Paradigmes de programmation (Cours)

S1 - Langages et programmation

Introduction et définition

Tout d'abord, nous pouvons nous demander ce que signifie le mot **paradigme**. Parmi les trois définitions fournies par le dictionnaire Le Robert, celle qui nous intéresse est la suivante :

> **Définition**
>
> **Paradigme** Modèle de pensée.

En programmation, un paradigme est donc une manière de penser un programme, une méthode de programmation.

Un programme est un texte, avec ses conventions d'écriture. Il s'agit bien d'un langage écrit, au sens commun, mais il doit toujours avoir un sens **univoque** et non contextuel.

Il faut que la formulation textuelle d'un programme soit :

- suffisamment proche d'un code réel, conforme à une famille d'ordinateurs particuliers ;
- standardisée et générale pour permettre une adaptation immédiate et automatique — on parle de « portabilité » — à d'autres contextes similaires ;
- parfaitement univoque, non ambiguë, puisque destinée à un traitement automatique ;
- intelligible par un être humain.

Vu le grand nombre de langages existants, une classification s'est fait jour. On regroupe en général les langages en « familles » selon le paradigme de programmation auquel ils sont (le mieux) adaptés.

Conformément au programme, nous allons définir les paradigmes **impératif**, **fonctionnel** et **objet**.

Notons tout d'abord que la plupart des langages de programmation modernes sont **multiparadigmes** : ils permettent de programmer aussi bien de façon impérative, fonctionnelle qu'avec des objets.

Paradigme impératif

La programmation **impérative** est la méthode de programmation que vous avez le plus couramment utilisée jusqu'à présent.

Il s'agit d'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.

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)
- les branchements.

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 nécessite que toutes les instructions précédentes aient déjà été appliquées correctement.

Les langages C, C++, Java, JavaScript, Python et beaucoup d'autres permettent la programmation impérative.

Paradigme fonctionnel

Introduction

Le paradigme fonctionnel est un paradigme de programmation qui reprend les principes du lambda-calcul introduit par Alonzo Church dans les années 1930.

L'idée fondamentale du lambda-calcul est de considérer que les fonctions sont des données comme les autres. Ainsi, elles peuvent être par exemple passées en paramètre à d'autres fonctions.

D'autres principes découlent également de la thèse de Church :

- les fonctions sont des fonctions au sens mathématique du terme : elles se contentent de renvoyer une valeur en fonction de leurs arguments ;
- il n'y a pas de notion « d'état », ni à l'extérieur des fonctions, ni dans les fonctions. Un programme n'est donc qu'une composition de fonctions.

Le paradigme fonctionnel a d'abord été implanté au sein de langages dédiés, plus ou moins « purement fonctionnel ». Parmi les langages dits fonctionnels, on peut citer :

- LISP (List Processing) : 1958 ;
- SML (Standard Meta Language) : 1983 ;
- CAML (Categorical Abstract Machine Language) : 1987, puis son extension objet OCAML ;
- Haskell : 1990 ;
- Clojure : 2007.

Mais certains aspects du paradigme fonctionnel ont fini par être intégrés dans des langages impératifs, car ils présentent certains avantages :

- fonctions pures ;
- fonctions d'ordre supérieur ;
- lambda-expressions ;
- évaluation paresseuse.

En programmation fonctionnelle, les variables sont toujours constantes : une fois qu'elles ont été affectées, leur valeur ne doit plus changer ; de plus les boucles sont remplacées par des appels récursifs.

Mise en œuvre en Python

Fonctions pures

Une fonction pure est une fonction qui ne modifie rien ; elle ne fait que renvoyer des valeurs en fonction de ses paramètres. Et les valeurs renvoyées ne doivent dépendre que de ses paramètres, et pas de variables extérieures à la fonction.

Les modifications qu'une fonction peut effectuer sur l'état du système sont appelées **effets de bord**. Un affichage à l'écran est un exemple d'effet de bord.

En Python, rien n'impose d'implémenter des fonctions pures. Notamment, étant donné la façon dont les arguments sont passés à une fonction en Python (utilisation d'une copie de la référence initiale), rien n'interdit qu'une fonction modifie l'objet référencé par l'un de ses paramètres.

