Flot d'exécution

Table des matières

1 Programmation procédurale

La *programmation procédurale* consiste à découper un programme en fonctions ou procédures élémentaires afin de rendre le programme modulaire. Chaque fonction a une responsabilité bien déterminée. Cela permet la réutilisation du programme ainsi défini : on dit que l'on *factorise* le code. Ainsi, il est plus facile de faire évoluer notre programme en remplaçant par exemple une fonction par une version plus efficace.

1.1 Fonction

Dans sa forme la plus simple, une fonction prend en entrée une valeur et en renvoie une autre. Par exemple, la fonction

```
def carre(n):
    return n * n
```

prend en entrée la valeur n et renvoie n^2. On utilise ensuite la fonction de la manière suivante :

```
In [1]: carre(3)
Out[1]: 9
```

Bien entendu, il est possible d'utiliser le résultat renvoyé par une fonction à l'intérieur d'une expression.

```
In [2]: carre(3) + carre(4)
Out[2]: 25
```

Une fonction peut prendre en entrée plusieurs paramètres :

```
def somme(a, b):
    return a + b
```

```
In [3]: somme(3, 5)
Out[3]: 8
```

Les fonctions que nous avons vues jusqu'à présent sont dites *pures*, dans la mesure où elles ne changent pas l'état du système.

Une fonction peut aussi ne rien renvoyer (en pratique elles renvoient `None`, mais c'est un détail que nous pouvons ignorer pour le moment). Elle fonctionne alors par effet de bord ; on dit que c'est une *procédure*. On peut par exemple afficher du texte sur la console :

```
def greetings(nom):
    print("Hello", nom)
```

```
In [4]: greetings("Paul")
Hello Paul
```

1.2 Liste

Bien que les listes n'aient pas de lien avec la programmation procédurale, nous les introduisons ici afin d'avoir des exemples plus intéressants dans la suite de ce chapitre. Une liste est une succession ordonnée de valeurs. Pour définir une liste, on énumére ses éléments entre crochets, en les séparant par des virgules. Les listes ont leur type `list` et il est possible de connaitre leur longueur à l'aide de la fonction `len`.

```
In [1]: note = [9, 10, 14]

In [2]: type(note)
Out[2]: list

In [3]: len(note)
Out[3]: 3
```

Si t est une liste de longueur n, ses valeurs sont indexées de 0 à $n-1$ et il est possible d'accéder directement à la valeur d'indice k grâce à `t[k]`. On peut imaginer que ses valeurs sont stockées dans un tableau les unes à la suite des autres : une liste peut ainsi avoir un accès direct à son k-ième élément.

```
In [4]: note[0]
Out[4]: 9

In [5]: moyenne = (note[0] + note[1] + note[2]) / len(note)

In [6]: moyenne
Out[6]: 11.0
```

Si l'on dépasse les bornes d'une liste, Python lève l'exception « list index out of range ». Par exemple `note[3]` va lever une telle exception. Même s'il est possible d'avoir des listes contenant des objets de types différents, en pratique, nous n'utiliserons que des listes constituées d'objets du même type.

Notons que les chaines de caractères ont un comportement comparable aux listes : si s est une chaine de caractères, `s[k]` permet d'accèder au caractère d'indice k. À noter que contrairement à de nombreux langages, il n'existe pas de type « caractère » et `s[k]` est tout simplement une chaine de caractères de longueur 1.

Les listes peuvent contenir d'autres listes. Par exemple, pour représenter une matrice, on utilise le plus souvent une liste formée des listes de ses vecteurs ligne. Par exemple, pour représenter la matrice

$$M \coloneqq \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix} \in \mathcal{M}_{2,3}(\mathbb{R})$$

on utilise :

```
In [7]: m = [[0, 1, 2], [3, 4, 5]]
```

On accède à l'élément $m_{i,j}$ à l'aide de `m[i][j]`. Si `m` représente une matrice à q lignes et p colonnes alors $0 \leqslant i < q$ et $0 \leqslant j < p$, contrairement à l'usage mathématique où $1 \leqslant i \leqslant q$ et $1 \leqslant j \leqslant p$. Si l'on souhaite récupérer le nombre de lignes et de colonnes, il suffit d'écrire :

```
In [8]: q = len(m)

In [9]: p = len(m[0])
```

1.3 Ordre d'évaluation

Lors de l'évaluation d'une fonction comportant des expressions comme arguments, Python évalue ces expressions avant d'appeler la fonction. Par exemple, si l'on évalue l'expression `f(2 + 3)`, Python va d'abord évaluer `2 + 3` en `5` puis appeler la fonction `f` avec l'argument `5`. Presque tous les langages de programmation fonctionnent de cette manière et seuls certains langages fonctionnels de niche comme Haskell ont un comportement différent.

