Apprentissage de la programmation

avec Python 3

Christian VINCENT

LETTRES SORBONNE UNIVERSITÉ

L1 - Semestre 1 - année 2023-2024

Contrôle du flux et répétition

- Chapitre II -

1 Contrôle du flux d'exécution
- Contrôle avec if
- Contrôle avec if et else
- Contrôle avec if, elif et else

2 Prédicats et opérateurs booléens

3 Les boucles

- "Contrôle du flux d'instructions" = "Structure conditionnelle"

- "Contrôle du flux d'instructions" = "Structure conditionnelle"
- Les programmes vus au TD1 se déroulaient de façon assez linéaire et ne prenaient aucune "décision" selon un critère donné.

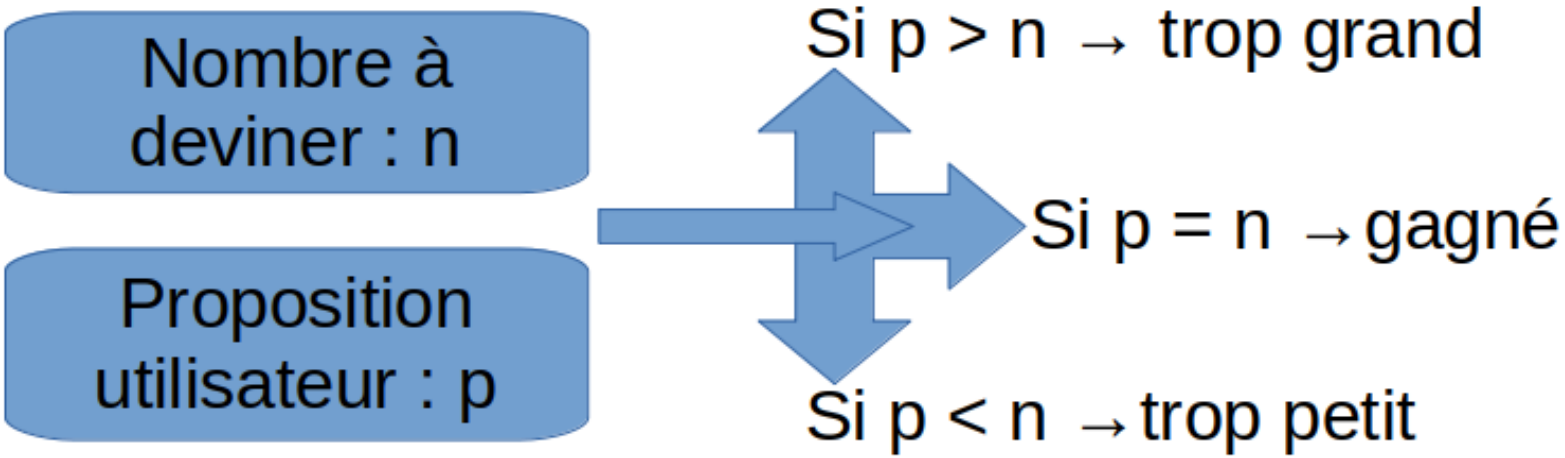
- "Contrôle du flux d'instructions" = "Structure conditionnelle"
- Les programmes vus au TD1 se déroulaient de façon assez linéaire et ne prenaient aucune "décision" selon un critère donné.
- Imaginez que vous écriviez un jeu qui consiste à deviner un nombre. Le programme en propose un.

- "Contrôle du flux d'instructions" = "Structure conditionnelle"
- Les programmes vus au TD1 se déroulaient de façon assez linéaire et ne prenaient aucune "décision" selon un critère donné.
- Imaginez que vous écriviez un jeu qui consiste à deviner un nombre. Le programme en propose un.
  - L'utilisateur propose à son tour un nombre (par exemple avec *input()*)

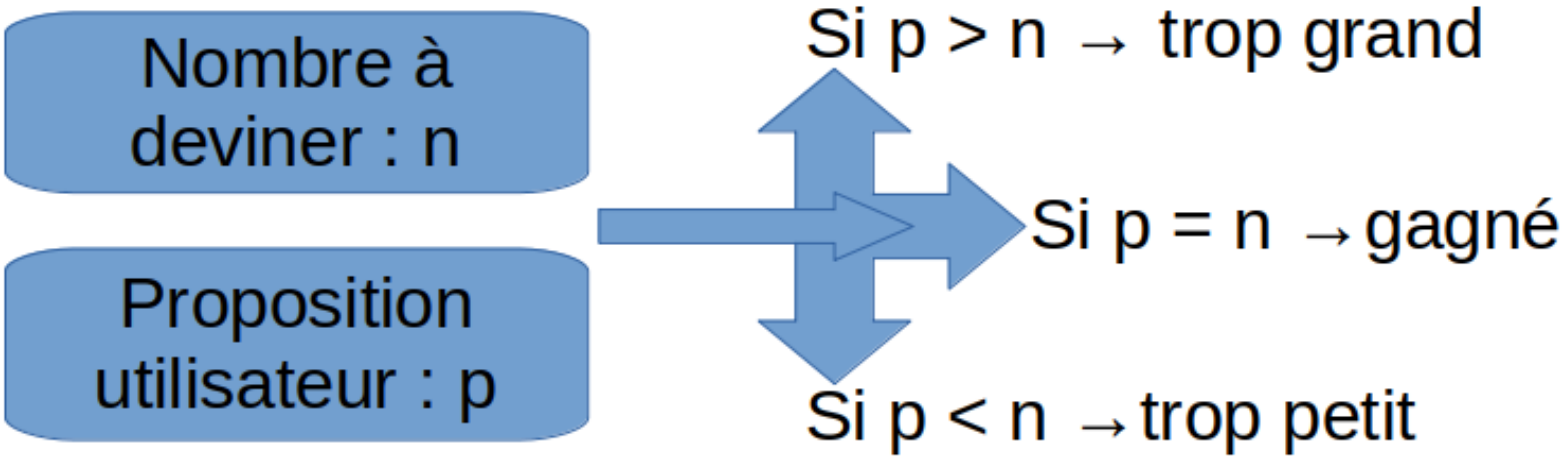
- "Contrôle du flux d'instructions" = "Structure conditionnelle"
- Les programmes vus au TD1 se déroulaient de façon assez linéaire et ne prenaient aucune "décision" selon un critère donné.
- Imaginez que vous écriviez un jeu qui consiste à deviner un nombre. Le programme en propose un.
  - L'utilisateur propose à son tour un nombre (par exemple avec *input()*)
  - Le programme doit répondre seul si la proposition de l'utilisateur est trop grande, trop petite, ou exacte.

