

Chap. 2 – Programmer en Python (pa.dilla.fr/15)



ACTIVITÉ

D'après toi, pourquoi HTML et CSS ne sont **pas** des langages de programmation ?

CORRECTION

HTML et CSS sont incapables d'effectuer le moindre calcul.

On peut aussi remarquer qu'il manque des instructions essentielles comme les répétitions, les branchements conditionnels ou encore les affectations.

Python est un langage de programmation qui peut être utilisé de différentes façon. Dans ce cours tu verras :

- le mode *interactif* dans des notebooks
- le mode *programme* dans l'interface de développement (IDE) VSCodium.

1 – Le mode interactif

Le mode interactif se comporte un peu comme une calculatrice. Nous allons explorer ce mode à travers les **notebooks**.

1.1 – Les notebooks

Un *notebook* est une page HTML qui contient des blocs de texte et des blocs de code :

- les blocs de **texte** permettent d'afficher et de modifier du texte écrit en langage Markdown. Cette simplification du HTML permet d'afficher du texte (normal, gras, italique, etc.), des listes, des tableaux, des liens, du code ou encore des média comme les images et les vidéos.
- les blocs de **code** permettent de saisir, modifier et d'exécuter du code Python. L'interprète Python est chargé avec la page HTML puis ensuite il calcule et affiche les résultats.

Exemple

La page de ce cours consultable à l'adresse <https://pa.dilla.fr/15> est un **notebook**.

Exemple

```
[ ]: # Ceci est un bloc de CODE.  
# Les lignes qui commencent pas un # sont des commentaires  
# Les commentaires ne seront pas interprétés par Python  
#  
# Les blocs de code se reconnaissent facilement car ils  
# possèdent une indication dans la marge "In []" ou "Entrée []"  
# Si le bloc a été exécuté, alors un compteur apparait dans "[...]"  
  
print("Dans le notebook, ceci est un bloc de code")
```

Pour **modifier un bloc de texte ou de code** tu peux utiliser la souris ou le clavier :

1. méthode avec la souris :

- double-cliquer sur ce bloc de texte pour l'*éditer*,
- modifier/corriger le texte
- cliquer sur le bouton `Run` ou `Exécuter` pour interpréter ce bloc et sortir du mode *édition*

2. méthode au clavier :


- utiliser les flèches <↑> et <↓> pour sélectionner le bloc de texte à modifier
- appuyer sur <Entrée> pour *éditer* le bloc
- modifier/corriger le texte
- pour interpréter le résultats tu peux utiliser au choix :
 - <Ctrl> + <Entrée> pour interpréter le bloc ou
 - <Shift> + <Entrée> pour interpréter le bloc et passer au bloc suivant.

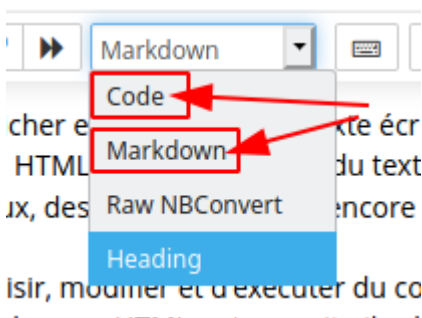
REMARQUE

Souvent quand on ouvre un notebook, aucun bloc de code n'a été exécuté. il faut donc passer *dans l'ordre* par tous les blocs de code et les exécuter.

C'est pourquoi, quand on lit notebook, il est pratique d'utiliser le raccourcis clavier <Shift> + <Entrée> permettant de passer d'un bloc au suivant.

Pour **ajouter des blocs** (de code ou de texte) :

1. tu peux utiliser dans les outils le bouton + .
2. Si tu souhaites que le bloc soit du code, sélectionne Code dans la liste déroulante.
3. Si tu souhaites un bloc de texte, sélectionne alors Markdown.



**ACTIVITÉ**

Accède à la version *notebook* de ce chapitre (<https://pa.dilla.fr/15>).

1. **Corrige** la faute d'orthographe de ce bloc de texte.
2. **Ajoute** juste en dessous un bloc de code et **écris** y : `1+2`.
3. Puis fait **calculer** le résultats par l'interprète Python.

1.2 – Arithmétique

En Python, on peut saisir des combinaisons arbitraires d'opérations arithmétiques, en utilisant notamment les quatre opérations les plus communes.

Exemple

```
2 + 5 * (10 - 1 / 2)
```

REMARQUE

Quelques remarques

1. Les symboles des quatre opérations sont + (addition), - (soustraction), * (multiplication) et / (division).
2. La priorité des opérations est usuelle et on peut utiliser les parenthèses.
3. Les espaces sont purement décoratifs : ils rendent plus lisibles mais n'agissent pas sur la priorité des opérations. Par exemple :
`1+2 * 3` renvoie 7 et non pas 9!

Remarque bien que l'interprète Python n'accepte pas les expressions qui sont mal formées ou comportent des erreurs.

**ACTIVITÉ**

Dans le bloc de code ci-dessous, **écris** $1 + * 2$.

Indique le type d'erreur affiché.

CORRECTION

Lorsque l'on fait des erreurs de syntaxe car l'expression est mal formée, l'interprète affiche `SyntaxError`. Pour aider l'utilisateur, l'interprète montre l'endroit exact de l'erreur avec le symbole `^` qui pointe sous le `*`.

**ACTIVITÉ**

Pour répondre aux questions suivantes, tu peux ajouter des blocs de texte ou des blocs de code...

En mathématique, est-il possible de **diviser un nombre par zéro** ?

Et en **Python** ?

1.3 – Les nombres

En Python, les nombres sont de deux *types* :

- soit des **entiers relatifs** (appelés *entiers*)
- soit des **nombres décimaux** (appelés *flottants*).

REMARQUE

Attention, les nombres flottants s'écrivent à l'*anglo-saxonne* avec un point 3.14 à la place de la virgule 3,14.

**ACTIVITÉ**

Indique ce qu'affiche les commandes suivantes :

1. Commande 1 : $1,2 + 3,4$.
2. Commande 2 : $1,2 * 3$

D'après toi, à quoi sert le symbole virgule , en Python ?

En Python :

- les *entiers* sont de **taille arbitraire** ;
- les *flottants* en revanche ont une **capacité limitée** et sont souvent des approximations qui ne représentent alors qu'une partie des nombres décimaux.

Quand tu en as le choix, travaille de préférence avec les entiers.

1.4 – Autres opérations

**ACTIVITÉ**

Détermine le type de nombre que l'on obtient lorsque l'on divise entre eux deux nombres entiers.

CORRECTION

La division avec l'opération `/` donne un nombre flottant. Ainsi `10/2` donne `5.0` et `7/2` donne `3.5`. Ces deux résultats sont de types *flottant* car ils sont écrits avec une virgule.

**ACTIVITÉ**

Effectue à la main les divisions euclidiennes `7/2` et `98/10`.

CORRECTION

`7/2` donne 3 et il reste 1. L'égalité $7 = 3 \times 2 + 1$ est vérifiée.

`98/10` donne 9 et il reste 8. L'égalité $98 = 9 \times 10 + 8$ est vérifiée.

Pour effectuer la **division entière**, il faut utiliser l'opération `//`. Ainsi, `a // b` peut être vu comme le plus grand nombre entier inférieur ou égal à a/b .

Pour obtenir le **reste de la division euclidienne**, on utilise l'opération `%` (qui se dit "*modulo*").

**ACTIVITÉ**

Vérifie à l'aide des opérations *division entière* et *modulo* les résultats de l'activité précédente.

CORRECTION

Utiliser quatre blocs de code pour afficher les résultats des calculs suivants :

```
7//2
7 % 2
98 // 10
98 % 10
```

L'opération `**` calcule la **puissance** d'un nombre.
Ainsi, 10^3 s'écrit `10**3`.

**ACTIVITÉ**

Calcule dans un bloc de code 2^{1024} puis dans un autre bloc de code : $(2.0)^{1024}$.

Propose une explication à tes observations.

CORRECTION

`2**1024` est un nombre entier et se calcule correctement.

En revance, `2.0**1024` est un nombre flottant trop grand pour être calculé. `OverflowError` indique que le résultat est hors intervalle.


