

Arithmétique –

Boucle tant que – I

Les activités de cette fiche sont centrées sur l'arithmétique : division euclidienne, nombres premiers... C'est l'occasion d'utiliser intensivement la boucle « tant que ».

Cours 1 (Arithmétique).

On rappelle ce qu'est la division euclidienne. Voici la division de a par b , a est un entier positif, b est un entier strictement positif (avec un exemple de 100 divisé par 7) :

The diagram illustrates the Euclidean division of a by b . It shows a vertical line separating a on the left and b on the right. Below this line, q is written under b and r is written under a . A blue arrow points from the word 'reste' to r , and another blue arrow points from the word 'quotient' to q . To the right, a numerical example is shown: 100 divided by 7, with 14 as the quotient and 2 as the remainder.

a	b
<hr/>	
r	q

reste quotient

100	7
<hr/>	
2	14

On a les deux propriétés fondamentales qui définissent q et r :

$$a = b \times q + r \quad \text{et} \quad 0 \leq r < b$$

Par exemple, pour la division de $a = 100$ par $b = 7$: on a le quotient $q = 14$ et le reste $r = 2$ qui vérifient bien $a = b \times q + r$ car $100 = 7 \times 14 + 2$ et aussi $r < b$ car $2 < 7$.

Avec Python :

- `a // b` renvoie le quotient,
- `a % b` renvoie le reste.

Il est facile de vérifier que :

b est un diviseur de a si et seulement si $r = 0$.

Activité 1 (Quotient, reste, divisibilité).

Objectifs : utiliser le reste pour savoir si un entier divise un autre.

1. Programme une fonction `quotient_reste(a,b)` qui fait les tâches suivantes à partir de deux entiers $a \geq 0$ et $b > 0$:

- Elle affiche le quotient q de la division euclidienne de a par b ,
- elle affiche le reste r de cette division,
- elle affiche `True` si le reste r est bien positif et strictement inférieur à b , et `False` sinon,
- elle affiche `True` si on a bien l'égalité $a = bq + r$, et `False` sinon.

Voici par exemple ce que doit afficher l'appel `quotient_reste(100,7)` :

Division de $a = 100$ par $b = 7$
 Le quotient vaut $q = 14$
 Le reste vaut $r = 2$
 Vérification reste $0 \leq r < b$? True
 Vérification égalité $a = bq + r$? True

Remarque. il faut que tu vérifies sans tricher que l'on a bien $0 \leq r < b$ et $a = bq + r$, mais bien sûr cela doit toujours être vrai !

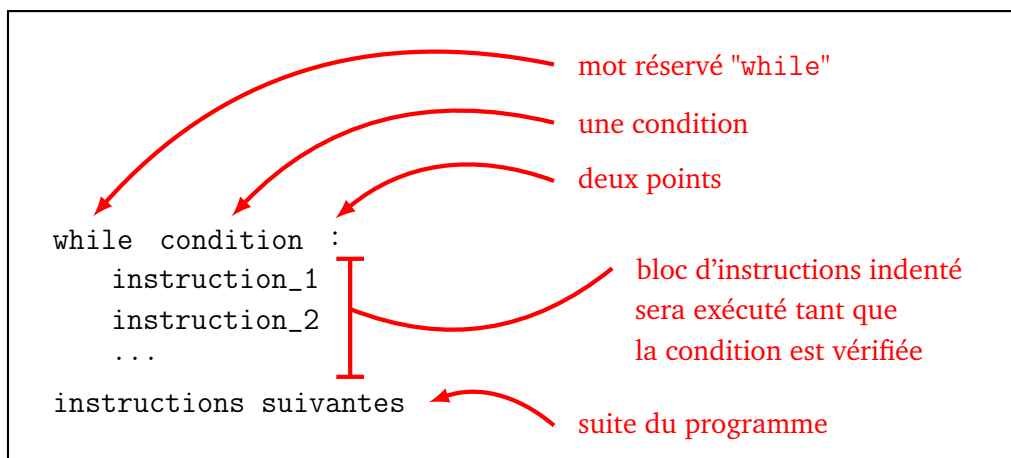
2. Programme une fonction `est_pair(n)` qui teste si l'entier n est pair ou pas. La fonction renvoie True ou False.

Indications

- Première possibilité : calculer $n \% 2$ et discuter selon les cas.
 - Seconde possibilité : calculer $n \% 10$ (qui renvoie le chiffre des unités) et discuter.
 - Les plus malins arriveront à écrire la fonction sur deux lignes seulement (une pour `def`... et l'autre pour `return`...)
3. Programme une fonction `est_divisible(a,b)` qui teste si b divise a . La fonction renvoie True ou False.

Cours 2 (Boucle « tant que »).

La boucle « tant que » exécute des instructions tant qu'une condition est vraie. Dès que la condition devient fausse, elle passe aux instructions suivantes.



Exemple.

