Académie de Rouen

# Programmer en lycée avec Python

## Éléments de formation à destination des enseignants de mathématiques

Pôle de compétences élargi de mathématiques

Novembre 2017

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
Région académique
NORMANDIE

mathématiques
académie de Rouen

# Préambule

Ce livret a été élaboré par des professeurs de mathématiques de l'académie de Rouen, et diffusé lors des journées de formation de l'automne 2017. Il a pour objectif d'accompagner les professeurs de lycée dans la pratique de l'algorithmique et de la programmation, dans le cadre des aménagements des programmes officiels de seconde générale et technologique de la rentrée 2017.

Le document d'accompagnement Algorithmique et programmation, publié au printemps 2017, fait le choix de Python comme support à l'apprentissage de la programmation en lycée général et technologique :

« *Le choix d'un langage textuel, comme Python, au lieu d'un langage par blocs, comme Scratch, permet aux élèves de se confronter à la précision et la rigidité d'une syntaxe proche de celle des expressions mathématiques, avec l'avantage de pouvoir bénéficier du contrôle apporté par l'analyseur syntaxique. [...]*
*Les programmes de ce document sont écrits dans le langage Python, choisi pour la concision et la simplicité de sa syntaxe, la taille de la communauté d'utilisateurs (en particulier dans le cadre éducatif), ainsi que la richesse des ressources disponibles.* »

L'ambition du présent livret est plurielle. Sa première partie accompagne le professeur débutant dans la découverte du langage Python, par des exemples simples, et le principe de l'essai-erreur. Viennent ensuite des applications simples et directes en classes de lycée, en privilégiant le niveau seconde, puis des scénarios plus complexes. La partie intitulée « Peaux de bananes » pointe des erreurs courantes en Python, afin que les enseignants les évitent. Dans le même esprit, un glossaire et des compléments sur les bibliothèques de Python peuvent servir de références, une fois les fondamentaux acquis. Le livret se clôture par un memento et les corrections des exercices de la première partie.

Ce livret, ainsi que ses ressources numériques (programmes rédigés en Python, aides diverses), sont disponibles sur le site académique de mathématiques : maths.spip.ac-rouen.fr, rubrique *Dans la classe / Ressources lycée*.

# Table des matières

# Python : prise en main

Python est un langage de programmation, couramment utilisé par les développeurs informatiques. Beaucoup de sites Internet, par exemple, sont aujourd'hui développés en Python.
Pas de « langage naturel » ici : la syntaxe des commandes et la structure des programmes nécessitent un apprentissage rigoureux.
Cependant, il s'agit toujours de mettre en œuvre des algorithmes…

## A) EduPython : un éditeur parmi d'autres

Pour écrire un programme en Python, un simple éditeur de texte (comme **Notepad**) suffit.
Mais il est plus pratique d'avoir un **environnement intégré**, où le programme et le résultat de son exécution s'affichent dans la même fenêtre.
Pour la présente formation, le choix s'est porté sur **EduPython**, développé par l'académie d'Amiens, pour les raisons suivantes :


- une application Windows autonome, qui peut même s'installer sur un support externe ou un partage réseau ; lien de téléchargement : edupython.tuxfamily.org ;
- Python en version 3 et toutes les bibliothèques nécessaires à une utilisation en lycée ;
- un éditeur intégré ;
- une installation possible sous Linux avec Wine.

Le présent livret et le site académique de mathématiques (maths.spip.ac-rouen.fr) fournissent plus de détails sur :

- la notion de bibliothèque (page 33), avec la description des bibliothèques les plus courantes ;
- les différents choix d'éditeurs (Idle, Jupyter Notebook, Pyzo…) ;
- les installations sur différents systèmes d'exploitation : Windows, Linux, Mac OS.

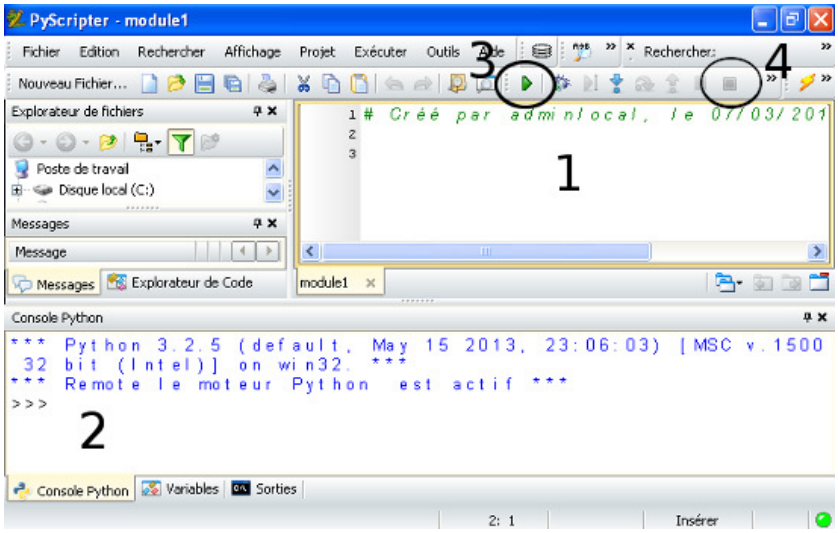
## B) L'interface d'EduPython

**Fenêtre 1 :** fenêtre de script.
C'est là qu'est écrit le programme.

**Fenêtre 2 :** interpréteur ou console.
Ici va s'exécuter le programme et vont se signaler les éventuelles erreurs.
On peut aussi y écrire des instructions simples, utilisant la console comme une calculatrice.

**Bouton 3 :** pour lancer le programme.

**Bouton 4 :** arrêt d'urgence.



# C) Premiers programmes

Conformément au document d'accompagnement Algorithmique et programmation, le présent livret prend le parti de présenter la programmation en Python sous son angle **fonctionnel** : il s'agit de définir des fonctions, qui seront ensuite soit exécutées dans la console, soit réemployées dans des programmes plus complexes, dans d'autres fonctions.
Les classiques instructions d'entrée / sortie n'apparaissent qu'occasionnellement ; elles sont décrites dans le Glossaire, page 30.

## 1. Notion de fonction

1. Dans la fenêtre de script, recopier et exécuter le programme ci-contre. Prendre garde à respecter les espaces en début de ligne (l'indentation).

   À quoi peut servir ce programme ?

```
def discr(a,b,c):
    return b**2-4*a*c

discr(5,3,1)
```

2. Effacer la dernière ligne du programme ; mettre à la place, en s'inspirant de la définition de la fonction `discr`, une fonction `x_sommet` prenant pour paramètres trois nombres `a`, `b` et `c` et renvoyant l'abscisse du sommet de la parabole d'équation $y = ax^2 + bx + c$.
   Exécuter le programme. Rien ne devrait s'afficher ; pourquoi ?

3. Dans la console (voir présentation de l'éditeur), écrire : `discr(5,3,1)` et valider ; puis `x_sommet(5,3,1)` et valider. Recommencer cette étape en variant les paramètres.

À retenir

Une **fonction** telle que `discr` :
- prend un certain nombre de paramètres (ici `a`, `b` et `c`), éventuellement aucun ;
- renvoie, via `return`, un ou plusieurs résultats.

L'objectif est que ces résultats soient exploitables par une autre fonction, ou un programme qui aurait par exemple pour but de les afficher.
Le `return` termine la fonction : les éventuelles instructions suivantes ne sont pas exécutées.

## 2. Appeler une fonction ; instruction conditionnelle

Retour à la fenêtre de script. À la suite des fonctions déjà mises en place, écrire la fonction ci-contre, en la modifiant et en la complétant de façon logique ; la fonction retournera une **chaîne de caractères**.
Exécuter le programme, puis tester la nouvelle fonction dans la console.

```
def nombre_racines_trinome(a,b,c):
    if discr(a,b,c)<0 :
        message = "Premier message"
    if (...) : ( ou else: )
        ...
    return message
```

Instructions utiles

- L'instruction `si ... alors ... sinon ...` se code comme ci-contre.
  Il n'y a pas de `Fin_si`. Remarquer l'indentation des lignes après les « deux points ».

```
if (condition):
    ...
else:
    ...
```

- Opérateurs de comparaison : `==` `!=` `<` `>` `<=` `>=`
- Opérateurs logiques : `and` `or`
- Concaténer deux chaînes de caractères : essayer le code ci-contre dans la console.

```
a = "bonjour "
b = "monde"
c = 4
a+b
a+str(c)
```

À retenir

L'indentation, obtenue avec la touche de tabulation ou avec des espaces, est **primordiale** : tout ce qui est indenté après le `if ()` : sera exécuté comme un bloc. Il ne faut pas que l'indentation varie (nombre d'espaces, passer de la tabulation à des espaces...) en cours de bloc.
Une fonction « retourne » aussi bien un résultat numérique qu'une chaîne de caractères.

### 3. Utiliser des bibliothèques

Modifier la fonction précédente, ou en créer une autre, de sorte que soit résolue dans $\mathbb{R}$ l'équation du second degré :

$$ax^2 + bx + c = 0$$

À retenir

La racine carrée s'écrit `sqrt()`, mais n'existe pas dans les instructions de base de Python. Il faut ajouter une *bibliothèque* spécifique, en début de programme.
Ces bibliothèques sont simplement des catalogues de fonctions. La racine carrée est définie dans la bibliothèque `math`, que l'on charge en écrivant en début de script :

```
from math import *
```

La description des bibliothèques les plus courantes est visible dans la partie Bibliothèques existantes, page 33. Différents exemples sont abordés dans les autres parties du présent livret.

### 4. Boucles Pour et Tant_que

Définitions

**Syntaxe boucle Pour**

```
for compteur in range(1,10):
    ...
```

La variable `compteur` va parcourir la liste des valeurs contenues dans `range(1,10)`. Cette liste est : [1,2,3,...,9] : elle s'arrête juste avant 10.
Autre syntaxe : avec `range(10)`, la variable `compteur` parcourt la liste [0,1,2,...,9] : cette liste a 10 valeurs.

Plus d'exemples dans le Memento, page 35.

**Syntaxe boucle Tant_que**

```
while (condition):
    ...
```

L'indentation a la même importance ici que pour le `if` ; elle définit les "blocs" d'instructions qui s'exécutent à l'intérieur des boucles.

Voir le glossaire, page 30, pour plus de détails.

**Application 1**

1. Recopier et compléter la fonction `fibo` ci-contre, de sorte qu'elle renvoie, sous forme de liste, les $k$ premiers termes de la suite $(u_n)$ définie par :
   $$u_0 = 1, \quad u_1 = 1 \quad \text{et} \quad u_{n+2} = u_n + u_{n+1}$$
2. Modifier cette fonction de sorte qu'elle prenne pour arguments, outre le nombre de termes de la suite à afficher, les valeurs de $u_0$ et $u_1$.

```
def fibo(k):
    # Initialisation de la liste
    L = [1,1]
    for compteur in range ( ... , ... ):
        ...
    return L
```

Instructions utiles

Si `L` est la liste `[1,1,2,3]`, l'instruction `L.append(5)` ajoute le nombre 5 au bout de cette liste. `L` contient alors : `[1,1,2,3,5]`.

