1 Notion de paradigme de programmation

> Un paradigme de programmation est une façon d'envisager la conception et la manière d'écrire un programme informatique.

Nous évoquerons ici les paradigmes **impératif**, **fonctionnel** ou **objet**. Mais d'autres existent encore.

Certains langages imposent la programmation selon un certain paradigme, mais beaucoup de langages, comme Python par exemple, permettent de programmer selon différents paradigmes au sein d'un même programme.

2 Paradigme impératif

Le paradigme principal que vous avez rencontré jusqu'à maintenant est le **paradigme impératif**.

> Le paradigme **impératif** consiste à écrire une suite d'instructions que le programme doit réaliser dans l'ordre où elles apparaissent, avec toutefois d'éventuels branchements (liés aux structures conditionnelles et aux boucles).

C'est le paradigme qui reprend de manière la plus proche (simplifiée, heureusement) celui du langage machine.

Le résultat produit par un programme écrit suivant le paradigme impératif dépend directement de sa chronologie d'exécution. En effet, les programmes impératifs contiennent des **variables qui peuvent êtres modifiées au cours de l'exécution** du programme. L'état de la mémoire varie au cours du temps.
Le résultat d'une instruction ou d'un appel de fonction dépend de l'état interne de l'exécution. Ce point rend a priori les programmes écrits selon ce paradigmes difficiles à prouver corrects ou à déboguer.

La programmation impérative est souvent accompagnée du **paradigme procédural** qui autorise la définition de **procédures ou fonctions**, annexes au corps du programme principal, qui ont pour rôle d'effectuer une tache particulière (pour structurer le programme et faciliter la modularité). Chacune de ces procédures respectent alors le paradigme impératif.

Exemple : le langage C est un langage impératif de bas niveaux développé par Dennis Ritchie et Kenneth Thompson en 1972 pour réécrire le système d'exploitation Unix. Il reste très utilisé aujourd'hui et il est le père de nombreux autres langages impératifs (ex : Perl, PHP, Java, Javascript).

3 Paradigme fonctionnel

Comme son nom l'indique, le **paradigme fonctionnel** repose sur l'utilisation de **fonctions**.

La notion de fonction existe aussi lorsqu'on programme de façon impérative, mais dans la programmation fonctionnelle cette notion correspond plus au concept mathématique de fonction :

> Une fonction reçoit une entrée et renvoie une **sortie qui ne dépend que de cette entrée** (et jamais d'un état antérieur d'exécution du programme par exemple).

La sortie est toujours la même pour une certaine entrée quel que soit « l'historique » de l'exécution du programme.

> Par ailleurs, l'exécution d'une fonction ne modifie jamais le contenu d'une variable : on dit qu'il n'y a **aucun effet de bord** (même pas d'écriture à l'écran).

Enfin, la programmation fonctionnelle repose beaucoup sur la composition de fonctions : on peut appliquer une fonction à une autre fonction :

> En programmation fonctionnelle, **une fonction peut être passée en entrée** d'une autre fonction.

Signalons enfin que la programmation fonctionnelle fait **très souvent appel à des structures récursives**. Comme il est interdit de modifier le contenu d'une variable, **les boucles (for ou while) n'existent pas** !

Exemple : le langage fonctionnel le plus ancien est Lisp, créé en 1958 par McCarthy. Lisp a donné naissance à des variantes telles que Scheme et Common Lisp. Des langages fonctionnels plus récents tels Haskell, Caml, ou Erlang sont aussi des représentants phares de langages fonctionnels.

