4

Instructions répétitives

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction while.

Ré-affectation

Nous ne l'avions pas encore signalé explicitement : il est permis de ré-affecter une nouvelle valeur à une même variable, autant de fois qu'on le souhaite.

L'effet d'une ré-affectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
>>> altitude = 320
>>> print(altitude)
320
>>> altitude = 375
>>> print(altitude)
375
```

Ceci nous amène à attirer une nouvelle fois votre attention sur le fait que le symbole *égale* utilisé sous Python pour réaliser une affectation ne doit en aucun cas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il est tentant d'interpréter l'instruction **altitude = 320** comme une affirmation d'égalité, mais ce n'en est pas une !

- Premièrement, l'égalité est *commutative*, alors que l'affectation ne l'est pas. Ainsi, en mathématique, les écritures **a = 7** et **7 = a** sont équivalentes, alors qu'une instruction de programmation telle que **375 = altitude** serait illégale.
- Deuxièmement, l'égalité est *permanente*, alors que l'affectation peut être remplacée comme nous venons de le voir. Lorsqu'en mathématique, nous affirmons une égalité telle que a = b au début d'un raisonnement, alors a continue à être égal à b durant tout le développement qui suit.
  En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre. Exemple :

```
>>> a = 5
>>> b = a          # a et b contiennent des valeurs égales
>>> b = 2          # a et b sont maintenant différentes
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément :

```
>>> a, b, c, d = 3, 4, 5, 7
```

Cette fonctionnalité de Python est bien plus intéressante encore qu'elle n'en a l'air à première vue. Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **c** (actuellement, **a** contient la valeur **3**, et **c** la valeur **5** ; nous voudrions que ce soit l'inverse). Comment faire ?

Exercice

4.1 Écrivez les lignes d'instructions nécessaires pour obtenir ce résultat.

À la suite de l'exercice proposé ci-dessus, vous aurez certainement trouvé une méthode, et un professeur vous demanderait certainement de la commenter en classe. Comme il s'agit d'une opération courante, les langages de programmation proposent souvent des raccourcis pour l'effectuer (par exemple des instructions spécialisées, telle l'instruction SWAP du langage *Basic*). Sous Python, *l'affectation parallèle* permet de programmer l'échange d'une manière particulièrement élégante :

```
>>> a, b = b, a
```

(On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction.)

Répétitions en boucle – l'instruction while

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle construite à partir de l'instruction **while**.

