Chapitre 3
Instructions itératives

1. Structuration et indentation

Les impératifs de la programmation structurée nécessitent la définition de blocs d'instructions au sein des structures de contrôles (`def`, `for`, `while`, `if`, ...). Certains langages utilisent des délimiteurs pour encadrer ces blocs d'instructions (des parenthèses en C ou en Caml, des mots-clés en Fortran, etc), mais le langage Python se distingue en utilisant l'*indentation*, qui favorise la lisibilité du code.
Le début d'un bloc d'instructions est défini par un double-point (`:`), la première ligne pouvant être considérée comme un en-tête. Le corps du bloc est alors indenté d'un nombre d'espaces fixes (quatre par défaut), et le retour à l'indentation de l'en-tête marque la fin du bloc.

```
en-tête:
    bloc ..........................
    ..............................
    d'instructions ................
```

Il est possible d'imbriquer des blocs d'instructions les uns dans les autres :

```
en-tête 1:
    ..............................
    ..............................
    en-tête 2:
        bloc ......................
        ..........................
        d'instructions ............
    ..............................
    ..............................
```

Cette structuration sert entre autre à définir de nouvelles fonctions, à réaliser des tests ou à effectuer des instructions répétitives.

1.1 Définition d'une fonction

On définit une fonction en python à l'aide du mot clé `def`. Il faut lui attribuer un nom, préciser la liste de ses paramètres et enfin décrire les différentes instructions à réaliser. La syntaxe générale est la suivante :

```
def nomdelafcn(liste de paramètres):
    bloc ..........................
    d'instructions ................
    à réaliser ....................
```

Dans l'histoire de l'informatique on distingue traditionnellement deux types de routines [1] : les procédures, qui ne retournent pas de résultat et se contentent d'agir sur l'environnement, et les fonctions proprement dites, qui retournent un résultat (en général par le biais d'une instruction `return`). En Python cette distinction n'existe pas car il n'existe que des fonctions : les procédures ne sont que des fonctions particulières qui retournent la valeur spéciale None lorsque l'instruction `return` n'est pas utilisée.

Il est facile de faire la distinction en utilisant l'interprète de commande IPython : lorsqu'on applique une fonction qui retourne un résultat, ce dernier est précisé à la suite du mot Out[..]. Par exemple, `print` est une procédure et `len` une fonction :

1. Une *routine* est une séquence d'instructions qui peut être réutilisée au sein d'un programme.

```
In [1]: print("Bonjour\ntout le monde")
Bonjour
tout le monde

In [2]: len("Bonjour\ntout le monde")
Out[2]: 21
```

On notera que bien que la procédure `print` retourne la valeur `None`, celle-ci est ignorée par l'interprète de commande.

• Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments (ou paramètres, c'est la même chose) séparés par une virgule [2]. Par exemple, pour définir la norme euclidienne d'un vecteur de coordonnées (x, y) on écrira :

```
from numpy import sqrt

def norme(x, y):
    return sqrt(x**2 + y**2)
```

Exemple d'utilisation :

```
In [1]: norme(3, 4)
Out[1]: 5.0
```

Vous apprendrez plus tard en cours de mathématique qu'il existe d'autres normes, en particulier celles-ci : $N_k(x, y) = (x^k + y^k)^{1/k}$, la norme euclidienne correspondant au cas $k = 2$. Pour les définir, il suffit de passer k en argument :

```
def norme(x, y, k):
    return (x**k + y**k)**(1/k)
```

Avec cette nouvelle définition, on a :

```
In [2]: norme(3, 4, 2)
Out[2]: 5.0

In [3]: norme(3, 4, 3)
Out[3]: 4.497941445275415

In [4]: norme(3, 4)
TypeError: norme() takes exactly 3 arguments (2 given)
```

Arguments optionnels
On peut constater sur ce dernier exemple qu'une erreur se produit si on ne donne pas exactement le bon nombre d'arguments d'une fonction. Il est possible de modifier cet état de fait en précisant les valeurs *par défaut* que doivent prendre les arguments d'une fonction : il suffit de préciser dans la liste des paramètres les valeurs prises par défaut :

```
def norme(x, y, k=2):
    return (x**k + y**k)**(1/k)
```

Avec cette nouvelle définition, si on omet de préciser le troisième paramètre, ce dernier sera pris égal à 2 :

```
In [5]: norme(3, 4, 3)        # ici k=3
Out[5]: 4.497941445275415

In [6]: norme(3, 4)           # ici k=2
Out[6]: 5.0
```

2. voire n'en posséder aucun, auquel cas la liste des arguments reste vide `()`.

