Paradigmes de programmation

Introduction :

Dans ce cours nous nous intéressons aux styles, ou paradigmes de programmation, qui sont différentes manières de penser les problèmes et d'écrire des programmes informatiques pour les résoudre. Certains langages imposent l'emploi d'un paradigme particulier, mais le langage Python offre une certaine souplesse, autorisant l'emploi de plusieurs d'entre eux. Nous pourrons donc employer chaque fois du code Python pour illustrer les différents paradigmes étudiés dans le cadre de ce cours.

Nous étudierons trois paradigmes de programmation très répandus : le paradigme impératif, le paradigme fonctionnel et le paradigme objet, dont nous présenterons les principales caractéristiques pour les mettre ensuite en perspective.
Un même exemple de base servira à l'introduction de ces différents paradigmes : l'addition d'une liste de nombres.

1 Programmation impérative

a. Définir la programmation impérative

Le **paradigme impératif** indique précisément **comment une tâche doit être menée à bien**. Il définit étape par étape la manière d'obtenir le résultat attendu. C'est le paradigme le plus répandu et le plus ancien. Il fait appel à la notion d'état que nous avons découverte avec les machines de Turing lors du cours précédent.

Définition

Programmation impérative :

> La programmation impérative est basée sur une séquence explicite d'instructions produisant des changements d'états.

Découvrons la programmation impérative avec un exemple concret.

Astuce

> La liste de nombres utilisée en exemple sera toujours la même.
> `nombres = [2, 9, 12, 3, 5, 11]`

Exemple

En décomposant étape par étape, on définit de manière explicite la façon dont on calcule la somme des nombres de la liste.

```
somme = 0
somme = somme + nombres[0]
somme = somme + nombres[1]
somme = somme + nombres[2]
somme = somme + nombres[3]
somme = somme + nombres[4]
somme = somme + nombres[5]

print(somme)
# affiche 42
```

Astuce

> En Python l'incrémentation d'une variable peut s'écrire avec la notation `+=`.

Ainsi `somme += nombre` est équivalent à `somme = somme + nombre`.

→ Nous emploierons préférentiellement la notation `+=` pour la suite de ce cours.

La valeur de la variable somme est modifiée à mesure que le programme s'exécute, pour contenir la somme totale à la fin de l'exécution. On pourrait

facilement faire afficher les valeurs intermédiaires de la variable entre chaque somme avec une instruction `print()`.

Exemple

Le code qui précède peut être réécrit à l'aide d'une **boucle**, à laquelle on ajoute une instruction `print()` pour matérialiser les étapes du calcul.

```
nombres = [2, 9, 12, 3, 5, 11]
somme = 0
for nombre in nombres:

    somme += nombre
    print(somme)
```

Exemple

Le code qui précède produit l'affichage suivant :

```
2
11
23
26
31
42
```

Dans ces deux versions de code impératif, **le programme s'exécute de manière séquentielle** et on observe que la valeur de somme change à chaque étape. Notre code produit des **changements d'état**.

À retenir

Les langages assembleurs sont naturellement basés sur le paradigme impératif. On trouve également C, C++, Java ou Python parmi les langages inspirés du paradigme impératif.

b. Paradigme de programmation procédural

Le paradigme de programmation procédurale est rattaché au style impératif. Il consiste à créer des **procédures** (ou routines) qu'il est possible d'appeler pour obtenir l'**exécution de courtes séquences d'instructions**. Ce découpage en procédures apporte de la modularité : cela facilite le réemploi de portions de code et améliore la maintenabilité des programmes.

À retenir

> Parmi les langages basés sur le paradigme procédural figurent C, C++, LISP, PHP et Python.

Voici une version procédurale de l'addition des éléments d'une liste :

```
def additionne(elements):
    somme = 0

    for element in elements:
        somme += element

    return somme
```

La procédure est ensuite appelée chaque fois que nécessaire :

```
print(additionne(nombres))
# affiche 42
```

La programmation impérative consiste en une succession d'étapes entraînant des modifications d'état.

Voyons maintenant comment le même problème peut être traité avec une approche fonctionnelle.

2 | Programmation fonctionnelle

a. Généralités sur la programmation fonctionnelle

Rappel

> La programmation fonctionnelle se rapproche du traitement mathématique par équations mathématiques. Les entrées sont traitées pour **fournir un résultat en sortie**, sans conservation d'état interne ni mutation de données.

