Tests et boucles en Python

Boucles bornées

Il y a deux types de boucles : celles dont on connait à l'avance le nombre d'itérations, qu'on appelle **boucles bornées** et les autres, les **boucles non bornées**.
En Python, pour faire une boucle bornée, on utilise la syntaxe suivante :

```
for VARIABLE in ITERABLE:
    instruction_1
    instruction_2
    ...
    intruction_n
```

Un **itérable** est un ensemble de valeurs que l'on peut parcourir, valeur par valeur. Nous utiliserons principalement 2 types d'itérables : les listes et les chaînes de caractères.
La plupart du temps, nous n'aurons besoin que de répéter n fois une boucle. Pour cela, nous utiliserons :

```
>>> for i in range(4):
        print("bonjour")

bonjour
bonjour
bonjour
bonjour
```

La **variable de boucle** prend des valeurs qui dépendent de l'itérable. Avec la commande `range(n)` ce sont les éléments de la liste de nombres entiers allant de 0 à $n-1$.

```
>>> for compteur in range(5):
        print("etape", compteur)

etape 0
etape 1
etape 2
etape 3
etape 4
```

La fonction `range` laisse la possibilité de commencer à partir d'une autre valeur que 0 en mettant un deuxième paramètre. Le premier correspond à la valeur de départ et le deuxième à la valeur d'arrêt. La commande `range(a, b)` prendra successivement toutes les valeurs entières dans l'intervalle $[a;b[$. On fera donc $b-a$ tours de boucles.

```
>>> for i in range(2, 5):
        print(i)

2
3
4
```

La commande `range(n)` fait donc la même chose que `range(0, n)`.

En rajoutant un troisième paramètre, on peut modifier l'écart entre chaque valeur. Au lieu d'aller de 1 en 1, on peut, par exemple, aller de 2 en 2.

```
>>> for i in range(0, 10, 2):
        print(i)

0
2
4
6
8
```

```
>>> for i in range(1, 10, 2):
        print(i)

1
3
5
7
9
```

Cela permet également de faire un compte à rebours :

```
>>> for i in range(10, 0, -1):
        print(i)

10
9
8
7
6
5
4
3
2
1
```

```
>>> for i in range(5, -1, -1):
        print(i)

5
4
3
2
1
0
```

Il faut bien mettre -1 pour le 2^e paramètre pour aller jusqu'à 0.

Exercice 1 :

Déterminer les valeurs à mettre à la place de `A`, `B` et `C` pour obtenir :

```
>>> for i in range(A, B, C):
        print(2*i+1)

41
35
29
23
17
11
```

Exercice 2 :

Un placement banquaire rapporte 3% d'intérêts par an. Cela veut dire que le montant placé est multiplié par 1,03 chaque année. Compléter le code de la fonction `placement(montant, nb_annees)` qui renvoie le montant sur le compte, qui initialement était de `montant` et qui a rapporté des intérêts pendant `nb_annees`.

```
def placement(montant, nb_annees):
    for i in range(...):
        ...
    return montant
```

```
>>> placement(100, 10)
134.39163793441222
>>> placement(20, 100)
384.3726396171258
```

Si . . . Alors . . . Sinon

En plus des calculs classiques, Python peut effectuer des comparaisons.

```
>>> 5 < 7
True
>>> 5 > 7
False
>>> 5 > 5
False
```

Les mots **True** et **False** sont des valeurs booléennes qui correspondent à "vrai" et "faux". Ces valeurs servent pour les tests et ainsi permettre d'éxecuter des instructions différentes selon si le test est réussi ou pas. En Python, la syntaxe est la suivante :

```
if test:
    code a executer
    si le test est reussi
else:
    code a executer
    si le test echoue
```

Comme toujours, en Python, l'indentation est importante pour délimiter les blocs. Il faut que le **else** soit aligné avec le **if**.

```
>>> a = 3
>>> b = 5
```

```
>>> if a < b:
        print("oui")
    else:
        print("non")

oui
```

```
>>> if a > b:
        print("oui")
    else:
        print("non")

non
```

La partie **else** n'est pas obligatoire.

```
>>> if a < b:
        print("oui")

oui
>>> if a > b:
        print("oui")
    print("fini")

fini
```

La fonction `maximum(a, b)` permet de déterminer le maximum des deux nombres.

```
def maximum(a, b):
    if a > b:
        return a
    else:
        return b
```

```
>>> maximum(5, 2)
5
>>> maximum(2, 5)
5
>>> maximum(5, 5)
5
```

Exercice 3 :

Rajoutez une fonction `minimum(a, b)` qui renvoie la valeur du plus petit des deux nombres donné en argument.

```
>>> minimum(5, 2)
2
>>> minimum(2, 5)
2
>>> minimum(5, 5)
5
```

Remarque :

En Python, il existe des fonctions **max**`(a, b)` et **min**`(a, b)` qui permettent de déterminer le maximum et le minimum de deux nombres.