1.5 – Variables

Les résultats calculés peuvent être **mémorisés** afin d'être utilisés plus tard.

```
[ ]: a = 1 + 1
```


Une variable peut être imaginée comme un **emplacement en mémoire** portant une **étiquette** et contenant une **valeur**.

Exemple

$a = 2$ se représente par un emplacement a contenant la valeur 2 : 

La notation $a =$ permet de donner un nom à l'information mémorisée. Ici, l'interprète Python calcule le résultat de $1 + 1$ et le mémorise dans la **variable** a .

Aucun résultat n'est affiché. On accède à la valeur mémorisée en utilisant le nom a .

```
[ ]: a
```

Le bloc ci-dessous affiche le résultat du calcul $a * (a + 1)$:

```
[ ]: a * (a+1)
```

Le symbole $=$ utilisé en Python pour définir la variable a désigne une opération d'**affectation**. En algorithmique, on note l'affectation par le symbole ' \leftarrow ', ce qui donne par exemple : $a \leftarrow 2$.

Ce symbole attend à *gauche* un nom de variable et à *droite* une expression.

On peut donner une nouvelle valeur à la variable a avec une nouvelle affectation. Cette valeur *remplace* la précédente.

```
[ ]: a = 3
a * (a+1)
```

Le calcul de la nouvelle valeur de a peut utiliser la valeur courante de a .

```
[ ]: a = a + 1
a
```

REMARQUE

Un nom de variable peut être formé de plusieurs caractères (lettre, chiffres et tiret bas). Il est recommandé de :

- ne pas utiliser de caractères accentués
- se limiter aux caractères minuscules.

```
[ ]: cube = a * a * a
      ma_variable = 42
      ma_variable2 = 2021
```

1.6 – État

L'ensemble des associations entre des noms de variables et des valeurs constitue **l'état** de l'interprète Python.

L'état évolue en fonction des instructions exécutées. Les instructions qui modifient l'état sont dites *à effet de bord*.



ACTIVITÉ

À partir de l'état initial

a	b	c	d
1	2	3	-12

, les instructions suivantes sont exécutées :

$a = c - a$

$e = b + c$

$d = a$

$a = 7$

Après la première instruction $a = c - a$, l'état initial devient alors :

a	b	c	d
2	2	3	-12

.

Indique les états suivants après l'exécution des trois dernières instructions.

CORRECTION

L'instruction `e = b + c` affecte la variable `e` pour la première fois. Celle-ci fait donc son apparition dans l'état :

a	b	c	d	e
2	2	3	-12	5

L'instructions `d = a` affecte à `d` la valeur pointée par `a`. Il est important de comprendre que `a` et `d` sont deux emplacements différents et donc indépendants. Ainsi, l'instruction suivante n'aura pas d'effet sur la variable `d`

a	b	c	d	e
2	2	3	2	5

La dernière instruction `a = 7` affecte à la variable `a` la valeur 7.

a	b	c	d	e
7	2	3	2	5

On peut voir une modélisation de l'état sur le site suivant : <https://pythontutor.com>

2 – Le mode programme

Le mode programme de Python consiste à écrire une suite d'instructions dans un fichier et à les faire exécuter par l'interprète Python.

Cette suite d'instruction s'appelle un **programme** ou encore un **code source**.

2.1 – VSCodium

Pour écrire des programmes, le plus simple est d'utiliser un **environnement de développement** (IDE). Cette année, nous allons essentiellement utiliser VSCodium.

- Après avoir ouvert le logiciel, créer un nouveau fichier (<Ctrl> + <N>).
- Sauvegarder le (<Ctrl> + <S>) en donnant un nom qui a pour *extension* `.py` (par exemple `test.py`).
- Puis une fois votre programme écrit, pour l'exécuter :
 - en mode *débogage* : <F5>
 - en mode *sans débogage* : <Ctrl> + <F5>

2.2 – Affichage

En mode programme, les résultats calculés ne sont plus affichés. Il faut utiliser pour ceci une instruction d'affichage.

En Python, elle s'appelle `print`. Ainsi le programme `test.py` contenant l'unique ligne `print(3)` affichera 3.

L'instruction `print` admet une expression arbitraire. Elle commence d'abord par calculer le résultat de cette expression puis l'affiche à l'écran.

Par exemple l'instruction `print(1+3)` calcule d'abord l'expression `1+3` puis affiche 4 à l'écran.

2.3 – Affichage des textes

On peut donner à l'instruction `print` un message à afficher, *écrit entre guillemets*.

L'instruction `print("Bonjour tout le monde")` affiche le message `Bonjour tout le monde` à l'écran.

Le texte écrit entre guillemets est appelé une **chaîne de caractères**.

REMARQUE

Les guillemets englobants ne sont pas affichés.

Les guillemets peuvent être doubles `"..."` ou simples `'...'`.

Les guillemets ouverts doivent être impérativement fermés sinon il y aura une exception (erreur) de type `syntaxError`.

Une chaîne de caractère est arbitraire et n'est pas interprétée par Python.



ACTIVITÉ

Indiquer ce qu'affichent les deux instructions suivantes :

- `print(1+3)`
- `print("1+3")`

Attention aux opérations avec les chaînes de caractères.

- `"Hello" + 2` est une expression **invalid**e car l'opération `+` entre une chaîne et un entier n'a aucun sens.
- `"Hello" + "World"` est une expression **valide**e car l'opération `+` entre deux chaînes de caractères les concatène (c'est-à-dire les assemble et produit une chaîne de caractère contenant le texte `"HelloWorld"` (sans espace))
- `"Hello" * "World"` est une expression **invalid**e car l'opération `*` n'a pas

de sens entre deux chaînes de caractères.

- `"Hello" * 3` est une expression **valide** car l'opération `*` entre une chaîne et un entier est définie et est équivalente à concaténer `n` fois la chaîne de caractère. Ici l'expression est équivalente à `"Hello" + "Hello" + "Hello"` et après interprétation produit comme résultat `"HelloHelloHello"`.

2.4 – Séquence d'instructions

Un programme est généralement constitué de plusieurs instructions. Chaque instruction est écrite sur une ligne.



ACTIVITÉ

Détailler ce que produit l'exécution du programme suivant :

```
a = 34
b = 21 + a
print(a)
print(b)
```

CORRECTION

Ce programme affiche deux entiers à l'écran, sur deux lignes :

```
34
55
```

REMARQUE

Pour afficher plusieurs expressions sur **une seule ligne**, il suffit d'utiliser

une seule instruction `print` et de mettre en argument les expressions séparées par une virgule :

- `print(a,b)` affichera les deux éléments sur la même ligne : 34 55
- `print("la somme de", a, "et de", b, "vaut", a+b)` affichera sur une seule ligne les 6 expressions et donnera alors : la somme de 34 et de 55 vaut 89

2.5 – Interagir avec l'utilisateur

Pour permettre l'interaction du programme avec l'utilisateur, il faut mettre en place une **interface**.

L'interface la plus simple consiste à utiliser l'instruction `input` qui permet de récupérer des caractères tapés au clavier par l'utilisateur.

Cette instruction interrompt l'exécution du programme et attend que la saisie se termine lorsque la touche <Entrée> est appuyée.

```
[ ]: saisie = input()
      print("la chaîne de caractère saisie est:", saisie)
```

La chaîne de caractère ainsi récupérée pourra être convertie, si besoin, en nombre entier en utilisant l'instruction `int`.

Par exemple le programme ci-dessous demande l'âge de l'utilisateur. Le nombre saisi est converti en nombre entier puis un calcul est effectué :

```
[ ]: texte = input("ton âge ?")
      age = int(texte)
      print("Dans 10 ans, tu auras", age+10)
```



ACTIVITÉ

Que renvoie le programme précédent si on supprime la deuxième ligne?

2.6 – Un programme complet



ACTIVITÉ

Écris un programme qui demande l'année de naissance à l'utilisateur puis affiche l'âge qu'il aura en 2048.

Pour chaque ligne du programme, **représente** l'état de l'interprète.

