Structures de contrôle

Pour le moment, nous sommes capables de faire exécuter par Python une série d'instructions. Cela ne permet pas encore beaucoup de souplesse. Si l'on regarde par exemple des recettes de cuisine, on peut y trouver des instructions un peu plus élaborées, telles que « *tant qu'il y a des grumeaux, fouetter le mélange* », pour lequel l'action dépend de l'état actuel du mélange, ou bien encore « remplir les six récipients avec le mélange », qui décrit une action que l'on effectuera plusieurs fois de suite.

Dans ce second cours, nous allons voir comment demander à Python d'exécuter plusieurs fois une série d'instruction, ou d'agir de plusieurs façons différentes en fonction d'une expression ou de la valeur désignée par un nom.

1 Boucles for

1.1 Principe et syntaxe d'une boucle

Assez fréquemment, on sera amené dans un programme à effectuer une tâche répétitive. Supposons par exemple que l'on souhaite afficher la table de multiplication de sept. Il serait possible d'écrire le programme de la façon suivante :

```
print(1, 'fois', 7, 'égale', 1*7)
print(2, 'fois', 7, 'égale', 2*7)
print(3, 'fois', 7, 'égale', 3*7)
print(4, 'fois', 7, 'égale', 4*7)
print(5, 'fois', 7, 'égale', 5*7)
print(6, 'fois', 7, 'égale', 6*7)
print(7, 'fois', 7, 'égale', 7*7)
print(8, 'fois', 7, 'égale', 8*7)
print(9, 'fois', 7, 'égale', 9*7)
print(10, 'fois', 7, 'égale', 10*7)
```

On voit tout de suite que ce n'est guère pratique, car on répète un grand nombre de fois des lignes quasiment identiques. Fort heureusement, les langages de programmations fournissent une syntaxe permettant d'effectuer successivement plusieurs instructions semblables. En Python, cela s'écrit de la façon suivante :

```
for nom in ensemble_de_valeurs :
    instruction_1
    instruction_2
    ...
    instruction_n
```

Dans cette situation, Python affectera successivement chacune des valeurs fournies au nom placé entre le **for** et le **in**, puis exécutera l'ensemble des instructions qui suivent.

Dans l'exemple précédent, certains éléments sont soulignés. Dans l'ensemble de ce cours, cela signifie que ce ne sont pas des mots-clés du langage, mais que ces « éléments » sont à remplacer par autre chose, en fonction de ce que vous voulez réaliser. Dans notre exemple, on pourra donc avantageusement réécrire le programme précédent en deux lignes :

```
for i in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 :
    print(i, 'fois', 7, 'égale', i*7)
```

Ce programme est équivalent à celui-ci (que l'on a raccourci) :

```
i = 1
print(i, 'fois', 7, 'égale', i*7)
i = 2
print(i, 'fois', 7, 'égale', i*7)
i = 3
print(i, 'fois', 7, 'égale', i*7)
...
i = 10
print(i, 'fois', 7, 'égale', i*7)
```

Les plus observateurs de nos lecteurs auront remarqué que les instructions constituant la boucle proprement dite sont décalées par rapport au mot-clé **for**. C'est ce que l'on appelle une *indentation*[1]. Python se sert de ce décalage pour déterminer ce qui fait partie de la boucle, et où la boucle se termine.

Ainsi, dans l'exemple suivant, la dernière instruction *print* ne fait pas partie de la boucle, et sera exécutée seulement une fois la boucle terminée :

```
for i in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 :
    print(i, 'fois', 7, 'égale', i*7)
print('fin de la table de 7')
```

Comme on le voit, le nom présent dans la boucle **for** permet de désigner, à l'intérieur de la boucle, les valeurs qui seront successivement considérées. Le choix du nom « i » est parfaitement arbitraire. Lorsque l'on a une petite boucle pour laquelle les valeurs itérées sont des entiers, les noms « i », « j » et « k » sont fréquemment utilisés, mais dans une boucle plus compliquée, il est recommandé de choisir un nom qui indiquera plus

1. On peut librement choisir la taille de cette indentation, en fonction de ses besoins et de ses préférences, même si le langage Python recommande d'utiliser quatre espaces.

clairement à quoi fait référence le nom. Nous verrons de nombreux exemples par la suite.

Précisons que si l'on n'a pas besoin des différentes valeurs, mais seulement de répéter une ou plusieurs instructions plusieurs fois de suite, il n'est pas requis que le nom soit utilisé dans la boucle [2] !

1.2 Génération de séries d'entiers

On remarque que dans l'exemple précédent, on a précisé l'ensemble des valeurs qui devront être successivement considérées, séparées par des virgules.

