Cours et Exercices de Python

niveau 1ère et Tale S

Cours et Énoncés

Ce cours Python n'est pas un chapitre individualisé du cours de mathématiques (que ce soit en classe de 1ère ou de Tale), selon les instructions officielles qui recommandent d'avoir des applications algorithmiques tout au long de l'année. Il s'agit d'ailleurs moins d'un chapitre d'algorithmique que d'un manuel de programmation en langage Python[1]. Les aspects purement algorithmiques s'y noient un peu dans les spécifications techniques du langage. Ce cours est donc une introduction au langage Python 3 (version la plus récente du langage) en particulier et à la programmation orientée objet en général. Nous n'avons pas la prétention d'écrire un manuel complet pour Python, mais si vous en cherchez, vous en trouverez de nombreux sur internet. J'aime bien le tutoriel de Kamel Naroun qui est à utiliser en ligne ou le manuel de Vincent Le Goff *Apprenez à programmer en Python. Développer en Python n'a jamais été aussi facile!* qui est un cours complet (de 418 pages) qui existe en version imprimée ou numérique (OpenClassrooms, collection Livre du Zéro). La Bible de Python reste tout de même l'ouvrage incontournable de Gérard Swinnen qui est libre : on peut le télécharger gratuitement (versions les plus récentes aux formats *pdf* et *odt*) ou en obtenir une version papier (chez Eyrolles).

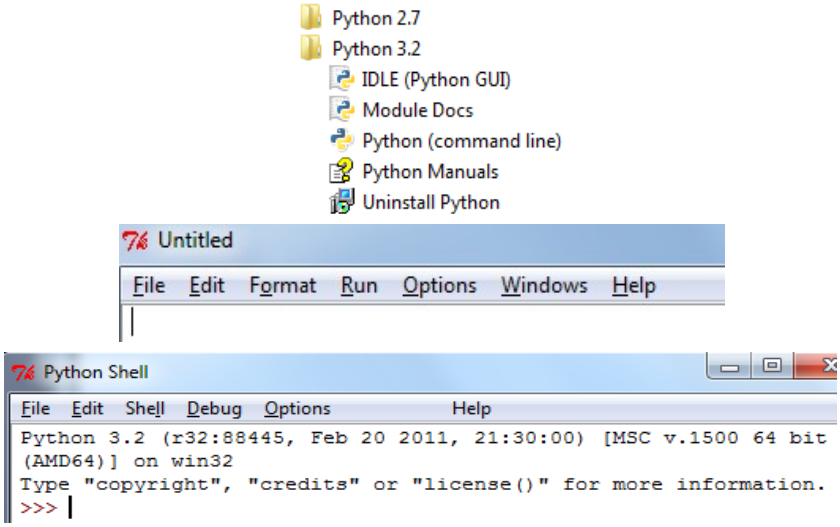
Parmi les dix premiers langages utilisés dans le monde en 2015, Python a été choisi pour être le langage enseigné en classes prépas. Il est relativement simple et suffisamment riche pour remplacer avantageusement, dès la classe de 1èreS, Algobox qui est un langage pédagogique, intéressant certes, mais limité et réservé de ce fait à l'initiation. La pratique de la programmation des calculatrices doit être maintenue puisque c'est la seule façon de programmer qui soit acceptée au Bac... Mais pour ces apprentissages, nous renvoyons les élèves à leurs manuels (manuel scolaire et manuel de la calculatrice) et au cours de seconde.

Contenu

1 Python n'est pas un acronyme. C'est un hommage aux *Monty Python*, une troupe d'acteurs humoristiques anglais des années 70 qui réalisa, entre autres, une série télévisée (le Flying Circus) et quelques films. Il faut croire que l'auteur du langage Python, le néerlandais Guido van Rossum, était *fan* des Monty Python car l'acronyme IDLE est le nom d'un des acteurs de la troupe et qu'une autre réalisation de van Rossum, le navigateur *Grail* est ainsi nommé d'après leur film *Holy Grail* (1975).

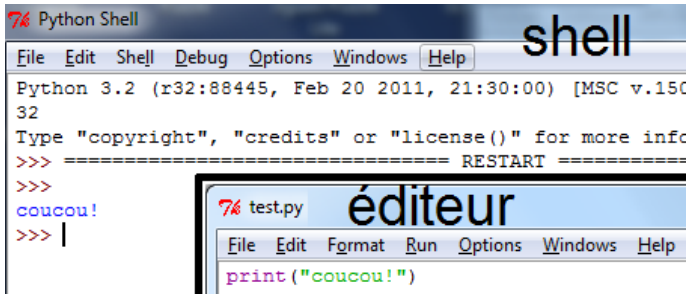
0. Installation de Python

L'ordre suivi dans cette présentation est assez arbitraire, mais nous souhaitons introduire les notions linéairement : au début ce qui est le plus important et le plus simple, les complications et les subtilités sont après. Il serait judicieux de lire ce cours en pouvant pratiquer et expérimenter. Pour utiliser Python sur un ordinateur, on commence par télécharger un installeur. Nous avons choisi une version assez récente et stable de Python (la version3.2, mais il en existe de multiples qui sont plus récentes, d'ailleurs pour les besoins de la partie 6 des questions nous avons basculé sur Python 3.4.3), il faut maintenant l'installer (sous windows en double-clickant sur l'installeur). Normalement, dans le dossier *Python 3.2* du menu « démarrer » vous devriez obtenir l'accès à *IDLE* (*Integrated DeveLopment Environment*) qui est un éditeur de Python avec coloration syntaxique, indentation automatique, etc. (*Python editor*) mais aussi un environnement d'exécution des programmes ou lignes de programme (*Python shell*). Vous pouvez utiliser uniquement le *shell*, au moins au début, et pour cela, il suffit de double-clicker sur *IDLE*. Vous devez obtenir quelque chose qui ressemble à ça, avec les trois chevrons >>> où le curseur attend vos instructions.



Je vous conseille d'*Envoyer Vers* ► *bureau (créer un raccourci)* le programme *IDLE* afin de le retrouver plus facilement. Par défaut, seul le *shell* s'ouvre, mais si vous modifiez les réglages (*Options, Configure IDLE, General*) en sélectionnant *At Startup Open Edit Window* vous devriez avoir les deux fenêtres qui s'ouvrent. Celle de l'éditeur (au dessus, avec le titre Untitled) ne contient pas exactement les mêmes onglets que celle du *shell* : il y a notamment l'onglet *Run* qui va permettre de lancer un programme (il s'exécutera dans le *shell*).

Si vous voulez enregistrer un programme, vous aller dans l'onglet *File* de l'éditeur et vous choisissez *Save As...* qui permet de nommer le fichier et de le ranger dans le dossier qui vous convient (dans ce cas, la fenêtre de l'éditeur porte le nom de votre fichier). Nommez votre fichier avec l'extension *.py* pour qu'il soit reconnu comme un fichier Python, par exemple *test.py*. Vous pouvez alors exécuter ce fichier. Si votre fichier *test.py* contient l'instruction *print*("coucou!") et que vous le lancez (*Run Module* ou F5 dans *windows*) à partir de l'éditeur, alors vous verrez s'afficher *coucou!* dans le *shell*. Dans ce qui suit, j'ai essayé de clarifier les choses en employant les italiques de préférences aux guillemets. Il se peut que les choix de mettre ou pas en italiques soient contestables. Signalez-moi les plus grosses aberrations et les erreurs que vous ne manquerez pas de trouver.



1. Notions de base

a) Type des variables, affectation, nommage

Une variable est une zone de la mémoire dans laquelle une valeur est stockée.
On stocke des nombres entiers, des nombres flottants (décimaux), du texte ou encore des résultats de test (Vrai ou Faux, notés *True* et *False*). Ces différents types de variable sont connus dés lors qu'on les déclare. Certains langage nécessitent des déclarations (Algobox), d'autres se contentent de reconnaître le type lors de l'initialisation. Ainsi, Python reconnaîtra que *A* est un entier si on écrit *A*=2 (type in*t*).
Si on écrit que *A*=2.0 (ou seulement *A*=2.) il en déduira que *A* est un flottant (type *float*). Noter le point décimal − une natation anglosaxonne − qui est la règle en informatique.
Si on écrit que *A*= "oui" (ou seulement *A*="") il en déduira que *A* est un chaîne de caractères (type *str*).
Si on écrit que *A*=*True* (ou *A*=*False*) il en déduira que *A* est une variable booléenne (type *bool*).
La division décimale en Python 3 est toujours de type flottant même si le résultat est un entier. Par exemple 3/2 donne 1.5 (c'est normal ici d'obtenir un nombre décimal, 3//2 donnerait 1) par contre 4/2 donne 2.0 qui est un flottant alors que 2 aurait pu aussi bien faire l'affaire ici. Il s'agit d'une conversion automatique.

Affecter une valeur à une variable *A* s'est associer une valeur à une variable donnée. L'initialisation est une forme d'affectation, qui fixe le type d'une variable en même temps que sa valeur initiale.
En pseudo-langage (un langage relativement codifié mais déconnecté de tout langage spécifique), on note l'affectation de 0 à la variable *A* : *A*←0 ou 0→*A* ou *A* prend_la_valeur 0 (Algobox) ou simplement *A*=0.
En Python, l'affectation se note avec le signe égal *A*=0 (on met 0 dans la mémoire *A*).
On peut affecter plusieurs variables dans une même instruction, en utilisant la virgule comme séparateur. Par exemple *A*,*B*=0,1 est une instruction qui initialise *A* à 0 et *B* à 1.

Attention, l'écriture *A*=*A*+1 a du sens pour une affectation : on affecte le résultat de la somme *A*+1 à la mémoire *A*. On remplace le contenu de *A* par l'ancienne valeur de *A* augmentée de 1.
En Python, on peut utiliser un raccourci pour cette instruction : *A*+=1.
Il est possible d'échanger des variables en une seule instruction avec Python. Par exemple *A,B*=*B,A*. Cette instruction permute *A* et *B* et évite d'avoir recours à une mémoire tampon *C*=*A,A*=*B,B*=*C*.

Les chaînes de caractères peuvent être additionnées (on dit concaténées), par exemple 'Papa'+' et '+'Maman' donne la chaîne 'Papa et Maman'. Plus étrange encore, elles peuvent être multipliées : 'Oui'+7*'i' donne la chaîne 'Ouiiiiiiii'. Par contre, n'essayez pas de faire des opérations invalides comme 'Papa'*'Maman' ou 3+'Non', car vous auriez un beau message d'erreur.

Avec les nombres, les opérations de base en Python sont les quatre opérations + - * et / (la typographie de ces caractères est celle du pavé numérique, c'est ce qui est utilisé en programmation, même si on préfère des affichages plus élégants comme + − × et ÷[2]). La division décimale se note / en Python 3 et la division euclidienne // (quotient entier). Le reste de la division euclidienne de A par B est noté A%B (opération *modulo*), l'élévation à la puissance N est notée avec deux étoiles A**N ou bien aussi *pow*(A,N).

Les nombres flottants (appelés aussi, de façon mathématiquement impropre, nombre réels) sont des valeurs approchées de nombres réels. Ce sont, intrinsèquement, des nombres décimaux, avec une notation scientifique pour les grands ou petits nombres 2**-25 renvoie 2.9802322387695312e-08 (le e est pour e*xposant*, c'est la contraction de $\times 10^{e}$) alors que $2^{-25} = 0{,}000000029802322387695312 5$ (la différence porte sur le 18ème chiffre). Pour des raisons techniques, la mémoire stocke le nombre 1.8 sous une forme qui n'est pas forcément exacte (comme 1.7999999999999999 ou 1.8000000000000001) ce qui peut perturber certains résultats. En général, on arrive néanmoins à ce que l'on souhaite. Dans les versions 3 de Python le nombre 1.8 est affiché correctement (même s'il continue à être stocké de façon un peu incorrecte).

2 Pour obtenir les caractères − × et ÷ on peut utiliser leur code unicode : u2212 u00D7 et u00F7. Par exemple, en tapant *print*(*str*(7), '\u00D7', *str*(8) ,'=',*str*(7*8)) vous obtiendrez 7 × 8 = 56 qui est mieux que 7 * 8 = 56 (le pire étant 7 x 8 = 56).

Les nombres entiers sont parfois limités en taille par certains langages de programmation. Le langage *Java* (un des plus employés) possède quatre sortes d'entiers : les *bytes* (stockés sur 1 octet, soit compris entre -128 et 127), les *short* (stockés sur 2 octets, soit compris entre -32768 et 32767), les *int* (stockés sur 4 octets, soit compris entre -2147483648 et 2147483647) et les *long* (stockés sur 8 octets, soit compris entre -9223372036854775808 et 9223372036854775807). Si on veut travailler avec des nombres entiers illimités, c'est aussi possible (mais moins simple, il faut passer par une classe spéciale). En Python, les nombres entiers sont d'une seule sorte : illimités, toujours.
Ainsi, 123^{45} est un très grand nombre, mais cela ne pose pas de problème à Python qui affiche bravement :
123**45=11110408185131956285910790587176451918559153212268021823629073199866111001242743283966127048043

Pour nommer une variable, on doit essayer d'être assez explicite pour que la relecture soit facilitée (la recherche éventuelle d'erreur dans un algorithme est l'étape la plus difficile). En Python, on a le droit aux chiffres, aux lettres minuscules et majuscules (attention : pas d'accents!) ainsi qu'à l'*underscore*. On doit faire commencer le nom d'une variable par un caractère minuscule. Appelons les indicateurs *i*, les nombres *n*, un taux de tva *taux* ou encore *taux_de_tva*, le numérateur et le dénominateur d'une fraction *n* et *d* ou mieux *num* et *denom*, et s'il y en a plusieurs *num1*, *denom1*, *num2* et *denom2*...

b) Instructions, blocs d'instructions

Les instructions sont les parties élémentaires d'un algorithme. Une affectation est une instruction mais il y en a d'autres, par exemple l'instruction conditionnelle *si <test> alors <bloc d'instructions> sinon <bloc d'instructions>* est une instruction. On voit qu'on peut insérer un bloc d'instructions (un ensemble de plusieurs instructions), dans une instruction. Généralement, les instructions sont séparées les unes des autres par un retour à la ligne ou bien un signe de ponctuation particulier (une virgule ou un point-virgule) et les blocs d'instructions sont marqués par des accolades (c'est le symbolisme que nous employons la plupart du temps dans nos algorithmes en pseudo-langage qui est copié de la syntaxe du Java).

En Python, les mots d'une instructions sont séparés par des espaces. Le retour à la ligne marque la fin d'une instruction. Si l'on veut créer un bloc d'instructions, il faut que les instructions soient à un même niveau d'indentation (un même retrait, ou décalage avec la marge de gauche) : on peut être collé à cette marge, ou bien espacé de celle-ci d'une tabulation (4 espaces), ou bien de deux espaces... L'important pour un bloc est d'avoir le même nombre d'espaces avec la marge (sinon il y a une erreur qui empêche l'exécution).
Le retour au retrait initial après une indentation est la manière de mettre fin au bloc d'instructions.
Par exemple, l'algorithme suivant qui calcule le suivant dans une suite Collatz :

Si x est impair alors {x est remplacé par $3x+1$; x remplace le maximum s'il le dépasse}
Sinon x est divisé par 2

est traduit en Python par la structure :

```
if x%2 != 0 :
        x =3x+1
        if x>max : max=x
else : x = x/2
```

Ce qui est remarquable est l'absence de signe indicateur du début et de la fin du bloc, hormis le retour à la ligne et l'indentation (ou plus exactement les changements d'indentation). Dans les autres langages, un mot est souvent réservé à cet effet : *end* ou *IfEnd ou endIf* (ici, ou *endWhile* après un *While*), ou *Fin_Si* (Algobox) ou bien des parenthèses ou des accolades (comme en Java). Une autre remarque : après le test, il y a deux points en Python. Cela marque la fin du test avec un caractère obligatoire, alors que les parenthèses autour du test sont facultatives. Ces deux points indiquent que le bloc d'instructions qui doit être exécuté si le test est vrai (*True*) suit. Dans notre exemple, vous noterez une deuxième utilisation du *if*, qui dispose l'instruction devant être exécutée si le test est vrai juste après les deux points (de même pour l'instruction suivant le *else*). Si il y a un bloc d'instructions, il faut utiliser le retour à la ligne avec rupture d'indentation pour marquer le début et la fin du bloc.

c) Entrées/Sorties

L'utilisateur d'un algorithme doit parfois effectuer des **entrées** : ce peut être un nombre, ou plusieurs, ou bien un texte. On peut aussi envisager (mais nous gardons cette possibilité pour plus tard) de lire un fichier de données, par exemple un tableau (fichier .xls) ou bien un fichier structurés par lignes avec des séparateurs de données sur une même ligne. Certains algorithmes lisent même des données en continue.

La lecture des données ponctuelles est la façon la plus simple d'entrer des données : des valeurs sont introduites au moment de l'exécution, avec un message à destination de l'opérateur (« Quelle est la longueur ? » ou, de plus en plus simplement « Entrer la longueur. » ou « Longueur= » ou « L ? »). Cette instruction de lecture s'écrit dans l'algorithme « Lire la valeur de L » ou « Lire L » ou « Entrer L ».