Attention, dans le cas d'une fonction avec des paramètres optionnels, les arguments doivent être ordonnés : les paramètres optionnels doivent suivre les paramètres obligatoires. Il est d'ailleurs préférable de les nommer pour éviter toute ambiguïté ; ainsi il est conseillé d'écrire l'instruction de la ligne 5 ainsi :

```
In [7]: norme(3, 4, k=3)
Out[7]: 4.497941445275415
```

Enfin, certaines fonctions peuvent avoir un nombre arbitraire de paramètres. Dans ce cas, les paramètres optionnels doivent *obligatoirement* être nommés. C'est le cas par exemple de la fonction `print` qui possède deux paramètres optionnels `sep` (valeur par défaut : un espace) qui est inséré entre chacun des arguments de la fonction et `end` (valeur par défaut : un passage à la ligne) qui est ajouté à la fin du dernier des arguments :

```
In [8]: print(1, 2, 3, sep='+', end='=6\n')
1+2+3=6
```

• Portée des variables

Fréquemment, de nouvelles variables sont définies et utilisées dans le bloc d'instructions d'une fonction. Néanmoins, celles-ci ne sont pas référencées au niveau global. En effet, chaque fonction possède sa propre table de référencement, ce qui permet d'en limiter la portée. Autrement dit, leur contenu est inaccessible depuis l'extérieur de la fonction et en particulier au niveau de l'interprète de commande.
De telles variables sont qualifiées de *locales*, par opposition aux variables *globales*, dont le contenu est accessible à tout niveau.

```
In [9]: a = 1            # définition d'une variable globale a

In [10]: def f():
    ...:     b = a        # définition d'une variable locale b
    ...:     return b

In [11]: f()
Out[11]: 1

In [12]: b
NameError: name 'b' is not defined
```

Figure 1 – Le contenu de la variable locale `b` n'est pas accessible en dehors de la fonction.

Il est même possible qu'une variable locale ait le même nom qu'une variable globale ; ce n'est vraiment pas souhaitable car générateur d'erreurs, mais sachez que par défaut, les variables définies à l'intérieur de la définition d'une fonction sont supposées locales. On s'en convaincra avec l'expérience présentée figure 2.

```
In [13]: def g():
    ...:     a = 2        # définition d'une variable locale a
    ...:     return a

In [14]: g()
Out[14]: 2

In [15]: a               # ici il s'agit de la variable globale définie ligne 9
Out[15]: 1
```

Figure 2 – Variables locales et globales peuvent porter le même nom.

Pour distinguer une variable locale d'une variable globale au sein de la définition d'une fonction, il faut suivre la règle suivante :

– si une variable se voit assigner une nouvelle valeur à l'intérieur de la fonction, cette variable est considérée comme locale (c'est le cas de `b` dans la définition de la ligne 10) ;

– si on se contente de faire appel au référencement d'une variable au sein d'une fonction, cette variable est considérée comme globale (c'est le cas de a dans la définition de la ligne 10).

Si vraiment on souhaite modifier le contenu d'une variable globale à l'intérieur du bloc d'instructions d'une fonction, il faut utiliser l'instruction `global` pour déclarer celles des variables qui doivent être traitées globalement (illustration figure 3).

```
In [16]: def h():
    ...:     global a     # déclaration d'une variable globale
    ...:     a = 2
    ...:     return a

In [17]: h()
Out[17]: 2

In [18]: a               # la variable globale définie ligne 9 a bien été modifiée
Out[18]: 2
```

FIGURE 3 – Une variable globale peut être modifiée au sein d'une fonction.

