FORMATION PYTHON POUR LES PROFESSEURS DU SECONDAIRE. QUELQUES THÈMES D'EXERCICES.

par

Landry Salle

Résumé. — Les deux premiers thèmes consistent en une prise en main du langage et de l'environnement et constituent une porte d'entrée pour les débutants. On peut ensuite piocher à l'envie dans ce qui suit.

Table des matières

Thème 1. Prise en main de l'environnement.

L'environnement *Idle* propose deux fenêtres, l'une étant une console interactive (qui permet d'exécuter directement des instructions simples), l'autre permettant d'éditer des fichiers (ce qui permet d'écrire des programmes complexes et de les sauvegarder).

Exercice 1. **Objectifs :** découverte de la console interactive ; exécuter une commande simple ; exécuter un bloc ; indentation automatique. **Mais aussi :** type entier et type flottant ; définition d'une fonction, appel à une fonction ; boucle bornée ; instruction conditionnelle et test.

Dans la console, exécuter successivement chacune des instructions suivantes (on valide après chaque ligne, les chevrons n'ont pas à être écrits, ils matérialisent ce qu'on appelle une « invite de commande ») :

```
>>> 2+2
>>> a = 3
>>> a + 5
>>> a + b
>>> b = 0
>>> a + b
>>> type(a)
>>> a = 3.0
>>> type(a)
>>> type(b)
>>> type(a+b)
```

On passe maintenant à des instructions plus complexes, comprenant une définition de fonction, une boucle, une instruction conditionnelle (ce qu'on appelle un « bloc »ci-dessous). En présence d'un bloc, la console interactive a un comportement un peu différent. Le code n'est pas exécuté après validation : la validation est interprétée comme un caractère de retour à la ligne, ce qui permet d'étendre la portée du bloc sur plusieurs lignes. De plus, on dispose d'une fonctionnalité d'indentation automatique : lors d'un retour à la ligne après un caractère « : », le curseur d'édition est directement positionné de manière à pouvoir éditer l'intérieur du bloc ainsi créé.

Deux validations successives permettent de déclencher l'exécution.

```
>>> def f(t): return(t**2)
>>> f(0), f(a)
>>> for i in range(10): f(i)
>>> for i in range(10):
        if i%2 == 0:
            f(i)
```

La fenêtre de la console propose quelques commandes usuelles sous forme d'onglets. Je commente brièvement deux onglets :

— l'onglet de redémarrage permet de redémarrer la console, ce qui a pour effet que toutes les variables définies sont « oubliées »(et de l'espace mémoire est libéré) ;

— l'onglet de débuggage : un outil de débuggage permet de tester des portions de code, de suivre les valeurs affectées dans différentes variables, ce qui permet de comprendre et corriger certaines erreurs. À réserver probablement à des utilisateurs avancés (pour ma part, je fais du débuggage « à la main »). On y reviendra.

◇

Exercice 2. Objectifs : la commande `help`.

La commande `help` permet d'obtenir, directement dans la console, des éléments de documentation sur le langage. On peut demander de l'aide (les exemples ci-dessous ne sont pas conçus pour que vous lisiez les aides concernées *in extenso* - je dirais même au contraire, dans certains cas) :

— sur un objet Python, que celui-ci soit entré directement comme une valeur (`help([])`, `help(2)` par exemple), ou comme une variable dans laquelle une valeur est stockée ;

— sur une commande du langage (`help('for')`, `help('def')` par exemple). Dans ce cas, la commande est passée entre guillemets simples droits. On peut aussi lire une aide sur une méthode ou un attribut d'un objet : par exemple `help([].append)` pour la méthode `append` sur les listes ;

— sur un module ou une bibliothèque, précédemment importée :

```
>>> import turtle
>>> help(turtle)
```

Une bonne partie des exemples ci-dessus montrent bien que l'aide en mode console est souvent assez touffue (volume du texte, vocabulaire technique), et qu'une lecture d'une documentation est souvent préférable. ◇

Exercice 3. Objectifs : créer, sauvegarder, coder, exécuter un fichier. Naviguer entre fichier et console. **Mais aussi :** création d'une liste vide, remplissage d'une liste avec la méthode `append`, boucle « non bornée », booléen.

Ouvrir un nouveau fichier (onglet File (Fichier) > New File dans la fenêtre comportant la console). Dans ce fichier, taper le code suivant (là aussi, l'indentation automatique lors des validations permet facilement de mettre en forme le code tout en respectant la syntaxe du langage ; la suppression d'une indentation superflue se fait facilement avec la touche de retour arrière) :

```
def carres(N):
    L = []
    i = 0
    while i**2 <= N:
        L.append(i**2)
        i = i+1
    return(L)
```

Il s'agit maintenant d'exécuter le code : on peut utiliser l'onglet Run (Exécuter) > Run Module, ou tout simplement la touche de raccourci F5. Il convient alors de sauvegarder le fichier, en le nommant, sur un disque facilement accessible (le disque dur de l'ordinateur, ou une clé USB). L'extension habituellement utilisée pour les fichiers contenant du code Python est `.py`.

