Syntaxe Python : fiche récapitulative

Lectures d'entrées (input())

Pour interagir au sein d'un programme avec un utilisateur, on peut **récupérer des informations** au clavier directement rentrées par celui-ci.

En général, cette entrée est ensuite conservée dans une **variable**, afin d'être manipulable par la suite.

Pour ce faire, on utilise la fonction `input()` :

```
print("Ecrivez ce que vous souhaitez !")
entreeUtilisateur = input()  # on récupère l'entrée utilisateur
print("Vous avez écrit : ", entreeUtilisateur)  # On affiche ce que
l'utilisateur
                                                # a entré
```

L'entrée utilisateur récupérée est une **chaîne de caractères**. Cependant, on cherche parfois à récupérer un nombre de la part de l'utilisateur. Il est alors nécessaire de modifier l'entrée de l'utilisateur pour faire réaliser au programme que ce qu'il récupère est en réalité un entier.

Dans ce cas précis, il est nécessaire de **modifier le type** de la variable (on parle de `cast` en anglais) :

```
print("Veuillez écrire un nombre.")
nombre = int(input())  # int() sert à transformer une variable en nombre entier
print("Le double de ce nombre est : ", nombre*2)
```

Conditions (if)

Fonctionnement du if

Jusqu'à présent, nos programmes réalisaient toujours les mêmes suites d'opérations. Il est cependant possible de **suivre des instructions en fonction d'une condition** déterminée dans le programme. Cette condition suit la **logique booléenne** évoquée au début de l'année. La condition est une **proposition** (par exemple, `maVariable > 5`). Cette proposition peut être **vraie** (true) ou **fausse** (false).

Dans notre exemple, la proposition `maVariable > 5` sera vraie si le contenu stocké dans `maVariable` est strictement supérieur à 5. Dans tous les autres cas, elle sera fausse.

Le mot-clé `if` permet de réaliser toutes les instructions qui sont contenues à l'intérieur, comme pour une boucle `for`, mais uniquement si la proposition logique qui lui est apposée est vraie.

```
print("Veuillez indiquer votre âge.")
age = int(input())

if age >= 18:  # notre condition est : l'âge rentré est supérieur ou égal à 18
    print("Vous êtes majeur !")  # On n'affiche ce texte que si age >= 18

if age < 18:
    print("Vous êtes mineur !")  # On n'affiche ce texte que si age < 18
```

De manière générale, une condition if peut tout à fait **accueillir un bloc** d'instructions :

```
print("instruction 0")  # est en dehors du if : se produit dans tous les cas
if ma_condition:
    print("instruction 1")  # se produit si ma_condition est vraie
    print("instruction 2")  # se produit si ma_condition est vraie
    print("instruction 3")  # se produit si ma_condition est vraie
print("instruction 4")  # est en dehors du if : se produit dans tous les cas
```

Gérer deux possibilités opposées : if / else

Dans l'exemple de l'âge que nous avons donné ci-dessus, nous avons dû utiliser deux `if` :

- Un if pour déterminer si l'âge donné par l'utilisateur était supérieur ou égal à 18
- in if pour déterminer si l'âge donné par l'utilisateur était strictement inférieur à 18.

Or, il est strictement impossible de rentrer dans les deux if en même temps : si l'âge est supérieur ou égal à 18, il ne peut être inférieur à 18 - et inversement.

Pour rendre plus facile l'écriture de ce genre de conditions, où l'on rentre forcément **soit dans un cas, soit dans l'autre**, on utilise le mot-clé `if` ("si", en anglais), en combinaison avec le mot-clé `else` ("sinon", en anglais).

```
print("Veuillez indiquer votre âge.")
age = int(input())

if age >= 18:  # soit l'âge est supérieur ou égal à 18
    print("Vous êtes majeur !")
else:  # soit il ne l'est pas
    print("Vous êtes mineur !")
```

Quelques précisions importantes sur le mot-clé `else` :

- Il **suit** forcément un `if`
- On le place au **même niveau d'indentation** que le `if` auquel il est lié
- Il ne **prend jamais de condition** (la condition est implicite : on rentre dans le `else` uniquement si la condition de notre `if` est fausse).

Gérer de multiples possibilités : if / elif / else

Un dernier cas de figure existe : on souhaite tester de **multiples possibilités** distinctes qui ne sont pas forcément mutuellement exclusives. Dans ce cas, on peut rajouter entre notre `if` et notre `else` un ou plusieurs `elif` (else if, c'est à dire "sinon si" en anglais).

Par exemple, l'eau existe en trois états différents selon sa température :

- A 0°C ou moins, l'eau est gelée (solide).

- Entre 1°C et 99°C, l'eau est liquide.
- A 100°C et plus, l'eau est sous forme de vapeur (gaz).

On peut représenter l'état de l'eau selon sa température dans le programme suivant :

```
print("Veuillez rentrer la température de l'eau.")
temperature = int(input())

if temperature <= 0:  # On présente un premier cas de figure
    print("l'eau est gelée (solide).")
elif temperature >= 100:  # Si le premier cas est faux, on teste un second cas
    print("l'eau est sous forme de vapeur (gaz).")
else:  # Si les deux cas précédents sont faux, else sert de cas général.
    print("l'eau est liquide."
```

A noter qu'il est possible d'avoir de multiples `elif` d'affilée. Chaque `elif` doit être accompagné d'une condition. Comme pour `else`, `elif` doit suivre un `if` initial et ne peut être utilisé seul. De plus, `elif` doit être au même niveau d'indentation que le `if` qu'il accompagne.

