2 Structures de contrôle

Dans les chapitres précédents, vous avez pu constater qu'un script Python était lu ligne par ligne, les instructions étant alors interprétées les unes après les autres dans l'ordre où elles apparaissent dans le fichier. Les *structures de contrôle* permettent de s'éloigner de ce comportement linéaire. Dans ce chapitre, nous verrons les deux structures de contrôles suivantes :

— les structures conditionnelles, qui permettent d'exécuter une partie du programme seulement si une condition donnée est vérifiée,

— les boucles, qui permettent de répéter une partie du programme un nombre variable de fois, ou bien jusqu'à ce qu'une condition ne soit plus vérifiée.

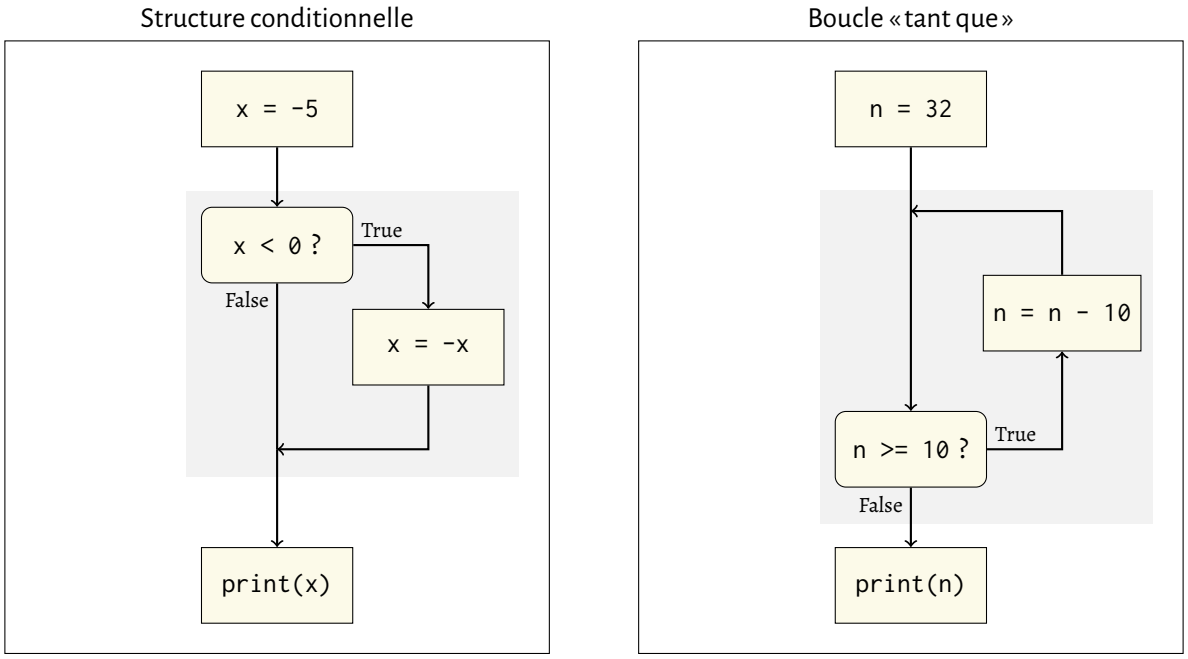

FIGURE 2.1 – Illustration des structures de contrôle élémentaires.

2.1 Structures conditionnelles

2.1.1 Forme minimale

La syntaxe minimale pour une structure conditionnelle en Python est la suivante :

```
if expr:
    instr1
    instr2
    ...
```

avec `expr` une expression renvoyant une valeur booléenne, et `instr1`, `instr2`, … des instructions qui ne seront exécutées que si l'expression `expr` renvoie `True`. Par exemple, l'algorithme illustré dans la partie gauche de la figure 2.1 s'écrit en Python de la manière suivante.

```
1 x = -5
2 if x < 0:
3     x = -x
4 print(x)
```

À tester. Testez le code précédent avec différentes valeurs initiales pour x.