Dans un programme, ces instructions d'entrée s'écrivent ?→A (calculatrices Casio), *input* A (calculatrices TI), *enter* A ou *read* A (les langages informatiques utilisent généralement l'anglais). En Python, cette instruction s'écrit *input*(), avec le message destiné à l'opérateur entre les parenthèses (avec des guillemets simples, doubles ou même triples pour un texte trop long pour entrer sur une ligne). La valeur qui est entrée par l'opérateur est alors toujours interprétée comme de type 'texte'. Si on veut que le type soit modifié, il faut forcer le type à changer, en utilisant une instruction de *cast* (traduction) comme *int*() pour convertir le texte en entier. Par exemple :

n = *input*("Entrez un nombre entier :")
n = *int*(*n*)

On peut contracter les deux instructions en une seule :

n = *int*(*input*("Entrez un nombre entier :")).

Si vous oubliez de convertir l'entrée en entier, les instructions suivantes provoqueront une erreur à l'exécution (car l'entier 2 ne peut pas être converti en texte, d'où le message *can't convert 'int' object to str implicitly*).

n = input("Entier :")
n=*n*+2

Pour information, la conversion de la variable en flottant (réel) s'écrit *n* = *float*(*n*).

Remarque : si l'opérateur entre une valeur qui n'est pas un entier et que le programme demande de convertir la valeur en entier, la réponse sera une erreur avec le message *invalid literal for int() with base 10:* '123v\r'. Pour obtenir ce message, nous avons entré 123v.

À la fin de l'exécution d'un algorithme (ou au cours de son exécution), on a souvent besoin de donner des résultats, de les afficher ou de les écrire dans un fichier. Toutes ces opérations sont des **sorties**. Une instruction qui permet d'effectuer une sortie du programme vers l'utilisateur ou vers un fichier externe est appelée une écriture. On peut noter cette instruction « affichage » ou « enregistrement » pour faire ressortir le moyen utilisé pour l'écriture (écran ou fichier), dans un programme ce sera des mots-clefs comme *output*, *write* ou *print* qui seront utilisés ou bien des symboles comme le triangle ◢ du langage Casio.

En Python, une écriture se fait généralement avec la syntaxe *print*(), avec, à l'intérieur de la parenthèse, une données ou plusieurs séparées par une virgule. Les données à afficher sont, soit des bouts de texte (avec guillemets), soit des valeurs de variables. Par exemple, *print*("En degrés Celcius, cela fait : ", *celcius*) affiche un texte suivi de la valeur contenue dans la mémoire *celcius* (bien sûr, il faut avoir déclaré cette variable au préalable). Si on veut afficher les coordonnées d'un point et son nom, d'une façon classique M(a;b) on écrit : *print*("M (",a, " ; ",b,")") où les cinq morceaux sont concaténés.

On peut formater un texte avec des données variables en Python en concaténant les différents morceaux dans une variable de texte. Pour ce faire, on utilise + qui est l'opérateur de concaténation. Mais si on veut insérer des variables numériques, il faut les convertir en texte avec la fonction de conversion *str*(). Notre affichage de M(a;b) se fera ainsi (on suppose que a et b sont des variables contenant des nombres) :

s="M ("+*str*(a)+" ; "+*str*(b)+")"
print(s)

Vous pensez que ce n'est pas très intéressant puisqu'on peut faire sans la variable s, et plus simplement. Certes, mais n'oublions pas cette méthode qui permet d'utiliser toutes les méthodes d'action sur les chaînes de caractères (on peut, par exemple, connaître la longueur d'une chaîne en écrivant *len*(s)). Il est aussi

possible de se passer de la variable s en écrivant directement *print*("M ("+*str*(a)+" ; "+*str*(b)+")"). Il y a une autre différence entre les deux méthodes : *print*("gauche", "droite") et *print*("gauche"+"droite"). Dans le 1er cas, Python met automatiquement un espace entre les mots et on obtient « gauche droite » alors que dans le 2d cas, il n'y en a pas et on obtient « gauchedroite ».

Une chaîne de caractères est une liste en Python, on peut accéder à n'importe lequel des caractères en désignant son rang, par exemple, si s contient la chaîne "Mr Moutou" alors s[0]="M", s[1]="r", s[2]=" "… L'inconvénient est qu'on ne peut pas modifier un des caractères directement car une chaîne est immutable en Python : s[1]="e" provoque l'erreur *'str' object does not support item assignment*.

Une autre façon de construire une chaîne de caractères en Python : la transformation d'une liste t en texte par l'instruction *join*(t). Une liste permet de stocker des éléments disparates comme du texte et des nombres. Mais si on veut créer une chaîne de caractères à afficher à partir des éléments de la liste, les nombres doivent être convertis en texte.

```
t=["M (", str(a), ";", str(b), ")"]
print(" ".join(t))
```

La fonction *join*() ne s'exécute pas sans un texte devant suivi d'un point (ici on a mis un espace " ") car il s'agit d'une méthode qui insère la chaîne de caractères (éventuellement vide) entre les éléments de la liste. L'avantage de cette méthode est la possibilité de modifier les différents éléments en les désignant par leur rang. Ici, t[0]="M (", t[1]="*contenu numérique de* a"). Si on veut remplacer ce "*contenu numérique de* a", on peut faire, par exemple, t[1]=*str*(3.14), ce qui ne provoquera pas d'erreur.

Signalons qu'on peut créer une liste à partir d'une chaîne de caractères en utilisant la fonction *list*(). Par exemple, l'instruction t=*list*("M(0;1)") conduit au tableau t=["M", "(", "0", ";", "1", ")"] et on peut alors en modifier les valeurs, par exemple en faisant t[2]=*str*(a).

À côté de toutes ces méthodes pour afficher une chaîne de caractères en Python, il en existe une autre, plus technique, qui permet d'intégrer des variables numériques sans les convertir, un peu comme la 1ère méthode présentée avec *print*(elt1,elt2,...). Cette fois, on commence par écrire le texte comme il vient en remplaçant les variables par des accolades. Pour notre point et ses coordonnées, cela donne s="M ({};{})". Pour indiquer les variables, on ajoute à cette chaîne incomplète un formatage : s.*format*(a,b). Sans la variable s et avec l'instruction d'affichage, cela s'écrit *print*("M ({};{})".*format*(a,b)).

d) Fonctions et procédures

Les fonctions sont des blocs d'instructions dédiées à l'exécution d'un calcul qui sont extraits de l'algorithme principal afin de séparer les difficultés, d'éviter de réécrire plusieurs fois le même calcul et de pouvoir être réemployé tel quel ailleurs (dans un autre algorithme) ou modifié. Pour être plus concret, si on a besoin de calculer l'image d'un nombre par une fonction à plusieurs endroits d'un algorithme (par exemple dans la recherche d'un maximum), on va externaliser ce calcul au moyen d'une fonction. L'avantage est multiple, on pourra effectuer le même algorithme pour une autre fonction en modifiant juste la fonction, on pourra employer cette fonction dans un autre algorithme...

Il existe une possibilité simple : les fonctions *lambda* qui se déclarent avec la syntaxe *lambda var*:<*exp*> où l'expression *exp* dépend de la variable *var*. Pour l'utiliser comme on fait habituellement avec les fonctions numériques, on l'affecte à un nom de fonction, disons *f* pour être original, et on l'utilise alors tout simplement en écrivant *f*(*x*). Par exemple, on peut calculer $y=4x^3-3x^2+2x-1$, en écrivant :

```
f=lambda x:4*x**3-3*x**2+2*x-1
print(f(3)) renvoie 86
```

Mis à part cette possibilité simple, les fonctions plus élaborées se déclarent avant de pouvoir être utilisées avec la syntaxe *def* <*nom de la fonction*>() : <bloc d'instructions>. Les instructions qui suivent le symbole « : » doivent respecter la règle de l'indentation. Il y a généralement une valeur de retour (ou plusieurs) qui est renvoyée à l'algorithme avec le mot-clef *return*. On peut réaliser des fonctions simples :

```
def f(x):
    y=4*x**3-3*x**2+2*x-1
    return y
```

La variable *x* est un paramètre donné en entrée de la fonction (on dit un argument) et *y* est une variable

locale (qui n'existe pas en dehors de cette définition). Ces deux variables se nomment indépendamment des noms employés dans l'algorithme principal. Par exemple, on pourra écrire dans celui-ci

```
b=f(a)
if(b>0) : b=math.sqrt(b)
else : b=math.sqrt(-b)
```

La valeur numérique contenue dans *a* est « envoyée » à la fonction qui le reçoit sous l'appellation locale de *x*. Noter que dans cet exemple, s'il s'agit de déterminer la valeur absolue du nombre *f*(*a*), il suffit d'écrire *b*=*abs*(*f*(*a*)).

Les fonctions en Python ne sont pas limitées aux fonctions à une variable. On peut entrer autant d'arguments que l'on veut dans une fonction, même aucun (dans ce cas on laisse les parenthèses vides). De même, la valeur de retour d'une fonction peut ne pas exister : il suffit de ne pas mettre d'instruction *return*. La fonction fait quelque chose sur les données (par exemple un affichage) sans renvoyer de valeur. On appelle alors ce type de fonction une *procédure*, mais, fondamentalement le mécanisme des fonctions et des procédures est le même. Une fonction peut également renvoyer plusieurs valeurs qu'on sépare alors par une virgule.

```
def trans(a,b,c,d):
    return a+c,b+d
```

Cette fonction *trans* est un peu simpliste mais opérationnelle. Si on envoie les coordonnées d'un point M(a;b) et celles d'un vecteur $\vec{v}$ (c;d), elle renvoie les coordonnées de l'image M' de M par la translation de vecteur $\vec{v}=\overrightarrow{MM'}$. Voici maintenant une fonction qui ne prend pas d'argument et renvoie une lettre de l'alphabet au hasard. Cette fonction pourrait être utile si on voulait créer un « petit bac », ce jeu si prisé par les (jeunes) élèves. Au passage, on verra comment s'agrandit l'espace des fonctions disponibles.

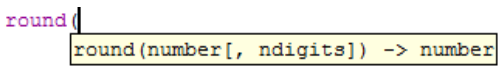
```
from random import choice
def lettre():
    return choice('abcdefghijklmnopqrstuvwxyz')
```

L'instruction *from random import choice* (cette fonction choisit au hasard un des caractères de la chaîne fournie en argument) importe la fonction *choice* du module *random*. Il y a de nombreuses fonctions, toutes en rapport avec les tirages aléatoires, dans le module *random*. Donnons un exemple plus élaboré qui utilise, cette fois, la fonction *randint* du module *random* (cette fonction choisit au hasard un entier compris entre les deux entiers fournis en argument). Supposons que l'on veuille fabriquer un petit programme pour s'entraîner aux tables de multiplications. On va avoir besoin d'une fonction qui génère une opération en choisissant deux entiers entre 2 et 9, l'opération renvoyée par notre fonction est une chaîne de caractères et la fonction doit aussi renvoyer le résultat sous la forme d'un entier (pour évaluer le résultat).

```
from random import randint
def table() :
    a=randint(2,9)
    b=randint(2,9)
    c=a*b
    return str(a)+'\u00D7'+str(b)+'=?',c
```

La valeur absolue et la racine carrée sont deux fonctions *abs*() et *sqrt*() qui sont prédéfinies dans Python. La 1ère est toujours disponible au même titre que les opérations élémentaires. La 2de est placée dans une bibliothèque, le module *math.* Lorsqu'on en a besoin, il faut l'importer de ce module, par exemple en écrivant *from math import sqrt.* On y accède alors simplement, en tapant *sqrt*(2) par exemple qui renvoie 1.4142135623730951.

La bibliothèque *math* contient de nombreuses autres fonctions comme *cos*, *sin*, … (fonctions trigonométriques), *ceil*, *floor*, *round*… (parties entières supérieure et inférieure, arrondis) ou des constantes (*pi*, …). Il y a de très nombreuses bibliothèques dans Python. Vous devrez sans doute vous tourner vers un catalogue en ligne qui détaille la liste des fonctions disponibles et leur syntaxe. Lorsque vous tapez le début d'une fonction dans IDLE, que ce soit dans le *shell* ou dans l'éditeur, une aide vous est donnée rappelant la syntaxe et les options éventuelles. Par exemple, *round*(5/6) renvoie l'entier 1 mais il est possible d'arrondir avec *ndigits* chiffres de précision (*ndigits*<18) *round*(5/6,3) renvoie 0.833. En tapant *help*(fonction), il y a davantage de détails à cette aide.



```
>>> help(round)
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

Pour utiliser une fonction d'un module, on peut importer le module en entier en écrivant *import math*. On utilise alors les fonctions du module en faisant précéder le nom de la fonction par le nom du module suivi d'un point *math.round*(*math.sin*(*math.pi*/6),3) renvoie ainsi 0.5 alors que *math.sin*(*math.pi*/6) renvoie le flottant mal arrondi 0.49999999999999994 (le dernier chiffre aurait été un 9 cela passerait, mais un 4!).

Une autre façon de procéder pour importer les fonctions d'un module est d'écrire *from math import* * où l'étoile indique que l'on veut importer toutes les fonctions du module. Dans ce cas, on n'a plus besoin de rappeler le nom du module (*round*(*sin*(*pi*/6),3) suffit donc). L'inconvénient est qu'on risque de se retrouver avec des noms de fonctions en conflit (une fonction d'un module importé qui porte le même nom qu'une de nos fonctions ou qu'une fonction d'un autre module importé). Pour éviter ces conflits, on peut choisir de n'importer que ce qui est nécessaire, par exemple *from math import sin, cos, pi, sqrt.* C'est la solution que nous avions employé dans notre exemple plus haut.

Vous avez remarqué que l'instruction d'importer est écrite avant la fonction qui emploie la fonction importée. C'est une structure générale que l'on peut retenir pour un programme : on écrit d'abord ce qui s'emploie ensuite. L'ordre logique est donc 1-les importations, 2-les fonctions, 3-le programme lui-même. Voici un petit exemple de programme qui respecte cet ordre et réalise le programme d'entraînement aux tables de multiplications

```
6×2=?12
Bravo!
Score=1/1. Taper 0 pour sortir
8×3=?24
Bravo!
Score=2/2. Taper 0 pour sortir
3×3=?9
Bravo!
Score=3/3. Taper 0 pour sortir
2×8=?166
Faux! La réponse est 16
Score=3/4. Taper 0 pour sortir
4×8=?32
Bravo!
Score=4/5. Taper 0 pour sortir
5×2=?10
Bravo!
Score=5/6. Taper 0 pour sortir0
```

```
from random import randint
def table() :
    a=randint(2,9)
    b=randint(2,9)
    c=a*b
    return str(a)+'\u00D7'+str(b)+'=?',c
total_question,bonne_reponse,continue =0,0,'1'
while continue!='0':
    question,reponse=table()
    essai=int(input(question))
        if(essai!=reponse) : print("Faux! La réponse est "+str(reponse))
        else :
            print("Bravo!")
            bonne_reponse+=1
    total_question+=1
    continue=input("Score="+str(bonne_reponse)+"/"+str(total_question)+". Taper 0 pour sortir"))
```

Dans une fonction le mot-clef *return* peut se situer à différents endroits (derrière des tests) mais si on attend que la fonction renvoie un nombre, il faut veiller à renvoyer toujours un nombre. Après le mot-clef *return* on sort de la fonction et ce qui suit n'est pas exécuté. Donnons un exemple :

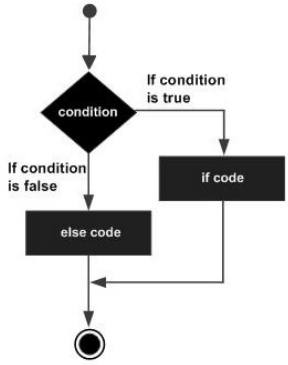
```
def nbr_jours_mois (m,a):
    if m==1 or m==3 or m==5 or m==7 or m==8 or m==10 or m==12 : return 31
    else :
        if m!=2 : return 30
        else :
            if a%4!=0 or (a%100==0 and a%400!=0) : return 28
    return 29
```

Cette fonction renvoie le nombre de jours dans un mois *m* de l'année *a* (sur quatre chiffres à cause des années 1600, 2000, etc. qui sont bissextiles alors que 1800 et 1900 ne le sont pas). Dans cet exemple, il y a plusieurs *return* soumis à des conditions (voir leur structure au paragraphe suivant), mais à la fin (si aucune condition n'a été satisfaite), on renvoie tout de même un nombre qui n'a pas besoin d'être soumis à une condition (un *else* final) puisque c'est la valeur par défaut.

2. Tests et boucles

a) Instructions conditionnelles et tests

L'instruction conditionnelle *si* <*test*> *alors* <*bloc* 1> *sinon* <*bloc* 2> dont nous avons déjà parlé (pour les blocs d'instructions) est le modèle d'instruction qui demande l'évaluation d'un test pour exécuter une action. Si le résultat du test est *True* une certaine action est faite (les instructions du bloc 1), sinon, si le résultat du test est *False* une autre action est faite (les instructions du bloc 2). Parfois, il n'y a pas de bloc 2 (la partie commençant à *sinon* est facultative), mais comme le programme passe à l'instruction suivante, il se peut que celle-ci ne soit exécutée que si le test est *False*, ce qui revient à un sinon (voir l'exemple de notre fonction *nbr_jours_mois*).



Un test simple consiste souvent en une comparaison : on évalue une égalité *A==B* (noter le double symbole égal qui réalise cette comparaison, *A=B* réaliserait une affectation), une différence *A!=B* (le symbole ! signifiant le contraire de ce qui suit), un ordre strict *A<B*, *A>B* ou large *A≤B*, *A≥B*.
Le test 3==3.0 renvoie *True*, *math.pi*==3.14 renvoie *False,* 1/3≥2/6 renvoie *True* (il y a égalité) et 1/3>2/6 renvoie *False*. On peut faire des opérations qui seront évaluées prioritairement, et insérer des variables dans un test : après *a*=1, le test *a*+1==2 renvoie *True* (a vaut toujours 1). On peut affecter le résultat d'un test dans une variable : avec *b*=*a*+1==2 la variable *b* contient le résultat (*True*) du test.

Parfois, on doit effectuer des tests plus complexes, en utilisant les opérateurs logiques *ou* et *et*. Ces opérateurs s'écrivent *or* et *and* en Python (comme dans de nombreux langages informatiques). Vous devez savoir que *A et B* est vrai quand *A* et *B* sont vrais simultanément alors que *A ou B* est vrai si l'un ou l'autre, ou les deux sont vrais (c'est le *ou inclusif*). Les opérateurs *ou* et *et* n'ont pas la même priorité : le *et* est prioritaire (exécuté en premier).
Dans le doute, on peut toujours mettre des parenthèses mais il faut savoir que, par exemple, le test *A==B et C==D ou E==F et G==H* équivaut à (*A==B et C==D*) *ou* (*E==F et G==H*).
Un exemple plus élaboré, extrait du livre du Zéro de V. Le Goff, qui détermine si une année est bissextile :

```
if annee%400 == 0 or (annee%4 == 0 and annee%100 != 0) : print("Année bissextile.")
else : print("Année non bissextile.")
```

En plus de cette possibilité habituelle d'employer le *et*, la structure d'encadrement *A<X<B* est acceptée par Python qui accepte donc des tests enchaînés comme celui-là (à condition d'être correct logiquement, un test comme *A<X>B* par exemple sera refusé car incorrect formellement) en lieu et place de *A<X et X<B*. La différence n'est pas grande direz-vous, mais l'écriture est simplifiée et l'exécution plus économique car s'il y a un calcul pour *X*, il n'est exécuté qu'une seule fois.

