

# Paradigmes de programmation

## Cours

Un **paradigme de programmation** est une façon d'approcher la programmation et de traiter les solutions aux problèmes et leur formulation dans un style approprié. La plupart des langages sont **multiparadigmes**.

La suite de ce chapitre présente certains paradigmes parmi les plus utilisés<sup>1</sup>.

## Langages de programmation impérative

C'est le paradigme **le plus courant**, car la quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative. Il s'agit historiquement des premiers langages, même si de nombreux langages modernes utilisent toujours ce principe de fonctionnement.

## Cours

Dans un langage **impératif** le programme est construit sous forme d'une **suite d'instructions**, regroupées par blocs et comprenant des sauts conditionnels pour revenir à un bloc d'instructions si une condition est réalisée.

Exemples : la plupart des langages courants, y compris des langages de programmation orientés objets tels que C#, Visual Basic, C++ et Java, PHP.

## Langages de programmation fonctionnelle

## Cours

Dans un langage **fonctionnel** le programme est construit comme un emboîtement de **fonctions** qui agissent comme des « boîtes noires » que l'on peut imbriquer les unes dans les autres.

Exemples : Lisp, la famille ML (Standard ML, OCaml et autres) et Haskell, mais aussi beaucoup de langages impératifs (Python ou C#) incluent des extensions de langage qui prennent en charge la programmation fonctionnelle, par exemple les fonctions lambda en Python.

Les principes de la programmation fonctionnelle sont les suivants :

- Les **variables ne sont pas modifiées**.

Les variables Python de type `list` avec les méthodes `.append()` ou `.pop()` ne respectent pas les principes de programmation fonctionnelle, voyons comment les utiliser différemment :

Exemple de programmation <u>non</u> fonctionnelle	Exemple de programmation fonctionnelle
<pre>&gt;&gt;&gt; l = [1, 2, 3, 4] &gt;&gt;&gt; l.append(5) &gt;&gt;&gt; l [1, 2, 3, 4, 5] &gt;&gt;&gt; l.pop() 5</pre>	<pre>&gt;&gt;&gt; l = [1, 2, 3, 4] &gt;&gt;&gt; m = l + [5] &gt;&gt;&gt; m [1, 2, 3, 4, 5] &gt;&gt;&gt; l [1, 2, 3, 4] &gt;&gt;&gt; n = m[:-1]</pre>

- Les **tableaux par compréhension**, inspirées du langage de programmation fonctionnel Haskell, sont adaptés à la programmation fonctionnelle.

Par exemple la liste de carrés de tous les nombres impairs de 0 à 9 :

Exemple de programmation <u>non</u> fonctionnelle	Exemple de programmation fonctionnelle
<pre>carres = [] for x in range(10):     if x%2 != 0:         carres.append(x**2)</pre>	<pre>[x**2 for x in range(10) if x%2 != 0]</pre>

- La **récurtivité** est souvent utilisée, en particulier à la place des boucles (puisque l'état du programme ne peut pas être modifié, en particulier avec un variant de boucle).

Voyons comment calculer le produit de tous les nombres entiers entre 1 et  $n$ , appelé factorielle de  $n$  et noté  $n!$  :

$$n! = 1 \times 2 \times 3 \times 4 \times \dots \times (n - 1) \times n$$

Exemple de programmation <u>non</u> fonctionnelle	Exemple de programmation fonctionnelle
<pre>def fact(n):     f = 1     for i in range(2, n+1):         f = f * i     return f</pre>	<pre>def fact(n):     if n == 1: return 1     return n * f(n - 1)  # ou en fonction lambda fact = lambda n: 1 if n == 1 else n * f(n - 1)</pre>

- Une fonction est une "**fonction pure**", elle renvoie une valeur qui ne dépend que de ses paramètres, et pas de valeur externes.

Exemple de programmation <u>non</u> fonctionnelle	Exemple de programmation fonctionnelle
<pre>n = 2 def inc(k):     """ incrémentation par effet de bord """     global n     n = n + k</pre>	<pre>def inc(k):     """ incrémentation sans effet de bord """     n = 2     n = n + k     return n</pre>

**Exemple de programmation non fonctionnelle**

```

return n

print(inc(1) + inc(1))
>>> 7

```

**Exemple de programmation fonctionnelle**

```

print(inc(1) + inc(1))
>>> 6

```

Dans l'exemple de programmation non fonctionnelle, la fonction `inc(k)` ne renvoie pas la même valeur lors des deux appels : le premier appel renvoie `3` (`=2 + 1`) et le second `4` (`=3 + 1`), il est donc impossible de remplacer `print(inc(1) + inc(1))` par `print(2 * inc(1))`.

