Informatique et Calcul Scientifique

Cours 2 : Flux de contrôle

18.09.2024

La fois passée on a vu

- Une introduction à l'informatique et au langage Python
- Les types `int`, `float`, `str` et des opérateurs associés
- La notion de variables
- Les fonctions `print()` et `input()` pour interagir avec l'utilisateur via l'entrée standard (le clavier) et la sortie standard (l'écran)

Aujourd'hui on verra

Les instructions de **flux de contrôle**, qui permettent de contrôler le déroulement d'un programme :

- L'instruction `if`
- Les boucles `while`
- Les boucles `for`

Comparaisons

Il est possible de comparer les valeurs de deux objets. On utilise pour cela des **opérateurs relationnels :**

Syntaxe	Signification
==	Egal à
!=	Différent de
<	Inférieur à
>	Supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
`is`	Même identité [1]
`is not`	Identité différente

Une comparaison consiste ainsi en une instruction qui sera évaluée, et dont le résultat peut prendre deux valeurs : `True` (vrai) ou `False` (faux).

1. cet opérateur compare si deux objets pointent vers le même espace mémoire

Comparaisons : remarques

Quelques remarques :

- On peut comparer deux nombres (`int` , `float`), mais également des objets d'autres types (`str` , `list` , `dict`).[2]
- Attention à ne pas confondre l'opérateur de comparaison " `==` " et l'opérateur d'affectation " `=` " !

2. Nous étudierons ces différents types lors des prochaines semaines.

Comparaisons : exemples

Voici quelques exemples de comparaisons sur deux variables :

```
x = 3
y = 4
print(x < 0)
print(x <= y)
print(y > -2)
print(y >= 2 * x)
print(x == y)
print(x != 2)
print(0 <= x <= 10)
```

```
Output :
False
True
True
False
False
True
True
```

Note : vous êtes encouragé.e.s à créer vos propres exemples, et à les tester dans un interpréteur Python !

Comparaisons : exemples

```
x = "I love Python"
y = "I love Python"
print(x == y, x!= y)
print("== teste l'egalite des valeurs:")
print(x, y)
print(x is y, x is not y)
print("is teste l'identite des objets:")
print(id(x), id(y))
```

Output :

```
True False
== teste l'egalite des valeurs:
I love Python I love Python
False True
is teste l'identite des objets:
4383263536 4383263472
```

Le type bool

Le résultat d'une comparaison est une valeur de type **booléen** (`bool`).

- `True` et `False` sont les deux seuls objets de type `bool` .
- `False` est assimilé à la valeur 0 et `True` à la valeur 1.

```
print(True == 1, True == 0)
print(False == 0, False == 1)
```

Output :

```
True False
True False
```

Il y a plusieurs manières d'affecter un booléen à une variable :

1. en affectant directement `True` ou `False` à une variable

```
x = True
print(x)
```

Output :

```
True
```

Le type bool

Il y a plusieurs manières d'affecter un booléen à une variable :

2. en affectant le *résultat* d'une comparaison à une variable

```
y = 3 > 5
print(y)
z =  x == False
print(z)
```

Output :

```
False
False
```

3. en castant un objet d'un autre type en un `bool`

```
s1 = ""
s2 = "non vide"
print(s1, bool(s1), sep = ' & ')
print(s2, bool(s2), sep = ' & ')
```

Output :

```
 & False
non vide & True
```

Structure conditionnelle

Jusqu'ici, on ne sait écrire qu'un programme **linéaire** : une suite d'instructions qui est exécutée ligne après ligne, quelle que soit l'entrée du programme.

```
# instruction 1
# instruction 2
# groupe d'instructions a repeter
# groupe d'instructions a repeter
# groupe d'instructions a repeter
```

Pour écrire des programmes plus intéressants, on utilisera des instructions qui permettent à un programme d'exécuter une séquence différente d'instructions selon qu'une certaine condition est vérifiée ou pas.

- On parle alors de **structures conditionnelles**.