Notez que la ligne 3 est *indentée*, c'est-à-dire qu'elle commence par plusieurs espaces ou caractères « tabulation »[1], ce qui fait qu'elle est visuellement décalée du reste du programme. En Python, l'indentation est primordiale car c'est elle qui permet de déterminer ou non l'appartenance à une structure de contrôle. Ici, l'indentation de la ligne 3 indique que celle-ci se trouve dans la structure « if » initiée à la ligne 2, tandis que l'absence d'indentation de la ligne 4 indique que cette dernière se situe en dehors de la structure conditionnelle, après celle-ci.

À tester. Comparez les effets des codes suivants (pour différentes valeurs initiales de x).

```
x = -5
if x < 0:
    x = -x
print(x)
print("Fin")
```

```
x = -5
if x < 0:
    x = -x
    print(x)
print("Fin")
```

2.1.2 Blocs « sinon, si »

Le langage Python propose aussi le mot-clé `elif` (contraction de « *else if* ») permettant des syntaxes telles que celle ci-dessous.

```
if expr1:
    instr1
elif expr2:
    instr2
elif expr3:
    instr3
...
```

Ici, `expr1`, `expr2` et `expr3` sont des expressions renvoyant des valeurs booléennes, et :

— `instr1` est une instruction (ou une suite d'instructions) qui sera exécutée seulement si l'expression `expr1` renvoie la valeur `True` ,

— `instr2` est une instruction (ou une suite d'instructions) qui sera exécutée seulement si l'expression `expr1` a renvoyé `False` et l'expression `expr2` renvoie `True` ,

— `instr3` est une instruction (ou une suite d'instructions) qui sera exécutée seulement si les expressions `expr1` et `expr2` ont renvoyé `False` et l'expression `expr3` renvoie `True` .

Notez que l'on peut ajouter de la sorte autant de blocs « elif » que l'on souhaite.

1. Dans un souci d'homogénéité lorsque vous partagez vos codes avec d'autres programmeurs, le PEP 8 recommande d'utiliser quatre espaces : `https://www.python.org/dev/peps/pep-0008/`

À tester. Comparez les effets des codes suivants (pour différentes valeurs initiales de n).

```
n = 8
if n % 4 == 0:
    print("n est divisible par 4")
elif n % 2 == 0:
    print("n est divisible par 2")
```

```
n = 8
if n % 4 == 0:
    print("n est divisible par 4")
if n % 2 == 0:
    print("n est divisible par 2")
```

Remarque. A noter que l'expression booléenne donnée après un `elif` n'est évaluée que lorsque c'est nécessaire (c'est-à-dire si les expressions booléennes utilisées dans le `if` et les éventuels `elif` précédents étaient toutes fausses). Par exemple, vous pouvez vérifier que le code suivant est interprété sans erreur en initialisant n à la valeur 2, mais produirait une erreur si l'on initialisait n à n'importe quelle autre valeur.

```
n = 2
if n + 1 == 3:
    print("A")
elif n / 0 == 2:
    print("B")
```

2.1.3 Bloc « sinon »

Enfin, le mot-clé `else` permet de rajouter un bloc d'instructions qui ne seront exécutées que si tous les tests précédents (`if` et éventuels `elif`) ont échoué au sein de la structure conditionnelle en cours. Par exemple :

```
n = 15
if n % 2 == 0:
    print("n est pair")
else:
    print("n est impair")
```

Le bloc « else » est facultatif, mais s'il est présent il doit toujours être placé en dernier dans la structure conditionnelle if/elif/else.

2.2 Boucles « tant que »

Une boucle « tant que » s'écrit en Python de la manière suivante :

```
while expr:
    instr1
    instr2
    ...
```

avec `expr` une expression renvoyant une valeur booléenne, et `instr1`, `instr2`, ... des instructions qui seront répétées en boucle tant que l'expression `expr` renvoie la valeur `True`. Par exemple, l'algorithme illustré dans la partie droite de la figure 2.1 s'écrit en Python de la manière suivante.

```
n = 32
while n >= 10:
    n = n - 10
print(n)
```

À tester. Exécutez le code suivant pour différentes valeurs initiales de `n`.

```
n = 32
while n >= 10:
    n = n - 10
    print("n vaut maintenant " + str(n))
print("Fin de la boucle")
print("n vaut finalement " + str(n))
```

Attention. Lorsqu'on utilise une boucle « tant que », il faut s'assurer que l'expression placée après le mot clé `while` finira par renvoyer la valeur `False`. Sinon, le programme ne se terminera jamais en théorie et on dit alors qu'il est entré dans une *boucle infinie*.

