2.1 Corrections

2.1.1 Introduction à la programmation en Python

Mode interactif

Correction de l'exercice 2.1 (Opérateurs)
1. Réponse de l'interpréteur :
- `2*3 : 6` (produit)
- `2**3 : 8` (puissance)
- `20/3 : 6.666666666666667` (quotient : nombre flottant)
- `20//3 : 6` (partie entière du quotient : nombre entier)
- `20%3 : 2` (reste de la division euclidienne : nombre entier)
- `11 == 132/12 : True` (résultat du test : variable booléenne) constater que la division est effectuée et que `11 == 11.0 : True`.
- `2 <= 8 and 15.0 != 15 : False` (résultat du test : variable booléenne) c'est faux à cause du `15.0 != 15` qui est faux car `15.0 == 15 : True`.

2. Le rôle de l'opérateur `&` sur des entiers est d'effectuer un « et » logique sur chaque bits des nombres exprimés en binaire. Exemple `2&3 : 2` car en binaire `2&3 : 10&11` et seul le bit de 2^1 est commun aux deux nombres ; on a `5&3 : 1` car en binaire `5&3 : 101&11` et seul le bit de 2^0 est commun aux deux nombres.
Les opérandes `|` et `^` effectuent, respectivement, un « ou inclusif » logique et un « ou exclusif » logique sur chaque bits des nombres exprimés en binaire.
`a<<b` et `a>>b` effectuent un décalage de `b` bits, respectivement, vers la gauche (avec adjonction de zéros à droite) ou vers la droite (avec suppression des chiffres perdus à droite) sur l'ensemble des bits du nombre `a`exprimé en binaire.

3. Le calcul `111**2` donne 12321 ; si on tape ensuite `_**2`, on obtient 151807041, soit `12321**2`.
Le rôle du caractère de soulignement `_` est de rappeler le dernier résultat.

Correction de l'exercice 2.2 (Types)
1. Le calcul `1-1/3-2/3` ne donne pas 0, contrairement à ce qu'on serait en droit d'attendre.
On obtient `1.1102230246251565e-16` soit $1,1102230246251565 \times 10^{-16} \approx 0,000000000000000111$.
La division `1/3` transforme le type des données : le résultat est un flottant, codé en double précision. Du fait des arrondis qui portent sur le 16^{e} chiffre décimal, le résultat n'est pas nul. D'ailleurs `1-1/3` donne `0.6666666666666667` alors que `2/3` donne `0.6666666666666666` : la différence de ces nombres décimaux n'est pas nulle.

2. Le calcul `88**99` donne un très grand nombre entier :
3189954899106468738519431331435374548486457306565071277011188404860475359372836550565046276541670 20...5152
alors que `88.1**99` donne le flottant `3.5695620257917767e+192` ce qui est évidemment un arrondi de la vraie valeur qui contient 193 chiffres (donnés par l'instruction précédente). On aurait pu comparer `88.**99` et `88**99` qui sont sensés calculer le même nombre mais le point décimal transformant automatiquement le type du calcul en nombre flottant, on obtient `3.1899548991064687e+192` pour la 1^{re} expression.

3. Le calcul `2**1000` donne un très grand nombre entier :
1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695 85...9376
alors que `_*1.` reprend ce nombre et le multiplie par le flottant `1.0` ce qui a pour effet de transformer le type du calcul en nombre flottant et on obtient `1.0715086071862673e+301` qui est une approximation flottante du 1^{er} nombre.

4. Type des expressions :
- `123/3` flottant (`<class 'float'>`)

- `123//3` entier (`<class 'int'>`)
- `123%3` entier
- `123.%3` flottant, du fait du point
- `123>3` booléen (`<class 'bool'>`), résultat d'un test
- `"123"+"3"` chaînes de caractères (en anglais *string*) (`<class 'str'>`), concaténation de chaînes

Correction de l'exercice 2.3 (Built-in Python)
1. `int(4.2)` donne l'entier `4` et `int(-4.2)` donne l'entier `-4`.
La fonction `int()` convertit un flottant en entier, en prenant la partie entière (devant la virgule). Ce n'est pas un arrondi car `floor(-4.9)` donne aussi l'entier `-4` ; ce n'est pas non plus la valeur plancher (obtenue avec la fonction `math.floor` du module `math`) car `math.floor(-4.2)` donne l'entier `-5`.
`int("57")` donne l'entier `57` (conversion d'une chaîne de caractères numérique en décimal)
`int("110",2)` donne l'entier `6` (conversion du binaire en décimal)
`int("ff",16)` donne l'entier `255` (conversion de l'hexadécimal en décimal).
La fonction `int("n",p)` où `n` et `p` sont des entiers convertit en décimal l'entier `n` qui est donné en base `p`. Attention, ça ne marche que si on donne une valeur possible : `int("ff",15)` conduit au message d'erreur suivant *invalid literal for int() with base 15 : 'ff'* .

2. `float(4)` donne le flottant `4.0` (conversion d'un entier en flottant)
`float(2**100)` donne le flottant `1.2676506002282294e+30` (idem)
La fonction `float()` convertit un entier en flottant ; si l'entier est grand, elle en arrondit la mantisse à 16 chiffres décimaux environ.

3. `round(5/6,7)` donne `0.8333333` (arrondi de `0.8333333333333334` à 7 chiffres après la virgule)
`round(5/6)` donne l'entier `1` (arrondi du flottant `0.8333333333333334` à l'entier le plus proche)
`round(5/6,0)` donne le flottant `1.0` (arrondi de `0.8333333333333334` au flottant entier le plus proche)
`round(1000/3,-1)` donne le flottant `330.0` (arrondi de `333.3333333333333` à la dizaine la plus proche)
L'aide de la fonction `round` nous indique les règles suivies par cette fonction, en particulier la syntaxe `round(number, ndigits=None)` indique que le 2e argument est optionnel, en son absence il est considéré comme `None` c'est-à-dire que le résultat sera entier.

4. `max(1,-2,5)` donne `5` (le maximum entre les trois nombres proposés).
`max([1,-2,5])` donne également `5` (le maximum entre les trois nombres de la liste).
`max({1,-2,5})` donne encore `5` (le maximum entre les trois nombres de l'ensemble).
`min("b","a","c")` donne `"a"` alors que `min("B","a","c")` donne `"B"` car le caractère `"B"` est avant le caractère `"a"` dans la table ASCII (les numéros d'ordre de ces caractères sont, respectivement, 66 et 97).
`min("BAC")` donne `'A'` (la chaîne de caractères `"BAC"` est considérée comme une liste) et `min("Bac","BAC")` donne `'BAC'` (car le caractère `"A"` est avant le caractère `"a"` dans la table ASCII).
Pour Python, les caractères sont rangés dans l'ordre de la table ASCII. Les caractères accentués, qui ne sont pas dans la table ASCII, ont le numéro d'ordre renvoyé par la fonction `ord`. Par exemple `ord("e")` renvoie `101` alors que `ord("é")` renvoie `233`. Du coup, `min("bébé","boule")` renvoie `'boule'`, ce qui n'est pas l'ordre suivi par le dictionnaire.

5. Autres fonctions pré-programmées :
- Conversions :
  `bin(2**10-1)` donne `'0b1111111111'` (valeur en binaire de $2^{10}-1$)
  `hex(125)` donne `'0x7d'` (valeur en hexadécimal de 125)
  `oct(9)` donne `'0o11'` (valeur en octal de 9)
  `str(5)` donne `'5'` (conversion en chaîne de caractères du numérique 5)
  `ord("H")` donne `72` (code numérique du caractère H)
  `ord("h")` donne `104` (code numérique du caractère h)
  `chr(35)` donne `'#'` (caractère ayant le code numérique 35)
- Affichage :
  `print("Il y a",3,"jours")` donne `Il y a 3 jours` (les trois données sont affichées à la suite

avec un espace pour les séparer)
`print("Il y a"+str(3)+"jours")` donne `Il y a3jours` (il manque des espaces autour de 3, ou plutôt les espaces n'ont pas été ajoutées automatiquement ; si on les veut, il faudrait écrire `print("Il y a "+str(3)+" jours")`).
`help(print)` nous donne la syntaxe complète de cette instruction d'affichage. En particulier l'option `sep` (le séparateur) est par défaut égale à un espace et l'option `end` (le caractère ajouté à la fin) est par défaut un retour à la ligne. On peut modifier ces options par défaut pour obtenir des effets inhabituels. Supposons par exemple que l'on veille afficher plusieurs durées au format hh :mm :ss, à la suite les unes des autres (au lieu de les avoir sur des lignes séparées). On pourra écrire, dans une boucle où sont définies les valeurs des variables h, m et s : `print(h,m,s,sep=':',end=' ')`.

- Tri de liste :
  `sorted([1,5,2,8,3])` donne `[1, 2, 3, 5, 8]` (la liste est triée selon les valeurs numériques)
  `sorted(["H","T","M","L"])` donne `['H', 'L', 'M', 'T']` (la liste est triée selon les valeurs ordinales des chaînes de caractères)
  `sorted("HTML")` donne également `['H', 'L', 'M', 'T']` (la chaîne de caractères est ici aussi considérée comme une liste de caractères)
- Longueur de liste :
  `len([1,5,2,8,3])` donne `5` (la longueur de la liste est son nombre d'éléments)
  `len('abcd')` donne `4` (la longueur de la chaîne de caractères)
  `len(range(1,10))` donne `9` car `range(1,10)` conduit à 9 éléments qui vont de 1 à 9 (la dernière borne est toujours exclue).

Correction de l'exercice 2.4 (Module math)
1. `cos(0)` conduit à l'erreur *name 'cos' is not defined* si on n'a pas importé le module `math` car la fonction `cos` n'est pas un built-in de Python. Il faut importer le module `math` : avec `import math`. Mais le module `math` étant importé, pour obtenir la valeur de $cos(0)$, il faut taper `math.cos(0)`. L'instruction d'importation ne dispense pas de préfixer le nom de la fonction par le nom du module.
Pour importer la fonction `cos` du module `math` dans l'espace des noms accessibles directement, il faut écrire `from math import cos` (cela n'importe que le nom et la fonction indiquée du module) ou bien `from math import *` (cela importe tous les noms et toutes les fonctions du module, ce qui n'est pas forcément une bonne pratique). On peut alors écrire `cos(0)` et ne pas avoir de message d'erreur, mais la bonne réponse qui est `1.0`.

2. La valeur de la constante `pi` donnée par le module `math` est `3.141592653589793` (on l'obtient en tapant `pi` si on a importé toutes les fonctions avec `from math import *` ou en tapant `math.pi` si on a importé le module avec `import math`.
Les autres constantes du module `math` sont `e = 2.718281828459045` (la base des logarithmes népériens), `tau = 6.283185307179586` (le double de π) et deux autres constantes plus surprenante : `inf` (l'infini) et `nan` (initiales de *not a number*, nono en français pour non-nombre).

3. Résultat des calculs :
- `floor(4.2)` donne `4` et `floor(-4.2)` donne `-5` (on rappelle que `int(-4.2)` donne `-4`).
  Il s'agit du plus grand entier inférieur (valeur plancher).
- `ceil(4.2)` donne `5` (on rappelle que `int(4.2)` donne `4`) et `ceil(-4.2)` donne `-4`.
  Il s'agit du plus petit entier supérieur (valeur plafond).
- `sqrt(49)` et `sqrt(2)` sont tous les deux des flottants alors que la première racine carrée a une valeur entière (on obtient `7.0` et `1.4142135623730951`).
- `pow(2,5)` est le flottant `32.0` alors que `2**5` est l'entier `32`, mais il s'agit fondamentalement du même nombre. Par contre, `pow(2,0.5)`, `2**0.5` et `sqrt(2)` donnent le même nombre flottant `1.4142135623730951`. De même avec `pow(49,0.5)`, `49**0.5` et `sqrt(49)` qui donnent le même flottant `7.0`.
- `degrees(pi)` donne `180.0`, `radians(90)` donne `1.5707963267948966` (la valeur flottante de $\frac{\pi}{2}$), `asin(0.5)` donne `0.5235987755982989` (la valeur flottante de $\frac{\pi}{6}$,

`sin(pi/6)` donne `0.49999999999999994` (une valeur flottante de 0,5).
Python considère qu'un angle, par défaut, est donné en radians. Si on tape `sin(90)`, on n'obtient pas 1 car 90 est un nombre de radians, on obtient `0.8939966636005579`. Si je veux le sinus de 90°, je dois taper `sin(radians(90))` qui me donne bien `1.0`.

Correction de l'exercice 2.5 (Autres modules)
1. Le module `random` étant importé, l'instruction `help(random)` renvoie un grand nombre d'informations sur le module et, en particulier, celles qui suivent mais regarder également les fonctions `choice` ou `shuffle` qui sont parfois bien utiles :

- la syntaxe de la fonction `randint` :
  randint(self, a, b)
  Return random integer in range [a, b], including both end points. Il s'agit donc de fixer les limites (entre `a` inclus et `b` inclus) de l'intervalle d'entiers dans lequel on souhaite en tirer un aléatoirement. Par exemple `randint(2,5)` tire au hasard, à chaque fois, un des quatre nombres de l'ensemble $\{2, 3, 4, 5\}$.
- la syntaxe de la fonction `randrange` :
  randrange(self, start, stop=None, step=1, _int=<class 'int'>)
  Choose a random item from range(start, stop[, step]). L'instruction `randrange(a,b)` donne donc un nombre entier aléatoire entre `a` inclus et `b` exclu). Par exemple, `randint(2,5)` donne un des nombres de $\{2, 3, 4\}$. On peut adjoindre le pas de l'incrémentation (il est de 1 par défaut) et `randrange(1,6,2)` donne un des nombres de l'ensemble $\{1, 3, 5\}$.
- la syntaxe de la fonction `random` :
  random() -> x in the interval [0, 1). Ici, il faut comprendre que l'on obtient un nombre flottant de l'intervalle $[0, 1[$. Par exemple `random()` peut donner 0.123456789101112 (15 à 16 chiffres décimaux). Associée à quelques instructions basiques, cette fonction suffirait, à elle seule, pour obtenir les autres fonctions du module. Si on veut un nombre aléatoire flottant sur un ensemble plus vaste, on pourrait utiliser une expression affine du genre `a*random()+b` ; par exemple, `6*random()-3` renvoie un nombre de l'intervalle $[-3, 3[$.
- la syntaxe de la fonction `uniform` :
  uniform(self, a, b)
  Get a random number in the range [a, b) or [a, b] depending on rounding. Cela remplace l'astuce signalée avec la fonction `random`. On peut ainsi obtenir un tirage sur l'intervalle de notre choix sans effectuer de calculs : `uniform(0.5,0.8)` donne un nombre flottant aléatoire de l'intervalle $[0.5, 0.8[$, mais il est précisé que la borne supérieure est accessible par une mesure d'arrondi.

2. Le module `time` contient de nombreuses fonctions pour manipuler des données temporelles. Il faut savoir qu'un moment du temps est représenté de deux façons en Python : le nombre de secondes depuis le 1er janvier 1970 (*the Epoch*) ou bien un tuple de 9 entiers (année, mois (de 1 à 12), jour (de 1 à 31), heure (de 0 à 23), minute (de 0 à 59), seconde (de 0 à 59), jour de la semaine (de 0 à 6), jour de l'année (de 1 à 366), DST). En plus des fonctions proposées, regarder `sleep` et `localtime()`. Je rappelle qu'après une instruction `import ...` il faut préfixer les fonctions avec le nom du module.
`time()` (après `from time import time`) ou `time.time()` (après `import time`) donne *the current time in seconds since the Epoch.* Avant d'écrire cette ligne, j'ai obtenu la valeur 1602232430.9904544, après l'avoir écrite 1602232500.1142845. Il s'est écoulé environ 70 secondes.
`time.clock()` donne *the CPU time or real time since the start of the process or since the first call to clock(). This has as much precision as the system records.* Le problème est que cette fonction est dépréciée, donc refusée par Python depuis la version 3.3 et sera enlevée définitivement à la version 3.8 (ma version de Python est la 3.7.7). Pour l'instant, j'obtiens 3399.9423183 un nombre qui indique qu'environ 56 minutes se sont écoulées depuis l'allumage de mon ordinateur.
`perf_counter()` est la fonction préconisée, avec `process_time` par Python pour remplacer `clock`.
`time.perf_counter()` me donne un nombre de secondes équivalent de `time.time()` ; par contre, `time.process_time()` me donne toujours la même valeur `0.390625` dont je ne comprends pas l'intérêt (l'indication de l'aide, *Process time for profiling : sum of the kernel and user-space CPU time*, ne

m'aide pas beaucoup en cela).

