3 Contrôle du flux d'exécution

1 Algorithme

Tout problème informatique peut être résolu par l'exécution ordonnée d'une série finie d'instructions. La procédure de résolution d'un tel problème repose en principe sur un **algorithme**, c'est-à-dire une suite d'opérations décrivant :

- un nombre fini d'**actions** à exécuter,
- l'**ordre** dans lequel ces actions doivent être effectuées.

Un **programme informatique** consiste dès lors en la traduction d'un algorithme dans un langage compilable ou interprétable par un ordinateur. Ce programme est généralement écrit en plusieurs parties dont une qui pilote les autres, appelée le **programme principal**. Cependant, avant d'écrire un programme résolvant un problème particulier, il est essentiel d'avoir bien compris et planifié la résolution de ce dernier. Afin d'aider le programmeur dans sa tâche, différents artifices ont été développés :

- le **pseudo-code** décrit en langage naturel les étapes de l'algorithme. Son aspect descriptif permet de commencer par une vision très large et de passer outre temporairement certains aspects complexes de l'algorithme. Une fois cette première description grossière réalisée, chacune de ses étapes est affinée en plus petites tâches jusqu'à obtenir un pseudo-code suffisamment précis pour être converti en langage formel. Cette approche de résolution de problème est appelée **top-down**.
- l'**organigramme** représente (une portion de) l'algorithme à l'aide d'éléments graphiques tels que des petits **cercles** (symbolisant le *début* et la *fin* de l'algorithme), des **rectangles** (symbolisant une *action* à entreprendre) et des **losanges** (symbolisant une *condition* à vérifier). Ces symboles graphiques sont reliés par des **flèches** qui indiquent l'ordre dans lequel les actions doivent être effectuées.

Ces artifices permettent au programmeur de prendre conscience de toute la difficulté de la mise en œuvre de l'algorithme et de développer une démarche structurée en vue de son implémentation.

A titre d'exemple, considérons l'algorithme permettant de calculer la factorielle [1] d'un entier naturel n. Sa description est la suivante :

Pseudo-code

Récupérer un entier n en entrée

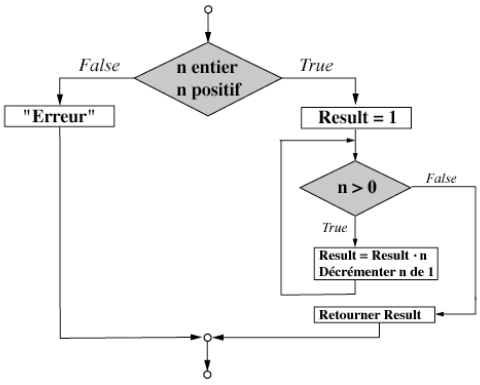
Si n n'est pas un entier naturel
  Retourner un message d'erreur

Sinon
  Initialiser une variable *result* à 1

  Tant que n est positif
    Affecter *result* de n*result
    Décrémenter n de 1

  Retourner *result*

Organigramme



Les instructions d'un programme sont généralement exécutées de manière séquentielle, c'est-à-dire selon l'ordre dans lequel elles ont été écrites. Néanmoins, comme l'illustre l'exemple ci-dessus, il existe bien des cas où le flux d'exécution du programme n'est pas séquentiel mais soumis à une condition ou une répétition. Les constructions permettant de modifier l'ordre d'exécution d'un programme sont appelées **instructions de contrôle de flux** et sont présentées dans les sections suivantes.

1. La factorielle d'un entier naturel n, notée $n!$, est le nombre défini par $n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$. Par exemple, $4! = 24$.

2 Structure séquentielle

Sauf mention explicite du contraire, les instructions d'un programme s'exécutent les unes après les autres, *dans l'ordre où elles ont été écrites à l'intérieur du script*. Cette structure **séquentielle** est symbolisée par l'organigramme ci-dessous :

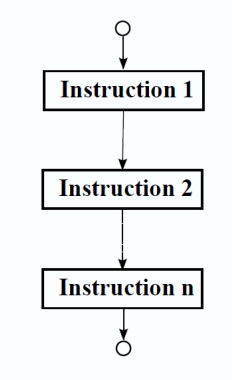


FIGURE 3.1 – Structure séquentielle

L'expérience montre qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres. Par exemple, dans la séquence d'instructions suivantes :

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
>>> print(a, b)
```

Vous obtiendrez un résultat différent si vous intervertissez les 2ème et 3ème lignes.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une **instruction conditionnelle** comme les instructions `if`, `if/else` ou `if/elif/else` décrites ci-après ou une **boucle de répétition** caractérisée par les instructions `while` ou `for`. De telles instructions vont permettre au programme de suivre différents chemins suivant les circonstances.

3 Structures conditionnelles

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 :

```
>>> note = 5.2
>>> if (note >= 4):          # parenthèses optionnelles
...
```

La première commande affecte la valeur 5.2 à la variable `note`. 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 [2].

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.

2. Dans certaines versions de l'éditeur Python pour ***Windows***, le prompt secondaire n'apparaît pas.

Votre écran devrait se présenter maintenant comme suit :

```
>>> note = 5.2
>>> if (note >= 4):              # parenthèses optionnelles
...     print("Promu")
...
```

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