Cependant, il est déconseillé d'utiliser des variables globales, car leur usage favorise la *programmation spaghetti* [3] : elles compliquent la compréhension et la modification d'un script dès lors que ce dernier devient un peu long. En effet, la présence de variables globales empêche le programmeur de restreindre son analyse à une petite partie du code car une variable globale peut avoir été définie ou modifiée à tout endroit du script. Toute modification ou débogage demande donc d'analyser le code *dans son entier*. Il est bien souvent préférable d'utiliser au sein des fonctions des paramètres avec des valeurs par défaut plutôt que des variables globales.
L'exercice suivant devrait vous convaincre des difficultés que peuvent engendrer l'usage de variables globales lorsque celle-ci sont modifiées :

Exercice 1 On considère les trois fonctions :

```
def f():
    global a
    a = a + 1
    return a
```

```
def g():
    a = 1
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

Qu'affiche le shell lorsqu'on exécute le script suivant ?

```
a = 1
print(f(), a)
print(a, f())
print(a, g())
print(a, h())
```

1.2 Instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide de l'instruction `if` et prennent la forme suivante :

```
if expression booléenne:
    bloc.............
    d'instructions 1..
else:
    bloc.............
    d'instructions 2..
```

(Notez bien l'indentation qui permet de délimiter chacun des deux blocs d'instructions.)
Le fonctionnement de cette instruction est le suivant : si l'expression booléenne de la première ligne s'évalue en `True`, le premier bloc d'instructions est exécuté, si elle s'évalue en `False` c'est le second bloc qui est exécuté.

3. On qualifie ainsi et de manière péjorative un code désordonné, à l'image d'un plat de spaghettis : il suffit de tirer sur un fil d'un côté de l'assiette pour que l'enchevêtrement des fils provoque des mouvements jusqu'au côté opposé.

Notez que l'instruction **else** est optionnelle si aucune instruction ne doit être réalisée dans le cas d'un test négatif.
Rappelons qu'outre les opérateurs booléens **not**, **and** et **or**, les opérateurs suivants sont à valeurs booléennes :

- `x < y` (x est strictement plus petit que y) ;
- `x > y` (x est strictement plus grand que y) ;
- `x <= y` (x est inférieur ou égal à y) ;
- `x >= y` (x est supérieur ou égal à y) ;
- `x == y` (x est égal à y) ;
- `x != y` (x est différent de y).

On observera que le test d'égalité utilise les caractères == pour ne pas être confondu avec l'opérateur d'affectation.
Ces opérateurs permettent de comparer des nombres (de type *int* ou *float* mais également des chaînes de caractères, comparées suivant l'ordre lexicographique :

```
In [1]: 'alpha' < 'omega'
Out[1]: True

In [2]: 'gamma' <= 'beta'
Out[2]: False
```

Notons enfin que l'opérateur == est plus général encore puisqu'il permet de tester l'*égalité de valeur* [4] entre deux objets Python quelconque.

• Instructions conditionnelles multiples

En informatique il est fréquent qu'on ait à imbriquer plusieurs tests, aussi existe-t-il en python un mot clé **elif** (qui est la contraction de **else if**) et qui fonctionne suivant le schéma :

