Vous trouverez ce document au format TeXmacs, avec l'extension `.tm`. Ce format nécessite l'installation du logiciel gratuit et opensource TeXmacs (à télécharger sur `http://www.texmacs.org`) mais il permet des fonctionalités supplémentaires, en particulier les réponses à quelques questions posées dans ce document et exercices, en dépliant une correction (chercher les petits triangles en fin de ligne)

Si vous essayez mais ne parvenez pas à installer TeXmacs, ne pas hésiter à venir me poser la question

Introduction à python

par Pierre-Henri Jondot

Le langage python a été inventé par Guido Van Rossum il y a un peu plus de 30 ans, nommé en hommage aux Monty Python, un groupe d'humoristes anglais que vous êtes probablement trop jeunes pour connaître (ils réalisaient des émissions pour la BBC dans les années 70, et on les connaît également par quelques films majeurs comme *Monty Python and the Holy Grail*, *the Life of Brian* etc…)

Bien entendu, le langage python n'est pas celui que comprend le microprocesseur de votre ordinateur, quel que soit celui-ci, et l'exécution d'un programme en python suppose donc qu'il soit traduit en des commandes que votre ordinateur saura alors exécuter. Dans le cas de python, cette traduction se fait à la volée : on dit que python est un langage interprété, par opposition à un langage compilé où le programme est traduit une fois pour toutes en langage machine que l'ordinateur est alors capable d'exécuter directement. (Ce n'est en fait pas tout à fait exact : il y a une très légère traduction d'un script python en un microcode qui ensuite est traduit à la volée pour l'exécution du programme)

Dans un programme compilé (exemples de langages qui sont compilés : C, C++, Swift, Rust) le programme s'exécute souvent plus rapidement que le même programme écrit dans un langage interprété, car dans ce dernier cas, il y a toujours une étape supplémentaire de traduction à l'exécution du programme.

Il y a toutefois un avantage à utiliser un langage interprété tel que python : il est immédiat de tester son code, de faire des modifications etc… La présence d'une console python permet également d'exécuter immédiatement des commandes python, d'invoquer une fonction que l'on vient d'écrire, ce qui rend à la fois l'apprentissage du langage et la mise au point de programmes bien plus simple.

Vous aurez besoin d'installer une distribution python sur votre ordinateur (ou du moins un ordinateur auquel vous avez accès) pour cet atelier. Plusieurs possibilités :

- Vous pouvez commencer avec Thonny : `https://thonny.org` qui est bien adapté à ses premiers pas avec Python. La distribution python fournie avec est assez minimaliste toutefois, comme l'est l'environnement de développement Thonny. (Cela dit, les bibliothèques qui manquent et dont on peut avoir besoin peuvent être installées à la demande, comme certains greffons qui étendent les fonctionalités de Thonny)

- L'environnement de développement Spyder (`https://www.spyder-ide.org`) est un peu plus complet, surtout qu'il est proposé ici en téléchargement avec une distribution python également très complète (le langage python a été enrichi de très nombreuses bibliothèques, et c'est la raison principale de son utilisation très fréquente dans de nombreux domaines, dont l'intelligence artificielle, le big-data)

- Vous pouvez aussi installer la distribution anaconda : `https://www.anaconda.com` elle aussi très complète, et fournie avec plusieurs environnements de développement python (dont spyder)

1 Première expérimentation avec python : types numériques

Les commandes qui suivent sont destinées à être exécutées dans un shell python, au sein de l'environnement Thonny ou Spyder (au choix). Le shell python se trouve dans Thonny en général en bas à gauche, et l'invite de commandes est `>>>` tandis qu'avec Spyder le shell python, en bas à droite en général (on peut réorganiser les fenêtres à sa convenance) dans une fenêtre qui peut présenter plusieurs onglets : assurez-vous que l'onglet « IPython Console » soit sélectionné. L'invite de commande d'une telle console prendra alors plutôt la forme `In [1]:` (tandis que la valeur de retour de la commande exécutée sera préfixée de `Out [1]:`)

1.1 Entiers (`int`)

Le type `int` en python sans surprise représente un entier. Contrairement à d'autres langages de programmation, l'occupation en mémoire d'un entier et de ce fait le nombre de valeurs représentables n'est pas limitée (si un entier signé est codé sur 32 bits, il peut prendre une valeur comprise entre, au sens large, -2147483648 et 2147483647, et s'il est codé sur 64 bits, il peut prendre une valeur comprise entre, au sens large, -9223372036854775808 et 9223372036854775807.

En python, on n'a pas cette limitation : la place mémoire destinée à représenter un entier s'adaptera à la taille de celui-ci, et le calcul sur de très grands entiers est donc possible, de base en python.

Les opérateurs agissant sur les entiers sont `+, -, *, //,**, %`. Faites l'essai de ces opérateurs dans la console python. Quelques exemples ci-dessous, mais ne vous en contentez pas.

```
>>> 2 + 3
>>> 2 * 10
>>> 23 // 10
>>> 23 % 10
>>> 13 ** 4
>>> 25 ** 100
```

Exercice 1. Etant donnés des entiers a et b, b étant strictement positif, que vaut `a % b` ?

Remarque 1. L'opérateur `/` est également défini, mais le résultat d'un calcul tel que `2/3` ou même `4/2` n'est pas un entier, mais un flottant. Si la valeur attendue est un entier, il convient d'utiliser plutôt `//`.

Un entier en python peut être de taille arbitraire (ou presque), comme les calculs précédents peuvent le laisser supposer. Sans surprise, le temps de calcul avec des entiers très grands est de plus en plus important.

1.2 Représentation binaire

Pour mieux comprendre certains calculs entiers (et pas seulement, on le verra) il peut être bon de connaître l'écriture binaire d'un entier. Une conséquence classique de la division euclidienne est que obtenir l'écriture en base b d'un entier revient à effectuer une suite de divisions euclidiennes successives, le chiffre des unités étant le premier reste obtenu. (Exemple : $123 = 12 \times 10 + 3$, puis $12 = 1 \times 10 + 2$...

Et la même chose en base 2 : $123 = 2 \times 61 + 1$, $61 = 2 \times 30 + 1$, $30 = 2 \times 15 + 0$, $15 = 2 \times 7 + 1$, $7 = 2 \times 3 + 1$, $3 = 2 \times 1 + 1$ et $1 = 2 \times 0 + 1$ et on en déduit que 123 en base 2 s'écrit 1111011.)

