On découvre

- comment exécuter certaines instructions et pas d'autres dans certaines situations
- ce qu'est une instruction conditionnelle

1 Pourquoi les instructions conditionnelles ?

Jusqu'à présent on a exécuté les instructions ligne par ligne, que ce soit dans le corps des fonctions des semaines 2 ou 3 ou dans les blocs de code dans les exercices de TD. On dit que les instructions sont exécutées *en séquence*.

Dans certaines situations, on peut vouloir n'exécuter une suite d'instructions que si une condition est vérifiée.

Par exemple : on peut établir que le nombre de jours du mois de février est 28 puis ajouter que dans le cas où l'année est bissextile, ce nombre de jours est 27. On dira aussi "si l'année est bissextile". Ou encore, "*à condition* que l'année soit bissextile".

Il nous faut pouvoir identifier :

- la condition : ce sera une expression booléenne du type "l'année est bissextile, vrai ou faux" ?
- la ou les instructions à exécuter : ce sera un bloc d'instructions

Dans le chapitre précédent, on a utilisé les booléens comme un type de données, au même titre que les entiers ou les flottants. À partir de ce chapitre on utilisera les booléens comme conditions de contrôle de l'exécution des instructions.

2 Instruction `if` : syntaxe

Les conditionnelles Python commencent par le mot clé `if`, suivi de la condition (une expression booléenne), suivi du caractère `:` qui termine la première ligne. On trouve ensuite le bloc d'instructions, indenté[1].

La syntaxe est la suivante :

```
if <expr_cond>:
    <bloc_instr>
```

Par exemple, en supposant que la variable `annee` est associée en mémoire à une valeur entière positive :

```
jours_mois = 28
jours_annee = 365
if est_bissextile(annee):
    jours_mois = 29
    jours_annee = 366
```

La fin du bloc de code est marquée par la reprise de l'indentation précédente. On pourrait écrire:

```
jours_mois = 28
if est_bissextile(annee) :
    jours_mois = 29
heures_mois = 24 * jours_mois
```

3 Sémantique de l'instruction `if`, flot d'exécution

Le **flot d'exécution** d'un bloc d'instructions est l'ordre dans lequel les instructions sont exécutées. Quand on exécute des instructions de tête, on suit intuitivement avec son doigt le flot d'instructions[2]. Le débogueur permet de détailler en couleurs le flot d'instructions.

Ce flot d'exécution suit les instructions d'un bloc en séquence, c'est-à-dire les unes après les autres[3].

[1] On retrouve le principe d'indentation d'un bloc d'instructions constituant le corps d'une fonction, ainsi que le caractère `:` de fin de ligne.
[2] Merci de ne pas mettre vos doigts sur les écrans des ordinateurs…
[3] Nous avons néanmoins vu qu'un appel de fonction permet 1.- d'interrompre la séquence pour exécuter les instructions de la fonction et, 2.- de reprendre ensuite l'exécution là où elle avait été laissée.

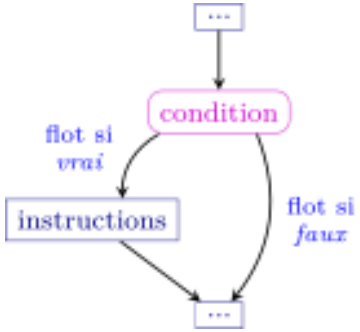
Université de Lille
Instruction conditionnelle
2025-2026
UE PROG, L1S1 MI

L'instruction conditionnelle `if` permet de potentiellement modifier ce flot d'exécution.

Considérons le code suivant :

```
...
if <condition> :
    <instructions>
...
# après le if
```

Le flot d'exécution correspondant peut être illustré par le graphe :



Dans l'ordre :

1. la condition est évaluée. On obtient une valeur booléenne.
2. si cette valeur vaut *vrai* (`True`):
   - les `<instructions>` associées au `if` sont exécutées
   - avant de poursuivre l'exécution des instructions qui suivent l'instruction conditionnelle `if` (`# après le if`).
3. si cette valeur vaut *faux* (`False`):
   - on passe directement à l'exécution des instructions qui suivent l'instruction conditionnelle `if` (`# après le if`).

