6.1 Les paradigmes de programmation

NSI Tle - JB Duthoit

6.1.1 Introduction

Définition 6.8

Un ***paradigme*** est un point de vue particulier sur la réalité, an angle d'attaque privilégié face à un problème, un état d'esprit.

Remarque

Un paradigme a lui seul ne permet qu'une vision limitée de la réalité.
Il existe des milliers de langages informatiques, souvent classés par catégorie suivant leur fonctionnement ; ces catégories sont appelés ***paradigmes de programmation*** .

6.1.2 le paradigme orienté objet

Programmer avec un langage orienté objet amène le programmeur à identifier les "acteurs" qui composent le problème, puis de déterminer ce qu'est et ce que doit savoir faire chaque acteur.
Ce type de programmation est très puissant, et offre un excellent outil de modularité.
Les programmes orientés objets sont modifiables et réutilisables.

Exemple

- C++
- Java
- Python :-)

Reprenons l'exemple :

```
liste = [2,1,6]
liste.sort()
```

Lorsque l'on crée la liste, on crée un objet liste qui est une instance de la classe liste. Cette classe possède beaucoup de méthodes publiques, dont *sort()* qui ordonne les éléments de façon croissante.

6.1.3 La programmation impérative

Jusqu'à présent nous avons vu un seul paradigme de programmation : la programmation impérative.
La programmation impérative respecte plusieurs points :

- La séquence d'instructions (les instructions d'un programme s'exécutent étape par étape). On parle d'instructions *séquentielles*.

- l'affectation (on attribue une valeur à une variable, par exemple : x = 15)
- l'instruction conditionnelle (if / else)
- les boucles (while et for)

La programmation impérative est la plus courante et la plus ancienne : elle a été inventé conjointement aux premiers ordinateurs.

Exemple

- C
- Pascal
- COBOL
- Fortran
- Python :-)

Exercice 6.59

Construire en langage python deux fonctions qui permettent à terme de renvoyer un tableau trié :

```
def insere(t,i):
    '''
    t tableau contenant au moins (i+1) entiers.
    les i premiers entiers sont triées: t[0,...,i-1] supposé
   trié
    la fonction insère t[i] à la bonne place
    pour que t[0,1,...,i] soit un tableau trié.
    exple
    t = [2,3,4,6,5,1]
    >>> insere(t,4)
    >>> t
    [2,3,4,5,6,1]
    >>> insere(t,5)
    >>> t
    [1,2,3,4,5,6]
    '''
    pass

def tri_insertion(t):
    '''
    trie le tableau t dans l'ordre croissant
    en utilisant la fonction insere(t,i)
    '''
    pass
```

Listing 6.1 – Le tri insertion en impératif

Exercice 6.60

On considère maintenant le tableau suivant, contenant des noms, prénoms d'élèves, avec leur date de naissance :

```
eleves = [  ('Théo','Courant',2,10,1942),
       ('Alain','Proviste',4,1,1955),
       ('Sarah','Vigotte',21,11,1978),
       ('Jean','Bonboeur',25,12,1988),
       ('Judas','Bricot',13,4,1973)]
```

1. Si on utilise la fonction `tri_insertion` précédente pour trier la liste `eleves`, quel résultat obtient-on ?
2. Modifier le programme précédent pour qu'il trie la liste `eleves` par ordre d'année de naissance croissante.
3. Modifier le programme précédent pour qu'il trie la liste `eleves` par ordre alphabétique des noms de famille.

Remarque

On comprend vite qu'il faudra faire des copier-coller de la fonction initiale pour l'adapter au tri demandé...
☛ Ce n'est clairement pas optimal, car une erreur dans le code initial sera dupliquée autant de fois que de copier-coller...Pire : si une erreur est trouvée et corrigée, elle ne le sera probablement que dans une partie du code...

6.1.4 Paradigme fonctionnel

Une bien meilleure solution consiste à passer la fonction de comparaison en argument de la fonction tri !

