

NIVEAU 1

1-AFFICHAGE DE TEXTE, SUITE D'INSTRUCTIONS :



Afficher du texte

En Python, pour afficher du texte, on écrit `print` puis, entre parenthèses, le texte à afficher, entre guillemets. Ainsi, pour afficher le mot « Bonjour », on écrira le code suivant :

```
print("Bonjour")
```

En effet, `print` en anglais signifie « imprimer ».

Chaque ligne du programme représente une *instruction*.



Afficher plusieurs lignes de texte

Pour afficher plusieurs lignes de texte, il faut écrire des instructions d'affichage successives (une par ligne) :

```
print("Bonjour !")
print("Comment vas-tu ?")
```

↳ Bonjour !
Comment vas-tu ?

Quand on écrit un programme informatique, il faut être très rigoureux car un simple caractère mal placé peut perturber l'ordinateur du robot. Si le format de votre code n'est pas bon, il ne va pas aller plus loin et va simplement vous indiquer que vous avez commis une erreur.

Quand ça arrive, il faut d'abord **tenter de comprendre le message d'erreur** et ensuite **relire attentivement la zone de code concernée**, en particulier pour s'assurer qu'il ne manque pas de symbole tel qu'une parenthèse ou un guillemet.

Voici quelques exemples de messages d'erreur.

Attention aux parenthèses !

Si on oublie une parenthèse, cela produit une des erreurs suivantes :

```
print("Bonjour"
```

↳ `SyntaxError: unexpected EOF while parsing`

```
print "Bonjour")
```

↳ `SyntaxError: invalid syntax`

Attention aux guillemets !

Si on ne place pas correctement les guillemets, cela produit une des erreurs suivantes :

```
print("Bonjour)
```

↳ `SyntaxError: EOL while scanning string literal`

```
print(Bonjour)
```

↳ `NameError: name 'Bonjour' is not defined`

```
print(Bonjour tout le monde)
```

↳ `SyntaxError: invalid syntax`

Conclusion

Il faut être précis et ne rien oublier. Si l'on rencontre une erreur, on pensera à bien tout vérifier. Les messages décrivent très souvent la source exacte du problème, en anglais : il faut les lire et essayer de comprendre leur signification.

Un message d'erreur vous donnera aussi toujours le numéro de la ligne qui a causé le souci. Ce numéro n'est pas toujours exact : l'ordinateur peut n'être dérangé qu'après votre erreur. Il vous faudra donc souvent regarder avant l'emplacement indiqué.

2 – REPETITIONS D'INSTRUCTIONS – boucle « FOR » :



Répéter une action

Si pour vous il est rébarbatif de répéter une tâche de nombreuses fois, cela ne pose aucun problème à votre robot. Celui-ci est donc très pratique pour cela.

Ainsi, imaginons que l'on souhaite écrire 5 fois « Coucou ». On pourrait le faire avec le programme suivant :

```
print("Coucou")
print("Coucou")
print("Coucou")
print("Coucou")
print("Coucou")
```

C'est plutôt convenable ici... Mais si on veut effectuer par exemple 1000 affichages, cela va devenir bien plus fastidieux !

Pour plus d'efficacité, on aimerait indiquer directement que l'on souhaite répéter l'affichage en *boucle*, de la même manière qu'on pourrait écrire en français :

Répéter 5 fois
Afficher "Coucou"

On décale l'instruction à répéter vers la droite avec des espaces, pour indiquer qu'elle « appartient » à l'instruction précédente. Effectuer des décalages comme ceci est très courant en programmation : on appelle ça l'*indentation*.

Ainsi, en Python, pour répéter 5 fois l'instruction qui affiche « Coucou », on va écrire le programme ci-dessous.

```
for loop in range(5):
    print("Coucou")
```

↳ Coucou
Coucou
Coucou
Coucou
Coucou

Pour coder la répétition, nous avons utilisé la structure suivante :

```
for loop in range(5):
    ...
```

et nous avons mis l'instruction à répéter à la place des ..., après trois espaces. Cette indentation est **obligatoire**.

Vous pouvez donc écrire une boucle de cette façon, en indiquant le nombre de répétitions à la place du chiffre 5. Nous éluciderons par la suite les mystères de cette écriture. Prenez garde à ne pas oublier le deux-points « : » à la fin de la ligne.

ERREURS POSSIBLES :

Il est facile de se tromper dans les boucles lorsqu'on n'a pas l'habitude. Ainsi, si on oublie le « : » à la fin de la ligne, on obtient une erreur :

```
for loop in range(5)
    print("Bonjour !")
```

↳ SyntaxError: invalid syntax

Et si l'on oublie d'indenter, c'est-à-dire si on oublie les trois espaces, on obtient également une erreur :

```
for loop in range(5):
    print("Bonjour !")
```

↳ SyntaxError: expected an indented block

Face à ce type d'erreur, on pensera donc à vérifier que le deux-points est bien présent et que l'indentation a été faite.

ATTENTION : si l'indentation n'est pas toujours la même, vous obtiendrez une erreur :

```
for loop in range(2):
    print("Bonjour !")
    print("Comment vas-tu ?")
```

↳ SyntaxError: unindent does not match any outer indentation level

Pour afficher les nombres de 0 à 9, c'est-à-dire de 0 (inclus) à 10 (non inclus) vous utilisez actuellement le code suivant :

```
nombre = 0
for loop in range(10):
    print(nombre)
    nombre = nombre + 1
```

Eh bien, sachez qu'il est également possible de faire ainsi :

```
for loop in range(10):
    print(loop)
```

Ce que nous ne vous avions donc pas encore dit c'est que `loop` est en fait une variable ! On peut donc l'utiliser comme n'importe quelle variable, en particulier pour l'afficher. On peut également utiliser un autre nom de variable (`loop` n'étant pas très clair). Ainsi on écrira plutôt :

```
for nombre in range(10):
    print(nombre)
```

En pratique `range(10)` se comporte comme la suite de valeurs 0, 1, ..., 9 et ce code revient à dire :

```
Pour chaque valeur dans [0, 1, ..., 9] affecter cette valeur à la variable "nombre" et
Afficher la variable "nombre"
```

Dans la suite de ce cours nous allons voir la puissance de cette écriture qui permet d'avoir des programmes plus courts et compacts.