Certains langages purement fonctionnels interdisent de déroger à ces principes, mais le modèle multi-paradigmes de Python nous permet de le faire pour illustrer par un exemple une programmation qui ne respecte pas le paradigme fonctionnel.


Exemple

```
a = 10

def somme_bancale(b):
    """Additionne deux nombres,
    l'un étant passé en argument (b)
    l'autre étant une variable globale (a)"""
    return a + b

print(somme_bancale(5))
# affiche 15

a = 12

print(somme_bancale(5))
# affiche 17
```

Dans cet exemple, le résultat de la fonction ne dépend pas uniquement de la valeur passée en argument en entrée : il dépend aussi de l'état de la variable a qui est extérieure à la fonction.

→ Cette fonction ne répond donc pas aux exigences du paradigme fonctionnel.


Attention

> L'emploi de fonctions ne suffit pas à lui seul à rendre un programme conforme au paradigme fonctionnel.

Définissons maintenant une fonction conforme au paradigme fonctionnel :

```
def somme_correcte(a, b):
    """Additionne deux nombres
    passés en arguments à la fonction."""
    return a + b

print(somme_correcte(10, 5))
# affiche 15
```

Cette fonction respecte le paradigme fonctionnel inspiré des mathématiques et retourne toujours le même résultat pour les mêmes valeurs d'entrée. Aucun état interne n'est conservé et la fonction n'a occasionné aucune mutation : elle n'a pas modifié les données reçues en entrée.

On observe que la création d'une variable globale portant le même nom qu'une des variables locales à la fonction n'a aucun effet sur le fonctionnement de cette dernière.

```
a = 37
print(somme_correcte(10, 5))
# affiche 15
```

En programmation fonctionnelle, une fonction ne doit pas modifier d'éléments extérieurs à son environnement local. Dans le cas contraire elle produit ce qu'on appelle un **effet de bord**. Illustrons cela avec un exemple d'appel de fonction sur un objet mutable comme une liste.

```
def retire_dernier(enumeration):
    enumeration.pop()
    return enumeration

nombres = [2, 3, 9]
print(nombres)
# affiche [2, 3, 9]

print(retire_dernier(nombres))
# affiche [2, 3]
```

```
print(nombres)
# affiche [2, 3]

print(retire_dernier(nombres))
# affiche [2]

print(nombres)
# affiche [2]
```

→ La variable `nombres`, extérieure à la fonction, est modifiée à chaque appel de fonction : c'est un effet de bord.

Pour l'éviter on redéfinit la fonction afin qu'elle effectue une **copie de la liste** et travaille uniquement sur cette copie. Ainsi la liste originale passée en argument n'est pas modifiée par la fonction.

```
def retire_dernier(enumeration):
    nouvelle_enumeration = enumeration[:]
    return nouvelle_enumeration

nombres = [2, 3, 9]
print(nombres)

print(retire_dernier(nombres))
#affiche [2, 3, 9]

print(retire_dernier(nombres))
#affiche [2, 3, 9]
```

La programmation fonctionnelle étant inspirée des fonctions mathématiques, plusieurs concepts ont été transposés en conservant leur nom. C'est le cas des expressions lambda.

b. Expression lambda

Les expressions lambda sont basées sur le lambda-calcul conceptualisé par Alonzo Church dans le cadre de ses recherches sur la calculabilité dans les années 1930, ainsi que nous l'avons vu dans le cours précédent sur les programmes et les données.

Pour découvrir les expressions lambda, prenons comme base une liste de noms de villes dont nous pouvons aisément produire un affichage trié.

```
villes = ['Paris', 'Strasbourg', 'Toulouse', 'Lyon']
print(sorted(villes))
# affiche ['Lyon', 'Paris', 'Strasbourg', 'Toulouse']
```

Rappel

> On rappelle que `sorted()` retourne par défaut une copie de la liste triée en ordre croissant pour des nombres et par ordre lexicographique (alphabétique) pour des chaînes de caractères. La liste d'origine n'est pas modifiée.

Nous pouvons modifier le comportement par défaut de `sorted()` et effectuer un tri avec un critère personnalisé. Nous précisons ce critère de tri avec le paramètre optionnel `key`.

- Ainsi pour obtenir une version de notre liste triée en fonction de la longueur croissante du nom de chaque ville, nous faisons référence à la fonction `len()`.

```
print(sorted(villes, key=len))
# affiche ['Lyon', 'Paris', 'Toulouse', 'Strasbourg']
```

Astuce

> On notera que l'argument du paramètre `key` est le nom de la fonction dépourvu de parenthèses.