Revenir alors sur la console. Celle-ci a redémarré. La fonction `carres` est définie. Exécuter dans la console :

```
>>> carres(100)
```

Écrire dans le fichier une fonction permettant de tester si un entier est un carré (on renverra un booléen, `True` ou `False`), puis tester cette fonction en mode console. On se servira de la fonction `carres` précédemment programmée, et du test d'appartenance à une liste. Le code pourra donc ressembler à cela :

```
def est_carre(N):
    L = carres(N)
    if ... :
        return(...)
    else ... :
        return(...)
```

Ceux qui sont le plus à l'aise éviteront l'utilisation de l'instruction conditionnelle et programmeront cette fonction en une ligne. Dans tous les cas, il convient de tester la fonction ainsi programmée en exécutant le fichier et en faisant quelques appels à cette fonction dans la console. ◇

Exercice 4. Objectifs : découvrir quelques messages d'erreurs ; boucle infinie ; interrompre une exécution.

Les quelques commandes suivantes devraient permettre de découvrir les erreurs les plus fréquentes (voir en ligne `http://www.fil.univ-lille1.fr/~wegrzyno/portail/Info/Doc/HTML/exceptions.html` pour des exemples similaires). On part d'une console redémarrée.

Erreur de nom :

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

Erreurs de syntaxe :

```
>>> a =/= b
```

```
>>> for i in range(10):
print(i)
```

```
>>> for i in range(10):
        print(i)
         print(i**2)
```

```
>>> for i in range(10)
        print(i)
```

Erreurs de type :

```
>>> []/8
```

```
>>> [] + 8
```

Erreur d'indice :

```
>>> L = [1]
>>> L[1]
```

Division par zéro :

```
>>> 10//0
```

Dépassements de capacité :

```
>>> 2.0**1024
```

```
>>> def facto(n):
        if n == 0:
            return(1)
        else:
            return(n*facto(n-1))

>>> facto(1000)
```

Enfin, il n'y aura pas de message d'erreur, mais une boucle infinie pose quand même problème. La combinaison de touche Ctrl + C permet d'interrompre l'exécution :

```
>>> while True:
      pass
```

◇

Thème 2. Boucles, `range`, listes et fonctions.

Exercice 1. Objectif : découvrir la syntaxe des boucles « bornées »(commande `for` en Python) ; boucler sur un `range`, boucler sur les éléments d'une liste.

Les boucles bornées sont adaptées pour répéter une instruction ou un bloc d'instructions un nombre de fois prescrit à l'avance. On distingue dans le langage Python deux manières d'aborder ce procédé :

— La manière classique qu'on trouve dans tous les langages, consiste à utiliser un indice de boucle (donc un nombre entier). Une commande spécifique permet de faire cela : la commande `range`. Dans son utilisation basique, l'indice varie entre la valeur 0 (incluse) et une valeur n (exclue), avec un pas de 1. La commande `range` offre toutefois une certaine souplesse sur la manière dont l'indice varie (voir exercice 2).

— Le langage Python offre toutefois une fonctionnalité plus avancée, qui s'avère très pratique dans certaines situations : la possibilité de parcourir directement les éléments d'une liste, en s'affranchissant de l'utilisation d'un indice de boucle. Cette fonctionnalité spécifique est obtenue grâce à l'introduction et la mise en œuvre des types de données *itérables* et des *itérateurs* : il n'est pas dans les objectifs de cet exercice de rentrer dans la description de ces notions, mais tout utilisateur de Python éprouvera certainement à un moment ou à un autre le besoin de les étudier pour comprendre plus en détail certains comportements observés.

Commençons par l'utilisation de `for i in range(n)` : une telle boucle comprend n passages, et l'indice i varie entre les valeurs 0 (incluse) et n (exclue). Cette convention de numérotation peut être un peu troublante, mais on s'y fait très vite : à partir du moment où la valeur 0 est prise comme valeur de départ d'une numérotation, il est nécessaire que la valeur d'arrivée n soit exclue lors de l'utilisation de `range(n)` afin que la boucle comporte bien n passages.

1. On considère le code suivant :

```
>>> u0, n, r = 3.4, 10, 0.2 # affectation simultanée (économie de papier)
>>> u = u0
>>> for i in range(n): u = u + 0.2
```

   Traduire dans le langage des suites ce que permet de calculer ce code (on pourra le tester bien entendu).

2. Écrire un code permettant de calculer le terme d'indice 100 de la suite géométrique de premier terme $u_0 = 3,4$ et de raison $0,9$.
3. Écrire un code permettant de calculer le terme d'indice 1000 de la suite $(u_n)_n$ définie par $u_0 = 0$ et la relation de récurrence $u_{n+1} = u_n + (-1)^n n$.
4. Écrire un code permettant de calculer le nombre 100! (factorielle de 100).

Passons maintenant à quelques boucles sur des listes.

5. On considère le code suivant :

```
>>> L = [i for i in range(100)] # création de la liste des entiers entre 0 et 100 (exclu)
>>> S = 0
>>> for x in S: S = S + x
```

   Quelle valeur est stockée dans la variable `S` après exécution de ce code ?

6. Comment modifier le code précédent afin de former le produit des éléments d'une liste ?
7. On considère le code suivant :

```
>>> from random import randrange # importation d'une fonction de bibliothèque
>>> L = [randrange(100) for i in range(10)] # création d'une liste de dix entiers
                                            # pris aléatoirement entre 0 et 100
>>> b = False
>>> for x in L:
```

```
if x == 50:
    b = True
```

Quelle information est codée par le booléen `b` à l'issue de l'exécution de ce code ?

◇

Exercice 2. Objectif : syntaxe de la commande `range`.