Voici un programme qui affiche le compte à rebours 10, 9, 8, ..., 3, 2, 1, 0. Tant que la condition $n \geq 0$ est vraie, on diminue n de 1. La dernière valeur affichée est $n = 0$, car ensuite $n = -1$ et la condition « $n \geq 0$ » devient fausse donc la boucle s'arrête.

On résume ceci sous la forme d'un tableau :

Entrée : $n = 10$

n	« $n \geq 0$ » ?	nouvelle valeur de n
10	oui	9
9	oui	8
...
1	oui	0
0	oui	-1
-1	non	

Affichage : 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

```
n = 10
while n >= 0:
    print(n)
    n = n - 1
```

Exemple.

Ce bout de code cherche la première puissance de 2 plus grande qu'un entier n donné. La boucle fait prendre à p les valeurs 2, 4, 8, 16, ... Elle s'arrête dès que la puissance de 2 est supérieure ou égale à n , donc ici ce programme affiche 128.

```
n = 100
p = 1
while p < n:
    p = 2 * p
print(p)
```

Entrées : $n = 100$, $p = 1$

p	« $p < n$ » ?	nouvelle valeur de p
1	oui	2
2	oui	4
4	oui	8
8	oui	16
16	oui	32
32	oui	64
64	oui	128
128	non	

Affichage : 128

Exemple.

Pour cette dernière boucle on a déjà programmé une fonction `est_pair(n)` qui renvoie `True` si l'entier n est pair et `False` sinon. La boucle fait donc ceci : tant que l'entier n est pair, n devient $n/2$. Cela revient à supprimer tous les facteurs 2 de l'entier n . Comme ici $n = 56 = 2 \times 2 \times 2 \times 7$, ce programme affiche 7.

```
n = 56
while est_pair(n) == True:
    n = n // 2
print(n)
```

Entrée : $n = 56$

n	« n est pair » ?	nouvelle valeur de n
56	oui	28
28	oui	14
14	oui	7
7	non	

Affichage : 7

Pour ce dernier exemple il est beaucoup plus naturel de démarrer la boucle par

```
while est_pair(n):
```

En effet `est_pair(n)` est déjà une valeur « vrai » ou faux ». On se rapproche d'une phrase « tant que n est pair... »

Opération « += ». Pour incrémenter un nombre tu peux utiliser ces deux méthodes :

```
nb = nb + 1    ou    nb += 1
```

La seconde écriture est plus courte mais rend le programme moins lisible.

Activité 2 (Nombres premiers).

Objectifs : tester si un entier est (ou pas) un nombre premier.

1. Plus petit diviseur.

Programme une fonction `plus_petit_diviseur(n)` qui renvoie, le plus petit diviseur $d \geq 2$ de l'entier $n \geq 2$.

Par exemple `plus_petit_diviseur(91)` renvoie 7, car $91 = 7 \times 13$.

Méthode.

- On rappelle que d divise n si et seulement si $n \% d$ vaut 0.
- La mauvaise idée est d'utiliser une boucle « pour d variant de 2 à n ». En effet, si par exemple on sait que 7 est diviseur de 91 cela ne sert à rien de tester si 8, 9, 10... sont aussi des diviseurs car on a déjà trouvé le plus petit.
- La bonne idée est d'utiliser une boucle « tant que » ! Le principe est : « tant que je n'ai pas obtenu mon diviseur, je continue de chercher ». (Et donc, dès que je l'ai trouvé, j'arrête de chercher.)
- En pratique voici les grandes lignes :
 - Commence avec $d = 2$.
 - Tant que d ne divise pas n alors, passe au candidat suivant (d devient $d + 1$).
 - À la fin d est le plus petit diviseur de n (dans le pire des cas $d = n$).

2. Nombres premiers (1).

Modifie légèrement ta fonction `plus_petit_diviseur(n)` pour écrire une première fonction

`est_premier_1(n)` qui renvoie « vrai » (True) si n est un nombre premier et « faux » (False) sinon.

Par exemple `est_premier_1(13)` renvoie True, `est_premier_1(14)` renvoie False.

3. Nombres de Fermat.

Pierre de Fermat (~1605–1665) pensait que tous les entiers $F_n = 2^{(2^n)} + 1$ étaient des nombres premiers. Effectivement $F_0 = 3$, $F_1 = 5$ et $F_2 = 17$ sont des nombres premiers. S'il avait connu Python il aurait sûrement changé d'avis ! Trouve le plus petit entier F_n qui n'est pas premier.

Indication. Avec Python b^c s'écrit `b ** c` et donc $a^{(b^c)}$ s'écrit `a ** (b ** c)`.

On va améliorer notre fonction qui teste si un nombre est premier ou pas, cela nous permettra de tester plus vite plein de nombres ou bien des nombres très grands.

4. Nombres premiers (2).

Améliore ta fonction en une fonction `est_premier_2(n)` qui ne teste pas tous les diviseurs d jusqu'à n , mais seulement jusqu'à \sqrt{n} .