```
if expression booléenne 1:
    bloc..............
    d'instructions 1..
elif expression booléenne 2:
    bloc..............
    d'instructions 2..
else:
    bloc..............
    d'instructions 3..
```

- Si l'expression booléenne 1 s'évalue en `True`, le bloc d'instructions 1 est réalisé ;
- Si l'expression booléenne 1 s'évalue en `False` et l'expression booléenne 2 en `True`, le bloc d'instructions 2 est réalisé ;
- dans les autres cas, le bloc d'instructions 3 est réalisé.

Évidemment, plusieurs **elif** à la suite peuvent être utilisés pour multiplier les cas possibles, mais n'oubliez pas que les tests sont effectués *séquentiellement*, c'est à dire les uns après les autres jusqu'à trouver un test positif (ou arriver au cas final **else**).

Exercice 2 Rédiger une fonction `tri(a, b, c)` qui prend en arguments trois nombres a, b et c et qui retourne ces trois valeurs triées par ordre croissant.

2. Instructions itératives

Réaliser une itération, ou encore une *boucle*, c'est répéter un certain nombre de fois des instructions semblables. Dans la plupart des langages de programmation, il existe deux instructions pour réaliser une boucle, suivant qu'on peut calculer à l'avance le nombre d'itérations à réaliser (on parle alors de boucles *énumérées*) ou que le nombre d'itérations dépend de la réalisation ou non d'une certaine condition (on parle dans ce cas de boucles *conditionnelles*) ; le langage Python ne fait pas exception à la règle.

4. À ne pas confondre avec l'*égalité physique* qui se teste à l'aide de l'opérateur **is** (voir le chapitre précédent).

2.1 Boucles énumérées

• La fonction range

La fonction `range` peut prendre entre 1 et 3 arguments entiers :

- `range(b)` énumère les entiers $0, 1, 2, \ldots, b-1$;
- `range(a, b)` énumère les entiers $a, a+1, a+2, \ldots, b-1$;
- `range(a, b, c)` énumère les entiers $a, a+c, a+2c \ldots, a+nc$ où n est le plus grand entier vérifiant $a+nc < b$.

(On observera la similitude qui existe avec le *slicing* des chaînes de caractères décrit au chapitre précédent.)
Pour pouvoir illustrer le contenu de cette énumération, il faut ranger cette dernière dans une *liste* [5] :

```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: list(range(5, 15))
Out[2]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

In [3]: list(range(1, 20, 3))
Out[3]: [1, 4, 7, 10, 13, 16, 19]
```

• Boucles indexées

On définit une boucle indexée à l'aide de la fonction `for`, en suivant la structure suivante :

```
for ... in range(...):
    bloc ..........................
    ..............................
    d'instructions ................
```

Immédiatement après le mot-clé `for` doit figurer le nom d'une variable, qui va prendre les différentes valeurs de l'énumération produite par l'instruction `range`. Pour chacune de ces valeurs, le bloc d'instructions qui suit sera exécuté.
Un premier exemple très simple :

```
In [4]: for x in range(2, 10, 3):
   ...:     print(x, x**2)
2 4
5 25
8 64
```

Il est bien entendu possible d'imbriquer des boucles à l'intérieur d'autres boucles ; attention seulement à respecter les règles d'indentation pour délimiter chacun des blocs d'instructions :

```
In [5]: for x in range(1, 6):
   ...:     for y in range(1, 6):
   ...:         print(x * y, end=' ')
   ...:     print('/')
1 2 3 4 5 /
2 4 6 8 10 /
3 6 9 12 15 /
4 8 12 16 20 /
5 10 15 20 25 /
```

• Invariant de boucle

On appelle *invariant de boucle* toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération de la boucle ;

Conservation si cette assertion est vraie avant une itération de la boucle, elle le reste avant l'itération suivante ;

5. Une liste est une structure de données que nous étudierons en détail dans un chapitre ultérieur ; pour l'instant, retenons seulement qu'une liste est une collection ordonnée d'objets.

Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile qui aide à établir/prouver/analyser l'algorithme.

Comme l'indique la troisième propriété, un invariant de boucle peut être utilisé en amont de la rédaction d'un algorithme pour rédiger ce dernier sans erreur ; il peut aussi servir à prouver la validité d'un algorithme existant, et enfin il peut servir à analyser un algorithme pour en découvrir le rôle.

Exemple. Commençons par un exemple simple : on souhaite étudier la suite $(u_n)_{n\in\mathbb{N}}$ définie par la donnée de la valeur initiale $u_0 = 0$ et de la relation de récurrence $u_{k+1} = 2u_k + 1$ en définissant une fonction Python qui permette le calcul du terme de rang n de cette suite.
Considérons l'expression Python suivante : `x = 2 * x + 1`. Si la variable x référence initialement la valeur de u_k, après cette instruction cette variable référencera la valeur de u_{k+1}. D'où l'idée, pour calculer u_n, d'initialiser la variable x avec la valeur de u_0 puis d'appliquer n fois cette instruction. Ceci conduit à définir la fonction suivante :

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

Dans ce cas, on choisira pour prouver la validité de cette fonction d'énoncer l'invariant suivant :

à l'entrée de la boucle indexée par k, la variable x *référence la valeur de* u_k.

Cette assertion est vraie avant la première itération de la boucle (*initialisation*).
Si elle est vraie à l'entrée de la boucle indexée par k, la variable x référencera à l'entrée de la boucle suivante la valeur $2u_k + 1 = u_{k+1}$ donc cette assertion restera vraie à l'entrée de la boucle indexée par $k+1$ (*conservation*).
Ainsi, une fois la boucle terminée la variable x référencera la valeur de u_n, ce qui prouve la validité de cette fonction (*terminaison*).

Exemple. Considérons maintenant le calcul de $n!$. À l'image de l'exemple précédent, nous allons chercher à respecter l'invariant suivant :

à l'entrée de la boucle indexée par k, la variable x *référence la valeur de* $k!$.

Une fois cet invariant énoncé, il devient évident que le corps de la boucle doit contenir l'instruction : `x = x * (k + 1)`. Pour respecter l'initialisation, il est nécessaire que la variable x référence $0! = 1$, ce qui conduit à la rédaction de la fonction :