Prenons l'exemple suivant, en supposant que la variable `annee` est associée en mémoire à une valeur entière positive:

```
jours_mois = 28               # 1
jours_annee = 365             # 2
if est_bissextile(annee):     # 3
    jours_mois = 29           # 4
    jours_annee = 366         # 5
heures_mois = 24 * jours_mois # 6
```

Supposons que `annee` vaut 2000 : `est_bissextile(annee)` vaut `True`. Le flot d'exécution est alors le suivant :

- exécution de la ligne 1 : `jours_mois` vaut 28 en mémoire
- exécution de la ligne 2 : `jours_annee` vaut 365 en mémoire
- exécution de la ligne 3 : évaluation de `est_bissextile(annee)` qui vaut `True`
- exécution de la ligne 4 : `jours_mois` vaut 29 en mémoire
- exécution de la ligne 5 : `jours_annee` vaut 366 en mémoire
- exécution de la ligne 6 : `heures_mois` vaut 696 (`24 * 29`) en mémoire

À la fin de l'exécution la mémoire contient les associations :

`annee`	2000
`jours_mois`	29
`jours_annee`	366

Département Informatique de la Faculté des Sciences et Technologies

heures_mois	696

Supposons que `annee` vaut `2019` : `est_bissextile(annee)` vaut `False`. Le flot d'exécution est alors le suivant :

- exécution de la ligne 1 : `jours_mois` vaut `28` en mémoire
- exécution de la ligne 2 : `jours_annee` vaut `365` en mémoire
- exécution de la ligne 3 : évaluation de `est_bissextile(annee)` qui vaut `False`
- exécution de la ligne 6 : `heures_mois` vaut `672` (`24 * 28`) en mémoire

À la fin de l'exécution la mémoire contient les associations :

annee	2019
jours_mois	28
jours_annee	365
heures_mois	672

NB : Pour les étudiant · es qui ont déjà programmé, on remarquera qu'il est tout à fait légitime

4 Imbrication des blocs d'instructions

Le bloc d'instructions d'un `if` peut lui-même comporter d'autres instructions conditionnelles.

Nous avons alors besoin de blocs d'instructions *imbriqués* dont l'imbrication est contrôlée par l'indentation:

```
instruction bloc 1
    instruction bloc 2
    instruction bloc 2
        instruction bloc 3
    instruction bloc 2
instruction bloc 1
instruction bloc 1
```

L'indentation des instructions doit donc être soignée.

Considérons le code suivant, en supposant que `mois` est associée à une variable de type `str` en mémoire :

```
if mois == "février" :
    jours_mois = 28
    if est_bissextile(annee) :
        jours_mois = 29
    heures_mois = 24 * jours_mois
```

L'affectation à `heures_mois` fait partie du bloc d'instructions associé au `if mois == "février"`, elle ne sera donc exécutée que si `mois` vaut `"février"`.

Le code suivant diffère très peu du précédent :

```
if mois == "février" :
    jours_mois = 28
    if est_bissextile(annee) :
        jours_mois = 29
heures_mois = 24 * jours_mois
```

L'affectation à `heures_mois` suit en séquence la conditionnelle `if mois == "février"`. Elle sera donc exécutée quelle que soit la valeur de `mois`.

Attention Considérons ce code :


```
if est_bissextile(annee):
    jours_mois = 29 # 1
heures_mois = 24 * jours_mois # 2
```

Ce code est syntaxiquement correct mais il produit une erreur de nom à l'exécution dans le cas où `annee` n'est pas bissextile.

En effet, si `annee` est bissextile la ligne `1` est exécutée, et la mémoire contient la valeur `29` associée à la variable `jours_mois`. Quand la ligne `2` est exécutée, la valeur de `jours_mois` est bien présente en mémoire.

Mais si `annee` n'est PAS bissextile, la ligne `1` n'est pas exécutée. Donc `jours_mois` n'est pas présent en mémoire et l'exécution de la ligne `2` déclenche une erreur de nom.

5 Bonnes pratiques

On n'écrira pas `x == True` mais `x`.

On n'écrira pas `x == False` mais `not x`.

Pour ne pas être tenté, on fera attention à choisir des noms de variables porteurs de sens. Il est plus facile de lire : `if est_fini` que `if x`.