Explications.

- Par exemple pour tester si 101 est un nombre premier, il suffit de voir s'il admet des diviseurs parmi 2, 3, ..., 10. Le gain est appréciable !
- Cette amélioration est due à la proposition suivante : si un entier n'est pas premier alors il admet un diviseurs d qui vérifie $2 \leq d \leq \sqrt{n}$.
- Au lieu de tester si $d \leq \sqrt{n}$, il est plus facile de tester $d^2 \leq n$!

5. Nombres premiers (3).

Améliore ta fonction en une fonction `est_premier_3(n)` à l'aide de l'idée suivante. On teste si $d = 2$ divise n , mais à partir de $d = 3$, il suffit de tester les diviseurs impairs (on teste d , puis $d + 2$...).

- Par exemple pour tester si $n = 419$ est un nombre premier, on teste d'abord si $d = 2$ divise n , puis $d = 3$ et ensuite $d = 5$, $d = 7$...
- Cela permet de faire environ deux fois moins de tests !
- Explications : si un nombre pair d divise n , alors on sait déjà que 2 divise n .

6. Temps de calcul.

Compare les temps de calcul de tes différentes fonctions `est_premier()` en répétant par exemple un million de fois l'appel `est_premier(97)`. Voir le cours ci-dessous pour savoir comment faire.

Cours 3 (Temps de calcul).

Il existe deux façons de faire tourner plus rapidement des programmes : une bonne et une mauvaise. La mauvaise, c'est d'acheter un ordinateur plus puissant. La bonne, c'est de trouver un algorithme plus efficace !

Avec Python, c'est facile de mesurer le temps d'exécution d'une fonction afin de le comparer avec le temps d'exécution d'une autre. Il suffit d'utiliser le module `timeit`.

Voici un exemple : on mesure le temps de calcul de deux fonctions qui ont le même but, tester si un entier n est divisible par 7.

```
# Première fonction (pas très maligne)
def ma_fonction_1(n):
    divis = False
    for k in range(n):
        if k*7 == n:
```

```

        divis = True
    return divis

# Seconde fonction (plus rapide)
def ma_fonction_2(n):
    if n % 7 == 0:
        return True
    else:
        return False

import timeit

print(timeit.timeit("ma_fonction_1(1000)",
    setup="from __main__ import ma_fonction_1",
    number=100000))
print(timeit.timeit("ma_fonction_2(1000)",
    setup="from __main__ import ma_fonction_2",
    number=100000))

```

Résultats.

Le résultat dépend de l'ordinateur, mais permet la comparaison des temps d'exécution des deux fonctions.

- La mesure pour la première fonction renvoie 5 secondes. L'algorithme n'est pas très malin. On teste si $7 \times 1 = n$, puis on teste $7 \times 2 = n$, $7 \times 3 = n$...
- La mesure pour la seconde fonction renvoie 0.01 seconde ! On teste si le reste de n divisé par 7 est 0. La seconde méthode est donc 500 fois plus rapide que la première.

Explications.

- On appelle le module `timeit`.
- La fonction `timeit.timeit()` renvoie le temps d'exécution en seconde. Elle prend comme paramètres :
 - une chaîne pour l'appel de la fonction à tester (ici est-ce que 1000 est divisible par 7),
 - un argument `setup="..."` qui indique où trouver cette fonction,
 - le nombre de fois qu'il faut répéter l'appel à la fonction (ici `number=100000`).
- Il faut que le nombre de répétitions soit assez grand pour éviter les incertitudes.

Activité 3 (Plus de nombres premiers).

Objectifs : programmer davantage de boucles « tant que » et étudier différentes sortes de nombres premiers à l'aide de ta fonction `est_premier()`.

1. Écris une fonction `nombre_premier_apres(n)` qui renvoie le premier nombre premier p supérieur ou égal à n .
Par exemple, le premier nombre premier après $n = 60$ est $p = 61$. Quel est le premier nombre premier après $n = 100\,000$?
2. Deux nombres premiers p et $p + 2$ sont appelés **nombres premiers jumeaux**. Écris une fonction `nombres_jumeaux_apres(n)` qui renvoie le premier couple $p, p + 2$ de nombres premiers jumeaux, avec $p \geq n$.
Par exemple, le premier couple de nombres premiers jumeaux après $n = 60$ est $p = 71$ et $p + 2 = 73$. Quel est le premier couple de nombres premiers jumeaux après $n = 100\,000$?

3. Un entier p est un **nombre premier de Germain** si p et $2p + 1$ sont des nombres premiers. Écris une fonction `nombre_germain_apres(n)` qui renvoie le couple $p, 2p + 1$ où p est le premier nombre premier de Germain $p \geq n$.

Par exemple, le premier nombre premier de Germain après $n = 60$ est $p = 83$ avec $2p + 1 = 167$. Quel est le premier nombre premier de Germain après $n = 100\,000$?