

INSTRUCTIONS DE CONNEXION

1. Notion de variable

1.1 Qu'est-ce qu'une variable ?

Définition. Une **variable** désigne une zone mémoire de l'ordinateur dans la RAM.

Il s'agit d'un endroit où l'on peut **stocker** une valeur, y **accéder** et **changer** cette valeur.

Pour faire référence à une variable, on utilise un nom de variable, en général composé d'une ou plusieurs lettres. On choisit un nom explicite, ce qui permet une meilleure lecture du programme.

1.2 Affectation

Définition.

- Quand on stocke une valeur **d** dans une variable **var**, on dit qu'on **affecte** **d** à la variable **var**. La valeur **d** est encore appelée **la donnée**.

En écriture algorithmique : $\text{var} \leftarrow d$

En Python, cette affectation est faite avec la commande `=`.

```
| var = d
```

- Une donnée est typée, autrement dit classée, selon l'objet qu'elle représente. Il y a des types simples comme les entiers ou les flottants (nombres à virgule) ou des types plus complexes comme les chaînes de caractères, les listes.

Remarque. Les variables (et donc l'affectation) sont incontournables : si vous ne stockez pas une donnée (résultat par exemple d'un calcul), vous ne pourrez pas la réutiliser, ni la modifier.

Exemple. On tape dans l'éditeur :

```
| x = 42 # x prend la valeur 42.  
| x = x + 2 # on additionne à la donnée stockée dans x  
| le nombre 2 et on stocke le résultat de nouveau dans x
```

Après compilation, on peut écrire dans le shell :

```
| >>> x # accès à la valeur mémorisée dans x.  
| 44 # réponse
```

Remarque. Dans la variable affectée, la nouvelle donnée "écrase" la donnée précédente : on perd son ancienne valeur.

2. Les booléens

2.1 Qu'est-ce que c'est ?

Définition.

Les **booléens** sont un type spécial ne contenant que deux valeurs **Vrai** et **Faux**, traduites en Python par **True** et **False** avec une majuscule.

2.2 Opérateurs sur les booléens

Les booléens permettent de représenter les résultats d'expressions logiques.

Exemple. Soit p l'expression logique " $0 < x < 3$ ".

Si x vaut 1 alors p a pour valeur **True**.

Si x vaut 4 alors p a pour valeur **False**.

Les opérateurs sur les booléens correspondent aux connecteurs logiques rencontrés en mathématiques. Leur traduction en **python** est donnée plus loin.

- la négation (**non**(p)),
- la conjonction (p **et** q),
- la disjonction (p **ou** q).

Les règles d'évaluation correspondent au tableau suivant :

p	q	non (p)	p et q	p ou q
Vrai	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Vrai	Faux	Vrai
Faux	Faux	Vrai	Faux	Faux

3. L'instruction conditionnelle

3.1 Algorithme

Quand on veut écrire un programme, on souhaite établir des connections logiques entre les instructions. Ainsi, l'instruction conditionnelle a pour objet d'intervenir dans le choix de l'instruction suivante en fonction d'une expression booléenne qu'on désignera par **condition** :

	Si condition alors bloc d'instructions 1 sinon bloc d'instructions 2 Fin-du-Si
	signifie que • si la condition est vérifiée (expression booléenne=True) alors le programme exécute les instructions du bloc 1 • si la condition n'est pas vérifiée (expression booléenne=False) alors le programme exécute les instructions du bloc 2

3.2 Syntaxe en Python

```

if condition:
    bloc d'instructions 1
else:
    bloc d'instructions 2

```

- **Si** et **Sinon** se traduisent par **if** et **else**. Ils sont situés au même niveau d'indentation.
- **alors** se traduit par **:** en bout de ligne et une indentation à la ligne suivante.
- **Fin-du-Si** se traduit par un retour à la ligne sans indentation.

3.3 Exemple

On veut tester si un nombre est proche de 3 à 10^{-4} près.

```
x = nombre
distance = abs(x-3) #la distance se mesure en valeur absolue ✓
if distance <= 10**(-4):
    print(True)
else:
    print(False)
```

Remarque. La partie **sinon** est optionnelle. Sans elle, si la condition n'est pas vérifiée, alors la machine n'exécute rien.