Les opérateurs `and` et `or` ont la particularité de fonctionner différemment. Étant donné que `a and b` est faux dès que `a` est faux, l'opérateur `and` évalue d'abord sa première opérande. Dans le cas où celle-ci s'évalue en `False`, *la seconde opérande n'est pas évaluée* et la valeur `False` est renvoyée. On dit que l'opérateur `and` est *paresseux* (*lazy* en anglais). Cette particularité est importante, notamment lorsque l'évaluation de la seconde opérande peut provoquer une erreur si la première est fausse. Par exemple, si $x = 0$, l'expression `x != 0 and 1 / x <= 1` ne lève pas d'exception et s'évalue en `False`. De même, l'opérateur `or` évalue d'abord sa première opérande. Si le résultat de cette évaluation est

`True`, la seconde opérande n'est pas évaluée et le résultat est `True`. Si par contre, l'évaluation de la première opérande est `False`, la seconde opérande est évaluée.

Exercice 1

⇨ Quel est le résultat de l'expression `k < len(t) and t[k] == 1` si les variables `k` et `t` contiennent respectivement les valeurs 3 et `[1, 1, 0, 1]` ? Et si `k` contient la valeur 2 ? Si elle contient la valeur 4 ?

2 Programmation structurée

En programmation impérative, l'ordre dans lequel les différentes instructions sont exécutées est essentiel. Jusqu'à présent, nous avons écrit des programmes dans lesquels les instructions s'exécutaient les unes après les autres, toujours dans le même ordre. Afin de changer cet ordre, les programmes sont capables d'effectuer des sauts dans ce flot d'instructions. Les premiers langages de programmation utilisaient une instruction nommée `goto` qui leur permettait, sous condition, de sauter d'un endroit à l'autre du programme. Bien que totalement adaptée à la manière dont fonctionne un processeur, cette instruction est beaucoup trop permissive et a abouti à l'écriture de « code spaghetti », difficilement compréhensible par des humains, et donc source de nombreux bugs. Dans un célèbre article publié en 1968 sous le titre « Goto statement considered harmful », Edsger Dijkstra a plaidé pour l'utilisation essentielle d'instructions conditionnelles (`if`) et de boucles (`for`/`while`). Parallèlement, on a montré que ces structures de contrôle suffisent pour l'écriture des programmes les plus complexes. Les années 1970 donnent ainsi naissance à la *programmation structurée*.

2.1 Branchement

L'instruction `if` permet de soumettre l'exécution d'une instruction ou d'un bloc d'instructions à une condition.

```
def banquier(solde):
    if solde < 0:
        print("Vous êtes à découvert.")
        print("Veuillez passer à la banque.")
    print("Bonne journée.")
```

```
In [1]: banquier(100)
Bonne journée.

In [2]: banquier(-10)
Vous êtes à découvert.
Veuillez passer à la banque.
Bonne journée.
```

Le bloc d'instructions soumis à condition est délimité par l'*indentation*. Par rapport à l'instruction `if`, on décale d'un même nombre d'espaces chaque instruction faisant partie de ce bloc. Par convention, nous choisissons une indentation de 4 espaces. L'instruction suivant la fin du bloc doit avoir le même niveau d'indentation que l'instruction `if`. Le programme précédent vous souhaitera donc une bonne journée quel que soit l'état de votre compte.

Exercice 2

⇨ Expliquer ce que font les deux fonctions suivantes.

```
def foo(n):
    if n % 2 == 1:
        n = n - 1
    print(n)

def bar(n):
    if n % 2 == 1:
        n = n - 1
        print(n)
```

Il est possible d'exécuter un autre bloc d'instructions dans le cas où la condition n'est pas vérifiée.

```
def salutation(est_femme):
    if est_femme:
        genre = "Madame"
    else:
        genre = "Monsieur"
```

```
    return "Bonjour " + genre + "."
```

```
In [3]: salutation(True)
Out[3]: 'Bonjour Madame.'

In [4]: salutation(False)
Out[4]: 'Bonjour Monsieur.'
```

Enfin, il est possible d'exécuter différents blocs si l'on a plusieurs conditions.

```
def bac(note):
    if note >= 16:
        print("Mention Tres Bien.")
    elif note >= 14:
        print("Mention Bien.")
    elif note >= 12:
        print("Mention Assez Bien.")
    elif note >= 10:
        print("Vous avez votre Bac.")
    else:
        print("Same player shoot again!")
```

```
In [5]: bac(13)
Mention Assez Bien.
```

Dans ce cas, seul le bloc correspondant à la première condition qui est vraie est exécuté.

