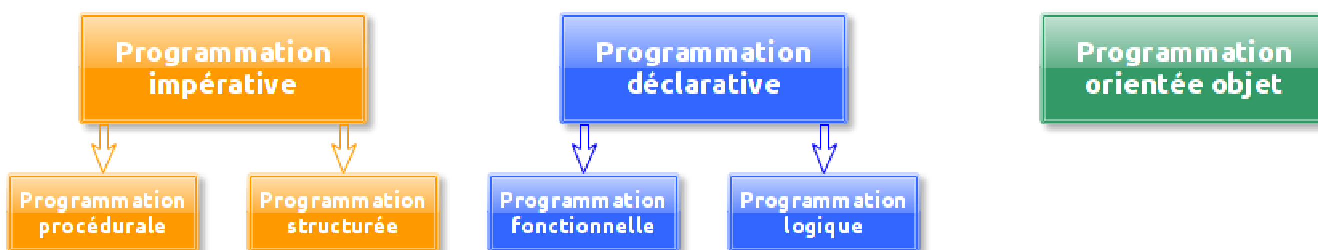
Chapitre 2

Paradigmes de Programmation

Les **paradigmes** en programmation désignent les styles de programmation fondamentaux divers générant des codes logiciels de structure différente. Il existe 3 grandes familles de paradigmes : la programmation **impérative**, la programmation **déclarative** et la programmation **orientée objet**.



Dans chaque paradigme, on trouve plusieurs dérivés comme présenté sur le schéma ci-dessus (liste non exhaustive).
Vous connaissez déjà la programmation impérative (vous l'utilisez constamment !) et avez découvert la programmation orientée objet cette année. Dans ce cours, nous allons nous intéresser aux principales caractéristiques de chacun de ces paradigmes, en nous focalisant notamment sur la programmation fonctionnelle.

2.1 Quelques rappels sur les variables

Variables locales, variables globales

Lorsqu'une variable est définie à l'intérieur d'une fonction, on parle de **variable locale**. Une telle variable n'est pas utilisable en dehors de la fonction où elle a été définie.
La variable locale s'oppose à la **variable globale**, utilisable dans tout le programme.

```python
L = [1,2,3]

def somme(liste): # Retourne la somme des éléments de liste
    S = 0
    for e in liste:
        S = S + e

    return S

sommeListe = somme(L)
```

Dans le programme ci-dessus, on dénombre :

- 2 variables globales : `L` et `sommeListe`
- 3 variables locales, définies dans la fonction `somme` : `liste` (copie de la variable passée en paramètre, ici `L`), `S` et `e`

Depuis l'extérieur de la fonction `somme`, on peut manipuler la variable `sommeListe`, mais il est impossible d'accéder à la variable `S`, qui n'existe pas en dehors de la fonction.

```
>>> sommeListe + 2
8
>>> S
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'S' is not defined
```

Effets de bord

En Python, les listes font partie des objets dits **mutables**, c'est à dire qu'ils sont modifiables de façon indirecte.
Prenons le programme suivant, et modifions les variables `b` et `M` depuis la console :

```python
a = 2
b = a

L = [1,2,3]
M = L
```

```
>>> b = 6
>>> b
6
>>> a
2
>>> M[1] = 10
>>> M
[1, 10, 3]
>>> L
[1, 10, 3]
```

On remarque que la modification du contenu de `M` a été appliquée à `L`.