6 Memento

- l'instruction conditionnelle `if` permet de n'exécuter un bloc d'instructions que si une condition est vraie
- les instructions conditionnelles peuvent être imbriquées
- l'indentation définit la manière dont les instructions seront exécutées. Le soin à apporter à cette indentation est capital.

On découvre

- comment exécuter certaines parties d'un programme dans certaines situations, et d'autres parties du programme dans les autres situations
- qu'une alternative peut être ajoutée à une instruction conditionnelle

1 Pourquoi les alternatives ?

Il existe des situations où l'on peut vouloir exécuter un bloc d'instructions si une condition est vérifiée (elle vaut vrai), et exécuter un autre bloc d'instructions si cette même condition n'est pas vérifiée (elle vaut faux).

Par exemple : on peut définir le nombre de jours du mois de février à 29 dans le cas où l'année est bissextile, et à 28 dans les autres cas. Ou encore : si l'année est bissextile alors le nombre de jours vaut 29, sinon il vaut 28.

Il nous faut pouvoir identifier

- la condition : ce sera une valeur booléenne
- la ou les instructions à exécuter quand la condition est vraie : ce sera un bloc d'instructions
- la ou les instructions à exécuter quand la condition n'est pas vraie (est fausse) : ce sera un autre bloc d'instructions

2 Syntaxe de l'alternative : Mot-clé `else`

En Python on utilise le mot clé `else` en association avec le mot clé `if` que nous connaissons.

La syntaxe est la suivante :

```
if <expr_cond>:
    <bloc_instr_if>
else:
    <bloc_instr_else>
```

À noter :

- le "si alors sinon" de la langue parlée est traduit en "si sinon";
- la ligne comportant le mot clé `else` est indentée au même niveau que celle du mot clé `if`, et se termine par un caractère `:`;
- la fin des blocs de code est marquée par la reprise de l'indentation précédente.

Par exemple, en supposant que la variable `annee` est associée en mémoire à une valeur entière positive :

```
if est_bissextile(annee):
    jours_mois = 29
    jours_annee = 366
else:
    jours_mois = 28
    jours_annee = 365
heures_mois = 24 * jours_mois
```

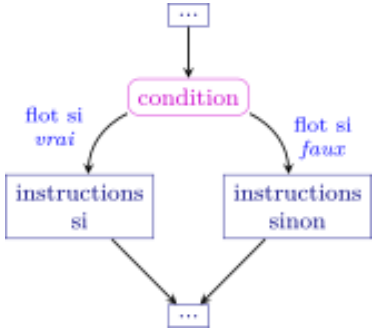
NB: certain · es étudiant · es veulent à tout prix écrire un `else` alors qu'on a vu dans le support précédent que dans certains cas un `if` sans `else` convient tout à fait. Ils ou elles cherchent alors à écrire quelque chose comme "`else` rien", avec du code plus ou moins affreux. **Quand il n'y a pas besoin d'un `else` on n'en met pas**.

3 Sémantique du `else`, flot d'exécution

Le flot d'exécution correspondant au code suivant :

```
...
if condition :
    instructions si
else :
    instructions sinon
...
# après le if
```

peut être illustré par :



L'exécution consistera, dans l'ordre, à

1. la condition est évaluée, on obtient une valeur booléenne.
2. Si cette valeur vaut *vrai* (`True`)
   - les instructions associées au `if` sont exécutées,
   - avant de poursuivre l'exécution des instructions qui suivent l'instruction conditionnelle (`# après le if`)
3. Si cette valeur vaut *faux* (`False`)
   - les instructions associées au `else` sont exécutées,
   - avant de poursuivre l'exécution des instructions qui suivent l'instruction conditionnelle (`# après le if`)

3.1 Exemple

Prenons l'exemple suivant, en supposant que la variable `annee` est associée en mémoire à une valeur entière positive:

```
if est_bissextile(annee):     # 1
    jours_mois = 29           # 2
    jours_annee = 366         # 3
else:                         # 4
    jours_mois = 28           # 5
    jours_annee = 365         # 6
heures_mois = 24 * jours_mois # 7
```