Le terme de rang `i` de la liste `L` est : `L[i]`.

L'indexation commence toujours à 0 : avec `L = [1,1,2,3]`, l'élément `L[0]` contient 1.

**Application 2**

La suite $(u_n)$ est définie par $u_0 = 7$ et, pour tout $n$ entier : $u_{n+1} = \frac{1}{2}\left(u_n + \frac{5}{u_n}\right)$.

On peut prouver que cette suite est décroissante et converge vers $\sqrt{5}$.
Créer une fonction qui calcule et renvoie la liste de tous les termes de $(u_n)$, jusqu'à obtenir une approximation à $10^{-6}$ près de la limite.
Améliorations possibles : la fonction pourrait s'adapter à n'importe quel nombre $a$ dont on cherche une approximation de la racine carrée ; d'autres paramètres pourraient être le seuil, le nombre de départ...

# D) Suite de Syracuse et représentations graphiques

Une suite de Syracuse est définie comme suit :
- Choisir un nombre entier strictement supérieur à 1.
- Si ce nombre est pair, le diviser par 2 ; sinon, le multiplier par 3 et ajouter 1.
- Recommencer avec le nouveau nombre.

**Conjecture** : quel que soit le nombre choisi au départ, la suite atteint le nombre 1.

## 1. Suite de Syracuse : calcul des termes

Programmer une fonction `syracuse` qui :
- prenne comme argument un entier, premier terme de la suite ;
- calcule les termes de la suite de Syracuse associée, avec comme condition d'arrêt le fait d'arriver au nombre 1 ;
- renvoie la liste des valeurs ainsi trouvées.

Instructions utiles

**Opérations sur les entiers**
- `a//b` donne le quotient de la division euclidienne de `a` par `b` : `11//4` donne 2
- `a%b` donne le reste de la division euclidienne de `a` par `b` : `11%4` donne 3

**Remarque**

Une fois la liste obtenue, on peut calculer sa longueur : `len(L)`

## 2. Représentation graphique d'une suite de Syracuse

L'objectif de cette partie est de représenter graphiquement, dans un repère, les points de coordonnées $(n; u_n)$ d'une suite de Syracuse $(u_n)$ — du moins jusqu'à la première occurrence de 1.

1. Programmer une fonction `graph_syracuse` :
   - qui prenne comme argument un entier, premier terme de la suite ;
   - qui fasse appel à la fonction `syracuse` créée précédemment ;
   - qui trace la représentation graphique de la suite.
2. Tracer deux suites de Syracuse, issues de deux entiers différents, sur le même graphique.
3. **Défi (optionnel) :** représenter sur un même graphique les suites de Syracuse issues des nombres 1 à 100.

Instructions utiles

- La bibliothèque graphique à charger en préambule est `matplotlib.pyplot`. Pour plus de confort de codage ensuite, il est d'usage de la renommer, par exemple en `plt`. L'instruction de chargement de cette bibliothèque est alors : `import matplotlib.pyplot as plt`
- L'instruction de **tracé de points** est : `plt.plot(X_liste , Y_liste , "paramètres")`
  Dans cette instruction :
  - `X_liste` est la liste des abscisses des points à tracer ;
  - `Y_liste` est la liste des ordonnées des points à tracer ;
  - `"paramètres"` (ne pas oublier les guillemets) définit le type de tracé : pour des petits points tracés en bleu, on écrira `"b."` ; pour des étoiles en rouge, on écrira `"r*"`. Une liste plus complète est disponible dans le Memento, page 35.

  Le couple de deux listes (`X_liste , Y_liste`) peut être considéré comme un tableau de valeurs. Il faut donc que ces deux listes aient le même nombre d'éléments.
  Mais pourquoi ne pas tracer un graphique point par point, comme sous Algobox ? L'explication est donnée dans la « peau de banane » Rapidité de tracé de graphiques, page 29.
- Par défaut, la fenêtre graphique est calibrée automatiquement ; des réglages plus fins sont listés dans le Memento, page 35.
- Python a créé un graphique... Mais il faut lui demander de l'afficher, à l'aide de l'instruction (sans argument) : `plt.show()`
  Cette instruction peut être donnée dans la console, après exécution d'une fonction.

# Premières applications de Python

## A) Python dès la seconde

### 1. Applications rapides en géométrie repérée

1. Que fait la fonction définie ci-contre ? Quel nom lui donner ?
   Remarquer que le résultat renvoyé est une liste, contenant deux nombres.

```
def ......... (xA,yA,xB,yB) :
    return [ (xA+xB)/2, (yA+yB)/2 ]
```

2. À la suite de la fonction évoquée ci-dessus, rédiger une nouvelle fonction :
   - qui prenne pour arguments les coordonnées des quatre sommets d'un quadrilatère $ABCD$ ;
   - qui appelle la fonction précédente ;
   - qui renvoie selon le cas le message « $ABCD$ est un parallélogramme » ou « $ABCD$ n'est pas un parallélogramme. »

Autres idées de programmes

- Coordonnées du quatrième sommet d'un parallélogramme.
- Droites parallèles ou sécantes.
- Coordonnées du point d'intersection de deux droites sécantes.
- Coordonnées d'un vecteur.
- Coordonnées de la somme de deux vecteurs.
- Coordonnées du produit d'un vecteur par un réel.
- Colinéarité de deux vecteurs.
- Translation…

### 2. Déterminer l'équation d'une droite passant par deux points

1. Rédiger une fonction `coeff` qui prenne comme arguments les coordonnées des deux points, et qui renvoie le coefficient directeur de la droite passant par ces deux points. Le cas de l'égalité entre les abscisses sera soit ignoré, soit traité d'une façon particulière, par exemple en retournant une chaîne de caractères.

2. Rédiger une fonction `ord_origine` qui prenne comme arguments les coordonnées d'un point et un nombre `a`, et qui renvoie l'ordonnée à l'origine de la droite de coefficient directeur `a` et passant par ce point.

3. Rédiger une fonction `equation` qui renvoie une chaîne de caractères : l'équation de la droite passant par les deux points dont les coordonnées ont été entrées en arguments. Cette fonction utilise les deux fonctions précédentes.

### 3. Petit jeu « Devine le nombre auquel je pense »

Un utilisateur doit trouver un nombre entier compris entre 1 et un entier $n$.

Instructions utiles

- La bibliothèque `random` est nécessaire : `from random import *`
- L'instruction `randint(a , b)` renvoie un entier aléatoire entre les entiers `a` et `b` inclus.
- Pour obtenir un flottant aléatoire dans $[0;1[$ : `random()`
- Dans ce contexte, les fonctions `input()` et `print()` sont nécessaires, pour interagir avec l'utilisateur : voir Glossaire, page 30.

1. Compléter le code ci-dessous, puis tester le programme dans la console.
2. Améliorer le programme en y ajoutant un compteur du nombre d'étapes réalisées.

```
from random import *

def jeu(n):
    nb=randint(1,n)
    ch=nb+1 # Quel est le rôle de cette initialisation ?
    print("Je pense à un nombre entier entre 1 et ",n ,", à vous de deviner lequel...")
    while nb!=ch:
        # La proposition devra être un entier.
        # Ainsi, afin de convertir la chaîne de caractères entrée en un entier,
        # on utilise la fonction int().
        ch=int(input("Votre proposition ?"))
        if ......... :
            print(".........")
        elif ......... :
            print(".........")
    return "Bravo, le nombre était " + str(ch)
```

Remarque

Un `elif` n'est pas un `else`, ni un `if`.
Sa traduction en Français serait : « sinon, si… » : la condition n'est testée que si la première condition (celle du `if` de départ) n'est pas vérifiée.

**Défi : une variante**

Écrire le programme qui fait l'inverse : l'utilisateur pense à un nombre entre 1 et 1000, et l'ordinateur propose successivement des valeurs afin de se rapprocher du nombre.

Pistes possibles

- Établir une stratégie, par exemple la dichotomie, et la coder.
- Prévoir deux variables, par exemple `minimum` et `maximum`, pour faire évoluer l'encadrement.

## 4. Statistiques

### Indicateurs de position et de dispersion

1. Inventer une série statistique avec une dizaine de données, non ordonnées ; puis, à l'aide de la console, stocker cette série dans une liste. Dans la suite de cette activité, cette variable sera appelée `serie`.

2. Programmer une fonction `moyenne` qui, appliquée à `serie`, renvoie la moyenne de la série statistique.

Instructions utiles

- Longueur d'une liste `L` : `len(L)`
- Élément d'indice `i` d'une liste `L` : `L[i]`
- La numérotation d'une liste démarre toujours à 0.
- Une boucle `for` peut s'appliquer à n'importe quelle liste, même non numérique :
  `for curseur in ["Lundi", "Mardi", "Mercredi"]`

3. Programmer une fonction `quartiles` qui, appliquée à `serie`, renvoie sa médiane et ses premier et troisième quartiles.

Instructions utiles

- Trier une liste `L` : `L.sort()`
- Trier une liste `L` et stocker le résultat dans une liste `L_triee` : `L_triee = sorted(L)`
  Différence par rapport à `L.sort()` : la liste `L` d'origine n'est pas modifiée.
- Division entière de `a` par `b` : `a//b`
- Retourner plusieurs résultats sous forme d'une liste : `return [..., ..., ...]`

### Échantillonnage

1. Définir, dans la console, une liste pouvant représenter les tirages d'un dé truqué (par exemple un dé cubique ayant 2 faces « trois »). Cette liste sera nommée `univers` dans la suite.

2. Toujours dans la console, importer la bibliothèque random, puis tester l'instruction : `choice(univers)`
   Que fait cette instruction ?

3. Programmer une fonction `echantillon` :
   - qui prenne pour paramètres une liste de données `univers` et un entier `n` ;
   - qui renvoie une liste, échantillon de taille `n` d'éléments de `univers`.

Instructions utiles

- Importer la bibliothèque `random` : `from random import *`
- Tirer au hasard un élément de la liste `L` : `choice(L)`
- Ajouter un élément `a` à la fin de la liste `L` : `L.append(a)`

Autres idées de programmes

- Calcul d'une moyenne pondérée, avec deux listes : une pour les données, l'autre pour les effectifs.
- Fréquence d'apparition d'un résultat dans un échantillon.
- Stabilisation des fréquences, avec représentation graphique.
- Intervalle de fluctuation au seuil 0,95 : calcul des bornes de l'intervalle, nuage de points, comptage sur une simulation…

Le document d'accompagnement Algorithmique et programmation présente plusieurs de ces programmes.