3. Le module `datetime` est un peu plus complexe car il contient plusieurs classes d'objets : `date`, `datetime`, `time`, `timedelta`, `timezone`, `tzinfo`. Ces noms de classe préfixent les fonctions utilisées : `datetime.date.today()` donne `datetime.date(2020, 10, 9)` (j'obtiens le jour d'aujourd'hui) tandis que `datetime.datetime.now()` donne `datetime.datetime(2020, 10, 9, 10, 59, 18, 377261)` (j'obtiens le jour et l'heure). Remarquez que le module `time` me donnait déjà ces informations puisque `time.localtime()` me donne `time.struct_time(tm_year=2020, tm_mon=10, tm_mday=9, tm_hour=10, tm_min=49, tm_sec=25, tm_wday=4, tm_yday=283, tm_isdst=1)`.

4. Le module `calendar` est bien compliqué aussi. Il donne des indications sur le calendrier, par exemple `calendar.weekday(2019,9,2)` donne `0` qui indique que ce jour était un dimanche. Les jours de la semaine sont codés par un entier allant de 0 (lundi) à 6 (dimanche).
Aujourd'hui, `calendar.weekday(2020,10,9)` me donne `4` : nous sommes vendredi.

5. Le module `turtle` est un module graphique (présent sur Numworks). `turtle.circle(100,180)` trace un demi-cercle de rayon 100 dans une fenêtre indépendante. Pour tracer le cercle complet, on écrit `turtle.circle(100,360)` ou, tout simplement, `turtle.circle(100)`. L'aide du module est un véritable manuel (8047 lignes). Pour la fonction `circle`, on obtient *circle(self, radius, extent=None, steps=None)*
Draw a circle with given radius. Arguments : radius – a number ; extent (optional) – a number ; steps (optional) – an integer. Avec l'argument optionnel `steps`, on peut réduire le nombre de côtés du polygone régulier qui est effectivement tracé, par exemple `turtle.circle(100, steps=6)` trace un hexagone régulier.
Les nombreuses autres fonctions disponibles du module ne seront pas commentées ici. Télécharger à ce sujet un aide-mémoire ou un manuel détaillé sur internet.

Correction de l'exercice 2.6 (Variables)

1. Lorsqu'on exécute successivement les instructions `a=3`, `b=a*a` et `a=b-2`, les variables `a` et `b` contiennent, respectivement, `7` et `9`. La dernière instruction (`a=b-2`) conduite à écraser la valeur précédente de `a` (3) en la remplaçant par le résultat du calcul `b-2`.

2. Les instructions successives `c=a`, `a=b`, `b=c` conduisent à un échange des valeurs contenues dans `a` et `b`. En Python, on peut obtenir le même résultat en écrivant sur une seule ligne `a,b=b,a`. Passer par une variable intermédiaire (la variable `c`) est motivé par le soucis de ne pas perdre la valeur de `a` lors de l'exécution de l'instruction `a=b`.

3. Les instructions successives `a=a+b`, `b=a-b`, `a=a-b` ont également pour effet d'échanger les valeurs contenues dans `a` et `b`. Cela peut paraître étrange, aussi examinons les contenus des deux variables pas-à-pas :

- Initialisons les valeurs : `a=1` ; `b=5`.
- Après `a=a+b`, on a : `a=1+5=6` ; `b=5`.
- Après `b=a-b`, on a : `a=6` ; `b=6-5=1` (l'ancienne valeur de `a`).
- Après `a=a-b`, on a : `a=6-1=5` (l'ancienne valeur de `b`) ; `b=1`.

La variable intermédiaire `c` n'est pas nécessaire pour effectuer l'échange : on peut confier ce rôle à une des variables, mais il faut admettre que ce n'est pas une construction facilement compréhensible.

4. Les instructions successives `a=input("Entrer un nombre :")` (j'entre le nombre 15) et `print("Le double de ce nombre est",a*2)` produisent une concaténation de mon nombre (j'obtiens l'affichage de 1515 au lieu de 30, le résultat attendu). Ce résultat provient du fait que l'instruction `input()` considère que le nombre entré est une chaîne de caractères. Pour que celle-ci soit interprétée comme un nombre entier, il faut écrire `a=int(input("Entrer un nombre :"))`. Si on veut pouvoir entrer un nombre décimal, il faut écrire `a=float(input("Entrer un nombre :"))`. Il faut convertir

(en franglais *caster*) la chaîne de caractères en un type numérique.

5. Pour demander le rayon d'un cercle et afficher son périmètre, on peut écrire successivement `from math import pi` et
`print("Le périmètre du cercle est",pi*2*float(input("Entrer le rayon : ")))`.
Noter qu'on peut faire plus simplement, en écrivant trois instructions : une pour l'importation, une pour l'affectation d'une variable `r` et l'autre pour l'affichage de `2*pi*r`. Certains se seraient contentés sans doute d'une valeur de π plus approximative (ça fait gagner une ligne) mais ce n'est pas une bonne pratique.

6. Pour demander deux nombres entiers `b` et `a` et afficher la conversion en base 10 de `a` donné originellement en base `b`, on peut écrire les trois instructions successives :
`b=int(input("Entrer la base d'entrée :"))`,
`a=input("Entrer le nombre :")` et
`print("Voici le nombre en base 10 :",int(a,b))`.
Bien sûr, des esprits aiguisés me feront remarquer qu'on peut tout faire en une seule ligne :
`print("Voici le nombre en base 10 :",int(input("Entrer le nombre :"),int(input("Entrer la base d'entrée :"))))`
J'avoue que cela fonctionne bien, mais cela demande d'abord la valeur du nombre avant de demander celle de la base.

Mode de programmation

Correction de l'exercice 2.7 (Entrées-sorties)

1. Pour demander la longueur et la largeur d'un rectangle (en mètres) et afficher le périmètre (en mètres) et l'aire (en mètres carrés) du rectangle, je peux écrire les instructions :
   - `a=int(input("longueur (en m) = "))`
   - `b=int(input("largeur (en m) = "))`
   - `print("périmètre =",(a+b)*2,"m \naire =",a*b,"m²")`

   J'enregistre ce programme, écrit dans l'éditeur de IDLE (pour l'ouvrir, choisir `new file` dans l'onglet `File`) dans mon dossier « Programmes » sous le nom « rectangle1.py ».
   Je l'exécute ensuite dans IDLE (Run>Run Module) ; j'entre `12` puis `15` et obtiens, sur deux lignes : `périmètre = 54 m` et `aire = 180 m²`.
2. Le programme précédent est transformé en la procédure « rectangle » par les instructions suivantes :
   ```
   def rectangle(a,b):
    print("périmètre =",(a+b)*2,"m \naire =",a*b,"m²")
   ```
   J'enregistre ce programme sous le nom « rectangle2.py » et l'exécute :
   après avoir entré `rectangle(12,15)`, j'obtiens les deux mêmes lignes-résultat.

2.1.2 Séquences conditionnelles et boucles

Séquences conditionnelles

Correction de l'exercice 2.8 (Min, Max & Étendue)
Pour déterminer le minimum et le maximum du nombre `d`, il faut :

- initialiser le minimum `mini` à la valeur maximale de `d` : ici le maximum que l'on va pouvoir obtenir est $\sqrt{2}$ (dans un carré de côté 1, la distance depuis un sommet ne peut dépasser la diagonale du carré) ; je majore cette valeur maximum en mettant $1,5$ mais toute valeur supérieure ou égale à $\sqrt{2}$ convient.
- initialiser le maximum `maxi` à la valeur minimale de `d` : ici c'est 0 (une distance étant toujours positive) ; je peux minorer cette valeur minimum en mettant -1 ou toute valeur inférieure ou égale à 0 mais 0 convient.

- Dans la boucle, je dois remplacer la valeur de `mini` par la distance `d` calculée si cette valeur la dépasse ; de même, je dois remplacer la valeur de `maxi` par la distance `d` calculée si cette valeur lui est inférieure.

Le programme complété suivant fait tout cela et donne le résultat attendu.
Je le teste en le lançant plusieurs fois :

1. `min= 0.175532631909694 max= 1.2721978270106926 étendue= 1.0966651951009987`
2. `min= 0.09302892471114922 max= 1.3603242799061777 étendue= 1.2672953551950283`
3. `min= 0.07733316268822003 max= 1.3298131075358703 étendue= 1.2524799448476502`

Remarquer que ce n'est pas une bonne idée d'appeler `min` et `max` les variables pour le minimum et le maximum, car ces noms sont réservés pour Python (surlignés avec un éditeur spécialisé). Pour cette raison j'ai ajouté un `i` à ces deux noms de variable. Voici donc le programme complet.

```
from math import sqrt
from random import random
n,mini,maxi=100,1.5,0
for i in range(n):
  x,y=random(),random()
  d=sqrt(x**2+y**2)
  if d<mini :
   d=mini
  if d>maxi :
   d=maxi
print('min=',mini,'max=',maxi,'étendue=',maxi-mini)
```

Correction de l'exercice 2.9 (Procédures conditionnelles)

1. La procédure « boite » prend deux nombres `a` et `b` comme arguments et affiche le nombre de boîtes contenant `b` objets, nécessaires au rangement de `a` objets.
   J'ai réalisé deux solutions :
   (a) la procédure `boite1` effectue un test pour savoir si le nombre `a` est divisible par `b`, auquel cas on donne le quotient. Sinon on donne le quotient augmenté de 1.
   (b) la procédure `boite2` n'effectue pas de test : elle donne toujours l'arrondi à l'entier par excès (valeur plafond ou `ceil` en Python).

```
from math import *

def boite1(a,b):
    if a%b==0 : print("nombre de boites=",a//b)
    else : print("nombre de boites=",a//b+1)

def boite2(a,b):
    print("nombre de boites=",ceil(a/b))

boite1(49,10)
boite1(50,10)
boite2(49,10)
boite2(50,10)
```

```
nombre de boites= 5
nombre de boites= 5
nombre de boites= 5
nombre de boites= 5
```

   J'enregistre ce programme sous le nom `boite.py` dans mon dossier `programmes`, puis je teste avec `(49,10)` et `(50,10)`. Dans les deux cas, je trouve qu'il faut 5 boites.

2. La procédure « trinome » prend trois nombres `a`, `b` et `c` comme arguments et affiche la ou les solutions de l'équation $ax^2+bx+c=0$ quand elle(s) existe(nt) et affiche `"pas de solution"` quand il n'y en a pas.

```
def trinome(a,b,c):
    delta=b**2-4*a*c
    if delta<0 : print("pas de solution")
    elif delta==0: print("une solution :",-b/(2*a))
    else: print("deux solutions :",(-b+sqrt(delta))/(2*a),"ou",(-b-sqrt(delta))/(2*a))

trinome(1,1,1)
trinome(1,2,1)
trinome(1,3,2)
```

```
pas de solution
une solution : -1.0
deux solutions : -1.0 ou -2.0
```

J'enregistre ce programme sous le nom `trinome.py` dans mon dossier `programmes`, puis je teste avec `(1,1,1)` (pas de solution), `(1,2,1)` (une solution) et `(1,3,2)` (deux solutions.
Remarquer l'écriture des instructions conditionnelles sur une seule ligne (possible seulement quand il n'y a qu'une seule instruction à exécuter) : c'est pratique et cela raccourcit les programmes. Pourtant ce n'est généralement pas conseillé (pour la clarté, mais aussi la possibilité d'ajouter des instructions).

3. La procédure « intersection » prend quatre nombres `a1`, `b1`, `a2` et `b2` comme arguments et affiche, selon les cas, les coordonnées du point d'intersection des droites d'équations `y=a1*x+b1` et `y=a2*x+b2` ou un des messages suivants : `"pas d'intersection"` ou `"droites confondues"`.
Il faut tester le parallélisme des droites en comparant leurs pentes et, dans le cas d'un non parallélisme, déterminer par le calcul approprié les coordonnées du point d'intersection. Il y a un aspect mathématiques à cette question : il faut en trouver la solution en résolvant l'équation $a_1x + b_1 = a_2x + b_2$; pour l'ordonnée, on remplace juste la valeur de l'abscisse trouvée dans une des équations.

```
def intersection(a1,b1,a2,b2):
    if a1==a2 :
        if b1==b2 : print("droites confondues")
        else : print("pas d'intersection")
    else :
        x=(b2-b1)/(a1-a2)
        print("une intersection (",x,";",a1*x+b1,")")

intersection(1,2,1,3)
intersection(1,2,1,2)
intersection(1,2,2,1)
```

```
pas d'intersection
droites confondues
une intersection ( 1.0 ; 3.0 )
```

J'enregistre ce programme sous le nom `intersection.py` dans mon dossier `programmes`, puis je teste avec `(1,2,1,3)` (pas d'intersection : droites distinctes), `(1,2,1,2)` (droites confondues) et `(1,2,2,1)` (une intersection).

Correction de l'exercice 2.10 (Âge)

1. La fonction `age(j,m,a)` détermine l'âge d'une personne à partir des trois arguments donnant sa date de naissance (`j` : jour, `m` : mois, `a` : année).
Avec l'instruction `d=date.today()` et ses attributs, on obtient la date du jour (variables notées `jj` : jour, `mm` : mois, `aa` : année). L'âge est alors calculé en effectuant la soustraction des années, éventuellement diminuée de 1 si l'anniversaire n'est pas encore passé (la condition `m<mm or m==mm and j<jj` est fausse).
Quelques tests sont effectués pour vérifier que la date entrée est cohérente (`0<j<32`, `0<m<13` et date entrée inférieure à date du jour avec le test `a>aa or a==aa and (m>mm or m==mm and j>jj)`).
La fonction est montrée sans les commentaires qui font l'objet de la question 4.

```
from datetime import date

def age(j,m,a):
    if not(0<j<32 and 0<m<13) : return None
    if j>j_mois(m,a) : return None
    d=date.today()
    age=0
    jj,mm,aa=d.day,d.month,d.year
    if a>aa or a==aa and (m>mm or m==mm and j>jj):
        return None
    if m<mm or m==mm and j<jj:
        age=aa-a
    else: age=aa-a-1
    return age
```

```
def j_mois(m,a):
    longMois=[1,3,5,7,8,10,12]
    if m in longMois : return 31
    elif m!=2 : return 30
    elif a%4!=0 or (a%100==0 and a%400!=0):
        return 28
    return 29
```

```
print(age(31,4,2019))
print(age(29,2,2004))
print(age(29,2,2003))
print(age(15,12,2020))
print(age(30,10,2021))
print(age(17,7,1959))
```

```
None
17
None
0
None
62
```

2. La fonction `j_mois(m,a)` prend en argument l'année (`a`) et le mois (`m`) et renvoie la longueur du mois entré.
Grâce à cette fonction, le programme précédent peut déterminer un dernier type d'incohérence dans la date entrée : si on demande un jour qui n'existe pas comme le 31/04/2019 (mois de 30 jours), le 29/02/2003 (année non bisextile, mois de 28 jours).