Les opérateurs de comparaison sont les suivants :

En math	En python
$a = b$	`a == b`
$a \neq b$	`a != b`
$a > b$	`a > b`
$a \geq b$	`a >= b`
$a < b$	`a < b`
$a \leq b$	`a <= b`

Il faut bien faire attention à distinguer l'affectation `a = b` et le test d'égalité `a == b`.

? Exercice 4 :

Compléter la fonction `pythagore(a, b, c)` qui revoie un booléen indiquant si les longueurs a, b et c, c étant la plus grande, permettent de former un triangle rectangle dont les côtés mesurent a, b et c. Pour rappel, pour déterminer si un triangle est rectangle, il faut vérifier que $a^2 + b^2 = c^2$.

```
def pythagore(a, b, c):
    if ...:
        return ...
    else:
        return ...
```

```
>>> pythagore(3, 4, 5)
True
>>> pythagore(3, 4, 6)
False
>>> pythagore(5, 4, 3)
False
```

Tests imbriqués et étude de cas

Parfois, pour déterminer dans quel cas on se trouve, il faut effectuer plusieurs tests.

```
def mention(note):
    if note >= 12:
        print("Mention AB")
    if note >= 14:
        print("Mention B")
    if note >= 16:
        print("Mention TB")
    else:
        print("Pas de mention")
```

Cette fonction ne donne pas le résultat prévu.

```
>>> mention(11)
Pas de mention
>>> mention(15)
Mention B
Mention AB
Pas de mention
>>> mention(17)
Mention TB
Mention B
Mention AB
```

Normalement, il ne devrait y avoir qu'un message d'affiché. C'est parce que les tests sont tous indépendants. Même si le premier test est réussi, les autres tests sont également effectués, alors qu'ils ne devraient l'être que si les précédents ont échoués. Il faut donc imbriquer les tests.

```
def mention(note):
    if note >= 16:
        print("Mention TB")
    else:
        if note >= 14:
            print("Mention B")
        else:
            if note >= 12:
                print("Mention AB")
            else:
                print("Pas de mention")
```

Cette version fonctionne correctement.

```
>>> mention(11)
Pas de mention
>>> mention(13)
Mention AB
>>> mention(15)
Mention B
>>> mention(17)
Mention TB
```

À "cause" du système d'indentation, la lecture de cette fonction n'est pas si simple. C'est pourquoi il existe une syntaxe permettant de simplifier la rédaction des études de cas.

```
if test1:
    bloc1
else:
    if test2:
        bloc2
    else:
        bloc3
```

$\Leftrightarrow$

```
if test1:
    bloc1
elif test2:
    bloc2
else:
    bloc3
```

Il est possible de mettre autant de **elif** que nécessaire et le **else** final n'est pas nécessaire. On obtient alors :

```
def mention(note):
    if note >= 16:
        print("Mention TB")
    elif note >= 14:
        print("Mention B")
    elif note >= 12:
        print("Mention AB")
    else:
        print("Pas de mention")
```

C'est le code du premier des tests réussi qui sera exécuté, et ce sera le seul, même si d'autre tests suivants auraient été réussis. Cela ne suffit pas et toujours à éviter les tests imbriqués.

```
def maximum3(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if b > c:
            return b
        else:
            return c
```

Cette fonction permet de trouver le maximum parmi 3 nombres. On commence par comparer a et b et on compare le plus grand des deux avec c pour trouver le maximum.

Cela revient à faire `maximum(maximum(a, b), c)`.
Pour éviter les tests imbriqués, on peut utiliser l'opération `TEST1 and TEST2` est vraie si et seulement si `TEST1` **et** `TEST2` sont vrais.

```
def max3(a, b, c):
    if a > b and a > c:
        return a
    elif ... and ...:
        return ...
    else:
        return ...
```

Exercice 5 :

Compléter la fonction ci-contre qui fait la même chose que `maximum3`.