### 5. Instruction conditionnelle

Programmer, en s'aidant de la structure ci-dessous :

1. une fonction `pgcd`, de paramètres `a` et `b`, qui calcule le plus grand commun diviseur des deux entiers $a$ et $b$ par l'algorithme des différences ;
2. une fonction `irreductible`, de paramètres `num` et `denum`, qui réduit la fraction $\frac{\texttt{num}}{\texttt{denum}}$.

```
def pgcd(a,b):
    while .......... :    # a différent de b
        if a>b :
            a = a-b
        else :
            b = b-a
    return a

def irreductible(num,denum):
    # Appel de la fonction pgcd
    d = pgcd(num,denum)
    if .......... :
        message = "la fraction est irréductible"
    else:
        num2 = ..........   # Division entière
        denum2 = ..........
        message = ".........." + str(num2) + ...
    return message
```

**Note**

Une fonction peut retourner une chaîne de caractères.
Dans la console, en écrivant `irreductible(45,85)`, le résultat devrait ressembler à :

```
La fraction 45/85
se simplifie en 9/17
```

Instructions utiles

- Concaténation de deux chaînes : `chaîne1 + chaîne2`
- Transformation d'un nombre `a` en chaîne de caractères : `str(a)`

Remarque

Selon les besoins, on peut modifier la fonction de sorte qu'elle renvoie :

- une chaîne de caractères comme ci-dessus ;
- le numérateur et le dénominateur, dans une liste ;
- un simple booléen (1 ou 0), selon que la fraction est irréductible ou pas.

Les deux dernières options permettent de réutiliser le résultat dans une autre fonction.

### 6. Arithmétique en seconde : *pgcd* et théorème de Césaro

Dans la multitude de scénarios envisageables en classe, on peut commencer par des séances d'algorithmique et programmation pour construire une fonction `pgcd`. À l'aide de cette fonction, il devient possible d'examiner le problème suivant.

**Problème :** un couple $(a;b)$ de deux nombres entiers compris entre 1 et 100, éventuellement égaux, est choisi aléatoirement. On s'intéresse à l'événement : « $a$ et $b$ sont premiers entre eux ».

**Travail demandé**

1. Rédiger une fonction `simulation_cesaro` qui simule la situation décrite ci-dessus $n$ fois ($n$ étant entré en paramètre), et qui donne en sortie la fréquence de couples premiers entre eux obtenus.

   Instructions utiles

   - Une fonction `pgcd` a déjà été définie dans l'activité précédente ; rien n'empêche d'en créer une autre, par exemple utilisant l'algorithme d'Euclide.
   - Le chargement de la bibliothèque `random` et la fonction `randint` ont déjà été décrits dans Petit jeu « Devine le nombre auquel je pense », page 9.

2. Calculer la probabilité de l'événement à l'aide d'une autre fonction, qui explore tous les tirages possibles.

# B) Python en prolongement de Scratch et de la calculatrice

## 1. La Tortue Python en prolongement de Scratch

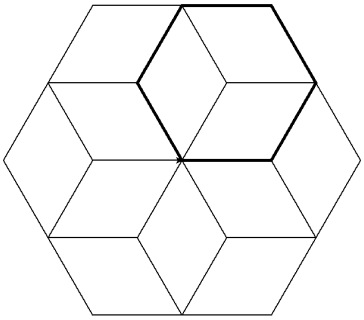
Python possède un module appelé `turtle` permettant de tracer des figures. Les élèves de collège sont déjà familiarisés avec ce type de programmation. Ils ont découvert les commandes de déplacement et de tracé grâce au langage Scratch. Les notions de boucle et de variable ont souvent été introduites au travers de la progammation de dessins sur un écran.
Le Wikibook Turtle détaille de nombreuses instructions. On peut par exemple manipuler plusieurs tortues…


Le programme suivant permet de tracer un triangle équilatéral avec la Tortue Python, sur le modèle du programme Scratch ci-contre.

```
from turtle import *

def triangle_equi(longueur):
    pensize(2)
    up()
    clear()
    setheading(0)    # Pointe la tortue vers la droite
    goto(-100,-100)
    down()
    for i in range(3):
        forward(longueur) # La tortue avance de longueur
        left(120) # La tortue tourne à gauche de 120°
    up()
```

1. Adapter ce programme en Python afin de créer une fonction `poly_reg` qui trace un polygone régulier. Cette fonction aura deux arguments : le nombre de côtés et la longueur d'un côté.

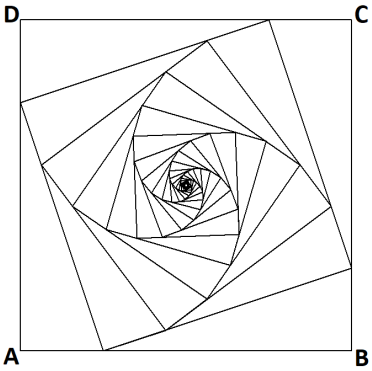
2. On désire tracer la figure ci-contre. Elle est constituée de six hexagones réguliers. Utiliser la fonction `poly_reg` pour créer une nouvelle fonction `rosace` permettant de tracer cette figure. Elle aura pour argument la longueur d'un côté d'un hexagone.



3. **Pour aller plus loin**

   $ABCD$ est un carré de côté 600. Chaque nouveau carré est construit en déplaçant chaque sommet d'un quart de la longueur de son côté.
   Écrire un programme faisant apparaître la figure ci-contre.

   Prolongement possible : une fourmi part du sommet $A$ et parcourt un quart du côté $[AB]$. Puis elle continue sur un quart du côté du nouveau carré. Et ainsi de suite.
   Calculer la longueur du chemin parcouru par la fourmi ; afficher le résultat sur la figure.




Instructions utiles et aide

- **Aide :** si `u` est une liste de deux nombres contenant les coordonnées d'un point (`u[0]` étant l'abscisse et `u[1]` l'ordonnée), que retourne la fonction ci-dessous ?

```
def prochain_point(u,v):
    return [u[0]+(v[0]-u[0])/4, u[1]+(v[1]-u[1])/4]
```

- `write(n)` écrit le contenu de la variable `n` à la position de la tortue.

## 2. Autour des courbes de fonctions

### Exploration des fonctions de variable réelle avec Python

1. Un peu de mathématiques : créer une fonction de variable réelle $x$, à valeurs dans $\mathbb{R}$.
   Implémenter cette fonction mathématique en une fonction Python en complétant le code ci-contre. Vérifier le fonctionnement de la fonction `ma_fonction` dans la console, en calculant quelques images.

```
def ma_fonction(x):
    return ...
```

Bibliothèques

- Pour certaines fonctions comme `sqrt` (racine carrée), `cos`, `exp`, `log`..., il est nécessaire d'importer la bibliothèque `math` : `from math import *`
- Pour la suite de cette activité, ne pas oublier de charger la bibliothèque graphique : `import matplotlib.pyplot as plt`

2. Tester alors les deux fonctions ci-dessous, et comparer leur action en les appliquant à `ma_fonction`.

```
def courbe(f,a,b):
    X = []
    Y = []
    for k in range(a,b+1) :
        X.append(k)
        Y.append(f(k))
    plt.plot(X,Y,"b-")
    plt.show()
```

```
def courbe_pas(f,a,b,pas):
    X = []
    Y = []
    while a < b :
        X.append(a)
        Y.append(f(a))
        a = a + pas
    plt.plot(X,Y,"b-")
    plt.show()
```

### Application à la parabole

**Contexte :** en seconde, une activité de recherche a été menée au préalable pour établir l'abscisse du sommet d'une parabole.

1. Rédiger une fonction :

   `abscisse_minmax`

   qui prenne pour arguments les coefficients `a`,`b` et `c` d'un trinôme du second degré, et qui renvoie l'abscisse du sommet de la parabole associée.
2. Rédiger une fonction :

   `image`

   de paramètres `a`,`b`,`c` et `x`, dont la signification semble évidente...
   Il est possible également d'utiliser la structure vue au début de la présente page.
3. Compléter le code de la fonction :

   `tracer_courbe`

   ci-contre, qui a pour but de tracer la parabole représentant le trinôme $ax^2 + bx + c$, sur un intervalle d'amplitude 10 centré sur l'abscisse du sommet, et avec un pas de pas ; le sommet de la parabole est marqué par un point bleu.

**Code à compléter (question 3)**

```
import matplotlib.pyplot as plt

def tracer_courbe(a,b,c,pas):
    # Calcul des coordonnées du min/max
    # pour définir un intervalle l'incluant
    x_minmax = ..........
    y_minmax = ..........
    debut = x_minmax-5
    fin = x_minmax + 5

    X_liste = []
    Y_liste = []
    curseur = debut
    while curseur < ..... :
        X_liste.append(..............)
        Y_liste.append(..............)
        curseur = ............

    plt.plot(..... , ........ , "r-")

    # Un point bleu pour le sommet
    plt.plot(x_minmax , y_minmax , "b.")
    plt.grid()   # Affichage de la grille
    plt.show()
```

Autres idées de programmes

- Calcul d'un tableau de valeurs d'une fonction, éventuellement définie par morceaux.
- Nuage de points.
- Calcul approché de la longueur d'un arc de courbe.
- Résolution d'équations de type $f(x) = k$ à l'aide d'un algorithme de dichotomie.
- Recherche de minimum local, selon un principe de dichotomie.
- Recherche de solutions approchées par balayage d'équations et d'inéquations de type :
$$f(x) = g(x) \qquad f(x) \leq g(x)$$
- Composition de fonctions…

Le document d'accompagnement Algorithmique et programmation présente plusieurs de ces programmes.

## 3. Méthode de Monte-Carlo

Il est très commode de mettre en place l'algorithme de Monte-Carlo en Python pour déterminer la valeur approchée de l'intégrale d'une fonction continue. La méthode peut être exploitée dès la classe de seconde pour déterminer, par exemple, une valeur approchée de l'aire d'un quart de disque.
Créer ainsi une fonction `montecarlo` :

- qui prend comme argument le nombre de tirages à effectuer,
- qui retourne la fréquence de points dans le quart de cercle de rayon 1 dans le premier quadrant.

**Début du code**

```
from random import *
from math import *

def montecarlo(n):
    # n est le nombre de tirages
    # de points aléatoires à effectuer
    nb_points_endessous = 0
    for ... :
        ...

    return f
```

**Complément graphique**

On peut enrichir la fonction en affichant également la figure correspondant aux tirages.
La construction d'un graphique point par point est très lente : voir la « peau de banane » Rapidité de tracé de graphiques, page 29. Il vaut mieux privilégier la construction de quatre listes :

- les deux listes des abscisses et ordonnées des points situés à l'intérieur du cercle ou en-dessous de la courbe d'équation $y = \sqrt{1 - x^2}$ (selon la modélisation choisie) ;
- les deux listes des abscisses et ordonnées des points situés au-dessus de la courbe.

Créer une fonction `montecarlo2` qui prend comme argument le nombre de tirages à effectuer, et qui, avant de renvoyer la fréquence de points dans le quart de disque, affiche la figure correspondant aux tirages.