CORRECTION

```
[ ]: # Calcul de l'âge en 2048
saisie = input("Entrez votre année de naissance : ")
annee = int(saisie)
age = 2048 - annee
print("Vous aurez", age, "ans en 2048.")
```

CORRECTION

ligne	état	interface
2	<div>saisie</div> <div>"2003"</div>	affichage : Entrez votre année de naissance : saisie : 2003
3	<div>saisie annee</div> <div>"2003" 2003</div>	
4	<div>saisie annee age</div> <div>"2003" 2003 45</div>	
5	<div>saisie annee age</div> <div>"2003" 2003 45</div>	affichage : Vous aurez 45 ans en 2048.

3 – Exercices



ACTIVITÉ 1

Observer les résultats obtenus par l'expression $5 - 3 - 2$ et par l'expression $1 / 2 / 2$.

En déduire la manière dont sont interprétées les soustractions et les divisions enchaînées.



ACTIVITÉ 2

Réécrire les expressions suivantes en explicitant toutes les parenthèses :

1. $1 + 2 * 3 - 4$
2. $1+2 / 4*3$
3. $1-a+a*a/2-a*a*a/6+a*a*a*a/24$

CORRECTION

1. $(1 + (2*3)) - 4$
2. $1 + ((2/4)*3)$
3. $((((1 - a) + ((a*a)/2)) - ((a*a*a)/6)) + ((a*a*a*a)/24))$



ACTIVITÉ 3

Réécrire les expressions suivantes en utilisant aussi peu de parenthèses

que possible sans changer le résultat.

1. $1 + (2 * (3 - 4))$
2. $(1 + 2) + ((5 * 3) + 4)$
3. $(1 - ((2 - 3) + 4)) + (((5 - 6) + ((7 - 8) / 2)))$

CORRECTION

1. $1 + 2 * (3 - 4)$
2. $1 + 2 + 5 * 3 + 4$
3. $1 - (2 - 3 + 4) + 5 - 6 + (7 - 8) / 2$



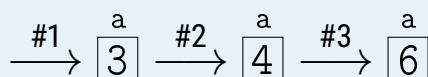
ACTIVITÉ 4

Déterminer la valeur affichée par l'interprète Python après la séquence d'instructions suivante :

```
a = 3
a = 4
a = a+2
a
```

CORRECTION

Après ces instruction, l'interprète affiche 6. Voici les états ligne par ligne :





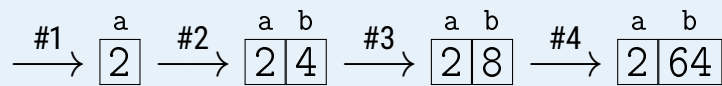
ACTIVITÉ 5

Déterminer la valeur affichée par l'interprète Python après la séquence d'instructions suivante :

```
a = 2
b = a*a
b = a*b
b = b*b
b
```

CORRECTION

L'interprète affiche la valeur de la variable `b`, soit 64. Voici les états successifs de l'interprète :



ACTIVITÉ 6

(Capytale : 3f7e-77330) Dans un notebook, **initialiser** une variable `a` avec la valeur 2, puis **répéter** dix fois l'instruction `a = a * a`.

Observer le résultat. Quelle puissance de 2 a-t-on ainsi calculé ?

Recommencer en affectant cette fois-ci la valeur 2.0 à la variable `a`. **Observer** puis **interpréter** le résultat.

CORRECTION

En procédant ainsi l'interprète a calculé 2^{1024} .

En affectant `2.0` à la variable `a`, on obtient `inf` qui signifie *infini* et indique que le nombre flottant n'est pas représentable car il est trop grand.

**ACTIVITÉ 7**

Indiquer ce qu'affichent les instructions suivantes `print("1+")` et `print(1+)`.

CORRECTION

Dans le premier cas la chaîne de caractère `1+` est affichée. Dans le second cas, l'expression (addition entre un nombre entier et rien du tout) comporte une erreur de syntaxe. L'exception `SyntaxError` est levée.

**ACTIVITÉ 8**

Indiquer ce qu'il se passe quand on exécute le code suivant :

```
a = input("saisir un nombre : ")
print("le nombre suivant est ", a+1)
```

Rectifier si nécessaire.

CORRECTION

L'expression `a+1` est incorrecte puisqu'elle demande d'effectuer une addition entre `a` qui est une chaîne de caractère et le nombre entier `1`. Cette

opération n'est pas définie en Python.

Pour corriger ce code, il faut par exemple ajouter dans une ligne intermédiaire le code `a = int(a)`.



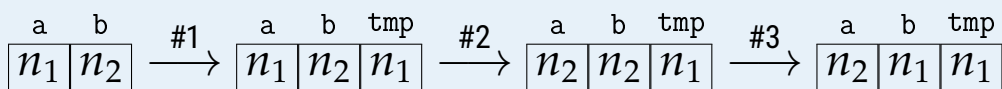
ACTIVITÉ 9

Indiquer ce que fait la séquence d'instruction suivante en supposant qu'à l'origine les variables `a` et `b` contiennent un nombre entier.

```
tmp = a
a = b
b = tmp
```

CORRECTION

Détaillons les états successifs de l'interprète en supposant que la variable `a` contienne la valeur n_1 et que la variable `b` contienne la valeur n_2 .



À la fin de l'instruction, les valeurs enregistrées dans les variables `a` et `b` ont été **permutées**.



ACTIVITÉ 10

(Capytale : f6db-77338) On met deux entiers dans deux variables `a` et `b`, par exemple 55 et 89. On remplace le contenu de `a` par la somme de celui

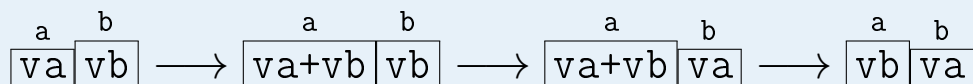
de a et de b . Puis on remplace le contenu de b par le contenu de a moins le contenu de b ? Enfin on remplace le contenu de a par son contenu moins celui de b .

Que contiennent a et b à la fin de ces opérations ?

Programme cet algorithme en Python.

CORRECTION

Notons va et vb les valeurs initiales des variables a et b .



À la fin de cette séquence d'instructions, les valeurs de a et de b ont été permutées.

```
[ ]: a = 55
      b = 89

      a = a + b
      b = a - b
      a = a - b
      print("a vaut", a, "et b vaut", b)
```



ACTIVITÉ 11

(Capytale : d677-77366) **Écrire** un programme qui demande à l'utilisateur les longueurs (entières) des deux côtés d'un rectangle et affiche son aire.

```
[ ]: texte_11 = input("Saisir la longueur du premier côté :")
      texte_12 = input("Saisir la longueur du second côté :")
      l1 = int(texte_11)
      l2 = int(texte_12)
      aire = l1 * l2
      print("L'aire du rectangle vaut", aire)
```

**ACTIVITÉ 12**

(Capytale : 5b27-77369) **Écrire** un programme qui demande d'entrer une base (entre 2 et 36) et un nombre dans cette base et qui affiche ce nombre en base 10.

La notation `int(chaine, base)` permet de convertir une chaîne représentant un entier dans une base donnée en un entier Python.

```
[ ]: txt_base = input("Saisir la base :")
txt_nb = input("Saisir le nombre :")
base = int(txt_base)
nb = int(txt_nb, base)
print("le nombre",txt_nb,"écrit en base",
      txt_base,"s'écrit en base 10 :", nb)
```

**ACTIVITÉ 13**

(Capytale : c513-77371) **Écrire** un programme qui demande à l'utilisateur d'entrer un nombre de secondes et qui l'affiche sous la forme d'heures/minutes/secondes.

```
[ ]: txt_seconde = input("Saisir un nombre de secondes :")
seconde = int(txt_seconde)
minute = seconde // 60
seconde = seconde % 60

heure = minute // 60
minute = minute % 60

print(heure,"h", minute,"min", seconde,"s")
```

**ACTIVITÉ 14**

(Capytale : 70d6-77374) On souhaite écrire un programme qui demande à l'utilisateur un nombre d'œufs et affiche le nombre de boîtes de 6 œufs nécessaires à leur transport. On considère ce programme qui utilise la division euclidienne.

```
n = int(input("combien d'œufs : "))  
print(n//6)
```

Tester ce programme sur différentes entrées.

1. Sur quelles valeurs de n ce programme est-il correct ?
2. Pourquoi n'est-il pas correct de remplacer $n // 6$ par $n // 6 + 1$?
3. Proposer une solution correcte.