Exercice 3

⇨ Une agence de voyages propose un voyage organisé où l'on peut s'inscrire en groupe. Le prix par personne est dégressif selon le nombre de personnes : 80 euros pour une ou deux personnes, 70 euros pour 3 à 5 personnes, 60 euros pour 6 à 9 personnes et 50 euros à partir de 10 personnes. On souhaite écrire une fonction ayant pour argument le nombre n de personnes et renvoyant le prix total pour l'ensemble du groupe.

1. Écrire une fonction qui effectue au plus 3 comparaisons à chaque exécution.
2. Écrire une nouvelle fonction qui effectue au plus 2 comparaisons.

2.2 Boucle for

Il est possible de répéter plusieurs fois la même séquence d'instructions en utilisant une boucle `for`. Comme pour l'instruction `if`, le bloc d'instructions à exécuter dans la boucle est indenté. La première instruction ne faisant pas partie de la boucle doit utiliser le même niveau d'indentation que la ligne du `for`.

```
def the_shining(n):
    for _ in range(n):
        print("All work and no play")
        print("makes Jack a dull boy.")
    print("Jack Torrance")
```

```
In [1]: the_shining(3)
All work and no play
makes Jack a dull boy.
All work and no play
makes Jack a dull boy.
All work and no play
makes Jack a dull boy.
Jack Torrance
```

Comme le nombre de fois où le corps de la boucle s'exécute est connu avant de rentrer dans la boucle, on parle de boucle *inconditionnelle*. En particulier, nous sommes certains d'en sortir avant même d'y rentrer ; on dit qu'elles sont *bornées*.

Pour calculer le n-ième terme de la suite (u_n) définie par

$$u_0 \coloneqq \alpha \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad u_{n+1} \coloneqq \cos(u_n)$$

on peut utiliser le programme suivant :

```
import math

def suite(alpha, n):
    u = alpha
    for _ in range(n):
        u = math.cos(u)
    return u
```

```
In [1]: suite(1.0, 1)
Out[1]: 0.5403023058681398

In [2]: suite(1.0, 10)
Out[2]: 0.7442373549005569

In [3]: suite(1.0, 100)
Out[3]: 0.7390851332151608
```

Exercice 4

⇨ On définit la suite de Fibonacci par

$$F_0 \coloneqq 0, \quad F_1 \coloneqq 1, \quad \text{et} \quad \forall n \in \mathbb{N}, \quad F_{n+2} \coloneqq F_{n+1} + F_n.$$

Écrire une fonction `fibo(n)` renvoyant F_n. Notre fonction pourra utiliser deux variables `a` et `b` contenant respectivement les valeurs F_k et F_{k+1}.

Il est souvent utile d'avoir une variable prenant des valeurs entières successives lors d'une boucle. Ainsi, dans le programme suivant, la variable `k` va prendre successivement les 10 valeurs : $0, 1, 2, 3, \ldots, 9$.

```
def table(n):
    for k in range(10):
        print(k, "*", n, "=", k * n)
```

```
In [4]: table(8)
0 * 8 = 0
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
```

Remarquons que dans les exemples précédents, `_` désigne un nom de variable qu'il est coutume d'utiliser en Python lorsque sa valeur ne nous est pas utile.

Les boucles `for` sont très utiles pour calculer des sommes. Par exemple, pour calculer les premiers termes de la suite (u_n) définie par

$$\forall n \in \mathbb{N}, \quad u_n \coloneqq \sum_{k=1}^{n} \frac{1}{k^2},$$

on utilise la fonction suivante :

```
def suite(n):
    s = 0.0
    for k in range(1, n + 1):
        s = s + 1 / k**2
    return s
```

Dans cette fonction, on dit que `s` est un *accumulateur*.

```
In [1]: suite(1)
Out[1]: 1.0
```

```
In [2]: suite(10)
Out[2]: 1.5497677311665408

In [3]: suite(100)
Our[3]: 1.6349839001848923
```

De manière générale, la boucle `for k in range(a, b)` permet à la variable `k` de prendre successivement les valeurs $a, a+1, \dots, b-1$. Dans notre cas, k va prendre les valeurs $1, 2, \dots, n$ pour ajouter les valeurs $1, 1/2^2, \dots, 1/n^2$ à s.

