

6 – Comparaisons, booléens, tests

6.1 Problème : compter les points au mölkky



ACTIVITÉ 1

Au jeu de mölkky, chaque joueur marque à son tour de jeu entre 0 et 12 points, qui viennent s'ajouter à son score précédent. Le premier à atteindre un score de 51 gagne. Mais gare ! Quiconque dépasse le score cible de 51 revient immédiatement à 25 points.

Écrire un algorithme demandant un score "Entrer le score : " et un nombre de points marqués "Entrer le gain : ", et qui affiche le nouveau score "Nouveau score : " ou signale une éventuelle victoire "Victoire".

Traiter chacun des cas de l'activité est facile. Pour traiter ces trois cas *simultanément*, il faut utiliser les instructions de **branchement** de ce chapitre.

En pseudo-code, on écrira `si ... sinon si ... sinon`, ce qui donnera dans le langage Python : `if ... elif ... else`.

6.2 Conditions et branchements

L'instruction conditionnelle `si` permet de soumettre l'exécution d'un bloc à une condition :

```
si n > 0 :  
    afficher "le nombre" n "est positif."
```

Pour l'implémentation de cet algorithme en Python, il faut utiliser le mot-clé `if` suivi d'une **condition** puis terminée par le symbole `:`.

Ce qui donne :

```
if n > 0:
    print("le nombre", n, "est positif.")
```

Les conditions Les conditions peuvent être exprimées en terme de **comparaisons numériques** entre deux expressions :

opérateur Python	rôle
>	plus grand que >
<	plus petit que <
>=	supérieur ou égal ≥
<=	inférieur ou égal ≤
==	égal à =
!=	différent de ≠

REMARQUE

Attention à ne pas confondre en python les opérateurs = et ==. Le premier est utiliser pour l'affectation de variable et le deuxième pour tester une condition. **Ce qui n'a RIEN à voir !**

Il n'y a pas ce problème lorsqu'on écrit des algorithmes en **pseudo-code** car l'affectation est notée \leftarrow et le test de condition =.

Par exemple, affecter à la variable `age` la valeur 85 s'écrit :

pseudo-code	$age \leftarrow 85$
Python	<code>age = 85</code>

De même tester si la variable `age` vaut la valeur 80 s'écrit :

pseudo-code	<code>age = 80</code>
-------------	-----------------------

Python	<code>age == 80</code>
--------	------------------------

Bloc alternatif

En plus d'un bloc à n'exécuter que lorsque sa condition est vérifiée, une instruction `si` peut contenir un bloc alternatif : **à n'exécuter que dans le cas contraire.**

Ce bloc alternatif s'écrit avec le mot-clé `sinon`.

On dit alors que l'instruction complète est constituée de deux branches, dont une seule sera choisie lors de l'exécution.