CORRECTION

1. Ce programme n'est correct que pour les nombres n multiples de 6.
2. Avec la modification proposée, le programme est correct pour les valeurs qui ne sont pas multiples de 6, mais est devenu incorrect pour les valeurs multiples de 6.
3. Un programme correct est :

```
n = int(input("combien d'œufs : "))  
print((n+5) // 6)
```


4 – La bibliothèque `microbit`

4.1 – Découverte de l'environnement `Micro:bit`

4.1.1 – Découverte de la carte `Micro:bit`

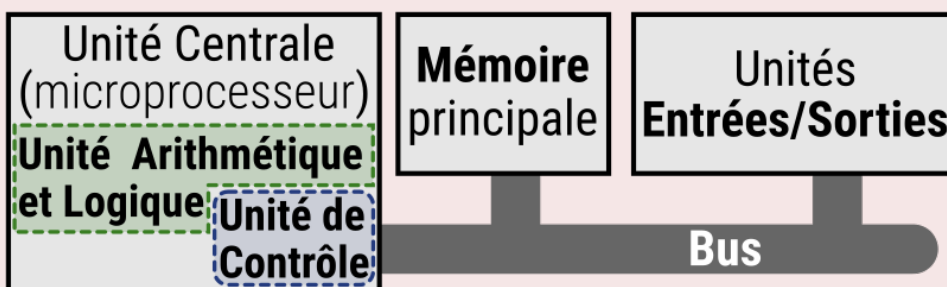


ACTIVITÉ 1

Tu as à ta disposition une carte `Micro:bit`. Répondre aux questions suivantes concernant la carte.

1. **Observer** la carte et **énumérer** les différents constituants de la carte `Micro:bit`.
2. **Indiquer** le rôle de chacun.
3. **D'après vous**, à quoi sert le composant *Processor* ? que contient-il ?

Comme tout modèle d'ordinateurs, la structure de la carte `Micro:bit` est conforme à un schéma inventé en 1945 et qui a peu évolué depuis : l'**architecture de Von Neumann**.



Cerveau de la machine, l'unité centrale (ou *microprocesseur*) est constituée d'une **unité arithmétique et logique** (pour effectuer les calculs) et d'une **unité de contrôle** (pour piloter les échanges à travers le **Bus**).

La **mémoire** contient à la fois les données et les programmes.

Les périphériques d'**entrées/sorties** permettent d'envoyer ou de recevoir des données avec l'extérieur (clavier, souris, écran, imprimante, etc.).

4.1.2 – Premier programme avec la carte Micro:bit



ACTIVITÉ 2

D'après toi, que fait le programme ci-dessous ?

```
from microbit import *  
e0 = button_a.is_pressed()  
if e0:  
    display.show("1")  
else:  
    display.show("0")
```

Ouvrir l'éditeur Mu-editor, **recopier** ce programme et le **téléverser** sur la carte (bouton Flasher )

2. **Observe** le fonctionnement de la carte et **propose** une ou plusieurs interprétations (pour confirmer/infirmes tes observations, tu peux manipuler les boutons présents sur la carte : A / B / Reset).

Présenter tes réponses au professeur.

3. **Indique** le nombre de fois qu'est exécutée l'instruction conditionnelle.
4. **D'après toi**, combien de fois doit être exécutée l'instruction conditionnelle pour que la carte reste à l'écoute des entrées et réagisse chaque fois que l'état du bouton A change ?
5. **Améliore** le programme en tenant compte de tes observations et de tes réponses aux questions précédentes.

Présente la carte programmée au professeur.

La fonction `button_a.is_pressed()` peut être avantageusement remplacée par un appel à la fonction `button_a.was_pressed()`.

Son appel renvoie une valeur booléenne `True` (vrai) ou `False` (faux) et permet de savoir si le bouton A a été pressé depuis la dernière fois que cette fonction a été appelée. Cette fonction renvoie `True` s'il a été pressé entre deux appels et `False` sinon.

6. **Modifie** le programme pour que l'affichage 0/1 change uniquement lorsque l'on presse le bouton A (et pas lorsqu'on le relâche).

4.2 – Commandes de base pour programmer avec `Micro:bit`

Dans la suite, tu utiliseras l'éditeur `mu-editor` pour programmer ta carte `Micro:bit`. Si tu le souhaites tu pourras, **sur ton temps personnel**, essayer de programmer la carte avec VSCode.

3.2.1 – La bibliothèque `microbit`

En Python, les entrées/sorties de la carte `Micro:bit` ne sont pas *nativement* accessibles. Afin de pouvoir utiliser les fonctions prévues à cet effet, il faut utiliser la bibliothèque `microbit`.

Exemple

Pour importer cette bibliothèque, il faut utiliser la commande :

```
from microbit import *
```

L'utilisation de cette bibliothèque nous permettra d'utiliser différentes fonctionnalités de la carte `Micro:bit` :

- **Image** pour créer et manipuler les images.
- **Button** avec deux *instances* `button_a` et `button_b` pour connaître l'état des boutons.
- **Pin** avec différentes *instances* en fonction du type de broche (par exemple `pin0`, `pin1` et `pin2`).
- **display** pour gérer l'écran de LED
- **accelerometer** pour interroger l'accéléromètre
- **compass** pour manipuler et interroger la boussole
- **music** pour créer et manipuler de la musique
- **speech** pour faire parler le `Micro:bit`
- **radio** pour communiquer entre plusieurs `Micro:bit` via un protocole simple.

4.2.2 – Micropython dans la carte `Micro:bit`

Lorsque la carte `Micro:bit` est flashée par `mu-editor` avec la bibliothèque `microbit`, elle contient un noyau *micropython*. Il est alors possible d'écrire du code qui sera interprété par ce noyau micropython de la carte.

Pour accéder à l'interprète Python de la carte, il suffit de cliquer sur l'icône REPL de l'application.

Exemple

Pour utiliser le terminal de la carte `Micro:bit` :

1. Flasher la carte `Micro:bit` avec micropython.
2. Ouvrir le terminal de la carte en cliquant sur REPL
3. Écrire le code ci-dessous :

```
print("Hello, World!")
```

4.2.3 – Afficher un texte

Le module `display` permet de gérer l'écran de LED de `Micro:bit`. Les fonctionnalités offertes par ce module sont accessibles en ajoutant un point `'.'` après `'display'` puis en écrivant le nom de la fonction à appeler :

- `display.show(mon_texte)` permet d'afficher la chaîne de caractère contenue dans la variable `mon_texte`
- `display.scroll(mon_texte)` permet de la `Micro:bit`
- `display.clear()` permet d'effacer l'écran.



ACTIVITÉ 3

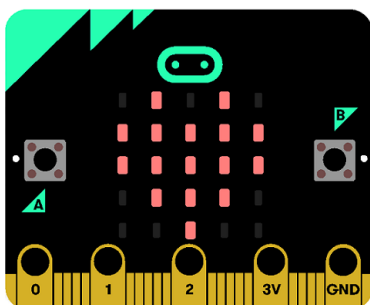
`Micro:bit` le mot Hello puis afficher la chaîne World!.

CORRECTION

```
[ ]: from microbit import *
display.scroll("Hello,")
display.show("World!")
```

4.2.4 – Des images

La classe `Image` contient comme *attributs* de nombreuses images pré-programmées. Ces constantes sont accessibles grâce à un point `'.'` placé après `'Image'`. Par exemple l'image du coeur est accessible via l'instruction `Image.HEART` :



Comme pour les textes, les images s'affichent grâce à l'instruction `display.show()`.

Exemple

Voici par exemple comment afficher successivement des images : une image est affichée pendant une seconde puis ensuite une autre image s'affiche. Une seconde plus tard, l'écran s'efface.

```
from microbit import *
display.show(Image.HAPPY)
sleep(1000)
display.show(Image.ANGRY)
sleep(1000)
display.clear()
```

Il faut mettre le `Micro:bit` en pause. Sinon l'utilisateur n'aura pas le temps de tout voir. La commande `sleep(duree)` arrête donc la carte pendant une durée `duree` exprimée en milliseconde par un nombre entier (int).

4.2.5 – Des boutons

Pour connaître les états des boutons de la carte `Micro:bit` on utilise les classes `button_a` (pour le bouton A de la carte) et `button_b` (pour le bouton B).

