Boucles "tant que"

Tant que

Les boucles **for** sont utiles quand on connait le nombre d'itérations dont on aura besoin. Mais parfois, on ne sait pas à l'avance combien de tours de boucle seront nécessaires. Pour cela, on utilise une boucle "tant que" dont la syntaxe en Python est la suivante :

```
while TEST:
    instruction_1
    instruction_2
    ...
    instruction_k
```

Les instructions seront répétées tant que TEST est vérifié.
Par exemple, c'est ainsi que fonctionne les algorithmes de seuil :

```
def seuil(s):
    n = 1
    while n < s:
        n *= 2
    return n
```

```
>>> seuil(7)
8
>>> seuil(8)
8
>>> seuil(35127)
65536
```

Cette fonction permet de déterminer la plus petite puissance de 2 supérieure ou égale à `s`.
Exercice 1 : Écrire une fonction `seuil2(s)` qui renvoie **l'exposant** de la plus petite puissance de 2 supérieure ou égale à `s`.

```
>>> seuil2(1)
0
>>> seuil2(3)
2
>>> seuil2(7)
3
>>> seuil2(35)
6
```

Remarque :
Contrairement à une boucle **for**, il n'est pas garanti qu'une boucle **while** se termine. Il faut donc bien faire attention à la condition utilisée. Nous verrons plus tard qu'il est possible de démontrer qu'une boucle se termine. Ce n'est pas toujours le cas, mais il est conseillé de bien vérifier que sa boucle peut se terminer. Si une boucle met trop de temps ou ne se termine pas, il faut appuyer sur le bouton "Stop" pour interrompre l'exécution.

L'utilisateur récalcitrant

Parfois, il faut demander une valeur à l'utilisateur en cours d'exécution et il faut que cette valeur vérifie certaines contraintes, comme par exemple être un entier positif. Il est possible de faire confiance à l'utilisateur, mais la confiance n'exclut pas le contrôle.
Dans Python, il y a une commande qui permet de tester une condition et qui provoque une erreur si ce n'est pas le cas.

```
>>> assert(3 > 2)
>>> assert(3 < 2)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
AssertionError
```

La commande **assert** peut être utilisée pour faire des tests d'une fonction. On fait une liste de tests que doit valider une fonction et si tout est correct, les tests sont passés sans renvoyer d'erreur. Elle peut aussi être utilisée pour tester des pré-conditions au début d'une fonction, comme **assert**(x > 0). Si la fonction est appelée avec une valeur ne respectant pas ses conditions d'utilisation, il y a une erreur qui est déclenchée.
Mais dans le cas de l'interaction avec un utilisateur, il vaut mieux redemander une nouvelle valeur.
Par exemple, pour avoir un entier positif, on peut utiliser la fonction suivante :

```
def donner_entier_pos():
    n = -1
    while n < 0:
        n = int(input("Donner un entier positif. "))
    print("merci")
    return n
```

La fonction **int**(n) permet de convertir un texte en un entier. Par contre si le texte ne correspond pas à un entier, cela produit une erreur :

```
>>> int("4")
4
>>> int("-4")
-4
>>> int("2.4")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2.4'
```

Exercice 2 : Écrire une fonction oui_ou_non() qui demande à l'utilisateur de répondre O ou N. Aucune autre réponse n'est acceptée. La première valide renvoyée est renvoyée par la fonction.

```
>>> oui_ou_non()
Oui ou non (O/N) ? a
Oui ou non (O/N) ? o
Oui ou non (O/N) ? n
Oui ou non (O/N) ? N
'N'
```

La suite de Syracuse

On appelle **suite de Syracuse** toute suite de nombres obtenue à l'aide de l'algorithme ci-contre, avec U un nombre entier strictement positif.
Même s'il n'y a aucune preuve dans le cas général, tous les nombres testés produisent une suite qui se termine par 1. L'existence, ou non, d'un nombre produisant une suite infinie reste un problème ouvert.

Tant que $U > 1$ **:**
  Si U est pair **:**
    $U \leftarrow \frac{U}{2}$
  Sinon :
    $U \leftarrow 3U + 1$
  Afficher U

Exercice 3 (sur papier) : Calculer les valeurs successives obtenues en partant de 5 et jusqu'à arriver à 1. Refaire de même en partant de 7 et de 21.