3. L'illustration ci-dessous montre le programme avec une petite batterie de test.
4. L'illustration montre aussi les commentaires sobres de fin de ligne (en rouge), l'entête du script donnant un titre et un objet (ici il n'y a pas la version, ni la date, ni l'auteur mais ces informations peuvent s'avérer utiles dans certains cas) et les commentaire entre triples guillemets (`"""..."""`) qui décrivent la fonction.

```
"""
Correction de l'exercice 2.10
objet  : Détermination de l'âge d'une personne ou d'un évèvènement du passé
auteur : PM
date   : 2021                                                    """

from datetime import date

def j_mois(m,a):
    """Les arguments sont supposés être des entiers (0<m<13 et a>0)
    >>> Retourne le nombre de jours du mois                      """
    longMois=[1,3,5,7,8,10,12]
    if m in longMois : return 31       #les mois ordinaires de 31 jours
    elif m!=2 : return 30              #les mois ordinaires de 30 jours
    elif a%4!=0 or (a%100==0 and a%400!=0):#teste l'année :si bisextile
        return 28                        #février une année bisextile
    return 29                            #février une année ordinaire

def age(j,m,a):
    """Les arguments doivent être cohérents avec la date du jour
    càd date 0<j<32 et 0<m<13 et aussi entrée < date du jour
    >>> Retourne le nombre entier d'années écoulées depuis la date
    et None si il y a une incohérence                             """
    if not(0<j<32 and 0<m<13) : return None              #incohérence1
    if j>j_mois(m,a) : return None                       #incohérence2
    d=date.today()
    age=0
    jj,mm,aa=d.day,d.month,d.year                  #date d'aujourd'hui
    if a>aa or a==aa and (m>mm or m==mm and j>jj):
        return None                                      #incohérence3
    if m<mm or m==mm and j<jj:                  #anniversaire dépassé
        age=aa-a
    else: age=aa-a-1                            #anniversaire à venir
    return age
```

5. Voir ci-dessous l'aide obtenue sur ces fonctions en tapant `help(age)` ou `help(j_mois)` dans la console (après exécution du programme).

```
>>> help(age)
Help on function age in module __main__:

age(j, m, a)
    Les arguments doivent être cohérents avec la date du jour
    càd date 0<j<32 et 0<m<13 et aussi entrée < date du jour
    >>> Retourne le nombre entier d'années écoulées depuis la date
    et None si il y a une incohérence

>>> help(j_mois)
Help on function j_mois in module __main__:

j_mois(m, a)
    Les arguments sont supposés être des entiers (0<m<13 et a>0)
    >>> Retourne le nombre de jours du mois
```

Boucles

Correction de l'exercice 2.11 (Suite de Syracuse)

1. La fonction `successeurs1(n)` affiche tous les successeurs de `n` jusqu'à 1. Cette fonction utilise une autre fonction – la fonction `successeur(n)` – qui renvoie le successeur de `n` : la moitié de `n` (si `n` pair) ou le suivant de son triple (si `n` impair).
2. La fonction `successeurs2(n)` reprend `successeurs1(n)` mais détermine et affiche à la fin, la longueur `long` de la suite et la valeur maximum `maxi` qui a été atteinte.

3. Je teste mes deux fonctions avec `n=15` puis avec `n=15` pour lequel `maxi=4372` et `long=46`.

```
"""
exercice 2.11
Suite de Syracuse  """
def successeur(n):
    if n%2==0 : return n//2
    return n*3+1

def successeurs1(n):
    """ se contente d'afficher la liste des successeurs du nombre de départ (question 1)"""
    while n!=1:
        print(n,end=" - ")
        n=successeur(n)
    print("1\n")

def successeurs2(n):
    """ afficher la liste des successeurs du nombre de départ
        la longueur de la suite et la valeur maximum (question 2)"""
    long,maxi=0,n
    while n!=1:
        print(n,end=" - ")
        n=successeur(n)
        long+=1
        if n>maxi : maxi=n
    print("1")
    print("longueur de la suite=",long)
    print("maximum de la suite=",maxi,'\n')

successeurs1(15)
successeurs2(15)
successeurs2(127)
```

```
15 - 46 - 23 - 70 - 35 - 106 - 53 - 160 - 80 - 40 - 20 - 10 - 5 - 16 - 8 - 4 - 2 - 1

15 - 46 - 23 - 70 - 35 - 106 - 53 - 160 - 80 - 40 - 20 - 10 - 5 - 16 - 8 - 4 - 2 - 1
longueur de la suite= 17
maximum de la suite= 160

127 - 382 - 191 - 574 - 287 - 862 - 431 - 1294 - 647 - 1942 - 971 - 2914 - 1457 - 4372 -
2186 - 1093 - 3280 - 1640 - 820 - 410 - 205 - 616 - 308 - 154 - 77 - 232 - 116 - 58 - 29
- 88 - 44 - 22 - 11 - 34 - 17 - 52 - 26 - 13 - 40 - 20 - 10 - 5 - 16 - 8 - 4 - 2 - 1
longueur de la suite= 46
maximum de la suite= 4372
```

Correction de l'exercice 2.12 (Facteurs premiers)

Petite remarque préliminaire : L'algorithme imposé par l'énoncé, pour déterminer si un nombre entier `n` est premier, est le plus simple que l'on puisse imaginer : on diviser `n` par les entiers allant de 2 à $p \geqslant \sqrt{n}$ (le premier entier qui vérifie cette inégalité convient). Sa justification est simple car elle traduit la contraposée de cette définition : si un nombre premier alors il n'est divisible que par lui-même et par 1. La contraposée est donc : si un nombre est divisible par un entier compris entre 2 et `n-1` alors il n'est pas premier. En outre, on n'a pas besoin d'aller jusqu'à `n-1` car si le nombre est composé (si $n = a \times b$) un des facteurs est inférieur ou égal à $\sqrt{n}$ tandis que l'autre est supérieur ou égal à $\sqrt{n}$.

1. La fonction `premier1(n)` teste si l'entier `n` fourni en argument est premier. Elle renvoie une chaîne de caractères indiquant le résultat.

2. La fonction `premier2` améliore `premier1` en enregistrant les facteurs premiers successifs, éventuellement facteurs multiples (si `n` est divisible par $4 = 2 \times 2$ et pas par $8 = 2 \times 2 \times 2$, la fonction enregistre deux fois le facteur 2, et ensuite aucun autre nombre pair ne peut être un diviseur de `n`). Il suffit ensuite, à ce stade, d'afficher le produit des facteurs enregistrés.

```
exercice 2.12
Décomposition d'un entier en facteurs premiers  """

from math import sqrt

def premier1(n):
    """ renvoie une chaîne de caractères indiquant si n est premier ou multiple (question 1)"""
    diviseur=2
    while diviseur<=sqrt(n) :
        if n%diviseur==0:
            return "non premier car divisible par "+str(diviseur)
        diviseur+=1
    return "premier"

def premier2(n):
    """ renvoie une chaîne de caractères indiquant le produit des facteurs premiers (question 2)
        il n'est pas nécessaire de tester si le facteur est premier comme indiqué dans l'énoncé """
    diviseur=2
    diviseurMax=sqrt(n)
    diviseurs=[]                                    #liste destinée à recevoir les diviseurs de n
    while diviseur<=diviseurMax and n>1 :
        while n%diviseur==0:
            diviseurs.append(diviseur)              #enregistrement d'un facteur premier
            n=n//diviseur
        diviseur+=1
    if len(diviseurs)==0 : return "est premier"

    produit="= "                                    #préparation de l'affichage pour un nombre multiple
    for facteur in diviseurs :
        produit+=str(facteur)+"\u00D7"
    return produit[:-1]                             #suppression du dernier symbole multiplicatif

n=1001
print(n,premier2(n))
n=96
print(n,premier2(n))
n=101
print(n,premier2(n))
```

```
1001 = 7×11×13
96 = 2×2×2×2×2×3
101 est premier
```

```
def premier3bis(n):
    """ renvoie une chaîne de caractères indiquant le produit des facteurs premiers (question 3)
        sous la forme du produit des facteurs à la puissance de sa multiplicité
        en n'écrivant l'exposant que si il n'est pas 1 """
    diviseur=2
    diviseurMax=sqrt(n)
    diviseurs=[]  #liste destinée à recevoir les diviseurs de n
    while diviseur<=diviseurMax and n>1 :
        while n%diviseur==0:
            diviseurs.append(diviseur) #enregistrement d'un facteur premier
            n=n//diviseur
        diviseur+=1
    if len(diviseurs)==0 : return "est premier"
    #préparation de l'affichage pour un nombre multiple
    produit="= "
    multiplicite=1
    facteur=diviseurs[0]
    for i in range(1,len(diviseurs)) :
        if diviseurs[i]==facteur :
            multiplicite+=1
            continue
        if multiplicite==1 : produit+=str(facteur)+"\u00D7"
        else : produit+=str(facteur)+"^"+str(multiplicite)+"\u00D7"
        multiplicite=1
        facteur=diviseurs[i]
    if multiplicite==1 : produit+=str(facteur)
    else : produit+=str(facteur)+"^"+str(multiplicite)
    return produit

n=1001
print(n,premier3bis(n))
n=96
print(n,premier3bis(n))
n=101
print(n,premier3bis(n))
n=38500
print(n,premier3bis(n))
```

```
1001 = 7×11×13
96 = 2^5×3
101 est premier
38500 = 2^2×5^3×7×11
```

3. Pour soigner l'affichage, la fonction `premier3` reprend `premier2` en ajoutant un dénombrement de la multiplicité de chacun des facteurs premiers. La chaîne de caractères `produit` se construit , facteur après facteur, en concaténant les termes `str(facteur)+"^"+str(multiplicite)+"\u00D7"` (je rappelle au passage que le symbole correct de la multiplication s'obtient par son code unicode "\u00D7"). Le dernier symbole multiplicatif ainsi créé est retiré (on aurait pu aussi éviter de l'écrire en testant à chaque fois si on traitait le dernier facteur).
   Avec la fonction `premier3bis`, j'évite d'afficher la multiplicité lorsque celle-ci est égale à 1 en la testant (c'était ce qui était demandé de façon implicite dans la question 3) ; dans ce cas, il suffit de concaténer `str(facteur)+"\u00D7"` au lieu de `str(facteur)+"^"+str(multiplicite)+"\u00D7"`.

Correction de l'exercice 2.13 (Conversion)

1. La fonction `convert(n,b,i=10)` effectue la conversion de `n` dans une base `b` $\leqslant 10$. Le 3^{e} argument est optionnel : en lui donnant une valeur par défaut (en écrivant `i=10`), on peut omettre de le préciser dans l'appel de fonction.
2. En fait, ma fonction `convert(n,b,i=10)` réalise la conversion vers toute base $2 \leqslant$ `b` $\leqslant 36$. Au moment d'écrire le chiffre correspondant au reste de la division euclidienne par `b`, il suffit de tester si ce reste est un chiffre (auquel cas on le convertit en chaîne de caractères avec `str(n%)`) ou s'il doit être écrit avec une lettre (auquel cas on écrit ce caractère avec `chr(87+n%b)`).
3. La dernière amélioration à apporter à cette fonction `convert(n,b,i=10)` est le traitement du nombre `n` lorsque la base initiale `i` est différente de 10. On utilise simplement la fonction `int(n,i)` qui réalise cela. Cette fonction nécessite de convertir `n` sous la forme d'une chaîne de caractères (même si `n` s'écrit avec des chiffres et même si la base est inférieure à 10).

```
def convert(n,b,i=10):
    """ Convertit n (donné en base i) en base b<=36 ; par défaut i=10
        la base optionnelle i doit aussi être comprise entre 2 et 36 """
    result=""
    if i!=10 : n=int(str(n),i)        #conversion si nécessaire de n en base 10
    while n>0:
        if n%b<10:
            result=str(n%b)+result    #concaténation par la gauche du reste<10 (chiffre)
        else :
            result=chr(87+n%b)+result #concaténation par la gauche du reste>=10 (lettre)
        n=n//b
    return result

n,b=2020,7                          # 2 arguments (i=10)
print("{} en base {} s'écrit {}".format(n,b,convert(n,b)))
n,b=2020,16
print("{} en base {} s'écrit {}".format(n,b,convert(n,b)))
n,b,i='7e4',10,16                   # 3 arguments
print("{} en base {} s'écrit {}".format(n,b,convert(n,b,i)))
n,b,i='7e4',36,16
print("{} en base {} s'écrit {}".format(n,b,convert(n,b,i)))
```

```
2020 en base 7 s'écrit 5614
2020 en base 16 s'écrit 7e4
7e4 en base 10 s'écrit 2020
7e4 en base 36 s'écrit 1k4
```

Correction de l'exercice 2.14 (Pyramide 3D)

Après avoir entré `from turtle import *` :

- `goto(50,150)` : la tortue se déplace à la position repérée par les coordonnées (50,150), l'origine du repère étant placé par défaut au centre de la fenêtre graphique.
- `forward(100)` : la tortue avance dans la direction où elle est (par défaut vers la droite, l'angle avec le haut étant de +90°).
- `left(90)` : la tortue modifie sa direction en pivotant d'un angle de +90° vers la gauche (dans le sens inverse des aiguilles d'une montre).
- `down()` : le stylo est abaissé sur la feuille, en position d'écriture. Tout déplacement de la tortue (elle matérialise la position du stylo) sera alors tracé sur la feuille (la fenêtre graphique). L'instruction `up()` permet de relever le stylo (pour effectuer un déplacement dont on ne veut pas laisser de trace)
- `dot(15)` : un point de taille 15 pixels est tracé à la position courante de la tortue.
- `ht()` : abréviation de l'anglais *hide turtle* qui signifie cacher la tortue. Cette instruction cache donc la tortue (on ne la voit plus, mais on voit les tracés qu'elle a effectué).

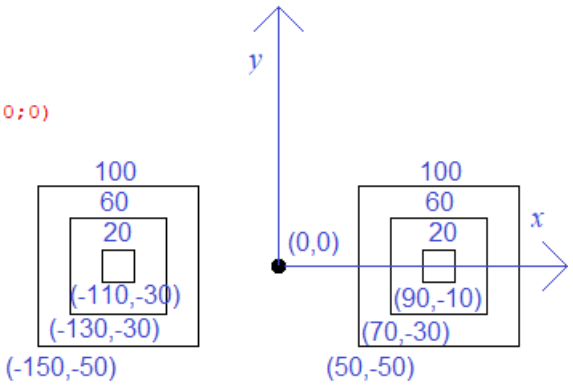
1. Pour dessiner un carré de `100` pixels de côté dont le sommet inférieur gauche a pour coordonnées `(50,-50)`, il faut écrire les instructions suivantes :

   - `up()`, `goto(50,-50)` et `down()` pour se positionner correctement à l'endroit indiqué
   - ensuite on écrit une boucle `for i in range(4):` contenant `forward(100)` et `left(90)`. Ceci afin d'avoir le sommet inférieur gauche aux coordonnées indiquées.

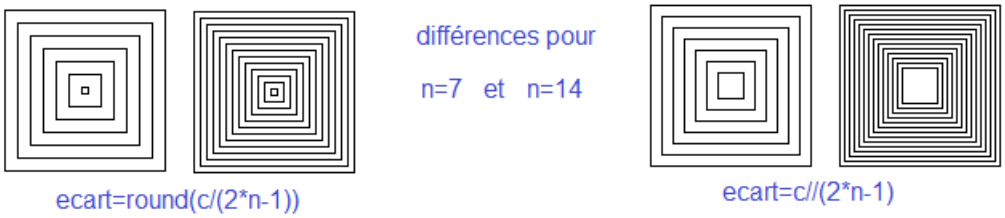
2. La fonction `carre(x,y,c)` dessine un carré de `c` pixels de côté dont le sommet inférieur gauche a pour coordonnées `(x,y)`. Pour tracer le carré demandé dans la question précédente, il suffit d'écrire l'instruction `carre(50,-50,100)`.

3. Le programme qui suit dessine trois carrés imbriqués (le 1er de 100 px de côté, le 2e de 60 px, le 3e de 20 px) à gauche et à droite du point de coordonnées `(0,0)`, comme sur la figure de l'énoncé. Sur l'illustration ci-dessous, j'ai tracé les axes de la fenêtre graphique et écrit les coordonnées des sommets inférieurs gauche des six carrés tracés ainsi que les côtés de ces carrés.

