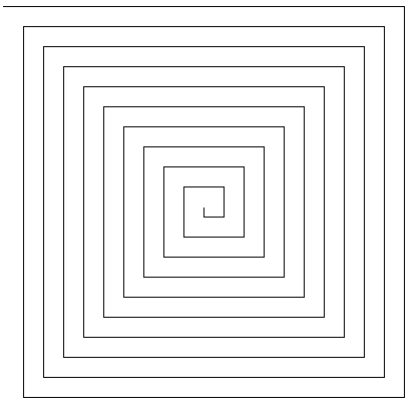
L'exécution d'un programme nécessite souvent la répétition d'un grand nombre d'instructions. Cependant, il n'est pas raisonnable d'écrire un programme en recopiant des millions de fois la même chose et, pour éviter de faire cela, on dispose de syntaxes dédiées.

1 Problème : Dessiner une spirale

On aimerait dessiner la spirale suivante :



Pour tracer cette figure avec Turtle, on voit facilement ce qu'il faut faire :

— on trace un trait
— on tourne de 90 degrés
— on trace un trait légèrement plus grand
— on tourne de 90 degré
— on trace un trait légèrement plus grand
— etc.

Il est donc possible d'en déduire un programme répondant au problème en écrivant :

```
from turtle import *

longueur = 10

forward(longueur)
right(90)
longueur = longueur + 10

<recopier 40 fois les lignes 5-7>
```

Ce programme répond au problème mais n'est pas très satisfaisant à cause de la répétition des commandes. De plus, il est difficile de le modifier pour dessiner une spirale plus grande.

2 Boucles bornées simples

Afin de corriger les inconvénients de la solution précédente, on peut utiliser une boucle bornée (ou boucle `for`). Essayer le programme suivant :

```
from turtle import *

longueur = 10

for i in range(40):
    forward(longueur)
    right(90)
    longueur = longueur + 10
```

On voit que l'on a réussi à répondre aux deux problèmes levés par la solution naïve :

— il n'y a pas de répétitions dans le code ;
— on comprend que le 40 dans `range(40)` correspond aux nombres de répétitions et on peut donc très facilement modifier le code pour faire plus de tours.

Boucle `for` : Pour répéter une partie de code `n` fois, on utilise la syntaxe suivante :

```
for i in range(n):
    <code a repeter>
```

Indentation : La partie du code à répéter doit être décalée d'une tabulation (touche avec deux flèches à gauche su clavier) par rapport au `for`, on parle d'indentation du code. Elle peut être constituée de plusieurs lignes (on parle alors de bloc ou séquence d'instructions) et peut faire appel à toutes les instructions Python. Dès que le code repasse à un niveau d'indentation inférieur (aligné avec le `for`) l'interpréteur considère que la partie à répéter est terminée.

Attention : Le `n` doit être un entier. Si ce n'est pas le cas, l'interpréteur affiche une erreur.

Essayer le programme suivant :

```
for i in range(10):
    print("Je suis dans la boucle.")
    print("Moi aussi.")

print("Pas moi !")

for i in range(5):
    print("Je suis dans une autre boucle, executee apres la premiere.")
```

Observons l'affichage :

- `"Je suis dans la boucle."` et `"Moi aussi."` sont affichés 10 fois en alternant car les deux `print` (l'un après l'autre) sont indentés après le premier `for`.
- `"Pas moi !"` n'est affiché qu'une fois car il n'est pas indenté et n'est donc pas affecté par le premier `for`.
- `"Je suis dans une autre boucle ..."` est affiché 5 fois car il est indenté après le deuxième `for`.

3 Utilisation de l'indice de boucle

Dans la commande `for i in range(10):` le `i` est en fait une variable (on peut d'ailleurs l'appeler comme on veut) qui va évoluer au cours des répétitions et dont le comportement dépend des paramètres donnés à `range`. Tester les programmes suivants :

```
for i in range(10):
    print("Valeur de i :", i)
```

```
for j in range(5,15):
    print("Valeur de j :", j)
```

```
for k in range(5,25,2):
    print("Valeur de k :", k)
```

Comportement de l'indice : Dans une boucle de la forme `for i in range(<paramètres>):`, `i` est appelé l'indice de la boucle et peut être utilisé dans la boucle pour savoir lors de son exécution à quelle étape on se trouve. Il varie de la façon suivante :

- Avec `for i in range(n):`, `i` varie de `0` à `n-1`.
- Avec `for i in range(b,e):`, `i` varie de `b` à `e-1`.
- Avec `for i in range(b,e,s):`, `i` varie de `b` à `e-1` (au maximum) avec un pas de `s`.

On en déduit un autre programme possible pour répondre au problème de la spirale :

```
from turtle import *

for i in range(40):
    forward(10+i*10)
    right(90)
```

En effet :

- dans le premier passage dans la boucle, `i=0` donc on avance de `10+0*10=10` ;
- dans le premier passage dans la boucle, `i=1` donc on avance de `10+1*10=20` ;
- dans le premier passage dans la boucle, `i=2` donc on avance de `10+2*10=30` ;
- etc.

Ce qui produit le même effet.