Tester les commandes suivantes afin de comprendre les différents usages envisageables de la commande `range` :

```
>>> list(range(10))
>>> list(range(1,10))
>>> list(range(1,10,3))
>>> list(range(1,10,-1))
>>> list(range(10,0,-1))
```

On notera que la commande `list` n'est utilisée ici que pour visualiser les objets générés par la commande `range` sous forme de liste. En effet, la commande `range` ne génère pas de liste, mais un objet d'un type spécifique (on peut y penser comme une « liste virtuelle » : c'est-à-dire que la liste n'est pas précalculée, ce qui permet d'économiser de la place en mémoire, et les éléments ne seront calculés qu'au fur et à mesure des besoins). Il ne faut pas en pratique convertir les objets `range` en liste, puisqu'on perdrait en faisant cela l'optimisation prévue par le langage. ◇

Exercice 3. Objectif : se familiariser avec la syntaxe sur les fonctions.

Cet exercice est jumeau de l'exercice 1 : on va reprendre les mêmes objectifs, en programmant des fonctions.

Voici le code d'une fonction :

```
1 def f(t):
2     ''' Calcule et retourne la valeur de l'expression t**2 + t + 1.'''
3     return(t**2 + t + 1)
```

Il convient de connaître le vocabulaire suivant :

- Ligne 1 : le mot-clef `def` indique que ce qui suit constitue la *définition* d'une fonction ; sont attendus le *nom de la fonction* (ici `f`), et le ou les arguments (ici `t`). Le *nom de l'argument* indique que dans le corps de la fonction, avant éventuelle modification, la variable `t` fait référence à la valeur passée en argument.
- Ligne 2 : c'est une ligne de *documentation* (optionnel, recommandé, mais dispensable sur un code aussi simple), qui décrit rapidement ce qu'on appelle la *spécification* de la fonction ;
- Ligne 3 : la commande `return` prend en argument une expression, qui sera évaluée pour obtenir la *valeur de retour*.

1. Voici le code d'une fonction :

```
def terme_suite_arithmetique(a,n,r):
    ''' Calcule et renvoie le terme u_n de la suite arithmétique de premier terme
        u_0 = a et de raison r.'''
    u = a
    for i in range(n):
        u = u + r
    return(u)
```

Sur le même modèle, écrire (dans un fichier) des fonctions répondant aux spécifications suivantes. Chaque fonction sera documentée, et testée dans la console.

- une fonction `terme_suite_geometrique` calculant et renvoyant le terme u_n d'une suite géométrique de premier terme $u_0 = a$ et de raison r (arguments a, n et r) ;

— une fonction `factorielle` prenant en argument un entier positif n qui calcule et renvoie $n!$;

— une fonction `suite_recurrente` prenant en arguments une fonction `f`, un flottant `a`, et un entier positif `n` et renvoyant le terme u_n de la suite définie par le premier terme $u_0 = a$ et la relation de récurrence $u_{n+1} = f(u_n)$;

— écrire une variante de la fonction précédente adaptée à une relation de récurrence de la forme $u_{n+1} = f(u_n, n)$.

2. Écrire une fonction prenant en argument une liste `L` et renvoyant la somme des éléments de cette liste.

◇

Exercice 4. Objectif : se familiariser avec les manipulations élémentaires sur les listes.
En mode console, exécuter les commandes suivantes (ne pas taper les commentaires) :

```
>>> L = []                          # Création d'une liste vide
>>> L.append(0)                     # Ajout d'un élément en fin de liste
>>> L[0]                            # Lecture du premier élément
>>> L[1]                            # Lecture du deuxième élément
>>> for i in range(10): L.append(i) # Ajout de plusieurs éléments à l'aide d'une boucle
>>> len(L), L[-1]                   # Longueur de la liste, lecture du dernier élément
>>> L[2:5]                          # Lecture d'une tranche
>>> L[2:5] = [3,4]                  # Modification d'une tranche
>>> L[2:5:2]                        # Lecture d'une tranche avec pas
```

1. Écrire une fonction qui prend en argument un entier naturel n, et renvoie la liste de tous les entiers compris entre 1 et n.
2. Écrire une fonction qui prend en argument deux entiers `n1` et `n2` et renvoie la liste des entiers compris entre `n1` (au sens large) et `n2` (au sens strict).
3. Écrire une fonction qui prend en argument un entier naturel n, et renvoie la liste de tous les nombres premiers inférieurs ou égaux à n. J'encourage à ne pas lire le code ci-dessous si possible, mais voici une trame possible en cas de nécessité :

```
def premiers(N):
    L = [i for i in range(2,n+1)] # liste des entiers entre 2 et n
    P = ...         # initialisation de la liste des nombres premiers
    for x in L:     # parcours de la liste L
        b = ...     # initialisation d'un booléen
        for p in P: # parcours des nombres premiers déjà calculés
            if ... : # test de divisibilité
                ... # modification de b
        if ... :    # mise à jour éventuelle de la liste des nombres premiers
            ...
    return(P)
```

◇

Thème 3. Modules Numpy, Scipy et Matplotlib.

On se contente de donner dans ce thème quelques brefs exemples d'utilisation des bibliothèques de tracé et de calcul numérique.