On peut résumer la situation par le schéma suivant.

On peut résumer la situation par le schéma suivant.



On peut résumer la situation par le schéma suivant.



La possibilité dans un programme de faire un choix est obtenu avec l'instruction **if** = si.

Observez les lignes de code suivantes :

Observez les lignes de code suivantes :

Usage du if

```
a = 5
if a > 0 :
    print("a est positif")
```

Observez les lignes de code suivantes :

Usage du if

```
a = 5
if a > 0 :
    print("a est positif")
```

Remarques :

- ici, la phrase "a est positif" va s'afficher car la condition $a > 0$ est vérifiée.

Observez les lignes de code suivantes :

Usage du if

```
a = 5
if a > 0 :
    print("a est positif")
```

Remarques :

- ici, la phrase "a est positif" va s'afficher car la condition $a > 0$ est vérifiée.
- notez bien les 2 points (:) à la fin de la ligne **if**... C'est une **instruction composée**.

Observez les lignes de code suivantes :

Usage du if

```
a = 5
if a > 0 :
    print("a est positif")
```

Remarques :

- ici, la phrase "a est positif" va s'afficher car la condition $a > 0$ est vérifiée.
- notez bien les 2 points (:) à la fin de la ligne **if**... C'est une **instruction composée**.
- juste après, le texte est indenté.

Une **instruction composée** se compose :

Une **instruction composée** se compose :

- d'une ligne introduite par un mot clé déterminé et se terminant par 2 points (:)

Une **instruction composée** se compose :

- d'une ligne introduite par un mot clé déterminé et se terminant par 2 points (:)
- d'un bloc d'instructions indenté (de 4 espaces). Dans un éditeur digne de ce nom, l'indentation se fait toute seule.

Une **instruction composée** se compose :

- d'une ligne introduite par un mot clé déterminé et se terminant par 2 points (:)
- d'un bloc d'instructions indenté (de 4 espaces). Dans un éditeur digne de ce nom, l'indentation se fait toute seule.

Usage du if

```
a = 5
if a > 0 :
        print("a est positif")
        print("Na !")
print("FIN")
```

- Dans le code précédent la fin du bloc qui commence après le **if** et les *2 points*, se termine quand l'indentation se termine.

- Dans le code précédent la fin du bloc qui commence après le **if** et les *2 points*, se termine quand l'indentation se termine.
- la dernière ligne est donc ici toujours exécutée

- Dans le code précédent la fin du bloc qui commence après le **if** et les *2 points*, se termine quand l'indentation se termine.
- la dernière ligne est donc ici toujours exécutée

Usage du if

```
a = -2
if a > 0 :
        print("a est positif")
        print("Na !")
print("FIN")
```

- Dans le code précédent la fin du bloc qui commence après le **if** et les *2 points*, se termine quand l'indentation se termine.
- la dernière ligne est donc ici toujours exécutée

Usage du if

```
a = -2
if a > 0 :
        print("a est positif")
        print("Na !")
print("FIN")
```

renvoie

- Dans le code précédent la fin du bloc qui commence après le **if** et les *2 points*, se termine quand l'indentation se termine.
- la dernière ligne est donc ici toujours exécutée

Usage du if

```
a = -2
if a > 0 :
        print("a est positif")
        print("Na !")
print("FIN")
```

renvoie

Titre du bloc

```
FIN
```

Le jeu énoncé précédemment pourrait être (n est calculé avant)

Patie du jeu

```
p = int(input("Propose un nombre : "))
if p > n :
    print("Trop grand")
if p < n :
    print("Trop petit")
if p == n :
    print("Gagné !")
```

Qu'est ce qui ne convient pas dans ce code qui est censé faire la même chose que précédemment ?

Qu'est ce qui ne convient pas dans ce code qui est censé faire la même chose que précédemment ?

Cherchez l'erreur

```
p = int(input("Propose un nombre : "))
if p > n :
    print("Trop grand")
    if p < n :
        print("Trop petit")
        if p == n :
            print("Gagné !")
```

Qu'est ce qui ne convient pas dans ce code qui est censé faire la même chose que précédemment ?

Cherchez l'erreur

```
p = int(input("Propose un nombre : "))
if p > n :
    print("Trop grand")
    if p < n :
        print("Trop petit")
        if p == n :
            print("Gagné !")
```

Il est clair que le programme n'affichera jamais "Trop petit" et "Gagné", quelque soit le nombre entré p

1 Contrôle du flux d'exécution
- Contrôle avec if
- Contrôle avec if et else
- Contrôle avec if, elif et else

2 Prédicats et opérateurs booléens

3 Les boucles

- On peut raccourcir une succession de **if** avec **if** et **else**.

- On peut raccourcir une succession de **if** avec **if** et **else**.
- **else** signifie *sinon*. C'est le cas par défaut si la condition exprimée dans le **if** n'est pas vérifiée.

- On peut raccourcir une succession de **if** avec **if** et **else**.
- **else** signifie *sinon*. C'est le cas par défaut si la condition exprimée dans le **if** n'est pas vérifiée.

Exemple : *n* est déjà calculé

```
p = int(input("Propose un nombre : "))
if p == n :
    print("Gagné")
else :
    print("Raté !")
```

- On peut raccourcir une succession de **if** avec **if** et **else**.
- **else** signifie *sinon*. C'est le cas par défaut si la condition exprimée dans le **if** n'est pas vérifiée.