Cas général

Le fonctionnement de `range()` est en réalité un peu plus complexe que cela et sous sa forme générale, il s'utilise ainsi :

```
for nombre in range(<debut>, <fin>, <saut>):
    ...
```

ce qui va faire prendre à la variable `nombre` toutes les valeurs entre `<debut>` (inclus) et `<fin>` (non inclus) en faisant des sauts de `<saut>` à chaque fois. Pour bien comprendre, le mieux est de regarder quelques exemples :

Afficher les nombres de 0 à 9

```
for nombre in range(0, 10, 1):
    print(nombre, end = " ")
```

→ 0 1 2 3 4 5 6 7 8 9

On va donc de 0 (inclus) à 10 (non inclus, c'est-à-dire 9 inclus) en faisant "+1" entre chaque valeur.

Afficher les nombres de 10 à 20

```
for nombre in range(10, 21, 1):
    print(nombre)
```

→ 10 11 12 13 14 15 16 17 18 19 20

On va donc de 10 (inclus) à 21 (non inclus, c'est-à-dire 20 inclus) en faisant "+1" entre chaque valeur.

Afficher les nombres pairs de 100 à 120

```
for nombre in range(100, 121, 2):
    print(nombre)
```

→ 100 102 104 106 108 110 112 114 116 118 120

On va donc de 100 (inclus) à 121 (non inclus, c'est-à-dire 120 inclus) en faisant "+2" entre chaque valeur.

Afficher les multiples de 5 de 20 à -20

On veut donc afficher des nombres de manière décroissante, donc le "saut" est négatif

```
for nombre in range(20, -21, -5):
    print(nombre)
```

→ 20 15 10 5 0 -5 -10 -15 -20

On va donc de 20 (inclus) à -21 (non inclus, c'est-à-dire -20 inclus) en faisant "-5" entre chaque valeur.

Interprétation

On commence donc par la valeur de début et tant qu'on est strictement plus petit (si le saut est positif) / grand (si le saut est négatif) que la valeur de fin, on passe d'une valeur à la valeur suivante en faisant le "saut" indiqué.

Notez que toutes les variations suivantes sont donc équivalentes !

```
range(10, 101, 5)
range(10, 102, 5)
range(10, 103, 5)
range(10, 104, 5)
range(10, 105, 5)
```

Elle vont toutes afficher les multiples de 5 entre 10 et 100.

Formes réduites

Saut non précisé

S'il n'est pas précisé, le "saut" vaut 1, on peut donc écrire

```
for nombre in range(100, 201):
    ...
```

à la place de

```
for nombre in range(100, 201, 1):
    ...
```

Ces deux codes sont équivalents.

Début et saut non précisé

Si on a un "début" égal à 0 et un "saut" égal à 1, alors il existe une écriture plus courte. Ainsi :

```
for nombre in range(0, 10, 1):
    ...
```

peut être remplacé par

```
for nombre in range(10):
    ...
```

C'est la forme que vous connaissiez et que vous utilisez jusqu'à présent. Elle permettait bien de répéter 10 fois une suite d'instruction puisque `range(10)` contient les valeurs 0, 1, ..., 9 soit 10 valeurs au total.



Afficher du texte sans retour à la ligne

L'instruction ci-dessous :

```
print("Bonjour")
```

effectue en fait deux choses distinctes :

- premièrement, afficher le mot « Bonjour » ;
- deuvièmement, passer à la ligne suivante ou, autrement dit, effectuer un *retour à la ligne*.

Nous pourrions l'écrire comme ceci en pseudo-code :

```
Afficher "Bonjour" (sans retour à la ligne)  
Aller à la ligne
```

Bien qu'il soit généralement pratique d'afficher du texte et de terminer la ligne en une seule commande, on peut vouloir effectuer l'une des deux actions indépendamment de l'autre. Pour afficher le mot « Bonjour » sans revenir à la ligne, on utilise la commande suivante :

```
print("Bonjour", end = "")
```

Le texte qui suit le « Bonjour » est une option qui permet de dire que l'on ne veut rien ajouter après « Bonjour ».

Voyons ce qui se passe si on écrit un programme contenant deux fois cette instruction :

```
print("Bonjour", end = "")  
print("Bonjour", end = "")
```

→ BonjourBonjour

On a donc deux « Bonjour » collés.

À l'inverse, comment aller à la ligne sans rien afficher du tout ? Pour cela, il suffit simplement d'utiliser l'instruction `print` en lui disant d'afficher un texte vide :

```
print("")
```

En fait, on peut même ne pas fournir le texte vide. Voici un programme d'exemple le démontrant :

```
print("Un ", end = "")  
print("deux ", end = "")  
print("trois", end = "")  
print()  
print("Soleil !")
```

→ Un deux trois
Soleil !

INSERER DES COMMENTAIRES :

Les langages de programmation proposent une notation pour insérer des *commentaires* dans le code, c'est-à-dire du texte qui va être ignoré par l'ordinateur. Les commentaires peuvent servir à expliquer une partie compliquée d'un programme, ou à mettre des indications dans le code, comme son auteur ou sa date.