Exercice 1. Un exemple de tracé. sous `matplotlib.pyplot`. Exécuter le code suivant :

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> T = np.arange(0,2*np.pi,0.05)
>>> X = np.sin(2*T)
>>> Y = np.sin(3*T)
>>> plt.plot(X,Y)
>>> plt.show()
```

Adapter ensuite pour tracer quelques courbes paramétrées, courbes représentatives de fonctions usuelles, etc. (on pourra travailler dans un fichier pour faire plus facilement varier des paramètres). ◇

Exercice 2. Le type `matrix` de la bibliothèque `Numpy`. Exécuter le code suivant :

```
>>> import numpy as np
>>> M = np.matrix([[2,1,1],[1,2,1],[1,1,2]])
>>> N = M**(-1)
>>> M*N
```

◇

Exercice 3. Problème de Cauchy. Exécuter le code suivant :

```
>>> import scipy.integrate as sci
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def f(y,t): return(-t*y)
>>> T = np.arange(0,5,0.01)
>>> S = sci.odeint(f,1,T)
>>> plt.plot(T,S)
>>> plt.show()
```

Adapter le code pour résoudre le problème du pendule simple $y'' = \sin(y)$: c'est une équation du deuxième ordre ; or, la fonction `odeint` attend un problème du premier ordre ; il faut donc s'y ramener en considérant une fonction vectorielle $Y = \begin{pmatrix} y \\ y' \end{pmatrix}$ à valeurs vectorielles ; on définira une fonction f par $f(Y,t) = [Y[1], np.sin(Y[0])]$. ◇

Exercice 4. Zéros d'une fonction, optimisation. Exécuter le code suivant :

```
>>> import scipy.optimize as sco
>>> def f(t): return(t**2 + t - 2)
>>> sco.root(f,0)
>>> sco.minimize(f,0)
```

◇

Thème 4. Statistiques.

L'objet des statistiques est en premier lieu de résumer des données numériques multiples à l'aide d'indicateurs simples. Pour représenter informatiquement ces données multiples, on utilise des types de données composés, ou types conteneurs. On s'autorise donc dans cette section à manipuler des listes à un niveau élémentaire, sans soulever aucune question sur la manière dont elles sont représentées en mémoire. L'étude des listes n'apparaît pas dans le programme de seconde, mais il me paraît difficile de faire l'économie de leur utilisation.

Exercice 1.

1. Écrire une fonction qui prend en argument une liste de valeurs numériques et renvoie leur moyenne.
2. Écrire une fonction qui prend en argument une liste de valeurs numériques et renvoie leur écart-type.
3. Écrire une fonction qui prend en argument deux listes de valeurs numériques, L et P, de même taille, et renvoie la moyenne des valeurs de L pondérées par les valeurs de P (autrement dit la valeur $\frac{1}{n}\sum_{i=0}^{n-1} l_i p_i$ si n est la taille commune des deux listes, les l_i les valeurs dans la liste L et les p_i les valeurs dans la liste P).

◇

Exercice 2. Le langage Python propose nativement une fonction permettant de trier une liste de valeurs numériques (ou plus généralement toute liste dont les valeurs peuvent être comparées deux à deux) : c'est la méthode `sort` (syntaxe `L.sort()` sur une liste `L`). En s'appuyant sur cette fonction, écrire une fonction qui renvoie la médiane d'une liste de valeurs numériques. ◇

Exercice 3.

1. Écrire une fonction qui prend en argument une liste `L` de valeurs numériques et un flottant `seuil` et renvoie le nombre d'éléments de la liste `L` qui sont inférieurs ou égaux à `seuil`.
2. Écrire une fonction `EffectifsCumulesCroissants` qui prend en argument une liste `L` et une liste `S`, et qui renvoie une liste `R` de même taille que `S`, et telle que pour chaque i, `R[i]` soit égal au nombre d'éléments de `L` inférieurs ou égaux à `S[i]`.
3. Visualiser le résultat des lignes de commandes suivantes :

```
>>> from random import randrange
>>> L = [randrange(0,100) for k in range(10**4)]
>>> R = EffectifsCumulesCroissants(L,range(101))
>>> import matplotlib.pyplot as plt
>>> plt.scatter(range(101),R)
```

◇

Thème 5. Suites.

Incontournable ! Les premiers exercices permettent de générer les termes d'une suite, de les stocker dans une liste, d'abord à l'aide d'une boucle `for`, puis, pour la suite de Syracuse, avec une boucle `while`. Les deux derniers exercices nécessitent de gérer manuellement des discrétisations du domaine d'étude.

Exercice 1. On travaille dans un fichier.

1. Écrire une fonction `suite1(n)` qui prend en argument un entier n, et renvoie le terme u_n, où $(u_n)_n$ est la suite définie par $u_0 = 1$, et $u_{n+1} = 3 * u_n/4$ pour tout n.
2. Écrire une fonction `suite2(n)` qui renvoie cette fois-ci la liste de taille $n+1$ contenant les termes $u_0, u_1, \dots, u_n$ de la suite définie à la question précédente.
3. Après avoir exécuté votre fichier, exécutez en console les instructions suivantes :

```
>>> import matplotlib.pyplot as plt
>>> Y = suite2(20) # liste de taille 21, qui va fournir des ordonnées
>>> X = range(20)  # liste de taille 21, qui va fournir des abscisses
>>> plt.scatter(X,Y) # prépare un objet graphique, sans affichage
>>> plt.show()     # déclenche l'affichage
```

◇

Exercice 2. Encore dans un fichier. On considère une suite $(u_n)_n$ définie par un premier terme $u_0 = a$ et une relation de récurrence de la forme $u_{n+1} = f(u_n)$ pour tout n.

1. Écrire une fonction `suite1(a,f,n)` qui prend en argument un flottant `a`, une fonction `f` et un entier positif `n`, et renvoie le terme u_n de la suite proposée ci-dessus.
2. Écrire une fonction `suite2(a,f,n)` qui à partir des mêmes arguments que la précédente renvoie cette fois la liste des termes de u_0 à u_n.
3. Tester à avec quelques jeux de valeurs, par exemple :

$$\begin{cases} a & = 1 \\ f: & t \mapsto \frac{3}{4}t\,, \\ n & = 20 \end{cases} \quad \begin{cases} a & = \frac{1}{2} \\ f: & t \mapsto t^2 - t\,, \\ n & = 20 \end{cases} \quad \begin{cases} a & = 1 \\ f: & t \mapsto \frac{1}{1+t^2}\,, \\ n & = 20 \end{cases} \quad \begin{cases} a & = 10 \\ f: & t \mapsto \ln(t) \\ n & = 5 \end{cases}$$

   La fonction ln est disponible dans la bibliothèque `math` :

