IngeSUP - Cours 03 - Les structures de répétition

Sommaire

Objectifs

- Maîtrîser l'implémentation des boucles `for` ;
- Maîtriser l'implémentation des boucles `while` .

Vidéos

Le sujet de ce notebook de cours est abordé dans le tutoriel vidéo suivant :

- Module 6 : Répétitions - Boucles en Python (https://courses.ionisx.com/courses/ref/m123/x/courseware/54c7a679a9354a2996ece5f4f11b02b9/4ca7a398

Introduction

Une boucle permet d'exécuter une portion de code plusieurs fois de suite. Python fournit deux manières d'exécuter les boucles grâce aux instructions `for` et `while` .

Bien que ces deux méthodes offrent des fonctionnalités de base similaires, leur syntaxe et leur manière de vérifier les conditions diffèrent.

Les boucles `for ... range`

Une boucle `for...range` est un bloc qui répète une ou plusieurs opérations un nombre de fois spécifié :

```
for variable in range(start, stop, step):
    instructions
```

En termes simples, `range()` permet à l'utilisateur de générer une série de nombres dans une plage donnée. En fonction du nombre d'arguments que l'utilisateur transmet à la fonction, l'utilisateur peut décider où cette série de nombres commencera, où cette cette serie de nombres se terminera ainsi que l'importance de la différence entre un nombre et le suivant.

`range()` peut prendre trois arguments :

- start : entier à partir duquel la séquence d'entiers doit être renvoyée.
- stop : entier avant lequel la séquence d'entiers doit être renvoyée. La plage d'entiers se termine à stop – 1.
- step : valeur entière qui détermine de combien on augmente pour passer au nombre suivant (incrément).

Entrée [ ]:

```
for n in range(0, 4):
    print("----")
    print(n)
```

La boucle ci-dessus répète les instructions 4 fois; n prenant les valeurs 0, 1, 2 et 3.

L'instruction :

```
for n in range(0, 4):
```

indique que nous souhaitons boucler sur des entiers qui vont de 0 (inclus) à 4 (exclu). On s'arrête donc à 3.

> **Note**
>
> Une fois de plus on constate que **for** (tout comme le if et le while) est une structure de contrôle qui utilise les deux points (:) et l'indentation pour caractériser l'appartenance d'un bloc d'instructions.

La valeur de la variable `n` (interne au for) est augmentée à chaque itération (tour de boucle). Les instructions que nous souhaitons éxécuter dans la boucle sont **indentées**:

```
for n in range(0, 4):
    print("----")
    print(n)
```

La boucle commence de zéro et n'inclus pas 4. Si nous le souhaitons, nous pouvons changer la valeur de départ:

Entrée [ ]:

```
for i in range(-2, 3):
    print(i)
```

> ⚠ **Attention**
>
> Lorsque l'on fait *for i in range(0, n)*, **les valeurs de i vont évoluer de 0 à n-1** (n est exclu).

Dans l'exemple précédent la boucle commence à -2 **mais n'inclus pas 3**. Cependant, si nous le souhaitons, nous pouvons utiliser un pas supérieur à 1 pour passer d'une valeur à l'autre...

> **Information**
>
> Lorsque vous appellez `range()` avec trois arguments, vous pouvez choisir non seulement où la série de nombres commencera et s'arrêtera, mais aussi **quelle sera la différence entre un nombre et le suivant**. Ce troisième paramètre s'appelle le pas. Si vous ne fournissez pas de pas, alors `range()` se comportera automatiquement comme si le pas était 1.

Entrée [ ]:

```
# On fait évoluer les valeurs de 3 en 3 selon l'intervalle [0, 10 [

for n in range(0, 10, 3):
    print(n)
```

Raccourci

> **Information**
>
> **for i in range(0,n)** peut être simplifié en **for i in range(n)**. Lorsque vous utilisez `range()` avec un seul argument, vous obtenez une série de nombres qui commence **automatiquement à 0** et qui inclut tous les nombres entiers jusqu'au nombre que vous avez défini comme arrêt **EXCLU**

Autres exemples

Essayez de deviner les affichages suivants :

Entrée [ ]:

```
for i in range(3):
    print(i)
```

Entrée [ ]:

```
for j in range(21, 25):
    print(j)
```

Entrée [ ]:

```
for num in range(7, 20, 3):
    print(num)
```

Afficher les données sur la même ligne

On utilise les paramètre `end=` pour préciser **ce qui se passe à la fin du print**. Par défaut, à la fin il saute la ligne.