Plus généralement, si $\delta > 0$, `range(a, b, delta)` est utilisé pour boucler sur les entiers $a, a+\delta, a+2\delta$ jusqu'au plus grand entier de la forme $a + k\delta$ strictement inférieur à b.

Exercices 5

⇨ Donner dans chacun des cas suivants les valeurs générées par le `range` :

1. `range(7)`
2. `range(2, 5)`
3. `range(3, 7, 2)`

⇨ Écrire une fonction permettant de calculer la somme de tous les nombres impairs entre 1 et n inclus.

⇨ Écrire une fonction permettant de calculer $n!$.

2.3 Réduction

Si l'on souhaite calculer la somme des éléments d'une liste d'entiers a, on peut initialiser une variable `acc` à 0 et lui ajouter successivement tous les éléments de a. On obtient alors la fonction :

```
1 def somme(a):
2     acc = 0
3     for i in range(len(a)):
4         acc = acc + a[i]
5     return acc
```

Pour prouver que cette fonction nous renvoie bien la somme des éléments de a, on définit, pour tout $i \in [\![0, n]\!]$

$$\mathcal{H}_i \coloneqq \text{« La variable } \texttt{acc} \text{ contient la valeur } \sum_{k=0}^{i-1} a_k \text{. »}$$

$\mathcal{H}_0$ est vraie avant de rentrer dans la boucle, et si $\mathcal{H}_i$ est vraie au début du corps de la boucle, alors $\mathcal{H}_{i+1}$ est vraie à la fin du corps de la boucle. Cela prouve que $\mathcal{H}_n$ est vraie en sortie de boucle et donc que la fonction renvoie bien la valeur souhaitée

$$\sum_{k=0}^{n-1} a_k.$$

On dit que $\mathcal{H}_i$ est un *invariant de boucle*.

On peut de même écrire une fonction calculant le produit des éléments d'une liste. Cette fois, la variable `prod` est initialisée à 1. En effet, 0 était l'élément neutre pour l'addition, puisque pour tout $v \in \mathbb{N}$, $0 + v = v$. Son équivalent pour la multiplication est 1, puisque pour tout $v \in \mathbb{N}$, $1 \times v = v$. On écrit donc :

```
1 def produit(a):
2     prod = 1
3     for i in range(len(a)):
4         prod = prod * a[i]
5     return prod
```

Dans ce cas, l'invariant de boucle est

$$\mathcal{H}_i \coloneqq \text{« La variable } \texttt{prod} \text{ contient la valeur } \prod_{k=0}^{i-1} a_k \text{. »}$$

et prouve que la fonction renvoie bien le produit des éléments de a.

Exercice 6

⇨ Écrire une fonction prenant en entrée une liste de booléens et renvoyant `True` si tous ces booléens sont égaux à `True`, et `False` sinon.

De la même manière, on peut écrire une fonction calculant le plus grand élément d'une liste non vide d'entiers.

```
def maximum(a):
    v_max = a[0]
    for i in range(1, len(a)):
        v_max = max(v_max, a[i])
    return v_max
```

On peut aussi adapter le programme afin qu'il nous renvoie l'indice de ce maximum :

```
def indice_maximum(a):
    v_max = a[0]
    i_max = 0
    for i in range(1, len(a)):
        if a[i] > v_max:
            v_max = a[i]
            i_max = i
    return i_max
```

2.4 Boucle while

Les boucles `for` nous ont permis d'exécuter plusieurs fois un bloc d'instructions dans le cas où le nombre d'itérations est connu avant de rentrer dans la boucle. Lorsque ce nombre n'est pas connu à priori, typiquement lorsque l'on doit exécuter un bloc tant qu'une condition est vérifiée, on utilise l'instruction `while`.

Supposons que l'on souhaite calculer la racine carrée entière de $n \in \mathbb{N}$, c'est-à-dire le plus grand $a \in \mathbb{N}$ tel que $a^2 \leqslant n < (a+1)^2$. Autrement dit, on souhaite calculer $\lfloor \sqrt{n} \rfloor$, mais sans utiliser de nombre flottant. Pour cela, on initialise a à 0 et on l'incrémente de 1 tant que $a^2 \leqslant n$. Dès que ce n'est plus le cas, on renvoie $a-1$ qui est la valeur cherchée. On obtient ainsi le code :

```
def int_sqrt(n):
    a = 0
    while a * a <= n:
        a = a + 1
    return a - 1
```

```
In [1]: int_sqrt(15)
Out[1]: 3
```

Exercices 7

⇨ Après avoir remarqué que

$$ne^{-n} \xrightarrow[n\to+\infty]{} 0,$$

écrire un programme prenant en entrée $\varepsilon > 0$ et permettant de trouver le plus petit entier $n \in \mathbb{N}^*$ tel que $ne^{-n} \leqslant \varepsilon$.