```
if 2000<=annee<2100 : print("Année du XXIe siècle")
else : print("Siècle inconnu")
```

L'instruction conditionnelle s'enrichit d'une possibilité supplémentaire en Python : le mot-clef *elif* (contraction de *else if*) peut suivre un *if*<test>. Le *elif* doit être suivi d'un test et il peut y en avoir plusieurs après un *if (*alors qu'il ne peut y avoir qu'un seul *else*) avec un *else* unique facultatif à la fin. La structure conditionnelle complète s'écrit donc :

```
if    <test 1> : <bloc 1>
elif <test 2> : <bloc 2>
elif <test 3> : <bloc 3>
...
else : <bloc n>.
```

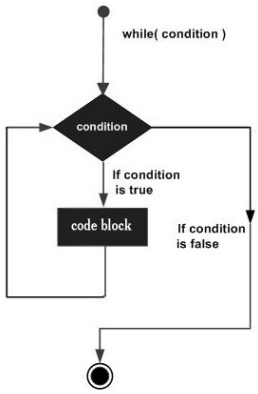
Alors que certains langages disposent d'une structure conditionnelle supplémentaire − la structure *switch* − qui permet d'examiner les différentes valeurs possibles d'une variable et d'exécuter des actions spécifiques dans chaque cas, le langage Python s'en passe. Il est toujours possible de traduire la suite d'instructions :
switch (*A*){case 0:<*bloc* 1>*break ;*case 1:<*bloc* 2>*break ;*case 2:<*bloc* 3>*break ; ... ;default:*<*bloc n*>}

Il suffit d'écrire une suite de *elif* où le *else* final joue le rôle de l'action par *default* du *switch* :

```
if    A==0 : <bloc 1>
elif A==1 : <bloc 2>
elif A==2 : <bloc 3>
...
else : <bloc n>.
```

L'inclusion dans une liste peut être testée avec le mot-clef *in,* par exemple 3 *in* [2,3,4] renvoie *True* et 3 *not in* [4,5,6] renvoie *True* aussi. L'opérateur logique not qui s'écrit aussi ! Peut être employé avec cet autre mot-clef is dans une instruction telle que *if* <test> *is not True* qui ressemble à de l'anglais.

b) La boucle While



Une boucle *while* permet de répéter une suite d'instructions *tant que* une condition est vraie. Sa syntaxe en Python est simple *while* <*test*> : <*bloc* d'instructions>. Dans le bloc d'instructions, il doit y en avoir qui assurent une fin à la boucle, sans quoi elle se prolongerait indéfiniment... Par exemple, *while True* : *A=A+1* est une magnifique boucle infinie qu'il faut arrêter en stoppant le programme de l'extérieur. On peut créer une pseudo-boucle infinie avec une possibilité de sortie dans le bloc d'instructions en créant, par exemple, un compteur *A* que l'on incrémente à chaque passage dans la boucle. S'il s'agit de définir une fonction et de renvoyer le nombre *B*, on peut écrire :

```
A,B=0,0
while True :
        B=fonction(B)
        A=A+1
        if A==100: return B
```

Les esprits vifs feront remarquer qu'alors le test est inutile puisqu'il suffit d'écrire :

```
A,B=0,0
while A<=100 :
        B=fonction(B)
        A=A+1
```

La valeur contenue dans *B* à l'issue des 101 passages dans la boucle sera la même dans les deux cas. Cette boucle n'est pas très intéressante et sera judicieusement remplacée par une boucle *for* (voir plus loin). On trouvera un meilleur exemple d'application avec l'algorithme d'Euclide pour lequel, on ne connait pas à l'avance le nombre de tours de boucle qu'il faut. Voici une version par soustractions successives de cet algorithme qui détermine PGCD(*a*;*b*).

```
def pgcd (a,b):
        while (a!=b) :
                if a>b : a=a−b
                else : b=b−a
        return a
```

Nous avons montré comment interrompre une boucle *while* (avec l'instruction *return*) à l'intérieur d'une fonction. Il existe deux autres possibilités, qui sont utilisables dans tous les contextes et pour les deux sortes de boucles (*while* et *for*) : l'instruction *break* qui stoppe la boucle la plus intérieure (s'il y a imbrication des boucles) et provoque la sortie de cette boucle (le programme continue avec ce qui suit la boucle) ; l'instruction *continue* qui court-circuite ce qui suit après, dans la boucle, et provoque le tour de boucle suivant. Ces instructions sont parfois inévitables.

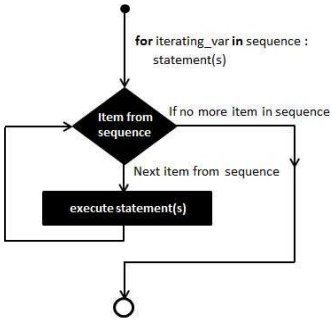
```
Entrez un nombre entier entre 0 et 1000 :250
C'est plus, recommencez!
Entrez un nombre entier entre 0 et 1000 :500
C'est moins, recommencez!
Entrez un nombre entier entre 0 et 1000 :350
C'est plus, recommencez!
Entrez un nombre entier entre 0 et 1000 :420
C'est plus, recommencez!
Entrez un nombre entier entre 0 et 1000 :490
C'est moins, recommencez!
Entrez un nombre entier entre 0 et 1000 :450
C'est plus, recommencez!
Entrez un nombre entier entre 0 et 1000 :465
C'est plus, recommencez!
Entrez un nombre entier entre 0 et 1000 :475
C'est plus, recommencez!
Entrez un nombre entier entre 0 et 1000 :485
Bravo! Vous avez trouvé après  9  étapes.
Voulez-vous tenter votre chance (oui:o,non:n):n
Bye bye!
```

Écrivons maintenant un petit programme en Python qui utilise un peu toutes les notions abordées jusque là. Réalisons le jeu « devine un nombre » : on demande à l'opérateur de saisir un nombre entre 0 et 1000 jusqu'à ce qu'il trouve la valeur

aléatoire choisie au départ, guidé par l'algorithme, qui lui indiquera à la fin son score (le nombre d'étapes nécessaires).

```
from random import randint
jeu="o"
while jeu=="o"
    secret,essai,etape=randint(0,1000),-1,0
    while essai!=secret
        essai=int(input("Entrez un nombre entier entre 0 et 1000 :"))
        if(essai<secret) : print("C'est plus, recommencez!")
        elif(essai>secret) : print("C'est moins, recommencez!")
        etape+=1
    print("Bravo! Vous avez trouvé après ",etape," étapes.")
    jeu=input("Voulez-vous tenter votre chance (oui:o,non:n):")
```

c) La boucle For



Dans la plupart des langages, cette structure de boucle est la plus simple à utiliser car il n'y a pas de test. Il en va de même en Python à un détail près : il va falloir employer, donc comprendre, ce qu'on appelle une liste. Nous avons déjà introduit la notion de liste en parlant des chaînes de caractères qui sont considérées comme des listes en Python. La boucle *for* de Python est l'équivalent du *for each* que l'on trouve dans d'autres langages et se traduit par : pour chaque élément d'un ensemble *L* faire <bloc d'instructions>.

La boucle *for* est employée lorsqu'on connait le nombre d'itérations à réaliser.
Par exemple, si on veut calculer la somme $P_n=\sum_{k=1}^{n}\frac{1}{k}$ pour une valeur donnée de *n,* il faut répéter *n* fois une suite particulière d'instructions. Pour $n=5$ on a $P_5=\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}=\frac{137}{60}\approx 2{,}28333\ldots$ et pour P_6 il faut ajouter $\frac{1}{6}$ à P_5. Il suffit donc d'ajouter, à chaque tour de boucle, dans une variable *P* initialisée à 0, l'inverse d'un compteur parcourant la liste des entiers de 1 à *n*. Voici donc notre algorithme :

```
Entrer n, Affecter 0 à P
Pour i allant de 1 à n {Affecter P+1÷i à P}
Afficher P
```

La boucle *for* correspond à la ligne centrale. On doit faire varier le compteur *i* entre 1 et *n*.
Cela peut s'écrire avec une boucle *while* en initialisant le compteur *i* à 1 et en utilisant un test approprié :

```
i=1
while i≤n :
    P=P+1÷i
    i=i+1
```

Mais avec une boucle *for* cette partie se simplifie puisqu'on n'a pas besoin d'incrémenter le compteur ni de l'initialiser (tout cela est réalisé par l'instruction *for*) : *pour I allant de 1 à N*{P=P+1÷*i*}. Généralement, l'incrémentation du compteur *i* est égale à 1 par défaut, mais certains langages nécessitent que l'on donne l'incrément (aussi appelé le pas).

En Python, on écrit la boucle *for* en utilisant la liste *range*(1,*n*+1) qui contient les entiers de 1 à *n*. C'est comme si on prenait les entiers de l'intervalle [1;*n*+1[où *n*+1 est exclu.

```
for i in range(1,n+1) :
    P=P+1÷i
```

La notation *range*(*a,b*) correspond aux entiers de *a* à $b-1$, mais on peut utiliser la notation simplifiée *range*(*n*) pour avoir tous les entiers de 0 à $n-1$ (il y en a *n*). Cet algorithme pourrait donc aussi s'écrire :

```
for i in range(n) :
    P=P+1÷(i+1)
```

On n'a pas besoin de s'occuper de l'incrémentation du compteur puisque celui-ci décrit toutes les valeurs de la liste [1,2,3,4, … ,*n*] ou, dans le 2ème algorithme, de la liste [0,1,2,3, ... , $n-1$].
On a dit plus haut qu'une chaîne de caractères est une liste en Python. On peut donc accéder à n'importe

lequel des caractères en désignant son rang, par exemple si s="bonjour" on a s[0]="b", s[1]=s[4]="o", s[6]="n". En fait, tout se passe comme si on avait la liste s=["b","o","n","j","o","u","r"]. Cette particularité de Python permet d'employer des mots pour contrôler une boucle *for*. Vous me direz « à quoi ça peut bien servir ? » et vous aurez raison de vous interroger jusqu'à ce que, un jour peut-être, cela vous soit nécessaire. On peut donc écrire quelque chose comme ça : *for* lettre *in* "bonjour" : *print*(lettre) qui affichera une lettre par ligne, ou bien, pour afficher un petit agenda :

```
for jour in ["Lun","Mar","Mer","Jeu","Ven","Sam","Dim"] :
    print(jour, end=" ")
    for heure in range(8,20) :
        print(heure+"h ....", end=" ")
    print()
```

On remarquera ici le saut de ligne final déclenché par *print*() alors que le reste est affiché sur une même ligne grâce à l'instruction *print*("xxx", *end*=" ") qui ne déclenche pas de saut de ligne en fin d'affichage. Une autre subtilité de l'affichage : *print*(heure,"h ....", *end*=" ") met un espace entre les morceaux affichés 8 h ..... alors que *print*(heure+"h.....", *end*=" ") ne met pas d'espace 8h .....

Voici donc notre petit agenda :

```
Lun 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
Mar 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
Mer 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
Jeu 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
Ven 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
Sam 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
Dim 8h .... 9h .... 10h .... 11h .... 12h .... 13h .... 14h .... 15h .... 16h .... 17h .... 18h .... 19h ....
```

Donnons un autre exemple qui peut être fait à titre d'exercice : afficher la table des multiples d'un nombre *n* compris inférieurs à *max* qui ne soient pas multiples de *m*. On veut définir une fonction *multiples*(*n*,*m*,*max*) qui affiche sur une ligne les multiples avec "...." lorsque le nombre est multiple de *m* et sur une deuxième ligne la liste des facteurs supprimés. Par exemple, on veut que *multiples*(7,5,100) affiche d'abord : 7 14 21 27 .... 42 49 56 63 .... 77 84 91 98, puis ensuite : Facteurs supprimés : 35 70

```
def multiples(n,m,maxi):
    supprime=list()
    for facteur in range(maxi//n) :
        if (facteur+1)*n%m!=0 : print((facteur+1)*n, end=" ")
        else :
            print("...", end=" ")
            supprime.append((facteur+1)*n)
    print()
    print("Facteurs supprimés :", end=" ")
    for facteur in supprime :
        print(facteur, end=" ")
a = int(input("Entrez un nombre entier dont vous voulez les multiples:"))
b = int(input("Entrez un nombre entier dont vous ne voulez pas les multiples:"))
c = int(input("Entrez le maximum pour les multiples:"))
multiples(a,b,c)
```

```
Entrez un nombre entier dont vous voulez les multiples:7
Entrez un nombre entier dont vous ne voulez pas les multiples:11
Entrez le maximum pour les multiples:500
7 14 21 28 35 42 49 56 63 70 ... 84 91 98 105 112 119 126 133 140 147 ... 161 168 175 18
2 189 196 203 210 217 224 ... 238 245 252 259 266 273 280 287 294 301 ... 315 322 329 33
6 343 350 357 364 371 378 ... 392 399 406 413 420 427 434 441 448 455 ... 469 476 483 49
0 497
Facteurs supprimés : 77 154 231 308 385 462
```

Nous utilisons la liste *supprime* pour stocker les multiples supprimés. Cette liste est initialisée sans rien à l'intérieur avec l'instruction : *supprime=list*(). On aurait aussi pu écrire *supprime*=[]. Vous constatez par ailleurs que l'on peut faire un calcul dans les parenthèses de *range*().

Si le test (*facteur+1*)***n*%*m*!=0 vous paraît compliqué, il faut l'examiner de plus près, en détaillant le pourquoi de chaque élément : !=0 teste si le nombre *A*=(*facteur+1*)***n* n'est pas divisible par *m* (car en effet, *A*%*m*==0 ssi *A* est divisible par *m*). Le produit (*facteur+1*)***n* correspond juste au multiple de *n* (on

augmente *facteur* de 1 car *facteur* commence à 0).

Un petit aparté concernant le mot-clef *in*. On l'utilise pour construire une boucle *for* mais il n'est pas réduit à cet usage. On peut aussi l'utiliser dans un test, par exemple *if* espaces *in* "Salut les copains!" : *print*("oui") affichera *oui* si on met dans *espace* la chaîne " " (deux espaces) car nous avons tapé deux espaces entre "salut" et "les". Cet autre exemple affichera le texte entré dans *texte* avec un astérisque à la place des consonnes.