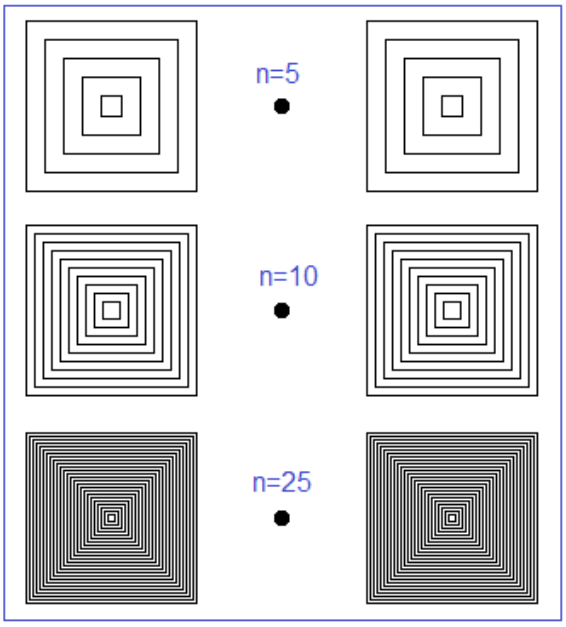
```
""" Dessine avec turtle une pyramide en 3D
    dessine 3 carrés imbriqués, de part et d'autre d'un point (question 3)"""

from turtle import *

def carre(x,y,c):
    """trace un carré de coté c
    dont le sommet inférieur gauche à les coordonnées (x;y) (question 2)"""
    up()     #lève le stylo
    goto(x,y)#se place au sommet de coordonnées (x;y)
    down()   #abaisse le stylo
    for i in range(4):
        forward(c) #avance de c pixels
        left(90)   #tourne de 90°

dot(10)            #place un point de coordonnées (0;0)
x,y,c=50,-50,100
for i in range(3): #trace les 3 carrés de droite
    carre(x,y,c)
    x+=20
    y+=20
    c-=40
x,y,c=-150,-50,100 #trace les 3 carrés de gauche
for i in range(3):
    carre(x,y,c)
    x+=20
    y+=20
    c-=40
ht()               #cache la tortue (hide turtle)
```



4. Le programme pour créer `n` carrés imbriqués est dérivé du précédent. Il suffit de calculer l'écart à ajouter aux coordonnées `x` et `y` en fonction de `n` et du côté du carré initial `c`. L'illustration montre le résultat pour `n=5`, `n=10` et `n=25`. On peut remarquer que la formule de l'écart donne un résultat satisfaisant pour la plupart des valeurs, mais comme il s'agit d'une valeur tronquée à l'entier inférieur, cela arrive que ce ne soit pas très bon (il reste un carré central dans lequel on pourrait tracer un petit carré supplémentaire). Pour remédier à cela, il faudrait remplacer la formule de l'énoncé par `ecart=round(c/(2*n-1))`. Observer les différences entre ces deux formules pour `n=7` et `n=14`.

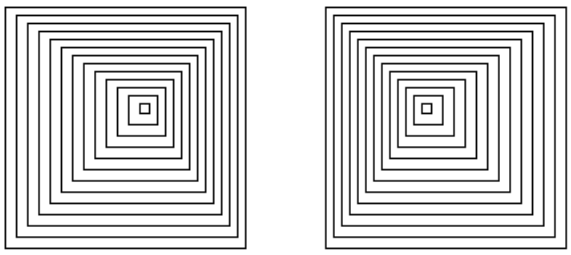


```
""" Dessine avec turtle une pyramide en 3D
    dessine n carrés imbriqués, de part et d'autre d'un point (question 4)"""

from turtle import *

def carre(x,y,c):
    up()
    goto(x,y)
    down()
    for i in range(4):
        forward(c)
        left(90)

ht()
dot(10)
speed(0) # pour accélérer les tracés
c,n=100,7
# écart régulier entre les n carrés
ecart=c//(2*n-1)    # ecart=round(c/(2*n-1))
# dessin du motif de droite
x,y=50,-50
for i in range(n):
    carre(x+i*ecart,y+i*ecart,c-2*i*ecart)
# dessin du motif de gauche
x,y=-150,-50
for i in range(n):
    carre(x+i*ecart,y+i*ecart,c-2*i*ecart)
```



5. Voici une version modifiée donnant des pyramides déformées par la perspective et permettant de restituer l'impression du relief (il faut regarder la figure en louchant un peu).

```
ht()
speed(0)
c,n,d=150,13,1 # d: parallaxe
# écart régulier entre les n carrés
ecart=c//(2*(n-1)+1)
# dessin du motif de droite
x,y=50,-50
for i in range(n):
    carre(x+i*(ecart-d),y+i*(ecart+d),c-2*i*ecart)
# dessin du motif de gauche
x,y=-150,-50
for i in range(n):
    carre(x+i*(ecart+d),y+i*(ecart+d),c-2*i*ecart)
```

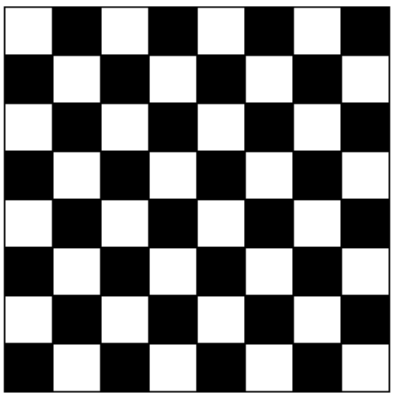


Correction de l'exercice 2.15 (Damier)

1. La fonction `carrePlein(x,y,c)` dessine un carré plein (elle reprend la fonction `carre(x,y,c)` de l'exercice précédent en y ajoutant les instructions `begin_fill()` au début du tracé et `end_fill()` à la fin).
2. Le programme ci-dessous dessine un damier de `2*n` cases, alternativement pleines et vides (il ne dessine que les pleines). Pour tracer une bordure à l'ensemble de ces carreaux blancs et noirs, il faut utiliser la fonction `carreVide(x,y,c)` (la fonction `carre(x,y,c)` rebaptisée).

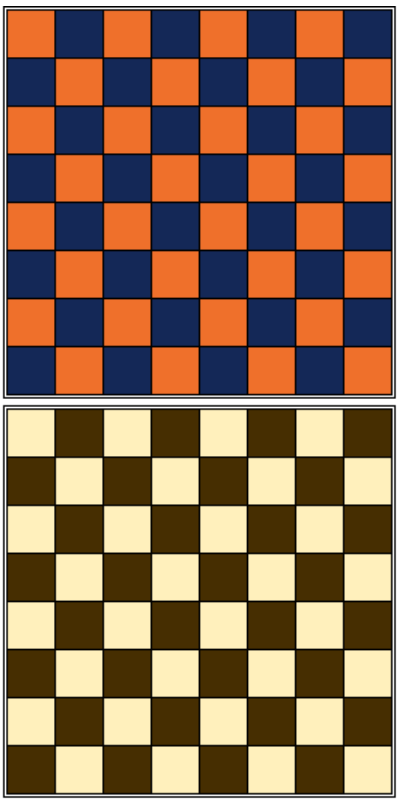
```
from turtle import *
def carrePlein(x,y,c):
    up()
    goto(x,y)
    down()
    begin_fill()
    for i in range(4):
        forward(c)
        left(90)
    end_fill()

def carreVide(x,y,c):
    up()
    goto(x,y)
    down()
    for i in range(4):
        forward(c)
        left(90)
```

```
c,n=30,4
#trace un damier de 2*n carrés de c de large sur chaque côté
ht()
speed(0)
x=-c*(n+1)
for i in range(2*n):
    x+=c
    y=-c*n
    for j in range(2*n):
        if (i+j)%2==0:
            carrePlein(x,y,c)
        y+=c
carreVide(-c*n,-c*n,2*c*n)
```



3. Le dessin est amélioré en utilisant deux couleurs. Pour cela, on peut tracer tous les carrés (c'est ce que fait mon programme 1) mais on peut préférer cette solution alternative, plus simple : tracer le carré de bordure en le remplissant de la couleur 2 et, à l'intérieur, tracer les petits carreaux de la couleur 1 (c'est ce que fait mon programme 2).
Le rendu des deux programmes est, bien sûr, identique.

```
c,n=30,4
ht()
speed(0)
x=-c*(n+1)

#programme 1 : petits carrés des deux couleurs
for i in range(2*n):
    x+=c
    y=-c*n
    for j in range(2*n):
        if (i+j)%2==0:
            color('black','#142857')
            carrePlein(x,y,c)
        else:
            color('black','#ef702b')
            carrePlein(x,y,c)
        y+=c
carreVide(-c*n-2,-c*n-2,2*c*n+4)

#programme 2 : petits carrés d'une seule couleur
color('black','#fff0bc')
carrePlein(-c*n,-c*n,2*c*n) # 1ère bordure (pleine)
for i in range(2*n):
    x+=c
    y=-c*n
    color('black','#462e01')
    for j in range(2*n):
        if (i+j)%2==0: carrePlein(x,y,c)
        y+=c
carreVide(-c*n-2,-c*n-2,2*c*n+4)# 2ème bordure(vide)
```



Correction de l'exercice 2.16 (Triangle des couleurs)

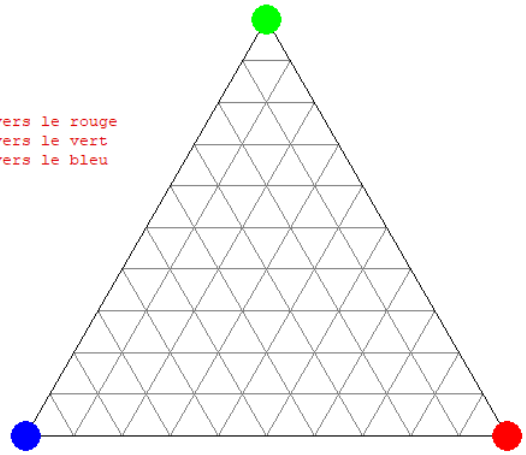
On veut visionner les couleurs RGB dans un triangle équilatéral de 400 pixels de côté, chaque sommet représentant une couleur primaire pure : à gauche Bleu, à droite Rouge, en haut Vert.

1. La façon de tracer ce quadrillage triangulaire peut varier beaucoup. On peut chercher les coordonnées des points d'intersection des segments avec les bords du triangle et tracer les segments avec des instructions de positionnement (`goto(x,y)`).
J'ai choisi une autre méthode qui fait faire des zig-zags à la tortue. Les seules instructions utilisées sont des rotations (`left(angle)` ou `right(angle)`) et des translations (`forward(long)`).
Bien sûr, dans tous les cas, il faut soigneusement analyser la figure pour sélectionner la méthode qui parait la plus simple à mettre en œuvre et effectuer les calculs (de coordonnées ou d'angles et de longueurs) nécessaires.

```
from turtle import *
couleur=['#0000ff','#ff0000','#00ff00']#bleu,rouge,vert

def triangle(n,c):
    for i in range(3):#le tour
        forward(c)
        left(120)
    for i in range(3):#les points
        color(couleur[i])
        dot(25)
        up()
        forward(c)
        left(120)
        down()
```

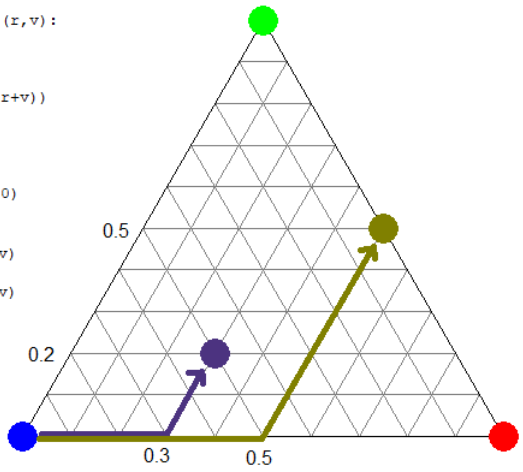
```
def segments(n,c):
    up()
    forward(ecart)
    left(60)
    for i in range(1,n):#le tour
        down()
        forward(c-i*ecart)
        left((-1)**i*120)
        up()
        forward(ecart)
        left((-1)**i*60)
    right(120)
```

```
ht()
speed(0)
c,n=400,10
ecart=c//n
up()
goto(-200,-100)
color('gray')
segments(n,c)#segments vers le rouge
segments(n,c)#segments vers le vert
segments(n,c)#segments vers le bleu
down()
color('black')
triangle(n,c)#le tour
up()
```



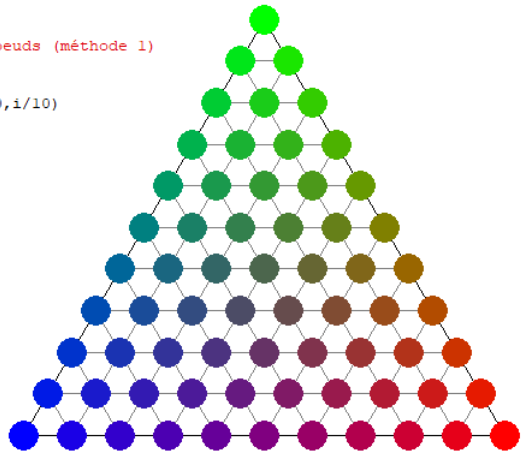
2. La fonction demandée dans cette question place un point de la couleur `r, v, 1-r-v` (`r` et `v` sont les fréquences de rouge et de vert) par la méthode suivante : en partant du point bleu (stylo levé), parcourir les `r%` du segment qui va jusqu'au point rouge, puis tourner de 60° et parcourir les `v%` du segment qui va vers le point vert (voir l'illustration où j'ai ajouté des indications montrant ce chemin de la tortue), abaisser alors le stylo et placer le point.

```
def pointCouleur1(r,v):
    forward(r*c)
    left(60)
    forward(v*c)
    color(r,v,1-(r+v))
    down()
    dot(25)
    up()
    left(-60)
    goto(-200,-100)

r,v=0.5,0.5
pointCouleur1(r,v)
r,v=0.3,0.2
pointCouleur1(r,v)
```



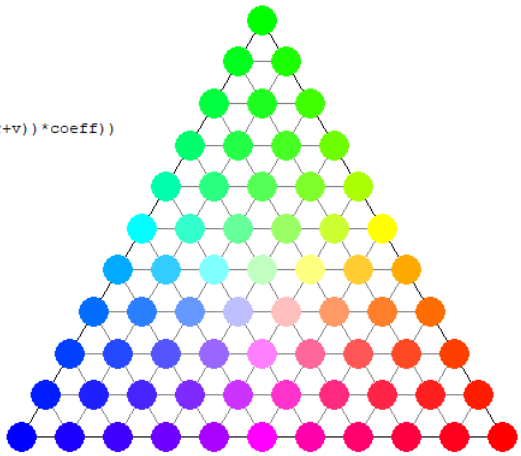
3. Il faut maintenant parcourir chaque nœud du triangle en y plaçant un point de la bonne couleur. Cela ne pose pas, à ce stade, de problèmes particuliers puisque l'on dispose déjà de toutes les fonctions nécessaires.

```
# coloriage de tous les noeuds (méthode 1)
for i in range(11):
    for j in range(11-i):
        pointCouleur1(j/10,i/10)
```



4. Pour donner aux couleurs leur brillance maximum, il faut calculer le coefficient de proportionnalité à appliquer à chaque couleur ; celui-ci demande de déterminer le maximum entre les trois fréquences et convertit toutes les couleurs en un nombre entier allant de 0 à 255 (la méthode RGB classique de représentation des couleurs) de façon à ce que celle qui a la fréquence maximum passe à 255.

```
def pointCouleur2(r,v):
    forward(r*c)
    left(60)
    forward(v*c)
    maxfcolor=max(r,v,1-(r+v))
    coeff=255/maxfcolor
    color(int(r*coeff),int(v*coeff),int((1-(r+v))*coeff))
    down()
    dot(25)
    up()
    left(-60)
    goto(-200,-100)


# coloriage de tous les noeuds (méthode 2)
colormode(255)
for i in range(11):
    for j in range(11-i):
        pointCouleur2(j/10,i/10)
```



Correction de l'exercice 2.17 (Opérations cryptées)