Exemple : *n* est déjà calculé

```
p = int(input("Propose un nombre : "))
if p == n :
    print("Gagné")
else :
    print("Raté !")
```

Remarque : **else** est au même niveau (indentation) que **if**, et **else** est suivi de 2 points (**:**)

Attention : la réunion des cas traités par le **if** et ceux traités par la clause **else** doit contenir l'ensemble des cas possibles !

Attention : la réunion des cas traités par le **if** et ceux traités par la clause **else** doit contenir l'ensemble des cas possibles !

Par exemple, ce code ne traite pas tous les cas :

Attention : la réunion des cas traités par le **if** et ceux traités par la clause **else** doit contenir l'ensemble des cas possibles !

Par exemple, ce code ne traite pas tous les cas :

Des cas non traités ! *n* déjà calculé

```
p = int(input("Propose un nombre : "))
if p > n :
    print("Trop grand")
else :
    print("Trop petit")
```

Attention : la réunion des cas traités par le **if** et ceux traités par la clause **else** doit contenir l'ensemble des cas possibles !

Par exemple, ce code ne traite pas tous les cas :

Des cas non traités ! *n* déjà calculé

```
p = int(input("Propose un nombre : "))
if p > n :
    print("Trop grand")
else :
    print("Trop petit")
```

Et quand c'est gagné ? ? ?

1 Contrôle du flux d'exécution
- Contrôle avec if
- Contrôle avec if et else
- Contrôle avec if, elif et else

2 Prédicats et opérateurs booléens

3 Les boucles

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$).

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$). On doit décomposer désormais le problème de la façon suivante grâce au nouveau mot clé **elif** :

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$). On doit décomposer désormais le problème de la façon suivante grâce au nouveau mot clé **elif** :

si $p > n$, alors on affiche "Trop grand"

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$). On doit décomposer désormais le problème de la façon suivante grâce au nouveau mot clé **elif** :

si $p > n$, alors on affiche "Trop grand"

sinon si $p < n$, alors on affiche "Trop petit"

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$). On doit décomposer désormais le problème de la façon suivante grâce au nouveau mot clé **elif** :

si $p > n$, alors on affiche "Trop grand"

sinon si $p < n$, alors on affiche "Trop petit"

sinon c'est gagné !

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$). On doit décomposer désormais le problème de la façon suivante grâce au nouveau mot clé **elif** :

si $p > n$, alors on affiche "Trop grand"

sinon si $p < n$, alors on affiche "Trop petit"

sinon c'est gagné !

Correspondance langage de réalisation et *Python*

Il est clair que dans l'exemple précédent, on a oublié de traiter le cas où l'utilisateur a gagné ($p = n$). On doit décomposer désormais le problème de la façon suivante grâce au nouveau mot clé **elif** :

si $p > n$, alors on affiche "Trop grand"

sinon si $p < n$, alors on affiche "Trop petit"

sinon c'est gagné !

Correspondance langage de réalisation et *Python*

$$\text{si} \leftrightarrow \text{if}$$

$$\text{sinon si} \leftrightarrow \text{elif}$$

$$\text{sinon} \leftrightarrow \text{else}$$

Le bon code :

Le bon code :

Bien traité...

```
p = int(input("Propose un nombre : ")
if p > n :
    print("Trop grand")
elif p < n :
    print("Trop petit")
else :
    print("Gagné !")
```

Attention : la réunion des cas traités par le **if**, le **elif** et le **else** doit contenir l'ensemble des cas possibles !

Attention : la réunion des cas traités par le **if**, le **elif** et le **else** doit contenir l'ensemble des cas possibles !

Par exemple, ce code ne traite pas tous les cas :

Attention : la réunion des cas traités par le **if**, le **elif** et le **else** doit contenir l'ensemble des cas possibles !

Par exemple, ce code ne traite pas tous les cas :

note, une variable qui correspond à une note d'examen

```
if note > 10 :
    print("Passable")
elif note > 12 :
    print("Assez bien")
elif note > 14 :
    print("Bien")
elif note > 16 :
    print("Très bien")
else :
    print("Refusé")
```

1 Contrôle du flux d'exécution

2 Prédicats et opérateurs booléens
- Notion de prédicat
- Avec les mots-clés and, or et not

3 Les boucles

Vous avez déjà rencontré des opérateurs de comparaison dans le cours ou TD.

Opérateurs	Significations
$<$	Strictement supérieur à
$>$	Strictement inférieur à
$<=$	Inférieur ou égal à
$>=$	Supérieur ou égal à
$==$	Égal à
$!=$	Différent de

Vous avez déjà rencontré des opérateurs de comparaison dans le cours ou TD.

Opérateurs	Significations
$<$	Strictement supérieur à
$>$	Strictement inférieur à
$<=$	Inférieur ou égal à
$>=$	Supérieur ou égal à
$==$	Égal à
$!=$	Différent de

Nous les avons utilisé avec **if** et **else** pour l'instant.

Les conditions qui se trouvent entre **if** et les deux points (:) sont appelés des **prédicats**.

Les conditions qui se trouvent entre **if** et les deux points (:) sont appelés des **prédicats**. Ces prédicats sont vrais ou faux selon le cas. C'est-à-dire qu'ils renvoient **True** ou **False**.

Les conditions qui se trouvent entre **if** et les deux points (:) sont appelés des **prédicats**. Ces prédicats sont vrais ou faux selon le cas. C'est-à-dire qu'ils renvoient **True** ou **False**.
Les 2 programmes ci-dessous sont équivalents

prog 1

```
age = 14
if age < 18 :
    print("Mineur")
```

prog 2

```
age = 14
if age < 18 == True :
    print("Mineur")
```

$a = 5$ est une affectation : on affecte à a la valeur 5.

$a = 5$ est une affectation : on affecte à a la valeur 5.
Par contre $a == 5$ signifie "est-ce que a est égal à 5 ?".

$a = 5$ est une affectation : on affecte à a la valeur 5. Par contre $a == 5$ signifie "est-ce que a est égal à 5 ?". Cette question renvoie *True* ou *False*.