Voici un tel exemple :

```
def retirer_dernier(liste) :
    liste.pop()
```

On utilise cette fonction ainsi :

```
>>> ma_liste = [1, 2, 3]
>>> retirer_dernier(ma_liste)
```

L'inconvénient de ce type de fonction est qu'elle modifie la variable `ma_liste` qui, à l'issue de l'exécution des deux lignes précédentes contient `[1, 2]`. Cela peut rendre le code plus difficile à comprendre et générer des comportements inattendus. Une fonction pure, au contraire, doit renvoyer la valeur calculée sans modifier ses paramètres. Ainsi, on peut réécrire le traitement précédent de la façon suivante :

```
def retirer_dernier_pure(liste) :
    retour = liste[:]
    retour.pop()
    return retour
```

Cette fonction s'utilise ainsi :

```
>>> l1 = [1, 2, 3]
>>> l2 = retirer_dernier_pure(l1)
```

Dans ce dernier cas, le fait que l'appel à `retirer_dernier_pure` ne modifie par `l1` est bien plus intuitif.

> ! À retenir
>
> Pour faciliter l'écriture de fonctions pures en Python, on peut :
>
> - utiliser au maximum des données non mutables (tuples plutôt que listes par exemple) ;
> - copier systématiquement au début des fonctions les paramètres référençant des données mutables et utiliser ces copies dans la fonction.
> - on veille à ne pas modifier de valeur existante, mais plutôt à créer une nouvelle valeur à partir de la valeur existante.

Essayer de n'écrire que des fonctions pures permet de limiter les risques de bugs et facilite la relecture des programmes. Il s'agit donc d'un style de programmation à privilégier.

Fonctions d'ordre supérieur

Les fonctions étant considérées comme des données comme les autres, il est possible de définir des fonctions dont les arguments sont d'autres fonctions. On parle alors de **fonctions d'ordre supérieur**.

Python fournit des fonctions d'ordre supérieur dans sa bibliothèque standard. Voyons par exemple la fonction `map` qui permet d'appliquer une fonction à tous les éléments d'une liste. Quelques remarques et explications s'imposent :

- `map` peut s'appliquer à tout objet *itérable*, donc aux chaînes de caractères, aux tuples, aux listes.
- `map` retourne un objet itérable : les valeurs résultat ne sont pas toutes calculées par avance, elles le seront à la demande. Cet itérable peut être transformé en liste en tapant `list(map(...))` ou être utilisé dans une boucle `for item in map(...):`. Ce calcul des valeurs à la demande est une mise en œuvre du principe de l'**évaluation paresseuse** caractéristique de la programmation fonctionnelle.

Considérons le programme suivant :

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


def capit(ch):
    return ch.capitalize()


ma_str = "azerty"
mon_tuple = (1, 2, 3, 4, 5)
ma_liste = [1, 2, 3, 4, 5]

iter1 = map(capit, ma_str)
iter2 = map(carre, mon_tuple)
iter3 = map(carre, ma_liste)

for car in iter1:
    print(car, end="")
print()
print(tuple(iter2))
print(list(iter3))
```

On obtient en sortie :

```
AZERTY
(1, 4, 9, 16, 25)
[1, 4, 9, 16, 25]
```

> **Remarque**
>
> Le programme ci-dessus est donné pour illustrer l'idée de fonction d'ordre supérieur, mais il n'est pas rédigé, notamment sa partie itérative, dans l'esprit de la programmation fonctionnelle !

Les langages OCamL, Haskell, F#, Rust par exemple sont des langages fonctionnels.

Paradigme objet

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** et
- des fonctions: ses **méthodes**

Les différents principes de la conception orientée objet aident à la réutilisation du code, au masquage des données, etc. Les bases de la POO sont détaillées dans le cours précédent, avec ses compléments.

À quel paradigme se vouer ?

Comment choisir entre les différents paradigmes existants ?

Il est important de bien comprendre qu'un programmeur doit maitriser plusieurs paradigmes de programmation (impératif, objet ou encore fonctionnelle). En effet, il sera plus facile d'utiliser le paradigme objet dans certains cas alors que dans d'autres situations, l'utilisation du paradigme fonctionnel sera préférable. Être capable de choisir le "bon" paradigme en fonction des situations fait partie du bagage de tout bon programmeur.

Il est aussi important de bien comprendre que la frontière entre ces différents paradigmes est parfois floue, par exemple on utilise très souvent de l'impératif en programmation orientée objet.

Dans l'article Perceiving Python programming paradigms du site opensource.com/, les conseils suivants sont donnés :

- Pour simplifier, si votre problème implique une série de manipulations séquentielles simples, suivre le paradigme de programmation impérative de la vieille école serait le moins cher en termes de temps et d'efforts et vous donnerait 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 pourrait être 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 sera certainement la plus naturelle.

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.

Notons pour finir que cette courte présentation ne recouvre pas tous les paradigmes de programmation existants. On rencontrera notamment l'idée de **programmation événementielle** lors du développement d'interfaces graphiques.