

PYTHON AU LYCÉE

ARNAUD BODIN

ALGORITHMES ET PROGRAMMATION



Python au lycée

C'est parti !

Tout le monde utilise un ordinateur, mais c'est une autre chose de le piloter ! Tu vas apprendre ici les bases de la programmation. L'objectif de ce livre est double : approfondir les mathématiques à travers l'informatique et maîtriser la programmation en s'aidant des mathématiques.

Python

Choisir un langage de programmation pour débiter est délicat. Il faut un langage avec une prise en main facile, bien documenté, avec une grande communauté d'utilisateurs. Python possède toutes ces qualités et davantage encore. Il est moderne, puissant et très utilisé, y compris par les programmeurs professionnels. Malgré toutes ces qualités, débiter la programmation (avec Python ou un autre langage) est difficile. Le mieux est d'avoir déjà une expérience du code, à l'aide de *Scratch* par exemple. Il reste quand même une grande marche à gravir et ce livre est là pour t'accompagner.

Objectif

Bien maîtriser Python te permettra d'apprendre facilement les autres langages. Surtout le langage n'est pas le plus important, l'essentiel ce sont les algorithmes. Les algorithmes sont comme des recettes de cuisine, il faut suivre pas à pas les instructions et ce qui compte, c'est le résultat final et non le langage avec lequel a été écrite la recette. Ce livre n'est donc ni un manuel complet de Python, ni un cours d'informatique, il ne s'agit pas non plus d'utiliser Python comme une super-calculatrice.

Le but est de découvrir des algorithmes, d'apprendre la programmation pas à pas à travers des activités mathématiques/informatiques. Cela te permettra de mettre en pratique des mathématiques avec ici la volonté de se limiter aux connaissances acquises au niveau seconde.

Mathématiques pour l'informatique

Informatique pour les mathématiques

Comme les ordinateurs ne manipulent que des nombres, les mathématiques sont indispensables pour communiquer avec eux. Un exemple est l'écriture binaire qui utilise les puissances de 2, la division euclidienne... Un autre exemple est l'affichage graphique à l'écran qui nécessite de bien maîtriser les coordonnées (x, y) , la trigonométrie...

L'informatique accompagne à merveille les mathématiques ! L'ordinateur devient indispensable pour manipuler de très grands nombres ou bien tester des conjectures sur de nombreux cas. Tu découvriras dans ce livre des fractales, des L-systèmes, des arbres browniens... et la beauté de phénomènes mathématiques complexes.

Sommaire

I	Mise en route	1
1	Premiers pas	2
2	Tortue (Scratch avec Python)	9
II	Fondamentaux	17
3	Si ... alors ...	18
4	Fonctions	24
5	Arithmétique – Boucle tant que – I	33
6	Chaînes de caractères – Analyse d'un texte	40
7	Listes I	50
III	Notions avancées	58
8	Statistique – Visualisation de données	59
9	Fichiers	68
10	Arithmétique – Boucle tant que – II	77
11	Binaire I	82
12	Listes II	88
13	Binaire II	94
IV	Projets	97
14	Probabilités – Paradoxe de Parrondo	98
15	Chercher et remplacer	101
16	Calculatrice polonaise – Piles	106

17	Visualiseur de texte – Markdown	118
18	L-système	125
19	Images dynamiques	133
20	Jeu de la vie	139
21	Graphes et combinatoire de Ramsey	145
22	Bitcoin	154
23	Constructions aléatoires	163
V	Guides	170
24	Guide de survie Python	171
25	Principales fonctions	180
26	Notes et références	196
	Index	

Résumé des activités

Premiers pas

Lance-toi dans la programmation ! Dans cette toute première activité, tu vas apprendre à manipuler des nombres, des variables et tu vas coder tes premières boucles avec Python.

Tortue (Scratch avec Python)

Le module `turtle` permet de tracer facilement des dessins en Python. Il s'agit de commander une tortue à l'aide d'instructions simples comme « avancer », « tourner »... C'est le même principe qu'avec *Scratch*, avec toutefois des différences : tu ne déplaces plus des blocs, mais tu écris les instructions ; et en plus les instructions sont en anglais !

Si ... alors ...

L'ordinateur peut réagir en fonction d'une situation. Si une condition est remplie il agit d'une certaine façon, sinon il fait autre chose.

Fonctions

Écrire une fonction, c'est la façon la plus simple de regrouper du code pour une tâche bien particulière, dans le but de l'exécuter une ou plusieurs fois par la suite.

Arithmétique – Boucle tant que – I

Les activités de cette fiche sont centrées sur l'arithmétique : division euclidienne, nombres premiers... C'est l'occasion d'utiliser intensivement la boucle « tant que ».

Chaînes de caractères – Analyse d'un texte

Tu vas faire quelques activités amusantes en manipulant les chaînes de caractères.

Listes I

Une liste est une façon de regrouper des éléments en un seul objet. Après avoir défini une liste, on peut récupérer un par un chaque élément de la liste, mais aussi en ajouter de nouveaux...

Statistique – Visualisation de données

C'est bien de savoir calculer le minimum, le maximum, la moyenne, les quartiles d'une série. C'est mieux de les visualiser tous sur un même graphique !

Fichiers

Tu vas apprendre à lire et à écrire des données dans des fichiers.

Arithmétique – Boucle tant que – II

On approfondit notre étude des nombres avec la boucle « tant que ». Pour cette fiche tu as besoin d'une fonction `est_premier()` construite dans la fiche « Arithmétique – Boucle tant que – I ».

Binaire I

Les ordinateurs transforment toutes les données en nombres et manipulent uniquement ces nombres. Ces nombres sont stockés sous la forme de listes de 0 et de 1. C'est l'écriture binaire des nombres ! Pour mieux comprendre l'écriture binaire, tu vas d'abord mieux comprendre l'écriture décimale.

Listes II

Les listes sont tellement utiles qu'il faut savoir les manipuler de façon simple et efficace. C'est le but de cette fiche !

Binaire II

On continue notre exploration du monde des 0 et des 1.

Probabilités – Paradoxe de Parrondo

Tu vas programmer deux jeux simples. Lorsque tu joues à ces jeux, tu as plus de chances de perdre que de gagner. Pourtant lorsque tu joues aux deux jeux en même temps, tu as plus de chances de gagner que de perdre ! C'est une situation paradoxale.

Chercher et remplacer

Chercher et remplacer sont deux tâches très fréquentes. Savoir les utiliser et comprendre comment elles fonctionnent te permettra d'être plus efficace.

Calculatrice polonaise – Piles

Tu vas programmer ta propre calculatrice ! Pour cela tu vas découvrir une nouvelle notation pour les formules et aussi découvrir ce qu'est une « pile » en informatique.

Visualiseur de texte – Markdown

Tu vas programmer un traitement de texte tout simple qui affiche proprement des paragraphes et met en évidence les mots en gras et en italiques.

L-système

Les L-systèmes offrent une façon très simple de coder des phénomènes complexes. À partir d'un mot initial et d'opérations de remplacement, on arrive à des mots compliqués. Lorsque l'on « dessine » ces mots, on obtient de superbes figures fractales. Le « L » vient du botaniste A. Lindenmayer qui a inventé les L-systèmes afin de modéliser les plantes.

Images dynamiques

Nous allons déformer des images. En répétant ces déformations, les images deviennent brouillées. Mais par miracle au bout d'un certain nombre de répétitions l'image de départ réapparaît !

Jeu de la vie

Le *jeu de la vie* est un modèle simple de l'évolution d'une population de cellules qui naissent et meurent au cours du temps. Le « jeu » consiste à trouver des configurations initiales qui donnent des évolutions intéressantes : certains groupes de cellules disparaissent, d'autres se stabilisent, certains se déplacent. . .

Graphes et combinatoire de Ramsey

Tu vas voir qu'un problème tout simple, qui concerne les relations entre seulement six personnes, va demander énormément de calculs pour être résolu.

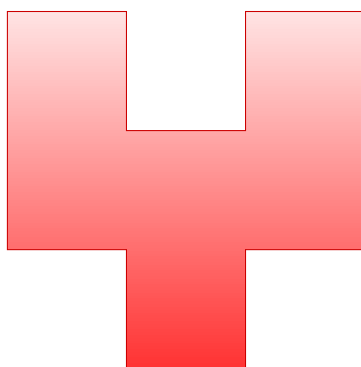
Bitcoin

Le *bitcoin* est une monnaie dématérialisée et décentralisée. Elle repose sur deux principes informatiques : la cryptographie à clé publique et la preuve de travail. Pour comprendre ce second principe, tu vas créer un modèle simple de *bitcoin*.

Constructions aléatoires

Tu vas programmer deux méthodes pour construire des figures qui ressemblent à des algues ou des coraux. Chaque figure est formée de petits blocs lancés au hasard et qui se collent les uns aux autres.

PREMIÈRE PARTIE



MISE EN ROUTE

Premiers pas

Lance-toi dans la programmation ! Dans cette toute première activité, tu vas apprendre à manipuler des nombres, des variables et tu vas coder tes premières boucles avec Python.

Cours 1 (Nombres avec Python).

Vérifie dans la console que Python fonctionne correctement, en tapant les commandes suivantes dans une console Python :

```
>>> 2+2
>>> "Bonjour le monde !"
```

Voici quelques instructions.

- **Addition.** $5+7$.
- **Multipliation.** $6*7$; avec des parenthèses $3*(12+5)$; avec des nombres à virgule $3*1.5$.
- **Puissance.** $3**2$ pour $3^2 = 9$; puissance négative $10**-3$ pour $10^{-3} = 0.001$.
- **Division réelle.** $14/4$ vaut 3.5 ; $1/3$ vaut 0.3333333333333333 .
- **Division entière et modulo.**
 - $14//4$ vaut 3 : c'est le quotient de la division euclidienne de 14 par 4 , note bien la double barre ;
 - $14\%4$ vaut 2 : c'est le reste de la division euclidienne de 14 par 4 , on dit aussi « 14 modulo 4 ».

Remarque. Dans tout ce cours, on écrira les « nombres à virgule » sous la forme 3.5 (et pas $3,5$). Le séparateur décimal est donc le point. En informatique les nombres à virgule sont appelés « nombres flottants ».

Activité 1 (Premiers pas).

Objectifs : faire tes premiers calculs avec Python.

1. Combien y a-t-il de secondes en un siècle ? (Ne tiens pas compte des années bissextiles.)
2. Jusqu'où faut-il compléter les pointillés pour obtenir un nombre plus grand qu'un milliard ?

$$(1 + 2) \times (3 + 4) \times (5 + 6) \times (7 + 8) \times \dots$$

3. Quels sont les trois derniers chiffres de

$$\underbrace{123456789 \times 123456789 \times \dots}_{7 \text{ occurrences de } 123456789} \quad ?$$

4. 7 est le premier entier tel que son inverse a une écriture décimale périodique de longueur 6 :

$$\frac{1}{7} = 0.\underbrace{142857}_{\text{142857}}\underbrace{142857}_{\text{142857}}\underbrace{142857}_{\text{142857}}\dots$$

Trouve le premier entier dont l'inverse a une écriture décimale périodique de longueur 7 :

$$\frac{1}{???} = 0.00 \underbrace{abcdefg}_{\text{période}} \underbrace{abcdefg}_{\text{période}} \dots$$

Indication. L'entier est plus grand que 230 !

5. Trouve l'unique entier :

- qui donne un quotient de 107 lorsque l'on effectue sa division (euclidienne) par 11,
- et qui donne un quotient de 90 lorsque l'on effectue sa division (euclidienne) par 13,
- et qui donne un reste égal à 6 modulo 7 !

Cours 2 (Travailler avec un éditeur).

À partir de maintenant, il est préférable que tu travailles dans un éditeur de texte dédié à Python. Tu dois alors explicitement demander d'afficher le résultat :

```
print(2+2)
print("Bonjour le monde !")
```

Cours 3 (Variables).

Variable. Une *variable* est un nom associé à un emplacement de la mémoire. C'est comme une boîte que l'on identifie par une étiquette. La commande « `a = 3` » signifie que j'ai une variable « `a` » associée à la valeur 3.

Voici un premier exemple :

```
a = 3 # Une variable
b = 5 # Une autre variable

print("La somme vaut",a+b) # Affiche la somme
print("Le produit vaut",a*b) # Affiche le produit

c = b**a # Nouvelle variable...
print(c) # ... qui s'affiche
```

Commentaires. Tout texte qui suit le caractère dièse « `#` » n'est pas exécuté par Python mais sert à expliquer le programme. C'est une bonne habitude de commenter abondamment ton code.

Noms. Il est très important de donner un nom clair et précis aux variables. Par exemple, avec les noms bien choisis tu devrais savoir ce que calcule le code suivant :

```
base = 8
hauteur = 3
aire = base * hauteur / 2
print(aire)
print(Aire) # !! Erreur !!
```

Attention ! Python distingue les majuscules des minuscules. Donc `mavariabLe`, `MavariabLe` et `MAVARIABLE` sont des variables différentes.

Réaffectation. Imaginons que tu veuilles tenir tes comptes journaliers. Tu pars d'une somme $S_0 = 1000$, le lendemain tu gagnes 100, donc maintenant $S_1 = S_0 + 100$; le jour d'après tu rajoutes 200, donc

$S_2 = S_1 + 200$; puis tu perds 50, donc au troisième jour $S_3 = S_2 - 50$. Avec Python tu peux n'utiliser qu'une seule variable S pour toutes ces opérations.

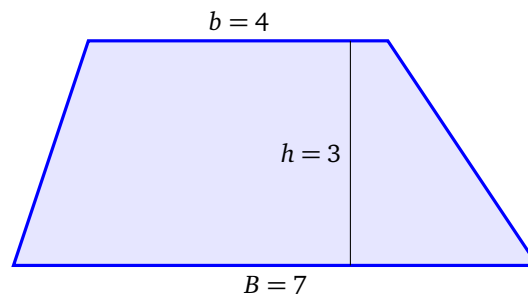
```
S = 1000
S = S + 100
S = S + 200
S = S - 50
print(S)
```

Il faut comprendre l'instruction « $S = S + 100$ » comme ceci : « je prends le contenu de la boîte S , je rajoute 100, je remets tout dans la même boîte ».

Activité 2 (Variables).

Objectifs : utiliser des variables !

1. (a) Définis des variables, puis calcule l'aire d'un trapèze. Ton programme doit afficher "L'aire vaut ..." en utilisant `print("L'aire vaut",aire)`.



- (b) Définis des variables pour calculer le volume d'une boîte (un parallélépipède rectangle) dont les dimensions sont 10, 8, 3.
- (c) Définis une variable PI qui vaut 3.14. Définis un rayon $R = 10$. Écris la formule de l'aire du disque de rayon R .
2. Remets les lignes dans l'ordre de sorte qu'à la fin x ait la valeur 46.
 - (1) $y = y - 1$
 - (2) $y = 2 * x$
 - (3) $x = x + 3 * y$
 - (4) $x = 7$
3. Tu places la somme de 1000 euros sur un compte d'épargne. Chaque année les intérêts sur l'argent placé rapportent 10% (le capital est donc multiplié par 1.10). Écris le code qui permet de calculer le capital pour les trois premières années.
4. Je définis deux variables par $a = 9$ et $b = 11$. Je souhaite échanger le contenu de a et b . Quelles instructions conviennent de sorte qu'à la fin a vaut 11 et b vaut 9 ?

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

```
c = b
a = b
b = c
```

```
c = a
a = b
b = c
```

```
c = a
a = c
c = b
b = c
```

Cours 4 (Utiliser des fonctions).• **Utiliser des fonctions de Python.**

Tu connais déjà la fonction `print()` qui affiche une chaîne de caractères (ou des nombres). Elle s'utilise ainsi `print("Coucou")` ou bien à travers une valeur :

```
chaine = "Bonjour"
print(chaine)
```

Il existe plein d'autres fonctions. Par exemple la fonction `abs()` calcule la valeur absolue : `abs(-3)` renvoie 3, `abs(5)` renvoie 5.

• **Le module math.**

Toutes les fonctions ne sont pas directement accessibles. Elles sont souvent regroupées dans des **modules**. Par exemple le module `math` contient les fonctions mathématiques. Tu y trouves par exemple la fonction racine carrée `sqrt()` (*square root*). Voici comment l'utiliser :

```
from math import *

x = sqrt(2)
print(x)
print(x**2)
```

La première ligne importe toutes les fonctions du module `math`, la seconde calcule $x = \sqrt{2}$ (en valeur approchée) et ensuite on affiche x et x^2 .

• **Sinus et cosinus.**

Le module `math` contient les fonctions trigonométriques sinus et cosinus et même la constante `pi` qui est une valeur approchée de π . Attention, les angles sont exprimés en radians.

Voici le calcul de $\sin(\frac{\pi}{2})$.

```
angle = pi/2
print(angle)
print(sin(angle))
```

• **Décimal vers entier.**

Dans le module `math` il y a aussi des fonctions pour arrondir un nombre décimal :

- `round()` arrondit à l'entier le plus proche : `round(5.6)` renvoie 6, `round(1.5)` renvoie 2.
- `floor()` renvoie l'entier inférieur ou égal : `round(5.6)` renvoie 5.
- `ceil()` renvoie l'entier supérieur ou égal : `round(5.6)` renvoie 6.

Activité 3 (Utiliser des fonctions).

Objectifs : utiliser des fonctions de Python et du module math.

1. La fonction Python pour le pgcd est `gcd(a, b)` (sans le « p », pour *greatest common divisor*). Calcule le pgcd de $a = 10\,403$ et $b = 10\,506$. Déduis-en le ppcm de a et b . La fonction ppcm n'existe pas, tu dois utiliser la formule :

$$\text{ppcm}(a, b) = \frac{a \times b}{\text{pgcd}(a, b)}.$$

2. Trouve par tâtonnement un nombre réel x qui vérifie toutes les conditions suivantes (plusieurs solutions sont possibles) :
 - `abs(x**2 - 15)` est inférieur à 0.5
 - `round(2*x)` renvoie 8
 - `floor(3*x)` renvoie 11

- `ceil(4*x)` renvoie 16

Indication. `abs()` désigne la fonction valeur absolue.

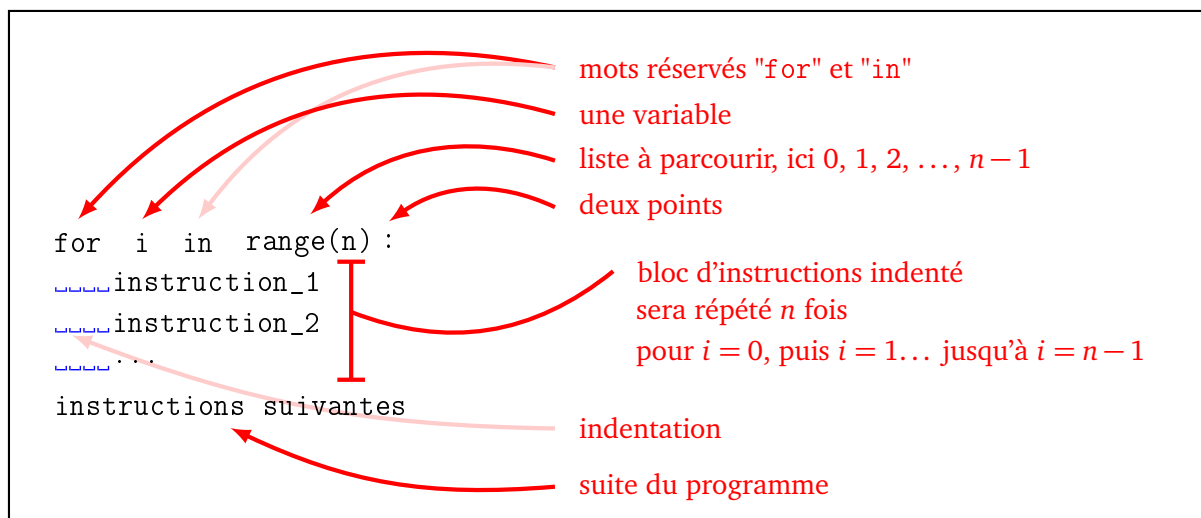
3. Tu connais la formule de trigonométrie

$$\cos^2 \theta + \sin^2 \theta = 1.$$

Vérifie que pour $\theta = \frac{\pi}{7}$ (ou d'autres valeurs) cette formule est numériquement vraie. (Ce n'est pas une preuve de la formule, car Python ne fait que des calculs approchés du sinus et du cosinus).

Cours 5 (Boucle « pour »).

La boucle « pour » est la façon la plus simple de répéter des instructions.



Note bien que ce qui délimite le bloc d'instructions à répéter, c'est **l'indentation**, c'est-à-dire les espaces placées en début de ligne qui décalent les lignes vers la droite. Toutes les lignes d'un bloc doivent avoir exactement la même indentation. Dans ce livre, nous choisissons une indentation de 4 espaces. N'oublie pas les deux points en fin de la ligne de la déclaration du `for` !

• Exemple de boucle « pour ».

Voici une boucle qui affiche les premiers carrés.

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

La seconde ligne est décalée et constitue le bloc à répéter. La variable `i` prend la valeur 0 et l'instruction affiche 0^2 ; puis `i` prend la valeur 1, et l'instruction affiche 1^2 ; puis 2^2 , $3^2 \dots$

Au final ce programme affiche :

0, 1, 4, 9, 16, 25, 36, 49, 64, 81.

Attention : la dernière valeur prise par `i` est bien 9 (et pas 10).

• Parcourir une liste quelconque.

La boucle « pour » permet de parcourir n'importe quelle liste. Voici une boucle qui affiche le cube des premiers nombres premiers.

```
for p in [2,3,5,7,11,13]:
    print(p**3)
```

- **Sommes des entiers.**

Voici un programme qui calcule

$$0 + 1 + 2 + 3 + \dots + 18 + 19.$$

```
somme = 0
for i in range(20):
    somme = somme + i
print(somme)
```

Comprends bien ce code : une variable `somme` est initialisée à 0. On va tour à tour lui ajouter 0, puis 1, puis 2... On peut mieux comprendre cette boucle en complétant un tableau :

Initialisation : `somme = 0`

i	somme
0	0
1	1
2	3
3	6
4	10
...	...
18	171
19	190

Affichage : 190

- `range()`.

— Avec `range(n)` on parcourt les entiers de 0 à $n - 1$. Par exemple `range(10)` correspond à la liste `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`.

Attention ! la liste s'arrête bien à $n - 1$ et pas à n . Ce qu'il faut retenir c'est que la liste contient bien n éléments (car elle commence à 0).

— Si tu veux afficher la liste des éléments parcourus, il faut utiliser la commande :

```
list(range(10))
```

— Avec `range(a, b)` on parcourt les éléments de a à $b - 1$. Par exemple `range(10, 20)` correspond à la liste `[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]`.

— Avec `range(a, b, pas)` on parcourt les éléments $a, a + \text{pas}, a + 2\text{pas} \dots$. Par exemple `range(10, 20, 2)` correspond à la liste `[10, 12, 14, 16, 18]`.

- **Imbrication de boucles.**

Il est possible d'imbriquer des boucles, c'est-à-dire que dans le bloc d'une boucle, on utilise une nouvelle boucle.

```
for x in [10, 20, 30, 40, 50]:
    for y in [3, 7]:
        print(x+y)
```

Dans ce petit programme x vaut d'abord 10, y prend la valeur 3 puis la valeur 7 (le programme affiche donc 13, puis 17). Ensuite $x = 20$, et y vaut de nouveau 3 puis 7 (le programme affiche donc ensuite 23, puis 27). Au final le programme affiche :

13, 17, 23, 27, 33, 37, 43, 47, 53, 57.

Activité 4 (Boucle « pour »).

Objectifs : construire des boucles simples.

- (a) Affiche les cubes des entiers de 0 à 100.
(b) Affiche les puissances quatrièmes des entiers de 10 à 20.
(c) Affiche les racines carrées des entiers 0, 5, 10, 15, ... jusqu'à 100.
- Affiche les puissances de 2, de 2^1 à 2^{10} , et apprends par cœur les résultats !
- Recherche de façon expérimentale une valeur approchée du minimum de la fonction

$$f(x) = x^3 - x^2 - \frac{1}{4}x + 1$$

sur l'intervalle $[0, 1]$.

Indications.

- Construis une boucle dans laquelle une variable i balaye les entiers de 0 à 100.
 - Définis $x = \frac{i}{100}$. Ainsi $x = 0.00$, puis $x = 0.01$, $x = 0.02$...
 - Calcule $y = x^3 - x^2 - \frac{1}{4}x + 1$.
 - Affiche les valeurs à l'aide de `print("x =", x, "y =", y)`.
 - Cherche à la main pour quelle valeur de x on obtient un y le plus petit possible.
 - N'hésite pas à modifier ton programme pour augmenter la précision.
- Cherche une valeur approchée que doit avoir le rayon R d'une boule afin que son volume soit 100 ?

Indications.

- Utilise une méthode de balayage comme à la question précédente.
- La formule du volume d'une boule est $V = \frac{4}{3}\pi R^3$.
- Affiche les valeurs à l'aide de `print("R =", R, "V =", V)`.
- Pour π tu peux prendre la valeur approchée 3.14 ou bien la valeur approchée `pi` du module `math`.

Activité 5 (Boucle « pour » (suite)).

Objectifs : construire des boucles plus compliquées.

- Définis une variable n (par exemple $n = 20$). Calcule la somme

$$1^2 + 2^2 + 3^2 + \dots + i^2 + \dots + n^2.$$

- Calcule le produit :

$$1 \times 3 \times 5 \times \dots \times 19.$$

Indications. Commence par définir une variable `produit` initialisée à la valeur 1. Utilise `range(a, b, 2)` pour obtenir un entier sur deux.

- Affiche les tables de multiplication entre 1 et 10. Voici un exemple de ligne à afficher :

$$7 \times 9 = 63$$

Utilise une commande d'affichage du style : `print(a, "x", b, "=", a*b)`.

Tortue (Scratch avec Python)

Le module turtle permet de tracer facilement des dessins en Python. Il s'agit de commander une tortue à l'aide d'instructions simples comme « avancer », « tourner »... C'est le même principe qu'avec Scratch, avec toutefois des différences : tu ne déplaces plus des blocs, mais tu écris les instructions ; et en plus les instructions sont en anglais !

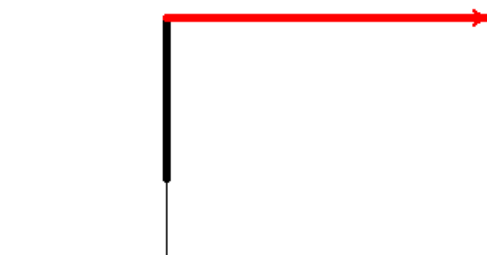
Cours 1 (La tortue Python).

La tortue c'est l'ancêtre de *Scratch* ! En quelques lignes tu peux faire de beaux dessins.

```
from turtle import *

forward(100)  # On avance
left(90)     # 90 degrés à gauche
forward(50)
width(5)     # Epaisseur du trait
forward(100)
color('red')
right(90)
forward(200)

exitonclick()
```



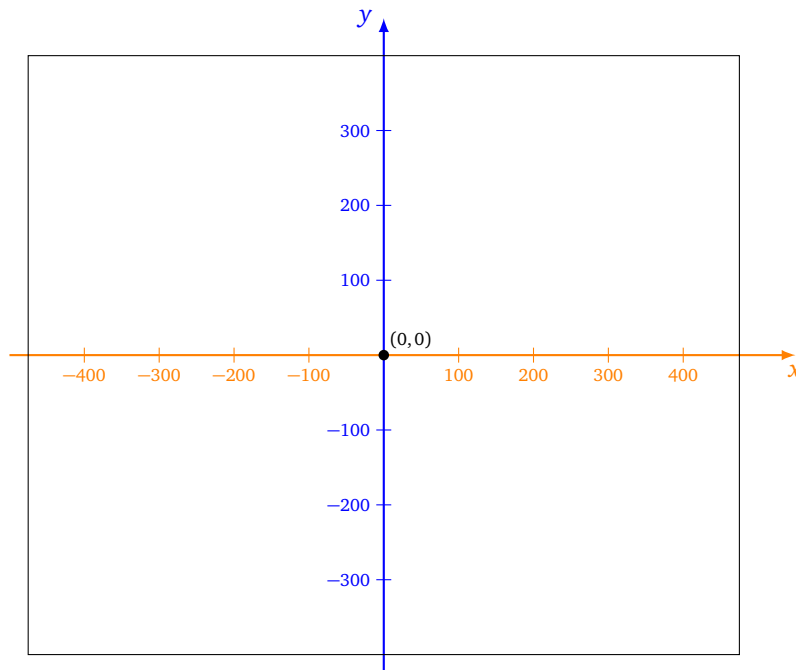
Voici une liste des principales commandes, accessibles après avoir écrit :

```
from turtle import *
```

- `forward(longueur)` avance d'un certain nombre de pas
- `backward(longueur)` recule
- `right(angle)` tourne vers la droite (sans avancer) selon un angle donné en degrés
- `left(angle)` tourne vers la gauche

- `setheading(direction)` s'oriente dans une direction (0 = droite, 90 = haut, -90 = bas, 180 = gauche)
- `goto(x,y)` se déplace jusqu'au point (x,y)
- `setx(newx)` change la valeur de l'abscisse
- `sety(newy)` change la valeur de l'ordonnée
- `down()` abaisse le stylo
- `up()` relève le stylo
- `width(epaisseur)` change l'épaisseur du trait
- `color(couleur)` change la couleur : "red", "green", "blue", "orange", "purple",...
- `position()` renvoie la position (x,y) de la tortue
- `heading()` renvoie la direction angle vers laquelle pointe la tortue
- `towards(x,y)` renvoie l'angle entre l'horizontale et le segment commençant à la tortue et finissant au point (x,y)
- `exitonclick()` termine le programme dès que l'on clique

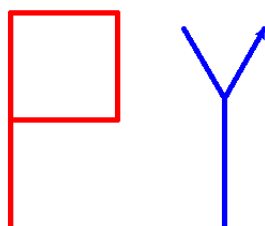
Les coordonnées de l'écran par défaut vont de -475 à +475 pour les x et de -400 à +400 pour les y ; (0,0) est au centre de l'écran.



Activité 1 (Premiers pas).

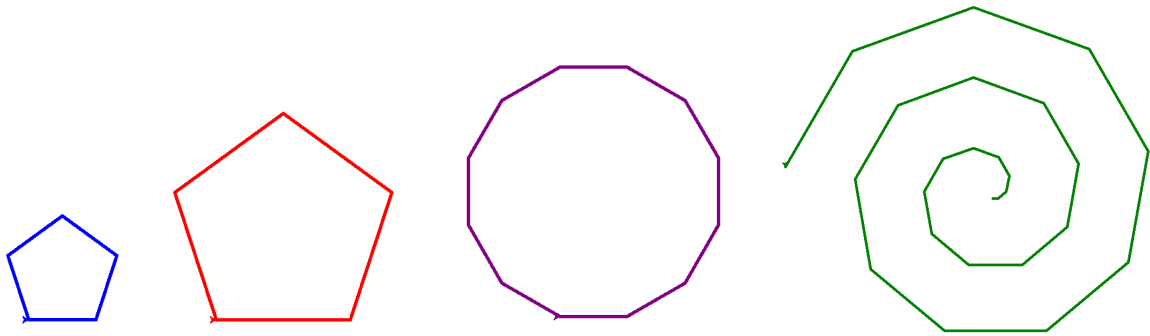
Objectifs : tracer tes premiers dessins.

Trace les premières lettres de Python, par exemple comme ci-dessous.



Activité 2 (Figures).

Objectifs : tracer des figures géométriques.



1. **Pentagone.** Trace un premier pentagone (en bleu). Tu dois répéter 5 fois : avancer de 100 pas, tourner de 72 degrés.

Indication. Pour construire une boucle utilise

```
for i in range(5):
```

(même si tu n'utilises pas ensuite la variable i).

2. **Pentagone (bis).** Définis une variable longueur qui vaut 200 et une variable angle qui vaut 72 degrés. Trace un second pentagone (en rouge) en avançant cette fois de longueur et en tournant de angle.

3. **Dodécagone.** Trace un polygone à 12 côtés (en violet).

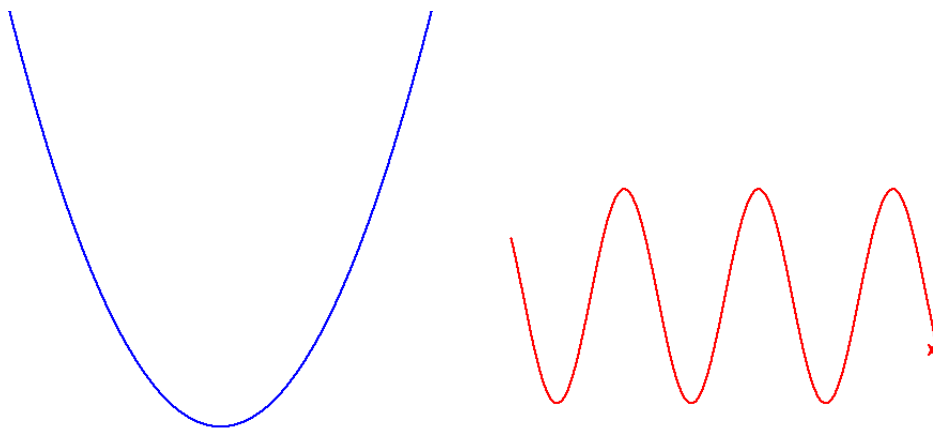
Indication. Pour tracer un polygone à n côtés, il faut tourner d'un angle de $360/n$ degrés.

4. **Spirale.** Trace une spirale (en vert).

Indication. Construis une boucle, dans laquelle tu tournes toujours du même angle, mais par contre tu avances d'une longueur qui augmente à chaque étape.

Activité 3 (Graphe de fonctions).

Objectifs : tracer le graphe d'une fonction.



Trace le graphe de la fonction carré et de la fonction sinus.

Afin d'obtenir une courbe dans la fenêtre de la tortue, répète pour x variant de -200 à $+200$:

- poser $y = \frac{1}{100}x^2$,

- aller à (x, y) .

Pour la sinusoïde, tu peux utiliser la formule

$$y = 100 \sin\left(\frac{1}{20}x\right).$$

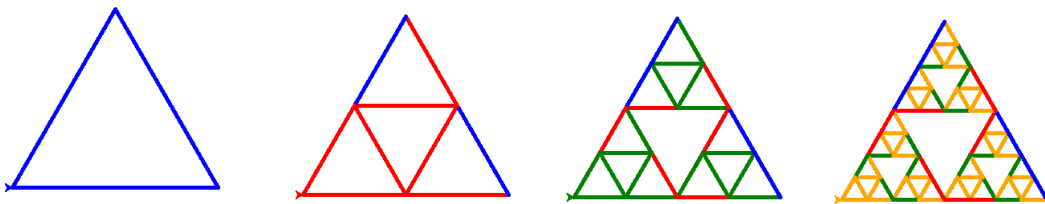
Par défaut Python ne connaît pas la fonction sinus, pour utiliser `sin()` il faut d'abord importer le module `math` :

```
from math import *
```

Pour que la tortue avance plus vite, tu peux utiliser la commande `speed("fastest")`.

Activité 4 (Triangle de Sierpinski).

Objectifs : tracer le début de la fractale de Sierpinski en imbriquant des boucles.



Voici comment tracer le second dessin. Analyse l'imbrication des boucles et trace les dessins suivants.

```
for i in range(3):
    color("blue")
    forward(256)
    left(120)

for i in range(3):
    color("red")
    forward(128)
    left(120)
```

Activité 5 (Le cœur des tables de multiplication).

Objectifs : dessiner les tables de multiplication. Pour une introduction en vidéo voir [la face cachée des tables de multiplication](#) sur Youtube par Micmaths.

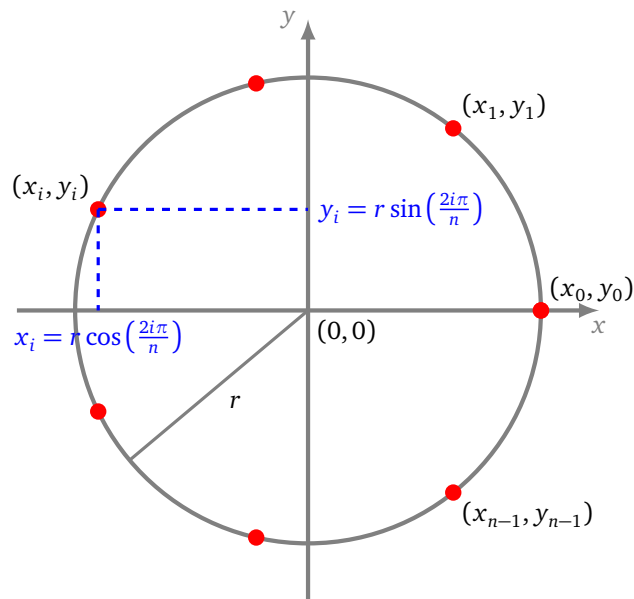
On fixe un entier n . On étudie la table de 2, c'est-à-dire que l'on calcule 2×0 , 2×1 , 2×2 , jusqu'à $2 \times (n-1)$. En plus les calculs se feront modulo n . On calcule donc

$$2 \times k \pmod{n} \quad \text{pour } k = 0, 1, \dots, n-1$$

Comment dessiner cette table ?

On place sur un cercle n points numérotés de 0 à $n-1$. Pour chaque $k \in \{0, \dots, n-1\}$, on relie le point numéro k et le point numéro $2 \times k \pmod{n}$ par un segment.

Voici le tracé, de la table de 2, modulo $n = 10$.



Cours 2 (Plusieurs tortues).

On peut définir plusieurs tortues qui se déplacent de façon indépendante chacune de leur côté. Voici comment définir deux tortues (une rouge et une bleue) et les déplacer.

```
tortue1 = Turtle()    # Avec un 'T' majuscule !
tortue2 = Turtle()

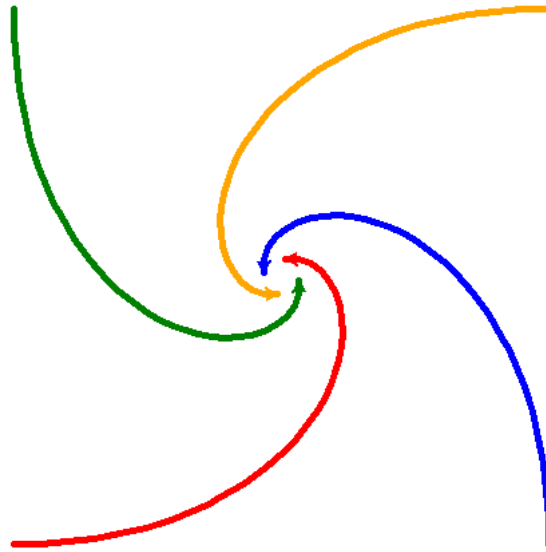
tortue1.color('red')
tortue2.color('blue')

tortue1.forward(100)
tortue2.left(90)
tortue2.forward(100)
```

Activité 6 (La poursuite des tortues).

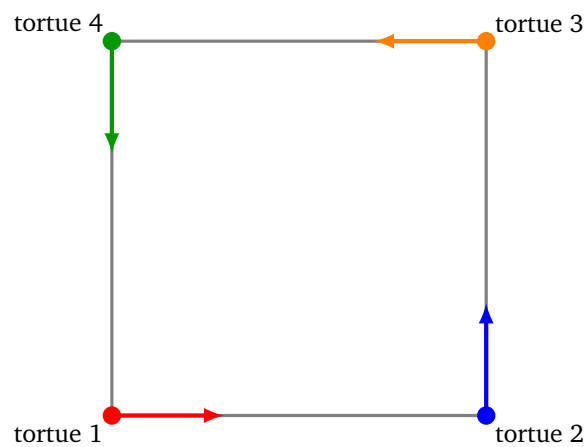
Objectifs : tracer des courbes de poursuite.

Programme quatre tortues qui courent les unes après les autres :



- la tortue 1 court après la tortue 2,
- la tortue 2 court après la tortue 3,
- la tortue 3 court après la tortue 4,
- la tortue 4 court après la tortue 1.

Voici les positions et orientations de départ :



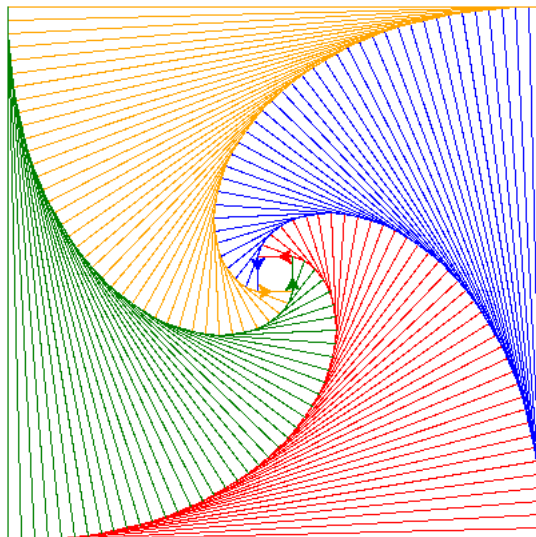
Indications. Utilise le bout de code suivant :

```
position1 = tortue1.position()
position2 = tortue2.position()
angle1 = tortue1.towards(position2)
tortue1.setheading(angle1)
```

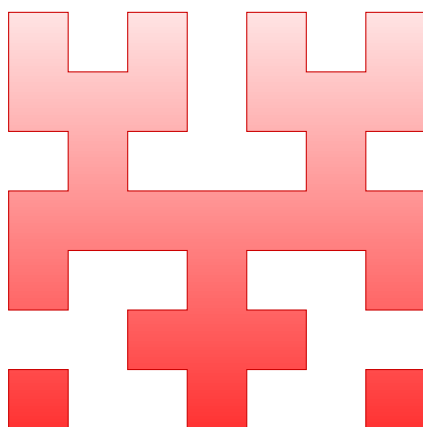
- Tu places les tortues aux quatre coins d'un carré, par exemple en $(-200, -200)$, $(200, -200)$, $(200, 200)$ et $(-200, 200)$.
- Tu récupères la position de la première tortue par `position1 = tortue1.position()`. Idem pour les autres tortues.
- Tu calcules l'angle entre la tortue 1 et la tortue 2 par la commande `angle1 = tortue1.towards(position2)`.

- Tu orientes la tortue 1 selon cet angle : `tortue1.setheading(angle1)`.
- Tu avances la tortue 1 de 10 pas.

Améliore ton programme en traçant à chaque fois un segment entre la tortue poursuivante et la tortue poursuivie.



DEUXIÈME PARTIE



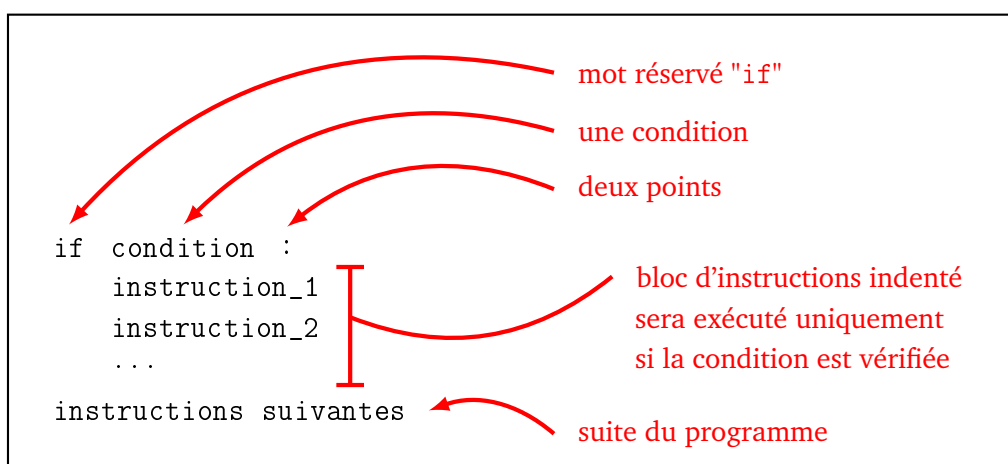
FONDAMENTAUX

Si ... alors ...

L'ordinateur peut réagir en fonction d'une situation. Si une condition est remplie il agit d'une certaine façon, sinon il fait autre chose.

Cours 1 (Si ... alors ...).

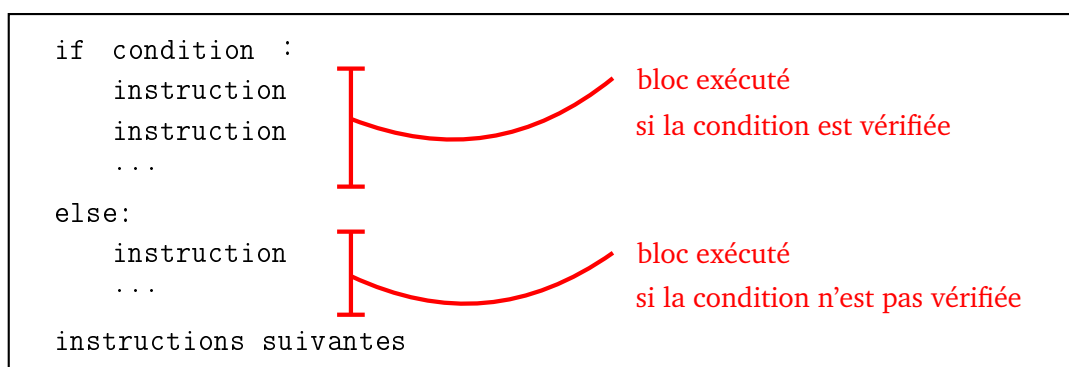
Voici comment utiliser le test « if » avec Python :



Voici un exemple, qui avertit un conducteur si une variable vitesse est trop grande.

```
if vitesse > 110:  
    print("Attention, tu roules trop vite.")
```

On peut aussi exécuter des instructions si la condition n'est pas remplie à l'aide du mot « else ».



Encore une fois c'est l'indentation qui délimite les différents blocs d'instructions. Voici un exemple qui affiche le signe d'un nombre x.


```

if x >= 0:
    print("Le nombre est positif.")
else:
    print("Le nombre est négatif.")

```

Cours 2 (Entrée au clavier).

Pour pouvoir interagir avec l'utilisateur, tu peux lui demander de saisir un texte au clavier. Voici un petit programme qui demande le prénom et l'âge de l'utilisateur et affiche un message du style « Bonjour Kevin » puis « Tu es mineur/majeur ! » selon l'âge.

```

prenom = input("Comment t'appelles-tu ? ")
print("Bonjour", prenom)

age_chaine = input("Quel âge as-tu ? ")
age = int(age_chaine)

if age >= 18:
    print("Tu es majeur !")
else:
    print("Tu es mineur !")

```

Explications.

- La commande `input()` met en pause l'exécution du programme et attend de l'utilisateur un texte (qu'il termine en appuyant sur la touche « Entrée »).
- Cette commande renvoie une chaîne de caractères.
- Si on veut un entier, il faut convertir la chaîne. Par exemple, ici `age_chaine` peut valoir "17" (ce n'est pas un nombre mais une suite de caractères), alors que `int(age_chaine)` vaut maintenant l'entier 17.
- L'opération inverse est aussi possible, `str()` convertit un nombre en une chaîne. Par exemple `str(17)` renvoie la chaîne "17"; si `age = 17`, alors `str(age)` renvoie également "17".