Exercice 6.61

On souhaite créer la fonction `inf1(x,y)` qui prend en argument deux entiers et qui renvoie `True` si `x` et inférieur ou égal à `y`, `False` sinon.

Exercice 6.62

Adapter les fonction du programme ?? pour passer en argument `inf1` :

```
def insere(inf,t,i):
    pass


def tri_insertion(inf,t):
    pass
```

Exercice 6.63

Trier la liste `eleves` en créant juste une autre fonction `inf2` que l'on passera en argument de la fonction `tri_insertion(inf,t)`

Une fonction accepte des données et renvoie des données.

Une fonction est un objet de première classe sur lequel d'autres fonctions peuvent opérer.

☞ Ce paradigme ne contient pas la notion de variable ! **Un programme en langage fonctionnel ne produit jamais d'effets secondaires !**.

Ce paradigme propose un point de vie "transformationnel" : tout comportement doit être vu comme un enchaînement de transformations sur un état initial et produisant un état final.

Exemple

Voici des exemples de langages adaptés pour la programmation fonctionnelle :

- Lisp
- Ocaml
- LOGO
- Python :-)

Les fonctions anonymes

L'expression `lamda x,y : x[3] <= y[3]` désigne une fonction, anonyme, qui prend 2 arguments x et y et qui renvoie le booléen `x[3] <= y[3]`

Exercice 6.64

Supposons qu'un tableau python contienne n valeurs x_1,x_2,...,x_n.
On souhaite calculer le résultat de :

$$x_1 \oplus x_2 \oplus ... \oplus x_n$$

pour une certaine opération $\oplus$.
Écrire un programme qui prend en argument un tableau de valeurs et une fonction `op` qui représente l'opération $\oplus$. On supposera que le tableau contienne au moins une valeur.
Il faudra que votre programme fonctionne dans les cas suivants, en utilisant une fonction anonyme :

- `t` est un tableau d'entiers et $\oplus$ représente l'addition
- `t` est un tableau de flottants et $\oplus$ représente le maximum.
- `t` est un tableau dont les valeurs sont des str, et $\oplus$ représente la concaténation.

6.1.5 Fonctions renvoyées comme résultat

Prenons l'exemple de la dérivée d'une fonction.
Considérons la fonction carré :

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

On rappelle que la dérivée $f'(x)$ d'une fonction peut être approchée par :

$$\frac{f(x+h)}{-} f(x)h$$

lorsque h est une valeur proche de zéro (raisonnablement proche de zéro).

Exercice 6.65

Ecrire une fonction `derive(f)` qui prend en argument une fonction `f` et qui renvoie une fonction qui est une approximation de la fonction dérivée de `f`.
On pourra vérifier avec la fonction `carre` :

```
>>> g = derive(carre)
>>> g
<function derive.<locals>.<lambda> at 0x03C376F0>
>>> g(3)
6.000000087880153
>>> g(-3)
-5.999999892480901
>>> g(5)
10.000000116860974
```

Vérifier que ces résultats correspondent bien aux valeurs approchées de la fonction f' quand f est définie sur $\mathbb{R}$ par $f(x) = x^2$ (On rappelle que la fonction f' est ici la fonction définie sur $\mathbb{R}$ par $f'(x) = 2x$). ☛ Essayer sur d'autres fonctions !

Attention aux effets de bord !

Considérons le programme suivant :

```
l = [4,7,3]
def ajout(i):
  l.append(i)
```

Le programme ci-dessus respecte-t-il la logique du paradigme fonctionnel ?
Et celui-ci :

```
def ajout(i,l):
        x = l + [i]
  return x
```

☞ Le paradigme fonctionnel va amener le programmeur non pas à modifier une valeur existante, mais plutôt à créer une nouvelle grandeur à partir de la grandeur existante : une grandeur existante n'est jamais modifiée, donc aucun risque d'effet de bord.

Selon vous, le programme ci-dessus respecte-t-il le paradigme fonctionnel ?

6.1.6 Utiliser le "bon" paradigme de programmation

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 fonctionnelle sera préférable.