Chapitre 4

Les structures conditionnelles

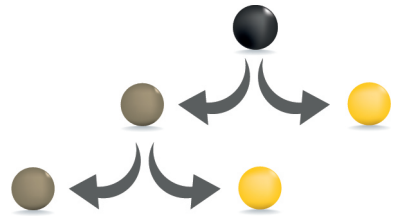
Difficulté :

Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous saisissiez dans la console. Mais nos programmes seraient bien pauvres si nous ne pouvions, de temps à autre, demander à exécuter certaines instructions dans un cas, et d'autres instructions dans un autre cas.

Dans ce chapitre, je vais vous parler des structures conditionnelles, qui vont vous permettre de faire des tests et d'aller plus loin dans la programmation.

Les conditions permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.

Vous finirez ce chapitre en créant votre premier « vrai » programme : même si vous ne pensez pas encore pouvoir faire quelque chose de très consistant, à la fin de ce chapitre vous aurez assez de matière pour coder un petit programme dans un but très précis.



Vos premières conditions et blocs d'instructions

Forme minimale en `if`

Les conditions sont un concept essentiel en programmation (oui oui, je me répète à force mais il faut avouer que des concepts essentiels, on n'a pas fini d'en voir). Elles vont vous permettre de faire une action précise si, par exemple, une variable est positive, une autre action si cette variable est négative, ou une troisième action si la variable est nulle. Comme un bon exemple vaut mieux que plusieurs lignes d'explications, voici un exemple clair d'une condition prise sous sa forme la plus simple.

Dès à présent dans mes exemples, j'utiliserai des commentaires. Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le code (car, vous vous en rendrez compte, relire ses programmes après plusieurs semaines d'abandon, sans commentaire, ce peut être parfois plus qu'ardu). En Python, un commentaire débute par un dièse (« # ») et se termine par un saut de ligne. Tout ce qui est compris entre ce # et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (on place le # en début de ligne) ou une partie seulement, après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement).

Cela étant posé, revenons à nos conditions :

```
1 >>> # Premier exemple de condition
2 >>> a = 5
3 >>> if a > 0: # Si a est supérieur à 0
4 ...     print("a est supérieur à 0.")
5 ...
6 a est supérieur à 0.
7 >>>
```

Détaillons ce code, ligne par ligne :

1. La première ligne est un commentaire décrivant qu'il s'agit du premier test de condition. Elle est ignorée par l'interpréteur et sert juste à vous renseigner sur le code qui va suivre.
2. Cette ligne, vous devriez la comprendre sans aucune aide. On se contente d'affecter la valeur 5 à la variable `a`.
3. Ici se trouve notre test conditionnel. Il se compose, dans l'ordre :
   - du mot clé `if` qui signifie « si » en anglais ;
   - de la condition proprement dite, `a > 0`, qu'il est facile de lire (une liste des opérateurs autorisés pour la comparaison sera présentée plus bas) ;
   - du signe deux points, « : », qui termine la condition et est indispensable : Python affichera une erreur de syntaxe si vous l'omettez.

4. Ici se trouve l'instruction à exécuter dans le cas où `a` est supérieur à `0`. Après que vous ayez appuyé sur `Entrée` à la fin de la ligne précédente, l'interpréteur vous présente la série de trois points qui signifie qu'il attend la saisie du **bloc d'instructions** concerné avant de l'interpréter. Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée, c'est-à-dire décalée vers la droite. Des explications supplémentaires seront données un peu plus bas sur les indentations.
5. L'interpréteur vous affiche à nouveau la série de trois points et vous pouvez en profiter pour saisir une nouvelle instruction dans ce bloc d'instructions. Ce n'est pas le cas pour l'instant. Vous appuyez donc sur `Entrée` sans avoir rien écrit et l'interpréteur vous affiche le message « a est supérieur à 0 », ce qui est assez logique vu que `a` est effectivement supérieur à `0`.

Il y a deux notions importantes sur lesquelles je dois à présent revenir, elles sont complémentaires ne vous en faites pas.