```
def fact(n):
    x = 1
    for k in range(n):
        x = x * (k + 1)
    return x
```

Exemple. Considérons enfin le calcul du terme de rang n de la suite de Fibonacci, définie par la donnée des valeurs initiales $u_0 = 0$, $u_1 = 1$ et la relation de récurrence $u_{k+2} = u_{k+1} + u_k$.
Compte tenu des deux exemples précédents, nous allons chercher à respecter l'invariant :

à l'entrée de la boucle indexée par k, la variable x *référence la valeur de* u_k.

Mais un problème apparaît rapidement : pour respecter ce seul invariant, il faut être capable de calculer u_{k+1} à l'aide de la seule valeur de u_k, ce qui semble difficile, pour ne pas dire impossible.

Énoncer cet invariant nous permet de mettre en évidence la nécessité de référencer en même temps que u_k la valeur de u_{k-1}. Ceci nous conduit à utiliser une deuxième variable y et à adopter l'invariant suivant :

à l'entrée de la boucle indexée par k, la variable x *référence la valeur de* u_k *et la variable* y *la valeur de* u_{k-1} *;*

Pour respecter ce nouvel invariant, il suffit que le corps de la boucle contienne l'instruction `x, y = x + y, x` et que les valeurs initiales de ces deux variables soient respectivement égales à $u_0 = 0$ et $u_{-1} = 1$. D'où la définition :

```
def fib(n):
    x, y = 0, 1
    for k in range(n):
        x, y = x + y, x
    return x
```

Exercice 3 Les exemples précédents ont montré comment l'énoncé d'un invariant en amont de la rédaction de l'algorithme pouvait servir à rédiger et à prouver la validité de ce dernier. Mais la recherche d'un invariant peut aussi servir à analyser un algorithme pour en deviner le rôle.

On considère un polynôme $p(x) = \sum_{i=0}^{n-1} a_i x^i$, représenté en PYTHON par le tableau `p = [a0, a1, a2, ...]`.
Déterminer un invariant pour établir le rôle de la fonction mystère ci-dessous :

```
def mystere(p, x):
    n = len(p)
    s = 0
    for k in range(n):
        s = x * s + p[n-1-k]
    return s
```

• Parcours d'une chaîne de caractères

Les boucles indexées que nous venons d'étudier ne sont qu'un cas particulier des possibilités offertes par la notion d'*énumération* en PYTHON. Celle-ci suit la syntaxe :

```
for ... in ...:
    bloc ..........................
    ..............................
    d'instructions ................
```

dans laquelle ce qui suit l'instruction `in` doit être une structure de données *énumérable*. Outre les énumérations engendrées la fonction `range`, nous avons déjà rencontré un autre exemple d'une telle structure de données : les chaînes de caractères [6]. Dans ce cas, la variable qui suit l'instruction `for` va prendre successivement la valeur des différents caractères qui composent cette chaîne.
Ce mécanisme est un des aspects remarquables du langage PYTHON, là où de nombreux langages de programmation n'autorisent que des itérations suivant une progression arithmétique. On appréciera la différence figure 4.

