COURS

Algorithmique et programmation Python

2de

Programme de seconde générale et technologique — BO spécial n°1 du 22 janvier 2019

Programmer, en mathématiques, ne sert pas à faire de l'informatique : cela sert à **écrire un raisonnement de façon si précise qu'une machine puisse l'exécuter**. Un algorithme qui ne marche pas n'est presque jamais un problème de langage — c'est un raisonnement incomplet, et le programme se contente de le dire tout haut.

1 Les briques du langage

DÉFINITION

Une **variable** est un nom qui désigne une valeur. L'**affectation** `n = 5` ne signifie pas « n égale 5 » mais « range 5 dans la case appelée n ». La différence apparaît dès qu'on écrit `n = n + 1` : mathématiquement absurde, cette ligne est parfaitement claire en programmation — elle remplace le contenu de n par ce contenu augmenté de 1.

les briques : variable et condition

```
n = 5                  # une variable, sans déclaration
if n > 3:               # deux points, puis un bloc indenté
    print("grand")
else:
    print("petit")
```

Deux points, puis un bloc indenté : l'indentation fait le programme.

En Python, l'**indentation** n'est pas une commodité de lecture : c'est la syntaxe elle-même. Ce qui est décalé sous un `if` s'exécute quand la condition est vraie ; ce qui revient à gauche s'exécute dans tous les cas. Un décalage mal placé change le programme sans provoquer d'erreur, ce qui en fait la faute la plus difficile à trouver.

LE PIÈGE QUI COÛTE LE PLUS DE POINTS

Écrire `if n = 3:` au lieu de `if n == 3:`. — un seul signe égal **affecte** une valeur ; le double signe **compare**. Python refuse la première forme dans un test, et signale une erreur de syntaxe — ce qui, ici, est une chance : dans d'autres langages, le programme s'exécuterait en donnant un résultat faux.

Le réflexe : un `=` range, un `==` demande. Dans un `if`, on demande toujours.

2 Les deux boucles

RÈGLE

La boucle **`for`** répète un nombre de fois **connu à l'avance** : `for i in range(1, 11)` fait prendre à `i` les valeurs 1 à 10 — la borne de droite est exclue, et c'est l'erreur d'un débutant sur deux. La boucle **`while`** répète **tant qu'une condition reste vraie** : le nombre de tours n'est pas connu, et il faut garantir que la condition finira par devenir fausse.

les deux boucles : nombre connu, condition

```
total = 0
for i in range(1, 11):     # i prend 1, 2, … 10 — jamais 11
    total = total + i
print(total)                # affiche 55

n = 100
while n > 1:                # tant que la condition est vraie
    n = n // 2              # division entière
```

Nombre de tours connu : for. Condition d'arrêt : while.

LE PIÈGE QUI COÛTE LE PLUS DE POINTS

Écrire une boucle `while` dont la condition ne peut jamais devenir fausse. — le programme tourne indéfiniment. Rien ne s'affiche, rien ne plante : la machine attend une fin qui ne viendra pas. C'est la seule erreur qui ne produit aucun message.

Le réflexe : avant d'écrire le corps de la boucle, se demander : **quelle ligne, à l'intérieur, rapproche de la sortie ?** Si la réponse n'est pas immédiate, la boucle est infinie.

3 Les fonctions

DÉFINITION

Une **fonction** regroupe un traitement sous un nom, avec des **paramètres** en entrée. L'instruction `return` **renvoie** une valeur à celui qui a appelé la fonction, et interrompt son exécution. Elle ne doit pas être confondue avec `print`, qui **affiche** sans rien renvoyer : une fonction qui affiche mais ne renvoie rien ne peut pas être réutilisée dans un calcul.

une fonction : elle renvoie une valeur

```
def maximum(liste):
    m = liste[0]              # on part du premier
    for x in liste:
        if x > m:
            m = x
    return m                  # renvoie, n'affiche pas

print(maximum([3, 9, 2, 7]))  # affiche 9
```

Une fonction renvoie ; elle n'affiche pas.