```
Promu
```

Recommencez le même exercice, mais avec `note = 3.5` en guise de première ligne : cette fois Python n'affiche plus rien.

L'expression que vous avez placée entre parenthèses 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. La structure conditionnelle simple **if** est symbolisée par l'organigramme suivant :

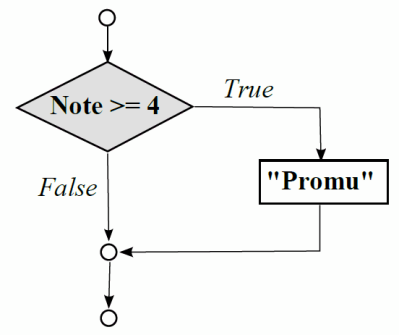


FIGURE 3.2 – Structure conditionnelle simple

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) :

```
>>> note = 3.5
>>> if (note >= 4):              # parenthèses optionnelles
...     print("Promu")
... else:
...     print("Non Promu")
...
```

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

```
Non Promu
```

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. La structure de sélection composée *if/else* est symbolisée par l'organigramme suivant :

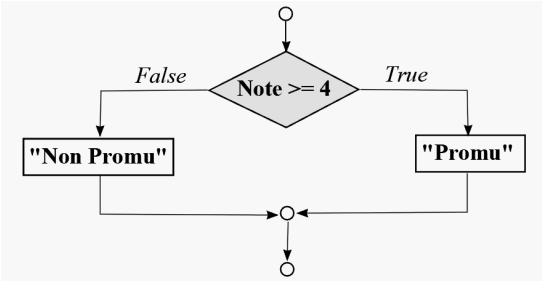


FIGURE 3.3 – Structure conditionnelle composée

On peut faire mieux encore en utilisant aussi l'instruction `elif` (contraction de *else if*) :

```
>>> note = 5.2
>>> if (note >= 5.5):
...     print("Très bien")
... elif (note >= 5):
...     print("Bien")
... elif (note >= 4.5):
...     print("Assez bien")
... elif (note >= 4):
...     print("Suffisant")
... else:
...     print("Insuffisant")
...
```

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

```
Bien
```

Comme vous pouvez le constatez, une *seule instruction* est exécutée. En fait, les conditions sont évaluées les unes après les autres selon leur ordre d'apparition. Si la première condition est fausse, la suivante est vérifiée et ainsi de suite. Si l'une des conditions est satisfaite, l'instruction correspondante est alors exécutée et le processus se termine. Même si plus d'une condition est satisfaite, seule le bloc d'instructions suivant la première est exécuté. Ainsi, dans l'exemple ci-dessus, le programme affiche uniquement *Bien,* même si la variable `note` vérifie également les conditions situées en-dessous de la ligne 5.

La structure de sélection multiple `if/elif/else` est symbolisée par l'organigramme suivant :

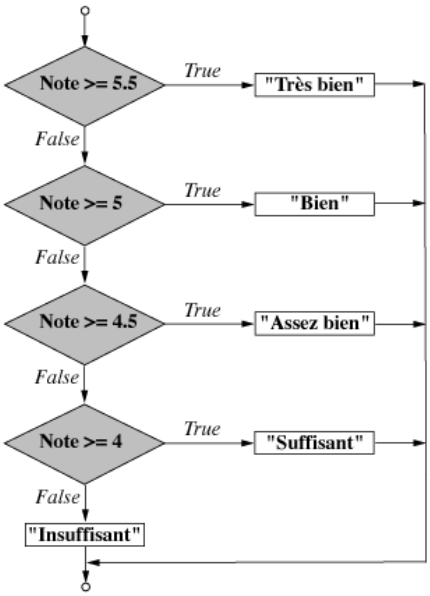


FIGURE 3.4 – Structure conditionnelle multiple

3.1 Opérateurs de comparaison

Une **expression booléenne** est une expression qui est vraie ou fausse. Les exemples suivants utilisent l'opérateur < qui comparent deux opérandes et retourne `True` si la première est strictement plus petite que la deuxième et `False` sinon :

```
>>> 5 < 6
True
>>> 6 < 6
False
```

`True` et `False` sont des valeurs spéciales appartenant au type `bool` mais ne sont pas de type `str`. La condition évaluée après l'instruction `if` peut contenir d'autres **opérateurs de comparaison** que < :

```
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
```

Par 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[3]. 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.

Tous les opérateurs de comparaison présentés ci-dessus fonctionnent aussi avec les chaînes de caractères. Cela peut vous être utile pour trier des mots par ordre alphabétique, comme l'illustre l'exemple ci-dessous :