En Python, on insère un commentaire avec le caractère `#` (un dièse). Voici un exemple :

```
# Ce programme a été écrit par Hermione Granger le 10/01/1994  
# Quatrième année, cours d'étude des moldus  
  
# Affiche un rectangle rempli de X  
for loop in range(5):  
    for loop in range(10):  
        print("X", end = "") # pas de retour à la ligne ici  
    print()
```

Toutefois, la priorité est toujours d'écrire le code le plus clair possible. En effet, il est très souvent bien plus efficace de comprendre un programme directement à partir de ses instructions, qu'en alternant la lecture des instructions avec celle d'explications spécifiques. N'utilisez donc de commentaires que lorsque cela s'avère nécessaire, une fois que vous avez travaillé la clarté de votre code au maximum.

Remarque : Une variante consiste à commencer un commentaire avec trois guillemets `"""` et à le finir de la même façon. C'est une façon aisée d'écrire un long commentaire qui court sur plusieurs lignes :

```
"""  
Nous rappelons la formule du produit scalaire de deux  
vecteurs que nous utilisons ci-dessous :  
Soit U(x;y) et V(x';y'), U.V = x * x' + y * y'  
"""
```

En réalité, il ne s'agit pas d'un commentaire mais d'un texte du programme s'étendant sur plusieurs lignes. Sans instruction pour l'utiliser, il est cependant ignoré à l'exécution.

AFFICHAGE SIMPLIFIE :

Imaginez qu'on vous demande de lire 3 entiers puis de les afficher l'un après l'autre, séparés par des tirets. Un premier programme pourrait être

```
nombre1 = int(input())
nombre2 = int(input())
nombre3 = int(input())
print(nombre1, end = "")
print("-", end = "")
print(nombre2, end = "")
print("-", end = "")
print(nombre3)
```

↳ 12
45
22

↳ 12-45-22

Le programme ci-dessus n'est pas difficile, mais il est très répétitif et surtout pas très amusant à écrire. Voici comment on peut faire plus simplement :

```
nombre1 = int(input())
nombre2 = int(input())
nombre3 = int(input())
print("{}-{}-{}".format(nombre1, nombre2, nombre3))
```

C'est beaucoup mieux, non ? Regardons comme cela fonctionne.

Dans le texte "`"{}-{}-{}"`", on met un `{}` à chaque endroit où on souhaite placer un entier donc ici, trois entiers séparés par des tirets. Ensuite, on donne comme arguments à la fonction `format()` les trois entiers qu'on veut insérer à la place des `{}`.

Cela marche de la même manière si au lieu d'avoir des entiers vous avez d'autres objets : des nombres à virgules, du texte...

3 – CALCULS ET DECOUVERTES DES VARIABLES :



Faire des calculs

Pour additionner deux nombres, on va les joindre avec le symbole `+`, comme en maths ! Ainsi, nous pouvons afficher le résultat d'une somme :

```
Afficher 2 + 3
```

Ce qui s'écrit en Python :

```
print(2 + 3)
```

```
5
```

Sur le même principe, on pourra multiplier, soustraire ou diviser deux nombres. On utilise les symboles `-`, `*` et `/` pour cela :

```
print(30 * 10)
print(10 - 100)
print(42 / 10)
```

```
300
-90
4.2
```

Pour la division, observez que la virgule s'affiche sous la forme d'un point (conformément à la convention anglo-saxonne).

Notez que les espaces autour des opérateurs arithmétiques (`+`, `-`, `*`, `/`) ne sont pas obligatoires, mais ils rendent le programme plus facile à lire.

Faire des calculs complexes

Il est possible de faire plusieurs opérations d'un coup. Par exemple :

```
print(1 + 2 + 3)
```

```
6
```

En programmation, certains opérateurs ont la priorité sur d'autres, comme en mathématiques. Pour faire certains calculs, il peut être nécessaire d'utiliser des parenthèses :

```
print((12 - 3) * 4)
```

```
36
```

Sans les parenthèses, on aurait ainsi obtenu un résultat différent :

```
print(12 - 3 * 4)
```

```
0
```



Mémoriser des informations

Supposons que l'on souhaite écrire un programme qui affiche la distance qui sépare la Terre et Mars le 27 août 2003 (55758000km), puis la distance à parcourir par la lumière pour faire l'aller-retour depuis la Terre (le double). On pourrait utiliser les instructions suivantes :

```
Afficher 55758000
Afficher 2 * 55758000
```

Cela fonctionnera parfaitement bien. Toutefois, si l'on veut afficher ces informations pour un autre jour, alors que la distance entre les deux planètes a changé, il faudra modifier le programme à deux endroits différents, puisque l'on y a écrit deux fois la distance 55758000.

Pour éviter de devoir faire des modifications en double, on va utiliser une *variable*, appelée *distance*, pour représenter la valeur 55758000. On peut alors exprimer notre programme de cette façon :

```
distance <- 55758000
Afficher distance
Afficher 2 * distance
```

La première ligne signifie « *distance* reçoit 55758000 ». La flèche <- matérialise le mouvement du nombre 55758000 qui va se placer dans *distance*.

Ce programme est un peu plus long, mais offre deux avantages :

- si l'on veut modifier la distance, on n'aura qu'une seule valeur à modifier dans notre programme ;
- la valeur porte un **nom**, qui permet de comprendre à quoi elle correspond.

En Python, cela s'écrit comme suit :

```
distance = 55758000
print(distance)
print(2 * distance)
```

```
↳ 55758000
    111516000
```

Une variable est comme une boîte nommée qui contient quelque chose : on peut y stocker des informations et les retrouver par la suite. Dans notre exemple, la boîte porte le nom *distance* et contient la valeur 55758000.



La boîte *distance* est créée par l'instruction suivante :

```
distance = 55758000
```

qui place la valeur 55758000 dans la variable.

Dans ce chapitre, nous ne nous servirons des variables que pour travailler avec des nombres entiers. Nous manipulerons d'autres types de valeurs par la suite.

