IV – Les structures algorithmiques

1/ Notions d’algorithme

1.1) Généralités

Définition Un *algorithme* est une suite d’opérations qui, une fois exécutée correctement, conduit à un résultat et permet de résoudre un problème donné.

Remarque : on en utilise depuis longtemps, que ce soit pour réaliser des tâches matérielles simples ou effectuer des calculs complexes (recettes de cuisine, algorithme d’Euclide).

Les ordinateurs permettent d’automatiser et d’accélérer l’exécution d’un algorithme. Pour cela, il faut d’abord l’écrire en français, en définissant précisément les instructions à effectuer, puis le traduire en langage machine.

1.2) Techniques de base

Passons en revue les différents types d’instructions qui nous permettent de structurer un algorithme, les instructions d’entrée / sortie ayant été vues dans le chapitre précédent.

Affectation de variable

Cette opération consiste à stocker dans une variable une valeur ou bien le résultat d’une opération. On la notera : $\text{variable} \leftarrow \text{valeur}$.

Exemple : on effectue $x \leftarrow 2, y \leftarrow 3, x \leftarrow x + y$; alors x vaut 5 et y vaut 3.

Instructions conditionnelles

Elles se présentent sous deux formes :

SI condition ALORS bloc d’instructions 1	SI condition ALORS bloc d’instructions 1 SINON bloc d’instructions 2
---	---

La condition est un test logique ou une simple variable booléenne. Le bloc d’instructions 1 est exécuté uniquement si la condition est vérifiée. Dans le second cas, on exécute en outre le bloc d’instructions 2 si la condition n’est pas vérifiée.

Remarque : on peut parfaitement imbriquer les instructions les unes dans les autres.

Exemple : le premier des algorithmes page suivante demande deux nombres et affiche le plus grand des deux ; le second (sur 2 colonnes) affiche le plus grand des deux s’il est positif et l’autre sinon.

Demander a et b SI $a \geq b$ ALORS Afficher a SINON Afficher b	Demander a et b SI $a \geq b$ ALORS SI $a \geq 0$ ALORS Afficher a SINON Afficher b	(<i>c'est la suite ...</i>) SINON SI $b \geq 0$ ALORS Afficher b SINON Afficher a
---	--	--

Boucles TANT QUE

Il n'est pas toujours possible de connaître à l'avance le nombre de répétitions nécessaires à l'exécution d'un calcul. Dans ce cas, on utilise la boucle

```

TANT QUE condition
    bloc d'instructions
```

qui répète le bloc d'instructions tant que la condition est vérifiée. Il faut évidemment que la condition évolue en cours de calcul si l'on veut que la boucle s'arrête.

Exemple : déterminons le plus petit entier naturel N tel que $5^N \geq 10\,000$.

```

N ← 0
TANT QUE  $5^N < 10\,000$ 
    N ← N + 1
Afficher N
```

Pour visualiser le fonctionnement d'un algorithme, on utilise un **tableau d'exécution** qui donne les valeurs des différentes variables étape après étape.

Exemple : voici ce que cela donne pour la boucle ci-dessus ... on obtient $N = 6$.

N	0	1	2	3	4	5	6
5^N	1	5	25	125	625	3125	15625

Boucles POUR

Elles permettent de répéter des instructions autant de fois qu'on le souhaite en faisant parcourir un ensemble de valeurs à une variable, **le compteur**. Elles s'écrivent sous la forme

```

POUR compteur DANS ensemble de valeurs
    bloc d'instructions
```

Le compteur est géré automatiquement par la boucle POUR : il n'y a pas besoin de l'initialiser ou de l'incrémenter à chaque étape, contrairement aux boucles TANT QUE. Il peut être utilisé dans les calculs, mais **il ne faut surtout pas modifier le compteur** sous peine d'obtenir des résultats imprévisibles.

Exemple : affichons les carrés des dix premiers entiers non nuls.

POUR N DANS (1,2,...,10) Afficher N^2	POUR N ALLANT DE 1 A 10 Afficher N^2
--	---

1.3) Traduction en langage machine

Quand on passe à l'écriture du programme proprement dit, il faut s'assurer que celui-ci est lisible, ne serait-ce que pour le corriger si nécessaire. De plus, d'autres personnes seront peut-être amenées à l'utiliser ou à le modifier. Pour cela, il faut impérativement :

- utiliser des noms de variables et de fonctions explicites (éviter `gzlurp137_bkz78`) ;
- insérer des commentaires dans le code (avec des `#`) ;
- bien documenter les programmes et les fonctions (on verra cela en temps utile).