- Nous pouvons également définir notre propre critère de tri : il suffit pour cela de créer une fonction qui l'exprime. Choisissons de trier la liste des villes en fonction de la dernière lettre de chaque mot.

Dans un premier temps, nous définissons de manière classique la fonction qui nous servira de critère de tri.

```
def derniere_lettre(chaine):
    return chaine[-1]
```

Effectuons maintenant un tri basé sur notre fonction nouvellement créée.

```
print(sorted(villes, key=derniere_lettre))
# affiche ['Toulouse', 'Strasbourg', 'Lyon', 'Paris']
```

→ La liste est bien triée en fonction de la dernière lettre de chaque mot : « e », « g », « n » et « s ».

- Voyons maintenant une autre notation, appelée notation lambda, qui permet d'obtenir le même résultat avec une syntaxe différente : on nomme et on définit une fonction sur une seule ligne.

```
derniere_lettre = lambda chaine: chaine[-1]
```

```
print(sorted(villes, key=derniere_lettre))
# affiche ['Toulouse', 'Strasbourg', 'Lyon', 'Paris']
```

→ Le résultat est identique.

Contrairement aux fonctions définies classiquement, les expressions lambda ne peuvent pas contenir d'instructions et se limitent à des expressions. Elles peuvent toutefois comporter plusieurs paramètres en entrée.

Les expressions lambda permettent aussi de créer des **fonctions anonymes**, c'est-à-dire dépourvues de noms, destinées à être utilisées ponctuellement. L'exemple précédent peut ainsi être reformulé comme suit :

```
print(sorted(villes, key=lambda chaine: chaine[-1]))
# affiche ['Toulouse', 'Strasbourg', 'Lyon', 'Paris']
```

Aucun nom n'est assigné à la fonction, qui est définie à la suite de `lambda` et aussitôt appliquée. Les fonctions anonymes sont en quelque sorte des fonctions « jetables ». Leur utilisation est tellement ponctuelle qu'il n'est même pas nécessaire de les nommer.

C. Transformation, filtrage et réduction

Voici trois fonctions complémentaires de traitement typiques de la programmation fonctionnelle : `map()`, `filter()` et `reduce()`.

→ Elles permettent, respectivement, d'appliquer des transformations, filtres et réductions.

Ces fonctions prennent en premier argument une fonction qui sera appliquée à chacun des éléments individuels composant le second argument (ici notre liste de nombres). Examinons dans un premier temps le fonctionnement de `map()` en élevant au carré chacun des éléments de la liste de nombres.

1. `map()`

```
nombres = [2, 9, 12, 3, 5, 11]
for element in map(lambda n: n** 2, nombres):
    print(element)
```

Produit l'affichage suivant :

```
4
81
144
9
25
121
```

Les valeurs itérées depuis la fonction `map()` peuvent facilement servir à former une nouvelle liste.

```
print(list(map(lambda n: n** 2, nombres)))
# affiche [4, 81, 144, 9, 25, 121]
```

2. `filter()`

De la même manière on peut aisément filtrer notre liste de nombre pour ne retenir que ceux supérieurs à dix.

```
print(list(filter(lambda n: n > 10, nombres)))
# affiche [12, 11]
```

Les fonctions `map()` et `filter()` existent dans de nombreux langages informatiques. Présentes en Python, elles peuvent la plupart du temps être avantageusement remplacées par des **compréhensions de listes**, également appelées **listes en intension**.

```
# nombres au carré
print([n** 2 for n in nombres])
# affiche [4, 81, 144, 9, 25, 121]
```

```
# nombres supérieurs à 10
print([n for n in nombres if n > 10])
# affiche [12, 11]
```

La bibliothèque standard de Python comporte le module de programmation fonctionnelle Functools, dédié aux fonctions de haut niveau, c'est-à-dire des fonctions pouvant agir sur des fonctions ou en retourner. On y trouve notamment la fonction `reduce()`.

3 `reduce()`

Dans l'exemple ci-dessous on utilise la fonction `reduce()` combinée à une fonction anonyme (`lambda`) pour effectuer la somme des éléments contenus dans la liste `nombres`.

```
import functools
print(functools.reduce(lambda m, n : m + n, nombres))
# affiche 42
```

Astuce

> Précisons qu'on aurait aussi pu utiliser la fonction native `sum()`.
> ```
> print(sum(nombres))
> # affiche 42
> ```

La programmation fonctionnelle s'articule autour de fonctions de traitement qui n'entraînent pas de mutation de données et ne conservent pas d'état.
Voyons maintenant comment le même problème peut être traité avec une approche objet.