Cours 3 (Le module « random »).

Le module `random` génère des nombres comme s'ils étaient tirés au hasard.

- Voici la commande à placer au début du programme pour appeler ce module :


```
from random import *
```
- La commande `randint(a, b)` renvoie un entier au hasard compris entre a et b .
Par exemple après l'instruction `n = randint(1, 6)`, n est un entier tiré au hasard avec $1 \leq n \leq 6$.
Si on recommence l'instruction `n = randint(1, 6)`, n prend une nouvelle valeur. C'est comme si on effectuait le lancer d'un dé à 6 faces.
- La commande `random()`, sans argument, renvoie un nombre flottant (c'est-à-dire un nombre à virgule) compris entre 0 et 1. Par exemple, après l'instruction `x = random()`, alors x est un nombre flottant avec $0 \leq x < 1$.

Cours 4 (Booléens).

- Un **booléen** est une donnée qui vaut soit la valeur « vrai », soit la valeur « faux ». En Python les valeurs sont `True` et `False` (avec une majuscule).
- On obtient un booléen par exemple comme résultat de la comparaison de deux nombres. Par exemple `7 < 4` vaut `False` (car 7 n'est pas plus petit que 4). Vérifie que `print(7 < 4)` affiche `False`.

Voici les principales comparaisons :

- **Test d'égalité** : `a == b`
- **Test inférieur strict** : `a < b`
- **Test inférieur large** : `a <= b`
- **Test supérieur** : `a > b` ou `a >= b`
- **Test non égalité** : `a != b`

Par exemple `6*7 == 42` vaut `True`.

•

ATTENTION ! L'erreur classique est de confondre « `a = b` » et « `a == b` ».

- **Affectation.** `a = b` met le contenu de la variable `b` dans la variable `a`.
- **Test d'égalité.** `a == b` teste si les contenus de `a` et de `b` sont égaux et vaut `True` ou `False`.

- On peut comparer autre chose que des nombres. Par exemple « `car == "A"` » teste si la variable `car` vaut `"A"`; « `il_pleut == True` » teste si la variable `il_pleut` est vraie...
 - Les booléens sont utiles dans le test « si ... alors ... » et dans les boucles « tant que ... alors ... ».
 - **Opérations entre les booléens.** Si `P` et `Q` sont deux booléens, on peut définir de nouveaux booléens.
 - **Et logique.** « `P and Q` » est vrai si et seulement si `P` et `Q` sont vrais.
 - **Ou logique.** « `P or Q` » est vrai si et seulement si `P` ou `Q` est vrai.
 - **Négation.** « `not P` » est vrai si et seulement si `P` est faux.
- Par exemple « `(2+2 == 2*2) and (5 < 3)` » renvoie `False`, car même si on a bien $2 + 2 = 2 \times 2$, l'autre condition n'est pas remplie car $5 < 3$ est faux.

Activité 3 (Chiffres d'un nombre).

Objectifs : trouver des nombres dont les chiffres vérifient certaines propriétés.

1. Le programme suivant affiche tous les entiers de 0 à 99. Comprends ce programme. Que représentent les variables `u` et `d` ?

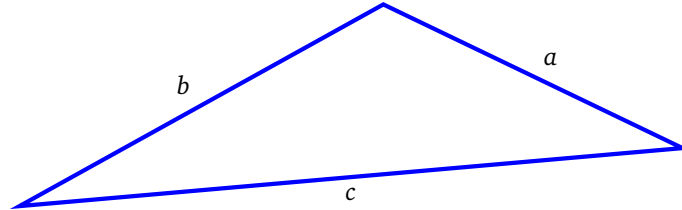
```
for d in range(10):
    for u in range(10):
        n = 10*d + u
        print(n)
```

2. Trouve tous les entiers compris entre 0 et 999 qui vérifient toutes les propriétés suivantes :
 - l'entier se termine par 3,
 - la somme des chiffres est supérieure ou égale à 15,
 - le chiffre des dizaines est pair.
3. Modifie ton programme précédent pour compter et afficher le nombre d'entiers vérifiant les propriétés.

Activité 4 (Triangles).

Objectifs : déterminer les propriétés d'un triangle à partir des trois longueurs des côtés.

On se donne trois longueurs a , b et c . Tu vas déterminer les propriétés du triangle dont les longueurs seraient a , b , c .



Définis trois variables a , b et c avec des valeurs entières et $a \leq b \leq c$ (ou bien demande à l'utilisateur trois valeurs).

1. **Ordre.** Demande à Python de tester si les longueurs vérifient bien $a \leq b \leq c$. Affiche une phrase pour la réponse.
2. **Existence.** Il existe un triangle correspondant à ces longueurs si et seulement si :

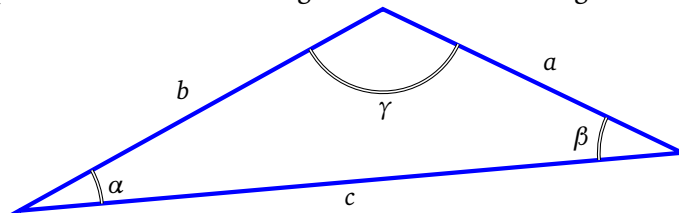
$$a + b \geq c.$$

Demande à Python de tester si c'est le cas et affiche la réponse.

3. **Triangle rectangle.** Demande à Python de tester si le triangle est un triangle rectangle. (Pense au théorème de Pythagore.)
4. **Triangle équilatéral.** Teste si le triangle est équilatéral.
5. **Triangle isocèle.** Teste si le triangle est isocèle.
6. **Angles aigus.** Teste si tous les angles sont aigus (c'est-à-dire inférieurs ou égaux à 90 degrés).

Indications.

- La loi des cosinus permet de calculer un angle en fonction des longueurs :



$$\cos \alpha = \frac{-a^2 + b^2 + c^2}{2bc}, \quad \cos \beta = \frac{a^2 - b^2 + c^2}{2ac}, \quad \cos \gamma = \frac{a^2 + b^2 - c^2}{2ab}.$$

- Pour tester si l'angle α est aigu il suffit de vérifier $\cos \alpha \geq 0$ (au final on ne calcule jamais α , mais juste $\cos \alpha$).

Trouve des exemples de longueurs a , b , c pour illustrer les différentes propriétés.

Activité 5 (Le nombre mystère).

Objectifs : coder le jeu incontournable lorsque l'on apprend à programmer. L'ordinateur choisit un nombre au hasard. L'utilisateur doit deviner ce nombre en suivant des indications « plus grand » ou « plus petit » données par l'ordinateur. Comme ce jeu est vite lassant, on introduit des variantes où l'ordinateur a le droit de mentir ou de tricher !

1. Le jeu classique.

- L'ordinateur choisit au hasard un nombre mystère entre 0 et 99.
- Le joueur propose une réponse.
- L'ordinateur répond « le nombre à trouver est plus grand » ou « le nombre à trouver est plus petit » ou « bravo, c'est le bon nombre ! ».
- Le joueur a sept tentatives pour trouver la bonne réponse.

Programme ce jeu !

Indications. Pour quitter une boucle `for` avant la dernière proposition, tu peux utiliser la commande `break`. Utilise ceci lorsque le joueur trouve la bonne réponse.

2. L'ordinateur ment.

Pour compliquer le jeu, l'ordinateur a le droit de mentir de temps en temps. Par exemple environ une fois sur quatre l'ordinateur donne la mauvaise indication « plus grand » ou « plus petit ».

Indications. Pour décider quand l'ordinateur ment, à chaque tour tire un nombre au hasard entre 1 et 4, si c'est 4 l'ordinateur ment !

3. L'ordinateur triche.

Maintenant l'ordinateur triche (mais il ne ment plus) ! À chaque tour l'ordinateur change un peu le nombre mystère à trouver.

Indications. À chaque tour, tire un nombre au hasard, entre -3 et $+3$ par exemple, et ajoute-le au nombre mystère. (Attention à ne pas dépasser les bornes 0 et 99.)

Fonctions

Écrire une fonction, c'est la façon la plus simple de regrouper du code pour une tâche bien particulière, dans le but de l'exécuter une ou plusieurs fois par la suite.

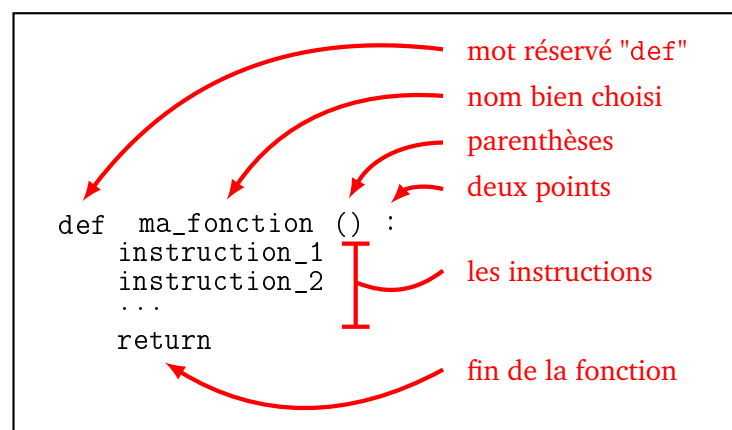
Cours 1 (Fonction (début)).

Une fonction en informatique est une portion réalisant un tâche bien précise et qui pourra être utilisée une ou plusieurs fois dans la suite du programme. Pour définir une fonction avec Python, c'est très simple. Voici deux exemples :

```
def dit_bonjour():  
    print("Bonjour le monde !")  
    return
```

```
def affiche_carres():  
    for i in range(20):  
        print(i**2)  
    return
```

Les instructions sont regroupées dans un bloc indenté. Le mot `return` (optionnel) indique la fin de la fonction. Ces instructions ne sont exécutées que si j'appelle la fonction. Par exemple, chaque fois que j'exécute la commande `dit_bonjour()`, Python affiche la phrase « Bonjour le monde ! ». Chaque fois que j'exécute la commande `affiche_carres()`, Python affiche 0, 1, 4, 9, 16, ..., c'est-à-dire les nombres i^2 pour $i = 0, \dots, 19$.



Cours 2 (Fonction (suite)).

Les fonctions informatiques acquièrent tout leur potentiel avec :

- une **entrée**, qui regroupe des variables qui servent de **paramètres**,
- une **sortie**, qui est un résultat renvoyé par la fonction (et qui souvent dépendra des paramètres d'entrée).

Voici deux exemples :

```
def affiche_mois(numero):  
    if numero == 1:  
        print("Nous sommes en janvier.")  
    if numero == 2:  
        print("Nous sommes en février.")  
    if numero == 3:  
        print("Nous sommes en mars.")  
    # etc.  
    return
```

Lorsqu'elle est appelée cette fonction affiche le nom du mois en fonction du nombre fourni en entrée. Par exemple `affiche_mois(3)` va afficher "Nous sommes en mars."

```
def calcule_cube(a):  
    cube = a * a * a    # ou bien a**3  
    return cube
```

Cette fonction calcule le cube d'un nombre, par exemple `calcule_cube(2)` n'affiche rien mais renvoie la valeur 8. Cette valeur peut être utilisée ailleurs dans le programme. Par exemple, que font les instructions suivantes ?

```
x = 3  
y = 4  
z = calcule_cube(x) + calcule_cube(y)  
print(z)
```

En terme mathématiques, on pose $x = 3$, $y = 4$, puis on calcule le cube de x , le cube de y et on les additionne :

$$z = x^3 + y^3 = 3^3 + 4^3 = 27 + 64 = 91$$

Ainsi le programme affiche 91.

```
# Définition de la fonction  
def ma_fonction (param) :  
    instruction_1  
    instruction_2  
    ...  
    return resultat  
  
# Appel de la fonction  
x = 7  
val = ma_fonction (x)
```

The diagram illustrates the components of a function definition and call. Red arrows point from labels to specific parts of the code:

- un paramètre**: Points to the parameter `param` in the function definition.
- renvoie un résultat**: Points to the `return` statement in the function definition.
- argument**: Points to the argument `x` in the function call.
- appel de la fonction**: Points to the function call `ma_fonction (x)`.
- résultat renvoyé**: Points to the variable `val` that receives the return value.

Les avantages de la programmation utilisant des fonctions sont les suivants :

- on écrit le code d'une fonction une seule fois, mais on peut appeler la fonction plusieurs fois ;
- en divisant notre programme en petits blocs ayant chacun leur utilité propre, le programme est plus facile à écrire, à lire, à corriger et à modifier ;
- on peut utiliser une fonction (écrite par quelqu'un d'autre, comme par exemple la fonction `sqrt()`) sans connaître tous les détails internes de sa programmation.

Activité 1 (Premières fonctions).

Objectifs : écrire des fonctions très simples.

1. Fonction sans paramètre ni sortie.

- Programme une fonction appelée `affiche_table_de_7()` qui affiche la table de multiplication par 7 : $1 \times 7 = 7$, $2 \times 7 = 14$...
- Programme une fonction appelée `affiche_bonjour()` qui demande à l'utilisateur son prénom et affiche ensuite « Bonjour » suivi du prénom de l'utilisateur.

Indication. Utilise `input()`.

2. Fonction avec un paramètre et sans sortie.

- Programme une fonction appelée `affiche_une_table(n)` qui dépend d'un paramètre `n` et qui affiche la table de multiplication par l'entier `n`. Par exemple, la commande `affiche_une_table(5)` doit afficher : $1 \times 5 = 5$, $2 \times 5 = 10$...
- Programme une fonction appelée `affiche_salutation(formule)` qui dépend d'un paramètre `formule`. Cette fonction demande le prénom de l'utilisateur et affiche une formule de salutation suivi du prénom. Par exemple `affiche_salutation("Coucou")` afficherait « Coucou » suivi du prénom donné par l'utilisateur.

3. Fonction sans paramètre et avec sortie.

Programme une fonction appelée `demande_prenom_nom()` qui demande d'abord le prénom de l'utilisateur, puis son nom et renvoie comme résultat l'identité complète avec le nom en majuscule. Par exemple, si l'utilisateur saisi « Dark » puis « Vador », la fonction renvoie la chaîne "Dark VADOR" (la fonction n'affiche rien).

Indications.

- Si `chaine` est une chaîne de caractères, alors `chaine.upper()` est la chaîne transformée avec les caractères en majuscules. Exemple : si `chaine = "Vador"` alors `chaine.upper()` renvoie "VADOR".
- On peut fusionner deux chaînes en utilisant le signe « + ». Exemple : "Dark" + "Vador" vaut "DarkVador". Autre exemple : si `chaine1 = "Dark"` et `chaine2 = "Vador"` alors `chaine1 + " " + chaine2` vaut "Dark Vador".

Cours 3 (Fonction (suite et fin pour l'instant)).

Une fonction peut avoir plusieurs paramètres et renvoyer plusieurs résultats. Par exemple, voici une fonction qui calcule et renvoie la somme et le produit de deux nombres donnés en entrée.

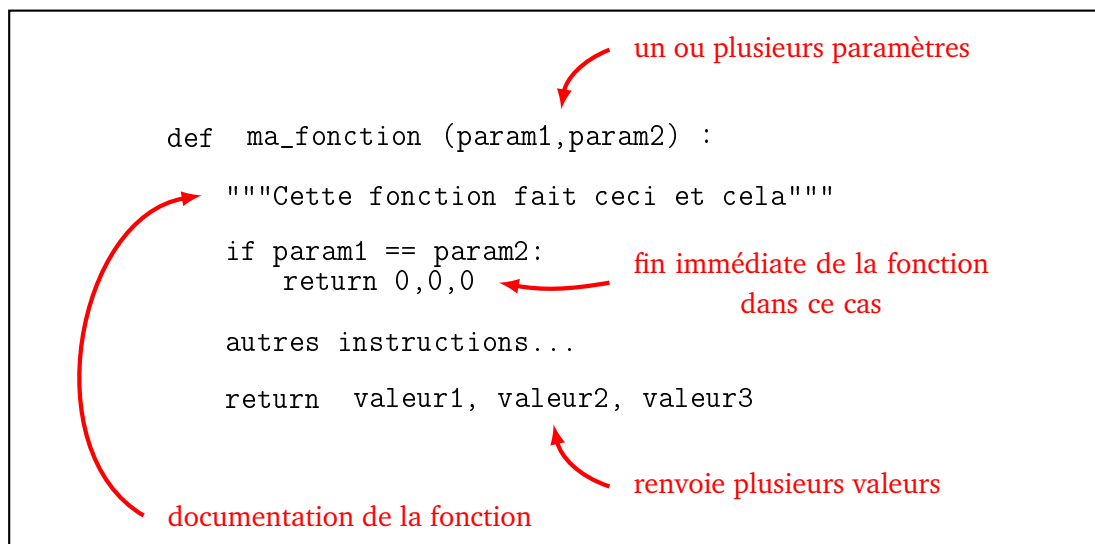
```
def somme_produit(a,b):
    """Calcule la somme et le produit de deux nombres"""
    s = a + b
```



```
p = a * b  
return s, p
```

```
som, pro = somme_produit(6,7)
```

La dernière ligne appelle la fonction avec les arguments 6 (pour le paramètre a) et 7 (pour le paramètre b). Cette fonction renvoie deux valeurs, la première est affectée à som (qui vaut donc ici 13) et la seconde à pro (qui vaut donc 42).



Retenons donc :

- Il peut y avoir plusieurs paramètres en entrée.
- Il peut y avoir plusieurs résultats en sortie.
- Très important ! Il ne faut pas confondre afficher et renvoyer une valeur. L'affichage (par la commande `print()`) affiche juste quelque chose à l'écran. La plupart des fonctions n'affichent rien, mais renvoient une valeur (ou plusieurs). C'est beaucoup plus utile car cette valeur peut être utilisée ailleurs dans le programme.
- Dès que le programme rencontre l'instruction `return`, la fonction s'arrête et renvoie le résultat. Il peut y avoir plusieurs fois l'instruction `return` dans une fonction mais une seule sera exécutée. On peut aussi ne pas mettre d'instruction `return` si la fonction ne renvoie rien.
- Dans les instructions d'une fonction, on peut bien sûr faire appel à d'autres fonctions !
- Il est important de bien commenter tes programmes. Pour documenter une fonction, tu peux décrire ce qu'elle fait en commençant par un *docstring*, c'est-à-dire une description (en français) entourée par trois guillemets : `""" Ma fonction fait ceci et cela. """` à placer juste après l'entête.
- Lorsque l'on définit une fonction, les variables qui apparaissent entre les parenthèses sont appelées les **paramètres** ; par contre, lorsque l'on appelle la fonction, les valeurs entre les parenthèses sont appelées les **arguments**. Il y a bien sûr une correspondance entre les deux.

Activité 2 (Encore des fonctions).

Objectifs : construire des fonctions avec différents types d'entrée et de sortie.

1. Trinômes.

- (a) Écris une fonction `trinome_1(x)` qui dépend d'un paramètre `x` et qui renvoie la valeur du

trinôme $3x^2 - 7x + 4$. Par exemple `trinome_1(7)` renvoie 102.

- (b) Écris une fonction `trinome_2(a,b,c,x)` qui dépend de quatre paramètres a , b , c et x et qui renvoie la valeur du trinôme $ax^2 + bx + c$. Par exemple `trinome_2(2,-1,0,6)` renvoie 66.

2. Devises.

- (a) Écris une fonction `conversion_euros_vers_dollars(montant)` qui dépend d'un paramètre et qui pour une somme d'argent `montant`, exprimée en euros, renvoie sa valeur en dollars (tu prendras par exemple 1 euro = 1,15 dollar).
- (b) Écris une fonction `conversion_euros(montant,devise)` qui dépend d'un paramètre `montant` et d'une monnaie `devise` et qui convertit la somme `montant` donnée en euros, dans la devise souhaitée. Exemples de devises : 1 euro = 1,15 dollar ; 1 euro = 0,81 livre ; 1 euro = 130 yens. Par exemple `conversion_euros(100,"livre")` renvoie 81.

Prends soin de donner un nom intelligible à tes fonctions ainsi qu'aux variables. N'oublie pas de documenter chaque fonction en ajoutant un petit texte explicatif entre triples guillemets au tout début de ta fonction.

3. Volumes.

Construis des fonctions qui calculent et renvoient des volumes :

- le volume d'un cube en fonction de la longueur d'un côté,
- le volume d'une boule en fonction de son rayon,
- le volume d'un cylindre en fonction du rayon de sa base et de sa hauteur,
- le volume d'une boîte parallélépipède rectangle en fonction de ses trois dimensions.

Pour la valeur de π , tu prendras soit la valeur approchée 3.14, soit la valeur approchée fournie par la constante `pi` du module `math`.

4. Périmètres et aires.

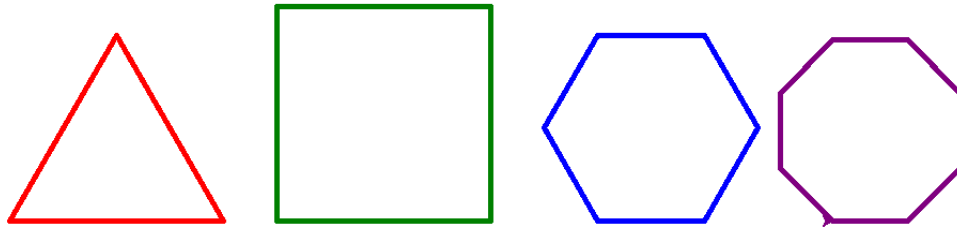
- (a) Écris une fonction dont l'usage est `perimetre_aire_rectangle(a,b)` et qui renvoie en sortie le périmètre et l'aire d'un rectangle de dimensions a et b .
- (b) Même question avec `perimetre_aire_disque(r)` pour le périmètre et l'aire d'un disque de rayon r .
- (c) Utilise ta fonction précédente pour conjecturer à partir de quel rayon, l'aire d'un disque est plus grande que le périmètre de ce disque.

Indication. Si tu veux balayer les rayons en incrémentant la valeur de 0.1 à chaque fois, tu peux construire une boucle ainsi :

```
for rayon in range(0,30):
    puis faire un appel à la fonction par perimetre_aire_disque(rayon/10).
```

Activité 3 (Tortue).

Objectifs : définir quelques fonctions qui dessinent des figures géométriques. Créer une fonction est similaire à créer un bloc avec Scratch.



1. Programme une fonction `triangle()` qui dessine un triangle (en rouge, chaque côté mesurant 200).
2. Programme une fonction `carre()` qui dessine un carré (en vert, chaque côté mesurant 200). Utilise une boucle « pour » afin de ne pas avoir à réécrire les mêmes instructions plusieurs fois.
3. Programme une fonction `hexagone(longueur)` qui trace un hexagone (en bleu) d'une longueur de côté donnée (l'angle pour tourner est de 60 degrés).
4. Programme une fonction `polygone(n, longueur)` qui trace un polygone régulier à n côtés et d'une longueur de côté donnée (l'angle pour tourner est alors de $360/n$ degrés).

Activité 4 (Toujours des fonctions).

Objectifs : créer de nouvelles fonctions.

1. (a) Voici la réduction pour le prix d'un billet de train en fonction de l'âge du voyageur :

- réduction de 50% pour les moins de 10 ans ;
- réduction de 30% pour les 10 à 18 ans ;
- réduction de 20% pour les 60 ans et plus.

Écris une fonction `reduction()` qui renvoie la réduction en fonction de l'âge et dont les propriétés sont rappelées dans le cadre ci-dessous :

`reduction()`

Usage : `reduction(age)`

Entrée : un entier correspondant à l'âge

Sortie : un entier correspondant à la réduction

Exemples :

- `reduction(17)` renvoie 30.
- `reduction(23)` renvoie 0.

- (b) Déduis-en une fonction `montant()` qui calcule le montant à payer en fonction du tarif normal et de l'âge du voyageur.

`montant()`

Usage : `montant(tarif_normal, age)`

Entrée : un nombre `tarif_normal` correspondant au prix sans réduction et `age` (un entier)

Sortie : un nombre correspondant au montant à payer après réduction

Remarque : utilise la fonction `reduction()`

Exemple : `montant(100, 17)` renvoie 70.

Une famille achète des billets pour différents trajets, voici le tarif normal de chaque trajet et les âges des voyageurs :

- tarif normal 30 euros, enfant de 9 ans ;
- tarif normal 20 euros, pour chacun des jumeaux de 16 ans ;
- tarif normal 35 euros, pour chacun des parents de 40 ans.

Quel est le montant total payé par la famille ?

2. On souhaite programmer un petit quiz sur les tables de multiplication.

- (a) Programme une fonction `calcul_est_exact()` qui décide si la réponse donnée à une multiplication est juste ou pas.

`calcul_est_exact()`

Usage : `calcul_est_exact(a,b,reponse)`

Entrée : trois entiers, `reponse` étant la réponse proposée au calcul de $a \times b$.

Sortie : « vrai » ou « faux », selon que la réponse est correcte ou pas

Exemples :

- `calcul_est_exact(6,7,35)` renvoie `False`.
- `calcul_est_exact(6,7,42)` renvoie `True`.

- (b) Programme une fonction qui affiche une multiplication, demande une réponse et affiche une petite phrase de conclusion. Tout cela en français ou en anglais !

`test_multiplication()`

Usage : `test_multiplication(a,b,lang)`

Entrée : deux entiers, la langue choisie (parmi "français" ou "anglais")

Sortie : rien

Remarque : utilise la fonction `calcul_est_exact()`

Exemple : `test_multiplication(6,7,"anglais")` demande, en anglais, la réponse au calcul 6×7 et répond si c'est correct ou pas.

Bonus. Améliore ton programme afin que l'ordinateur propose tout seul des opérations aléatoires au joueur. (Utilise la fonction `randint()` du module `random`.)

Activité 5 (Égalité expérimentale).

Objectifs : utiliser l'ordinateur pour expérimenter des égalités de fonctions.

- (a) Construis une fonction `valeur_absolue(x)` qui renvoie la valeur absolue d'un nombre (sans utiliser la fonction `abs()` de Python!).
- (b) Construis une fonction `racine_du_carre(x)` qui correspond au calcul de $\sqrt{x^2}$.
- (c) On dit que deux fonctions (d'une variable) f et g sont **expérimentalement égales** si $f(i) = g(i)$ pour $i = -100, -99, \dots, 0, 1, 2, \dots, 100$. Vérifie par ordinateur que les deux fonctions définies par

$$|x| \quad \text{et} \quad \sqrt{x^2}$$

sont expérimentalement égales.

- (a) Construis une fonction à deux paramètres $F1(a,b)$ qui renvoie $(a+b)^2$. Même chose avec $F2(a,b)$ qui renvoie $a^2 + 2ab + b^2$.

(b) On dit que deux fonctions de deux variables F et G sont **expérimentalement égales** si $F(i, j) = G(i, j)$ pour tout $i = -100, -99, \dots, 100$ et pour tout $j = -100, -99, \dots, 100$. Vérifie par ordinateur que les fonctions définies par $(a + b)^2$ et $a^2 + 2ab + b^2$ sont expérimentalement égales.

(c) Je sais que l'une des deux égalités suivantes est vraie :

$$(a - b)^3 = a^3 - 3a^2b - 3ab^2 + b^3 \quad \text{ou} \quad (a - b)^3 = a^3 - 3a^2b + 3ab^2 - b^3.$$

Aide-toi de l'ordinateur pour décider laquelle c'est !

3. (a) Construis une fonction `sincos(x)` qui renvoie $(\sin(x))^2 + (\cos(x))^2$ et une autre `un(x)` qui renvoie toujours 1. Ces deux fonctions sont-elles expérimentalement égales (au sens de la première question) ? Cherche quelle peut être la cause de cette réponse.

(b) On pose $\epsilon = 0.00001$. On dit que deux fonctions (d'une variable) f et g sont **expérimentalement approximativement égales** si $|f(i) - g(i)| \leq \epsilon$ pour $i = -100, -99, \dots, 100$. Est-ce que maintenant les deux fonctions définies par `sincos(x)` et `un(x)` vérifient ce critère ?

(c) Vérifie de façon expérimentale et approchée les égalités :

$$\sin(2x) = 2 \sin(x) \cos(x), \quad \cos\left(\frac{\pi}{2} - x\right) = \sin(x).$$

(d) **Bonus. Un contre-exemple.** Montre que les fonctions définies par $g_1(x) = \sin(\pi x)$ et $g_2(x) = 0$ sont expérimentalement égales (avec notre définition donnée plus haut). Mais montre aussi que l'on n'a pas $g_1(x) = g_2(x)$ pour tout $x \in \mathbb{R}$.

Cours 4 (Variable locale).

Voici une fonction toute simple qui prend en entrée un nombre et renvoie le nombre augmenté de un.

```
def ma_fonction(x):
    x = x + 1
    return x
```

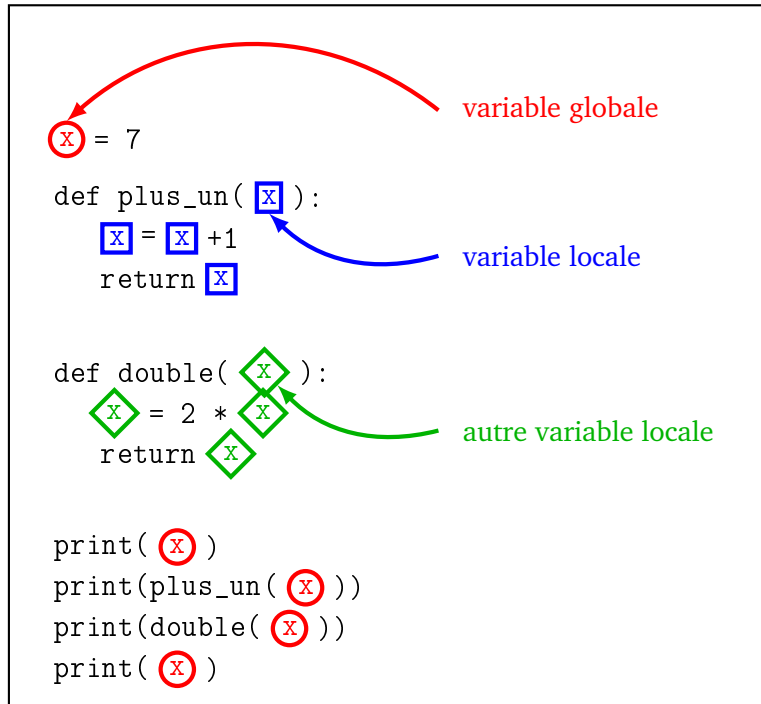
- Bien évidemment `ma_fonction(3)` renvoie 4.
- Si je définis une variable par `y = 5` alors `ma_fonction(y)` renvoie 6. Et la valeur de `y` n'a pas changé, elle vaut toujours 5.
- Voici la situation délicate qu'il faut bien comprendre :

```
x = 7
print(ma_fonction(x))
print(x)
```

- La variable `x` est initialisée à 7.
- L'appel de la fonction `ma_fonction(x)` est donc la même chose que `ma_fonction(7)` et renvoie logiquement 8.
- Que vaut la variable `x` à la fin ? La variable `x` est inchangée et vaut toujours 7 ! Même s'il y a eu entre temps une instruction `x = x + 1`. Cette instruction a changé le `x` à l'intérieur de la fonction, mais pas le `x` en dehors de la fonction.

- Les variables définies à l'intérieur d'une fonction sont appelées **variables locales**. Elles n'existent pas en dehors de la fonction.
- S'il existe une variable dans une fonction qui porte le même nom qu'une variable dans le programme (comme le `x` dans l'exemple ci-dessus), c'est comme si il y avait deux variables distinctes ; la variable locale n'existant que dans la fonction.

Pour bien comprendre la portée des variables, tu peux colorier les variables globales d'une fonction en rouge, et les variables locales avec une couleur par fonction. Le petit programme suivant définit une fonction qui ajoute un et une autre qui calcule le double.



Le programme affiche d'abord la valeur de `x`, donc 7, puis il l'augmente de 1, il affiche donc 8, puis il affiche le double de `x`, donc 14. La variable globale `x` n'a jamais changé, le dernier affichage de `x` est donc encore 7.

Il est tout de même possible de forcer la main à Python et de modifier une variable globale dans une fonction à l'aide du mot clé `global`. Voir la fiche « Calculatrice polonaise – Pile ».

Arithmétique – Boucle tant que – I

Chapitre 5

Les activités de cette fiche sont centrées sur l'arithmétique : division euclidienne, nombres premiers...
C'est l'occasion d'utiliser intensivement la boucle « tant que ».

Cours 1 (Arithmétique).

On rappelle ce qu'est la division euclidienne. Voici la division de a par b , a est un entier positif, b est un entier strictement positif (avec un exemple de 100 divisé par 7) :

The diagram illustrates the Euclidean division of a by b . It shows two representations: a general one with variables and a specific one with the values 100 and 7. In the general case, a is divided by b to yield a quotient q and a remainder r . Blue arrows point from the labels 'reste' and 'quotient' to r and q respectively. In the specific example, 100 is divided by 7 to yield a quotient of 14 and a remainder of 2.

a		b		100		7
				2		14

On a les deux propriétés fondamentales qui définissent q et r :

$$a = b \times q + r \quad \text{et} \quad 0 \leq r < b$$

Par exemple, pour la division de $a = 100$ par $b = 7$: on a le quotient $q = 14$ et le reste $r = 2$ qui vérifient bien $a = b \times q + r$ car $100 = 7 \times 14 + 2$ et aussi $r < b$ car $2 < 7$.

Avec Python :

- `a // b` renvoie le quotient,
- `a % b` renvoie le reste.

Il est facile de vérifier que :

b est un diviseur de a si et seulement si $r = 0$.

Activité 1 (Quotient, reste, divisibilité).

Objectifs : utiliser le reste pour savoir si un entier divise un autre.

1. Programme une fonction `quotient_reste(a,b)` qui fait les tâches suivantes à partir de deux entiers $a \geq 0$ et $b > 0$:

- Elle affiche le quotient q de la division euclidienne de a par b ,
- elle affiche le reste r de cette division,
- elle affiche `True` si le reste r est bien positif et strictement inférieur à b , et `False` sinon,
- elle affiche `True` si on a bien l'égalité $a = bq + r$, et `False` sinon.

Voici par exemple ce que doit afficher l'appel `quotient_reste(100,7)` :

Division de $a = 100$ par $b = 7$
 Le quotient vaut $q = 14$
 Le reste vaut $r = 2$
 Vérification reste $0 \leq r < b$? True
 Vérification égalité $a = bq + r$? True

Remarque. il faut que tu vérifies sans tricher que l'on a bien $0 \leq r < b$ et $a = bq + r$, mais bien sûr cela doit toujours être vrai !

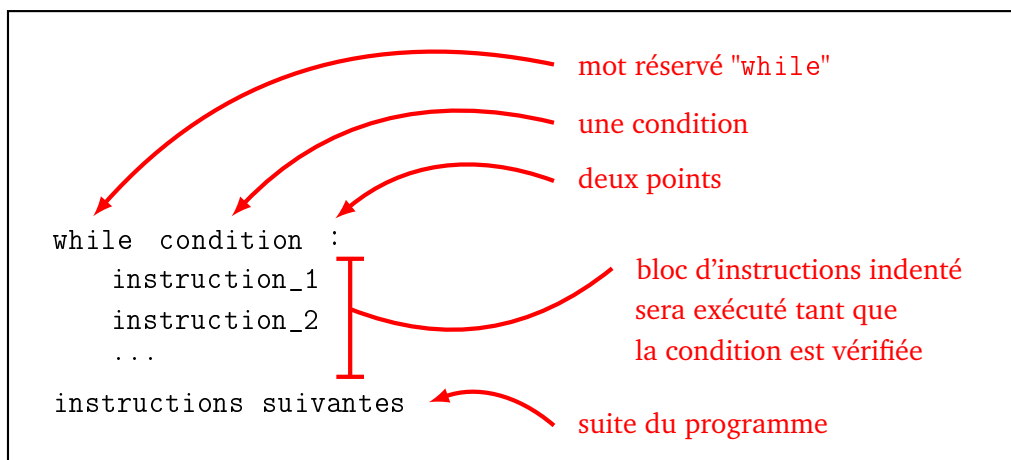
2. Programme une fonction `est_pair(n)` qui teste si l'entier n est pair ou pas. La fonction renvoie True ou False.

Indications

- Première possibilité : calculer $n \% 2$ et discuter selon les cas.
 - Seconde possibilité : calculer $n \% 10$ (qui renvoie le chiffre des unités) et discuter.
 - Les plus malins arriveront à écrire la fonction sur deux lignes seulement (une pour `def` . . . et l'autre pour `return` . . .)
3. Programme une fonction `est_divisible(a,b)` qui teste si b divise a . La fonction renvoie True ou False.

Cours 2 (Boucle « tant que »).

La boucle « tant que » exécute des instructions tant qu'une condition est vraie. Dès que la condition devient fausse, elle passe aux instructions suivantes.



Exemple.

Voici un programme qui affiche le compte à rebours 10, 9, 8, ..., 3, 2, 1, 0. Tant que la condition $n \geq 0$ est vraie, on diminue n de 1. La dernière valeur affichée est $n = 0$, car ensuite $n = -1$ et la condition « $n \geq 0$ » devient fausse donc la boucle s'arrête.

On résume ceci sous la forme d'un tableau :

Entrée : $n = 10$

n	« $n \geq 0$ » ?	nouvelle valeur de n
10	oui	9
9	oui	8
...
1	oui	0
0	oui	-1
-1	non	

Affichage : 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

```
n = 10
while n >= 0:
    print(n)
    n = n - 1
```

Exemple.

Ce bout de code cherche la première puissance de 2 plus grande qu'un entier n donné. La boucle fait prendre à p les valeurs 2, 4, 8, 16, ... Elle s'arrête dès que la puissance de 2 est supérieure ou égale à n , donc ici ce programme affiche 128.

```
n = 100
p = 1
while p < n:
    p = 2 * p
print(p)
```

Entrées : $n = 100$, $p = 1$

p	« $p < n$ » ?	nouvelle valeur de p
1	oui	2
2	oui	4
4	oui	8
8	oui	16
16	oui	32
32	oui	64
64	oui	128
128	non	

Affichage : 128

Exemple.

Pour cette dernière boucle on a déjà programmé une fonction `est_pair(n)` qui renvoie `True` si l'entier n est pair et `False` sinon. La boucle fait donc ceci : tant que l'entier n est pair, n devient $n/2$. Cela revient à supprimer tous les facteurs 2 de l'entier n . Comme ici $n = 56 = 2 \times 2 \times 2 \times 7$, ce programme affiche 7.

```
n = 56
while est_pair(n) == True:
    n = n // 2
print(n)
```

Entrée : $n = 56$

n	« n est pair » ?	nouvelle valeur de n
56	oui	28
28	oui	14
14	oui	7
7	non	

Affichage : 7

Pour ce dernier exemple il est beaucoup plus naturel de démarrer la boucle par

```
while est_pair(n):
```

En effet `est_pair(n)` est déjà une valeur « vrai » ou faux ». On se rapproche d'une phrase « tant que n est pair... »

Opération « += ». Pour incrémenter un nombre tu peux utiliser ces deux méthodes :

```
nb = nb + 1    ou    nb += 1
```

La seconde écriture est plus courte mais rend le programme moins lisible.

Activité 2 (Nombres premiers).

Objectifs : tester si un entier est (ou pas) un nombre premier.

1. Plus petit diviseur.

Programme une fonction `plus_petit_diviseur(n)` qui renvoie, le plus petit diviseur $d \geq 2$ de l'entier $n \geq 2$.

Par exemple `plus_petit_diviseur(91)` renvoie 7, car $91 = 7 \times 13$.

Méthode.

- On rappelle que d divise n si et seulement si $n \% d$ vaut 0.
- La mauvaise idée est d'utiliser une boucle « pour d variant de 2 à n ». En effet, si par exemple on sait que 7 est diviseur de 91 cela ne sert à rien de tester si 8, 9, 10... sont aussi des diviseurs car on a déjà trouvé le plus petit.
- La bonne idée est d'utiliser une boucle « tant que » ! Le principe est : « tant que je n'ai pas obtenu mon diviseur, je continue de chercher ». (Et donc, dès que je l'ai trouvé, j'arrête de chercher.)
- En pratique voici les grandes lignes :
 - Commence avec $d = 2$.
 - Tant que d ne divise pas n alors, passe au candidat suivant (d devient $d + 1$).
 - À la fin d est le plus petit diviseur de n (dans le pire des cas $d = n$).

2. Nombres premiers (1).

Modifie légèrement ta fonction `plus_petit_diviseur(n)` pour écrire une première fonction

`est_premier_1(n)` qui renvoie « vrai » (True) si n est un nombre premier et « faux » (False) sinon.

Par exemple `est_premier_1(13)` renvoie True, `est_premier_1(14)` renvoie False.

3. Nombres de Fermat.

Pierre de Fermat (~1605–1665) pensait que tous les entiers $F_n = 2^{(2^n)} + 1$ étaient des nombres premiers. Effectivement $F_0 = 3$, $F_1 = 5$ et $F_2 = 17$ sont des nombres premiers. S'il avait connu Python il aurait sûrement changé d'avis ! Trouve le plus petit entier F_n qui n'est pas premier.

Indication. Avec Python b^c s'écrit `b ** c` et donc $a^{(b^c)}$ s'écrit `a ** (b ** c)`.

On va améliorer notre fonction qui teste si un nombre est premier ou pas, cela nous permettra de tester plus vite plein de nombres ou bien des nombres très grands.

4. Nombres premiers (2).

Améliore ta fonction en une fonction `est_premier_2(n)` qui ne teste pas tous les diviseurs d jusqu'à n , mais seulement jusqu'à \sqrt{n} .

Explications.

- Par exemple pour tester si 101 est un nombre premier, il suffit de voir s'il admet des diviseurs parmi 2, 3, ..., 10. Le gain est appréciable !
- Cette amélioration est due à la proposition suivante : si un entier n'est pas premier alors il admet un diviseurs d qui vérifie $2 \leq d \leq \sqrt{n}$.
- Au lieu de tester si $d \leq \sqrt{n}$, il est plus facile de tester $d^2 \leq n$!

5. Nombres premiers (3).

Améliore ta fonction en une fonction `est_premier_3(n)` à l'aide de l'idée suivante. On teste si $d = 2$ divise n , mais à partir de $d = 3$, il suffit de tester les diviseurs impairs (on teste d , puis $d + 2$...).

- Par exemple pour tester si $n = 419$ est un nombre premier, on teste d'abord si $d = 2$ divise n , puis $d = 3$ et ensuite $d = 5$, $d = 7$...
- Cela permet de faire environ deux fois moins de tests !
- Explications : si un nombre pair d divise n , alors on sait déjà que 2 divise n .

6. Temps de calcul.

Compare les temps de calcul de tes différentes fonctions `est_premier()` en répétant par exemple un million de fois l'appel `est_premier(97)`. Voir le cours ci-dessous pour savoir comment faire.

Cours 3 (Temps de calcul).

Il existe deux façons de faire tourner plus rapidement des programmes : une bonne et une mauvaise. La mauvaise, c'est d'acheter un ordinateur plus puissant. La bonne, c'est de trouver un algorithme plus efficace !

Avec Python, c'est facile de mesurer le temps d'exécution d'une fonction afin de le comparer avec le temps d'exécution d'une autre. Il suffit d'utiliser le module `timeit`.

Voici un exemple : on mesure le temps de calcul de deux fonctions qui ont le même but, tester si un entier n est divisible par 7.

```
# Première fonction (pas très maligne)
def ma_fonction_1(n):
    divis = False
    for k in range(n):
        if k*7 == n:
```

```

        divis = True
    return divis

# Seconde fonction (plus rapide)
def ma_fonction_2(n):
    if n % 7 == 0:
        return True
    else:
        return False

import timeit

print(timeit.timeit("ma_fonction_1(1000)",
    setup="from __main__ import ma_fonction_1",
    number=100000))
print(timeit.timeit("ma_fonction_2(1000)",
    setup="from __main__ import ma_fonction_2",
    number=100000))

```

Résultats.

Le résultat dépend de l'ordinateur, mais permet la comparaison des temps d'exécution des deux fonctions.

- La mesure pour la première fonction renvoie 5 secondes. L'algorithme n'est pas très malin. On teste si $7 \times 1 = n$, puis on teste $7 \times 2 = n$, $7 \times 3 = n$...
- La mesure pour la seconde fonction renvoie 0.01 seconde ! On teste si le reste de n divisé par 7 est 0. La seconde méthode est donc 500 fois plus rapide que la première.

Explications.

- On appelle le module `timeit`.
- La fonction `timeit.timeit()` renvoie le temps d'exécution en seconde. Elle prend comme paramètres :
 - une chaîne pour l'appel de la fonction à tester (ici est-ce que 1000 est divisible par 7),
 - un argument `setup="..."` qui indique où trouver cette fonction,
 - le nombre de fois qu'il faut répéter l'appel à la fonction (ici `number=100000`).
- Il faut que le nombre de répétitions soit assez grand pour éviter les incertitudes.

Activité 3 (Plus de nombres premiers).

Objectifs : programmer davantage de boucles « tant que » et étudier différentes sortes de nombres premiers à l'aide de ta fonction `est_premier()`.

1. Écris une fonction `nombre_premier_apres(n)` qui renvoie le premier nombre premier p supérieur ou égal à n .
Par exemple, le premier nombre premier après $n = 60$ est $p = 61$. Quel est le premier nombre premier après $n = 100\,000$?
2. Deux nombres premiers p et $p + 2$ sont appelés **nombres premiers jumeaux**. Écris une fonction `nombres_jumeaux_apres(n)` qui renvoie le premier couple $p, p + 2$ de nombres premiers jumeaux, avec $p \geq n$.
Par exemple, le premier couple de nombres premiers jumeaux après $n = 60$ est $p = 71$ et $p + 2 = 73$. Quel est le premier couple de nombres premiers jumeaux après $n = 100\,000$?

3. Un entier p est un **nombre premier de Germain** si p et $2p + 1$ sont des nombres premiers. Écris une fonction `nombre_germain_apres(n)` qui renvoie le couple $p, 2p + 1$ où p est le premier nombre premier de Germain $p \geq n$.

Par exemple, le premier nombre premier de Germain après $n = 60$ est $p = 83$ avec $2p + 1 = 167$. Quel est le premier nombre premier de Germain après $n = 100\,000$?

Chaînes de caractères

– Analyse d'un texte

Chapitre 6

Tu vas faire quelques activités amusantes en manipulant les chaînes de caractères.

Cours 1 (Caractère et chaîne).

1. Un **caractère** est un symbole unique, par exemple une lettre minuscule "a", une lettre majuscule "B", un symbole spécial "&", un symbole représentant un chiffre "7", une espace " " que l'on notera aussi "␣".
Pour désigner un caractère, il faut le mettre entre guillemets simples 'z' ou entre guillemets doubles "z".
2. Une **chaîne de caractères** est une suite de caractères, comme un mot "Bonjour", une phrase 'Il fait beau.', un mot de passe "N[w5ms}e!".
3. Le type d'un caractère ou d'une chaîne est str (pour *string*).

Cours 2 (Opérations sur les chaînes).