$a = 5$ est une affectation : on affecte à a la valeur 5.
Par contre $a == 5$ signifie "est-ce que a est égal à 5 ?". Cette question renvoie *True* ou *False*.
Avec une variable booléenne, on peut raccourcir l'écriture :

prog 1 : etat est une variable booléenne

```
if etat == True :
    print("C'est vrai !")
```

prog 2 : équivaut à prog 1

```
if etat :
    print("C'est vrai !")
```

1 Contrôle du flux d'exécution

2 Prédicats et opérateurs booléens
- Notion de prédicat
- Avec les mots-clés and, or et not

3 Les boucles

- Il arrive souvent de devoir tester plusieurs prédicats, par exemple quand on cherche à vérifier si une variable quelconque, de type entier par exemple, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres).

- Il arrive souvent de devoir tester plusieurs prédicats, par exemple quand on cherche à vérifier si une variable quelconque, de type entier par exemple, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres).
- Le mot clé **and** (et) permet de tester simultanément plusieurs conditions.

- Il arrive souvent de devoir tester plusieurs prédicats, par exemple quand on cherche à vérifier si une variable quelconque, de type entier par exemple, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres).
- Le mot clé **and** (et) permet de tester simultanément plusieurs conditions.

Exemple :

Avec and

```
if note > 10 and note <= 20 :
    print("Vous avez plus de 10 !")
```

- Il arrive souvent de devoir tester plusieurs prédicats, par exemple quand on cherche à vérifier si une variable quelconque, de type entier par exemple, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres).
- Le mot clé **and** (et) permet de tester simultanément plusieurs conditions.

Exemple :

Avec and

```
if note > 10 and note <= 20 :
    print("Vous avez plus de 10 !")
```

Les 2 conditions doivent être réunies pour afficher la phrase.

Le mot-clé **and** a pour but de relier deux ou plusieurs conditions qui doivent être vraies simultanément. la ligne :

Le mot-clé **and** a pour but de relier deux ou plusieurs conditions qui doivent être vraies simultanément. la ligne :

if d > 5 and d <= 10 :

Le mot-clé **and** a pour but de relier deux ou plusieurs conditions qui doivent être vraies simultanément. la ligne :

if d > 5 and d <= 10 :

signifie que

Le mot-clé **and** a pour but de relier deux ou plusieurs conditions qui doivent être vraies simultanément. la ligne :

if d > 5 and d <= 10 :

signifie que

si (d est supérieure à 5) et (inférieure ou égale à 10), alors

Le mot-clé **or** a pour but de discerner deux conditions.

Le mot-clé **or** a pour but de discerner deux conditions.la ligne

if res == 1 or res == 6 :

Le mot-clé **or** a pour but de discerner deux conditions.la ligne

if res == 1 or res == 6 :

signifie que

si (res est égal à 1) ou (res est égal à 6), alors

Le **ou** (**or**) en programmation est un "ou inclusif", comme en mathématique, c'est-à-dire que les conditions qui doivent être vérifiées sont

Le **ou** (**or**) en programmation est un "ou inclusif", comme en mathématique, c'est-à-dire que les conditions qui doivent être vérifiées sont

- la première,

Le **ou** (**or**) en programmation est un "ou inclusif", comme en mathématique, c'est-à-dire que les conditions qui doivent être vérifiées sont

- la première,
- ou la deuxième,

Le **ou** (**or**) en programmation est un "ou inclusif", comme en mathématique, c'est-à-dire que les conditions qui doivent être vérifiées sont

- la première,
- ou la deuxième,
- ou les deux à la fois.

Le **ou** (**or**) en programmation est un "ou inclusif", comme en mathématique, c'est-à-dire que les conditions qui doivent être vérifiées sont

- la première,
- ou la deuxième,
- ou les deux à la fois.

Par exemple le code suivant affiche "GAGNE !" si l'entier n est divisible par 3 ou s'il est divisible par 5 . "GAGNE !" sera affiché pour $n = 6$, pour $n = 10$, pour $n = 15$ mais pas pour $n = 14$.

Avec or : *n* est un entier

```
if n % 3 == 0 or n % 5 == 0 :
    print("GAGNÉ !")
```

Avec or : *n* est un entier

```
if n % 3 == 0 or n % 5 == 0 :
    print("GAGNÉ !")
```

On peut bien sûr combiner ces 2 mots-clés :

Avec or : *n* est un entier

```
if n % 3 == 0 or n % 5 == 0 :
    print("GAGNÉ !")
```

On peut bien sûr combiner ces 2 mots-clés :

Avec and et or : *n* un entier

```
if (n > 7 and n <= 9) or n ==3 :
    print("Bravo !")
```

Notez l'usage des parenthèses !

Avec or : *n* est un entier

```
if n % 3 == 0 or n % 5 == 0 :
    print("GAGNÉ !")
```

On peut bien sûr combiner ces 2 mots-clés :

Avec and et or : *n* un entier

```
if (n > 7 and n <= 9) or n ==3 :
    print("Bravo !")
```

Notez l'usage des parenthèses !
Que s'affiche-t-il pour $n = 3$? $n = 7$? $n = 8$? $n = 11$?

Avec and et or : *n* un entier

```
if n > 7 and (n <= 9 or n ==3) :
    print("Bravo !")
```

Notez le déplacement des parenthèses !

Avec and et or : *n* un entier

```
if n > 7 and (n <= 9 or n ==3) :
    print("Bravo !")
```

Notez le déplacement des parenthèses !

Que s'affiche-t-il pour $n = 3$? $n = 7$? $n = 8$? $n = 11$?

Avec and et or : *n* un entier

```
if n > 7 and (n <= 9 or n ==3) :
    print("Bravo !")
```

Notez le déplacement des parenthèses !
Que s'affiche-t-il pour $n = 3$? $n = 7$? $n = 8$? $n = 11$?

Exercice : pour quel *n* entier a-t-on "bravo" qui s'affiche ?

```
if (n > 5 or n < 3) and (n < 7 and n > 4) :
    print("bravo")
```

Le mot-clé **not** permet de prendre la négation d'un prédicat (qui renvoie donc **True** ou **False**).