```
from math import log
```

4. Écrire des variantes des fonctions `suite1` et `suite2` qui permettent de traiter les relations de récurrence de la forme :

$$u_{n+1} = g(u_n, n).$$

◇

Exercice 3. La suite de Syracuse. Pour $a \in \mathbf{N}^*$, on considère la suite définie par :

$$u_0 = a, \forall n \in \mathbf{N}, u_{n+1} = \begin{cases} u_n/2 \text{ si } u_n \text{ est pair} \\ 3u_n + 1 \end{cases}$$

1. Écrire une fonction `syracuse(a,n)` qui renvoie le terme u_n de la suite de Syracuse initialisée à $u_0 = a$.
2. Écrire une fonction `temps_de_vol(a)` qui calcule des termes de la suite de Syracuse initialisée à $u_0 = a$ jusqu'à obtenir un terme $u_n = 1$ et qui renvoie l'indice de ce terme. On dit que n est le temps de vol associé à a.
3. Représenter graphiquement le temps de vol associé à a en fonction de a, pour a entier compris entre 1 et 100.

◇

Exercice 4. Ensemble de Mandelbrot. Pour tout $c \in \mathbf{C}$, on définit une suite $(z_n)_n$ de nombres complexes par :

$$z_0 = 0, \forall n \in \mathbf{N}, z_{n+1} = z_n^2 + c.$$

On s'intéresse au caractère borné ou non de cette suite en fonction de c. Pour étudier cette question numériquement, on se content d'étudier si un des $N = 20$ premiers termes de la suite $(z_n)_n$ excède ou non $M = 20$ en module.

1. Écrire une fonction `test(c,M,N)` qui prend en argument un complexe `c`, un flottant positif `M` et un entier `N` et renvoie un booléen indiquant si un des N premiers termes de la suite $(z_n)_n$ excède `M` en module (la programmation ne nécessite aucun traitement spécifique aux complexes : les opérateurs d'addition, de multiplication, et la fonction `abs` de calcul de module sont les mêmes pour les flottants et les complexes).

On note $c = x+iy$, avec x et y réels. On souhaite utiliser la fonction test avec `c` qui varie dans le rectangle $[-2, 1] \times [-1, 1]$ (on identifie $\mathbf{C}$ au plan complexe). Il convient de discrétiser ce rectangle, pour n'avoir qu'un nombre fini de points à traiter. Pour cela, on commence par créer une liste de liste, donc chaque valeur représente un point de la grille de discrétisation :

```
>>> Nx, Ny = 150, 100
>>> grille = [[i/50 -2 + 1j*(k/50 -1) for i in range(Nx)] for k in range(Ny)]
```

On remarque que `1j` représente en Python l'unité imaginaire. La variable `grille` fait donc référence à une liste de listes, et l'accès à un élément se fait donc par une instruction de la forme `grille[k][i]`. De même, tout élément peut être modifié par une affectation de la forme `grille[k][i] = valeur`.

2. Écrire un script qui modifie `grille`, en remplaçant chaque valeur `c` par son image par `test(c,20,20)`.
3. Exécuter :

   ```
   >>> import matplotlib.pyplot as plt
   >>> plt.matshow(grille)
   >>> plt.show()
   ```

4. Améliorer ce qui précède en le rendant plus modulaire : le nombre de points de subdivision horizontal et vertical, les valeurs extrêmes de x et y, les valeurs de M et N doivent pouvoir être modifiées.

◇

Exercice 5. Suite logistique. On considère les suites $(u_n)_n$ définies par un premier terme u_0, et une relation de récurrence de la forme $u_{n+1} = au_n(1 - u_n)$ pour a un réel positif fixé.

1. Écrire une fonction `logistique1(u0,a,n)` qui renvoie le terme u_n de la suite logistique de premier terme u_0 et dont la relation de récurrence est définie à l'aide du facteur a.
2. Écrire une fonction `logistique2(u0,a,n)` qui renvoie cette fois la liste des termes de u_0 jusqu'à u_n.
3. Écrire une fonction `extraction(a,Nmin,Nmax)` qui renvoie la liste des termes d'indice compris entre `Nmin` et `Nmax` pour la suite logistique initialisée à $u_0 = \frac{1}{2}$, et dont la relation de récurrence est définie à l'aide du facteur a.
4. Écrire une fonction `discretisation(amin,amax,Na,Nmin,Nmax)` qui calcule la liste `extraction(a,Nmin,Nmax)` pour `a` prenant `Na` valeurs équiréparties entre `amin` et `amax`. La liste de ces listes sera renvoyée.
5. Tracer sur un même graphique les points de coordonnées (a, u_n) pour a prenant 100 valeurs équiréparties entre $2,5$ et 4, et n allant de 30 à 50.

◇

Thème 6. Équations et inéquations.

Exercice 1. On choisit de représenter un trinôme du second degré $P(x) = ax^2 + bx + c$ par la liste `L = [a,b,c]` de ses coefficients.

1. Écrire une fonction `discriminant(P)` qui prend en argument une liste de trois flottants représentant les coefficients d'un trinôme du second degré, et qui renvoie son discriminant.
2. Tester la commande `discriminant([0.25,0.1,0.01]) == 0`. Cela correspond-il au résultat obtenu par un calcul à la main ?
3. Que penser de la fonction suivante ?