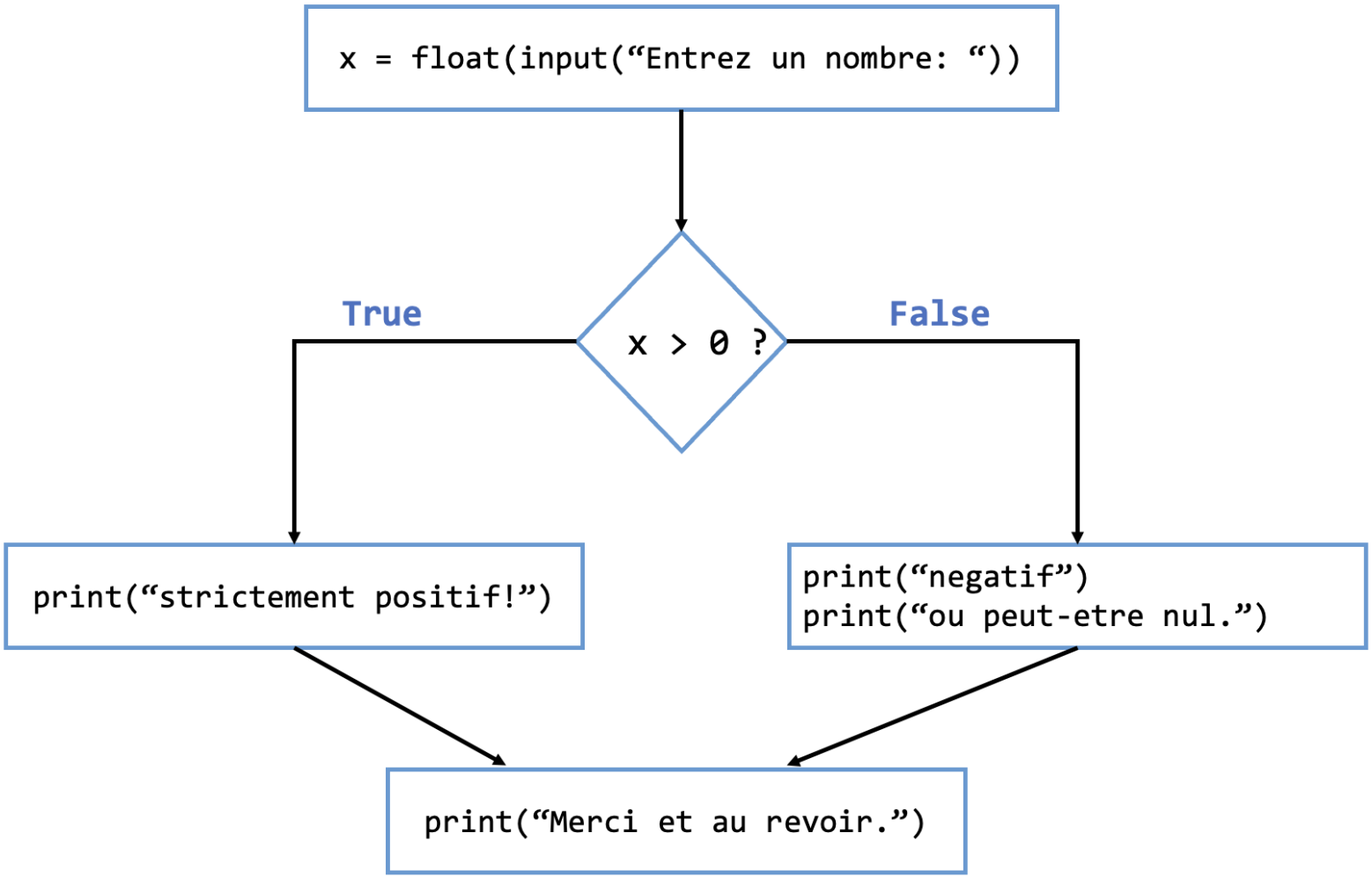
L'instruction `if`

L'instruction `if... else` permet de tester une condition, puis d'effectuer un bloc d'instructions différent en fonction du résultat du test.