2/ Les instructions conditionnelles

2.1) Les tests simples

En Python, les tests SI ... ALORS ... vont s'écrire

```
if condition:
    instructions
```

Il ne faut pas oublier les deux points en fin de première ligne, ni l'indentation de 4 espaces du bloc d'instructions (elle est automatique lors du retour à la ligne). Pour sortir du bloc d'instructions (et du test), il faudra se recaler à gauche à l'aide de la touche `backspace`.

Exemple : comparons les deux codes ci-dessous

<pre>x=4 y=8 if x<2: y=3 y=y/x print("x=",x," et y=",y)</pre>	<pre>x=4 y=8 if x<2: y=3 y=y/x print("x=",x," et y=",y)</pre>
--	--

Comme $x > 2$, on passe directement au `print()` final dans le premier cas, sans modifier y . Dans le second cas, on exécute l'instruction $y = y/x$ d'abord.

Remarque : en cas d'erreur de syntaxe, des petites icônes apparaissent dans la marge de l'éditeur pour les signaler. De même, après la compilation, des messages signalent les erreurs (et même les numéros de ligne) dans la console.

2.2) Les tests avec alternative

Les tests SI ... ALORS ... SINON ... s'écrivent quant à eux

```
if condition:
    instructions1
else:
    instructions2
```

Attention, **un bloc d'instructions ne peut pas et ne doit pas être vide**. Si l'on est tentés de mettre un bloc vide après le `if`, c'est qu'il faut utiliser la condition contraire !

2.3) Les tests imbriqués

La structure précédente est parfaite s'il n'y a que deux cas de figure. Lorsque l'on doit traiter plus de cas, on peut enchaîner les tests mais il faut faire très attention aux indentations. On pensera alors à utiliser l'instruction `elif`, qui est la contraction de `else if` et qui permet de multiplier à volonté les cas envisagés tout en restant lisible.

```

if condition1:
    if condition2:
        instructionsA
    else:
        instructionsB
    instructionsC
elif condition3:
    instructionsD
else:
    instructionsE

```

```

if note_bac<10:
    print('Recalé !')
elif note_bac<12:
    print('Passable')
elif note_bac<14:
    print('Assez bien')
elif note_bac<16:
    print('Bien')
else:
    print('Très bien')

```

Exemple : ci-dessus à droite, on affiche la mention à partir de la note obtenue au bac.

3/ Les boucles

Tout comme les instructions conditionnelles, les boucles TANT QUE et les boucles POUR vont pouvoir s'imbriquer les unes dans les autres sans aucun problème.

Deux instructions spéciales ... et utiles

<code>continue</code>	Passer à la prochaine itération d'une boucle.
<code>break</code>	Sortir de la plus petite boucle englobante.

3.1) Les boucles TANT QUE

Elles s'écrivent

```

while condition:
    instructions

```

On sort de la boucle lorsque la condition devient fausse. Il est parfois utile de créer une boucle infinie que l'on interrompt le moment voulu.

Exemples : le premier code renvoie le plus petit entier $n \in \mathbb{N}$ tel que $5^n \geq 10^4$; le second code réitère la demande « Tapez Q pour quitter : » jusqu'à obtenir satisfaction.

```

n=0
while 5**n < 10000:
    n=n+1
print(n)

```

```

while 1>0:
    entree=input("Tapez Q pour quitter : ")
    if entree=="Q":
        print("Fin du programme")
        break

```

3.2) Les boucles POUR

Elles s'écrivent

```
| for variable in itérable:
|     instructions
```

un **itérable** étant un objet dont on peut énumérer les éléments successifs (voir le chapitre 5). Pour le moment, on va se contenter des itérables suivants :

$(x_1, x_2, x_3, \dots)$	Ensemble des valeurs $(x_1, x_2, x_3, \dots)$
<code>range(b)</code>	Ensemble des entiers de 0 à $b - 1$.
<code>range(a, b)</code>	Ensemble des entiers de a à $b - 1$.
<code>range(a, b, c)</code>	Même chose avec un <i>pas</i> de c .

Remarques :

- ❶ On peut utiliser le compteur lors des calculs, mais **attention à ne pas le modifier!**
- ❷ Pour répéter N fois une tâche, on utilisera `for n in range(N)`
- ❸ Quand on utilise `range`, les nombres a , b et c peuvent parfaitement être négatifs.