À retenir

> Voici quelques langages orientés programmation fonctionnelle : Sccala, LISP, OCaml, Haskell, Python, C++, Ruby ou encore Java.

3 Programmation objet

À retenir

> Le paradigme de **programmation orientée objet** s'articule autour du concept central d'objets. Ces objets définis en **classes** peuvent contenir à la fois des données et du code et interagir avec d'autres objets.

Dans ce paradigme :

- les données rattachés aux objets sont appelées **attributs** ou **propriétés** ;
- les procédures ou fonctions rattachées aux objets sont appelées **méthodes**.

a. Définition d'une classe

La définition d'une classe prend la forme suivante :

```
class Enumerations:
    def __init__(self, une_liste):
        self.liste = une_liste

    def additionne(self):
        self.somme = sum(self.liste)
```

On définit comment les objets de la classe sont initialisés avec la méthode spéciale `__init__()`. Cette méthode est appelée « constructeur » dans d'autres langages informatiques.
Le paramètre `self` fait référence à l'**instance**, autrement dit à l'objet appartenant à la classe, lequel peut aussi posséder différents attributs. Selon les besoins, ces attributs peuvent être définis ou modifiés lors de l'initialisation ou bien ultérieurement.

On définit ensuite des méthodes applicables à chacun des objets de la classe ainsi créée. Ces méthodes peuvent créer, accéder, modifier, ou supprimer des attributs de l'objet considéré.

Dans le cas présent nous créons une méthode `additionne()` qui prend comme argument `self` et génère un attribut `somme` spécifique à chaque objet.

Une fois la classe créée, il est possible de créer des instances, c'est-à-dire des objets individuels appartenant à la classe « Enumerations » que nous venons de définir.
L'instanciation s'effectue de la manière que nous allons détailler en suivant.

b. Instanciation d'un objet

La création d'un objet s'effectue depuis la classe, en passant les arguments attendus au niveau de la méthode `__init__()`.

```
mon_enumeration = Enumerations(nombres)
```

Les méthodes sont ensuite accessibles au niveau de l'objet nouvellement créé.

```
mon_enumeration.additionne()
print(monenumeration.somme)
# affiche 42
```

Le code du paradigme objet est très différent des paradigmes impératif et fonctionnel présentés précédemment, mais il produit bien le résultat attendu.

Nous pouvons créer un second objet appartenant à cette même classe, avec une autre liste de nombres.

```
monautreenumeration = Enumerations([3, -4, 19, 2, 0])
```

Cet objet possède lui aussi la méthode `additionne()`, que nous pouvons appeler afin de générer l'attribut somme rattaché à cet objet (et différent de l'objet précédent).

```
monautreenumeration.additionne()
print(monautreenumeration.somme)
# affiche 20
```

Les valeurs des attributs sont spécifiques à chaque instance ou objet de la classe.

À retenir

> Dans le paradigme objet, l'**instanciation** consiste à créer un objet appartenant à une classe donnée.
> L'**encapsulation** désigne le regroupement des données et des procédures au sein de l'objet.
>
> → L'accès aux données s'effectue donc au travers des mécanismes d'interface proposés par l'objet.

La programmation objet comporte deux autres notions essentielles : l'héritage et le polymorphisme.

c. Héritage et polymorphisme

Définition

> **Héritage :**
>
> L'héritage consiste à définir des sous-classes, ou classes-enfants, capables d'hériter des propriétés de la classe parente.

Définition

> **Polymorphisme :**
>
> Le polymorphisme consiste à pouvoir adapter le comportement d'une méthode d'une classe à l'autre.