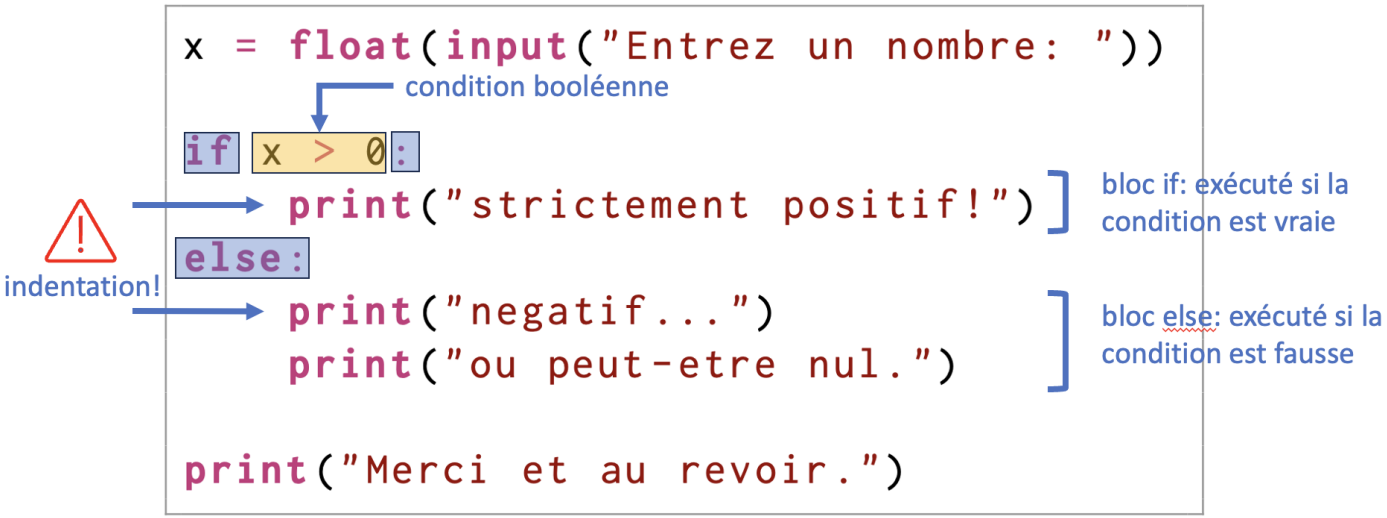
- Le programme ci-dessous demande un nombre à l'utilisateur et se comporte différemment si ce nombre est strictement positif, ou s'il est négatif ou nul.

```
x = float(input("Entrez un nombre svp: "))

if x > 0:
    print("Votre nombre est strictement positif.")
else:
    print("Votre nombre est negatif ou nul.")

print("Merci et au revoir.")
```

L'instruction if



Structure d'une instruction `if`



- L'instruction `if` est toujours suivie d'une **condition booléenne**
- Les **deux points** `:` après `if` et `else` indiquent le début d'un bloc d'instructions
- Chaque bloc d'instructions est délimité par un tab (ou quatre espaces), aussi appelé **indentation**
- La fin de l'indentation indique la fin de l'instruction `if`... `else`.

Structure d'une instruction `if`

- La clause `else` est optionnelle :

```
x = int(input("x: "))
if x % 2 == 0:
    print(f"{x} est pair")
print("au revoir")
```

Output :

```
x: 3
au revoir
```

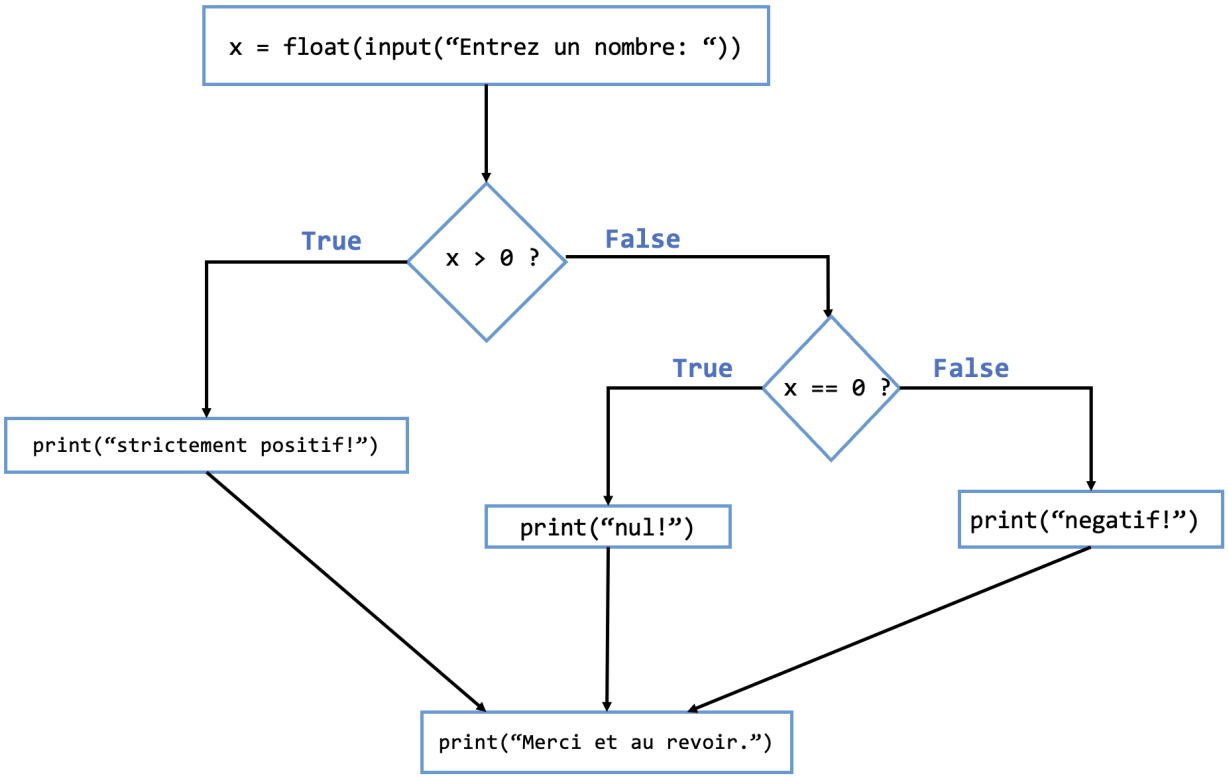
- Par contre, un bloc `else` vide (ou un bloc `if` vide) génère une erreur (testez-le !)
- Pour avoir un bloc `if` ou un bloc `else` qui "ne fait rien", on peut utiliser l'instruction `pass` :

```
# ce code ne fait rien
if 1 > 0:
    pass
```