Exemple

Pseudo-code

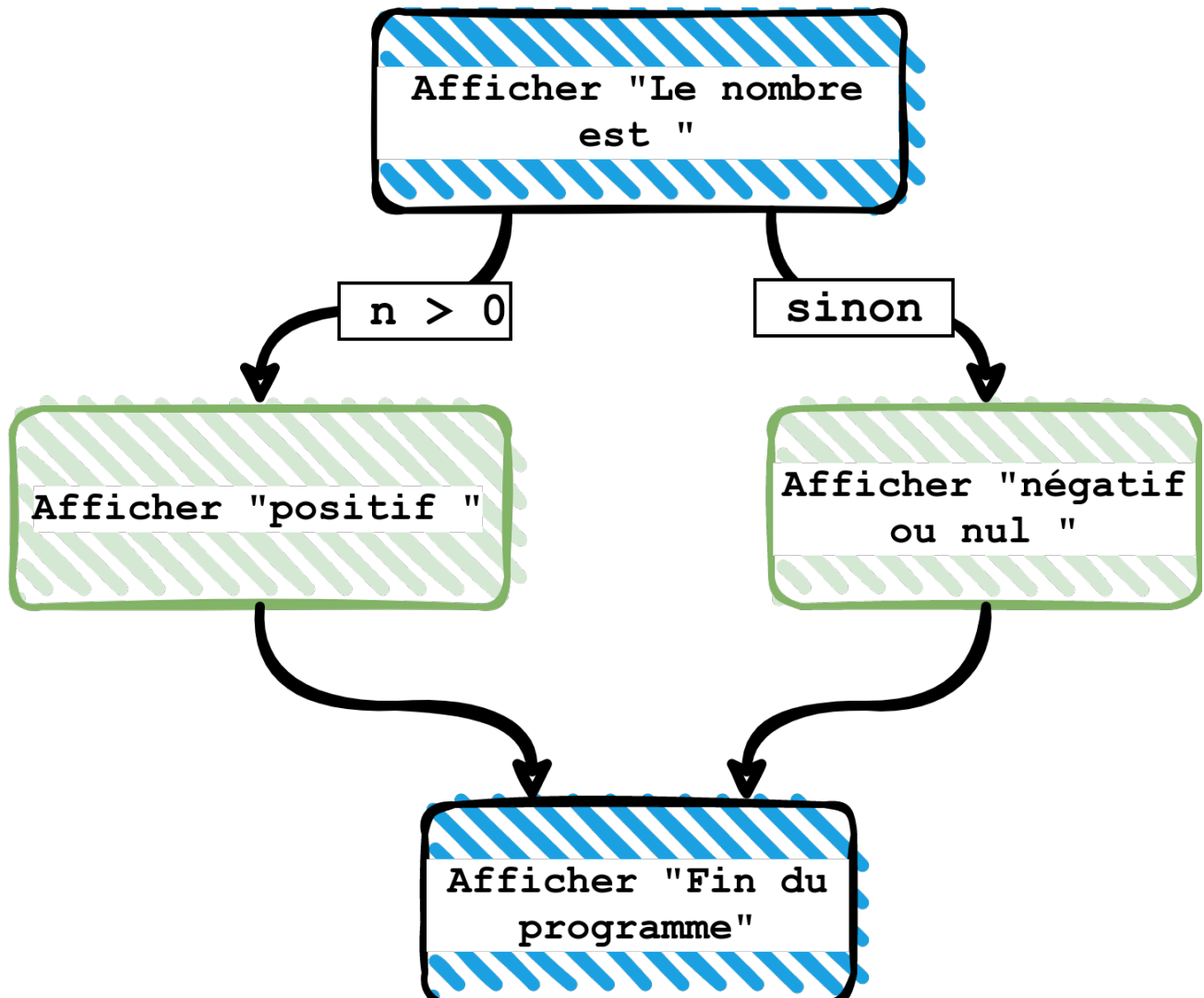
```
si n > 0 :  
    afficher "Positif"  
sinon :  
    afficher "Négatif ou nul"
```

En langage Python, l'instruction `sinon` est traduit par le mot-clé `else` suivi du symbole `:`.

Python

```
if n > 0:  
    print("Positif")  
else:
```

```
print("Négatif ou nul")
```



Trois branches ou plus Enfin, il est possible d'introduire trois branches ou plus. Après un premier test avec le mot-clé `si`, chaque branche suivante peut être ajoutée avec **sa propre condition**.

Pour cela on utilise en pseudo-code le mot clé `sinon si` qui est traduit en Python par `elif`.