La première est celle de bloc d'instructions. On entend par bloc d'instructions une série d'instructions qui s'exécutent dans un cas précis (par condition, comme on vient de le voir, par répétition, comme on le verra plus tard...). Ici, notre bloc n'est constitué que d'une seule instruction (la ligne 4 qui fait appel à `print`). Mais rien ne vous empêche de mettre plusieurs instructions dans ce bloc.

```
1 a = 5
2 b = 8
3 if a > 0:
4     # On incrémente la valeur de b
5     b += 1
6     # On affiche les valeurs des variables
7     print("a =",a,"et b =",b)
```

La seconde notion importante est celle d'indentation. On entend par indentation un certain décalage vers la droite, obtenu par un (ou plusieurs) espaces ou tabulations.

Les indentations sont essentielles pour Python. Il ne s'agit pas, comme dans d'autres langages tels que le C++ ou le Java, d'un confort de lecture mais bien d'un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc.

Forme complète (`if`, `elif` et `else`)

Les limites de la condition simple en `if`

La première forme de condition que l'on vient de voir est pratique mais assez incomplète.

Considérons, par exemple, une variable `a` de type entier. On souhaite faire une action si cette variable est positive et une action différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :

```
>>> a = 5
>>> if a > 0: # Si a est positif
...     print("a est positif.")
... if a < 0: # a est négatif
...     print("a est négatif.")
```

Amusez-vous à changer la valeur de `a` et exécutez à chaque fois les conditions ; vous obtiendrez des messages différents, sauf si `a` est égal à 0. En effet, aucune action n'a été prévue si a vaut 0.

Cette méthode n'est pas optimale, tout d'abord parce qu'elle nous oblige à écrire deux conditions séparées pour tester une même variable. De plus, et même si c'est dur à concevoir par cet exemple, dans le cas où la variable remplirait les deux conditions (ici c'est impossible bien entendu), les deux portions de code s'exécuteraient.

La condition `if` est donc bien pratique mais insuffisante.

L'instruction `else` :

Le mot-clé `else`, qui signifie « sinon » en anglais, permet de définir une première forme de complément à notre instruction `if`.

```
>>> age = 21
>>> if age >= 18: # Si age est supérieur ou égal à 18
...     print("Vous êtes majeur.")
... else: # Sinon (age inférieur à 18)
... print("Vous êtes mineur.")
```

Je pense que cet exemple suffit amplement à exposer l'utilisation de `else`. La seule subtilité est de bien se rendre compte que Python exécute soit l'un, soit l'autre, et jamais les deux. Notez que cette instruction `else` doit se trouver au même niveau d'indentation que l'instruction `if` qu'elle complète. De plus, elle se termine également par deux points puisqu'il s'agit d'une condition, même si elle est sous-entendue.

L'exemple de tout à l'heure pourrait donc se présenter comme suit, avec l'utilisation de `else` :

```
>>> a = 5
>>> if a > 0:
...     print("a est supérieur à 0.")
... else:
...     print("a est inférieur ou égal à 0.")
```

Mais... le résultat n'est pas tout à fait le même, si ?

Non, en effet. Vous vous rendrez compte que, cette fois, le cas où `a` vaut `0` est bien pris en compte. En effet, la condition initiale prévoit d'exécuter le premier bloc d'instructions si `a` est strictement supérieur à `0`. Sinon, on exécute le second bloc d'instructions.

Si l'on veut faire la différence entre les nombres positifs, négatifs et nuls, il va falloir utiliser une condition intermédiaire.

L'instruction `elif` :

Le mot clé `elif` est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons juste de voir, l'idéal serait d'écrire :

– si `a` est strictement supérieur à `0`, on dit qu'il est positif ;
– sinon si `a` est strictement inférieur à `0`, on dit qu'il est négatif ;
– sinon, (`a` ne peut qu'être égal à `0`), on dit alors que `a` est nul.