Les deux instructions d'affichage ont consulté le contenu de la boîte *distance* pour y lire la valeur 55758000 :

```
print(distance)
print(2 * distance)
```

Afficher des nombres avec des espaces

Pour afficher trois nombres sur la même ligne séparés par une espace, appuyez-vous sur le code suivant :

```
print(1, end = " ")
print(2, end = " ")
print(3)
```

```
↳ 1 2 3
```

On remplace donc la terminaison (qui est par défaut une fin de ligne) par une espace avec l'option `end = " "`.

NOM d'UNE VARIABLE :

Dans les langages de programmation, le choix du nom d'une variable est assez libre. Voici les règles générales.

- L'identifiant se constitue de caractères collés (pas d'espace).
- Les caractères autorisés sont essentiellement :
 - les lettres majuscules et minuscules naturelles : abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ;
 - les chiffres 0123456789 ;
 - le caractère « _ » (appelé « sous-tiret »).
- Le premier caractère du nom d'une variable ne peut pas être un chiffre ; le nom *1erNombre* est donc invalide.
- Les mots-clés du langage ne peuvent être utilisés pour nommer des variables. En Python, c'est par exemple le cas du mot `for`.

Dans le cas du langage Python, les accents sont acceptés dans les identifiants.

Vous devez profiter de cette flexibilité pour choisir des noms qui aideront à comprendre votre programme. Nous vous en reparlerons dans un prochain cours.



Modifier une variable

Comme son nom l'indique, une variable a vocation à **varier**, c'est-à-dire à représenter différentes valeurs. En programmation, il est notamment intéressant de changer la valeur que nous mettons dans nos boîtes. La flèche que nous employons en pseudo-code :

```
variable <- valeur
```

signifie que *valeur* devient le nouveau contenu de *variable*. On dit qu'on *affecte* la valeur à la variable.

Une même variable peut se voir affecter différentes valeurs au cours du programme. C'est comme si on prenait une vraie boîte et qu'on y mettait des objets, un seul restant à chaque fois :

```
boîte <- chaussure  
boîte <- poupée
```

Ainsi, considérons un programme qui se sert d'une variable nommée *contenance* pour représenter la contenance d'une bouteille d'eau d'un litre. Cette bouteille contient initialement 60cl, puis on la remplit en entier (jusqu'à 100cl). Cela donne :

```
contenance <- 60  
Afficher "Départ : ", contenance  
contenance <- 100  
Afficher "Remplissage : ", contenance
```

Maintenant, supposons que l'on boive 15cl d'eau. Comment calculer la nouvelle contenance ? Il s'agit de la soustraction « *contenance* - 15 ». On l'affecte donc à la variable :

```
contenance <- contenance - 15  
Afficher "Consommation : ", contenance
```

En Python, c'est l'opérateur `=` qui symbolise l'affectation, plutôt que la flèche. Voici donc le programme final :

```
contenance = 60  
print("Départ :", contenance)  
contenance = 100  
print("Remplissage :", contenance)  
contenance = contenance - 15  
print("Consommation :", contenance)
```

```
Départ : 60  
Remplissage : 100  
Consommation : 85
```

L'utilisation du symbole d'égalité peut vous paraître particulière. En effet, en mathématiques, l'égalité énonce un fait. Ainsi, dans un contexte donné, $x = y - z$ signifie qu'il est vrai que x a la même valeur que $y - z$.

L'affectation `contenance = contenance - 15` en revanche, décrit une action : **l'enregistrement de la valeur placée à droite du = dans la variable indiquée à gauche**. Le contenu de la variable *contenance* changera donc la prochaine fois que l'on affectera une valeur à la variable.

Observez aussi les instructions d'affichage :

```
print("Départ :", contenance)
```

Avec une seule instruction `print`, on peut afficher plusieurs données sur une ligne en les séparant par des virgules. Il y a alors une espace entre chaque donnée et la suivante (on n'a donc pas mis d'espace à la fin du texte).

L'exécution du programme est détaillée ci-dessous, la ligne rouge indiquant à quel endroit on est arrivé.

Début du programme. La variable *contenance* n'existe pas encore.

```
contenance = 60  
print("Départ :", contenance)  
contenance = 100  
print("Remplissage :", contenance)  
contenance = contenance - 15  
print("Consommation :", contenance)
```

Exécution de l'initialisation. La variable *contenance* est créée, initialisée avec la valeur 60.

```
contenance = 60  
print("Départ :", contenance)  
contenance = 100  
print("Remplissage :", contenance)  
contenance = contenance - 15  
print("Consommation :", contenance)
```

contenance

60

On affiche un message avec le contenu actuel de la variable *contenance*.

```
contenance = 60  
print("Départ :", contenance)  
contenance = 100  
print("Remplissage :", contenance)  
contenance = contenance - 15  
print("Consommation :", contenance)
```

contenance

60

→ Départ : 60

On affecte la valeur 100 à la variable *contenance*.

```
contenance = 60
print("Départ :", contenance)
contenance = 100
print("Remplissage :", contenance)
contenance = contenance - 15
print("Consommation :", contenance)
```

contenance

60 100

→ Départ : 60

On affiche la nouvelle valeur de la variable.

```
contenance = 60
print("Départ :", contenance)
contenance = 100
print("Remplissage :", contenance)
contenance = contenance - 15
print("Consommation :", contenance)
```

contenance

100

→ Départ : 60
Remplissage : 100

On récupère la contenance actuelle, on y retire 15, et on retient le résultat (85) dans la variable.

```
contenance = 60
print("Départ :", contenance)
contenance = 100
print("Remplissage :", contenance)
contenance = contenance - 15
print("Consommation :", contenance)
```

contenance

100 85

→ Départ : 60
Remplissage : 100

On affiche un dernier message.

```
contenance = 60
print("Départ :", contenance)
contenance = 100
print("Remplissage :", contenance)
contenance = contenance - 15
print("Consommation :", contenance)
```

contenance

85

→ Départ : 60
Remplissage : 100
Consommation : 85

Plusieurs variables

Précédemment, nous n'avons utilisé qu'une seule variable, mais les programmes en utilisent généralement plusieurs. Le programme suivant illustre cela : il utilise deux variables nommées *largeur* et *longueur* afin de calculer l'aire en mm² d'une feuille A4 (21cm × 29,7cm), et il enregistre le résultat dans une variable nommée *surface*. Le contenu de cette dernière variable est ensuite affiché.

```
largeur = 210
longueur = 297
surface = longueur * largeur

print(surface)
```

