1 Introduction

Les **structures de boucle** sont des structures algorithmiques qui permettent de **répéter** des instructions.

Le nombre de répétitions que l'on souhaite réaliser peut être prédéfini à l'avance. On parle alors de **boucle itérative**, ou **boucle bornée**.

Mais, le nombre de répétitions que l'on va réaliser peut aussi être inconnu, car dépendant d'une condition qui doit être vérifiée. Dans ce cas, on parle alors de **boucle conditionnelle**, ou **boucle non bornée**.

2 Boucles itératives (ou boucles bornées)

2.1 Syntaxe et exemples

En programmation, comme dans la vie courante, on s'aperçoit que l'on est amené, dans certains cas, à répéter les mêmes instructions.

Exemple 2.1 On place une somme de 1000 euros sur un compte rémunéré à 2% (on multiplie donc chaque année la somme disponible sur le compte par 1,02) et on aimerait savoir la somme dont on disposerait dans 10 ans.

- Une première approche consisterait à définir onze variables `somme0`, `somme1`, `somme2`, ... correspondant chacune à la somme disponible sur le compte au début, à la 1ère année, à la 2ème année, etc. On écrirait donc le code suivant :

```
somme0 = 1000
somme1 = somme0 * 1.02
somme2 = somme1 *1.02
somme3 = somme2 * 1.02
...
```

  On s'aperçoit tout de suite que cela est fastidieux. De plus, si on veut connaître la somme disponible dans 25 ans, il faudrait utiliser vingt-six variables !!!
- Pour corriger ce problème, on utilise une seule variable `somme` que l'on réactualise chaque année. On aura donc le code suivant :

```
somme = 1000 # somme déposée
somme = somme*1.02 # somme au bout dune année
somme = somme*1.02 # somme au bout de deux années
somme = somme*1.02 # somme au bout de trois années
...
```

- Si on a limité le nombre de variables, on n'a toujours pas diminué le nombre de lignes de code et donc le travail reste toujours fastidieux. Toutefois, on remarque que l'on écrit à chaque ligne toujours la même instruction : `somme = somme*1.02`. On répète donc la même instruction dix fois.

Cette répétition d'instructions est une structure algorithmique connue sous le nom de **boucle itérative** (ou **boucle bornée** ou encore **boucle pour**).

Définition 2.2 Une *boucle itérative* permet de répéter une séquence d'instructions un nombre fixé de fois.

On donne ci-dessous la structure algorithmique d'une telle boucle, ainsi que son implémentation `Python`.

```
pour compteur allant de valeur_initiale à valeur_finale
   bloc d'instructions
fin pour
```

```
for compteur in range(valeur_initiale, valeur_finale + 1):    # ne pas oublier les :
   bloc d'instructions                                        # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

Remarque 2.3

- Si `valeur_finale < valeur_initiale`, il n'y a pas d'erreur : la boucle ne s'exécute simplement pas et on passe directement à la suite de l'algorithme écrite après le `fin pour`.
- En `Python`, l'instruction `range(valeur_initiale, valeur_finale + 1)` définit une plage de valeurs entières consécutives commençant à `valeur_initiale` et se terminant à `valeur_finale`. Ainsi :
  - l'instruction `range(1,6)` correspond à la plage de valeurs 1, 2, 3, 4, 5.
  - l'instruction `range(0,3)` correspond à la plage de valeurs 0, 1, 2.

  Dans le cas où `valeur_initiale = 0`, on peut simplement écrire `range(valeur_finale + 1)` au lieu de `range(0, valeur_finale + 1)`. Ainsi, `range(0,3)` est la même instruction que `range(3)`.

Exemple 2.4 On reprend l'énoncé de l'exemple 2.1 : il s'agit de répéter dix fois l'instruction `somme = somme*1.02`. On donne ci-dessous la version algorithmique et l'implémentation `Python` d'une fonction `placement` qui renvoie la somme disponible sur le compte au bout de 10 ans.

Pseudo-code :

```
fonction placement()
  somme ← 1000
  pour i allant de 1 à 10
    somme ← somme × 1.02
  fin pour
  renvoyer somme
fin fonction
```

Python :