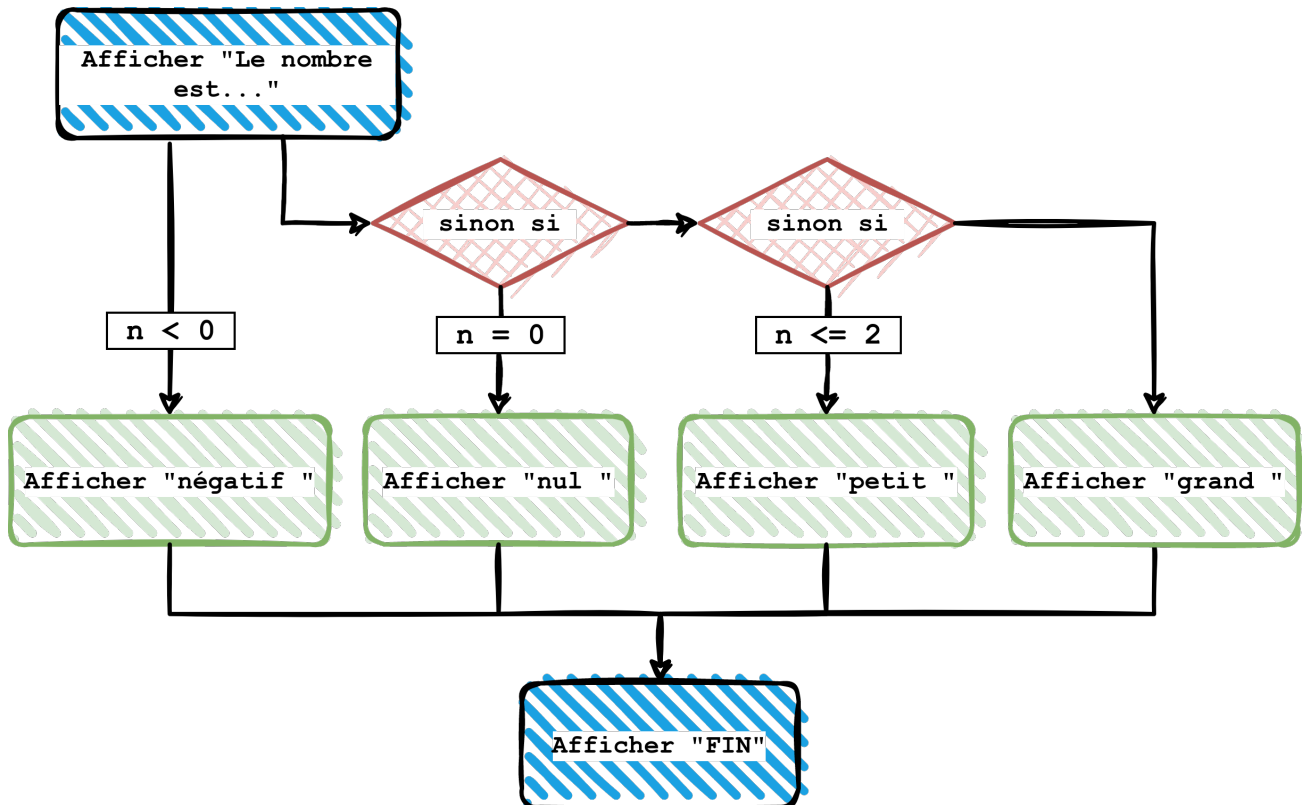
Ce mot-clé est suivi d'une condition terminée par le symbole `:`.

Exemple**Pseudo-code**

```
si n < 0:
    afficher "Négatif"
sinon si n = 0:
    afficher "Nul"
sinon si n <= 2:
    afficher "Petit"
sinon:
    afficher "Grand"
```

Python

```
if n < 0:
    print("Négatif")
elif n == 0:
    print("Nul")
elif n <= 2:
    print("Petit")
else:
    print("Grand")
```



REMARQUE

Remarquez bien que l'**ordre** dans lequel apparaissent les conditions est important.

REMARQUE

Les opérateur de comparaison (tout comme on l'a vu + et *) s'utilisent **aussi** avec **autre chose** que les nombres !

Python compare les chaînes en fonction de l'**ordre lexicographique**.

Exemples :

- "abcd" < "b" est une condition qui est Vrai
- "123" < "13" est une condition qui est Vrai

- attention aux majuscules : "Z" < "a" est Vrai
- attention aux accents : "wagon" < "été" est Vrai

Attention, on ne peut pas comparer une **chaîne de caractère** et un nombre **entier** ou un **flottant**.



ACTIVITÉ 2

Au bowling, on a deux chances pour faire tomber un total de dix quilles. Écrire un algorithme qui demande le nombre de quilles renversées avec chacune des deux boules et affiche "X" si toutes les quilles sont tombées à la première boule, "/" si toutes les quilles sont tombées, et sinon le nombre de quille renversées.

Exemples :

```
>>> Score première boule : 2
>>> Score deuxième boule : 3
5
```

```
>>> Score première boule : 7
>>> Score deuxième boule : 3
/
```

```
>>> Score première boule : 10
>>> Score deuxième boule : 0
X
```

6.3 Expressions booléennes

Les *conditions* sont des expressions algorithmique ordinaire qui produisent un résultat. On les appelle des **expressions booléennes**.

Une expression booléenne admet deux résultats possibles :

pseudo-code	vrai ou faux
Python	True ou False

Exemple

Par exemple, l'expression booléenne $10 < 5$ est évaluée par l'algorithme et produit le résultat `faux`.