1. Prouvons que, chaque lettre différente représentant un chiffre différent, l'addition SEND+MORE=MONEY n'admet qu'une seule solution.
   Ce problème a 8 variables : S, E, N, D, M, O, R, Y.
   Le programme ci-dessous essaie toutes les combinaisons possibles et affiche toutes les solutions.
   Ce que j'avais oublié dans l'énoncé : pour que ce soit un bon problème il faut que le chiffre M ne soit pas nul. Sans cette contrainte, il y a 25 solutions différentes et ce n'est donc pas un bon problème.

```
C'est un bon problème qui n'a qu'une solution : 9567+1085=10652
```

```
solutions=[]
for S in range(10) :
    for E in range(10)  :
        if E==S : continue
        for N in range(10)  :
            if N==E or N==S : continue
            for D in range(10)  :
                if D==N or D==E or D==S : continue
                for M in range(1,10)  : # pour empêcher un chiffre M nul (avec range(10) il y a 25 solutions)
                    if M==D or M==N or M==E or M==S : continue
                    for O in range(10)  :
                        if O==M or O==D or O==N or O==E or O==S : continue
                        for R in range(10)  :
                            if R==O or R==M or R==D or R==N or R==E or R==S : continue
                            for Y in range(10)  :
                                if Y==R or Y==O or Y==M or Y==D or Y==N or Y==E or Y==S : continue
                                if (S+M-O)*10**3+(E+O-N)*10**2+(N+R-E)*10+(D+E-Y)-M*10**4==0 :
                                    solutions.append(str(S*10**3+E*10**2+N*10+D)+'+'+str(M*10**3+O*10**2+R*10+E)+\
                                                     '='+str(M*10**4+O*10**3+N*10**2+E*10+Y))
if len(solutions)==0 :
    print("Ce problème n'a pas de solution.")
elif len(solutions)==1 :
    print("C'est un bon problème qui n'a qu'une solution : "+solutions[0])
else :
    print("Il y a {} solutions : ".format(len(solutions)))
    for i in range(len(solutions)):
        print("Solution n°{} : {}".format(i+1,solutions[i]))
```

2. En modifiant très légèrement le programme précédent, on résout le problème à 9 variables (C, I, N, Q, S, X, T, R, E) : « CINQ×SIX=TRENTE ». Comme je n'ai pas précisé ici que je voulais des premiers chiffres non nuls, je trouve 2 solutions. Mais avec cette contrainte supplémentaire, il n'en reste plus qu'une $5409 \times 142 = 768078$.

```
Il y a 2 solutions :
Solution n°1 : 2567×54=138618
Solution n°2 : 5409×142=768078
```

```
for C in range(10) :
    for I in range(10)  :
        if I==C : continue
        for N in range(10)  :
            if N==I or N==C : continue
            for Q in range(10)  :
                if Q==N or Q==I or Q==C : continue
                for S in range(10)  :
                    if S==Q or S==N or S==I or S==C : continue
                    for X in range(10)  :
                        if X==S or X==Q or X==N or X==I or X==C : continue
                        for T in range(10)  :
                            if T==X or T==S or T==Q or T==N or T==I or T==C : continue
                            for R in range(10)  :
                                if R==T or R==X or R==S or R==Q or R==N or R==I or R==C : continue
                                for E in range(10)  :
                                    if E==R or E==T or E==X or E==S or E==Q or E==N or E==I or E==C : continue
                                    num1=C*10**3+I*10**2+N*10+Q
                                    num2=S*10**2+I*10+X
                                    num3=T*10**5+R*10**4+E*10**3+N*10**2+T*10+E
                                    if num1*num2==num3 :
                                        solutions.append(str(num1)+'\u00D7'+str(num2)+'='+str(num3))
```

3. Mon problème à 9 variables (P, H, I, L, E, M, O, U, T) : « PHILIPPE+MOUTOU=PILEPOIL » a 4 solutions différentes. C'est un assez bon problème néanmoins (dans le sens qu'il n'a pas beaucoup de solutions).
   Si on veut néanmoins en faire un bon problème, il faut ajouter une condition supplémentaire. Si je ne souhaite aucun chiffre nul, il n'y a plus qu'une seule solution : $17828119+463563=18291682$.
   Mais on peut sans doute trouver d'autres contraintes spécifiques qui privilégie une autre solution.

```
for P in range(10) :
    for H in range(10)  :
        if H==P : continue
        for I in range(10)  :
            if I==H or I==P : continue
            for L in range(10)  :
                if L==I or L==H or L==P : continue
                for E in range(10)  :
                    if E==L or E==I or E==H or E==P : continue
                    for M in range(10)  :
                        if M==E or M==L or M==I or M==H or M==P : continue
                        for O in range(10)  :
                            if O==M or O==E or O==L or O==I or O==H or O==P : continue
                            for U in range(10)  :
                                if U==O or U==M or U==E or U==L or U==I or U==H or U==P : continue
                                for T in range(10)  :
                                    if T==U or T==O or T==M or T==E or T==L or T==I or T==H or T==P : continue
                                    num1=P*10**7+H*10**6+I*10**5+L*10**4+I*10**3+P*10**2+P*10+E
                                    num2=M*10**5+O*10**4+U*10**3+T*10**2+O*10+U
                                    num3=P*10**7+I*10**6+L*10**5+E*10**4+P*10**3+O*10**2+I*10+L
                                    if num1+num2==num3 :
                                        solutions.append(str(num1)+'+'+str(num2)+'='+str(num3))
```

```
Il y a 4 solutions :
Solution n°1 : 17828119+463563=18291682
Solution n°2 : 28909227+163463=29072690
Solution n°3 : 38909336+154254=39063590
Solution n°4 : 86737882+591091=87328973
```

4. Le problème à 6 variables (N, E, U, F, O, Z) : « NEUF+UN+UN=ONZE » est un bon problème car il n'a qu'une solution.
   Il va sans dire qu'il n'est pas simple d'inventer de tels problèmes, surtout lorsqu'on ne dispose pas du programme qui explore toutes les possibilités. Sur internet on trouve de nombreux problèmes de ce type (il y a toujours un jeu avec les mots), en trouverez-vous de nouveaux ?

```
for N in range(10) :
    for E in range(10)  :
        if E==N : continue
        for U in range(10)  :
            if U==E or U==N : continue
            for F in range(10)  :
                if F==U or F==E or F==N : continue
                for O in range(10)  :
                    if O==F or O==U or O==E or O==N : continue
                    for Z in range(10)  :
                        if Z==O or Z==F or Z==U or Z==E or Z==N : continue
                        num1=N*10**3+E*10**2+U*10+F
                        num2=U*10+N
                        num3=O*10**3+N*10**2+Z*10+E
                        if num1+num2*2==num3 :
                            solutions.append(str(num1)+'+'+str(num2)+'+'+str(num2)+'='+str(num3))
```

```
C'est un bon problème qui n'a qu'une solution : 1987+81+81=2149
```

Correction de l'exercice 2.18 (Exploration numérique)

1. La fonction `successeur(n)` détermine et retourne le successeur d'un nombre `n` de trois chiffres `str(n)="abc"`. J'ai choisi de rester avec des valeurs numériques dans `successeur1` et j'ai fait selon l'indication du texte (en passant par des chaînes de caractères) pour `successeur2`.
   La difficulté de ces fonctions est de traiter les cas où le nombre `n` n'a pas vraiment 3 chiffres, comme par exemple 45 qui doit être traité comme 045. Dans ce cas, on doit ajouter les '0' qui manquent (cette remarque ne concerne que `successeur2`).

```
def successeur1(n):                         # recherche du successeur d'un nombre abc (question 1)
    if int(n)!=n or n>999 or n<0 : return -1 # le successeur n'est défini que pour les entiers positifs de 3 chiffres
    c,b,a=n%10,(n//10)%10,n//100            # obtention des chiffres a, b et c
    e=(a+c)%10                              # calcul du chiffre e
    return b*100+c*10+e                     # calcul et retour de la valeur de bce

def successeur2(n):                                 # recherche du successeur d'un nombre abc (question 1)
    n=str(n)                                        # conversion de n en chaine de caractères
    if len(n)>3 or n[0]=="-" or "." in n : return -1 # le successeur n'est défini que pour les entiers positifs de 3 chiffres
    while len(n)<3 : n='0'+n                        # ajout de "0" devant si n<100
    return int(n[1:]+str(int(n[-1])+int(n[0]))[-1]) # calcul et retour de la valeur de bce (n[1:] donne bc, le reste calcule e)

#tests
print(successeur1(157))      print(successeur2(157))
print(successeur1(15))       print(successeur2(15))
print(successeur1(757))      print(successeur2(757))
print(successeur1(1270))     print(successeur2(1270))
```

```
578
155
574
-1
```

```
578
155
574
-1
```

2. La fonction `successeurs(n)` prend un nombre de trois chiffres `abc` comme argument et affiche la succession des nombres obtenus avec un des fonctions `successeur(n)`, la longueur de la succession, le minimum et le maximum qui ont été atteint.

On voit que la succession de 157 contient 296 nombres compris entre 1 et 997.

```
def successeurs(n):
    """ afficher la liste des successeurs du nombre de départ
        la longueur de la suite, les valeurs minimum et maximum (question 2)"""
    long,maxi,mini=0,0,999
    initial=n
    texte="successeurs de "+str(n)+" : "
    if n==-1 : print("le nombre entré n'est pas un nombre entier positif de 3 chiffres au plus")
    while True:
        n=successeur1(n)
        texte+=str(n)+" - "
        long+=1
        if n>maxi : maxi=n
        if n<mini : mini=n
        if n==initial: break
    print(texte[:-3])
    print("longueur de la suite=",long)
    print("minimum de la suite=",mini)
    print("maximum de la suite=",maxi,'\n')
```

```
successeurs(157)
successeurs de 157 : 578 - 783 - 830 - 308 - 81 - 811 - 119 - 190 - 901 - 10 - 100 - 1 - 11 - 111 - 112 - 123 - 234 -
346 - 469 - 693 - 939 - 398 - 981 - 810 - 108 - 89 - 899 - 997 - 976 - 765 - 652 - 528 - 283 - 835 - 353 - 536 - 361 -
614 - 140 - 401 - 15 - 155 - 556 - 561 - 616 - 162 - 623 - 239 - 391 - 914 - 143 - 434 - 348 - 481 - 815 - 153 - 534 -
349 - 492 - 926 - 265 - 657 - 573 - 738 - 385 - 858 - 586 - 861 - 619 - 195 - 956 - 565 - 650 - 506 - 61 - 611 - 117 -
178 - 789 - 896 - 964 - 643 - 439 - 393 - 936 - 365 - 658 - 584 - 849 - 497 - 971 - 710 - 107 - 78 - 788 - 885 - 853 -
531 - 316 - 169 - 690 - 906 - 65 - 655 - 551 - 516 - 161 - 612 - 128 - 289 - 891 - 919 - 198 - 989 - 898 - 986 - 865 -
653 - 539 - 394 - 947 - 476 - 760 - 607 - 73 - 733 - 330 - 303 - 36 - 366 - 669 - 695 - 951 - 510 - 105 - 56 - 566 -
661 - 617 - 173 - 734 - 341 - 414 - 148 - 489 - 893 - 931 - 310 - 103 - 34 - 344 - 447 - 471 - 715 - 152 - 523 - 238 -
380 - 803 - 31 - 311 - 114 - 145 - 456 - 560 - 605 - 51 - 511 - 116 - 167 - 678 - 784 - 841 - 419 - 193 - 934 - 343 -
436 - 360 - 603 - 39 - 399 - 992 - 921 - 210 - 102 - 23 - 233 - 335 - 358 - 581 - 816 - 164 - 645 - 451 - 515 - 150 -
501 - 16 - 166 - 667 - 673 - 739 - 396 - 969 - 698 - 984 - 843 - 431 - 315 - 158 - 589 - 894 - 942 - 421 - 215 - 157
longueur de la suite= 217
minimum de la suite= 1
maximum de la suite= 997
```

3. L'ensemble des nombres à trois chiffres contient 1000 nombres, de 000 à 999. Chacun de ces nombres appartient à un sous-ensemble : le sous-ensemble de 157 contient 1 et 997 mais ne contient pas 2 par exemple qui appartient à un autre sous-ensemble. J'ai écrit une autre fonction, la fonction `sousensemble(n)` qui ordonne les éléments de la liste des successeurs d'un nombre `n`. Ainsi je trouve que chacun des 10 chiffres conduit à un sous-ensemble distinct.
   - 4 sous-ensembles contiennent 217 nombres : ceux qui contiennent les chiffres 1, 3, 7 et 9.
   - 4 sous-ensembles contiennent 31 nombres : ceux qui contiennent les chiffres 2, 4, 6 et 8.
   - 1 sous-ensemble contient 7 nombres (les nombres 5, 50, 55, 500, 505, 550 et 555) : celui qui contient le chiffre 5.
   - 1 sous-ensemble contient 1 seul nombre : celui qui contient le chiffre 0.

```
def sousensemble(n):
    """ afficher la liste des successeurs du nombre de départ
        ordonnée afin d'identifier des sous-ensembles distincts (question 3)"""
    long=0
    initial=n
    successeurs=[]
    while True:
        n=successeur1(n)
        successeurs.append(n)
        long+=1
        if n==initial: break
    print(sorted(successeurs))
    print("longueur de la suite=",long)
sousensemble(1)
```

```
[1, 10, 11, 15, 16, 23, 31, 34, 36, 39, 51, 56, 61, 65, 73, 78, 81, 89, 100, 102, 103, 105, 107, 108, 111, 112
, 114, 116, 117, 119, 123, 128, 140, 143, 145, 148, 150, 152, 153, 155, 157, 158, 161, 162, 164, 166, 167, 169
, 173, 178, 190, 193, 195, 198, 210, 215, 233, 234, 238, 239, 265, 283, 289, 303, 308, 310, 311, 315, 316, 330
, 335, 341, 343, 344, 346, 348, 349, 353, 358, 360, 361, 365, 366, 380, 385, 391, 393, 394, 396, 398, 399, 401
, 414, 419, 421, 431, 434, 436, 439, 447, 451, 456, 469, 471, 476, 481, 489, 492, 497, 501, 506, 510, 511, 515
, 516, 523, 528, 531, 534, 536, 539, 551, 556, 560, 561, 565, 566, 573, 578, 581, 584, 586, 589, 603, 605, 607
, 611, 612, 614, 616, 617, 619, 623, 643, 645, 650, 652, 653, 655, 657, 658, 661, 667, 669, 673, 678, 690, 693
, 695, 698, 710, 715, 733, 734, 738, 739, 760, 765, 783, 784, 788, 789, 803, 810, 811, 815, 816, 830, 835, 841
, 843, 849, 853, 858, 861, 865, 885, 891, 893, 894, 896, 898, 899, 901, 906, 914, 919, 921, 926, 931, 934, 936
, 939, 942, 947, 951, 956, 964, 969, 971, 976, 981, 984, 986, 989, 992, 997]
```

Vérification : $4 \times 217 + 4 \times 31 + 7 + 1 = 868 + 124 + 7 + 1 = 1000$.
Comme on a trouvé les 1000 nombres attendus, on peut conclure que tous les nombres ont été rangés dans l'un ou l'autre de ces 10 sous-ensembles. Un prolongement amusant est d'effectuer la même étude pour les nombres de 2 chiffres ou pour les nombres de 4 chiffres...