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, le-

quel 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[14].
- 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 **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 !")
```

[14]... du moins dans cet exemple. Nous verrons un peu plus loin qu'en fait l'exécution continue avec la première instruction qui suit le bloc indenté, et qui fait partie du même bloc que l'instruction **while** elle-même.

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 l'un en-dessous de l'autre. Pour obtenir plusieurs colonnes bien alignées à l'aide de tabulations, utilisez l'argument **"\t"**: **print(**a,**"\t"**,b,**"\t",**c**) .**

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.)

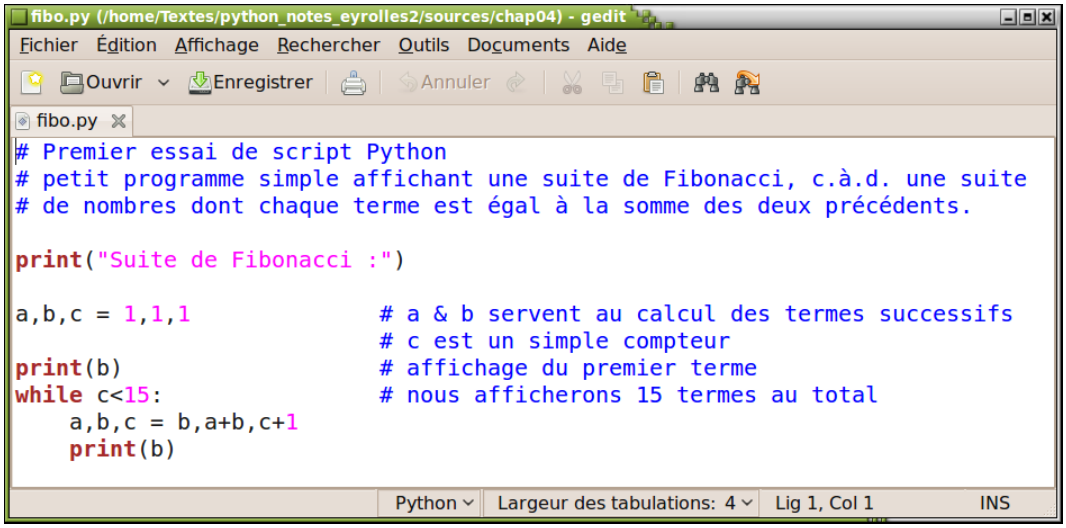
4.4 Écrivez un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.

Premiers scripts, ou comment conserver nos programmes

Jusqu'à présent, vous avez toujours utilisé Python *en mode interactif* (c'est-à-dire que vous avez à chaque fois entré les commandes directement dans l'interpréteur, sans les sauvegarder au préalable dans un fichier). Cela vous a permis d'apprendre très rapidement les bases du langage, par expérimentation directe. Cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions que vous avez écrites disparaissent irrémédiablement dès que vous fermez l'interpréteur. Avant de poursuivre plus avant votre étude, il est donc

temps que vous appreniez à sauvegarder vos programmes dans des fichiers, sur disque dur ou clef USB, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, vous allez désormais rédiger vos séquences d'instructions dans un *éditeur de texte* quelconque (par exemple *Kate*, *Gedit*, *Geany*, ... sous *Linux*, *Wordpad, Geany, Komodo editor,* ... sous *Windows*, ou encore l'éditeur incorporé dans l'interface de développement *IDLE* qui fait partie de la distribution de Python pour Windows). Ainsi vous écrirez *un script*, que vous pourrez ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur[15]. La figure ci-dessus illustre l'utilisation de l'éditeur *Gedit* sous *Linux (Ubuntu)* :



Par la suite, lorsque vous voudrez tester l'exécution de votre programme, il vous suffira de lancer l'interpréteur Python en lui fournissant (comme argument) le nom du fichier qui contient le script. Par exemple, si vous avez placé un script dans un fichier nommé « MonScript », il suffira d'entrer la commande suivante dans une fenêtre de terminal pour que ce script s'exécute :

```
python3 MonScript [16]
```

Pour faire mieux encore, veillez à choisir pour votre fichier un nom qui se termine par l'extension **.py**

Si vous respectez cette convention, vous pourrez aussi lancer l'exécution du script, simplement en cliquant sur son nom ou sur l'icône correspondante dans le gestionnaire de fichiers (c'est-à-dire l'explorateur, sous *Windows*, ou bien *Nautilus*, *Konqueror*... sous *Linux*).

---

[15] Il serait parfaitement possible d'utiliser un système de traitement de textes, à la condition d'effectuer la sauvegarde sous un format « texte pur » (sans balises de mise en page). Il est cependant préférable d'utiliser un véritable éditeur « intelligent » tel que *Gedit*, *Geany*, ou *IDLE*, muni d'une fonction de coloration syntaxique pour Python, qui vous aide à éviter les fautes de syntaxe. Avec *IDLE*, suivez le menu : File → New window (ou tapez Ctrl-N) pour ouvrir une nouvelle fenêtre dans laquelle vous écrirez votre script. Pour l'exécuter, il vous suffira (après sauvegarde), de suivre le menu : Edit → Run script (ou de taper Ctrl-F5).

[16] Si l'interpréteur Python 3 a été installé sur votre machine comme interpréteur Python par défaut, vous devriez pouvoir aussi entrer tout simplement : `python MonScript` . Mais attention : si plusieurs versions de Python sont présentes, il se peut que cette commande active plutôt une version antérieure (Python 2.x).

Ces gestionnaires graphiques « savent » en effet qu'il doivent lancer l'interpréteur Python chaque fois que leur utilisateur essaye d'ouvrir un fichier dont le nom se termine par .py (cela suppose bien entendu qu'ils aient été correctement configurés). La même convention permet en outre aux éditeurs « intelligents » de reconnaître automatiquement les scripts Python, et d'adapter leur coloration syntaxique en conséquence.

Un script Python contiendra des séquences d'instructions identiques à celles que vous avez expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues plus tard par vous-même ou par d'autres, *il vous est très fortement recommandé d'expliciter vos scripts le mieux possible, en y incorporant de nombreux commentaires* . La principale difficulté de la programmation consiste en effet à mettre au point des algorithmes corrects. Afin que ces algorithmes puissent être vérifiés, corrigés, modifiés, etc. dans de bonnes conditions, il est essentiel que leur auteur les décrive le plus complètement et le plus clairement possible. Et le meilleur emplacement pour cette description est le corps même du script (ainsi elle ne peut pas s'égarer).

> *Un bon programmeur veille toujours à insérer un grand nombre de commentaires dans ses scripts. En procédant ainsi, non seulement il facilite la compréhension de ses algorithmes pour d'autres lecteurs éventuels, mais encore il se force lui-même à avoir les idées plus claires.*

On peut insérer des commentaires quelconques à peu près n'importe où dans un script. Il suffit de les faire précéder d'un caractère #. Lorsqu'il rencontre ce caractère, l'interpréteur Python ignore tout ce qui suit, jusqu'à la fin de la ligne courante.

Comprenez bien qu'il est important d'inclure des commentaires *au fur et à mesure de l'avancement de votre travail de programmation*. N'attendez pas que votre script soit terminé pour les ajouter « après coup ». Vous vous rendrez progressivement compte qu'un programmeur passe *énormément de temps* à relire son propre code (pour le modifier, y rechercher des erreurs, etc.). Cette relecture sera grandement facilitée si le code comporte de nombreuses explications et remarques.

Ouvrez donc un éditeur de texte, et rédigez le script ci-dessous :