```
mot = input("Entrez un mot quelconque : (<enter> pour terminer)")
if mot < "limonade":
    place = "précède"
elif mot > "limonade":
    place = "suit"
else:
    place = "se confond avec"
print("Le mot", mot, place, "le mot 'limonade' dans l'ordre alphabétique")
```

Ces comparaisons sont possibles, parce que dans toutes les normes d'encodage, les codes numériques représentant les caractères ont été attribués dans l'ordre alphabétique, tout au moins pour les caractères non accentués. Dans le système de codage **ASCII**, par exemple, A correspond à 65, B à 66, C à 67, etc.

0	32	64 @	96 ‘	128	160	192 À	224 à
1	33 !	65 A	97 a	129	161 ¡	193 Á	225 á
2	34 "	66 B	98 b	130	162 ¢	194 Â	226 â
3	35 #	67 C	99 c	131	163 £	195 Ã	227 ã
4	36 $	68 D	100 d	132	164 ¤	196 Ä	228 ä
5	37 %	69 E	101 e	133	165 ¥	197 Å	229 å
6	38 &	70 F	102 f	134	166 ¦	198 Æ	230 æ
7	39 ’	71 G	103 g	135	167 §	199 Ç	231 ç
8	40 (	72 H	104 h	136	168 ¨	200 È	232 è
9	41)	73 I	105 i	137	169 ©	201 É	233 é
10	42 *	74 J	106 j	138	170 ª	202 Ê	234 ê
11	43 +	75 K	107 k	139	171 «	203 Ë	235 ë
12	44 ,	76 L	108 l	140	172 ¬	204 Ì	236 ì
13	45 -	77 M	109 m	141	173 -	205 Í	237 í
14	46 .	78 N	110 n	142	174 ®	206 Î	238 î
15	47 /	79 O	111 o	143	175 ¯	207 Ï	239 ï
16	48 0	80 P	112 p	144	176 °	208 Ð	240 ð
17	49 1	81 Q	113 q	145	177 ±	209 Ñ	241 ñ
18	50 2	82 R	114 r	146	178 ²	210 Ò	242 ò
19	51 3	83 S	115 s	147	179 ³	211 Ó	243 ó
20	52 4	84 T	116 t	148	180 ´	212 Ô	244 ô
21	53 5	85 U	117 u	149	181 µ	213 Õ	245 õ
22	54 6	86 V	118 v	150	182 ¶	214 Ö	246 ö
23	55 7	87 W	119 w	151	183 ·	215 ×	247 ÷
24	56 8	88 X	120 x	152	184 ¸	216 Ø	248 ø
25	57 9	89 Y	121 y	153	185 ¹	217 Ù	249 ù
26	58 :	90 Z	122 z	154	186 º	218 Ú	250 ú
27	59 ;	91 [	123 {	155	187 »	219 Û	251 û
28	60 <	92 \	124 |	156	188 ¼	220 Ü	252 ü
29	61 =	93]	125 }	157	189 ½	221 Ý	253 ý
30	62 >	94 ^	126 ~	158	190 ¾	222 Þ	254 þ
31	63 ?	95 _	127	159	191 ¿	223 ß	255 ÿ

FIGURE 3.5 – Table des caractères de l'ASCII étendu

3. Pour rappel, l'opérateur % est l'opérateur modulo : il calcule le reste d'une division entière. Ainsi par exemple, $a\%2$ fournit le reste de la division de a par 2.

Comprenez cependant que cela ne fonctionne bien que pour des mots qui sont tous entièrement en minuscules, ou entièrement en majuscules, et qui ne comportent aucun caractère accentué. Les majuscules et minuscules utilisent en effet des ensembles de codes distincts : les codes 65 à 90 sont réservés pour les majuscules alors que les minuscules prennent des codes situés entre 97 et 122. Quant aux caractères accentués, vous remarquerez dans la table ci-dessus qu'ils sont encodés en dehors de l'ensemble constitué par les caractères du standard ASCII. Dès lors, construire un algorithme de tri alphabétique qui prenne en compte à la fois la casse des caractères et tous leurs accents n'est pas une mince affaire !

3.2 Opérateurs logiques

Les opérateurs de comparaison ci-dessus permettent de définir *une seule* condition. Afin de créer des conditions plus complexes combinant plusieurs conditions simples, il est possible de recourir aux **opérateurs logiques** `and` (et), `or` (ou) et `not` (négation). Le tableau suivant décrit, pour chacun de ces trois opérateurs, tous les résultats possibles en fonction des conditions en présence. Un tel tableau est appelé **table de vérité** :

Opérateurs	and				or				not	
Condition 1	True	True	False	False	True	True	False	False	True	False
Condition 2	True	False	True	False	True	False	True	False		
Résultat	True	False	False	False	True	True	True	False	False	True

Exemple :

```
>>> (3 == 3) and (9 > 15)
False
>>> (3 == 3) or (9 > 15)
True
>>> not (9 > 15)
True
```

Stricto sensu, les opérandes d'un opérateur logique doivent être de type *booléen* comme le montrent les trois premiers exemples ci-dessus. Néanmoins, - et vous le savez déjà – un ordinateur ne manipule strictement que des nombres. Tout ce qu'un ordinateur doit traiter doit d'abord toujours être converti en valeur numérique. Ainsi, en Python, on interprète comme `True` tout nombre non nul. Seule la valeur zéro est `False`.

Exemple :

```
ch = input('Entrez un nombre entier quelconque: ')
n =int(ch)
if n:
    print("vrai")
else:
    print("faux")
```

Le petit script ci-dessus n'affiche *faux* que si vous entrez la valeur 0. Pour toute autre valeur numérique, vous obtiendrez *vrai*.

Ce qui précède signifie donc qu'une expression à évaluer, telle par exemple la condition `a > 5`, est d'abord convertie par l'ordinateur en une valeur numérique (**1** si l'expression est vraie et **0** si l'expression est fausse). Ce n'est pas tout à fait évident, parce que l'interpréteur Python est doté d'un dispositif qui traduit ces deux valeurs en `True` ou `False` lorsqu'on les lui demande explicitement.

Exemple :