Correction de l'exercice 2.19 (Décomposition des entiers)

1. Pour l'exploration systématique, la fonction `decomposition1(N)` cherche une décomposition en somme d'entiers consécutifs pour tous les nombres entiers inférieurs ou égaux à `N`. Le programme affiche pour chaque nombre, la ou les décompositions qui conviennent. Lorsque je lance

`decomposition1(100)`, je m'aperçois que tous les nombres ont une ou des décompositions, sauf 4, 8, 16, 32 et 64 qui n'en ont pas. J'émets alors la conjecture que seules, les puissances de 2 ne peuvent pas se décomposer comme somme d'entiers consécutifs.

```
def decomposition(N):
    for n in range(3,N+1) :
        solutions=[]
        for premier_terme in range(1,n//2+1) :
            somme_string=str(premier_terme)+"+"
            somme,k=premier_terme,premier_terme
            while somme<n :
                k+=1
                somme+=k
                somme_string+=str(k)
                if somme<n : somme_string+="+"
            if somme==n :
                solutions.append(somme_string)
        print("{} = {}".format(n,solutions))

decomposition(16)
```

```
3 = ['1+2']
4 = []
5 = ['2+3']
6 = ['1+2+3']
7 = ['3+4']
8 = []
9 = ['2+3+4', '4+5']
10 = ['1+2+3+4']
11 = ['5+6']
12 = ['3+4+5']
13 = ['6+7']
14 = ['2+3+4+5']
15 = ['1+2+3+4+5', '4+5+6', '7+8']
16 = []
```

2. Pour savoir quel est le 1er nombre à avoir `k=1,2,3,...` décomposition(s), je n'enregistre dans la version `decomposition2(N)` que le nombre de décompositions pour chaque entier. Je peux ainsi afficher la liste des nombres ayant 1, 2, 3, etc. décompositions. Dans la version `decomposition3(N)`, je n'affiche que la tête de liste, c'est-à-dire le premier nombre ayant 1, 2, 3, etc. décomposition. Pour limiter l'affichage, j'utilise une variable `maxi` qui indique au programme jusqu'à combien de décompositions on peut avoir.

```
def decomposition3(N):
    tableau=[]
    maxi=0
    for n in range(1,N+1) :
        solutions=[]
        for premier_terme in range(1,n//2+1) :
            somme_string=str(premier_terme)+"+"
            somme,k=premier_terme,premier_terme
            while somme<n :
                k+=1
                somme+=k
                somme_string+=str(k)
                if somme<n : somme_string+="+"
            if somme==n :
                 solutions.append(somme_string)
        tableau.append([n,len(solutions)])
        if len(solutions)>maxi : maxi=len(solutions)
    for i in range(maxi+1) :
        string="Premier nombre ayant {} décompositions : ".format(i)
        for j in range(N) :
            if tableau[j][1]==i :
                string+=" "+str(tableau[j][0])
                print(string)
                break
decomposition3(10000)
```

```
Premier nombre ayant 0 décompositions :  1
Premier nombre ayant 1 décompositions :  3
Premier nombre ayant 2 décompositions :  9
Premier nombre ayant 3 décompositions :  15
Premier nombre ayant 4 décompositions :  81
Premier nombre ayant 5 décompositions :  45
Premier nombre ayant 6 décompositions :  729
Premier nombre ayant 7 décompositions :  105
Premier nombre ayant 8 décompositions :  225
Premier nombre ayant 9 décompositions :  405
Premier nombre ayant 11 décompositions :  315
Premier nombre ayant 13 décompositions :  3645
Premier nombre ayant 14 décompositions :  2025
Premier nombre ayant 15 décompositions :  945
Premier nombre ayant 17 décompositions :  1575
Premier nombre ayant 19 décompositions :  2835
Premier nombre ayant 23 décompositions :  3465
```

Listes

Correction de l'exercice 2.20 (Cartes)

a) Le programme suivant renvoie une main de `n` cartes prises au hasard dans un jeu de 32 cartes.

```
from random import *

def cartes(n):
    coul=['Trèfle','Carreau','Coeur','Pique']
    vale=['7','8','9','10','Valet','Dame','Roi','As']
    main=[]
    while len(main)<n:
        carte="{} de {}".format(choice(vale),choice(coul))
        if carte not in main : main.append(carte)
    return main

print(" - ".join(cartes(4)))
print(" - ".join(cartes(3)))
```

```
8 de Pique - 7 de Carreau - 10 de Coeur - 9 de Carreau
Roi de Carreau - Valet de Trèfle - As de Coeur
```

Pour ne pas avoir de doublons (une carte ne peut être tirée qu'une seule fois d'un jeu si on ne remet pas les cartes tirées dans le jeu), je n'enregistre la carte tirée que si elle n'est pas déjà présente dans la main en cours de constitution. La main est donc une liste, vide au départ, qui se remplit progressivement et

est renvoyée lorsqu'elle a atteint le nombre d'éléments souhaité.
La syntaxe choisie `"{} de {}".format(choice(vale),choice(coul))` utilise la méthode `.format()` des chaînes de caractères. Mais on pourrait aussi écrire `choice(vale)+" de "+choice(coul)` qui donnerait le même résultat.
La chaîne `" - ".join(cartes(4))` permet d'écrire la main renvoyée (une liste) en utilisant un trait de séparation entre chaque carte, ce qui est un peu plus propre que d'afficher directement la liste.

b) Le programme suivant distribue un jeu de 32 cartes entre `n` joueurs. On calcule le nombre de cartes par joueur et on lance la fonction `cartes` avec ces deux arguments.

```
def cartes(n,p):
    coul=['Trèfle','Carreau','Coeur','Pique']
    vale=['7','8','9','10','Valet','Dame','Roi','As']
    jeu,mains=[],[]
    for c in coul:
        for v in vale:
            jeu.append("{} de {}".format(v,c))
    for i in range(p):
        main=[]
        while len(main)<n:
            main.append(jeu.pop(randrange(len(jeu))))
        mains.append(main)
    return mains
```

```
from random import *


nb_joueurs=4
nb_cartes=32//nb_joueurs
jeux=cartes(nb_cartes,nb_joueurs)
for i in range(nb_joueurs):
    print("Joueur {}:".format(i+1))
    print(" - ".join(jeux[i]))
```

```
Joueur 1:
Roi de Coeur - 8 de Carreau - Dame de Carreau - 7 de Carreau - 9 de Trèfle - 10 de Trèfle - 10 de Coeur - Roi de Pique
Joueur 2:
As de Trèfle - 9 de Pique - 9 de Coeur - 10 de Carreau - 7 de Trèfle - As de Coeur - 10 de Pique - As de Pique
Joueur 3:
Dame de Coeur - 7 de Coeur - 8 de Pique - Roi de Trèfle - Valet de Coeur - Dame de Trèfle - Roi de Carreau - 8 de Coeur
Joueur 4:
8 de Trèfle - 9 de Carreau - Valet de Trèfle - Valet de Pique - Valet de Carreau - Dame de Pique - 7 de Pique - As de Carreau
```

Dans un premier temps, cette fonction génère toutes les cartes du jeu. Puis, elle crée les mains une après l'autre, en retirant au fur-et-à-mesure du jeu les cartes tirées. La fonction utilisée est `pop(i)` qui retire du jeu en le renvoyant, l'élément de rang `i`. Une fois qu'une liste `main` est constituée, on l'ajoute dans la liste de listes `mains` (noter le pluriel) qui est renvoyée à la fin de la procédure.

Correction de l'exercice 2.21 (Calendrier)

a) Dans un premier temps, on se contente d'afficher les jours de la semaine pour un mois `m` et une année `a` fournis en arguments.

```
import calendar

def nbr_jours_mois1(m,a):
    longMois=[1,3,5,7,8,10,12]
    if m in longMois : return 31       #les mois ordinaires de 31 jours
    elif m!=2 : return 30              #les mois ordinaires de 30 jours
    elif a%4!=0 or (a%100==0 and a%400!=0):#teste l'année : bisextile (False) ou ordinaire (True)?
        return 28                          #février une année ordinaire
    return 29                              #février une année bisextile

def nbr_jours_mois2(m,a):              # fonction alternative utilisée ici
    long=[31,28,31,30,31,30,31,31,30,31,30,31]
    if m==2 and a%4==0 and (a%100!=0 or a%400==0): return 29
    return long[m-1]

mois,annee=11,2019  # modifier ici le mois et l'année
nom_jours=['Lundi','Mardi','Mercredi','Jeudi','Vendredi','Samedi','Dimanche']
nom_mois=['Janvier','Février','Mars','Avril','Mai','Juin','Juillet','Aout',\
          'Septembre','Octobre','Novembre','Décembre']
print(nom_mois[mois-1],annee)
for jour in range(1,nbr_jours_mois2(mois,annee)+1):
    jour_semaine=calendar.weekday(annee,mois,jour)
    print(nom_jours[jour_semaine],jour)
```

```
Novembre 2019
Vendredi 1      Samedi 16
Samedi 2        Dimanche 17
Dimanche 3      Lundi 18
Lundi 4         Mardi 19
Mardi 5         Mercredi 20
Mercredi 6      Jeudi 21
Jeudi 7         Vendredi 22
Vendredi 8      Samedi 23
Samedi 9        Dimanche 24
Dimanche 10     Lundi 25
Lundi 11        Mardi 26
Mardi 12        Mercredi 27
Mercredi 13     Jeudi 28
Jeudi 14        Vendredi 29
Vendredi 15     Samedi 30
```

La fonction `calendar.weekday(annee,mois,jour)` donne le jour de la semaine d'une date donnée avec le code Lundi :0, Mardi :1, etc. Ma liste `nom_jours` permet de rétablir le jour en français ; de même pour la liste `nom_mois` qui permet de rétablir le nom du mois (attention au décalage : janvier correspond à 0 et non à 1).
J'ai mis deux fonctions `nbr_jours_mois` mais je n'en utilise qu'une. Il n'y a jamais une seule façon d'obtenir un résultat. La 2^{e} me parait plus judicieuse car plus simple : il n'y a que le mois de février qui pose problème, d'où le test un peu sophistiqué qui examine si, dans ce cas, l'année est bissextile ou non.

b) Pour écrire une ligne par semaine, une colonne par jour, il faut formater l'affichage en console. J'ai réalisé cela en attribuant une largeur de 10 caractères à chaque jour (pour écrire mercredi et 1

espace de chaque côté) au moyen de la syntaxe `"{:10}".format(nom_jours[jour]),end=" "` qui est répétée après pour le jour du mois `"{:10}".format(jour),end=" "`. Il y a bien d'autres façons de procéder, par exemple en ajoutant des espaces tant qu'une certaine largeur n'a pas été atteinte.
Le mois ne commençant pas forcément par un Lundi, il faut déterminer le jour du 1er du mois, ce que fait ma boucle `while jour_avant<calendar.weekday(annee,mois,1)` en positionnant, étape par étape, le stylo de la console en dessous du `jour_avant`. Ainsi, lorsque le jour du 1er du mois est atteint, on peut écrire 1 au bon endroit. Ensuite, il faut aller à la ligne après avoir écrit le jour du dimanche. C'est le rôle de la ligne `if calendar.weekday(annee,mois,jour)==6 : print()`.

```
import calendar

def nbr_jours_mois(m,a):
    long=[31,28,31,30,31,30,31,31,30,31,30,31]
    if m==2 and a%4==0 and (a%100!=0 or a%400==0): return 29
    return long[m-1]

mois,annee=12,2021  # modifier ici le mois et l'année
nom_jours=['Lundi','Mardi','Mercredi','Jeudi','Vendredi','Samedi','Dimanche']
nom_mois=['Janvier','Février','Mars','Avril','Mai','Juin','Juillet','Aout',\
          'Septembre','Octobre','Novembre','Décembre']
print(nom_mois[mois-1],annee)
print("     ",end=" ")
for jour in range(7):
    print("{:10}".format(nom_jours[jour]),end=" ")
print()
jour_avant=0
while jour_avant<calendar.weekday(annee,mois,1) :
    print("{:10}".format(" "),end=" ")
    jour_avant+=1
for jour in range(1,nbr_jours_mois(mois,annee)+1):
    print("{:10}".format(jour),end=" ")
    if calendar.weekday(annee,mois,jour)==6 : print()
```

```
Décembre 2021
     Lundi      Mardi      Mercredi   Jeudi      Vendredi   Samedi     Dimanche
                                1          2          3          4          5
          6          7          8          9         10         11         12
         13         14         15         16         17         18         19
         20         21         22         23         24         25         26
         27         28         29         30         31
```

Correction de l'exercice 2.22 (Nombres premiers)
a) La méthode du crible d'Ératosthène est relativement simple à mettre en oeuvre : on parcourt la liste des nombres entiers inférieurs ou égaux à `n` autant de fois qu'il est possible en supprimant les multiples du premier nombre non supprimé.
La liste de départ, obtenue avec `list(range(2,n+1))`, est `[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]` lorsque `n=12`.
La suppression des multiples de `k=2` (`k` indique le premier nombre non supprimé) s'obtient en redéfinissant la liste `prem`, par l'instruction `prem=[p for p in prem if p<=k or p%k!=0]` qui opère le tri (`p<=k or p%k!=0` signifiant : soit `p` est un nombre premier inférieur au dernier nombre non supprimé, soit `p` n'est pas divisible par `k`). On recommence l'opération après avoir choisi la nouvelle valeur de `k` avec `k=prem[prem.index(k)+1]`.
On arrive ainsi à `[2, 3, 5, 7, 11]` après l'avoir parcourue pour `k=2` et `k=3`.

```
def premiers(n):
    prem=list(range(2,n+1))
    k=2
    nRacine=n**0.5
    while k<=nRacine :
        prem=[p for p in prem if p<=k or p%k!=0]
        k=prem[prem.index(k)+1]   # nouveau nombre premier
    return prem

Liste_premiers=premiers(100)
print("Plus grand premier =",Liste_premiers[-1])
print("Nombre de premiers =",len(Liste_premiers))
print("Liste premiers :",Liste_premiers)
```

```
Plus grand premier = 97
Nombre de premiers = 25
Liste premiers : [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Le prochain nombre non supprimé est `k=5`, mais on n'a pas besoin de supprimer ses multiples car le

premier qui n'a pas déjà été supprimé est 5×5, un nombre supérieur à `n=12`. Pour cette raison, on arrête la procédure quand `k` atteint ou dépasse $\sqrt{n}$.

b) Pour déterminer le nombre de nombres premiers inférieurs ou égaux à tous les entiers inférieurs ou égaux à `n`, il suffit de relancer textttn fois la fonction `premiers(nb)` en modifiant l'argument `nb`. C'est ce que fait ce petit script, qui se charge aussi d'une mise en page adaptée, sous la forme d'un tableau.

```
N=100
L=[0]
for nb in range(1,N+1):
    n_premiers=len(premiers(nb))
    L.append(n_premiers)
print("Nombre | Premiers inférieurs")
for nb in range(N+1):
    print("{:6} | {:8}".format(nb,L[nb]))
```

```
Nombre | Premiers inférieurs          . . .
     0 |        0                    88 |       23
     1 |        0                    89 |       24
     2 |        1                    90 |       24
     3 |        2                    91 |       24
     4 |        2                    92 |       24
     5 |        3                    93 |       24
     6 |        3                    94 |       24
     7 |        4                    95 |       24
     8 |        4                    96 |       24
     9 |        4                    97 |       25
    10 |        4                    98 |       25
    11 |        5                    99 |       25
    12 |        5                   100 |       25
     . . .
```

Correction de l'exercice 2.23 (Écriture décimale)

a) Il faut effectuer la division euclidienne de `a` par `b`, puis successivement, du reste multiplié par 10 jusqu'à reconnaître, dans le nouveau reste, un reste déjà obtenu. Les restes successifs sont donc mis dans une liste `suiteRestes` tant qu'ils n'y sont pas déjà. On doit alors déterminer le rang du reste (dans la liste de restes) qui a permis de détecter la répétition. Cela conduit à connaître la longueur de la séquence périodique qu'il faut extraire de la chaîne des caractères du quotient.
La fonction `decimal(a,b)` réalise cela et renvoie un quotient décimal contenant une séquence périodique suivie de trois petits points lorsque le rationnel n'est pas décimal. L'instruction `quotient[longueurQ+rang:]` permet de retrouver la séquence en découpant dans la chaîne de caractères `quotient` la partie finale qui commence par le caractère d'indice `longueurQ+rang` (`longueurQ` étant la longueur de la partie entière complétée par la virgule, tandis que `rang` est le rang du reste qui a permis de détecter la répétition).