La fonction `somme`, définie plus haut, prend une liste en argument. Lors de l'appel de `somme(L)`, la variable `L` passée en paramètre est copiée dans la variable locale `liste`, à la manière de l'instruction `M = L` précédente.
Bien que locale, la variable `liste` peut alors modifier la variable `L`, bien que ce soit une copie. Si une telle modification survient, on parle d'**effet de bord**. Plus généralement :

Définition (Wikipédia)**.** En informatique, une fonction est dite à **effet de bord** (traduction mot à mot de l'anglais *side effect*, dont le sens est plus proche d'effet secondaire) si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

Imaginons qu'un programmeur code la fonction `somme` de la façon suivante :

```
1 def somme(liste):
2     S = 0
3     while len(liste) > 0:
4         S = S + liste.pop()
5 
6     return S
```

Algorithmiquement parlant, la fonction est correcte : elle renvoie bien la somme de la liste. Mais en pratique, la liste passée en paramètre est maintenant vide ! En effet, la méthode `pop` modifie le contenu de la liste (elle retire le dernier élément et le retourne), on a donc un effet de bord.

```
>>> sommeListe
6
>>> L
[]
```

Les effets de bord sont donc à éviter autant que possible.
Dans l'exemple précédent, on force l'effet de bord, et personne ne code la fonction `somme` de cette manière. Mais parfois, notre programme présente un bug difficilement compréhensible, pouvant résulter d'un effet de bord peu visible.

Pour exclure tout effet de bord, on peut coder de manière particulière : c'est la **programmation fonctionnelle**, qui dans sa forme la plus « pure », ne permet pas la déclaration de variables ! Tout y est fonction.

Mais avant de parler de programmation fonctionnelle, revenons quelques instants sur les paradigmes que vous connaissez déjà : l'impératif et l'orienté objet.

2.2 Programmation impérative

Parmi les paradigmes de programmation, la programmation impérative est considérée comme la plus « traditionnelle ».

Les premiers langages de programmation et, de ce fait, les premiers programmes informatiques, ont été intégralement conçus sur cette base, qui énonce une séquence stricte d'ordres déterminés (du lat. imperare « ordonner »), ou instructions d'exécution. La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter une suite d'instructions élémentaires, codées sous forme d'*opcodes*. C'est pourquoi ce paradigme est à la base de tous les langages d'assemblage.

En programmation impérative, l'accent est notamment mis sur une collaboration aussi étroite que possible avec le système. Le **code** de programmation en résultant est alors **facile à comprendre**, mais très **volumineux**.

La plupart des langages de haut niveau comporte 5 instructions de base en programmation impérative :

- la séquence d'instructions (la liste des instructions à exécuter les unes à la suite des autres)
- l'assignation ou l'affectation de valeurs en mémoire
- l'instruction conditionnelle
- la boucle
- les branchements

En Python, les branchements correspondent aux appels de fonctions au cours du programme.
Certains langages implémentent une instruction *goto* qui permet de se déplacer à un certain endroit du programme.

On distingue trois approches majeures, les programmations **structurée**, **procédurale** et **modulaire**, qui sont en quelque sorte des « sous-familles » de programmation impérative.

Programmation Structurée

L'approche de **programmation structurée** est une forme simplifiée de programmation impérative. La différence déterminante par rapport au principe de base : au lieu d'instructions de saut absolu (instructions entraînant le traitement, non pas avec l'ordre suivant, mais à un autre endroit), ce paradigme de programmation prévoit l'**utilisation de boucles et structures de contrôle**. Un exemple en est l'utilisation de « do...while » pour l'exécution automatique d'une instruction dès lors qu'une certaine condition est remplie (au moins une fois).

Programmation Procédurale

Le paradigme de **programmation procédurale** étend l'approche impérative à la possibilité de subdiviser des algorithmes en plusieurs parties plus facilement maîtrisables. Celles-ci sont alors désignées par les termes procédures ou, selon le langage de programmation, **sous-programmes**, **routines** ou **fonctions**. L'objectif de cette subdivision est de faciliter la compréhension du code de programmation et d'éviter les répétitions de code inutiles. De par l'abstraction des algorithmes, le paradigme logiciel procédural s'impose comme une avancée déterminante entre les langages d'assemblage simples et les langages standard plus complexes.