3.4 À propos des conditions

L'expression booléenne derrière le **si** joue le rôle de test. Pour exprimer cette condition, on a besoin des **opérateurs de comparaison** (inférieur strict, supérieur strict, inférieur ou égal, supérieur ou égal, égal à, différent de) et des **connecteurs logiques** (non, et, ou).

Syntaxe Python pour les opérateurs de comparaison et les connecteurs logiques

En mathématiques	En Python	En mathématiques	En Python
<	<	≤	<=
>	>	≥	>=
=	==	≠	!=
et	and	ou	or
non	not		

Exemples.

- Calcul du carré d'un nombre positif.

On écrit dans l'éditeur :

```
x = 4
if x >= 0:
    rac = x**0.5
```

Après compilation, on peut écrire dans le shell :

```
>>> rac
2.0 # réponse
```

- Comprenez-vous le message d'erreur ?

```
x = -5
if x >= 0:
    carr = x**2

>>> carr
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    carr
NameError: name 'carr' is not defined
```

- Condition avec un " et ".

```
x = 0.5
if x >= -1 and x <= 1:
    print("Il existe un angle theta tel que x = ✓
          cos(theta).")
```

3.5 Imbrication de plusieurs conditions

On peut se trouver face à un problème qui se scinde en plus de deux cas (par exemple, dans le cas des équations du second degré, on teste si le discriminant est strictement positif, nul ou strictement négatif). Dans ce cas, voici comment procéder :

```
Si condition 1
    alors bloc d'instructions 1
sinon si condition 2
    alors bloc d'instructions 2
    sinon si condition 3
        alors bloc d'instructions 3
    ...
    sinon bloc final
Fin-du-Si
```

signifie que

- si la condition 1 est vérifiée (expression booléenne=True)
alors le programme exécute les instructions du bloc 1
- si la condition 1 n'est pas vérifiée (expression booléenne=False) mais que la condition 2 est vérifiée
alors le programme exécute les instructions du bloc 2
- si les conditions 1 et 2 ne sont pas vérifiées mais que la condition 3 est vérifiée
alors le programme exécute les instructions du bloc 3...
- si aucune condition n'est vérifiée
alors le programme exécute les instructions du bloc final

Remarque. Si la condition 1 est vérifiée alors le programme ne se préoccupe pas des conditions suivantes.

3.6 Syntaxe en Python

```
if condition 1:
    bloc d'instructions 1
elif condition 2:
    bloc d'instructions 2
elif condition 3:
    bloc d'instructions 3
.
.
.
else:
    bloc final
```

- Sinon si se traduit par elif.

3.7 Exemple : la bataille navale (simplifiée !)

Le bateau ayant pour coordonnées a, b et le joueur rentre deux valeurs c, d :

```
x = c
y = d
if x == a and y == b:
    print("Touché, coulé !")
elif x == a or y == b:
    print("Bateau en vue")
else:
    print("Coup dans l'eau")
```

Remarque. Il est important de tester son algorithme sur un ensemble de valeurs représentatives.

Que choisiriez-vous pour tester l'exemple ci-dessus si $(a, b) = (2, 6)$?

4. La boucle itérative

4.1 Algorithme

Il arrive souvent que l'on ait besoin de répéter une même opération plusieurs fois. Le cas des suites définies par u_0 et $\forall n \in \mathbb{N}, u_{n+1} = f(u_n)$ en est la parfaite illustration : pour calculer u_5 , il est nécessaire de calculer de proche en proche u_1, u_2, u_3, u_4 et enfin u_5 .

Pour éviter d'écrire les cinq instructions l'une après l'autre, on utilisera la boucle itérative.

Vous serez certainement convaincus de son utilité si on vous demande de calculer u_{100} !