```
texte = "Salut les copains!"
texte_voyelles= ""
for lettre in texte :
        if lettre in " AEIOUYaeiouy ": texte_voyelles+=lettre
        else : texte_voyelles+="*"
print(texte_voyelles)
```

Ce petit programme affiche ainsi **a*u* *e* *o*ai****.

3. Listes, chaînes et autres objets

a) Listes

On sait déjà plus ou moins ce qu'est une liste pour Python : une structure qui conserve les données qu'on y entre dans un ordre précis, qui permet de les retrouver, de les changer, de les supprimer, d'en ajouter d'autres, etc.. De nombreuses manipulations sont possibles avec les listes qui sont un outil indispensable de la programmation. Dans d'autres langages, on utilise plus généralement ce qui s'appelle des *tableaux* : des structures assez semblables aux listes mais qui ont l'inconvénient principal d'avoir une taille fixée dès l'initialisation. En Python, il y a un équivalent aux tableaux, ce sont les *tuples*.

Pour instancier une liste, on peut déclarer une liste vide *L* en écrivant *L*=*list*() ou *L*=[].
On peut aussi déclarer et initialiser une liste en donnant ses éléments : *prem*=[2,3,5,7,11,13].
Avec des chaînes de caractères, le principe est le même : *jours*=["Lu","Ma","Me","Je","Ve","Sa","Di"].
Une liste peut contenir des listes : *points*=[[0,0],[0,1],[1,1],[1,0]] ou des objets hétéroclites *bazar*=[1, 2.0, 'hello!, [3.14, 'pi']]. La fonction *range*(*a*,*b*), employée dans les boucles *for*, permet de créer une liste virtuelle d'entiers consécutifs allant de *a* inclus à *b* exclu. Pour donner une existence concrète à cette liste, on la capture avec *list*() : *list*(*range*(1,10)) renvoie la liste [1,2,3,4,5,6,7,8,9]. On peut omettre le 1er paramètre, Python considère alors que c'est 0 : *list*(*range*(10)) contient les 10 chiffres. On peut ajouter un 3ème paramètre pour modifier l'incrément (par défaut c'est 1) : *list*(*range*(1,10,2)) renvoie [1, 3, 5, 7, 9]. Enfin, on peut utiliser l'opérateur * pour dupliquer une liste 9*[0] renvoie la liste [0, 0, 0, 0, 0, 0, 0, 0, 0].

On peut accéder à un élément quelconque d'une liste en donnant le nom de la liste et le rang de l'élément. Par exemple, *jours*[6] contient la valeur "Di". On peut également parcourir une liste en partant de la fin, en sens inverse, en indiquant un rang négatif : *jours*[-1] contient "Di" et *jours*[-2] contient la valeur "Sa".
On peut modifier un élément quelconque par une affectation ordinaire : *jours*[6]="Dim".
On peut aussi connaître le nombre d'éléments d'une liste avec la fonction *len* (de *length*, longueur en anglais) : *len*(*prem*) contient la valeur 6 et *len*(*points*) la valeur 4.
On peut afficher la liste avec l'instruction *print*(). Par exemple, *print*(jours) conduit à l'affichage ["Lu","Ma","Me","Je","Ve","Sa","Di"]. Si on ne veut pas afficher les crochets et les guillemets, on peut écrire une petite boucle : *for* jour *in* jours : *print*(jour, *end*=" ") qui conduit à Lu Ma Me Je Ve Sa Dim.

La structure de liste est une classe qui possède de nombreuses méthodes. Les méthodes sont des fonctions que l'on peut utiliser en écrivant leur nom après le nom de l'objet (ici, de la liste), avec un point entre ces deux noms. Nous avons déjà mentionné la methode *append*() qui permet d'ajouter un élément en fin de liste : *points.append*([0.5,0.5]) ajoute un 5ème élément dans la liste *points*. On peut également ajouter une autre liste avec la méthode *extend*() : *prem.extend*([17,19,23,27,29]) étend la liste *prem* qui est maintenant [2,3,5,7,11,13,17,19,23,27,29]. On peut, à la place de *extend*(), utiliser l'addition qui produit le même résultat : *prem*=*prem*+[31,37,41] ou encore *prem*+=[43,47]. On n'est pas forcé d'ajouter toujours les éléments en fin de liste. La méthode *insert*(rang,élément) insère l'élément au rang voulu, en décalant tous les éléments suivants d'un rang. Par exemple, *points.insert*(2,[0.5,1.5]) insère l'élément [0.5,1.5] au rang 2, c'est-à-dire à la place de l'élément [1,1] qui occupe maintenant le rang 3 comme on le voit en exécutant *print*(*points*) qui affiche [[0,0],[0,1],[0.5,1.5],[1,1],[1,0],[0.5,0.5]].

On peut supprimer la première occurrence d'une valeur dans un tableau (celle qui a le rang le plus bas) en utilisant la méthode *remove*(valeur). Par exemple, *points.remove*([0,1]) enlève cet élément de la liste points qui devient [[0,0],[0.5,1.5],[1,1],[1,0],[0.5,0.5]]. Une autre manière de supprimer un élément d'une liste est d'en indiquer le rang dans la fonction *del*. On supprime ainsi la variable désignée : *del* liste[rang] supprime l'élément liste[rang] de la liste, par exemple *del points*[3] supprime l'élément [1,0] de la liste points qui devient [[0,0],[0.5,1.5],[1,1],[0.5,0.5]] (l'élément de rang 4 se retrouve au rang 3).

Lorsqu'on parcourt une liste dans une boucle *for* v *in* liste, en Python, on capture dans la variable v la valeur de ses éléments successifs. Par exemple en écrivant : *for* jour *in* jours, la variable *jour* contient successivement "Lu","Ma", etc. Parfois, il peut être utile de disposer du rang de l'élément dans le tableau. On peut, bien sûr, créer et incrémenter un compteur, mais la fonction *enumerate*() met à disposition le rang

et la valeur de l'élément. En écrivant : *for* rang,valeur *in enumerate*(jours), on obtient chacune de ces informations dans des variables séparées.

Une particularité originale de Python est d'offrir une syntaxe simple pour fabriquer une nouvelle liste en se basant sur le parcours d'une liste existante. Si *L1* est une liste contenant les nombres de 1 à 10 (*L1*=[1,2,3,4,5,6,7,8,9,10]), on peut fabriquer les listes *L2* et *L3* contenant les doubles et les carrés de ces nombres en écrivant *L2*=[2**n for n in L1*] et *L3*=[*n*n for n in L1*]. On peut également appliquer un test, effectuer un filtrage, pour les valeurs prises dans *L1*, par exemple *L4*=[*n+1 for n in L1 if n%2==0*]. Les listes obtenues par ces affectations sont [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] pour *L2*, [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] pour *L3* et [3, 5, 7, 9, 11] pour *L4*.

La méthode *sum*(), permet d'obtenir la somme des termes d'une liste. Par exemple *sum*([1,2,3,4,5,6,7,8,9,10]) renvoie le nombre 55. Couplée avec les déclarations de liste en compréhension, on obtient en une ligne la somme d'une suite : *sum*([0.5**k for k in range(1,11)]) permet de vérifier que $\sum_{k=1}^{10}(\frac{1}{2})^k$ vaut 0.9990234375.

Nous avons déjà présenté la transformation d'une liste en texte avec l'instruction *join*(t). Avec notre liste *jours*=["Lu","Ma","Me","Je","Ve","Sa","Di"], on peut créer le texte *semaine*=" ".*join*(*jours*)) qui contient la chaîne de caractères "Lu Ma Me Je Ve Sa Di". La transformation inverse est possible : on découpe une chaîne de caractères en morceaux qui constituent une liste. La fonction *split* utilisée est une fonction de la classe *str* qui prend en argument le séparateur : ce peut être un point, un espace, ou n'importe quelle chaîne. On peut, par exemple, découper un texte s en mots en écrivant *mots*=s.*split*(" "). La variable *mots* ainsi instanciée est alors une liste.

Pour compléter notre brève description des méthodes de la classe *list*, signalons la possibilité de renverser l'ordre des éléments d'une liste avec *reverse*() ainsi ["Lu","Ma","Me","Je","Ve","Sa","Di"].*reverse*() est la liste ['Di', 'Sa', 'Ve', 'Je', 'Me', 'Ma', 'Lu']. On peut également obtenir la liste classée dans l'ordre alphanumérique avec *sort*(), ainsi ["Lu","Ma","Me","Je","Ve","Sa","Di"].*sort*() est la liste ['Di', 'Je', 'Lu', 'Ma', 'Me', 'Sa', 'Ve']. Ces méthodes trient « sur place » : elles ne renvoient rien, mais modifient la liste initiale. Des fonctions réalisent la même chose sans modifier la liste initiale, la fonction *reversed*() et la fonction *sorted*() qui prennnent en argument ce qu'il faut inverser ou trier, avec une option pour la dernière afin d'indiquer dans quel ordre : *jours_inverse=sorted*(*jours*, *reverse=True*) met dans la liste *jours_inverse* la liste des jours triée dans l'ordre alphanumérique décroissant sans affecter la liste *jours*. La liste *jours_inverse* contient donc ['Ve', 'Sa', 'Me', 'Ma', 'Lu', 'Je', 'Di']. On peut également tout mélanger avec *shuffle*(), mais cette fonction est une méthode appartenant au module *random*. On doit l'importer avec l'instruction *from random import shuffle* et ensuite *shuffle*(*jours*) peut renvoyer quelque chose comme ['Me', 'Ve', 'Di', 'Ma', 'Sa', 'Je', 'Lu'].

b) Chaînes

Généralement, on déclare une chaîne, éventuellement vide, en écrivant une affectation comme s='*Bonjour!*' ou bien s="*Bonjour!*". Apostrophe ou guillemets ? Si on a la chaîne '*Comment vas-tu mon ami? Depuis si longtemps que l'on ne s'est vu...*' alors il va y avoir une erreur car l'interpréteur de Python ferme la chaîne après la 1ère apostrophe trouvée (*...que l'on...*) et ce qui suit constitue une erreur de syntaxe. On peut contourner ce genre de difficulté en écrivant des guillemets "*Comment vas-tu mon ami? Depuis si longtemps que l'on ne s'est vu...*". Une autre façon d'éviter l'erreur est « d'échapper » l'apostrophe en plaçant un *antislash* \ avant le caractère ambigu qui serait mal interprété '*c\'est l\'hiver que j\'préfère*'. On peut faire de même avec les guillemets, par exemple "*L'ouvreuse m'a dit : \"Donnez-moi votre ticket.\" Je le lui ai donné.*"

Le nombre de caractères d'une chaîne est donné avec l'instruction *len*() qui s'emploie aussi avec les listes : *len*("Bonjour!") renvoie le nombre 8. La concaténation des chaînes est obtenue avec l'opérateur +, et on peut même répéter plusieurs fois une chaîne avec *, par exemple 'NOM='+10*' _' renvoie la chaîne 'NOM= _ _ _ _ _ _ _ _ _ _'. L'afficher d'une chaîne s est réalisé avec l'instruction *print*(s). Nous avons vu tout cela déjà. Nous avons aussi mentionné que l'objet *str* est une liste de Python et qu'on peut en extraire un caractère de rang donné avec l'expression *string*[*rang*] par exemple "Bonjour!"[2] renvoie le caractère 'n'.

On peut aussi extraire une sous-chaîne en partant du rang *deb* jusqu'au rang *fin*. Cela s'écrit *string*[*deb*:*fin*], ainsi "Bonjour!"[1:3] renvoie les caractères 'on'. On peut omettre le rang de début si c'est 0 ou le rang de fin si c'est le dernier. Par exemple, "Bonjour!"[:3] renvoie 'Bon' alors que "Bonjour!"[3:] renvoie 'jour!'.
Les chaînes sont les objets de la classe *str* (*string* signifie corde, file (de voitures), chaîne (de caractères) en anglais) en Python. Ces objets sont manipulables de bien des façons, avec des méthodes propres à la classe *str*. On accède à ces méthodes avec un point derrière la chaîne. Voici quelques méthodes de la classe *str*.

- La méthode *join*(*L*) fabrique une chaîne à partir d'une liste *L* en insérant la chaîne séparatrice ' '.*join*(*L*) insère un espace entre les éléments de la liste *L*.
- La méthode *format*(*var1, var2, ...*) permet de remplacer des variables par leur valeur dans une chaîne, par exemple "M ({};{})".*format*(a,b) ou mieux "{} ({};{})".*format*(nom_point,*x*,*y*).
- La méthode *split*(*sep*) permet de découper une chaîne selon une chaîne séparatrice *sep*, le résultat constituant une liste 'Viens dans 2 jours!'.*split*(' ') fabrique la liste ['Viens', 'dans', '2', 'jours!'].
- La méthode *upper*() permet de changer les caractères en majuscules 'Viens dans 2 jours!'.upper() renvoie la chaîne 'VIENS DANS 2 JOURS!' alors que *lower*() fait l'inverse 'Mon nom est MOUTOU'.*lower*() renvoie 'mon nom est moutou'. Si vous voulez juste mettre les initiales en majuscules, il faut employer la méthode *title*() car 'jean-hugues anglade'.*title*() renvoie 'Jean-Hugues Anglade'.
- La méthode *count*(*sub*) compte le nombre d'occurrences de la chaîne *sub* dans une chaîne 'Philippe'.*count*('p') renvoie 2 car 'P' n'est pas compté (la casse des caractères est différente). Si on veut compter les 3 'p' de mon prénom, on peut faire 'Philippe'.*lower*().*count*('p').
- La méthode *find*(*sub*) donne le rang de la 1ère occurrence de la chaîne *sub* dans la chaîne et donne -1 si la chaîne *sub* est absente 'Leonard de Vinci'.*find*(' ') renvoie 7 alors que 'Leonard'.*find*(' ') renvoie -1. On peut parcourir la chaîne à partir de la fin 'Leonard de Vinci'.*rfind*(' ') renvoie 10 qui est le rang de la dernière occurrence de ' ' dans cette chaîne.
- La méthode *isalpha*() renvoie *True* s'il n'y a que des caractères alphabétiques dans la chaîne (il n'existe que 52 caractères alphabétiques abc...zABC...Z) et *False* sinon ' '.*isalpha*() renvoie *False*, comme 'p m'.*isalpha*() ou '12h'.*isalpha*() mais 'deux'.*isalpha*() renvoie *True*.

Nous ne pouvons détailler toutes les méthodes de la classe *str* car elles sont nombreuses et parfois plus raffinées que nous ne le signalons. Par exemple, la méthode *find*(*sub*) accepte comme argument optionnel qu'on indique à partir de quel rang de début *deb* on souhaite effectuer la recherche s.*find*(' ',s.*find*(' ')+1) cherche ainsi la 2ème occurrence de ' ' dans la chaîne s (si s='Leonard de Vinci', on trouve 10). On peut aussi indiquer le rang de *fin* pour cette recherche avec la notation, cette fois complète, s.*find*(*sub*,*deb*,*fin*).
Afin de connaître toutes les méthodes d'une classe *class* quelconque de Python, il suffit de taper *help*(*class*). Si on tape *help*(*str*), on trouvera toutes les méthodes décrites avec leurs raffinements éventuels ainsi que de nombreuses autres qui paraissent moins utiles. La méthode *replace*() par exemple n'est pas forcément inutile. Elle peut même se révéler indispensable dans certaines situations. Comme le dit le commentaire, elle permet de rechercher et d'échanger une chaîne *old* par une autre *new* dans une chaîne 'Certains enfants ont des problèmes avec les s en français'.*replace*('s','S') renvoie la chaîne 'CertainS enfantS ont deS problèmeS avec leS S en françaiS' ou bien s.*replace*('x','\u00D7') remplacera les inesthétiques x multiplicatifs '3xa+3xb=3(a+b)'.*replace*('x','\u00D7') renvoie la chaîne '3×a+3×b=3(a+b)'.

```
replace(...)
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

c) Fabriquer et utiliser les objets d'une classe

Les méthodes que nous venons de voir pour les listes et les chaînes sont spécifiques à ces classes d'objets. En Python « tout est objet », même les nombres, et on peut appliquer à ces objets les méthodes de la classe d'objets dont ils font partie. Mais en dehors des nombres, des chaînes et des listes, on peut avoir besoin de définir notre propre classe d'objets. On choisit alors les données qu'elle peut contenir et les méthodes qu'elle peut appliquer à ces données (recevoir des nouvelles données, renvoyer des données, effectuer certains traitement, etc.).