Programmation Modulaire

La **programmation modulaire** est également comprise comme une sous-catégorie du paradigme de programmation impérative. Elle est fondamentalement très similaire à l'approche procédurale et applique ce style de programmation aux exigences de **projets plus volumineux** et plus complets. Le code source est ici subdivisé de manière ciblée en blocs logiques indépendants les uns des autres pour plus de clarté et pour faciliter le processus de débogage (dépannage). Les différents blocs, encore appelés **modules**, peuvent être testés séparément avant d'être liés au sein d'une application commune.

☞ *Python permet d'utiliser le paradigme impératif. Il dispose de boucles, de structures conditionnelles, on peut y définir des fonctions et séparer le code en modules pour plus de clarté*

Exemple. Supposons que l'on dispose d'une liste L dont on veut calculer la somme des carrés des éléments.
En programmation procédurale, on peut procéder de la manière suivante :

```
1  L = [8, 5, 7]
2  s = 0
3
4  for n in L:
5      s = s + n**2
6
7  print(s) # Affiche 138 = 8^2 + 5^2 + 7^2
```

2.3 Programmation Orientée Objet

La **programmation orientée objet** (POO), ou programmation par objet, consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

Exemple. Pour résoudre le problème de l'exemple précédent, on peut créer en Python une classe `ListeNombres` qui possède une méthode `sommeCarres()` :

```
1  class ListeNombres:
2      def __init__(self, liste):
3          self.__liste = liste
4
5      def sommeCarres(self):
6          s = 0
7          for n in self.__liste:
8              s = s + n**2
9
10         return s
11
12 liste = ListeNombres([8, 5, 7])
13 print(liste.sommeCarres)
```

2.4 Programmation fonctionnelle

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. On évite au maximum l'affectation de variables, afin de minimiser les effets de bord.

En programmation fonctionnelle, « on ne définit pas de variables, mais on définit des fonctions » !

Problème. *Comment calculer la somme des carrés des éléments d'une liste en fonctionnel ?*

Avant de traiter cet exemple, essayons de saisir la différence entre un programme impératif et un programme fonctionnel.
Considérons les programmes suivants, produisant un résultat identique :

```
a = 0

def increment():
    global a
    a = a + 1

increment()
print(a)
```

```
def increment(a):
    return a + 1

print(increment(0))
```

Le programme de **gauche** est un programme **impératif**, dans lequel on incrémente la valeur d'une variable `a` extérieure à la fonction. Cela est rendu possible par le mot-clé `global`, qui rend la variable `a` accessible depuis la fonction `increment`.

Le programme de **droite** est un programme **fonctionnel** : aucune variable n'est définie, la fonction `increment` se contente de retourner la valeur incrémentée passée en paramètre.

Autre exemple : on souhaite incrémenter toutes les valeurs d'une liste. Voici deux façons de faire :

```
# Incrément d'une liste, impératif
L = [1,2,3]

def increment_liste(liste):
    for i in range(len(liste)):
        liste[i] = liste[i] + 1

increment_liste(L)
print(L)
```

```
# Incrément d'une liste, fonctionnel
def increment(a):
    return a + 1

def increment_liste(liste):
    return list(map(increment, liste))

print(increment_liste([1,2,3]))
```

Côté impératif (gauche), il y a plusieurs affectations de variables : la liste `L`, l'élément d'indice i de la liste `liste[i]` ... mais aussi la variable `i` elle-même : sa valeur est re-affectée à chaque tour de boucle ! **Pas de boucle itérative en fonctionnel** !

À la place, on dispose de la fonction `map`, qui prend en arguments une fonction et une collection de données, généralement une liste ou un dictionnaire, et applique la fonction à chaque élément de la collection.
Ainsi, l'instruction `map(increment, liste)` retourne une nouvelle collection composée des éléments de `liste`, auxquels on a préalablement appliqué la fonction `increment`. On transforme alors cette collection en liste avec la fonction `list`.