→ 62370

À la fin du programme, nous avons donc les boîtes suivantes :

largeur	longueur	surface
210	297	62370

Variable inexisteante

Si on fait appel à une variable qui n'existe pas (ou pas encore), on obtient une erreur. Par exemple, le programme suivant définit une variable *longueur*, et tente ensuite d'afficher le contenu d'une variable nommée *largeur* qui n'a jamais été définie.

```
longueur = 297
print(largeur)
```

→ NameError: name 'largeur' is not defined

Il faut faire particulièrement attention au fait que les minuscules et majuscules ne sont pas considérées comme équivalentes. Ainsi, la variable nommée *longueurFil* n'a strictement rien à voir avec la variable nommée *longueurfil*.

```
longueurFil = 10
print(longueurfil)
```

→ NameError: name 'longueurfil' is not defined

Si on rencontre une erreur de la forme `NameError: name 'xxxxx' is not defined`, on pensera donc à vérifier que l'on a bien initialisé la variable correspondante et que l'on a correctement saisi l'identifiant aux deux endroits.

Faute de frappe à l'affectation

Observez le programme suivant, dans lequel on a commis une faute de frappe sur la troisième ligne :

```
taille = 180
print(taille)
tauelle = taille + 20
print(taille)
```

Au lieu d'afficher 180 puis 200 comme on voudrait, ce programme affiche deux fois de suite le nombre 180 :

```
→ 180
180
```

car la valeur 200 a en fait été affectée à une autre variable, nommée *tauelle*.

METTRE À PROFIT LES IDENTIFIANTS :

Les identifiants comme les noms de variable sont un très bon support pour concrétiser un programme : ils permettent de placer directement dans les instructions des mots au choix du programmeur, avec lesquels il peut décrire ce qu'il souhaite faire. Nous présentons ci-dessous des techniques à prendre en main, qui seront très appréciées par les membres du forum, et qui vous aideront vous-même, à relire, analyser et corriger vos programmes.

Choisir un bon nom

Lorsqu'on a besoin d'une nouvelle variable, on cherchera toujours à **trouver un nom pour cette variable qui décrive le mieux possible ce qu'elle représente**. Il ne faut pas hésiter à mettre à la suite plusieurs mots pour construire un nom précis. Nous vous conseillons de nommer vos variables par une suite de mots ou abréviations accolés les uns aux autres, en mettant en majuscule la première lettre de chaque mot sauf le premier. Voici quelques exemples :

```
longueurFeuille
nbPiècesJaunes
maxHauteurPiquets
```

Ce style de nommage est très répandu : on l'appelle couramment [CamelCase](#), ou « casse ChatMot » en français, en référence aux bosses du chameau.

On évitera en particulier l'utilisation de noms réduits à une simple lettre, comme *i* ou *a*. Les lignes suivantes montrent un exemple de très mauvais code :

```
a = 1000
b = 50
c = a + b
```

En effet, on comprend beaucoup mieux ce qui se passe lorsque des noms plus parlants sont employés :

```
prixFour = 1000
prixBatteur = 50
prixTotal = prixFour + prixBatteur
```

Le nom doit placer la variable dans son contexte afin d'être efficace. Ci-dessus, lors d'achats culinaires, nous calculons le coût total à partir du prix de nos deux produits.

Nommer une valeur

Nous avons vu au départ qu'une variable permettait d'éviter de recopier une valeur. On peut également créer une variable, simplement dans le but de décrire une valeur ; c'est notamment pratique pour diviser un calcul compliqué en plusieurs parties identifiables.

Imaginons par exemple un marchand de fruits et légumes, qui vend les produits suivants :

Nom	Quantité (kg)	Prix (centimes/kg)
Pommes	32	90
Pêches	12	96
Aubergines	23	120
Carottes	17	102
Patates	22	85

On souhaite calculer la valeur totale de ses produits. On pourrait écrire le programme comme ceci :

```
Afficher 32 * 90 + 12 * 96 + 23 * 120 + 17 * 102 + 22 * 85
```

Toutefois, si le marchand décide de changer un prix ou nous apprend qu'il s'est trompé dans une quantité, il n'est pas forcément facile de déterminer où effectuer la correction. Mais si on introduit des variables :

```
prixPommes <- 32 * 90
prixPêches <- 12 * 96
prixAubergines <- 23 * 120
prixCarottes <- 17 * 102
prixPatates <- 22 * 85
Afficher prixPommes + prixPêches + prixAubergines + prixCarottes + prixPatates
```

Il devient bien plus aisés de se repérer dans le programme.

4 – LECTURE DE L’ENTREE :



Lire des entiers

On dispose de paquets de friandises dans lesquels les bonbons sont répartis dans des sachets de même contenance. On souhaite écrire un programme calculant le nombre total de bonbons dans un paquet, à partir du nombre de sachets dans le paquet et du nombre de bonbons par sachet. En Python, on fera ainsi :

```
nbSachets = int(input())
contenanceSachets = int(input())

print(nbSachets * contenanceSachets)
```

La ligne :

```
nbSachets = int(input())
```

récupère le premier entier de l'entrée (qui correspond au nombre de sachets) et le stocke dans la variable `nbSachets`. Il en est de même avec la ligne suivante pour le nombre de biscuits par sachet.