Pour fixer les idées, prenons une exemple simple : nous voulons créer une classe d'objets « points du plan ». La position d'un point étant définie par deux coordonnées, il faut pouvoir déclarer un point en donnant deux nombres, par exemple en écrivant M=Point(2,3). Ensuite, pour retrouver l'abscisse et l'ordonnée de notre point, il suffira d'écrire M.x ou M.y. Imaginons qu'en plus d'enregistrer les coordonnées d'un point, on veut pouvoir appliquer une translation de vecteur $\vec{v}$ (*a*;*b*) et on veut aussi pouvoir afficher les coordonnées. Nous pouvons définir notre classe « Point » qui réalisera tous nos vœux :

```
class Point :
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def translat(self,a,b):
        self.x+=a
        self.y+=b
    def affiche(self):
        print('(', self.x, ';', self.y, ')')
```

Cela suffit pour l'instant. La classe d'objets « Point » est déclarée. Les objets de cette classe ont deux attributs x et y. La méthode *__init__* (avec deux *undercores* avant et après le mot-clef *init*) est une classe spéciale et indispensable appelée *constructeur* qui permet d'instancier un objet (de déclarer un nouveau point). Le mot-clef *self* est utilisé au sein de la classe pour remplacer le nom de l'objet auquel appartiennent les attributs. La méthode *translat* est la méthode qui va permettre de translater un point et la méthode affiche réalisera l'affichage des coordonnées. L'utilisation de notre classe est simple : A=Point(2,3) crée l'objet, A.x vaut 2 et A.y vaut 3. On peut afficher (2;3) en écrivant A.affiche(). On peut éventuellement modifier les attributs en écrivant, par exemple, A.x=5 et A.y+=1 maintenant, A.affiche() donne (5,4). Pour utiliser notre méthode *translat*, avec une vecteur $\vec{v}$ (*-10*;*15*), il suffit d'écrire A.translat(-10,15). Maintenant A.affiche() donne (-5,19).

```
class Point :
    '''Classe Point prenant les coordonnées en argument'''
    def __init__(self,x,y):
        #constructeur de la classe Point
        self.x=x
        self.y=y
    def translat(self,a,b):
        '''methode translat pour opérer une translation de (a,b)'''
        self.x+=a
        self.y+=b
    def affiche(self):
        '''methode d'affichage'''
        print('(', self.x, ';', self.y, ')')
```

```
>>> help(Point)
Help on class Point in module __main__:
class Point(builtins.object)
 |  Classe Point prenant les coordonnées en argument
 |
 |  Methods defined here:
 |
 |  __init__(self, x, y)
 |
 |  affiche(self)
 |      methode d'affichage
 |
 |  translat(self, a, b)
 |      methode translat pour opérer une translation de (a,b)
```

Pour que notre classe « Point » soit plus compréhensible (pour un utilisateur extérieur par exemple, ou pour un réemploi après une longue période), on conseille d'écrire des commentaires (des *docstrings*), en utilisant une syntaxe avec trois guillemets consécutifs. Une fois notre classe créée on peut avoir accès à sa documentation en tapant *help*(nom_de_la_classe). Les *docstrings* sont différents des commentaires ordinaires, ces lignes ou fins de ligne explicatives que l'on peut écrire à la suite d'un #. Les commentaires ordinaires n'apparaissent pas sur la documentation de la classe.

```
class Point :
    '''Classe Point prenant les coordonnées en argument'''
    nombre=0#attribut de la classe Point
    def __init__(self,x,y):
        #constructeur de la classe Point
        self.x=x
        self.y=y
        Point.nombre+=1#augmente à chaque nouvelle instanciation d'un Point
        self.distance=x**2+y**2
    def translat(self,a,b):
        '''methode translat pour opérer une translation de (a,b)'''
        self.x+=a
        self.y+=b
    def affiche(self):
        '''methode d'affichage'''
        print('(', self.x, ';', self.y, ')')
```

```
>>> A=Point(2,3)
>>> A.nombre
1
>>> B=Point(-1,5)
>>> A.nombre
2
>>> B.nombre
2
>>> B.distance
26
>>> B.translat(1,5)
>>> B.affiche()
( 0 ; 10 )
>>> B.distance
26
```

Les attributs d'un objet ne sont pas seulement limités aux variables passées en argument au moment de la déclaration de notre objet. Il peut être utile de définir d'autres attributs ayant, par défaut, une certaine valeur, donnée dans le constructeur. Ces attributs appartiennent tous à un seul objet, l'objet désigné par *self*, on les appelle des attributs d'instance. À côté de ces attributs d'instance, il peut exister des attributs de

classe qui auront la même valeur pour tous les objets de la classe. Enrichissons notre classe « Point » d'un attribut d'instance *distance* qui est la distance du point à l'origine, ainsi qu'un attribut de classe *nombre* qui est le nombre de points créés.
Ces deux nouveaux attributs s'utilisent simplement comme on le voit dans cette copie du *shell* (à droite). Remarquez que A.nombre et B.nombre donne la même valeur (on aurait aussi pu écrire Point.nombre pour avoir cette valeur). Pour le paramètre distance, c'est aussi simple, sauf que c'est très gênant : B.distance vaut toujours 26, même après le déplacement de B (normalement, on devrait avoir alors B.distance=100). Pour éviter de conserver une valeur fausse après une translation, on peut ajouter à la fin de la méthode *translat* l'instruction self.distance=x**2+y**2. Mais cela ne résout pas le problème si l'on s'autorise à modifier les valeurs des attributs x ou y en écrivant, par exemple, B.x=0 ou B.y+=0.5. La solution est de créer une méthode spéciale pour la modification des attributs x et y. On nomme souvent ce genre de méthode *setAttribut*(), ainsi *setX*() sera utilisée pour modifier la valeur de l'attribut x en écrivant, par exemple B.setX(0) au lieu du B.x=0 qui n'agit pas sur l'attribut distance. Muni de cette méthode ainsi qu'une semblable pour modifier l'attribut y, la classe Point devient fonctionnelle. Nous avons ajouté une méthode interne *recalcule*() qui évite d'avoir à effectuer le calcul de la distance à différents endroits de la classe. Ici, cela apparaît comme un détail, mais lorsque le calcul impliqué sera plus complexe, il est intéressant de ne l'écrire qu'une seule fois et d'y accéder au moyen d'une fonction interne qui ne sert qu'à ça.

```
class Point :
    '''Classe Point prenant les coordonnées en argument'''
    nombre=0#attribut de la classe Point
    def __init__(self,x,y):
        #constructeur de la classe Point
        self.x=x
        self.y=y
        Point.nombre+=1#augmente à chaque nouvelle instanciation d'un Point
        self.distance=x**2+y**2
    def recalcule(self):
        '''methode interne qui calcule la distance à l'origine'''
        return self.x**2+self.y**2
    def setX(self, a):
        '''methode à utiliser pour modifier x'''
        self.x=a
        self.distance=self.recalcule()
    def setY(self, a):
        '''methode à utiliser pour modifier y'''
        self.y=a
        self.distance=self.recalcule()
    def translat(self,a,b):
        '''methode translat pour opérer une translation de (a,b)'''
        self.x+=a
        self.y+=b
        self.distance=self.recalcule()
    def affiche(self):
        '''methode d'affichage des coordonnées'''
        print('(', self.x, ';', self.y, ')', ' distance=', self.distance)
```

```
>>> A=Point(2,3)
>>> A.distance
13
>>> A.translat(1,5)
>>> A.affiche()
( 3 ; 8 )  distance= 73
>>> A.setX(0)
>>> A.affiche()
( 0 ; 8 )  distance= 64
>>> A.translat(-5,-2)
>>> A.affiche()
( -5 ; 6 )  distance= 61
>>>
```

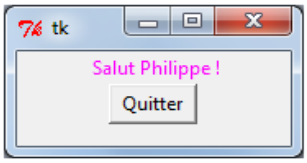
4. Affichage graphique et utilisation d'un fichier

a) Graphique

Le module à importer pour réaliser un affichage graphique (plutôt que textuel comme nous le faisons depuis le début) est *tkinter*. Nous créons ensuite une instance de l'objet *Tk* (une fenêtre d'affichage) que nous nommons ici *fenetre* dans laquelle nous déposons des objets graphiques (on dit des *widgets*) comme un *texte* (un objet *Label*, une étiquette) ou un *bouton* (un objet *Button*, qui pourra déclencher une action). Voici ce programme minimaliste dans l'éditeur, son exécution dans le *shell* à droite et la fenêtre qu'il produit et qui se fermera en cliquant sur le bouton (ou sur la croix, la méthode par défaut).

```
from tkinter import *
nom=input("Quel est ton nom ? ")
fenetre=Tk()
texte=Label(fenetre,text=" Salut {} !".format(nom),fg='magenta')
texte.pack ()
bouton=Button(fenetre,text =" Quitter ",command=fenetre.destroy)
bouton.pack()
fenetre.mainloop ()
```

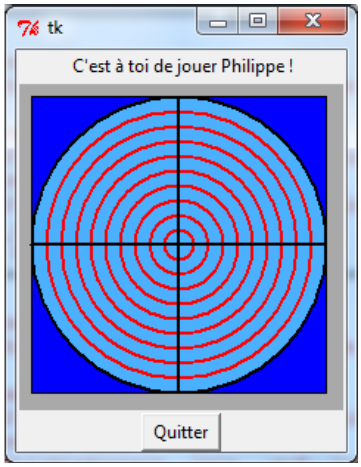
```
>>>
Quel est ton nom ? Philippe
```



Pour dessiner des formes géométriques, il faut disposer d'un objet appelé *canvas* (une toile en français) qu'on ajoute à la fenêtre en lui assignant ses dimensions (hauteur, largeur), sa couleur, sa place dans la fenêtre, etc. Ensuite on peut, couche après couche, tracer des lignes, des cercles, des rectangles, des polygones, etc. Chaque objet est créé par une syntaxe particulière, par exemple :

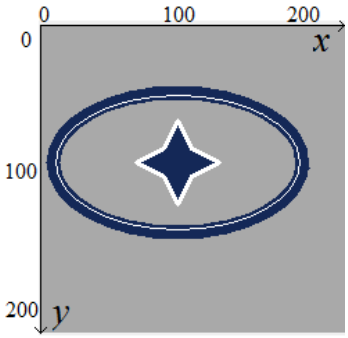
- create_line(10,110,210,110,width=2,fill='black') trace une ligne du point de coordonnées (10;110) jusqu'au point de coordonnées (210;110) d'épaisseur 2 et de couleur noire.
- create_rectangle(10,10,210,210,width=1,fill='blue') trace un rectangle de sommet supérieur gauche le point de coordonnées (10;10) et de sommet inférieur droite le point de coordonnées (210;210) d'épaisseur 1 et le remplit d'une couleur bleue (*fill*=remplir).
- create_oval(110-r,110-r,110+r,110+r,width=2,outline='red') trace un cercle de centre (110;110) et de rayon r, d'épaisseur 2 et de couleur bleue (*outline*=bord).

```
from tkinter import *
nom='Philippe'
fenetre=Tk()
texte=Label(fenetre,text=" C'est à toi de jouer {} !".format(nom),fg='black')
texte.pack ()
toile = Canvas(fenetre,bg='dark grey',height=220,width=220)
toile.pack()
toile.create_rectangle(10,10,210,210,width=1,fill='blue')
toile.create_oval(10,10,210,210,width=2,fill='#4AB0FF')
rayon=10
while rayon<100:
    toile.create_oval(110-rayon,110-rayon,110+rayon,110+rayon,width=2,outline='red')
    rayon+=10
toile.create_line(10,110,210,110,width=2,fill='black')
toile.create_line(110,10,110,210,width=2,fill='black')
bouton=Button(fenetre,text =" Quitter ",command=fenetre.destroy)
bouton.pack()
fenetre.mainloop ()
```



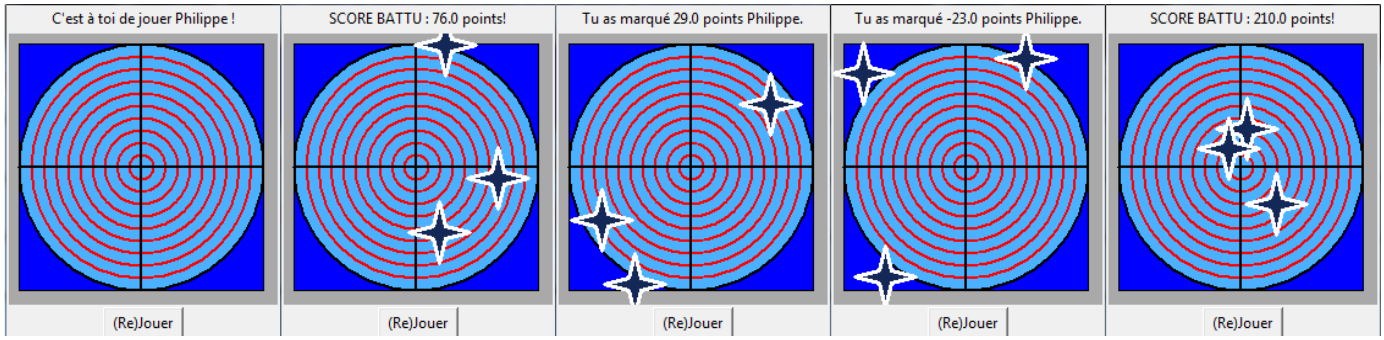
L'option *side*=BOTTOM de la commande *pack*() assure de placer la *toile* en bas de la *fenetre* (on dispose de trois autres options LEFT, RIGHT et TOP). Les couleurs principales sont connues ('black', 'magenta', 'purple', 'cyan', 'maroon', 'green', 'red', 'blue', 'orange', 'yellow', etc.) mais on peut entrer un nombre hexadécimal (en base 16, les chiffres sont 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E et F) de deux caractères pour chacune des composantes RGB (Red Green Blue), par exemple "#4AB0FF" (dans ce cas Red:$4A_{(16)}=4\times16+10=74_{(10)}$, Green:$B0_{(16)}=11\times16+0=176_{(10)}$, BLUE:$FF_{(16)}=15\times16+15=255_{(10)}$, la couleur obtenue est un bleu clair). Le dessin d'un polygone prend une liste de coordonnées [x_1, y_1, x_2, y_2, etc.] en argument : points=[100, 140, 110, 110, 140, 100, 110, 90, 100, 60, 90, 90, 60, 100, 90, 110]
toile.create_polygon(points, *outline*='white', fill='#142857', width=3)
Les sommets de cette étoile à quatre branches sont énumérés à partir du point le plus bas, et en tournant dans le sens inverse trigonométrique. Remarquez le système des coordonnées avec un axe des ordonnées

(*Oy*) inversé par rapport au sens qu'on lui donne habituellement.
toile.create_oval(10,50,190,150,width=10,outline='#142857')
toile.create_oval(12,52,188,148,width=1,outline='white')
La méthode *create_oval*(x_1, y_1, x_2, y_2,...) crée des ovales (des ellipses) lorsque les deux « diamètres » perpendiculaires x_2-x_1 et y_2-y_1 sont différents.



Nous pouvons interagir avec la fenêtre en utilisant les boutons pour déclencher des fonctions. Ces fonctions peuvent modifier le dessin (généralement, on commence par tout effacer avec *toile.delete*(ALL) et puis on redessine), elles peuvent aussi modifier les *widgets*, par exemple un Label qui affiche un texte modifié. Le dessin précédent peut servir de cible pour un joueur aléatoire qui tire trois flèches (dessinées sur le modèle polygonal ci-dessus). À chaque pression sur le bouton « (Re)Jouer » la cible et les flèches sont redessinées et un score est calculé puis affiché.
Voici le programme qui réalise cela :

```
from tkinter import *
from random import randint
from math import sqrt

nom='Philippe'
fenetre=Tk()
texte=Label(fenetre,text="C'est à toi de jouer {} !".format(nom),fg='black')
texte.pack ()
toile=Canvas(fenetre,bg='dark grey',height=220,width=220)
toile.pack()

def fleche(x,y):
    ma,mi=25,5
    points=[x,y+ma,x+mi,y+mi,x+ma,y,x+mi,y-mi,x,y-ma,x-mi,y-mi,x-ma,y,x-mi,y+mi]
    toile.create_polygon(points,outline='white',fill='#142857',width=3)

def cible() :
    toile.delete(ALL)
    #dessin de la cible
    toile.create_rectangle(10,10,210,210,width=1,fill='blue')
    toile.create_oval(10,10,210,210,width=2,fill='#4AB0FF')
    r=10
    while r<100:
        toile.create_oval(110-r,110-r,110+r,110+r,width=2,outline='red')
        r+=10
    toile.create_line(10,110,210,110,width=2,fill='black')
    toile.create_line(110,10,110,210,width=2,fill='black')

def jouer() :
    cible()
    #dessin des flèches
    score=0
    for f in range(3) :
        xfleche=randint(10,210)
        yfleche=randint(10,210)
        fleche(xfleche,yfleche)
        score+=100-10*sqrt((xfleche-110)**2+(yfleche-110)**2)//10
    maj(score)

def maj(s) :
    global scoreMax
    if s>scoreMax :
        texte.configure(text="SCORE BATTU : {} points!".format(s))
        scoreMax=s
    else :
        texte.configure(text="Tu as marqué {} points {}.".format(s,nom))

cible()
scoreMax=0.0
bouton1=Button(fenetre,text =" (Re)Jouer ",command=jouer)
bouton1.pack()
bouton2=Button(fenetre,text =" Arrêter ",command=fenetre.destroy)
bouton2.pack()
fenetre.mainloop()
```

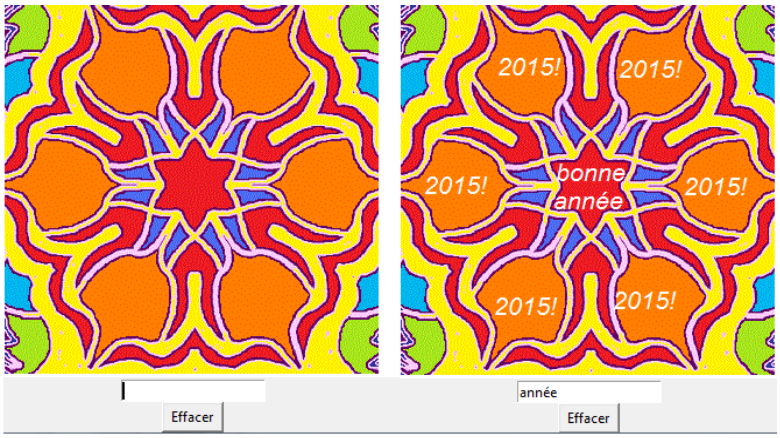


L'essentiel du côté dynamique vient de la commande *command=jouer* associée au bouton « (Re)Jouer » qui déclenche la fonction jouer() qui elle-même redessine la cible (fonction *cible*()), les flèches (fonction *fleche*()), recalcule le score et l'affiche (fonction *maj*()). Pour modifier le texte du Label *texte*, il suffit d'écrire *texte.configure*(*text*="...nouveau texte...").
Une instruction n'a pas été expliquée jusqu'ici : *global scoreMax*. Le mot-clef *global* indique que la variable *scoreMax* n'est pas une variable locale, interne à la fonction *maj*(), mais une variable globale, qui a été déclarée à l'extérieur de la fonction. Ce statut de variable globale permet à la fonction de la modifier et assure qu'elle conserve la valeur du score maximum quelque soit le moment du jeu. Il n'y a pas que les boutons pour créer de l'interactivité avec la fenêtre. Voici comment entrer un texte dans une zone de texte (*entree*), et comment provoquer son affichage dans un champ de texte (*texte*).

```
def fonction(event):
  texte.configure(text="Vous avez tapé= "
    +entree.get())
  entree = Entry(fenetre)
  entree.bind("<Return>",fonction)
  entree.pack()
```