Cela n'est cependant souvent guère très pratique : on perd encore beaucoup de temps à écrire l'ensemble des valeurs qui seront associées au nom `i`. Heureusement, il est possible de faire mieux que cela, grâce à la fonction [3] `range` qui permet de générer une séquence d'entiers. Ainsi, `range(10)` correspond à l'ensemble des entiers de 0 à 9 :

`range(10)` ≙ 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9.

Comme souvent en Python, la séquence s'arrête *juste avant la valeur finale*, ici 10. Ainsi, `range(n)` correspond à la séquence des entiers i (par ordre croissant) vérifiant $0 \leqslant i < n$.

Si l'on souhaite ne pas débuter de 0 mais d'une autre valeur, on peut fournir deux paramètres à `range()` : ainsi, `range(a, b)` correspond à la séquence d'entiers i par ordre croissant vérifiant $a \leqslant i < b$. Attention, on débute avec a, mais s'arrête *avant* b.

`range(1, 11)` ≙ 1, 2, 3, 4, 5, 6, 7, 8, 9 et 10.

Ainsi, le programme précédent affichant la table de sept peut être simplifié en

```
for i in range(1, 11):
    print(i, 'fois', 7, 'égale', i*7)
```

On peut enfin spécifier un troisième et dernier paramètre à la fonction `range()`, qui représentera un pas entre chaque valeurs. Par exemple, `range(2, 20, 2)` fournira l'ensemble des nombres pairs entre 2 et 18 (20, comme d'habitude, est exclu !).

`range(2, 20, 2)` ≙ 2, 4, 6, 8, 10, 12, 14, 16 et 18.

Cela permet aussi de compter à rebours, en spécifiant un pas négatif. Ainsi, lorsque l'on écrit `range(10, 0, -1)` cela fait référente aux entiers de 10 à 1 rangés par ordre décroissant (là encore, on s'arrête avant d'atteindre la valeur 0).

`range(10, 0, -1)` ≙ 10, 9, 8, 7, 6, 5, 4, 3, 2 et 1.

2. Si l'on n'utilise pas le nom dans la boucle, il est d'usage de choisir comme nom le tiret inférieur _ pour avertir l'utilisateur que le nom ne sera pas utilisé.
3. En fait, un *générateur* dans le formalisme de Python, nous y reviendrons.

1.3 Boucles imbriquées

Il est bien évidemment possible de créer une boucle à l'intérieur d'une autre boucle. Il faut faire très attention à l'indentation car c'est ce qui permet à Python de savoir ce qui doit être répété et ce qui ne doit pas l'être. Ainsi, si l'on veut afficher d'un coup l'ensemble des tables de multiplications de 2 à 9, on peut utiliser le programme suivant :

```
for j in range(2, 10) :
    print('Début de la table de', j)
    for i in range(1, 11) :
        print(i, 'fois', j, 'égale', i*j)
    print('Fin de la table de', j)
```

Il est bien évidemment vivement recommandé d'utiliser des noms *différents* pour chacune des boucles imbriquées, même si Python ne protestera pas dans le cas contraire !

1.4 Exemples d'entrainement

• Écrire un programme affichant les cubes des dix premiers entiers non nuls.

• Écrire un programme calculant et affichant la somme de ces dix cubes.

• Écrire un programme affichant $k!$ pour les valeurs de k allant de 2 à 10 (on rappelle que $k!$ correspond au produit des entiers strictement positifs inférieurs ou égaux à k).

• On définit la suite (u_n) par $u_0 = 0$ et $u_{n+1} = \sqrt{2 + u_n}$. Écrire un programme affichant les termes u_1 à u_{10} (inclus).

On attend, pour chaque terme, un affichage de la forme suivante :

```
u_1 = 1.4142135623730951
```

```

```

• Pour la même suite, écrire un programme affichant les termes u_{1000} à u_{1010} (inclus) sans que n'apparaissent les termes précédents.

```

```

2 Structures conditionnelles

2.1 Algèbre de Boole et expressions booléennes

Pour l'instant, nos programmes effectuent toujours exactement la même séquence d'instructions, quoi qu'il arrive. Pour aller plus loin, nous allons avoir besoin d'instructions qui seront exécutées ou non en fonction de la situation.

Les structures permettant d'obtenir un tel comportement utilisent des « *expressions booléennes* ». George Boole était un mathématicien et philosophe anglais du début du XIX^e siècle, qui a posé les bases de la logique moderne en tentant de traduire les raisonnements humains en équations mathématiques.

L'algèbre de Boole est basée sur l'idée qu'une affirmation donnée peut être *vraie* ou *fausse*, et pose les bases permettant de combiner plusieurs affirmations. Les travaux de Boole ont été inappréciables pour le développement de l'informatique, un siècle plus tard.

