Instructions itératives

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Structuration et indentation

Un bloc d'instructions PYTHON est défini par son indentation. Le début d'un bloc est marqué par un double-point (`:`) ; le retour à l'indentation de l'en-tête marque la fin du bloc.

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

Structuration et indentation

Un bloc d'instructions PYTHON est défini par son indentation. Le début d'un bloc est marqué par un double-point (`:`) ; 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 ............
    ..............................
    ..............................
```

Définition d'une fonction

On définit une fonction à l'aide du mot clé **def** :

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

Définition d'une fonction

On définit une fonction à l'aide du mot clé **def** :

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

Traditionnellement on distingue deux types de routines : les procédures ne retournent pas de résultat et se contentent d'agir sur l'environnement, les fonctions retournent un résultat.

Définition d'une fonction

On définit une fonction à l'aide du mot clé **def** :

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

Traditionnellement on distingue deux types de routines : les procédures ne retournent pas de résultat et se contentent d'agir sur l'environnement, les fonctions retournent un résultat.

En Python la distinction n'existe pas réellement : les procédures sont des fonctions qui retournent la valeur None.

Définition d'une fonction

On définit une fonction à l'aide du mot clé **def** :

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

Traditionnellement on distingue deux types de routines : les procédures ne retournent pas de résultat et se contentent d'agir sur l'environnement, les fonctions retournent un résultat.

L'interprète de commande IPython permet de faire la distinction : lorsqu'on applique une fonction qui retourne un résultat, ce dernier est précisé à la suite du mot Out[..].

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

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

Définition d'une fonction

On définit une fonction à l'aide du mot clé **def** :

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

Traditionnellement on distingue deux types de routines : les procédures ne retournent pas de résultat et se contentent d'agir sur l'environnement, les fonctions retournent un résultat.

Un résultat retourné par une fonction peut être réutilisé dans un calcul, à l'inverse d'une procédure :

```
In [3]: 1 + len("45")
Out[3]: 3

In [4]: 1 + print(45)
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule.

Pour définir la fonction $(x, y) \mapsto \sqrt{x^2 + y^2}$:

```
from numpy import sqrt

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

Une fois définie, une fonction s'utilise à l'instar de toute autre fonction prédéfinie :

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

En effet, $\sqrt{3^2 + 4^2} = 5$.

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule.

Pour définir la fonction $(x, y) \mapsto (x^k + y^k)^{1/k}$:

```
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
```

En effet, $\sqrt[3]{3^3 + 4^3} = 4,4979\ldots$.

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule.

Pour définir la fonction $(x, y) \mapsto (x^k + y^k)^{1/k}$:

```
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)
```

Une erreur est déclenchée dès lors que le nombre d'arguments donnés est incorrect.

Arguments d'une fonction

Arguments optionnels

Il est possible de préciser les valeurs par défaut que doivent prendre certains arguments. Par exemple :

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

Si on omet de préciser le troisième paramètre, ce dernier sera égal à 2 :

```
In [5]: norme(3, 4, 3)
Out[5]: 4.497941445275415

In [6]: norme(3, 4)
Out[6]: 5.0
```

Arguments d'une fonction

Arguments optionnels

Il est possible de préciser les valeurs par défaut que doivent prendre certains arguments. Par exemple :

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

Si on omet de préciser le troisième paramètre, ce dernier sera égal à 2 :

```
In [5]: norme(3, 4, 3)
Out[5]: 4.497941445275415

In [6]: norme(3, 4)
Out[6]: 5.0
```

Il est préférable de nommer les arguments optionnels pour éviter toute ambiguïté :

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

Arguments d'une fonction

Arguments optionnels

Certaines fonctions peuvent avoir un nombre arbitraire de paramètres. Dans ce cas, les paramètres optionnels doivent obligatoirement être nommés.
C'est le cas de la fonction **print** qui possède deux paramètres optionnels sep (valeur par défaut : `' '`) qui est inséré entre chacun des arguments de la fonction et end (valeur par défaut : `'\n'`) 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

Les variables définies dans une fonction ne sont accessibles *que dans la fonction elle-même*. De telles variables sont qualifiées de locales, par opposition aux variables globales.