Exemples : affichons les 20 premiers éléments de la table de 7, puis dans un second temps les cubes des 10 premiers entiers impairs.

```
| for n in range(20):
|     print("7*", n, "=", 7*n)
|
| for n in range(1, 20, 2):
|     print(n, "***3=", n**3)
```

Remarque : pour faire la même chose avec une boucle `while`, il faut s'occuper de créer, initialiser et faire varier le compteur n . C'est un peu plus fastidieux.

4/ Les assertions

Une assertion est une aide au débogage qui teste une condition. Elle s'écrit en Python

```
| assert condition, texte facultatif
```

- Si la condition est vraie, il ne se passe rien et l'exécution du programme continue.
- Si elle est fausse, l'exécution du programme s'interrompt et l'on voit apparaître un message d'erreur `AssertionError` assorti d'un texte facultatif.

Les assertions sont des aides au développement, qui n'ont pas vocation à faire partie du programme final. On les utilise quand on cherche à identifier la source d'une erreur imprévue.

Exemple : avant de calculer le logarithme de x , on pourra placer l'instruction

```
assert x>0, "x doit être strictement positif"
```

5/ Les grands classiques

5.1) Calcul d'une somme

On dispose d'une fonction f définie sur $\mathbb{N}$ et l'on désire calculer la somme

$$f(a) + f(a + 1) + \dots + f(b) = \sum_{k=a}^b f(k)$$

- ❶ On crée une variable `somme` initialement égale à 0.
- ❷ À l'aide d'une boucle `for` conjuguée à l'instruction `range()`, on ajoute à la variable `somme` les valeurs successives de $f(k)$.
- ❸ En fin de boucle, `somme` contient la valeur recherchée.

Exemple : calculons les sommes $\sum_{k=1}^{100} k^2$ et $\sum_{n=2}^{19} \frac{1}{n(n+1)}$.

```
somme=0
for k in range(1,101):
    somme=somme+k**2
print(somme)
```

```
somme=0
for n in range(2,20):
    somme=somme+1/(n*(n+1))
print(somme)
```

5.2) Calcul d'un produit

Si l'on veut calculer cette fois

$$f(a) \times f(a + 1) \times \dots \times f(b) = \prod_{k=a}^b f(k)$$

on procède quasiment pareil :

- ❶ On crée une variable `produit` initialement égale à 1.
- ❷ À l'aide d'une boucle `for` conjuguée à l'instruction `range()`, on multiplie la variable `produit` par les valeurs successives de $f(k)$.
- ❸ En fin de boucle, `produit` contient la valeur recherchée.

Exemple : calculons $100! = \prod_{k=1}^{100} k$.

```
produit=1
for k in range(1,101):
    produit=produit*k
print(produit)
```

5.3) Calcul des termes successifs d'une suite récurrente

On sera amenés à croiser principalement deux types de récurrence en informatique.

Les récurrences simples

On désire calculer les termes successifs de la suite $(u_n)_{n \in \mathbb{N}}$ définie par

$$\begin{cases} u_0 = a \\ u_{n+1} = f(u_n) \end{cases}$$

Pour cela, on va utiliser une variable u qui contiendra les valeurs successives de u_n :

- ❶ Initialement, $n = 0$ donc $u = a$.
- ❷ À l'étape n , supposons que la variable u contient la valeur u_n .
Il suffit d'effectuer l'instruction $u = f(u)$ pour qu'elle contienne u_{n+1} .

Exemple : on va afficher les 20 premiers termes de la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 2$ et $u_{n+1} = \sqrt{1 + u_n}$ de deux façons, puis simplement calculer le terme u_{15} .

<pre>u = 2 for n in range(20): print(n, ':', u) u = (1+u)**0.5</pre>	<pre>u, n = 2, 0 while n < 20: print(n, ':', u) u = (1+u)**0.5 n = n+1</pre>	<pre>u = 2 for n in range(15): u = (1+u)**0.5 print("u_15=", u)</pre>
--	---	---

Les récurrences doubles

Si la suite $(u_n)_{n \in \mathbb{N}}$ est définie par une récurrence double

$$\begin{cases} u_0 = a, u_1 = b \\ u_{n+2} = f(u_n, u_{n+1}) \end{cases}$$

il faut utiliser une seconde variable v qui contiendra les valeurs successives de u_{n+1} :

- ❶ Initialement, $n = 0$ donc $u = a$ et $v = b$.
- ❷ À l'étape n , supposons que les variables u et v contiennent les valeurs u_n et u_{n+1} .
Il faut alors affecter v à u et $f(u, v)$ à v ... en s'aidant d'une affectation multiple.