Python sait réaliser ce genre de conversion, du moins pour certaines bases. Essayez les commandes suivantes (et d'autres) :

```
>>> bin(123)
>>> int('10010010',2)
>>> int(bin(123),2)
```

Profitez-en pour aller lire la documentation (en anglais, bien sûr) de ces deux fonctions :

```
>>> help(bin)
>>> help(int)
```

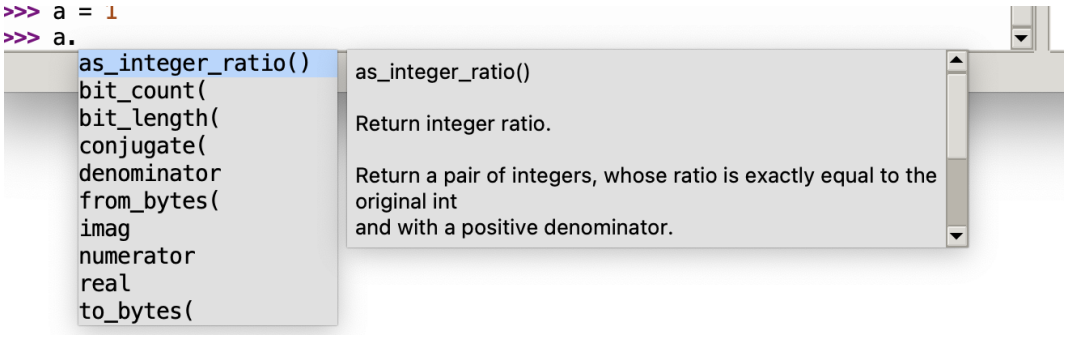
Notez déjà la première ligne de l'aide concernant `bin` : `Help on built-in function bin` qui indique que c'est une fonction interne à Python, laquelle est toujours disponible tandis que nous verrons que nombre de fonctions font partie de bibliothèques (on parle de modules en python) externes qu'il faudra importer avant de pouvoir faire appel à ces fonctions.

L'aide concernant `int` est beaucoup plus longue, et pour cause... C'est bien plus qu'une fonction, `int` est en fait ce qu'on appelle une classe, et les instances de cette classe sont appelés objets. Même si on n'abordera qu'en pointillé les principes de la programmation orientée objets, on ne pourra éviter de rencontrer des objets, car en python, tout est un objet !

L'autocomplétion de Thonny (ou de Spyder) permet aussi de parcourir par exemple toutes les méthodes d'un objet. Essayez ce qui suit :

```
>>> a = 1
```

puis dans la console, tapez : `a.` puis la touche de tabulation, et vous devriez voir apparaître :



(N.B. : lorsqu'on ajoute après un objet python le point . python s'attend à ce que cette expression soit suivie d'une méthode qui va agir sur cet objet, c'est-à-dire une fonction, ou d'une variable d'instance… Désolé pour le formalisme !)

Un exemple de méthode sur les entiers : `bit_length()` renvoie le nombres de chiffres binaires (bits) permettant l'écriture d'un entier, tandis que `bit_count()` renvoie le nombre de 1 dans ladite écriture binaire de l'entier. Ainsi :

```
>>> (123).bit_length()
7
>>> (123).bit_count()
6
```

1.3 Comparaisons, tests d'égalité

En programmation, on a souvent besoin de comparer des valeurs, de voir si une condition est satisfaite ou non, et décider selon le résultat du test de faire ceci ou cela.

Les opérateurs permettant de réaliser une comparaison sont les suivants : `<` , `>` , `<=` , `>=`, `==`, `!=` . Les quatre premiers se passent sans doute de commentaire, mais les deux derniers sont respectivement le test d'égalité et sa négation.

Attention au symbole `=` seul, sa signification est toute autre, on verra bientôt qu'il s'agit du symbole d'*affectation* qui permet de mettre un nom sur un objet, une valeur.

Le résultat d'un test est ce qu'on appelle un booléen, lequel vaut `True` ou `False`. Essayer :

```
>>> 1 < 2**3 < 128 > 5
```

A noter que ce qui précède aurait également pu être écrit ainsi : `(1 < 2**3) and (2**3 < 128) and (128 > 5)` ou même sans les parenthèses, car `and` est de précédence inférieure aux tests $<$ et $>$.

Outre la conjonction `and`, on peut combiner des valeurs booléennes avec `or` ou `not` dont on devine le sens.

Comme beaucoup de langages, python utilise l'évaluation paresseuse : par exemple pour évaluer `A and B` où `A` et `B` peuvent être des expressions complexes, il commence à évaluer `A` et si `A` est faux, alors `B` n'est même pas évalué, puisque la valeur de la conjonction est alors connue.

Ce peut nous permettre d'éviter quelques plantages. Un exemple simple avec le branchement conditionnel suivant : `if (y != 0) and (x / y > M)...` où il vaut mieux que la comparaison `x / y > M` ne soit pas effectuée lorsque `y` est nul...

(Dans l'expression `(y != 0) and (x / y > M)`, les parenthèses ne sont pas indispensables, mais améliorent la lisibilité)

Exercice 2. Il est une lourdeur souvent commise par les débutants en python, d'écrire quelque chose du genre :

```
if resultatBooléenDeMonTest == True:
    faire ceci
else:
    faire cela
```

Ceci est a priori (voir ce qui suit) tout à fait correct, mais, au fond, ceci exprime ce qu'en français on écrirait : s'il est vrai qu'il est vrai que, alors faire ceci sinon faire cela, ce qui est pour le moins lourd.

Mais ce qui n'est dans un premier temps qu'une lourdeur peut vite avoir des effets indésirables.

Le test suivant `'o' in 'mot' == True` s'évalue en `False`. Pourquoi ? (Vous pouvez bien sûr vous aider de la console python pour confirmer ou infirmer une conjecture, mais aussi relire l'intégralité du paragraphe…)

1.4 Nombres en virgule flottante

Dès lors qu'un nombre est introduit avec le séparateur décimal (le point) ou construit avec `float(`*une valeur*`)`, on obtient un objet de type `float` que l'on traduit en français par un nombre en virgule flottante.

On peut associer l'idée de la virgule flottante à la représentation scientifique des nombres décimaux. Par exemple, plutôt que d'écrire $1354,236$ on pourra écrire $1,354236 \times 10^3$. La représentation interne d'un `float` n'est pas très différente comme on le verra, si ce n'est que l'ordinateur travaille en base 2 (ainsi l'exposant s'écrirait plutôt 2^e pour un e convenable...)

