Python 12 - Boucles non bornées

I – Module random

Ce module comporte des fonctionnalités liées au hasard. Deux façons d'obtenir un nombre entre 10 et 50 inclus :

```
import random
a = random.randint(10, 50)
```

```
from random import randint
a = randint(10, 50)
```

II – Déclaration d'une boucle non bornée

1 - DEFINITIONS

Définition d'une boucle

Une boucle est un bloc d'instructions qui va être réalisé zéro, une ou plusieurs fois d'affilée.

Définition d'une boucle non bornée

Une boucle non bornée est une boucle dont l'exécution dépend de l'évaluation d'une expression qu'on nommera **condition de poursuite.**

Le principe est simple :

1) On évalue la **condition de poursuite**

2) On effectue la boucle si la **condition de poursuite** est évaluée à VRAI puis on revient à l'étape 1.

On parle de **boucle non bornée** puisqu'on ne connaît pas à l'avance le nombre de fois où la boucle va être réalisée : on réalise la boucle TANT QUE la condition de poursuite sera évaluée à VRAI. WHILE en anglais. Il n'y a pas de nombre limite au bout duquel on quitte la boucle. Ca peut tourner à l'infini.

2 - Déclaration d'une boucle non bornée

On utilise le mot-clé **`while`** de Python

```
1 r = 40
2 while r < 100:
3     print(r)
4     r = r + 30
5 print("Fin TANT QUE")
```

r va prendre successivement les valeurs 40 puis 70

(L1) **r** vaut _____ initialement

(L2) **r** vaut ____ donc **`r < 100`** évaluée à _____

(L3-L4) : on affiche _____ et on incrémente **r** à _____

(L2) **r** vaut ____ donc **`r < 100`** évaluée à _____

(L3-L4) : on affiche _____ et on incrémente **r** à _____

(L2) **r** vaut ____ donc **`r < 100`** évaluée à _____

(L5) : On sort de la boucle

3 - POUR ou TANT QUE ? Bornée ou non bornée

Si on connaît à l'avance le nombre de fois où on va devoir agir, on utilise une boucle POUR.
La boucle POUR est une boucle bornée.

Si on ne peut pas savoir à l'avance le nombre de fois où on va devoir agir, on utilise une boucle TANT QUE.
La boucle TANT est une boucle non bornée.

4 - Choix attentif de la valeur initiale

Les variables nécessaires à l'évaluation du TANT QUE doivent toutes être définies avant la ligne du TANT QUE.

Si la condition est basée sur une variable qui est calculée dans le bloc de la boucle, comment faire ? Facile : on initialise cette variable avec une "fausse valeur" initiale pour rentrer dans la boucle au départ.

```
total = 0
while total < 16:
    total = de(6) + de(6) + de(6)
print(total)
```

5 - Un peu de logique : poursuite et arrêt

Lien entre poursuite et arrêt

Parfois, lorsqu'on réfléchit, on tombe plutôt sur une **condition d'arrêt** de la boucle...
Or si **`poursuite == True`**, c'est que **`arret ==`** _____
Or si **`poursuite == False`**, c'est que **`arret ==`** _____
On peut donc donc trouver la condtion de poursuite en utilisant la condition d'arrêt :

poursuite = _____

Conclusion : deux façons de faire

Utilisation de la condition de **poursuite** `argent < 2000`

```
while argent < 2000:
```

Utilisation de la condition d'**arrêt** `argent >= 2000`

```
while not argent >= 2000:
```

6 – Boucle infini

Il s'agit d'une boucle pour laquelle la condition de poursuite ne passe jamais à True. Si on veut en faire une volontairement : **`while True:`**

III – Jeu interactif avec input()

Cette fonction renvoie la saisie clavier de l'utlisateur.
Prototype : **`input(prompt:str) -> str:`**

Cette fonction fait trois choses :

1. La fonction **affiche** le string **`prompt`** qu'elle a reçu (c'est la question qu'on veut poser).
2. La fonction **attend** que l'utilisateur tape quelque chose sur le clavier et finalise par l'appui sur ENTREE.
3. La fonction **renvoie** le texte tapé par l'utilisateur, sous forme d'un **`string`**.

On peut donc l'utiliser comme une sorte de PAUSE lorsqu'on utilise l'interface CONSOLE en insérant juste input().

Voici l'exemple typique :

```
1  print("Début du programme")
2  nom = input("Quel est votre nom ? ")
3
4  print(f"Bonjour {nom}")
```

IV – Equivalence

1 - Remplacer un POUR par un TANT QUE

Nous avons vu que :

- La boucle bornée POUR / FOR est programmable naturellement en Python avec un for.
- La boucle non bornée TANT QUE / WHILE est programmable naturellement avec un while.

Néanmoins, on peut remplacer les for par des while. Ce n'est pas "naturel" mais on peut. Exemples :

```
1  def somme_v1(n:int) -> int:
2      '''Renvoie la somme de 1 + 2 + 3... jusqu'à n.'''
3      reponse = 0
4      for x in range(n+1): # ou range(0, n+1, 1)            04 : _____
5          reponse = reponse + x
6      return reponse
7
8  def somme_v2(n:int) -> int:
9      '''Renvoie la somme de 1 + 2 + 3... jusqu'à n.'''
10     reponse = 0
11     x = 0                                                 11 : _____
12     while x < n+1:
13         reponse = reponse + x
14         x = x + 1                                         14 : _____
15     return reponse
```

Il est clair qu'on a recrée la variable de boucle x à l'aide d'un while et que la fonction **`somme_v2()`** est donc plus longue et plus complexe à comprendre.

2 - Remplacer un TANT QUE par un POUR

On peut donc **parfois** programmer une boucle non bornée en utilisant un **`for`** associé à **`return`** plutôt que d'utiliser directement un **while**. J'ai noté **parfois** car cela n'est pas possible si il n'est pas possible de connaître la valeur maximale du nombre de tours de boucle.

Exemple : deux versions d'une fonction qui renvoie le premier mot de plus de 5 lettres trouvés dans une phrase.

On notera qu'on utilise l'espace comme caractère de séparation dans la phrase pour créer le tableau des mots.

```
1  def premier_mot_v1(phrase:'str NON VIDE') ->'int|None':
2      '''Renvoie le premier mot de plus de 5 lettres de la phrase'''
3      t = phrase.split(" ")
4      i = 0
5      while len(t[i]) < 5:
6          i = i + 1
7      if i < len(t[i]:
8          return t[i]
9
10 def premier_mot_v2(phrase:'str NON VIDE') ->'int|None':
11     '''Renvoie le premier mot de plus de 5 lettres de la phrase'''
12     t = phrase.split(" ")
13     for i in range(len(t)):
14         if len(t[i]) >= 5:
15             return t[i]
```

Possible ici car on sait que dans le pire des cas, on avancera jusqu'à la fin de la chaîne de caractères.