⇨ Montrer comment une boucle `for`

```
for k in range(a, b):
    bloc.....................
    ..........d'instructions
```

peut s'écrire à l'aide d'une boucle `while`.

⇨ Le but de cet exercice est d'écrire une fonction prenant en entrée une liste de booléens et renvoyant `True` si un de ces booléens est `True`. Elle doit renvoyer `False` sinon.

1. Écrire une première version de cette fonction en utilisant une boucle `for`.
2. La version précédente a le défaut de parcourir toute la liste, même si elle trouve le booléen `True` dans les premiers éléments. On souhaite donc écrire une nouvelle version qui, dès qu'elle découvre un `True`, arrête le parcours. Pour cela, compléter la fonction suivante :

```
def possede_un_true(a):
```

```
    n = len(a)
    k = 0
    while k < n and ...:
        k = k + 1
    return ...
```

Contrairement aux boucles inconditionnelles pour lesquelles on est assuré de sortir de la boucle, il est possible qu'une boucle conditionnelle ne termine jamais. On dit alors que le programme part en *boucle infinie*.

```
def un_jour_sans_fin():
    while True:
        print("This is Groundhog day!")
```

```
In [2]: un_jour_sans_fin()
This is Groundhog day!
This is Groundhog day!
This is Groundhog day!
...
```

Vous pouvez interrompre un tel programme en appuyant à la fois sur la touche « Ctrl » et la touche « c ».

Une boucle `while` a donc le défaut de ne pas être assurée de terminer. Il est cependant essentiel de pouvoir prouver que dans les conditions normales d'exécution, votre boucle se termine bien. Pour cela, on cherche souvent une grandeur entière positive qui diminue strictement à chaque itération. Comme il n'existe pas de suite infinie strictement décroissante d'entiers positifs, on aura ainsi prouvé que la boucle termine. Une telle grandeur est appelé un *variant*.

Le calcul du pgcd par l'algorithme d'Euclide est basé sur le principe suivant : si $a, b \in \mathbb{N}$, le pgcd de a et b est a lorsque $b = 0$ et est égal au pgcd de b et du reste de la division euclidienne de a par b lorsque $b > 0$. Le programme suivant permet donc de calculer ce pgcd.

```
def pgcd(a, b):
    while b > 0:
        a, b = b, a % b
    return a
```

```
In [3]: pgcd(15, 21)
Out[3]: 3
```

Exercice 8

⇨ Prouver que le programme précédent termine.

2.5 Boucles imbriquées

Il est possible d'imbriquer les boucles les unes dans les autres. Vous pouvez par exemple générer les tables de multiplication très facilement de la manière suivante :

```
def tables(n):
    for a in range(2, n + 1):
        for b in range(2, n + 1):
            print(a, "*", b, "=", a * b)
```

```
In [1]: tables(4)
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
```

Comme les produits `2 * 3` et `3 * 2` sont égaux, ont peut chercher à limiter ces produits aux cas où $a \leqslant b$. On écrit alors :

```
def tables_bis(n):
    for a in range(2, n + 1):
        for b in range(a, n + 1):
            print(a, "*", b, "=", a * b)
```

```
In [2]: tables_bis(4)
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 3 = 9
3 * 4 = 12
4 * 4 = 16
```

Les boucles imbriquées nous seront utiles pour parcourir les éléments d'une matrice. Notons au passage, qu'il est possible de mélanger les boucles `for` et `while`. Supposons par exemple qu'étant donnée une matrice de 0 et de 1, on souhaite calculer le nombre de lignes possèdant au moins un 1.

```
def nb_lignes_avec_un(m):
    q = len(m)
    p = len(m[0])
    nb = 0
    for i in range(q):
        j = 0
        while j < p and m[i][j] != 1:
            j = j + 1
        if j < p:
            nb = nb + 1
    return nb
```

Exercice 9

⇨ Écrire une fonction prenant en entrée une matrice de 0 et de 1 et renvoyant l'indice d'une des lignes possédant le plus de 1.