Pour tester si un nombre est pair en Python, il faut utiliser le reste de la division euclidienne par 2. Si le reste est 0, le nombre est pair, sinon il est impair.

```
>>> 1563 % 2 == 0    # nombre impair
False
>>> 4594 % 2 == 0    # nombre pair
True
```

Exercice 4 : Compléter la fonction `prochain_syracuse(u)` qui renvoie la valeur suivant `n` dans la suite de Syracuse.

```
def prochain_syracuse(u):
    if ...:
        return u // 2   # division entière
    else:
        return ...
```

La fonction suivante renvoie la liste des valeurs prises par la suite en partant de `u`.

```
def suite_syracuse(u):
    suite = [u]   # on commence avec u
    while u > 1:
        u = prochain_syracuse(u)
        suite.append(u)  # on rajoute u
    return suite
```

```
>>> suite_syracuse(5)
[5, 16, 8, 4, 2, 1]
>>> suite_syracuse(136)
[136, 68, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Cette fonction vous permettra de vérifier les résultats des exercices suivants.

Exercice 5 : On appelle **altitude maximale** d'une suite de Syracuse le plus grand nombre de la suite. Compléter la fonction `altitude_max(u)` qui renvoie l'altitude maximale en partant de `u`.

```
def altitude_max(u):
    alt_max = ...
    while ...:
        u = ...
        if ...:
            alt_max = u
    return alt_max
```

```
>>> altitude_max(11)
52
>>> suite_syracuse(11)
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
>>> altitude_max(32)
32
>>> suite_syracuse(32)
[32, 16, 8, 4, 2, 1]
```

Exercice 6 : On appelle **durée de vol** la longueur d'une suite de Syracuse. Écrire une fonction `duree_vol(u)` qui renvoie la durée de vol en partant de `u`.

```
>>> duree_vol(7)
17
>>> suite_syracuse(7)
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Altitudes et durées records

Exercice 7 : Compléter la fonction `record_altitude(u_limite)` qui renvoie la valeur de `u` qui donne la suite avec l'altitude maximale, en partant entre 1 et `u_limite`. La fonction renvoie également l'altitude maximale atteinte. En cas d'égalité en partant de deux valeurs différentes de `u`, c'est la plus petite qui est renvoyée.

```
def record_altitude(u_limite):
    a_record = 0          # la plus haute altitude qu'on a vu
    u_record = 0          # le point de départ pour cette altitude
    for u in range(1, u_limite+1):   # pour aller de 1 à u_limite inclus
        a = altitude_max(u)   # on calcule l'alt max en partant de u
        if ...:               # on regarde si on a un nouvau record
            a_record = ...
            u_record = ...
    return u_record, a_record
```

```
>>> record_altitude(10)
(7, 52)     # le record est en partant de 7 et on atteint 52
>>> suite_syracuse(7)
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Exercice 8 : Trouver le nombre inférieur à 10 000 ayant l'altitude maximale.

Exercice 9 : Écrire une fonction `record_duree(u_limite)` qui trouve le nombre ayant la plus grande durée de vol, en partant entre 1 et `u_limite`. La fonction doit renvoyer la valeur de `u` et la durée de vol maximale obtenue.

```
>>> record_duree(10)
(9, 20)   # en partant de 9, on a une durée de vol de 20
>>> suite_syracuse(9)
[9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Exercice 10 : Trouver le nombre inférieur à 10 000 ayant la durée de vol maximale.

Stop ou encore ?

Parfois il est difficile d'écrire une condition pour la boucle. Il est alors possible d'utiliser **`while True:`**, qui va donner une boucle infinie. Pour sortir de la boucle, il faut utiliser la commande **`break`** qui termine immédiatement l'exécution de la boucle et passe à la première instruction après la boucle.

```
>>> i = 0
>>> while True:
        if i == 5:
            break
        i += 1

>>> i
5
```

Dans le cas de boucle imbriquées, cette commande n'arrête que la boucle à l'intérieur des autres.
Il existe aussi une commande **`continue`** qui permet d'interrompre l'exécution d'itération actuelle et repart au début de la boucle pour une nouvelle itération. Cette commande est surtout utilise dans le cas de boucle très longues dont chaque itération prend un certain temps d'exécution.

Pour aller plus loin

Exercice 11 : Écrire une fonction `altitude_donnee(alt)` qui trouve le plus petit nombre dont l'altitude maximale est supérieure ou égale à `alt`.
Trouver le plus petit nombre dont l'altitude maximale dépasse 100 000 000.

Exercice 12 : Écrire une fonction `duree_donnee(duree)` qui trouve le plus petit nombre dont la durée de vol est supérieure ou égale à `duree`.
Trouver le plus petit nombre dont la durée de vol dépasse 300.