Pour i allant de n_1 à n_2 par pas de p faire bloc d'instructions Fin-du-Pour

signifie que

- i prend la valeur n_1
- **si** la condition $i \leq n_2$ est vérifiée
alors le programme exécute le bloc d'instructions et i prend la valeur $i + p$
- puis le programme teste à nouveau la condition sur i .

4.2 Syntaxe en Python

```
for i in range(n1, n2 + 1, p):
    bloc d'instructions
```

- **Pour** se traduit par **for**
- i allant de n_1 à n_2 par pas de p se traduit par `range(n1, n2 + 1, p)`.
 Notez le `n2 + 1`, nous y reviendrons de nombreuses fois.
- **faire** se traduit par les " : " et l'indentation du bloc d'instructions.
- **Fin-du-Pour** se traduit par un retour à la ligne au même niveau que le **for** (fin d'indentation).

4.3 Exemples

1. Calcul de 2^5 avec une boucle itérative.

```
n = 1      # initialisation
for i in range(1, 6, 1):    # on peut améliorer ce ✓
    range , voir ci-dessous
    n = 2 * n
```

Après compilation :

```
>>> n
32                                # réponse
```

2. Calcul de $6!$.

```
fact = 1      # initialisation
for i in range(2, 7):
    fact = i * fact
```

```
>>> fact
720                                # réponse
```

4.4 La commande `range` en Python

Nous venons de voir que l'instruction

Pour i allant de n_1 à n_2 par pas de p

se traduit en Python par

```
for i in range(n1, n2 + 1, p)
```

- Dans `range`, les variables `n1`, `n2` et `p` sont de type **entier**.
- Si on ne précise pas le pas `p`, alors par défaut il vaut 1.
Dans l'instruction `for i in range(n1, n2+1)`, i prend successivement les valeurs `n1`, `n1+1`, `n1+2`, ..., `n2`.
- Dans le calcul de 2^5 précédent, la valeur de i n'est pas utilisée. Seul le nombre de tours de boucle est important, ici 5 tours.
La commande `range(1, 6, 1)` peut être améliorée en `range(1, 6)` ou `range(0, 5)` ou, par la syntaxe très efficace `range(5)`.
À chaque fois il y a 5 tours de boucles.

À retenir :

La commande `range(n)` "représente" les n valeurs entières consécutives de 0 à $n - 1$.

5. La boucle conditionnelle

5.1 Algorithme

On peut aussi être amené à répéter un bloc d'instructions sans savoir combien de fois on devra le répéter.

Exemple. Disposant d'une suite croissante, non majorée, on cherche à trouver le plus petit entier p tel que la valeur au rang p dépasse 10000.

Dans ce cas, on utilise la boucle **Tant que** qui permet de répéter le bloc d'instructions tant qu'une certaine condition est vérifiée.

	Tant que condition faire bloc d'instructions Fin-du-Tant-que
	signifie que
	Tant que la condition est vérifiée (expression booléenne=True) Faire le bloc d'instructions.

5.2 Syntaxe en Python

```
while condition:
    bloc d'instructions
```

- **Tant que** se traduit par **while**.
- **faire** se traduit encore par **:** et l'indentation du bloc d'instructions.
- **Fin-du-Tant-que** se traduit encore par la fin de l'indentation.

5.3 Exemple

Recherche du premier entier n tel que la somme des entiers de 1 à n dépasse 11.

```
n = 0
s = 0    # initialisations
while s < 11:
    n = n + 1
    s = s + n
    # à la fin du k ième tour de boucle, n vaut k et s
    # vaut 0 + ... + k
```

Après compilation,

```
>>> n
5      # réponse : n=5 est l'entier cherché et dans ce cas s=15
```

Le commentaire expliquant la situation à la fin de k ième tour de boucle s'appelle un **invariant de boucle**.

5.4 Comparaison avec la boucle itérative

Dans la boucle **Pour**, on connaît par avance le nombre d'étapes effectuées dans la boucle.

En revanche, on ne sait pas au bout de combien de temps la boucle **Tant que** s'arrête. Il peut parfois être utile d'ajouter un compteur. On l'initialise à 0 avant la boucle et on termine le bloc d'instructions par `compteur = compteur + 1`.