Illustrons ces deux propriétés en créant des classes d'animaux, en commençant par une classe « Animal » adaptée à tous les animaux sans

distinction.
Considérant que tous les animaux sont capables de pousser un cri, nous créons une méthode dédiée appelée `crie()`. Le cri émis pourra cependant être différent d'un animal à l'autre : le cri du chat est le miaulement et le cri du chien l'aboiement. Un animal d'une espèce indéterminée pousse un cri indéterminé.

```
class Animal:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def crie(self):
        return 'cri indéterminé'

animal1 = Animal('Finaud', 9)

print(animal1.nom)
# affiche Finaud

print(animal1.crie())
# affiche cri indéterminé
```

Si nous définissons une classe enfant, dérivée de la classe Animal, elle hérite automatiquement de ses méthodes et de ses propriétés.

```
class Chat(Animal):
    pass
```

L'instruction `pass` nous permet de définir notre classe sans méthodes ni attributs, afin de mettre en évidence le caractère automatique du mécanisme d'héritage.

```
chat1 = Chat('Félix', 7)

print(chat1.nom)
# affiche Félix

print(chat1.crie())
# affiche cri indéterminé
```

On constate bien que la sous-classe « Chat » a hérité des caractéristiques de la classe parente « Animal » : nous pouvons créer des objets de classe « Chat » capables de pousser un cri (indéterminé) alors que rien n'a été défini au niveau de la classe « Chat ».

À retenir

> Ce principe d'héritage nous offre la possibilité de redéfinir uniquement certaines caractéristiques en fonction des besoins, et de conserver toutes les autres.

Nous allons redéfinir la sous-classe « Chat » de manière plus utile, en lui attribuant également un surnom optionnel et en personnalisant la méthode `crie()`.

```
class Chat(Animal):
    def __init__(self, nom, age, surnom=None):
        Animal.__init__(self, nom, age)
        self.surnom = surnom

    def crie(self):
        return 'Miaou'
```

L'instanciation s'appuie sur celle de la classe parente mais ajoute la création de l'attribut `surnom` spécifiquement pour les chats. Ayant indiqué une valeur `None` par défaut, on peut créer un chat sans surnom, mais s'il en a un et qu'on le précise, il sera pris en compte.

```
chat1 = Chat('Gavroche', 4)
chat2 = Chat('Félix' , 5, 'Diesel')

print(chat1.nom)
# affiche Gavroche

print(chat2.nom)
# affiche Félix

print(chat1.surnom)
# affiche None
```

```
print(chat2.surnom)
# affiche Diesel

print(chat1.crie())
# affiche Miaou

print(chat2.crie())
# affiche Miaou
```

Les objets partagent les méthodes, éventuellement héritées et éventuellement modifiées, tout en possédant des valeurs d'attributs distinctes.

À retenir

> Parmi les langages de programmation orientée objet, on compte C++, Javascript, Python, Java, Ruby ainsi que SmallTalk.

> Conclusion :
>
> Dans ce cours nous avons présenté successivement trois styles majeurs de programmation informatique : le paradigme impératif, le paradigme fonctionnel et le paradigme objet, dont nous avons présenté les principales caractéristiques. Il existe d'autres paradigmes, parmi lesquels la programmation événementielle ou encore la programmation concurrente, qui ne sont pas étudiés dans le cadre de ce cours.
> On pourrait se demander quel est le meilleur paradigme parmi tous ceux-ci ? Cette question fait souvent l'objet de débats animés et passionnés entre développeur·se·s, mais elle n'a pas de réponse unique car les paradigmes constituent autant de manières, parfois très différentes, d'apporter une solution à un problème donné. Nous l'avons montré avec un même exemple abordé selon trois paradigmes distincts. Certains langages informatiques imposent le recours à un paradigme donné, tandis que d'autres (comme Python) en supportent plusieurs, et ces paradigmes peuvent même être panachés le cas échéant au sein d'un même programme. Les préférences stylistiques du·de la développeur·se et les pratiques en vigueur ont également une influence sur le choix d'un paradigme de programmation.

On peut toutefois noter que la programmation orientée objet est assez fréquemment employée pour la réalisation d'interfaces graphiques ou de jeux, fortement liés à des changements d'état. Par ailleurs, on ne peut ignorer l'essor actuel que connaît le paradigme fonctionnel : celui-ci semble en effet mieux répondre que les autres aux contraintes de fiabilité et de performance, en particulier pour la mise en œuvre de l'exécution de portions de code en parallèle.
Enfin, le choix d'un paradigme de programmation influence celui du langage de programmation dont les critères suivants viennent compléter celui d'implémenter ou non le paradigme de son choix :

- Existence d'une grande communauté de développeur·se·s : partage de bibliothèques, d'expériences, etc.
- Facilité d'apprentissage
- Capacité à mettre en œuvre plusieurs paradigmes
- Langage compilé ou interprété
- Avec ou sans pointeur, selon qu'on privilégie l'optimisation dans le premier cas où la rapidité de développement et la fiabilité dans le second cas.