Remarque :
Les fonctions **min** et **max** peuvent prendre plus de 2 arguments et donc peuvent donner le minimum ou le maximum de 3, 4 valeurs ou plus encore.

Exercice 6 :

Écrire une fonction `pythagore2(a, b, c)` pour que les valeurs a, b et c puissent être données dans n'importe quel ordre.

```
>>> pythagore2(3, 4, 5)
True
>>> pythagore2(5, 3, 4)
True
>>> pythagore2(4, 5, 3)
True
>>> pythagore2(6, 3, 4)
False
```

Compléments sur les chaînes de caractères

En Python, les textes sont encadrés par des guillemets ou des apostrophes : `"Un texte"` ou `'Un autre texte'`.
Il est possible de **concaténer** deux textes en les "additionnant" :

```
>>> "bon"+"jour"
'bonjour'
```

On peut aussi multiplier un texte par un entier :

```
>>> "bla"*3
'blablabla'
```

Si on multiplie un texte par 0, on obtient le texte vide :

```
>>> "bla"*0
''
```

Il est possible d'accéder à n'importe quel symbole d'un texte à l'aide de sa position. Les symboles sont numérotés de gauche à droite, en commençant à 0. On appelle la position d'un symbole son **index**.

```
>>> "avion"[0]
'a'
>>> "avion"[1]
'v'
>>> "avion"[2]
'i'
>>> "avion"[3]
'o'
>>> "avion"[4]
'n'
```

On peut aussi y accéder en comptant de droite à gauche, en commençant à -1, puis -2...

```
>>> "avion"[-1]
'n'
>>> "avion"[-2]
'o'
>>> "avion"[-3]
'i'
>>> "avion"[-4]
'v'
>>> "avion"[-5]
'a'
```

Pour résumer :

Symboles	a	v	i	o	n
De gauche à droite	0	1	2	3	4
De droite à gauche	-5	-4	-3	-2	-1

Si on donne un index en dehors du mot, on obtient un message d'erreur.

```
>>> "avion"[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Les chaînes de caractères sont également des itérables. Cela permet de les parcourir lettre par lettre :

```
>>> for lettre in "rap":
        print(lettre)

r
a
p
```

Application aux chaînes de caractères

On peut combiner boucles et tests pour (re)programmer les fonctions de base pour les chaînes de caractères.
Rajouter la fonction suivante :

```
def longueur(mot):
    n = 0
    for a in mot:
        n += 1
    return n
```

```
>>> longueur("un")
2
>>> longueur("deux")
4
>>> longueur("trois")
5
```

Cette fonction existe déjà en Python :

```
>>> len("un")
2
>>> len("deux")
4
>>> len("trois")
5
```

On peut aussi définir une fonction qui teste si une lettre est dans un texte :

```
def est_dans(l, mot):
    for a in mot:
        if l == a:
            return True
    return False
```

```
>>> est_dans("a", "abc")
True
>>> est_dans("c", "abc")
True
>>> est_dans("d", "abc")
False
```

On peut remarquer qu'il y a un **return** dans une boucle. Lors de l'appel d'une fonction, son exécution s'arrête lorsque toutes les instructions ont été exécutées ou lorsqu'un **return** est rencontré, même en plein milieu d'une boucle.
Cette fonction existe déjà dans Python :

```
>>> "a" in "abc"
True
>>> "c" in "abc"
True
>>> "d" in "abc"
False
```

Cette fonction permet même de tester si un mot est dans un autres :

```
>>> "gre" in "tigre"
True
>>> "tige" in "tigre"
False
```

Exercice 7 :

Créer une fonction `compter(l, mot)` qui compte combien de fois la lettre `l` apparaît dans `mot`.

```
>>> compter("j", "bonjour")
1
>>> compter("o", "bonjour")
2
>>> compter("z", "bonjour")
0
```

Exercice 8 :

Créer une fonction `index(l, mot)` qui renvoie la position de la lettre `l` dans `mot` si elle y est et -1 sinon. Attention, la position de la première lettre est la position 0.

```
>>> index("a", "abc")
0
>>> index("c", "abc")
2
>>> index("d", "abc")
-1
```

Ces deux fonctions existent également en Python :

```
>>> "bonjour".count("j")
1
>>> "bonjour".count("o")
2
>>> "bonjour".count("z")
0
>>> "abc".index("a")
0
>>> "abc".index("c")
2
>>> "abc".index("d")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: substring not found
```