```
def epeler(mot):
    for c in mot:
        print(c, end=' ')
```

```
def epeler(mot):
    for i in range(len(mot)):
        print(mot[i], end=' ')
```

```
In [1]: epeler('Louis-Le-Grand')
L o u i s - L e - G r a n d
```

FIGURE 4 – Deux fonctions conduisant au même résultat.

Outre les intervalles et les chaînes de caractères, les listes, les tuples, les dictionnaires, les ensembles, les fichiers (et d'autres encore) sont des objets énumérables.

Autres itérateurs
Il est parfois nécessaire de connaître à la fois l'indice et la valeur d'un élément d'un objet énumérable ; on utilise dans ce cas l'instruction `enumerate`, qui retourne un couple formé de l'indice et de l'objet qui lui est associé :

6. Les listes, déjà évoquées dans une note précédente, en sont un autre exemple.

```
In [2]: for (i, c) in enumerate('Louis-Le-Grand'):
   ...:     print(i, c, sep='->', end=' ')
0->L 1->o 2->u 3->i 4->s 5->- 6->L 7->e 8->- 9->G 10->r 11->a 12->n 13->d
```

Enfin, il est possible d'énumérer deux énumérables en parallèle à l'aide de l'instruction `zip` (l'énumération se termine quand l'énumération du plus petit des deux énumérables est achevée).

```
In [3]: for (i, c) in zip(range(1, 9), 'Louis-Le-Grand'):
   ...:     print(i, c, sep='->', end=' ')
1->L 2->o 3->u 4->i 5->s 6->- 7->L 8->e
```

2.2 Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée ; elle peut donc tout aussi bien ne jamais réaliser cette suite d'instructions (lorsque la condition n'est pas réalisée au départ) que de les réaliser un nombre infini de fois (lorsque la condition reste éternellement vérifiée). La syntaxe d'une boucle conditionnelle est la suivante :

```
while condition:
    bloc ..........................
    ..............................
    d'instructions ................
```

La condition doit être une expression à valeurs booléennes. Par exemple,

```
In [1]: while 1 + 1 == 3:
   ...:     print('abc', end=' ')
   ...: print('def')
def
```

L'instruction `print('abc', end='')` n'est jamais exécutée puisque la condition est fausse.
En revanche, on évitera d'écrire dans l'interprète les instructions suivantes :

```
In [2]: while 1 + 1 == 2:
   ...:     print('abc', end=' ')
   ...: print('def')
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc ...
```

car la condition étant éternellement vérifiée cette suite d'instructions conduit à bloquer l'interprète.

Ainsi, sauf exception la condition dans une telle boucle va dépendre d'une variable au moins dont le contenu sera susceptible d'être modifié dans le corps de la boucle. Par exemple :