Une expression booléenne est une expression dont le résultat est un « *booléen* », c'est-à-dire vrai ou faux. On peut par exemple construire de telles expressions en Python en comparant deux valeurs ou deux expressions mathématiques :

```
In [1]: 14 < 17
Out[1]: True

In [2]: 42 >= 1023
Out[2]: False
```

Les opérateurs de comparaison sont les suivants :

`expr_1 < expr_2`	inférieur strictement à
`expr_1 <= expr_2`	inférieur ou égal à
`expr_1 > expr_2`	supérieur strictement à
`expr_1 >= expr_2`	supérieur ou égal à
`expr_1 == expr_2`	égal à
`expr_1 != expr_2`	non égal à

Attention, pour tester l'égalité de deux expressions, on utilise un *double* signe égal (pour éviter toute confusion avec l'affectation).

Quelques autres exemples :

```
In [3]: x = 5

In [4]: y = 2

In [5]: x**2 == 25
Out[5]: True

In [6]: x**2 != y*12 + 1
Out[6]: False
```

On remarque dans les exemples précédents que les opérateurs de calcul ont priorité sur ceux de comparaison. Les expressions de part et d'autre sont évalués avant d'être comparées.

Signalons par ailleurs qu'il est possible d'utiliser simultanément plusieurs opérateurs de comparaison dans une même expression (le résultat sera vrai si et seulement si toutes les comparaisons sont vraies, comme on pouvait logiquement s'y attendre) :

```
In [7]: 14 <= 17 <= 42
Out[7]: True

In [8]: 1 > sqrt(2) > 0
Out[8]: False
```

`True` et `False` sont des objets Pythons comme les autres (attention à la majuscule), et le résultat d'une expression booléenne peut dont être affecté à un nom :

```
In [9]: b = 1023%2 == 0

In [10]: b
Out[10]: False

In [11]: b == False
Out[11]: True
```

Sur la première ligne de cet exemple, l'interpréteur Python commence par calculer l'expression `1023%2` (de résultat `1`), puis effectue la comparaison avec `0` (ce qui donne un résultat `False`), avant d'affecter ce résultat au nom `b`. Dans la suite, `b` désigne donc la valeur booléenne `False`.

2.2 Combiner des expressions

George Boole a, en définissant les règles de la logique booléenne, introduit des règles permettant de combiner plusieurs expressions booléennes. Il a notamment défini, entre autres choses, un « et » logique, un « ou » logique et un « non » logique (ce sont ces règles qui permettent de faire de l'ensemble une algèbre au sens mathématique du terme). En Python, on peut faire référence à ces règles grâce aux mots-clés `and`, `or` et `not`.

Ainsi, l'expression `expr_bool_1 and expr_bool_2` est vraie si et seulement si les *deux* expressions booléennes `expr_bool_1` et `expr_bool_2` sont vraies (et fausse dans le cas contraire). On peut représenter ça par la table de vérité suivante :

`expr_bool_1`	`expr_bool_2`	`expr_bool_1 and expr_bool_2`
`False`	`False`	`False`
`True`	`False`	`False`
`False`	`True`	`False`
`True`	`True`	`True`

L'expression `expr_bool_1 or expr_bool_2` est, elle, vraie si et seulement si *au moins une* des expressions `expr_bool_1` et `expr_bool_2` est vraie. Soit la table de vérité suivante :

`expr_bool_1`	`expr_bool_2`	`expr_bool_1 or expr_bool_2`
`False`	`False`	`False`
`True`	`False`	`True`
`False`	`True`	`True`
`True`	`True`	`True`

Enfin, l'expression `not expression_booléenne` est vraie si et seulement si `expression_booléenne` est fausse (et inversement). Soit la table de vérité suivante :

`expression_booléenne`	`not expression_booléenne`
`False`	`True`
`True`	`False`

La priorité de ces opérateurs logiques est inférieure aux opérateurs de comparaison vus tantôt, mais elles sont calculées avant une éventuelle affectation. La relation `not` est prioritaire sur `and`, elle même prioritaire sur `or`.

Dans l'algèbre de Boole, il existe d'autres relations, tel que le « ou exclusif » de deux expressions qui est vrai si et seulement si une expression et une seule est vraie.

`expr_bool_1`	`expr_bool_2`	`expr_bool_1` « ou exclusif » `expr_bool_2`
`False`	`False`	`False`
`True`	`False`	`True`
`False`	`True`	`True`
`True`	`True`	`False`

Cette relation logique n'existe pas en Python, mais on peut néanmoins en écrire un équivalent en combinant les opérateurs précédents :

```
(expr_bool_1 and not expr_bool_2) or (expr_bool_2 and not expr_bool_1)
```

On remarquera ici l'usage des parenthèses pour assurer le bon ordre dans l'évaluation de l'expression !