Différencier un plus grand nombre de cas

Que se passe-t-il si on aimerait choisir parmi plusieurs conditions ?

- Exemple : demander un nombre `x` à l'utilisateur, et distinguer les cas : `x` strictement positif, `x` strictement négatif, `x` nul.



Différencier un plus grand nombre de cas

- On peut imbriquer un `if... else` dans le bloc `else` ...

```
x = float(input("Entrez un nombre: "))

if x > 0:
    print("strictement positif!")
else:
    if x == 0:
        print("nul!")
    else:
        print("negatif!")

print("Merci et au revoir.")
```

Différencier un plus grand nombre de cas

- ... ou on peut utiliser une clause `elif` (“`else if`”) !

```
x = float(input("Entrez un nombre: "))

if x > 0:
    print("strictement positif!")
elif x == 0:
    print("nul!")
else:
    print("negatif!")

print("Merci et au revoir.")
```

On peut rajouter un nombre arbitraire de clauses `elif`, du moment qu'on associe un bloc d'instructions à chacune.

Combiner des expressions booléennes

Comment faire si on aimerait que plusieurs conditions soient vérifiées simultanément ?

- Exemple : on veut tester si un nombre est à la fois pair **et** strictement positif.

⇒ On peut imbriquer deux expressions `if` ... :

```
x = float(input("Entrez un nombre pair et positif: "))

if x % 2 == 0:
    if x > 0:
        print("OK")
    else:
        print("pas OK")
else:
    print("pas OK")
```

Combiner des expressions booléennes

... mais on peut aussi utiliser le mot-clé `and` pour tester la **conjonction** de deux conditions booléennes !

- On peut combiner les valeurs de plusieurs expressions booléennes avec les opérateurs `and`, `or` et `not`.

```
x = float(input("Entrez un nombre pair et positif: "))

if x % 2 == 0 and x > 0:
    print("OK")
else:
    print("pas OK")
```

Combiner des expressions booléennes

- Conjonction :

and	True	False
True	True	False
False	False	False

- Disjonction :

or	True	False
True	True	True
False	True	False

- Négation :

x	**not x**
True	False
False	True

Combiner des expressions booléennes

Quelques exemples :

```
x = 2
y = 3.0
print(x < y and type(y) == float)
print(x < y or type(y) == str)
print(not x < y)
```

Output :

```
True
True
False
```

Boucles et itération

En programmation, il est fréquent qu'on veuille répéter un bloc d'instructions un certain nombre de fois. On utilise pour ceci des **boucles**. Il existe deux types de boucles différentes selon l'utilisation désirée :

- Pour répéter une portion de code *tant qu'une* certaine condition est vraie, on utilisera une boucle `while`. On ne sait pas d'avance combien d'itérations la boucle fera !
- Pour répéter une portion de code un *nombre prédéfini* de fois, on utilisera en général une boucle `for`.

Boucles while

Une boucle while permet à un programme de continuer à exécuter la même portion de code tant qu'une condition booléenne est vraie.