```
def racines(P):
    a, b, c = P[0], P[1], P[2]
    Delta = discriminant(P)
    if Delta > 0:
        return([(-b-Delta**0.5)/(2*a),(-b+Delta**0.5)/(2*a)])
    elif Delta == 0:
        return([-b/(2*a)])
    else:
        return([])
```

On pourra la tester avec les listes `[0.25,0.1,0.01]`, `[7,420**0.5,15]`, `[1,10**8,1]`.

◇

Exercice 2.

1. Exécuter les commandes suivantes dans un terminal :

```
>>> a = 3.1
>>> b = a/3
>>> a == 3*b
```

Ceci illustre que la multiplication et la division par 3 ne sont pas des opérations réciproques l'une de l'autre sur les flottants.

2. Écrire une fonction `test_multiplication(a,N)` qui prend en arguments un flottant `a` et un entier `N`, qui répète N fois le test d'égalité entre u et $(u/a) * a$ pour u des flottants générés aléatoirement (fonction `random` du module `random`) et qui renvoie un booléen valant `True` si tous ces tests sont passés, et valant `False` si au moins un test a échoué.
3. Expérimenter afin de former une conjecture sur quels sont les entiers a pour lesquels la multiplication et la division flottantes par a sont réciproques l'une de l'autre.
4. Procéder de même pour l'addition et la soustraction.

◇

Exercice 3.

1. La suite $(u_n)_n$ de premier terme $u_0 = 1$ et satisfaisant la relation de récurrence $u_{n+1} = u_n^{3/2} + \frac{1}{2}$ est croissante, et tend vers $+\infty$. Programmer une fonction `seuil_particulier(A)` qui prend en argument un flottant `A` et renvoie un entier `N` tel que $u_n \geq A$ pour tout $n \geq N$.
2. Généraliser cela en une fonction `seuil(suite,A)` qui renvoie le même résultat, mais pour une suite $(u_n)_n$ croissante tendant vers $+\infty$ quelconque, entendu que l'argument `suite` est une fonction (au sens informatique), qui prend en argument un entier n et renvoie u_n.
3. La fonction précédente est un peu décevante si la suite $(u_n)_n$ peut se calculer par récurrence. Expliquer pourquoi, et modifier la fonction en une fonction `seuil` en une fonction `seuil_bis(iteration,A)`, où l'argument `iteration` est une fonction qui prend en argument un terme de la suite $(u_n)_n$ et renvoie le terme suivant.

4. On considère cette fois f une fonction croissante et qui tend vers $+\infty$. Adapter la fonction `seuil` pour déterminer une approximation de la borne inférieure des réels x pour lesquels $f(x) \geq A$.

◇

Exercice 4. Programmer l'algorithme d'approximation dichotomique du zéro d'une fonction f sur un segment $[a, b]$, sous hypothèse de continuité de f sur $[a, b]$, avec la condition $f(a)f(b) < 0$. On pourra programmer une fonction `dichotomie(f,a,b,eps)`, où l'argument `eps` sera un flottant strictement positif, et qui renverra deux flottants `xmin` et `xmax` satisfaisant aux conditions :

1. `xmin < xmax`, `xmax - xmin < eps`,
2. la fonction f s'annule au moins une fois dans l'intervalle `[xmin,xmax]`.

◇

Thème 7. Tracés avec `turtle`.

La bibliothèque `turtle` permet de disposer d'une fenêtre de tracé, sur laquelle quelques commandes basiques permettent de mimer le comportement d'un crayon. Il s'agit d'un classique en informatique, qui remonte aux années 1960 avec le langage Logo (le crayon était représenté par une tortue). Quelques règles d'utilisation :

— à tout moment le crayon peut être sur la papier (*down*), ou relevé (*up*) ; une couleur lui est attribuée ; ainsi qu'une direction, qui sera celle du prochain tracé ; ces trois attributs peuvent être modifiés ;

— on peut avancer le crayon d'une distance donnée à partir de la position courante, ce qui aboutit à un tracé si le crayon était en position basse ;

— on peut aussi amener directement le crayon à une position donnée, lire la position courante, tracer des arcs de cercle, etc.

Quelques commandes de base (après avoir importé le module `turtle`, `help(turtle.home)` pour obtenir par exemple l'aide de la commande `home`) :

`forward, back, left, right, penup, pendown, color, pos, goto, home, setx, sety, distance, circle`

Exercice 1. Que fait le programme suivant ?