Ainsi les deux pseudo-code suivant sont équivalent et n'affichent jamais rien :

```
# algo 1
si 10 < 5:
    afficher "jamais"

# algo 2
si faux:
    afficher "jamais"
```

Il est possible de faire des opérations avec les expressions booléennes grâce à trois **opérateurs** :

- `et`, appelé la *conjonction*
- `ou`, appelé la *disjonction*
- `non`, appelé la *négation*

Exemple

Par exemple, prenons deux expressions booléennes e_1 et e_2 , alors :

$(e_1 \text{ et } e_2)$ est une expression booléenne vraie	si et seulement si e_1 est vraie et e_2 est vraie
$(e_1 \text{ ou } e_2)$ est une expression booléenne vraie	si et seulement si e_1 est vraie ou e_2 est vraie
$(\text{non } e_1)$ est une expression booléenne vraie	si et seulement si e_1 est fausse

Les opérateur booléens permettent donc de combiner plusieurs tests de comparaison et d'égalité **en une seule expression**.

Exemple

Par exemple l'algorithme suivant compare les coordonnées (x_a, y_a) et (x_b, y_b) de deux points x et y et affiche des informations sur leurs positions relatives.

```
si  $x_a = x_b$  et  $y_a = y_b$  :  
    afficher "points confondus"  
sinon si  $x_a = x_b$  ou  $y_a = y_b$  :  
    afficher "points alignés horizontalement ou verticalement"  
sinon :  
    afficher "points indépendants"
```

ce qui se traduit en Python par :

```
if  $x_a == x_b$  and  $y_a == y_b$ :  
    print("points confondus")
```

```
elif xa == xb or ya == yb:  
    print("points alignés horizontalement ou verticalement")  
else:  
    print("points indépendants")
```

REMARQUE

Priorité des opérateurs booléens Comme pour les expressions arithmétiques, on a établi des conventions de priorité :

la négation est **plus prioritaire que** la conjonction qui est **plus prioritaire que** la disjonction.

L'expression

a ou non b et c

doit être comprise comme

a ou $((\text{non } b) \text{ et } c)$.

Mais, **il ne faut pas hésiter à mettre des parenthèses** pour faciliter la lecture !

REMARQUE

Paresse des opérateurs booléens en Python

conjonction : L'expression booléenne $e1$ and $e2$ n'est vraie que si $e1$ est vraie et $e2$ est vraie. Ainsi, **il suffit que** $e1$ soit fausse pour que l'expression complète soit fausse, et ceci, sans même avoir à évaluer l'expression $e2$!

Et c'est exactement ce que fait l'interprète Python : il tente d'éviter

d'évaluer une expression si le résultat final est déjà connu. Dans notre exemple, l'interprète fonctionne ainsi :

1. évaluer e_1
2. si e_1 est `faux` alors la conjonction vaut `faux`
3. sinon, évaluer e_2 (et la conjonction à la même valeur que e_2)

Exemple

Par exemple, imaginons un algorithme qui indique si un nombre a est divisible par un nombre b . Cette situation est caractérisée par le fait que a vaut 0 ou que le reste de la division de a par b vaut 0. Le reste s'obtient grâce à l'opérateur $a \bmod b$ (appelé *modulo* et qui se calcule en Python en écrivant `a % b`)

Cependant, l'opérateur *modulo* n'a pas de sens si b vaut 0 (car on ne peut pas, mais **vraiment pas** diviser par 0!!!!), ce qui déclenche une erreur en Python.

Ainsi, la condition complète pour savoir si a est divisible par b sera donc :

- soit $a = 0$;
- soit $b \neq 0$ **et** $a \bmod b = 0$

En l'écrivant ainsi dans un programme en Python, l'évaluation paresseuse du **et** évite l'évaluation problématique de $a \bmod b$ lorsque b vaut 0.

Algorithme correct :

```
si  $a = 0$  ou ( $b \neq 0$  et  $a \bmod b = 0$ ) :  
    afficher "divisible"  
sinon :  
    afficher "pas divisible"
```

Algorithme incorrect qui engendre une erreur en Python lorsque $b = 0$:

```
si  $a = 0$  ou ( $a \bmod b = 0$  et  $b \neq 0$ )  
    afficher "divisible"  
sinon :  
    afficher "pas divisible"
```



ACTIVITÉ 3

Implémenter un programme qui demande deux nombres à l'utilisateur et affiche "divisible" si le premier est divisible par le second et qui affiche "pas divisible" sinon.