Exercice 2.1. Parmi les programmes suivants, lesquels se terminent (c'est-à-dire ne rentrent pas dans une boucle infinie) ?

Programme 1 :

```
x = 22
while x > 0:
    x = x - 5
```

Programme 2 :

```
x = 22
while x != 0:
    x = x - 5
```

Programme 3 :

```
x = -17
while x > 0:
    x = x - 5
```

Programme 4 :

```
x = 22
while x != 0:
    x = x - 1
```

Programme 5 :

```
x = -17
while x != 0:
    x = x - 1
```

Programme 6 :

```
x = -17
while x < 0:
    x = x + 2
```

Exercice 2.2. On suppose déjà définie une variable `M` contenant un entier positif. Écrire un programme Python qui affiche (ligne par ligne) tous les multiples de 3 compris entre 0 et M (éventuellement inclus).

Exemples.

— *Si M vaut 10, alors votre programme devra afficher :*

```
0
3
6
9
```

— *Si M vaut 15, alors votre programme devra afficher :*

```
0
3
6
9
12
15
```

2.3 Itérables et boucles « for »

2.3.1 Itération sur une chaîne de caractères

Itération via les indices

Le type `string` est *indiçable* [2]. Cela signifie que si `S` est une chaîne de caractères de taille n, alors on peut utiliser la syntaxe `S[i]` pour accéder à l'élément d'indice i de la chaîne `S`, où i est un entier compris entre 0 et $n-1$.

À tester.

1. Éxécutez le code suivant :

```
word = "Bonjour"
letter = word[0]
print(letter)
print(word[2])
print(word[3])
print(word[6])
```

   Que se passe-t-il si vous exécutez l'instruction `print(word[7])` ?

2. Analysez le message d'erreur produit par le programme ci-dessous.

```
a = 3
print(a[0])
```

   Parmi les types vus dans le chapitre précédent (tableau 1.1), lesquels sont indiçables (*subscriptable*) ?

Attention. Rappelez-vous bien que les éléments sont indexés à partir de 0 et non pas de 1. Cela implique que, par exemple, une chaîne formée de 7 caractères aura ses éléments indexés de 0 jusqu'à 6 (il n'existe pas d'élément d'indice 7).

Remarque. La fonction `len` renvoie la taille d'un objet indiçable (donc, par exemple, le nombre de caractères d'une chaîne de caractères).

L'avantage des types indiçables est que l'on peut très facilement parcourir leurs éléments à l'aide d'une boucle « tant que ». Par exemple, pour obtenir tous les caractères (un par un) d'une chaîne de caractères, on peut utiliser un programme de la forme suivante.

```
word = "Bonjour"
i = 0
while i < len(word):
    letter = word[i]
    print(letter)
    i = i + 1
```

2. On utilisera souvent le mot anglais *indexable*, ou encore son synonyme *subscriptable*.

Itération sur les valeurs directement

En Python, une variable indiçable est automatiquement *itérable*, c'est-à-dire que l'on peut encore plus simplement parcourir tous ses éléments à l'aide de l'opérateur `for`. Concrètement, pour une chaîne de caractères, l'opérateur `for` s'utilise de la manière suivante :

```
word = "Bonjour"
for letter in word:
    print(letter)
```

Vous pouvez remarquer que ce programme donne exactement le même résultat que le programme précédent reposant sur l'opérateur `while`, mais en utilisant une syntaxe plus simple et plus courte. On privilégiera donc l'utilisation de `for` lorsqu'on voudra seulement accéder à tous les caractères d'une chaîne dans l'ordre où ils apparaissent, mais il faudra savoir utiliser `while` pour des itérations plus subtiles.

Exercice 2.3. On suppose déjà définie une variable `mot` contenant une chaîne de caractères.