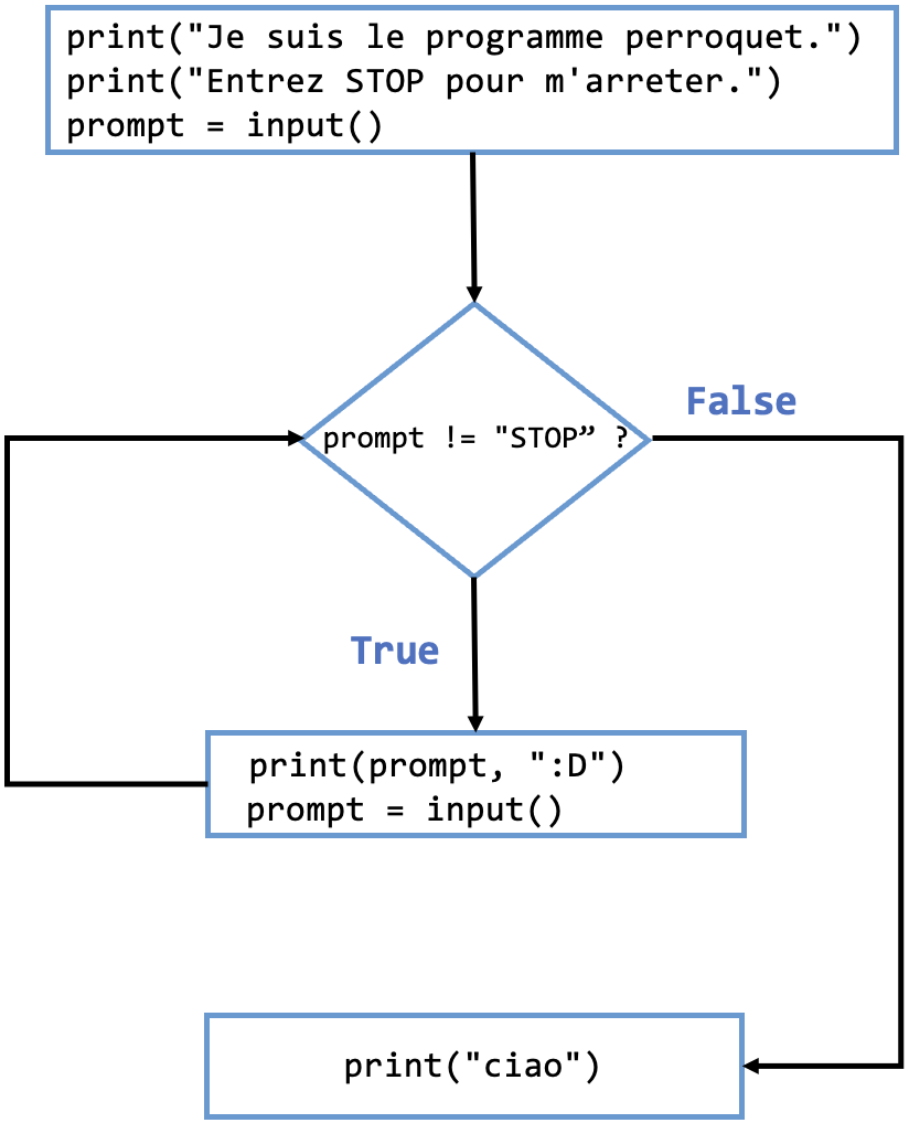
Exemple : On veut écrire un programme qui continue à

- prendre une chaîne de caractères de l'entrée standard
- afficher cette chaîne de caractères

jusqu'à ce que l'utilisateur entre "STOP".