Conditions multiples

Il est possible d'enchainer plusieurs `if` les uns à l'intérieur des autres :

```
print("Veuillez indiquer votre âge.")
age = int(input())

if age >= 18:  # soit l'âge est supérieur ou égal à 18
    print("Etes-vous étudiant ? (oui / non)")
    reponse = input()
    # le if ci-dessous est à l'intérieur du bloc "if age >= 18:"
    if reponse = "oui":  # ce test ne se produit donc que lorsque age >= 18
        print("Vous êtes majeur et étudiant !")
    else:
        print("Vous êtes majeur, mais n'êtes pas étudiant !")
else:  # soit il ne l'est pas
    print("Vous êtes mineur !")
```

Cependant, un tel code peut vite devenir difficile à comprendre, en particulier si l'on a de nombreux tests imbriqués les uns dans les autres. On peut alors **combiner des conditions** simples au sein d'une seule et unique condition multiple, à l'aide de deux **opérateurs** de la logique booléenne : `and` et `or`.

- `condition1 and condition2` ne sera vraie que si condition1 **ET** condition2 sont toutes les deux vraies
- `condition1 or condition2` sera vraie si condition1, condition2, ou les deux sont vraies.

Dans notre exemple précédent, cela donnerait :

```
print("Veuillez indiquer votre âge.")
age = int(input())
print("Etes-vous étudiant ? (oui / non)")
reponse = input()

if age >= 18 and reponse = "oui":
    print("Vous êtes majeur et étudiant !")
elif age >= 18:
    print("Vous êtes majeur, mais n'êtes pas étudiant.")
else:
    print("Vous êtes mineur !")
```

Comme pour des opérations mathématiques classiques, il est possible de combiner autant de propositions avec autant d'opérateurs que l'on souhaite. Au lieu d'utiliser les opérateurs `+ - * /` , on utilise les opérateurs `and or` .

Exemple :

- `(condition1 or condition2) and (condition3 or condition4)`

  Pour que cette proposition soit vraie, il faut que condition1 ou condition2 (ou les deux) soit vraie ET que condition3 ou condition4 (ou les deux) soit vraie.

Répétitions conditionnées (while)

Nous avons vu un premier moyen de répéter des instructions : la boucle `for` . Ce type de boucle, appelé **boucle bornée**, nécessite de préciser un nombre de fois où l'on va répéter notre série d'instructions.

Cependant, il existe des cas pour lesquels on ne sait pas à l'avance combien de fois on va devoir faire un tour de boucle. On utilise, dans ces cas-là, une **boucle non-bornée**, aussi appelée **boucle conditionnée**, grâce au mot-clé `while`.

`while` (que l'on peut traduire en français par "tant que"), permet de rester à l'intérieur d'une boucle tant qu'une **condition** donnée en parallèle de notre `while` reste vraie.

```
print("Quel montant dans votre compte en banque ?")
argent = int(input())

while argent > 0:  # Tant que la variable argent est positive, on reste dans la
                   # boucle
    print("Combien d'euros souhaitez-vous retirer ?")
    montant = int(input())
    print("Vous avez retiré ", montant)
    argent = argent - montant  # on soustrait le montant retiré du compte.

print("Vous n'avez plus assez d'argent sur votre compte !")
```

Dans l'exemple ci-dessus, on ne peut pas prédire combien de tours de boucle seront nécessaires. Ce nombre est variable : il dépend du montant initial (`argent`) et de la façon dont l'utilisateur retire l'argent de son compte en banque (`montant`) :

- Si `argent = 1000` et que l'utilisateur retire toujours un euro par un euro (`montant = 1`), il faudra mille tours de boucle.

- Si `argent = 1000` et que l'utilisateur retire 500, puis 250, puis 200, puis 50, il ne faudra que 4 tours.

Attention à **bien réfléchir aux conditions** de vos `while` ! Si la condition est inatteignable, vous ne rentrerez jamais dans la boucle comme désiré. Dans l'exemple précédent :

```
while argent < 0:
```

signifierait que l'on souhaite rester dans la boucle tant que le compte est vide, ce qui n'est pas sensé arriver.

A l'inverse, il est possible de créer une condition qui sera toujours vraie. Dans ce cas, il sera impossible de sortir de la boucle. on parle de **boucle infinie** (Il est parfois possible, bien que ce soit rare, que l'on souhaite obtenir une boucle infinie).

```
#  Ce programme ne s'arrêtera jamais d'écrire "je suis bloqué"
while true:
    print("Je suis bloqué !")
```

```
# Ce programme ne s'arrêtera jamais non plus : on utilise la valeur de a pour
déterminer quand sortir de la boucle, mais on ne change pas la valeur de a au
sein de notre boucle : a restera toujours à 5, et sera donc toujours positif.
a = 5
while a > 0:
    print("Je suis bloqué aussi !")
```

Ecrire une boucle bornée à l'aide d'une boucle while

En réalité, la boucle `for` n'est qu'une implémentation (une façon d'écrire) de la boucle while.

Le code suivant :

```
for i in range(20):
    print("Bonjour")
```

Correspond exactement au code suivant :

```
i = 0  # i démarre à 0
while i < 20:  # Tant que i n'a pas atteint notre plafond (la valeur de "range")
    print("Bonjour")
    i = i + 1  # On augmente i à chaque tour de boucle
```

On pourrait donc théoriquement complètement se passer de boucles `for`, mais celles-ci sont faciles à lire. On les privilégie donc lorsque l'on connait le nombre d'itérations (tour de boucles) à réaliser.