- Une fonction est dite "**d'ordre supérieur**", elle peut être utilisée comme paramètre d'une autre fonction ou renvoyée comme résultat d'une autre fonction. Une fonction est aussi dite "**de première classe**", elle est manipulable comme un type de base, elle peut être assignée à une variable (fonction lambda) ou même stockée dans une structure de données.

En Python, les fonctions lambda sont un exemple de programmation fonctionnelle. Créons une fonction qui renvoie la fonction "à la puissance n" :

```

def puissance(n):
    return lambda x: x**n

```

que l'on peut l'utiliser pour créer les fonctions `carre` et `cube` :

```

>>> carre = puissance(2)
>>> cube = puissance(3)
>>> cube(10)
1000

```

`puissance(5)` est la fonction qui renvoie la puissance 5 d'un nombre, par exemple pour calculer  $10^5$  :

```

>>> puissance(5)(10)
10000

```

On peut aussi stocker des fonctions dans une structure de données, par exemple créer un tableau avec toutes les fonctions `puissance()` entre de 0 et 100 :

```

>>> p = [puissance(i) for i in range(101)]
>>> p[5](10)
10000

```

## Langages de programmation orientée objet

### Cours

La **programmation orientée objet**, ou **POO** 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 (une voiture, une personne, etc.).

Exemple de langages orientés objets : Java, Javascript C++, Python, PHP

Une classe regroupe des fonctions et des attributs qui définissent la structure interne et le comportement de ses objets.



#### PEP 8

Les noms de classes s'écrivent en CamelCase <https://www.python.org/dev/peps/pep-0008/#class-names>

Prenons un exemple, créons une classe `Voiture` :

```
class Voiture:
    pass
```

Notre classe `Voiture` est une sorte d'usine à créer des voitures, ce n'est pas une voiture. Elle permet de créer un ou plusieurs objets, appelés **instances** de la classe.

Créons maintenant notre premier objet `Voiture` :

```
ma_voiture = Voiture()
```

Les attributs permettent de stocker des informations à propos d'un objet `Voiture`. Dans notre exemple, donnons plusieurs attributs à une voiture :

```
ma_voiture = Voiture()
ma_voiture.marque="Peugeot"
ma_voiture.modele = "308"
ma_voiture.km = 25645
```

Et les lire ainsi. L'instruction `print("Ma voiture est une ",ma_voiture.marque, ma_voiture.modele, 'de ', ma_voiture.km, 'km')` affiche dans la console :

```
>>> Ma voiture est une  Peugeot 308 de 25645 km
```

Les **méthodes** sont des fonctions qui s'appliquent aux objets de cette classe. Créons deux nouvelles méthodes dans notre classe `Voiture` :

```
class Voiture:

    def get_km(self):
        return self.km

    def roule(self, k):
        self.km = self.km + k
```

Utilisons ces méthode dans un programme :

```
def main():
    ma_voiture = Voiture()
    ma_voiture.marque="Peugeot"
    ma_voiture.modele = "308"
    ma_voiture.km = 25645
```

```
ma_voiture.roule(10000)
print(ma_voiture.get_km() )
```

Plutôt que de rajouter les attributs les uns après les autres après avoir créé un objet, on peut ajouter une méthode spéciale appelée `__init__()`<sup>2</sup> pour les renseigner directement à la création de l'objet.

```
class Voiture:
    def __init__(self, ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self.km = k

def main():
    ma_voiture = Voiture("Peugeot", "308", 25645)
    ma_voiture.roule(10000)
    print(ma_voiture.get_km() )
```

## Langages de programmation logique

### Cours

La **programmation logique** est un paradigme de programmation qui définit les applications à l'aide d'un ensemble de **faits élémentaires** les concernant et de **règles de logique** leur associant des conséquences plus ou moins directes. Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférence, en réaction à une question ou requête.

Exemple : Prolog<sup>3</sup> (Prolog est aussi un langage impératif).


Prenons l'exemple du problème logique suivant : Alice et Luc sont mariés. Luc est le père de Jean. Qui est la mère de Jean ?