1. La **concaténation**, c'est-à-dire la mise bout à bout de deux chaînes, s'effectue à l'aide de l'opérateur +. Par exemple "para"+"pluie" donne la chaîne "parapluie".
2. La chaîne vide "" est utile lorsque l'on veut initialiser une chaîne avant d'y ajouter d'autres caractères.
3. La **longueur** d'une chaîne est le nombre de caractères qu'elle contient. Elle s'obtient par la fonction len(). Par exemple len("Hello␣World") renvoie 11 (l'espace compte comme un caractère).
4. Si mot est une chaîne alors on peut récupérer chaque caractère par mot[i]. Par exemple si mot = "avion" alors :
 - mot[0] est le caractère "a",
 - mot[1] est le caractère "v",
 - mot[2] est le caractère "i",
 - mot[3] est le caractère "o",
 - mot[4] est le caractère "n".

Lettre	a	v	i	o	n
Rang	0	1	2	3	4

Note qu'il y a 5 lettres dans le mot "avion" et qu'on y accède par les indices en commençant par 0. Les indices sont donc ici 0, 1, 2, 3 et 4 pour la dernière lettre. De façon plus générale, si mot est une chaîne, les caractères s'obtiennent par mot[i] pour i variant de 0 à len(mot)-1.

Cours 3 (Sous-chaînes).

On peut extraire plusieurs caractères d'une chaîne à l'aide de la syntaxe `mot[i : j]` qui renvoie une chaîne formée des caractères numéro i à $j - 1$ (attention le caractère numéro j n'est pas inclus !).

Par exemple si `mot = "vendredi"` alors :

- `mot[0 : 4]` renvoie la sous-chaîne "vend" formée des caractères de rang 0, 1, 2 et 3 (mais pas 4),
- `mot[3 : 6]` renvoie "dre" correspondant aux rangs 3, 4 et 5.

Lettre	v	e	n	d	r	e	d	i
Rang	0	1	2	3	4	5	6	7

Autre exemple : `mot[1 : len(mot) - 1]` renvoie le mot privé de sa première et dernière lettre.

Activité 1 (Pluriels des mots).

Objectifs : écrire petit à petit un programme qui renvoie le pluriel d'un mot donné.

1. Pour une chaîne `mot`, par exemple "chat", affiche le pluriel de ce mot en rajoutant un "s".
2. Pour un mot, par exemple "souris", affiche la dernière lettre de cette chaîne (ici "s"). Améliore ton programme de la première question, en testant si la dernière lettre est déjà un "s" :
 - si c'est le cas, il n'y a rien à faire pour le pluriel,
 - sinon il faut ajouter un "s".
3. Teste si un mot se termine par "al". Si c'est le cas, affiche le pluriel en "aux" (le pluriel de "cheval" est "chevaux"). (Ne tiens pas compte des exceptions.)
4. Rassemble tout ton travail des trois premières questions dans une fonction `met_au_pluriel()`. La fonction n'affiche rien, mais renvoie le mot au pluriel.

met_au_pluriel()

Usage : `met_au_pluriel(mot)`

Entrée : un mot (une chaîne de caractères)

Sortie : le pluriel du mot

Exemples :

- `met_au_pluriel("chat")` renvoie "chats"
- `met_au_pluriel("souris")` renvoie "souris"
- `met_au_pluriel("cheval")` renvoie "chevaux"

5. Écris une fonction `affiche_conjugaison()` qui conjugue un verbe du premier groupe au présent.

affiche_conjugaison()

Usage : `affiche_conjugaison(verbe)`

Entrée : un verbe du premier groupe (une chaîne de caractères se terminant par "er")

Sortie : pas de résultat mais l'affichage de la conjugaison du verbe au présent

Exemple :

- `affiche_conjugaison("chanter"),` affiche
"je chante, tu chantes,..."
- `affiche_conjugaison("choisir"),` affiche
"Ce n'est pas un verbe du premier groupe."

Cours 4 (Un peu plus sur les chaînes).

1. Une boucle `for ... in ...` permet de parcourir une chaîne, caractère par caractère :

```
for carac in mot:
    print(carac)
```

2. On peut tester si un caractère appartient à une certaine liste de caractères. Par exemple :

```
if carac in ["a", "A", "b", "B", "c", "C"]:
```

permet d'exécuter des instructions si le caractère `carac` est l'une des lettres a, A, b, B, c, C.

Pour éviter certaines lettres, on utiliserait :

```
if carac not in ["X", "Y", "Z"]:
```

Activité 2 (Jeux de mots).

Objectifs : manipuler des mots de façon amusante.

1. Distance entre deux mots.

La distance de Hamming entre deux mots de même longueur est le nombre d'endroits où les lettres sont différentes.

Par exemple :

JAPON SAVON

La première lettre de **JAPON** est différente de la première lettre de **SAVON**, les troisièmes aussi sont différentes. La distance de Hamming entre **JAPON** et **SAVON** vaut donc 2.

Écris une fonction `distance_hamming()` qui calcule la distance de Hamming entre deux mots de même longueur.

distance_hamming()

Usage : `distance_hamming(mot1,mot2)`

Entrée : deux mots (des chaînes de caractères)

Sortie : la distance de Hamming (un entier)

Exemple : `distance_hamming("LAPIN", "SATIN")` renvoie 2

2. Latin-cochon.

On transforme un mot commençant par une consonne selon la recette suivante :

- on déplace la première lettre à la fin du mot ;
- on rajoute le suffixe **UM**.

Par exemple **VITRE** devient **ITREVUM** ; **BLANCHE** devient **LANCHEBUM** ; **CARAMEL** devient **ARAMELCUM**. Les mots commençant par une voyelle ne changent pas. Écris une fonction `latin_cochon()` qui transforme un mot selon ce procédé.

`latin_cochon()`

Usage : `latin_cochon(mot)`

Entrée : un mot (une chaîne de caractères)

Sortie : le mot transformé en latin-cochon, s'il commence par une consonne.

Exemple : `latin_cochon("BONJOUR")` renvoie "ONJOURBUM"

3. Verlan.

Écris une fonction `verlan()` qui renvoie un mot à l'envers : **SALUT** devient **TULAS**.

`verlan()`

Usage : `verlan(mot)`

Entrée : un mot (une chaîne de caractères)

Sortie : le mot à l'envers

Exemple : `verlan("TOCARD")` renvoie "DRACOT"

4. Palindrome.

Déduis-en une fonction qui teste si un mot est un palindrome ou pas. Un *palindrome* est un mot qui s'écrit indifféremment de gauche à droite ou de droite à gauche ; par exemple **RADAR** est un palindrome.

`est_un_palindrome()`

Usage : `est_un_palindrome(mot)`

Entrée : un mot (une chaîne de caractères)

Sortie : « vrai » si le mot est un palindrome, « faux » sinon.

Exemple : `est_un_palindrome("KAYAK")` renvoie `True`

Activité 3 (ADN).

*Une molécule d'ADN est formée d'environ six milliards de nucléotides. L'ordinateur est donc un outil indispensable pour l'analyse de l'ADN. Dans un brin d'ADN il y a seulement quatre types de nucléotides qui sont notés **A**, **C**, **T** ou **G**. Une séquence d'ADN est donc un long mot de la forme : **TAATTACAGACCTGAA...***

1. Écris une fonction `presence_de_A()` qui teste si une séquence contient le nucléotide **A**.

presence_de_A()

Usage : `presence_de_A(sequence)`

Entrée : une séquence d'ADN (une chaîne de caractères parmi A, C, T, G)

Sortie : « vrai » si la séquence contient « A », « faux » sinon.

Exemple : `presence_de_A("CTTGCT")` renvoie `False`

2. Écris une fonction `position_de_AT()` qui teste si une séquence contient le nucléotide A suivi du nucléotide T et renvoie la position de la première occurrence trouvée.

position_de_AT()

Usage : `position_de_AT(sequence)`

Entrée : une séquence d'ADN (une chaîne de caractères parmi A, C, T, G)

Sortie : la position de la première séquence « AT » trouvée (commence à 0) ;

`None` si n'apparaît pas

Exemple :

- `position_de_AT("CTTATGCT")` renvoie 3
- `position_de_AT("GATATAT")` renvoie 1
- `position_de_AT("GACCGTA")` renvoie `None`

Indication. `None` est affecté à une variable pour signifier l'absence de valeur.

3. Écris une fonction `position()` qui teste si une séquence contient un code donné et renvoie la position de la première occurrence.

position()

Usage : `position(code, sequence)`

Entrée : un code et une séquence d'ADN

Sortie : la position du début du code trouvé ; `None` si n'apparaît pas

Exemple : `position("CCG", "CTCCGTT")` renvoie 2

4. Un crime a été commis dans le château d'Adéno. Tu as récupéré deux brins d'ADN, provenant de deux positions éloignées de l'ADN du coupable. Il y a quatre suspects, dont tu as séquencé l'ADN. Sauras-tu trouver qui est le coupable ?

Premier code du coupable : **CATA**

Second code du coupable : **ATGC**

ADN du colonel Moutarde :

CCTGGAGGGTGGCCCCACCGGCCGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGC

ADN de Mlle Rose :

CTCCTGATGCTCCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGCCAGTGCCGGGGCCCCTCATAGGAGAGG

ADN de Mme Pervenche :

AAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCAGTACTCCGCGCGCCGGGACAGAATGCC

ADN de M. Leblanc :

CTGCAGGAACCTTCTTCTGGAAGTACTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAG

Cours 5 (Codage des caractères).

Un caractère est stocké par l'ordinateur sous la forme d'un entier. Pour le codage ASCII/unicode, la lettre majuscule « A » est codé par 65, la lettre minuscule « h » est codée par 104, le symbole « # » par 35.

Voici la table des premiers caractères. Les numéros 0 à 32 ne sont pas des caractères imprimables. Cependant le numéro 32 est le caractère espace " ".

33	!	43	+	53	5	63	?	73	I	83	S	93]	103	g	113	q	123	{
34	"	44	,	54	6	64	@	74	J	84	T	94	^	104	h	114	r	124	
35	#	45	-	55	7	65	A	75	K	85	U	95	_	105	i	115	s	125	}
36	\$	46	.	56	8	66	B	76	L	86	V	96	'	106	j	116	t	126	~
37	%	47	/	57	9	67	C	77	M	87	W	97	a	107	k	117	u	127	-
38	&	48	0	58	:	68	D	78	N	88	X	98	b	108	l	118	v		
39	'	49	1	59	;	69	E	79	O	89	Y	99	c	109	m	119	w		
40	(50	2	60	<	70	F	80	P	90	Z	100	d	110	n	120	x		
41)	51	3	61	=	71	G	81	Q	91	[101	e	111	o	121	y		
42	*	52	4	62	>	72	H	82	R	92	\	102	f	112	p	122	z		

1. La fonction `chr()` est une fonction Python qui renvoie le caractère associé à un code.

python : `chr()`

Usage : `chr(code)`
 Entrée : un code (un entier)
 Sortie : un caractère
 Exemple :
 • `chr(65)` renvoie "A"
 • `chr(123)` renvoie "{"

2. La fonction `ord()` est une fonction Python correspondant à l'opération inverse : elle renvoie le code associé à un caractère.

python : `ord()`

Usage : `ord(carac)`
 Entrée : un caractère (une chaîne de longueur 1)
 Sortie : un entier
 Exemple :
 • `ord("A")` renvoie 65
 • `ord("*")` renvoie 42

Activité 4 (Majuscules/minuscules).

Objectifs : convertir un mot en majuscules ou en minuscules.

1. Décode à la main le message chiffré sous les codes suivants :

80-121-116-104-111-110 101-115-116 115-121-109-112-64

2. Écris une boucle qui affiche les caractères codés par les entiers de 33 à 127.
3. Que renvoie la commande `chr(ord("a")-32)` ? Et `chr(ord("B")+32)` ?
4. Écris une fonction `lettre_majuscule()` qui transforme une lettre minuscule en sa lettre majuscule.

lettre_majuscule()

Usage : `lettre_majuscule(carac)`
 Entrée : un caractère minuscule parmi "a", ..., "z"
 Sortie : la même lettre en majuscule
 Exemple : `lettre_majuscule("t")` renvoie "T"

5. Écris une fonction `majuscules()` qui à partir d'une phrase écrite en minuscules renvoie la même phrase écrite en majuscules. Les caractères qui ne sont pas des lettres minuscules restent inchangés.

majuscules()

Usage : `majuscules(phrase)`
 Entrée : une phrase
 Sortie : la même phrase en majuscules
 Exemple : `majuscules("Bonjour le monde !")` renvoie "BONJOUR LE MONDE !"

Fais le travail semblable pour une fonction `minuscules()`.

6. Écris une fonction `formate_prenom_nom()` qui renvoie le prénom et le nom formatés suivant le style **Prénom NOM**.

formate_prenom_nom()

Usage : `formate_prenom_nom(personne)`
 Entrée : le prénom et le nom d'une personne (sans accent, séparés par une espace)
 Sortie : le nom complet au format "Prénom NOM"
 Exemple :
 • `formate_prenom_nom("harry Potter")` renvoie "Harry POTTER"
 • `formate_prenom_nom("LORD Voldemort")` renvoie "Lord VOLDEMORT"

Activité 5.

Objectifs : déterminer la langue d'un texte à partir de l'analyse des fréquences des lettres.

1. Écris une fonction `occurrences_lettre()` qui compte le nombre de fois où la lettre donnée apparaît dans une phrase (en majuscules et sans accents).

occurrences_lettre()

Usage : `occurrences_lettre(lettre, phrase)`

Entrée : une lettre et une phrase en majuscules (une chaîne de caractères)

Sortie : le nombre d'occurrences de la lettre (un entier)

Exemple : `occurrences_lettre("E", "ESPRIT ES TU LA")` renvoie 2

2. Écris une fonction `nombre_lettres()` qui compte le nombre total de lettres qui apparaissent dans une phrase (en majuscules et sans accents). Ne pas compter les espaces, ni la ponctuation.

nombre_lettres()

Usage : `nombre_lettres(phrase)`

Entrée : une phrase en majuscules (une chaîne de caractères)

Sortie : le nombre total de lettres de « A » à « Z » (un entier)

Exemple : `nombre_lettres("ESPRIT ES TU LA")` renvoie 12

3. La **fréquence d'apparition** d'une lettre dans un texte ou une phrase est le pourcentage donné selon la formule :

$$\text{fréquence d'apparition d'une lettre} = \frac{\text{nombre d'occurrences de la lettre}}{\text{nombre total de lettres}} \times 100.$$

Par exemple, la phrase **ESPRIT ES TU LA** contient 12 lettres ; la lettre **E** y apparaît 2 fois. La fréquence d'apparition de **E** dans cette phrase est donc :

$$f_E = \frac{\text{nombre d'occurrences de E}}{\text{nombre total de lettres}} \times 100 = \frac{2}{12} \times 100 \simeq 16.66$$

La fréquence est donc d'environ 17%.

Écris une fonction `pourcentage_lettre()` qui calcule cette fréquence d'apparition.

pourcentage_lettre()

Usage : `pourcentage_lettre(lettre, phrase)`

Entrée : une lettre et une phrase en majuscules (une chaîne de caractères)

Sortie : la fréquence d'apparition de la lettre (un nombre inférieur à 100)

Exemple : `pourcentage_lettre("E", "ESPRIT ES TU LA")` renvoie 16.66...

Utilise cette fonction pour afficher proprement la fréquence d'apparition de toutes les lettres d'une phrase.

4. Voici la fréquence d'apparition des lettres selon la langue utilisée (source : en.wikipedia.org/wiki/Letter_frequency). Par exemple, la lettre la plus courante en français est le « e » avec une fréquence de plus de 16%. Le « w » représente environ 2% des lettres en anglais et en allemand, mais n'apparaît presque pas en français et en espagnol. Ces fréquences varient aussi en fonction du texte analysé.

Lettre	Anglais	Français	Allemand	Espagnol
a	8.167%	8.173%	7.094%	12.027%
b	1.492%	0.901%	1.886%	2.215%
c	2.782%	3.345%	2.732%	4.019%
d	4.253%	3.669%	5.076%	5.010%
e	12.702%	16.734%	16.396%	12.614%
f	2.228%	1.066%	1.656%	0.692%
g	2.015%	0.866%	3.009%	1.768%
h	6.094%	0.737%	4.577%	0.703%
i	6.966%	7.579%	6.550%	6.972%
j	0.153%	0.613%	0.268%	0.493%
k	0.772%	0.049%	1.417%	0.011%
l	4.025%	5.456%	3.437%	4.967%
m	2.406%	2.968%	2.534%	3.157%
n	6.749%	7.095%	9.776%	7.023%
o	7.507%	5.819%	3.037%	9.510%
p	1.929%	2.521%	0.670%	2.510%
q	0.095%	1.362%	0.018%	0.877%
r	5.987%	6.693%	7.003%	6.871%
s	6.327%	7.948%	7.577%	7.977%
t	9.056%	7.244%	6.154%	4.632%
u	2.758%	6.429%	5.161%	3.107%
v	0.978%	1.838%	0.846%	1.138%
w	2.360%	0.074%	1.921%	0.017%
x	0.150%	0.427%	0.034%	0.215%
y	1.974%	0.128%	0.039%	1.008%
z	0.074%	0.326%	1.134%	0.467%

D'après toi, dans quelles langues ont été écrits les quatre textes suivants (les lettres de chaque mot ont été mélangées).

TMAIER BERACUO RSU NU REBRA PRCEEH EIAN TT NE ONS EBC NU GAOFREEM EIMATR RERNAD APR L RDUOE LAHECLE UIL TTNI A EUP SREP EC LGNGAEA TE RBONUJO ERMNOUSI DU UBRACEO QUE OVSU EEST LIJO UQE OUVS EM MSZELBE BAEU ASNS MIERNT IS RVETO AGRAME ES PRARPTOE A OEVTR AMGUPLE VUOS SEET EL PNIHXE DSE OSHET ED CSE BIOS A ESC MSOT LE OUBRCEA NE ES ESTN ASP DE IEJO TE OUPR ERRNOTM AS BELEL XOVI IL OREU NU RGLEA ECB ILESSA EBOMTR AS PIOER EL NRDAER S EN ISIAST TE ITD MNO NOB EUSRMNOI NRPEEAZP QEU UTOT EUTLRFTA IVT XUA SPNEDE DE UECIL UQI L TECEOUE TECET NEOCL VATU BNEI UN GMAEORF SNAS TUOED LE EOABURC OHENTXU TE NSCOFU UJRA SMIA UN EPU TRDA UQ NO EN L Y ARRPEIDNT ULSP

WRE TREITE SO TSPA CUDHR AHNCT UND WIND SE STI RED AEVRT MTI ESEIMN IDNK RE ATH END NEABNK WLOH IN EMD AMR ER AFTSS HIN IHSERC RE AHTL HIN MRWA EINM SHNO SAW SRTIBG UD SO NGBA DNEI EIHSBTC ESISTH RAETV UD DEN LERNIOKG NITHC NDE LOENINKGRE TIM OKRN UDN CHWFSEI NEIM NSOH ES STI IEN BIFTRLSEEN DU BILESE IKDN OMKM EHG MIT MIR RAG ECHNOS EPELSI EIPSL IHC ITM RDI HNCMA BEUTN MBLUNE DINS NA DEM TNDRA NMIEE UTETMR AHT CAMHN UDNGEL GDWEN MIEN EATRV MENI VEART DUN OSTHER DU CINTH SAW KNOEIREGL RIM ILEES PRSTVRCIEH ISE IHGRU BEEILB RIGUH MNEI KNDI NI RDNEUR NATBRLET STAESUL EDR WNID

DSNOACAIF ORP ANU DAEDALRI DNAAEIMTI EQU NNCOSETE EL RSTEOUL SMA AACTFAITNS UQE LE TSVAO OINSRVUE DE US ANIGIICANOM EIORDP TOOD RTEIENS RPO LE ITOABOLRROA ED QIUAMALI

USOP A NSSRCAEAD LA TMREAAI NXTADAUEE ROP GOARLS EMESS DE NNAMICLUIAPO Y LOVOIV A RES
LE RHMEOB EOMDNEERPRD DE LOS RSOPMRIE OMTSIPE UEQ CIIDADE LE RTDAAOZ ED LSA CELSAL Y
LA NICOIOPS ED LAS UESVNA SSACA Y ES ITRMNEEOD QEU AERFU EL UEQIN IIIRDEGAR LA
NAIORTREICP DE AL RRTEIA

IMTRUESMME DNA TEH LNGIIV SI EYAS SIFH REA GJPNUIM DNA HET TTNOCO IS GHIH OH OUYR DDADY
SI IRHC DAN ROUY MA SI DOGO GKOILON OS USHH LTLIET BBYA NDOT OUY CYR NEO OF HESET
GNSRONIM YUO RE NANGO SIER PU SNIGING NAD OULLY EPADRS YUOR GINSW DAN LYOLU KATE OT
HET KSY TUB ITLL TATH MGNIRNO EREHT NATI INTGOHN ACN AHMR OYU TWIH DADYD NDA MYMMA
NSTIDGAN YB

Listes I

Une liste est une façon de regrouper des éléments en un seul objet. Après avoir défini une liste, on peut récupérer un par un chaque élément de la liste, mais aussi en ajouter de nouveaux...

Cours 1 (Liste (1)).

Une **liste** est une suite d'éléments. Cela peut être une liste d'entiers, par exemple `[5, -7, 12, 99]`, ou bien une liste de chaînes de caractères, par exemple `["Mars", "Avril", "Mai"]` ou bien les objets peuvent être de différents types `[3.14, "pi", 10e-3, "x", True]`.

- **Construction d'une liste.** Une liste se définit par des éléments entre crochets :
 - `liste1 = [5, 4, 3, 2, 1]` une liste de 5 entiers,
 - `liste2 = ["Vendredi", "Samedi", "Dimanche"]` une liste de 3 chaînes de caractères,
 - `liste3 = []` la liste vide (très utile pour la compléter plus tard !).
- **Accéder à un élément.** Pour obtenir un élément de la liste, il suffit d'écrire `liste[i]` où *i* est le rang de l'élément souhaité.

Attention ! Le piège c'est que l'on commence à compter à partir du rang 0 !

Par exemple après l'instruction `liste = ["A", "B", "C", "D", "E", "F"]` alors

- `liste[0]` renvoie "A"
- `liste[1]` renvoie "B"
- `liste[2]` renvoie "C"
- `liste[3]` renvoie "D"
- `liste[4]` renvoie "E"
- `liste[5]` renvoie "F"

"A"	"B"	"C"	"D"	"E"	"F"
-----	-----	-----	-----	-----	-----

rang : 0 1 2 3 4 5

- **Ajouter un élément.** Pour ajouter un élément à la fin de la liste, il suffit d'utiliser la commande `maliste.append(element)` (*to append* signifie « ajouter »). Par exemple si `premiers = [2, 3, 5, 7]` alors `premiers.append(11)` rajoute 11 à la liste, si ensuite on exécute la commande `premiers.append(13)` alors maintenant la liste `premiers` vaut `[2, 3, 5, 7, 11, 13]`.
- **Exemple de construction.** Voici comment construire la liste qui contient les premiers carrés :

```
liste_carres = []           # On part d'une liste vide
for i in range(10):
    liste_carres.append(i**2) # On ajoute un carré
```

À la fin `liste_carres` vaut :

`[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

Cours 2 (Liste (2)).

- **Longueur d'une liste.** La longueur d'une liste est le nombre d'éléments qu'elle contient. La commande `len(liste)` renvoie la longueur (*length* en anglais). La liste `[5, 4, 3, 2, 1]` est de longueur 5, la liste `["Vendredi", "Samedi", "Dimanche"]` de longueur 3, la liste vide `[]` de longueur 0.
- **Parcourir une liste.** Voici la façon la plus simple de parcourir une liste (et ici d'afficher chaque élément) :

```
for element in liste:
    print(element)
```

- **Parcourir une liste (bis).** Parfois on a besoin de connaître le rang des éléments. Voici une autre façon de faire (qui affiche ici le rang et l'élément).

```
n = len(liste)
for i in range(n):
    print(i, liste[i])
```

- Pour obtenir une liste à partir de `range()` il faut écrire :
`list(range(n))`

Activité 1 (Intérêts simples ou composés).

Objectifs : créer deux listes afin de comparer deux types d'intérêts.

1. **Intérêts simples.** On dispose d'une somme S_0 . Chaque année ce placement rapporte des intérêts en fonction de la somme initiale.

Par exemple avec une somme initiale $S_0 = 1000$ et des intérêts simples de $p = 10\%$. Les intérêts sont de 100. Donc au bout d'un an, je dispose d'une somme de $S_1 = 1100$, au bout de deux ans $S_2 = 1200$...

Programme une fonction `interets_simples(S0,p,n)` qui renvoie les listes des sommes des n premières années. Par exemple `interets_simples(1000,10,3)` renvoie `[1000, 1100, 1200, 1300]`.

2. **Intérêts composés.** Une somme S_0 rapporte selon des intérêts composés. Cette fois les intérêts sont calculés chaque année sur la base de la somme de l'année précédente, c'est-à-dire selon la formule :

$$I_{n+1} = S_n \times \frac{p}{100}$$

Programme une fonction `interets_composes(S0,p,n)` qui renvoie la liste des sommes des n premières années. Par exemple `interets_composes(1000,10,3)` renvoie `[1000, 1100, 1210, 1331]`.

3. J'ai le choix entre un placement à intérêts simples de 10% et un placement à intérêts composés de 7%. Quelle est la solution la plus avantageuse en fonction de la durée du placement ?

Cours 3 (Liste (3)).

- **Concaténer deux listes.** Si on a deux listes, on peut les fusionner par l'opérateur « + ». Par exemple avec `liste1 = [4, 5, 6]` et `liste2 = [7, 8, 9]`
`liste1 + liste2` vaut `[4, 5, 6, 7, 8, 9]`.
- **Ajouter un élément à la fin.** L'opérateur « + » fournit une autre méthode permettant d'ajouter un élément à une liste :

```
liste = liste + [element]
```

Par exemple `[1,2,3,4] + [5]` vaut `[1,2,3,4,5]`. Attention ! Il faut entourer l'élément à ajouter de crochets. C'est une méthode alternative à `liste.append(element)`.

- **Ajouter un élément au début.** Avec :

```
liste = [element] + liste
```

on ajoute l'élément en début de liste. Par exemple `[5] + [1,2,3,4]` vaut `[5,1,2,3,4]`.

- **Trancher des listes.** On peut extraire d'un seul coup toute une partie de la liste : `liste[a:b]` renvoie la sous-liste des éléments de rang a à $b - 1$.

"A"	"B"	"C"	"D"	"E"	"F"	"G"
-----	-----	-----	-----	-----	-----	-----

rang : 0 1 2 3 4 5 6

Par exemple si `liste = ["A", "B", "C", "D", "E", "F", "G"]` alors

— `liste[1:4]` renvoie `["B", "C", "D"]`

— `liste[0:2]` renvoie `["A", "B"]`

— `liste[4:7]` renvoie `["E", "F", "G"]`

Il faut encore une fois faire attention à ce que le rang d'une liste commence à 0 et que le tranchage `liste[a:b]` s'arrête au rang $b - 1$.

Activité 2 (Manipulation de listes).

Objectifs : programmer des petites routines qui manipulent des listes.

1. Programme une fonction `rotation(liste)` qui décale d'un rang tous les éléments d'une liste (le dernier élément devenant le premier). La fonction renvoie une nouvelle liste.
Par exemple `rotation([1,2,3,4])` renvoie la liste `[4,1,2,3]`.
2. Programme une fonction `inverser(liste)` qui inverse l'ordre des éléments d'une liste.
Par exemple `inverser([1,2,3,4])` renvoie la liste `[4,3,2,1]`.
3. Programme une fonction `supprimer_rang(liste, rang)` qui renvoie une liste formée de tous les éléments, sauf celui au rang donné.
Par exemple `supprimer_rang([8,7,6,5,4], 2)` renvoie la liste `[8,7,5,4]` (l'élément 6 qui était au rang 2 est supprimé).
4. Programme une fonction `supprimer_element(liste, element)` renvoyant une liste qui contient tous les éléments sauf ceux égaux à l'élément spécifié.
Par exemple `supprimer_element([8,7,4,6,5,4], 4)` renvoie la liste `[8,7,6,5]` (tous les éléments égaux à 4 ont été supprimés).

Cours 4 (Manipulation de listes).

Tu peux maintenant utiliser les fonctions Python qui font certaines de ces opérations.

- **Inverser une liste.** Voici trois méthodes :
 - `maliste.reverse()` modifie la liste sur place (c'est-à-dire que `maliste` est maintenant renversée, la commande ne renvoie rien) ;
 - `list(reversed(maliste))` renvoie une nouvelle liste ;
 - `maliste[::-1]` renvoie une nouvelle liste.
- **Supprimer un élément.** La commande `liste.remove(element)` supprime la première occurrence trouvée (la liste est modifiée). Par exemple avec `liste = [2,5,3,8,5]` la commande

`liste.remove(5)` modifie la liste qui maintenant vaut `[2, 3, 8, 5]` (le premier 5 a disparu).

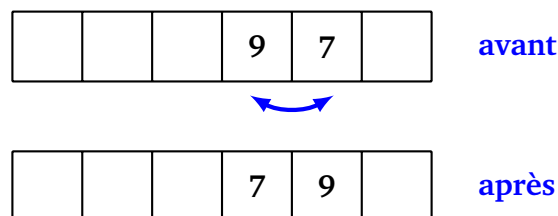
- **Supprimer un élément (bis).** La commande `del liste[i]` supprime l'élément de rang i (la liste est modifiée).

Activité 3 (Tri à bulles).

Objectifs : ordonner une liste du plus petit au plus grand élément.

Le tri à bulles est une façon simple d'ordonner une liste, ici ce sera du plus petit au plus grand élément. Le principe est le suivant :

- On parcourt la liste en partant du début. Dès que l'on rencontre deux éléments consécutifs dans le mauvais ordre, on les échange.
- À la fin du premier passage, le plus grand élément est à la fin et il ne bougera plus.
- On recommence du début (jusqu'à l'avant-dernier élément), cette fois les deux derniers éléments sont bien placés.
- On continue ainsi. Il y a en tout $n - 1$ passages si la liste est de taille n .



Voici l'algorithme du tri à bulles :

Algorithme.

- Entrée : une liste ℓ de n nombres
- Sortie : la liste ordonnée du plus petit au plus grand
- Pour i allant de $n - 1$ à 0 :
 - Pour j allant de 0 à $i - 1$:
 - Si $\ell[j + 1] < \ell[j]$ alors échanger $\ell[j]$ et $\ell[j + 1]$.
- Renvoyer la liste ℓ .

Programme l'algorithme du tri à bulles en une fonction `trier(liste)` qui renvoie la liste ordonnée des éléments. Par exemple `trier([13, 11, 7, 4, 6, 8, 12, 6])` renvoie la liste `[4, 6, 6, 7, 8, 11, 12, 13]`.

Indications.

- Commence par définir `nouv_liste = list(liste)` et travaille uniquement avec cette nouvelle liste.
- Pour que l'indice i parcourt les indices à rebours de $n - 1$ à 0 , tu peux utiliser la commande :


```
for i in range(n-1, -1, -1):
```

 En effet `range(a, b, -1)` correspond à la liste décroissante des entiers i vérifiant $a \geq i > b$ (comme d'habitude la borne de droite n'est pas incluse).

Cours 5 (Tri).

Tu peux maintenant utiliser la fonction `sorted()` de Python qui ordonne des listes.

python : sorted()

Usage : `sorted(liste)`

Entrée : une liste

Sortie : la liste ordonnée des éléments

Exemple : `sorted([13,11,7,4,6,8,12,6])` renvoie la liste `[4,6,6,7,8,11,12,13]`.

Attention ! Il existe aussi une méthode `liste.sort()` qui fonctionne un peu différemment. Cette commande ne renvoie rien, mais par contre la liste `liste` est maintenant ordonnée. On parle de modification *sur place*.

Activité 4 (Arithmétique).

Objectifs : améliorer quelques fonctions de la fiche « Arithmétique – Boucle tant que – I ».

1. **Facteurs premiers.** Programme une fonction `facteurs_preiers(n)` qui renvoie la liste de tous les facteurs premiers d'un entier $n \geq 2$. Par exemple, pour $n = 12936$, dont la décomposition en facteurs premiers est $n = 2^3 \times 3 \times 7^2 \times 11$, la fonction renvoie `[2, 2, 2, 3, 7, 7, 11]`.

Indications. Consulte la fiche « Arithmétique – Boucle tant que – I ». Le corps de l'algorithme est le suivant :

Tant que $d \leq n$:

Si d est un diviseur de n , alors :

ajouter d à la liste,

n devient n/d .

Sinon incrémenter d de 1.

2. **Liste de nombres premiers.** Écris une fonction `liste_preiers(n)` qui renvoie la liste de tous les nombres premiers strictement inférieurs à n . Par exemple `liste_preiers(100)` renvoie la liste :

`[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]`

Pour cela, tu vas programmer un algorithme qui est une version simple du crible d'Ératosthène :

Algorithme.

- — Entrée : un entier $n \geq 2$.
- — Sortie : la liste des nombres premiers $< n$.
- Initialiser `liste` qui contient tous les entiers de 2 à $n-1$.
- Pour d allant de 2 à $n-1$:
 - Pour k parcourant `liste` :
 - Si d divise k et $d \neq k$, alors retirer l'élément k de `liste`
- Renvoyer `liste`.

Indications.

- `Pars de liste = list(range(2,n)).`
- Utilise `liste.remove(k)`.

Explications. Voyons comment fonctionne l'algorithme avec $n = 30$.

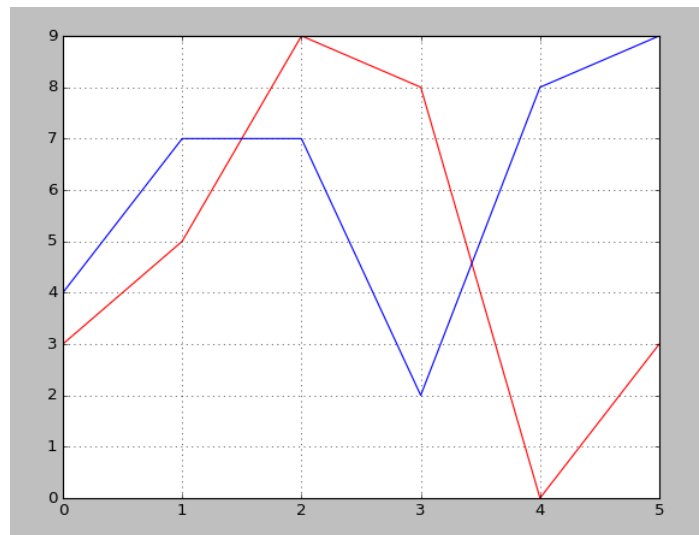
- Au départ la liste est

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]

- On part avec $d = 2$, on élimine tous les nombres divisibles par 2, sauf si c'est le nombre 2 : on élimine donc 4, 6, 8, ..., la liste est maintenant : [2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29].
- On continue avec $d = 3$, on élimine les multiples de 3 (sauf 3), après ces opérations la liste est : [2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29].
- Avec $d = 4$, on élimine les multiples de 4 (mais il n'y en a plus).
- Avec $d = 5$ on élimine les multiples de 5 (ici on élimine juste 25), la liste devient [2, 3, 5, 7, 11, 13, 17, 19, 23, 29].
- On continue (ici il ne se passe plus rien).
- À la fin, la liste vaut [2, 3, 5, 7, 11, 13, 17, 19, 23, 29].

Cours 6 (Visualiser une liste).

Avec le module `matplotlib` il est très facile de visualiser les éléments d'une liste de nombres.



```
import matplotlib.pyplot as plt
```

```
liste1 = [3,5,9,8,0,3]
```

```
liste2 = [4,7,7,2,8,9]
```

```
plt.plot(liste1,color="red")
```

```
plt.plot(liste2,color="blue")
```

```
plt.grid()
```

```
plt.show()
```

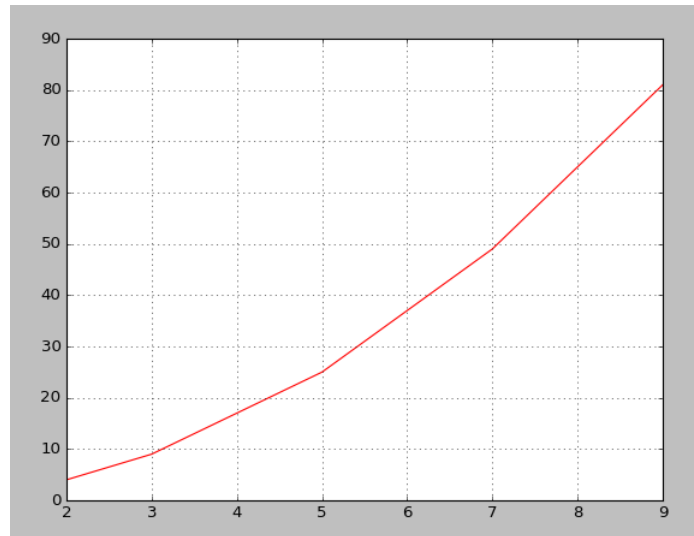
Explications.

- Le module s'appelle `matplotlib.pyplot` et on lui donne le nouveau nom plus simple de `plt`.
- Attention! Le module `matplotlib` n'est pas toujours installé par défaut avec Python.
- `plt.plot(liste)` trace les points d'une liste (sous la forme (i, ℓ_i)) qui sont reliés par des segments.
- `plt.grid()` trace une grille.
- `plt.show()` affiche tout.

Pour afficher des points (x_i, y_i) il faut fournir la listes des abscisses puis la listes des ordonnées :

```
plt.plot(liste_x,liste_y,color="red")
```

Voici un exemple de graphe obtenu en affichant des points de coordonnées du type (x, y) avec $y = x^2$.



```
import matplotlib.pyplot as plt
```

```
liste_x = [2, 3, 5, 7, 9]
```

```
liste_y = [4, 9, 25, 49, 81]
```

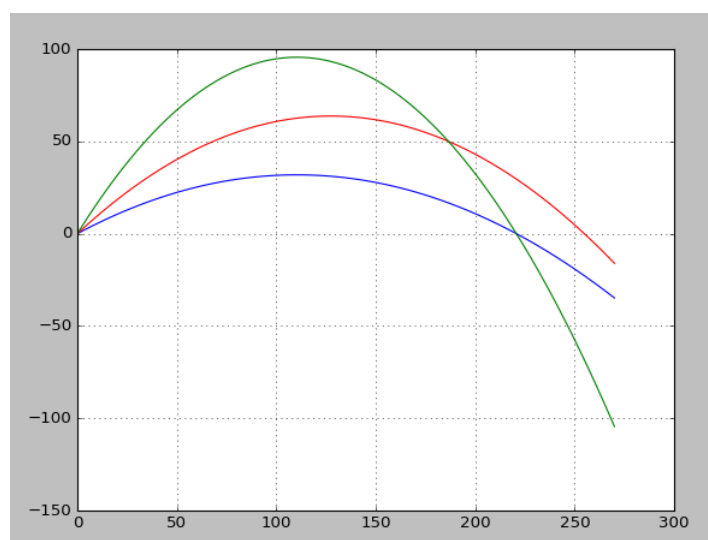
```
plt.plot(liste_x,liste_y,color="red")
```

```
plt.grid()
```

```
plt.show()
```

Activité 5 (Tir balistique).

Objectifs : visualiser le tir d'un boulet de canon.

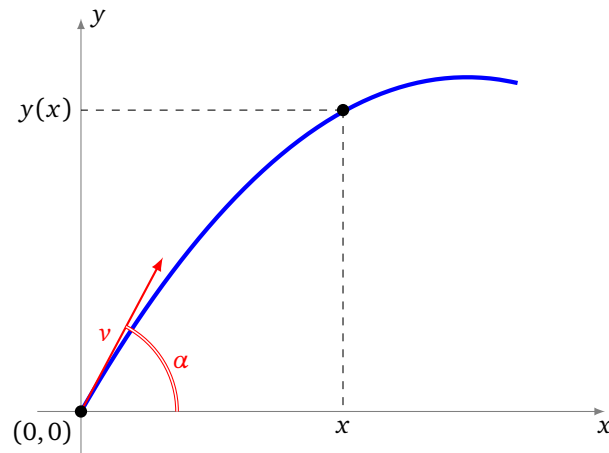


On tire un boulet de canon depuis l'origine $(0, 0)$. L'équation de la trajectoire est donnée par la formule :

$$y(x) = -\frac{1}{2}g \frac{1}{v^2 \cos^2(\alpha)} x^2 + \tan(\alpha)x$$

où

- α est l'angle du tir,
- v est la vitesse initiale,
- g est la constante de gravitation : on prendra $g = 9.81$.



1. Programme une fonction `tir_parabolique(x, v, alpha)` qui renvoie la valeur $y(x)$ donnée par la formule.

Indication. Fais attention aux unités pour l'angle α . Si par exemple tu choisis que l'unité pour l'angle soit les degrés, alors pour appliquer la formule avec Python il faut d'abord convertir les angles en radians :

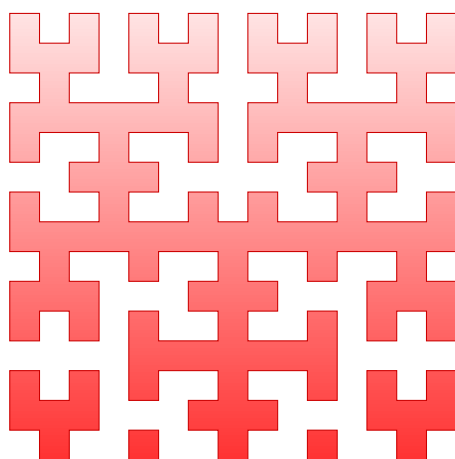
$$\alpha_{\text{radians}} = \frac{2\pi}{360} \alpha_{\text{degrés}}$$

2. Programme une fonction `liste_trajetoire(xmax, n, v, alpha)` qui calcule la liste des ordonnées y des $n + 1$ points de la trajectoire dont les abscisses sont régulièrement espacées entre 0 et x_{max} .

Méthode. Pour i allant de 0 à n :

- calcule $x_i = i \cdot \frac{x_{\text{max}}}{n}$,
 - calcule $y_i = y(x_i)$ par la formule de la trajectoire,
 - ajoute y_i à la liste.
3. Pour $v = 50$, $x_{\text{max}} = 270$ et $n = 100$, affiche différentes trajectoires selon les valeurs de l'angle α . Quel angle α permet d'atteindre le point $(x, 0)$ au niveau du sol le plus éloigné possible du point de tir ?

TROISIÈME PARTIE



NOTIONS AVANCÉES

Statistique – Visualisation de données

Chapitre

8

C'est bien de savoir calculer le minimum, le maximum, la moyenne, les quartiles d'une série. C'est encore mieux de les visualiser tous sur un même graphique !

Activité 1 (Statistique de base).

Objectifs : calculer les principales caractéristiques d'une série de données : minimum, maximum, moyenne et écart-type.

Dans cette activité `liste` désigne une liste de nombres (entiers ou flottants).

1. Écris ta propre fonction `somme(liste)` qui calcule la somme des éléments d'une liste donnée. Compare ton résultat avec la fonction `sum()` décrite ci-dessous qui existe déjà en Python. En particulier pour une liste vide vérifie que ton résultat est bien 0.

python : `sum()`

Usage : `sum(liste)`
Entrée : une liste de nombres
Sortie : un nombre
Exemple : `sum([4,8,3])` renvoie 15

Tu peux maintenant utiliser la fonction `sum()` dans tes programmes !

2. Écris une fonction `moyenne(liste)` qui calcule la moyenne des éléments d'une liste donnée (et renvoie 0 si la liste est vide).
3. Écris ta propre fonction `minimum(liste)` qui renvoie la plus petite valeur des éléments d'une liste donnée. Compare ton résultat avec la fonction Python `min()` décrite ci-dessous (qui en plus sait calculer le minimum de deux nombres).

python : `min()`

Usage : `min(liste)` ou `min(a,b)`
Entrée : une liste de nombres ou bien deux nombres
Sortie : un nombre
Exemple :

- `min(12,7)` renvoie 7
- `min([10,5,9,12])` renvoie 5

Tu peux maintenant utiliser la fonction `min()`, et aussi bien sûr la fonction `max()` dans tes programmes !

4. La **variance** d'une série de données (x_1, x_2, \dots, x_n) est définie comme la moyenne des carrés des écarts à la moyenne. C'est-à-dire :

$$v = \frac{1}{n}((x_1 - m)^2 + (x_2 - m)^2 + \dots + (x_n - m)^2)$$

où m est la moyenne de (x_1, x_2, \dots, x_n) .

Écris une fonction `variance(liste)` qui calcule la variance des éléments d'une liste.

Par exemple, pour la série (6, 8, 2, 10), la moyenne est $m = 8$, la variance est

$$v = \frac{1}{4} = ((6 - 8)^2 + (8 - 8)^2 + (2 - 8)^2 + (10 - 8)^2) = 8.75.$$

5. L'**écart-type** d'une série (x_1, x_2, \dots, x_n) est la racine carrée de la variance :

$$e = \sqrt{v}$$

où v est la variance. Programme une fonction `ecart_type(liste)`.

6. Voici les températures mensuelles moyennes à Brest et à Strasbourg.

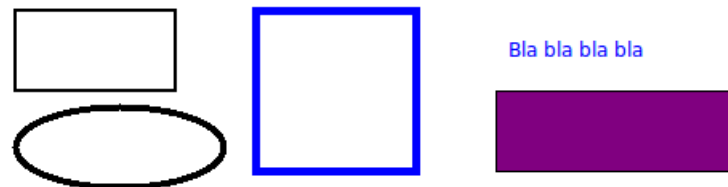
`temp_brest = [6.4, 6.5, 8.5, 9.7, 11.9, 14.6, 15.9, 16.3, 15.1, 12.2, 9.2, 7.1]`

`temp_strasbourg = [0.9, 2.4, 6.1, 9.7, 13.8, 17.2, 19.2, 18.6, 15.7, 10.7, 5.3, 2.1]`

Calcule la température moyenne sur l'année à Brest puis à Strasbourg. Calcule l'écart-type des températures à Brest puis à Strasbourg. Quelles conclusions en tires-tu ?

Cours 1 (Graphiques avec tkinter).