```
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
```

La variable locale b n'est pas référencée au niveau global.

Portée des variables

Les variables définies dans une fonction ne sont accessibles *que dans la fonction elle-même*. De telles variables sont qualifiées de locales, par opposition aux variables globales.

Il est même possible qu'une variable locale ait le même nom qu'une variable globale (même si ce n'est pas souhaitable).

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

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

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

Portée des variables

Les variables définies dans une fonction ne sont accessibles *que dans la fonction elle-même*. De telles variables sont qualifiées de locales, par opposition aux variables globales.

Règle pour distinguer variable locale et variable globale :

- si une variable se voit assigner une nouvelle valeur à l'intérieur de la fonction, cette variable est considérée comme locale ;
- si on se contente de faire appel au référencement d'une variable au sein d'une fonction, cette variable est considérée comme globale.

Portée des variables

Les variables définies dans une fonction ne sont accessibles *que dans la fonction elle-même*. De telles variables sont qualifiées de locales, par opposition aux variables globales.

Si 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.

```
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 définie ligne 9 a bien été modifiée
Out[18]: 2
```

Portée des variables

Les variables définies dans une fonction ne sont accessibles *que dans la fonction elle-même*. De telles variables sont qualifiées de locales, par opposition aux variables globales.

Si 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.

```
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 définie ligne 9 a bien été modifiée
Out[18]: 2
```

En règle générale, il est conseillé de peu faire usage de variables globales.

Portée des variables

Exercice

```
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
```

Que produit le script suivant ?

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

Portée des variables

Exercice

```
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
```

Que produit le script suivant ?

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

Les arguments d'une fonction sont évalués de la gauche vers la droite.

Portée des variables

Exercice

```
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
```

Que produit le script suivant ?

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

```
2 2
```

Portée des variables

Exercice

```
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
```

Que produit le script suivant ?

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

```
2 2
2 3
```

Portée des variables

Exercice

```
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
```

Que produit le script suivant ?

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

```
2 2
2 3
3 2
```

Portée des variables

Exercice

```
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
```

Que produit le script suivant ?

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

```
2 2
2 3
3 2
UnboundLocalError: local variable 'a' referenced before assignment
```

Portée des variables

Exercice

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

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

On utilise la fonction `dis` du module du même nom qui désassemble le bytecode :

```
In [1]: dis.dis(f)
  4         0 LOAD_GLOBAL              0 (a)
            3 LOAD_CONST               1 (1)
            6 BINARY_ADD
            7 STORE_GLOBAL             0 (a)

  5        10 LOAD_GLOBAL              0 (a)
           13 RETURN_VALUE
```

```
In [2]: dis.dis(h)
  8         0 LOAD_FAST                0 (a)
            3 LOAD_CONST               1 (1)
            6 BINARY_ADD
            7 STORE_FAST               0 (a)

  9        10 LOAD_FAST                0 (a)
           13 RETURN_VALUE
```

Dans la fonction *h*, la variable locale a est référencée (`LOAD_FAST`) avant d'être assignée (`STORE_FAST`). Il n'y a pas d'erreur dans *f* puisque la variable globale a est déjà assignée lorsqu'elle est référencée par `LOAD_GLOBAL`.

Instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

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

Si l'expression booléenne est vraie, le premier bloc d'instructions est réalisé, dans le cas contraire c'est le second.

Instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

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

Si l'expression booléenne est vraie, le premier bloc d'instructions est réalisé, dans le cas contraire c'est le second.

Opérateurs courants (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).

Instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

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

Si l'expression booléenne est vraie, le premier bloc d'instructions est réalisé, dans le cas contraire c'est le second.

Les chaînes de caractères peuvent être comparées suivant l'ordre lexicographique :

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

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

Instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

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

Si l'expression booléenne est vraie, le premier bloc d'instructions est réalisé, dans le cas contraire c'est le second.

Expressions booléennes.

L'évaluation d'une expression logique n'est qu'un calcul dont le résultat ne peut prendre que deux valeurs : le vrai (`True`) et le faux (`False`).

Les deux fonctions suivantes sont équivalentes :

```
def est_pair(n):
    return n % 2 == 0
```

```
def est_pair(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

Instructions conditionnelles multiples

Il est possible d'imbriquer plusieurs tests à l'aide du mot-clé **elif** :

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

- Si l'expression 1 est vraie, le bloc d'instructions 1 est réalisé ;
- Si l'expression 1 est fausse et l'expression 2 vraie, le bloc d'instructions 2 est réalisé ;
- Si les deux expressions sont fausses, le bloc d'instructions 3 est réalisé.

