Sélection ou exécution conditionnelle :

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de *tester une certaine condition* et de modifier le comportement du programme en conséquence.
La plus simple de ces instructions conditionnelles est l'instruction **if**. Pour expérimenter son fonctionnement, veuillez entrer dans votre éditeur Python les deux lignes suivantes:

```
>>> a = 150
>>> if (a > 100):
...
```

La première commande affecte la valeur 150 à la variable **a**. Jusqu'ici rien de nouveau. Lorsque vous finissez d'entrer la seconde ligne, par contre, vous constatez que Python réagit d'une nouvelle manière. En effet, et à moins que vous n'ayez oublié le caractère « : » à la fin de la ligne, vous constatez que le *prompt principal* (>>>) est maintenant remplacé par un *prompt secondaire* constitué de trois points[11].
Si votre éditeur ne le fait pas automatiquement, vous devez à présent effectuer une tabulation (ou entrer 4 espaces) avant d'entrer la ligne suivante, de manière à ce que celle-ci soit *indentée* (c'est-à-dire en retrait) par rapport à la précédente. Votre écran devrait se présenter maintenant comme suit :

```
>>> a = 150
>>> if (a > 100):
...     print("a dépasse la centaine")
...
```

Frappez encore une fois <*Enter*>. Le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec **a = 20** en guise de première ligne : cette fois Python n'affiche plus rien.
L'expression que vous avez placée entre parenthèses après **if** est ce que nous appellerons désormais une *condition*. L'instruction **if** permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons *indentée* après le **:** est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici avec l'instruction **if** sont optionnelles : nous les avons utilisées pour améliorer la lisibilité. Dans d'autres langages, il se peut qu'elles soient obligatoires. Recommencez encore, en ajoutant deux lignes comme indiqué ci-dessous. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

```
>>> a = 20
>>> if (a > 100):
...     print("a dépasse la centaine")
... else:
...     print("a ne dépasse pas cent")
...
```

Frappez <*Enter*> encore une fois. Le programme s'exécute, et affiche cette fois :

```
a ne dépasse pas cent
```

Comme vous l'aurez certainement déjà compris, l'instruction **else** (« sinon », en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de « else if ») :

```
>>> a = 0
>>> if a > 0 :
...     print("a est positif")
... elif a < 0 :
...     print("a est négatif")
... else:
...     print("a est nul")
...
```

Opérateurs de comparaison :

La condition évaluée après l'instruction if peut contenir les *opérateurs de comparaison* suivants :

```
x == y       # x est égal à y
x != y       # x est différent de y
x > y        # x est plus grand que y
x < y        # x est plus petit que y
x >= y       # x est plus grand que, ou égal à y
x <= y       # x est plus petit que, ou égal à y
```

Exemple :

```
>>> a = 7
>>> if (a % 2 == 0):
...     print("a est pair")
...     print("parce que le reste de sa division par 2 est nul")
... else:
...     print("a est impair")
...
```

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes « égale » et non d'un seul[12]. Le signe « égale » utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison. Vous retrouverez le même symbolisme en *C++* et en *Java.*

Instructions composées – blocs d'instructions :

La construction que vous avez utilisée avec l'instruction **if** est votre premier exemple d'**instruction composée**. Vous en rencontrerez bientôt d'autres. Sous Python, les

instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête. Exemple :

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau* (comptez un décalage de 4 caractères, par exemple). Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple du paragraphe précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction **if** constituent un même bloc logique : ces deux lignes ne sont exécutées – toutes les deux – que si la condition testée avec l'instruction **if** se révèle vraie, c'est-à-dire si le reste de la division de **a** par 2 est nul.

Instructions imbriquées :