1. Écrire un programme Python qui affiche (ligne par ligne) les caractères de la chaîne `mot` dans l'ordre inverse où ils apparaîssent dans la chaîne,
   Exemple. Si `mot` *contient la chaîne* `"Bonjour"`*, alors votre programme devra afficher :*

   ```
   r
   u
   o
   j
   n
   o
   B
   ```

2. Écrire un programme Python qui affiche (ligne par ligne) seulement un caractère sur deux de la chaîne `mot`, dans l'ordre où ils apparaîssent dans la chaîne et en commençant par le premier.
   Exemple. Si `mot` *contient la chaîne* `"Bonjour"`*, alors votre programme devra afficher :*

   ```
   B
   n
   o
   r
   ```

2.3.2 Itération sur un objet « range »

Un autre type itérable en Python est le type *range* utilisé pour représenter une progression arithmétique d'entiers. On construit un objet de type `range` à l'aide de la fonction du même nom :

— **avec un seul paramètre :** l'instruction `range(n)` renvoie un objet de type `range` représentant la suite $0, 1, \ldots n-1$ formée des entiers consécutifs de 0 (inclus) jusqu'à n (exclus),

— **avec deux paramètres :** l'instruction `range(a, b)` renvoie un objet de type `range` représentant la suite $a, a+1, \ldots b-1$ formée des entiers consécutifs de a (inclus) jusqu'à b (exclus),

— **avec trois paramètres :** l'instruction `range(a, b, h)` renvoie un objet de type `range` représentant la suite $a, a+h, a+2h, \ldots$ formée des entiers de a (inclus) jusqu'à b (exclus) avec un pas égal à h.

À tester.

1. Observez le résultat produit par le programme suivant. Quel est le type de `T` ?

```
T = range(8)
for x in T:
    print(x)
```

2. Analysez les résultats produits par chacun des programmes suivants.

```
for a in range(2, 10):
    print(a)
```

```
for a in range(10, 2):
    print(a)
```

```
for num in range(0, 32, 5):
    print(num)
```

```
for x in range(8, 3, -1):
    print(x)
```

Exercice 2.4. Reprenez l'exercice 2.3 en remplaçant l'utilisation de `while` par une utilisation de `for` et `range`.

2.4 Algorithmes classiques

2.4.1 Compter ou sommer

Lorsqu'un programme informatique a pour but de *compter* (ou *sommer*) un nombre de choses, il est fréquent de rencontrer le schéma suivant :

1. une variable servant de compteur est créée et initialisée à la valeur zéro,
2. une boucle (`for` ou `while`) est exécutée et ajoute occasionnellement 1 (ou plus) à cette variable.

Par exemple, le programme suivant compte le nombre de « e » présents dans la chaîne de caractères stockée dans la variable `msg`.

```
msg = "Ceci est une petite phrase d'exemple."
n = 0
for c in msg:
    if c == "e":
        n = n + 1
print("La chaine contient " + str(n) + " fois la lettre e")
```

Attention. Ce programme ne compte que les lettres « e » minuscules sans accent, car en Python `"e"`, `"E"`, `"é"`, `"è"`, `"ê"`, … sont tous des caractères différents.

À tester. Reprenez le programme ci-dessus en initialisant la chaîne `msg` aux valeurs suivantes :

1. `"Un autre test."`
2. `"Est-ce que toutes les lettres ont bien été comptées ?."`
3. `"Son pouls battait trop fort. Il avait chaud. Il ouvrit son vasistas, scruta la nuit."` [3]

3. *La Disparition*, Georges Perec (1969).

Exercice 2.5.

1. Écrire un programme qui affiche (ligne par ligne) tous les nombres entiers entre 1 et 20 (inclus) qui sont divisibles par 2 ou par 3.
2. Écrire un programme qui compte combien d'entiers entre 1 et 20 sont divisibles par 2 ou par 3.
3. Écrire un programme qui calcule la somme des entiers entre 1 et 20 divisibles par 2 ou par 3.

2.4.2 Vérifier l'existence (ou non) de quelque chose

Premières pistes

Mettons que l'on veuille maintenant écrire un programme qui détermine si une chaîne de caractère stockée dans la variable `msg` contient ou non le caractère `"e"`. Une première approche pourrait être de reprendre le programme précédent (celui qui compte le nombre d'apparitions de `"e"`) et de conclure suivant si la valeur trouvée est nulle ou strictement positive. Ceci donnerait un code comme suit.

```
msg = "Ceci est une petite phrase d'exemple."
n = 0
for c in msg:
    if c == "e":
        n = n + 1
print("La lettre e est " + ("présente" if n > 0 else "absente"))
```