Plusieurs **elif** à la suite peuvent être utilisés pour multiplier les cas possibles.

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 $a+nc < b$.

```
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 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 $a+nc < b$.

```
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]
```

- L'énumération `range(b)` comporte b éléments ;
- l'énumération `range(a, b)` comporte $b-a$ éléments.

Boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

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

Boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

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

Immédiatement après le mot clé **for** figure le nom d'une variable, qui va prendre les différentes valeurs de l'énumération produite par **range**. Pour chacune de ces valeurs, le bloc d'instructions qui suit sera exécuté.

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

Boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

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

Il est possible d'imbriquer des boucles à l'intérieur d'autres boucles :

```
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 /
```

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Exemple : $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

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

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Exemple : $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

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

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

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Exemple : $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

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

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

Initialisation : à l'entrée de la boucle indexée par 0, $x = 0 = u_0$.

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Exemple : $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

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

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

Conservation : si à l'entrée de la boucle indexée par k, $x = u_k$, alors à l'entrée de la boucle indexée par $k+1$, $x = 2u_k + 1 = u_{k+1}$.

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Exemple : $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

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

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

Terminaison : Le résultat retourné est la valeur de x à l'entrée de la boucle indexée par n, soit u_n.

Boucles énumérées

Invariant de boucle

$\longrightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

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

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

Exemple : $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

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

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

Ici, l'invariant de boucle énoncé prouve la validité de l'algorithme.

Boucles énumérées

Invariant de boucle

On souhaite calculer $u_n = n!$. Pour cela, on cherche à obtenir l'invariant :

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

Boucles énumérées

Invariant de boucle

On souhaite calculer $u_n = n!$. Pour cela, on cherche à obtenir l'invariant :

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

Cet invariant conditionne l'initialisation et la conservation :

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

Boucles énumérées

Invariant de boucle

On souhaite calculer $u_n = n!$. Pour cela, on cherche à obtenir l'invariant :

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

Cet invariant conditionne l'initialisation et la conservation :

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

Ici, l'invariant de boucle permet de rédiger l'algorithme.

Boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = u_{k+1} + u_k$.

On cherche à obtenir l'invariant :

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

Boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = u_{k+1} + u_k$.

On cherche à obtenir l'invariant :

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

Problème : comment calculer u_{k+1} à l'aide de la seule valeur de u_k ?
Il manque la valeur de u_{k-1}.

Boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = u_{k+1} + u_k$.

On utilise une deuxième variable y avec l'invariant :

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

Boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = u_{k+1} + u_k$.

On utilise une deuxième variable y avec l'invariant :

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

On en déduit la fonction :

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

Boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$, représenté par `p = [a0, a1, a2, ...]`.

Déterminer un invariant pour établir le rôle de la fonction :

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

Boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$, représenté par `p = [a0, a1, a2, ...]`.

Déterminer un invariant pour établir le rôle de la fonction :

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

À l'entrée de la boucle indexée par k, s référence $\sum_{i=n-k}^{n-1} a_i x^{i+k-n}$.

Boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$, représenté par `p = [a0, a1, a2, ...]`.

Déterminer un invariant pour établir le rôle de la fonction :

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

À l'entrée de la boucle indexée par k, s référence $\sum_{i=n-k}^{n-1} a_i x^{i+k-n}$.

On en déduit que cet algorithme retourne la valeur de $\sum_{i=0}^{n-1} a_i x^i = p(x)$.

Boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$, représenté par `p = [a0, a1, a2, ...]`.

Déterminer un invariant pour établir le rôle de la fonction :

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

À l'entrée de la boucle indexée par k, s référence $\sum_{i=n-k}^{n-1} a_i x^{i+k-n}$.

On en déduit que cet algorithme retourne la valeur de $\sum_{i=0}^{n-1} a_i x^i = p(x)$.

Ici, l'invariant de boucle permet d'analyser l'algorithme.

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

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

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers ...).

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

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

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers ...).

Par exemple :

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

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

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

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

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers ...).

Dans un langage de programmation n'autorisant que des itérations suivant une progression arithmétique, il faudrait écrire :

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

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

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

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers ...).

Il est possible d'énumérer à la fois l'indice et la valeur d'un élément :

```
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 ...
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

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

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers ...).