Le mot-clé **not** permet de prendre la négation d'un prédicat (qui renvoie donc **True** ou **False**).

Avec not : que s'affiche-t-il ici ?

```
etat = True
if not etat :
    print("Bon")
else :
    print("Mauvais")
```

Le mot-clé **not** permet de prendre la négation d'un prédicat (qui renvoie donc **True** ou **False**).

Avec not : que s'affiche-t-il ici ?

```
etat = True
if not etat :
    print("Bon")
else :
    print("Mauvais")
```

Évidemment, **not** se combine très bien avec **and** et **or**

Que va-t-il s'afficher dans ce programme ?

Que va-t-il s'afficher dans ce programme ?

Exercice

```
n = 6
if n > 7 and n <= 9 :
    print("UN")
if not(n > 5) :
    print("DEUX")
else :
    print("TROIS")
```

Que va-t-il s'afficher dans ce programme ?

Exercice

```
n = 6
if n > 7 and n <= 9 :
    print("UN")
if not(n > 5) :
    print("DEUX")
else :
    print("TROIS")
```

Réponse :

Que va-t-il s'afficher dans ce programme ?

Exercice

```
n = 6
if n > 7 and n <= 9 :
    print("UN")
if not(n > 5) :
    print("DEUX")
else :
    print("TROIS")
```

Réponse : TROIS bien sûr !

Que va-t-il s'afficher dans ce programme ?

Exercice

```
n = 6
if n > 7 and n <= 9 :
    print("UN")
if not(n > 5) :
    print("DEUX")
else :
    print("TROIS")
```

Réponse : TROIS bien sûr ! Dans le 1^e if, la condition n'est pas vérifiée, dans le 2^e if non plus et c'est donc la clause else qui est exécutée.

1 Contrôle du flux d'exécution

2 Prédicats et opérateurs booléens

3 Les boucles
- Avec while
- Avec for
- enumerate, continue

1 Contrôle du flux d'exécution

2 Prédicats et opérateurs booléens

3 Les boucles
- Avec while
- Avec for
- enumerate, continue

- Les tâches répétitives sont lassantes pour l'humain.

- Les tâches répétitives sont lassantes pour l'humain.
- L'ordinateur les exécute très bien et facilement au travers de fonctionnalités logicielles ou bien simplement au travers d'un langage comme *Python*.

- Les tâches répétitives sont lassantes pour l'humain.
- L'ordinateur les exécute très bien et facilement au travers de fonctionnalités logicielles ou bien simplement au travers d'un langage comme *Python*.
- l'*instruction composée* **while**, qui signifie **tant que**, est là pour cela.

- Les tâches répétitives sont lassantes pour l'humain.
- L'ordinateur les exécute très bien et facilement au travers de fonctionnalités logicielles ou bien simplement au travers d'un langage comme *Python*.
- l'*instruction composée* **while**, qui signifie **tant que**, est là pour cela. La syntaxe est la suivante :

- Les tâches répétitives sont lassantes pour l'humain.
- L'ordinateur les exécute très bien et facilement au travers de fonctionnalités logicielles ou bien simplement au travers d'un langage comme *Python*.
- l'*instruction composée* **while**, qui signifie **tant que**, est là pour cela. La syntaxe est la suivante :

Usage de while

```
while condition :
    instruction 1
    instruction 2
    ...
```

- Comme dans toute instruction composée, l'identation du bloc qui débute après les 2 points (:) est obligatoire.

- Comme dans toute instruction composée, l'identation du bloc qui débute après les 2 points (:) est obligatoire.
- ATTENTION : pour ne pas avoir une boucle qui s'exécute indéfiniment, la condition (située après le **while**) doit à un moment évoluer.

- Comme dans toute instruction composée, l'identation du bloc qui débute après les 2 points (:) est obligatoire.
- ATTENTION : pour ne pas avoir une boucle qui s'exécute indéfiniment, la condition (située après le **while**) doit à un moment évoluer.
- Pour pallier au 2^e point, dans l'usage de **while**, est associé souvent un compteur qui évolue dans la boucle.

- Comme dans toute instruction composée, l'identation du bloc qui débute après les 2 points (:) est obligatoire.
- ATTENTION : pour ne pas avoir une boucle qui s'exécute indéfiniment, la condition (située après le **while**) doit à un moment évoluer.
- Pour pallier au 2^e point, dans l'usage de **while**, est associé souvent un compteur qui évolue dans la boucle.

Usage de while : boucle infinie ...

```
n = 3
while n < 7 :        # condition toujours vraie !
    print("coucou")
```

Usage de while : boucle finie.

```
n = 3
while n < 7 :
    print("coucou")
    n = n + 1        # ou n += 1
```

Usage de while : boucle finie.

```
n = 3
while n < 7 :
    print("coucou")
    n = n + 1        # ou n += 1
```

À chaque passage dans le boucle, *n* s'incrémente de 1, de fait, la boucle va s'exécuter 4 fois, et passe à la suite du programme.

Usage de while : boucle finie.

```
n = 3
while n < 7 :
    print("coucou")
    n = n + 1        # ou n += 1
```

À chaque passage dans le boucle, *n* s'incrémente de 1, de fait, la boucle va s'exécuter 4 fois, et passe à la suite du programme.
le mot "coucou" sera donc écrit à l'écran 4 fois, avec $4 = 7 - 3$ (en fait $6 - 3 + 1$)

Exemple

```
i = 0
while i < 5 :
    print(i, "est plus petit que 5")
    i += 1
print("Fin")
```

Exemple

```
i = 0
while i < 5 :
    print(i, "est plus petit que 5")
    i += 1
print("Fin")
```

- la boucle s'exécute 5 fois, de 0 à 4.

