FICHE DE COURS

Python : les bases de la programmation

Variables, tests, boucles et fonctions

Un **algorithme** est une suite finie d'instructions qui résout un problème. Pour le faire exécuter par une machine, on le traduit dans un **langage de programmation** : ce sera Python. Cette fiche présente les briques qui suffisent à écrire la quasi-totalité des programmes du lycée.

1. Variables et affectation

Une **variable** est un emplacement de la mémoire, désigné par un nom, dans lequel on range une valeur. L'instruction qui y range une valeur s'appelle une **affectation** et s'écrit avec le signe `=`.

Une affectation `variable = expression` s'exécute **de droite à gauche** : Python calcule d'abord la valeur de l'expression, puis la range dans la variable.

Le signe `=` n'est donc *pas* le signe d'égalité des mathématiques. L'instruction `x = x + 1` n'est pas une équation — elle n'aurait aucune solution — mais un ordre : « augmenter x de 1 ».

Exemple : suivre l'exécution pas à pas

```
a = 7
b = 2
a = a + 10
b = a
a = 0
```

On dresse un **tableau de suivi** : une ligne par instruction, une colonne par variable.

Instruction exécutée	a	b
`a = 7`	7	—
`b = 2`	7	2
`a = a + 10`	17	2
`b = a`	17	17
`a = 0`	0	17

À la fin, $a = 0$ et $b = 17$. La ligne `b = a` a *copié* la valeur de a à cet instant : modifier a ensuite ne change plus b.

2. Les nombres en Python

Python distingue deux types de nombres :

- les **entiers** (type `int`), stockés de façon exacte : `7`, `-3` ;
- les **nombres à virgule** (type `float`), stockés de façon *approchée*, avec un point comme séparateur décimal : `3.5`, `-0.25`.

Les cinq opérations :

Opérateur	Rôle	Exemple
`+ - *`	somme, différence, produit	`3 * 4` vaut `12`
`/`	division décimale, résultat *toujours* `float`	`7 / 2` vaut `3.5`
`//`	quotient de la division euclidienne	`7 // 2` vaut `3`
`%`	reste de la division euclidienne	`7 % 2` vaut `1`
`**`	puissance	`2 ** 10` vaut `1024`

Exemple : division euclidienne et parité
Pour 47 divisé par 5, on demande le quotient et le reste :

```
q = 47 // 5      # q vaut 9
r = 47 % 5       # r vaut 2
```

On retrouve l'égalité $47 = 5 \times 9 + 2$ avec $0 \leqslant 2 < 5$.

Le reste sert aussi à tester la divisibilité : un entier n est pair si et seulement si `n % 2 == 0`, et divisible par 3 si et seulement si `n % 3 == 0`.

La machine ne connaît pas $\mathbb{R}$. Les `float` sont des valeurs approchées : on peut vérifier que `0.1 + 0.2 == 0.3` renvoie `False`. On ne teste donc **jamais** l'égalité de deux nombres à virgule ; on compare plutôt leur écart à une petite valeur.

Autre conséquence : `6 / 3` renvoie `2.0` et non `2`. Le résultat vaut bien $2 \in \mathbb{N}$, mais l'opérateur `/` produit toujours un `float`.

3. Communiquer avec l'utilisateur

`print(...)` **affiche** une ou plusieurs valeurs à l'écran. `input(message)` affiche le message, attend une saisie au clavier, et renvoie ce qui a été tapé.

`input` renvoie **toujours** une chaîne de caractères, jamais un nombre. Pour calculer ensuite, il faut convertir : `int(...)` pour un entier, `float(...)` pour un nombre à virgule.

Exemple : périmètre et aire d'un rectangle

```
L = float(input("Longueur : "))
l = float(input("Largeur : "))
p = 2 * (L + l)
a = L * l
print("Perimetre :", p)
print("Aire :", a)
```

Pour $L = 7{,}5$ et $\ell = 4$, le programme affiche $p = 23$ et $a = 30$.

Sans les `float(...)`, la ligne `2 * (L + l)` concaténerait les deux textes saisis et afficherait `"7.547.54"` : un résultat absurde, mais sans message d'erreur.

4. Tester : les instructions conditionnelles

L'instruction `if` exécute un bloc **seulement si** une condition est vraie. On la complète par `elif` (« sinon, si ») et `else` (« sinon »). Les deux points en fin de ligne et l'**indentation** du bloc sont obligatoires : c'est le décalage vers la droite qui délimite le bloc.

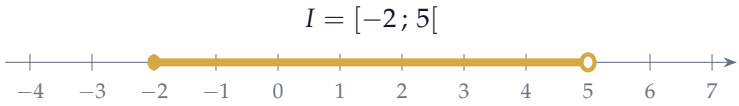
```
if condition :
    instructions
elif autre_condition :
    instructions
else:
    instructions
```

Une **condition** est une expression qui vaut `True` ou `False`. On la construit avec les comparaisons `==`, `!=`, `<`, `<=`, `>`, `>=`, puis on les combine avec les connecteurs logiques :

Connecteur	Sens	Traduction ensembliste
`and`	les deux conditions sont vraies	intersection $\cap$
`or`	au moins une est vraie	réunion $\cup$
`not`	inverse la condition	complémentaire

Attention : `=` affecte, `==` teste. Les confondre est l'erreur la plus fréquente.