Ce code renvoie bien le résultat voulu (et ce quelque soit la valeur initiale de la variable `msg`), mais il n'est pas satisfaisant car il demande à l'ordinateur une tâche *plus précise* que ce qui est attendu. En effet, ce programme impose à l'ordinateur de calculer et stocker le nombre d'apparitions de `"e"` alors qu'en réalité on se pose une question ayant seulement deux réponses possibles : ou bien la lettre est là, ou bien elle n'est pas là. Autrement dit, l'algorithme ne devrait pas avoir à stocker un *entier* mais seulement un *booléen*. Voici donc une nouvelle proposition d'algorithme dans lequel la variable `n` a été remplacée par une variable de type `bool`.

```
msg = "Ceci est une petite phrase d'exemple."
is_found = False
for c in msg:
    if c == "e":
        is_found = True
print("La lettre e est " + ("présente" if is_found else "absente"))
```

Remarque. Cette modification peut paraître superflue lorsqu'on travaille sur un ordinateur personnel car de nos jours ces derniers possèdent une capacité mémoire largement suffisante pour la plupart des programmes que vous écrirez. Mais si jamais vous travailliez sur un système embarqué que l'on cherchait à équiper du moins de mémoire vive possible (pour des raisons de coût, de masse, de consommation, ...), ce genre d'optimisation pourrait s'avérer cruciale car un booléen occupe moins de place en mémoire vive qu'un entier.

Au-delà de cette raison très technique, c'est par ailleurs une bonne habitude à prendre que de toujours choisir le type de variable le plus adapté à l'information que vous souhaitez stocker, et il ne faut surtout pas oublier l'existence du type booléen qui se prête parfaitement à la représentation d'informations du type « oui ou non ».

Toutefois, le code Python proposé est encore perfectible. En effet, une fois que le caractère `"e"` a été trouvé, il est inutile de poursuivre le parcours de la chaîne de caractères `msg` car on peut d'ores et déjà répondre à la question posée. Ceci nous mène à la section suivante.

Arrêt prématuré

Rappelons tout d'abord que le programme précédent pourrait être réécrit en parcourant la chaîne `msg` à l'aide d'une boucle `while` plutôt que d'une boucle `for`, ce qui donnerait le programme suivant.

```
1 msg = "Ceci est une petite phrase d'exemple."
2 is_found = False
3 i = 0
4 while i < len(msg):
5     if msg[i] == "e":
6         is_found = True
7     i = i + 1
8 print("La lettre e est " + ("présente" if is_found else "absente"))
```

Pour arrêter la boucle `while` une fois que la lettre `"e"` a été trouvée, c'est-à-dire une fois que le booléen `is_found` vaut `True`, on pourrait remplacer la ligne 4 du programme précédent par :

```
4 while i < len(msg) and not is_found:
```

(que l'on pourrait traduire par « tant que l'indice i ne dépasse pas la taille de `msg` **et** que le caractère recherché n'a pas été trouvé »).

Remarque. Notez que pour des raisons de simplicité et de lisibilité, on préfèrera généralement l'expression `not is_found` à `is_found != True` ou `is_found == False`.

Toutefois, il existe une solution alternative avec une syntaxe beaucoup plus légère qui repose sur l'utilisation du mot-clé `break`. En effet, `break` et `continue` sont deux instructions permettant d'interrompre le bloc d'instructions à l'intérieur d'une boucle `for` ou `while`, avec les particularités suivantes :

— l'instruction `continue` fait passer directement à l'itération suivante en ignorant les instructions qu'il restait à exécuter dans le bloc en cours,

— l'instruction `break` fait passer directement à la fin de la boucle en ignorant les instructions qu'il restait à exécuter dans le bloc en cours.

À tester. Analysez les résultats produits par les deux programmes suivants.

```
for i in range(10):
    print(i)
    if i == 3:
        continue
    print("Youpi")
```

```
for i in range(10):
    print(i)
    if i == 3:
        break
    print("Youpi")
```

Pour conclure cette partie, voici donc ci-dessous la version finale proposée pour déterminer la présence ou non du caractère `"e"` dans la chaîne `msg`.

```
msg = "Ceci est une petite phrase d'exemple."
is_found = False
for c in msg:
    if c == "e":
        is_found = True
        break
print("La lettre e est " + ("présente" if is_found else "absente"))
```

Exercice 2.6. Répondre aux questions ci-dessous à l'aide de Python.