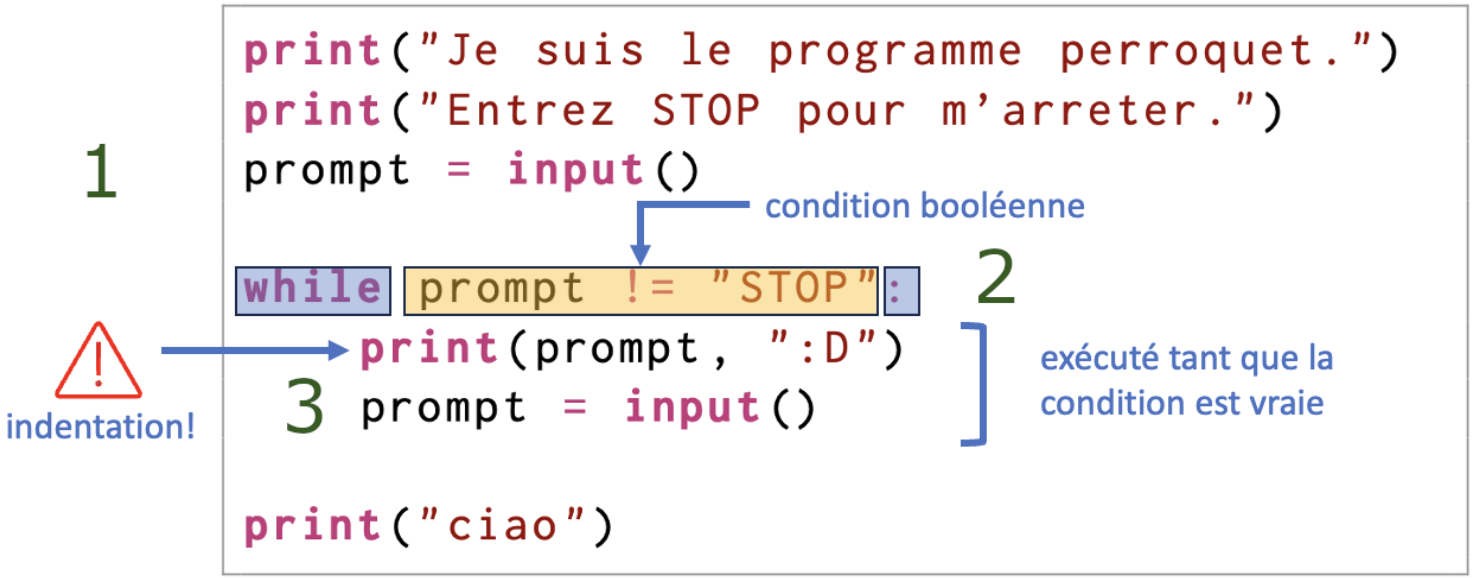
```
print("Je suis le programme perroquet.")
print("Entrez STOP pour m'arreter.")
prompt = input()

while prompt != "STOP":
    print(prompt, ":D")
    prompt = input()

print("ciao")
```

Structure d'une boucle while



Structure d'une boucle `while`

Pour fonctionner, une boucle `while` doit être composée de trois parties distinctes :



1. L'**initialisation** de la variable d'itération
2. L'**évaluation** de la condition booléenne testée par la boucle `while`
3. La **mise à jour** de la variable d'itération

Boucle while : exemples

Ecrire un programme qui demande à l'utilisateur un entier `n` et qui affiche les nombres de `1` à `n` inclus.

```
n = int(input("Entrez un entier: "))

i = 1
while i <= n:
    print(i)
    i += 1

print("c'est fini!")
```

- Testez ce programme en entrant différentes valeurs (essayez aussi des valeurs plus petites que 1).
- Ce programme aurait aussi pu être implémenté avec une boucle `for` (plus tard...)

Boucle while : exemples

Ecrire un programme qui affiche toutes les puissances de 2 entre 1 et 10^6.

```
x = 1
while x < 1_000_000:
    print(x)
    x *= 2
```

Ou encore

```
i = 0
while 2**i <= 1_000_000:
    print(2**i)
    i += 1
```

Boucles infinies

Attention ! Si la condition booléenne de la boucle `while` s'évalue toujours à `True`, la boucle va itérer à l'infini.

- En général, ce n'est pas le comportement attendu d'une boucle `while` !

Exemple : Le code suivant va afficher `Hello world!` jusqu'à ce qu'on interrompe l'exécution du programme :

```
while True:
    print("Hello world!")
```

Boucles infinies

Une boucle infinie peut se produire si la condition booléenne de la boucle `while` est mal définie.

- Reprenons l'exemple du programme qui prend un entier `n` et affiche les nombres de `1` à `n` inclus :

```
n = int(input("Entrez un entier: "))

i = 1
while i >= 1:
    print(i)
    i += 1

print("c'est fini!")
```

- La condition booléenne vaudra toujours `True`, donc le bloc d'instructions sera toujours exécuté

Boucles infinies

Une boucle infinie peut également se produire si on ne modifie pas le corps de la boucle, et ne met donc pas à jour la variable d'itération.

- Dans le cas du code ci-dessous, on ne met pas le compteur `i` à jour et par conséquent la condition `i <= n` restera vraie (pour tout $n > 1$).

```
n = int(input("Entrez un entier: "))

i = 1
while i <= n:
    print(i)

print("c'est fini!")
```

Instruction break

On peut utiliser une instruction `break` pour quitter immédiatement l'exécution du corps de la boucle `while` [3], même si la condition de la boucle est encore satisfaite.