```
def placement():
    somme = 1000
    for i in range(1,11):
        somme = somme * 1.02
    return somme
```

Remarque. En `Python`, on aurait pu remplacer l'instruction `range(1,11)` par l'instruction `range(10)`, puisqu'il faut simplement que la boucle soit réalisée dix fois, c'est-à-dire que la variable `i` prenne dix valeurs.

2.2 Exercices

Exercice 2.5 Compléter le Notebook *NSI Première Partie 1 Chapitre 5 Boucles itératives*.

Exercice 2.6 (QCM)

1. On considère l'instruction suivante : `for i in range(8)`. Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) `i` parcourt les valeurs de `1` à `8`
   (b) `i` parcourt les valeurs de `0` à `8`
   (c) l'instruction est identique à : `for i in range(1,8)`
   (d) `i` parcourt les valeurs de `0` à `7`

2. On considère la fonction suivante :

```
def somme(n):
    S = 0
    for i in range(n):
        S = S + i
    return S
```

   Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) `somme(n)` renvoie la somme des entiers de `0` à `n`
   (b) `somme(0)` renvoie `0`
   (c) la fonction admet trois variables locales
   (d) `somme(4)` renvoie `6`

3. On considère la fonction suivante :

```
def fact(n): # n entier >=1
    p = 1
    for i in range(1,n+1):
        p = p*i
        return p
```

   Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) `fact(n)` renvoie la valeur de `1*2*...*n`
   (b) `fact(n)` renvoie toujours `1`
   (c) `i` parcourt les valeurs de `1` à `n+1`
   (d) le code n'est pas correct

4. On considère la fonction suivante :

```
1 from random import randint
2
3 def six(n):
4     cpt = 0
5     for i in range(n):
6         de = randint(1,6)
7         if de == 6:
8             cpt = cpt + 1
9     return cpt
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) les lignes 6 et 7 peuvent être remplacées par : `if randint(1,6) == 6:`

(b) la fonction simule `n` lancers d'un dé équilibré à 6 faces et renvoie le nombre de fois où le 6 est apparu au cours de ces `n` lancers

(c) `six(4)` renvoie un nombre entier entre `0` et `4`

(d) la fonction peut renvoyer `0`

5. On considère la fonction suivante :

```
def double_boucle(n):
    S = 0
    for i in range(n):
        for j in range(n):
            S = S + 1
    return S
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) `double_boucle(3)` renvoie `3`

(b) `double_boucle(0)` renvoie `0`

(c) `double_boucle(n)` renvoie n^2

(d) `double_boucle(3)` renvoie `9`

3 Boucles conditionnelles (ou boucles non bornées)

3.1 Syntaxe et exemples

Exemple 3.1 Reprenons l'exemple du placement de l'exemple 2.1 : on place une somme de 1000 euros sur un compte rémunéré à 2% (on multiplie donc chaque année la somme disponible sur le compte par 1,02). Maintenant, au lieu de déterminer la somme disponible sur le compte au bout d'un certain nombre d'années, on aimerait plutôt savoir combien d'années on doit attendre pour disposer d'une somme d'au moins 2000 euros.

- On raisonne comme précédemment en définissant une variable `somme = 1000` et en exécutant plusieurs fois l'instruction `somme = somme*1.02` jusqu'à ce que la variable `somme` contienne une valeur supérieure ou égale à `2000`. Il ne nous reste plus qu'à compter le nombre d'instructions que l'on a réalisées.
- A nouveau, nous sommes dans le cadre d'une répétition d'instructions, c'est-à-dire dans le cadre d'une structure de boucle. Mais, à la différence des boucles itératives abordées au paragraphe précédent, on ne sait pas ici à l'avance combien de fois on va devoir répéter les instructions. Ce nombre de répétitions dépend en effet d'une condition (*ma somme actuelle est-elle supérieure ou égale à* 2000 *?*) que l'on doit vérifier à chaque fois que l'on veut répéter une instruction supplémentaire.

Cette structure de boucle est connue sous le nom de **boucle conditionnelle** (ou **boucle non bornée** ou encore **boucle tant que**).

Définition 3.2 Une *boucle conditionnelle* permet de répéter une séquence d'instruction tant qu'une condition est vérifiée.

On donne ci-dessous la structure algorithmique d'une telle boucle, ainsi que son implémentation `Python`.

```
tant que condition vraie
   bloc d'instructions
fin tant que
```

```
while condition vraie:    # ne pas oublier les :
   bloc d'instructions    # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

Remarque 3.3

- La boucle arrête de s'exécuter dès que la condition qui doit être vérifiée devient fausse.
- Si la condition est toujours vraie, on a une *boucle infinie*, ce qui peut planter le programme.

Exemple 3.4 On reprend l'énoncé de l'exemple 3.1 : il s'agit de répéter l'instruction `somme = somme*1.02` jusqu'à ce que la variable `somme` contienne une valeur supérieure ou égale à `2000`. On donne ci-dessous la version algorithmique et l'implémentation `Python` d'une fonction `seuil_placement` qui renvoie le nombre d'années à attendre pour que la somme disponible sur le compte soit d'au moins 2000 euros.

Pseudo-code :

```
fonction seuil_placement()
   somme ← 1000
   nb_annees ← 0
   tant que somme < 2000
      somme ← somme × 1.02
      nb_annees ← nb_annees + 1
   fin tant que
   renvoyer nb_annees
fin fonction
```

Python :

```
def seuil_placement():
   somme = 1000
   nb_annees = 0
   while somme < 2000:
      somme = somme * 1.02
      nb_annees = nb_annees + 1
   return nb_annees
```

Remarque. La boucle tant que s'arrêtant lorsque la condition est fausse et continuant lorsqu'elle est vraie, on choisit comme condition à tester le contraire de celle que l'on veut afin de pouvoir réaliser les instructions voulues. Dans notre cas, on veut s'arrêter lorsque la variable `somme` contient une valeur supérieure ou égale à `2000` et continuer lorsque `somme < 2000`. C'est donc cette dernière que l'on va utiliser comme test.

3.2 Exercices

Exercice 3.5 Compléter le Notebook *NSI Première Partie 1 Chapitre 5 Boucles conditionnelles*.

Exercice 3.6 (QCM)

1. On considère l'instruction suivante : `while i%2 == 1`. Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) si `i` contient la valeur `4`, alors la boucle continue
   (b) si `i` contient la valeur `7`, alors la boucle s'arrête
   (c) la boucle s'arrête lorsque `i` est impair
   (d) la boucle s'arrête lorsque `i` est pair

2. On considère la fonction suivante :

```
def seuil():
    n = 1
    a = 0
    while n < 8:
        n = n + a
        a = a + 2
    return a
```

   Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) la fonction renvoie `13`
   (b) la fonction renvoie `8`
   (c) la fonction renvoie `6`
   (d) la fonction renvoie la plus petite valeur de `a` pour laquelle `n` est supérieur ou égal à `8`

3. On considère la fonction suivante :

```
from random import randint

def obt1():
    de = 0
    lancer = 0
    while de != 1:
        de = randint(1,6)
        lancer = lancer + 1
    return lancer
```

   Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) la fonction renvoie toujours la même valeur
   (b) la boucle s'arrête lorsque `de` contient la valeur `1`
   (c) la fonction peut renvoyer `0`
   (d) la fonction simule le lancer d'un dé équilibré à 6 faces et renvoie le nombre de lancers nécessaire pour obtenir `1` pour la première fois

4. On considère la fonction suivante :

```
from random import randint

def recherche2():
    lancer = 1
    while lancer <= 5:
        if randint(1,6) == 2:
            return lancer
        lancer = lancer + 1
    return -1
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) la fonction renvoie l'une des 6 valeurs suivantes : `-1`, `1`, `2`, `3`, `4` ou `5`

(b) la fonction renvoie l'un des nombres entiers entre `-1` et `5`

(c) si la fonction renvoie `-1`, alors la variable `lancer` contient la valeur `6`

(d) la fonction simule 5 lancers d'un dé équilibré à 6 faces et renvoie, soit le numéro du premier lancer pour lequel on a obtenu la face 2, soit `-1` si on n'a jamais obtenu la face 2 au cours des 5 lancers

5. La fonction suivante permet de simuler le lancer de deux dés équilibrés à 6 faces :

```
from random import randint

def double3():
    de1 = 0
    de2 = 0
    lancer = 0
    while ... :
        de1 = randint(1,6)
        de2 = randint(1,6)
        lancer = lancer + 1
    return lancer
```

Parmi les conditions proposées ci-dessous, quelles sont celles qui permettent à cette fonction de renvoyer le nombre de lancers nécessaires pour obtenir un double 3 ?

(a) `de1 != 3 or de2 != 3`

(b) `de1 + de2 != 6`

(c) `not(de1 == 3 and de2 == 3)`

(d) `de1 == 3 and de2 == 3`

4 Exercices complémentaires

Exercice complémentaire 4.1 On donne la fonction `equi` suivante. Tester la et expliquer ce qu'elle fait :

```
from turtle import *

def equi(cote) :
    setup(200,200)
    for i in range(3):
        forward(cote)
        right(120)
```

Exercice complémentaire 4.2 (Figures géométriques) Dans une fenêtre graphique de taille 600×600 :

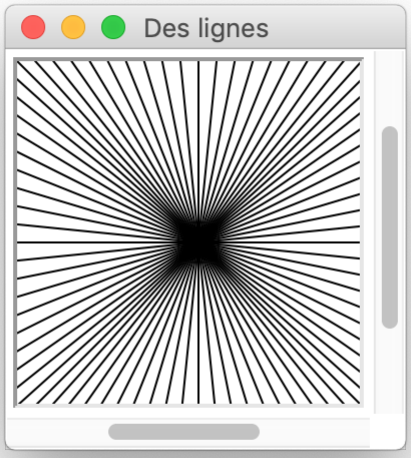
1. Ecrire un programme permettant de tracer un carré de côté 100 avec la tortue.
2. Modifier le programme précédent pour tracer un pentagone régulier de côté 50.
3. Améliorer encore le programme en écrivant une fonction `polygoneRegulier` prenant en entrée le nombre de côté et leur longueur afin de tracer un polygone régulier.

Exercice complémentaire 4.3 (Cercle, carré ou triangle ?) Compléter la fonction suivante :

```
from turtle import *

def Trace(figure):
    """Trace la figure choisie (un carre, un cercle ou un triangle),
    la variable figure est une chaine de caractères.
    Si la chaine de caractères saisie n'est pas "carre" ou "cercle",
    la figure tracée sera un triangle."""
    if figure == "cercle":
        circle(100)
    elif figure == "carre":
        ..........
    else:
        ..........
```

Exercice complémentaire 4.4 (Des lignes) Ecrire un programme permettant de réaliser, dans une fenêtre de taille 200×200, le dessin suivant :



Modifier ensuite ce programme pour qu'il puisse fonctionner avec n'importe quelle fenêtre graphique.

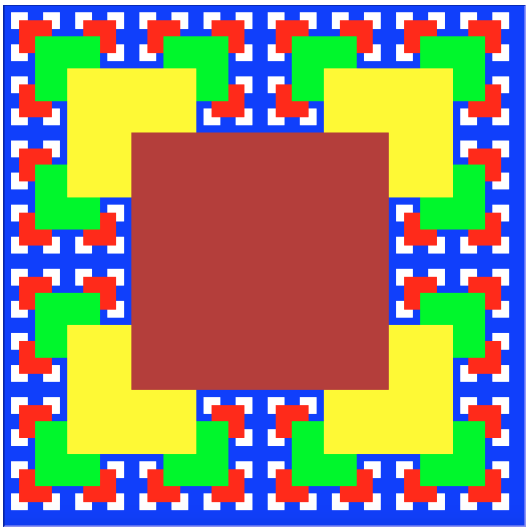
Exercice complémentaire 4.5 (Des étoiles)

1. Ecrire une fonction `etoile5` traçant une étoile à cinq branches dont la longueur `taille` d'une branche est donnée.
2. Utiliser cette fonction pour reproduire le modèle ci-dessous :

Exercice complémentaire 4.6 (Echiquier)

1. Ecrire un programme permettant de tracer un échiquier dans une fenêtre de taille 800×800.
2. Modifier le programme précédent afin de pouvoir tracer un échiquier dont la taille est adaptée automatiquement à la taille de la fenêtre graphique dont les dimensions sont fournies par l'utilisateur.

Exercice complémentaire 4.7 (Pyramides chinoises) A l'aide d'une fonction traçant un carré de dimensions et de couleur donnés et de boucles, reproduire les pyramides chinoises suivantes (les couleurs sont au choix). On pourra utiliser une liste pour gérer les couleurs.



Exercice complémentaire 4.8 Tester la fonction ci-dessous et expliquer chaque ligne de code.

```
from turtle import *

def spirale():
    speed("fastest")
    rayon = 1
    rayonS = 100
    while (rayon < rayonS):
        circle(rayon, 180)
        rayon += 2
```

Exercice complémentaire 4.9 (Rebonds) Ecrire un programme permettant de simuler, dans une fenêtre graphique de taille 400×400, le rebond d'une balle sur les deux côtés verticaux (utilisez les fonctions `shape` et `clear`). On se limitera à trois allers-retours.