Pour afficher ceci :



Le code est :

```
# Module tkinter
from tkinter import *

# Fenêtre tkinter
root = Tk()

canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)

# Un rectangle
canvas.create_rectangle(50, 50, 150, 100, width=2)

# Un rectangle à gros bords bleus
canvas.create_rectangle(200, 50, 300, 150, width=5, outline="blue")
```

```
# Un rectangle rempli de violet
canvas.create_rectangle(350,100,500,150,fill="purple")

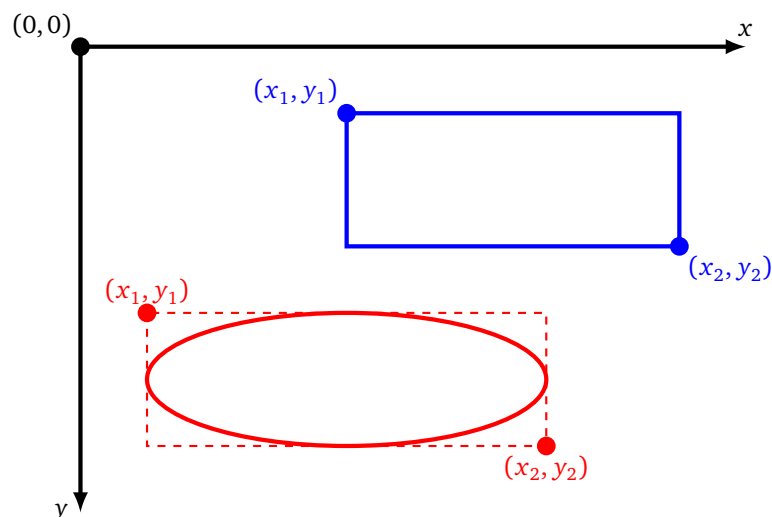
# Un ovale
canvas.create_oval(50,110,180,160,width=4)

# Du texte
canvas.create_text(400,75,text="Bla bla bla bla",fill="blue")

# Ouverture de la fenêtre
root.mainloop()
```

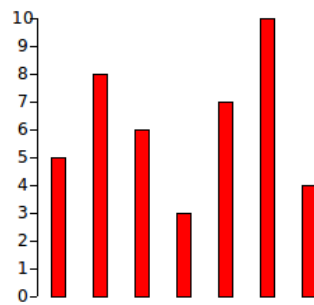
Quelques explications :

- Le module tkinter nous permet de définir des variables `root` et `canvas` qui définissent une fenêtre graphique (ici de largeur 800 et de hauteur 600 pixels). On décrit ensuite tout ce que l'on veut ajouter dans la fenêtre. Et enfin la fenêtre est affichée par la commande `root.mainloop()` (tout à la fin).
- Attention ! Le repère graphique de la fenêtre a son axe des ordonnées dirigé vers le bas. L'origine $(0,0)$ est le coin en haut à gauche (voir la figure ci-dessous).
- Commande pour tracer un rectangle : `create_rectangle(x1, y1, x2, y2)` ; il suffit de préciser les coordonnées (x_1, y_1) , (x_2, y_2) de deux sommets opposés. L'option `width` ajuste l'épaisseur du trait, `outline` définit la couleur de ce trait, `fill` définit la couleur de remplissage.
- Une ellipse est tracée par la commande `create_oval(x1, y1, x2, y2)`, où (x_1, y_1) , (x_2, y_2) sont les coordonnées de deux sommets opposés d'un rectangle encadrant l'ellipse voulue (voir la figure). On obtient un cercle lorsque le rectangle correspondant est un carré !
- Du texte est affiché par la commande `canvas.create_text(x, y, text="Mon texte")` en précisant les coordonnées (x, y) du point à partir duquel on souhaite afficher le texte.

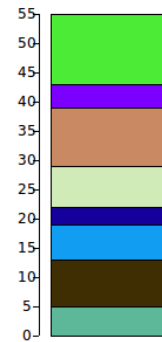


Activité 2 (Graphiques).

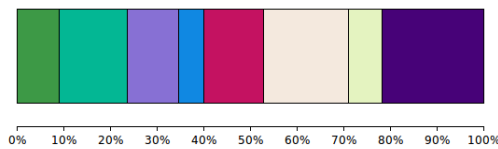
Objectifs : visualiser des données par différents types de graphiques.



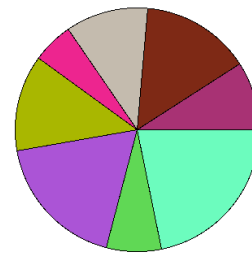
Graphique en barres



Graphique cumulatif



Graphique en pourcentage



Graphique en secteurs

1. **Graphique en barres.** Écris une fonction `graphique_barres(liste)` qui affiche les valeurs d'une liste sous la forme de barres verticales.

Indications :

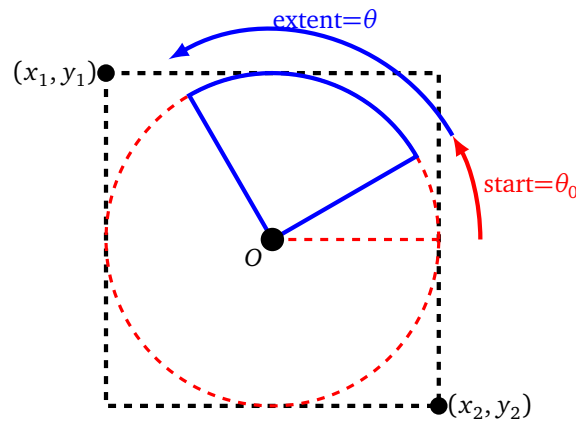
- Dans un premier temps, ne t'occupe pas de tracer l'axe vertical des coordonnées avec les indications chiffrées.
- Tu peux définir une variable `echelle` qui permet d'agrandir tes rectangles, afin qu'ils aient une taille adaptée à l'écran.
- Si tu souhaites tester ton graphique avec une liste au hasard voici comment construire une liste aléatoire de 10 entiers compris entre 1 et 20 :

```
from random import *
liste = [randint(1,20) for i in range(10)]
```

2. **Graphique cumulatif.** Écris une fonction `graphique_cumulatif(liste)` qui affiche les valeurs d'une liste sous la forme de rectangles les uns au-dessus des autres.
3. **Graphique en pourcentage.** Écris une fonction `graphique_pourcentage(liste)` qui affiche les valeurs d'une liste sous la forme d'un rectangle horizontal de taille fixe (par exemple 500 pixels) et qui est divisé en sous-rectangles représentant les valeurs.
4. **Graphique en secteurs.** Écris une fonction `graphique_secteurs(liste)` qui affiche les valeurs d'une liste sous la forme d'un disque de taille fixe et divisé en secteurs représentant les valeurs.

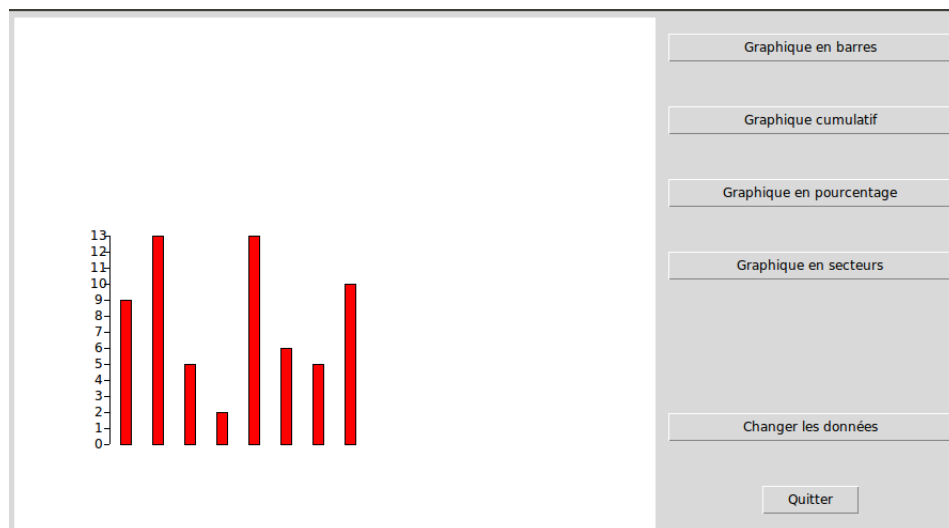
La fonction `create_arc()`, qui permet de dessiner des arcs de cercles, n'est pas très intuitive. Il faut penser que l'on dessine un cercle, en précisant les coordonnées des coins d'un carré qui l'entoure, puis en précisant l'angle de début et l'angle du secteur (en degrés).

```
canvas.create_arc(x1,y1,x2,y2,start=debut_angle,extent=mon_angle)
```



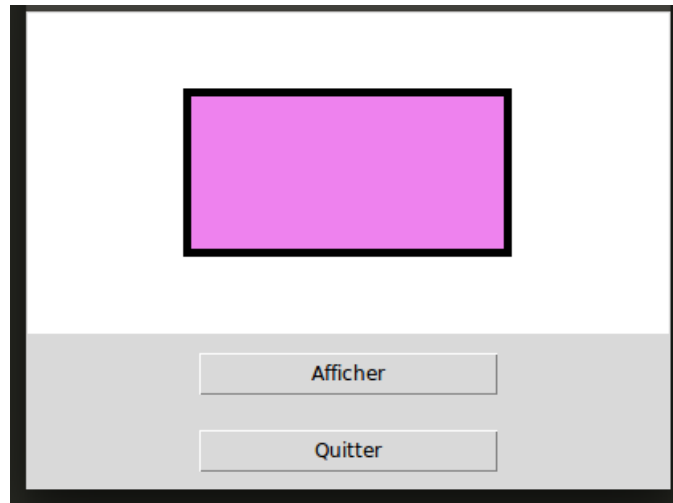
L'option `style=PIESLICE` affiche un secteur au lieu d'un arc.

5. **Bonus.** Rassemble ton travail en un programme qui laisse la possibilité à l'utilisateur de choisir le diagramme qu'il souhaite en cliquant sur des boutons, et aussi la possibilité d'obtenir une nouvelle série aléatoire de données. *Pour afficher et gérer les boutons avec `tkinter`, vois le cours ci-dessous.*



Cours 2 (Boutons avec `tkinter`).

Il est plus ergonomique d'afficher des fenêtres où les actions sont exécutées en cliquant sur des boutons. Voici la fenêtre d'un petit programme avec deux boutons. Le premier bouton change la couleur du rectangle, le second termine le programme.



Le code est :

```
from tkinter import *
from random import *

root = Tk()
canvas = Canvas(root, width=400, height=200, background="white")
canvas.pack(fill="both", expand=True)

def action_bouton():
    canvas.delete("all")          # Efface tout
    couleurs = ["red", "orange", "yellow", "green", "cyan", "blue", "violet", "purple"]
    coul = choice(couleurs)       # Couleur au hasard
    canvas.create_rectangle(100, 50, 300, 150, width=5, fill=coul)
    return

bouton_couleur = Button(root, text="Afficher", width=20, command=action_bouton)
bouton_couleur.pack()

bouton_quitter = Button(root, text="Quitter", width=20, command=root.quit)
bouton_quitter.pack()

root.mainloop()
```

Quelques explications :

- On crée un bouton par la commande `Button`. L'option `text` personnalise le texte qui s'affiche sur le bouton. On ajoute le bouton créé à la fenêtre par la méthode `pack`.
- Le plus important est l'action associée au bouton ! C'est l'option `command` qui reçoit le nom de la fonction à exécuter lorsque le bouton est cliqué. Pour notre exemple `command=action_bouton`, associe au clic sur le bouton un changement de couleur.
- Attention ! il faut donner le nom de la fonction sans parenthèses : `command=ma_fonction` et pas `command=ma_fonction()`.
- Pour associer au bouton « Quitter » la fermeture du programme, l'argument est `command=root.quit`.
- La commande `canvas.delete("all")` efface tous les dessins de notre fenêtre graphique.

Activité 3 (Médiane et quartiles).

Objectifs : calculer la médiane et les quartiles d'un effectif.

1. Écris une fonction `mediane(liste)` qui calcule la valeur médiane des éléments d'une liste donnée. Par définition, la moitié des valeurs est inférieure ou égale à la médiane, l'autre moitié est supérieure ou égale à la médiane.

Rappels. On note n la longueur de la liste et on suppose que la liste est ordonnée (du plus petit au plus grand élément).

- **Cas n impair.** La médiane est la valeur de la liste au rang $\frac{n-1}{2}$. Exemple avec `liste = [12,12,14,15,19]` :
 - la longueur de la liste est $n = 5$ (les indices vont de 0 à 4),
 - l'indice du milieu est l'indice 2,
 - la médiane est la valeur `liste[2]`, c'est donc 14.
- **Cas n pair.** La médiane est la moyenne entre la valeur de la liste au rang $\frac{n}{2} - 1$ et au rang $\frac{n}{2}$. Exemple avec `liste = [13,14,19,20]` :
 - la longueur de la liste est $n = 4$ (les indices vont de 0 à 3),
 - les indices du milieu sont 1 et 2,
 - la médiane est la moyenne entre `liste[1]` et `liste[2]`, c'est donc $\frac{14+19}{2} = 16.5$.

2. Les résultats d'une classe sont collectés sous la forme suivante d'un effectif par note :

`effectif_notes = [0,0,0,0,0,1,0,2,0,1,5,1,2,3,2,4,1,2,0,1,0]`

Le rang i va de 0 à 20. Et la valeur au rang i indique le nombre d'élèves ayant eu la note i . Par exemple ici, 1 élève a eu la note 5, 2 élèves ont eu la note 7, ..., 5 élèves ont obtenus 10, ... Écris une fonction `notes_vers_liste(effectif_notes)` qui prend en entrée un effectif de notes et renvoie la liste des notes. Pour notre exemple la fonction doit renvoyer `[5,7,7,9,10,10,10,10,10,10,...]`.

Déduis-en une fonction qui calcule la médiane des notes d'une classe à partir d'un effectif par note.

3. Écris une fonction `calcule_quartiles(liste)` qui calcule les quartiles Q_1 , Q_2 , Q_3 des éléments d'une liste donnée. Les quartiles répartissent les valeurs en : un quart en-dessous de Q_1 , un quart entre Q_1 et Q_2 , un quart entre Q_2 et Q_3 , un quart au-dessus de Q_3 . Pour le calcul, on utilisera que :
 - Q_2 est simplement la médiane de la liste entière (supposée ordonnée),
 - Q_1 est la médiane de la sous-liste formée de la première moitié des valeurs,
 - Q_3 est la médiane de la sous-liste formée de la seconde moitié des valeurs.

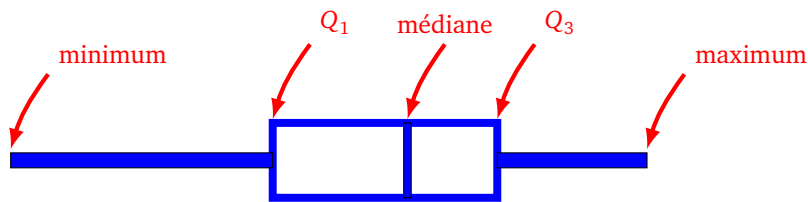
Pour l'implémentation, il faut une nouvelle fois discuter selon que la longueur n de la liste est paire ou pas.

Déduis-en une fonction qui calcule les quartiles des notes d'une classe à partir d'un effectif par note.

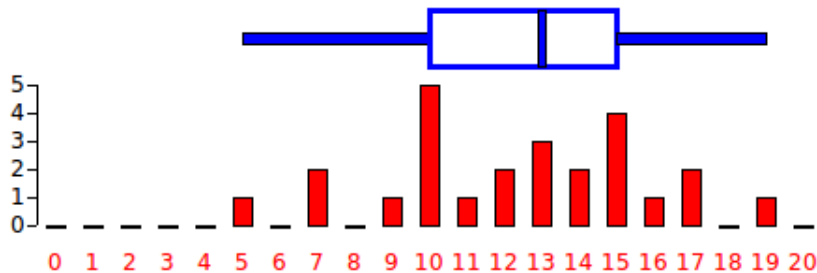
Activité 4 (Diagramme en boîte).

Objectifs : tracer des diagrammes en boîte.

Un **diagramme en boîte** (appelé aussi **boîte à moustaches**) est un graphique qui représente les principales caractéristiques d'une série statistique : minimum, maximum, médiane et quartiles. Le schéma de principe est le suivant :



Écris une fonction `diagramme_boite(effectif_notes)` qui trace le diagramme en boîte des notes d'une classe à partir d'un effectif par note (voir l'activité précédente).



Activité 5 (Moyenne mobile).

Objectifs : calculer des moyennes mobiles afin de « lisser » des courbes.

1. Simule le cours de la bourse de l'indice *Top 40* sur 365 jours. Au jour $j = 0$, l'indice vaut 1000. Ensuite l'indice d'un jour est déterminé en ajoutant une valeur au hasard (positive ou négative) à la valeur de l'indice de la veille :

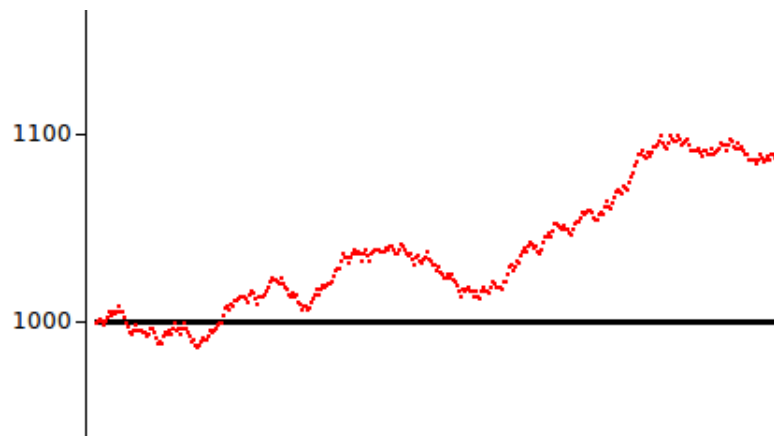
$$\text{indice du jour } j = \text{indice du jour } (j - 1) + \text{valeur au hasard}$$

Pour cette valeur au hasard, tu peux essayer une formule du style :

$$\text{valeur} = \text{randint}(-10, 12) / 3$$

Écris une fonction `cours_bourse()`, sans paramètre, qui renvoie une liste de 365 valeurs de l'indice *Top 40* selon cette méthode.

2. Trace point par point la courbe du cours sur une année. (Pour tracer un point, tu peux afficher un carré de taille 1 pixel.)

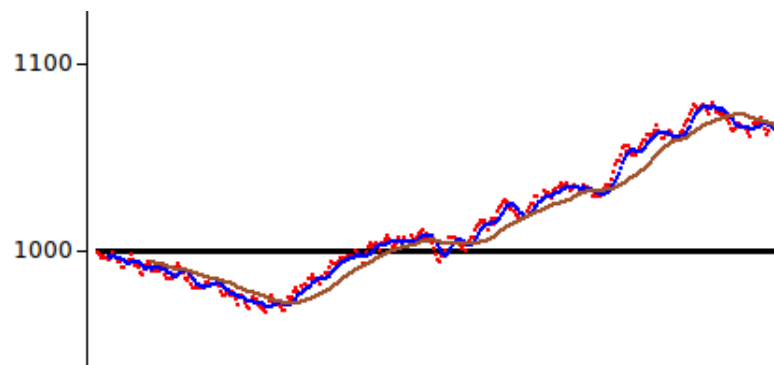


3. Comme la courbe du cours journalier est très chaotique, nous souhaitons la lisser afin de la rendre plus lisible. Pour cela nous calculons des moyennes mobiles.

La moyenne mobile à 7 jours pour le jour j , est la moyenne des 7 derniers cours. Par exemple : la moyenne mobile (à 7 jours) pour le jour $j = 7$ est la moyenne des cours des jours $j = 1, 2, 3, 4, 5, 6, 7$. On peut changer la durée : par exemple la moyenne mobile à 30 jours est la moyenne des 30 derniers cours.

Écris une fonction `moyenne_mobile(liste, duree)` qui renvoie la liste de toutes les moyennes mobiles d'une série de données, pour une durée fixée.

4. Trace point par point sur un même graphique : la courbe du cours sur une année (en rouge ci-dessous), la courbe de ses moyennes mobiles à 7 jours (en bleu ci-dessous) et la courbe des ses moyennes mobiles à 30 jours (en marron ci-dessous). Note que plus la durée est longue plus la courbe est « lisse ». (Bien sûr la courbe des moyennes mobiles à 30 jours ne commence qu'à partir du trentième jour.)



Fichiers

Tu vas apprendre à lire et à écrire des données dans des fichiers.

Cours 1 (Écrire dans un fichier).

Écrire dans un fichier est presque aussi facile que d'afficher une phrase à l'écran. Voici à gauche un programme qui écrit deux lignes dans un fichier appelé `mon_fichier.txt` ; à droite le fichier résultant qui s'affiche dans un éditeur de texte.

```
fic = open("mon_fichier.txt", "w")

fic.write("Bonjour le monde\n")

ligne = "Coucou\n"
fic.write(ligne)

fic.close()
```

Bonjour le monde
Coucou

Explications.

- La commande `open` permet d'ouvrir un fichier. Le premier argument est le nom du fichier. Le second argument est ici `"w"` pour dire que l'on veut écrire dans le fichier (*write* en anglais).
- On ne travaille pas avec le nom du fichier, mais avec la valeur renvoyée par la fonction `open`. Ici nous avons nommé `fic` ce fichier-objet. C'est avec cette variable `fic` que l'on travaille désormais.
- On écrit maintenant dans le fichier presque comme on afficherait une phrase à l'écran. L'instruction est `fic.write()` où l'argument est une chaîne de caractères.
- Pour passer à la ligne, il faut ajouter le caractère de fin de ligne `"\n"`.
- Il est important de fermer son fichier quand on a fini d'écrire. La commande est `fic.close()`.
- Les données à écrire sont des chaînes, donc pour écrire un nombre, il faut d'abord le transformer par `str(nombre)`.

Cours 2 (Lire un fichier).

C'est tout aussi facile de lire un fichier. Voici comment faire (à gauche) et l'affichage par Python à l'écran (à droite).

```

fic = open("mon_fichier.txt", "r")

for ligne in fic:
    print(ligne)

fic.close()

```

Bonjour le monde

Coucou

Explications.

- La commande `open` est cette fois appelée avec l'argument `"r"` (pour *read*), elle ouvre le fichier en lecture.
- On travaille de nouveau avec un fichier-objet nommé ici `fic`.
- Une boucle parcourt tout le fichier ligne par ligne. Ici on demande juste l'affichage de chaque ligne.
- On ferme le fichier avec `fic.close()`.
- Les données lues sont des chaînes, donc pour obtenir un nombre, il faut d'abord le transformer par `int(chaine)` (pour un entier) ou `float(chaine)` (pour un nombre à virgule).

Activité 1 (Lire et écrire un fichier).

Objectifs : écrire un fichier de notes, puis le lire pour calculer les moyennes.

1. Génère au hasard un fichier de notes, nommé `notes.txt`, qui est composé de lignes ayant la structure :

```
Prenom Nom note1 note2 note3
```

Par exemple :

```

Robin Dubois 5.0 16.5 19.5
Tintin Dubois 8.5 15.5 16.5
Gargamel Lupin 15.0 18.0 5.5
Tintin Tchoupi 18.0 11.0 13.0
Hermione Lupin 19.5 10.5 9.0
James Tchoupi 10.0 11.0 6.0
Alice Voldemort 13.0 16.5 20.0

```

Indications.

- Construis une liste de prénoms `liste_prenoms = ["Tintin", "Harry", "Alice", ...]`. Puis choisis un prénom au hasard par la commande `prenom = choice(liste_prenoms)`. (il faut importer le module `random`).
 - Même chose pour les noms !
 - Pour une note, tu peux choisir un nombre au hasard avec la commande `randint(a,b)`.
 - **Attention !** N'oublie pas de convertir les nombres en une chaîne de caractères pour l'écrire dans le fichier : `str(note)`.
2. Lis le fichier `notes.txt` que tu as produit. Calcule la moyenne de chaque personne et écrit le résultat dans un fichier `moyennes.txt` où chaque ligne est de la forme :

```
Prenom Nom moyenne
```

Par exemple :

```
Robin Dubois 13.67
Tintin Dubois 13.50
Gargamel Lupin 12.83
Tintin Tchoupi 14.00
Hermione Lupin 13.00
James Tchoupi 9.00
Alice Voldemort 16.50
```

Indications.

- Pour chaque ligne lue du fichier `notes.txt`, tu récupères les données dans une liste par la commande `ligne.split()`.
- **Attention !** Les données lues sont des chaînes de caractères. Tu peux convertir une chaîne "12.5" en le nombre 12.5 par la commande `float(chaine)`.
- Pour convertir un nombre en une chaîne avec seulement deux décimales après la virgule, tu peux utiliser la commande `'{0:.2f}'.format(moyenne)`.
- N'oublie pas de fermer tous tes fichiers.

Cours 3 (Fichiers au format csv).

Le format `csv` (pour *comma-separated values*) est un format très simple de fichier texte contenant des données. Chaque ligne du fichier contient des données (des nombres ou du texte). Sur une même ligne les données sont séparées par une virgule (d'où le nom du format, même si d'autres séparateurs sont possibles).

Exemple. Voici un fichier qui contient les noms, prénoms, années de naissance, la taille ainsi que le nombre de prix Nobel reçus :

```
CURIE,Marie,1867,1.55,2
EINSTEIN,Albert,1879,1.75,1
NOBEL,Alfred,1833,1.70,0
```

Activité 2 (Format csv).

Objectifs : écrire un fichier de données au format csv, puis le lire pour un affichage graphique.

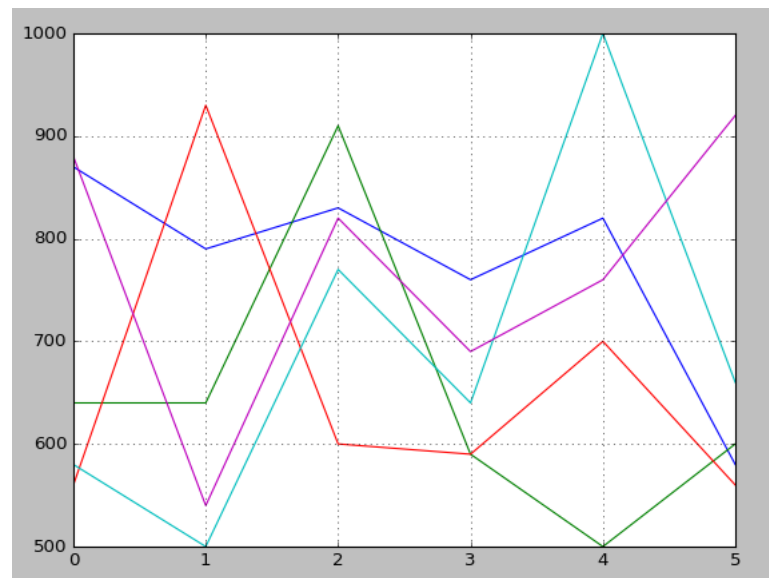
1. Génère un fichier `ventes.csv` des chiffres de ventes (tirés au hasard) d'une enseigne de sport. Voici un exemple :

```
Meilleures ventes de la société 'Pentathlon'
,2013,2014,2015,2016,2017,2018
Vélo VTT,870,790,830,760,820,580
Planche de surf,640,640,910,590,500,600
Chaussures de courses,560,930,600,590,700,560
Raquette de badminton,580,500,770,640,1000,660
Ballon de volley,880,540,820,690,760,920
```

- Les données débutent à partir de la cinquième ligne.
- Le fichier produit respecte le format `csv` et doit pouvoir être lu par *LibreOffice Calc* par exemple.

	A	B	C	D	E	F	G	H
1	Meilleures ventes de la société 'Pentathlon'							
2								
3		2013	2014	2015	2016	2017	2018	
4								
5	Vélo VTT	870	790	830	760	820	580	
6	Planche de surf	640	640	910	590	500	600	
7	Chaussures de courses	560	930	600	590	700	560	
8	Raquette de badminton	580	500	770	640	1000	660	
9	Ballon de volley	880	540	820	690	760	920	
10								

2. Lit le fichier `ventes.csv` pour afficher les courbes de ventes.



Indications.

- Le package `matplotlib` permet d'afficher facilement des graphiques, il s'appelle souvent avec l'instruction :

```
import matplotlib.pyplot as plt
```

- Voici comment visualiser deux listes de données `liste1` et `liste2` :

```
plt.plot(liste1)
plt.plot(liste2)
plt.grid()
plt.show()
```

Cours 4 (Images *bitmap*).

Il existe un format simple de fichier, appelé format *bitmap*, qui décrit pixel par pixel une image. Ce format se décline en trois variantes selon que l'image est en noir et blanc, en niveaux de gris ou bien en couleurs.

Image noir et blanc, le format « **pbm** ».

L'image est décrite par des 0 et des 1.

Voici un exemple : le fichier `image_nb.pbm` à gauche (lu comme un fichier texte) et à droite sa visualisation (à l'aide d'un lecteur d'images, ici très agrandi).

```
P1
4 5
1 1 1 1
1 0 0 0
1 1 1 0
1 0 0 0
1 1 1 1
```



Voici la description du format :

- Première ligne : l'identifiant P1.
- Deuxième ligne : le nombre de colonnes puis le nombre de lignes (ici 4 colonnes et 5 lignes).
- Puis la couleur de chaque pixel ligne par ligne : 1 pour un pixel noir, 0 pour un pixel blanc. Attention : c'est contraire à la convention habituelle !

Image en niveaux de gris, le format « pgm ».

L'image est décrite par différentes valeurs pour différents niveaux de gris. Voici un exemple : le fichier `image_gris.pbm` à gauche et à droite sa visualisation.

```
P2
4 5
255
0 0 0 0
192 192 192 192
192 255 128 128
192 255 64 64
192 0 0 0
```



Voici la description du format :

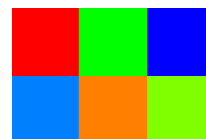
- Première ligne : l'identifiant est cette fois P2.
- Deuxième ligne : le nombre de colonnes puis le nombre de lignes.
- Troisième ligne : la valeur maximale du niveau de gris (ici 255).
- Puis le niveau de gris de chaque pixel ligne par ligne : cette fois 0 pour un pixel noir, la valeur maximale pour un pixel blanc et les valeurs intermédiaires donnent des gris.

Image en couleurs, le format « ppm ».

L'image est décrite par trois valeurs par pixel : une pour le rouge, une pour le vert, une pour le bleu.

Voici un exemple : le fichier `image_coul.ppm` à gauche et à droite sa visualisation.

```
P3
3 2
255
255 0 0 0 255 0 0 0 255
0 128 255 255 128 0 128 255 0
```



Voici la description du format :

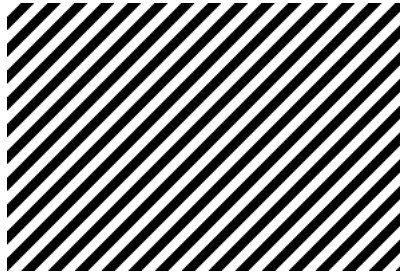
- Première ligne : l'identifiant est maintenant P3.
- Deuxième ligne : le nombre de colonnes puis le nombre de lignes.
- Troisième ligne : la valeur maximale des niveaux de couleur (ici 255).

- Puis chaque pixel est décrit par 3 nombres : le niveau de rouge, celui de vert puis celui de bleu (système RVB, *RGB* en anglais). Par exemple le premier pixel est codé par (255, 0, 0) c'est donc un pixel rouge.

Activité 3 (Images *bitmap*).

Objectifs : définir tes propres images pixel par pixel.

1. Génère un fichier `image_nb.pbm` qui représente une image en noir et blanc (par exemple de taille 300×200) selon le motif suivant :

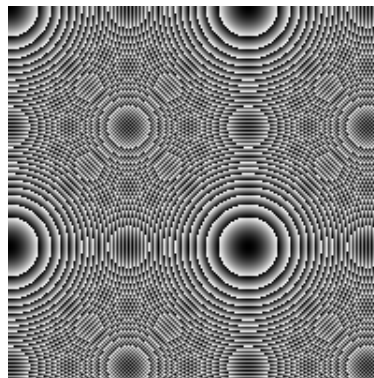


Indications. Si i désigne le numéro de ligne et j le numéro de colonne (en partant du haut à gauche), alors le pixel en position (i, j) est blanc si $i + j$ est compris entre 0 et 9, ou compris entre 20 et 29, ou entre 40 et 49, ... Ce qui s'obtient par la formule :

$$\text{coul} = (i+j) // 10 \% 2$$

qui renvoie 0 ou 1 comme désiré.

2. Génère un fichier `image_gris.pgm` qui représente une image en niveaux de gris (par exemple de taille 200×200 avec 256 niveaux de gris) selon le motif suivant :

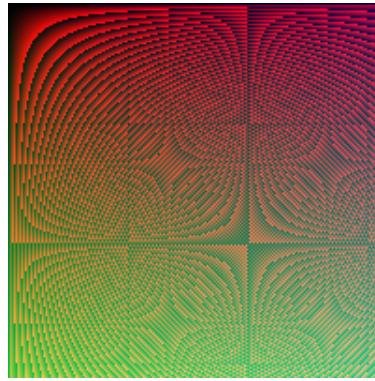


Indications. Cette fois la formule est :

$$\text{coul} = (i**2 + j**2) \% 256$$

qui renvoie un entier entre 0 et 255.

3. Génère un fichier `image_coul.ppm` qui représente une image en couleurs (par exemple de taille 200×200 avec 256 niveaux de rouge, vert et bleu) selon le motif suivant :



Indications. Cette fois la formule est :

$$R = (i*j) \% 256$$

$$V = i \% 256$$

$$B = (i + j) // 3 \% 256$$

qui donne les niveaux de rouge, vert et bleu du pixel (i, j) .

4. Écris une fonction `inverser_couleurs_nb(fichier)` qui lit un fichier image noir et blanc .pbm et crée un nouveau fichier dans lequel les pixels blancs sont devenus noirs et inversement.

Exemple : à gauche l'image de départ, à droite l'image d'arrivée.



5. Écris une fonction `couleurs_vers_gris(fichier)` qui lit un fichier image couleur au format .ppm et crée un nouveau fichier au format .pgm dans lequel les pixels couleurs sont transformés en niveaux de gris.

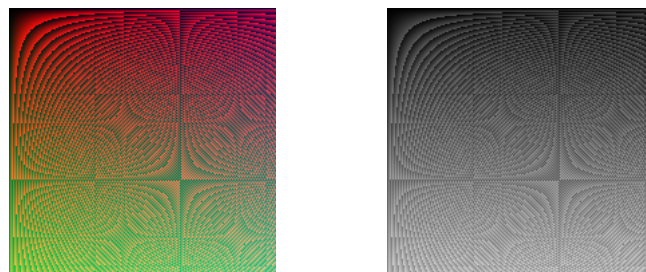
Tu peux utiliser la formule :

$$G = 0,21 \times R + 0,72 \times V + 0,07 \times B$$

où

- R, V, B sont les niveaux de rouge, vert et bleu du pixel coloré,
- G est le niveau de gris du pixel transformé.

Exemple : à gauche l'image de départ en couleur, à droite l'image d'arrivée en niveau de gris.



Activité 4 (Distance entre deux villes).

Objectifs : lire les coordonnées des villes et écrire les distances entre elles.

1. Distance dans le plan.

Écris un programme qui lit un fichier contenant les coordonnées (x, y) de villes, puis qui calcule et écrit dans un autre fichier les distances (dans le plan) entre deux villes.

La formule pour la distance entre deux points (x_1, y_1) et (x_2, y_2) du plan est :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Exemple. Voici un exemple de fichier en entrée :

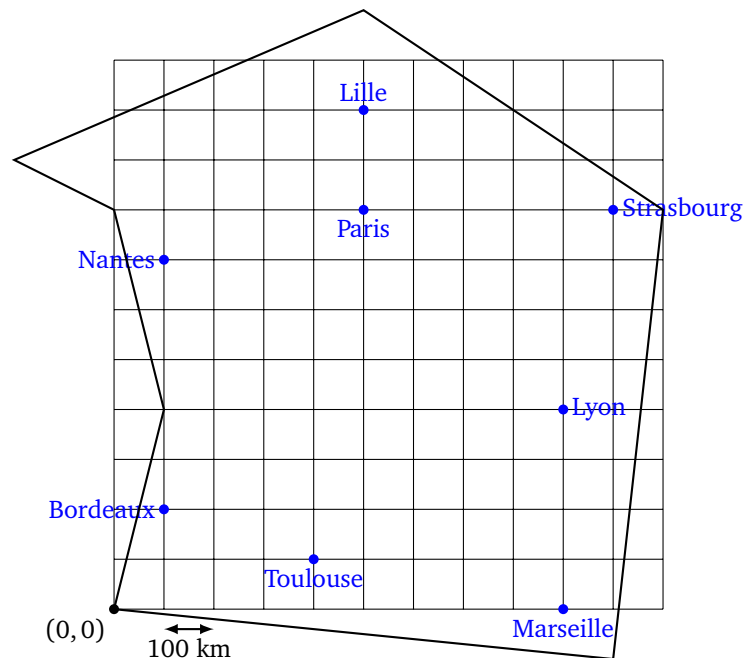
```
Paris 500 800
Lille 500 1000
Nantes 100 700
Marseille 900 0
```

Et voici le fichier de sortie produit par le programme :

	Paris	Lille	Nantes	Marseille
Paris	0	200	412	894
Lille	200	0	500	1077
Nantes	412	500	0	1063
Marseille	894	1077	1063	0

On lit sur ce fichier que la distance entre Lille et Marseille est de 1077 kilomètres.

Ci-dessous la carte de France qui a fourni des données (très approximatives) pour le fichier d'entrée. L'origine est en bas à gauche, chaque côté d'un carré représente 100 km. Par exemple, dans ce repère, Paris a pour coordonnées (500, 800).



2. Distance sur la sphère.

Sur la Terre, la distance entre deux villes correspond à un trajet suivant un *grand cercle* à la surface de la sphère et pas selon une ligne droite. C'est la distance que parcourt un avion pour relier deux villes. Écris un programme qui lit les latitudes et longitudes des villes, puis qui calcule et écrit dans un autre fichier les distances (à la surface de la Terre) entre deux villes.

Exemple. Voici un exemple de fichier en entrée :

```

Paris 48.853 2.350
New-York 40.713 -74.006
Vancouver 49.250 -123.119
Lima -12.043 -77.0282
Hong-Kong 22.286 114.158
Addis-Abeba 9.0250 38.747

```

Et voici le fichier de sortie produit par le programme :

	Paris	New-York	Vancouver	Lima	Hong-Kong	Addis-Abeba
Paris	0	5837	7924	10253	9629	5573
New-York	5837	0	3905	5874	12959	11207
Vancouver	7924	3905	0	8168	10257	13298
Lima	10253	5874	8168	0	18371	13001
Hong-Kong	9629	12959	10257	18371	0	8135
Addis-Abeba	5573	11207	13298	13001	8135	0

Mise en œuvre et explications.

- Le fichier d'entrée contient la latitude (notée φ) et la longitude (notée λ) en degrés de chaque ville. Par exemple Paris a pour latitude $\varphi = 48.853$ degrés et pour longitude $\lambda = 2.350$ degrés.
- Pour les formules, il faudra utiliser les angles en radians. La formule de conversion de degrés vers radians est :

$$\text{angle en radians} = \frac{2\pi}{360} \times \text{angle en degrés}$$

- Formule de la distance approchée.**

Il existe une formule simple qui donne une bonne estimation pour la distance la plus courte entre deux points d'une sphère de rayon R . Poser d'abord :

$$x = (\lambda_2 - \lambda_1) \cdot \cos\left(\frac{\varphi_1 + \varphi_2}{2}\right) \quad \text{et} \quad y = \varphi_2 - \varphi_1$$

La distance approchée est alors

$$\tilde{d} = R\sqrt{x^2 + y^2}$$

(φ_1, λ_1) et (φ_2, λ_2) sont les latitudes/longitudes de deux villes exprimées en radians.

- Formule de la distance exacte.**

Les plus courageux peuvent utiliser la formule exacte pour calculer la distance. Poser d'abord :

$$a = \left(\sin\left(\frac{\varphi_1 + \varphi_2}{2}\right)\right)^2 + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \left(\sin\left(\frac{\lambda_2 - \lambda_1}{2}\right)\right)^2$$

La distance exacte est alors :

$$d = 2 \cdot R \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$$

où $\text{atan2}(y, x)$ est la fonction « arctangente » qui s'obtient par la commande $\text{atan2}(y, x)$ du module `math`.

- Pour le rayon de la Terre on prendra $R = 6371$ km.

Arithmétique – Boucle tant que – II

Chapitre 10

On approfondit notre étude des nombres avec la boucle « tant que ». Pour cette fiche tu as besoin d'une fonction `est_premier()` construite dans la fiche « Arithmétique – Boucle tant que – I ».

Activité 1 (Conjecture(s) de Goldbach).

Objectifs : étudier deux conjectures de Goldbach. Une conjecture est un énoncé que l'on pense vrai mais que l'on ne sait pas démontrer.

1. La bonne conjecture de Goldbach : *Tout entier pair plus grand que 4 est la somme de deux nombres premiers.*

Par exemple $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 3 + 7$ (mais aussi $10 = 5 + 5$), $12 = 5 + 7$,... Pour $n = 100$ il y a 6 solutions : $100 = 3 + 97 = 11 + 89 = 17 + 83 = 29 + 71 = 41 + 59 = 47 + 53$.

Personne ne sait démontrer cette conjecture, mais tu vas voir qu'il y a de bonnes raisons de penser qu'elle est vraie.

- (a) Programme une fonction `nombre_solutions_goldbach(n)` qui pour un entier pair n donné, trouve combien il existe de décompositions $n = p + q$ avec p et q deux nombres premiers et $p \leq q$. Par exemple pour $n = 8$, il n'y a qu'une seule solution $8 = 3 + 5$, par contre pour $n = 10$ il y a deux solutions $10 = 3 + 7$ et $10 = 5 + 5$.

Indications.

- Il faut donc tester tous les p compris 2 et $n/2$;
- poser $q = n - p$;
- on a une solution quand $p \leq q$ et que p et q sont tous les deux des nombres premiers.

- (b) Prouve avec la machine que la conjecture de Goldbach est vérifiée pour tous les entiers n pairs compris entre 4 et 10 000.

2. La mauvaise conjecture de Goldbach : *Tout entier impair n peut s'écrire sous la forme*

$$n = p + 2k^2$$

où p est un nombre premier et k un entier (éventuellement nul).

- (a) Programme une fonction `existe_decomposition_goldbach(n)` qui renvoie « vrai » lorsqu'il existe une décomposition de la forme $n = p + 2k^2$.
- (b) Montre que cette seconde conjecture de Goldbach est fausse ! Il existe deux entiers plus petits que 10 000 qui n'admettent pas une telle décomposition. Trouve-les !

Activité 2 (Nombres ayant 4 ou 8 diviseurs).

Objectifs : réfuter une conjecture en faisant beaucoup de calculs !

Conjecture : Entre 1 et N , il y a plus d'entiers qui ont exactement 4 diviseurs que d'entiers qui ont exactement 8 diviseurs.

Tu vas voir que cette conjecture a l'air vrai pour N assez petit, mais tu vas montrer que cette conjecture est fausse en trouvant un N grand qui contredit cet énoncé.

1. Nombre de diviseurs.

Programme une fonction `nombre_de_diviseurs(n)` qui renvoie le nombre d'entiers divisant n .

Par exemple : `nombre_de_diviseurs(100)` renvoie 9 car il y a 9 diviseurs de $n = 100$:

1, 2, 4, 5, 10, 20, 25, 50, 100

Indications.

- N'oublie pas 1 et n comme diviseurs.
- Essaie d'optimiser ta fonction car tu l'utiliseras intensivement : par exemple il n'y a pas de diviseurs strictement plus grands que $\frac{n}{2}$ (à part n).

2. 4 ou 8 diviseurs.

Programme une fonction `quatre_et_huit_diviseurs(Nmin, Nmax)` qui renvoie deux nombres : (1) le nombre d'entiers n avec $N_{\min} \leq n < N_{\max}$ qui admettent exactement 4 diviseurs et (2) le nombre d'entiers n avec $N_{\min} \leq n < N_{\max}$ qui admettent exactement 8 diviseurs.

Par exemple `quatre_et_huit_diviseurs(1, 100)` renvoie (32, 10) car il y a 32 entiers entre 1 et 99 qui admettent 4 diviseurs, mais seulement 10 entiers qui en admettent 8.

3. Preuve que la conjecture est fausse.

Expérimente que pour des « petites » valeurs de N (jusqu'à $N = 10\,000$ par exemple) il y a plus d'entiers ayant 4 diviseurs que 8. Mais calcule que pour $N = 300\,000$ ce n'est plus le cas.

Indications. Comme il y a beaucoup de calculs, tu peux les séparer en tranches (la tranche des entiers $1 \leq n < 50\,000$, puis $50\,000 \leq n < 100\,000$,...) puis fais la somme. Tu peux ainsi partager tes calculs entre plusieurs ordinateurs.

Activité 3 (121111... n'est jamais premier?).

Objectifs : étudier une nouvelle conjecture fausse !

On appelle U_k l'entier :

$$U_k = 12 \underbrace{111 \dots 111}_{k \text{ occurrences de } 1}$$

formé du chiffre 1, puis du chiffre 2, puis de k fois le chiffre 1.

Par exemple $U_0 = 12$, $U_1 = 121$, $U_2 = 1211$,...

1. Écris une fonction `un_deux_un(k)` qui renvoie l'entier U_k .

Indications. Tu peux remarquer qu'en partant de $U_0 = 12$, on a la relation $U_{k+1} = 10 \cdot U_k + 1$. Donc tu peux partir de $u = 12$ et répéter un certain nombre de fois $u = 10 * u + 1$.

2. Vérifie à l'aide de la machine que U_0, \dots, U_{20} ne sont pas des nombres premiers.

On pourrait croire que c'est toujours le cas, mais ce n'est pas vrai ! L'entier U_{136} est un nombre premier ! Malheureusement il est trop grand pour qu'on puisse le vérifier avec nos algorithmes. Dans la suite on va définir ce qu'est un nombre presque premier pour pouvoir pousser plus loin les calculs.

3. Programme une fonction `est_presque_premier(n, r)` qui renvoie « vrai » si l'entier n n'admet aucun diviseur d tel que $1 < d \leq r$ (on suppose $r < n$).

Par exemple : $n = 143 = 11 \times 13$ et $r = 10$, alors `est_presque_premier(n, r)` est « vrai » car n n'admet aucun diviseur inférieur ou égal à 10. (Mais bien sûr, n n'est pas un nombre premier.)

Indications. Modifie ta fonction `est_premier(n)` !

4. Trouve tous les entiers U_k avec $0 \leq k \leq 150$ qui sont presque premiers pour $r = 1\,000\,000$ (c'est-à-dire qu'ils ne sont divisibles par aucun entier d avec $1 < d \leq 1\,000\,000$).

Indications. Dans la liste tu dois retrouver U_{136} (qui est un nombre premier) mais aussi U_{34} qui n'est pas premier mais dont le plus petit diviseur est 10 149 217 781.

Activité 4 (Racine carrée entière).

Objectifs : calculer la racine carrée entière d'un entier.

Soit $n \geq 0$ un entier. La **racine carrée entière de n** est le plus grand entier $r \geq 0$ tel que $r^2 \leq n$. Une autre définition est de dire que la racine carrée entière de n est la partie entière de \sqrt{n} .

Exemples :

- $n = 21$, alors la racine carrée entière de n est 4 (car $4^2 \leq 21$, mais $5^2 > 21$). Autre façon, $\sqrt{21} = 4.58\dots$, on ne retient que la partie entière (l'entier à gauche de la virgule), c'est donc 4.
- $n = 36$, alors la racine carrée entière de n est 6 (car $6^2 \leq 36$, mais $7^2 > 36$). Autre façon, $\sqrt{36} = 6$ et la racine carrée entière est bien sûr aussi 6.

1. Écris une première fonction qui calcule la racine carrée entière d'un entier n , en calculant d'abord \sqrt{n} , puis en prenant la partie entière.

Indications.

- Pour cette question uniquement, tu peux utiliser le module `math` de Python.
- Dans ce module `sqrt()` renvoie la racine carrée réelle.
- La fonction `floor()` du même module renvoie la partie entière d'un nombre.

2. Écris une deuxième fonction qui calcule la racine carrée entière d'un entier n , mais cette fois selon la méthode suivante :

- Pars de $p = 0$.
- Tant que $p^2 \leq n$, incrémente la valeur de p .

Teste bien quelle doit être la valeur renvoyée (attention au décalage !).