En général, l'instruction `break` sera dans une structure conditionnelle (une instruction `if`).

- Exemple : demander un entier `p` à l'utilisateur et afficher le *premier* multiple de 17 et de 19 plus grand ou égal à `p` :

```
p = int(input("entier de depart: "))

while True:
    if p % 17 == 0 and p % 19 == 0:
        break
    p += 1

print(p)
```

3. ou de la boucle `for` que nous verrons par la suite

Instruction break

L'instruction `break` peut être utile, mais en général il vaut mieux éviter de l'utiliser : ce n'est pas une bonne pratique de programmation d'avoir des instructions qui permettent à l'exécution du code de sauter d'une ligne à l'autre d'une manière qui peut ne pas être prévisible (et ce dans n'importe quel langage de programmation). [4]

- On peut toujours se débarasser d'un `break` dans une instruction `if` à l'intérieur d'une boucle `while` en incorporant la négation de la condition booléenne du `if` à celle du `while`.

```
while condition_1:
    # on fait quelque chose
    if condition_2:
        break
```

```
while condition_1 and not condition_2:
    # on fait quelque chose
```

- Exercice : réécrire sans instruction `break` le programme du slide précédent.

4. dijkstra, xkcd

Exemple : calculer une moyenne

On veut écrire un programme qui prend des nombres entrés par l'utilisateur et calcule la moyenne de ces nombres.

- On ne sait pas d'avance combien de nombres l'utilisateur désirera entrer : c'est typiquement un cas où il nous faudra une boucle `while`.
- Problème : Comment sortir de la boucle ? On utilise une **valeur sentinelle** !
- Voir le Jupyter notebook "Moyenne - boucle while" sur Moodle.

Boucle for

On utilisera une boucle `for` lorsqu'on veut exécuter une portion de code un nombre connu de fois. Plus particulièrement, lorsqu'on veut parcourir tous les éléments d'un objet itérable.

- Exemple : on veut écrire un code qui affiche un par un tous les caractères d'un `str` :

```
text = 'salut'
for i in text:
    print(i)
```

Output :

```
s
a
l
u
t
```

Boucle for : remarques

```
text = 'salut'
for i in text:
    print(i)
```

- Ici, `i` est une variable dite d'**itération**, qu'on définit directement dans la boucle `for` (et qui aurait pu avoir n'importe quel autre nom).
- `text` est une variable de type `str` qui est **itérable**. C'est-à-dire qu'on peut la parcourir à l'aide de la variable d'itération, qui prendra à chaque tour les valeurs successives contenues dans la variable `text`.
  - On verra par la suite qu'il existe d'autres structures de données qui sont itérables (listes, dictionnaires)
  - Il n'est pas possible d'itérer sur une structure de données qui n'est pas itérable (`int` , `float` , ...)
- Tout comme la boucle `while`, le début du bloc d'instruction est indiqué par les **deux points** et par l'**indentation**

range(n)

Comment fait-on si on ne veut pas parcourir un objet itérable, mais plutôt répéter un bloc d'instructions n fois ?

$\Longrightarrow$ On utilise la fonction `range(n)` !

Pour `n` un entier arbitraire, `range(n)` crée une structure de données itérable qui contient les valeurs `0`, `1`, ..., `n - 1`.

- `range(5)` contient les valeurs `0, 1, 2, 3, 4`
- `range(1)` contient uniquement la valeur `0`
- `range(0)`, `range(-1)`, `range(-2)` ... sont vides.

```
for i in range(5):
    print(f"iteration {i}.")
```

Output :

```
iteration 0.
iteration 1.
iteration 2.
iteration 3.
iteration 4.
```

`range(i, j)`

Si on ne veut pas commencer à `0`, on peut utiliser `range(i, j)` qui contient les valeurs `i, i + 1, ..., j - 1`.