Traduit en langage Python, cela donne :

```
>>> if a > 0: # Positif
...     print("a est positif.")
... elif a < 0: # Négatif
...     print("a est négatif.")
... else: # Nul
    print("a est nul.")
```

De même que le `else`, le `elif` est sur le même niveau d'indentation que le `if` initial. Il se termine aussi par deux points. Cependant, entre le `elif` et les deux points se trouve une nouvelle condition. Linéairement, le schéma d'exécution se traduit comme suit :

1. On regarde si `a` est strictement supérieur à `0`. Si c'est le cas, on affiche « a est positif » et on s'arrête là.
2. Sinon, on regarde si `a` est strictement inférieur à `0`. Si c'est le cas, on affiche « a est négatif » et on s'arrête.
3. Sinon, on affiche « a est nul ».

Attention : quand je dis « on s'arrête », il va de soi que c'est uniquement pour cette condition. S'il y a du code après les trois blocs d'instructions, il sera exécuté dans tous les cas.

Vous pouvez mettre autant de `elif` que vous voulez après une condition en `if`. Tout comme le `else`, cette instruction est facultative et, quand bien même vous construiriez une instruction en `if`, `elif`, vous n'êtes pas du tout obligé de prévoir un `else` après. En revanche, l'instruction `else` ne peut figurer qu'une fois, clôturant le bloc de la condition. Deux instructions `else` dans une même condition ne sont pas envisageables et n'auraient de toute façon aucun sens.

Sachez qu'il est heureusement possible d'imbriquer des conditions et, dans ce cas, l'indentation permet de comprendre clairement le schéma d'exécution du programme. Je vous laisse essayer cette possibilité, je ne vais pas tout faire à votre place non plus :)

De nouveaux opérateurs

Les opérateurs de comparaison

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison**. Je vais les présenter très brièvement, vous laissant l'initiative de faire des tests car ils ne sont réellement pas difficiles à comprendre.

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Attention : l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ». Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

Prédicats et booléens

Avant d'aller plus loin, sachez que les conditions qui se trouvent, par exemple, entre `if` et les deux points sont appelés des **prédicats**. Vous pouvez tester ces prédicats directement dans l'interpréteur pour comprendre les explications qui vont suivre.

```
>>> a = 0
>>> a == 5
False
>>> a > -8
True
>>> a != 33.19
True
>>>
```

L'interpréteur renvoie tantôt `True` (c'est-à-dire « vrai »), tantôt `False` (c'est-à-dire « faux »).

`True` et `False` sont les deux valeurs possibles d'un type que nous n'avons pas vu jusqu'ici : le type booléen (bool).

N'oubliez pas que `True` et `False` sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrire `True` sans un 'T' majuscule, Python ne va pas comprendre.

Les variables de ce type ne peuvent prendre comme valeur que vrai ou faux et peuvent être pratiques, justement, pour stocker des prédicats, de la façon que nous avons vue ou d'une façon plus détournée.

```
>>> age = 21
>>> majeur = False
>>> if age >= 18:
>>>     majeur = True
>>>
```

À la fin de cet exemple, majeur vaut `True`, c'est-à-dire « vrai », si l'âge est supérieur ou égal à 18. Sinon, il continue de valoir `False`. Les booléens ne vous semblent peut-être pas très utiles pour l'instant mais vous verrez qu'ils rendent de grands services !

Les mots-clés `and`, `or` et `not`

Il arrive souvent que nos conditions doivent tester plusieurs prédicats, par exemple quand l'on cherche à vérifier si une variable quelconque, de type entier, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres). Avec nos méthodes actuelles, le plus simple serait d'écrire :

```
# On fait un test pour savoir si a est comprise dans l'
    intervalle allant de 2 à 8 inclus
a = 5
if a >= 2:
    if a <= 8:
        print("a est dans l'intervalle.")
    else:
        print("a n'est pas dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Cela marche mais c'est assez lourd, d'autant que, pour être sûr qu'un message soit affiché à chaque fois, il faut fermer chacune des deux conditions à l'aide d'un `else` (la seconde étant imbriquée dans la première). Si vous avez du mal à comprendre cet exemple, prenez le temps de le décortiquer, ligne par ligne, il n'y a rien que de très simple.

Il existe cependant le mot clé `and` (qui signifie « et » en anglais) qui va nous rendre ici un fier service. En effet, on cherche à tester à la fois si `a` est supérieur ou égal à 2 et inférieur ou égal à 8. On peut donc réduire ainsi les conditions imbriquées :