Précisons enfin qu'il convient d'être prudent... Python accepte à peu près n'importe quoi de chaque côté des mots-clé `and` et `or`. Il nous laisse par exemple écrire

```
In [12]: sin and sqrt(2)
Out[12]: 1.4142135623730951
```

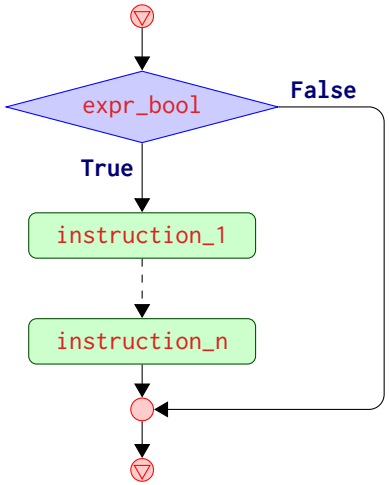
Aussi curieux et incompréhensible que cela puisse paraître, il y a des règles pour préciser ce qui se passe dans une telle situation. Nous n'entrerons pas dans ces subtilités du langage pour le moment, et nous nous limiterons à des expressions booléennes. Mais soyez attentifs lorsque vous écrivez un test, la tolérance de Python à ce niveau peut occasionner des « bogues » difficile à retrouver !

2.3 Structure conditionnelle

Il est temps, à présent, d'employer ces expressions booléennes. Il est possible de demander à l'interpréteur Python conditionner l'exécution d'une série d'instruction à une condition booléenne. Pour ce faire, on dispose de la structure de contrôle suivante :

```
if expression_booléenne :
    instruction_1
    instruction_2
    ...
    instruction_n
```

On retrouve, ici aussi, le principe de l'indentation. Les instructions indentées ne sont évaluées que si l'expression booléennne est vraie. Le fonctionnement de cette structure de contrôle peut être comparée à l'organigramme suivant :



Ainsi, le morceau de programme suivant :

```
if x%2 == 0 :
    print("x est pair")
```

affichera « x est pair » si le nom x désigne un nombre pair, et ne fera rien dans le cas contraire.

Pour afficher les nombres pairs entre 2 et 18, on peut donc utiliser une syntaxe différente de ce que l'on a déjà utilisé tantôt :

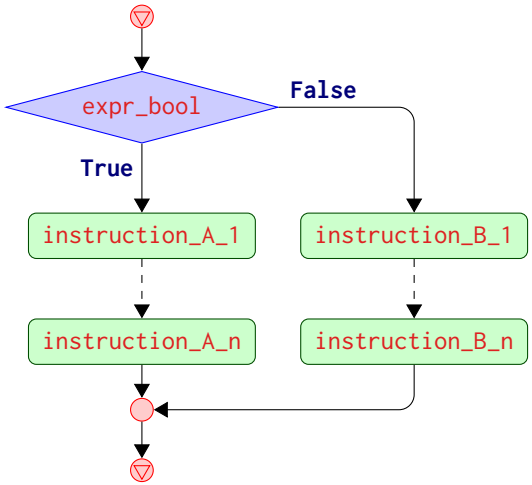
```
for i in range(2, 20) :
    if i%2 == 0 :
        print(k)
```

2.4 Variantes

Parfois, on peut souhaiter effectuer des traitements différents selon qu'une expression booléenne est vraie ou fausse. Pour ce faire, on peut utiliser le mot-clé **else** :

```
if expression_booléenne :
    instruction_A_1
    instruction_A_2
    ...
    instruction_A_n
else :
    instruction_B_1
    instruction_B_2
    ...
    instruction_B_n
```

Dans cette situation, Python exécutera l'ensemble des instructions de la séquence A si l'expression booléenne est vraie, et celles de la séquence B si elle est fausse. Ce comportement correspond à l'organigramme suivant :



Par exemple, le morceau de programme suivant :

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

affiche « x est pair » ou « x est impair » en fonction de la parité de la valeur désignée par le nom x.

Il est important de comprendre que la syntaxe `if ... : ... else : ...` est différente d'un test de l'expression booléenne suivi d'un test de l'expression inverse : en effet, le programmes suivant, où x fait référence à une valeur numérique :

```
if x > 0 :
    x = -x
else :
    x = 0
```

n'est pas équivalent à ce second programme, pour lequel x désignera toujours 0 à la fin !

```
if x > 0 :
    x = -x
if x <= 0 :
    x = 0
```

Enfin, s'il y a plus de deux cas à examiner, on peut utiliser le mot-clé `elif` pour tester d'autres expressions, comme ci-dessous, où x fait référence à une valeur numérique :

```
if x > 0 :
    print('x est positif')
elif x < 0 :
    print('x est positif')
else :
    print('x est nul')
```