```
>>> a, b = 3, 8
>>> c = (a < b)
>>> d = (a > b)
>>> c
True
>>> d
False
```

(L'expression `a < b` est évaluée et son résultat (*vrai*) est mémorisé dans la variable `c`. De même pour le résultat de l'expression inverse, dans la variable `d`.)

À l'aide d'une petite astuce, nous pouvons quand même vérifier que ces valeurs `True` et `False` sont en réalité les deux nombres 1 et 0 *déguisés* :

```
>>> chaine = "HS"
>>> chaine[d]
H
>>> chaine[c]
S
```

En utilisant les valeurs des variables `c` et `d` comme indices pour extraire les caractères de la chaîne de caractères `chaine`, nous confirmons bien que `False` = 0 et `True` = 1.

Le petit script ci-après est très similaire au précédent. Il nous permet de tester que `False` = 0 et `True` = 1 :

```
n = int(input("Veuillez entrer un nombre : "))
if n % 2:
    print("Il s'agit d'un nombre impair.")
else:
    print("Il s'agit d'un nombre pair.")
```

L'instruction `if n % 2:`, à la troisième ligne de cet exemple, est équivalente à une instruction du type : `if n % 2 != 0:`, du moins de notre point de vue d'êtres humains.

Pour l'ordinateur, cependant, ce n'est pas tout à fait pareil. Pour lui, l'instruction `if n % 2:` consiste à vérifier directement si la valeur de l'expression `n % 2` est nulle ou non. La seconde formulation, `if n % 2 != 0:`, lui impose par contre de commencer par comparer la valeur de `n % 2` *à la valeur d'une autre donnée que nous lui fournissons par notre programme* (l'entier 0), puis à évaluer *ensuite* si le résultat de cette comparaison est lui-même *vrai* ou *faux* (ou en d'autres termes, à vérifier si ce résultat est lui-même `True` ou `False`). Cela lui demande donc deux opérations successives, là où la première formulation ne lui en demande qu'une seule. La première formulation est donc plus performante.

Ce raisonnement *proche de la machine* vous paraîtra probablement un peu subtil au début, mais croyez bien que cette forme d'écriture vous deviendra très vite tout à fait familière.

3.3 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.

Une instruction composée est donc schématisée de la manière suivante :

```
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 une tabulation, 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.

Vous pouvez aussi indenter à l'aide d'espaces, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, et même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.

3.4 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, comme l'illustre l'exemple suivant :

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

Ce fragment de programme n'imprime la phrase « on a peut-être affaire à un chat » que dans le cas où les quatre premières conditions testées sont vraies.

Pour que la phrase « on a en tous cas affaire à un mammifère » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 6) 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 « on a peut-être affaire à 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.

3.5 Quelques règles de syntaxe Python

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

- **Les limites des instructions et des blocs sont définies par la mise en page**
  Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de fin de ligne [4] qui joue ce rôle. Pour étendre une instruction complexe sur plusieurs lignes, il est néanmoins possible d'outrepasser cette règle en utilisant l'antislash \ permettant de continuer une même instruction sur la ligne suivante.

  Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que `begin` et `end`). En C++ et en Java, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper ni d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire pour les pauvres humains que nous sommes. On conseille donc à tous les programmeurs qui utilisent ces langages de se servir **aussi** des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

  Avec Python, vous **devez** utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.
- **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. Néanmoins, pour l'instant, gardez en mémoire les éléments suivants :
  — Les blocs d'instructions sont toujours associés à une **ligne d'en-tête** contenant une instruction bien spécifique (`if`, `elif`, `else`, `while`, `for`, `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 ou de tabulations).
  — Notez que le code du bloc le plus externe ne peut pas lui-même être écarté de la marge de gauche (il n'est imbriqué dans rien).

4. Ce caractère n'apparaît ni à l'écran, ni sur les listings imprimés. Il est cependant bien présent, à un point tel qu'il fait même problème dans certains cas, parce qu'il n'est pas encodé de la même manière par tous les systèmes d'exploitation. Nous en reparlerons plus loin, à l'occasion de notre étude des fichiers texte.

Le schéma ci-dessous résume le principe des blocs d'instructions :

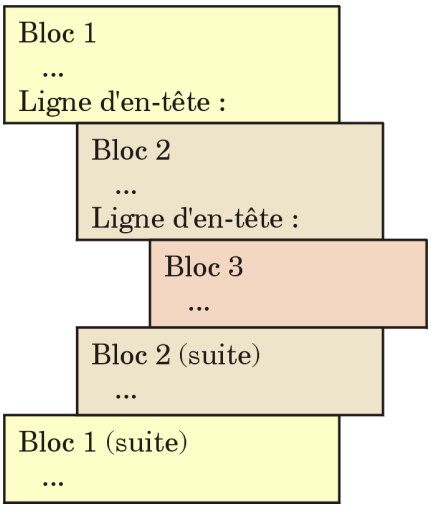


FIGURE 3.6 – Structure des blocs d'instructions en Python

- **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.

4 Structures répétitives

L'une des tâches que les machines font le mieux est la *répétition* sans erreur de tâches identiques. Il existe deux méthodes pour programmer ces tâches répétitives : les boucles construites à partir de l'instruction `while` et celles gérées par l'instruction `for`. Schématiquement, une structure répétitive est symbolisée par l'organigramme suivant :

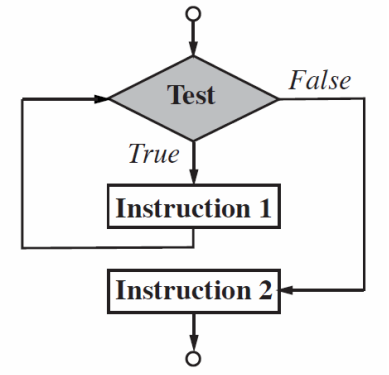


FIGURE 3.7 – Structure répétitive

Il s'agit d'une instruction composée indiquant qu'il faut répéter continuellement le bloc d'instructions suivant le `while` ou le `for`, tant que le test défini dans l'en-tête est satisfait.

