Informatique 3 : Programmation structurée

I Introduction

Pour le moment, les fonctions Python qu'on a créées ne comportent que des commandes simples. Dans ce chapitre, on verra comment faire des structures conditionnelles ou répétitives. On utilisera ces structures logiques en faisant attention à leur construction, par **bloc d'instructions**, c'est-à-dire comportant des indentations :

Instruction initiale :
indentation Instruction 1
(...)
Instruction n

On peut combiner ces structures à condition d'imbriquer les blocs et de jouer avec l'`indentation`. On distinguera les instructions conditionnelles et les instructions répétitives qui constitueront les deux parties suivantes du chapitre.

II Les instructions conditionnelles

1 Notion de booléen

Si dans Python on tape

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

ce qui fait fait apparaître un nouveau type de variable : les **booléens**. Un booléen n'a que deux valeurs possibles : vrai `True` ou faux `False`. Les booléens s'utilisent pour réaliser des tests :

```
>>> 2 > 3
False
>>> 4 > 3
True
```

Pour effectuer un test faisant intervenir des expressions x et y, on utilisera les commandes suivantes :

commande Python	Interprétation	commande Python	Interprétation
`x == y`		`x <= y`	
`x != y`		`x in y`	
`x > y`		`x and y`	
`x < y`		`x or y`	
`x >= y`		`not x`	

Remarque 1

Exemple 2

On tape dans la console de Python les différentes commandes :

```
>>> 2 <= 3
>>> 2 == 2.0
>>> 'a' < 'b'
>>> 'b' == 'ba'
>>> a == a
>>> 3 < 'a'
>>> 2 in [2,3,5]
>>> 2 in [1,3,5]
>>> 2.0 in [2,3,5]
>>> (2 < 3) or (2 > 3)
>>> (2 < 3) and (2 > 3)
>>> not 2
```

Expliquons le résultat de ces différentes commandes :

2 Structures conditionnelles

Il s'agit d'une structure permettant l'exécution d'une série d'instructions selon qu'une certaine condition est réalisée ou non. Dans le langage Python, on a recours aux commandes `if`, `elif`, `else`, et la syntaxe d'un tel **bloc d'instructions** sera toujours la suivante :

`if` *condition 1* :
indentation Instruction 1
(...)
`elif` *condition 2* :
indentation Instruction 1
(...)
`elif` *condition 3* :
indentation Instruction 1
(...)
(...)
`else` :
indentation Instruction 1
(...)

Pour séparer les instructions, on pourra encore utiliser le symbole ; ou bien pour faciliter le **debogage**, on pourra préférer un **saut de ligne**. Par contre, on veillera à ce que les *conditions* énoncées soient bien des variables de **type booléen** que nous avons vues précédemment.

Exemple 3

On donne les fonctions Python :

```
def f(x):
    if x >= 0:
        return x
    else:
        return -x

def g(x):
    if x = 0:
        return 'x est nul'
    else:
        return 'x n est pas nul'

def h(x):
    if x > 0:
        return x
    elif x < 0:
        return -x
```

Expliquons le fonctionnement de chaque fonction.

Exemple 4

Écrivons une fonction Python `racines_reelles(a,b,c)` qui, en fonction $a, b, c \in \mathbb{R}$ renvoie les racines réelles éventuelles du polynôme $aX^2 + bX + c$ et retourne un message du type `'pas de racines reelles'` lorsque les racines ne sont pas réelles.

III Les instructions répétitives

1 Notion de liste

1.1 Définition d'une liste et opérations de base

Une liste est une collection d'objets, qui peuvent être de types différents, entre crochets et séparés par des virgules, ces objets étant numérotés de 0 à $k-1$ où k est le nombre d'objets. En Python, une liste est représentée entre crochets et ses éléments séparés par des virgules. Par exemple, on peut définir une liste `L` de la façon suivante :

```
>>> L = [1,2,4,5,7,1,0]
```

Une liste peut contenir des objets de n'importe quel type et peut mélanger différents types d'objets voire contenir elle-même une liste :

```
>>> L = [1.24, 1 , 'ok', [1,4], True ]
```

Une liste peut être vide :

```
>>> L = []
```

On accède aux éléments individuels d'une liste en indiquant leur indice entre crochets. La numérotation des éléments commence à zéro et on peut utiliser des indices négatifs pour compter à partir de la fin de la liste. Par exemple :

```
>>>L=[1,2,5,1,6]
>>> L[1] ; L[-1] ; L[-2]
2
6
1
```

La fonction `len` donne la longueur de la liste. Avec l'exemple précédent,

```
>>> len(L)
5
```

Remarque 5

L'opération `+`, comme pour les chaines de caractères, concatènent les listes :

```
>>> [1,2,4]+['a',1.1]
[1,2,4,'a',1.1]
```

et la multiplication par un entier k concatène k fois la liste avec elle-même :

```
>>> [1,2,4]*4
[1,2,4,1,2,4,1,2,4,1,2,4]
```

Pour ajouter un élément au bout d'une liste, on peut utiliser la commande `append` :

```
>>> L = []
>>> L.append(2)
>>> L
[2]
```

Pour modifier un élément dans une liste, on peut procéder de la façon suivante :

```
>>> L = [1,4,5]
>>> L[1] = 0
>>> L
[1,0,5]
```

Exemple 6

Écrire une fonction Python `base(k,n)` qui crée une liste de longueur n contenant uniquement des zéros sauf en position k où la liste contient un 1.