Supposons que `annee` vaut 2000 : `est_bissextile(annee)` vaut `True`. Le flot d'exécution est alors le suivant :

- exécution de la ligne 1 : évaluation de `est_bissextile(annee)` qui vaut `True`
- exécution de la ligne 2 : `jours_mois` vaut 29 en mémoire
- exécution de la ligne 3 : `jours_annee` vaut 366 en mémoire
- exécution de la ligne 7 : `heures_mois` vaut 696 (`24 * 29`) en mémoire

À la fin de l'exécution la mémoire contient les associations :

`annee`	2000
`jours_mois`	29
`jours_annee`	366
`heures_mois`	696

Supposons que `annee` vaut 2019 : `est_bissextile(annee)` vaut `False`. Le flot d'exécution est alors le suivant :

Université de Lille
Instruction conditionnelle et alternative
2025-2026
UE PROG, L1S1 MI
Département Informatique de la Faculté des Sciences et Technologies

- exécution de la ligne 1 : évaluation de `est_bissextile(annee)` qui vaut `False`
- exécution de la ligne 5 : `jours_mois` vaut 28 en mémoire
- exécution de la ligne 6 : `jours_annee` vaut 365 en mémoire
- exécution de la ligne 7 : `heures_mois` vaut 672 (`24 * 28`) en mémoire

À la fin de l'exécution la mémoire contient les associations :

`annee`	2019
`jours_mois`	28
`jours_annee`	365
`heures_mois`	672

3.2 Un peu de logique

Les instructions du bloc *sinon* sont exécutées uniquement dans le cas où la condition est fausse.

Donc quand on exécute les instructions du bloc *sinon* on est sûr que la négation de la condition vaut vrai.

Dans l'exemple ci-dessus, quand les instructions 5 et 6 sont exécutées, on est sûr que `not est_bissextile(annee)` vaut vrai.

4 Imbrication des blocs d'instructions

Les blocs d'instructions conditionnels et alternatifs peuvent eux-mêmes contenir des instructions conditionnelles.

Considérons le code suivant, en supposant que la variable `mois` est associée en mémoire à la valeur `"janvier"`, `"février"` ou `"mars"`:

```
if mois == "janvier" or mois == "mars":
    jours_mois = 31
else:  # else 1) mois vaut "février"
    if est_bissextile(annee):
        jours_mois = 29
    else: # else 2) mois vaut "février" et not est_bissextile(annee) vaut vrai
        jours_mois = 28
heures_mois = 24 * jours_mois
```

Pour comprendre qu'au niveau de la ligne `else 1)` (le premier `else`) on sait que `mois` vaut `"février"`, il faut se rappeler que `mois == "janvier" or mois == "mars"` vaut `False` donc `mois` ne vaut ni `"janvier"` ni `"mars"`. De plus on a dit que `mois` vaut soit `"janvier"`, soit `"février"` soit `"mars"`. Donc `mois` vaut nécessairement `"février"`.

Au niveau de la ligne `else 2)` (le 2nd `else`), `mois` vaut `"février"` car ce bloc est sous la portée du premier `else`. De plus la condition `est_bissextile(annee)` vaut `False`.

Bien évidemment, la bonne indentation des instructions est essentielle. Comparez les deux codes :

```
if mois == "février":
    jours_mois = 28
    if est_bissextile(annee) :
        jours_mois = 29
else : # janvier ou mars
    jours_mois = 31
heures_mois = 24 * jours_mois
```

et

```
# version erronée
if mois == "février" :
    jours_mois = 28
    if est_bissextile(annee) :
        jours_mois = 29
    else : # ce n'est plus janvier ou mars mais non est_bissextile
        jours_mois = 31
heures_mois = 24 * jours_mois
```

Dans ce dernier exemple, le `else` de la ligne 6 se rapporte au `if` de la ligne 4, et ne correspond donc pas aux mois de janvier ou mars, mais aux mois de février des années non bissextiles…

5 Bonnes pratiques

5.1 Ce qu'on n'écrira pas...

On pourrait être tenté d'utiliser une conditionnelle là où des opérateurs booléens suffisent. On n'écrira pas:

```
def est_pair(n:int) -> bool:
    if n%2==0:
        return True
    else:
        return False
```

Le corps de la fonction sera avantageusement remplacé par:

```
    return n%2 == 0
```

On n'écrira pas `else: x = x`. S'il n'y a pas besoin de code associé à une condition valant `False`, on utilise un `if` sans `else`. L'absence de `else` n'est pas une erreur de syntaxe.

