17.1 Les paradigmes de programmation

NSI terminale - JB Duthoit

17.1.1 Introduction

Définition

Un **paradigme** est un point de vue particulier sur la réalité, un 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** .

17.1.2 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 :-)

17.1.3 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.

17.1.4 Paradigme fonctionnel

L'idée fondamentale de la programmation fonctionnelle 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.

Dans le cadre de la programmation fonctionnelle, on utilise des **fonctions pures**. Une fonction pure est une fonction qui ne produit jamais d'effets de bord.

Fonctions pures ?

Exercice 17.1

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 ?

Exercice 17.2

Et celui-ci :

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

☞ 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.

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 - Hors programme mais pratiques

L'expression `lambda 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 17.3

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 une fonction `calcul(t,op)` qui prend en argument un tableau de valeurs `t` et une fonction `op` qui représente l'opération $\oplus$. La fonction `calcul` renvoie le résultat attendu en fonction de l'opération souhaitée $\oplus$.
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 ou des fonctions spécifiques :

- `t` est un tableau d'entiers et $\oplus$ représente l'addition. On utilisera ici la fonction `op1`.
- `t` est un tableau d'entiers et $\oplus$ représente la multiplication. On utilisera ici la fonction `op2`.
- `t` est un tableau de flottants et $\oplus$ représente le maximum. On utilisera ici la fonction `op3`.
- `t` est un tableau dont les valeurs sont des str, et $\oplus$ représente la concaténation. On utilisera ici la fonction `op4`.

```
def op1(a,b):
    ...
def op2(a,b):
    ...
def op3(a,b):
    ...
def op4(a,b):
    ...
def calcul(t,op, indice = 0):
    """
    t est un tableau,
```

```
  op une fonction (addition, multiplication, maximum, concaté
   nation)
  la fonction renvoie le résultat attendu en fonction de l'op
   ération
  """
  if indice == len(t) - 1:
    return ...
  return ...

t = [1,2,4]
assert calcul(t,op1) == 7
assert calcul(t,op2) == 8
assert calcul(t,op3) == 4
t = ['1','2','4']
assert calcul(t,op1) == '124'
```

☛ **Le paradigme de la programmation fonctionnelle repose sur plusieurs concepts, en voici deux :**

- **On ne modifie pas les valeurs déjà construites : on en construit de nouvelles.**
- **Les fonctions sont des valeurs comme les autres : passées en arguments, renvoyées comme résultat ou encore stockées comme des données**

Exercice 17.4

Ecrire une fonction `trouve(p,t)` qui reçoit en argument une fonction `p` (`p` pour propriété) et un tableau `t`. La fonction renvoie le premier élément `x` de `t` tel que `p(x)` vaut `True`. Si aucun élément ne satisfait `p` alors la fonction renvoie `None`.

```
def p1(elt):
  """"
  renvoie True si n est pair, False sinon.
  """
  ...

def trouve(p,t,indice = 0):
  if indice = len(t) :
    return ...
  if p(t[indice]):
     return ...
  return ...
t = [1,31,4,18,7]
assert trouve(p1,t) == 4
```

Exercice 17.5

Les fonctions `map` et `reduce` sont (avec la fonction `filter`), les fondements de la programmation fonctionnelle.
On les traduit ici par `applique` et `reduit`, et on va chercher à les construire !.

1. La fonction `applique` a deux paramètres : une fonction et une liste. Elle renvoie une nouvelle liste contenant les images par la fonction des éléments de la liste initiale.

a) Créer la fonction `applique(fonction,liste)` qui réalise cela :

```
>>> applique(len,['Paul','Sarah','Charlotte'])
[4, 5, 9]
>>> applique(lambda x: x ** 2,[2,3,4,5])
[4, 9, 16, 25]
```

b) Utiliser cette fonction `applique` pour mettre les éléments d'une liste de caractères en majuscule.

2. La fonction `reduit(fonction, depart, liste)` prend une fonction , une valeur de départ, une liste , et renvoie une combinaison des éléments de la liste.
Par exemple :

```
>>> reduit(lambda x,y : x + y,'',['voi','tu','re'])
'voiture'
```

a) Créer la fonction `reduit`.

b) En utilisant la fonction `reduit`, créer une fonction `tous_ vrais` qui vérifie qu'une liste de booléens sont tous égaux à `True` :

```
>>> tous_vrais([3 > 2,4 % 2 == 0, len('voiture') == 7])
True
```

c) Utiliser `reduit` pour calculer le nombre d'éléments d'une liste :

```
>>> taille('voiture')
7
```

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 17.6

Écrire 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 !

17.1.5 Paradigme événementiel

La programmation événementiel consiste à associer du code pour qu'il soir exécuté lorsque survient un événement. Observer, analyser et tester le code ci-dessous.

```
import tkinter as tk
fenetre = tk.Tk()
b = tk.Button(fenetre, text="0")
b.pack()

compteur = 0
def clic(evenement):
    global compteur
    compteur += 1
    evenement.widget.configure(text=str(compteur))

b.bind("<Button-1>",clic)

fenetre.title("Cliquer sur le bouton")
fenetre.geometry("800x400")
fenetre.mainloop()
```

Exercice 17.7

Soit le code suivant, qui définit une étiquette et 10 boutons. Compléter la fonction `changer(c)` pour que le chiffre passé en argument soit utilisé comme étiquette pour l'étiquette.

```
import tkinter as tk
fenetre = tk.Tk()
chiffre = tk.Label(fenetre)
chiffre.grid(row=1,column=1,columnspan=10)

def changer(c):
    return

for i in range(10):
    c = str(i)
    b = tk.Button(fenetre, text = c)
    b.grid(row=2,column=i+1)
    b.bind("<Button-1>",changer(c))

fenetre.mainloop()
```

***

17.1.6 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.