3

Contrôle du flux d'exécution

Dans notre premier chapitre, nous avons vu que l'activité essentielle d'un programmeur est la résolution de problèmes. Or, pour résoudre un problème informatique, il faut toujours effectuer une série d'actions dans un certain ordre. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un ***algorithme****.*

Le « chemin » suivi par Python à travers un programme est appelé un ***flux d'exécution****, et les constructions qui le modifient sont appelées des* ***instructions de contrôle de flux****.*

Les ***structures de contrôle*** *sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois : la séquence*[10] *et la sélection, que nous allons décrire dans ce chapitre, et la répétition que nous aborderons au chapitre suivant.*

Séquence d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, *dans l'ordre où elles ont été écrites à l'intérieur du script*.

Cette affirmation peut vous paraître banale et évidente à première vue. L'expérience montre cependant qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres. Par exemple, dans la séquence d'instructions suivantes :

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
>>> print(a, b)
```

Vous obtiendrez un résultat contraire si vous intervertissez les 2e et 3e lignes.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une *instruction conditionnelle* comme l'instruction **if** décrite ci-après (nous en rencontrerons d'autres plus loin, notamment à propos des boucles de répétition). Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

---

[10] Tel qu'il est utilisé ici, le terme de ***séquence*** désigne donc une série d'instructions qui se suivent. Nous préférerons dans la suite de cet ouvrage réserver ce terme à un concept Python précis, lequel englobe les ***chaînes de caractères***, les ***tuples*** et les ***listes*** (voir plus loin).

Sélection ou exécution conditionnelle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de *tester une certaine condition* et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction **if**. Pour expérimenter son fonctionnement, veuillez entrer dans votre éditeur Python les deux lignes suivantes :

```
>>> a = 150
>>> if (a > 100):
...
```

La première commande affecte la valeur 150 à la variable **a**. Jusqu'ici rien de nouveau.
Lorsque vous finissez d'entrer la seconde ligne, par contre, vous constatez que Python réagit d'une nouvelle manière. En effet, et à moins que vous n'ayez oublié le caractère « : » à la fin de la ligne, vous constatez que le *prompt principal* (**>>>**) est maintenant remplacé par un *prompt secondaire* constitué de trois points[11].

Si votre éditeur ne le fait pas automatiquement, vous devez à présent effectuer une tabulation (ou entrer 4 espaces) avant d'entrer la ligne suivante, de manière à ce que celle-ci soit *indentée* (c'est-à-dire en retrait) par rapport à la précédente. Votre écran devrait se présenter maintenant comme suit :

```
>>> a = 150
>>> if (a > 100):
...     print("a dépasse la centaine")
...
```

Frappez encore une fois <*Enter*>. Le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec `a = 20` en guise de première ligne : cette fois Python n'affiche plus rien.

L'expression que vous avez placée entre parenthèses est ce que nous appellerons désormais une *condition*. L'instruction **if** permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons *indentée* après le « : » est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici avec l'instruction if sont optionnelles : nous les avons utilisées pour améliorer la lisibilité. Dans d'autres langages, il se peut qu'elles soient obligatoires.

Recommencez encore, en ajoutant deux lignes comme indiqué ci-dessous. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

```
>>> a = 20
>>> if (a > 100):
...     print("a dépasse la centaine")
... else:
```

[11] Dans certaines versions de l'éditeur Python pour ***Windows***, le prompt secondaire n'apparaît pas.

```
...     print("a ne dépasse pas cent")
...
```

Frappez <*Enter*> encore une fois. Le programme s'exécute, et affiche cette fois :

```
a ne dépasse pas cent
```

Comme vous l'aurez certainement déjà compris, l'instruction **else** (« sinon », en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de « else if ») :

```
>>> a = 0
>>> if a > 0 :
...     print("a est positif")
... elif a < 0 :
...     print("a est négatif")
... else:
...     print("a est nul")
...
```

Opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les *opérateurs de comparaison* suivants :

```
x == y       # x est égal à y
x != y       # x est différent de y
x > y        # x est plus grand que y
x < y        # x est plus petit que y
x >= y       # x est plus grand que, ou égal à y
x <= y       # x est plus petit que, ou égal à y
```

Exemple

