h
```

- ❖ Expliquez la 3ème ligne.
- ❖ À partir du module math de Python, importez la fonction sinus et à l'aide de votre fonction `deriveImperative`, calculez la dérivée du sinus. Stockez le résultat dans une variable d. Donnez le type de cette variable.
- ❖ Calculez à présent la valeur de la dérivée du sinus en $x_0 = 0$ puis en $x = \frac{\pi}{2}$.
- ❖ Faire de même pour la fonction $x \mapsto x^3 - x^2 + 1$ pour x_0 variant de -3 à 3.

Conclusion :

La programmation fonctionnelle permet donc de manipuler des fonctions plutôt que des variables. Elle est utilisée principalement pour simplifier les opérations vectorielles et fonctionnelles. En Python, cela se traduit par l'utilisation intensive des fonctions anonymes lambda, créées dans le but spécifique.