4.1 Boucle conditionnelle - l'instruction while

En anglais, le mot `while` signifie *tant que*. Comme l'instruction `if` abordée dans la section précédente, l'instruction `while` amorce une **instruction composée**. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lesquelles doivent obligatoirement être indentées au même niveau :

```
1  total = 0
2  prix = input("Entrez le prix du premier article ou -1 pour terminer:")
3  prix = float(prix)                # conversion de la chaine en float
4
5  while (prix != -1):               # double point obligatoire
6          total += prix      # indentation obligatoire
7          prix = input("Entrez le prix du prochain article ou -1 pour terminer:")
8          prix = float(prix)
9
10 print("Le prix total à payer est de " + str(total) + " franc(s).")
```

Nous avons ainsi construit notre première **boucle de programmation** (ln 5 à 8), laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici son fonctionnement :

- 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 continue avec la première instruction qui suit le bloc indenté et qui fait partie du même bloc que l'instruction `while` elle-même, c'est-à-dire la ligne 10.
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant **le corps de la boucle**, c'est-à-dire :
  - l'instruction `total += prix` (ln 6), équivalente à `total = total + prix`, qui incrémente le contenu de la variable `total` d'autant d'unités qu'en contient la variable `prix`.
  - l'appel de la fonction `input()` (ln 7) pour mettre à jour le contenu de la variable `prix`.
  - l'appel de la fonction `float()` (ln 8) pour convertir la valeur de la variable `prix` en nombre réel.
- Lorsque ces trois 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 5 contenant l'instruction `while`. La condition qui s'y trouve est à nouveau évaluée et ainsi de suite.
  Dans notre exemple, si la condition `prix != -1` est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit [5].

Remarques

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

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

(c) 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 :

```
prix = float(input("Entrez le prix du premier article ou -1 pour terminer:"))

while (prix != -1):
        print("Le prix de l'article est de " + str(prix) + " franc(s).")
```

5. La variable `prix` est une sorte d'*interrupteur* (*flag* en anglais) car elle gère la condition d'exécution de la boucle. Cette boucle est exécutée aussi longtemps que l'utilisateur n'a pas rentré la valeur -1. Cette valeur, signalant la fin de la boucle, est appelée *valeur sentinelle*. En fait, toute valeur ne pouvant être confondue avec une valeur d'entrée acceptable peut être choisie comme valeur sentinelle. Dans notre exemple, comme le prix d'un article est toujours positif, la valeur -1 convient parfaitement.

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 l'un en-dessous de l'autre.

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.

4.2 Parcours d'une séquence : l'instruction for ... in ...

Il arrive très souvent que l'on doive traiter l'intégralité d'une chaîne caractère par caractère, du premier jusqu'au dernier, pour effectuer à partir de chacun d'eux une opération quelconque. Nous appellerons cette opération un **parcours** ou une **boucle avec compteur**. En nous limitant aux outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours à l'aide d'une boucle, articulée autour de l'instruction `while` et d'un `compteur` gérant l'indice du caractère de la chaîne à traiter à chaque itération :

```
>>> nom ="Joséphine"
>>> index =0
>>> while index < len(nom):
...     print(nom[index], end =' *')
...     index += 1
...
J * o * s * é * p * h * i * n * e *
```

Cette boucle **parcourt** donc la chaîne `nom` pour en extraire un à un tous les caractères, lesquels sont ensuite imprimés avec interposition d'astérisques. Notez bien que la condition utilisée avec l'instruction `while` est `index < len(nom)`, ce qui signifie que le bouclage, géré par le compteur `index`, doit s'effectuer jusqu'à ce que l'on soit arrivé à l'indice numéro 9 (la chaîne compte en effet 10 caractères). Nous aurons effectivement traité tous les caractères de la chaîne, puisque ceux-ci sont indicés de 0 à 9.

Le **parcours d'une séquence**[6] est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée que la boucle `while`, basée sur le couple d'instructions `for ... in ...`

Avec ces instructions, le programme ci-dessus devient :

```
>>> nom ="Joséphine"
>>> for car in nom:
...         print(car, end ='*')
...
J * o * s * é * p * h * i * n * e *
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle vous évite d'avoir à définir et à incrémenter un compteur pour gérer l'indice du caractère que vous voulez traiter à chaque itération (c'est Python qui s'en charge). La structure `for ... in ...` ne montre donc que l'essentiel, à savoir que la **variable de contrôle** `car` contiendra successivement tous les caractères de la chaîne, du premier jusqu'au dernier.

L'instruction `for ... in ...` est un nouvel exemple d'**instruction composée**. Le nom qui suit le mot réservé `in` est celui de la chaîne qu'il faut traiter. Le nom qui suit le mot réservé `for` est celui que vous choisissez pour la **variable de contrôle** destinée à contenir successivement tous les caractères de la chaîne. Cette variable est définie automatiquement (c'est-à-dire qu'il est inutile de la définir au préalable).

Appartenance d'un élément à une séquence : l'instruction in utilisée seule

L'instruction `in` peut être utilisée indépendamment de `for`, pour *vérifier si un caractère donné fait partie ou non d'une chaîne :*