Comme `int` signifie « entier » et `input` signifie « entrée », `int(input())` permet de prendre un entier sur l'entrée.

Erreur si l'on ne donne pas un entier :

Imaginons que le programme demande un entier mais que l'utilisateur fournit un texte, par exemple « coucou ». Le programme déclare alors une erreur, car « coucou » ne peut pas être interprété comme un nombre.

```
taille = int(input())
print(taille)

↳ coucou

↳ ValueError: invalid literal for int() with base 10: 'coucou'
```

Notez qu'une erreur similaire peut se produire si vous ajoutez des lignes vides avant un nombre.

Lectures d'entiers : autre erreur possible :

Si on oublie le `int(...)` et que l'on écrit juste `input()` à la place de `int(input())`, on peut avoir de grosses surprises, comme le montre l'exemple suivant.

```
valeur1 = input()
valeur2 = input()
print(valeur1 + valeur2)

↳ 11
22

↳ 1122
```

Si on oublie le `int(...)` autour du `input()`, les valeurs ne sont pas traitées comme des entiers mais comme du texte. Le symbole `+` agit alors comme un opérateur qui **concatène** (c'est-à-dire qui met bout à bout) deux textes, et du coup on obtient 11 collé à 22 (c'est-à-dire 1122) à la place de 11 additionné à 22 (c'est-à-dire 33).

Faites donc toujours attention à ne pas utiliser `input()` tout seul. De plus, si les résultats des calculs sont manifestement faux, pensez à vérifier si les nombres ne sont pas traités comme du texte.

INSTRUCTIONS CONDENSEES :

Jusqu'à présent, chaque fois qu'on voulait afficher une chaîne de caractères, une variable ou une autre valeur, on utilisait une instruction `print`. Prenons par exemple ce programme qui lit la longueur et la largeur d'un rectangle puis calcule son périmètre et son aire et les affiche séparés par une espace :

```
longueur = int(input())
largeur = int(input())
périmètre = (longueur + largeur) * 2
aire = longueur * largeur
print(périmètre, end = " ")
print(aire)
```

```
↳ 12
7

↳ 38 84
```

En fait, avec un seul `print`, il est possible d'afficher autant de valeurs que l'on veut ! Il suffit pour cela de les séparer par une virgule entre les parenthèses. On pourrait en effet écrire le programme précédent ainsi :

```
longueur = int(input())
largeur = int(input())
périmètre = (longueur + largeur) * 2
aire = longueur * largeur
print(périmètre, aire)
```

Mais où est passée l'espace ? Les différentes valeurs sont automatiquement séparées par une espace à l'affichage. Cela permet d'écrire facilement de jolies phrases :

```
âge = 20
tempsProgrammation = 8
print("J'ai", âge, "ans et je programme depuis", tempsProgrammation, "ans")
```

sans risque d'oublier l'espace au bout des chaînes de caractères.

En fait, tout comme le retour à la ligne à la fin, cette espace séparatrice peut être affectée par une option : `sep`. Juste avant de fermer la parenthèse d'une instruction `print`, on peut indiquer les valeurs de `sep` et de `end`. Par exemple :

```
print(1, 2, 3, 4, 5, sep = ", ", end = " | ")
print(6, 7, 8, 9, 10, sep = ";" )
```

↳ 1,2,3,4,5 | 6;7;8;9;10

Toutefois, dans certains cas, il peut être difficile de se repérer dans une instruction `print` très longue. N'hésitez donc pas à en utiliser plusieurs à la suite si cela rend la lecture plus agréable !

De manière générale, le langage de programmation permet de formuler la même chose de différentes façons. Certaines écritures sont cependant plus agréables pour un humain : c'est à **vous** de les trouver ; en aucun cas l'ordinateur ne vous y aidera !

Lorsqu'une écriture vous est inhabituelle, prenez un peu de temps pour chercher comment l'écrire de sorte qu'elle soit la plus « propre » (jolie, optimale, demandant moins d'effort de lecture à une personne quelconque) possible, afin de vous en servir automatiquement la prochaine fois. Les codes que nous présentons dans les cours et les corrections sont généralement de bonnes références (sachant que nous n'utilisons que les notations que nous avons vues jusqu'alors).

PORTEE D'UNE VARIABLE :

Observez le programme suivant :

```
nbValeurs = int(input())
for iValeur in range(nbValeurs):
    laValeur = int(input())
print(laValeur)
```

Il demande un nombre `nbValeurs`, puis chacune des valeurs. À la fin, il affiche le dernier entier qui a été saisi. Voici un exemple d'exécution :

↳ 2
10
25
↳ 25

Ainsi, la variable `laValeur`, qui reçoit des valeurs au sein de la boucle, peut être réutilisée après la boucle. Cela peut s'avérer dangereux : en effet, que se passe-t-il dans le programme si 0 est saisi pour `nbValeurs` ? Dans ce cas, `laValeur` n'est jamais initialisée, et on obtient l'erreur suivante :

↳ Traceback (most recent call last):
 File "./run/exe", line 4, in <module>
 print(laValeur)
NameError: name 'laValeur' is not defined

On appelle *portée* d'une variable l'ensemble des endroits du programme où elle existe. En Python, la portée d'une variable s'étend donc dans tout le programme dès qu'elle reçoit sa première valeur. Toutefois, nous verrons plus tard des cas où la portée d'une variable est limitée à un bloc, par exemple dans une fonction.

5 – TESTS ET CONDITIONS :



Conditionner une action

La célèbre attraction du train fou est interdite aux moins de 10 ans. On souhaite écrire un programme qui demande à l'utilisateur son âge et qui, **si** la personne a moins de 10 ans, affiche le texte « Accès interdit » ; ce qui peut se rédiger comme cela :

```
âge <- lireEntier()
Si âge < 10
    Afficher "Accès interdit"
```

Voyons comment cela se traduit en Python :

```
âge = int(input())
if âge < 10:
    print("Accès interdit")
```

On écrit donc le mot-clé **if**, la traduction en anglais de « si », la condition à tester, à savoir `âge < 10`, puis on termine la ligne avec un deux-points, comme on le faisait pour la boucle de répétition.