☞ *La programmation fonctionnelle, c'est l'art de composer les fonctions*

Nous sommes maintenant capables d'écrire un programme qui retourne la somme des carrés d'une liste en fonctionnel :

```python
def carre(x):
    return x**2

def somme_carres(liste):
    return sum(list(map(carre, liste)))

print(somme_carres([8,5,7]))
```

On peut encore raccourcir le programme précédent en utilisant une fonction lambda :

```python
print(sum(list(map(lambda x : x**2, [8,5,7]))))
```

Dernier exemple : on souhaite écrire une fonction `valeurs_paires(liste)` qui retourne une liste composée uniquement des nombres pairs de la liste d'entiers `liste`.

```python
# Valeurs paires, impératif
L = [1,2,3,4]

def valeurs_paires(liste):
    liste_pairs = []

    for n in liste:
        if n % 2 == 0:
            liste_pairs.append(n)

    return liste_pairs

print(valeurs_paires(L))
```

```python
# Valeurs paires, fonctionnel

def valeurs_paires(liste):
    if len(liste) == 0:
        return []
    else:
        if liste[0] % 2 == 0:
            return [liste[0]] + valeurs_paires(liste[1:])
        else:
            return valeurs_paires(liste[1:])

print(valeurs_paires([1,2,3,4]))
```

Remarquez l'emploi de la **récursivité** en fonctionnel.

Dans ce dernier exemple, on « filtre » une liste en ne gardant que les termes pairs. On aurait pu utiliser la fonction `filter`, qui prend en entrée une fonction et une collection, et renvoie une nouvelle collection contenant tous les éléments de la collection d'origine pour laquelle la fonction renvoie une valeur vraie :

```python
def estPair(n):
    return n % 2 == 0

def valeurs_paires(liste):
    return list(filter(estPair, liste))

print(valeurs_paires([1,2,3,4]))
```

Pour la beauté du code, le même programme en « one-line » :

```python
print(list(filter(lambda n : n % 2 == 0, [1,2,3,4])))
```

2.5 Exercices

Exercice 01 Réecrire de façon fonctionnelle le programme suivant :

```
prenoms = ["Jean", "Luc", "Lucie", "Marie"]

for i in range(len(prenoms)):
    prenoms[i] = hash(prenoms[i])
```

☞ *La fonction `hash` renvoie un entier permettant d'indentifier de façon unique l'objet donné.*

Dans les exercices suivants, on programmera de façon impérative puis de façon fonctionnelle.

Exercice 02 Écrire un programme permettant de multiplier tous les nombres d'une liste par un nombre donné.

```
>>> multiplier_liste([1,2,3], 4)
[4,8,12]
```

Exercice 03 Écrire une fonction `carres(n)` qui retourne la liste des carrés des entiers de 0 à n inclus.

```
>>> carres(7)
[0,1,4,9,16,25,36,49]
```

Exercice 04 Écrire un programme permettant de calculer les sommes suivantes :

$$S_1 = 1 + 2 + 3 + \ldots + 2020 \qquad S_2 = 3^2 + 6^2 + \ldots + 999^2 \qquad S_3 = 1^1 + 2^2 + 3^3 + \ldots + 20^{20}$$

Exercice 05 Calculer la somme des entiers naturels multiples de 5 ou 6 inférieurs ou égaux à 100 000.

Exercice 06 Écrire une fonction `compteur(chaine, caractere)` qui retourne le nombre d'occurences de `caractere` dans `chaine`.

```
>>> compteur("J'adore la programmation fonctionnelle !", "o")
4
```

Exercice 07 Écrire une fonction `map_liste(fonction, liste)` qui reproduit le comportement de la fonction `map`.

Exercice 08 Réecrire les fonctions `sum`, `len` et `range`.

Exercice 09 Trouver tous les couples d'entiers naturels (x, y) vérifiant l'équation :

$$x^2 + 2y^2 = 1000$$