- `range(10, 15)` contient les valeurs `10, 11, 12, 13, 14`
- `range(-5, -4)` contient la valeur `-5`
- `range(10, 10)`, `range(10, 8)`, `range(0, -3)` ... sont vides.

range(i, j) : exemple

`range(i, j)` contient les valeurs `i, i + 1, ..., j - 1`.

```
for i in range(10, 15):
    print(i, end = " ")
print("\n********************")

for i in range(-5, -4):
    print(i, end = " ")
print("\n********************")

for i in range(10, 10):
    print(i, end = " ")
print("********************")

for i in range(10, 8):
    print(i, end = " ")
print("********************")
```

Output :

```
10 11 12 13 14
********************
-5
********************
********************
********************
```

range(i, j, k)

Un `range` peut également contenir des valeurs non consécutives mais qui diffèrent par un **pas** constant. On utilise pour cela `range(i, j, k)` qui contient les valeurs `i`, `i + k`, `i + 2k`, ... jusqu'à `j` non inclus :

- `range(10, 20, 2)` contient les valeurs `10, 12, 14, 16, 18`
- `range(-5, 17, 5)` contient les valeurs `-5, 0, 5, 10, 15`
- `range(10, 15, 6)` contient uniquement la valeur `10`
- `range(10, 8, 2)` est vide.

De plus, le pas `k` peut être négatif !

- `range(5, 0, -1)` contient les valeurs `5`, `4`, `3`, `2`, `1`
- `range(5, 0, -2)` contient les valeurs `5`, `3`, `1`
- `range(0, 5, -1)` et `range(0, 5, -2)` sont vides.

range(i, j, k) : exemple

`range(i, j, k)` contient les valeurs `i`, `i + k`, `i + 2k`, ... jusqu'à `j` non inclus.

```
for i in range(-2, 11, 3):
    print(i, end = " ")
print("\n********************")

for i in range(11, -2, -3):
    print(i, end = " ")
print("\n********************")

for i in range(10, 5, 2):
    print(i, end = " ")
print("********************")

for i in range(5, 10, -2):
    print(i, end = " ")
print("********************")
```

Output :

```
-2 1 4 7 10
********************
11 8 5 2 -1
********************
********************
********************
```

range(i, j, k)

Dans `range(i, j, k)` :

- `i` est la valeur de départ, incluse. Si on ne spécifie pas ce paramètre, il est pris à `0` par défaut.
- `j` est la valeur d'arrivée, non incluse. Ce paramètre est obligatoire.
- `k` est le pas. Si on ne spécifie pas ce paramètre, il est pris à `1` par défaut.

- Si on spécifie deux paramètres dans `range(i, j, k)`, ils correspondent aux valeurs de `i` et de `j` (dans cet ordre).
- Si on spécifie un seul paramètre, il correspond à la valeur de `j`.
- Donc `range(0, n, 1)`, `range(0, n)` et `range(n)` correspondent à la même suite de valeurs `0`, `1`,... `n-1`.

Boucle `while` vs boucle `for`

On aimerait écrire un programme qui demande à l'utilisateur un entier `n` et qui affiche les nombres de `1` à `n` inclus.

Il existe deux manières équivalentes de le faire, en utilisant une boucle `while` ou une boucle `for` :

```
n = int(input("Entrez un entier: "))

i = 1
while i <= n:
    print(i)
    i += 1

print("c'est fini!")
```

```
n = int(input("Entrez un entier: "))

for i in range(n):
    print(i+1)

print("c'est fini!")
```

Question : Peut-on toujours utiliser de manière interchangeable une boucle `while` ou une boucle `for` ?

Boucle `while` vs boucle `for`

On peut toujours simuler une boucle `for` par une boucle `while`, en utilisant un **compteur**.

- Ainsi, les deux codes ci-dessous sont équivalents (quel affichage produisent-ils ?)

```
for i in range(3, 12, 2):
    print(i)
```

```
k = 3
while k < 12 :
    print(k)
    k += 2
```

Par contre, on ne peut pas toujours simuler une boucle `while` par une boucle `for` : si on ne sait pas d'avance combien de fois la boucle doit tourner, on utilisera une boucle `while`.[5]

5. Revoyez les exemples de boucles `while` de ce cours et voyez lesquelles vous pouvez remplacer par des boucles `for`.

Exemple : afficher tous les multiples de 19 entre 0 et 1000

- Avec une boucle `for` :

```
for i in range(1001):
    if i % 19 == 0:
        print(i, end = " ")
```

- Peut-on le faire sans instruction `if` ?

```
for i in range(0, 1001, 19):
    print(i, end = " ")
```

Exemple : afficher tous les multiples de 19 entre 0 et 1000

- Avec une boucle `while` :

```
i = 0
while i <= 1000:
    if i % 19 == 0:
        print(i, end = " ")
    i += 1
```

- Ou encore :

```
i = 0
while i <= 1000:
    print(i, end = " ")
    i += 19
```

- Ou encore :

```
i = 0
while 19 * i <= 1000:
    print(19 * i, end = " ")
    i += 1
```