Comme d'habitude, les fonctionnalités offertes par ces classes sont accessibles via un point `'.'` placé après le nom de la classe :

- `.get_presses()` renvoie le **nombre** de fois que le bouton a été pressé depuis le dernier appel de cette fonction
- `.is_pressed()` renvoie une **valeur booléenne** (vrai `True` ou fausse `False`) qui indique si le bouton est *actuellement* pressé

- `.was_pressed()` renvoie une **valeur booléenne** qui indique si le bouton a été pressé depuis le dernier appel de cette méthode.



ACTIVITÉ 4

Écris un code permettant d'afficher le nombre de fois que le bouton B a été pressé dans les 10 secondes qui ont suivies l'alimentation de la carte.

(bon à savoir) : la fonction `str(nb)` permet de transformer le nombre en entier `nb` en une chaîne de caractère qu'on peut `Micro:bit`.

CORRECTION

```
[ ]: from microbit import *
      sleep(10000)
      display.scroll(str( button_a.get_presses() ))
```

Une carte `Micro:bit` est un objet connecté qui, entres autres, est à l'écoute de certains **évènements**. Pour programmer cette capacité d'écoute, il est possible d'appliquer le principe fondamental des objets connectés : créer une **boucle infinie** `while True:`.

À chaque tour de cette boucle, le programme doit vérifier la réalisation de l'évènement attendu.

Lorsque l'évènement se réalise, il est possible d'utiliser le mot clé `break` qui permet alors de quitter la boucle infinie.

Exemple

Le programme ci-dessous tourne en boucle :

- Lorsque aucun évènement n'est détecté, c'est l'image triste

- Image.SAD qui s'affiche.
- Pendant que le bouton A est pressé, l'image joyeuse Image.HAPPY s'affiche.
 - Pendant que la broche pin1 est touchée (pour cela, pincer en même temps que la broche GND avec la main droite et la broche pin1 avec la main gauche), l'image endormie Image.ASLEEP s'affiche.
 - Enfin, lorsque le bouton B est pressé, l'écran s'efface et le programme quitte la boucle.

```
from microbit import *
while True:
    if button_a.is_pressed():
        display.show(Image.HAPPY)
    elif pin1.is_touched():
        display.show(Image.ASLEEP)
    elif button_b.is_pressed():
        display.clear()
        break
    else:
        display.show(Image.SAD)
```

REMARQUE

Les broches pin0, pin1 et pin2 peuvent aussi servir de bouton. Ainsi la méthode `.is_touched()` permet de renvoyer une valeur booléenne qui vaut True lorsqu'une personne en contact avec la masse (broche GND) touche puis relâche la broche en question.

En effet quand l'instruction est exécutée, la carte mesure la résistance entre la broche à laquelle l'instruction s'applique et la masse (broche

GND). Si cette dernière a varié et est passée d'une valeur quasi infinie à une valeur faible, le test devient vrai (`True`). Cet événement arrive lorsqu'une personne en contact avec GND touche la broche puis la relâche.

4.2.6 Le hasard

La bibliothèque `random` est utilisable avec `Micro:bit`. Grâce à cette bibliothèque, il est très simple de générer des nombres aléatoires avec les fonctions :

- `random()` pour tirer un nombre décimal (`float`) compris entre 0 (inclus) et 1 (exclu)
- `randint(a,b)` pour tirer un nombre entier (`int`) appartenant à `a..b` (inclus).

Exemple

Le programme ci-dessous affiche très rapidement 50 nombres aléatoires tirés entre 1 et 6. Le dernier nombre tiré est affiché pendant une seconde puis effacé.

```
from microbit import *
from random import randint
for i in range(50):
    display.show(random.randint(1, 6))
    sleep(20)
sleep(1000)
display.clear()
```

4.2.7 – Le mouvement

L'accéléromètre de la carte Micro:bit est accessible par le module `accelerometer`.

Il est alors possible de récupérer une des coordonnées du vecteur accélération. Par exemple avec un appel à la fonction `.get_x()`.

Exemple

Le programme ci-dessous affiche une flèche en fonction de l'inclinaison de la carte Micro:bit.

Le bouton B permet de quitter la boucle infinie.

```
from microbit import *
button_b.was_pressed()
while True:
    capteur = accelerometer.get_x()
    if capteur > 40:
        display.show(Image.ARROW_E)
    elif capteur < -40:
        display.show(Image.ARROW_W)
    else:
        display.show("-")

    if button_b.was_pressed():
        display.clear()
        break
```

4.2.8 – Les gestes

Le module `accelerometer` peut aussi détecter des mouvements ou des positions pré-programmés : les *gestes* ('up', 'down', 'left', 'right', 'face up', 'face down', 'freefall', '3g', '6g', '8g', 'shake').

REMARQUE

L'utilisation des gestes est très lente.

Pour cela, il existe :

- `accelerometer.get_gestures()` qui renvoie un ensemble appelé `tuple` contenant l'historique des gestes. Le dernier élément du `tuple` est le geste le plus récent. Le `tuple` est réinitialisé à chaque appel de cette fonction.
- `accelerometer.is_gesture(nom_du_geste)` qui renvoie une **valeur booléenne** indiquant si le geste en cours est `nom_du_geste`
- `accelerometer.was_pressed(nom_du_geste)` qui renvoie une **valeur booléenne** indiquant si le geste `nom_du_geste` a été pressé depuis le dernier appel de cette fonction.

Exemple

Le programme ci-dessous affiche le nombre "8".

Lorsque la carte `Micro:bit` est secoué, il s'affiche une seconde plus tard de manière aléatoire et équiprobable soit "Oui", soit "Non".

Après une petite pause, le jeu recommence sauf si on appui sur le bouton B ce qui fait sortir de la boucle.

```
from microbit import *  
from random import choice
```

```
button_b.was_pressed()
while True:
    display.show("8")
    if accelerometer.was_gesture("shake"):
        display.clear()
        sleep(1000)
        display.scroll(random.choice(["Oui", "Non"]))
        sleep(250)
    if button_b.was_pressed():
        break
```

4.2.9 – La radio

Les cartes `Micro:bit` peuvent communiquer entre elles au moyen du module `radio`.

- `radio.send(texte)` permet d'envoyer par radio la chaîne de caractère `texte`.
- `radio.receive()` renvoie les données reçues par radio converties en une chaîne de caractère. Si rien n'a été reçu, la chaîne vaut `None`.

Exemple

Le programme ci dessous est à téléverser sur 2 cartes `Micro:bit`. Il contient à la fois une partie émetteur et une partie récepteur. Un appui sur les boutons A ou B de l'une ou l'autre des cartes `Micro:bit` affiche les texte "A" ou "B" sur l'autre. Un appui sur la broche `pin0` arrête le programme.

```
from microbit import *
button_a.was_pressed()
```

```
button_b.was_pressed()
while True:
    # émetteur
    if button_a.was_pressed():
        radio.send("A")
    if button_b.was_pressed():
        radio.send("B")
    # récepteur
    message = radio.receive()
    if message == "A":
        display.scroll("A")
    if message == "B":
        display.scroll("B")
    # pause pour éviter de saturer la carte
    sleep(20)
    # sortir de la boucle
    if pin0.is_touched():
        display.clear()
        break
```

4.2.10 – La boussole

Pour utiliser la boussole de la carte Micro:bit, il faut utiliser le module `compass`.

- `compass.get_x()` (ou `.get_y()` ou `.get_y()`) renvoient une des composantes du vecteur champ magnétique.
- `compass.heading()` renvoie un nombre entier (`int`) correspondant à l'angle en degré entre l'orientation de la carte Micro:bit et le nord magnétique.

REMARQUE

Avant d'utiliser une fonctionnalité du module `compass`, il faut obligatoirement *calibrer* la carte. Sans cela, les valeurs renvoyées sont fausses à cause du *bruit magnétique* présent dans l'environnement de la carte.

Au moment de calibrer la boussole, l'utilisateur doit bouger la carte dans différentes positions jusqu'à faire passer le point clignotant par toutes les LED de l'écran.

L'outil de calibration se lance automatiquement mais il est possible de programmer son exécution en appelant l'instruction `compass.calibrate()`.

Exemple

Le programme ci-dessous fait office de boussole et indique le nord magnétique.