5.2 Mais quel code choisir ?

Considérons les codes suivants précédemment décrits :

```
jours_mois = 28             # version 1
if est_bissextile(annee):
    jours_mois = 29
```

et

```
if est_bissextile(annee):   # version 2
    jours_mois = 29
else:
    jours_mois = 28
```

Si vous avez bien compris la sémantique du `if` et du `else`, vous aurez constaté qu'après l'exécution de ces 2 codes les valeurs de `jours_mois` en mémoire sont les mêmes.

Cette troisième version produit la même valeur pour `jours_mois` avec des instructions conditionnelles en séquence:

```
if est_bissextile(annee):       # version 3
    jours_mois = 29
if not est_bissextile(annee):
    jours_mois = 28
```

Alors, quelle version préférer ?

L'inconvénient de la version 3 est qu'on ne saisit pas nécessairement en la lisant que les 2 affectations sont exclusives, alors que ça saute aux yeux dans la version 2. De plus l'expression `est_bissextile(annee)` est écrite et évaluée 2 fois systématiquement.

La version 1 utilise une valeur initialisée puis modifiée, qui peut ne pas être claire à la première lecture[1].

[1] On l'a utilisée car elle était bien pratique pour les besoins du cours.

Université de Lille
Instruction conditionnelle et alternative
2025-2026
UE PROG, L1S1 MI
Département Informatique de la Faculté des Sciences et Technologies

On découvre

- comment exécuter certaines parties d'un programme dans différentes situations
- que plusieurs alternatives peuvent être énumérées dans une instruction conditionnelle

1 Énumération d'alternatives

Il existe des séries de situations pour lesquelles on peut vouloir exécuter différents blocs d'instructions quand différentes conditions sont vérifiées.

Par exemple, nous souhaitons définir le nombre de jours du mois pour le premier semestre civil, de janvier à juin (pas le premier semestre de la L1). Ce nombre peut être 31 pour les mois de janvier, mars et mai, 30 pour les mois d'avril et juin, 28 ou 29 jours pour le mois de février.

Il nous faut pouvoir identifier :

- une première condition : ce sera une expression booléenne
- la ou les instructions à exécuter quand cette condition est vraie : ce sera un bloc d'instructions
- une deuxième condition, toujours une expression booléenne
- la ou les instructions à exécuter quand cette condition est vraie : ce sera un autre bloc d'instructions
- … autant d'autres conditions que nécessaire
- … autant d'autres blocs d'instructions associés à ces conditions
- éventuellement la ou les instructions à exécuter quand aucune de ces conditions n'est vraie : ce sera un dernier bloc d'instructions

2 Mot-clé `elif` : syntaxe

En Python on utilise le mot clé `elif` (pour *else if*), en association avec le mot clé `if` que nous connaissons.

La syntaxe est la suivante:

```
...
if <expr_cond>:
    <bloc_instr_si>
elif <condition_sinon_si1>:
    <bloc_instr_sinon_si1>
elif <condition_sinon_si2>:
    <bloc_instr_sinon2_si2>
...
else:
    <bloc_instr_sinon_sinon>
...
```

Attention On peut utiliser des `elif` sans `else`. La partie `else` est optionnelle.

On écrira par exemple, en supposant que la variable `mois` est associée en mémoire à la valeur `"janvier"`, `"février"`, `"mars"`, `"avril"`, `"mai"` ou `"juin"`:

```
if mois == "janvier" or mois == "mars" or mois == "mai":
    jours_mois = 31
elif mois == "avril" or mois == "juin":
    jours_mois = 30
else:  # février
    if est_bissextile(annee) :
        jours_mois = 29
    else:
        jours_mois = 28
heures_mois = 24 * jours_mois
```


Notez la construction identique du `if` et du `elif` avec une condition suivie du caractère `:` pour introduire un bloc d'instructions à exécuter dans le cas où la condition est vraie.