```
car = input("Entrez un caractère alphabétique : ")
voyelles = "aeiouyAEIOUYàâäéèêîôûù"
if car in voyelles:
    print(car, "est une voyelle")
```

Cette instruction très puissante effectue donc à elle seule un véritable parcours de la chaîne. À titre d'exercice, écrivez les instructions qui effectueraient le même travail à l'aide d'une boucle classique utilisant l'instruction `while`.

6. Une **séquence** est un type de données formant un ensemble *ordonné* d'éléments indicés par des entiers. Nous étudierons d'autres exemples de séquences comme les *listes* et les *tuples* dans le chapitre 5.

4.3 Structures de rupture de séquence

Il arrive qu'il ne soit pas possible de déterminer à l'avance à quel moment un programme doit *sortir* d'une boucle. Afin de pallier à ce type de situations, Python offre deux **structures de rupture de séquence** permettant de sortir immédiatement de la boucle en cours d'exécution (`break`) ou de reprendre l'exécution à la ligne d'en-tête de la boucle en laissant sous silence l'exécution des instructions de l'itération en cours (`continue`).

Lorsqu'elle est utilisée à l'intérieur d'une boucle, l'instruction `break` cause la sortie immédiate de la structure répétitive. L'exécution du programme continue alors avec la première instruction située *après* la boucle :

```
total = 0

while 1:
        prix = input("Entrez le prix de l'article suivant ou -1 pour terminer:")
        prix = float(prix)
        if prix == -1:
            break
        total = total + prix

print("Le prix total à payer est de " + str(total) + " franc(s).")
```

Une boucle `while` construite comme nous l'avons fait ci-dessus devrait boucler indéfiniment, puisque la condition de continuation reste toujours vraie[7]. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction `break`, laquelle permettra à l'exécution du programme de passer *directement* à l'affichage du prix total à payer lorsque la valeur sentinelle -1 sera contenue dans la variable `prix`. L'instruction `break` permet de mettre en place plusieurs mécanismes de sortie différents pour une même boucle :

```
while <condition 1> :
    --- instructions diverses ---
    if <condition 2> :
        break
    --- instructions diverses ---
    if <condition 3>:
        break
    etc.
```

Lorsqu'une instruction `continue` est placée à l'intérieur d'une boucle, l'exécution du programme saute les instructions restantes du corps de la boucle et **passe directement à l'itération suivante** en reprenant au niveau de l'en-tête de la boucle :

```
result = ""

for car in "abracadabra":
    if car == "a":
        continue
    result = result + car

print("En supprimant la lettre 'a', on obtient ", result)
```

Dans l'exemple ci-dessus permettant de supprimer toutes les occurrences de la lettres « a » dans le mot « abracadabra », la variable de contrôle `car` est affectée du caractère suivant de la chaîne à chaque saut d'itération.

7. La boucle `while` utilisée dans cette fonction est construite d'une manière différente de ce que vous avez rencontré précédemment. Nous avons remplacé la condition à évaluer par une simple constante non nulle, rendant ainsi la condition suivant le `while` toujours vraie.

5 Structures de gestion d'exceptions

Les **exceptions** sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue et un message d'erreur plus ou moins explicite est affiché.

Exemple :

```
>>> print(55/0)
ZeroDivisionError: int division or modulo by zero
```

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le **type d'erreur** et ensuite une **information spécifique** de cette erreur.

Dans de nombreux cas, il est possible de prévoir certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à *tout un ensemble d'instructions*, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Un mécanisme de ce type s'appelle en général mécanisme de **traitement des exceptions**. Celui de Python utilise l'ensemble d'instructions `try` - `except` – `else`, qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Son fonctionnement est le suivant :

- Le bloc d'instructions qui suit directement une instruction `try` est exécuté par Python *sous réserve*.
- Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction `except`.
- Si aucune erreur ne s'est produite dans les instructions qui suivent `try`, alors c'est le bloc qui suit l'instruction `else` qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

Considérons par exemple un script qui demande à l'utilisateur d'entrer deux nombres, lesquels étant destinés à représenter le numérateur et le dénominateur d'une fraction. Si le dénominateur est nul, nous ne voulons pas que le programme « plante ». Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer une nouvelle valeur :

```
1  a = input("Entrez la valeur du numérateur: ")
2  b = input("Entrez la valeur du dénominateur: ")
3
4  a = float(a)
5  b = float(b)
6
7  try:                # double point obligatoire
8      result = a/b    # indentation obligatoire
9
10 except:
11     print("Le dénominateur doit être non nul !")
12
13 else:
14     print("a / b = ", result)
```

Dans l'exemple ci-dessus, si l'utilisateur entre 0 pour le dénominateur, une exception va être détectée par l'interpréteur Python lorsqu'il essayera d'exécuter la division par 0 située dans la structure `try` (ln 7-8) et sera gérée par la structure `except`. Au contraire, si les nombres entrés par l'utilisateur n'engendre aucune division par 0, l'interpréteur passe sous silence l'instruction située dans la structure `except` et exécute directement le code situé dans la structure `else`.

Il est également possible de faire suivre l'instruction `try` de plusieurs blocs `except`, chacun d'entre eux traitant un type d'erreur spécifique. Ainsi, dans l'exemple précédent, un autre type d'erreur pourrait survenir si l'utilisateur entrait des données non numériques. Dans ce cas, l'erreur surviendrait au moment de la conversion des valeurs entrées en `float` (ln 4-5) et ne serait pas gérée par la structure `try` – `except` – `else`. Afin de pallier à ce manquement, il convient de faire suivre le `try` de deux blocs `except` : le premier traitant l'erreur générée par

la conversion de type (qui est une exception de type `ValueError` pour Python) et le deuxième gérant l'erreur occasionnée par une division par 0 (qui est une exception de type `ZeroDivisionError` pour Python) :