Attention, le capteur est sensible aux objets tels que téléphones, ordinateurs ou aux lieux tels que ascenseurs ou salle informatique...

```
from microbit import *
compass.calibrate()
button_b.was_pressed()
while True:
    cap = ((15 - compass.heading()) // 30) % 12
    display.show(Image.ALL_CLOCKS[cap])
    if button_b.was_pressed():
        display.clear()
        break
```

4.2.11 – Nuages de points avec Mu-editor

Il est possible de tracer un **nuage de points** avec l'éditeur Mu-editor.

Pour cela, il faut

- utiliser l'outil Graphique de Mu-editor,
- faire afficher par le programme en cours d'exécution un tuple de nombres.

Exemple

Le programme ci-dessous affiche dans le terminal (ou dans le **Graphique** si l'outil de l'IDE est cliqué) deux nombres aléatoires appartenant à -100..100.

```
from microbit import *
from random import randint
drapeau = True
while True:
    sleep(50)
    if button_a.was_pressed():
        drapeau = not drapeau
    if drapeau:
        nb1 = randint(-100,100)
        nb2 = randint(-100,100)
        print( (nb1,nb2) )
```

(Notebook Capytale : 0fcb-77439)

5 – Boucle `for`

5.1 – Boucle bornée simple

5.1.1 – Répétition d'une instruction

Pour exécuter plusieurs fois la même instruction, il suffisait de recopier plusieurs fois la même instruction :

```
print("A")
print("A")
print("A")
```

Cette solution n'est pas raisonnable et elle n'est pas envisageable si on ne connaît pas le nombre de répétitions.

Python (et de nombreux langages de programmation) propose une instruction, appelée **la boucle** `for` permettant de gérer les **répétitions**.

Exemple

Forme la plus simple :

```
for _ in range(3): print("A")
```

Le nombre de tours de boucles peut dépendre d'une variable :

```
n = int(input())
for _ in range(2*n): print("A")
```


5.1.2 – Répétition d'un bloc d'instruction

La répétition n'est pas limitée à une instruction.

Exemple

```
[ ]: from random import randint

a = randint(1,6)
b = randint(1,6)
print("somme des deux dés : ",a+b)
a = randint(1,6)
b = randint(1,6)
print("somme des deux dés : ",a+b)
a = randint(1,6)
b = randint(1,6)
print("somme des deux dés : ",a+b)
a = randint(1,6)
b = randint(1,6)
print("somme des deux dés : ",a+b)
```

Pour cela les instructions formant le *corps de la boucle* doivent être regroupés en un **bloc** : une suite de lignes en retraits du même nombre d'espace.

Exemple

```
[ ]: from random import randint

for _ in range(4):
    a = randint(1,6)
    b = randint(1,6)
    print("somme des deux dés : ",a+b)
```

Une instruction supplémentaire qui n'est plus alignée avec le corps de boucle ne sera exécutée qu'une seule fois et après tous les tours de boucle :

Exemple

```
[ ]: from random import randint

print("Effectuons 4 expériences avec 2 dés :")
for _ in range(4):
    a = randint(1,6)
    b = randint(1,6)
    print("somme des deux dés : ",a+b)
print("Les 4 tirages sont effectués.")
```

5.2 – Utilisation de l'indice de boucle

Dans une boucle bornée, il est possible d'introduire une **nouvelle variable**

- accessible à l'intérieur du corps de boucle
- dont la valeur donne le numéro du tour de boucle
- appelée *indice de boucle* ou *compteur de boucle*

Exemple

```
[ ]: for i in range(10): print(i)
```

Le premier tour est 0, le deuxième 1, etc. Le numéro du dernier tour est donc égal à **un de moins que le nombre total de tours**.

5.3 – Utilisation des accumulateurs

On peut utiliser dans une boucle un *accumulateur* : une variable dont la valeur progresse à chaque tour de boucle.

```
[ ]: a = 1
for _ in range(4):
    a = a + 2
print(a)
```

Ce qui est équivalent au code suivant et affiche 9 après avoir donné à la variable *a* les valeurs 1, 3, 5, 7 et 9.

```
[ ]: a = 1
a = a + 2
a = a + 2
a = a + 2
a = a + 2
print(a)
```

Une boucle `for` permet de répéter une suite d'instructions regroupées en un **bloc**. **Le nombre de tours est prédéfini** et chacun des tours est associé à un **indice**.

5.4 Activités



ACTIVITÉ 1

(Capitale : 0fcb-77439) Écrire un programme qui demande un entier n à l'utilisateur, puis calcule et affiche le résultat de la multiplication

$$\underbrace{2 \times 2 \times 2 \times \dots \times 2}_{n \text{ fois}}$$



ACTIVITÉ 2

Écrire un programme qui calcule et affiche $1 \times 2 \times \dots \times 100$.



ACTIVITÉ 3

Écrire un programme qui demande un entier n à l'utilisateur puis calcule et affiche

1. $1 + 2 + \dots + n$
2. le nombre entier $n*(n+1)//2$

CORRECTION

```
[ ]: n = int(input("entrer un nombre"))

# Question 1
r = 0
for i in range(n):
    r = r + i + 1
print("réponse 1:", r)

# Question 2
print("réponse 2:", n*(n+1)//2)
```

**ACTIVITÉ 4**

(Capytale : 1983-77502) Écrire un programme qui demande à l'utilisateur

- une somme d'argent initiale s déposée sur un livret,
- un taux d'intérêt annuel t exprimé en pourcents
- un nombre d'année n

et qui affiche les intérêts perçus chaque année ainsi que le montant total présent sur le livret après n années.

(chaque année, il faut ajouter à s la quantité $s*t/100$)

CORRECTION

```
[ ]: s = 10_000 # 10000,00 mais écrit plus lisible
      t = 1.5  # 1,50 %
      n = 5

      for _ in range(n):
          interet = s*t/100
          s = s + interet

      print("somme finale :", s)
```

**ACTIVITÉ 5**

(Capytale : e7b7-77504)

1. Écrire un programme qui demande à l'utilisateur un nombre de chiffres n puis n chiffres, et qui calcule et affiche le nombre formé avec les n chiffres fournis dans l'ordre.
2. Écrire une variante du programme précédent dans lequel les chiffres sont donnés dans l'ordre inverse.

CORRECTION

L'idée du code suivant est d'écrire le prochain chiffre à droite de tous les précédents. Cela revient donc à multiplier par 10 le nombre obtenu jusque là et à ajouter le nouveau chiffre.

```
[ ]: n = int(input("nombre de chiffres"))
r = 0
for i in range(n):
    chiffre = int(input("Entrer le prochain chiffre : "))
    r = 10 * r + chiffre
print(r)
```

Dans le cas où les chiffres sont données en sens inverses, chaque chiffre à ajouter est multiplié par une puissance de 10 correspondant à sa position.

CORRECTION

```
[ ]: n = int(input("Entrer le nombre de chiffres : "))
r = 0
for i in range(n):
    chiffre = int(input("Entrer le prochain chiffre : "))
    r = r + chiffre * (10**i)
print(r)
```

**ACTIVITÉ 6**

(Capytale : a7bc-77505) Écrire un programme qui demande à l'utilisateur un nombre entier n et un nombre de chiffres k , et qui affiche successivement les k derniers chiffres de n , en commençant par les unités.

Si n contient moins de k chiffres, il suffira d'afficher des zéros à la fin.

(on rappelle que $n \% 10$ renvoie le chiffre des unités de n)

CORRECTION

```
[ ]: n = int(input("Entrer un nombre entier : "))
      k = int(input("Entrer un nombre de chiffres : "))

      for _ in range(k):
          print(n % 10)
          n = n // 10
```

**ACTIVITÉ 7**

(Capytale : 2294-77506) La suite de Fibonacci est la suite d'entiers (F_n) définie par $F_0 = 0$, $F_1 = 1$ et, pour tout entier n à partir de 2, $F_n = F_{n-1} + F_{n-2}$.

Écrire un programme qui demande à l'utilisateur un entier n qu'on supposera supérieur ou égal à 1, et qui affiche la valeur de F_n .