Exemple

```
i = 0
while i < 5 :
        print(i, "est plus petit que 5")
        i += 1
print("Fin")
```

- la boucle s'exécute 5 fois, de 0 à 4.
- le bloc d'instruction qui suit le **while** n'a que 2 lignes : conséquence

Exemple

```
i = 0
while i < 5 :
        print(i, "est plus petit que 5")
        i += 1
print("Fin")
```

- la boucle s'exécute 5 fois, de 0 à 4.
- le bloc d'instruction qui suit le **while** n'a que 2 lignes : conséquence
- Le mot "Fin" n'est écrit qu'une seule fois, à la fin, quand la boucle a fini de s'exécuter.

Autre exemple : que fait ce programme ?

```
statut, age = False, 16
while not statut :
    print("âge :", age, "ans, mineur")
    age += 1
    if age == 18 :
        statut = not statut
print("Enfin 18 ans, donc majeur")
```

Autre exemple : que fait ce programme ?

```
statut, age = False, 16
while not statut :
    print("âge :", age, "ans, mineur")
    age += 1
    if age == 18 :
        statut = not statut
print("Enfin 18 ans, donc majeur")
```

Réponse :

Autre exemple : que fait ce programme ?

```
statut, age = False, 16
while not statut :
    print("âge :", age, "ans, mineur")
    age += 1
    if age == 18 :
        statut = not statut
print("Enfin 18 ans, donc majeur")
```

Réponse : il affiche ...

âge : 16 ans, mineur

âge : 17 ans, mineur

Enfin 18 ans, donc majeur

- L'instruction **break** permet de forcer la sortie d'une boucle.

- L'instruction **break** permet de forcer la sortie d'une boucle.
- C'est-à-dire, si *Python* rencontre dans une boucle cette instruction, il sort de la boucle et continue immédiatement après la fin du bloc constituant cette boucle.

- L'instruction **break** permet de forcer la sortie d'une boucle.
- C'est-à-dire, si *Python* rencontre dans une boucle cette instruction, il sort de la boucle et continue immédiatement après la fin du bloc constituant cette boucle.

Usage de break : $n = 2$ au départ

```
while True :
        print(n)
        n += 1
        if n % 5 == 0 :
                break
print("fin")
```

Emboîtement de boucles :

Emboîtement de boucles : Il est bien sûr possible de concevoir des boucles à l'intérieur d'autres boucles. Exemple :

Emboîtement de boucles : Il est bien sûr possible de concevoir des boucles à l'intérieur d'autres boucles. Exemple :

Deux boucles imbriquées : $n = 3$ au départ

```
while n < 5 :
    m = 5
    while m < 7 :
        print("n =", n, ", m =", m)
        m += 1
    n += 1
```

Emboîtement de boucles : Il est bien sûr possible de concevoir des boucles à l'intérieur d'autres boucles. Exemple :

Deux boucles imbriquées : $n = 3$ au départ

```
while n < 5 :
    m = 5
    while m < 7 :
        print("n =", n, ", m =", m)
        m += 1
    n += 1
```

affiche :

n = 3 , m = 5
n = 3 , m = 6
n = 4 , m = 5
n = 4 , m = 6

Qu'affiche ce programme ? ($n = 1$ et $m = 1$ au départ)

```
while n <= 5 :
    while m <= n :
        print("*", end = "")
        m += 1
    m, n = 1, n + 1
    print()                 # seul, provoque un saut de ligne
```

Qu'affiche ce programme ? ($n = 1$ et $m = 1$ au départ)

```
while n <= 5 :
    while m <= n :
        print("*", end = "")
        m += 1
    m, n = 1, n + 1
    print()                 # seul, provoque un saut de ligne
```

Réponse :

Qu'affiche ce programme ? ($n = 1$ et $m = 1$ au départ)

```
while n <= 5 :
    while m <= n :
        print("*", end = "")
        m += 1
    m, n = 1, n + 1
    print()                 # seul, provoque un saut de ligne
```

Réponse : il affiche

```
*
**
***
****
*****
```

1 Contrôle du flux d'exécution

2 Prédicats et opérateurs booléens

3 Les boucles
- Avec while
- Avec for
- enumerate, continue

- On peut aussi construire des boucles avec l'instruction **for** qui signifie **pour** ...

- On peut aussi construire des boucles avec l'instruction **for** qui signifie **pour** ...
- **for** est souvent utilisé pour l'énumération des éléments d'un *itérable* (plage de nombres, chaîne de caractères, liste, ...)

- On peut aussi construire des boucles avec l'instruction **for** qui signifie **pour** ...
- **for** est souvent utilisé pour l'énumération des éléments d'un *itérable* (plage de nombres, chaîne de caractères, liste, ...)
- L'usage de for est lié à range() quand il s'agit de plage de nombres.

- On peut aussi construire des boucles avec l'instruction **for** qui signifie **pour** ...
- **for** est souvent utilisé pour l'énumération des éléments d'un *itérable* (plage de nombres, chaîne de caractères, liste, ...)
- L'usage de for est lié à range() quand il s'agit de plage de nombres.
- Avec **for**, l'utilité d'un compteur est inutile, le nombre de fois où la boucle est parcourue est a priori déterminé à l'avance.

- On peut aussi construire des boucles avec l'instruction **for** qui signifie **pour** ...
- **for** est souvent utilisé pour l'énumération des éléments d'un *itérable* (plage de nombres, chaîne de caractères, liste, ...)
- L'usage de for est lié à range() quand il s'agit de plage de nombres.
- Avec **for**, l'utilité d'un compteur est inutile, le nombre de fois où la boucle est parcourue est a priori déterminé à l'avance.

Commençons par l'usage de range().

Voici d'abord la syntaxe de range()

Voici d'abord la syntaxe de range()

range(deb, fin, pas)

défini une plage de d'entiers de **deb** à **fin-1** par pas de **pas**.

Voici d'abord la syntaxe de range()

range(deb, fin, pas)

défini une plage de d'entiers de **deb** à **fin-1** par pas de **pas**.

- si **deb** n'est pas précisé, la plage commence à 0

Voici d'abord la syntaxe de range()

range(deb, fin, pas)

défini une plage de d'entiers de **deb** à **fin-1** par pas de **pas**.