4 Accumulateurs

Quand on utilise une boucle, on peut utiliser des variables. Cela permet de faire des calculs mais aussi d'interagir de manière plus complexe avec l'utilisateur. Le programme suivant (à tester) permet à l'utilisateur de se déplacer point par point sur turtle :

```
from turtle import *

n = int(input("Combien de segments ?"))

for i in range(n):
    x = int(input("Entrer l'abscisse :"))
    y = int(input("Entrer l'ordonnee :"))
    goto(x,y)
```

On commence par demander à l'utilisateur combien de mouvements il compte faire puis, grâce à une boucle `for` de taille `n`, on lui demande l'abscisse et l'ordonnée de sa prochaine destination le bon nombre de fois.

Dans ce programme, les variables `x` et `y` sont redéfinies à chaque passage dans la boucle. À l'inverse, il est également possible de partir une certaine valeur et de modifier cette valeur plusieurs fois. Tester le programme suivant :

```
a = 15
for i in range(10):
    a = a+8
print(a)
```

Ici, la variable `a` est initialisée à la valeur 15 puis, 10 fois, on lui ajoute 10. À la fin, `a` contient donc $15 + 10 \times 8 = 95$. On a a accumulé des résultats intermédiaires dans `a` pour aboutir à un résultat : on dit alors que `a` est un accumulateur. Bien sûr, pour cet exemple, il aurait été plus rapide de calculer directement $15 + 10 \times 8$ mais on peut penser à des calculs plus complexes, par exemple le calcul de la somme $1 + 2 + 3 + ... + 100$. Tester le programme suivant :

```
somme = 0
for i in range(1,101):
    somme = somme + i
print(somme)
```

Analysons ce programme :

- D'abord, la variable `somme` est initialisée à `0`.
- Puis, pour des valeurs de `i` allant de `1` à `100` on ajoute la valeur de `i` à la variable `somme` donc :
  - au premier passage, `i=1` et `a` passe de `0` à `0+1` ;
  - au deuxième passage, `i=2` et `a` passe de `0+1` à `0+1+2` ;
  - ...
  - au dernier passage, `i=100` et `a` passe de `0` à `0+1+...+99` à `0+1+...+100`.
- On affiche donc la bonne valeur à la fin du programme.

La variable `somme` est encore un accumulateur.

Voyons un dernier exemple de programme utilisant un accumulateur. On veut permettre à un élève de calculer sa moyenne dans une matière (on suppose pour ne pas trop complexifier l'exemple que chaque note a le même coefficient) :

```
somme_notes = 0

nb_notes = int(input("Nombre de notes ?"))
for i in range(nb_notes):
    note = int(input("Entrer une note :"))
    somme_notes = somme_notes + note

moyenne = somme_notes/nb_notes
print("Votre moyenne est de", moyenne)
```

Ici, on demande à l'utilisateur combien de notes il va rentrer puis, à l'aide de la boucle, on lui demande toutes ses notes. On se sert alors de la variable `somme_notes` valant initialement zéro, pour calculer au fur et à mesure des passages dans la boucle la somme de toutes les notes. Pour finir on affiche la moyenne qui est égale à (Somme des notes)/(Nombre de notes).

Utiliser un accumulateur : On utilise un accumulateur pour réaliser un calcul qui peut s'effectuer étape par étape. On peut utiliser le squelette de programme suivant :

```
accumulateur = <valeur de depart>

for i in range(<nombre d etapes>):
    <mise a jour de l accumulateur>
```

Attention : Il ne faut surtout pas oublier d'initialiser l'accumulateur avant la boucle car sinon, lors du premier passage, celui-ci ne sera pas défini et Python affichera une erreur.

5 Boucles imbriquées

Il est possible d'utiliser une boucle dans une boucle et on parle dans ce cas de boucles imbriquées. Tester le programme suivant :

```
for i in range(5):
    print("Boucle sur i")
for j in range(10):
    print("Boucle sur j")
```

Rien de nouveau ici : on a deux boucles de tailles 5 et 10 qui s'exécutent l'une après l'autre, ce qui donne 15 affichages : 5 pour `i` et 10 pour `j`. Essayer le programme suivant :

```
for i in range(5):
    print("Boucle sur i")
    for j in range(10):
        print("Boucle sur j")
```

Le résultat est complètement différent. Cette fois en effet, on voit grâce à l'indentation que la boucle sur `j` est lancée non pas après mais à l'intérieur de la boucle sur `i` donc à chaque passage dans celle-ci. Ce qui donne 5 affichages pour la boucle `i` et 10 exécutions de la boucle `j` par passage dans la boucle `i`, soit 50 affichages pour la boucle `j`.

Boucles imbriquées : Pour l'instant on n'utilisera des boucles imbriquées que dans des cas très simples. On voit cependant qu'il est très important d'indenter correctement son code Python lorsque l'on utilise plusieurs boucles :

— si les deux boucles sont au même niveau d'indentation, elle vont s'exécuter l'une après l'autre ;

— si la deuxième boucle se trouve à l'intérieur de la première, elle va s'exécuter à chaque passage dans celle-ci.