```
if a>=2 and a<=8:
    print("a est dans l'intervalle.")
```

```
3 else:
4     print("a n'est pas dans l'intervalle.")
```

Simple et bien plus compréhensible, avouez-le.

Sur le même mode, il existe le mot clé `or` qui signifie cette fois « ou ». Nous allons prendre le même exemple, sauf que nous allons évaluer notre condition différemment.

Nous allons chercher à savoir si `a` n'est pas dans l'intervalle. La variable ne se trouve pas dans l'intervalle si elle est inférieure à 2 ou supérieure à 8. Voici donc le code :

```
1 if a<2 or a>8:
2     print("a n'est pas dans l'intervalle.")
3 else:
4     print("a est dans l'intervalle.")
```

Enfin, il existe le mot clé `not` qui « inverse » un prédicat. Le prédicat `not a==5` équivaut donc à `a!=5`.

`not` rend la syntaxe plus claire. Pour cet exemple, j'ajoute à la liste un nouveau mot clé, `is`, qui teste l'égalité non pas des valeurs de deux variables, mais de leurs références. Je ne vais pas rentrer dans le détail de ce mécanisme avant longtemps. Il vous suffit de savoir que pour les entiers, les flottants et les booléens, c'est strictement la même chose. Mais pour tester une égalité entre variables dont le type est plus complexe, préférez l'opérateur « == ». Revenons à cette démonstration :

```
1 >>> majeur = False
2 >>> if majeur is not True:
3 ...     print("Vous n'êtes pas encore majeur.")
4 ...
5 Vous n'êtes pas encore majeur.
6 >>>
```

Si vous parlez un minimum l'anglais, ce prédicat est limpide et d'une simplicité sans égale.

Vous pouvez tester des prédicats plus complexes de la même façon que les précédents, en les saisissant directement, sans le `if` ni les deux points, dans l'interpréteur de commande. Vous pouvez utiliser les parenthèses ouvrantes et fermantes pour encadrer des prédicats et les comparer suivant des priorités bien précises (nous verrons ce point plus loin, si vous n'en comprenez pas l'utilité).

Votre premier programme !

À quoi on joue ?

L'heure du premier TP est venue. Comme il s'agit du tout premier, et parce qu'il y a quelques indications que je dois vous donner pour que vous parveniez jusqu'au bout, je vous accompagnerai pas à pas dans sa réalisation.

Avant de commencer

Vous allez dans cette section écrire votre premier programme. Vous allez sûrement tester les syntaxes directement dans l'interpréteur de commandes.

Vous pourriez préférer écrire votre code directement dans un fichier que vous pouvez ensuite exécuter. Si c'est le cas, je vous renvoie au chapitre traitant de ce point, que vous trouverez à la page 349 de ce livre.

Sujet

Le but de notre programme est de déterminer si une année saisie par l'utilisateur est bissextile. Il s'agit d'un sujet très prisé des enseignants en informatique quand il s'agit d'expliquer les conditions. Mille pardons, donc, à ceux qui ont déjà fait cet exercice dans un autre langage mais je trouve que ce petit programme reprend assez de thèmes abordés dans ce chapitre pour être réellement intéressant.

Je vous rappelle les règles qui déterminent si une année est bissextile ou non (vous allez peut-être même apprendre des choses que le commun des mortels ignore).

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
  - Si c'est le cas, on regarde si elle est multiple de 400.
    - Si c'est le cas, l'année est bissextile.
    - Sinon, elle n'est pas bissextile.
  - Sinon, elle est bissextile.

Solution ou résolution

Voilà. Le problème est posé clairement (sinon relisez attentivement l'énoncé autant de fois que nécessaire), il faut maintenant réfléchir à sa résolution en termes de programmation. C'est une phase de transition assez délicate de prime abord et je vous conseille de schématiser le problème, de prendre des notes sur les différentes étapes, sans pour l'instant penser au code. C'est une phase purement algorithmique, autrement dit, on réfléchit au programme sans réfléchir au code proprement dit.

Vous aurez besoin, pour réaliser ce petit programme, de quelques indications qui sont réellement spécifiques à Python. Ne lisez donc ceci qu'après avoir cerné et clairement écrit le problème d'une façon plus algorithmique. Cela étant dit, si vous peinez à trouver

une solution, ne vous y attardez pas. Cette phase de réflexion est assez difficile au début et, parfois il suffit d'un peu de pratique et d'explications pour comprendre l'essentiel.

La fonction `input()`

Tout d'abord, j'ai mentionné une année saisie par l'utilisateur. En effet, depuis tout à l'heure, nous testons des variables que nous déclarons nous-mêmes, avec une valeur précise. La condition est donc assez ridicule.

`input()` est une fonction qui va, pour nous, caractériser nos premières interactions avec l'utilisateur : le programme réagira différemment en fonction du nombre saisi par l'utilisateur.

`input()` accepte un paramètre facultatif : le message à afficher à l'utilisateur. Cette instruction interrompt le programme et attend que l'utilisateur saisisse ce qu'il veut puis appuie sur `Entrée`. À cet instant, la fonction renvoie ce que l'utilisateur a saisi. Il faut donc piéger cette valeur dans une variable.