Instructions utiles

- Bibliothèque graphique à importer : `import matplotlib.pyplot as plt`
- Pour avoir un repère orthonormé : `plt.axis("equal")`
- Pour une grille : `plt.grid()`
- Rappel de l'instruction de tracé, à adapter selon les besoins :
`plt.plot(X_liste_rouge , Y_liste_rouge , "r.")`
- Ne pas oublier d'afficher l'image finale : `plt.show()`

# Pour aller plus loin

## A) Manipulations plus expertes

### 1. Fonction exponentielle et courbe d'Euler

La fonction exponentielle est définie à partir du système :

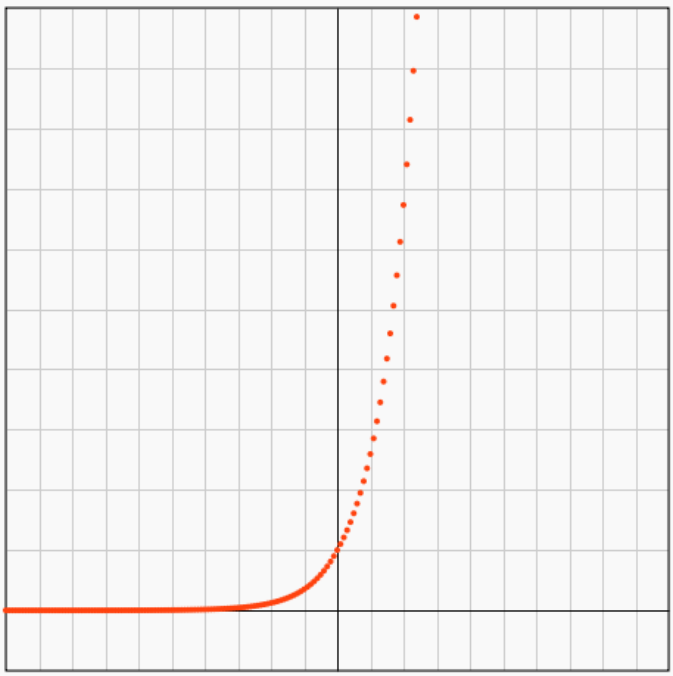
$$(S) : \begin{cases} f' = f \\ f(0) = 1 \end{cases}$$

La **méthode d'Euler** permet de construire la courbe d'une solution approchée d'une équation différentielle avec condition initiale, comme $(S)$.
Grâce à l'équation de la tangente à une courbe en un point, on sait que, pour une fonction $f$ dérivable en $x_0$, et un réel $h$ petit :

$$f(x_0 + h) \approx f(x_0) + f'(x_0)h$$

La méthode d'Euler utilise l'égalité précédente par récurrence pour produire des couples $(x_n; y_n)$ représentant des points tels que $y_n \approx f(x_n)$.

Le programme sous Algobox ci-contre permet de tracer la courbe représentée ci-dessous.



```
VARIABLES
xi EST_DU_TYPE NOMBRE
yi EST_DU_TYPE NOMBRE
h EST_DU_TYPE NOMBRE
i EST_DU_TYPE NOMBRE
n EST_DU_TYPE NOMBRE
x0 EST_DU_TYPE NOMBRE
y0 EST_DU_TYPE NOMBRE

DEBUT_ALGORITHME
LIRE h
LIRE n
xi PREND_LA_VALEUR 0
yi PREND_LA_VALEUR 1
TRACER_POINT (xi,yi)

POUR i ALLANT_DE 1 A n
DEBUT_POUR
   xi PREND_LA_VALEUR xi+h
   yi PREND_LA_VALEUR yi+yi*h
   TRACER_POINT (xi,yi)
FIN_POUR

xi PREND_LA_VALEUR 0
yi PREND_LA_VALEUR 1
h PREND_LA_VALEUR -h

POUR i ALLANT_DE 1 A n
DEBUT_POUR
   xi PREND_LA_VALEUR xi+h
   yi PREND_LA_VALEUR yi+yi*h
   TRACER_POINT (xi,yi)
FIN_POUR

FIN_ALGORITHME
```

**Remarque :** la deuxième boucle permet d'obtenir les points d'abscisse négative.

**Travail demandé**

1. À l'aide du programme sous Algobox, compléter le code ci-contre, dans lequel la fonction `methode_euler` prend pour arguments les coordonnées d'un point de départ, un pas `h` et un nombre d'itérations `n`.

   Elle doit renvoyer la liste des abscisses et celle des ordonnées des points de la courbe d'Euler.

```
def methode_euler(x,y,h,n):
    X_liste = [x]
    Y_liste = [y]

    for ...
        ...

    return X_liste , Y_liste
```

2. Voici une mise en œuvre de cette fonction permettant de récupérer les deux listes de coordonnées, dans deux variables `X` et `Y` :

   ```
   X , Y = methode_euler(0 , 1 , 0.1 , 100)
   ```

   Le programme ci-contre démarre par le paramétrage d'un graphique et par la définition de la fonction `methode_euler`.
   Compléter ce programme, en appelant deux fois la fonction `methode_euler` et en concaténant les listes renvoyées. L'objectif est de tracer ensuite la courbe d'Euler sur un intervalle centré en 0.

   **Remarques**
   - Ne pas hésiter à manipuler le paramétrage du graphique, pour en comprendre le fonctionnement.
   - On concatène deux listes comme on concatène deux chaînes de caractères : en les additionnant.

```
import matplotlib.pyplot as plt
# Configuration des axes
ax=plt.gca()
ax.spines["bottom"].set_position("zero")
ax.spines["left"].set_position("zero")
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
ax.xaxis.set_ticks_position("bottom")
ax.yaxis.set_ticks_position("left")
plt.grid()

def methode_euler(x,y,h,n):
    ...

Xliste1 , Yliste1 = ...
...

plt.show()
```

## 2. Algorithme de Kaprekar

*Attatreya Ramachandra Kaprekar (1905-1988), mathématicien indien, a travaillé sur beaucoup de nombres entiers particuliers. Il fut fort méconnu, un peu moins avec Martin Garner, spécialiste en énigmes mathématiques, qui l'a cité dans ses travaux en 1975.*

Les suites de Kaprekar sont définies comme suit :
1. choisir un nombre entier à 3 chiffres avec au moins 2 chiffres différents ;
2. former par permutation avec les 3 chiffres le nombre le plus grand possible ;
3. former par permutation avec les 3 chiffres le nombre le plus petit possible ;
4. effectuer la différence du plus grand par le plus petit ;
5. revenir à l'étape 2 avec le nouveau nombre.

Conjecture : quel que soit le nombre choisi au départ, la suite atteint le nombre 495.

### Suites de Kaprekar avec calcul arithmétique

Programmer :
1. une fonction `grand` qui, à partir d'un nombre de 3 chiffres, renvoie le nombre le plus grand possible formé de ces 3 chiffres ;
2. une fonction `petit` similaire qui renvoie le nombre le plus petit possible ;
3. une fonction `kaprekar` qui, à partir d'un nombre de 3 chiffres, renvoie la suite de Kaprekar associée.
   On pourra utiliser une boucle « Tant que » avec comme condition d'arrêt le fait d'arriver à 495 ; ou bien une boucle « Pour », dans une perspective d'exploration des suites de Kaprekar.

Instructions utiles

- Division entière de `a` par `b` : `a//b`
- Reste de la division euclidienne de `a` par `b` : `a%b`
- Maximum d'autant de termes que l'on veut : `max(a,b,...)` (s'applique aussi à une liste)
- Minimum, sur le même principe : `min(a,b,...)`
- `append` a déjà été décrit, voir le Memento page 35.

### Suites de Kaprekar avec des chaînes de caractères

Python dispose d'outils puissants de tri de listes. Par ailleurs, une chaîne de caractères est quasiment une liste : on dit que c'est un élément « itérable ». Cela mène à une deuxième approche pour calculer les suites de Kaprekar.

Programmer une fonction `petit_chaine` qui :
1. prenne en argument un nombre entier, comme 436 ;
2. transforme ce nombre en chaîne de caractères : `"436"` ;
3. transforme cette chaîne en une liste ordonnée : `[ "3" , "4" , "6" ]` ;
4. transforme cette liste en chaîne de caractères (`"346"`), puis en entier (346).

Programmer une deuxième fonction, `inversion`, qui prenne en argument l'entier renvoyé par `petit_chaine`, transforme celui-ci en chaîne de caractères, et recrée une chaîne en prenant ses éléments dans l'ordre inverse.
La troisième fonction est la fonction `kaprekar` de la première version, mais utilisant les fonctions `petit_chaine` et `inversion`.

Instructions utiles

- Convertir l'entier `n` en chaîne de caractères : `str(n)`
- Convertir la chaîne de caractères `chaine` en entier : `int(chaine)`
- Trier une liste (ou une chaîne) `L` dans l'ordre croissant : `sorted(L)` (le résultat est une liste)
- `"sep".join(L)` renvoie une chaîne de caractères constituée des éléments de la liste `L`, séparés par ce qui est entre guillemets (ici `"sep"`).
  Ainsi, `"".join(liste_triee)` renvoie le contenu de `liste_triee` concaténé en une chaîne, sans séparateur.
- Voir également le traitement de chaînes de caractères dans le Memento, page 35.

### Suites de Kaprekar avec recherche de minimum dans une liste

Considérons encore un nombre à trois chiffres (517 par exemple) converti en une chaîne de caractères : `"517"`.
Parallèlement, considérons une chaîne de caractères vide : `""`.
Cette troisième variante fait évoluer parallèlement ces deux chaînes, prélevant le minimum (au sens alphabétique) de l'une pour l'ajouter dans l'autre par concaténation, comme ci-contre.
Le résultat final de la deuxième chaîne est alors converti en entier et renvoyé.

| | |
|---|---|
| `"517"` | `""` |
| `"57"` | `"1"` |
| `"7"` | `"15"` |
| `""` | `"157"` |

Programmer cette fonction `petit_par_minimum`.

Les deux autres fonctions `inversion` et `kaprekar` permettant de calculer les suites de Kaprekar sont à adapter si nécessaire avant la mise en œuvre.

Instructions utiles

- Convertir l'entier `n` en chaîne de caractères : `str(n)`
- Convertir la chaîne de caractères `chaine` en entier : `int(chaine)`
- Minimum d'une liste `L`, y compris au sens « alphabétique » : `min(L)`
- Supprimer la première occurence de `e` dans la liste `L` : `L.remove(e)`
- Voir également le traitement de chaînes de caractères dans le Memento, page 35.

### Pour aller plus loin

Idées et remarques

**Remarque 1**

À partir des fonctions décrites ci-dessus, il serait facile de programmer une fonction testant tous les nombres à trois chiffres (chiffres non tous égaux).
L'examen des termes de rang 1 des suites de Kaprekar est aussi intéressant.