```
In [3]: x = 10

In [4]: while x > 0:
   ...:     print(x, end=' ')
   ...:     x -= 1
10 9 8 7 6 5 4 3 2 1
```

(rappelons que l'instruction `x -= 1` est équivalente à `x = x - 1`).

• Terminaison d'une boucle conditionnelle

Le fait qu'une boucle conditionnelle puisse boucler éternellement doit nous amener à nous préoccuper de la *terminaison* de celle-ci, c'est-à-dire être à même de prouver que celle-ci retourne un résultat *en un temps fini*. Là encore, c'est la recherche d'un invariant de boucle adéquat qui fournit le plus souvent la solution.
Considérons la définition suivante :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

et cherchons à déterminer son rôle (on suppose que les arguments a et b sont des entiers strictement positifs).

À l'évidence, la boucle conditionnelle réalise l'itération de deux suites $(q_n)_{n\in\mathbb{N}}$ et $(r_n)_{n\in\mathbb{N}}$ définies par la donnée des valeurs initiales $q_0 = 0$ et $r_0 = a$ et les relations : $q_{n+1} = q_n + 1$ et $r_{n+1} = r_n - b$. Il s'agit de deux suites arithmétiques, on en déduit l'invariant suivant :

à l'entrée de la boucle indexée par n la variable `q` *référence l'entier n et* `r` *l'entier $a - nb$.*

Sachant que $\lim(a - nb) = -\infty$, il existe un entier n vérifiant $a - nb < b$, ce qui prouve la *terminaison* de cette fonction.
En outre, on peut affirmer que cette fonction retourne la valeur d'un couple (q, r) vérifiant : $a = qb + r$ et $r < b \leqslant r + b$, soit encore $0 \leqslant r < b$. Autrement dit, cette fonction calcule le quotient et le reste de la division euclidienne de a par b.

Exercice 4 En déterminant un invariant, donner le rôle de la fonction suivante :

```
def mystere(n):
    i, s = 0, 0
    while s < n:
        s += 2 * i + 1
        i += 1
    return i
```

2.3 Forcer la sortie d'une boucle

Il existe deux façons de sortir prématurément d'une boucle : en utilisant l'instruction `return` (dans le cadre de la définition d'une fonction) ou l'instruction `break`.
Prenons l'exemple de la recherche d'un caractère particulier dans une chaîne de caractères : il faut parcourir la chaîne caractère par caractère en comparant à chaque fois le caractère de la chaîne avec celui que l'on cherche mais bien évidemment, une fois le caractère trouvé (s'il existe) il n'est plus nécessaire de continuer à parcourir le reste de la chaîne. On va donc écrire :

```
def cherche(c, chn):
    for x in chn:
        if x == c:
            return True
    return False
```

Si le caractère `c` ne figure pas dans la chaîne `chn`, celle-ci sera parcourue dans son entier, mais s'il figure dans la chaîne, le parcours sera interrompu dès la première occurrence rencontrée.

La fonction `break` quant à elle, permet d'interrompre le déroulement des instructions du bloc interne à la boucle pour passer à la suite. En règle générale, on évite d'employer à tort et à travers cette instruction car elle a souvent pour effet de rendre moins claires les conditions d'arrêt ; on fera néanmoins des exceptions le script s'en trouvera notablement simplifié.
C'est le cas en particulier lorsque la boucle consiste à attendre le déclenchement d'un événement exceptionnel. Pour illustrer cette situation, nous allons chercher à évaluer le nombre moyen de jets de deux dés à six faces qu'il faut réaliser avant d'obtenir un double-six.
Pour simuler un jet de dé, nous allons utiliser la fonction `randint(1, 7)` qui retourne un entier pris au hasard entre 1 et 6 (la fonction `randint` se trouve dans le module `numpy.random` ; il faut en charger la définition en mémoire au préalable). Le script qui consiste à simuler un jet de deux dés jusqu'à obtenir un double-six est :

```
s = 0
while True:
    s += 1
    if randint(1,7) == 6 and randint(1,7) == 6:
        break
```

Cette boucle a pour effet d'itérer la variable s jusqu'à la réalisation de l'événement attendu : un double-six.

Pour obtenir le nombre moyen de jets nécessaire à la réalisation de cet événement, il suffit de réaliser cette expérience un nombre suffisant de fois puis de calculer la moyenne de la valeur prise par s au sortir de cette boucle.

```
def test(n):
    e = 0
    for k in range(n):
        s = 0
        while True:
            s += 1
            if randint(1, 7) == 6 and randint(1, 7) == 6:
                break
        e += s
    return e / n
```

Voici quelques expériences avec 100 000 essais :

```
In [1]: test(100000)
Out[1]: 35.96509

In [2]: test(100000)
Out[2]: 36.19394

In [3]: test(100000)
Out[3]: 36.00176

In [4]: test(100000)
Out[4]: 35.98994
```

On obtient (c'était prévisible) un nombre moyen de lancer égal à 36.