```
def decimal(a,b):
    quotient=str(a//b)  # la partie entière du quotient
    reste=a%b           # le premier reste
    decimal=0           # le nombre est un décimal par défaut
    if reste!=0:
        restes=[reste]         # on va mettre dans cette liste tous les restes obtenus
        quotient+=","          # le quotient s'allonge avec une partie décimale non nulle
        longueurQ=len(quotient)# on enregistre la longueur de la partie entière avec virgule
        while decimal==0 :
            quotient+=str(reste*10//b) # on ajoute un chiffre au quotient
            reste=(reste*10)%b         # on recalcule le reste
            if reste==0 : break        # le nombre est décimal, la division s'arrête
            if reste in restes :       # si le reste a déjà été obtenu on arrête la division
                rang=restes.index(reste)          # rang du reste déjà obtenu dans la liste
                sequence=quotient[longueurQ+rang:]# extraction de la séquence périodique
                quotient+="..."
                print("Séquence périodique de longueur "+str(len(sequence))+" : "+sequence)
                decimal=1                         # le nombre est un rationnel non décimal
            else : restes.append(reste)# sinon on place le reste obtenur dans la liste
    return quotient,decimal

num,denom=1,700 #2021,2022
q,i=decimal(num,denom)
nature=["décimal","rationnel non décimal"]
print(num,"/",denom,"est un nombre "+nature[i],":",q)
```

```
Séquence périodique de longueur 6 : 142857
1 / 7 est un nombre rationnel non décimal : 0,142857...
```

Note pour mes élèves de la 1re3 : Le programme a été simplifié par rapport à ce qui a été montré en classe de maths. J'y ai aussi ajouté des commentaires pour qu'il soit plus compréhensible. Si vous avez une meilleure (plus simple) solution, n'hésitez pas à me la transmettre.

b) Le programme précédent a été adapté pour qu'il fasse la même chose que précédemment pour une base quelconque inférieure à 37 (pour utiliser les 26 lettres). J'ai utilisé la fonction `conversionBase(n,base)` pour convertir la partie entière dans la base concernée. Sinon le reste du programme n'utilise pas cette fonction.
On constate que la fraction $\frac{1}{7}$ qui a une écriture illimitée en base 10 est un nombre décimal en base 7

(son écriture dans cette base étant $0,1$). De même $\frac{10}{7}$ s'écrit $1,3$ en base 7.
Remarquez que les divisions s'effectuent toujours en base 10, mais les chiffres du quotient sont convertis dans la base concernée grâce à la syntaxe `ecriture+=chiffre[chiffreQuotient]`.

```
def conversionBase(n,base):
    reste,quotient,num,ecriture=list(),n,n,""
    while quotient>0:
        quotient=num//base
        reste.append(num%base)
        num=quotient
    for j in range(len(reste)): ecriture=chiffre[reste[j]]+ecriture
    return ecriture

chiffre=["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f","g","h",\
         "i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]
a=int(input("Entrer le dividende (base 10) : "))
b=int(input("Entrer le diviseur (base 10) : "))
c=int(input("Entrer la base de sortie (<36) : "))
if a//b>0 : ecriture=conversionBase(a//b,c)  #la partie entière du quotient en base c
else : ecriture="0"
if a%b==0: print("Votre nombre est entier, le voici dans cette base : "+ecriture)
else :
    reste,longueur,suiteRestes,fin=a%b,0,list(),0
    ecriture+=",";
    suiteRestes.append(reste)
    while fin==0:
        reste*=c
        chiffreQuotient=reste//b
        suiteRestes.append(reste%b)
        ecriture+=chiffre[chiffreQuotient]
        if suiteRestes[-1]==0 : fin=1
        else:
            for i in range(len(suiteRestes)-1) :
                if suiteRestes[-1]==suiteRestes[i] :
                    fin=2
                    longueur=len(suiteRestes)-i-1
                    sequence=str(ecriture[-longueur:])
                    ecriture+=sequence+"..."
                else : reste=suiteRestes[-1]
    print("Voici le développement illimité du quotient : "+ecriture)
    if fin==2 : print("Votre nombre a une écriture décimale illimitée périodique\
 de période "+str(longueur)+", la séquence qui se répète est "+sequence)
    else : print("Votre nombre a une écriture finie en base "+str(c)+" : "+ecriture)
```

```
Entrer le dividende (base 10) : 1
Entrer le diviseur (base 10) : 7
Entrer la base de sortie (<36) : 7
Voici le développement illimité du quotient : 0,1
Votre nombre a une écriture finie en base 7 : 0,1

Entrer le dividende (base 10) : 10
Entrer le diviseur (base 10) : 7
Entrer la base de sortie (<36) : 7
Voici le développement illimité du quotient : 1,3
Votre nombre a une écriture finie en base 7 : 1,3

Entrer le dividende (base 10) : 1
Entrer le diviseur (base 10) : 7
Entrer la base de sortie (<36) : 10
Voici le développement illimité du quotient : 0,142857142857...
Votre nombre a une écriture décimale illimitée périodique de période 6, la séquence qui se répète est 142857
```

Correction de l'exercice 2.24 (Élimination cyclique)

a) Commençons par construire la liste initial des 41 numéros (de 1 à 41) attribués aux personnes en cercle.

```
def josephe(n,p):
    rg=0
    initial,ordre=list(range(1,n+1)),[]
    while len(initial)>0:
        rg=(rg+p-1)%len(initial)
        ordre.append(initial.pop(rg))
    return ordre

n,p=41,3
L=josephe(n,p)
print('Ordre de décimation:',L)
#pour savoir dans quel ordre seront supprimés les personnes
passage=n*[0]
for i in range(n): passage[L[i]-1]=i+1
print('Ordre de passage:',passage)
```

```
Ordre de décimation: [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 1, 5, 10, 14, 19,
23, 28, 32, 37, 41, 7, 13, 20, 26, 34, 40, 8, 17, 29, 38, 11, 25, 2, 22, 4, 35, 16, 31]
Ordre de passage: [14, 36, 1, 38, 15, 2, 24, 30, 3, 16, 34, 4, 25, 17, 5, 40, 31, 6, 18,
26, 7, 37, 19, 8, 35, 27, 9, 20, 32, 10, 41, 21, 11, 28, 39, 12, 22, 33, 13, 29, 23]
```

Dans cette liste, on supprime un numéro sur 3 avec la fonction `pop(rg)` qui élimine le numéro de rang `rg` et on enregistre ce numéro dans la liste ordre. La longueur de la liste `len(initial)` changeant à chaque suppression d'une personne, pour savoir le rang `rg` de la prochaine personne à supprimer, on ajoute `p -1` à l'ancien rang (le nombre de personnes sautées) et on se ramène au début de la liste en examinant le reste de la division par `len(initial)`. En effet, arrivés en fin de la liste, il faut continuer

comme si le 1er était derrière le dernier. La boucle `while` qui contient ces deux instructions assure que le processus ne s'achève qu'avec la suppression de la dernière personne.
En remplaçant 41 et 3 par les lettres `n` et `p`, ce programme pourra être réutilisé avec d'autres valeurs. D'après le résultat, Josèphe doit se mettre en 31e position pour avoir le privilège d'être le dernier à s'autodétruire comme le capitaine d'un navire en perdition qui met un point d'honneur à être le dernier à le quitter. La liste `passage` qui a été ajoutée à la fin du programme nous donne le rang de l'exécution de chaque personne : le 1er est tué en 14e, le 2e est tué en 36e, le 3e en 1er, etc.

b) Supposons qu'il y a `n=5` bambins et `p=10` syllabes dans la comptine de sélection (« plouf, plouf, une poule en or c'est toi qui sort »). J'entre `n=5` et `p=10` dans le programme et trouve qu'il faut me mettre en avant-dernier (la position 4).

```
n,p=5,10
L=josephe(n,p)
print('Ordre de décimation:',L)
#pour savoir dans quel ordre seront supprimés les personnes
passage=n*[0]
for i in range(n): passage[L[i]-1]=i+1
print('Ordre de passage:',passage)
```

```
Ordre de décimation: [5, 2, 3, 1, 4]
Ordre de passage: [4, 2, 3, 5, 1]
```

Correction de l'exercice 2.25 (Énigme)

Cet exercice peut paraître un peu abstrait et/ou arbitraire mais il s'agit du premier exemple de système formel donné par Hofstadter [1] dans son monumental livre. Comme il le fait remarquer, les deux premières règles augmentent la longueur des chaînes alors que les deux dernières la font diminuer. Ce peut-il qu'on obtienne `MU` en partant de `MI` ?
Le programme qui traduit cette recherche est une boucle non bornée qui semble infinie. Sur mon ordinateur, je laisse tourner le programme quelques secondes pour obtenir les 8 premières étapes mais la 9e tarde vraiment à s'achever (20 minutes environ), la chaîne `MU` n'ayant pas été trouvée.
J'utilise deux listes `anciens` et `nouveaux` : parcourant la 1re et appliquant les 4 règles à chaque élément, la 2e liste se constitue et, en fin d'étape, se déverse dans la 1re, écrasant celle-ci.
Ce fonctionnement est assez simple, ce sont les règles elles-mêmes qui sont plus délicates à traduire :

- `m[-1:]` désigne le dernier caractère de la chaîne `m`
- `m[1:]` désigne la chaîne `m` privée de son 1er caractère (le fameux `x` de la règle 2)
- `m[0:R]+'U'+m[R+3:]` est la chaîne obtenue en substituant `U` à `III` se trouvant au rang `R`
- `m[0:R]+m[R+2:]` est la chaîne obtenue en éliminant `UU` se trouvant au rang `R`

L'instruction `R=m.find('III',R+1)` recherche `III` dans la chaîne `m` à partir du rang `R+1` et affecte le rang trouvé à `R`. Le grand intérêt de la fonction `find` est de renvoyer 0 si la chaîne n'est pas trouvée au lieu de déclencher une erreur comme le font d'autres fonctions (par exemple la fonction `index`).

```
def regle1(m):
    if m[-1:]=='I'and m+'U' not in nouveaux:
        nouveaux.append(m+'U')

def regle2(m):
    X=m[1:]
    if m+X not in nouveaux:
        nouveaux.append(m+X)

def regle3(m):
    R=m.find('III',0)
    while R>=0:
        if m[0:R]+'U'+m[R+3:] not in nouveaux:
            nouveaux.append(m[0:R]+'U'+m[R+3:])
        R=m.find('III',R+1)

def regle4(m):
    R=m.find('UU',0)
    while R>=0:
        if m[0:R]+m[R+2:] not in nouveaux:
            nouveaux.append(m[0:R]+m[R+2:])
        R=m.find('UU',R+1)
```

```
anciens,n=['MI'],0
print('Génération 0:',anciens)
while 'MU' not in anciens:
    nouveaux,n=[],n+1
    for m in anciens:
        regle1(m)
        regle2(m)
        regle3(m)
        regle4(m)
    if n<3:print('Génération {}:'.format(n),nouveaux)
    else:print('Génération {}: {} chaines'.format(n,len(nouveaux)))
    anciens=nouveaux
print('MU trouvé à la génération',n)
```

```
Génération 0: ['MI']
Génération 1: ['MIU', 'MII']
Génération 2: ['MIUIU', 'MIIU', 'MIIII']
Génération 3: 6 chaines
Génération 4: 15 chaines
Génération 5: 48 chaines
Génération 6: 232 chaines
Génération 7: 1544 chaines
Génération 8: 14959 chaines
Génération 9: 203333 chaines
```

b) Pour effectuer la vérification demandée, j'ai juste affiché le contenu de la liste `nouveaux` aux étapes 0, 1 et 2. Après j'ai affiché le nombre de mots de la liste à chaque étape : ces nombres augmentent de plus en plus et les mots eux-mêmes sont de plus en plus longs (la règle 2 les fait doubler quasiment d'une étape à l'autre) ce qui explique que le programme tarde de plus en plus à achever une étape.

Correction de l'exercice 2.26 (Code César)

a) J'utilise les fonctions `chr` et `ord` pour obtenir le décalage souhaité. On peut s'en servir aussi pour convertir les lettres en majuscules : comme `ord('a')` vaut `97` tandis que `chr(97)` vaut `'a'`, pour convertir en majuscule un caractère ASCII on peut enlever 32 à son code, par exemple `chr(ord('a')-32)` vaut `'A'`. Néanmoins, la syntaxe `'a'.upper()` est préférée ici car elle est plus commode.
Finalement, j'ai utilisé l'instruction `chr((ord(lettre)-65-c)%26+65)` pour obtenir le décalage de `c` lettres. Dans l'énoncé, ceci était mal expliqué puisque j'ai écrit `E code A`, c'est plutôt l'inverse : `A code E` car la lettre `A`, déplacée de `c=4` rangs vers la droite devient un `E`. Si je veux coder un `E` il me faut enlever `c` à son ordre ASCII.
Ces préliminaires étant posés, voici un premier script pour la fonction `encode()`. J'y ai traduit l'encodage d'un texte non préformaté où minuscules, majuscules, caractères accentués, chiffres et signes de ponctuation peuvent être présents. Dans le texte encodé, par contre, la ponctuation a disparu, tout est en majuscule, les lettres étant par ailleurs groupées par 5.
Pourquoi ces choix ? Le texte de l'énoncé semble les suivre et elles sont justifiées car si on conservait la ponctuation, les espaces et la casse des caractères (majuscule/minuscule), le message serait trop facile à décoder. Pour encoder les chiffres je les décale de la clé `c`. Avec `c=4`, `0 code 4` tandis que `6 code 0`. Le texte `"XKJFKQN"` codé avec un code César `c=4` signifie `"Bonjour"` (pour décoder ce texte, on tape `encode("XKJFKQN",-4)`.

```
def encode(t,c):
    tt,nbr_lettres='',0
    for lettre in t :
        if lettre in ['é','è','ê','ë','È','É','Ê','Ë']:lettre='E'
        if lettre in ['à','À']:lettre='A'
        if lettre in ['ç']:lettre='C'
        lettre=lettre.upper()
        if 64<ord(lettre)<91:  #pour encoder les majuscules (les lettres)
            tt+=chr((ord(lettre)-65-c)%26+65)
            nbr_lettres+=1
        if 47<ord(lettre)<58:  #pour encoder les chiffres
            tt+=chr((ord(lettre)-48-c)%10+48)
            nbr_lettres+=1
        if nbr_lettres%5==0 : tt+=" "
    return tt

texte='abcdefghijklmnopqrstuvwxyz0123456789'
clef=4
print('alphabet clair >> CODE ({}):'.format(clef))
print(encode(texte,0))
print(encode(texte,clef))
print(encode("Bonjour César, ça va bien?",0))
print(encode("Bonjour César, ça va bien?",clef))
```

```
alphabet clair >> CODE (4):
ABCDE FGHIJ KLMNO PQRST UVWXY Z0123 45678 9
WXYZA BCDEF GHIJK LMNOP QRSTU V6789 01234 5
BONJO URCES ARCAV ABIEN
XKJFK QNYAO WNYWR WXEAJ
>>> encode("XKJFKQNYAOWNYWRWXEAJ",-4)
'BONJO URCES ARCAV ABIEN '
```

b) La fonction `analyse` procède au décompte des lettres du message, l'effectif de chacune étant enregistré dans le dictionnaire `alpha` avec les lettres de l'alphabet présentes dans le texte comme clés et le nombre d'occurrences de chacune comme valeur.