Contrairement aux entiers, la place mémoire réservée au stockage d'un nombre en virgule flottante n'est pas extensible (elle est de 8 octets, c-à-d 64 bits) et un nombre réel est a priori représenté avec une valeur approchée dont la précision est d'environ 15 chiffres décimaux.

On retrouve les opérateurs `+, -, *, /, //, **, %`

Essayer ces opérateurs, avec les commandes de votre choix. Que se passe-t-il lorsqu'on combine un flottant et un entier ?

Exercice 3. Etant donnés `a` et `b` deux flottants, le second étant strictement positif, deviner la définition de `a // b` et `a % b`. (On prendra exemple sur leurs définitions lorsque a et b sont des entiers.)

Qu'advient-il si $b = 0$? Quelle relation y a-t-il entre `a // b` et `-a // -b`, entre `a % b` et `-a % -b` ?

Exercice 4. Par tâtonnements, en calculant des flottants tels que `2.0**n` avec plusieurs valeurs de n, chercher l'ordre de grandeur du plus grand flottant représentable, ainsi que du plus petit strictement positif.

Exercice 5. Un mystère : comment s'évalue le test d'égalité : `0.1 + 0.2 == 0.3` ? Comment expliquer ce phénomène ?

1.5 Module `math`

Les fonctions mathématiques usuelles ne font pas partie du noyau python (autrement dit, il ne s'agit pas de fonctions "built-in") mais font partie d'un module externe, nommé `math`.

Pour l'importer exécuter : `import math`

Pour voir ce que contient le module en question, exécuter : `dir(math)`.

Contrairement à ce qu'on pourrait penser, la commande suivante échoue :

```
>>> exp(1.0)
```

La raison est que si le module a bien été chargé, on accède aux fonctions et variables définies par le module en préfixant le nom de la fonction ou de la variable de `math`.

Ainsi, $\exp(1.0)$ peut s'obtenir grâce à la commande `math.exp(1.0)` (ce qui correspond également à `math.e`)

Calculez la valeur de $\sin(\pi)$, convertissez en degrés l'angle de $\frac{\pi}{2}$ radians. (Je vous laisse deviner quelle fonction du module `math` réalise cette conversion.)

Avouez que de devoir préfixer toutes les commandes par `math` devient un peu long ! Pour éviter cette corvée, on peut importer le module `math` ainsi :

```
>>> from math import *
```

Le caractère `*` indique de charger l'intégralité des variables et fonctions du module `math`.

On peut fort bien décider de ne charger que quelques fonctions. Par exemple :

```
>>> from math import sin, cos
```

Enfin, on peut aussi garder un préfixe, mais le changer (par exemple pour le raccourcir) :

```
>>> import math as m
>>> m.sin(m.pi)
```

2 Variables, Affectation

On est vite obligé de nommer des nombres, des objets pour s'y référer facilement. C'est le rôle de l'affectation, qui dans Python est réalisée par l'opérateur `=`. Quelques règles quant aux noms de variables acceptables en python :

- Un nom de variable doit commencer par une lettre ou bien le caractère de soulignement : `_`.
- Les autres caractères peuvent être des chiffres, des lettres ou encore le caractère de soulignement.
- Python fait la différence entre minuscules et majuscules, donc les noms de variables `truc` et `Truc` diffèrent...
- Python définit un certain nombre de noms qu'il réserve à son usage et qui ne sauraient être des variables modifiables par un programme (parmi lesquels : `as, assert, break, del, if, else, while, True, False, continue, and, or`...)

La commande `a = 1000` (l'exécuter) crée un objet entier de valeur 1000, ajoute le symbole `a` à l'espace des noms et fait pointer le nom `a` sur l'objet nouvellement créé.

Par la suite, à chaque fois que le nom `a` apparaîtra dans une expression python, `a` sera évalué en 1000. On fera attention à ce que, contrairement au symbole d'égalité mathématique, l'opérateur d'affectation n'est pas symétrique, et si l'instruction `a = 1000` a du sens en python, `1000 = a` n'en a pas. D'ailleurs, en langage naturel pour écrire un algorithme, on présente souvent l'affectation avec une flèche qui ne permet pas l'inversion : $a \leftarrow 1000$

Pour bien comprendre la signification profonde de l'affectation en python, penser qu'il ne s'agit guère que de poser une étiquette sur un objet pour le nommer et s'y référer plus tard. Contrairement à d'autres langages, si `x` est le nom choisi pour désigner tel ou tel objet, il n'est pas nécessaire de déclarer à l'avance le type d'objet que va représenter `x`, et de plus, rien n'interdit de réutiliser le nom `x` pour que `x` fasse référence à un autre objet, y compris d'un type différent.

Par exemple : en exécutant consécutivement les commandes `x = 2` puis `x = x / 3`, `x` est d'abord une étiquette vers un entier, puis devient une étiquette vers un nombre flottant.

Bref, la notion de « variable » est en python assez mal nommée, car au fond, une « variable » n'est jamais qu'un nom grâce auquel on accède à un objet donné, là ou dans bien d'autres langages de programmation, une variable correspond à un emplacement mémoire et où les affectations consécutives conduisent à changer la valeur stockée à cet emplacement.

Un exemple très simple pour comparer le C et Python :

En C :

```
int unEntier; // déclaration d'une variable de type entier. Selon le système d'exploitation,
// cette variable pourra occuper 4 ou 8 octets (32 ou 64 bits). En l'absence d'initialisation,
// la valeur que contient cette variable est inconnue.
unEntier = 12345 // la place mémoire qu'occupe notre variable est modifiée pour représenter
// désormais l'entier 12345
unEntier = unEntier + 1 // l'entier 12346 vient d'être calculé, et à l'emplacement de notre
// variable, la nouvelle valeur 12346 remplace l'ancienne.
```

En python :

```
unEntier = 12345 # nul besoin de déclaration ici. Un objet entier, de valeur 12345 est créé
# et on lui donne le nom de unEntier.
unEntier = unEntier + 1 # la valeur 12346 est dans un premier temps calculée, et forme un nouvel
# objet de type entier. Puis on fait en sorte que l'étiquette qu'est unEntier soit désormais
# associée à ce nouvel entier.
# Le premier entier 12345 n'est alors plus référencé, et, pour cette raison, l'emplacement mémoire
# qui lui correspondait peut alors être libéré pour être réutilisé plus tard.
```