```
>>> # Test de la fonction input
>>> annee = input("Saisissez une année : ")
Saisissez une année : 2009
>>> print(annee)
'2009'
>>>
```

Il subsiste un problème : le type de la variable `annee` après l'appel à `input()` est... une chaîne de caractères. Vous pouvez vous en rendre compte grâce aux apostrophes qui encadrent la valeur de la variable quand vous l'affichez directement dans l'interpréteur.

C'est bien ennuyeux : nous qui voulions travailler sur un entier, nous allons devoir convertir cette variable. Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction (c'est d'ailleurs exactement ce que c'est).

```
>>> type(annee)
<type 'str'>
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

Bon, parfait ! On a donc maintenant l'année sous sa forme entière. Notez que, si vous saisissez des lettres lors de l'appel à `input()`, la conversion renverra une erreur.

L'appel à la fonction `int()` en a peut-être déconcerté certains. On passe en paramètre de cette fonction la variable contenant la chaîne de caractères issue de `input()`, pour tenter de la convertir. La fonction `int()` renvoie la valeur convertie en entier et on la récupère donc dans la même variable. On évite ainsi de travailler sur plusieurs variables, sachant que la première n'a plus aucune utilité à présent qu'on l'a convertie.

Test de multiples

Certains pourraient également se demander comment tester si un nombre a est multiple d'un nombre b. Il suffit, en fait, de tester le reste de la division entière de b par a. Si ce reste est nul, alors a est un multiple de b.

```
>>> 5 % 2 # 5 n'est pas un multiple de 2
1
>>> 8 % 2 # 8 est un multiple de 2
0
>>>
```

À vous de jouer

Je pense vous avoir donné tous les éléments nécessaires pour réussir. À mon avis, le plus difficile est la phase de réflexion qui précède la composition du programme. Si vous avez du mal à réaliser cette opération, passez à la correction et étudiez-la soigneusement. Sinon, on se retrouve à la section suivante.

Bonne chance !

Correction

C'est l'heure de comparer nos méthodes et, avant de vous divulguer le code de ma solution, je vous précise qu'elle est loin d'être la seule possible. Vous pouvez très bien avoir trouvé quelque chose de différent mais qui fonctionne tout aussi bien.