**Remarque 2**

En considérant initialement des nombres entiers à trois chiffres, on peut prouver des curiosités en travaillant en base $p$.
Pour un entier $p$ pair, la limite est unique, est systématiquement atteinte et vaut : $\overline{\left(\frac{p}{2}-1\right)(p-1)\frac{p}{2}}^{p}$.
On constate au passage que $\overline{\left(\frac{p}{2}-1\right)(p-1)\frac{p}{2}}^{p} = \frac{p^3-p}{2}$. Pour $p=10$, cela donne… 495.
Pour un entier $p$ impair, l'algorithme oscille entre deux valeurs uniques pour chaque base, $\overline{\frac{p-1}{2}(p-1)\frac{p-1}{2}}^{p}$ et $\overline{\left(\frac{p-1}{2}-1\right)(p-1)\left(\frac{p-1}{2}+1\right)}^{p}$.

### 3. La traversée du pont

Un homme à la démarche titubante tente de rentrer chez lui à pied. Pour cela, il doit emprunter un pont sans garde-corps de 10 pas de long et 4 pas de large. Sa démarche est très particulière :

- soit il avance d'un pas en avant ;
- soit il se déplace d'un pas en diagonale vers la gauche (c'est-à-dire l'équivalent d'un pas en avant et d'un pas vers la gauche) ;
- soit il se déplace d'un pas en diagonale vers la droite (c'est-à-dire l'équivalent d'un pas en avant et d'un pas vers la droite).

Ces trois déplacements sont aléatoires et équiprobables. On suppose qu'au début de la traversée, notre individu se situe au milieu du pont (dans le sens de la largeur). Sur le bord du pont, l'homme arrive encore à être en équilibre.

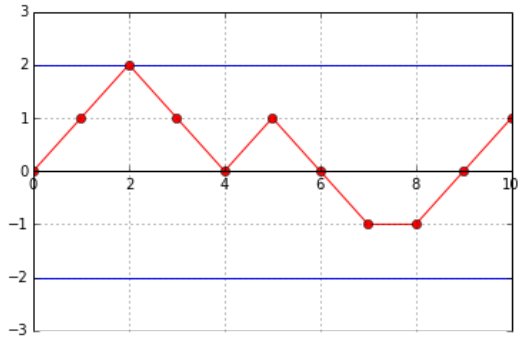
**Question 1**

La fonction `traversee_une_fois` ci-contre a pour but de simuler une traversée du pont, et de représenter graphiquement cette simulation. Elle retourne 1 si la traversée a réussi, 0 sinon.

Compléter cette fonction, et la tester.

**Mise en œuvre**

```
traversee_une_fois()
1
```




```
import matplotlib.pyplot as plt
from random import *

def traversee_une_fois():
    # Réglages d'appoint de la fenêtre graphique
    ax = plt.gca()
    ax.spines["bottom"].set_position("zero")
    plt.grid(True)
    plt.plot([0,10],[-2,-2],"-b")    # Trace des
    plt.plot([0,10],[2,2],"-b")      # bords du pont
    plt.axis([0,10,-3,3])

    # Initialisation des listes
    # des abscisses et ordonnées du marcheur
    X = [0]
    Y = [0]

    while X[-1]<..... and abs(........)<=2:
        de = randint(... , ...)
        X.append(X[-1]+1)
        Y.append(........)
    plt.plot(X,Y,"-or")
    plt.show()

    if ........ and ........ :
        s = 1
    else:
        s = 0
    return s
```

Instructions utiles

- `plt.plot([0,10],[-2,-2],"-b")` trace un segment bleu continu entre les points de coordonnées $(0;-2)$ et $(10;-2)$.
- `X[-1]` est équivalent à `X[len(X)-1]` : cela permet d'aller chercher le dernier élément de la liste `X`.
- `append` et `randint` peuvent servir : voir le Memento, page 35.

**Question 2**

Recopier et modifier rapidement la fonction de la question 1 de sorte qu'elle ne fasse plus le graphique. On pourra l'appeler `traversee_une_fois_sans_graphe`.

Programmer alors une nouvelle fonction `traversee_multiple` qui, à partir d'un entier $n$, simule $n$ traversées du pont et renvoie la fréquence de réussite.

**Question 3**

La fonction `explore_chemin` ci-contre utilise le principe de récursivité (voir Glossaire, page 30), qui n'est pas un objectif du programme. Elle a pour but de dénombrer toutes les traversées possibles.

**Mise en œuvre**

```
explore_chemin(3,0)
25
```

```
def explore_chemin(n,y):
    r = 0
    if n==0:
        if (abs(y)<=2):
            r = 1
        else:
            r = 0
    else:
        if ..... :
            for ......... :
                r = r + explore_chemin(.... , ....)

    return r
```

Les variables essentielles de cette fonction sont décrites ci-dessous.

- `n` est le nombre d'étapes à réaliser pour traverser le pont.
- `y` est la position verticale du marcheur.
- La variable locale `r` permet de compter le nombre de chemins à `n` étapes, partant de la position `y`, menant au succès. Cette variable `r` est retournée à la fin.

Principe de fonctionnement :

- si `n` vaut 0, alors le marcheur est au bout du pont : `r` ne peut valoir que 0 ou 1 ;
- si `n` $> 0$, et si la position `y` du marcheur est « sur le pont », alors il faut compter tous les chemins partant de cette position `y`.
  Cela se fait en cumulant dans `r` les résultats renvoyés par trois applications de la fonction `explore_chemin`, avec les paramètres `n`$-1$ d'une part et `y`$-1$, `y`, `y`$+1$ d'autre part ;
- enfin le contenu de `r` est retourné.

Compléter cette fonction, puis l'utiliser pour calculer la probabilité que le marcheur réussisse à traverser le pont.

Remarque

Il est possible de réaliser cette exploration de tous les chemins sans passer par une fonction récursive.
Mais c'est bien plus pénible, et demande un code beaucoup plus long.

## B) Albums de vignettes et rareté ressentie : le problème du collectionneur

*Cette activité suppose l'introduction préalable des listes en Python et des principales fonctionnalités associées (*`append`, `min`, `max`, `sum`*), ou peut être le prétexte pour les introduire, mais cette introduction n'est pas détaillée ici. L'activité suppose également d'utiliser la bibliothèque* `random` *à partir de laquelle on importera la fonction* `randint`.

Maxime décide d'acheter l'album *Les grandes mathématiciennes* qui contient 480 vignettes à collectionner. Au fur et à mesure de ses achats de vignettes, il a le sentiment que certaines images sont beaucoup plus rares que d'autres car il a du mal à les obtenir pour finir son album alors que d'autres sont très communes car il en possède beaucoup de doubles. Pourtant, alors qu'il consulte la foire aux questions du site internet de la société *Les génies en vignettes autocollantes*, à la question :

« Existe-t-il des images rares ou imprimées en plus grand nombre que d'autres ? »

La réponse est :

« Non. Les images d'une même collection sont imprimées en nombre égal, avant d'être mélangées et conditionnées en pochettes. »

Peut-on expliquer le sentiment de Maxime sur la rareté de certaines vignettes tout en supposant comme l'annonce la société que toutes les vignettes sont imprimées en nombre égal ?


### 1. Écriture d'un algorithme en Python modélisant le problème

On décide de modéliser l'album de Maxime par une liste Python `ALBUM` de taille 480. L'élément numéro $i$ de la liste correspond au nombre de vignettes numéro $i+1$ que Maxime possède. Par exemple, si `ALBUM`$[6] = 3$ cela signifie que Maxime possède 3 vignettes numérotées 7 (Python compte de 0 à 479 mais les vignettes sont numérotées de 1 à 480). Pour simplifier le problème, on suppose que les vignettes sont vendues à l'unité, et qu'à chaque achat, Maxime obtient une vignette au hasard parmi les 480 possibles, en supposant l'équiprobabilité de chaque vignette.

Écrire un programme en Python qui permette de modéliser les achats de Maxime jusqu'à ce qu'il ait terminé son album. En utilisant le programme, répondre aux questions suivantes :

1. Combien de vignettes identiques a-t-il obtenu au maximum ?
2. Combien de vignettes a-t-il obtenu en tout ?
3. À partir de la liste `ALBUM`, créer une liste `FREQ` qui contient la fréquence d'apparition de chaque vignette. Si Maxime décide d'estimer les probabilités d'apparition de chaque vignette par leur fréquence, pourquoi est-il naturel qu'il croie que certaines vignettes sont plus rares que d'autres ?
4. Calculer l'intervalle de fluctuation associé à une probabilité $\frac{1}{480}$ pour un échantillon de taille $n =$ nombre de vignettes obtenues en tout.
   Pourquoi ce phénomène de « rareté ressentie » n'est-il finalement pas si surprenant ?

### 2. Taille de la collection variable

Maxime s'interroge : « S'il y avait deux fois moins de vignettes à collectionner, est-ce que j'aurais deux fois moins de vignettes à acheter pour obtenir la collection complète ? »

On cherche donc cette fois à savoir si le nombre d'achats à faire pour obtenir la collection complète est proportionnel au nombre de vignettes à collectionner.

Écrire une fonction `album` qui prend en argument la taille de la collection et qui renvoie le nombre d'achats nécessaire avant de terminer la collection. Écrire ensuite une fonction `achats_moyen` qui prend en arguments la taille $n$ de la collection et le nombre $e$ d'essais, faisant appel $e$ fois à la fonction précédente et qui renvoie le nombre d'achats moyen sur les $e$ appels.

Écrire ensuite un programme qui permet d'obtenir une représentation graphique avec en abscisse la taille de la collection et en ordonnée le nombre d'achats moyen. Pouvez-vous alors répondre à l'interrogation de Maxime ?

### 3. Nombre de collectionneurs variable

On imagine cette fois que Maxime a une amie Jeanne qui a, elle aussi, acheté l'album. Ils décident d'un commun accord qu'ils achèteront chacun le même nombre de vignettes jusqu'à ce que les deux albums soient finis et qu'ils se donneront sans contrepartie les cartes qu'ils possèdent en double dont l'autre a besoin. Maxime espère qu'ainsi il aura deux fois moins de cartes à acheter. A-t-il raison ?

Écrire une fonction `album_echange` qui prend en arguments la taille $n$ de la collection et le nombre $p$ de collectionneurs et qui renvoie le nombre d'achats nécessaires avant de terminer les $p$ collections.

Sur le même modèle que la section précédente, écrire une fonction `achats_moyen_echange` qui renvoie le nombre d'achats moyen *par collectionneur* pour obtenir les $p$ collections complètes.

Écrire ensuite un programme qui permet d'obtenir une représentation graphique avec en abscisse le nombre de collectionneurs et en ordonnée le nombre d'achats moyen par collectionneur *pour une taille de collection fixée*. Commenter le choix de Jeanne et Maxime.

# « Peaux de bananes » : exemples de manipulations fautives

Les activités de cette partie comportent des erreurs de syntaxe, de manipulation… Il s'agit ici de rencontrer au moins une fois un certain nombre de pièges, pour éviter de tomber dedans plus tard.