Exemple strict :
La fonction `map` est une fonction « classique » des langages fonctionnels : elle permet d'appliquer une fonction à chaque élément d'une liste. Voici une implémentation en Python de cette fonction :

```
def mapper(f, tab):
    if tab == []:
        return []
    else:
        return [f(tab[0])] + mapper(f, tab[1:])

def carre(x):
   """ Renvoie le carré du nombre x """
   return x**2

>>> tab = [0, 1, 2, 3, 4, 5]
>>> mapper(carre, tab)
[0, 1, 4, 9, 16, 25]
>>> tab
[0, 1, 2, 3, 4, 5] # tab n'a pas été modifié !
```

Contre-exemple :

```
def incremente(t):
   """ Incrémente chaque élément d'un tableau t (modification en place).
   Entrée : t : tableau d'éléments à incrémenter
   """
   for i in range(len(t)):
      t[i] += increment # paramètre externe
   return t # facultatif

>>> tab = [0, 1, 2, 3, 4, 5]
>>> increment = 5
>>> incremente(tab)
[5, 6, 7, 8, 9, 10]
# tab a été modifié : ne respecte pas le paradigme fonctionnel !
>>> tab
[5, 6, 7, 8, 9, 10]
```

```
# autre interdit : l'appel de la même fonction avec la même entrée produit un autre
↪  effet !
>>> tab = [0, 1, 2, 3, 4, 5] # même tableau initial
>>> increment = 1
>>> incremente(tab)
[1, 2, 3, 4, 5, 6]
```

Le respect du paradigme fonctionnel peut **ajouter de la robustesse** au programme et éviter certains bugs difficiles à déceler, notamment lorsque les fonctions modifient des variables en place ou utilisent des paramètres externes (voir contre-exemple ci-dessus).
Cela facilite aussi la programmation parallèle. Deux processus ne risquent pas de modifier des variables qu'ils pourraient partager.

Voir complément sur la doc de Python (How to) :
`https://docs.python.org/fr/3/howto/functional.html`

Voir aussi mon cours de 1ère P6-2-Diversité des langages.

4 Paradigme objet

Le paradigme objet conduit à la **programmation orientée objet** POO.
Ce paradigme repose sur la création de données, appelées objets, modélisant des concepts quelconques.
Exemples : un personnage de jeu (nom, force, vie...), une fenêtre graphique (dimension, titre, ...), une carte à jouer (couleur, hauteur), une liste ...

Un objet est caractérisé par ses attributs et ses méthodes.

Les **attributs** sont des variables qui caractérisent un objet, et les **méthodes** sont des fonctions qui peuvent s'appliquer spécifiquement à l'objet, par exemple pour agir sur ses attributs ou interagir avec d'autres objets du programme.

Voir plus spécifiquement le cours P1.II-Vocabulaire de la programmation objet.

Exemple : Python propose d'écrire des programmes en suivant le paradigme objet (en réalité, en Python tout est objet !). C++ ou Ocaml permettent aussi l'usage de ce paradigme de programmation.

On reconnaît un programme orienté objet avec la « notation pointée » qui désigne l'accès à un attribut ou l'application d'une méthode à un objet.

Exemples :

```
>>> l = [1, 2, 3]
>>> l.append(4) # application de la méthode append à l'objet list [1, 2, 3]
                # modifie l'objet en lui ajoutant un élément
>>> l.pop() # application de la méthode pop à l'objet list [1, 2, 3, 4]
            # modifie l'objet en lui retirant le dernier élément
4
```

On pourrait imaginer des objets représentant des concepts de géométrie (point, segment, cercle...) et quelques manipulations sur les attributs ou méthodes de ces objets :

```
>>> p1 = Point(2, 3)       # instanciation d'un objet Point
>>> p2 = Point(-1, 5)
>>> AB = Segment(p1, p2)   # instanciation d'un objet Segment
>>> AB.extremite1          # accès à l'attribut extremite1 de l'objet AB
(2, 3)
>>> AB.translater((2 ,4)) # application de la méthode translater à l'objet AB
>>> AB.extremite2
(1, 9)
>>> c1 = Cercle(p1, 3)     # instanciation d'un objet Cercle
>>> c1.surface()           # application de la méthode surface à l'objet c1
28.27
```

Remarque : le paradigme de programmation orientée objet repose aussi sur d'autres concepts comme l'**encapsulation**, l'**héritage** ou le **polymorphisme**, mais ces notions (pourtant très importantes) restent hors programme en Terminale.