```
# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

a, b, c = 1, 1, 1            # a & b servent au calcul des termes successifs
                             # c est un simple compteur
print(b)                     # affichage du premier terme
while c<15:                  # nous afficherons 15 termes au total
    a, b, c = b, a+b, c+1
    print(b)
```

Afin de vous montrer tout de suite le bon exemple, nous commençons ce script par trois lignes de commentaires, qui contiennent une courte description de la fonctionnalité du programme. Prenez l'habitude de faire de même dans vos propres scripts.

Certaines lignes de code sont également elle-mêmes documentées. Si vous procédez comme nous l'avons fait, c'est-à-dire en insérant des commentaires à la droite des instructions correspondantes, veillez à les écarter suffisamment de celles-ci, afin de ne pas gêner leur lisibilité.

Lorsque vous aurez bien vérifié votre texte, sauvegardez-le et exécutez-le.

> *Bien que ce ne soit pas indispensable, nous vous recommandons une fois encore de choisir pour vos scripts des noms de fichiers se terminant par l'extension .py. Cela aide beaucoup à les identifier comme tels dans un répertoire. Les gestionnaires graphiques de fichiers (explorateur Windows, Nautilus, Konqueror) se servent d'ailleurs de cette extension pour leur associer une icône spécifique.* ***Évitez cependant de choisir des noms qui risqueraient d'être déjà attribués à des modules python existants*** *: des noms tels que math.py ou tkinter.py, par exemple, sont à proscrire !*

Si vous travaillez en mode texte sous *Linux*, ou dans une fenêtre *MS-DOS*, vous pouvez exécuter votre script à l'aide de la commande **python3** suivie du nom du script. Si vous travaillez en mode graphique sous *Linux*, vous pouvez ouvrir une fenêtre de terminal et faire la même chose :

```
python3 monScript.py
```

Dans un gestionnaire graphique de fichiers, vous pouvez en principe lancer l'exécution de votre script en effectuant un (double ?) clic de souris sur l'icône correspondante. Ceci ne pourra cependant fonctionner que si c'est bien **Python 3** qui a été désigné comme interpréteur par défaut pour les fichiers comportant l'extension .py (des problèmes peuvent en effet apparaître si plusieurs versions de Python sont installées sur votre machine – voyez votre professeur ou votre gourou local pour détailler ces questions).

Si vous travaillez avec *IDLE*, vous pouvez lancer l'exécution du script en cours d'édition, directement à l'aide de la combinaison de touches <**Ctrl**-**F5**>. Dans d'autres environnements de travail spécifiquement dédiés à Python, tels que *Geany*, vous trouverez également des icônes et/ou des raccourcis clavier pour lancer l'exécution (il s'agit souvent de la touche F5). Consultez votre professeur ou votre *gourou* local concernant les autres possibilités de lancement éventuelles sur différents systèmes d'exploitation.

Problèmes éventuels liés aux caractères accentués

Si vous avez rédigé votre script avec un logiciel éditeur récent (tels ceux que nous avons déjà indiqués), le script décrit ci-dessus devrait s'exécuter sans problème avec la version actuelle de Python 3. Si votre logiciel éditeur est ancien ou mal configuré, par contre, il se peut que vous obteniez un message d'erreur similaire à celui-ci :