## A) Simple ou double égal ?

En Python, le « = » simple sert à l'affectation de variables :

```
a = 3
```

Cette instruction devrait se lire : « a prend la valeur 3 » ou « affecter 3 à la variable a ».
L'erreur à ne pas commettre est d'utiliser le simple = pour tester une égalité. Comme en Algobox, la bonne syntaxe est « == ».
**Exemple avec les deux syntaxes :**

```
def est_egal_a_trois(a):
    if a == 3:          # test d'égalité
        msg = "vrai"    # affectation
    else:
        msg = "faux"    # affectation
    return msg
```

**Mise en œuvre :**
Testons le typage dynamique des variables : Python reconnaîtra-t-il l'entier 3 si on lui entre le flottant 3.0 ?

```
est_egal_a_trois(3.0)
```

## B) L'erreur introuvable de la parenthèse oubliée

**Contexte :** une ligne du code présente une erreur de parenthésage.

**Code incorrect et sa mise en œuvre :**

```
message = input("Texte à afficher ?"  # Il manque la parenthèse fermante
print (message)
```

```
  File "<ipython-input-41-4c065c099e63>", line 2
    print (message)
        ^
SyntaxError: invalid syntax
```

**Explication :** une syntaxe erronée est indiquée sur la ligne 2, alors que la parenthèse manquante est dans la ligne 1. Python considère en conséquence que la ligne 2 prolonge la ligne 1, comme suit :

```
message=input("Texte à afficher ?" print (message)
```

**Code correct :**

```
message = input("Texte à afficher ?")
print (message)
```

## C) Initialiser les listes

**Code incorrect :**

```
def table_multiplication(n):
   for i in range(0,10):
      table.append(i*n)
   return table
```

**Mise en œuvre :**

```
table_multiplication(3)
    ...
----> 3         table.append(i*n)
      4     print(table)

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

**Code corrigé :**
Il faut initialiser une table, même si elle ne contient rien.

```
def table_multiplication_correction(n):
   table = [] # on aurait aussi pu mettre table=[0] et commencer ensuite à range(1,...)

   for i in range(0,10+1):
   # +1 pour aller jusqu'à 10 car le premier nombre est inclus mais le deuxième exclu
      table.append(i*n)
   return table
```

**Variante de codage :** on pouvait aussi utiliser `table = table + [i*n]` au lieu de `table.append(i*n)`.

## D) Le problème des nombres décimaux

Les opérations avec les nombres décimaux peuvent poser des soucis, par exemple la simple addition 0.1+0.2 affiche un résultat vraiment étonnant.

**Codes à tester dans la console :**

```
0.1+0.2
```

**Résultats :**

```
0.30000000000000004
```

```
1/10+2/10
```

```
0.30000000000000004
```

Pour en savoir plus sur ce type d'erreur, dû à la conversion des décimaux en binaire, consulter : docs.python.org/fr/2.7/tutorial/floatingpoint.html, et surtout le blog de Yann Salmon, professeur d'informatique en classe préparatoire.
On peut contourner le problème à l'aide de la bibliothèque `decimal` (docs.python.org/3.3/library/decimal.html) :

**Code à tester dans la console :**

```
from decimal import *
Decimal("0.1") + Decimal("0.2")
```

**Résultat :**

```
Decimal('0.3')
```

On retiendra qu'il convient de travailler le plus souvent possible avec des nombres entiers. Lorsqu'il est indispensable de travailler avec des nombres décimaux, l'erreur commise étant petite, il est possible de résoudre de nombreux problèmes malgré tout (même si cela peut poser des soucis lors de tests d'égalités : le test `0.1+0.2-0.3==0` renverra "faux"). Il est en revanche important d'y penser lors de phénomènes exponentiels par exemple.

## E) Fonctions ne retournant pas de valeur

On rencontre parfois des fonctions qui ne retournent pas de valeur. Suivant l'objectif, on peut avoir quelques surprises...

**Exemple 1 :**

```
a=3

def incremente():
    a=a+1

incremente()
```

Ce code exécuté en console génère une erreur :

```
UnboundLocalError: local variable 'a' referenced
before assignment
```

En effet dans la fonction `incremente`, la ligne commençant par `a=...` crée une variable `a` qui est locale à la fonction `incremente`. En la définissant à `a+1`, on tente donc de faire référence à une variable locale `a`, qui n'est alors pas définie. La variable `a` précédant la fonction est, elle, définie de manière globale.

```
a=3

def incremente():
  b=a+1

incremente()
```

Le code ci-contre ne génère plus d'erreur lors de son exécution.

En effet la fonction `incremente` créé une variable `b` qui est locale à la fonction (et ne sera donc pas visible en dehors de la fonction) en faisant référence à la variable globale `a` (qui est bien globale puisqu'il n'y a plus de déclaration de variable locale `a` dans la fonction).

Pour réussir à incrémenter la variable `a`, on peut penser à utiliser un paramètre. Il est d'ailleurs conseillé — même si ce n'est pas obligatoire — de ne pas lui donner le même nom que la variable globale :

```
a=3

def incremente(nombre):
    nombre = nombre+1

incremente(a)
```

L'exécution de ce code ne génère plus d'erreur, mais la variable globale `a` n'est toujours pas modifiée (essayez). En effet le paramètre `nombre` reste une variable locale à la fonction `incremente`.

*Question :* qu'affiche le code ci-contre ?
Vérifiez !

```
a=3

def incremente():
    a=5
    b=a+1
    print(b)

incremente()
print(a)
```

**Exemple 2 :**

Dans le premier exemple précédent, on peut éviter que l'éxécution génère une erreur en précisant à la fonction que la variable utilisée est globale. On utilise pour ceci la commande `global a` (voir ci-contre).

```
a=3

def incremente():
    global a
    a=a+1

incremente()
```

Mais modifier une variable globale dans une fonction n'est pas forcément une bonne pratique. Retrouverez-vous, dans un programme complexe, dans quelle fonction a été modifiée une variable globale ?

**Note :** les listes représentent un cas particulier, il n'est pas nécessaire de les déclarer en variables globales pour pouvoir les modifier dans une fonction (voir la « peau de banane » suivante).

## F) Return or not return ? (avec un ajout dans une liste)

**Contexte :** il s'agit d'ajouter un élément à une liste prédéfinie, à l'aide soit de `append`, soit d'une concaténation de deux listes.
Dans un programme simple, les deux méthodes donneront le même résultat.
Mais il n'en est pas de même si on veut utiliser une procédure ou une fonction.

**Exemple 1 : code correct avec une procédure et `append`**

```
def ajoute_liste(L,val):
    L.append(val)
```

**Note :** une *procédure* est définie comme une *fonction*, mais elle ne comprend pas l'instruction `return`.

Mise en œuvre dans la console :

```
liste = [ 3 ]
ajoute_liste(liste,2)
liste  # Pour voir la liste
```

Résultat :

```
[3, 2]
```

**Exemple 2 : code incorrect avec une procédure et une concaténation**

```
def ajoute2_liste(L,val):
    L = L+[val]
```

Mise en œuvre dans la console :

```
liste = [ 3 ]
ajoute2_liste(liste,2)
liste # pour voir la liste
```

Résultat :

```
[3]
```

**Exemple 3 : code correct avec une fonction et une concaténation**

```
def ajoute3_liste(L,val):
    L = L+[val]
    return L
```

Mise en œuvre dans la console :

```
liste = [ 3 ]
liste = ajoute3_liste(liste,2)
liste # pour voir la liste
```

Résultat :

```
[3, 2]
```

## G) Ne pas oublier int() ou float()

L'objectif de la fonction ci-contre est de retourner la table de multiplication du nombre `n` entré en paramètre.

```
def table_multiplication(n):
    table=[]
    for i in range(0,10):
        table.append(i*n)
    return table
```

**Mise en œuvre incorrecte avec `input`**

```
n = input("Quelle table de multiplication ?")
table_multiplication(n)
```

Résultat :

```
Quelle table de multiplication ? 3
['','3','33','333','3333',...]
```

**Code corrigé et sa mise en œuvre**

En Python 3 (contrairement à Python 2), la variable `n` alimentée par un `input` est une chaîne de caractères. Il faut transformer ce résultat, selon les besoins :

- en entier avec `int()` ;
- en flottant avec `float()`.

```
n = int(input("Quelle table de multiplication ?"))
table_multiplication(n)
```

```
Quelle table de multiplication  ? 3
[0, 3, 6, 9, 12, 15, 18, 21,...]
```

## H) Deux types de division : euclidienne ou décimale

La division décimale classique est codée par un simple « / », et renvoie un flottant.
La division euclidienne est codée par « // », et renvoie un entier.
Rappelons au passage que le reste de la division euclidienne de `a` par `b` se code : `a%b`

**Code incorrect**

```
def divi_euclidienne(a,b):
    r = a%b
    q = a/b
    return [r,q]
    # Retour d'une liste de 2 valeurs
```

**Mise en œuvre et résultat**

```
divi_euclidienne(8,5)
[3 , 1.6]
```

**Code corrigé**

```
def divi_euclidienne_correction(a,b):
    r = a%b
    q = a//b
    return [r,q]
```

**Mise en œuvre et résultat**

```
divi_euclidienne_correction(8,5)
[3 , 1]
```

# I) Listes liées

Créer une liste `X`, par exemple `[1,2,3]`.
La liste `Y` est créée par copie de la liste `X` : `Y=X`
Modifier alors la liste `X` (ajouter un élément, ou modifier l'élément de rang 0).
Observer ce qu'il se passe sur `Y`.

**Code de test en console**

```
X = [1,2,3]
Y = X
print("X : ",X)
print("Y, copie de X : ",Y)
X[0] = 3
print("X modifié : ",X)
print("Qu'est devenu Y : ",Y)
Y.append(4)
print("Nouvel X après modification de Y : ",X)
```

**Résultat après exécution**

```
X : [1, 2, 3]
Y, copie de X : [1, 2, 3]
X modifié : [3, 2, 3]
Qu'est devenu Y :
[3, 2, 3]
Nouvel X après modification de Y :
[3, 2, 3, 4]
```

L'intruction `Y = X` ne se contente donc pas de recopier `X` dans `Y` : elle lie les deux listes ; toute modification de l'une sera répercutée sur l'autre.
Pour réaliser en pratique une copie de `X` indépendante, il faut recopier le contenu de `X` dans `Y`.

**Code de test corrigé**

```
X = [1,2,3]
Y = list(X)      # ou Y = X[:]
print("X : ",X)
print("Y, copie du contenu de X : ",Y)
X[0] = 3
print("X modifié : ",X)
print("Y n'a pas changé : ",Y)
```

**Résultat après exécution**