Exemple : appartenance à un intervalle
On veut savoir si un nombre x appartient à $I = [-2\,;\,5[$.



Le crochet fermé en -2 donne une inégalité **large**, le crochet ouvert en 5 une inégalité **stricte** :

```
def appartient(x) :
    return -2 <= x < 5
```

On vérifie toujours sur les **bornes**, là où se cachent les erreurs de crochet : `appartient(-2)` renvoie `True`, `appartient(5)` renvoie `False`, et `appartient(4.999)` renvoie `True`.

L'ordre des tests compte. Dans une cascade `if / elif / else`, seul le premier test vrai est exécuté : il faut donc aller du cas le plus *restrictif* vers le plus *général*. Pour reconnaître à quel ensemble appartient un entier, on teste $\mathbb{N}$ *avant* $\mathbb{Z}$: dans l'autre ordre, $\mathbb{N}$ ne serait jamais atteint, puisque $\mathbb{N} \subset \mathbb{Z}$.

5. Répéter : les boucles

Deux boucles, pour deux situations différentes :

- la boucle `for` répète un bloc un nombre de fois **connu à l'avance** ;
- la boucle `while` répète un bloc **tant qu'**une condition reste vraie ; le nombre de tours n'est pas connu à l'avance.

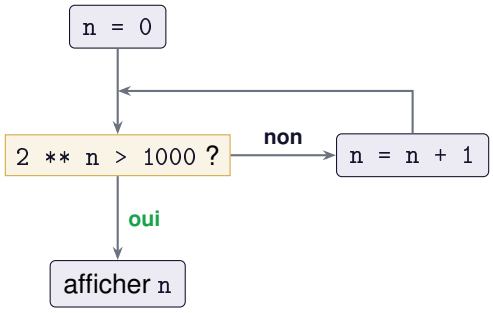
`range(a, b)` engendre les entiers de a **inclus** à b **exclu**, soit $b - a$ valeurs. En particulier, pour parcourir tous les entiers de $[a\,;\,b]$ il faut écrire `range(a, b + 1)` : cet intervalle contient $b - a + 1$ entiers.

Exemple : somme des entiers de 1 à 100
On crée un accumulateur *avant* la boucle, on l'enrichit *dedans*, on l'affiche *après*.

```
s = 0
for k in range(1, 101) :
    s = s + k
print(s)
```

Le programme affiche `5050`, ce que confirme la formule $\dfrac{100 \times 101}{2} = 5050$.

Exemple : chercher le premier entier n tel que $2^n > 1000$
Ici le nombre de tours est inconnu : c'est une boucle `while`.




```
n = 0
while 2 ** n <= 1000 :
    n = n + 1
print(n)
```

La boucle s'arrête au premier n pour lequel la condition devient fausse. Comme $2^9 = 512 \leqslant 1000$ et $2^{10} = 1024 > 1000$, le programme affiche $n = 10$.

Boucle infinie. Une boucle `while` ne s'arrête que si une instruction du bloc finit par rendre la condition fausse. Oublier le `n = n + 1` bloque le programme indéfiniment : c'est l'erreur classique du `while`.

6. Fonctions

Une **fonction** est un bloc d'instructions nommé, que l'on définit une fois avec `def` et que l'on appelle ensuite autant de fois qu'on veut. L'instruction `return` renvoie un résultat et met fin à la fonction.

Exemple : programmer la fonction $f : x \mapsto 3x - 5$

```
def f(x) :
    return 3 * x - 5

print(f(4))          # affiche 7
```

Une fois `f` définie, on peut dresser un tableau de valeurs en une boucle :

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

x	-2	-1	0	1	2
$f(x)$	-11	-8	-5	-2	1

`return` **n'est pas** `print`**.** `return` transmet une valeur, réutilisable dans un calcul : `2 * f(4)` vaut 14. `print` se contente d'afficher à l'écran ; une fonction qui se termine par un `print` ne renvoie rien, et `2 * f(4)` provoquerait alors une erreur.

À retenir

1. `=` affecte, `==` teste. L'affectation se lit de droite à gauche, et `x = x + 1` signifie « augmenter x de 1 ».

2. `/` donne un `float` même quand la division tombe juste ; `//` et `%` donnent le quotient et le reste de la division euclidienne.

3. Les crochets se traduisent en inégalités : fermé $\Rightarrow$ `<=`, ouvert $\Rightarrow$ `<`. Un intervalle se teste avec `a <= x < b`, une intersection avec `and`, une réunion avec `or`.

4. `range(a, b)` s'arrête à $b - 1$. Pour couvrir $[a\,;b]$, écrire `range(a, b + 1)`, soit $b - a + 1$ entiers.

5. `for` quand le nombre de tours est connu, `while` sinon — en s'assurant que la condition finira par devenir fausse.

Le piège à connaître : les `float` sont approchés, donc `0.1 + 0.2 == 0.3` renvoie `False`. On ne teste jamais l'égalité de deux nombres à virgule.