```
File "fibo2.py", line 2
SyntaxError: Non-UTF-8 code starting with '\xe0' in file fibo2.py on line 2,
but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Ce message vous indique que le script contient des caractères typographiques encodés suivant une norme ancienne (vraisemblablement la norme ISO-8859-1 ou Latin-1).

Nous détaillerons les différentes normes d'encodage plus loin dans ce livre. Pour l'instant, il vous suffit de savoir que vous devez dans ce cas :

- soit reconfigurer votre éditeur de textes pour qu'il encode les caractères en Utf-8 (ou vous procurer un autre éditeur fonctionnant suivant cette norme). C'est la meilleure solution, car ainsi vous serez certain à l'avenir de travailler en accord avec les conventions de standardisation actuelles, qui finiront tôt ou tard par remplacer partout les anciennes.
- soit inclure le pseudo-commentaire suivant au début de tous vos scripts (obligatoirement à la 1e ou à la 2e ligne) :

```
# -*- coding:Latin-1 -*-
```

Le pseudo-commentaire ci-dessus indique à Python que vous utilisez dans votre script le jeu de caractères accentués *ASCII étendu* correspondant aux principales langues de l'Europe occidentale (Français, Allemand, Portugais, etc.), encodé sur un seul octet suivant la norme *ISO-8859-1*, laquelle est souvent désignée aussi par l'étiquette *Latin-1* .

Python peut traiter correctement les caractères encodés suivant toute une série de normes, mais il faut alors lui signaler laquelle vous utilisez à l'aide d'un pseudo-commentaire en début de script. Sans cette indication, Python considère que vos scripts ont été encodés en Utf-8[17], suivant en cela la nouvelle norme *Unicode* , laquelle a été mise au point pour standardiser la représentation numérique de tous les caractères spécifiques des différentes langues mondiales, ainsi que les symboles mathématiques, scientifiques, etc. Il existe plusieurs représentations ou *encodages* de cette norme, et nous devrons approfondir cette question plus loin[18], mais pour l'instant il vous suffit de savoir que l'encodage le plus répandu sur les ordinateurs récents est *Utf-8*. Dans ce système, les caractères standard (*ASCII*) sont encore encodés sur un seul octet, ce qui assure une certaine compatibilité avec l'ancienne norme d'encodage *Latin-1*, mais les autres caractères (parmi lesquels nos caractères accentués) peuvent être encodés sur 2, 3, ou même parfois 4 octets.

Nous apprendrons comment gérer et convertir ces différents encodages, lorsque nous étudierons plus en détail le traitement des fichiers texte (au chapitre 9).

Exercices

4.5 Écrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.

4.6* Modifiez le programme sur la suite de Fibonacci. Faites deux colonnes. Dans la première vous écrirez les termes t_k de la suite, avec t_k <100'000. Dans la seconde, vous écrirez le rapport t_k / t_{k-1}. Les deux colonnes doivent être alignées sur la gauche. Voici le début de ce vous devez obtenir :

```
1
1       1.0
2       2.0
3       1.5
5       1.66666666667
```

[17] Dans les versions de Python antérieures à la version 3.0, l'encodage par défaut était ASCII. Il fallait donc toujours préciser en début de script les autres encodages (y compris Utf-8).

[18] Voir page 129

8 1.6
13 1.625
Vers quel nombre converge le rapport t_k / t_{k-1}?

4.7 Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
Exemple : 7 14 21 * 28 35 42 * 49 ...

4.8 Écrivez un programme qui calcule les 50 premiers termes de la table de multiplication par 13, mais n'affiche que ceux qui sont des multiples de 7.

4.9 Écrivez un programme qui affiche la suite de symboles suivante :

*
**
***
****
*****
******
****