Remarque 2. Si un petit nombre de noms sont réservés, il en est d'autres, pourtant importants, qui ne le sont pas, ce qui peut conduire à quelques surprises :

```
>>> print = 5
>>> print(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Tout n'est pas perdu après cette boulette qui nous a conduits à perdre une fonction importante : l'instruction `del print` permet à l'interpréteur python d'oublier l'affectation précédente, et le nom `print` est de nouveau associé à la fonction dédiée à l'affichage de texte dans la console.

Il n'en va pas de même, attention, avec les fonctions qui ne sont pas de base dans python, mais qui sont importées, comme pourraient l'être les fonctions de la bibliothèque `math` :

```
>>> from math import *
>>> sin(3)
0.14112000805986721
>>> sin = 3
>>> sin(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>> del sin
>>> sin(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'sin' is not defined
```

(Le plus simple, si on a besoin de retrouver la vraie fonction `sin` sera alors d'importer à nouveau le module `math`, ou seulement la fonction concernée ici, avec `from math import sin`)

On peut réaliser des affectations doubles :

```
>>> a = b = 10**3
```

En pratique ici, il est calculé 10^3 et créé un objet entier, vers lequel pointe `a` comme `b`. (On obtient la même chose en exécutant : `b = 10**3; a = b`)

On peut s'en rendre compte grâce à la fonction `id` qui renvoie un identifiant (unique) de l'objet qui lui est donné en argument : `id(a); id(b)`. (En pratique : on peut considérer que la valeur renvoyée par `id` correspond à une adresse mémoire et elle identifie uniquement l'objet référencé)

Si on exécute plutôt consécutivement les deux instructions `a = 1000` puis `b = 1000`, on peut s'attendre à ce que deux objets entiers soient créés, de même valeur 1000. (Rien n'interdit en la circonstance que `a` et `b` pointent encore sur le même objet.)

Expérimentez :

```
>>> a = 1000
>>> b = 1000
>>> id(a); id(b)
>>> a = b = 1000
>>> id(a); id(b)
```

Reprendre les commandes précédentes en remplaçant la valeur 1000 par une petite valeur telle que 10 ou 15... Que constate-t-on ? (Pour avoir une explication, déplier à gauche)

On peut enfin réaliser des affectations simultanées :

```
>>> a = 1; b = 2
>>> a, b = b, a
>>> a; b
```

Exercice 6. La commande `a, b = b, a` qui suppose que les variables `a` et `b` aient été définies, réalise l'échange de leurs objets : l'objet que référençait `a` est référencé par `b`, et réciproquement.

Comment réaliser cet échange sans utiliser le mécanisme d'affectations simultanées ?

Remarque 3. Dans l'affectation en python, ne pas oublier que le membre droit est dans un premier temps évalué, puis seulement alors les affectations ont lieu. Ainsi dans l'opération `a, b = b, a` ci-dessus, le membre droit est d'abord évalué en $2, 1$ puis les deux affectations `a = 2` et `b = 1` ont alors lieu.

3 Structures de contrôle

Lorsqu'une suite d'instructions doit être réalisée conditionnellement (branchement conditionnel) ou bien de manière répétée (boucle) il faut un moyen d'identifier ce qu'on appelle un *bloc* d'instructions. Beaucoup de langages utilisent des accolades pour entourer les instructions d'un tel bloc, mais python préfère utiliser l'*indentation*. C'est un retrait à la ligne de quelques caractères qui détermine le début et la fin d'un bloc d'instruction. On choisit souvent une indentation de 4 caractères. Bien sûr, un bloc d'instruction peut se trouver à l'intérieur d'un premier bloc, et ainsi de suite, ce qui encourage à ne pas trop multiplier l'imbrication de blocs...

3.1 Test conditionnel

La syntaxe est la suivante :

```
if condition1:
    instruction1
    ...
    instruction1
elif condition2:
    instruction2
    ...
```

```
    instruction2
else:
    instruction3
    ....
    instruction3
```

Un branchement conditionnel `if` peut ou non être suivi d'un ou plusieurs `elif` et/ou d'un `else:` final.

Dans l'exemple présent, si `condition1` s'évalue en `True` le bloc formé des instructions notées `instruction1` est exécuté est seulement celui-là. Si `condition1` s'évalue en `False`, alors `condition2` est évalué, et selon sa valeur, le bloc formé des instructions `instruction2` sera évalué ou celui formé des instructions `instruction3`.

3.2 Boucles conditionnelles

Syntaxe :

```
while condition:
    instruction
    ...
    instruction
```

On le devine, le bloc d'instructions qui suit la ligne `while condition` est exécuté *tant que* la condition s'évalue en `True` (peut-être 0 fois si cette condition se révèle fausse au premier passage, et peut-être une infinité de fois, du moins un nombre juste limité par la prochaine coupure de courant ou l'intervention autoritaire de l'utilisateur)

Exercice 7. Si `i` est un entier, la commande `print(i)` affiche dans la console celui-ci. Calculer la somme des 100 premiers carrés (de 1^2 à 100^2) et afficher celle-ci.

Même question avec la somme des carrés des entiers de 1 à 1000 qui sont multiples de 4 mais pas de 3 ni de 5.

Quelles sont les valeurs affichées par le script suivant ?

```
u = 1
v = 4
while v >= u:
    v = v + 1
    u = u + v - 1
    print(u)
print(v)
```

3.3 La fonction range

La fonction `range` est essentielle en python. Elle s'invoque avec 1,2 ou 3 arguments et renvoie ce qu'on appelle un itérateur, lequel va permettre d'obtenir un par un des entiers dans un certain intervalle, séparés chacun d'une même valeur (1 le plus souvent bien sûr).

L'appel le plus complet prend la forme `range(deb, fin, pas)`.

Si `pas` > 0, alors les entiers ainsi obtenus sont `deb` pour le premier, `deb + pas` pour le second, et le dernier est le plus grand entier de la forme `deb + k*pas` strictement inférieur à `fin` (extrémité supérieure non comprise donc).

Si `pas` < 0, on commence toujours par `deb`, puis `deb - pas` et ainsi de suite...

A noter qu'on peut invoquer `range` avec seulement deux arguments, sous la forme `range(deb, fin)`. Dans ce cas, le pas par défaut vaut 1 (et donc `range(deb, fin)` permettra d'obtenir un par un les entiers de `deb` compris à `fin-1`) et enfin, on peut aussi invoquer `range` avec un seul argument, à savoir `fin` (et dans ce cas `deb` prend la valeur par défaut de 0)

Ainsi `range(10)` permettra d'obtenir dans l'ordre 0,1,2,3,4,5,6,7,8 et enfin 9. (10 entiers consécutifs donc, à partir de 0)

A noter que pour faire apparaître tous les entiers qui seront générés par un `range`, on peut convertir celui-ci en une liste :