```
def analyse(t):
    alpha,beta={},[]
    for lettre in t :
        if 64<ord(lettre)<91:# pour enregistrer seulement les lettres
            if lettre in alpha:
                alpha[lettre]+=1
            else : alpha[lettre]=1
    print('alpha',alpha)
    val=[]                  # constitution de la liste des effectifs
    for c in alpha:
        if alpha[c] not in val : val.append(alpha[c])
    val.sort(reverse=True)
    for n in val:
        for c in alpha:
            if alpha[c]==n:
                beta.append(c)
    print('beta',beta)
    return beta

texte='XQEBX MUEMZ FQDUQ EXQEB XGEOA GDFQE EAZFX QEYQU XXQGD QE'
Ordre=analyse(texte)
for i in range(3):
    print('clé={}'.format(((ord(Ordre[i])-ord('E'))%26)),encode(texte,((ord(Ordre[i])-ord('E'))%26)))
```

```
alpha {'X': 7, 'Q': 9, 'E': 9, 'B': 2, 'M': 2, 'U': 3, 'Z': 2, 'F': 3, 'D': 3, 'G': 3, 'O': 1, 'A': 2, 'Y': 1}
beta ['Q', 'E', 'X', 'U', 'F', 'D', 'G', 'B', 'M', 'Z', 'A', 'O', 'Y']
clé=12 LESPL  AISAN  TERIE  SLESP  LUSCO  URTES  SONTL  ESMEI  LLEUR  ES
clé=0 XQEBX  MUEMZ  FQDUQ  EXQEB  XGEOA  GDFQE  EAZFX  QEYQU  XXQGD  QE
clé=19 EXLIE  TBLTG  MXKBX  LEXLI  ENLVH  NKMXL  LHGME  XLFXB  EEXNK  XL
```

Je crée ensuite une liste `val` qui va contenir les effectifs trouvés, sans doublon. Cette liste est ensuite

1. Douglas Hofstadter, *Gödel, Escher, Bach, les Brins d'une Guirlande Éternelle*, Dunod 2000, pp.38-47.

triée dans le sens décroissant (avec l'option `reverse=True` de la fonction `sort`). La liste `beta` est alors constituée par la liste des lettres les plus fréquentes du message dans l'ordre décroissant. Les lettres les plus fréquentes du message de l'énoncé sont le `'E'` et le `'Q'` (9 occurrences chacune), suivie du `'X'` (7 occurrences), etc.
J'affiche les trois premiers choix, espérant que le `'E'` est codé par une de ces trois lettres.
Il s'avère ici que ce n'est pas le `'E'` ; on s'en serait douté car si le `'E'` était codé `'E'`, le message codé serait en clair. La clé est donc trouvée en enlevant au rang de `'Q'` le rang de `'E'`. Avec cette clé `c=12`, on retrouve le sens du message dans lequel il n'y a qu'à rétablir les césures correctes des mots :
LES PLAISANTERIES LES PLUS COURTES SONT LES MEILLEURES

Correction de l'exercice 2.27 (Stock)

a) La fonction `commande(livre,nombre)` examine le dictionnaire `stock` et renvoie le message approprié après avoir mis à jour le dictionnaire.

```
def commande(livre,nombre):
    reponse=["référence absente","livre épuisé","commande satisfaite","commande partiellement exécutée"]
    if livre not in stock :
        return reponse[0]
    elif stock[livre]==0 :
        return reponse[1]
    elif stock[livre]>=nombre :
        stock[livre]-=nombre
        return reponse[2]
    stock[livre]=0
    return reponse[3]

stock={"Python en 100 leçons":1500,"Je programme comme un pro":2500,"Jeux en Python":900}

#question 1
print(commande("Jeux en Python",750))
print(commande("Jeux en Python",500))
print(commande("Je programme en Python",100))
print(commande("Python en 100 leçons",1500))
```

```
commande satisfaite
commande partiellement exécutée
référence absente
commande satisfaite
```

```
>>> print(stock)
{'Python en 100 leçons': 0, 'Je programme comme un pro': 2500, 'Jeux en Python': 0}
```

b) La fonction `ajout(livre,quantité)` réalise simplement cette mise à jour du dictionnaire `stock`.

```
def ajout(livre,quantite):
    if livre not in stock :
        stock[livre]=quantite
    else :
        stock[livre]+=quantite
    print(stock)
    return "ajout pris en compte"

print(ajout("Je programme en Python",1000))
print(ajout("Jeux en Python",5000))
```

```
{'Python en 100 leçons': 0, 'Je programme comme un pro': 2500, 'Jeux en Python': 0, 'Je programme en Python': 1000}
ajout pris en compte
{'Python en 100 leçons': 0, 'Je programme comme un pro': 2500, 'Jeux en Python': 5000, 'Je programme en Python': 1000}
ajout pris en compte
```

c) La nouvelle fonction `commande` prend le nom du client comme 3e argument et, en cas de commande partiellement exécutée, enregistre le reliquat dans une liste `reliquats` contenant des tuples de la forme `(livre,reliquat,client)`. La fonction `ajout` permet de servir un reliquat quand c'est possible et affiche, en plus du stock, la liste `reliquats` si elle n'est pas vide.

```
def commande(livre,nombre,client):
    reponse=["référence absente","livre épuisé","commande satisfaite","commande partiellement exécutée"]
    if livre not in stock :
        reliquats.append((livre,nombre,client))# commande non satisfaite > reliquat total
        return reponse[0]
    elif stock[livre]==0 :
        reliquats.append((livre,nombre,client))# commande non satisfaite > reliquat total
        return reponse[1]
    elif stock[livre]>=nombre :
        stock[livre]-=nombre
        return reponse[2]
    reliquats.append((livre,nombre-stock[livre],client))# commande partiellement satisfaite > reliquat partiel
    stock[livre]=0
    return reponse[3]
```

```
def ajout(livre,quantite):
    if livre not in stock :
        stock[livre]=quantite
    else :
        stock[livre]+=quantite
    print(stock)
    for reference in reliquats :
        if reference[0]==livre:
            print(commande(livre,reference[1],reference[2]))
    return "ajout pris en compte"

stock={"Python en 100 leçons":1500,"Je programme comme un pro":2500,"Jeux en Python":900}
reliquats=list() # contiendra les reliquats de commande sous la forme (livre,reliquat,client)

print(commande("Jeux en Python",1500,"FNOC"))
print(ajout("Jeux en Python",5000))
```

```
commande partiellement exécutée
{'Python en 100 leçons': 1500, 'Je programme comme un pro': 2500, 'Jeux en Python': 5000}
commande satisfaite
ajout pris en compte
```

Correction de l'exercice 2.28 (Association sportive)

a) La fonction `ins(nom,liste)`complète les listes `natation`, `paddling` et `voile` en y enregistrant les noms des personnes inscrites (initialement, ces listes sont créées vides). Pour éviter les doublons (deux inscriptions avec le même nom), on utilise des ensembles

```
def ins(nom,sports):
    if "natation" in sports:
       natation.add(nom)
    if "paddling" in sports:
       paddling.add(nom)
    if "voile" in sports:
       voile.add(nom)
    print("Insertion de",nom,"effectuée")

natation=set()
paddling=set()
voile=set()
#question 1
ins("AB",["natation","paddling"])
ins("BC",["natation","voile"])
ins("CD",["voile"])
ins("DE",["paddling"])
ins("EF",["natation"])
ins("FG",["paddling","natation"])
ins("GH",["paddling","natation","voile"])
```

```
Insertion de AB effectuée
Insertion de BC effectuée
Insertion de CD effectuée
Insertion de DE effectuée
Insertion de EF effectuée
Insertion de FG effectuée
Insertion de GH effectuée
```

b) La fonction `affichage` donne le contenu des trois listes simples, des trois listes des double inscriptions et la liste des triple inscriptions. On crée ici les intersections d'ensemble avec l'opérateur `&` dont c'est la fonction.

```
def affichage():
    print()
    print("natation =",natation,len(natation),"p.")
    print("paddling =",paddling,len(paddling),"p.")
    print("voile =",voile,len(voile),"p.")
    natPadd=natation & paddling
    print("natation+paddling =",natPadd,len(natPadd),"p.")
    natVoil=natation & voile
    print("natation+voile =",natVoil,len(natVoil),"p.")
    padVoil=paddling & voile
    print("paddling+voile =",padVoil,len(padVoil),"p.")
    natPadVoil= natation & paddling & voile
    print("natation+paddling+voile =",natPadVoil,len(natPadVoil),"p.")

affichage()
```

```
natation = {'GH', 'AB', 'BC', 'EF', 'FG'} 5 p.
paddling = {'GH', 'DE', 'AB', 'FG'} 4 p.
voile = {'CD', 'GH', 'BC'} 3 p.
natation+paddling = {'GH', 'AB', 'FG'} 3 p.
natation+voile = {'GH', 'BC'} 2 p.
paddling+voile = {'GH'} 1 p.
natation+paddling+voile = {'GH'} 1 p.
```

p-uplets

Correction de l'exercice 2.29 (Fractions)

a) La fonction `irreductible(a,b)` transforme une fraction quelconque en sa forme irréductible en utilisant le PGCD des deux nombres calculé directement dans le corps de la fonction (on aurait pu l'extérioriser en utilisant une fonction `PGCD(a,b)`). Cette fonction renvoie un tuple constitué du numé-

rateur et du dénominateur de la fraction irréductible.

```
def irreductible(a,b):
    assert type(a)==int and type(b)==int and b>0
    num,denom=abs(a),b # recherche du pgcd
    if a==0 : return a,b
    while num!=denom :
        if num>denom :num-=denom
        else : denom-=num
    return a//num,b//num
```

```
>>> irreductible(490,210)
(7, 3)
>>> irreductible(4896,68946)
(816, 11491)
```

b) La fonction `operation(a1,b1,op,a2,b2)` effectue sur les fractions `(a1,b1)` et `(a2,b2)` l'opération `op` (le caractère `op` étant choisi dans le p-uplet `("+","-","*","/")`). Cette fonction retourne le résultat sous la forme d'une fraction irréductible en utilisant la fonction précédente.

```
def operation(a1,b1,op,a2,b2):
    a,b=0,0
    if op=="+":
        a=a1*b2+a2*b1
        b=b1*b2
    if op=="-":
        a=a1*b2-a2*b1
        b=b1*b2
    if op=="*":
        a=a1*a2
        b=b1*b2
    if op=="/":
        a=a1*b2
        b=b1*a2
    return irreductible(a,b)
```

```
print(operation(1,2,"+",1,6))
print(operation(1,12,"+",15,6))
print(operation(-1,123,"+",15,456))
print(operation(-1,123,"/",15,456))
print(operation(112,123,"*",-125,456))
```

```
(2, 3)
(31, 12)
(463, 18696)
(-152, 615)
(-1750, 7011)
```

c) La fonction `affichage(a1,b1,op,a2,b2)` écrit sur la console l'opération ainsi que son résultat sous la forme habituelle des fractions, sur 3 lignes. Pour réaliser cela, j'ai écrit une fonction `long(a,b)` qui détermine la longueur du trait de fraction (elle est égale à la longueur du plus long des deux nombres) et qui renvoie les numérateur, dénominateur et trait de fraction formaté selon cette longueur. L'affichage peut alors être réalisé, la ligne 1 pour les numérateurs, la ligne 2 pour les traits de fraction et les symboles opératoires et la ligne 3 pour les dénominateurs.

```
def long(a,b): # nécessaire à l'affichage
    A,B,C=str(a),str(b),"-"
    m=max(len(A),len(B))
    while len(A)<m : A=" "+A
    while len(B)<m : B=" "+B
    while len(C)<m : C="-"+C
    return A,B,C

def affichage(a1,b1,op,a2,b2):
    a3,b3=operation(a1,b1,op,a2,b2)
    A1,B1,C1=long(a1,b1)
    A2,B2,C2=long(a2,b2)
    A3,B3,C3=long(a3,b3)
    print(A1+"   "+A2+"   "+A3)
    print(C1+" "+op+" "+C2+" = "+C3)
    print(B1+"   "+B2+"   "+B3)

affichage(1,2,"+",1,6)
affichage(1,12,"+",15,6)
affichage(-1,123,"+",15,456)
affichage(-1,123,"/",15,456)
affichage(112,123,"*",-125,456)
```

```
1   1   2
- + - = -
2   6   3
 1   15   31
-- + -- = --
12    6   12
 -1    15     463
--- + --- = -----
123   456   18696
 -1    15   -152
--- / --- = ----
123   456    615
112   -125   -1750
--- * ---- = -----
123    456    7011
```

Correction de l'exercice 2.30 (Croisement)

a) La fonction `droite(xa,ya,xb,yb)` renvoie les coefficients `(a,b,c)` de l'équation de droite `(AB)`. Cette fonction peut servir pour toute droite : je l'utilise ci-dessous pour la droite `(CD)`.

```
def droite(xa,ya,xb,yb):
    a=yb-ya
    b=xa-xb
    c=xa*(ya-yb)+ya*(xb-xa)
    return a,b,c

a1,b1,c1=droite(1,1,5,2) # coordonnées de A et B
print("equation de la droite (AB):",a1,"x+",b1,"y+",c1,"=0")
a2,b2,c2=droite(5,1,4,4) # coordonnées de C et D
print("equation de la droite (CD):",a2,"x+",b2,"y+",c2,"=0")
```

```
equation de la droite (AB): 1 x+ -4 y+ 3 =0
equation de la droite (CD): 3 x+ 1 y+ -16 =0
```

b) La fonction `intersection(a1,b1,c1,a2,b2,c2)` renvoie les coordonnées `(x,y)` du point d'intersection unique des droites d'équation $a_1x + b_1y + c_1 = 0$ et $a_2x + b_2y + c_2 = 0$ quand il existe et

`(None,None)` sinon (cas de droites parallèles et disjointes ou confondues).

```
def intersection(a1,b1,c1,a2,b2,c2):
    """ ne traite que du cas des droites sécantes
        les droites parallèles (disjointes ou confondues) retournent 'none' """
    if a1*b2==a2*b1: # cas des droites parallèles
        return None,None
    y=(a1*c2-a2*c1)/(a2*b1-a1*b2)
    x=-(b1*y+c1)/a1
    return x,y

print("coordonnées de l'intersection entre (AB) et (CD):")
x0,y0=intersection(a1,b1,c1,a2,b2,c2)
print("x0=",x0,"y0=",y0)
```

```
coordonnées de l'intersection entre (AB) et (CD):
x0= 4.6923076923076925 y0= 1.9230769230769231
```

c) La formule du produit scalaire des vecteurs $\vec{u}$`(x1,y1)` et $\vec{v}$`(x2,y2)` ($\vec{u}\cdot\vec{v}=$`x1×x2+y1×y2`) sert à déterminer si le point d'intersection trouvé (on va le noter `O`) appartient au segment `[AB]` : si $\vec{AO}\cdot\vec{BO}>0$ les deux vecteurs ont le même sens et le point `O` n'appartient pas au segment `[AB]`, sinon ils ont un sens contraire et le point `O` appartient bien au segment, autrement dit $O\in[AB]\iff\vec{AO}\cdot\vec{BO}<0$.

```
def interieur(xa,ya,xb,yb,xo,yo): # question 3
    """ renvoie True si le point O(xo,yo) appartient au segment [AB] """
    vec1=(xo-xb,yo-yb) #coordonnées du vecteur BO
    vec2=(xo-xa,y0-ya) #coordonnées du vecteur AO
    produit_scalaire=vec1[0]*vec2[0]+vec1[1]*vec2[1]
    if produit_scalaire<0 :
        return True
    return False

print(interieur(1,1,5,2,x0,y0))
```

```
True
```

d) La fonction `coupe(xa,ya,xb,yb,xc,yc,xd,yd)` renvoie `True` si `[CD]` coupe `[AB]` et `False` sinon. J'ai testé cette fonction avec les trois quadruplets de l'énoncé auxquels j'en ai ajouté un 4e pour la situation où les droites se coupent mais pas les segments. Ce point n'était pas très clair dans l'énoncé, mais le but était de tester le croisement des segments (pas des droites). Le cas ajouté est donc le suivant :
`A(1,1)`, `B(4,2)`, `C(5,1)` et `D(4,4)` (voir l'illustration)

```
def coupe(xa,ya,xb,yb,xc,yc,xd,yd):
    """ renvoie True si [CD] coupe [AB] """
    a1,b1,c1=droite(xa,ya,xb,yb)
    a2,b2,c2=droite(xc,yc,xd,yd)
    x0,y0=intersection(a1,b1,c1,a2,b2,c2)
    if x0 != None and interieur(xa,ya,xb,yb,x0,y0):
        return True
    return False

print(coupe(1,1,5,2,5,1,4,4))
print(coupe(1,1,5,2,1,2,4,4))
print(coupe(1,1,5,2,1,2,3,2.5))
print(coupe(1,1,4,2,5,1,4,4))#jeu de test supplémentaire
```

```
True
False
False
False
```