Notez aussi que les lignes comportant les mot-clés `if`, `elif` et `else` sont indentées au même niveau.

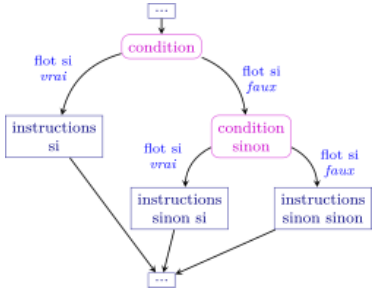
Comme toujours, la fin des blocs de code est marquée par la reprise de l'indentation précédente.

3 Sémantique du `elif`, flot d'exécution

Le flot d'exécution correspondant au code suivant :

```
...
if condition :
    bloc d'instructions si
elif condition sinon :
    bloc d'instructions sinon si
else :
    bloc d'instructions sinon sinon
...
# après le if
```

peut être illustré par :



Le flot d'exécution passera par *un et un seul* des blocs d'instructions.

L'exécution consistera, dans l'ordre

- la première condition est évaluée. On obtient une valeur booléenne.
- Si cette valeur vaut *vrai*
  - les instructions associées au `if` sont exécutées
  - avant de poursuivre l'exécution des instructions qui suivent l'instruction `if` (`# après le if`)
- Si cette valeur vaut *faux*, la deuxième condition est évaluée. On obtient une nouvelle valeur booléenne.
- Si cette nouvelle valeur vaut *vrai*
  - les instructions associées au `elif` sont exécutées
  - avant de poursuivre l'exécution des instructions qui suivent l'instruction `if` (`# après le if`)
- Les conditions suivantes sont évaluées tant qu'aucune d'entre elles ne vaut *vrai*. Si aucune d'entre elles ne vaut *vrai*
  - les instructions associées au `else` sont exécutées
  - avant de poursuivre l'exécution des instructions qui suivent l'instruction `if` (`# après le if`).

3.1 Exemple

Considérons le code suivant:

```
if mois == "janvier" or mois == "mars" or mois == "mai":
```

```
    jours_mois = 31
elif mois == "avril" or mois == "juin":
    jours_mois = 30
else:  # février
    if est_bissextile(annee) :
        jours_mois = 29
    else:
        jours_mois = 28
heures_mois = 24 * jours_mois
```

Supposons que `mois` vaut `"mars"`. Le flot d'exécution est alors le suivant:

- exécution de la ligne 1 : la condition est évaluée à la valeur `True`
- exécution de la ligne 2: `jours_mois` vaut `31` en mémoire
- exécution de la ligne 10: `heures_mois` vaut `744` (`24 * 31`) en mémoire

Supposons que `mois` vaut `"juin"`. Le flot d'exécution est alors le suivant:

- exécution de la ligne 1 : la condition est évaluée à la valeur `False`
- exécution de la ligne 3 : la condition est évaluée à la valeur `True`
- exécution de la ligne 4: `jours_mois` vaut `30` en mémoire
- exécution de la ligne 10: `heures_mois` vaut `720` (`24 * 30`) en mémoire

Supposons que `mois` vaut `"février"`. Le flot d'exécution est alors le suivant:

- exécution de la ligne 1 : la condition est évaluée à la valeur `False`
- exécution de la ligne 3 : la condition est évaluée à la valeur `False`
- exécution de la ligne 6 : la condition est évaluée à la valeur `True` ou `False`
- selon cette valeur, exécution des lignes 7 ou 9
- exécution de la ligne 10

3.2 Un peu de logique

Les différentes conditions (expressions booléennes) sont évaluées du haut vers le bas, et la suivante est évaluée si seulement si la précédente a été évaluée à faux.

Quand on évalue une condition, on sait donc que la conjonction des conditions précédentes (c'est-à-dire les conditions précédentes reliées par l'opérateur `and`) vaut faux.

```
...
if <expr_cond>:
    <bloc_instr_si>
elif <condition_sinon_si1>: # ici on sait que <expr_cond> vaut faux
    <bloc_instr_sinon_si1>
elif <condition_sinon_si2>: # ici on sait que <expr_cond> and <condition_sinon_si1> vaut faux
    <bloc_instr_sinon2_si2>
...
else: # ici on sait que <expr_cond> and <condition_sinon_si1>
      # and <condition_sinon_si2> and .... vaut faux
    <bloc_instr_sinon_sinon>
...
```

4 Ordre des conditions

Un seul des blocs d'instructions est exécuté, celui qui correspond à la première condition évaluée à vrai.