```
import turtle

turtle.colormode(255) # on choisit de représenter les couleurs
                      # en mode (R,G,B), codées sur 255 valeurs
for k in range(10):
    for i in range(3):
        turtle.forward(k*20)   # on trace un trait de longueur 20*k
        turtle.left(120)       # on tourne de 120 degrés vers la gauche
    turtle.color((k,10*k,25*k)) # choix d'une couleur en (R,G,B)
```

◇

Exercice 2. Écrire un programme qui trace dix lignes verticales parallèles, de même longueur, deux à deux équidistantes. ◇

Exercice 3. Écrire un programme qui trace dix cercles concentriques dont les rayons sont en progression arithmétique (non triviale). ◇

Exercice 4.

1. Écrire un programme qui trace dix cercles, dont les rayons et les centres sont choisis aléatoirement (fonction `random` du module `random`).
2. Adapter le programme précédent pour assurer que les cercles sont deux à deux disjoints et qu'aucun n'est inclus dans le disque délimité par un autre.
3. Adapter pour que les cercles soient toujours deux à deux disjoints, mais sans exclure que l'un soit tracé dans le disque délimité par un autre.

◇

Thème 8. Probabilités.

Le module `scipy.stats` propose des modélisations des variables aléatoires usuelles. On propose ici de programmer quelques-unes de ces variables à partir de la fonction `random` du module `random`.

Exercice 1.

1. Écrire une fonction `bernoulli(p)` qui prend en argument un flottant `p` compris entre 0 et 1 et qui modélise une variable aléatoire suivant une loi de Bernoulli de paramètre p (c'est-à-dire qu'elle renvoie 0 ou 1, et la probabilité que 1 soit renvoyée est égale à p - on pourra choisir un flottant (pseudo)-aléatoire avec `random` et le comparer à p).
2. Écrire une fonction `binomiale(n,p)` qui modélise une variable aléatoire suivant la loi binomiale de paramètres (n, p), vue comme le nombre de succès dans une répétition de n épreuves de Bernoulli indépendantes et de même paramètre p.
3. Écrire une fonction `geometrique(p)` qui modélise une variable aléatoire suivant la loi géométrique de paramètre p, vue comme le rang du premier succès dans une répétition d'épreuves de Bernoulli indépendantes et de même paramètre p.
4. Pour chacune des lois programmées, procéder à une répétition de N appels à cette loi, et mettre en place une méthode de comptage des effectifs observés pour chaque issue. Représenter graphiquement avec `matplotlib.pyplot`.

◇

Exercice 2. Reprendre l'exercice précédent avec les lois continues du programme. ◇

Exercice 3. Écrire un programme qui fournit une approximation de π par la méthode de Monte-Carlo : génération d'un grand nombre de couples de coordonnées dans $[0, 1]^2$, et calcul de la proportion de ces points situés dans le quart de disque centré en l'origine et de rayon 1. ◇

Exercice 4. Écrire un programme qui simule l'expérience de l'aiguille de Buffon. On pourra considérer que les jointures entre les lames du parquet sont les droites d'équation $x = k$, pour $k \in \mathbf{Z}$, qu'une extrémité de l'aiguille est située en un point de la forme $(x, 0)$, pour $x \in [0, 1[$, générer aléatoirement ce réel x, ainsi que l'angle $\theta \in]-\pi/2, \pi]$ que fait l'aiguille avec l'horizontal. Il ne reste ainsi plus qu'à tester l'existence d'une intersection entre l'aiguille et une droite. ◇

Thème 9. Arithmétique.

Exercice 1. Programmer des fonctions de conversion d'un entier entre écritures binaire et décimale (on pourra créer des listes de chiffres). ◇

Exercice 2. Écrire une fonction prenant en argument deux entiers a et b, avec b non nul, et qui renvoie le couple (q, r) constitué du quotient et du reste de la division euclidienne de a par b.

On pourra comparer l'algorithme naïf (effectuer des soustractions par b successives jusqu'à obtention du reste), et l'algorithme dichotomique (on cherche d'abord i tel que $2^i b \leq a < 2^{i+1}b$, puis on cherche q par dichotomie entre 2^i et 2^{i+1}). ◇

Exercice 3. Programmer une exponentiation rapide.

On rappelle le principe de cet algorithme. Il s'agit de calculer x^n pour x flottant et n entier. L'algorithme naïf nécessite $n-1$ multiplications. On peut faire beaucoup mieux. On considère l'écriture binaire $n = \sum_{i=0}^{d} a_i 2^i$, avec $a_i \in \{0, 1\}$, pour tout i, de sorte que d est de l'ordre du logarithme de n. En termes de puissances de x, cela s'écrit :

$$x = \prod_{i=0}^{d} (x^{2^i})^{a_i}.$$

On en déduit l'algorithme :

- Initialisations : `puiss = 1`, `X = x`, `N = n`.
- Traitement : tant que `N` est non nul, faire :
  - calculer `a` le reste de la division euclidienne de `N` par 2, et stocker dans `N` le quotient `(N-a)//2` ;
  - stocker dans `puiss` le produit de `puiss` par `X**a` (puisque `a` vaut 0 ou 1, cela ne nécessite pas de calcul de puissance) ;
  - élever `X` au carré.
- Sortie : la valeur cherchée est stockée dans `puiss`.

◇

Exercice 4.

1. Programmer l'algorithme d'Euclide de calcul du pgcd.
2. Programmer l'algorithme d'Euclide étendu aux coefficients de Bézout.

◇

Exercice 5. Programmer un crible d'Eratosthène. ◇

Exercice 6.

1. Programmer un test de primalité de Fermat : soit n l'entier à tester, on choisit un entier a au hasard entre 2 et $n-1$ (on dit que a est une « base »), si a et n ne sont pas premiers entre eux ou si a^{n-1} n'est pas congru à 1 modulo n, on déclare que n est composé et que a est un témoin de non primalité de n, sinon, n est déclaré pseudo-premier en base a. On pourra choisir une valeur de sortie booléenne.
2. Pour un entier n fixé, faire une estimation de la proportion de témoin de non primalité parmi les bases possibles. Tracer un graphe de cette proportion.

◇

Exercice 7. Reprendre l'exercice précédent avec le test de primalité de Miller-Rabin, et comparer les graphes de proportion des témoins de non primalité.

On rappelle que le test de primalité de Miller-Rabin repose sur le constat que si n est premier, alors l'anneau $\mathbf{Z}/n\mathbf{Z}$ est un corps, de sorte que si a est premier à n, non seulement a^{n-1} est congru à 1 modulo n, mais de plus, par extraction de racines carrées successives, soit tous les $a^{(n-1)/2^i}$, tant que 2^i divise $n-1$, sont congrus à 1 modulo n, soit l'un de ces nombres est congru à -1. Pour mettre en place ce test, on peut procéder comme suit :

— par division consécutives par 2, calculer f entier et m impair tels que $n-1=2^f m$;
— on considère les a^{m2^i} modulo n pour i allant de 0 à f :
  — si a^m est congru à 1 modulo n, alors n est pseudo-premier en base a ;
  — si la valeur -1 est rencontrée avant la valeur 1 et pour $i \leq f-1$, alors n est pseudo-premier en base a ;
  — sinon, n est composé, et a est un témoin de non primalité.

◇

Exercice 8. Écrire une fonction prenant en argument un entier n, et renvoyant sa décomposition en facteurs premiers (qui pourra être écrite comme une liste de couples (p, m), où p parcourt les diviseurs premiers de n, et pour tout p, m est la valuation p-adique de n).

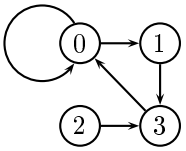
Parvenez-vous à ce que le temps d'exécution soit polynomial en $log(n)$? ◇

Thème 10. Graphes.

Exercice 1. On considère un graphe à n sommets, numérotés de 0 à $n-1$. On représente un graphe (orienté et non pondéré) par une liste de listes `G`, de la manière suivante : pour deux sommets `i` et `j`, le coefficient `G[i][j]` vaut 0 s'il n'y a pas d'arête de `i` vers `j`, et 1 s'il y en a une. Par exemple, la liste :