La fonction *bind*() de l'objet *Entry* (une zone de texte) associe la touche « retour chariot » (la touche *Enter*) à une commande, la fonction *fonction*() qui prend comme 1er argument le mot-clef *event*. Dans cette fonction, la méthode *get*() de l'objet *Entry* permet de récupérer le texte entré et donc de l'afficher à l'aide de la méthode *configure*() de l'objet *Label*.

```
from tkinter import *
def pointeur(event):
    xcurseur=event.x
    ycurseur=event.y
    txt=cadre.create_text(xcurseur,ycurseur,text=entree.get(),\
                          font="Arial 16 italic",fill="white")
def initial():
    cadre.delete(ALL)
    mon_image=cadre.create_image(200,200,image=photo)
fen=Tk()
cadre=Canvas(fen,width=400,height=400,bg='white')
cadre.bind("<Button-1>",pointeur)
cadre.pack()
photo=PhotoImage(file ='etoile.gif')
initial()
entree=Entry(fen)
entree.bind("<FocusOut>",pointeur)
entree.pack()
bouton=Button(fen,text=" Effacer ",command=initial)
bouton.pack()
fen.mainloop()
```



On peut également rendre la fenêtre clickable, la position du curseur au moment du click pouvant être reçue et utilisée pour un traitement à la volée, par exemple pour dessiner (ou écrire). Dans le petit programme qui suit, nous affichons une image (photo) dans un *Canvas* (cadre) et une zone de texte permet d'entrer un texte qui sera écrit sur l'image quand on clique quelque part sur l'image. On peut changer le texte pour varier les écritures sur l'image ou tout effacer (il faut alors redessiner l'image initiale).
La fonction *pointeur*() est lancée quand la zone de texte perd le focus (cela arrive lorsqu'on clique dans l'image). Elle récupère les coordonnées du click (un des attributs de l'évènement *event*) et les utilise pour afficher le texte saisi (qu'elle récupère grâce à la fonction *entree.get*()) à l'endroit du click.
De nombreuses autres méthodes sont utilisables pour créer des graphiques aussi complexes, interactifs, animés que l'on peut l'imaginer. On peut fabriquer ainsi des applications avec un menu déroulant ou *pop-up* (*Menu*), contenant des boîtes de choix (*Listbox*), des cases à cocher (*Checkbutton, Radiobutton*), des curseurs (*Scale*), des ascenseurs (*Scrollbar*). Bien sûr, nous n'irons pas beaucoup plus loin ici sur ces sujets. Nous voulons juste illustrer la possibilité de gérer un mouvement automatique, contrôlé par une variable de temps. La méthode *after*() permet l'appel d'une méthode après un certain temps écoulé en millisecondes. Nous l'avons utilisée dans l'instruction *id=texte.after*(100,*compte_Rebours*) qui, toutes les 100 millisecondes lance la fonction *compte_Rebours*(). Cette commande est référencée par le mot-clef *id* qui permet de mettre fin au processus avec la commande *texte.after_cancel*(*id*).
Notre programme (voir illustration à la page suivante) dessine de façon aléatoire un motif inspiré de la marche de l'ivrogne et ses images par des quart de tour autour du centre de l'image. Ce centre est aussi l'endroit où se prolonge la marche aléatoire quand celle-ci sort de la fenêtre impartie. Non seulement la position du point est modifié mais aussi sa couleur. Nous utilisons le convertisseur *hex*() en nombre hexadécimal de Python qui donne un nombre sous la forme '*0x1a*' où les deux premiers caractères sont inutiles pour le format couleur que nous avons adopté '*#aabbcc*'. Quand on exécuté *hex*(15), on obtient '*0xf*' (le zéro n'est pas affiché). Il nous faut donc adjoindre ce zéro pour une couleur qui aurait une de ces composantes égale à $15_{(10)}=f_{(16)}$ (la couleur '*#0f0f0f*' existe mais pas la couleur '*#fff*').
Il y a certainement pas mal de choses à améliorer pour ce programme : on aurait sans doute mieux fait d'utiliser deux tableaux, un pour *x* et un pour *y*, plutôt que d'utiliser huit variables. Une autre amélioration serait d'utiliser les fonctions prédéfinies qui réalisent une rotation ou une symétrie... On aurait aussi pu proposer davantage d'interactions, etc. Mais l'intérêt d'un programme comme celui-là est, avant tout, de servir d'exemple pour illustrer une ou deux nouvelles fonctionnalités, éventuellement de donner quelques idées pour créer de meilleurs programmes.

b) Fichiers

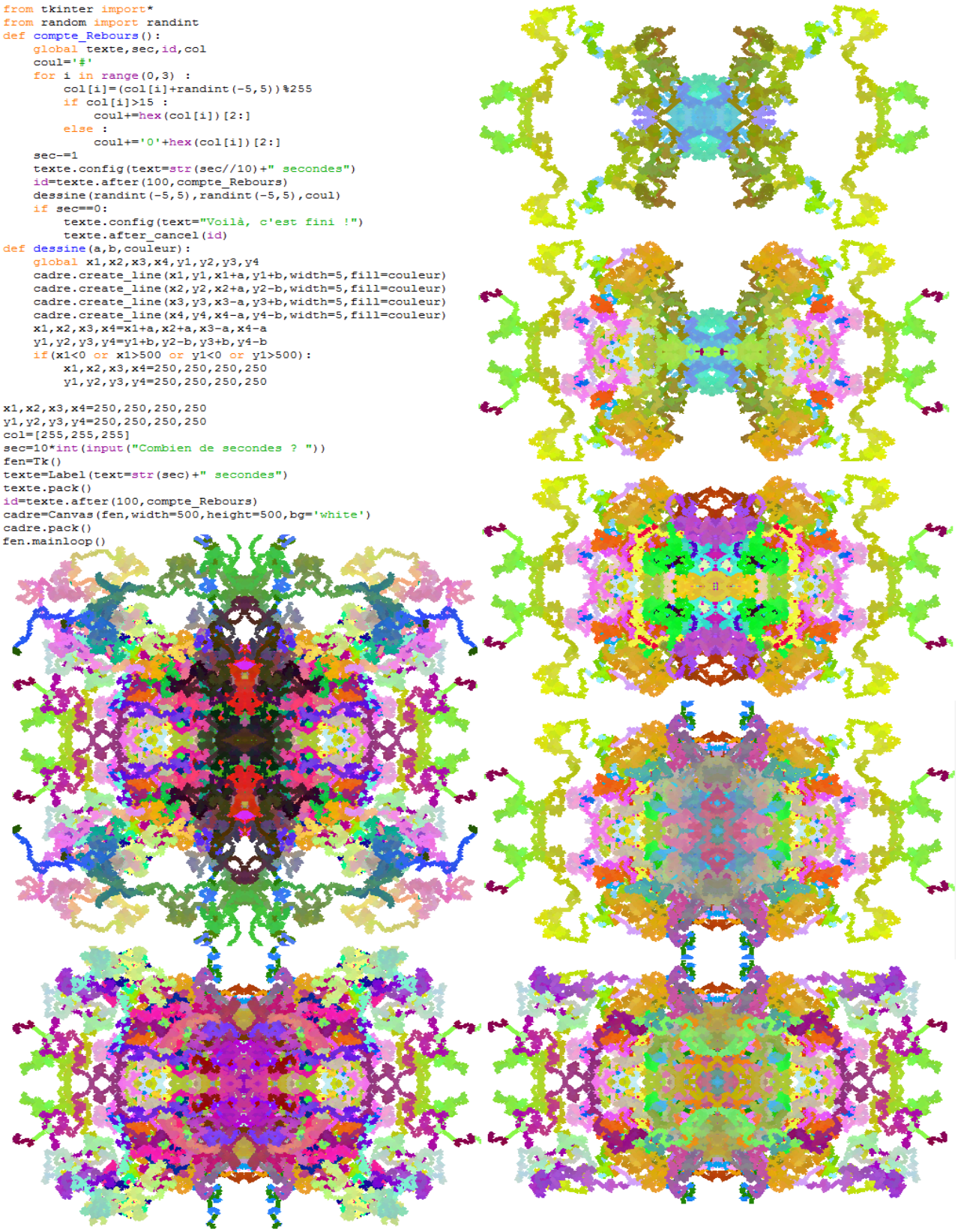
Un fichier externe est une source de données potentielles pour un programme, mais ce peut être aussi un moyen d'enregistrer les données produites. Il y a donc deux usages des fichiers à examiner : en entrée, ce peut être des lignes de données formatées, un flux continu de données, une image ou un son numérisé, les enregistrements d'une base de donnée ; en sortie, ce peut être aussi toutes ces différentes sortes de données. Nous n'envisagerons ici que des fichiers contenant des lignes de données (des lignes de texte), que ce soit en lecture ou en écriture.

Pour écrire ou lire dans un fichier, il faut indiquer l'emplacement du fichier à Python qui, sinon, le cherchera dans le répertoire courant (celui contenant le dossier Python si vous passez par IDLE). Un début

de script incluant ces deux lignes : *from os import getcwd,chdir* et *chdir*(*getcwd*()), vous assurera que, dorénavant, vos fichiers lus et écrits seront dans le répertoire courant (celui dans lequel vous avez enregistré votre fichier Python).

L'ouverture d'un fichier pour l'écriture se fait avec l'instruction : nom_fichier=*open*('monFichier.txt','*w*') où *monFichier.txt* est le nom du fichier qui va être écrit (s'il existe déjà, l'ancien sera écrasé) et *nom_fichier* est la référence de ce fichier pour les manipulations ultérieures que l'on va y faire. L'option *w* précise que l'on veut écrire (*write*) sur un fichier vierge. Si l'on veut écrire à la suite des données (éventuellement à la suite de rien si le fichier est une nouvelle création), on écrit *a* (*append*) comme option à la place de *w*. Il y a

```
from tkinter import*
from random import randint
def compte_Rebours():
    global texte,sec,id,col
    coul='#'
    for i in range(0,3) :
        col[i]=(col[i]+randint(-5,5))%255
        if col[i]>15 :
            coul+=hex(col[i])[2:]
        else :
            coul+='0'+hex(col[i])[2:]
    sec-=1
    texte.config(text=str(sec//10)+" secondes")
    id=texte.after(100,compte_Rebours)
    dessine(randint(-5,5),randint(-5,5),coul)
    if sec==0:
        texte.config(text="Voilà, c'est fini !")
        texte.after_cancel(id)
def dessine(a,b,couleur):
    global x1,x2,x3,x4,y1,y2,y3,y4
    cadre.create_line(x1,y1,x1+a,y1+b,width=5,fill=couleur)
    cadre.create_line(x2,y2,x2+a,y2-b,width=5,fill=couleur)
    cadre.create_line(x3,y3,x3-a,y3+b,width=5,fill=couleur)
    cadre.create_line(x4,y4,x4-a,y4-b,width=5,fill=couleur)
    x1,x2,x3,x4=x1+a,x2+a,x3-a,x4-a
    y1,y2,y3,y4=y1+b,y2-b,y3+b,y4-b
    if(x1<0 or x1>500 or y1<0 or y1>500):
        x1,x2,x3,x4=250,250,250,250
        y1,y2,y3,y4=250,250,250,250

x1,x2,x3,x4=250,250,250,250
y1,y2,y3,y4=250,250,250,250
col=[255,255,255]
sec=10*int(input("Combien de secondes ? "))
fen=Tk()
texte=Label(text=str(sec)+" secondes")
texte.pack()
id=texte.after(100,compte_Rebours)
cadre=Canvas(fen,width=500,height=500,bg='white')
cadre.pack()
fen.mainloop()
```

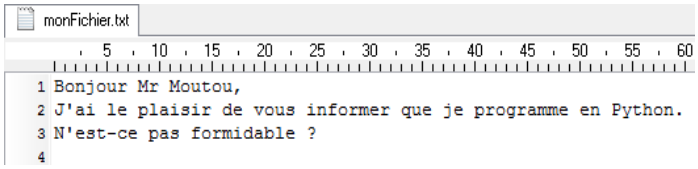


d'autres options, par exemple la lecture seule *r*(*read*, c'est le mode par défaut), la lecture/écriture *r+*.

Pour écrire une donnée *txt='mon texte de données'* dans le fichier *nom_fichier*, il suffit d'écrire l'instruction nom_fichier.*write*(txt). Si on veut écrire des lignes de données formatée, on doit insérer le caractère de fin de ligne \n. Ajoutons ces quelques lignes à notre programme :

```
testFichier.write('Bonjour Mr Moutou,\n')
testFichier.write('J'ai le plaisir de vous informer que je programme en Python.\n')
testFichier.write('N'est-ce pas formidable ?\n')
testFichier.close()
```

La dernière ligne provoque la fermeture du fichier, et sa libération pour d'autres usages (par exemple la consultation). Après exécution de ce script d'écriture, le fichier référencé par *testFichier* (en l'occurrence *monFichier.txt*) contient le texte ci-contre (le numéro de ligne a été ajouté par mon éditeur de texte, remarquer la 4ème ligne vierge déclenchée par le \n à la fin de la ligne 3). On peut ajouter une ligne à ce fichier texte, à condition de l'ouvrir en mode '*a*' : *testFichier.write*('Signé : votre ex-élève préférée.') qui complète cette gentille lettre.

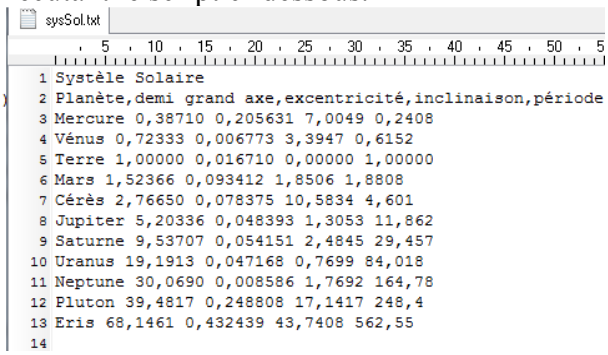


Évidemment, personne n'utilise Python pour écrire des lettres de ce type. On choisira plutôt d'écrire dans un fichier si l'on dispose de données formatées que l'on voudrait pouvoir traiter ensuite. Par exemple, dans un lexique français-anglais, on a un mot en français suivi de sa nature (au choix parmi certains mots-clés) et de ses traductions en anglais (plusieurs propositions de traduction pouvant être proposées). Par exemple, on peut avoir *lien,n.m.,link,bond* ou *faible,adj.,weak,light,low,small,dim,faint,slight,narrow* ou encore *réprimander,vt,to reprimand*. Nous donnons juste un exemple de formatage des données, pour lequel on utilise des chaines de caractères (contenant des espaces éventuels), un séparateur (ici la virgule), des mots-clés, un ordre précis même si le nombre de chaînes n'est pas limité (il doit y en avoir au moins trois). Ainsi formatées, les données de notre fichier texte pourront être exploitées par un script Python qui les lira en séparant les différentes chaînes de chaque ligne.

L'instruction pour lire un fichier est, tout d'abord *nom_fichier=open*('*monFichier.txt*','r') et puis ensuite, pour chaque ligne *nom_fichier.readline*() ou bien pour toutes les lignes d'un coup *nom_fichier.readlines*(). La fonction *readline*() renvoie la ligne avec un \n à la fin tandis que *readlines*() renvoie une liste contenant toutes les lignes du fichier (avec un \n à la fin de chaque ligne). On peut ainsi lire toutes les lignes du fichier en écrivant *for* ligne *in* nom_fichier.*readlines*() : <bloc d'instructions>. Si on doit réutiliser ensuite les lignes, il vaut mieux alors affecter la liste à une variable *lignes=nom_fichier.readlines*(). Ainsi, notre lexique français-anglais peut être décrypté en utilisant l'instruction *texte.split*(*sep*) qui, on l'a vu, découpe un *texte* en une liste sur la base du séparateur *sep*. Par exemple, mots='*lien,n.m.,link,bond*\n'.*split*(',') est une liste : mots[0] contient '*lien*', mots[0] contient '*n.m.*', etc. et *print*(mots) affiche ['lien', 'n.m.', 'link', 'bond\n'].