### Syntaxe Prolog

Le symbole `:-` dans une règle se lit "si" ; la virgule `,` dans une règle se lit "et", tout ce qui suit le caractère `%` sur une ligne est un commentaire, le nom des variables commencent toujours par une lettre majuscule (exemple : `X`, `Voiture`, `NUMERO`, etc.).

La solution peut être trouvée par un simple programme Prolog réalisé sur <https://swish.swi-prolog.org/>.

Indiquons les deux faits connus dans le code source: 1. alice est l'épouse de luc 2. luc est le père de jean

 Ne pas oublier le point à la fin de chaque instruction.

```
% les faits :
epouse(alice,luc). % alice est l'épouse de luc
pere(luc,jean). % luc est le père de jean
```

Puis la règle que si un père est marié à une femme, alors cette dernière est la mère du fils (pas de famille recomposée ici !) :

% les règles :

```
mere(M,E):-pere(P,E),epouse(M,P). % M est la mère de E si P est le père de E et si M est l'épouse de P
```

Avant de répondre à notre problème, posons quelques questions :

j'utiliser le bouton Run ou CTRL+Enter pour executer.

1. Qui est l'épouse de luc ?

```
?- epouse(X, luc).
X = alice.
```

2. Qui est l'époux d'alice ? (ou qui a pour épouse alice ?)

```
?- epouse(alice,X).
X = luc.
```

3. Qui est le père de jean ?

```
?- pere(X, jean).
X = luc.
```

4. Qui est le fils de luc ? (ou qui a pour père luc ?)

```
?- pere(luc,X).
X = jean.
```

Noter que pour répondre à ces 4 premières questions Prolog n'utilise que les faits, pas la règle définissant la mère.

Posons maintenant la question qui nous intéresse. Prolog utilise cette fois la règle définissant la mère.

1. Qui est la mere de jean ?

```
?- mere(X, jean).
X = alice.
```

2. Qui est le fils d'alice ? (ou qui a pour mère alice ?)

```
?- mere(alice,X).
X = jean.
```

3. Qui est le fils de jean ? (ou Qui a pour père jean ?)

```
?- pere(jean,X).
false.
```

4. Qui est le fils de Lucienne ? (ou qui a pour mère Lucienne ?)

```
?- mere(lucienne,X).
false.
```

Prolog n'a pas la réponse à ces 2 dernières questions : il répond alors false.

1. Qui est la mère de qui ?

```
?- mere(X,Y).
X = alice,
Y = jean..
```

2. Combien de liens de parenté "mère/fils" existe-t-il ?

```
?- mere(_,_).
true    % 1 seul.
```

Le soulignement (   ) signale à Prolog que la variable en question n'est pas utilisée ("placeholder")