3. Écris une troisième fonction qui calcule encore la racine carrée entière d'un entier n avec l'algorithme décrit ci-dessous. Cet algorithme s'appelle la méthode babylonienne ou bien méthode de Héron ou bien encore méthode de Newton !

Algorithme.

Entrée : un entier positif n

Sortie : sa racine carrée entière

- Partir avec $a = 1$ et $b = n$.
- Tant que $|a - b| > 1$:
 — $a \leftarrow (a + b) / 2$;
 — $b \leftarrow n / a$
- Renvoyer le minimum entre a et b : c'est la racine carrée entière de n .

Nous n'expliquons pas comment fonctionne cet algorithme, mais il faut savoir que c'est l'une des méthodes les plus efficaces pour calculer les racines carrées. Les nombres a et b fournissent, au cours de l'exécution, un encadrement de plus en plus précis de \sqrt{n} .

Voici un tableau qui détaille un exemple de calcul pour la racine carrée entière de $n = 1664$.

Étape	a	b
$i = 0$	$a = 1$	$b = 1664$
$i = 1$	$a = 832$	$b = 2$
$i = 2$	$a = 417$	$b = 3$
$i = 3$	$a = 210$	$b = 7$
$i = 4$	$a = 108$	$b = 15$
$i = 5$	$a = 61$	$b = 27$
$i = 6$	$a = 44$	$b = 37$
$i = 7$	$a = 40$	$b = 41$

À la dernière étape, l'écart entre a et b est inférieur ou égal à 1, donc la racine carrée entière est 40. On peut vérifier que c'est exact car : $40^2 = 1600 \leq 1664 < 41^2 = 1681$.

Bonus. Compare les vitesses d'exécution des trois méthodes à l'aide de `timeit()`. Voir la fiche « Fonctions ».

Cours 1 (Quitter une boucle).

Il n'est pas toujours facile de trouver la condition adéquate pour une boucle « tant que ». Python possède une commande pour quitter immédiatement une boucle « tant que » ou une boucle « pour » : c'est l'instruction `break`.

Voici des exemples qui utilisent la commande `break`. Comme c'est rarement une façon élégante d'écrire son programme, des alternatives sont aussi présentées.

Exemple.

Voici différents codes pour un compte à rebours de 10 à 0.

<pre># Compte à rebours n = 10 # Boucle infinie while True: print(n) n = n - 1 if n < 0: break # Arrêt</pre>	<pre># Mieux (avec un drapeau) n = 10 termine = False while not termine: print(n) n = n - 1 if n < 0: termine = True</pre>	<pre># Encore mieux n = 10 while n >= 0: print(n) n = n - 1</pre>
---	---	--

Exemple.

Voici des programmes qui cherchent la racine carrée entière de 777, c'est-à-dire le plus grand entier i qui vérifie $i^2 \leq 777$. La recherche est limitée aux entiers i entre 0 et 99.

```
# Racine carrée entière
n = 777
for i in range(100):
    if i**2 >= n:
        break
print(i-1)
```

```
# Mieux
n = 777
i = 0
while (i**2 < n) and (i < 100):
    i = i + 1
print(i-1)
```

Exemple.

Voici des programmes qui calculent les racines carrées réelles des éléments d'une liste, sauf bien sûr si le nombre est négatif. Le code de gauche s'arrête avant la fin de la liste, alors que le code de droite gère proprement le problème.

```
# Racines carrées des éléments
# d'une liste
liste = [3,7,0,10,-1,12]
for element in liste:
    if element < 0:
        break
    print(sqrt(element))
```

```
# Mieux avec try/except
liste = [3,7,0,10,-1,12]
for element in liste:
    try:
        print(sqrt(element))
    except:
        print("Problème avec",element)
```

Binaire I

Les ordinateurs transforment toutes les données en nombres et manipulent uniquement ces nombres. Ces nombres sont stockés sous la forme de listes de 0 et de 1. C'est l'écriture binaire des nombres ! Pour mieux comprendre l'écriture binaire, tu vas d'abord mieux comprendre l'écriture décimale.

Cours 1 (Écriture décimale).

L'écriture habituelle des entiers se fait dans le système décimal (en base 10). Par exemple, 70 685 c'est $7 \times 10\,000 + 0 \times 1\,000 + 6 \times 100 + 8 \times 10 + 5 \times 1$:

7	0	6	8	5
10000	1000	100	10	1
10^4	10^3	10^2	10^1	10^0

(on voit bien que 5 est le chiffre des unités, 8 celui des dizaines, 6 celui des centaines...).

Il faut bien comprendre les puissances de 10. On note 10^k pour $10 \times 10 \times \dots \times 10$ (avec k facteurs).

d_{p-1}	d_{p-2}	\dots	d_i	\dots	d_2	d_1	d_0
10^{p-1}	10^{p-2}	\dots	10^i	\dots	10^2	10^1	10^0

On calcule donc l'entier correspondant aux chiffres $[d_{p-1}, d_{p-2}, \dots, d_2, d_1, d_0]$ par la formule :

$$n = d_{p-1} \times 10^{p-1} + d_{p-2} \times 10^{p-2} + \dots + d_i 10^i + \dots + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

Activité 1 (De l'écriture décimale vers l'entier).

Objectifs : à partir de l'écriture décimale, retrouver l'entier.

Écris une fonction `decimale_vers_entier(liste_decimale)` qui à partir d'une liste représentant l'écriture décimale calcule l'entier correspondant.

`decimale_vers_entier()`

Usage : `decimale_vers_entier(liste_decimale)`

Entrée : une liste de chiffres entre 0 et 9

Sortie : l'entier dont l'écriture décimale est la liste

Exemple : si l'entrée est `[1, 2, 3, 4]`, la sortie est 1234.

Indications. Il faut faire la somme d'éléments du type

$$d_i \times 10^i \quad \text{pour } 0 \leq i < p$$

où p est la longueur de la liste et d_i est le chiffre en position i en comptant à partir de la fin (c'est-à-dire de droite à gauche). Pour gérer le fait que l'indice i utilisé pour parcourir la liste ne correspond pas à la puissance de 10, il y a deux solutions :

- comprendre que $d_i = \text{liste}[p - 1 - i]$ où `liste` est la liste des p chiffres,
- ou bien commencer par inverser `liste`.

Cours 2 (Binaire).

- **Puissances de 2.** On note 2^k pour $2 \times 2 \times \dots \times 2$ (avec k facteurs). Par exemple, $2^3 = 2 \times 2 \times 2 = 8$.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

- **Écriture binaire : un exemple.**

Tout entier admet une écriture binaire, c'est-à-dire une écriture où les seuls chiffres sont des 0 ou des 1. En binaire, les chiffres sont appelés des **bits**. Par exemple, 1.0.1.1.0.0.1 (prononce les chiffres un par un) est l'écriture binaire de l'entier 89. Comment faire ce calcul ? C'est comme pour la base 10, mais en utilisant les puissances de 2 !

1	0	1	1	0	0	1
64	32	16	8	4	2	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

Donc l'écriture 1.0.1.1.0.0.1 représente l'entier :

$$1 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 64 + 16 + 8 + 1 = 89.$$

- **Écriture binaire : formule.**

b_{p-1}	b_{p-2}	\dots	b_i	\dots	b_2	b_1	b_0
2^{p-1}	2^{p-2}	\dots	2^i	\dots	2^2	2^1	2^0

On calcule donc l'entier correspondant aux bits $[b_{p-1}, b_{p-2}, \dots, b_2, b_1, b_0]$ comme une somme de termes $b_i \times 2^i$, par la formule :

$$n = b_{p-1} \times 2^{p-1} + b_{p-2} \times 2^{p-2} + \dots + b_i 2^i + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

- **Python et le binaire.** Python accepte que l'on écrive directement les entiers en écriture binaire, il suffit d'utiliser le préfixe « 0b ». Exemples :
 - avec `x = 0b11010`, alors `print(x)` affiche 26,
 - avec `y = 0b11111`, alors `print(y)` affiche 31,
 - et `print(x+y)` affiche 57.

Activité 2 (De l'écriture binaire vers l'entier).

Objectifs : à partir de l'écriture binaire, retrouver l'entier (en écriture décimale usuelle).

1. Calcule les entiers dont l'écriture binaire est donnée ci-dessous. Tu peux le faire à la main ou t'aider de Python. Par exemple 1.0.0.1.1 vaut $2^4 + 2^1 + 2^0 = 19$ ce que confirme la commande `0b10011` qui renvoie 19.

- 1.1, 1.0.1, 1.0.0.1, 1.1.1.1
- 1.0.0.0.0, 1.0.1.0.1, 1.1.1.1.1
- 1.0.1.1.0.0, 1.0.0.0.1.1
- 1.1.1.0.0.1.0.1

2. Écris une fonction `binaire_vers_entier(liste_binaire)` qui à partir d'une liste représentant l'écriture binaire calcule l'entier correspondant.

binaire_vers_entier()

Usage : `binaire_vers_entier(liste_binaire)`

Entrée : une liste de *bits* 0 et 1

Sortie : l'entier dont l'écriture binaire est la liste

Exemples :

- entrée `[1, 1, 0]`, sortie 6
- entrée `[1, 1, 0, 1, 1, 1]`, sortie 55
- entrée `[1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1]`, sortie 3383

Indications. Il faut cette fois faire la somme d'éléments du type

$$b_i \times 2^i \quad \text{pour } 0 \leq i < p$$

où p est la longueur de la liste et b_i est le *bit* (0 ou 1) en position i de la liste en comptant à partir de la fin.

3. Voici un algorithme qui effectue le même travail : il permet le calcul de l'entier à partir de l'écriture binaire, mais il a l'avantage de ne jamais calculer directement des puissances de 2. Programme cet algorithme en une fonction `binaire_vers_entier_bis()` qui a les mêmes caractéristiques que la fonction précédente.

Algorithme.

Entrée : `liste` : une liste de 0 et de 1

Sortie : le nombre binaire associé

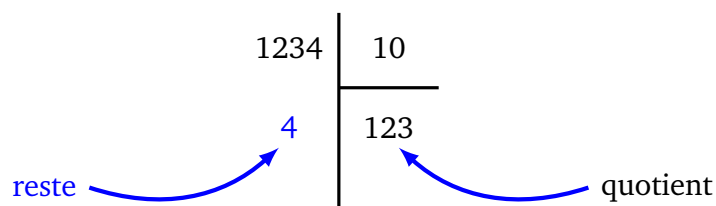
- Initialiser une variable n à 0.
- Pour chaque élément b de `liste` :
 - si b vaut 0, alors faire $n \leftarrow 2n$,
 - si b vaut 1, alors faire $n \leftarrow 2n + 1$.
- Le résultat est la valeur de n .

Cours 3 (Écriture décimale (bis)).

Bien sûr, quand tu vois le nombre 1234 tu sais tout de suite trouver la liste de ses chiffres `[1, 2, 3, 4]`. Mais comment faire en général à partir d'un entier n ?

1. On aura besoin de la division euclidienne par 10 :

- on calcule par `n % 10` le **reste**, on l'appelle aussi **n modulo 10**
- on calcule par `n // 10` le **quotient** de cette division.



2. Les commandes Python sont simplement `n % 10` et `n // 10`.

Exemple : `1234 % 10` vaut 4 ; `1234 // 10` vaut 123.

3.
 - Le chiffre des unités de n s'obtient comme le reste modulo 10 : c'est $n\%10$. Exemple $1234\%10 = 4$.
 - Le chiffre des dizaines s'obtient à partir du quotient de la division de n par 10, puis en prenant le chiffre des unités de ce nombre : c'est donc $(n//10)\%10$. Exemple : $1234//10 = 123$, puis on a $123\%10 = 3$; le chiffre des dizaines de 1234 est bien 3.
 - Pour le chiffre de centaines, on calcule le quotient de la division de n par 100, puis on prend le chiffre des unités. Exemple $1234//100 = 12$; 2 est le chiffre des unités de 12 et c'est le chiffre des centaines de 1234. Remarque : diviser par 100, c'est diviser par 10, puis de nouveau par 10.
 - Pour le chiffre des milliers on calcule le quotient de la division par 1000 puis on prend le chiffre des unités...

Activité 3 (Trouver les chiffres d'un entier).

Objectifs : décomposer un entier en la liste de ses chiffres (en base 10).

Programme l'algorithme suivant en une fonction `entier_vers_decimale()`.

entier_vers_decimale()

Usage : `entier_vers_decimale(n)`

Entrée : un entier positif

Sortie : la liste de ses chiffres

Exemple : si l'entrée est 1234, la sortie est `[1, 2, 3, 4]`.

Algorithme.

Entrée : un entier $n > 0$

Sortie : la liste de ses chiffres

- Partir d'une liste vide.
- Tant que n n'est pas nul :
 - ajouter $n\%10$ au début de la liste,
 - faire $n \leftarrow n//10$.
- Le résultat est la liste.

Cours 4 (Écriture binaire avec Python).

Python calcule très bien l'écriture binaire d'un entier grâce à la fonction `bin()`.

python : bin()

Usage : `bin(n)`

Entrée : un entier

Sortie : l'écriture binaire de n sous la forme d'une chaîne de caractères commençant par `'0b'`

Exemple :

- `bin(37)` renvoie `'0b100101'`
- `bin(139)` renvoie `'0b10001011'`

Cours 5 (Calcul de l'écriture binaire).

Pour calculer l'écriture binaire d'un entier n , c'est la même méthode que pour calculer l'écriture décimale mais en remplaçant les divisions par 10 par des divisions par 2.

Nous avons donc besoin :

- de $n\%2$: le reste de la division euclidienne de n par 2 (appelé aussi n modulo 2) ; le reste vaut soit 0 soit 1.
- de $n//2$: le quotient de cette division.

Note que le reste $n\%2$ vaut soit 0 (quand n est pair) soit 1 (quand n est impair).

Voici la méthode générale pour calculer l'écriture binaire d'un entier :

- On part de l'entier dont on veut l'écriture binaire.
- On effectue une suite de divisions euclidiennes par 2 :
 - à chaque division, on obtient un reste qui vaut 0 ou 1 ;
 - on obtient un quotient que l'on divise de nouveau par 2... On s'arrête quand ce quotient est nul.
- On lit l'écriture binaire comme la suite des restes, mais en partant du dernier reste.

Exemple.

Calcul de l'écriture binaire de 14.

- On divise 14 par 2, le quotient est 7, le reste est 0.
- On divise 7 (le quotient précédent) par 2 : le nouveau quotient est 3, le nouveau reste est 1.
- On divise 3 par 2 : quotient 1, reste 1.
- On divise 1 par 2 : quotient 0, reste 1.
- C'est terminé (le dernier quotient est nul).
- Les restes successifs sont 0, 1, 1, 1. On lit l'écriture binaire à l'envers c'est 1.1.1.0.

Les divisions se font de gauche à droite, mais on lit les restes de droite à gauche.

14	2	7	2	3	2	1	2
0	7	1	3	1	1	1	0

←

Exemple.

Écriture binaire de 50.

50	2	25	2	12	2	6	2	3	2	1	2
0	25	1	12	0	6	0	3	1	1	1	0

Les restes successifs sont 0, 1, 0, 0, 1, 1, donc l'écriture binaire de 50 est 1.1.0.0.1.0.

Activité 4.

Objectifs : trouver l'écriture binaire d'un entier.

1. Calcule à la main l'écriture binaire des entiers suivants. Vérifie tes résultats à l'aide de la fonction

`bin()` de Python.

- 13, 18, 29, 31,
- 44, 48, 63, 64,
- 100, 135, 239, 1023.

2. Programme l'algorithme suivant en une fonction `entier_vers_binaire()`.

Algorithme.

Entrée : un entier $n > 0$

Sortie : son écriture binaire sous la forme d'une liste

- Partir d'une liste vide.
- Tant que n n'est pas nul :
 - ajouter $n\%2$ au début de la liste,
 - faire $n \leftarrow n//2$.
- Le résultat est la liste.

entier_vers_binaire()

Usage : `entier_vers_binaire(n)`

Entrée : un entier positif

Sortie : son écriture binaire sous forme d'une liste

Exemple : si l'entrée est 204, la sortie est `[1, 1, 0, 0, 1, 1, 0, 0]`.

Vérifie que tes fonctions marchent bien :

- pars d'un entier n ,
- calcule son écriture binaire,
- calcule l'entier associé à cette écriture,
- tu dois retrouver l'entier de départ !

Les listes sont tellement utiles qu'il faut savoir les manipuler de façon simple et efficace. C'est le but de cette fiche !

Cours 1 (Manipuler efficacement les listes).

- **Trancher des listes.**

- Tu connais déjà `maliste[a:b]` qui renvoie la sous-liste des éléments du rang a au rang $b - 1$.
- `maliste[a:]` renvoie la liste des éléments du rang a jusqu'à la fin.
- `maliste[:b]` renvoie la liste des éléments du début jusqu'au rang $b - 1$.
- `maliste[-1]` renvoie le dernier élément, `maliste[-2]` renvoie l'avant-dernier élément,...
- **Exercice.**

7	2	4	5	3	10	9	8	3
rang : 0	1	2	3	4	5	6	7	8

Avec `maliste = [7, 2, 4, 5, 3, 10, 9, 8, 3]`, que renvoient les instructions suivantes ?

- `maliste[3:5]`
- `maliste[4:]`
- `maliste[:6]`
- `maliste[-1]`

- **Trouver le rang d'un élément.**

- `liste.index(element)` renvoie la première position à laquelle l'élément a été trouvé. Exemple : avec `liste = [12, 30, 5, 9, 5, 21]`, `liste.index(5)` renvoie 2.
- Si on souhaite juste savoir si un élément appartient à une liste, alors l'instruction :

$$\text{element in liste}$$
renvoie `True` ou `False`. Exemple : avec `liste = [12, 30, 5, 9, 5, 21]`, «`9 in liste`» est vrai, alors que «`8 in liste`» est faux.

- **Liste par compréhension.**

On peut définir un ensemble en donnant la liste de tous ses éléments, par exemple $E = \{0, 2, 4, 6, 8, 10\}$. Une autre façon est de dire que les éléments de l'ensemble doivent vérifier une certaine propriété. Par exemple le même ensemble E peut se définir par :

$$E = \{x \in \mathbb{N} \mid x \leq 10 \text{ et } x \text{ est pair}\}.$$

Avec Python il existe un tel moyen de définir des listes. C'est une syntaxe extrêmement puissante et efficace. Voyons des exemples :

- Partons d'une liste, par exemple `maliste = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1]`.
- La commande `liste_doubles = [2*x for x in maliste]` renvoie une liste qui contient les doubles des éléments de la liste `maliste`. C'est donc la liste `[2, 4, 6, 8, ...]`.
- La commande `liste_carres = [x**2 for x in maliste]` renvoie la liste des carrés des

éléments de la liste initiale. C'est donc la liste `[1,4,9,16,...]`.

- La commande `liste_partielle = [x for x in maliste if x > 2]` extrait la liste composée des seuls éléments strictement supérieurs à 2. C'est donc la liste `[3,4,5,6,7,6,5,4,3]`.

• Liste de listes.

Une liste peut contenir d'autres listes, par exemple :

```
maliste = [ ["Harry", "Hermione", "Ron"], [101,103] ]
```

contient deux listes. Nous allons nous intéresser à des listes qui contiennent des listes d'entiers, que nous appellerons des **tableaux**. Par exemple :

```
tableau = [ [2,14,5], [3,5,7], [15,19,4], [8,6,5] ]
```

Alors `tableau[i]` renvoie la sous-liste de rang i , alors que `tableau[i][j]` renvoie l'entier situé au rang j dans la sous-liste de rang i . Par exemple :

- `tableau[0]` renvoie la sous-liste `[2,14,5]`,
- `tableau[1]` renvoie la sous-liste `[3,5,7]`,
- `tableau[0][0]` renvoie l'entier 2,
- `tableau[0][1]` renvoie l'entier 14,
- `tableau[2][1]` renvoie l'entier 19.

Activité 1 (Listes par compréhension).

Objectifs : mettre en pratique les listes par compréhension. Dans cette activité les listes sont des listes d'entiers.

1. Programme une fonction `multiplier(liste,k)` qui multiplie chaque élément de la liste par k . Par exemple `multiplier([1,2,3,4,5],2)` renvoie `[2,4,6,8,10]`.
2. Programme une fonction `puissance(liste,k)` qui élève chaque élément de la liste à la puissance k . Par exemple `puissance([1,2,3,4,5],3)` renvoie `[1,8,27,64,125]`.
3. Programme une fonction `addition(liste1,liste2)` qui additionne terme à terme les éléments de deux listes de même longueur. Par exemple `addition([1,2,3],[4,5,6])` renvoie `[5,7,9]`.
Indication. C'est un exemple de tâche où on n'utilise pas les listes par compréhension !
4. Programme une fonction `non_zero(liste)` qui renvoie la liste de tous les éléments non nuls. Par exemple `non_zero([1,0,2,3,0,4,5,0])` renvoie `[1,2,3,4,5]`.
5. Programme une fonction `pairs(liste)` qui renvoie la liste de tous les éléments pairs. Par exemple `pairs([1,0,2,3,0,4,5,0])` renvoie `[0,2,0,4,0]`.

Activité 2 (Atteindre une somme fixée).

Objectifs : chercher à atteindre le total de 100 dans une liste de nombres.

On considère une liste de n entiers compris entre 1 et 99 (inclus). Par exemple la liste de $n = 20$ entiers :

```
[16,2,85,27,9,45,98,73,12,26,46,25,26,49,18,99,10,86,7,42]
```

qui a été obtenue au hasard par la commande :

```
liste_20 = [randint(1,99) for i in range(20)]
```

On cherche différentes manières de trouver des nombres de la liste dont la somme fait exactement 100.

1. Programme une fonction `somme_deux_consecutifs_100(liste)` qui teste s'il existe deux éléments consécutifs de la liste dont la somme vaut 100. La fonction renvoie « vrai » ou « faux » (mais elle peut aussi afficher les nombres et leur position pour vérification). Pour l'exemple donné la fonction renvoie `False`.

2. Programme une fonction `somme_deux_100(liste)` qui teste s'il existe deux éléments de la liste, situés à des positions différentes, dont la somme vaut 100. Pour l'exemple donné la fonction renvoie `True` et peut afficher les entiers 2 et 98 (aux rangs 1 et 6 de la liste).
3. Programme une fonction `somme_suite_100(liste)` qui teste s'il existe des éléments consécutifs de la liste dont la somme vaut 100. Pour l'exemple donné la fonction renvoie `True` et peut afficher les entiers à suivre 25, 26, 49 (aux rangs 11, 12 et 13).
4. (*Facultatif*) Plus la taille de la liste est grande plus il y a de chances d'obtenir des entiers dont la somme vaut 100. Pour chacune des trois situations précédentes, détermine à partir de quelle taille n de la liste, la probabilité d'obtenir une somme de 100 est plus grande que $1/2$.

Indications. Pour chaque cas, tu obtiens une estimation de cet entier n , en écrivant une fonction `proba(n, N)` qui effectue un grand nombre N de tirages aléatoires de listes à n éléments (avec par exemple $N = 10\,000$). La probabilité est approchée par le nombre de cas favorables (où la fonction renvoie vraie) divisé par le nombre total de cas (ici N).

Activité 3 (Tableau).

Objectifs : travailler avec des listes de listes.

Dans cette activité nous travaillons avec des tableaux carrés de taille $n \times n$ contenant des entiers. L'élément `tableau` est donc une liste de n listes ayant chacune n éléments.

Par exemple (avec $n = 3$) :

```
tableau = [ [1,2,3], [4,5,6], [7,8,9] ]
```

représente le tableau :

```
1 2 3
4 5 6
7 8 9
```

1. Écris une fonction `somme_diagonale(tableau)` qui calcule la somme des éléments situés sur la diagonale principale. La diagonale principale de l'exemple donné est constituée de 1, 5, 9, la somme vaut donc 15.
2. Écris une fonction `somme_anti_diagonale(tableau)` qui calcule la somme des éléments situés sur l'autre diagonale. L'anti-diagonale de l'exemple donné est constituée de 3, 5, 7, la somme vaut encore 15.
3. Écris une fonction `somme_tout(tableau)` qui calcule la somme totale de tous les éléments. Pour l'exemple la somme totale vaut 45.
4. Écris une fonction `affiche_tableau(tableau)` qui affiche proprement à l'écran un tableau. Tu peux utiliser la commande :

```
print('{:>3d}'.format(tableau[i][j]), end="")
```

Explications.

- La commande `print(chaine, end="")` permet d'afficher une chaîne de caractères sans passer à la ligne.
- La commande `'{:>3d}'.format(k)` affiche l'entier k sur trois cases (même s'il n'y a qu'un chiffre à afficher).

Activité 4 (Carrés magiques).

Objectifs : construire des carrés magiques de taille aussi grande que l'on souhaite ! Il faut d'abord avoir fait l'activité précédente.

Un **carré magique** est un tableau carré de taille $n \times n$ qui contient tous les entiers de 1 à n^2 et qui vérifie que : la somme de chaque ligne, la somme de chaque colonne, la somme de la diagonale principale et la somme de l'anti-diagonale ont toutes la même valeur.

Voici un exemple de carré magique de taille 3×3 et un de taille 4×4 .

4	9	2	→ 15
3	5	7	→ 15
8	1	6	→ 15
↓ 15	↓ 15	↓ 15	↘ 15

1	14	15	4	→ 34
7	9	6	12	→ 34
10	8	11	5	→ 34
16	3	2	13	→ 34
↘ 34	↓ 34	↓ 34	↓ 34	↘ 34

Pour un carré magique de taille $n \times n$, la valeur de la somme est :

$$S_n = \frac{n(n^2 + 1)}{2}.$$

- Exemples.** Définis un tableau pour chacun des exemples 3×3 et 4×4 ci-dessus et affiche-les à l'écran (utilise l'activité précédente).
- Être ou ne pas être.** Définis une fonction `est_carre_magique(carre)` qui teste si un tableau donné est (ou pas) un carré magique (utilise l'activité précédente pour les diagonales).
- Carrés aléatoires.** (Facultatif.) Génère de façon aléatoire des carrés contenant les entiers de 1 à n^2 grâce à une fonction `carre_aléatoire(n)`. Vérifie expérimentalement qu'il est rare d'obtenir ainsi un carré magique !

Indications. Pour une liste `maliste`, la commande `shuffle(maliste)` (issue du module `random`) mélange aléatoirement la liste (la liste est modifiée sur place).

Le but des questions restantes est de créer des carrés magiques de grande taille.

- Addition.** Définis une fonction `addition_carre(carre, k)` qui ajoute un entier k à tous les éléments du carré. Avec l'exemple du carré 3×3 , la commande `addition_carre(carre, -1)` soustrait 1 à tous les éléments et renvoie donc un tableau qui s'afficherait ainsi :

```
3 8 1
2 4 6
7 0 5
```

Indications. Pour définir un nouveau carré, commence par le remplir avec des 0 :

```
nouv_carre = [[0 for j in range(n)] for i in range(n)]
```

puis remplis-le avec les bonnes valeurs par des commandes du type :

```
nouv_carre[i][j] = ...
```

- Multiplication.** Définis une fonction `multiplication_carre(carre, k)` qui multiplie par k tous les éléments du carré. Avec l'exemple du carré 3×3 , la commande

`multiplication_carre(carre,2)` multiplie tous les éléments par 2 et renvoie donc un tableau qui s'afficherait ainsi :

8	18	4
6	10	14
16	2	12

6. **Homothétie.** Définis une fonction `homothetie_carre(carre,k)` qui agrandit le carré d'un facteur k comme sur les exemples ci-dessous. Voici un exemple du carré 3×3 avec une homothétie de rapport $k = 3$.

4	9	2
3	5	7
8	1	6

 \longrightarrow

4	4	4	9	9	9	2	2	2
4	4	4	9	9	9	2	2	2
4	4	4	9	9	9	2	2	2
3	3	3	5	5	5	7	7	7
3	3	3	5	5	5	7	7	7
3	3	3	5	5	5	7	7	7
8	8	8	1	1	1	6	6	6
8	8	8	1	1	1	6	6	6
8	8	8	1	1	1	6	6	6

Voici l'exemple d'un carré 4×4 avec une homothétie de rapport $k = 2$.

1	14	15	4
7	9	6	12
10	8	11	5
16	3	2	13

 \longrightarrow

1	1	14	14	15	15	4	4
1	1	14	14	15	15	4	4
7	7	9	9	6	6	12	12
7	7	9	9	6	6	12	12
10	10	8	8	11	11	5	5
10	10	8	8	11	11	5	5
16	16	3	3	2	2	13	13
16	16	3	3	2	2	13	13

7. **Addition de blocs.** Définis une fonction `addition_bloc_carre(grand_carre,petit_carre)` qui ajoute par bloc un petit carré au grand carré comme sur l'exemple ci-dessous. Le petit carré 2×2 à gauche est ajouté au grand carré au centre pour donner le résultat à droite. Pour cette addition le grand carré est décomposé en 9 blocs, il y a en tout 36 additions.

		4	4	9	9	2	2
		4	4	9	9	2	2
1	2	3	3	5	5	7	7
3	4	3	3	5	5	7	7
		8	8	1	1	6	6
		8	8	1	1	6	6

 \longrightarrow

		5	6	10	11	3	4
		7	8	12	13	5	6
		4	5	6	7	8	9
		6	7	8	9	10	11
		9	10	2	3	7	8
		11	12	4	5	9	10

8. **Produits de carrés magiques.** Définis une fonction `produit_carres(carre1,carre2)` qui à partir de deux carrés magiques, calcule un grand carré magique appelé le produit des deux carrés. L'algorithme est le suivant :

Algorithme.

- — Entrées : un carré magique C_1 de taille $n \times n$ et un carré magique C_2 de taille $m \times m$.
— Sortie : un carré magique C de taille $(nm) \times (nm)$.
 - Définis le carré C_{3a} en retirant 1 à tous les éléments de C_2 . (Utilise la commande `addition(carre2,-1)`.)
 - Définis le carré C_{3b} comme l'homothétie du carré C_{3a} de rapport n . (Utilise la commande `homothetie(carre3a,n)`.)
 - Définis le carré C_{3c} en multipliant tous les termes du carré C_{3b} par n^2 . (Utilise la commande `multiplication_carre(carre3b,n**2)`.)
 - Définis le carré C_{3d} en ajoutant par bloc le carré C_1 au carré C_{3c} . (Utilise la commande `addition_bloc_carre(carre3c,carre1)`.)
 - Renvoie le carré C_{3d} .
-
- Implémente cet algorithme.
 - Teste-le sur des exemples, en vérifiant que le carré obtenu est bien un carré magique.
 - Construis un carré magique de taille 36×36 !
 - Vérifie aussi que l'ordre du produit est important ($C_1 \times C_2$ n'est pas le même carré que $C_2 \times C_1$).

On continue notre exploration du monde des 0 et des 1.

Activité 1 (Palindromes).

Objectifs : trouver des palindromes en écriture binaire et en écriture décimale.

En français un palindrome est un mot (ou une phrase) qui se lit dans les deux sens, par exemple « **RADAR** » ou « **ENGAGE LE JEU QUE JE LE GAGNE** ». Dans cette activité, un **palindrome** sera une liste, qui a les mêmes éléments lorsqu'on la parcourt de gauche à droite ou de droite à gauche.

Exemples :

- $[1, 0, 1, 0, 1]$ est un palindrome (avec une écriture binaire),
- $[2, 9, 4, 4, 9, 2]$ est un palindrome (avec une écriture décimale).

1. Programme une fonction `est_palindrome(liste)` qui teste si une liste est un palindrome ou pas.
Indications. Tu peux comparer les éléments en position i et $p - 1 - i$ ou bien utiliser `list(reversed(liste))`.

2. On cherche des entiers n tels que leur écriture binaire soit un palindrome. Par exemple l'écriture binaire de $n = 27$ est le palindrome $[1, 1, 0, 1, 1]$. C'est le dixième entier n ayant cette propriété.
Quel est le millièmème entier $n \geq 0$ dont l'écriture binaire est un palindrome ?

3. Quel est le millièmème entier $n \geq 0$ dont l'écriture décimale est un palindrome ?
Par exemple les décimales de $n = 909$ forment le palindrome $[9, 0, 9]$. C'est le centièmème entier n ayant cette propriété.

4. Un entier n est un **bi-palindrome** si son écriture binaire *et* son écriture décimales sont des palindromes. Par exemple $n = 585$ a une écriture décimale qui est un palindrome et son écriture binaire $[1, 0, 0, 1, 0, 0, 1, 0, 0, 1]$ aussi. C'est le dixième entier n ayant cette propriété.
Quel est le vingtièmème entier $n \geq 0$ à être un bi-palindrome ?

Cours 1 (Opérations logiques).

On considère que 0 représente le « faux » et 1 le « vrai ».

- Avec l'opération logique « OU », le résultat est vrai dès que l'un au moins des deux termes est vrai.
Cela s'écrit :
 - $0 \text{ OU } 0 = 0$
 - $0 \text{ OU } 1 = 1$
 - $1 \text{ OU } 0 = 1$
 - $1 \text{ OU } 1 = 1$

- Avec l'opération logique « ET », le résultat est vrai uniquement lorsque les deux termes sont vrais. Cela s'écrit :
 - $0 \text{ ET } 0 = 0$
 - $0 \text{ ET } 1 = 0$
 - $1 \text{ ET } 0 = 0$
 - $1 \text{ ET } 1 = 1$
- L'opération logique « NON », échange vrai et faux :
 - $\text{NON } 0 = 1$
 - $\text{NON } 1 = 0$
- Pour des nombres en écriture binaire, ces opérations s'étendent *bits à bits*, c'est-à-dire chiffre par chiffre (en commençant par les chiffres de droite) comme on poserait une addition (sans retenue). Par exemple :

	1.0.1.1.0		1.0.0.1.0
ET	1.1.0.1.0	OU	0.0.1.1.0
	1.0.0.1.0		1.0.1.1.0

Si les deux écritures n'ont pas le même nombre de *bits*, on rajoute des 0 non significatifs à gauche (exemple de 1.0.0.1.0 OU 1.1.0 sur la figure de droite).

Activité 2 (Opérations logiques).

Objectifs : programmer les principales opérations logiques.

- (a) Programme une fonction `NON()` qui correspond à la négation pour une liste donnée. Par exemple `NON([1,1,0,1])` renvoie `[0,0,1,0]`.
- (b) Programme une fonction `OUeg()` qui correspond au « OU » avec en entrée deux listes qui ont la même longueur. Par exemple, avec `liste1 = [1,0,1,0,1,0,1]` et `liste2 = [1,0,0,1,0,0,1]`, la fonction renvoie `[1,0,1,1,1,0,1]`.
- (c) Même travail avec `ETeg()` pour deux listes de longueurs égales.
- Écris une fonction `ajouter_zeros(liste,p)` qui rajoute des zéros au début de la liste afin d'obtenir une liste de longueur `p`. Exemple : si `liste = [1,0,1,1]` et `p = 8`, alors la fonction renvoie `[0,0,0,0,1,0,1,1]`.
- Écris deux fonctions `OU()` et `ET()` qui correspondent aux opérations logiques « OU » et « ET », mais avec deux listes qui n'ont pas nécessairement la même longueur.

Exemple :

- `liste1 = [1,1,1,0,1]` et `liste2 = [1,1,0]`,
- il faut considérer que `liste2` est équivalente à la liste `liste2bis = [0,0,1,1,0]` de même longueur que `liste1`,
- donc `OU(liste1,liste2)` renvoie `[1,1,1,1,1]`,
- puis `ET(liste1,liste2)` renvoie `[0,0,1,0,0]` (ou bien `[1,0,0]` selon ton choix).

Indications : tu peux reprendre le contenu de tes fonctions `OUeg` et `ETeg`, ou bien tu peux d'abord ajouter des zéros à la liste la plus courte.

Activité 3 (Lois de Morgan).

Objectifs : générer toutes les listes possibles de 0 et 1 afin de vérifier une proposition.

1. Première méthode : utiliser l'écriture binaire.

On souhaite générer toutes les listes possibles de 0 et de 1 d'une taille p donnée. Voici comment faire :

- Un entier n parcourt tous les entiers de 0 à $2^p - 1$.
- Pour chacun de ces entiers n , on calcule son écriture binaire (sous la forme d'une liste).
- On rajoute (si besoin) des 0 en début de liste, afin d'obtenir une liste de longueur p .

Exemple : avec $n = 36$, son écriture binaire est $[1, 0, 0, 1, 0, 0]$. Si on veut une liste de $p = 8$ bits, on rajoute deux 0 : $[0, 0, 1, 0, 0, 1, 0, 0]$.

2. Seconde méthode (optionnelle) : un algorithme récursif.

On souhaite de nouveau générer toutes les listes possibles de 0 et de 1 d'une taille donnée. On adopte la procédure suivante : si on sait trouver toutes les listes de taille $p - 1$, alors pour obtenir toutes les listes de taille p , il suffit de rajouter un 0 en début de chacune des listes de taille $p - 1$, puis de recommencer en rajoutant un 1 en début de chacune des listes de taille $p - 1$.

Par exemple : il y a 4 listes de longueur 2 : $[0, 0]$, $[0, 1]$, $[1, 0]$, $[1, 1]$. J'en déduis les 8 listes de longueur 3 :

- 4 listes en rajoutant un 0 devant : $[0, 0, 0]$, $[0, 0, 1]$, $[0, 1, 0]$, $[0, 1, 1]$,
- 4 listes en rajoutant un 1 devant : $[1, 0, 0]$, $[1, 0, 1]$, $[1, 1, 0]$, $[1, 1, 1]$.

Cela donne l'algorithme suivant, qui est un algorithme récursif (car la fonction s'appelle elle-même).

Algorithme.

Usage : `tous_les_binaires(p)`

Entrée : un entier $p > 0$

Sortie : la liste de toutes les listes possibles de 0 et de 1 de longueur p

- Si $p = 1$ renvoyer la liste $[[0], [1]]$.
- Si $p \geq 2$, alors :
 - obtenir toutes les listes de taille $p-1$ par l'appel `tous_les_binaires(p-1)`
 - pour chaque élément de cette liste, construire deux nouveaux éléments :
 - d'une part ajouter 0 en début de cet élément ;
 - d'autre part ajouter 1 en début de cet élément ;
 - ajouter ensuite ces deux éléments à la liste des listes de taille p .
- Renvoyer la liste des listes de taille p .

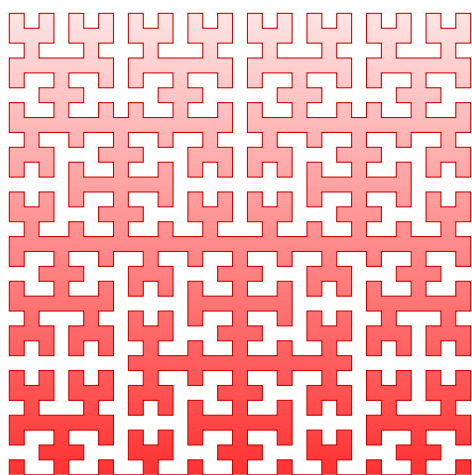
3. Les lois de Morgan.

Les lois de Morgan affirment que pour des booléens (vrai/faux) ou des bits (1/0), on a toujours les égalités :

$$\text{NON}(b_1 \text{ OU } b_2) = \text{NON}(b_1) \text{ ET } \text{NON}(b_2), \quad \text{NON}(b_1 \text{ ET } b_2) = \text{NON}(b_1) \text{ OU } \text{NON}(b_2).$$

Vérifie expérimentalement que ces égalités sont encore vraies pour n'importe quelles listes ℓ_1 et ℓ_2 d'exactly 8 bits.

QUATRIÈME PARTIE



PROJETS

Probabilités – Paradoxe de Parrondo

Chapitre 14

Tu vas programmer deux jeux simples. Lorsque tu joues à ces jeux, tu as plus de chances de perdre que de gagner. Pourtant lorsque tu joues aux deux jeux en même temps, tu as plus de chances de gagner que de perdre ! C'est une situation paradoxale.

Cours 1 (Hasard – Gain – Espérance).

Un joueur joue à un jeu de hasard : il lance une pièce de monnaie ; en fonction du résultat il gagne ou perd de l'argent. On ne peut pas prévoir combien le joueur va gagner ou perdre à coup sûr, mais on va introduire une valeur qui estime combien le joueur peut espérer gagner en moyenne s'il joue de nombreuses fois.

- Au départ le total de gain du joueur est nul : $g = 0$.
- À chaque tirage, il lance une pièce de monnaie. S'il gagne, il obtient un euro, s'il perd il doit un euro.
- Dans les jeux que l'on étudie, la pièce n'est pas équilibrée (elle est un peu truquée). Le joueur n'a pas autant de chances de gagner que de perdre.
- On répète des tirages un certain nombre N de fois. Au bout de N tirages, on totalise le gain du joueur (qui peut être positif ou négatif).
- L'**espérance**, c'est la somme que peut espérer gagner le joueur à chaque lancer. On estime l'espérance par la moyenne des gains d'un grand nombre de tirages. Autrement dit, on a la formule :

$$\text{espérance} \simeq \frac{\text{gain après } N \text{ tirages}}{N} \quad \text{avec } N \text{ grand.}$$

Pour nos jeux, l'espérance sera un nombre réel entre -1 et $+1$.

- Exemples :
 - Si la pièce est bien équilibrée (50 chances sur 100 de gagner), alors pour un grand nombre N de tirages, le joueur va gagner à peu près autant de fois qu'il va perdre ; ses gains seront proches de 0 et donc l'espérance sera proche de $\frac{0}{N} = 0$. En moyenne il gagne 0 euro par tirage.
 - Si la pièce est truquée et que le joueur gagne tout le temps, alors au bout de N tirages, il a empoché N euros. L'espérance est donc $\frac{N}{N} = 1$.
 - Si la pièce est truquée afin que le joueur perde tout le temps, alors au bout de N tirages, son gain est de $-N$ euros. L'espérance est donc $\frac{-N}{N} = -1$.
 - Une espérance de -0.5 signifie qu'en moyenne le joueur perd 0.5 euro par tirage. C'est possible avec une pièce déséquilibrée qui fait gagner dans un cas sur quatre seulement. (Vérifie le calcul !) Si le joueur joue 1000 fois, on peut estimer qu'il va perdre 500 euros ($-0.5 \times 1000 = -500$).

Activité 1 (Jeu A : premier jeu perdant).

Objectifs : modéliser un premier jeu simple, qui en moyenne est perdant pour le joueur.

Jeu A. Dans ce premier jeu, on lance une pièce de monnaie légèrement déséquilibrée : le joueur gagne

un euro dans 49 cas sur 100 ; il perd un euro dans 51 cas sur 100.

1. **Tirage.** Écris une fonction `tirage_jeu_A()` qui ne dépend d'aucun argument et qui modélise un tirage du jeu A. Pour cela :
 - Tire un nombre au hasard $0 \leq x < 1$ à l'aide de la fonction `random()` du module `random`.
 - Renvoie +1 si x est plus petit que 0.49 ; et -1 sinon.
2. **Gain.** Écris une fonction `gain_jeu_A(N)` qui modélise N tirages du jeu A et renvoie le gain total de ces tirages. Bien sûr, le résultat dépend des tirages, il peut varier d'une fois sur l'autre.
3. **Espérance.** Écris une fonction `esperance_jeu_A(N)` qui renvoie une estimation de l'espérance du jeu A selon la formule :

$$\text{espérance} \simeq \frac{\text{gain après } N \text{ tirages}}{N}, \quad \text{avec } N \text{ grand.}$$

4. **Conclusion.**

- (a) Estime l'espérance en effectuant au moins un million de tirages.
- (b) Que signifie le fait que l'espérance soit négative ?
- (c) Dédus de la valeur de l'espérance, le gain (ou la perte) que je peux espérer en jouant 1000 fois au jeu A.

Activité 2 (Jeu B : second jeu perdant).

Objectifs : modéliser un second jeu un peu plus compliqué et qui, en moyenne, est encore perdant pour le joueur.

Jeu B. Le second jeu est un peu plus compliqué. Au début, le joueur part avec un gain nul : $g = 0$. Puis en fonction du gain, il joue à un des deux sous-jeux suivants :

- **Sous-jeu B1.** Si le gain g est un multiple de 3, alors il lance un pièce très désavantageuse : le joueur gagne un euro dans seulement 9 cas sur 100 (il perd donc un euro dans 91 cas sur 100).
- **Sous-jeu B2.** Si le gain g n'est pas un multiple de 3, alors il lance un pièce avantageuse : le joueur gagne un euro dans 74 cas sur 100 (il perd donc un euro dans 26 cas sur 100).

1. **Tirage.** Écris une fonction `tirage_jeu_B(g)` qui dépend du gain déjà acquis et modélise un tirage du jeu B. Tu peux utiliser le test `g%3 == 0` pour savoir si g est multiple de 3.
2. **Gain.** Écris une fonction `gain_jeu_B(N)` qui modélise N tirages du jeu B (en partant d'un gain initial nul) et renvoie le gain total de ces tirages.
3. **Espérance.** Écris une fonction `esperance_jeu_B(N)` qui renvoie une estimation de l'espérance du jeu B.
4. **Conclusion.**
 - (a) Estime l'espérance en effectuant au moins un million de tirages.
 - (b) Combien puis-je espérer gagner ou perdre en jouant 1000 fois au jeu B ?

Activité 3 (Jeux A et B : un jeu gagnant !).

Objectifs : inventer un nouveau jeu en jouant à chaque tour au jeu A ou au jeu B ; bizarrement ce jeu est en moyenne gagnant ! C'est le paradoxe de Parrondo.

Jeux AB. Dans ce troisième jeu, on joue à chaque tour ou bien au jeu A ou bien au jeu B (le choix est fait au hasard). Au début le joueur part avec un gain nul : $g = 0$. À chaque étape, il choisit au hasard (50% de chance chacun) :

- de jouer une fois au jeu A,
 - ou de jouer une fois au jeu B ; plus précisément avec le sous-jeu B1 ou le sous-jeu B2 en fonction du gain déjà acquis g .
1. **Tirage.** Écris une fonction $\text{tirage_jeu_AB}(g)$ qui dépend du gain déjà acquis et modélise un tirage du jeu AB.
 2. **Gain.** Écris une fonction $\text{gain_jeu_AB}(N)$ qui modélise N tirages du jeu AB (en partant d'un gain initial nul) et renvoie le gain total de ces tirages.
 3. **Espérance.** Écris une fonction $\text{esperance_jeu_AB}(N)$ qui renvoie une estimation de l'espérance du jeu AB.
 4. **Conclusion.**
 - (a) Estime l'espérance en effectuant au moins un million de tours de jeu.
 - (b) Que dire cette fois du signe de l'espérance ?
 - (c) Combien puis-je espérer gagner ou perdre en jouant 1000 fois au jeu AB ? Surprenant, non ?

Référence : « Paradoxe de Parrondo », Hélène Davaux, La gazette des mathématiciens, juillet 2017.

Chercher et remplacer

Chercher et remplacer sont deux tâches très fréquentes. Savoir les utiliser et comprendre comment elles fonctionnent te permettra d'être plus efficace.

Activité 1 (Chercher).

Objectifs : apprendre différentes façons de chercher avec Python.

1. L'opérateur « in ».

La façon la plus simple de savoir si une sous-chaîne est présente dans une chaîne de caractères est d'utiliser l'opérateur « in ». Par exemple, l'expression :

```
"PAS" in "ETRE OU NE PAS ETRE"
```

vaut « vrai » car la sous-chaîne **PAS** est bien présente dans la phrase **ETRE OU NE PAS ETRE**.