(on utilisera deux variables pour mémoriser F_n et F_{n-1} ainsi qu'une variable temporaire)

CORRECTION

```
[ ]: n = int(input("Entrer un nombre entier : "))

fib_m = 0    # valeur en n-1
fib_n = 1    # valeur en n

for _ in range(2, n+1):
    tmp = fib_m
    fib_m = fib_n
    fib_n = fib_m + tmp

print(fib_n)
```

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

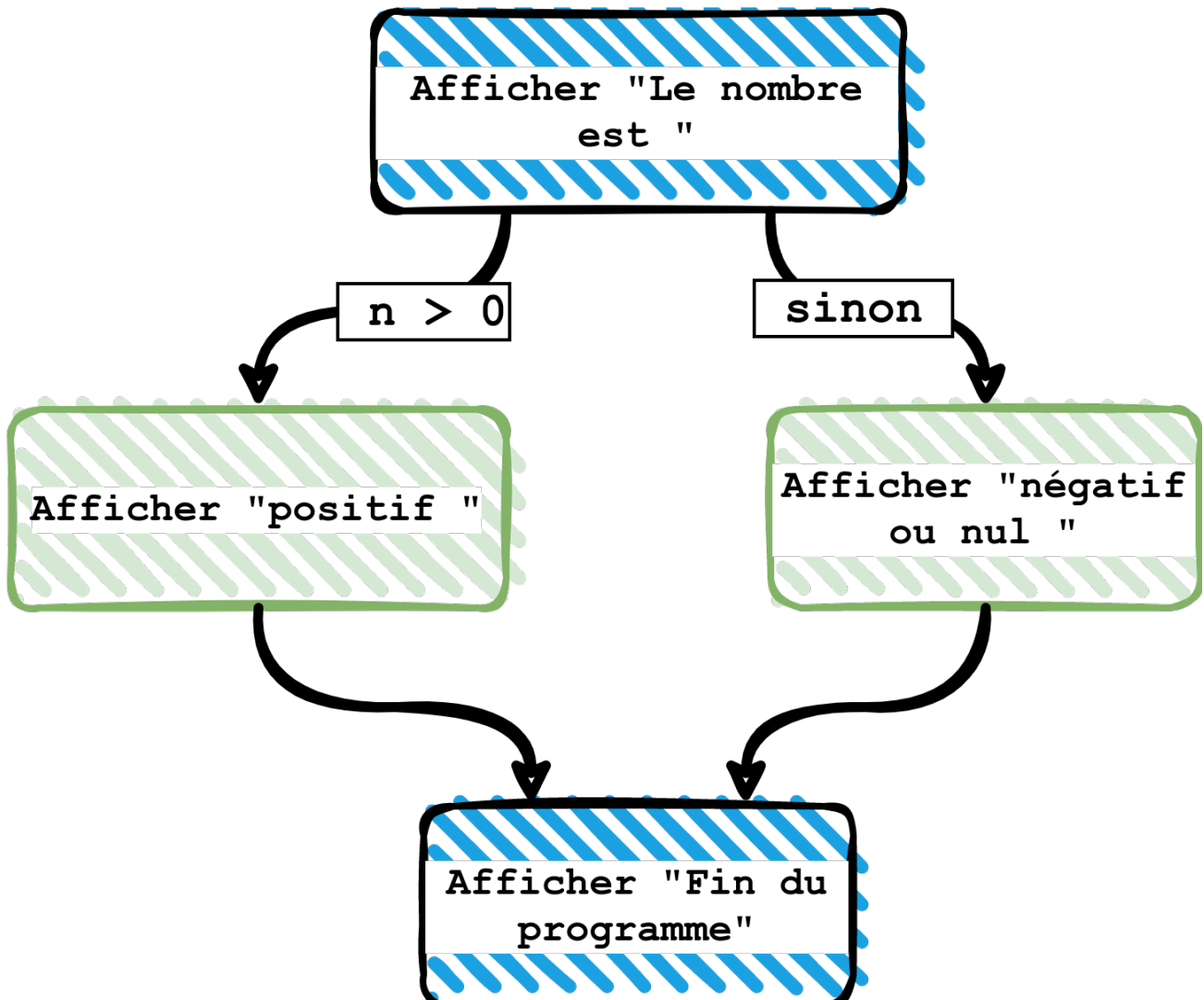
```
si n > 0 :  
    afficher "Positif"  
sinon :  
    afficher "Négatif ou nul"
```

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`.

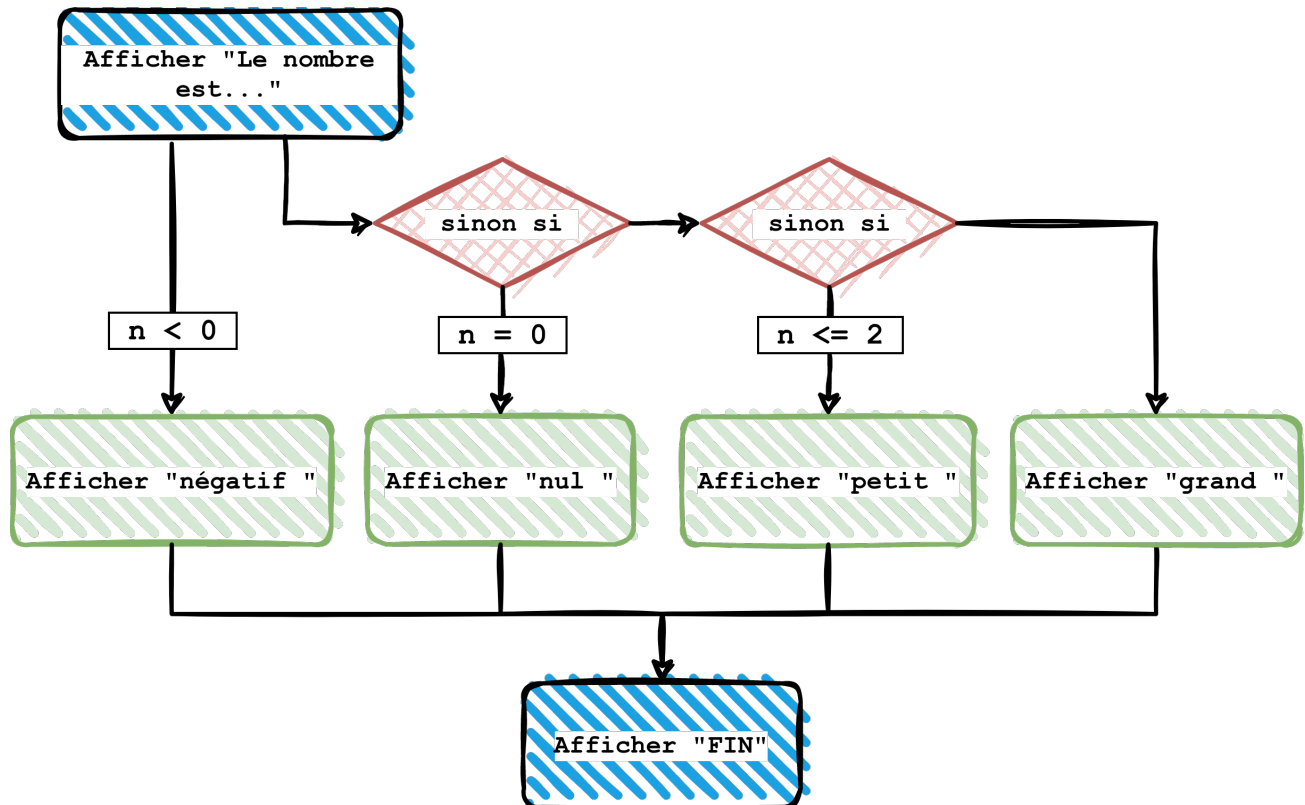
Ce mot-clé est suivi d'une condition terminée par le symbole `:`.

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.

6.4 – Activités



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".