```
X :  [1, 2, 3]
Y, copie du contenu de X :  [1, 2, 3]
X modifié :  [3, 2, 3]
Y n'a pas changé :  [1, 2, 3]
```

Pour plus de détails : apprendre-python.com/page-apprendre-listes-list-tableaux-tableaux-liste-array-python-cours-debutant

# J) Rapidité de tracé de graphiques

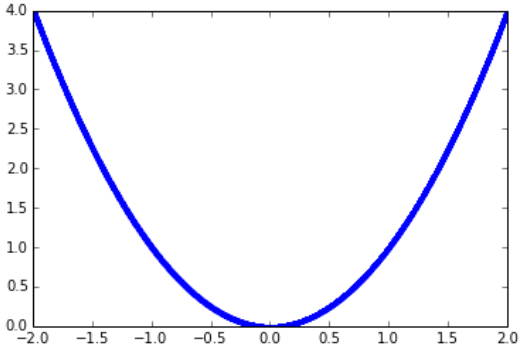
La bibliothèque `matplotlib` et son module `pyplot` sont conçus de manière à tracer des graphiques point par point de façon rapide… à condition de se plier à leur pratique.
Soit à tracer la parabole d'équation $y = x^2$ sur l'intervalle $[-2;2]$, avec un pas de 0.001 : quatre milliers de points suffisent à faire une étude comparative de vitesse de tracé, à l'aide de la bibliothèque `time`.

**Programme « naïf »**

```
# Chargement de matplotlib.pyplot,
# renommée plt pour simplifier
import matplotlib.pyplot as plt
import time  # Bibliothèque time

def trace_parabole_naif():
    # Enregistrement de l'heure actuelle
    depart = time.clock()   # Top départ

    a = -2      # Initialisation
    h = 0.001
    while a<2:
        # Tracé d'un petit point bleu (".b")
        # de coordonnées (a,a*a)
        plt.plot(a , a*a , "b.")
        a = a+h   # Le curseur avance d'un pas

    fin = time.clock()   # Top de fin
    # Retour du temps de calcul, en secondes
    return "Temps :" + str(fin-depart) + " s."
```

**Mise en œuvre et résultat**

```
trace_parabole_naif()
Temps : 5.703140999999999 s.
plt.show()  # Affichage du graphique
```




**Programme utilisant des listes**

L'instruction `plot` de `matplotlib.pyplot` est optimisée pour tracer des graphiques utilisant des listes. Il faut raisonner en termes de « tableau de valeurs » :

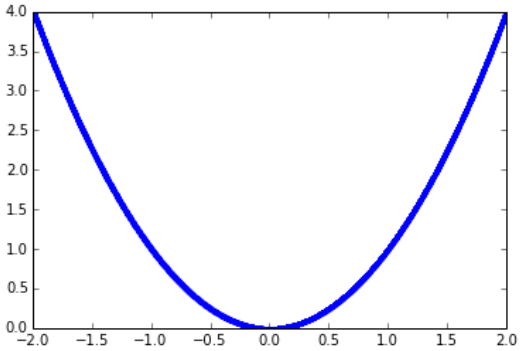
- `X` et `Y` sont deux listes de même longueur : par exemple [1,2,3] et [1,4,9]
- `plt.plot(X,Y,".b")` trace l'ensemble des points `( X[i] , Y[i] )` : (1,1), (2,4) et (3,9).

L'essentiel du programme consiste donc à alimenter les deux listes `X` et `Y`, à l'aide de la commande `append`.

```
import matplotlib.pyplot as plt
import time

def trace_parabole_tableaux():
    depart=time.clock()

    X = []  # Initialisation des listes
    Y = []

    a = -2
    h = 0.001
    while a<2:
        X.append(a)   # Ajout des valeurs
        Y.append(a*a) # au "bout" de X et Y
        a = a+h

    # Tracé de l'ensemble du tableau de valeurs
    plt.plot(X,Y,".b")

    fin=time.clock()
    return "Temps : " + str(fin-depart) + " s."
```

```
trace_parabole_tableaux()
Temps : 0.04798300000000211 s.
plt.show()
```

Une différence de temps d'exécution spectaculaire…



# Glossaire

## A) Console et fenêtre de script

La console d'un éditeur Python est le lieu où s'appliquent les programmes. Dans la fenêtre de script, les fonctions et programmes peuvent être enregistrés.
Plusieurs scénarios sont possibles.

1. La console est utilisée seule. L'utilisateur y écrit toutes ses instructions, qui s'exécutent à chaque appui sur la touche `Entrée`. Il peut aussi y définir des fonctions, qu'il pourra exécuter ensuite. Mais ces fonctions ne sont pas enregistrées à la fermeture de l'éditeur.
2. Les fonctions sont écrites dans la fenêtre de script, puis l'ensemble est exécuté. Les fonctions sont alors disponibles pour une exécution dans la console. Le contenu de la fenêtre de script peut être enregistré, pour utilisation et modification ultérieures.
3. Dans la fenêtre de script sont écrites non seulement les fonctions, mais aussi un programme qui utilise ces fonctions dans un certain but. Il est alors souvent nécessaire de prévoir un affichage pour certaines étapes de ce programme : voir `print` : pour afficher un message, page 31.

## B) Indentation

L'indentation consiste à aligner verticalement des instructions qui doivent s'effectuer successivement dans un même bloc. Pour créer cet alignement vertical, on utilise les espaces et / ou les tabulations.
Dans de nombreux langages, les instructions d'un même bloc sont signifiées de manière explicite :

en **Pascal**, un bloc d'instructions est délimité par les « balises » `BEGIN` et `END` :

```
BEGIN
instruction
instruction
...
instruction
END
```

en **C**, ce sont les accolades qui délimitent le début et la fin du bloc d'instruction :

```
{
instruction
instruction
...
instruction
}
```

Les exemples ci-dessus ne comportent aucune indentation : elle n'est pas utile au bon déroulement du programme. Cela n'est cependant pas une bonne pratique ; l'indentation, dans tout langage, sert à rendre le programme plus lisible et compréhensible.
En revanche, dans le langage Python, c'est l'indentation et uniquement l'indentation qui détermine les instructions d'un même bloc.

**Un exemple d'erreur d'indentation**

La fonction `SommeDesPremiersTermes` ci-contre a pour but de calculer la somme des termes d'une suite arithmétique. Mais une erreur d'indentation a été commise : le résultat renvoyé n'est pas correct...
Saurez-vous trouver l'erreur ?

```
def SommeDesPremiersTermes(u0,raison,nb_termes):
    somme = 0
    for i in range(0,nb_termes):
        ui = u0+i*raison
    somme = somme + ui
    return somme
```

## C) `input` : pour interroger l'utilisateur

L'instruction `input()` permet de dialoguer avec l'utilisateur via un formulaire :
1. l'argument passé en paramètre est affiché ;
2. ce que l'utilisateur entre dans le formulaire est retourné.

Tester le programme ci-contre, qui devrait échouer à son exécution.

```
a = input("Entrer un nombre : ")
b = a+5
print(b)
```

En réalité, la variable entrée grâce à `input()` est une chaîne de caractères : on ne peut pas la manipuler comme des nombres.

Voici 3 types de variables :
- `string` : chaîne de caractères ;
- `int` : entier ;
- `float` : virgule flottante (nombre décimal).

L'instruction `int(a)` convertit la variable `a` en entier, si cela est possible. L'instruction `float(a)` agit de même.

**Code corrigé**

```
# La chaîne de caractères retournée
# par input est convertie en entier
a = int(input("Entrer un nombre : "))
b = a+5
print(b)
```

Essayer en entrant un nombre non entier ; puis en utilisant `float` au lieu de `int`. Quelle est la différence ?

**Note importante :** `input()`, et à un moindre degré `print()`, relèvent de la notion de **programme**, et non plus de fonctions où les paramètres sont passés en arguments.
Citons le document d'accompagnement Algorithmique et programmation :
« *…les notions d'entrées-sorties (fonctions input et print) ne sont pas développées dans ce document : elles ne relèvent pas de la pensée algorithmique et l'accent mis par le programme sur la notion de fonction permet de s'en libérer complètement.* »

## D) `print` : pour afficher un message

Pour imprimer sur écran le contenu d'une variable `a` : `print(a)`

Pour un affichage complexe, mélangeant par exemple une chaîne de caractères et une variable : `print("La variable a contient : ", a)`

## E) Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même.
L'usage de fonctions récursives fournit un code élégant, mais qui demande un certain effort intellectuel pour comprendre son fonctionnement. De plus, il faut être attentif aux conditions utilisées : encore plus qu'avec un `while`, le danger d'une boucle infinie est présent.

**Exemple : *pgcd* de deux entiers avec une fonction récursive**

À comparer avec le *pgcd* classique fondé sur une boucle `while`.

```
def pgcd_recursif(a,b):
    r = a%b # Division euclidienne de a par b
    if r==0 :
        # Si le reste est nul : on renvoie b
        return b
    else :
        # Sinon : on calcule le pgcd de b et r
        return pgcd_recursif(b,r)
```

**Mise en œuvre dans la console**

```
pgcd_recursif(48945778,58268)

14
```

# F) Typage des variables et vérification

La programmation en Python est facilitée par le **typage dynamique des variables** : une variable est de tel ou tel type selon le contexte des affectations qu'elle subit.
Mieux vaut, cependant, avoir une idée des objets manipulés et du comportement de Python.
Dans ce livret sont manipulés quatre types de variables :

- les **entiers**, comme 125 ;
- les **flottants**, comme 5.0 ou 3.141592 ; les nombres sont représentés avec environ 16 chiffres significatifs ;
- les **chaînes de caractères**, comme `"Pi vaut environ 3.14"` (chaîne comportant 20 caractères numérotés de 0 à 19) ;
- les **listes**, dont les éléments peuvent eux-mêmes être des entiers, des flottants, des chaînes de caractères… voire des listes.

Le statut d'une variable peut changer selon les calculs. Tester en console le code ci-contre.