Déduis-en une fonction `chercher_in(chaine,sous_chaine)` qui renvoie « vrai » ou « faux », selon que la sous-chaîne est (ou non) présente dans la chaîne.

2. La méthode `find()`.

La méthode `find()` s'utilise sous la forme `chaine.find(sous_chaine)` et renvoie la position à laquelle la sous-chaîne a été trouvée.

Teste ceci sur l'exemple précédent. Que renvoie la fonction si la sous-chaîne n'est pas trouvée ?

3. La méthode `index()`.

La méthode `index()` a la même utilité, elle s'utilise sous la forme `chaine.index(sous_chaine)` et renvoie la position à laquelle la sous-chaîne a été trouvée.

Teste ceci sur l'exemple précédent. Que renvoie la fonction si la sous-chaîne n'est pas trouvée ?

4. Ta fonction `chercher()`.

Écris ta propre fonction `chercher(chaine,sous_chaine)` qui renvoie la position de départ de la sous-chaîne si elle est trouvée (et renvoie `None` si elle ne l'est pas).

Tu n'as pas le droit d'utiliser les fonctions Python, tu as seulement droit de tester si deux caractères sont égaux !

Activité 2 (Remplacer).

Objectifs : remplacer des portions de texte par d'autres.

1. La méthode `replace()` s'utilise sous la forme :

```
chaine.replace(sous_chaine,nouv_sous_chaine)
```

Chaque fois que la séquence `sous_chaine` est trouvée dans `chaine`, elle est remplacée par `nouv_sous_chaine`.

Transforme la phrase **ETRE OU NE PAS ETRE** en **ETRE OU NE PLUS ETRE**, puis en **AVOIR OU NE PLUS AVOIR**.

2. Écris ta propre fonction `remplacer()` que tu appelleras sous la forme suivante :

```
remplacer(chaine,sous_chaine,nouv_sous_chaine)
```

et qui remplace seulement la première occurrence de `sous_chaine` trouvée. Par exemple `remplacer("ABBA","B","XY")` renvoie "AXYBA".

Indication. Tu peux utiliser ta fonction `chercher()` de l'activité précédente pour trouver la position de départ de la séquence à remplacer.

3. Améliore ta fonction pour construire une fonction `remplacer_tout()` qui remplace cette fois toutes les occurrences rencontrées.

Cours 1 (Expressions rationnelles *regex*).

Les **expressions rationnelles** permettent de chercher des sous-chaînes avec une plus grande liberté : par exemple on autorise un caractère joker ou bien plusieurs choix possibles pour un caractère. Il existe plein d'autres possibilités, mais nous étudions seulement ces deux-là.

1. On s'autorise une lettre joker symbolisée par un point « . ». Par exemple si on cherche l'expression « **P . R** » alors :
 - **PAR, EMPIRE, PURE, APPORTE** contiennent ce groupe (par exemple pour **PAR** le point joue le rôle de A),
 - mais pas les mots **CAR, PEUR, RAP**.
2. On cherche toujours des groupes de lettres, on s'autorise maintenant plusieurs options. Par exemple « **[CT]** » signifie « **C** ou **T** ». Ainsi le groupe de lettres « **[CT]O** » correspond aux groupes de lettres « **CO** » ou « **TO** ». Ce groupe est donc contenu dans **TOTEM, COTE, TOCARD** mais pas dans **VOTER**. De même « **[ABC]** » désignerait « **A** ou **B** ou **C** ».

Nous utiliserons les expressions rationnelles à travers la commande :

```
python_regex_chercher(chaine,exp)
```

```
from re import *
```

```
def python_regex_chercher(chaine,exp):
    trouve = search(exp,chaine)
    if trouve:
        return trouve.group(), trouve.start(), trouve.end()
    else:
        return None
```

Programme-la et teste-la. Elle renvoie : (1) la sous-chaîne trouvée, (2) la position de début et (3) la position de fin.

python : re.search() - python_regex_chercher()

Usage : `search(exp, chaine)`

ou `python_regex_chercher(chaine, exp)`

Entrée : une chaîne de caractères `chaine` et une expression rationnelle `exp`

Sortie : le résultat de la recherche (la sous-chaîne trouvée, sa position de début, celle de fin)

Exemple avec `chaine = "ETRE OU NE PAS ETRE"`

- avec `exp = "P.S"`, alors `python_regex_chercher(chaine, exp)` renvoie ('PAS', 11, 14).
- avec `exp = "E..E"`, la fonction renvoie ('ETRE', 0, 4).
- avec `exp = "[OT]U"`, la fonction renvoie ('OU', 5, 7).
- avec `exp = "[MN]..P[AI]S"`, la fonction renvoie ('NE PAS', 8, 14).

Activité 3 (Expressions rationnelles *regex*).

Objectifs : programmer la recherche d'expressions rationnelles simples.

1. Programme ta fonction `regex_chercher_joker(chaine, exp)` qui cherche une sous-chaîne qui peut contenir un ou plusieurs jokers « . ». La fonction doit renvoyer : (1) la sous-chaîne trouvée, (2) la position de début et (3) la position de fin (comme pour la fonction `python_regex_chercher()` ci-dessus).
2. Programme ta fonction `regex_chercher_choix(chaine, exp)` qui cherche une sous-chaîne qui peut contenir un ou plusieurs choix contenus dans des balises « [] ». La fonction doit de nouveau renvoyer : (1) la sous-chaîne trouvée, (2) la position de début et (3) la position de fin.

Indication. Tu peux commencer par écrire une fonction `genere_choix(exp)` qui génère toutes les chaînes possibles à partir de `exp`. Par exemple si `exp = "[AB]X[CD]Y"` alors `genere_choix(exp)` renvoie la liste formée de : "AXCY", "BXCY", "AXDY" et "BXDY".

Cours 2 (Remplacer des 0 et des 1 et recommencer!).

On considère une « phrase » composée de seulement deux lettres possibles 0 et 1. Dans cette phrase nous allons chercher un motif (une sous-chaîne) et le remplacer par un autre.

Exemple.

Appliquer la transformation $01 \rightarrow 10$ à la phrase **10110**.

On lit la phrase de gauche à droite, on trouve le premier motif **01** à partir de la seconde lettre, on le remplace par **10** :

$$1(01)10 \mapsto 1(10)10$$

On peut recommencer à partir du début de la phrase obtenue, avec toujours la même transformation $01 \rightarrow 10$:

$$11(01)0 \mapsto 11(10)0$$

Le motif **01** n'apparaît plus dans la phrase **11100** donc la transformation $01 \rightarrow 10$ laisse maintenant cette phrase inchangée.

Résumons : voici l'effet de la transformation itérée $01 \rightarrow 10$ à la phrase **10110** :

$$10110 \mapsto 11010 \mapsto 11100$$

Exemple.

Appliquer la transformation $001 \rightarrow 1100$ à la phrase **0011**.

Une première fois :

$$(001)1 \mapsto (1100)1$$

Une seconde fois :

$$11(001) \mapsto 11(1100)$$

Et ensuite la transformation ne modifie plus la phrase.

Exemple.

Voyons un dernier exemple avec la transformation $01 \rightarrow 1100$ pour la phrase de départ **0001** :

$$0001 \mapsto 001100 \mapsto 01100100 \mapsto 1100100100 \mapsto \dots$$

On peut itérer la transformation, pour obtenir des phrases de plus en plus longues.

Activité 4 (Itérations de remplacements).

Objectifs : étudier quelques transformations et leurs itérations.

On considère ici uniquement des transformations du type $0^a 1^b \rightarrow 1^c 0^d$, c'est-à-dire un motif avec d'abord des **0** puis des **1** est remplacé par un motif avec d'abord des **1** puis des **0**.

1. Une itération.

En utilisant ta fonction `remplacer()` de l'activité 1, vérifie les exemples précédents. Vérifie bien que tu ne remplaces qu'un motif à chaque étape (celui le plus à gauche).

Exemple : la transformation $01 \rightarrow 10$ appliquée à la phrase **101**, se calcule par `remplacer("101", "01", "10")` et renvoie "110".

2. Plusieurs itérations.

Programme une fonction `iterations(phrase, motif, nouv_motif)` qui, à partir d'une phrase, itère la transformation. Une fois que la phrase est stabilisée, la fonction renvoie le nombre d'itérations effectuées ainsi que la phrase obtenue. Si le nombre d'itérations n'a pas l'air de s'arrêter (par exemple quand il dépasse 1000) alors renvoie None.

Exemple. Pour la transformation $0011 \rightarrow 1100$ et la phrase **000011011**, les phrases obtenues sont :

000011011 $\xrightarrow{1}$ 001100011 $\xrightarrow{2}$ 110000011 $\xrightarrow{3}$ 110001100 $\xrightarrow{4}$ 110110000 $\xrightarrow{\text{idem}}$

Pour cet exemple l'appel à la fonction `iterations()` renvoie alors 4 (le nombre de transformations avant stabilisation) et "110110000" (la phrase stabilisée).

3. Le plus d'itérations possibles.

Programme une fonction `iteration_maximale(p,motif,nouv_motif)` qui, parmi toutes les phrases de longueur p , cherche l'une de celles qui met le plus de temps à se stabiliser. Cette fonction renvoie :

- le nombre maximum d'itérations,
- la première phrase qui réalise ce maximum,
- la phrase stabilisée correspondante.

Exemple : pour la transformation $01 \rightarrow 100$, parmi toutes les phrases de longueur $p = 4$, le maximum d'itérations possibles est 7. Un tel exemple de phrase est 0111, qui va se stabiliser (après 7 itérations donc) en 1110000000. Ainsi la commande `iteration_maximale(4,"01","100")` renvoie :

7, '0111', '1110000000'

Indication. Pour générer toutes les phrases de longueur p formées de 0 et 1, tu peux consulter la fiche « Binaire II » (activité 3).

4. Catégories de transformations.

- **Transformation linéaire.** Vérifie expérimentalement que la transformation $0011 \rightarrow 110$ est *linéaire*, c'est-à-dire que pour toutes les phrases de longueur p , il y aura au plus de l'ordre de p itérations au maximum. Par exemple pour $p = 10$, quel est le nombre maximum d'itérations ?
- **Transformation quadratique.** Vérifie expérimentalement que la transformation $01 \rightarrow 10$ est *quadratique*, c'est-à-dire que pour toutes les phrases de longueur p , il y aura au plus de l'ordre de p^2 itérations au maximum. Par exemple pour $p = 10$, quel est le nombre maximum d'itérations ?
- **Transformation exponentielle.** Vérifie expérimentalement que la transformation $01 \rightarrow 110$ est *exponentielle*, c'est-à-dire que pour toutes les phrases de longueur p , il y aura un nombre fini d'itérations, mais que ce nombre peut être très grand (beaucoup plus grand que p^2) avant stabilisation. Par exemple pour $p = 10$, quel est le nombre maximum d'itérations ?
- **Transformation sans fin.** Vérifie expérimentalement que pour la transformation $01 \rightarrow 1100$, il existe des phrases qui ne vont jamais se stabiliser.

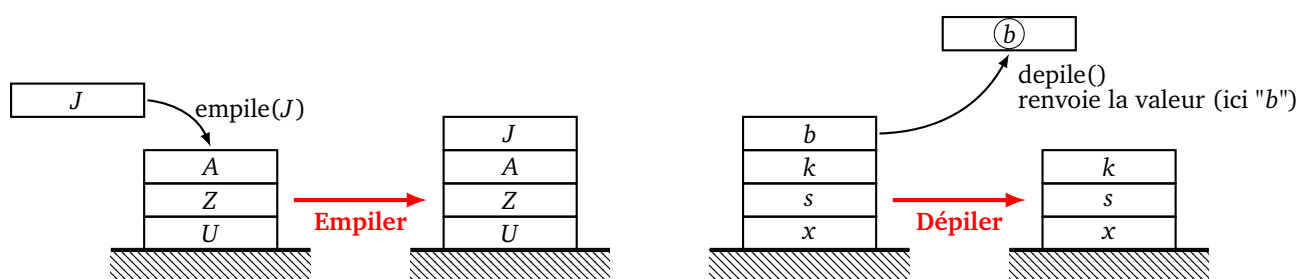
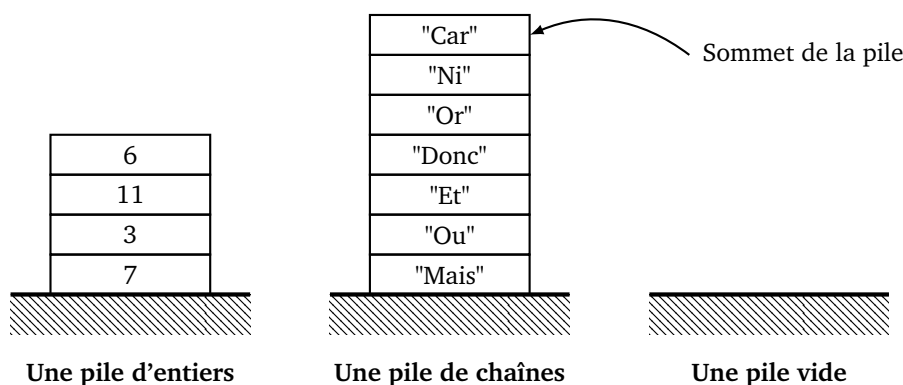
Calculatrice polonaise – Piles

Tu vas programmer ta propre calculatrice ! Pour cela tu vas découvrir une nouvelle notation pour les formules et aussi découvrir ce qu'est une « pile » en informatique.

Cours 1 (Pile).

Une **pile** est une suite de données munie de trois opérations de base :

- **empiler** : on ajoute un élément au sommet de la pile,
- **dépiler** : on lit la valeur de l'élément au sommet de la pile et on retire cet élément de la pile,
- et enfin, on peut tester si la pile est vide.



Remarques.

- **Analogie.** Tu peux faire le lien avec une pile d'assiettes. On peut déposer, une à une, des assiettes sur une pile. On peut retirer, une à une, les assiettes en commençant bien sûr par celle du haut. En plus, il faut considérer que sur chaque assiette est dessinée une donnée (un nombre, un caractère, une chaîne...).
- **Dernier entré, premier sorti.** Dans une file d'attente, le premier qui attend, est le premier qui est servi et ressort. Ici c'est le contraire ! Une pile fonctionne selon le principe « dernier entré, premier sorti ».

- Dans une liste, on peut accéder directement à n'importe quel élément ; dans une pile on n'accède directement qu'à l'élément au sommet de la pile. Pour accéder aux autres éléments, il faut dépiler plusieurs fois.
- L'avantage d'une pile est que c'est une structure de données très simple qui correspond bien à ce qui se passe dans la mémoire d'un ordinateur.

Cours 2 (Variable globale).

Une **variable globale** est une variable qui est définie pour l'ensemble du programme. Il n'est généralement pas recommandé d'utiliser de telles variables mais cela peut être utile dans certains cas. Voyons un exemple. On déclare la variable globale, ici la constante de gravitation, en début de programme comme une variable classique :

```
gravitation = 9.81
```

La contenu de la variable `gravitation` est maintenant accessible partout. Par contre, si on souhaite changer la valeur de cette variable dans une fonction, il faut bien préciser à Python que l'on est conscient de modifier une variable globale !

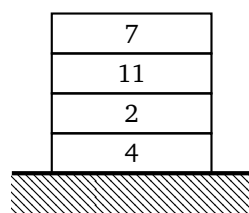
Par exemple pour des calculs sur la Lune, il faut changer la constante de gravitation qui y est beaucoup plus faible.

```
def sur_la_lune():
    global gravitation    # Oui, je veux modifier cette variable globale !
    gravitation = 1.625   # Nouvelle valeur pour tout le programme
    ...
```

Activité 1 (Manipulations de la pile).

Objectifs : définir les trois commandes (très simples) pour utiliser les piles.

Dans cette fiche, une pile sera modélisée par une liste. L'élément en fin de liste correspond au sommet de la pile.



Une pile

```
pile = [4,2,11,7]
```

Modèle sous forme de liste

La pile sera stockée dans une variable globale `pile`. Il faut commencer chaque fonction qui modifie la pile par la commande :

```
global pile
```

1. Écris une fonction `empile()` qui ajoute un élément au sommet de la pile.

empile()

Usage : `empile(element)`

Entrée : un entier, une chaîne...

Sortie : rien

Action : la pile contient un élément en plus

Exemple : si au départ `pile = [5,1,3]` alors, après l'instruction `empile(8)`, la pile vaut `[5,1,3,8]` et si on continue avec l'instruction `empile(6)`, la pile vaut maintenant `[5,1,3,8,6]`.

2. Écris une fonction `depile()`, sans paramètre, qui retire l'élément au sommet de la pile et renvoie sa valeur.

depile()

Usage : `depile()`

Entrée : rien

Sortie : l'élément du sommet de la pile

Action : la pile contient un élément de moins

Exemple : si au départ `pile = [13,4,9]` alors l'instruction `depile()` renvoie la valeur 9 et la pile vaut maintenant `[13,4]` ; si on exécute une nouvelle instruction `depile()`, elle renvoie cette fois la valeur 4 et la pile vaut maintenant `[13]`.

3. Écris une fonction `pile_est_vide()`, sans paramètre, qui teste si la pile est vide ou non.

pile_est_vide()

Usage : `pile_est_vide()`

Entrée : rien

Sortie : vrai ou faux

Action : ne fait rien sur la pile

Exemple :

- si `pile = [13,4,9]` alors l'instruction `pile_est_vide()` renvoie `False`,
- si `pile = []` alors l'instruction `pile_est_vide()` renvoie `True`.

Activité 2 (Opérations sur la pile).

Objectifs : manipuler les piles en utilisant seulement les trois fonctions `empile()`, `depile()` et `pile_est_vide()`.

Dans cet exercice, on travaille avec une pile formée d'entiers. Les questions sont indépendantes.

1. (a) En partant d'une pile vide, arrive à une pile `[5,7,2,4]`.
(b) Exécute ensuite les instructions `depile()`, `empile(8)`, `empile(1)`, `empile(3)`. Que vaut-maintenant la pile ? Que renvoie maintenant l'instruction `depile()` ?

2. Pars d'une pile. Écris une fonction `pile_contient(element)` qui teste si la pile contient un élément donné.
3. Pars d'une pile. Écris une fonction qui calcule la somme des éléments de la pile.
4. Pars d'une pile. Écris une fonction qui renvoie l'avant-dernier élément de la pile (le dernier élément est celui tout en bas ; si cet avant-dernier élément n'existe pas, la fonction renvoie `None`).

Cours 3 (Manipulation de chaînes).

1. La fonction `split()` est une méthode Python qui sépare une chaîne de caractères en morceaux. Si aucun séparateur n'est précisé, le séparateur est le caractère espace.

python : `split()`

Usage : `chaine.split(separateur)`

Entrée : une chaîne de caractères `chaine` et éventuellement un séparateur `separateur`

Sortie : une liste de chaînes de caractères

Exemple :

- `"Etre ou ne pas etre.".split()` renvoie `['Etre', 'ou', 'ne', 'pas', 'etre.']`
- `"12.5;17.5;18".split(";")` renvoie `['12.5', '17.5', '18']`

2. La fonction `join()` est une méthode Python qui recolle une liste de chaînes en une seule chaîne. C'est l'opération inverse de `split()`.

python : `join()`

Usage : `separateur.join(liste)`

Entrée : une liste de chaînes de caractères `liste` et un séparateur `separateur`

Sortie : une chaîne de caractères

Exemple :

- `"".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etreounepasetre.'` Il manque les espaces.
- `" ".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre ou ne pas etre.'` C'est mieux lorsque le séparateur est une espace.
- `--".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre--ou--ne--pas--etre.'`

3. La fonction `isdigit()` est une méthode Python qui teste si une chaîne de caractères ne contient que des chiffres. Cela permet donc de tester si une chaîne correspond à un entier positif. Voici des exemples : `"1789".isdigit()` renvoie `True` ; `"Coucou".isdigit()` renvoie `False`.

Rappelons que l'on peut convertir une chaîne en un entier par la commande `int(chaine)`. Le petit programme suivant teste si une chaîne peut être convertie en un entier positif :

```
machaine = "1789"           # Une chaîne
if machaine.isdigit():
```

```

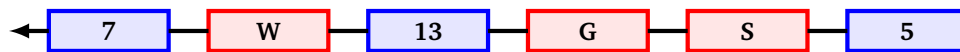
monentier = int(chaine)    # monentier est un entier
else:                      # Problème
    print("Je ne sais pas convertir la chaîne en un entier !")

```

Activité 3 (Gare de triage).

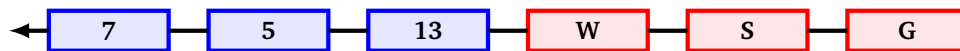
Objectifs : résoudre un problème de triage en modélisant une zone de stockage par la pile.

Un train comporte des wagons bleus qui portent un numéro et des wagons rouges qui portent une lettre.



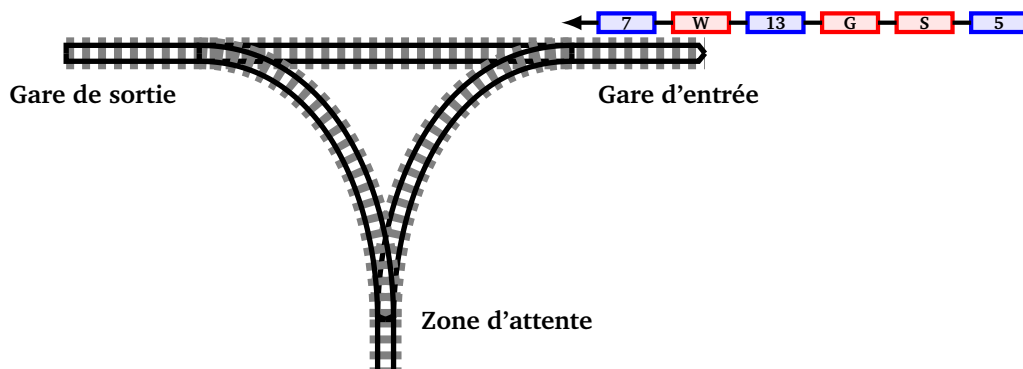
Un train non trié.

Le chef de gare souhaite séparer les wagons : d'abord tous les bleus et ensuite tous les rouges (l'ordre des wagons bleus n'a pas d'importance, l'ordre des wagons rouges non plus).



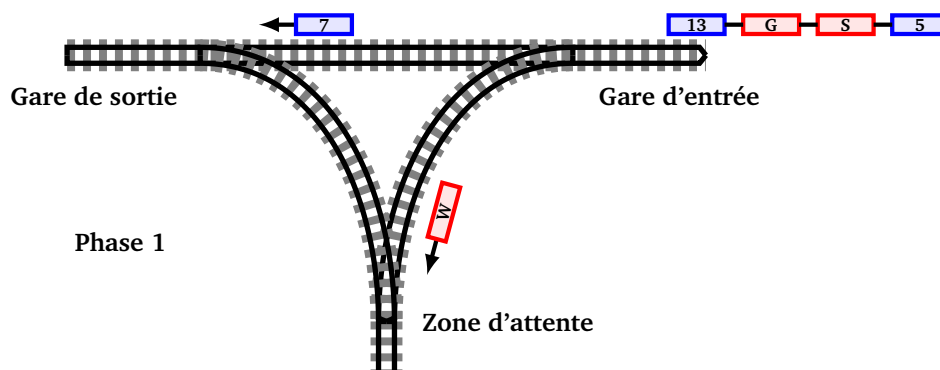
Un train trié.

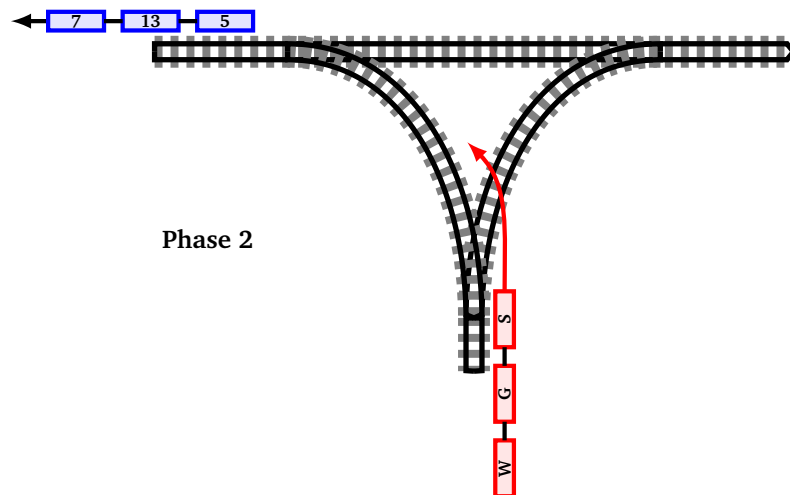
Pour cela, il dispose d'une gare de sortie et d'une zone d'attente : un wagon peut soit être directement envoyé à la gare de sortie, soit être momentanément stocké dans la zone d'attente.



Voici les instructions du chef de gare.

- **Phase 1.** Pour chaque wagon du train :
 - si c'est un wagon bleu, envoyez-le directement en gare de sortie ;
 - si c'est un wagon rouge, envoyez-le dans la zone d'attente.
- **Phase 2.** Ensuite, déplacez un par un les wagons (rouges) de la zone d'attente vers la gare de sortie en les rattachant aux autres.





Voici comment nous allons modéliser le train et son triage.

- Le train est une chaîne de caractères formée d'une suite de nombres (les wagons bleus) et de lettres (les wagons rouges) séparés par des espaces. Par exemple `train = "G 6 Z J 14"`.
- On obtient la liste des wagons par la commande `train.split()`.
- On teste si un wagon est bleu en regardant s'il est marqué d'un nombre, par la fonction `wagon.isdigit()`.
- Le train reconstitué par les wagons triés est aussi une chaîne de caractères. Au départ, c'est la chaîne vide.
- La zone d'attente sera la pile. Au départ la pile est vide. On va y ajouter uniquement les wagons rouges. À la fin, on vide la pile vers la queue du train reconstitué.

En suivant les instructions du chef de gare, écris une fonction `tri_wagons()` qui sépare les wagons bleus et rouges d'un train.

`tri_wagons()`

Usage : `tri_wagons(train)`

Entrée : une chaîne de caractères avec des wagons bleus (nombres) et des wagons rouges (lettres)

Sortie : les wagons bleus d'abord et les rouges ensuite

Action : utilise la pile

Exemple :

- `tri_wagons("A 4 C 12")` renvoie "4 12 C A"
- `tri_wagons("K 8 P 17 L B R 3 10 2 N")` renvoie "8 17 3 10 2 N R B L P K"

Cours 4 (Notation polonaise).

L'écriture en notation polonaise (de son vrai nom, notation polonaise inverse) est une autre façon d'écrire une expression algébrique. Son avantage est que cette notation n'utilise pas de parenthèses et qu'elle est plus facile à manipuler pour un ordinateur. Son inconvénient est que nous n'y sommes pas habitués.

Voici la façon classique d'écrire une expression algébrique (à gauche) et son écriture polonaise (à droite).

Dans tous les cas, le résultat sera 13 !

Classique : $7 + 6$

Polonaise : $7 \ 6 \ +$

Autres exemples :

- classique : $(10 + 5) \times 3$; polonaise : $10 \ 5 \ + \ 3 \ \times$
- classique : $10 + 2 \times 3$; polonaise : $10 \ 2 \ 3 \ \times \ +$
- classique : $(2 + 8) \times (6 + 11)$; polonaise : $2 \ 8 \ + \ 6 \ 11 \ + \ \times$

Voyons comment calculer la valeur d'une expression en écriture polonaise.

- On lit l'expression de gauche à droite :

$2 \ 8 \ + \ 6 \ 11 \ + \ \times$ →

- Lorsque l'on rencontre un premier opérateur (+, ×, ...) on calcule l'opération *avec les deux membres juste avant cet opérateur* :

$\underbrace{2 \ 8 \ +}_{2+8} \ 6 \ 11 \ + \ \times$

- On remplace cette opération par le résultat :

$\underbrace{10}_{\text{résultat de } 2+8} \ 6 \ 11 \ + \ \times$

- On continue la lecture de l'expression (on cherche le premier opérateur et les deux termes juste avant) :

$10 \ \underbrace{6 \ 11 \ +}_{6+11=17} \ \times$ devient $10 \ 17 \ \times$ qui vaut 170

- À la fin il ne reste qu'une valeur, c'est le résultat ! (Ici 170.)

Autres exemples :

- $8 \ 2 \ \div \ 3 \ \times \ 7 \ +$

$\underbrace{8 \ 2 \ \div}_{8 \div 2 = 4} \ 3 \ \times \ 7 \ +$ devient $\underbrace{4 \ 3 \ \times}_{4 \times 3 = 12} \ 7 \ +$ devient $12 \ 7 \ +$ qui vaut 19

- $11 \ 9 \ 4 \ 3 \ + \ - \ \times$

$11 \ 9 \ \underbrace{4 \ 3 \ +}_{4+3=7} \ - \ \times$ devient $11 \ \underbrace{9 \ 7 \ -}_{9-7=2} \ \times$ devient $11 \ 2 \ \times$ qui vaut 22

Exercice. Calcule la valeur des expressions :

- $13 \ 5 \ + \ 3 \ \times$
- $3 \ 5 \ 7 \ \times \ +$
- $3 \ 5 \ 7 \ + \ \times$
- $15 \ 5 \ \div \ 4 \ 12 \ + \ \times$

Activité 4 (Calculatrice polonaise).

Objectifs : programmer une mini-calculatrice qui calcule les expressions en écriture polonaise.

1. Écris une fonction `operation()` qui calcule la somme ou le produit de deux nombres.

operation()

Usage : `operation(a,b,op)`

Entrée : deux nombres a et b , un caractère d'opération "+" ou "*"

Sortie : le résultat de l'opération $a + b$ ou $a * b$

Exemple :

- `operation(2,4,"+")` renvoie 6
- `operation(2,4,"*")` renvoie 8

2. Programme une calculatrice polonaise, selon l'algorithme suivant :

Algorithme.

- Entrée : une expression en écriture polonaise (une chaîne de caractères).
- Sortie : la valeur de cette expression.
- Exemple : "2 3 + 4 *" (le calcul $(2 + 3) \times 4$) donne 20.
- Partir avec une pile vide.
- Pour chaque élément de l'expression (lue de gauche à droite) :
 - si l'élément est un nombre, alors ajouter ce nombre à la pile,
 - si l'élément est une opération, alors :
 - dépiler une fois pour obtenir un nombre b ,
 - dépiler une seconde fois pour obtenir un nombre a ,
 - calculer $a + b$ ou $a \times b$ selon l'opération,
 - ajouter ce résultat à la pile.
- À la fin, la pile ne contient qu'un seul élément, c'est le résultat du calcul.

calculatrice_polonaise()

Usage : `calculatrice_polonaise(expression)`

Entrée : une expression en notation polonaise (chaîne de caractères)

Sortie : le résultat du calcul

Action : utilise une pile

Exemple :

- `calculatrice_polonaise("2 3 4 + +")` renvoie 9
- `calculatrice_polonaise("2 3 + 5 *")` renvoie 25

Bonus. Modifie ton code pour prendre en charge la soustraction et la division !

Activité 5 (Expression bien parenthésée).

Objectifs : déterminer si les parenthèses d'une expression sont placées de façon cohérente.

Voici des exemples d'expressions bien et mal parenthésées :

- $2 + (3 + b) \times (5 + (a - 4))$ est correctement parenthésée ;
- $(a + 8) \times 3 + 4$ est mal parenthésée : il y a une parenthèse fermante «) » seule ;
- $(b + 8/5)) + (4$ est mal parenthésée : il y a autant de parenthèses ouvrantes « (» que de parenthèses fermantes «) » mais elles sont mal positionnées.

1. Voici l'algorithme qui décide si les parenthèses d'une expression sont bien placées. La pile joue le rôle d'une zone de stockage intermédiaire pour les parenthèses ouvrantes " (". Chaque fois que l'on trouve une parenthèse fermante ") " dans l'expression on supprime une parenthèse ouvrante de la pile.

Algorithme.

Entrée : une expression en écriture habituelle (une chaîne de caractères).

Sortie : « vrai » si les parenthèses sont cohérentes, « faux » sinon.

- Partir avec une pile vide.
- Pour chaque caractère de l'expression lue de gauche à droite :
 - si le caractère n'est ni " (", ni ") " alors ne rien faire !
 - si le caractère est une parenthèse ouvrante " (" alors ajouter ce caractère à la pile ;
 - si le caractère est une parenthèse fermante ") " :
 - tester si la pile est vide, si elle vide alors renvoyer « faux » (le programme se termine là, l'expression est mal parenthésée), si la pile n'est pas vide continuer,
 - dépiler une fois, on dépile un " (".
- Si à la fin, la pile est vide alors renvoyer la valeur « vrai », sinon renvoyer « faux ».

parentheses_correctes()

Usage : parentheses_correctes(expression)

Entrée : une expression (chaîne de caractères)

Sortie : vrai ou faux selon que les parenthèses sont correctes ou pas

Action : utilise une pile

Exemple :

- parentheses_correctes("(2+3)*(4+(8/2))") renvoie True
- parentheses_correctes("(x+y)*((7+z)") renvoie False

2. Améliore cette fonction pour tester une expression avec des parenthèses et des crochets. Voici une expression cohérente : $[(a+b)*(a-b)]$, voici des expressions non correctes : $[a+b]$, $(a+b)*[a-b]$. Voici l'algorithme à programmer en une fonction `crochets_parentheses_correctes()`.

Algorithme.

Entrée : une expression en écriture habituelle (une chaîne de caractères).

Sortie : « vrai » si les parenthèses et les crochets sont cohérents, « faux » sinon.

- Partir avec une pile vide.
- Pour chaque caractère de l'expression lue de gauche à droite :
 - si le caractère n'est ni "(", ni ")", ni "[", ni "]" alors ne rien faire ;
 - si le caractère est une parenthèse ou un crochet ouvrant "(" ou "[", alors ajouter ce caractère à la pile ;
 - si le caractère est une parenthèse ou un crochet fermant ")" ou "]" :
 - tester si la pile est vide, si elle est vide alors renvoyer « faux » (le programme se termine là, l'expression n'est pas cohérente), si la pile n'est pas vide continuer,
 - dépiler une fois, on dépile un "(" ou un "[",
 - si le caractère dépilé (ouvrant) ne correspond pas au caractère lu dans l'expression, alors renvoyer « faux ». Le programme se termine là, l'expression n'est pas cohérente ; dire que les caractères correspondent c'est avoir "(" avec ")" et "[" avec "]" .
- Si à la fin, la pile est vide alors renvoyer la valeur « vrai », sinon renvoyer « faux ».

Cette fois la pile peut contenir des parenthèses ouvrantes "(" ou biens des crochets ouvrants "[". Chaque fois que l'on trouve une parenthèse fermante ")" dans l'expression, il faut que le haut de la pile soit une parenthèse ouvrante "(" . Chaque fois que l'on trouve un crochet fermant "]" dans l'expression, il faut que le haut de la pile soit un crochet ouvrant "[".

Activité 6 (Conversion en écriture polonaise).

Objectifs : transformer une expression algébrique classique avec parenthèses en une écriture polonaise. L'algorithme est une version très améliorée de l'activité précédente. Nous ne donnerons pas de justification.

Tu es habitué à l'écriture « $(13+5) \times 7$ » ; tu as vu que l'ordinateur savait facilement calculer « $13\ 5 + 7 \times$ ». Il ne reste plus qu'à passer de l'expression algébrique classique (avec parenthèses) à l'écriture polonaise (sans parenthèses) !

Voici l'algorithme pour des expressions ne comportant que des additions et des multiplications.

Algorithme.

Entrée : une expression en écriture habituelle

Sortie : l'expression écrite en notation polonaise

- Partir avec une pile vide.
- Partir avec une chaîne vide polonaise qui à la fin contiendra le résultat.
- Pour chaque caractère de l'expression (lue de gauche à droite) :
 - si le caractère est un nombre, alors ajouter ce nombre à la chaîne de sortie polonaise ;
 - si le caractère est une parenthèse ouvrante "(", alors ajouter ce caractère à la pile ;
 - si le caractère est l'opérateur de multiplication "*", alors ajouter ce caractère à la pile ;
 - si le caractère est l'opérateur d'addition "+", alors
 - tant que la pile n'est pas vide :
 - dépiler un élément,
 - si cet élément est l'opérateur de multiplication "*", alors :
 - ajouter cet élément à la chaîne de sortie polonaise
 - sinon :
 - empiler cet élément (on le remet sur la pile après l'avoir enlevé)
 - terminer immédiatement la boucle « tant que » (avec break)
 - enfin, ajouter l'opérateur d'addition "+" à la pile.
 - si le caractère est une parenthèse fermante ")", alors
 - tant que la pile n'est pas vide :
 - dépiler un élément,
 - si cet élément est une parenthèse ouvrante "(", alors :
 - terminer immédiatement la boucle « tant que » (avec break)
 - sinon :
 - ajouter cet élément à la chaîne de sortie polonaise
- Si à la fin, la pile n'est pas vide, alors ajouter chaque élément de la pile à la chaîne de sortie polonaise.

écriture_polonaise()

Usage : `écriture_polonaise(expression)`

Entrée : une expression classique (avec les éléments séparés par des espaces)

Sortie : l'expression en notation polonaise

Action : utilise la pile

Exemple :

- `écriture_polonaise("2 + 3")` renvoie "2 3 +"
- `écriture_polonaise("4 * (2 + 3)")` renvoie "4 2 3 + *"
- `écriture_polonaise("(2 + 3) * (4 + 8)")` renvoie "2 3 + 4 8 + *"

Dans cet algorithme, on appelle abusivement « caractère » d'une expression chaque élément entre deux espaces. Exemple : les caractères de "(17 + 10) * 3" sont (, 17, +, 10,), * et 3.

Tu vois que l'addition a un traitement plus compliqué que la multiplication. C'est dû au fait que la multiplication est prioritaire devant l'addition. Par exemple $2+3 \times 5$ signifie $2+(3 \times 5)$ et pas $(2+3) \times 5$. Si

tu souhaites prendre en compte la soustraction et la division, il faut faire attention à la non-commutativité ($a - b$ n'est pas égal à $b - a$, $a \div b$ n'est pas égal à $b \div a$).

Termine cette fiche en vérifiant que tout fonctionne correctement avec différentes expressions. Par exemple :

- Définis une expression `exp = "(17 * (2 + 3)) + (4 + (8 * 5))"`
- Demande à Python de calculer cette expression : `eval(exp)`. Python renvoie 129.
- Convertis l'expression en écriture polonaise : `ecriture_polonaise(exp)` renvoie
`"17 2 3 + * 4 8 5 * + +"`
- Avec ta calculatrice calcule le résultat : `calculatrice_polonaise("17 2 3 + * 4 8 5 * + +")` renvoie 129. On obtient bien le même résultat !

Visualiseur de texte – Markdown

Chapitre 17

Tu vas programmer un traitement de texte tout simple qui affiche proprement des paragraphes et met en évidence les mots en gras et en italiques.

Cours 1 (Texte avec tkinter).

Voici comment afficher du texte avec Python et le module des fenêtres graphiques tkinter.

Du texte avec Python !

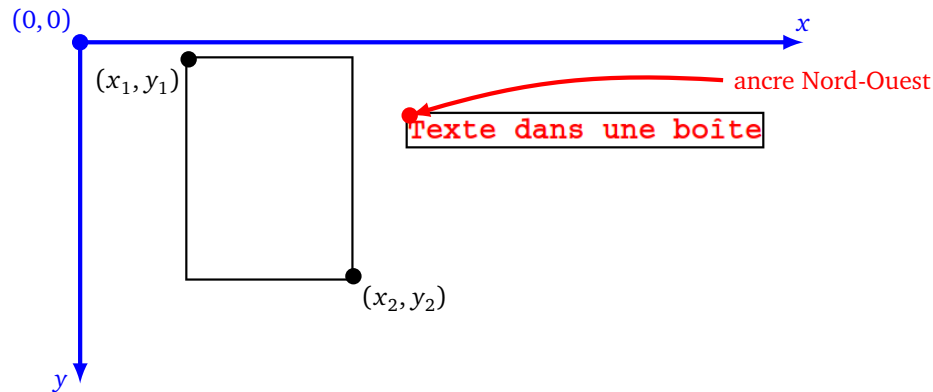
Le code est :

```
from tkinter import *
from tkinter.font import Font
# Fenêtre tkinter
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)
# Fonte
mafonte = Font(family="Times", size=20)
# Le texte
canvas.create_text(100,100, text="Du texte avec Python !",
anchor=NW, font=mafonte, fill="blue")
# Ouverture de la fenêtre
root.mainloop()
```

Quelques explications :

- `root` et `canvas` sont les variables qui définissent une fenêtre graphique (ici de largeur 800 et de hauteur 600 pixels). Cette fenêtre est visualisée par la commande de la fin : `root.mainloop()`.
- On rappelle que pour le repère graphique l'axe des ordonnées est dirigé vers le bas. Pour définir un rectangle, il suffit de préciser les coordonnées (x_1, y_1) et (x_2, y_2) de deux sommets opposés (voir la figure ci-dessous).
- Le texte est affiché par la commande `canvas.create_text()`. Il faut préciser les coordonnées (x, y) du point à partir duquel on souhaite afficher le texte.
- L'option `text` permet de passer la chaîne de caractères à afficher.
- L'option `anchor` permet de préciser le point d'ancrage du texte, `anchor=NW` signifie que la zone de texte est ancrée au point Nord-Ouest (NW) (voir la figure ci-dessous).
- L'option `fill` permet de préciser la couleur du texte.

- L'option `font` permet de définir la fonte (c'est-à-dire le style et la taille des caractères). Voici des exemples de fontes, à toi de les tester :
 - `Font(family="Times", size=20)`
 - `Font(family="Courier", size=16, weight="bold")` en **gras**
 - `Font(family="Helvetica", size=16, slant="italic")` en *italique*



Activité 1 (Afficher un texte avec tkinter).

Objectifs : afficher du texte avec le module graphique `tktinter`.

Du texte avec Python

- (a) Définis une fenêtre `tktinter` de taille 800×600 par exemple.
 - (b) Dessine un rectangle gris (qui sera notre zone de texte) de taille largeur \times hauteur (par exemple 700×500).
 - (c) Définis plusieurs types de fontes : `fonte_titre`, `fonte_sous_titre`, `fonte_gras`, `font_italique`, `fonte_texte`.
 - (d) Affiche des textes avec différentes fontes.
- Écris une fonction `encadre_mot(mot, fonte)` qui dessine un rectangle autour d'un texte. Pour cela utilise la méthode `canvas.bbox(monobjet)` qui renvoie les coordonnées x_1, y_1, x_2, y_2 du rectangle voulu. (Ici `monobjet = canvas.create_text(...)`).
 - Écris une fonction `longueur_mot(mot, fonte)` qui calcule la longueur d'un mot en pixels (c'est la largeur du rectangle de la question précédente).
 - Écris une fonction `choix_fonte(mode, en_gras, en_italique)` qui renvoie le nom d'une fonte adaptée (parmi celles définies en première question) selon un mode (parmi "titre", "sous_titre", "texte") et selon des booléens `en_gras`, `en_italique`.
Par exemple `choix_fonte("texte", True, False)` renvoie la fonte `fonte_gras`.

Cours 2 (Markdown).

Le *Markdown* est un langage de balisage simple qui permet d'écrire un fichier texte propre et éventuellement de le convertir vers un autre format (html, pdf...).

Voici un exemple de fichier texte avec une syntaxe *Markdown* avec juste en dessous son rendu graphique.

```
# L'Origine des Espèces
## par Charles Darwin

Les rapports géologiques qui existent entre la ** faune
actuelle ** et la ** faune éteinte ** de l'Amérique
méridionale, ainsi que certains faits relatifs à la
distribution des êtres organisés qui peuplent ce continent,
m'ont profondément frappé lors de mon voyage à bord du
navire le * Beagle * en qualité de naturaliste.

** Chapitres **

+ De la variation des espèces à l'état domestique.
+ De la variation à l'état de nature.
+ La lutte pour l'existence.
+ La sélection naturelle ou la persistance du plus apte.
+ ...
```

L'Origine des Espèces
par Charles Darwin

Les rapports géologiques qui existent entre la **faune actuelle** et la **faune éteinte** de l'Amérique méridionale, ainsi que certains faits relatifs à la distribution des êtres organisés qui peuplent ce continent, m'ont profondément frappé lors de mon voyage à bord du navire le *Beagle* en qualité de naturaliste.

Chapitres

- De la variation des espèces à l'état domestique.
- De la variation à l'état de nature.
- La lutte pour l'existence.
- La sélection naturelle ou la persistance du plus apte.
- ...

La syntaxe est simple, avec un fichier texte bien lisible. Voici quelques éléments de cette syntaxe :

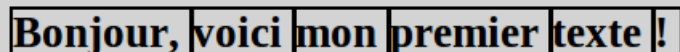
- un **texte en gras** s'obtient en entourant le texte par deux astérisques ** ;
- un *texte en italique* s'obtient en entourant le texte par un astérisque * ;
- la ligne d'un titre commence par dièse # ;
- la ligne d'un sous-titre commence par deux dièses ## ;
- pour les éléments d'une liste, chaque ligne commence par un symbole spécial, pour nous ce sera le symbole « plus » +.
- Il existe aussi une syntaxe pour afficher des liens, des tableaux, du code...

Dans la suite nous utiliserons la syntaxe simplifiée comme elle est décrite ci-dessus.

Activité 2 (Visualiser du Markdown).

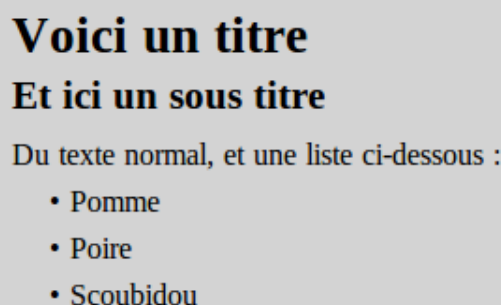
Objectifs : visualiser du texte avec la syntaxe Markdown simplifiée.

1. Écris une fonction `afficher_ligne_v1(par, posy)` qui affiche *un par un* les mots d'un paragraphe `par` (sur la ligne d'ordonnée `posy`).



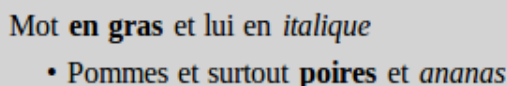
Indications :

- Ces mots sont obtenus grâce à la commande `par.split()`.
 - La ligne affichée commence tout à gauche, elle déborde à droite si elle trop longue.
 - Après chaque mot on place une espace puis le mot suivant.
 - Sur l'image ci-dessus les mots sont encadrés.
2. Améliore ta fonction en `afficher_ligne_v2(par, posy)` pour tenir compte des titres, sous-titres et listes.



Indications :

- Pour savoir dans quel mode il faut afficher la ligne, il suffit de tester les premiers caractères de la ligne. La ligne d'un titre commence par `#`, celle d'un sous-titre par `##`, celle d'une liste par `+`.
 - Pour les listes, tu peux obtenir le caractère « • » par le caractère unicode `u'\u2022'`. Tu peux aussi indenter les éléments de la liste pour plus de lisibilité.
 - Utilise la fonction `choix_fonte()` de la première activité.
 - Sur l'image ci-dessus, chaque ligne est produite par un appel à la fonction. Par exemple `afficher_ligne_v2("## Et ici un sous titre", 100)`
3. Améliore encore ta fonction en `afficher_ligne_v3(par, posy)` pour tenir compte des mots en gras et en italique dans le texte.