L'ordre dans lequel les différentes conditions apparaissent est donc important. Les parties `elif` ne sont pas interchangeables, on ne peut pas changer leur ordre sans modifier le comportement du code.

Considérons l'exemple suivant.

Les catégories en tir à l'arc sont définies comme suit :

- les moins de 10 ans inclus sont en catégorie *poussin*
- les sportif · ves entre 10 ans exclus et 12 ans inclus sont en catégorie *benjamin*
- entre 12 ans exclus et 14 ans inclus iels sont en catégorie *minime*
- entre 14 ans exclus et 17 ans inclus iels sont en catégorie *cadet*
- entre 17 ans exclus et 20 ans inclus iels sont en catégorie *junior*
- après 20 ans exclus iels sont en catégorie *senior*

Le code suivant reflète la spécification, avec une suite de `if`.

```
if age <= 10:
    categorie = "poussin"
if 10 < age and age <= 12:
    categorie = "benjamin"
if 12 < age and age <= 14:
    categorie = "minime"
if 14 < age and age <= 17:
    categorie = "cadet"
if 17 < age and age <= 20:
    categorie = "junior"
if age > 20:
    categorie = "senior"
```

Comme dit précédemment, on ne voit pas que les différents comportements sont exclusifs les uns des autres, et chaque condition sera évaluée. De plus on pourrait être tenté de remplacer le dernier `if` par un `else` associé au `if 17 < age and age <= 20:`, ce qui donnerait un code incorrect :

```
...
if 17 < age and age <= 20:
    categorie = "junior"
else:
    categorie = "senior"
```

Pour `age` valant `10`, la valeur `"poussin"` serait dans un premier temps associée à la variable `categorie`, puis cette valeur serait écrasée en mémoire par la valeur `"senior"` car la condition `if 17 < age and age <= 20` est fausse et donc l'instruction `categorie = "senior"` est exécutée.

L'utilisation d'une construction `elif` avec un *ordre des conditions bien choisi* peut permettre de simplifier les conditions :

```
if age <= 10:
    categorie = "poussin"
elif age <= 12:
    categorie = "benjamin"
elif age <= 14:
    categorie = "minime"
elif age <= 17:
    categorie = "cadet"
elif age <= 20:
    categorie = "junior"
else:
    categorie = "senior"
```

Ici le `else` est correct, il sera exécuté si la conjonction des conditions précédentes est fausse, donc si `age > 10 and age > 12 and age > 14 and age > 17 and age > 20` vaut vrai, ce qui est équivalent à `age > 20` vaut vrai.

Attention, un changement dans l'ordre de ces conditions ne permettra plus d'avoir la bonne catégorie en fonction de l'âge.

Supposons qu'on échange les 2 premières conditions :

```
if age <= 12:
    categorie = "benjamin"
```

```
elif age <= 10: # ici on sait que age > 12
    categorie = "poussin"
...
```

On voit le piège : la condition `age <= 10` peut être remplacée par `False` car `age > 12 and age <= 10` est équivalent à `False`. Pour `age` valant `9`, la condition `age <= 12` vaut vrai et la valeur de `categorie` sera `"benjamin"`, ce qui n'est pas ce qu'on veut.

Il faut donc bien écrire les conditions qui représentent les plus petits ensembles de valeurs en premier (ex : `age <= 10` décrit un ensemble inclus dans l'ensemble décrit par `age <= 12`). Sinon les conditions qui représentent des ensembles plus grands capteront le comportement associé aux conditions "plus petites".

5 Memento

- la construction `if` … `elif` … `else` permet d'exécuter différents blocs d'instructions en fonction de différentes conditions
- les conditions d'une construction `if` … `elif` … `else` sont énumérées tant que l'une d'elles n'est pas vraie
- un seul des blocs d'instructions d'une construction `if` … `elif` … `else` est exécuté