Ainsi, l'accès est interdit à un enfant de 8 ans :

```
↳ 8
↳ Accès interdit
```

À l'opposé, le programme n'affiche rien pour un âge de 13 ans :

```
↳ 13
↳
```

Pour exprimer la condition du « si » dans le programme, on a utilisé le symbole `<`, qui est l'**opérateur de comparaison strictement inférieur**. De manière symétrique, l'opérateur `>` permet de tester si un nombre est **strictement supérieur** à un autre.

Lorsqu'on veut tester si un nombre est **inférieur ou égal** à un autre, on utilise le symbole `<=`. De manière symétrique, le symbole `>=` permet de tester si un nombre est **supérieur ou égal** à un autre.

Par exemple, le code suivant permet de tester si la température de l'eau a atteint 100 degrés.

```
température = int(input())
if température >= 100:
    print("L'eau bout !")
```



Effectuer une action dans le cas opposé

Il arrive que l'on ait besoin de tester une condition puis son contraire. Par exemple, supposons que l'on souhaite tester si l'on peut ranger tous ses œufs dans une seule boîte de 12 compartiments. On pourrait écrire :

```
nbOeufs <- lireEntier()
Si nbOeufs <= 12
    Afficher "Une boîte suffit"
Sinon
    Afficher "Plusieurs boîtes nécessaires"
```

Ce programme traduit donc la phrase suivante : si le nombre d'œufs est plus petit que 12, alors afficher « Une boîte suffit », et si c'est strictement supérieur à 12 alors afficher « Plusieurs boîtes nécessaires ».

Cependant, cette phrase s'exprime plus simplement sous la forme suivante : si le nombre d'œufs est plus petit que 12 alors afficher « Une boîte suffit », et sinon afficher « Plusieurs boîtes nécessaires » :

```
nbOeufs <- lireEntier()
Si nbOeufs <= 12
    Afficher "Une boîte suffit"
Sinon
    Afficher "Plusieurs boîtes nécessaires"
```

Il est possible d'exprimer en Python la notion de **sinon**, ce qui évite de tester le contraire de la condition que l'on vient de tester. Ainsi, le code précédent peut s'écrire de manière beaucoup plus élégante :

```
nbOeufs = int(input())
if nbOeufs <= 12:
    print("Une boîte suffit")
else:
    print("Plusieurs boîtes nécessaires")
```

On utilise donc le mot-clé **else**, sans condition. Pensez bien toujours au deux-points : !

Il est important de toujours utiliser un « sinon » lorsque c'est possible. Cela permet non seulement d'éviter d'introduire des informations redondantes, comme par exemple `nbOeufs <= 12` et `nbOeufs > 12`, mais aussi de rendre la logique du programme plus facile à comprendre.

CONDITIONS ERREUR POSSIBLE :

Si l'on place un `else` sans `if` qui le précède, on obtient une erreur :

```
else:  
    print("Oh non")
```

↳ `SyntaxError: invalid syntax`

Si vous obtenez une telle erreur, vérifiez donc que chacun de vos `else` est bien connecté à un `if` qui le précède.

BLOCS CONDITIONNELS FORMES DE PLUSIEURS INSTRUCTIONS :

On peut bien sûr placer plusieurs instructions à la suite d'un test. L'exemple suivant permet d'afficher deux messages à la suite dans le cas où la température de l'eau est supérieure ou égale à 100 degrés.

```
Si température >= 100  
    Afficher "L'eau bout !"  
    Afficher "Préparons le thé"
```

En Python :

```
if température >= 100:  
    print("L'eau bout !")  
    print("Préparons le thé")
```

Comme avec les boucles, il est important d'indenter toutes les instructions de la condition. Si l'on n'indente pas la seconde instruction, c'est-à-dire si l'on écrit :

```
temperature = int(input())  
if temperature >= 100:  
    print("L'eau bout !")  
print("Préparons le thé")
```

alors la phrase « Préparons le thé » va s'afficher quelle que soit la température de l'eau, ce qui n'est pas ce que l'on souhaite ici.

Notez que, de la même manière, il est possible de placer plusieurs instructions dans un bloc `else`.



Tester l'égalité et la différence

Vous avez vu comment tester des inégalités strictes, à l'aide des opérateurs `<` et `>`, ainsi que des inégalités larges, à l'aide des opérateurs `<=` et `>=`. Voyons maintenant comment tester l'égalité de deux valeurs, ou leur non-égalité.

Par exemple, pour tester si Marie et Robin ont le même âge, on utilise l'opérateur `==`, comme illustré ci-dessous.

```
ageMarie = int(input())  
ageRobin = int(input())  
  
if ageMarie == âgeRobin:  
    print("Marie et Robin ont le même âge")  
else:  
    print("Marie et Robin n'ont pas le même âge")
```

Il faut faire bien attention à ne pas confondre l'opérateur `==` avec le simple `=`, car les deux ont des rôles très différents :

- `=` sert à affecter une valeur à une variable ;
- `==` sert à tester l'égalité de deux valeurs.

Lorsqu'on veut uniquement tester si deux valeurs sont différentes, on utilise l'opérateur `!=`, qui se lit « *différent de* ». Par exemple, le code suivant affiche un message si un animal n'a aucune chance d'être une araignée car il n'a pas 8 pattes.

```
nbPattes = int(input())  
if nbPattes != 8:  
    print("L'animal n'est pas une araignée")
```



Faire des tests : le « sinon si »

Un grand magasin propose une offre spéciale : **si** on achète pour plus de 300 euros on a une remise de 40 euros, **sinon si** on achète pour plus de 200 euros on a une remise de 25 euros, **sinon si** on achète pour plus de 100 euros on a une remise de 10 euros **sinon** on a aucune remise.