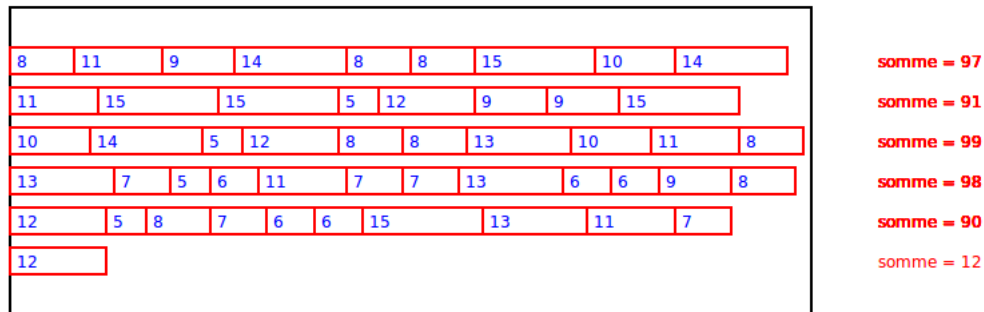


Indications :

- Les mots en gras sont entourés par la balise `**`, les mots en italique par la balise `*`. Dans notre syntaxe simplifiée, les balises sont séparées des mots par des espaces, par exemple : `"Mot ** en gras ** et lui en * italique *"`.
 - Définis une variable booléenne `en_gras` qui est fausse au départ ; chaque fois que tu rencontres la balise `**` alors inverse la valeur de `en_gras` (« vrai » devient « faux », « faux » devient « vrai »). Tu peux utiliser l'opérateur `not`.
 - Utilise encore la fonction `choix_fonte()` de la première activité.
 - Sur l'image ci-dessus, chaque ligne est produite par un appel à la fonction. Par exemple `afficher_ligne_v3("+ Pommes et surtout ** poires ** et * ananas *", 100)`
4. Améliore encore ta fonction en `afficher_paragraphe(par, posy)` qui gère l'affichage d'un paragraphe (c'est-à-dire une chaîne de caractères qui peut être très longue) sur plusieurs lignes.


```
from random import randint
longueurs = [randint(5,15) for i in range(50)]
```

1. Écris une fonction `coupures_simples()` qui calcule les indices permettant de réaliser les coupures correspondant à la figure ci-dessous, c'est-à-dire un alignement à gauche (sans espaces) et sans dépasser la longueur totale de la ligne (ici de longueur 100).



coupures_simples()

Usage : `coupures_simples(long)`

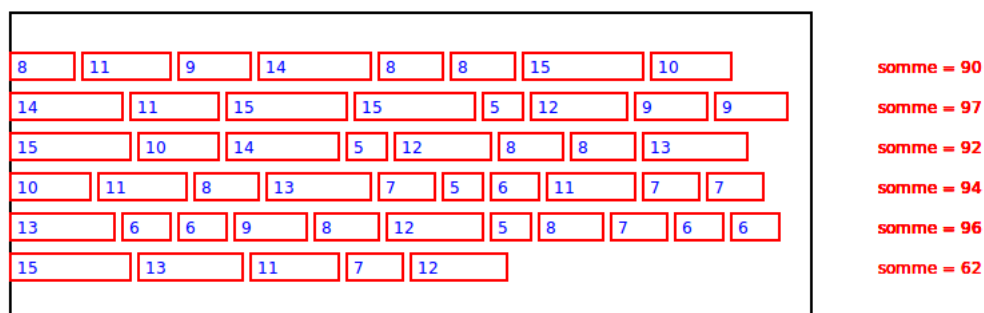
Entrée : une suite de longueurs (une liste d'entiers)

Sortie : la liste des indices où effectuer une coupure

Exemple : `coupures_simples(longueurs)` où `longueurs` est l'exemple donné ci-dessus, renvoie la liste `[0, 9, 17, 27, 39, 49, 50]`. C'est-à-dire que :

- la première ligne correspond aux indices 0 à 8 (donnés par `range(0,9)`),
- la deuxième ligne correspond aux indices 9 à 16 (donnés par `range(9,17)`),
- la troisième ligne correspond aux indices 17 à 26 (donnés par `range(17,27)`),
- ...
- la dernière ligne correspond à l'indice 49 (donné par `range(49,50)`).

2. Modifie ton travail en une fonction `coupures_espaces()` qui rajoute une espace (avec `longueur_espace = 1`) entre deux mots d'une même ligne (mais pas au début de ligne, ni à la fin de la ligne). Cela correspond au dessin suivant :



Pour notre exemple, les coupures renvoyées sont `[0, 8, 16, 24, 34, 45, 50]`.

3. Afin de pouvoir justifier le texte, tu autorises les espaces à être plus grandes que la longueur initiale de 1. Sur chaque ligne, les espaces entre les mots sont toutes de la même longueur (supérieure ou égale 1) de sorte que le dernier mot soit aligné à droite. D'une ligne à l'autre, la longueur des espaces peut changer.

8	11	9	14	8	8	15	10	somme = 100.0
14	11	15	15	5	12	9	9	somme = 100.0
15	10	14	5	12	8	8	13	somme = 100.0
10	11	8	13	7	5	6	11	somme = 100.0
13	6	6	9	8	12	5	8	somme = 100.0
15	13	11	7	12				somme = 62

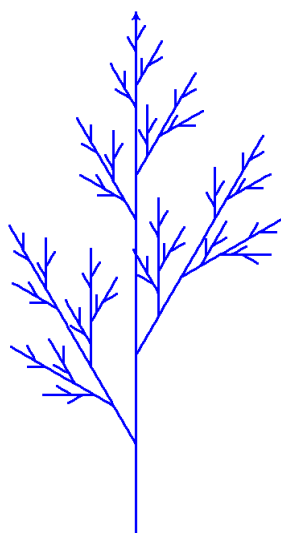
Écris une fonction `calcul_longueur_espaces()` qui renvoie la longueur que doivent avoir les espaces de chaque ligne pour que le texte soit justifié. Pour notre exemple, on obtient la liste `[2.43, 1.43, 2.14, 1.67, 1.40, 1.00]`, c'est-à-dire que pour la première ligne les espaces doivent être de longueur 2.43, pour la seconde 1.43,...

Pour trouver la formule qui convient, il suffit de reprendre les résultats de la fonction `coupures_espaces()` puis, pour chaque ligne, compter le nombre de mots qu'elle contient, ainsi que la valeur qu'il manque pour arriver au total de 100.

Tu as maintenant tout en main pour visualiser du texte écrit avec la syntaxe Markdown et le justifier. Cela représente quand même encore du travail ! Tu peux aussi améliorer la prise en charge de la syntaxe Markdown : prendre en charge le code, les listes numérotées, les sous-listes, les mots en gras et en italique en même temps...

L-système

Les L-systèmes offrent une façon très simple de coder des phénomènes complexes. À partir d'un mot initial et d'opérations de remplacement, on arrive à des mots compliqués. Lorsque l'on « dessine » ces mots, on obtient de superbes figures fractales. Le « L » vient du botaniste A. Lindenmayer qui a inventé les L-systèmes afin de modéliser les plantes.



Cours 1 (L-système).

Un **L-système** est la donnée d'un mot initial et de règles de remplacement. Voici un exemple avec le mot de départ et une seule règle :

$$\mathbf{BgAdB} \quad \mathbf{A} \rightarrow \mathbf{ABA}$$

Le **k-ème itéré** du L-système s'obtient en appliquant k fois la substitution au mot de départ. Avec notre exemple :

- Première itération. Le mot de départ est **BgAdB**, la règle est $\mathbf{A} \rightarrow \mathbf{ABA}$: on remplace le **A** par **ABA**. On obtient le mot **BgABAdB**.
- Deuxième itération. On part du mot obtenu **BgABAdB**, on remplace les deux **A** par **ABA** : on obtient le mot **BgABABABAdB**.
- Le troisième itéré est **BgABABABABABABAdB**, etc.

Lorsqu'il y a deux règles (ou plus) il faut les appliquer en même temps. Voici un exemple de L-système à deux règles :

$$\mathbf{A} \quad \mathbf{A} \rightarrow \mathbf{BgA} \quad \mathbf{B} \rightarrow \mathbf{BB}$$

Avec notre exemple :

- Première itération. Le mot de départ est **A**, on applique la première règle $\mathbf{A} \rightarrow \mathbf{BgA}$ (la seconde règle

ne s'applique pas, car il n'y a pas encore de **B**) : on obtient le mot **BgA**.

- Deuxième itération. On part du mot obtenu **BgA**, on remplace les **A** par **BgA** et en même temps les **B** par **BB** : on obtient le mot **BBBgBgA**.
- Le troisième itéré est **BBBBBgBBBgBgA**, etc.

Cours 2 (Argument optionnel d'une fonction).

Je veux programmer une fonction qui trace un trait d'une longueur donnée, avec la possibilité de changer l'épaisseur du trait et la couleur.

Une méthode serait de définir une fonction par :

```
def tracer(longueur, epaisseur, couleur):
```

Je l'appellerais alors par exemple par :

```
tracer(100, 5, "blue"):
```

Mais comme mes traits auront le plupart du temps l'épaisseur 5 et la couleur bleue, je perds du temps et de la lisibilité en redonnant ces informations à chaque fois.

Avec Python il est possible de donner des arguments optionnels. Voici une meilleure façon de faire en donnant des valeurs par défaut :

```
def tracer(longueur, epaisseur=5, couleur="blue"):
```

- La commande `tracer(100)` trace mon trait, et comme je n'ai précisé que la longueur, les arguments `epaisseur` et `couleur` prennent les valeurs par défaut (5 et bleu).
- La commande `tracer(100, epaisseur=10)` trace mon trait avec une nouvelle épaisseur (la couleur est celle par défaut).
- La commande `tracer(100, couleur="red")` trace mon trait avec une nouvelle couleur (l'épaisseur est celle par défaut).
- La commande `tracer(100, epaisseur=10, couleur="red")` trace mon trait avec une nouvelle épaisseur et une nouvelle couleur.
- Voici aussi ce que tu peux utiliser :
 - `tracer(100, 10, "red")` : ne pas préciser les noms des options si on fait attention à l'ordre.
 - `tracer(couleur="red", epaisseur=10, longueur=100)` : on peut nommer n'importe quelle variable, les variables nommées peuvent passer en argument dans n'importe quel ordre !

Activité 1 (Tracer un mot).

Objectifs : tracer un dessin à partir d'un « mot ». Chaque caractère correspond à une instruction de la tortue.

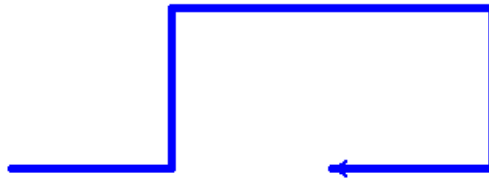
On te donne un mot (par exemple **AgAdAAAdAdA**) dans lequel chaque lettre (lues de gauche à droite) correspond à une instruction pour la tortue Python.

- **A** ou **B** : avance d'une quantité fixée (en traçant),
- **g** : tourne à gauche, sans avancer, d'un angle fixé (le plus souvent 90 degrés),
- **d** : tourne à droite d'un angle fixé.

Les autres caractères ne font rien. (On ajoutera d'autres commandes plus loin).

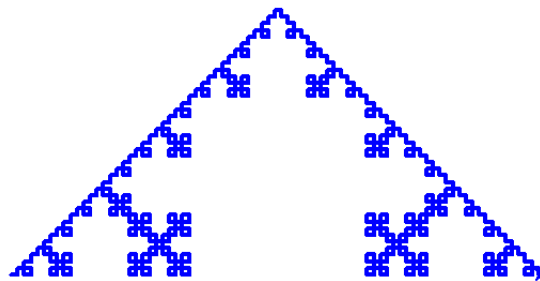
Programme une fonction `trace_lsystem(mot, angle=90, echelle=1)` qui affiche le dessin correspondant aux lettres de mot. Par défaut l'angle est de 90 degrés, et à chaque fois que l'on avance, c'est de $100 \times \text{echelle}$.

Par exemple : `trace_lsystem("AgAdAAAdAdA")` affiche ceci :



Activité 2 (Une seule règle – Flocon de Koch).

Objectifs : tracer le flocon de Koch à partir d'un mot obtenu par itérations.



1. Programme une fonction `remplacer_1(mot, lettre, motif)` qui remplace une lettre par un motif dans un mot.

Par exemple avec `mot = "AdAag"`, `remplacer_1(mot, "A", "Ag")` renvoie le mot `AgdAgAgg` : chaque lettre `A` a été remplacée par le motif `Ag`.

2. Programme une fonction `iterer_1systeme_1(depart, regle, k)` qui calcule le k -ème itéré du L-système associé au mot initial `depart` selon la règle `regle` qui contient le couple formé de la lettre et de son motif de remplacement. Par exemple, avec :

- `depart = "A"`
- `regle = ("A", "AgAdAdAgA")` c'est-à-dire $A \rightarrow AgAdAdAgA$
- pour $k = 0$, la fonction renvoie le mot de départ `A`,
- pour $k = 1$, la fonction renvoie `AgAdAdAgA`,
- pour $k = 2$, la fonction renvoie :

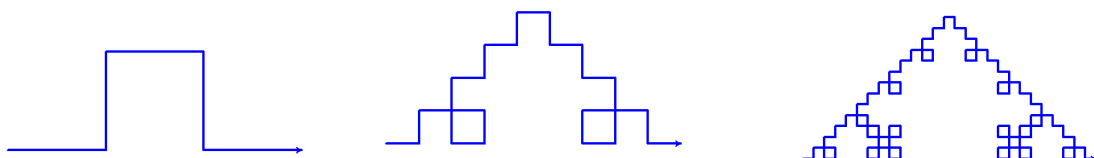
`AgAdAdAgAgAgAdAdAgAdAgAdAdAgAdAgAdAdAgAgAgAdAdAgA`

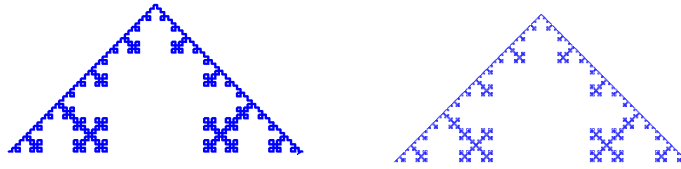
- pour $k = 3$, la fonction renvoie : `AgAdAdAgAgA . . .` un mot de 249 lettres.

3. Trace les premières images du flocon de Koch donné comme ci-dessus par :

départ : `A` règle : $A \rightarrow AgAdAdAgA$

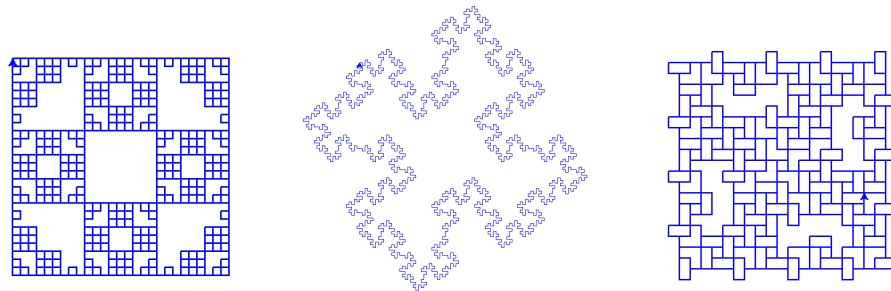
Ici les images pour $k = 1$ jusqu'à $k = 5$. Pour $k = 1$, le mot est `AgAdAdAgA` et tu peux vérifier le tracé sur la première image.





4. Trace d'autres figures fractales à partir des L-systèmes suivants. Pour tous ces exemples le mot de départ est "AdAdAdA" (un carré) et la règle est à choisir parmi :

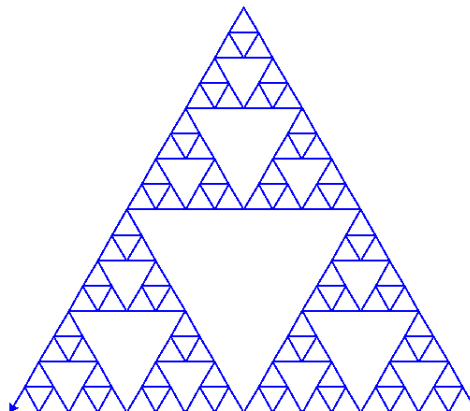
- ("A", " AdAgAgAAAdAdAgA")
- ("A", "AgAAAdAAAdAdAgAgAAAdAdAgAgAAAdAdAgA")
- ("A", "AAAdAdAdAdAA")
- ("A", "AAAdAddAdA")
- ("A", "AAAdAdAdAdAdAgA")
- ("A", "AAAdAgAdAdAA")
- ("A", "AdAAAddAdA")
- ("A", "AdAgAdAdA")



Invente et trace tes propres L-systèmes !

Activité 3 (Deux règles – Triangle de Sierpinski).

Objectifs : calculer des L-systèmes plus compliqués en autorisant cette fois deux règles de remplacement au lieu d'une seule.



1. Programme une fonction `remplacer_2(mot, lettre1, motif1, lettre2, motif2)` qui remplace une première lettre par un motif et une seconde lettre par un autre.

Par exemple avec `mot = "AdBgA"`, `remplacer_2(mot, "A", "ABg", "B", "Bd")` renvoie le mot `ABgdBdgABg` : chaque lettre **A** a été remplacée par le motif **ABg** et en même temps chaque lettre **B** a été remplacée par le **Bd**.

Attention ! Il ne faut pas obtenir `ABgdBdgABdg`. Si c'est le cas c'est que tu as utilisé la fonction `remplacer_1()` pour d'abord remplacer les A, puis une seconde fois pour les B (mais après le premier remplacement de nouveaux B sont apparus). Il faut reprogrammer une nouvelle fonction pour éviter cela.

2. Programme une fonction `iterer_lsystème_2(depart, regle1, regle2, k)` qui calcule le k -ème itéré du L-système associé au mot initial `depart` selon les règles `regle1` et `regle2`. Par exemple, avec :

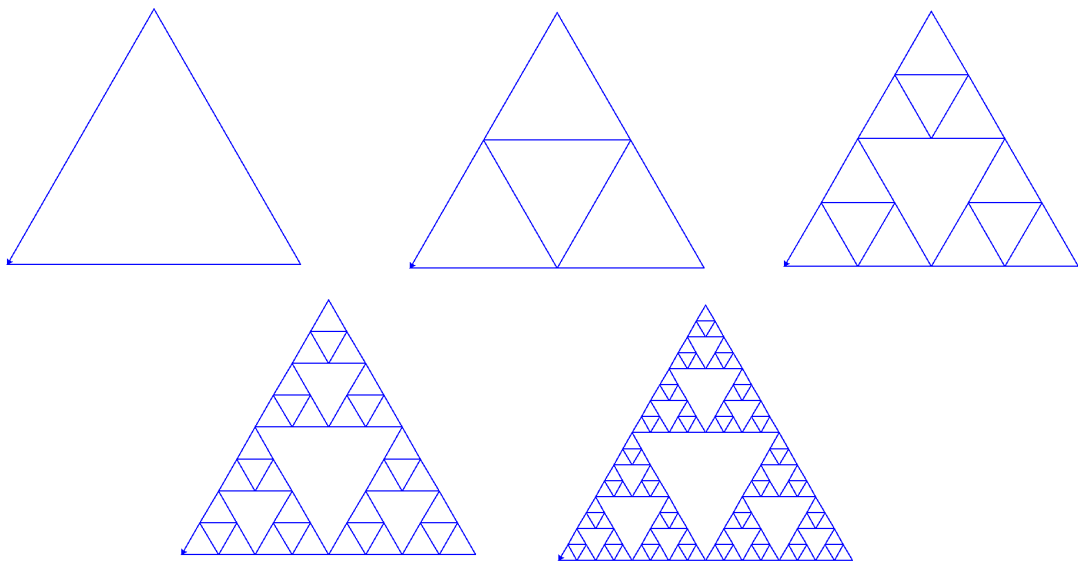
- `depart = "AdBdB"`
- `regle1 = ("A", "AdBgAgBdA")` c'est-à-dire **A** → **AdBgAgBdA**
- `regle2 = ("B", "BB")` c'est-à-dire **B** → **BB**
- pour $k = 0$, la fonction renvoie le mot de départ `AdBdB`,
- pour $k = 1$, la fonction renvoie `AdBgAgBdAdBBdBB`,
- pour $k = 2$, la fonction renvoie :

`AdBgAgBdAdBBgAdBgAgBdAgBBdAdBgAgBdAdBBBBdBBBB`

3. Trace les premières images du triangle de Sierpinski donné comme ci-dessus par :

départ : **AdBdB** règles : **A** → **AdBgAgBdA** **B** → **BB**

L'angle est de -120 degrés. Voici les images pour $k = 0$ jusqu'à $k = 4$.



4. Trace d'autres figures fractales à partir des L-systèmes suivants.

- La courbe du dragon :

`depart="AX" regle1=("X", "XgYAg") regle2=("Y", "dAXdY")`

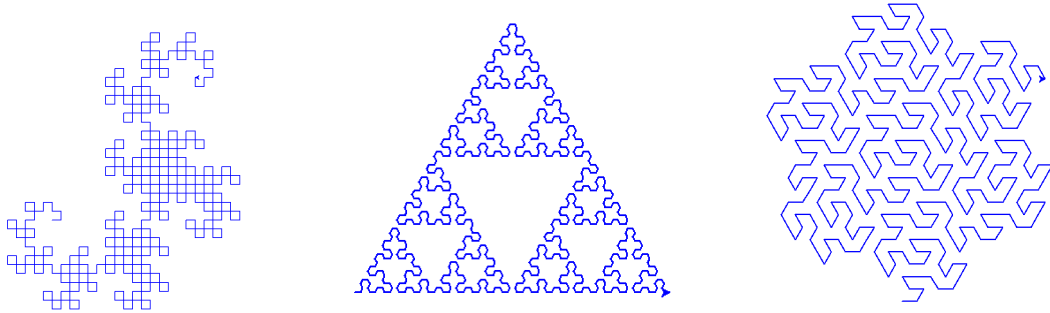
Les lettres X et Y ne correspondent à aucune action.

- Une variante du triangle de Sierpinski, avec `angle = 60` :

`depart="A" regle1=("A", "BdAdB") regle2=("B", "AgBgA")`

- La courbe de Gosper, avec `angle = 60` :

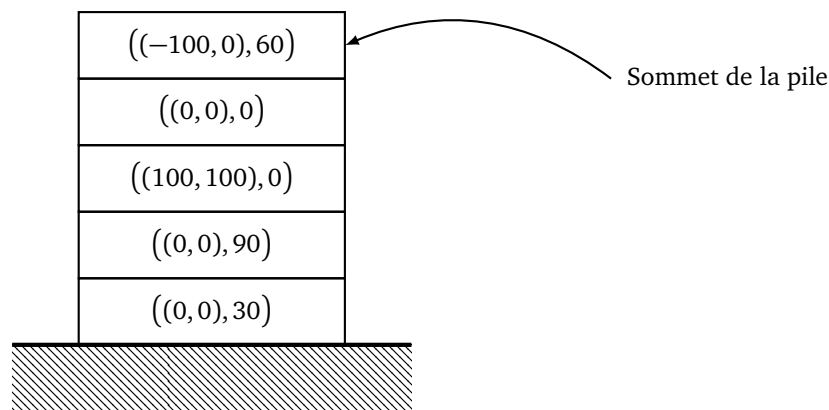
`depart="A" regle1=("A", "AgBggBdAddAAdBg") regle2=("B", "dAgBBggBgAddAdB")`



Invente et trace tes propres L-systèmes avec deux règles !

Cours 3 (Piles).

Une **pile** est une zone de stockage temporaire. Les détails sont dans la fiche « Calculatrice polonaise – Piles ». Voici juste quelques rappels.



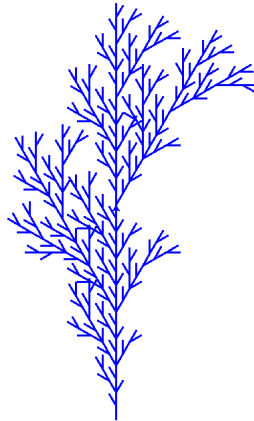
Une pile

- Une pile c'est comme un pile d'assiettes ; on pose des éléments un par un au-dessus de la pile ; on retire les éléments un par un, également à partir du dessus. C'est le principe : « dernier arrivé, premier parti » (*last in, first out*).
- On modélise une pile par une liste.
- Au départ la pile est vide : `pile = []`.
- **Empiler.** On ajoute les éléments en fin de liste : `pile = pile + [element]`.
- **Dépiler.** On retire un élément par la commande `pop()` :

$$\text{element} = \text{pile.pop}()$$
 qui renvoie le dernier élément de la pile et le supprime de la liste.
- Sur le dessin et dans l'activité suivante, les éléments de la pile sont du type $((x, y), \theta)$ qui stockeront un état de la tortue : (x, y) est la position et θ sa direction.

Activité 4 (L-système et tortue à pile).

Objectifs : améliorer nos tracés en autorisant d'avancer sans tracer et aussi à l'aide d'une sorte de retour en arrière, afin de tracer des plantes.

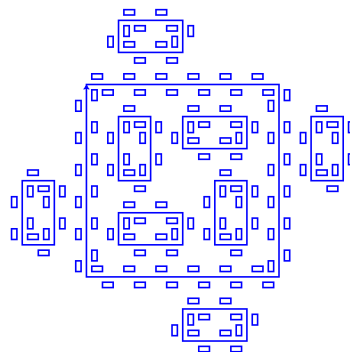


1. **Avancer sans tracer.**

Augmente les possibilités en autorisant la tortue à avancer sans tracer de trait, lorsque l'instruction est la lettre **a** (en minuscule). (Il suffit de modifier la fonction `trace_lsysteme()`.)

Trace alors le L-système suivant :

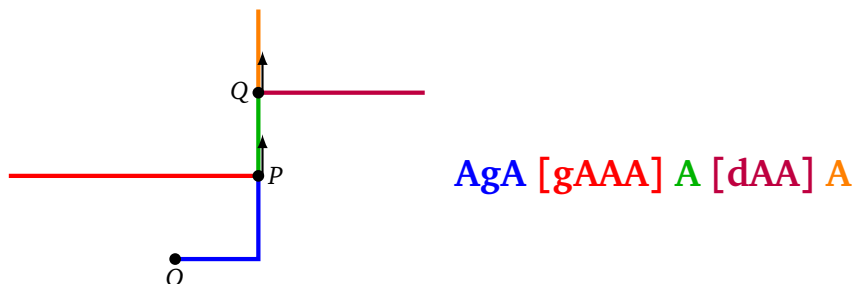
- depart = "AdAdAdA"
- regle1 = ("A", "AgadAAGAgAAGAgAAdagAAdAdAAdAAdAAA")
- regle2 = ("a", "aaaaaa")



2. Retour en arrière.

On autorise maintenant des crochets dans nos mots. Par exemple **AgA[gAAA]A[dAA]A**. Lorsque l'on rencontre un crochet ouvrant « [», on mémorise la position de la tortue, puis les commandes entre crochets sont exécutées comme d'habitude, lorsque l'on trouve un crochet fermant «] » on repart de la position mémorisée auparavant.

Comprenons l'exemple du tracé du **AgA [gAAA] A [dAA] A**.

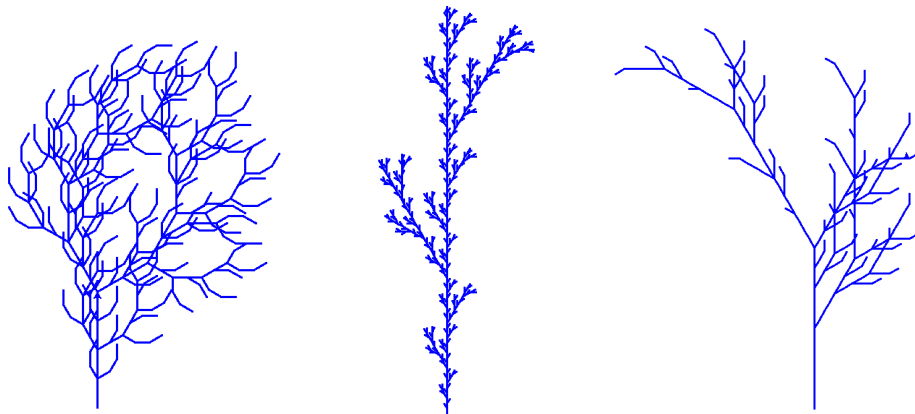


- **AgA** : on part du point O , on avance, on tourne, on avance.

- **[gAAA]** : on retient la position actuelle (le point P) et aussi la direction ; on tourne, on avance trois fois (on trace le segment rouge) ; à la fin on replace la tortue à la position P (sans tracer et avec la même direction que celle auparavant).
- **A** : depuis P on avance (segment vert).
- **[dAA]** : on retient la position Q et la direction, on tourne et on trace le segment violet. On revient en Q avec l'ancienne direction.
- **A** : depuis Q on trace le dernier segment.

Voici comment tracer un mot contenant des crochets à l'aide d'une pile :

- Au départ la pile est vide.
 - On lit un par un les caractères du mot. Les actions sont les mêmes qu'auparavant.
 - Si le caractère est le crochet ouvrant « [» alors on ajoute à la pile la position et la direction courante de la tortue $((x, y), \theta)$ que l'on obtient par `position()`, `heading()`.
 - Si le caractère est le crochet fermant «] » alors dépiler (c'est-à-dire lire l'élément du haut de la pile et le retirer). Mettre la position de la tortue et l'angle avec les valeurs lues, utiliser `goto()` et `setheading()`.
3. Trace les L-systèmes suivants, où l'on donne le mot de départ et la règle (ou les règles). L'angle est à choisir entre 20 et 30 degrés.
- "A" ("A", "A[gA]A[dA][A]")
 - "A" ("A", "A[gA]A[dA]A")
 - "A" ("A", "AA[dAgAgA]g[gAdAdA]")
 - "X" ("X", "A[gX][X]A[gX]dAX") ("A", "AA")
 - "X" ("X", "A[gX]A[dX]AX") ("A", "AA")
 - "X" ("X", "Ad[[X]gX]gA[gAX]dX") ("A", "AA")



Invente ta propre plante !

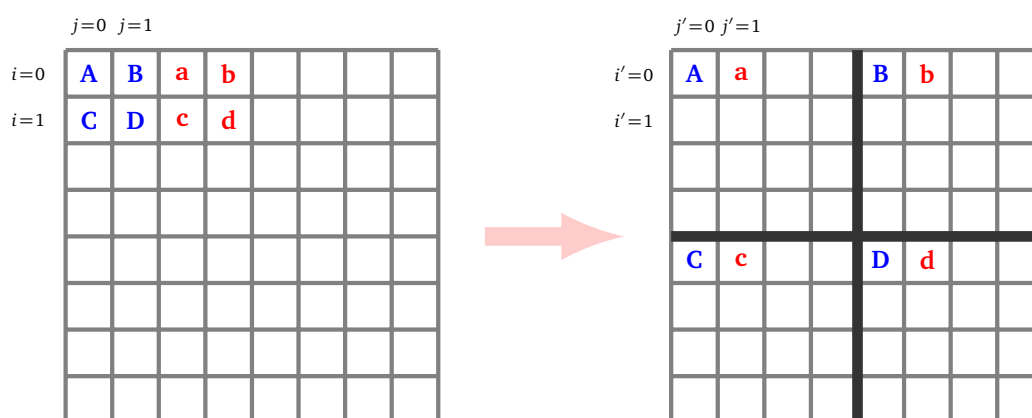
Images dynamiques

Nous allons déformer des images. En répétant ces déformations, les images deviennent brouillées. Mais par miracle au bout d'un certain nombre de répétitions l'image de départ réapparaît !

Cours 1 (Transformation du photomaton).

On part d'un tableau $n \times n$, avec n pair, chaque élément du tableau représente un pixel. Les lignes sont indexées de $i = 0$ à $i = n - 1$, les colonnes de $j = 0$ à $j = n - 1$. À partir de cette image on calcule une nouvelle image en déplaçant chaque pixel selon une transformation, appelée **transformation du photomaton**.

On découpe l'image de départ selon des petits carrés de taille 2×2 . Chaque petit carré est donc composé de quatre pixels. On envoie chacun de ces pixels à quatre endroits différents de la nouvelle image : le pixel en haut à gauche reste dans une zone en haut à gauche, le pixel en haut à droite du petit carré, s'envoie dans une zone en haut à droite de la nouvelle image,...



Par exemple le pixel en position (1, 1) (symbolisé par la lettre **D**) est envoyé en position (4, 4).

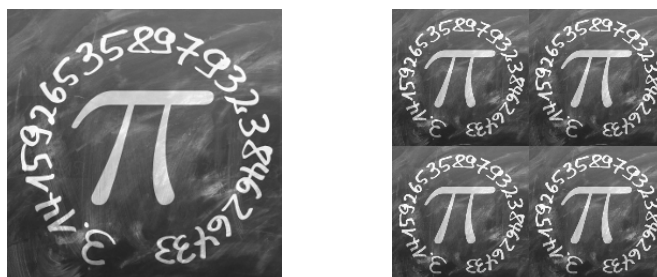
Explicitons ce principe par des formules. Pour chaque couple (i, j) , on calcule son image (i', j') par la transformation du photomaton selon les formules suivantes :

- Si i et j sont pairs : $(i', j') = (i//2, j//2)$.
- Si i est pair et j est impair : $(i', j') = (i//2, (n + j)//2)$.
- Si i est impair et j est pair : $(i', j') = ((n + i)//2, j//2)$.
- Si i et j sont impairs : $(i', j') = ((n + i)//2, (n + j)//2)$.

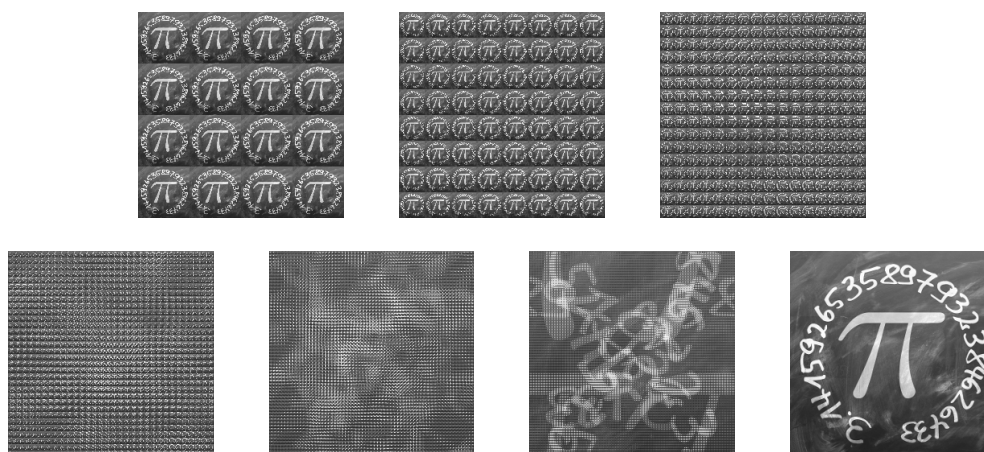
Voici un exemple d'un tableau 4×4 avant (à gauche) et après (à droite) la transformation du photomaton.

1	2	3	4	1	3	2	4
5	6	7	8	9	11	10	12
9	10	11	12	5	7	6	8
13	14	15	16	13	15	14	16

Voici une image 256 × 256 et sa première transformation :



Voici ce qui se passe si on répète plusieurs fois la transformation du photomaton :



L'image devient de plus en plus brouillée, mais au bout d'un certain nombre de répétitions de la transformation, on retombe sur l'image de départ !

Activité 1 (Transformation du photomaton).

Objectifs : programmer la transformation du photomaton qui décompose une image en sous-images. Lorsque l'on itère cette transformation l'image se déstructure petit à petit, puis d'un coup se reforme !

1. Programme une fonction `transformation(i, j, n)` qui met en œuvre la formule de la transformation du photomaton et renvoie les coordonnées (i', j') de l'image du pixel (i, j) .

Par exemple `transformation(1, 1, 8)` renvoie $(4, 4)$.

2. Programme une fonction `photomaton(tableau)` qui renvoie le tableau calculé après transformation.

Par exemple le tableau de gauche est transformé en le tableau de droite.

1	2	3	4	1	3	2	4
5	6	7	8	9	11	10	12
9	10	11	12	5	7	6	8
13	14	15	16	13	15	14	16

Indications. Tu peux initialiser un nouveau tableau par la commande :

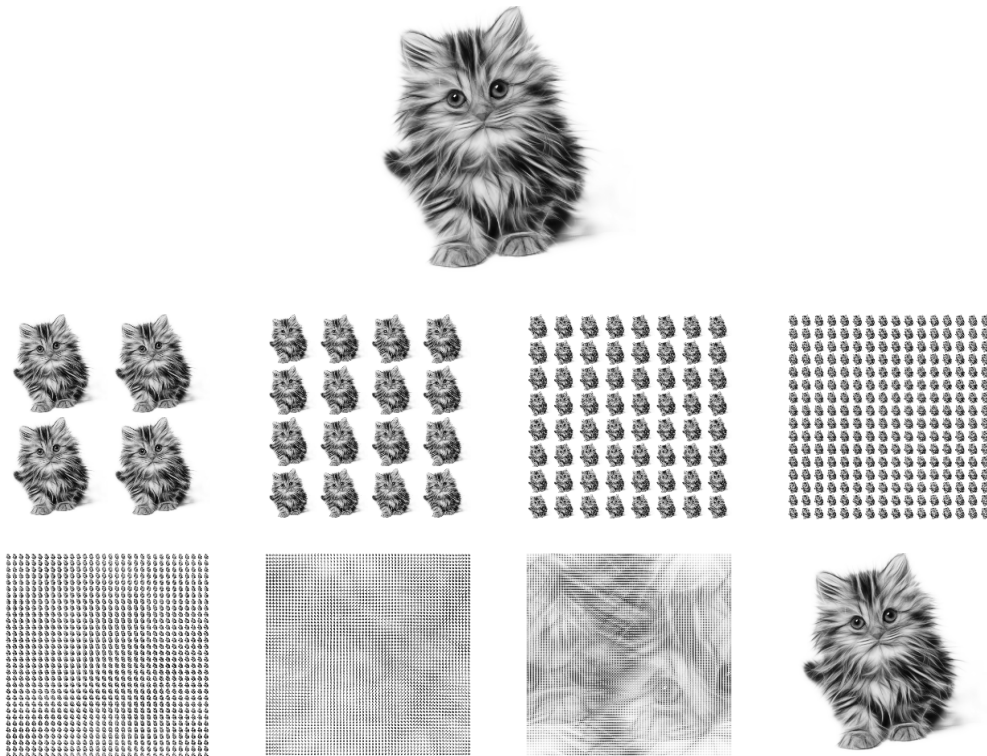
```
nouv_tableau = [[0 for j in range(n)] for i in range(n)]
```

Puis le remplir par des commandes du type :

```
nouv_tableau[ii][jj] = tableau[i][j]
```

3. Programme une fonction `photomaton_iterer(tableau, k)` qui renvoie le tableau calculé après k itérations de la transformation du photomaton.
4. À finir après avoir fait l'activité 2.
Programme une fonction `photomaton_images(nom_image, kmax)` qui calcule les images correspondant à la transformation du photomaton, pour toutes les itérations allant de $k = 1$ à $k = k_{\max}$.
5. Expérimente pour différentes valeurs de la taille n , afin de voir au bout de combien d'itérations on retrouve l'image de départ.

Voici l'image de départ de taille 256×256 et les images obtenues par itérations de la transformation du photomaton pour $k = 1$ jusqu'à $k = 8$. Au bout de 8 itérations on retrouve l'image initiale !



Activité 2 (Conversion tableau/image).

Objectifs : passer d'un tableau à un fichier d'image et inversement. Le format pour afficher les images est le format « pgm » qui a été manipulé dans la fiche « Fichiers ».

1. Tableau vers image.

Programme une fonction `tableau_vers_image(tableau, nom_image)` qui écrit un fichier image au format « pgm » à partir d'un tableau de niveaux de gris.

```
P2
5 5
255
128 192 128 192 128
224 0 228 0 224
228 228 228 228 228
224 64 64 64 224
192 192 192 192 192
```



Par exemple avec `tableau = [[128,192,128,192,128], [224,...]]`, la commande `tableau_vers_image(tableau,"test")` écrit un fichier `test.pgm` (à gauche) qui s'afficherait comme l'image à droite.

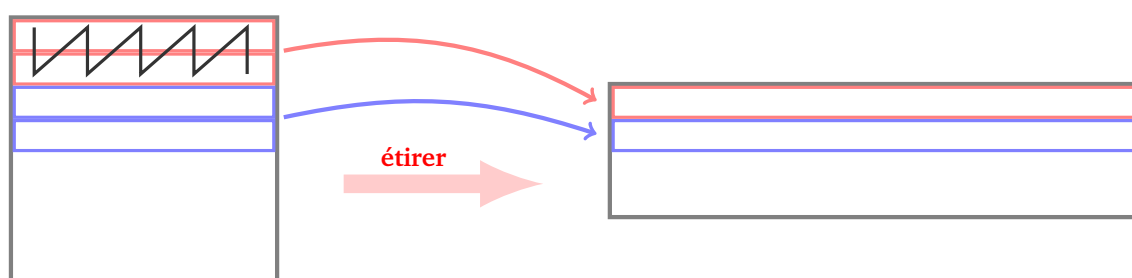
2. Image vers tableau.

Programme une fonction `image_vers_tableau(nom_image)` qui à partir d'un fichier image au format « pgm », renvoie un tableau des niveaux de gris.

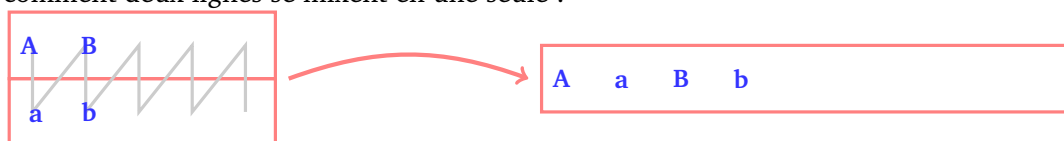
Cours 2 (Transformation du boulanger).

On part d'un tableau $n \times n$, avec n pair dont chaque élément représente un pixel. On va appliquer deux transformations élémentaires à chaque fois :

- **Étirer.** Le principe est le suivant : les deux premières lignes (chacune de longueur n) produisent une seule ligne de longueur $2n$ en mixant les valeurs de chaque ligne en alternant un élément du haut, un élément du bas.



Voici comment deux lignes se mixent en une seule :

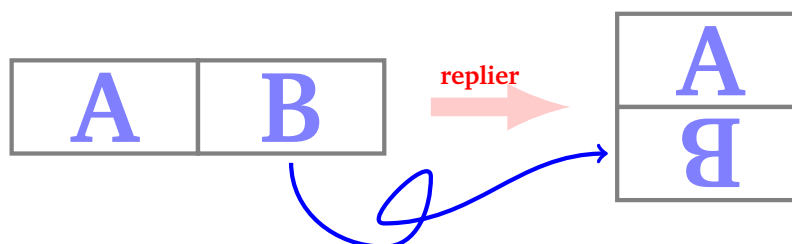


Formules. Un élément en position (i, j) du tableau d'arrivée, correspond à un élément $(2i, j//2)$ (si j est pair) ou bien $(2i + 1, j//2)$ (si j est impair) avec ici $0 \leq i < \frac{n}{2}$ et $0 \leq j < 2n$.

Exemple. Voici un tableau 4×4 à gauche, et le tableau étiré 2×8 à droite. Les lignes 0 et 1 à gauche donnent la ligne 0 à droite. Les lignes 2 et 3 à gauche donnent la ligne 1 à droite.

1	2	3	4						
5	6	7	8	1	5	2	6	3	7
9	10	11	12	9	13	10	14	11	15
13	14	15	16						

- **Replier.** Le principe est le suivant : la partie droite d'un tableau étiré est retournée, puis ajoutée sous la partie gauche. Partant d'un tableau $\frac{n}{2} \times 2n$ on obtient un tableau $n \times n$.



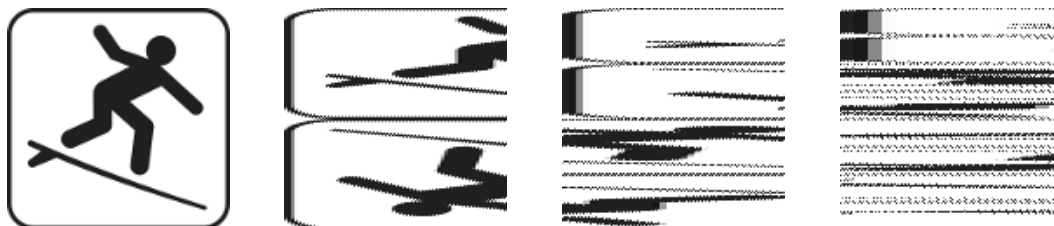
Formules. Pour $0 \leq i < \frac{n}{2}$ et $0 \leq j < n$ les éléments en position (i, j) du tableau sont conservés. Pour $\frac{n}{2} \leq i < n$ et $0 \leq j < n$ un élément du tableau d'arrivée (i, j) , correspond à un élément $(\frac{n}{2} - i - 1, 2n - 1 - j)$ du tableau de départ.

Exemple. À partir du tableau étiré 2×8 à gauche, on obtient un tableau replié 4×4 à droite.

1	5	2	6	3	7	4	8	1	5	2	6
9	13	10	14	11	15	12	16	9	13	10	14
								16	12	15	11
								8	4	7	3

La **transformation du boulanger** est la succession d'un étirement et d'un repliement. Partant d'un tableau $n \times n$ on obtient encore un tableau $n \times n$.

Voyons un exemple de l'action de plusieurs transformations du boulanger. À gauche l'image initiale de taille 128×128 , puis le résultat de $k = 1, 2, 3$ itérations.



Voici les images pour $k = 12, 13, 14, 15$ itérations :



Activité 3 (Transformation du boulanger).

Objectifs : programmer une nouvelle transformation qui étire puis replie une image. Encore une fois l'image se déforme de plus en plus mais, au bout d'un certain nombre d'itérations, on retrouve l'image de départ.

1. Programme une fonction `boulanger_etirer(tableau)` qui renvoie un nouveau tableau obtenu en « étirant » le tableau donné en entrée.
2. Programme une fonction `boulanger_replier(tableau)` qui renvoie un tableau obtenu en « repliant » le tableau donné en entrée.
3. Programme une fonction `boulanger_iterer(tableau, k)` qui renvoie le tableau calculé après k itérations de la transformation du boulanger.

Par exemple, voici un tableau 4×4 à gauche, son image par la transformation ($k = 1$) et son image après une seconde transformation ($k = 2$).