- si **deb** n'est pas précisé, la plage commence à 0
- si **pas** n'est pas précisé, le pas est de 1 par défaut

Voici d'abord la syntaxe de range()

range(deb, fin, pas)

défini une plage de d'entiers de **deb** à **fin-1** par pas de **pas**.

- si **deb** n'est pas précisé, la plage commence à 0
- si **pas** n'est pas précisé, le pas est de 1 par défaut
- le type rendu est **range** qui est un *itérable* (ie : que l'on peut parcourir)

Voici d'abord la syntaxe de range()

range(deb, fin, pas)

défini une plage de d'entiers de **deb** à **fin-1** par pas de **pas**.

- si **deb** n'est pas précisé, la plage commence à 0
- si **pas** n'est pas précisé, le pas est de 1 par défaut
- le type rendu est **range** qui est un *itérable* (ie : que l'on peut parcourir)

Ainsi,

for k in range(deb, fin) :

signifie : pour tous les k de la plage d'entiers de deb à $fin - 1$:...

range(fin)

```
for k in range(4) :
    print(k, end="")
```

range(fin)

```
for k in range(4) :
    print(k, end="")
```

affiche : 0 1 2 3

range(fin)

```
for k in range(4) :
    print(k, end="")
```

affiche : 0 1 2 3

range(deb, fin)

```
for k in range(3, 8) :
    print(k, end="")
```

range(fin)

```
for k in range(4) :
    print(k, end="")
```

affiche : 0 1 2 3

range(deb, fin)

```
for k in range(3, 8) :
    print(k, end="")
```

affiche : 3 4 5 6 7

range(fin)

```
for k in range(4) :
    print(k, end="")
```

affiche : 0 1 2 3

range(deb, fin)

```
for k in range(3, 8) :
    print(k, end="")
```

affiche : 3 4 5 6 7

range(deb, fin, pas)

```
for k in range(2, 13, 3) :
    print(k, end="")
```

range(fin)

```
for k in range(4) :
    print(k, end="")
```

affiche : 0 1 2 3

range(deb, fin)

```
for k in range(3, 8) :
    print(k, end="")
```

affiche : 3 4 5 6 7

range(deb, fin, pas)

```
for k in range(2, 13, 3) :
    print(k, end="")
```

affiche : 2 5 8 11

Dans la cas d'une chaîne de caractères :

Dans la cas d'une chaîne de caractères :

- **for** énumère chacune des lettres de la chaîne.

Dans la cas d'une chaîne de caractères :

- **for** énumère chacune des lettres de la chaîne.

for et chaîne

```
mot = "lapin"
for l in mot :
    print("lettre", l, end = ", ")
```

Dans la cas d'une chaîne de caractères :

- **for** énumère chacune des lettres de la chaîne.

for et chaîne

```
mot = "lapin"
for l in mot :
    print("lettre", l, end = ", ")
```

affiche : lettre l, lettre a, lettre p, lettre i, lettre n,

Dans la cas d'une chaîne de caractères :

- **for** énumère chacune des lettres de la chaîne.

for et chaîne

```
mot = "lapin"
for l in mot :
    print("lettre", l, end = ", ")
```

affiche : lettre l, lettre a, lettre p, lettre i, lettre n,

- Le comportement de **for** avec une *liste* (nous le verrons plus tard) est le même, les lettres sont remplacées par les éléments de la liste.

1 Contrôle du flux d'exécution

2 Prédicats et opérateurs booléens

3 Les boucles
- Avec while
- Avec for
- enumerate, continue

- *Python* propose une option encore plus performante pour à la fois numéroter et énumérer les lettres d'un mot, d'une plage de nombres, les éléments d'une liste, etc.

- *Python* propose une option encore plus performante pour à la fois numéroter et énumérer les lettres d'un mot, d'une plage de nombres, les éléments d'une liste, etc.
- Cela se fait avec **enumerate()** qui s'emploie comme cela :

- *Python* propose une option encore plus performante pour à la fois numéroter et énumérer les lettres d'un mot, d'une plage de nombres, les éléments d'une liste, etc.
- Cela se fait avec **enumerate()** qui s'emploie comme cela :

avec enumerate()

```
p = "anaconda"
for ind, lettre in enumerate(mot) :
    print("la lettre", lettre, "a pour indice", ind)
```

- *Python* propose une option encore plus performante pour à la fois numéroter et énumérer les lettres d'un mot, d'une plage de nombres, les éléments d'une liste, etc.
- Cela se fait avec **enumerate()** qui s'emploie comme cela :

avec enumerate()

```
p = "anaconda"
for ind, lettre in enumerate(mot) :
        print("la lettre", lettre, "a pour indice", ind)
```

affiche :
la lettre a a pour indice 0
la lettre n a pour indice 1
la lettre a a pour indice 2
etc.

 Remarquez qu'avec **enumerate()**, on récupère en 1^{e} l'*indice* et ensuite l'*élément*.

—

 Remarquez qu'avec **enumerate()**, on récupère en 1^{e} l'*indice* et ensuite l'*élément*.

—

Qu'affiche le programme suivant ?

Remarquez qu'avec **enumerate()**, on récupère en 1^{e} l'*indice* et ensuite l'*élément*.

—

Qu'affiche le programme suivant ?

Boucle

```
for i in range(3) :
    j = 2
    while j < 4 :
        print(i * j, end = " $$ ")
        j += 1
```

 Remarquez qu'avec **enumerate()**, on récupère en 1^e l'*indice* et ensuite l'*élément*.

—

Qu'affiche le programme suivant ?

Boucle

```
for i in range(3) :
    j = 2
    while j < 4 :
        print(i * j, end = " $$ ")
        j += 1
```

affiche : 0 $$ 0 $$ 2 $$ 3 $$ 4 $$ 6

Et celui-ci ?

Et celui-ci ?

Boucle

```
for i in range(1,11) :
    for j in range(1, 11) :
        print(i, "x", j, "=", i*j)
```

Et celui-ci ?