Il est possible d'énumérer deux énumérables à la fois :

```
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
```

L'énumération se termine quand l'énumération du plus petit des deux énumérables est achevée.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

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

La condition doit être une expression à valeurs booléennes.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
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.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

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

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

```
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 ...
```

La condition est éternellement vérifiée, ce qui conduit au blocage de l'interprète.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

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

La condition doit être une expression à valeurs booléennes.

En général, la condition dépend 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
```

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat *en un temps fini*. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

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

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat *en un temps fini*. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

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

La boucle conditionnelle réalise l'itération de deux suites $(q_n)_{n\in\mathbb{N}}$ et $(r_n)_{n\in\mathbb{N}}$ définies par $q_0 = 0$, $r_0 = a$ et : $q_{n+1} = q_n + 1$, $r_{n+1} = r_n - b$.

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat *en un temps fini*. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

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

La boucle conditionnelle réalise l'itération de deux suites $(q_n)_{n\in\mathbb{N}}$ et $(r_n)_{n\in\mathbb{N}}$ définies par $q_0 = 0$, $r_0 = a$ et : $q_{n+1} = q_n + 1$, $r_{n+1} = r_n - b$.
On en déduit l'invariant suivant :

> à l'entrée de la n^e boucle q contient l'entier n et r l'entier $a - nb$.

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat *en un temps fini*. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

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

La boucle conditionnelle réalise l'itération de deux suites $(q_n)_{n\in\mathbb{N}}$ et $(r_n)_{n\in\mathbb{N}}$ définies par $q_0 = 0$, $r_0 = a$ et : $q_{n+1} = q_n + 1$, $r_{n+1} = r_n - b$.
On en déduit l'invariant suivant :

> à l'entrée de la n^e boucle q contient l'entier n et r l'entier $a - nb$.

Terminaison : il existe un entier n vérifiant $a - nb < b$.

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat *en un temps fini*. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

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

Conclusion : cette fonction retourne la valeur d'un couple (q, r) vérifiant

$$\begin{cases} q = n \\ r = a - nb \end{cases} \text{ avec } a - nb < b \leqslant a - (n-1)b$$

soit encore :

$$a = qb + r \text{ avec } 0 \leqslant r < b.$$

C'est le quotient et le reste de la division euclidienne de a par b.

Terminaison d'une boucle conditionnelle

Exercice

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
```

Terminaison d'une boucle conditionnelle

Exercice

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
```

À l'entrée de la boucle de rang k, $i = k$ et $s = \sum_{i=0}^{k-1}(2i+1) = k^2$.

Terminaison d'une boucle conditionnelle

Exercice

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
```

À l'entrée de la boucle de rang k, $i = k$ et $s = \sum_{i=0}^{k-1}(2i+1) = k^2$.

Il existe un unique entier k tel que $(k-1)^2 < n \leqslant k^2$ donc l'algorithme se termine et retourne cette valeur de $k = \lceil \sqrt{n} \rceil$.

Forcer la sortie d'une boucle

Pour sortir prématurément d'une boucle : **`return`** ou **`break`**.
Exemple : recherche d'un caractère dans une chaîne de caractères.

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

Dès que le caractère c est trouvé dans la chaîne chn, le parcours cesse.

Forcer la sortie d'une boucle

Pour sortir prématurément d'une boucle : **`return`** ou **`break`**.
`break` permet d'interrompre le déroulement des instructions du bloc interne à la boucle.

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

La boucle ne se termine que si on réalise un double 6 (la terminaison n'est que *probable*).

Forcer la sortie d'une boucle

Pour obtenir le nombre moyen de jets nécessaire à l'obtention d'un double 6, on réalise cette expérience un nombre suffisant de fois :

```
from numpy.random import randint

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
```

Forcer la sortie d'une boucle

Pour obtenir le nombre moyen de jets nécessaire à l'obtention d'un double 6, on réalise cette expérience un nombre suffisant de fois :

```
from numpy.random import randint

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
```

```
In [5]: test(100000)
Out[5]: 35.96509

In [6]: test(100000)
Out[6]: 36.19394
```

```
In [7]: test(100000)
Out[7]: 36.00176

In [8]: test(100000)
Out[8]: 35.98994
```