CORRECTION

```
[ ]: score = int(input("Entrer le score : "))
gain = int(input("Entrer le gain : "))
nouveau_score = score + gain
if nouveau_score == 51:
    print("Victoire")
elif nouveau_score < 51:
    print("Nouveau score :", nouveau_score)
else:
    print("Nouveau score : 25")
```



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
```

CORRECTION

```
[ ]: score_1 = int(input("Score première boule : "))
score_2 = int(input("Score deuxième boule : "))
if score_1 == 10:
    print("X")
elif score_1 + score_2 == 10:
    print("/")
else:
    print(score_1 + score_2)
```

**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.

CORRECTION

```
[ ]: a = int(input("saisir un premier nombre: "))
      b = int(input("saisir un premier nombre: "))

      if a % b == 0:
          print("divisible")
      else:
          print("pas divisible")
```



ACTIVITÉ 4

Écrire un programme qui demande un entier "Entrer un nombre entier : " à l'utilisateur et affiche tous ses diviseurs.

Exemples :

```
>>> Entrer un nombre entier : 26
1
2
13
26
```

```
>>> Entrer un nombre entier : 10
1
2
5
10
```

CORRECTION

```
[ ]: n = int(input("Entrer un nombre entier positif : "))
      for i in range(1, n+1):
          if n % i == 0:
              print(i)
```

**ACTIVITÉ 5**

Écrire un programme qui demande un entier n à l'utilisateur et affiche tous ses diviseurs en précisant à la fin de l'affichage si le nombre est premier ou pas.

Exemples :

```
>>> Entrer un nombre entier positif : 26
1
2
13
26
26 n'est pas premier
```

```
>>> Entrer un nombre entier positif : 41
1
41
41 est premier
```

CORRECTION

```
[ ]: n = int(input("Entrer un nombre entier positif : "))
if n < 0:
    print(n, "n'est pas positif")
else:
    nb_diviseurs = 2
    print(1)
    for i in range(2, n//2 + 1):
        if n % i == 0:
            nb_diviseurs = nb_diviseurs + 1
            print(i)
    print(n)
    if nb_diviseurs == 2:
        print(n, "est premier")
    else:
        print(n, "n'est pas premier")
```

**ACTIVITÉ 6**

Au bowling, on a deux chances pour faire tomber un total de dix quilles. Écrire un programme 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.

Améliorer votre programme en ne demandant les informations de la deuxième boule que si elle a besoin d'être lancée.

Exemples :

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

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

>>> Score première boule : 10
X
```

CORRECTION

```
[ ]: score_1 = int(input("Score première boule : "))
if score_1 == 10:
    print("X")
else:
    score_2 = int(input("Score deuxième boule : "))
    if score_1 + score_2 == 10:
        print("/")
    else:
        print(score_1 + score_2)
```


**ACTIVITÉ 7**

Au bowling, on a deux chances pour faire tomber un total de dix quilles. Écrire un programme 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.

Améliorer votre programme en ne demandant les informations de la deuxième boule que si elle a besoin d'être lancée.

Améliorer votre programme en affichant ! si les scores saisis sont impossibles.

Exemples :

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

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

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

```
>>> Score première boule : -5
!
```

```
>>> Score première boule : 13
!
```

```
>>> Score première boule : 2
>>> Score deuxième boule : -3
!

>>> Score première boule : 2
>>> Score deuxième boule : 9
!
```

CORRECTION

```
[ ]: score_1 = int(input("Score première boule : "))
if score_1 < 0 or score_1 > 10:
    print("!")
elif score_1 == 10:
    print("X")
else:
    score_2 = int(input("Score deuxième boule : "))
    if score_2 < 0 or score_1 + score_2 > 10:
        print("!")
    elif score_1 + score_2 == 10:
        print("/")
    else:
        print(score_1 + score_2)
```



ACTIVITÉ 8

Écrire un programme qui demande trois longueurs à l'utilisateur, indique si ces trois longueurs peuvent être les longueurs des trois côtés d'un triangle et, le cas échéant, s'il s'agit d'un triangle équilatéral, isocèle, rectangle ou scalène (trois côtés de longueurs différentes)

Exemples :

```
>>> Entrer la première distance : 2
>>> Entrer la deuxième distance : 3
>>> Entrer la troisième distance : 10
Ceci n'est pas un triangle
```

```
>>> Entrer la première distance : 3
>>> Entrer la deuxième distance : 5
>>> Entrer la troisième distance : 6
Ceci est un triangle
scalène
```

```
>>> Entrer la première distance : 3
>>> Entrer la deuxième distance : 3
>>> Entrer la troisième distance : 5
Ceci est un triangle
isocèle
```

```
>>> Entrer la première distance : 5
>>> Entrer la deuxième distance : 5
>>> Entrer la troisième distance : 5
Ceci est un triangle
équilatéral
```

```
>>> Entrer la première distance : 3
>>> Entrer la deuxième distance : 4
>>> Entrer la troisième distance : 5
Ceci est un triangle
scalène
rectangle
```

CORRECTION

```
[ ]: a = int(input("Entrer la première distance : "))
b = int(input("Entrer la deuxième distance : "))
c = int(input("Entrer la troisième distance : "))
if a+b >= c and b+c >= a and c+a >= b:
    print("Ceci est un triangle")
    if a == b and b == c:
        print("équilatéral")
    elif a == b or b == c or c == a:
        print("isocèle")
    else:
        print("scalène")

    if a*a+b*b == c*c or b*b+c*c == a*a or c*c+a*a == b*b:
        print("rectangle")

else:
    print("Ceci n'est pas un triangle")
```



ACTIVITÉ 9

Pour limiter les erreurs dans la transmission d'un message, une technique simple appelée code de répétition d'ordre k consiste à répéter k fois chaque lettre.

Lors de la réception du message, il suffit alors de regarder quelle lettre est majoritaire dans chaque paquet de lettres.

Dans cet exercice, on ne considérera comme lettre que les nombres 0 et 1.

Écrire un programme qui demande à l'utilisateur de saisir deux lettres et qui affiche la lettre majoritaire, ou X si aucune n'est majoritaire.

Exemples :

```
>>> Première lettre : 0
>>> Deuxième lettre : 0
0
```

```
>>> Première lettre : 1
```

```
>>> Deuxième lettre : 0
X

>>> Première lettre : 0
>>> Deuxième lettre : 1
X

>>> Première lettre : 1
>>> Deuxième lettre : 1
1
```

CORRECTION

```
[ ]: l1 = input("Première lettre : ")
     l2 = input("Deuxième lettre : ")
     if l1 == l2:
         print(l1)
     else:
         print("X")
```



ACTIVITÉ 10

Pour limiter les erreurs dans la transmission d'un message, une technique simple appelée code de répétition d'ordre k consiste à répéter k fois chaque lettre.

Lors de la réception du message, il suffit alors de regarder quelle lettre est majoritaire dans chaque paquet de lettres.

Dans cet exercice, on ne considérera comme lettre que les nombres 0 et 1.

Écrire un programme qui demande à l'utilisateur de saisir trois lettres et qui affiche la lettre majoritaire.

Exemples :

```
>>> Première lettre : 0
>>> Deuxième lettre : 1
>>> Troisième lettre : 1
1

>>> Première lettre : 1
>>> Deuxième lettre : 0
>>> Troisième lettre : 0
0
```

CORRECTION

```
[ ]: 11 = input("Première lettre : ")
      12 = input("Deuxième lettre : ")
      13 = input("Troisième lettre : ")

      if 11 == 12 or 11 == 13:
          print(11)
      else:
          print(13)
```

**ACTIVITÉ 11**

Pour limiter les erreurs dans la transmission d'un message, une technique simple appelée code de répétition d'ordre k consiste à répéter k fois chaque lettre.

Lors de la réception du message, il suffit alors de regarder quelle lettre est majoritaire dans chaque paquet de lettres.

Dans cet exercice, on ne considérera comme lettre que les nombres 0 et 1.

Écrire un programme qui demande à l'utilisateur la valeur de l'ordre k, puis demande de saisir k lettres. Le programme affiche la lettre majoritaire, ou X si aucune n'est majoritaire.

Exemples :

```
>>> Entrer l'ordre du code : 5
>>> Saisir le lettre : 0
>>> Saisir le lettre : 1
>>> Saisir le lettre : 1
>>> Saisir le lettre : 1
>>> Saisir le lettre : 0
1
```

CORRECTION

```
[ ]: n = 0
k = int(input("Entrer l'ordre du code : "))
for _ in range(k):
    l = input("Saisir le lettre : ")
    if l == "1":
        n = n + 1
if n > k - n:
    print("1")
elif n < k - n:
    print("0")
else:
    print("X")
```