Entrée [ ]:

```
#... Mais là on finit chaque print par un espace

for i in range(3):
    print(i, end=" ")
```

Entrée [ ]:

```
#... Mais là on finit chaque print par virgule et espace

for k in range(3):
    print(k, end=", ")
```

Boucler par valeurs descendantes

En prenant un pas négatif on peut boucler selon des valeurs décroissantes:

Entrée [ ]:

```
for l in range(0, -10, -2):
    print(l)
```

Entrée [ ]:

```
for j in range(10, -1, -1):
    print(j, end="  ")
```

Exemple : Tables de multiplication

La création d'une table de multiplication paraît plus simple avec une boucle `for` :

Entrée [ ]:

```
# Par exemple avec la table de 9

for compteur in range(1,11):
    print(compteur, '* 9 =', compteur*9)
print("Et voilà !")
```

Les boucles `for ... in`

la boucle `for...in` est utilisée pour les parcours séquentiels. Par exemple **pour parcourir une chaîne de caractères**.

```
for element in sequence:
    instructions
```

Entrée [ ]:

```
msg = 'Bonjour'
for lettre in msg:
    print(lettre)   # On parcourt la chaine lettre par lettre
```

Comment afficher les lettres côte à côte ? C'est simple on a encore recours à **end=**

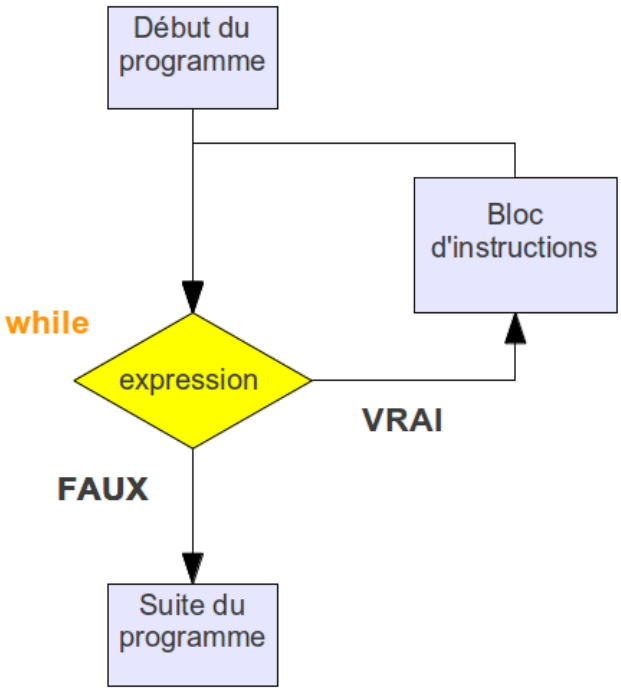
Entrée [ ]:

```
v = "Salut toi!"
for lettre in v:
    print(lettre, end="")    # Chaque print se termine par "le vide", les lettres sont donc
```

> **Note**
>
> Ce qu'on appelle **sequence** ça peux être une liste, un tuple, un dictionnaire, une chaine de caractères ou un fichier. Nous verrons tous ces éléments dans la suite du cours...

Les boucles `while`

Nous venons de voir que les boucles `for` permettent d'éxécuter des instructions **un nombre de fois spécifié**.
La boucle `while` éxécute une ou plusieurs instructions **tant qu'une condition est vraie**.




```
while expression:
        bloc d'instructions
```

> ⚠ **Attention**
>
> N'oubliez pas de mettre les **:** juste après le condition du *while*
>
> N'oubliez pas d'utiliser **l'indentation** pour écrire le bloc d'instruction qui appartient au *while* !

Exemple :

Entrée [ ]:

```
print("Début de l'instruction while")
x = -2
while x < 5:
    print(x)
    x += 1  # Augmenter la valeur de x d'un cran
print("Fin de l'instruction while")
```

affiche la valeur de `x` **tant que** x est inférieur à 5.

> **Note**
>
> En anglais " while " signifie "Tant que". Pour créer une boucle , il faut donc utiliser ce mot clé suivi d'une condition qui dit quand la boucle s'arrête.
>
> Si l'expression est vraie (***True***) le bloc d'instructions est exécuté, puis l'expression est à nouveau évaluée. Le cycle continue jusqu'à ce que l'expression soit fausse (***False***) : on passe alors à la suite du programme.