Nous avons tous les outils pour écrire un programme qui traite des données. Prenons un fichier texte qui contient des données sur le système solaire. Nous sommes allés chercher ces données sur Wikipédia : elles concernent les planètes principales et les petites planètes de notre système solaire. La ligne n°1 contient un titre. Le format des données est indiqué dans la 2ème ligne du fichier. Les données proprement dites sont les lignes qui suivent. Nous avons créé le fichier *systSol.txt* en exécutant le script ci-dessous.

```
from os import getcwd,chdir
chdir(getcwd())
systSol=open('sysSol.txt','w')
testFichier.write('Systèle Solaire\n')
testFichier.write('Planète,demi grand axe,excentricité,inclinaison,période\n')
testFichier.write("Mercure 0,38710 0,205631 7,0049 0,2408\n")
testFichier.write("Vénus 0,72333 0,006773 3,3947 0,6152\n")
testFichier.write("Terre 1,00000 0,016710 0,00000 1,00000\n")
testFichier.write("Mars 1,52366 0,093412 1,8506 1,8808\n")
testFichier.write("Cérès 2,76650 0,078375 10,5834 4,601\n")
testFichier.write("Jupiter 5,20336 0,048393 1,3053 11,862\n")
testFichier.write("Saturne 9,53707 0,054151 2,4845 29,457\n")
testFichier.write("Uranus 19,1913 0,047168 0,7699 84,018\n")
testFichier.write("Neptune 30,0690 0,008586 1,7692 164,78\n")
testFichier.write("Pluton 39,4817 0,248808 17,1417 248,4\n")
testFichier.write("Eris 68,1461 0,432439 43,7408 562,55\n")
testFichier.close()
```



Vous remarquerez que le séparateur, ici n'est pas la virgule (employée comme séparateur décimal) mais l'espace. Les nombres ne sont pas au format Python qui exige d'avoir un point comme séparateur décimal.

On peut utiliser la fonction *replace*(',','.') qui change le séparateur décimal et ensuite la fonction *float*() qui convertit la chaîne en flottant. Par exemple, ligne="Terre 1,00000 0,016710 0,00000 1,00000\n".*split*(' ') est une liste contenant les caractéristique de l'orbite terrestre, et *float*(ligne[1].*replace*(',','.')) contient la valeur du demi-grand axe de cette orbite qui est égale à 1.0 (l'unité est l'UA, Unité Astronomique définie justement à partir de ce demi-grand-axe) comme d'ailleurs *float*(ligne[4].*replace*(',','.')) alors que le mot contenu dans ligne[4] est '1,00000\n' (le caractère de fin de ligne n'empêche pas la conversion de la chaîne en nombre).

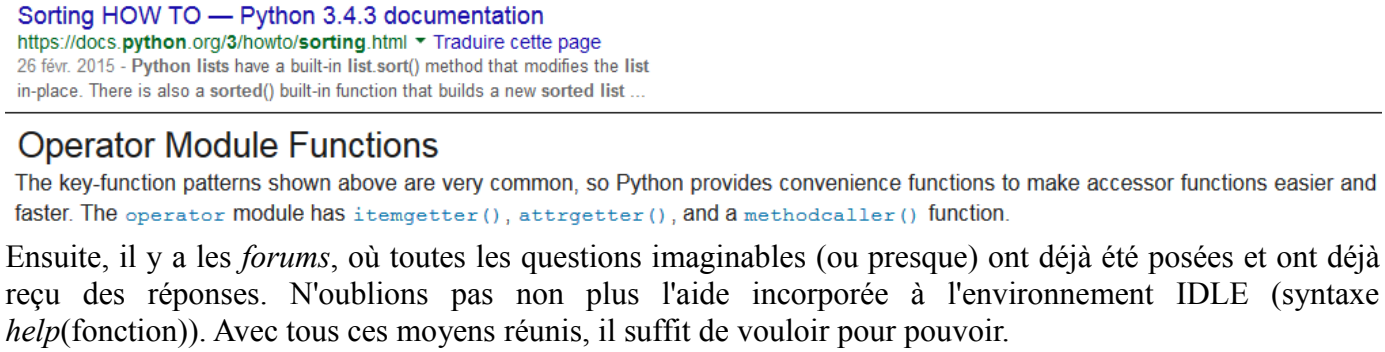
Que peut-on faire d'un tel fichier ? D'abord le lire, mettre chacune des données dans des variables, et selon les cas, effectuer les traitements requis par la situation. Dans le cas qui nous occupe, les planètes sont ordonnées selon le demi-grand axe (approximativement égale à une distance moyenne par rapport au soleil). Supposons que l'on veuille en afficher la liste dans l'ordre des excentricités croissantes (l'excentricité *e* exprime l'écart de forme entre l'orbite et le cercle parfait dont l'excentricité est nulle ; entre 0 et 1 l'excentricité signale une orbite elliptique ; pour *e*=1 on aurait une parabole, cela n'arrive jamais pour un corps en gravitation).

```
from os import getcwd,chdir
import operator
testFichier=open('sysSol.txt','r')
excentricite=list()
rang=-1
for ligne in testFichier.readlines() :
    if rang>0 :
        mots=ligne.split(' ')
        excentricite.append([mots[0],float(mots[2].replace(',','.')),rang])
    rang+=1
testFichier.close()
excentricite.sort(key=operator.itemgetter(1))
print('Planète \t  e \t \t rang')
for ligne in excentricite :
    nom_planete=ligne[0]
    while len(nom_planete)<8 :
        nom_planete+=' '
    print(nom_planete,'\t',ligne[1],'\t',ligne[2])
```

```
Planète          e              rang
Vénus            0.006773       2
Neptune          0.008586       9
Terre            0.01671        3
Uranus           0.047168       8
Jupiter          0.048393       6
Saturne          0.054151       7
Cérès            0.078375       5
Mars             0.093412       4
Mercure          0.205631       1
Pluton           0.248808       10
Eris             0.432439       11
```

Nous utilisons l'instruction *for* ligne *in* nom_fichier.*readlines*() pour lire d'un coup notre fichier. Les données d'une ligne sont alors dans la chaîne *ligne*. Nous créons une autre liste *excentricite*, qui contient la partie des informations qui nous intéresse (le nom et l'excentricité) ainsi que le *rang* dans l'ordre des grand-axes (il s'agit juste de l'indice de la ligne quand on fait commencer les indices à -1, car les deux premières lignes ne concernent pas une planète en particulier).
L'instruction *excentricite.sort*(*key=operator.itemgetter*(1)) permet de classer directement la liste selon le 2ème item. Nous avons créé nos enregistrements comme des listes de trois items : pour la Terre, on a créé la liste *excentricite*[2] qui est ['Terre', '0.016710', '3'] et nous voulons trier les éléments selon la valeur de l'excentricité qui a le rang dans cette liste. L'argument *key=operator.itemgetter*(1) de la fonction *sort*() permet de réaliser cela.

Bien sûr, on ne peut pas tout savoir, sur toutes les fonctions et tous les modules. C'est pour cela qu'il est bon de savoir où trouver de la documentation. Les ressources pour Python sur internet sont immense. Un moyen très simple et de taper l'interrogation dans un moteur de recherche comme Google. Ici, j'ai tapé « sort list python 3 » dans *Google* et la 1ère réponse est la documentation de python.org qui donne tout ce qu'on veut savoir, et le lien nous donne la bonne page de cette documentation.


Sorting HOW TO — Python 3.4.3 documentation
https://docs.python.org/3/howto/sorting.html ▾ Traduire cette page
26 févr. 2015 - Python lists have a built-in list.sort() method that modifies the list in-place. There is also a sorted() built-in function that builds a new sorted list ...

Operator Module Functions

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The `operator` module has `itemgetter()`, `attrgetter()`, and a `methodcaller()` function.

Ensuite, il y a les *forums*, où toutes les questions imaginables (ou presque) ont déjà été posées et ont déjà reçu des réponses. N'oublions pas non plus l'aide incorporée à l'environnement IDLE (syntaxe *help*(fonction)). Avec tous ces moyens réunis, il suffit de vouloir pour pouvoir.

```
Planète         e       rang
Vénus       0.006773            2
Neptune             0.008586            9
Terre       0.01671             3
Uranus      0.047168            8
Jupiter             0.048393            6
Saturne             0.054151            7
Cérès       0.078375            5
Mars        0.093412            4
Mercure             0.205631            1
Pluton      0.248808            10
Eris        0.432439            11
```

Nous n'avons pas expliqué ce qui ressemble à un « bidouillage » et qui permet de réaliser un affichage propre (les données en colonnes). Le caractère de tabulation est \t mais cela est interprété par une tabulation de huit espaces, or quatre planètes ont des noms de sept lettres (Neptune, Jupiter, Saturne et Mercure), ce qui fait qu'avec l'espace supplémentaire ajouté par *print*(), on arrive à huit caractères. L'usage de \t sans précaution conduit à l'affichage ci-contre. Nous avons donc complété les noms de toutes les planètes pour qu'ils aient tous la même longueur. De cette façon on corrige ce problème. La bonne façon de faire est certainement plus raffinée : il doit exister une classe d'objets qui permet un affichage sous la forme d'un tableau bien propre, encadré, etc. mais chercher un moyen puissant pour réaliser une chose simple n'est pas toujours le plus facile... mais cela peut être payant à long terme. On aurait aussi pu utiliser l'environnement graphique *tkinter* vu précédemment pour créer un plus joli tableau.

Nos études astronomiques et pythonesques s'arrêtent ici, nous invitons nos lecteurs à poursuivre le voyage, non pas seuls, mais accompagnés par la multitude des utilisateurs/créateurs que nous sommes, tous, en puissance.

5. Énoncés des questions

Les questions qui suivent n'ont d'autre but que de vous donner quelques sujets de réflexion pour écrire des programmes Python. Les situations à programmer sont potentiellement infinies. Celles-ci ont le mérite d'avoir été corrigées : vous trouverez mes « propositions de correction » avec les commentaires les accompagnant ainsi que les programmes dans la même page de mathadomicile que ce document.

Partie 1: Primalité

a) Écrire un programme qui détermine si un nombre entier *n* est premier (si *n* n'a que deux diviseurs) ou s'il est composé. Dans ce dernier cas, donner la décomposition en facteurs premiers de *n*.

b) Utiliser ce programme pour montrer qu'à partir de k=5 les nombres de Fermat $F_k = 2^{2^k} + 1$ ne sont pas premiers.

c) Montrer, de même, que pour k=2, 3, 5, 7 les nombres de Mersenne, notés $M_k = 2^k - 1$, avec k premier, sont premiers (on les note alors *M1, M2, M3* et *M4*), alors que pour k=11, M_k n'est pas premier. Déterminer la valeur de k pour le 5ème nombre de Mersenne premier, noté *M5*.

d) Il est possible de se focaliser sur la primalité des nombres de Mersenne car un théorème dû au mathématicien français Lucas (1842-1891), permet d'accroître notablement l'efficacité du test : étant donnée la suite (s_n) définie par s_1=4 et, pour tout n>1, $s_n = (s_{n-1})^2 - 2$, le nombre $M_n = 2^n - 1$ est premier si et seulement si il divise s_{n-1}. La difficulté ici va être la croissance extrêmement rapide du nombre de chiffres des termes de la suite (s_n). Par exemple, pour tester si $M_{11} = 2^{11} - 1 = 2047$ est premier, il suffit de tester si ce nombre divise s_{10}. Le problème est que s_{10} s'écrit avec près de 300 chiffres ! Essayer de voir jusqu'où, avec Python sur une machine ordinaire et dans des temps raisonnables, cette simple suite permet d'examiner la primalité des nombres de Mersenne.

Partie 2: Liste de nombres premiers

a) Écrire un programme qui détermine la liste des nombres premiers inférieurs ou égaux à un entier *n* donné. On pourra afficher proprement cette liste (tableau) et donner des informations sur chaque nombre premier (son rang, sa valeur, son résidu modulo 4, modulo 6 ou modulo 10, etc.) ainsi qu'un récapitulatif statistique (effectif total des nombres premiers, pourcentages par résidu dans les différents modulo, etc.). Pour ce faire, on peut partir de rien, et utiliser la méthode du crible d'Ératosthène (un algorithme qui part de la liste des nombres entiers compris entre 2 et un entier *n* donné, et qui raye successivement tous les multiples du premier nombre non rayé, puis du 2ème nombre non rayé, etc.).

b) On peut aussi partir d'une première liste des premiers nombres premiers fournie en argument, et déterminer ceux qui manquent par une adaptation de la méthode précédente.

c) Application n°1 : Une représentation graphique amusante et instructive des nombres premiers consiste à enrouler ceux-ci à la manière d'un escargot autour d'un premier nombre (le germe). L'image ci-contre nous donne une idée de ce que l'on veut obtenir (le germe y est égal à 7). Pour bien identifier les nombres premiers des autres, nous allons dessiner un gros point de couleur pour les nombres premiers et un plus petit point d'une couleur différente pour les autres. Pour se donner quelques repères, on peut écrire les valeurs des nombres premiers ou seulement celles des nombres de la forme $10k$+1 (un quart des nombres premiers) par dessus le point correspondant.

d) Application n°2 : Lorsqu'on additionne tous les diviseurs stricts (inférieurs au nombre) d'un nombre *n*, on fabrique un nouveau nombre *n′* qui peut lui-même suivre le même sort : on additionne tous ses diviseurs pour fabriquer un 3ème nombre *n″*, etc. Si n=10, les diviseurs stricts de 10 étant 1, 2 et 5, on fabrique le nombre n'=1+2+5=8, puis, comme les diviseurs stricts de 8 sont 1, 2 et 4 on fabrique le nombre n''=1+2+4=7. Comme 7 est premier, on se retrouve au nombre n'''=1 et on s'arrête là car 1 n'a pas de diviseur strict. Il se trouve qu'en essayant avec plusieurs valeurs *n* de départ, on s'arrête presque toujours sur 1. Prouver cela en écrivant un programme qui affiche cette suite de nombres partant d'un nombre *n*

quelconque. Explorer le comportement des premiers entiers : longueur de la suite et sens de variation. Essayer de repérer les exceptions à la règle énoncée.

Partie 3: Le petit théorème de Fermat

Ce théorème a été souvent utilisé pour montrer qu'un nombre n'est pas premier, voici comment. Ce théorème s'énonce en disant que, si *p* est un nombre premier et *a*≥*2* un nombre entier non divisible par *p*, alors $a^{p-1}-1$ est un multiple de *p*.

a) Le nombre 7 est premier donc quelque soit l'entier *a*≥2 non divisible par 7, on doit avoir a^6-1 divisible par 7. Vérifier cela, en testant la divisibilité de a^6-1 par 7 pour toutes les valeurs inférieures à 14 qui ne soient pas dans la table de 7. Recommencer le même travail pour d'autres nombres premiers inférieurs à 100.

b) Pour prouver qu'un nombre *n* n'est pas premier, il suffit de montrer que $a^{n-1}-1$ n'est pas divisible par *n* (c'est la forme contraposée du théorème) pour au moins une valeur de *a*≥*2* non divisible par *n*. Par exemple, 91 n'est pas premier car $2^{90}-1=1237940039285380274899124223$ n'est pas divisible par 91 (le reste est 63). Et, en effet, $91=7\times13$, 91 est un nombre composé (on aurait pu le savoir bien plus facilement, sans utiliser ce théorème).

Mettre au point un algorithme qui teste, avec cette méthode la non-primalité d'un entier *n*. On pourra par exemple tenter de prendre *a*=2 et montrer que $b=2^{p-1}-1$ n'est pas divisible par *n* (utiliser la fonction % : reste de la division euclidienne), si ce n'est pas le cas, on tente la division par le nombre premier suivant, etc. Tester, par exemple, la non-primalité de $10^3+1=1001$ puis de $10^7+1=10000001$.

c) Par contre, on ne peut utiliser directement la réciproque qui est fausse généralement : ce n'est pas parce qu'il existe un entier *a*≥2 premier avec l'entier impair *n* tel que $a^{n-1}-1$ soit un multiple de *n* (on écrit cela aussi $a^{n-1}\equiv1(mod\ n)$, que *n* est un nombre premier... De tels nombres impairs *n* sont appelés *nombres pseudo-premiers de base a* (on précise parfois que ce sont des nombres pseudo-premiers de Fermat car il existe d'autres sortes de nombres pseudo-premiers). Si *a* est égal à 2, et si *a* est quelconque (pas nécessairement premier avec *p*) de tels nombres pseudo-premiers sont appelés *nombres de Poulet*. Si la propriété est vérifiée pour tout entier *a* compris entre 2 et *n*, le nombre est appelé *nombre de Carmichael*.

Retrouver, à l'aide d'un algorithme, les dix premiers nombres de Poulet (la liste commence par 341, 561, 645, 1105 et 1387) et les dix premiers nombres de Carmichael (la liste commence par 561, 1105 et 1729).

Quels sont les premiers nombres pseudo-premiers de base 2, 3, 4, 5, etc. qui soient supérieurs à leur base ? (pour 2, 3 et 4 la réponse est 341, 91 et 15).

Partie 4: Récursivité et efficacité

a) Écrire un programme itératif (utilisant uniquement des boucles *for* et *while*) qui calcule le PGCD de deux nombres entiers *a* et *b* en traduisant l'algorithme d'Euclide. Écrire ce programme en mode récursif (utilisant une fonction qui s'appelle elle-même).

b) Parfois l'écriture récursive est moins efficace que son équivalent itératif, car les appels multiples à la fonction créent autant d'environnements d'exécution qu'il en faut. Ces environnements s'ignorant les uns les autres peuvent être redondants. Pour mieux comprendre cela, programmer le calcul d'un terme de la célèbre suite de Fibonacci : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc. (chacun des termes étant la somme des deux précédents) dans les deux modes (itératif et récursif). Mesurer les temps d'exécution avec la fonction *clock*() du module *time* : cette fonction renvoie un temps très précis de l'horloge interne en secondes ; pour mesurer une durée, il faut faire la différence entres le temps de fin et de début d'exécution.

c) Écrire un programme itératif qui calcule la factorielle $n!=1\times2\times...\times n=\prod_{k=1}^{n}k$ d'un nombre entier *n*.