1.2 Définition énumérative d'une liste et commande `range`

La commande `range(a,b)` crée une collection d'entiers de `a` jusqu'à `b-1` et espacés de 1. La commande `range(a,b,p)` crée une collection d'entiers de `a` jusqu'à `b-1` et espacés de `p`.

Remarque 7

Exemple 8

Définissons une liste `L` donnant les multiples de 3 de 0 à 30.

On peut utiliser également, pour définir une liste, l'instruction `for` pour parcourir un objet range :

```
>>> [ i**2 for i in range(1,4)]
[1,4,9]
```

Expliquons les lignes de code précédentes :

Exemple 9

Expliquer le contenu des listes `L1` suivante :

```
L1 = [j for i in range(2,8) for j in range(i,50,i)]
```

1.3 Quelques autres fonctions sur les listes

On peut transformer une liste en liste triée avec la commande `sort` :

```
>>> a = [1,5,4]
>>> a.sort()
>>> a
[1,4,5]
```

Remarque 10

On peut tester l'appartenance d'un élément à une liste avec `in`, trouver le minimum et le maximum avec `min`, `max`, effacer un élément d'une liste avec `del` :

```
>>> L = [1,2,4,10,8,9]
>>> 2 in L ; max(L)
True
10
>>> del L[1]
>>> L
[1,4,10,8,9]
```

Exemple 11

Écrire une fonction Python `caracteristiques(L)` qui renvoie

1. `'ceci n est pas une liste'` lorsque `L` n'est pas une liste ;
2. une liste à trois élément contenant dans l'ordre le plus grand élément, le plus petit élément et la longueur de la liste lorsque `L` est une liste.

On peut créer une liste à partir d'une autre liste moyennant une condition :

```
>>> >>> liste = [1,2,4,5,6,7]
>>> sousliste = [x for x in liste if x < 2]
>>> sousliste
[1]
```

1.4 Conversion en liste

On peut transformer certains objets en listes à l'aide de la commande `list` :

```
>>> list('test')
['t', 'e', 's', 't']
>>> list(range(10))
[0,1,2,3,4,5,6,7,8,9]
```

Remarque 12

Exemple 13

Créer une liste Python `change_caractere(c)` qui à partir d'une chaîne de caractères `c` renvoie la chaîne `c` mais où le second caractère est remplacé par un `a`.

2 Boucles de répétition

Il s'agit de **structures itératives** qui permettent d'effectuer une série d'instructions un nombre de fois donné ou tant qu'une condition est réalisée. Dans le langage Python, on distingue donc la boucle `while` (boucle *tant que*) dépendant d'une **condition booléenne** :

`while` *condition* :
indentation Instruction 1
(...)
Instruction n

et la boucle `for` (boucle *pour*) associée à une liste donnée (un objet `range` ou une liste `L`), qu'elle soit constituée d'entiers obtenus par la fonction `range`, ou constituée de valeurs quelconques :

`for k in range(N):` *indentation* Instruction 1 (...) Instruction n	ou plus généralement,	`for x in L:` *indentation* Instruction 1 (...) Instruction n

Comme pour les structures conditionnelles, on présentera dans Python ce type de structure par **bloc d'instructions** comme dans le schéma précédent.

2.1 Boucle `for`

Quand le nombre d'itérations est déterminé, on peut utiliser une boucle `for` avec les listes présentées dans la partie précédente. Donnons une fonction Python `multiplesneuf(n)` affichant à la suite les multiples positifs de 9 de 0 jusqu'à $9n$:

```
def multiplesneuf(n):
    for i in list(range(0,n+1)):
        print(9*i)
```

Exemple 14

Donnons une fonction Python `sommeentiers(n)` qui retourne $1 + 2 + \cdots + n$.

Exemple 15

Donnons une fonction Python `racines_unite(n)` qui retourne la liste des racines n-ièmes de l'unité sous forme algébrique. Pour convertir un nombre complexe sous forme trigonométrique en forme algébrique, on pourra utiliser `rect(r,theta)` où r est le module du complexe et *theta* un argument de celui-ci.

2.2 Boucle `while`

Quand le nombre d'itérations n'est pas déterminée, on choisit la boucle `while` mais on doit veiller à ce qu'on puisse sortir de la boucle, c'est à dire qu'à un moment la *condition* énoncée ne devra plus être réalisée.

Remarque 16

Donnons l'exemple d'une fonction `multtrois(n)` qui affiche tous les multiples de 3 inférieurs ou égaux à n :

```
def multrois(n):
    k = 0
    while 3*k <= n:
        print(3*k)
        k = k+1
```

On pourra noter que lorsque la condition de boucle n'est plus réalisée, le programme s'arrête.

Exemple 17

Réécrivons la fonction précédente mais celle-ci devra retourner la liste des multiples positifs de 3 inférieur ou égaux à n.

Exemple 18

1. Donnons une fonction Python `sommeentiers(n)` qui retourne $1+2+\cdots+n$ à l'aide d'une boucle `while`.
2. Donnons une fonction Python `sommeentierspairs(n)` qui retourne la somme des entiers pairs inférieurs ou égaux à n :

$$\sum_{0\leq 2k\leq n} 2k$$