3 Boucles while

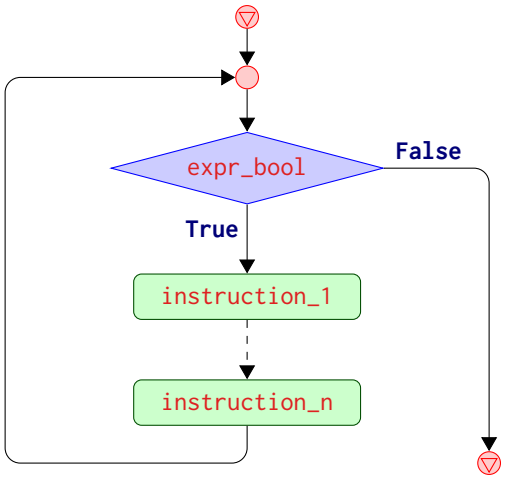
Pour effectuer plusieurs fois une série d'instructions, nous avons vu que nous disposions des boucles `for`. Toutefois, ces dernières supposent que l'on connaisse à l'avance le nombre d'itérations à effectuer. Cela ne convient pas à toutes les situations, par exemples celles similaires à « tant qu'il reste des grumeaux, fouetter le mélange ».

Le langage Python fournit donc un second type de boucle, la boucle `while`. Dans ce cas, on va itérer un ensemble d'instruction tant qu'une condition, exprimée sous la forme d'une expression booléenne, est vraie. La syntaxe d'une telle boucle est la suivante :

```
while expression_booléenne :
    instruction_1
    instruction_2
    ...
    instruction_n
```

On retrouve le principe de l'indentation des instructions faisant partie de la boucle.

Cette structure de contrôle se comporte comme l'organigramme suivant :



Ainsi, dans l'exemple ci-dessous :

```
u = 1
while u<100 :
    u = 2*u

print(u)
```

on calcule les termes de la suite définie par $u_0 = 1$ et $u_{n+1} = 2u_n$ tant que u_k est strictement inférieur à 100. Dès qu'un terme de la suite dépasse 100, on quitte la boucle, et on affiche le résultat (la plus grande puissance de 2 supérieure ou égale à 100, soit 128).

De même, considérons la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_n = \sum_{k=1}^{n} \frac{1}{k}$.

C'est une suite croissante qui n'est pas majorée [4], et peut donc dépasser n'importe quelle valeur x choisie à l'avance. Pour calculer le rang n à partir duquel $u_n > x$, on peut écrire :

```
u, k = 0, 0
while u <= x :
    k = k + 1
    u = u + 1/k

print(k)
```

On remarquera à travers cet exemple que dans une boucle while, si l'on a besoin d'une variable incrémentée, il convient de gérer cela soi-même.

4. On peut aisément montrer que $u_n > \int_1^n \frac{1}{x}\,\mathrm{d}x = \ln(n)$.

4 Écrire une boucle correcte

Écrire une boucle qui fait exactement ce que l'on souhaite n'est pas toujours chose facile. On peut aisément se perdre dans les instructions constituant la boucle, ou bien écrire des boucles qui ne se terminent jamais (si par exemple, dans une boucle `while` l'expression booléenne est toujours fausse). Nous allons voir dans la suite quelques méthodes permettant de vérifier, voire de prouver, que les programmes que l'on écrit fournissent bien le résultat attendu.

4.1 Invariants de boucle

Pour aider à bien écrire une boucle, on peut essayer de définir un ou plusieurs *invariants de boucles*. Ce sont des affirmations qui restent vraies à chaque itération de la boucle.

Reprenons par exemple le programme calculant et affichant les termes u_1 à u_{10} de la suite (u_n) définie par $u_0 = 0$ et $u_{n+1} = \sqrt{2+u_n}$:

```
u = 0
for i in range(1, 11) :
    u = sqrt(2+u)
    print(u)
```

Pour cette boucle, on peut définir les invariants de boucle suivants :

- Au début d'une quelconque itération de la boucle, u désigne u_{i-1}
- À la fin d'une quelconque itération de la boucle, u désigne u_i

Pour aider à la compréhension d'un programme, il est possible d'y insérer des commentaires qui ne seront pas pris en compte par Python. Tout ce qui suit le signe # est ainsi ignoré. Incluons donc nos invariants de boucle dans le programme :

```
u = 0
for i in range(1, 11) :
    # la valeur désignée par u correspond à u_{i-1}
    u = sqrt(2+u)
    print(u)
    # la valeur désignée par u correspond à u_i
```

Vérifions à présent que ces informations sont exactes :

- lors de l'entrée dans la boucle, la première fois, u contient u_0 (c'est-à-dire 0), et $i = 1$, ce qui est en accord avec le premier invariant ;
- si, lors d'une itération donnée, le premier invariant est juste, alors le second l'est aussi, puisque l'instruction entre les deux invariants permet de passer de u_{i-1} à u_i ;
- si, lors d'une itération donnée, le second invariant est juste, alors le premier le sera à l'itération suivante (sauf, évidemment, si l'on est sorti de la boucle) puisque i sera incrémenté à l'itération suivante.