1. Existe-t-il un entier compris entre 1 et 50 (inclus) divisible à la fois par 2, 3 et 4?
2. Existe-t-il un entier compris entre 1 et 50 (inclus) divisible à la fois par 3, 4 et 5?

Exercices supplémentaires

Exercice 2.7. On suppose déjà définie une variable `n` contenant un nombre entier positif. Écrire un programme Python qui affiche :

— une première ligne formée de une fois le caractère 1,

— une deuxième ligne formée de deux fois le caractère 2,

— ...

et ainsi de suite jusqu'à la n-ième ligne formée de n fois le chiffre n.

Par exemple si `n` *vaut* 4*, votre programme devra afficher :*

```
1
22
333
4444
```

Exercice 2.8. On suppose déjà définies deux chaînes de caractères `S1` et `S2` de même longueur. Écrire un programme Python qui compte et affiche le nombre de différences trouvées lorsqu'on compare ces chaînes caractère par caractère.

Par exemple si `S1` *est la chaîne* `"voiture"` *et* `S2` *est la chaîne* `"tonsure"`*, alors votre programme devra afficher* 3*, car on trouve trois différences entre ces deux chaînes :*

v	o	**i**	**t**	u	r	e
t	o	**n**	**s**	u	r	e

Exercice 2.9. On vous propose un job étudiant au salaire plutôt déroutant : le premier jour on vous paie 1 euro, le second jour 2 euros, le troisième jour 3 euros et ainsi de suite (sans aucun plafond). Répondez aux questions suivantes à l'aide d'un programme Python.

1. Combien d'argent gagnerez-vous en 30 jours?
2. Au bout de combien de jours aurez-vous gagné 2000 euros? et 10 000 euros?

Exercice 2.10. On suppose déjà définie une variable `msg` contenant une chaîne de caractères. Écrire un programme qui stocke dans une variable `n` le nombre d'occurrences de la séquence `"le"` au sein de la chaîne `msg`.

Par exemple si `msg` *est la chaîne* `"Oui, le Soleil est une étoile"`*, alors* `n` *devra valoir* 3.

Exercice 2.11. On suppose déjà définie une variable `s` de type entier contenant un nombre de secondes. Écrire un programme Python qui convertit ce nombre en heures, minutes et secondes et affiche le résultat sous la forme « `HH:MM:SS` ».

Par exemple si `s` *vaut* 30 186*, alors votre programme devra afficher «* `08:23:06` *» (pour* 8 *heures,* 23 *minutes et* 6 *secondes).*

Exercice 2.12. Pour un $u_0 \in \mathbb{N}^*$ fixé, la suite de SYRACUSE est définie par la relation de récurrence

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } n \text{ est pair,} \\ 3u_n + 1 & \text{si } n \text{ est impair.} \end{cases}$$

Par exemple en partant de $u_0 = 10$, on trouve ensuite successivement les termes $5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$ Une conjecture encore non-résolue à ce jour émet l'hypothèse que pour n'importe quelle valeur de départ u_0, on finira toujours par tomber sur le cycle $4, 2, 1, \ldots$

Écrire un programme Python qui, pour une variable `u0` initialisée en début de programme, affiche tous les termes obtenus jusqu'à tomber sur la valeur 1.

Exercice 2.13. On suppose déjà définie une variable `n` contenant un nombre entier positif. Écrire un programme Python qui détermine le plus petit entier positif (non-nul) divisible par tous les entiers de 1 à n.

À retenir

— Savoir analyser et écrire des structures conditionnelles à l'aide de `if`, `elif` et `else`.
— Savoir analyser et écrire des boucles à l'aide du mot-clé `while`.
— Connaître les différentes façons de parcourir les caractères d'une chaîne.
— Savoir créer un objet de type `range` et itérer dessus.
— Savoir reconnaître et résoudre un problème demandant de compter, sommer ou rechercher l'existence de certains éléments.
— Connaître les effets des mots-clés `break` et `continue` au sein d'une boucle `while` ou `for`.
— Nouvelles fonctions à connaître : `len()`, `range()`.