Attention... la voiiiciiiiiii...

```
# Programme testant si une année, saisie par l'utilisateur,
# est bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'
    utilisateur saisisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre
bissextile = False # On crée un booléen qui vaut vrai ou faux
                   # selon que l'année est bissextile ou non

```

```
9  if annee % 400 == 0:
10     bissextile = True
11 elif annee % 100 == 0:
12     bissextile = False
13 elif annee % 4 == 0:
14     bissextile = True
15 else:
16     bissextile = False
17
18 if bissextile: # Si l'année est bissextile
19     print("L'année saisie est bissextile.")
20 else:
21     print("L'année saisie n'est pas bissextile.")
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers afin de les exécuter. Je vous renvoie à la page 349 pour plus d'informations.

Je pense que le code est assez clair, reste à expliciter l'enchaînement des conditions. Vous remarquerez qu'on a inversé le problème. On teste en effet d'abord si l'année est un multiple de 400, ensuite si c'est un multiple de 100, et enfin si c'est un multiple de 4. En effet, le `elif` garantit que, si `annee` est un multiple de 100, ce n'est pas un multiple de 400 (car le cas a été traité au-dessus). De cette façon, on s'assure que tous les cas sont gérés. Vous pouvez faire des essais avec plusieurs années et vous rendre compte si le programme a raison ou pas.

L'utilisation de `bissextile` comme d'un prédicat à part entière vous a peut-être déconcertés. C'est en fait tout à fait possible et logique, puisque `bissextile` est un booléen. Il est de ce fait vrai ou faux et donc on peut le tester simplement. On peut bien entendu aussi écrire `if bissextile==True:`, cela revient au même.

Un peu d'optimisation

Ce qu'on a fait était bien mais on peut l'améliorer. D'ailleurs, vous vous rendrez compte que c'est presque toujours le cas. Ici, il s'agit bien entendu de notre condition, que je vais passer au crible afin d'en construire une plus courte et plus logique, si possible. On peut parler d'optimisation dans ce cas, même si l'optimisation intègre aussi et surtout les ressources consommées par votre application, en vue de diminuer ces ressources et d'améliorer la rapidité de l'application. Mais, pour une petite application comme celle-ci, je ne pense pas qu'on perdra du temps sur l'optimisation du temps d'exécution.

Le premier détail que vous auriez pu remarquer, c'est que le `else` de fin est inutile. En effet, la variable `bissextile` vaut par défaut `False` et conserve donc cette valeur si le cas n'est pas traité (ici, quand l'année n'est ni un multiple de 400, ni un multiple de 100, ni un multiple de 4).

Ensuite, il apparaît que nous pouvons faire un grand ménage dans notre condition car les deux seuls cas correspondant à une année bissextile sont « si l'année est un multiple

de 400 »ou « si l'année est un multiple de 4 mais pas de 100 ».

Le prédicat correspondant est un peu délicat, il fait appel aux priorités des parenthèses. Je ne m'attendais pas que vous le trouviez tout seuls mais je souhaite que vous le compreniez bien à présent.

```
1 # Programme testant si une année, saisie par l'utilisateur, est
      bissextile ou non
2
3 annee = input("Saisissez une année : ") # On attend que l'
    utilisateur saisisse l'année qu'il désire tester
4 annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre
5
6 if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
7     print("L'année saisie est bissextile.")
8 else:
9     print("L'année saisie n'est pas bissextile.")
```

▷ Copier ce code
Code web : 886842

Du coup, on n'a plus besoin de la variable `bissextile`, c'est déjà cela de gagné. Nous sommes passés de 16 lignes de code à seulement 7 (sans compter les commentaires et les sauts de ligne) ce qui n'est pas rien.

En résumé

- Les conditions permettent d'exécuter certaines instructions dans certains cas, d'autres instructions dans un autre cas.
- Les conditions sont marquées par les mot-clés `if` (« si »), `elif` (« sinon si ») et `else` (« sinon »).
- Les mot-clés `if` et `elif` doivent être suivis d'un test (appelé aussi prédicat).
- Les booléens sont des données soit vraies (`True`) soit fausses (`False`).