4 Quatre algorithmes à connaître

CHERCHER UN MAXIMUM

1. **Initialiser** avec le premier élément de la liste, jamais avec 0 : une liste de nombres négatifs donnerait un maximum faux.
2. **Parcourir** tous les éléments, y compris le premier — le comparer à lui-même ne coûte rien et simplifie l'écriture.
3. **Remplacer** la valeur retenue chaque fois qu'on rencontre plus grand.
   à la fin du parcours, la variable contient nécessairement le plus grand élément rencontré.

EXEMPLE

L'algorithme d'Euclide calcule le PGCD de deux entiers. Le suivre à la main sur 48 et 18.

1. Le principe : on remplace le plus grand par le **reste** de sa division par le plus petit, et on recommence jusqu'à ce que le reste soit nul.
2. Premier tour : reste de 48 par 18.
   $48 - 2 \times 18 = 12$
3. Deuxième tour : reste de 18 par 12.
   $18 - 12 = 6$
4. Troisième tour : reste de 12 par 6.
   $12 - 2 \times 6 = 0$
5. Le reste est nul : le PGCD est le dernier diviseur employé, soit **6**.
   en Python, la boucle s'écrit `while b != 0 :` puis `a, b = b, a % b`, et l'on renvoie `a`.

PGCD(48 ; 18) = 6, en trois tours de boucle

Le **test de primalité** illustre le seul souci d'efficacité du programme de seconde. Tester tous les diviseurs de 2 à $n - 1$ fonctionne, mais devient très lent pour un grand n. Or si n a un diviseur supérieur à $\sqrt{n}$, il en a nécessairement un inférieur : il suffit donc de tester jusqu'à $\sqrt{n}$. Pour $n = 10^6$, on passe d'un million de tests à mille.

test de primalité, arrêté à la racine

```
def est_premier(n):
    if n < 2:
        return False          # 0 et 1 ne sont pas premiers
    d = 2
    while d * d <= n:         # équivaut à d <= racine de n
        if n % d == 0:
            return False      # un diviseur trouvé : on s'arrête
        d = d + 1
    return True
```

Le test `d * d <= n` évite d'appeler une racine carrée : plus rapide, et exact.

Deux détails de ce programme méritent d'être remarqués. Le `return False` **à l'intérieur** de la boucle est ici volontaire : dès qu'un diviseur est trouvé, il n'y a plus rien à chercher, et sortir immédiatement fait gagner tout le reste du parcours. Et la condition s'écrit `d * d <= n` plutôt que `d <= sqrt(n)` : on compare deux entiers au lieu d'un entier et d'un décimal approché, ce qui supprime tout risque d'arrondi.

LE PIÈGE QUI COÛTE LE PLUS DE POINTS

Écrire `return True` à l'intérieur de la boucle, symétriquement au `return False`. — la fonction renverrait `True` dès le premier diviseur qui ne convient pas — c'est-à-dire dès $d = 2$ pour tout nombre impair. Elle déclarerait 9 premier.

Le réflexe : on ne peut conclure « premier » qu'**après** avoir tout testé : le `return True` va donc à l'extérieur de la boucle, aligné avec le `while`.

À VÉRIFIER

- `=` range une valeur, `==` compare : ai-je le bon signe dans mon test ?
- L'**indentation** correspond-elle exactement à ce que je veux répéter ?
- `range(1, 11)` s'arrête à **10** : ai-je vérifié mes bornes ?
- Dans un `while` : quelle ligne rapproche de la sortie ?
- Ma fonction **renvoie**-t-elle avec `return`, ou se contente-t-elle d'afficher ?
- Ai-je testé mon programme sur un cas dont je connais la réponse ?

À RETENIR

- `=` **range** une valeur · `==` **compare**. Dans un `if`, toujours `==`.
- L'**indentation** est la syntaxe : elle décide de ce qui se répète.
- `range(a, b)` s'arrête à $b - 1$ · le nombre de tours vaut $b - a$.
- `for` : nombre de tours connu · `while` : condition d'arrêt.
- Dans un `while`, une ligne doit **rapprocher de la sortie**.
- `return` renvoie et interrompt · `print` affiche et ne renvoie rien.
- Un maximum s'initialise avec le **premier élément**, jamais avec 0.
- Primalité : tester jusqu'à $\sqrt{n}$ suffit.