On montre ainsi que les deux invariants sont bien corrects. Cela permet de justifier que ce qui est affiché avec l'instruction `print` correspond bien aux termes u_1 à u_{10}.

Commenter les boucles (et les programmes en général), par exemple en précisant les invariants de boucles, est une très bonne habitude car cela permet d'aider la personne qui vous lira à comprendre ce que vous avez souhaité faire !

4.2 Correction et terminaison

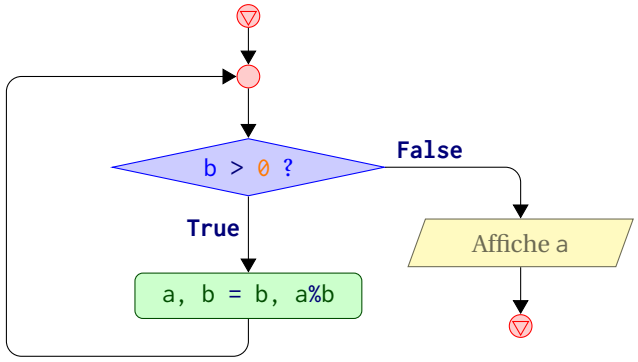
Une seconde difficulté qui peut se présenter est de justifier que le programme fournira bien un résultat en un temps fini (qui peut dépendre des conditions initiales), et ne bouclera pas indéfiniment à cause par exemple d'une boucle `while` mal écrite [5].

Il peut être utile de montrer qu'une boucle va bien fournir un résultat en un temps fini (on parle de *terminaison* de la boucle), et que ce résultat sera celui attendu (on parle de *correction* de la boucle). Parfois, la démonstration peut être complexe, mais souvent des arguments assez simples suffiront.

Prenons par exemple l'algorithme [6] d'Euclide permettant de calculer le PGCD (plus grand commun diviseur) de deux entiers positifs a et b, c'est-à-dire le plus grand entier c qui divise à la fois a et b. Cet algorithme est basé sur deux règles :

- le PGCD de a et de 0 est a ;
- si $b > 0$, le PGCD de a et de b est égal au PGCD de b et du reste de la division entière de a par b.

En utilisant autant de fois que nécessaire la seconde règle, on finit par tomber dans une situation où la première règle s'applique, ce qui permet d'obtenir le PGCD recherché. La recherche du PGCD correspond donc à cet organigramme :



Par exemple, pour déterminer le PGCD de 276 et 126, on peut écrire :

$$\text{PGCD}(276,126) = \text{PGCD}(126,24) = \text{PGCD}(24,6) = \text{PGCD}(6,0) = 6$$

5. Profitons de l'occasion pour signaler que pour interrompre un programme qui ne s'arrête pas, le raccourci dans IEP est ctrl-I !

6. Un algorithme est une « méthode » (une suite d'opérations ou d'instructions) permettant d'effectuer une tâche, de résoudre un problème ou d'obtenir un résultat donné.

Écrivons un programme mettant en œuvre cet algorithme :

```
while b>0 :
    # le résultat recherché est le PGCD de a et b
    a, b = b, a%b
    # le résultat recherché est toujours le PGCD de a et b

print("le PGCD est", a)
```

On remarquera d'abord notre invariant de boucle : le résultat recherché reste le PGCD des entiers désignés par a et b, même si ces derniers changent à chaque itération. En effet, la seconde règle indique que $\text{PGCD}(a,b) = \text{PGCD}(b, a\%b)$.

Si l'on sort de la boucle, alors a désignera bien le résultat recherché (le programme est *correct*) ! Il ne reste donc qu'à montrer sa *terminaison*, autrement dit prouver que l'on finit effectivement par sortir de la boucle après un nombre fini d'itérations.

Ici, ce n'est en fait pas bien difficile. Si l'on regarde bien, a et b désigneront toujours des entiers positifs ou nuls. Par ailleurs, a%b est nécessairement inférieur à b par définition de la division euclidienne. Les valeurs successivement désignées par b forment donc une suite strictement décroissante d'entiers positifs. Après un nombre fini d'itérations (inférieur ou égal, quoi qu'il arrive, à la valeur initiale de b), b finira donc par atteindre 0, ce qui assure une sortie de la boucle.

On qualifiera parfois b de *variant de boucle*. Ce terme désigne une grandeur qui évolue à chaque itération de la boucle de façon à ce que l'on puisse prouver que la boucle va bien se terminer, après un nombre fini d'itérations.

Le programme basé sur l'algorithme d'Euclide fournit donc bien le PGCD de a et b (sous réserve que a et b correspondent bien à deux entiers positifs, évidemment !).