Exemple : prenons la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 1$, $u_1 = 2$ et $u_{n+2} = 3u_n - 2u_{n+1}$. On va d'abord afficher les 20 premiers termes de la suite de deux façons, puis calculer u_{15} .

<pre>u, v = 1, 2 for n in range(20): print(n, ':', u) u, v = v, 3*u-2*v</pre>	<pre>u, v, n = 1, 2, 0 while n < 20: print(n, ':', u) u, v = v, 3*u-2*v n = n+1</pre>	<pre>u, v = 1, 2 for n in range(14): u, v = v, 3*u-2*v print('u_15=', v)</pre>
---	--	--

Exercices

Exercice 1.

Donner l'état des différentes variables après l'exécution des algorithmes suivants :

$$\left| \begin{array}{l} x \leftarrow 2 \\ x \leftarrow x + 1 \end{array} \right. \quad \left| \begin{array}{l} y \leftarrow 2 \\ y \leftarrow y + 1 \\ y \leftarrow 4 \times y \end{array} \right. \quad \left| \begin{array}{l} x \leftarrow 7 \\ y \leftarrow x - 2 \\ z \leftarrow -3 \times y - 4 \end{array} \right.$$

Exercice 2.

- 1) Écrire un algorithme qui demande un nombre et renvoie sa valeur absolue.
- 2) Écrire un algorithme qui demande un nombre x et renvoie $1/(x - 1)$, si possible.
- 3) Écrire un algorithme qui demande un nombre, renvoie sa racine carrée s'il est positif et un message d'erreur sinon.

Exercice 3.

- 1) Écrire un algorithme qui affiche les 20 premiers éléments de la table de 8.
- 2) Écrire un algorithme calculant la somme des entiers de 5 à 47.

Dans chaque cas, on proposera deux versions, la première avec une boucle TANT QUE et la seconde avec une boucle POUR.

Exercice 4.

- 1) Écrire un algorithme qui, en utilisant uniquement des soustractions, trouve le reste de la division euclidienne de 583 par 25.
- 2) Compléter cet algorithme pour qu'il donne aussi le quotient.

Exercice 5.

Programmer les algorithmes des exercices 2 à 4.

Exercice 6.

- 1) Écrire un programme qui demande deux nombres et affiche le plus grand.
- 2) Faire la même chose avec trois nombres.

Exercice 7.

Écrire un programme qui détermine si un automobiliste est en excès de vitesse, connaissant sa vitesse (en $\text{km} \cdot \text{h}^{-1}$) et le type de route sur laquelle il se trouve ('ville', 'départementale', '4 voies' ou 'autoroute').

Exercice 8.

Écrire un programme qui demande l'année en cours et détermine si elle est bissextile ou non. Attention, parmi les années qui sont des multiples de 100, seules les multiples de 400 sont effectivement bissextiles.

Exercice 9.

Afficher tous les multiples de 8 (entre 0 et 100) qui ne sont pas des multiples de 5.

Exercice 10.

Afficher les 20 premiers termes de la suite réelle $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 3$ et $u_{n+1} = 2 + \frac{1}{u_n}$.

Exercice 11.

Procéder de même avec la suite de Fibonacci définie par $u_0 = 0$, $u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$.

Exercice 12.

Déterminer le rang du dernier terme strictement positif de la suite réelle $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 20\,182\,019$ et $u_{n+1} = \frac{2}{3}u_n - 3n$ pour tout $n \in \mathbb{N}$.

Exercice 13.

Demander un nombre à l'utilisateur puis déterminer si c'est une puissance de 2 ou non.

Exercice 14.

Demander à l'utilisateur d'entrer un entier, en réitérant jusqu'à ce qu'il soit pair.

Exercice 15.

Indiquer les résultats des suites d'instructions suivantes :

```
for x in range(1,11):
    if x==5:
        break
    print(x)
```

```
for x in range(1,11):
    if x==5:
        continue
    print(x)
```

Exercice 16.

On dispose d'un entier naturel n et l'on considère le programme ci-contre. Quel est l'état final des variables ? À quoi sert ce programme ?

```
c,p = n,1
while c>0:
    p=p*2
    c=c-1
```

Exercice 17.

Quel est l'effet du code suivant ?

```
for x in range(1,43):
    if 42%x != 0:
        continue
    print(x)
print("C'est fini !")
```