```
In[n]: list(range(5))
Out[n]: [0, 1, 2, 3, 4]
In[n+1]: list(range(2, 7, 2)):
Out[n+1]: [2, 4, 6]
In[n+2]: list(range(6, 3))
```

```
Out[n+2]: []
```

Question 1. Comment obtenir les listes suivantes (avec une commande de la forme `list(range(...))`)

1. `[2, 5, 8, 11]`
2. `[2, 3, 4, 5, 6]`
3. `[5, 4, 3, 2]`

3.4 Boucles `for`

Au fond, on n'a pas vraiment besoin d'un autre type de boucle, on peut tout faire avec des boucles conditionnelles. Cependant, il peut-être plus naturel d'utiliser une boucle `for` lorsque l'objet est de répéter une suite d'instructions un certain nombre de fois, connu à l'avance. En python, une boucle `for` permet d'appliquer un même traitement à tous les objets contenus dans une structure de données dite *itérable*. Un premier exemple d'itérable est la liste de valeurs (on va y revenir). Voici un exemple d'utilisation :

```
for i in [3, 7, 2, 5]:
    instruction
    ...
    instruction
```

Ici, le bloc d'instruction en retrait sera exécuté exactement 4 fois, et i prendra pour valeur 3 à la première exécution du bloc, 7 à la seconde, 2 à la troisième et 5 à la quatrième et dernière. (Il gardera d'ailleurs cette valeur par la suite)

Le plus souvent, ce n'est pas directement une liste qu'on itèrera, mais on utilisera la fonction `range` décrite dans le paragraphe précédent.

Exercice 8. Qu'affichent les petits programmes suivants :

```
for i in range(2, 11, 2):
    print(i)

for i in range(5, 124, -1):
    print(i)

for i in range(23, 0, -3):
    print(i)
```

4 Structures de données

4.1 Chaînes de caractères

Pour écrire une chaîne de caractère, on l'entoure d'apostrophes `'` ou de guillemets `"` :

```
s = 'une␣chaîne␣de␣caractères'
```

Si on choisit de délimiter une chaîne de caractères avec des apostrophes, et qu'on a besoin du caractère apostrophe dans ladite chaîne de caractères, on peut la préfixer d'un caractère \ (Alt-gr+8 sur un clavier pc, shift+option+/ sur un clavier mac) ainsi : `s = 'si on a besoin d\'une apostrophe'` et la même règle s'applique pour les guillemets.

On peut aussi délimiter une chaîne de caractères avec les symboles d'apostrophe ou de guillemets triplés. Dans ces conditions, la chaîne de caractères peut être écrite sur plusieurs lignes. Tout retour à la ligne fera alors partie de la chaîne, dont l'affichage par une fonction telle que `print()` s'étendra lui aussi sur plusieurs lignes. A noter qu'à l'aide du caractère \ à la fin d'une ligne, on peut reprendre l'écriture de la chaîne de caractères à la ligne suivante sans que le retour à la ligne soit pris en compte dans la ligne (et on peut aussi dans des cas exceptionnels utiliser ce caractère pour permettre d'écrire une ligne de code sur plusieurs lignes dans un éditeur python).

Au contraire, insérer `\n` dans une chaîne de caractères est rajouter un saut de ligne :

```
>>> s = 'une␣chaîne\nde␣caractères'
>>> print(s)
une chaîne
```

de caractères

Une chaîne de caractères (`str`) est un objet en python (à vrai dire, tout y est objet...) qui est immuable : on ne peut pas modifier une chaîne de caractères, on ne peut qu'en créer une nouvelle.

Quelques opérations classiques sur les chaînes (la liste est loin d'être complète) :

```
>>> s = 'une␣chaîne␣de␣caractères'
>>> len(s)
24
>>> s.capitalize()
'Une␣chaîne␣de␣caractères'
>>> s.find('de')
11
>>> s[11:13]
'de'
>>> s[2:10:2]
'ecan'
```

On vient d'utiliser ici, dans les deux dernières commandes, ce qu'on appelle une *tranche* (vous trouverez souvent l'anglicisme *slice*) : avec deux indices séparés de `:` on récupère la chaîne extraite depuis le caractère au premier indice jusqu'à celui, non compris, du second (comme range(a,b) commence à a et s'arrête à b non compris). Avec trois indice, le troisième donne le pas.

4.2 Tuples

Un tuple est une collection d'objets, séparés par des virgules, et entourée de parenthèses. Les parenthèses peuvent toutefois souvent être omises. Un tuple est, comme une chaîne de caractères, un objet immuable. Exemple :

```
>>> x = 'chaine', 3
>>> x
('chaine', 3)
>>> len(x)
2
>>> x[1]
3
>>> x[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Pour créer un tuple formé d'un seul terme (de peu d'intérêt certes, mais...), l'entourer de parenthèses n'est pas suffisant, car l'interpréteur python va supprimer ces parenthèses qu'il n'estime pas nécessaires. On écrira par exemple : `t = (2,)`. On peut même créer un tuple vide avec `t = ()`.

A noter que les tranches s'appliquent aux tuples de la même manière que pour les chaînes de caractères (ainsi qu'aux listes, voir le paragraphe suivant)

4.3 Listes

Une liste est comme un tuple une collection d'objets, séparés par des virgules, mais entourée de crochets `[o1, o2, ...]`. Une différence fondamentale est qu'une liste est un objet mutable : on peut ajouter, retirer et modifier des éléments très facilement.

Pour créer une liste, ou bien on entoure une collection d'objets des crochets ouvrant `[` et fermant `]`, ou bien on utilise le constructeur `list` avec un objet *itérable* (comme le sont les chaînes de caractères, les tuples, les listes elle-mêmes, les ensembles et dictionnaires...) :

```
>>> L = list('mot')
>>> L + L
['m', 'o', 't', 'm', 'o', 't']
>>> 3*L
['m', 'o', 't', 'm', 'o', 't', 'm', 'o', 't']
>>> L.append(0)
>>> len(L)
4
```

```
>>> L.pop()
0
>>> L
['m', 'o', 't']
>>> L.pop()
't'
```

Quelques opérations fondamentales avec les listes et points à connaître :

- Si une liste compte n éléments (ce qu'on obtient avec la fonction `len`), le premier élément est celui d'indice 0, le n-ième et dernier est d'indice $n-1$, mais python autorise également l'accès au dernier élément d'une liste L avec l'indice -1, l'avant-dernier avec l'indice -2...