```
a = input("Entrez la valeur du numérateur: ")
b = input("Entrez la valeur du dénominateur: ")

try: # double-point obligatoire
    a = float(a) # indentation obligatoire
    b = float(b)
    result = a/b

except ValueError: # nom de l'exception à traiter
    print("Les valeurs doivent être numériques !")

except ZeroDivisionError: # nom de l'exception à traiter
    print("Le dénominateur doit être non nul !")

else:
    print("a / b = ", result)
```

Les noms des différentes exceptions levées par Python peuvent être consultés dans un ouvrage de référence ou à l'adresse https://docs.python.org/3/tutorial/errors.html.

6 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

algorithme : enchaînement ordonné d'actions nécessaires à l'accomplissement d'une tâche.

boucle infinie : boucle pour laquelle la condition d'arrêt n'est jamais vérifiée.

compteur : variable utilisée pour compter quelque chose, habituellement initialiser à zéro et incrémenter par la suite.

condition multiple : structure conditionnelle présentant une série d'alternatives possibles.

condition : expression booléenne située dans l'en-tête d'une structure conditionnelle et déterminant l'alternative à exécuter.

corps : bloc d'instructions situées à l'intérieur d'une instruction composée.

décrémenter : diminuer la valeur d'une variable (le plus souvent d'une unité).

exception : erreur détectée par l'interpréteur Python.

expression booléenne : expression dont la valeur est vraie (`True`) ou fausse (`False`).

flux d'exécution : ordre dans lequel les instructions sont réalisées lors de l'exécution d'un programme.

incrémenter : augmenter la valeur d'une variable (le plus souvent d'une unité).

initialiser : affecter une première valeur à une variable qui pourra être mise à jour par la suite.

instructions composées : structure composée d'une en-tête qui se termine par " :" et d'un corps composé d'un bloc d'instructions indenté relativement à l'en-tête.

itération : exécution répétée d'un bloc d'instructions situées dans une boucle.

opérateur de comparaison : opérateur permettant de comparer ses opérandes (= =, !=, >, <, >= et <=).

opérateur logique : opérateur permettant de combiner des expressions booléennes (`and`, `or` et `not`).

organigramme : représentation graphique du déroulement d'un algorithme.

parcours : boucle opérant sur les éléments d'une chaîne de caractères (ou plus généralement d'une séquence), exécutant une instruction similaire sur chacun d'eux.

programme : ensemble d'instructions, rédigé dans un langage de programmation, permettant à un système informatique d'exécuter une tâche donnée.

pseudo-code : descriptif du déroulement d'un algorithme en langage naturelle.

structure conditionnelle : structure de contrôle de flux d'exécution dépendant d'une condition.

structure de contrôle : groupes d'instructions déterminant l'ordre dans lequel les actions sont effectuées.

structure répétitive : structure de contrôle permettant de répéter sous certaines condition un bloc d'instructions.

structure séquentielle : groupe d'instructions exécutées dans l'ordre où elles ont été écrites à l'intérieur du script.

traiter une exception : prévenir l'interruption d'un programme causée par la détection d'une exception en appliquant la structure `try-except-else`.

variable sentinelle : variable utilisée pour vérifier si une condition est vérifiée.

7 Exercices

Exercice 3.1
Ecrivez un script qui demande à l'utilisateur son nom et son sexe (M ou F) et en fonction de ces données, retourne « Cher Monsieur » ou « Chère Madame » suivi du nom de l'utilisateur.

Exercice 3.2
Ecrivez un script qui calcule le minimum de 3 nombres entiers entrés par l'utilisateur.

Exercice 3.3
Construisez l'organigramme décrivant l'algorithme qui prend en arguments trois mots écrits en minuscules et les affiche triés alphabétiquement puis implémentez un script traduisant cet algorithme en Python.

Exercice 3.4
Ecrivez un script qui détermine si une année entrée par l'utilisateur est bissextile ou non.
(Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400)

Exercice 3.5
Ecrivez un script qui affiche horizontalement les 20 premiers multiples d'un entier n choisi par l'utilisateur.

Exercice 3.6
Ecrivez un script qui affiche horizontalement les 20 premiers multiples d'un entier n choisi par l'utilisateur, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
(Si $n = 7$, le programme retournera : 7 14 21* 28 35 42* 49...)

Exercice 3.7
Ecrivez un script qui calcule les 50 premiers multiples d'un entier n choisi par l'utilisateur mais qui n'affiche verticalement que ceux qui sont des multiples de 7.

Exercice 3.8
(1) Ecrivez un script qui, étant donnés deux bornes entières a et b données en entrée par l'utilisateur, additionne les multiples de 3 et 5 compris entre ces bornes. Ainsi, pour $a = 0$ et $b = 32$, le résultat doit être $0 + 15 + 30 = 45$.

(2) Modifier légèrement le programme précédent pour qu'il additionne les multiples de 3 ou 5 compris entre les bornes a et b. Ainsi, pour $a = 0$ et $b = 32$, le résultat doit être $0 + 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 + 20 + 21 + 24 + 25 + 27 + 30 = 225$.

Exercice 3.9
Ecrivez un script qui affiche une suite de 12 nombres dont chaque terme est égal au triple du terme précédent. Le premier terme de la suite est choisi par l'utilisateur et l'affichage se fait horizontalement.

Exercice 3.10
Ecrivez un script qui calcule la factorielle [8] $n!$ d'un entier n entrée par l'utilisateur. Votre programme veillera à ce que l'entrée fournie soit un entier positif.

Exercice 3.11
Ecrivez un script qui affiche chacune des suites de symboles suivantes à l'aide de boucles :

(1)
```
*
**
***
****
*****
******
*******
********
*********
**********
```

(2)
```
**********
*********
********
*******
******
*****
****
***
**
*
```

(3)
```
**********
 *********
  ********
   *******
    ******
     *****
      ****
       ***
        **
         *
```

(4)
```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