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if embranchement == "vertébrés":                    # 1
    if classe == "mammifères":                      # 2
        if ordre == "carnivores":                   # 3
            if famille == "félins":                 # 4
                print("c'est peut-être un chat")    # 5
        print("c'est en tous cas un mammifère")     # 6
    elif classe == "oiseaux":                       # 7
        print("c'est peut-être un canari")          # 8
print("la classification des animaux est complexe") # 9
```

Analysez cet exemple. Ce fragment de programme n'imprime la phrase « c'est peut-être un chat » que dans le cas où les quatre premières conditions testées sont vraies.

Pour que la phrase « c'est en tous cas un mammifère » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 4) se trouve en effet au même niveau d'indentation que l'instruction : **if ordre == "carnivores"**: (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 et 2 sont vraies.

Pour que la phrase « c'est peut-être un canari » soit affichée, il faut que la variable **embranchement** contienne « vertébrés », et que la variable **classe** contienne « oiseaux ».

Quant à la phrase de la ligne 9, elle est affichée dans tous les cas, parce qu'elle fait partie du même bloc d'instructions que la ligne 1.

Quelques règles de syntaxe Python :

Tout ce qui précède nous amène à faire le point sur quelques règles de syntaxe :

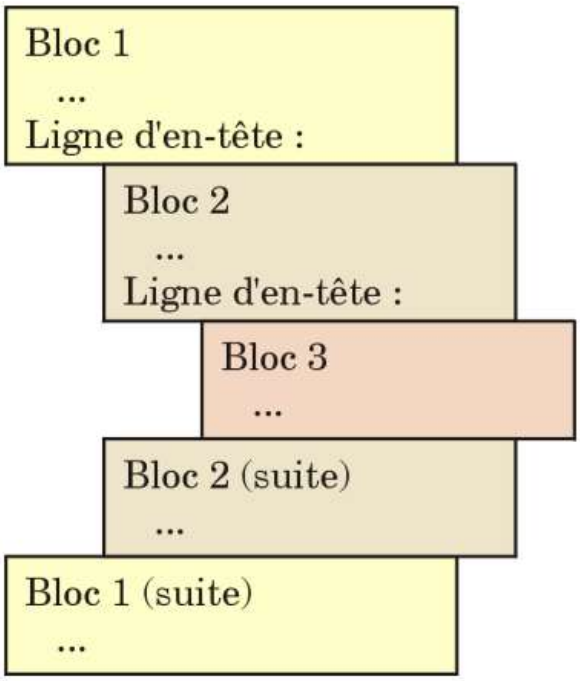
Instruction composée : en-tête, double point, bloc d'instructions indenté

Nous aurons de nombreuses occasions d'approfondir le concept de « bloc d'instructions » et de faire des exercices à ce sujet dès le chapitre suivant.
Le schéma ci-contre en résume le principe.

• Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, etc.) *se terminant par un double point.*

• Les blocs sont *délimités par l'indentation* : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c'est-à-dire décalées vers la droite d'un même nombre d'espaces). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.

• Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (il n'est imbriqué dans rien).



Les espaces et les commentaires sont normalement ignorés :

À part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés (sauf s'ils font partie d'une chaîne de caractères). Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante.

Répétitions en boucle – L'instruction while :

En programmation, on appelle *boucle* un système d'instructions qui permet de répéter un certain nombre de fois (voire indéfiniment) toute une série d'opérations. Python propose deux instructions particulières pour construire des boucles : l'instruction **for … in …** , très puissante, et l'instruction **while** que nous allons découvrir tout de suite.
Veuillez donc entrer les commandes ci-dessous :

```
>>> a = 0
>>> while (a < 7):          # (n'oubliez pas le double point !)
...     a = a + 1           # (n'oubliez pas l'indentation !)
...     print(a)
```

Frappez encore une fois <*Enter*>. Que se passe-t-il ?
Avant de lire les commentaires de la page suivante, prenez le temps d'ouvrir votre cahier et d'y noter cette série de commandes. Décrivez aussi le résultat obtenu, et essayez de l'expliquer de la manière la plus détaillée possible.

Commentaires :

Le mot **while** signifie « tant que » en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut *répéter continuellement le bloc d'instructions qui suit, tant que* le contenu de la variable **a** reste inférieur à 7.
Comme l'instruction **if** abordée au chapitre précédent, l'instruction **while** amorce une *instruction composée*. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces).

Nous avons ainsi construit notre première *boucle de programmation*, laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici comment cela fonctionne:

• Avec l'instruction **while**, Python commence par évaluer la validité de la *condition* fournie entre parenthèses (celles-ci sont optionnelles, nous ne les avons utilisées que pour clarifier notre explication).
• Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine.
• Si la codition est vraie, alors Python exécute tout le bloc d'instructions constituant *le corps de la boucle*, c'est-à-dire :
– l'instruction **a = a + 1** qui incrémente d'une unité le contenu de la variable **a** (ce qui signifie que l'on affecte à la variable **a** une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
– l'appel de la fonction **print()** pour afficher la valeur courante de la variable **a**.
• lorsque ces deux instructions ont été exécutées, nous avons assisté à une première *itération*, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction **while**. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite.

Dans notre exemple, si la condition **a < 7** est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.

Remarques :

• La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur).

• Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.

• Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner). Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par while, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer.

Exemple de boucle sans fin (à éviter !) :