- L'opérateur + agit sur les listes, en les concaténant. Ainsi, si `L1` et `L2` sont deux listes, `L1 + L2` est la liste formée des objets de `L1` suivis de ceux de `L2`.

- Si `n` est un entier et `L` une liste, `n * L` (ou `L * n`) correspond, sans surprise à `L + L + ... + L` (n fois)

- La méthode `append()` est très importante : étant donnée une liste `L`, exécuter `L.append(objet)` modifie la liste `L` pour y rajouter, à la fin, `objet`. Attention, la fonction `L.append(objet)` ne renvoie rien (plus exactement renvoie `None`)

- La méthode `pop()` elle aussi est très utilisée : elle retire le dernier objet de la liste et le renvoie. On peut rajouter un argument : l'indice de l'élément à retirer de la liste, ainsi `L.pop(0)` retire le premier élément de `L` et renvoie celui-ci. (Prendre conscience que cela prend plus de temps de retirer un autre élément du tableau que le denier : au fond, tous ceux qui le suivent sont alors décalés d'une case)

- Tout objet valide python peut prendre place dans une liste, dont les listes elles-mêmes (et on peut même faire d'une liste un des objets qu'elle contient !)

  Une conséquence de la mutabilité d'une liste (les méthodes `pop` et `append` par exemple en modifient le contenu) est que si deux variables référencent le même objet, toute modification apportée à l'objet référencé par la première variable est également répercutée sur l'objet référencé par la seconde, puisque c'est le même objet ! Exemple :

  ```
  >>> L1 = L2 = []
  >>> L1.append(5)
  >>> L2
  [5]
  >>> L2.pop()
  5
  >>> L1
  []
  ```

  Plus délicat à comprendre (venez poser des questions si besoin est !) :

  ```
  >>> L  = [[]] * 5
  >>> L[0].append(1)
  >>> L
  [[1], [1], [1], [1], [1]]
  >>> L[5] = [2]
  >>> L
  [[1], [1], [1], [1], [2]]
  ```

- Les tranches permettent d'obtenir d'une liste `L` la liste obtenue en extrayant certains objets de `L`. Exemple : si `L` est la liste `[1, 3, 5, 2, 4, 6, 3, 5, 7]`, `L[2:4]` est `[5, 2]`, `L[1:8:3]` est `[3, 4, 5]`, `L[3:]` est `[2, 4, 6, 3, 5, 7]`, `L[:3]` est `[1, 3, 5]`.

  (Exercice : devinez ce que sont `L[:3:-1]`, `L[:-1]`, `L[::-1]`)

- Si l'affectation `L2 = L1` conduit à ce que, si `L1` désigne une liste, `L2` référence le *même* objet, on peut souhaiter effectuer une copie de la liste `L1` afin que toute modification à l'avenir de l'objet référencé par `L2` ne modifie pas `L1`. Plusieurs manières de faire :

  - Avec le constructeur : `L2 = list(L1)`
  - Avec une tranche : `L2 = L1[:]`

- Avec la fonction `copy()` du module `copy` (`from copy import copy`) : `L2 = copy(L1)`

Petit souci cependant : les trois opérations présentées ci-dessus conduisent à ce que les objets `L1` et `L2` diffèrent et que toute modification de `L2` avec par exemple `append`, `pop` ne modifie pas `L1`. En revanche, à sa création, `L2` est constituée des mêmes objets que `L1`. Si ceux-là sont mutables, toute modification d'un tel objet présent dans `L2` modifiera aussi celui référencé dans `L1` :

```
>>> L1 = [[1], [2]]
>>> L2 = list(L1)
>>> L2.append(3)
>>> L2[1].append(2)
>>> L1 + L2
[[1], [2, 2], [1], [2, 2], 3]
```

Pas de solution simple, à part celle qui consiste à faire appel à la fonction `deepcopy` du module `copy` :

```
>>> L1 = [[0]]
>>> L2 = deepcopy(L1)
>>> L2[0].append(1)
>>> L1 + L2
[[0], [0, 1]]
```

4.4 Dictionnaires

Un *dictionnaire* est un tableau associatif, où à des clés sont associées des valeurs. Une clé est forcément un objet non mutable (valeur numérique, chaîne de caractères, tuple) tandis que la valeur associée peut être n'importe quel objet python.

Pour créer un dictionnaire vide, on peut écrire : `d = {}` puis on ajoute des clés ainsi : `d[cle] = valeur` où donc `cle` est un objet immuable (un nombre, une chaîne de caractères, un tuple) et `valeur` un objet python quelconque.

On peut également initialiser un dictionnaire avec quelques couples de clé et valeur pré-établies. Par exemple : `d = {0: [], 'clé' : 0}` où à la clé 0 est associée la liste `[]` et à la clé `'clé'` est associé l'entier 0.

Le mécanisme d'un dictionnaire est très astucieux. Grâce à celui-ci (fonctions de hachage en particulier) l'accès en lecture à la valeur associée à une clé est garantie (ou presque) se faire en temps constant (quand bien même donc le dictionnaire serait très grand, pourvu bien entendu que le dictionnaire dans son intégralité figure en mémoire vive...)

L'accès en lecture à la valeur associée à une clé qui n'est pas présente dans le tableau est une erreur et génère une exception de type `KeyError`. On peut donc utiliser le test suivant : `key in d` pour s'assurer que la clé `key` figure parmi les clés du dictionnaire `d`, et seulement alors aller lire la valeur correspondante avec `d[key]`.

A connaître aussi : `len(d)` pour connaître le nombre d'entrées (ou de clés) du dictionnaire `d`.

5 Fonctions

On a vu qu'une boucle permet de répéter une série d'instructions un nombre de fois prédéterminé (boucle `for`) ou tant qu'une condition est réalisée (boucle `while`). Une fonction permet de regrouper un bloc d'instructions pour s'y référer et y faire appel un nombre indéterminé de fois, à la demande. Une fonction peut être nommée ou non (voir les fonctions lambda un peu plus loin) admettre un certain nombre de paramètres et renvoie toujours une valeur (fût-elle la valeur `None` qui au fond correspond un peu à « rien » en Français...)