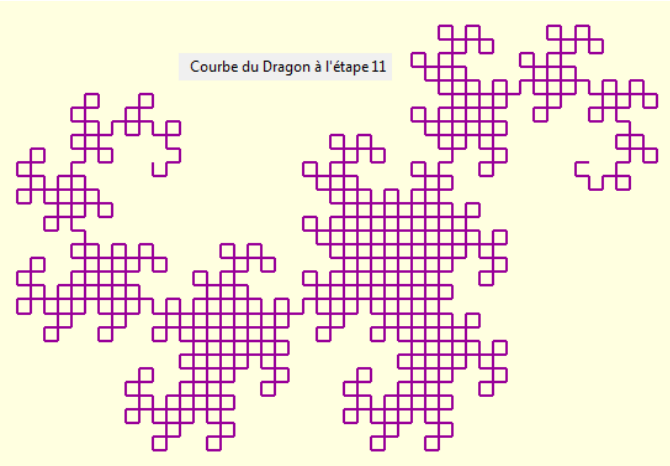
En déduire un programme qui calcule les coefficients binomiaux $\binom{n}{k}=\frac{n!}{k!(n-k)!}$, aussi appelés (en dénombrement) nombres de combinaisons de *k* éléments pris parmi *n*, ou coefficients du triangle de Pascal.

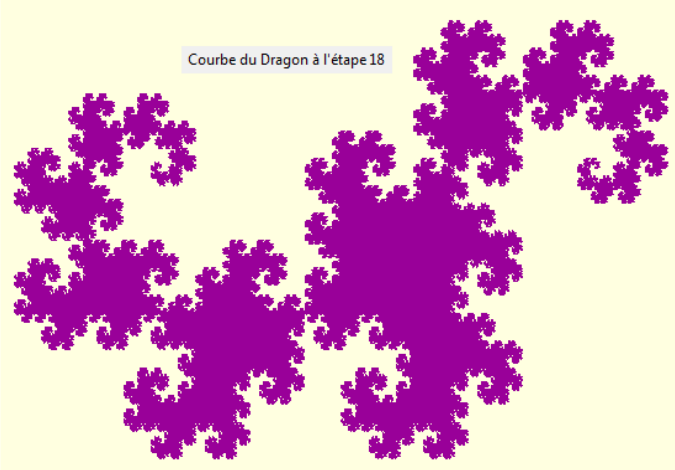
Écrire ces deux programmes en mode récursif (utilisant une fonction qui s'appelle elle-même).

d) Le mode récursif est réputé plus efficace que le mode itératif dans certains cas. Pour trier une liste de nombres initialement dans le désordre, on peut procéder ainsi : choisir un des nombres comme pivot, comparer les nombres de la liste au pivot et les ranger dans deux listes distinctes (les nombres inférieurs dans *listInf* et les nombres supérieurs dans *listSup*), appliquer la même méthode de tri sur les deux listes. Écrire le programme récursif qui effectue le tri de cette façon. Tester ce programme sur une liste de n nombres tirés au hasard. Écrire un programme itératif qui réalise le même travail et comparer les deux sur leurs performances respectives.

e) Pour prolonger la réflexion sur les méthodes récursives de programmation, nous avons vu qu'il faut éviter les redondances qui sont la plaie des programmes récursifs (du moins pour des traitements de grande ampleur). Comment alors programmer la situation suivante où on a calculé qu'une probabilité p_n vaut $p_n = p_{n-1} + \frac{1-p_{n-6}}{26^6}$? Il s'agit de la probabilité d'écrire un mot spécifique de six lettres comme « COGITO » (*je pense*, en latin) lorsqu'on tape n lettres au hasard[3]. La situation est aggravée par le fait que n est nécessairement grand (n=2000 est un minimum envisageable) sinon p_n va être ridiculement petit.

f) Pour finir notre incursion au pays des fonctions récursives, et pour montrer que leur domaine d'application n'est absolument pas limité aux nombres, laissons nous aller à dessiner la *courbe du dragon*. Cette courbe fractale a de nombreuses propriétés comme celle, inattendue, de paver le plan avec des répliques d'elle-même. Pour la construire, on part d'un segment à la 1ère étape, que l'on remplace par un chevron (si le segment initial est la diagonale d'un carré, le chevron est constitué par un des demi-carrés que découpe cette diagonale), et on fait ensuite de même avec les deux nouveaux segments. Les chevrons nouvellement créés sont alternativement tournés de part et d'autre de la courbe dont ils sont issus. Pour mieux comprendre cela, on peut observer le début de la construction sur l'illustration ci-dessous. Si D symbolise un virage à Droite et G un virage à Gauche : à la 2ème étape on a G, à la 3ème on a GGD (nous indiquons avec la couleur verte la séquence précédente, et avec la couleur rose la séquence doublement inversée qui s'ajoute, après un G médian). La séquence GGD devient, à la 4ème étape, GGDGGDD. La 1ère inversion (le début devenant la fin) de GGD conduit à DGG, et il s'ajoute une 2de inversion (les D se changeant en G) qui donne finalement le GDD final. Le processus continue ainsi jusqu'à l'infini, du moins en théorie, mais nous voulons obtenir l'étape n.





Partie 5: Modules et classes

Les fractions permettent d'écrire les nombres rationnels à partir de deux nombres entiers, le numérateur et le dénominateur, le second devant être non nul. Parmi toutes les fractions égales, il existe une unique fraction plus simple que toutes les autres : la *fraction irréductible*. Les nombres rationnels ont une autre propriété : tous ont une écriture décimale qui est périodique à partir d'un certain rang. Pour $\frac{2}{3} = 0{,}66666\ldots$, c'est le chiffre 6 qui se répète à partir du rang des dixièmes (-1), précédé par le nombre 0 ; pour $\frac{13}{11} = 1{,}181818\ldots$ c'est la suite de chiffres 18 qui se répète à partir du rang des dixièmes (-1), précédée par le nombre 1 ; pour $\frac{15}{14} = 1{,}071428571428571\ldots$, c'est la suite de chiffres 714285 qui se répète à partir du

3 Cette situation d'*un singe dactylographe*, imaginée par Émile Borel au début du XXe, est mise en scène dans le livre de Benoît Rittaud : *L'assassin des échecs et autres fictions mathématiques* (coll. Plumes de Sciences, ed. Le Pommier, 1992), pp.165-179.

rang des centièmes (−2), précédée par le nombre 1,0 (le 0 est important ici).

a) Écrire les fonctions nécessaires à l'obtention de l'écriture irréductible d'un nombre rationnel donné par deux nombres *a* et *b* (le numérateur et le dénominateur de la fraction) ainsi que celles donnant la suite de chiffres se répétant, le rang à partir duquel cette suite apparaît dans la partie décimale et la partie précédent cette suite (appelé partie non périodique du quotient). Toutes ces informations doivent pouvoir être obtenues à partir d'une classe que vous appellerez « Frac » (on ne se sert pas encore de la classe « Fraction » qui existe déjà dans Python).

b) Notre classe *Frac* peut s'étoffer encore un peu : nous voulons pouvoir obtenir l'écriture de notre fraction sous sa forme de fraction continue. Par exemple $\frac{105}{8}=13+\frac{1}{8}$ ou $\frac{51}{7}=7+\frac{1}{3+\frac{1}{2}}$. On notera ces fractions avec une notation linéaire [13,8] et [7,3,2]. Ainsi, toutes les fractions peuvent s'écrire sous la forme d'une liste finie $[a_0, a_1, a_2, \ldots, a_n]$, cette notation ayant un intérêt qui dépasse largement ce niveau anecdotique. Comment obtient-on les coefficients de cette notations : avec l'algorithme d'Euclide (encore), les quotients partiels de chaque étape de la recherche du PGCD nous les fournit : $51=7\times\underline{7}+2$ (donc $a_0=7$) puis $7=2\times\underline{3}+1$ (donc $a_1=3$) et enfin $2=1\times\underline{2}+0$ (donc $a_2=2$). Une fois définie cette fonction *fracContinue*(), nous voudrions pouvoir instancier un objet de la classe *Frac* avec une liste d'entiers. Par exemple, nous voudrions pouvoir faire *a=Frac*([7,3,2]) et que dans *a.num* on trouve 51.

c) Notre classe *Frac* peut sans doute accepter encore un autre type de déclaration : nous voulons pouvoir définir une fraction par la partie non-périodique de son développement décimal et la suite de chiffres qui se répète (deux chaînes de caractères). Par exemple *a=Frac*("2.1","6") doit être interprété comme le nombre 2,16666... qui est égal à la fraction $\frac{13}{6}$. Pour réaliser cela, on calcule $10^1 a-a=9a$ (1 car il n'y a qu'un seul chiffre qui se répète) qui vaut autant que 21,6−2,1 (les chiffres d'après sont identiques dans les écritures décimales) soit 19,5. Il ne reste plus qu'à simplifier $\frac{19,5}{9}=\frac{195}{90}=\frac{13\times15}{6\times15}=\frac{13}{6}$. Une fois que la classe *Frac* est au point, la transposer comme une classe héritée de la classe *Fraction* du module *fractions* de Python (*from fractions import Fraction*).

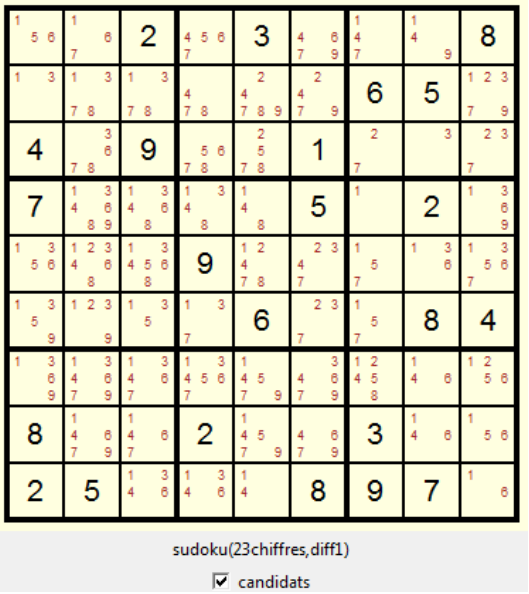
d) Pour compléter notre étude des classes Python et enrichir le domaine de nos investigations (il n'y a pas que les nombres!), intéressons nous à un jeu. Il est assez simple de résoudre un sudoku quand on envisage toutes les possibilités, les unes après les autres. L'algorithme de *backtracking* réalise cela : on essaie méthodiquement le remplissage de la grille (sans réfléchir), chiffre après chiffre, et si cela conduit à une impasse, on revient au point précédent où l'on augmente le chiffre, jusqu'à ce qu'on ai essayé toutes les possibilités, dans ce cas on revient encore plus en arrière, etc. En principe une bonne grille de sudoku ne doit avoir qu'une seule solution, nous la cherchons. L'objectif fixé ici : la grille initiale (une liste de 81 éléments donnée par un fichier) devient une instance de la classe *Grille* qui contient les méthodes nécessaires à la recherche de la solution et à son affichage.

e) Vous noterez que l'algorithme employé ici pour résoudre les sudokus n'est pas du tout intelligent : il essaie bêtement toutes les possibilités jusqu'à trouver une solution. S'il n'y a en a pas, il va produire une erreur (rien n'est prévu pour ce cas) et s'il y en a deux ou plus, il ne nous le dira pas (et c'est pourtant disqualificatoire pour un sudoku). Nous voulons améliorer notre programme et compter les solutions (aller, tout aussi bêtement, jusqu'au bout du processus). Pour prolonger cette étude, nous pouvons essayer de générer une grille de sudoku valide (a une solution unique) et dont le nombre de chiffres de la grille initiale n'est pas trop grand (entre 17 qui est la valeur minimum et 25, une valeur raisonnable arbitraire) .

Partie 6: Interactivité et widgets graphiques

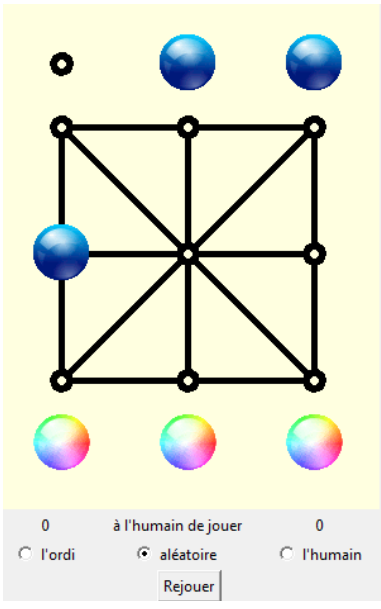
Nous avons déjà rencontré quelques situations interactives – où l'utilisateur peut converser avec le programme – quand nous avons utilisé le module *tkinter* et découvert quelques uns de ses nombreux dispositifs graphiques. Mais l'interactivité est également présente dans un simple programme qui demanderait à l'opérateur d'entrer son nom jusqu'à ce que le nom entré ne contienne aucun chiffre. Cela pour dire que la notion d'interactivité recouvre des fonctionnalités très diverses, pas forcément compliquées, qui permettent de moduler les actions du programme.

a) Pour continuer avec les sudokus, nous aimerions disposer d'un programme qui affiche une grille initiale et offre la possibilité de la compléter avec des chiffres entrés au clavier. La fenêtre d'affichage doit être rendue sensible (la méthode *bind*() de la classe *Tk* réalise cela) et réagir aux entrées du clavier en lançant un évènement (*KeyPress*) qui permet à la commande *ajoute*() de se servir du chiffre entré. Le programme doit reconnaître une grille complète et valide en affichant le message « Bravo ! La solution a été trouvée en … secondes »). On peut prévoir aussi une case à cocher *Checkbutton*() qui, lorsqu'elle est cochée, déclenche l'affichage des chiffres « candidats » dans les cases vides.



b) Changeons de sujet et faisons un pas de plus dans le monde merveilleux des fractales ! Les ensembles de Julia sont des images fractales construites selon un procédé assez simple. À chaque pixel P de coordonnées $(x;y)$ nous allons associer une couleur $coul(r,b,v)$ selon la 1ère valeur de l'entier n pour laquelle l'image P_n du point P s'écarte de l'origine $O(0;0)$ du repère d'une différence supérieure ou égale à 2. Pour trouver les coordonnées $(x';y')$ de l'image P_n, à partir de celles $(x;y)$ de P_{n-1} on applique les relations : $x'=x^2-y^2+x_I$ et $y'=2xy+y_I$ où $(x_I ; y_I)$ sont les coordonnées d'un point I du plan. Lorsque le point P_n ne sort pas (en essayant les valeurs de n jusqu'à une certaine valeur maximum appelée *prof* pour profondeur) du disque de rayon 2 centré sur O, la couleur du point est noire, sinon la couleur est définie par un choix arbitraire, par exemple le dégradé de vert $coul(0, \frac{(prof-n)\times 255}{prof}, 0)$. En procédant ainsi, on obtient l'image de l'ensemble de Julia associé au point I. Programmer l'affichage d'une telle image pour x et y compris entre -2 et 2, avec la possibilité de modifier les coordonnées de I et la possibilité de zoomer sur une partie de cette image (par exemple en cliquant dans le cadre deux extrémités de la nouvelle fenêtre en diagonale).

c) Donnons-nous un objectif un peu plus ludique que la simple exploration d'une image statique, fusse t-elle magnifique, et un peu plus ambitieux que la programmation du sudoku où les contraintes ne laissent aucune possibilité de variation. Un jeu simple à deux joueurs dont l'un des joueurs est l'ordinateur et qui nécessite une forme d'intelligence de sa part : le jeu des neuf trous appelé aussi *tapatan*. Le plateau de jeu est une grille carrée de 3 sur 3 et chaque joueur dispose de trois pions qu'il doit aligner sur le plateau. Au début, chacun à tour de rôle pose un pion, n'importe où sur le plateau. Une fois les six pions posés, chacun peut déplacer un de ses pions d'une position, selon un des tracés figurés sur le plateau (le long des six lignes ou des deux diagonales), si le nouvel emplacement est libre. Programmer ce jeu avec un compteur des parties gagnées/perdues et un dispositif permettant de choisir si c'est le joueur qui commence (plus facile), si c'est l'ordinateur (plus difficile) ou bien si ce choix est laissé au hasard (équilibré). Dans un 1er temps, on peut se consacrer à la réalisation du jeu, sans chercher une réponse intelligente de la part de l'ordinateur (rendre ses choix aléatoires).



d) Pour finir en beauté, envisageons une application graphique qui utilise un menu. Fixons un but à ce programme : réaliser des rosaces, des figures ayant n axes de symétrie concourants (n est variable, supérieur ou égal à 2). Les « pinceaux » ont une couleur et une taille réglables. On trace des lignes de plusieurs types (des segments, des arcs de cercles, des lignes libres, etc.) et on doit pouvoir gommer (supprimer des éléments). Dans un 2ème temps, on peut prévoir un enregistrement de l'image ainsi crée au format *png* ou une exportation de cette image dans le « presse-papier ».