```
G = [[1,1,0,0],[0,0,0,1],[0,0,0,1],[1,0,0,0]]
```

représente le graphe :



1. Écrire une fonction `voisins(G,i)` prenant en argument une liste de listes qui représente un graphe, et le numéro `i` d'un sommet, et renvoyant la liste des numéros des sommets qui sont un but d'une arête de source `i`.
2. Écrire une fonction `exploration(G,i)` de mêmes argument que la fonction `voisins`, et qui renvoie une liste dont l'élément en position `d` est la liste des sommets `j` tels que le chemin minimal de `i` à `j` comprenne `d` arêtes. Par exemple, sur le graphe précédent avec `i = 0`, la fonction renvoie `[[0],[1],[3]]`.
3. On suppose désormais le graphe non orienté, ou, ce qui revient au même, pour tous sommets `i` et `j`, s'il existe une arête de `i` vers `j`, alors il en existe une de `j` vers `i`. Écrire une fonction `composantes(G)` qui renvoie la liste des composantes connexes de `G` (chaque composante est représentée par la liste des sommets qui la composent).

◇

Exercice 2. On considère un graphe à n sommets, numérotés de 0 à $n-1$. On considère que le graphe est connexe, pondéré et non orienté. L'objectif est de chercher le chemin de longueur minimale d'un sommet à un autre (un chemin est une suite d'arêtes telle que deux arêtes consécutives de la suite ont un sommet en commun, sa longueur étant la somme des poids des arêtes qui le composent).

Le graphe est représenté en mémoire par une liste de longueur n, l'élément en position i fournissant une information sur les sommets connectés au sommet i sous forme d'une liste de couples (j, d), l'indice j représentant un sommet connecté au sommet i, et d le poids de l'arête correspondante.

On considère pour cela l'algorithme de Dijkstra :

— Données : un graphe représenté par une liste L, deux indices i et j compris entre 0 et la longueur de L moins 1.

— Variables locales : une liste `distances` de longueur n ; on initialise `distances[i]` à 0, et tous les autres éléments à `float("inf")` (pour $+\infty$) ; une deuxième liste `visites` de longueur n : on initialise tous ses éléments à `False`.

— Tant que le sommet `j` n'a pas été visité, faire :

  — choisir le sommet `k` tel que `distances[k]` est minimal ;

  — pour chaque sommet `l` tel qu'il existe une arête entre `k` et `l`, faire :

    — si la somme `d[k] + poids(k,l)` est strictement inférieure à `d[l]`, affecter à `d[l]` cette somme ;

  — passer `visites[k]` à `True`.

1. Programmer cet algorithme.
2. Permet-il de trouver le chemin le plus court entre les sommets considérés ? Y remédier.

L'utilisation de deux listes comme suggéré ici ne permet pas de programmer l'algorithme de Dijkstra de la manière la plus efficace. D'autres structures de données sont plus adaptées. ◇

Thème 11. Chaînes de caractères.

Manque de temps pour écrire une section spécifique. Voici quelques pistes d'activité :

— tester si un caractère apparaît dans une chaîne ; compter le nombre d'occurrences ;

— classer les lettres par fréquence dans une chaîne ;

— tester la présence d'un motif dans une chaîne de caractères ;

— ouvrir un fichier texte en mode lecture avec la commande `open`, et le parcourir. Voici une trame :

```
fichier = open('nom_du_fichier','r')

for ligne in fichier:
    traitement
```

— ouvrir un fichier `csv` afin de traiter des données numériques.