```
a = 10
a = a/2
a  # Pour afficher a
```

- au départ, `a` est de type **entier** ;
- la division par 2 impose le type **flottant** ;
- on ne peut plus faire d'arithmétique en toute confiance avec le résultat final : 5.0.

Voir également les « peaux de bananes » Deux types de division : euclidienne ou décimale (page 27) et Le problème des nombres décimaux (page 24).

### Vérifications diverses avec `assert`

Le document d'accompagnement Algorithmique et programmation donne plusieurs exemples d'utilisation de cette fonction. L'objectif d'`assert` est de déclencher une erreur d'exécution quand les conditions qu'elle énonce ne sont pas vérifiées.

#### Exemple 1 : dichotomie

Soit la fonction `dichotomie` ci-contre.
Quand on travaille par dichotomie sur un intervalle $[a;b]$, la première vérification à faire est que $a$ est bien inférieur à $b$.
Cela peut se traduire par la ligne de code, à placer au début de la fonction :

```
assert a<b
```

```
def dichotomie(f,a,b,h):
    while b-a>h:
        c = (a+b)/2
        if f(a)*f(c)<0:
            b = c
        else:
            a = c
    return [a,b]
```

#### Exemple 2 : typage d'une variable

Les fonctions de l'activité Algorithme de Kaprekar, page 17, s'appliquent à un entier `n` à trois chiffres. Si elles sont appliquées à un flottant ou une chaîne de caractères, il est probable qu'une erreur survienne ; mais il serait intéressant qu'un message d'erreur adapté soit émis.

La ligne de code à ajouter au début de la fonction est alors :

```
assert type(n)==int, "n n'est pas entier"
```

Si le contenu du paramètre `n` n'est pas entier, le programme s'interrompt et le message d'erreur `"n n'est pas entier"` est affiché.

# Les bibliothèques et modules

## A) Bibliothèques existantes

### 1. La bibliothèque standard

La bibliothèque standard de Python est un ensemble de fonctions prédéfinies qui sont considérées comme suffisamment génériques pour être utilisées par tous les utilisateurs.
Cette bibliothèque contient également un certain nombre de modules (regroupements de fonctions) à importer en fonction des besoins. Il en existe un grand nombre, parmi lesquels :

- le module `math` qui contient des fonctions mathématiques comme le cosinus (`cos`), la racine carrée (`sqrt`), le nombre $\pi$ (`pi`), etc ;
- le module `random` qui permet de générer des nombres aléatoires, notamment à l'aide de la fonction `random()` qui génère un nombre aléatoire compris entre 0 inclus et 1 exclu ;
- le module `turtle` qui permet de réaliser des figures géométriques (voir activité La Tortue Python en prolongement de Scratch, page 12) ;
- le module `decimal` qui permet de travailler avec des nombres décimaux (voir « peau de banane » Le problème des nombres décimaux, page 24).

Pour importer une bibliothèque ou un module, on peut procéder de plusieurs façons. Tester les codes suivants dans la console.

**Importer une seule fonction**

```
from math import sqrt
sqrt(100)
```

**Importer tout le module**

```
import math
math.cos(0)
```

**Importer une seule fonction** en lui donnant un nom (alias)

```
from math import sqrt as racine
racine(36)
```

**Importer tout le module**, en lui donnant un nom (alias)

```
import math as m
m.sin(m.pi/2)
```

**Importer tout le module**, sans besoin de « nommer » les fonctions

```
from math import *
sin(pi/2)
```

### 2. La bibliothèque `matplotlib`

`matplotlib` est une bibliothèque graphique de visualisation 2D (avec un support pour la 3D et l'animation). Un exemple d'utilisation, parmi d'autres, est proposé page 7.

### 3. Autres bibliothèques

- La bibliothèque `numpy` est une bibliothèque de calcul numérique apportant, entre autres, le support de tableaux multidimensionnels (`array`).
- La bibliothèque `scipy` concerne le calcul scientifique.
- La bibliothèque `lycee`, développée par des enseignants de l'académie d'Amiens, regroupe un grand nombre des bibliothèques précédentes. Son exploration est conseillée aux utilisateurs curieux de voir comment certaines fonctions y sont construites.
  Pour la télécharger : http://download.tuxfamily.org/edupython/lycee.py

## B) Se constituer une bibliothèque personnelle

Il est possible d'alimenter une bibliothèque personnelle avec toutes les fonctions réalisées durant l'année. Prenons un exemple.
La fonction codée dans le premier encadré ci-contre a pour but de simuler $n$ tirages indépendants de probabilité de succès $p$.

1. Rédiger ce programme et l'enregistrer dans un fichier nommé : `fonctions.py`
2. Ouvrir un autre fichier **dans le même dossier que** `fonctions.py` et y écrire le code du second encadré. L'exécuter ensuite.

```
import random as alea
def BernoulliCompteSucces(n,p):
      NbSucces=0
      for i in range(n):
         if alea.random()<p:
            NbSucces=NbSucces+1
      return NbSucces
```

```
from fonctions import *
print(BernoulliCompteSucces(30,0.2))
```

# Memento

## Variables : types, conversions

| Chaînes de caractères | Listes |
|---|---|
| Concaténer deux chaînes "abc" et " def" :<br>*"abc" + " def"*<br>Obtenir le caractère numéro `i` de la chaîne `s` :<br>`s[i]`<br>(comme pour une liste)<br>Convertir la chaîne de caractères "123" en un entier :<br>`int("123")`<br>Convertir l'entier 123 en chaîne de caractères :<br>`str(123)`<br>Convertir la liste `L` en chaîne de caractères, les éléments étant séparés par "-" ; stocker le résultat dans `chaine` :<br>`chaine = "-".join(L)` | Initialiser une liste `L` vide :<br>`L = []`<br>Ajouter l'élément `x` au bout de la liste `L` :<br>`L.append(x)`<br>Ajouter l'élément `x` au rang `i` de la liste `L` :<br>`L.insert(i,x)`<br>Supprimer la première occurrence de `x` dans la liste `L` :<br>`L.remove(x)`<br>Supprimer l'élément d'indice `i` dans la liste `L` :<br>`L.pop(i)`<br>Longueur d'une liste `L` :<br>`len(L)`<br>Trier une liste `L` et stocker le résultat dans `L_triee` :<br>`L_triee = sorted(L)` |

## Calculs

| | |
|---|---|
| 4 opérations classiques : $+$ $-$ `*` `/`<br>`a` exposant `b` : `a**b`<br>Division entière de `a` par `b` : `a//b`<br>Reste de la division euclidienne de `a` par `b` : `a%b`<br>Définir `z`, nombre complexe : `z=complex(3,2)`<br>En particulier, définir `i` : `i=complex(0,1)`<br>Valeur absolue, module de `z` : `abs(z)`<br>Partie réelle, partie imaginaire de `z` :<br>`z.real` `z.imag`<br>Argument de `z` (avec bibliothèque `cmath`) :<br>`phase(z)` | **Bibliothèque `math`**<br>Racine carrée de `a` : `sqrt(a)`<br>Exponentielle de `a` : `exp(a)`<br>Logarithme népérien de `a` : `log(a)`<br>et `cos`, `sin`, `tan`... (angles en radians)<br>**Bibliothèque `random`**<br>Nombre aléatoire dans $[0;1[$ : `random()`<br>Nombre aléatoire entier entre `a` et `b` inclus :<br>`randint(a,b)` |

## Tests

| | |
|---|---|
| `if (condition):`<br>`  ...`<br>`else:`<br>`  ...` | Opérateurs de comparaison :<br>`==` `!=` `<` `>` `<=` `>=`<br>Opérateurs logiques :<br>`and` `or` |

## Boucles

| Boucle `Pour` | Boucle `Tant que` |
|---|---|
| Avec exactement 10 itérations, le `compteur` variant de 0 à 9 :<br>`for compteur in range(10):`<br>`...`<br>Avec exactement 9 itérations, le `compteur` variant de 1 à 9 :<br>`for compteur in range(1,10):`<br>`...`<br>Parcours d'une liste non numérique :<br>`for compteur in ["lundi", "mardi", "mercredi"]:`<br>`...` | `while (condition):`<br>`...`<br>(les parenthèses autour de la condition sont optionnelles) |

## Graphiques

En début de programme, charger la bibliothèque `matplotlib.pyplot` :
`import matplotlib.pyplot as plt`

Placer un point de coordonnées $(x;y)$, bleu, gros :
`plt.plot(x,y,"bo")`

Tracer un segment entre deux points $A$ et $B$, rouge :
`plt.plot([`$x_A$`,`$x_B$`],[`$y_A$`,`$y_B$`],"r")`

Tracer les points dont les coordonnées sont données par les listes `X_liste` et `Y_liste`, en noir, en reliant les points :
`plt.plot(X_liste,Y_liste,"k-")`

Pour avoir un repère orthonormé :
`plt.axis("equal")`

Pour une grille :
`plt.grid()`

Afficher la fenêtre graphique :
`plt.show()`

| couleur | | style | |
|---|---|---|---|
| b | bleu | - | ligne continue |
| g | vert | – | tirets |
| r | rouge | : | pointillés |
| c | cyan | . | points |
| m | magenta | o | billes |
| y | jaune | x | croix |
| k | noir | v | triangles |
| w | blanc | -. | point-tirets |
| | ... | | ... |

# Correction des exercices de la première partie

### C-1. x_sommet

```
def x_sommet(a,b,c):
    return -b/(2*a)
```

### C-2. nombre_racines_trinome

```
def nombre_racines_trinome(a,b,c):
   # La fonction discr a déjà été définie
   if discr(a,b,c)<0:
      message = "Pas de racine dans R"
   if discr(a,b,c)==0:
      message = "1 racine"
   if discr(a,b,c)>0:
      message = "2 racines"
   return message
```

### C-3. racines_trinome

```
from math import *

def racines_trinome(a,b,c):
   d = discr(a,b,c)     # La fonction discr a déjà été définie
   if d<0:
      message = "Pas de racine dans R"
   if d==0:
      message = "1 racine "+str(x_sommet(a,b,c))
   if d>0:
      message = "2 racines : "+str((-b-sqrt(d))/(2*a)) +" et : "+str((-b+sqrt(d))/(2*a))
   return message
```

### C-4. Application 1 : fibo

```
def fibo(k):
   # Initialisation de la liste
   L = [1,1]
   for compteur in range(2,k):
      L.append(L[compteur-1]+L[compteur-2])
   return L
```

```
def fibo(u0,u1,k):
   # Initialisation de la liste
   L = [u0,u1]
   for compteur in range(2,k):
      L.append(L[compteur-1]+L[compteur-2])
   return L
```

## C-4. Application 2 : Héron d'Alexandrie et les racines carrées

**Méthode de Héron d'Alexandrie pour $\sqrt{5}$**

```
from math import *

def approx_racine():
    u = 7
    while ((u-sqrt(5))>10**-6):
        u = 0.5*(u+5/u)
    return u
```

**Méthode de Héron d'Alexandrie pour $\sqrt{a}$**

Il faut partir d'un nombre supérieur à $\sqrt{a}$.

```
from math import *

def approx_racine(a):
    u = a + 2
    while ((u-sqrt(a))>10**-6):
        u = 0.5*(u+a/u)
    return u
```

**Pour aller plus loin**

Un critère de fin n'utilisant pas la bibliothèque `math` et sa fonction `sqrt` pourrait être d'observer l'écart entre le carré de `u` et `a`. Mais comment alors mesurer précisément l'approximation ?

## D-1. Suite de Syracuse

```
def syracuse(n):
    L = [n]
    while (n!=1):
        if (n%2==0):     # Test de parité
            n = (n//2)   # Division entière
        else:
            n = n*3+1
        L.append(n)
    return L
```

Mise en œuvre en console, et calcul de la longueur de la liste :

```
liste = syracuse(9)
len(liste)
```

## D-2. Suite de Syracuse graphique

```
import matplotlib.pyplot as plt

# La fonction syracuse a déjà été définie (voir D-1.)

def graph_syracuse(nombre):
    Y_liste = syracuse(nombre)          # On remplit la liste Y_liste
                                        #     avec les valeurs de la suite de Syracuse
    X_liste = []                        # Initialisation de la liste des abscisses
    for i in range(1,len(Y_liste)+1):   # Affectation des indices de la suite
        X_liste.append(i)
    plt.plot(X_liste,Y_liste,"b.")      # Création du graphique et affichage
    plt.show()
```