Paradigmes de programmation

1. Définition

Paradigmes : représentation du monde ; manière de voir les choses ; modèle cohérent de pensée, de vision du monde qui repose sur une base définie, sur un système de valeurs.

Un **paradigme de programmation** est une façon d'approcher la programmation informatique et de traiter les solutions aux problèmes et leur formulation dans un langage de programmation approprié.

Un **paradigme de programmation** fournit la vue qu'a le développeur de l'exécution de son programme. Par exemple, en **programmation orientée objet**, les développeurs peuvent considérer le programme comme une **collection d'objets en interaction**, tandis qu'en **programmation fonctionnelle** un programme peut être vu comme une **suite d'évaluations de fonctions sans états**. Lors de la programmation d'ordinateurs ou de systèmes multiprocesseurs, la **programmation orientée processus** permet aux développeurs de voir les applications comme des **ensembles de processus agissant sur les structures de données localement partagées**.

Le paradigme a une influence forte sur la manière de concevoir un programme : il constitue un des critères principaux pour le choix d'un langage.

2. Les trois principaux paradigmes

Il existe trois grands types de programmation :

- La programmation **impérative** : le paradigme originel est le plus courant.
- La programmation **orientée objet** (POO) consistant en la définition et l'assemblage de briques logicielles appelées objets.
- La programmation déclarative consistant à déclarer les données du problème, puis à demander au programme de le résoudre. La programmation **fonctionnelle** en est un exemple représentatif dans lequel le résultat souhaité est déclaré comme la valeur d'une série d'applications de fonctions.

Même si certains langages forcent à utiliser un paradigme de programmation (Smalltalk : POO, Haskell : fonctionnelle), de nombreux langages modernes comme Python (ou Javascript) sont multiparadigmes et permettent la programmation impérative structurée, fonctionnelle et orientée objet ; laissant ainsi le choix au programmeur du paradigme à utiliser en fonction du problème posé.

2.1. Exemples d'approches en Python

Pour illustrer les différences entre ces paradigmes, nous allons utiliser un exemple très simple.
On a une liste de caractères que nous souhaitons concaténer en une chaine de caractères.

```
entrée = ['p','y','t','h','o','n']
sortie = 'python'
```

2.2. Programmation impérative : le comment

La programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.
La programmation impérative se concentre sur la description du fonctionnement d'un programme : **le comment**.
La plupart des langages de haut niveau comporte cinq types d'instructions principales :

- la séquence d'instructions ;
- l'assignation ou affectation ;
- l'instruction conditionnelle (if, else) ;
- la boucle (for, while).

```
entrée = ['p','y','t','h','o','n']
# on initialise une chaine vide pour la sortie
sortie = ""
# On concatène à l'aide d'une boucle
for c in entrée:
     sortie = sortie + c

print(sortie)
```

Ce type de programmation est le plus ancien et utilisé, il est facile à comprendre, souvent efficace car proche des instructions réalisées par les processeurs. Par contre, il est assez difficile à tester car l'état du programme ne cesse de changer et il est difficile de tester une petite partie du programme au milieu de son exécution par exemple car elle ne nécessite que toutes les instructions précédentes aient déjà été appliquées correctement.

2.3. Programmation fonctionnelle : le quoi

En programmation fonctionnelle **on décrit les résultats que l'on veut obtenir à partir des données** plutôt que la séquence d'instructions qui permettent d'obtenir les résultats (c'est un paradigme déclaratif).
L'approche fonctionnelle considère le calcul en tant qu'évaluation de **fonctions** mathématiques. Vous donnez vos données en entrée aux fonctions, qui vous renvoient les valeurs calculées en sortie.

L'utilisation massive de fonctions a amené à la création d'une syntaxe raccourcie pour la définition de fonctions anonymes, les fonctions **lambda** :

```
lambda param1, ... , paramN: valeur_retournée
```

Au lieu de :

```
def ma_fonction(param1, ... , paramN):
    ...
    return valeur_retournée
```

En programmation fonctionnelle, **il n'y a pas d'état, l'opération d'affectation est interdite**, ce qui permet de s'affranchir des effets secondaires (ou **effets de bord**).
Ceci rend les programmes parfaitement prédictibles, faciles à tester et à paralléliser, par contre il est souvent compliqué de se débarrasser complétement de l'état du programme.

En programmation fonctionnelle, on remplace souvent les boucles par des **fonctions récursives**. Une approche fonctionnelle par la récursion de notre problème pourrait être :

```
entrée = ['p','y','t','h','o','n']

def list_to_string(ma_liste, ma_chaine=""):
    """Fonction récursive pour concaténer les éléments d'une liste"""
    if ma_liste:
        # on enlève le premier élement de la liste qu'on ajoute à la chaine de caractères
        ma_chaine += ma_liste.pop(0)
        # application récursive
        return list_to_string(ma_liste, ma_chaine)
    else:
        # cas de base
        return ma_chaine

list_to_string(entrée)
```

2.4. Programmation objet : POO

La 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, comme une voiture, une personne ou encore une page d'un livre.
Un objet possède :

- des données: ses **attributs** ;
- des fonctions: ses **méthodes**.

```
class ListeLettres:
    "Classe permettant de lier une chaîne de caractères à une liste de caractères"
    def __init__(self, lettres=[]):
        """Intialisation de l'objet
        """
        # initialisation des attributs de l'objet
        self.lettres = lettres
        # Conversion en chaine de caractères
        self.string = ''.join(lettres)

    # définition d'une méthode
    def get_string(self):
        return self.string

entrée = ['p','y','t','h','o','n']

# instanciation de l'objet avec les données de la liste
objet_py = ListeLettres(entrée)

# récupération de l'attribut string grâce à la méthode get_string
objet_py.get_string() # renvoie 'python'
```

Les différents principes de la conception orientée objet aident à la réutilisation du code, au masquage des données…

2.5. Comment choisir le paradigme à utiliser ?

Pour simplifier, si le problème implique une série de manipulations séquentielles simples, suivre le paradigme de programmation **impérative** est le moins cher en termes de temps et d'efforts et donne potentiellement les meilleures performances.