8. La **factorielle** d'un nombre entier positif n, notée $n!$, correspond au produit de n avec tous les entiers inférieurs à n et ceci jusqu'à atteindre le nombre 1. Par exemple, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Exercice 3.12
Un triplet pythagoricien est un ensemble de trois entiers (a, b, c) pouvant représenter les trois côtés d'un triangle rectangle c'est-à-dire vérifiant l'égalité $c^2 = a^2 + b^2$. Ecrivez un script qui génère tous les triplets pythagoriciens (sans doublon) où a, b et c sont inférieurs à 50.

Exercice 3.13
Ecrivez un script qui prend en entrée une chaîne de caractères et retourne en sortie une copie de la chaîne entrée après avoir inséré un astérisque entre tous les caractères : `gaston` devra donner `g*a*s*t*o*n`.

Exercice 3.14
Ecrivez un script qui compte le nombre de « e » dans une chaîne de caractères entrée par l'utilisateur :

(1) avec l'instruction `while` (2) avec l'instruction `for`

Exercice 3.15
Ecrivez un script qui prend en entrée une chaîne de caractères et retourne en sortie une copie de la chaîne entrée après avoir inversé l'ordre des caractères qui la composent : `gaston` devra donner `notsag`.

(1) avec l'instruction `while` (2) avec l'instruction `for`

Exercice 3.16
Ecrivez un script qui renvoie le nombre de mots contenus dans une chaîne de caractères entrée par l'utilisateur :

(1) avec l'instruction `while` (2) avec l'instruction `for`

On considère comme *mots* les ensembles de caractères inclus entre des espaces.

Exercice 3.17
Ecrivez un script qui remplace tous les caractères « e » d'une chaîne de caractères par un astérisque.

Exercice 3.18
Ecrivez un script qui détermine si une chaîne de caractères donnée en entrée contient ou non le caractère « e ».

Exercice 3.19
Ecrivez un programme qui affiche tous les caractères de l'alphabet ASCII-étendu en regard de leur code correspondant. A partir de cette table, établissez la relation numérique simple reliant chaque caractère majuscule au caractère minuscule correspondant.

Exercice 3.20
Ecrivez un script qui prend en entrée un caractère et teste si celui-ci est un chiffre, une lettre ou un caractère spécial.

Exercice 3.21
Sachant que les codes *Unicode* attribués aux caractères minuscules de l'alphabet grec vont de 945 à 969, écrivez un programme qui affiche tous ces caractères.
(Le programme devra afficher : `Alphabet grec (minuscule) :` $\alpha\beta\gamma\delta\varepsilon\zeta\eta\theta\iota\kappa\lambda\mu\nu\xi o\pi\rho\varsigma\sigma\tau\upsilon\varphi\chi\psi\omega$)

Exercice 3.22
Ecrivez un script prenant en entrée une chaîne de caractères à traiter et un caractère à trouver. Le programme fournira en retour l'indice du premier caractère de ce type dans la chaîne. Veiller à ce que le programme renvoie une valeur particulière, par exemple -1, si le caractère recherché n'existe pas dans la chaîne traitée.
(Par exemple, en entrant « Juliette & Roméo » et « & », le programme devra retourner 9)

Exercice 3.23
Ecrivez un programme qui demande en boucle à l'utilisateur d'entrer des notes d'élèves. En entrant une note négative, l'utilisateur met fin à la boucle et alors le programme lui retourne le nombre de notes entrées, la note la plus élevée, la note la plus basse ainsi que la moyenne de toutes les notes entrées.

Exercice 3.24
Ecrivez un programme proposant à l'utilisateur 10 questions de livret à partir de deux entiers tirés au sort dans l'intervalle $[2; \texttt{Nmax}]$, où `Nmax` est l'entier le plus grand choisi par l'utilisateur. En cas de réponse erronée, le programme reposera la question tant que la bonne réponse n'aura pas été donnée. A la fin, le programme indique le nombre total d'erreurs réalisées.

Exercice 3.25
Ecrivez un programme qui détermine si un mot entré par l'utilisateur est un palindrome, c'est-à-dire un mot lisible de manière analogue de gauche à droite ou de droite à gauche.
(Le programme devra afficher : `sugus est un palindrome` et `banane n'est pas un palindrome`)