5.1 Syntaxe

```
def nomDeLaFonction(arg1: type, arg2: type, ..., argN: type) -> type:
    """Une␣plus␣ou␣moins␣longue␣chaîne␣de␣caractères␣qui␣décrit␣ce␣que␣fait␣la
    'fonction.␣Quelques␣exemples␣d'utilisation␣peuvent␣également␣être␣donnés␣:
    >>> nomDeLaFonction(val1,...)
    valeur de retour
    >>> nomDeLaFonction(autres paramètres)
    valeur de retour
    """
    instruction 1
    ...
    instruction n
```

```
    return valeurDeRetour
```

La directive `return` peut se trouver plusieurs fois dans le corps d'instructions d'une fonction (après un branchement conditionnel par exemple), mais dès lors qu'elle se trouve exécutée, alors on sort du corps de la fonction et l'exécution reprend après l'instruction qui avait conduit à l'appel de la fonction.

La chaîne de caractères présente après l'entête de la fonction est très utile, particulièrement si cette fonction sera utilisée par d'autres personnes que celui ou celle qui a écrit la-dite fonction (mais documenter son code est très utile aussi pour soi...)

La présence de quelques exemples peut également permettre de réaliser une vérification automatique de la fonction (qui doit donner les valeurs de retour attendues pour les arguments proposés en exemple) à l'aide du module `doctest`. Un exemple d'utilisation :

```
def fact(n: int) -> int:
    """calcule la factorielle de n
    >>> fact(0)
    1

    >>> fact(3)
    6
    """
    f = 1
    for i in range(n):
        f = f * i
    return f

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Le test `if __name__ == '__main__':` permet d'éviter que les tests soient réalisés lorsque le script est importé sous forme d'un module, mais en revanche les tests seront réalisés lorsque, par exemple, le script est exécuté pour lui-même par l'interpréteur courant dans python (touche F5).

Ici un test échoue et on obtient l'affichage suivant (tronqué) :

```
Failed example:
    fact(3)
Expected:
    6
Got:
    0
```

5.1.1 Arguments par défaut

La fonction `range` admet, on l'a vu, un, deux ou trois arguments. Voici comment on pourrait écrire une fonction `autreRange` qui renvoie une liste d'entiers numérotés à partir de 1 (contrairement à `range` qui non seulement numérote à partir de 1 mais aussi ne renvoie pas une liste (on peut obtenir une liste avec la syntaxe `list(range(...))`

```
def autreRange(deb:int, fin:int = None, pas:int = 1) -> list:
    """ Renvoie une liste d'entiers de deb à fin (extrémités comprises)
    séparés de pas. Avec un seul argument, le premier entier est 1.
    >>> autreRange(5)
    [1, 2, 3, 4, 5]

    >>> autreRange(2, -3, -2)
    [2, 0, -2]"""
    if fin == None:
        deb, fin = 1, deb
    i = deb
    L = []
    if pas > 0:
        while i <= fin:
            L.append(i)
            i = i + pas
```

```
    elif pas < 0:
        while i >= fin:
            L.append(i)
            i = i + pas
    return L
```

5.2 Variables locales et globales

Dès lors qu'une affectation est réalisée telle que `i = 1`, alors une variable *locale* est créée, ce qui indique que la portée de cette variable se cantonnera à la fonction proprement dite. Si depuis le point d'appel à la fonction, l'étiquette `i` était déjà définie, alors ce que référence `i` ne sera pas perdu. La valeur que référence `i` à l'intérieur de la fonction sera en revanche perdue en sortant de la fonction. Exemple :

```
def f() -> None:
    i = 2
    j = 1
i = 1
f()
print(i)
print(j)
```

Affichera :

```
1
.......
NameError: name 'j' is not defined
```

5.3 Fonctionnement des appels imbriqués de fonctions

Dans un script python, il y a le code « principal » formé de toutes les instructions qui ne prennent pas place au sein d'un bloc (à l'exception de certains blocs liés à un branchement conditionnel, une boucle etc…), et il y a les fonctions, qui peuvent être invoquées depuis le code principal, mais également depuis une autre fonction. Ainsi, une fonction peut être invoquée depuis le code principal, et à son tour elle peut en invoquer une autre, qui peut en invoquer une autre, et ainsi de suite.

Ce qui permet de reprendre l'exécution du code juste après la commande qui invoquait telle ou telle fonction, lorsque celle-ci a terminé son travail et renvoie, ou non, une certaine valeur, c'est ce qu'on appelle une *pile d'exécution*. Une pile est une structure de données qui s'apparente à une pile d'assiette ou de documents : le principe est qu'on peut rajouter des éléments sur la pile, un par un, et en retirer, un par un, toujours par le même côté (pour une pile d'assiettes, c'est plus simple par le dessus !)

A chaque appel de fonction, on empile la position actuelle dans le code qui s'exécute (afin de pouvoir y revenir plus tard) et on se sert également de la pile pour transmettre les arguments d'une fonction, définir le contexte local d'une fonction (les variables locales en dépendront, et seront alors détruites lorsque la fonction aura terminé son travail).

Si on invoque une fonction `a()` : on rajoute un étage à la pile, lequel disparaîtra lorsque la fonction `a()` aura terminé son travail. Si la fonction `a()` invoque à son tour une fonction `b()` un étage supplémentaire s'ajoute à la pile, qui se superpose donc au premier. (On ne pourra du reste retirer le premier étage que lorsque le second aura à son tour disparu).

Pour que la fonction `a()` puisse terminer son travail, il faudra bien sûr déjà que `b()` en ait terminé (ce faisant l'étage ajouté à la pile sera supprimé) et ait donc rendu le contrôle à la fonction `a()` et seulement alors la fonction `a()` pourra à son tour terminer son travail (le plus souvent à l'aide de la directive `return`)

Lorsqu'un programme plante, python nous montre l'état de la pile d'exécution, comme dans l'exemple suivant :

```
1: def a() -> None:
2:     return b()
3: def b() -> None:
4:     return c()
5: def c() -> None:
6:     return d()
7: print(a())
```

dont l'exécution conduit, sans grande surprise je crois, à une erreur, et voici (en partie) ce que la console montre alors :

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "plante.py", line 7, in <module>
    print(a())
  File "plante.py", line 2, in a
    return b()
  File "plante.py", line 4, in b
    return c()
  File "plante.py", line 6, in c
    return d()
NameError: name 'd' is not defined
```

On y apprend que la commande initiée ligne 7 échoue, car la fonction invoquée `a()` échoue elle-même, car la fonction qu'invoque `a()`, à savoir `b()` a échoué, puisque la fonction `c()` qu'a invoquée `b()` a échoué, et enfin `c()` a échoué car elle a tenté d'invoquer une fonction `d()` dont la définition manque.