```
>>> n = 3
>>> while n < 5:
...     print("hello !")
```

Élaboration de tables :

Recommencez à présent le premier exercice, mais avec la petite modification ci-dessous:

```
>>> a = 0
>>> while a < 12:
...     a = a +1
...     print(a , a**2 , a**3)
```

Vous devriez obtenir la liste des carrés et des cubes des nombres de 1 à 12.

Notez au passage que la fonction **print()** permet d'afficher plusieurs expressions l'une à la suite de l'autre sur la même ligne : il suffit de les séparer par des virgules. Python insère automatiquement un espace entre les éléments affichés.

Construction d'une suite mathématique :

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite appelée « *Suite de Fibonacci* ». Il s'agit d'une suite de nombres dont chaque terme est égal à la somme des deux termes qui le précèdent. Analysez ce programme (qui utilise judicieusement l'affectation parallèle) et décrivez le mieux possible le rôle de chacune des instructions.

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...     print(b, end =" ")
...     a, b, c = b, a+b, c+1
```

Lorsque vous lancez l'exécution de ce programme, vous obtenez :

```
1 2 3 5 8 13 21 34 55 89
```

Les termes de la suite de Fibonacci sont affichés sur la même ligne. Vous obtenez ce résultat grâce au second argument **end =" "** fourni à la fonction **print()**. Par défaut, la

fonction **print()** ajoute en effet un caractère de saut à la ligne à toute valeur qu'on lui demande d'afficher. L'argument **end =" "** signifie que vous souhaitez remplacer le saut à la ligne par un simple espace. Si vous supprimez cet argument, les nombres seront affichés les uns en-dessous des autres.

Dans vos programmes futurs, vous serez très souvent amenés à mettre au point des boucles de répétition comme celle que nous analysons ici. Il s'agit d'une question essentielle, que vous devez apprendre à maîtriser parfaitement. Soyez sûr que vous y arriverez progressivement, à force d'exercices.

Lorsque vous examinez un problème de cette nature, vous devez considérer les lignes d'instruction, bien entendu, mais surtout décortiquer *les états successifs des différentes variables* impliquées dans la boucle. Cela n'est pas toujours facile, loin de là. Pour vous aider à y voir plus clair, prenez la peine de dessiner sur papier une table d'états similaire à celle que nous reproduisons ci-dessous pour notre programme « suite de Fibonacci » :

Variables	a	b	c
Valeurs initiales	1	1	1
Valeurs prises successivement, au cours des itérations	1 2 3 5 ...	2 3 5 8 ...	2 3 4 5 ...
Expression de remplacement	b	a+b	c+1

Dans une telle table, on effectue en quelque sorte « à la main » le travail de l'ordinateur, en indiquant ligne par ligne les valeurs que prendront chacune des variables au fur et à mesure des itérations successives. On commence par inscrire en haut du tableau les noms des variables concernées. Sur la ligne suivante, les valeurs initiales de ces variables (valeurs qu'elles possèdent avant le démarrage de la boucle). Enfin, tout en bas du tableau, les expressions utilisées dans la boucle pour modifier l'état de chaque variable à chaque itération.

On remplit alors quelques lignes correspondant aux premières itérations. Pour établir les valeurs d'une ligne, il suffit d'appliquer à celles de la ligne précédente, l'expression de remplacement qui se trouve en bas de chaque colonne. On vérifie ainsi que l'on obtient bien la suite recherchée. Si ce n'est pas le cas, il faut essayer d'autres expressions de remplacement.

Exercices :

4.2 Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.

4.3 Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous :

1 euro(s) = 1.65 dollar(s)
2 euro(s) = 3.30 dollar(s)

4 euro(s) = 6.60 dollar(s)
8 euro(s) = 13.20 dollar(s)
etc. (S'arrêter à 16384 euros.)

4.4 Écrivez un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.