5 Les mots-clés `break` et `continue`

Python fournit par ailleurs deux instructions supplémentaires peuvent être parfois utiles dans une boucle. **Dans la mesure du possible, il est recommandé de s'en passer car elles rendent fréquemment la lecture des programmes plus difficile**.

Dans les situations où leur utilisation vous permet plutôt de simplifier un programme, vous pouvez toutefois vous en servir, mais pensez à glisser dans la boucle un commentaire !

5.1 Quitter une boucle avec `break`

L'instruction **`break`** a pour effet de quitter immédiatement la boucle dans laquelle on se trouve [7], l'exécution reprenant juste après la fin de celle-ci.

7. Dans le cas de boucles imbriquées, on ne sort que de la boucle la plus « interne »

Prenons par exemple le cas d'un programme permettant de déterminer si n est un nombre premier :

```
est_premier = True

for i in range(2, int(sqrt(n))+1) :
    # i est-il un diviseur de n ?
    if n%i == 0 :
        est_premier = False

if est_premier == True :
    print(n, "est premier")
else
    print(n, "n'est pas premier")
```

Le principe du programme est simple : n est premier si et seulement si il n'a aucun diviseur entier compris entre 2 (inclus) et $\lfloor\sqrt{n}\rfloor$ (inclus également).

Pour implémenter cette propriété, on a procédé de la manière suivante : le nom `est_premier` fait référence à un booléen indiquant si n *peut* être premier.

Au début, on suppose que oui (donc on l'initialise avec la valeur **`True`**), puis on essaie tous les diviseurs possibles un par un. Si l'on trouve un diviseur propre, c'est que le nombre n'était pas premier, aussi `est_premier` désignera dorénavant la valeur **`False`** ! Une fois la boucle terminée, on a une réponse définitive quant à la primalité de n.

Attention, ce programme ne fonctionne que pour des entiers supérieurs ou égaux à 2 ! Pour des entiers inférieurs à 3, en effet, on n'entre pas dans la boucle **`for`** (car $2 \geqslant \lfloor\sqrt{n}+1\rfloor$), aussi ces valeurs sont considérées comme des nombres premiers. C'est correct pour 2 et 3, mais pas pour 0 ou 1. Il faudrait traiter ces cas à part.

On voit ici qu'il y a un gâchis : dès que l'on a trouvé un diviseur, il n'est plus utile de tester les suivants... on peut vouloir quitter la boucle immédiatement. Cela peut être fait grâce au mot-clé **`break`**, en écrivant :

```
est_premier = True

for i in range(2, int(sqrt(n))+1) :
    # i est-il un diviseur de n ?
    if n%i == 0 :
        est_premier = False
        break   # plus besoin de continuer à chercher, on sort !

if est_premier == True :
    print(n, "est premier")
else
    print(n, "n'est pas premier")
```

Notons qu'utiliser **break** n'est pas indispensable [8], on aurait pu écrire :

```
est_premier = True

i = 2

while est_premier == True and i <= sqrt(n) :
    # i est-il un diviseur de n ?
    if n%i == 0 :
        est_premier = False
    i = i+1

if est_premier == True :
    print(n, "est premier")
else
    print(n, "n'est pas premier")
```

L'utilisation d'une boucle **while** à la place de la boucle **for** demande que l'on gère manuellement l'incrémentation de `i`, mais permet en revanche de sortir dès que l'on a trouvé un diviseur.

Pour clore sur cet exemple, signalons que l'on peut simplifier, dans les deux dernières versions du programme, les deux lignes

```
    if n%i == 0 :
        est_premier = False
```

en

```
    est_premier = n%i != 0
```

En effet, si `i` divise `n`, alors `n%i == 0`, et cela revient à

```
    est_premier = False
```

et dans le cas contraire, `n%i != 0`, ce qui revient à

```
    est_premier = True
```

ce qui ne change rien puisque `est_premier` désignait déjà cette même valeur **True**.

Cela ne fonctionnerait en revanche pas avec la première version, puisque l'on pourrait réassigner au nom `est_premier` la valeur **True** après avoir trouvé un diviseur si un entier i considéré ultérieurement n'en est pas un !

8. Il est *toujours* possible de se passer d'un **break**, au prix d'une conversion éventuelle d'un **for** en **while** (puisque l'on ne sait plus avec certitude combien d'itérations auront lieu) et de l'ajout ou de la modification de quelques tests.

5.2 Poursuivre une boucle avec `continue`

Supposons à présent que l'on souhaite afficher tous les diviseurs d'un nombre n, puis leur somme. Pour ce faire, on peut écrire :

```
somme = 0

for i in range(1, n+1) :
    if n%i == 0 :
        print(i, "est un diviseur de", n)
        somme = somme + i

print("La somme des diviseurs de", n, "est", somme)
```

L'instruction **continue** permet d'ignorer les instructions restant dans le corps de la boucle, mais pas nécessairement d'en sortir. S'il reste des itérations à faire (dans le cas d'une boucle **for**) ou si l'expression booléenne régissant la boucle est toujours vraie (dans le cas d'une boucle **while**), l'exécution se poursuivra.