```
>>> a = 7
>>> if (a % 2 == 0):
...     print("a est pair")
...     print("parce que le reste de sa division par 2 est nul")
... else:
...     print("a est impair")
...
```

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes « égale » et non d'un seul[12]. Le signe « égale » utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison. Vous retrouverez le même symbolisme en *C++* et en *Java*.

Instructions composées – blocs d'instructions

La construction que vous avez utilisée avec l'instruction **if** est votre premier exemple d'**instruction composée**. Vous en rencontrerez bientôt d'autres. Sous Python, les instructions composées

[12] Rappel : l'opérateur % est l'opérateur *modulo* : il calcule le reste d'une division entière. Ainsi par exemple, **a % 2** fournit le reste de la division de **a** par 2.

ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête. Exemple :

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau* (comptez un décalage de 4 caractères, par exemple). Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple du paragraphe précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction **if** constituent un même bloc logique : ces deux lignes ne sont exécutées – toutes les deux – que si la condition testée avec l'instruction **if** se révèle vraie, c'est-à-dire si le reste de la division de **a** par 2 est nul.

Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if embranchement == "vertébrés":                    # 1
    if classe == "mammifères":                      # 2
        if ordre == "carnivores":                   # 3
            if famille == "félins":                 # 4
                print("c'est peut-être un chat")    # 5
        print("c'est en tous cas un mammifère")     # 6
    elif classe == "oiseaux":                       # 7
        print("c'est peut-être un canari")          # 8
print("la classification des animaux est complexe") # 9
```

Analysez cet exemple. Ce fragment de programme n'imprime la phrase « c'est peut-être un chat » que dans le cas où les quatre premières conditions testées sont vraies.

Pour que la phrase « c'est en tous cas un mammifère » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 4) se trouve en effet au même niveau d'indentation que l'instruction : **if ordre == "carnivores"**: (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 et 2 sont vraies.

Pour que la phrase « c'est peut-être un canari » soit affichée, il faut que la variable **embranchement** contienne « vertébrés », et que la variable **classe** contienne « oiseaux ».

Quant à la phrase de la ligne 9, elle est affichée dans tous les cas, parce qu'elle fait partie du même bloc d'instructions que la ligne 1.

Quelques règles de syntaxe Python

Tout ce qui précède nous amène à faire le point sur quelques règles de syntaxe :

Les limites des instructions et des blocs sont définies par la mise en page

Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de fin de ligne[13] qui joue ce rôle. (Nous verrons plus loin comment outrepasser cette règle pour étendre une instruction complexe sur plusieurs lignes.) On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial **#**. Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur.

Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que **begin** et **end**). En *C++* et en *Java*, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper ni d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire pour les pauvres humains que nous sommes. On conseille donc à tous les programmeurs qui utilisent ces langages de se servir *aussi* des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

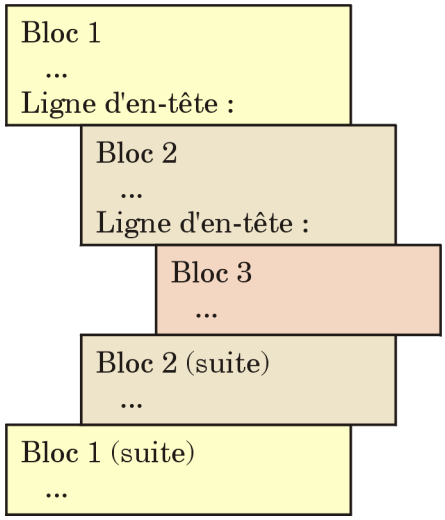
Avec Python, vous *devez* utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

Instruction composée : en-tête, double point, bloc d'instructions indenté

Nous aurons de nombreuses occasions d'approfondir le concept de « bloc d'instructions » et de faire des exercices à ce sujet dès le chapitre suivant.

Le schéma ci-contre en résume le principe.

- Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, etc.) *se terminant par un double point*.
- Les blocs sont *délimités par l'indentation* : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c'est-à-dire décalées vers la droite d'un même nombre d'espaces). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.
- Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (il n'est imbriqué dans rien).



[13] Ce caractère n'apparaît ni à l'écran, ni sur les listings imprimés. Il est cependant bien présent, à un point tel qu'il fait même problème dans certains cas, parce qu'il n'est pas encodé de la même manière par tous les systèmes d'exploitation. Nous en reparlerons plus loin, à l'occasion de notre étude des fichiers texte (page 116).

> ***Important :*** *vous pouvez aussi indenter à l'aide de tabulations, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, et même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.*
> *En conséquence, la plupart des programmeurs préfèrent se passer des tabulations. Si vous utilisez un éditeur de texte « intelligent », vous avez tout intérêt à activer son option « Remplacer les tabulations par des espaces ».*

Les espaces et les commentaires sont normalement ignorés

À part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés (sauf s'ils font partie d'une chaîne de caractères). Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante.