CHAPITRE 11

Les boucles

On ne sait prévoir que des répétitions et comprendre, c'est dégager le quelque chose qui se répète.

Antoine de Saint-Exupéry [1]

Thème : Langages et programmation

Contenus	Capacités attendues	Commentaires
Constructions élémentaires	Mettre en évidence un corpus de constructions élémentaires.	Séquences, affectation, conditionnelles, boucles bornées, boucles non bornées, appels de fonction.

I. Introduction

Activité 1.
Le jour de sa naissance, la grand-mère de Robin dépose sur un compte en banque bloqué, la somme de 1 000 euros. Les intérêts, qui s'élèvent à 1,75 % par an, sont acquis sur le compte tous les ans. Le jour anniversaire de la création du compte, le montant des interêts est calculé sur le montant disponible sur le compte et est déposé dessus. En supposant qu'il n'effectue pas de nouveau dépôt, calculez le montant disponible sur son compte :

1. le jour de son premier anniveraire :
2. le jour de son deuxième anniversaire :
3. le jour de sa majorité.

Définition

Une boucle est une structure permettant de répéter une séquence d'instructions tant qu'une condition est vraie. On parle d'itérations.

II. Les boucles bornées

Les boucles bornées sont des structures itératives dont on connaît à l'avance le nombre d'itérations. La séquence d'instructions est exécutée **pour** toutes les valeurs d'une structure itérable (c'est à dire dont on peut parcourir les éléments).

1. Carnets - 1953 (`https://www.antoinedesaintexupery.com/ouvrage/carnets-1953/`)

II. 1. structure algorithmique

```
1 début
2 |  pour variable dans itérable faire
3 |  |  bloc à exécuter
4 |  |  si la variable est dans l'itérable
5 |  fin pour
6 |  Instructions qui seront TOUJOURS exécutées
7 fin
```

Exemple 1

Algorithme : somme(n)

```
/* algorithme qui calcule la somme des n premiers entiers          */
```
Entrées : n : un nombre
Sorties : *somme* : un nombre
```
/* Variables                                                       */
/* n, somme : des entiers                                          */
```

```
1 début
2 |  somme ← 0
3 |  pour i dans [1, n] faire
4 |  |  somme ← somme + i
5 |  fin pour
6 |  renvoyer somme
7 fin
```

Activité 2.

1. Complétez l'historique d'exécution lorsque n prend la valeur 4. (voir tableau ci-dessous).

2. Combien de fois la ligne 3 est-elle exécutée ?

3. Combien de fois la ligne 4 est-elle exécutée ?

II. 2. Implémentation en Python

Les mots-clé utilisés sont simplement `for` et `in`.

```
for element in objet_iterable:
    blocs d'instructions
```

Remarque

- ▷ La ligne commençant par le mot clef `for` doit être terminée par un double-point :
- ▷ les instructions à exécuter si la condition est vraie doivent être indentées.

II. 3. Usage de la fonction `range`

La fonction `range` génère une structure itérable. Elle contient 3 paramètres :

- ▷ *debut* : la borne inférieure ;
- ▷ *fin* : la borne supérieure **toujours exclue** ;
- ▷ *pas* : le pas d'incrémentation.

`range(0, 10, 1)` va renvoyer une structure itérable dont les éléments sont 0, 1, 2, ... , 9.

Exemple 2

```
def somme(n):
somme = 0
for i in range (1, n+1) :
somme = somme + i
return somme
```

Remarque

- ▷ Le paramètre *debut* est optionnel. Par défaut, il vaut 0.
- ▷ La paramètre *pas* est optionnel. Par défaut, il vaut 1.

L'instruction `range(0, 10, 1)` est équivalente à `range(10)`.

Activité 3.
Rédigez une fonction python *telephone*, prenant un entier n en paramètre, et affichant n fois `allô` avant d'afficher `t'es où ?`.

Activité 4.
On reprend l'activité 1.

1. Créez une fonction *compte*, prenant n, le nombre d'années, en argument, et renvoyant le montant disponible au bout de n années.
2. Quelle instruction permettrait d'afficher la somme disponible au bout de 1 an, de 2 ans, de 18 ans ?

La suite page suivante

Suite de l'exercice 0.

3. Modifiez la fonction précédente afin qu'elle prenne en entrée également la somme d'argent présente sur le compte à la naissance.
4. À l'aide de cette fonction, calculez la somme qui sera disponible sur son compte à sa majorité avec un versement initial de 3 000 euros.

III. Les boucles non bornées

Les boucles non bornées sont des structures itératives dont on ne connaît pas à l'avance le nombre d'itérations. La séquence d'instructions est exécutée **tant que** l'expression testée est vraie (le test de l'expression renvoie `True`).

III. 1. structure algorithmique

```
1 début
2     tant que expression faire
3         bloc a exécuter
4         si l'expresison est vraie
5     fin tq
6     Instructions qui seront TOUJOURS exécutées
7 fin
```

Exemple 3

Algorithme : difference

```
/* algorithme qui calcule la différence entre deux entiers        */
Entrées : x, y : des entiers
Sorties : difference : des entiers
/* Variables                                                       */
/* x, y, difference : des entiers                                  */

1 début
2 |   difference ← 0
3 |   tant que x - y ≠ 0 faire
4 |   |   difference ← difference + 1
5 |   |   y ← y + 1
6 |   fin tq
7 |   renvoyer difference
8 fin
```

III. 2. Implémentation en python

Le mot-clé utilisé est `while`.

```
1 while condition :
2     blocs d'instructions
```

Remarque

- ▷ La ligne commençant par le mot clef `while` doit être terminée par un double-point :
- ▷ les instructions à exécuter si la condition est vraie doivent être indentées.

Exemple 4

```
1 def difference(x, y):
2     difference = 0
3     while x - y != 0:
4         difference = difference + 1
5         y = y + 1
```

IV. Le problème de l'arrêt

On considère l'algorithme suivant, permettant de calculer 2^n :

Algorithme : puissance_2(n)

```
/* algorithme qui calcule 2^n                                   */
Entrées : n : un entier
Sorties : resultat : un entier
/* Variables                                                    */
/* n, resultat : des entiers                                    */
1 début
2     resultat ← 1
3     tant que n ≠ 0 faire
4         resultat ← 2× resultat
5         n ← n - 1
6     fin tq
7     renvoyer resultat
8 fin
```

Activité 5.

1. Identifiez une valeur de n pour laquelle la boucle se termine.
2. Identifiez une valeur de n pour laquelle la boucle ne se termine pas.

Bonnes Pratiques

La condition qui régit une boucle `while` doit nécessairement être modifiée à l'intérieur de celle-ci. Dans le cas contraire, on appelle une telle boucle **une boucle infinie** puisqu'il est impossible d'en sortir.

Je retiens

- ▷ Une itération est la répétition d'une séquence d'instructions.
- ▷ Un itérable est une strcuture de données que l'on peut parcourir une par une.
- ▷ La boucle `for ... in` est une boucle bornée. Elle est utilisée lorsqu'on connaît à l'avance le nombre d'itérations que l'on souhaite faire.
- ▷ L'instruction `range(debut, fin, pas)` permet de créer une structure de données itérable. Les paramètres *debut* et *pas* sont optionnels. La valeur correspondant au paramètre *fin* est exclu.
- ▷ La boucle `while` est une boucle non bornée. Elle est utilisée lorsqu'on ne connaît pas à l'avance le nombre d'itérations que l'on souhaite faire.
- ▷ Les boucles sont à manier avec précautions. Elle peuvent ne jamais finir. C'est le problème de l'arrêt.

V. Exercices

Exercice 1 (QCM).

1. Combien de fois la fonction `print` est-elle appelée dans le code Python qui suit ?

```
n = 4
for i in range(2, n):
    print(i)
```

- ☐ Jamais
- ☐ Une fois
- ☐ Deux fois
- ☐ Trois fois

2. Voici un code Python :

```
x = 1
for i in range(4):
    x = x + i
```

Quelle est la valeur finale de x ?

- ☐ 6
- ☐ 7
- ☐ 10
- ☐ 11

3. Après le code Python qui suit, quelles sont les valeurs finales de x et de y ?

```
x = 4
while x > 0:
    y = 0
    while y < x:
        y = y + 1
        x = x - 1
```

- ☐ La valeur finale de x est -1 et celle de y est 0.
- ☐ La valeur finale de x est 0 et celle de y est 0.
- ☐ La valeur finale de x est 0 et celle de y est 1.
- ☐ La boucle `while x > 0` est une boucle infinie. Le programme ne se termine jamais.

4. On a saisi le code suivant :

```
n = 8.0
while n > 1.0 :
    n = n / 2
```

Quelle est la valeur de n après l'exécution du code ?

- ☐ 4.0
- ☐ 2.0
- ☐ 1.0
- ☐ 0.5

Exercice 2.
Codez en Python les algorithmes suivants :

1.

Fonction racine(a)

```
/* renvoie la racine carré de a                                    */
```
Entrées : a : un réel
Sorties : x : un réel

```
/* Variables                                                       */
/* a, x : des réels                                                */
/* i : un entier                                                   */
```

```
1 début
2 |  x ← 1
3 |  pour 0 ⩽ i ⩽ 9 par pas de 1 faire
4 |  |  x ← (x + a/x) / 2
5 |  fin pour
6 |  renvoyer x
7 fin
```

2.

Fonction augmentation(quantite_initiale, nb_annee)

```
/* renvoie la quantité initiale augmentée de 33 % par an           */
```
Entrées : *quantite_initiale* : un réel
nb_annee : un entier
Sorties : *quantite_finale* : un réel

```
/* Variables                                                       */
/* quantite_initiale, quantite_finale : des réels                  */
/* i, nb_annee : des entiers                                       */
```

```
1 début
2 |  quantite_finale ← quantite_initiale
3 |  pour 1 ⩽ i ⩽ nb_annee par pas de 1 faire
4 |  |  quantite_finale ← quantite_finale ×1,33
5 |  fin pour
6 |  renvoyer quantite_finale
7 fin
```

3.

Fonction compte bits(valeur)

```
/* renvoie le nombre de bits nécessaire pour coder un entier       */
```
Entrées : *valeur* : un entier
Sorties : *nb_bits* : un entier

```
/* Variables                                                       */
/* nb_bits, valeur : des entiers                                   */
```

```
1 début
2 |  nb_bits ← 1
3 |  tant que valeur ⩽ 2^nb_bits faire
4 |  |  nb_bits ← nb_bits + 1
5 |  fin tq
6 |  renvoyer nb_bits
7 fin
```

Remarque

Récupérez le fichier **exercices.py** et copiez le dans l'espace **Mes Documents** de votre ordinateur. Exécutez le fichier pour vous assurer de l'absence d'erreur.

Exercice 3 (Usages de la boucle for).

1. On considère le code Python suivant :

```
1 nombre = 0
2 for i in range(0,5,1):
3     nombre = nombre + i
```

a. Quelle sera la valeur de la variable *nombre* lorsque le programme sera totalement exécuté ?

b. Ce code correspond-t-il à une fonction ?

c. Si vous deviez transformer ce code en fonction, existerait-il une entrée ? quelle serait la variable renvoyée ?

d. Complétez la fonction *mystere()* reprenant ce code.
Exécutez en console l'instruction `test_mystere()` pour vous assurer de son bon fonctionnement.

e. Que doît-on saisir dans la console Python pour exécuter cette fonction ?

2. Complétez la fonction Python *somme()* prenant un entier n positif en paramètre, et renvoyant la somme des n premiers entiers. Pour cela :

a. Indiquez deux préconditions sur le type et la valeur de l'argument d'entrée.

b. Complétez le code de la fonction permettant de renvoyer le résultat.

Exécutez en console l'instruction `run_docstring_examples(somme, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_somme()` pour exécuter des tests supplémentaires.

3. Complétez la fonction Python *somme2()* prenant un entier n positif en paramètre, et renvoyant la somme des n premiers entiers pairs.
Exécutez en console l'instruction `run_docstring_examples(somme2, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_somme2()` pour vous assurer de son bon fonctionnement.

4. Complétez la fonction Python *multiple3()*, prenant un entier positif n en paramètre, et affichant les entiers inférieurs ou égal à n qui sont multiples de 3.
Exécutez en console l'instruction `run_docstring_examples(multiple3, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_multiple3()` pour vous assurer de son bon fonctionnement.

5. Complétez la fonction Python *somme_inverse()*, prenant un entier n en paramètre, et renvoyant la valeur de $\sum_{i=1}^{n} \frac{1}{i}$.
Exécutez en console l'instruction `run_docstring_examples(somme_inverse, globals(), verbose = Tru` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_somme_inverse()` pour vous assurer de son bon fonctionnement.

Exercice 4 (Usages de la boucle `while`).

1. Complétez la fonction *boom()*, prenant un entier n en argument, et affichant un compte à rebours, puis une fois arrivé à 0, affiche "BOOOMMMM".
Par exemple, `boom(5)` affichera :

```
5
4
3
2
1
BOOOMMMM
```

Exécutez en console l'instruction `test_boom()` pour vous assurer de son bon fonctionnement.

2. La fonction *increment()* ci-dessous, prend deux entiers a et b tels que $a < b$ en argument, et incrémente la valeur de a tant qu'elle reste inférieure à celle de b.

```
1 def increment(a, b):
2     while a < b:
3         a = a + 1
```

Copiez la fonction *increment()* dans `http://pythontutor.com/visualize.html`, puis indiquez dessous l'appel `increment(1, 5)`.

a. Combien de fois la ligne 2 est-elle exécutée ?

b. Combien de fois la ligne 3 est-elle exécutée ?

3. En vous aidant de la structure précédente, complétez la fonction *increment2()*, prenant deux entiers a et b tels que $a < b$ en argument, incrémentant la valeur de a tant qu'elle reste inférieure à celle de b, et **renvoyant** le nombre de fois où la boucle a été exécutée.
Exécutez en console l'instruction `run_docstring_examples(increment2, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.

4. Complétez la fonction *impaire()*, prenant deux entiers a et b tels que $a < b$ en argument, et **affichant** les différentes valeurs de b (affichage dans l'ordre décroissant) si elles sont impaires, tant que b est supérieur à a.
Par exemple, `impaire(2, 4)` affiche `3` alors que `impaire(2, 5)` affiche `5` puis `3`.
Exécutez en console l'instruction `run_docstring_examples(impaire, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_impaire()` pour exécuter des tests supplémentaires.

Aide à la résolution : il est possible de s'aider de la structure précédente, mais cette fois, c'est b qui va décroître.

5. Complétez la fonction *impaire2()*, prenant deux entiers a et b tels que $a < b$ en argument, et **renvoyant** le nombre de valeurs impaires de b tant que b est supérieur àa.
Par exemple, `impaire(2, 4)` renvoie `1` alors que `impaire(2, 5)` renvoie `2`.
Exécutez en console l'instruction `run_docstring_examples(increment2, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_impaire2()` pour exécuter des tests supplémentaires.

Exercice 5 (Division euclidienne utilisant une boucle `while`).
La division euclidienne de deux nombres peut être déterminée uniquement avec des soustractions et des additions.

Par exemple, la division euclidienne de 9 par 2 donne le résultat suivant :

$$\begin{array}{r|l} 9 & 2 \\ \hline 1 & 4 \end{array}$$

Pour un « premier tour », on soustrait 2 à 9. On obtient 7, qui est supérieur à 2
Pour un « deuxième tour », on soustrait 2 à 7. On obtient 5, qui est supérieur à 2
Pour un « troisième tour », on soustrait 2 à 5. On obtient 3, qui est supérieur à 2
Pour un « quatrième tour », on soustrait 2 à 3. On obtient 1, qui est inférieur à 2. On s'arrête là !

Il a fallut quatre tours, donc le quotient est 4. Le reste de la division est 1.

1. Quelle opération effectue-t-on à chaque tour ?
2. Quelle est la condition qui arrête cette opération ?
3. Comment peut-on compter le nombre de « tours » jusqu'à l'arrêt ?
4. Quelles doivent être les préconditions sur le diviseur ?
5. Écrivez les assertions correspondantes.
6. Quelles doivent être les préconditions sur le dividende ?
7. Écrivez les assertions correspondantes.
8. Quelles doivent être les postconditions sur le quotient ?
9. Écrivez les assertions correspondantes.
10. Quelles doivent être les postconditions sur le reste ?
11. Écrivez les assertions correspondantes.
12. Soient deux entiers x et y. Complétez la fonction Python *reste()*, prenant x et y en argument, et renvoyant le reste de x divisé par y en n'utilisant que des soustractions. Indiquez les précoditions et les postconditions.
    Exécutez en console l'instruction `run_docstring_examples(reste, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
    Exécutez en console l'instruction `test_reste()` pour exécuter des tests supplémentaires.

13. Soient deux entiers x et y. Complétez la fonction Python *quotient()*, prenant x et y en argument, et renvoyant le quotient de x divisé par y en n'utilisant que des soustractions et additions. Indiquez les précoditions et les postconditions.
    Exécutez en console l'instruction `run_docstring_examples(quotient, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
    Exécutez en console l'instruction `test_quotient()` pour exécuter des tests supplémentaires.

Exercice 6 (Conjecture de Syracuse).
Étant donné un entier n strictement positif, on applique les règles suivantes : si n est pair on le divise par 2 ; si n est impair on le multiplie par 3 et on ajoute 1. Puis on applique ces règles sur le résultat ainsi obtenu, et ainsi de suite. La conjecture de Syracuse, qui n'a pas encore été prouvée à ce jour, dit que, quelle que soit la valeur initiale de n, on finit toujours par atteindre la valeur 1.

On appelle la durée de vol de cette suite, le nombre de fois où l'on doit appliquer les règles pour atteindre la valeur 1. Par exemple pour 5, la durée de vol est 5 car les termes de la suite sont 5, 16, 8, 4, 2 puis 1.

Complétez la fonction Python *syracuse()*, prenant un entier n en paramètre, et renvoyant la durée de vol de cette suite.
Exécutez en console l'instruction `run_docstring_examples(syracuse, globals(), verbose = True)` pour vous assurer de son bon fonctionnement.
Exécutez en console l'instruction `test_syracuse()` pour exécuter des tests supplémentaires.