3. Existe-t-il une personne qui est sa propre mère ?

```
?- mere(X,X).
false   % non.
```

## Langages de programmation événementielle

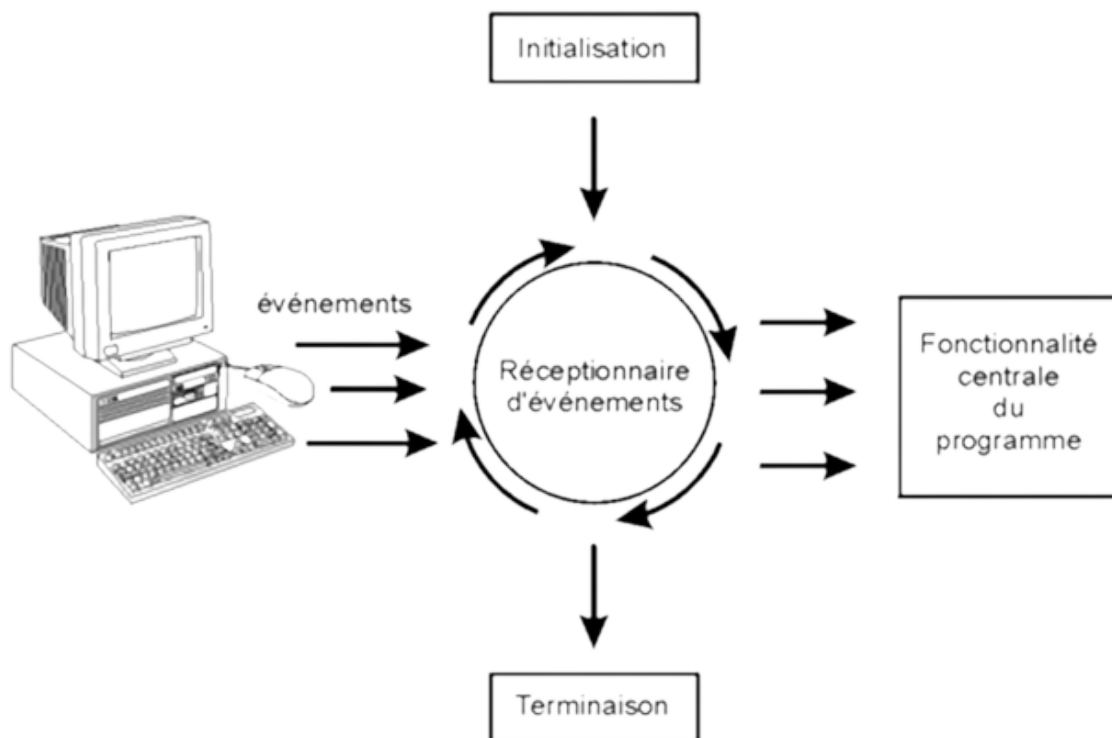
### Cours

En **programmation événementielle** le flux du programme est déterminé par des événements tels que les actions de l'utilisateur (clics de souris, pressions de touches), les sorties de capteur ou les messages d'autres programmes ou threads.

Exemple : JavaScript

La programmation événementielle est le paradigme dominant utilisé dans la programmation d'Interfaces Homme Machine, ou **IHM** (**GUI** en anglais pour *graphical user interface*) et d'autres applications d'automates.

Après une phase d'initialisation, le programme « tourne » en permanence « en attente » de détecter un **événement** : une action de l'utilisateur (déplacement de la souris, appui sur une touche, etc.) ou un événement externe (top d'horloge, etc.)



Le programme réagit aux événements, l'ordre dans lequel il est exécuté n'est pas connu à l'avance, ce sont les événements qui le déterminent.

En pratique :

1. Javascript avec `<element onclick="myFunction">` .

Dans notepad, créons un fichier test.html et ouvrons-le dans un navigateur

### Syntaxe Prolog

On peut accéder à un élément HTML en JavaScript avec `document.getElementById(id)` où `id` désigne l'élément HTML. L'attribut `innerHTML` définit le contenu de cet élément HTML.

```

<!DOCTYPE html>
<html>
<body>

<p id="demo" onclick="myFunction()">Cliquez ici</p>

<script>
function myFunction() {
document.getElementById("demo").innerHTML = "Vous avez cliqué"
}
</script>
</body>
</html>
  
```

2. Javascript avec `object.onclick = function(){myScript};`

```

<!DOCTYPE html>
<html>
<body>
<p id="demo"> Cliquez ici</p>
  
```



```
<script>
document.getElementById("demo").onclick = function() {myFunction()};

function myFunction() {
document.getElementById("demo").innerHTML = "Vous avez cliqué"
}
</script>
</body>
</html>
```

## Langages de requêtes ou de bases de données

### Cours

Le langage SQL (Structured Query Languages) spécialement conçu pour faire des requêtes (sélectionner, filtrer, mettre à jour) sur les systèmes de bases de données.

## Langages descriptifs (ou de balisage)

### Cours

Les langages **descriptifs** (ou de balisage) sont spécialisés dans l'enrichissement d'information textuelle. Ils utilisent des balises, permettant de transférer à la fois la structure du document et son contenu.

Cette structure est compréhensible par un programme informatique, ce qui permet un traitement automatisé du contenu. Si **ce sont des langages informatiques, ce ne sont pas des langages de programmation** à proprement parler (on ne peut pas écrire de programme avec).

Exemples: HTML, mais aussi Latex, XML

1. Une liste complète de paradigmes de programmation est disponible sur : [https://en.wikipedia.org/wiki/programming\\_paradigm](https://en.wikipedia.org/wiki/programming_paradigm). ←
2. La méthode `__init__()` est appelée le constructeur de la classe. ←
3. Le nom Prolog est un acronyme de PROgrammation en LOGique. Il a été créé par Alain Colmerauer et Philippe Roussel vers 1972 à Luminy, Marseille. Le but était de créer un langage de programmation où seraient définies les règles logiques attendues d'une solution et de laisser le compilateur la transformer en séquence d'instructions. (Source : Wikipedia) ←