Dans le cas de problèmes nécessitant des transformations mathématiques des valeurs, le filtrage des informations, le mappage (transformer une liste en une autre) et les réductions (transformer une liste en une valeur), la programmation fonctionnelle est adaptée.

Si le problème est structuré comme un tas d'objets interdépendants avec certains attributs qui peuvent changer avec le temps, en fonction de certaines conditions, la programmation orientée objet est la plus adaptée

Bien sûr, il n'y a pas de règle simple, car le choix du paradigme de programmation dépend également fortement du type de données à traiter, des connaissances des programmeurs et de diverses autres choses comme l'évolutivité.

3. Les autres

3.1. Programmation évènementielle

La programmation événementielle est un paradigme de programmation dans lequel l'exécution d'actions est déclenchée automatiquement lorsqu'un **évènement** survient. Un **évènement** correspond en général à un changement d'état dans l'univers, ou bien à une intervention explicite de l'utilisateur (ou d'un système externe). On peut noter ces associations (événement, action) sous la forme : **évènement → action**.
La programmation événementielle est souvent utilisée dans les cas suivants :

- La programmation d'automates (systèmes de régulation par exemple).
  - Par exemple : température < 20 → déclencher chauffage.
- La programmation d'interface graphique. En effet, chaque action de l'utilisateur (clic souris...) peut être vu comme un événement.
  - Elle est utilisée dans la bibliothèque TKinter, le langage Javascript ou avec l'IDE Processing (détection touche clavier ou clic souris).

3.2. Programmation parallèle

Pour les programmes nécessitant beaucoup de calculs (simulation par exemple), il est rapidement devenu nécessaire de lancer plusieurs tâches en même temps. Pour cela, il a fallu développer des machines faites de nombreux processeurs. Au départ, cela ne concernait que les supercalculateurs, mais avec le développement des processeurs multi-core, il est devenu possible d'exécuter plusieurs tâches en même temps sur un ordinateur personnel.
Cependant, l'intérêt d'effectuer plusieurs tâches en même temps n'est pas lié qu'à cela. Par exemple, l'humain étant particulièrement long à taper sur un clavier, un logiciel de traitement de texte passe 99% de temps à ne rien faire. Il est donc intéressant, pendant ses longues périodes d'inactivité, de laisser d'autres logiciels s'exécuter. Mais si l'un de ces logiciels est un programme de calcul intensif, le système ne va pas attendre la fin de son exécution avant de repasser la main au traitement de texte. Le système va ainsi partager le temps du micro-processeur entre les différents logiciels, ou plus exactement entre les différents **processus**.
Utiliser plusieurs processus peut également servir au sein d'une même application. Par exemple, si votre application graphique doit lancer un calcul qui prend plusieurs secondes, il n'est pas concevable, que cela bloque l'application. Aussi, on peut être amené, au sein d'une même application, à créer plusieurs **threads**, qui sont des sortes de processus légers.
En ce qui concerne l'exécution de tâches en parallèle, certains langages, comme ADA, comportent des instructions spéciales pour décrire ce genre de traitement, mais ces langages sont peu nombreux. Dans la plupart des langages, c'est notamment le cas de Python, il existe des bibliothèques qui permettent de créer et lancer des threads.

Un exemple de programme multi-thread en Python

Voici un programme Python constitué de 3 threads : le thread principal, et 2 threads, créés grâce au constructeur **`Thread()`**, qui prend en paramètre la fonction qui devra être exécutée par le thread en question, et lancés par l'appel à la méthode **`start()`** :

```
from threading import Thread
from time import sleep
from random import randint

def traitementA():
    i = 0
    while i < 100:
        print("A"+str(i))
        sleep(randint(1,10)/10)
        i += 1
```

```
def traitementB():
    i = 0
    while i < 100:
        print("B"+str(i))
        sleep(randint(1,10)/10)
        i += 1

t1 = Thread(target=traitementA)
t2 = Thread(target=traitementB)

t1.start()
t2.start()

for i in range(20):
    print("Thread principal")
    sleep(1)
print("Thread principal terminé")
```

À l'exécution, on peut s'apercevoir que les exécutions des 3 threads sont entrelacées, et que l'arrêt du thread principal n'empêche pas les autres threads de continuer à s'exécuter.

3.3. Programmation logique

La programmation logique est un paradigme de programmation déclaratif très différent de tout ce que vous avez pu voir jusqu'à présent. Il repose essentiellement sur la logique des prédicats (ou logique du premier ordre). Il y a peu de langages mettant en œuvre la programmation logique. Le principal est Prolog.
Prolog est un langage créé en 1972 par un français : Alain Colmerauer. C'est un langage principalement utilisé dans le domaine de l'intelligence artificielle, mais aussi dans le traitement du langage.

Source : https://lyceum.fr/tg/nsi/4-langages-et-programmation/5-paradigmes-de-programmation/#les-trois-grands-types-de-programmation
https://mermet.users.greyc.fr/Enseignement/EnseignementInformatiqueLycee/Havre/Bloc4/progLogique.html