On peut donc réécrire le programme précédent par :

```
somme = 0

for i in range(1, n+1) :
    if n%i != 0 :
        continue   # ce i n'est pas un diviseur, on passe au i suivant

    print(i, "est un diviseur de", n)
    somme = somme + i

print("La somme des diviseurs de", n, "est", somme)
```

5.3 Du bon usage de `break` et `continue`

Il est fréquent que l'on passe davantage de temps à lire et modifier un programme qu'à l'écrire. Pour cette raison, il est important qu'il soit aussi lisible que possible. Les mots-clés **break** et **continue** peuvent être dangereux de ce point de vue.

Par exemple, un **break** dans une boucle **while** peut provoquer la sortie de la boucle alors que la condition du **while** est encore vraie. Quelqu'un qui lirait le programme un peu rapidement et manquant le **break** pourrait ne pas voir qu'il existe plusieurs circonstances provoquant la sortie de la boucle.

Pour cette raison, il est indispensable de ne pas abuser de ces mots-clés et de les réserver quand il simplifient nettement les choses, et ne surtout pas négliger les commentaires, pour faciliter la tâche à qui relira le programme.

Exercices

Ex. 1 – Somme de carrés

On suppose que le nom n désigne un entier positif.

1. Écrire un programme calculant $\sum_{i=0}^{n} i^2$.

2. Utiliser le programme pour calculer la somme précédente pour $n = 100$ et $n = 1000$.

3. Vérifier, sur ces exemples, que la somme est égale à $\frac{n(n+1)(2n+1)}{6}$.

Ex. 2 – Nombres particuliers

On suppose que le nom n désigne un entier naturel.

1. Proposer un programme indiquant si n est le carré d'un entier, en affichant l'entier en question le cas échéant.

Un nombre parfait est un nombre égal à la somme de ses diviseurs stricts positifs. Par exemple, 28 est parfait car $28 = 1 + 2 + 4 + 7 + 14$.

2. Écrire un programme déterminant et affichant si n est parfait ou non.

Ex. 3 – Multiples de 3 et 5

E désigne l'ensemble des entiers strictement positifs divisibles par trois et/ou par cinq.

1. Écrire un programme affichant les éléments de E strictement inférieurs à 100.

2. Proposer un programme comptant ces éléments de E strictement inférieurs à 100, et affichant leur nombre.

3. Modifier le programme précédent pour qu'il affiche non pas le nombre mais la somme de ces éléments.

Ex. 4 – Multiplicité des diviseurs

Écrire un programme affichant les entiers compris entre 1 et 100 dont la décomposition en facteurs premiers ne fait apparaître que des diviseurs de multiplicité égale à 1. On réfléchira à la façon la plus simple de tester si un nombre répond à ce critère (le programme ne devrait faire que trois ou quatre lignes).

Ex. 5 – Triplets pythagoriciens

Un triplet pythagoricien est un ensemble de trois entiers strictement positifs (a, b, c) vérifiant $a^2 + b^2 = c^2$. Par exemple, le triplet $(3, 4, 5)$ est un triplet pythagoricien.

Il existe une infinité de tels triplets. Proposer un programme trouvant le triplet vérifiant également $a + b + c = 1000$.

Ex. 6 – Suite de Fibonacci et nombre d'or

La suite de Fibonacci est définie par $u_0 = 1$, $u_1 = 1$ et, pour tout $n > 1$, $u_n = u_{n-1} + u_{n-2}$.

1. Proposer un programme affichant les termes u_1 à u_50.

On définit la suite v_n pour tout $n > 0$ par $v_n = u_n / u_{n-1}$

2. Modifier le programme précédent pour qu'il affiche les termes v_1 à v_50, et vérifier que cette suite semble tendre vers le nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$.

Ex. 7 – Emballage

On considère un objet ayant la forme d'un parallélépipède rectangle, dont les dimensions (en mètres) sont désignées par les noms l, h et p.

On souhaite déterminer la surface (en mètres carrés) de papier nécessaire pour emballer l'objet. Cette surface est égale à la surface des six côtés, plus la surface du plus petit des côtés pour permettre de fermer proprement l'emballage.

Par exemple, un objet de taille 0,2 m × 0,3 m × 0,5 m nécessite une surface égale à

$$2 \times (0.2 \times 0.3) + 2 \times (0.2 \times 0.5) + 2 \times (0.3 \times 0.5) + 0.2 \times 0.3 = 0{,}68\,\text{m}^2$$

Écrire un programme calculant et affichant la surface de papier nécessaire.