Si on traduit ce programme en Python cela donne :

```
prix = int(input())
if prix >= 300:
    prix = prix - 40
else:
    if prix >= 200:
        prix = prix - 25
    else:
        if prix >= 100:
            prix = prix - 10
print(prix)
```

On a cependant beaucoup de `if/else` imbriqués et le programme est très indenté, imaginez si on avait 10 conditions différentes !

Mais, dans la phrase du début on a beaucoup utilisé le terme "**sinon si**" et il existe une structure en Python qui correspond exactement à cela, il s'agit de la construction `elif` qui s'utilise ainsi :

```
prix = int(input())
if prix >= 300:
    prix = prix - 40
elif prix >= 200:
    prix = prix - 25
elif prix >= 100:
    prix = prix - 10
print(prix)
```

On utilisera donc le `elif` quand il y a beaucoup de cas différents à tester.

7 – REPETITIONS CONDITIONNEES :



Faire la même chose plusieurs fois : le « tant que »

On a parfois besoin de répéter certaines instructions jusqu'à ce qu'un certain changement ce soit produit. Par exemple, demander un mot de passe *tant que* l'utilisateur n'a pas donné le bon.

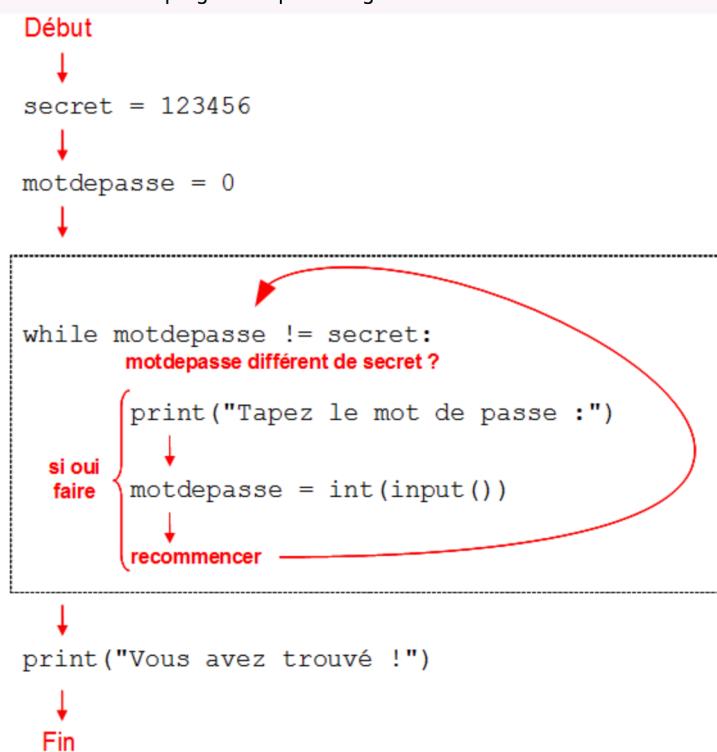
On a ici utilisé dans la phrase le terme « *tant que* », ce qui signifie qu'on a bien une condition pour savoir quand s'arrêter. On ne peut pas utiliser notre boucle « *répéter* » habituelle, car on ne sait pas combien de fois l'utilisateur va se tromper !

On va donc faire intervenir une autre boucle : la boucle « *tant que* », que nous allons manipuler dans ce chapitre. Elle se nomme `while` dans les langages de programmation (traduction en anglais de « *tant que* »).

```
secret = 123456
motDePasse = 0
while motDePasse != secret:
    print("Tapez le mot de passe :")
    motDePasse = int(input())
print("Vous avez trouvé !")
```

Ainsi, tant que la condition `motDePasse ≠ secret` est vraie, on continue à demander un nouveau mot de passe.

On peut représenter l'exécution du programme par le diagramme suivant :



Il est bien sûr possible d'utiliser des opérateurs booléens pour combiner des conditions et les valeurs booléennes sont également utilisables. Voici quelques extraits de code à titre d'exemple :

```
while True:  
    print("J'attends")
```

```
while motDePasse != secret or âgePersonne <= 3:  
    print("Accès refusé : mauvais mot de passe ou personne trop jeune")  
    âgePersonne = int(input())  
    motDePasse = int(input())
```

```
while nbPersonnes <= nbMax and température <= 45:  
    print("Portes ouvertes")  
    nbPersonnes = nbPersonnes + 1  
    température = int(input())
```

Le premier de ces trois exemples est ce qu'on appelle une « boucle infinie », c'est-à-dire que le programme ne s'arrête jamais : comme `True` est toujours vrai, on ne quitte jamais la boucle.

BOUCLE INFINIE :

Dans un code utilisant une boucle « tant que », si on utilise une mauvaise condition, il est possible que le programme ne s'arrête jamais. Par exemple :

```
valeur <- 1  
Tant que valeur <= 10 faire  
    valeur <- valeur - 1
```

Ici, on a mis `valeur - 1` au lieu de `valeur + 1` ; du coup, la variable `valeur` ne fait que diminuer au lieu d'augmenter. Elle ne sera donc jamais plus grande que 10 et le programme ne s'arrêtera jamais (en supposant les limites des entiers infinies) !

Si vous écrivez un tel programme, le système d'évaluation automatique du site vous indiquera que « votre programme a dépassé la limite de temps ». En effet, pour éviter les programmes qui ne s'arrêtent jamais, nous avons un système qui les bloque automatiquement s'ils mettent trop de temps à donner leur réponse.

Si vous voyez un tel message, essayez de vérifier les conditions dans vos boucles.