Boucle

```
for i in range(1,11) :
    for j in range(1, 11) :
        print(i, "x", j, "=", i*j)
```

affiche : les tables de multiplication de 1 à 10.

Et celui-ci ?

Boucle

```
for i in range(1,11) :
    for j in range(1, 11) :
        print(i, "x", j, "=", i*j)
```

affiche : les tables de multiplication de 1 à 10.

1 x 1 = 1

1 x 2 = 2

1 x 3 = 3

etc.

10 x 9 = 90

10 x 10 = 100

Il se peut parfois que se dresse la necessité d'ignorer des cas lors d'une boucle.

Il se peut parfois que se dresse la necessité d'ignorer des cas lors d'une boucle. Imaginez que vous vouliez afficher les nombres entiers de 10 à 30 en ignorant les multiples de 5 et les multiples de 7.

Il se peut parfois que se dresse la necessité d'ignorer des cas lors d'une boucle. Imaginez que vous vouliez afficher les nombres entiers de 10 à 30 en ignorant les multiples de 5 et les multiples de 7. Voici comment l'on peut s'y prendre avec l'instruction **continue**

Il se peut parfois que se dresse la necessité d'ignorer des cas lors d'une boucle. Imaginez que vous vouliez afficher les nombres entiers de 10 à 30 en ignorant les multiples de 5 et les multiples de 7. Voici comment l'on peut s'y prendre avec l'instruction **continue**

continue

```
for nb in range(10,31) :
    if nb % 5 == 0 or nb % 7 == 0 :
        continue
    print("nombre :", nb)
```

Il se peut parfois que se dresse la necessité d'ignorer des cas lors d'une boucle. Imaginez que vous vouliez afficher les nombres entiers de 10 à 30 en ignorant les multiples de 5 et les multiples de 7. Voici comment l'on peut s'y prendre avec l'instruction **continue**

continue

```
for nb in range(10,31) :
    if nb % 5 == 0 or nb % 7 == 0 :
        continue
    print("nombre :", nb)
```

continue arrête l'exécution de la boucle et reprend à l'itération suivante. Dès que *nb* sera égal à 5, **continue** renvoie au cas suivant et donc relance la suite de la boucle...

En conclusion :

En conclusion :
Le rôle de ces boucles est d'itérer (de répéter) un bloc d'instructions,

En conclusion :
Le rôle de ces boucles est d'itérer (de répéter) un bloc d'instructions,

- soit un nombre précis de fois, c'est la boucle **for**

En conclusion :
Le rôle de ces boucles est d'itérer (de répéter) un bloc d'instructions,

- soit un nombre précis de fois, c'est la boucle **for**
- soit relativement à une condition, c'est la boucle **while**

En conclusion :
Le rôle de ces boucles est d'itérer (de répéter) un bloc d'instructions,

- soit un nombre précis de fois, c'est la boucle **for**
- soit relativement à une condition, c'est la boucle **while**

Toutes ces boucles peuvent être interrompues dans leur exécution grâce à l'instruction **break**.

En conclusion :
Le rôle de ces boucles est d'itérer (de répéter) un bloc d'instructions,

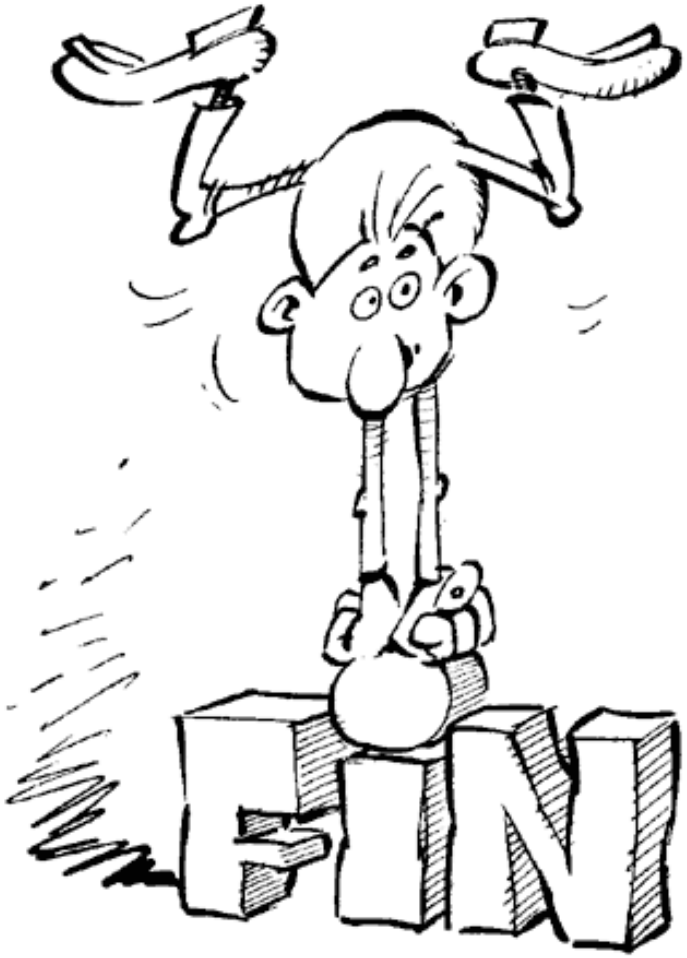
- soit un nombre précis de fois, c'est la boucle **for**
- soit relativement à une condition, c'est la boucle **while**

Toutes ces boucles peuvent être interrompues dans leur exécution grâce à l'instruction **break**.
Les boucles peuvent être imbriquées (les unes dans les autres).

En conclusion :
Le rôle de ces boucles est d'itérer (de répéter) un bloc d'instructions,

- soit un nombre précis de fois, c'est la boucle **for**
- soit relativement à une condition, c'est la boucle **while**

Toutes ces boucles peuvent être interrompues dans leur exécution grâce à l'instruction **break**.
Les boucles peuvent être imbriquées (les unes dans les autres).
Pour passer à l'itération suivante d'une boucle (et ainsi ignorer un cas), on utilise l'instruction **continue**


FIN