Y a-t-il une limite au nombre d'appels imbriqués de fonctions ? Et bien oui... Déjà, on se doute que la pile d'exécution ne saurait être extensible à l'infini. Le nombre d'appels imbriqués possible peut être obtenu par la commande `sys.getrecursionlimit()` (qui avec une version récente de python doit valoir 3000 par défaut) et on peut en changer par la commande `sys.setrecursionlimit(n)`.

6 Entrées-sorties

6.1 La fonction `input`

Une interaction élémentaire avec l'utilisateur d'un programme en python passe par la fonction `input`. Elle admet un paramètre (une chaine de caractères) qui est affiché dans la console, et le programme s'interrompt en attendant une réponse de l'utilisateur. La réponse donnée est bien entendu la valeur retournée par la fonction `input`.

La chaîne peut alors être convertie en un entier avec le constructeur `int()`, en un flottant avec le constructeur `float()`...

6.2 Lecture-écriture dans un fichier (pas à connaître par coeur)

Dès lors que l'on a à travailler sur des données diverses et dont la taille peut être importante, il est malcommode de coder celles-ci au sein d'un script, pas plus qu'il n'est raisonnable d'attendre de l'utilisateur que ligne après ligne il introduise ces données en exécutant le programme.

Par ailleurs, il peut être souhaitable d'enregistrer sur disque (ou sur un support externe non volatil, comme une clé usb par exemple) le résultat d'un long calcul.

Ce qui suit est particulièrement adapté à la lecture et l'écriture de fichiers texte. Pour des données numériques autres (images, sons) des bibliothèques python spécialisées sont plus adaptées.

Pour ouvrir un fichier en lecture, on peut exécuter `f = open("nom_fichier")` ou, ce qui revient au même : `f = open("nom_fichier", "r")`.

A noter que le shell python qui exécute un script cherche dans un répertoire "courant" un fichier donné (en pratique, il s'agit le plus souvent du répertoire où figure le script en question), et que si le fichier à ouvrir ne s'y trouve pas, alors une erreur va survenir (`FileNotFoundError`). Pour connaître quel est le répertoire courant, on peut faire appel à la fonction `getcwd()` du module `os`. Pour changer le répertoire courant, on peut faire appel à la fonction `chdir()` toujours dans le module `os`, mais on peut aussi compléter le nom du fichier à ouvrir du chemin permettant d'y accéder. Par exemple, si le répertoire "courant" est `\home\user\python` et que le fichier à ouvrir se trouve plutôt dans `\home\user\donnees\` on peut ouvrir le fichier `fichier.txt` ou bien avec le chemin *absolu* : `f = open("\home\user\donnees\fichier.txt")` ou bien avec le chemin *relatif* : `f = open("..\donnees\fichier.txt")`.

Une fois un fichier ouvert en lecture, voici les différentes fonctions permettant d'accéder à son contenu :

- `f.read()` lit l'intégralité du fichier et retourne une (grande) chaîne de caractères.
- `f.readline()` retourne une ligne du fichier, sous la forme d'une chaîne de caractères. Attention, le caractère de retour chariot (`\n`) fait en général partie de cette chaîne (sauf s'il s'agit de la dernière ligne du fichier et que celle-ci est non vide)

  Si la chaîne vide `''` est retournée, c'est que la fin du fichier a été atteinte.
- Avec `f.readlines()` tout le fichier est lu est retourné sous la forme d'une liste de chaînes de caractères, chacune formant une ligne du fichier.

- On parcourt en fait souvent un fichier en lecture avec une boucle for :

```
for ligne in f:
    print(ligne)
```

Lorsque un fichier a été lu, ne pas oublier de "fermer" le fichier avec la commande `f.close()`.

Pour ouvrir un fichier en écriture, on utilisera plutôt la commande `f = open("nom_fichier", "w")`. Si un fichier du même nom existe déjà, il sera alors remplacé par celui-ci, et sinon il sera créé. Avec la commande `f.open("nom_fichier", "a")`, le fichier existant ne sera pas effacé, mais les nouvelles données écrites seront ajoutées à la fin du fichier existant (`a` comme append bien sûr...)

Une fois un fichier ouvert en écriture, voici quelques fonctions pour y écrire du contenu :

- `f.write(chaine)` écrit la chaîne de caractères `chaine`. Le caractère de retour chariot n'est pas ajouté à la fin (et si on n'en met pas, le fichier texte sera alors formé d'une seule et longue ligne...)

- `f.writelines(L)` ajoute les chaînes que contient l'itérable (liste, tuple, ensemble) `L`. Ici non plus, le caractère de retour chariot n'est pas ajouté.

Ici encore, et de manière encore plus criante que pour un fichier ouvert en lecture, il convient une fois le traitement terminé de "fermer" le fichier ouvert en écriture avec la commande `f.close()`. Ce n'est qu'alors que les données à écrire seront gravées sur le disque (et encore, selon le traitement réalisé par le système d'exploitation, elle ne le seront physiquement qu'après peut-être un certain retard, d'où la nécessité d'éjecter par exemple proprement un support de stockage tel qu'une clé usb avant de le retirer de l'ordinateur...)

Exercice 9. Ecrire une fonction `tableMultiplication(n)` qui écrit dans un fichier `table.txt` une table de multiplication de tous les $i \times j$ pour $1 \leqslant i, j \leqslant n$. Bien sûr, on tâchera d'aligner parfaitement verticalement les multiples de j...

Exercice 10. On suppose que `"fichier.txt"` contient des nombres (entiers et flottants). Ecrire une fonction `lire()` qui parcourt ce fichier et qui construit, et renvoie, une liste de listes des nombres qu'il contient. Par exemple, si le fichier est constitué de :

```
2 3.5 5

42

3.14159
```

alors la liste retournée sera `[[2, 3.5, 5], [42], [3.14159]]`. (Indication : on pourra utiliser la méthode `split` d'une chaîne de caractères, qui découpe celle-ci autour d'un séparateur, par défaut le symbole d'espacement. Ainsi `'4 6  3.5'.split()` renvoie la liste `['4', '6', '3.5']`.