Un autre exemple sera plus parlant:

On désire écrire 100 fois cette phrase:

" Je ne dois pas poser une question sans lever la main "

Entrée [ ]:

```
i = 0
while i < 100:
    print("Je ne dois pas poser une question sans lever la main")
    i += 1   # Ne pas oublier d'augmenter la valeur de i. C'est l'incrémentation
```

Autre exemple un script qui compte de 1 à 4 :

Entrée [ ]:

```
# initialisation de la variable de comptage
compteur = 1
while compteur < 5:
    # ce bloc est exécuté tant que la condition (compteur < 5) est vraie
    print(compteur, compteur < 5)
    compteur += 1    # Incrémentation du compteur, compteur = compteur + 1
print(compteur < 5)
print("Fin de la boucle")
```

> **Attention**
>
> **Attention aux boucles infinies !**
>
> Dans le code suivant:
>
> ```
> i = 11
> while i != 10:
>     print("Je ne bavarde pas en classe")
>     i += 1    # La valeur n'atteindra jamais 10...
> ```

Entrée [ ]:

```
# Vous pouvez tester la boucle infinie ici !
# Si vous êtes satisfait(e) appuyer sur le carré noir
# pour l'arrêter
i = 11
while i != 10:
    print("Je ne bavarde pas en classe")
    i += 1    # La valeur n'atteindra jamais 10...
```

Autre exemple

Le code ci-dessous remplace la valeur de `x` par son carré tant que le carré de x est supérieur à 0.001 :

Entrée [ ]:

```
x = 0.9
while x > 0.001:
    # Calcul du carré de x
    x = x*x
    print(x)
```

Mais il serait réducteur de limiter le `while` aux valeurs numériques.

C'est une boucle qui existe avant tout pour répéter des instructions **tant qu'une condition est vérifiée**. Voici un code plus représentatif de son fonctionnement :

Entrée [ ]:

```
reponse = input('Voulez-vous commencer ?')
while reponse != "oui" and reponse != "non":
    print("Répondez à la question par oui par non !")
    # On repose la question
    reponse = input('Voulez-vous commencer ?')
```

Boucles imbriquées

Python permet d'utiliser une boucle dans une autre boucle (boucles imbriquées).

```
for iterateur1 in sequence1:
    for iterateur2 in sequence2:
        instructions
```

```
while condition1:
    while condition2:
        instructions
```

Entrée [ ]:

```
for i in range(2):
    for j in range(3):
        print(i, ' - ', j)
```

break, continue et pass

break

Il est parfois utile de sortir d'une boucle `for` ou une boucle `while`. Par exemple, dans une boucle `for`, nous pouvons vérifier si une condition est vérifiée et si oui, sortir prématurément de la boucle. Par exemple :

Entrée [ ]:

```
for x in range(10):
    print(x)
    if x == 5:
        print("Il est temps de sortir de la boucle")
        break
```

continue

Parfois, nous souhaitons aller prématurément à la prochaine itération d'une boucle, sautant ainsi l'execution des instructions de l'itération en cours.

Nous pouvons utiliser pour cela `continue`. Voici un exemple d'une boucle de 20 itérations (de 0 à 19) qui vérifie si l'incrément est divisible par 4.

Entrée [ ]:

```
for j in range(20):
    if j % 4 == 0:  # Vérifie si j est divisible par 4
        # passe au tour suivant (itération suivante)
        continue
    print("Cette valeur n'est pas divisible par 4:", j)
```

pass

Parfois, il est utilse d'avoir une instruction qui permet de ne rien faire. On permet à une valeur de la boucle de "passer son tour". Par exemple :

Entrée [ ]:

```
for x in range(-10,10):
    if x == 0:
        pass
    else:
        print(1/x)
```

Cela peut aider à rendre le programme plus lisible. Il existe des cas spécifiques où aucune instruction n'est à éxécuter.

Utiliser `pass` indique clairement à celui qui lit le code que l'intention du programmeur ou de la programmeuse était de ne rien faire.

Exercices de TD

Vous pouvez maintenant vous exercer à partir du notebook TD 03 - Les structures de répétition (../TD/TD%2003%20-%20Les%20structures%20de%20r%C3%A9p%C3%A9tition.ipynb).