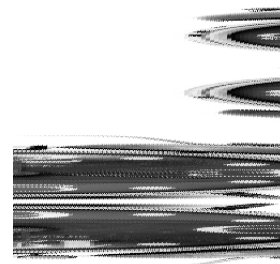
1	2	3	4	1	5	2	6	1	9	5	13
5	6	7	8	9	13	10	14	16	8	12	4
9	10	11	12	16	12	15	11	3	11	7	15
13	14	15	16	8	4	7	3	14	6	10	2

4. Programme une fonction `boulanger_images(nom_image, kmax)` qui calcule les images correspondant à la transformation du boulanger, avec des itérations allant de $k = 1$ à $k = k_{\max}$.

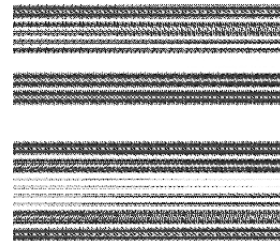
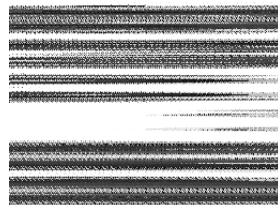
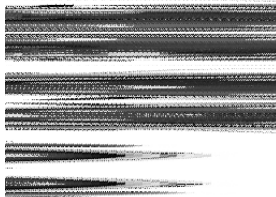
5. Expérimente pour différentes valeurs de la taille n , afin de voir au bout de combien d'itérations on retrouve l'image de départ.

Attention ! Il faut parfois itérer beaucoup avant de retrouver l'image de départ. Par exemple avec $n = 4$, on retrouve l'image de départ au bout de $k = 5$ itérations, avec $n = 256$ c'est $k = 17$. Conjecture une valeur de retour dans le cas où n est une puissance de 2. Par contre pour $n = 10$, il faut $k = 56\,920$ itérations !

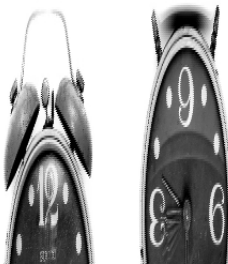
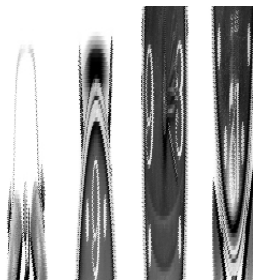
Voici un exemple avec une image de taille 256×256 , d'abord l'image initiale, puis une première itération ($k = 1$) et une deuxième itération ($k = 2$).



$k = 3, 4, 5 :$



$k = 15, 16, 17 :$

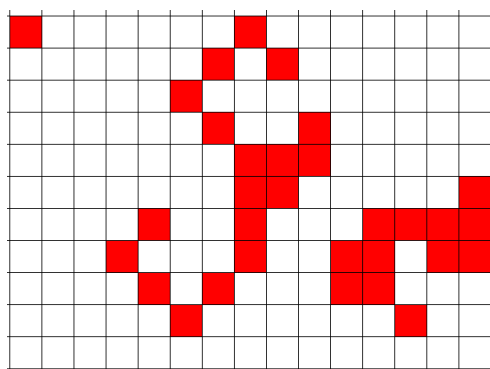


Pour $k = 17$ on retrouve l'image de départ !

Cette fiche est basée sur l'article « Images brouillées, images retrouvées » par Jean-Paul Delahaye et Philippe Mathieu (Pour la Science, 1997).

Jeu de la vie

Le jeu de la vie est un modèle simple de l'évolution d'une population de cellules qui naissent et meurent au cours du temps. Le « jeu » consiste à trouver des configurations initiales qui donnent des évolutions intéressantes : certains groupes de cellules disparaissent, d'autres se stabilisent, certains se déplacent...

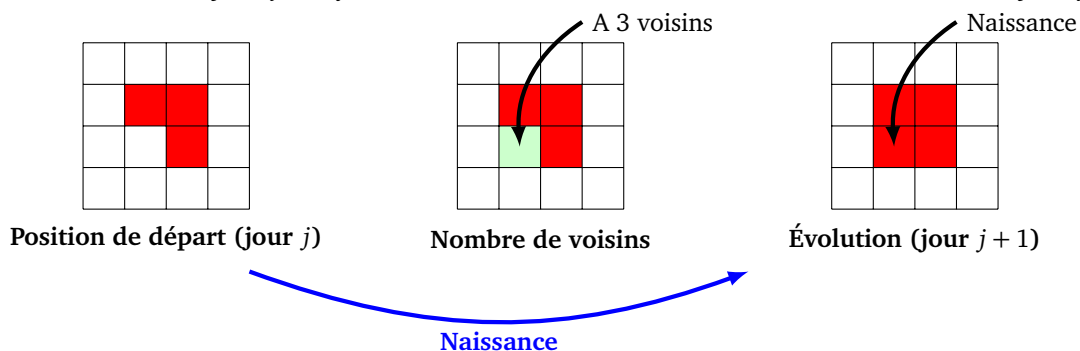


Cours 1 (Règles du jeu de la vie).

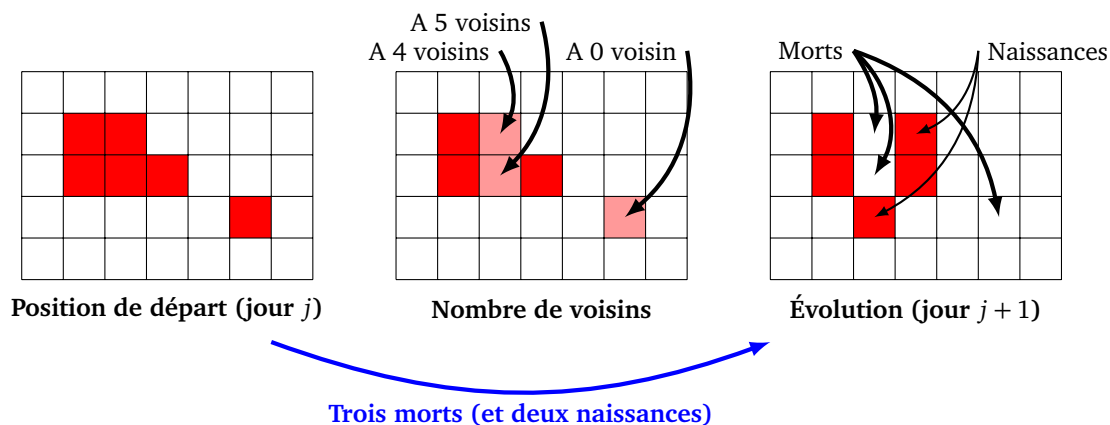
Le jeu de la vie se déroule sur une grille. Chaque case peut contenir une cellule. Partant d'une configuration initiale, chaque jour des cellules vont naître et d'autres mourir en fonction du nombre de ses voisins.

Voici les règles :

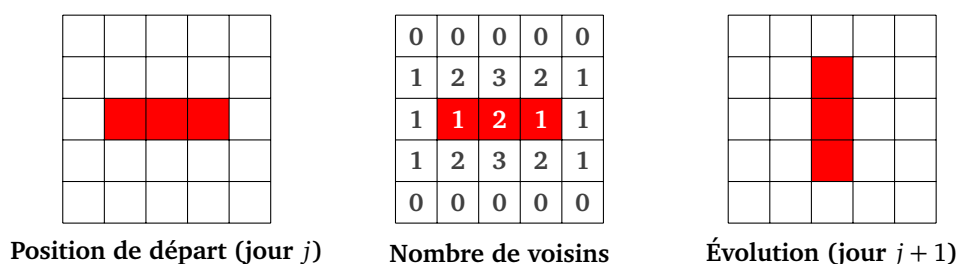
- Pour une case vide au jour j et ayant exactement 3 cellules voisines : une cellule naît au jour $j + 1$.



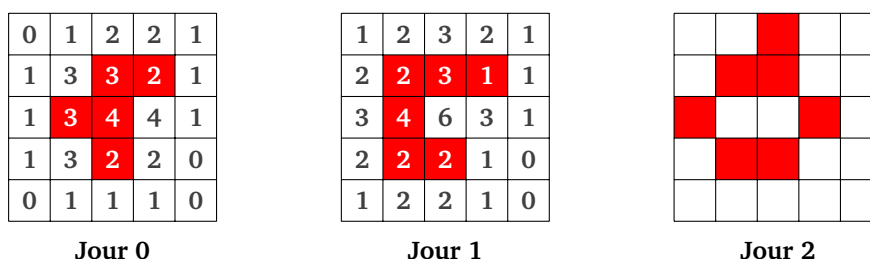
- Pour une case contenant une cellule au jour j , ayant soit 2 ou soit 3 cellules voisines : alors la cellule continue de vivre. Dans les autres cas la cellule meurt (avec 0 ou 1, elle meurt d'isolement, avec plus de 4 voisins, elle meurt de surpopulation !).



Voici un exemple simple, le « clignotant » :



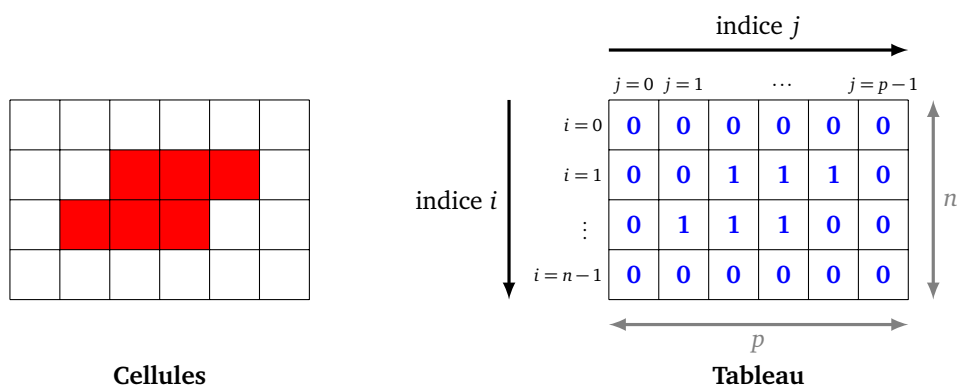
Voici un autre exemple (avec directement le nombre de voisins), la première évolution et l'évolution de l'évolution.



Activité 1 (Tableaux à deux dimensions).

Objectifs : définir et afficher des tableaux indexés par deux indices.

Nous modélisons l'espace de vie des cellules par un tableau à double entrée, contenant des entiers, 1 pour signifier la présence d'une cellule, 0 sinon. Voici en exemple la configuration « bouche » et son tableau :



- Initialise deux variables n (la hauteur du tableau) et p (la largeur) (par exemple à 5 et 8).
 - Définis un tableau à deux dimensions rempli de zéros par la commande :

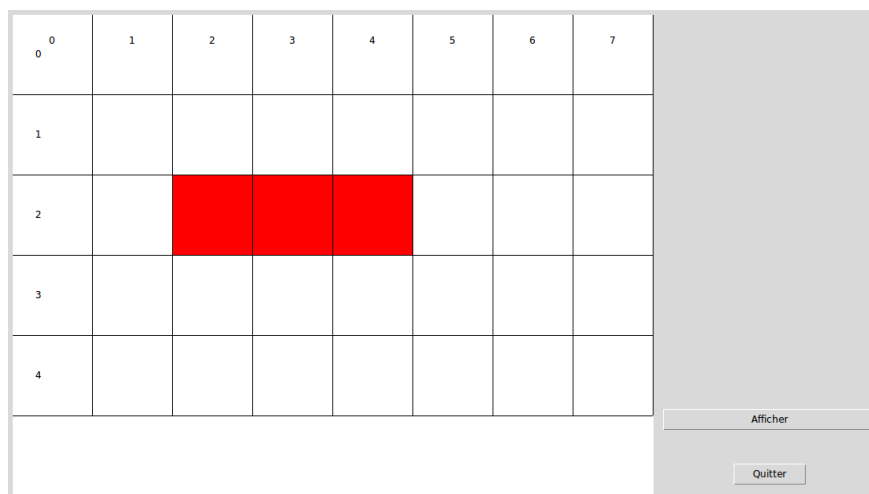

```
tableau = [[0 for j in range(p)] for i in range(n)]
```
 - Par des instructions du type `tableau[i][j] = 1` remplis le tableau afin de définir la configuration du clignotant, de la bouche...
- Programme l'affichage à l'écran d'un tableau donné. Par exemple, le clignotant s'affiche ainsi :

```
00000000
00000000
00111000
00000000
00000000
```

Indication : par défaut la commande `print()` passe à la ligne suivante à chaque appel (il rajoute le caractère `"\n"` qui est le caractère de fin de ligne). On peut lui spécifier de ne pas le faire par l'option `print("Mon texte",end="")`.

Activité 2 (Affichage graphique).

Objectifs : réaliser l'affichage graphique d'une configuration de cellules.



- Programme la création d'une fenêtre `tkinter` (voir la fiche « Statistique – Visualisation de données »). Écris une fonction `afficher_lignes()`, sans paramètre, qui affiche une grille vierge de coordonnées. Tu auras besoin d'une constante `echelle` (par exemple `echelle = 50`) pour transformer les indices i et j en des coordonnées graphiques x et y , suivant la relation : $x = e \times j$ et $y = e \times i$ (e étant l'échelle).
Bonus. Tu peux aussi rajouter la valeur des indices i et j en haut et à gauche pour faciliter la lecture de la grille.
- Construis une fonction `afficher_tableau(tab)` qui affiche graphiquement les cellules d'un tableau.
Bonus. Rajoute un bouton « Afficher » et un bouton « Quitter » (voir la fiche « Statistique – Visualisation de données »).

Activité 3 (Évolution).

Objectifs : calculer l'évolution d'une configuration au jour d'après.

1. Programme une fonction `nombre_voisins(i, j, tab)` qui calcule le nombre de cellules voisines vivantes de la cellule (i, j) .

Indications.

- Il y a en tout au maximum 8 voisins possibles. Le plus simple est de les tester un par un !
 - Pour le comptage, il faut faire bien attention aux cellules qui sont placées près d'un bord (et ont moins de 8 voisins).
2. Programme une fonction `evolution(tab)` qui en entrée reçoit un tableau et qui en sortie renvoie un nouveau tableau correspondant à la situation au jour d'après, selon les règles du jeu de la vie décrites au début. Par exemple, si le tableau de gauche correspond à l'entrée, alors la sortie correspond au tableau de droite :

00000000		00000000
00000000		00010000
00111000	évolue en	00010000
00000000		00010000
00000000		00000000

Indications. Pour définir un nouveau tableau reprend la commande :

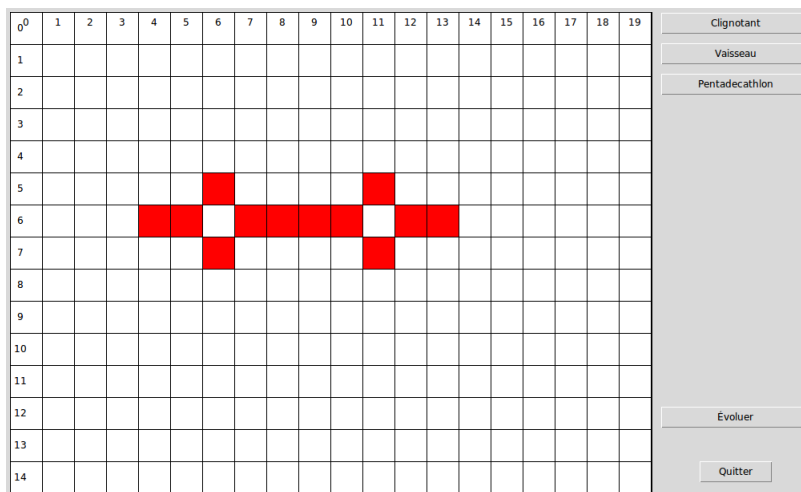
```
nouveau_tableau = [[0 for j in range(p)] for i in range(n)]
```

puis modifie le tableau comme voulu.

Activité 4 (Itérations).

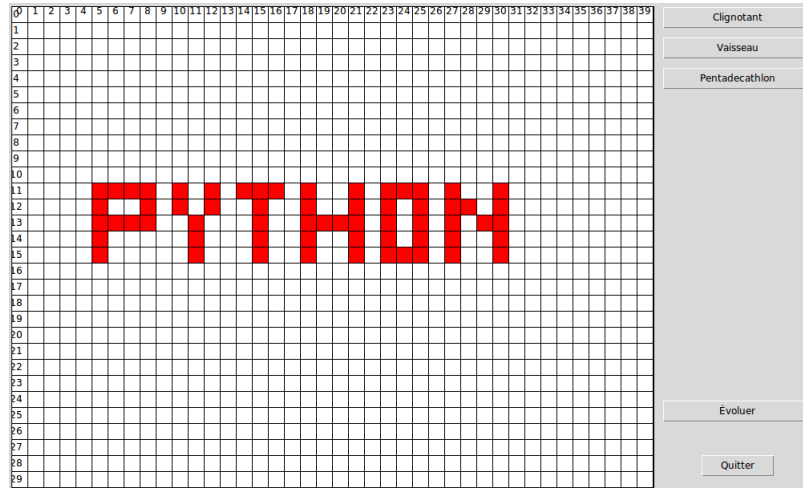
Objectifs : finir le programme graphique afin que l'utilisateur puisse définir des configurations et les faire évoluer d'un simple clic.

1. Améliore la fenêtre graphique afin de faciliter la vie de l'utilisateur :
 - Un bouton « évoluer » qui à chaque clic affiche l'évolution.
 - Des boutons permettant d'afficher des configurations pré-définies (sur la capture d'écran ci-dessous c'est la configuration « pentadecathlon »).



2. Perfectionne ton programme afin que l'utilisateur dessine la configuration qu'il souhaite par des clics de la souris. Un clic sur une case éteinte l'allume, un clic sur une case allumée l'éteint. Tu peux décomposer ce travail en trois fonctions :

- `allumer_eteindre(i,j)`, qui commute la cellule (i,j) ;
- `xy_vers_ij(x,y)` qui convertit des coordonnées graphiques (x,y) en des coordonnées entières (i,j) (utiliser la variable `echelle` et la division entière).
- `action_clic_souris(event)` pour récupérer les coordonnées (x,y) d'un clic de souris (voir le cours ci-dessous) et commuter la case cliquée.



Cours 2 (Clic de souris).

Voici un petit programme qui affiche une fenêtre graphique. Chaque fois que l'utilisateur clique (avec le bouton gauche de la souris) le programme affiche un petit carré (sur la fenêtre) et affiche « Clic à $x = \dots$, $y = \dots$ » (sur la console).

```
from tkinter import *

# Fenêtre
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

# Capture des clics de souris
def action_clic_souris(event):
    canvas.focus_set()
    x = event.x
    y = event.y
    canvas.create_rectangle(x,y,x+10,y+10,fill="red")
    print("Clic à x =",x," y =",y)
    return

# Association clic/action
canvas.bind("<Button-1>", action_clic_souris)

# Lancement
root.mainloop()
```

Voici quelques explications :

- La création de la fenêtre est habituelle. Le programme se termine par le lancement avec la commande

`mainloop()`.

- Le premier point clé est d'associer un clic de souris à une action, c'est ce que fait la ligne

```
canvas.bind("<Button-1>", action_clic_souris)
```

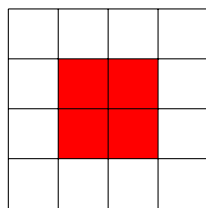
Chaque fois que le bouton gauche de la souris est cliqué, Python exécute la fonction `action_clic_souris`. (Note qu'il n'y a pas de parenthèses pour l'appel à la fonction.)

- Second point clé : la fonction `action_clic_souris` récupère les coordonnées du clic et ici ensuite fait deux choses : elle affiche un petit rectangle à l'endroit du clic et affiche dans la fenêtre du terminal les coordonnées (x, y) .
- Les coordonnées x et y sont exprimées en pixels ; $(0, 0)$ désigne le coin en haut à gauche de la fenêtre (la zone délimitée par `canvas`).

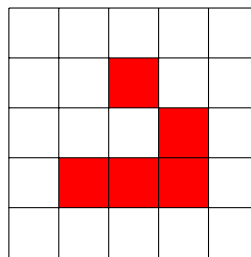
Voici quelques idées pour aller plus loin :

- Fais en sorte que la grille s'adapte automatiquement à la largeur de l'écran, c'est-à-dire calcule `echelle` en fonction de `n` et `p`.
- Rajoute une étape intermédiaire avant d'évoluer : colorie en vert une cellule qui va naître et en noir une cellule qui va mourir. Pour cela les éléments de `tableau` pourront prendre des valeurs autres que 0 et 1.

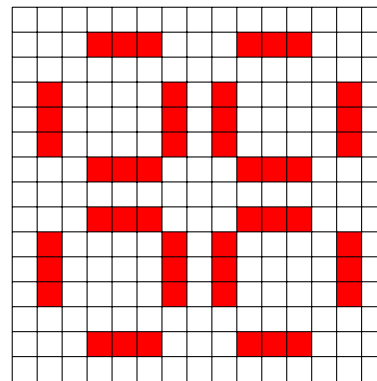
Voici quelques configurations intéressantes.



Carré



Vaisseau



Pulsar

Tu en trouveras plein d'autres sur internet mais surtout amuse-toi à en découvrir de nouvelles !

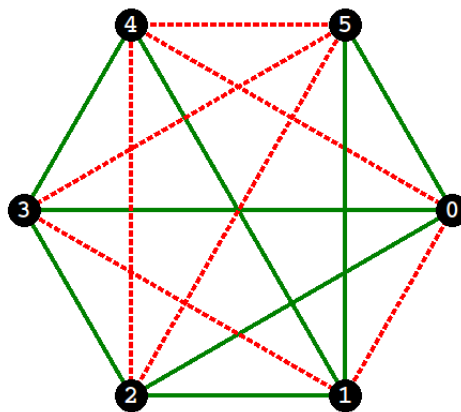
En particulier, trouve des configurations :

- qui restent fixes au cours du temps ;
- qui évoluent, puis deviennent fixes ;
- qui sont périodiques (les mêmes configurations reviennent en boucles) avec une période de 2, 3 ou plus ;
- qui voyagent ;
- qui propulsent des cellules ;
- dont la population augmente indéfiniment !

Graphes et combinatoire de Ramsey

Chapitre 21

Tu vas voir qu'un problème tout simple, qui concerne les relations entre seulement six personnes, va demander énormément de calculs pour être résolu.



Cours 1 (Le problème de Ramsey).

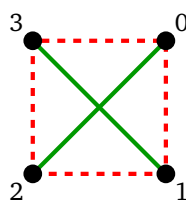
Proposition. Dans un groupe de 6 personnes, il y a toujours 3 amis (les trois se connaissent deux à deux) ou 3 étrangers (les trois sont tous des inconnus les uns pour les autres).

Le but de cette fiche est de faire démontrer à l'ordinateur cet énoncé. Pour cela nous allons modéliser le problème par des graphes, puis nous allons vérifier l'énoncé pour chacun des 32 768 graphes possibles.

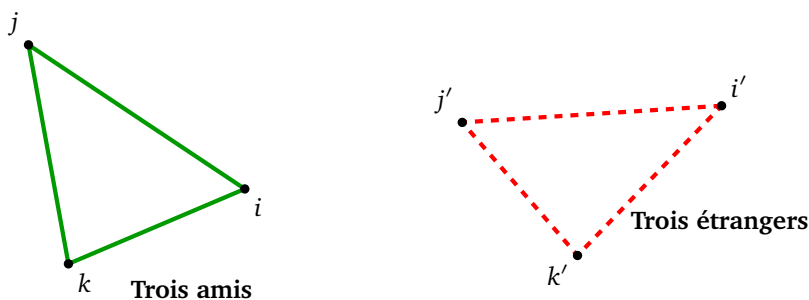
On considère n personnes. Pour deux personnes parmi elles, soit elles se connaissent (elles sont amies), soit elles ne se connaissent pas (elles sont étrangères l'une à l'autre). Nous schématisons cela par un graphe :

- une personne est représentée par un sommet (numéroté de 0 à $n - 1$) ;
- si deux personnes sont amies, on relie les sommets correspondants par une arête verte ;
- sinon (elles sont étrangères), on relie les sommets correspondants par une arête pointillée rouge.

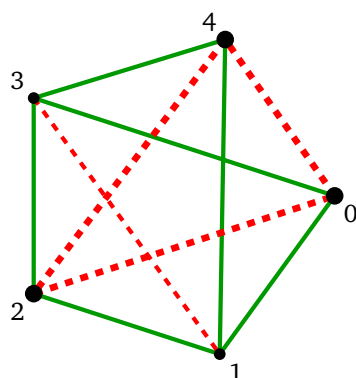
Le graphe ci-dessous signifie que 0 est ami avec 2 ; 1 est ami avec 3. Les autres paires ne se connaissent pas.



Un graphe vérifie le problème de Ramsey, s'il y a parmi ses sommets, ou bien 3 amis, ou bien s'il y a 3 étrangers.



Voici un exemple de graphe à 5 sommets qui vérifie l'énoncé : il possède bien 3 sommets étrangers (les sommets 0, 2 et 4), même s'il ne possède pas trois amis.



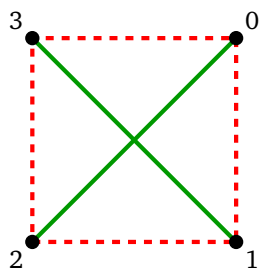
Un graphe avec $n = 5$ qui vérifie l'énoncé de Ramsey

Cours 2 (Modélisation).

Nous modélisons un graphe par un tableau à double entrée, contenant des 0 et des 1.

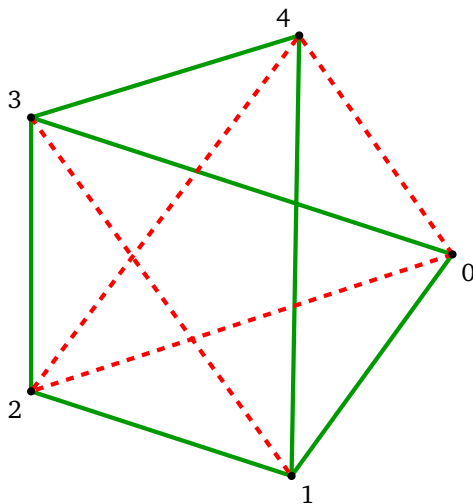
Soit un graphe ayant n sommets, numérotés de 0 à $n - 1$. Le **tableau du graphe** est un tableau de taille $n \times n$ dans lequel on place un 1 en position (i, j) si les sommets i et j sont reliés par une arête, sinon on place un 0.

Premier exemple ci-dessous : les sommets 0 et 2 sont amis (car reliés par une arête verte) donc le tableau contient un 1 en position $(0, 2)$ et aussi en $(2, 0)$. De même 1 et 3 sont amis, donc le tableau contient un 1 en position $(1, 3)$ et $(3, 1)$. Le reste du tableau contient des 0.



		indice j				
		$j = 0$	$j = 1$	$j = 2$	$j = 3$	
indice i	$i = 0$	0	0	1	0	n
	$i = 1$	0	0	0	1	
	$i = 2$	1	0	0	0	
	$i = 3$	0	1	0	0	
		n				

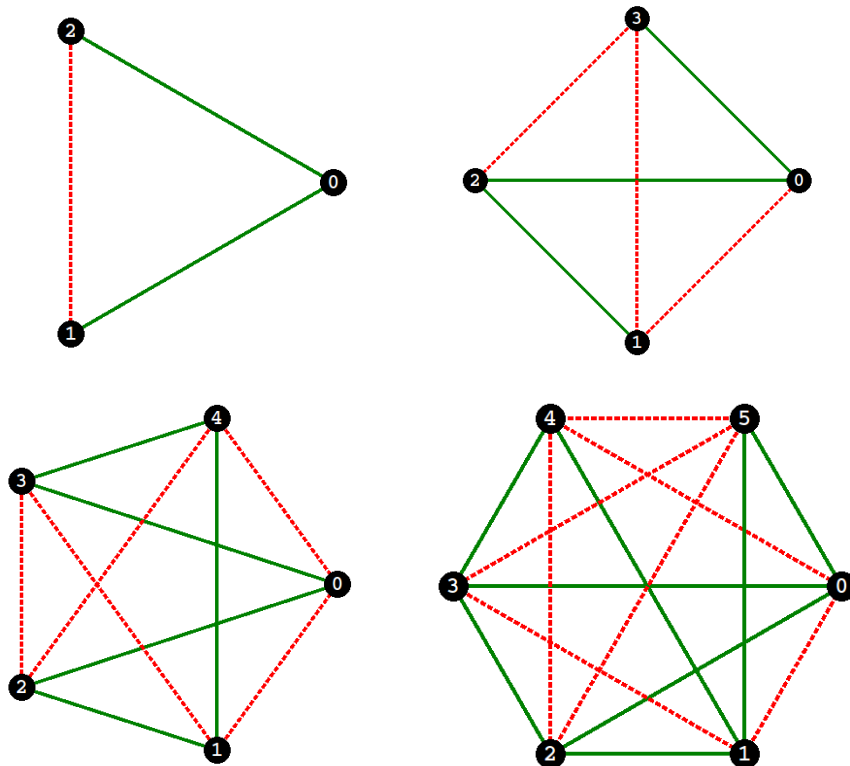
Voici un graphe plus compliqué et son tableau :



	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 0$	0	1	0	1	0
$i = 1$	1	0	1	0	1
$i = 2$	0	1	0	1	0
$i = 3$	1	0	1	0	1
$i = 4$	0	1	0	1	0

Activité 1 (Construire des graphes).

Objectifs : définir des graphes et tester si trois sommets donnés sont amis.



1. Définis le tableau des graphes des quatre exemples ci-dessus. Tu peux commencer par initialiser le tableau par

```
graphe = [[0 for j in range(n)] for i in range(n)]
```

Puis ajoute des commandes :

```
graphe[i][j] = 1 et graphe[j][i] = 1
```

N'oublie pas que si le sommet i est relié au sommet j par une arête, alors il faut mettre un 1 en position (i, j) mais aussi en position (j, i) .

2. Définis une fonction `voir_graphe(graphe)` qui permet d'afficher à l'écran le tableau d'un graphe. Ainsi le troisième exemple ci-dessus (avec $n = 5$) doit s'afficher ainsi :

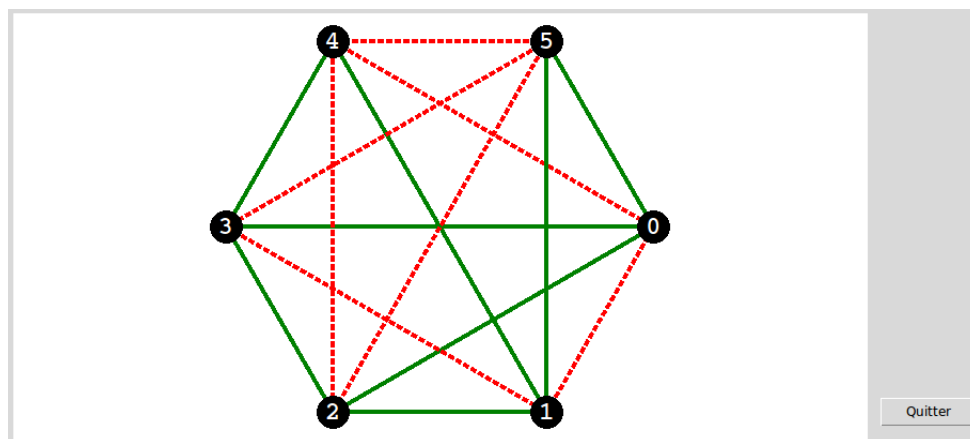
00110
 00101
 11000
 10001
 01010

3. On fixe trois sommets i, j, k d'un graphe. Écris une fonction `contient_3_amis_fixes(graphe,i,j,k)` qui teste si les sommets i, j, k sont trois amis (la fonction renvoie « vrai » ou « faux »). Fais le même travail avec une fonction `contient_3_etrangers_fixes(graphe,i,j,k)` pour savoir si ces sommets sont étrangers.

Trouve à la main sur le quatrième exemple, trois sommets amis ou étrangers et vérifie ta réponse à l'aide des fonctions que tu viens de définir.

Activité 2 (Afficher des jolis graphes).

Objectifs : dessiner un graphe ! Activité facultative.



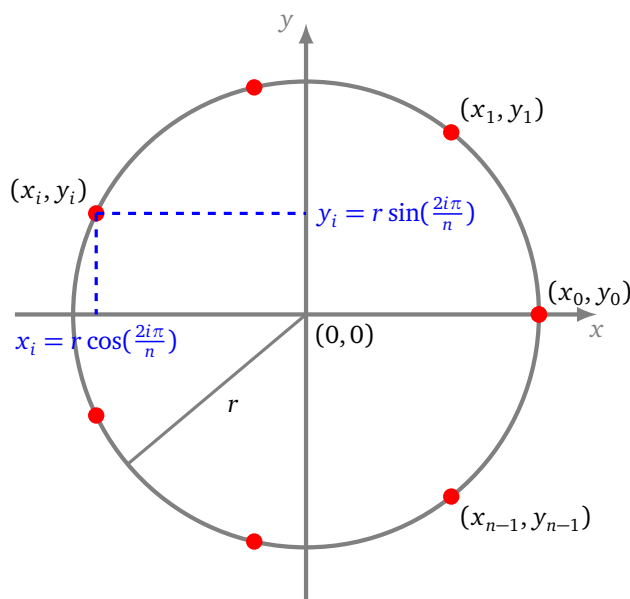
Programme l'affichage graphique d'un graphe par une fonction `afficher_graphe(graphe)`.

Indications. Cette activité n'est pas nécessaire pour la suite, elle aide juste à visualiser les graphes. Il faut utiliser le module `tkinter` et les fonctions `create_line()`, `create_oval()` et éventuellement `create_text()`.

Le point le plus délicat est d'obtenir les coordonnées des sommets. Tu auras besoin des fonctions sinus et cosinus (disponibles dans le module `math`). Les coordonnées (x_i, y_i) du sommet numéro i d'un graphe à n éléments peuvent être calculées par les formules :

$$x_i = r \cos\left(\frac{2i\pi}{n}\right) \quad \text{et} \quad y_i = r \sin\left(\frac{2i\pi}{n}\right).$$

Ces sommets sont situés sur le cercle de rayon r , centré en $(0,0)$. Tu devras choisir r assez grand (par exemple $r = 200$) et décaler le cercle pour bien l'afficher à l'écran.



Activité 3 (Écriture binaire avec des 0 non significatifs).

Objectifs : convertir un entier en écriture binaire avec éventuellement des zéros non significatifs.

Programme une fonction `decimal_vers_binaire(p,n)` qui affiche l'écriture binaire d'un entier p sur n bits. Le résultat est une liste de 0 et de 1.

Exemple.

- L'écriture binaire de $p = 37$ est 1.0.0.1.0.1. Si on veut son écriture binaire sur $n = 8$ bits alors il faut rajouter deux 0 non significatifs devant : 0.0.1.0.0.1.0.1.
- Ainsi le résultat de la commande `decimal_vers_binaire(37,8)` doit être `[0, 0, 1, 0, 0, 1, 0, 1]`.
- La commande `decimal_vers_binaire(37,10)` renvoie l'écriture de 37 en binaire sur 10 bits : `[0, 0, 0, 0, 1, 0, 0, 1, 0, 1]`.

Indications.

- Tu peux utiliser la commande `bin(p)` !
- La commande `list(ma_chaine)` renvoie la liste des caractères composant `ma_chaine`.
- Attention ! On veut une liste d'entiers 0 ou 1, pas des caractères '0' ou '1'. La commande `int('0')` renvoie 0 et `int('1')` renvoie 1.
- `ma_liste = ma_liste + [element]` ajoute un élément en fin de liste, alors que `ma_liste = [element] + ma_liste` ajoute l'élément en début de liste.

Cours 3 (Sous-ensembles).

Soit $E_n = \{0, 1, 2, \dots, n-1\}$ l'ensemble des entiers de 0 à $n-1$. L'ensemble E_n contient donc n éléments. Par exemple $E_3 = \{0, 1, 2\}$, $E_4 = \{0, 1, 2, 3\}$...

Sous-ensembles.

Quels sont les sous-ensembles de E_n ? Par exemple il y a 8 sous-ensembles de E_3 , ce sont :

- le sous-ensemble $\{0\}$ composé du seul élément 0 ;
- le sous-ensemble $\{1\}$ composé du seul élément 1 ;

- le sous-ensemble $\{2\}$ composé du seul élément 2 ;
- le sous-ensemble $\{0, 1\}$ composé de l'élément 0 et de l'élément 1 ;
- le sous-ensemble $\{0, 2\}$;
- le sous-ensemble $\{1, 2\}$;
- le sous-ensemble $\{0, 1, 2\}$ composé de tous les éléments ;
- l'ensemble vide \emptyset qui ne contient aucun élément !

Proposition. L'ensemble E_n contient 2^n sous-ensembles.

Par exemple $E_4 = \{0, 1, 2, 3\}$ possède $2^4 = 16$ sous-ensembles possibles. Amuse-toi à les trouver tous !
Pour E_6 il y a $2^6 = 64$ sous-ensembles possibles.

Sous-ensembles de cardinal fixé.

On cherche seulement les sous-ensembles ayant un nombre k fixé d'éléments.

Exemples :

- Pour $n = 3$ et $k = 2$, les sous-ensembles à deux éléments contenus dans $E_3 = \{0, 1, 2\}$ sont les trois paires : $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$.
- Pour $n = 5$ et $k = 3$, les sous-ensembles à trois éléments contenus dans $E_5 = \{0, 1, 2, 3, 4\}$ sont les 10 triplets : $\{0, 1, 2\}$, $\{0, 1, 3\}$, $\{0, 2, 3\}$, $\{1, 2, 3\}$, $\{0, 1, 4\}$, $\{0, 2, 4\}$, $\{1, 2, 4\}$, $\{0, 3, 4\}$, $\{1, 3, 4\}$, $\{2, 3, 4\}$.

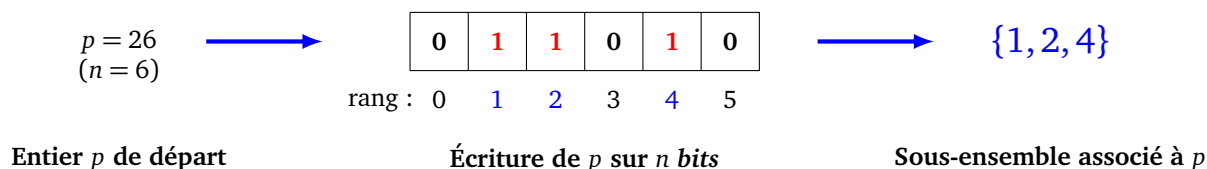
Activité 4 (Sous-ensembles).

Objectifs : générer tous les sous-ensembles afin de tester tous les triplets de sommets. Pour cela nous utiliserons l'écriture binaire.

Voici comment nous associons à chaque entier p vérifiant $0 \leq p < 2^n$ un sous-ensemble de $E_n = \{0, 1, \dots, n-1\}$.

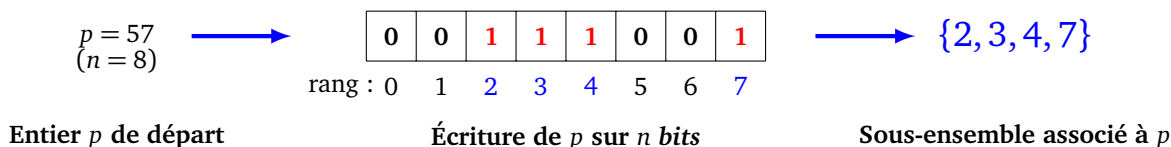
Commençons par un exemple, avec $n = 6$ et $p = 26$:

- l'écriture binaire de $p = 26$ sur $n = 6$ bits est $[0, 1, 1, 0, 1, 0]$;
- il y a des 1 au rang 1, 2 et 4 (en commençant au rang 0 à gauche) ;
- le sous-ensemble associé est alors $\{1, 2, 4\}$.



Autres exemples.

- Avec $n = 8$ et $p = 57$ dont l'écriture binaire sur 8 bits est $[0, 0, 1, 1, 1, 0, 0, 1]$, le sous-ensemble associé correspond aux rangs 2, 3, 4, 7, c'est donc $\{2, 3, 4, 7\}$.



- Avec $p = 0$, l'écriture binaire est formée uniquement de 0, le sous-ensemble associé est l'ensemble vide.
- Avec $p = 2^n - 1$, l'écriture binaire est formée uniquement de 1, le sous-ensemble associé est $E_n = \{0, 1, \dots, n-1\}$ tout entier.

Nous modélisons un ensemble comme une liste d'éléments. Par exemple :

- L'ensemble E_4 est pour nous la liste $[0, 1, 2, 3]$.
- Un sous-ensemble de E_4 est par exemple la paire $[1, 3]$.
- L'ensemble vide est représenté par la liste vide $[]$.

1. Programme la fonction `sous_ensembles(n)` qui renvoie la liste de tous les sous-ensembles possibles de $E_n = \{0, 1, 2, \dots, n-1\}$.

Par exemple, pour $n = 3$, `sous_ensembles(n)` renvoie la liste (qui contient elle-même des listes) :

`[[], [2], [1], [1, 2], [0], [0, 2], [0, 1], [0, 1, 2]]`

C'est-à-dire les 8 sous-ensembles (en commençant par l'ensemble vide) :

$\emptyset \quad \{2\} \quad \{1\} \quad \{1, 2\} \quad \{0\} \quad \{0, 2\} \quad \{0, 1\} \quad \{0, 1, 2\}.$

Indication. Pour tester ton programme, vérifie que la liste renvoyée contient bien 2^n sous-ensembles.

2. Dédus-en une fonction `sous_ensembles_fixe(n, k)` qui renvoie seulement les sous-ensembles de E_n ayant k éléments.

Par exemple, pour $n = 3$ et $k = 2$, `sous_ensembles_fixe(n, k)` renvoie la liste des paires :

`[[0, 1], [0, 2], [1, 2]]`

Teste ton programme :

- Pour $n = 4$ et $k = 3$, la liste renvoyée par `sous_ensembles_fixe(n, k)` contient 4 triplets.
- Pour $n = 5$ et $k = 3$, il y a 10 triplets possibles.
- Pour $n = 10$ et $k = 4$, il y a 210 sous-ensembles possibles !

Dans la suite nous utiliserons surtout les sous-ensembles à 3 éléments. En particulier, pour $n = 6$, les triplets inclus dans $\{0, 1, 2, 3, 4, 5\}$ sont au nombre de 20 :

`[[3, 4, 5], [2, 4, 5], [2, 3, 5], [2, 3, 4], [1, 4, 5],
[1, 3, 5], [1, 3, 4], [1, 2, 5], [1, 2, 4], [1, 2, 3],
[0, 4, 5], [0, 3, 5], [0, 3, 4], [0, 2, 5], [0, 2, 4],
[0, 2, 3], [0, 1, 5], [0, 1, 4], [0, 1, 3], [0, 1, 2]]`

Activité 5 (Théorème de Ramsey pour $n = 6$).

Objectifs : vérifier que tous les graphes ayant 6 sommets contiennent trois amis ou bien trois étrangers.

1. Programme une fonction `graphe_contient_3(graphe)` qui teste si un graphe contient 3 amis ou bien 3 étrangers. Il faut donc appeler les fonctions `contient_3_amis_fixes(graphe, i, j, k)` et `contient_3_etrangers_fixes(graphe, i, j, k)` de la première activité pour tous les triplets possibles de sommets (i, j, k) .

Pour les quatre exemples de la première activité, seul le quatrième (avec 6 sommets) vérifie le test.

2. Programme une fonction `voir_tous_graphes(n)` qui affiche tous les tableaux possibles de graphes à n sommets. Il y a $N = \frac{(n-1)n}{2}$ tableaux possibles. Tu peux les générer par une méthode similaire à celle pour les sous-ensembles :

- pour chaque entier p qui vérifie $0 \leq p < 2^N$,
- calcule l'écriture binaire de p sur N bits,
- remplis le tableau élément par élément, avec les 0 et les 1 de l'écriture binaire.

Indications. Pour remplir un tableau à partir d'une écriture binaire b donnée, tu peux utiliser une double boucle du type :

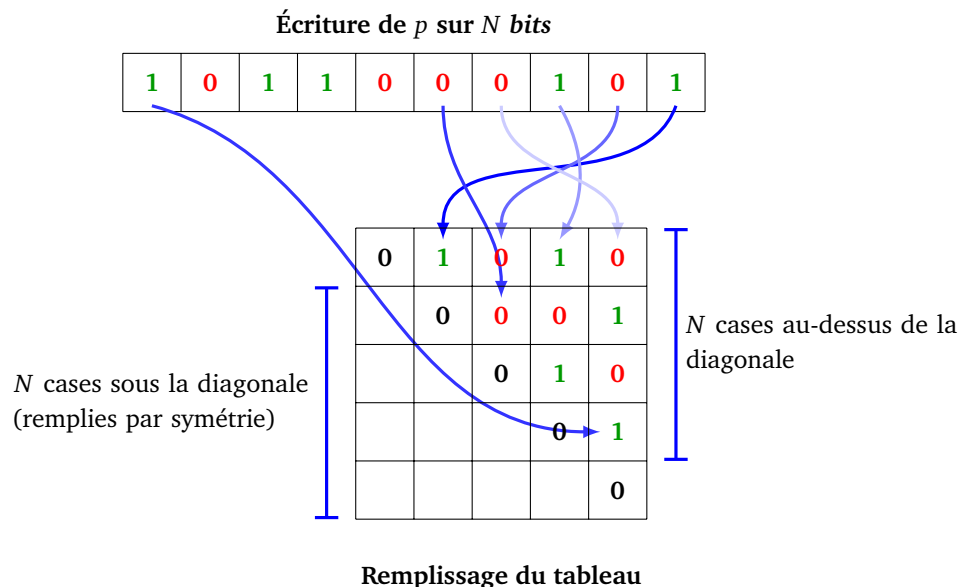
```
for j in range(0, n):
    for i in range(j+1, n):
```

```

b = liste_binaire.pop()
graphe[i][j] = b
graphe[j][i] = b

```

Voici le principe de cette boucle qui remplit la partie au-dessus de la diagonale (et aussi la partie en-dessous par symétrie). Cette boucle prend le dernier *bit* de la liste et le place sur la première case libre au-dessus de la diagonale ; puis l'avant-dernier *bit* est placé sur la seconde case libre... ; le premier *bit* de la liste remplit la dernière case libre.



3. Transforme la fonction précédente en une fonction `test_tous_graphes(n)` qui teste la conjecture « il y a trois amis ou trois étrangers » pour tous les graphes à n sommets. Tu dois trouver que :
 - pour $n = 4$ et $n = 5$ la conjecture est fausse. Donne un graphe à 4 sommets (puis à 5 sommets) qui n'a ni 3 amis, ni 3 étrangers ;
 - pour $n = 6$ laisse l'ordinateur vérifier que, pour chacun des $N = 2^{\frac{5 \times 6}{2}} = 32\,768$ graphes ayant 6 sommets, soit il possède 3 amis, soit il possède 3 étrangers.

Activité 6 (Pour aller plus loin).

Objectifs : améliorer ton programme et prouver d'autres conjectures. Activité facultative.

1. Améliore ton programme afin qu'il vérifie la conjecture pour $n = 6$ en moins d'une seconde.

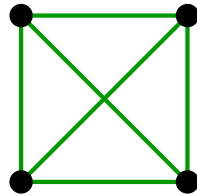
Idées.

- Il faut générer la liste des triplets une fois pour toute au début du programme (et non à chaque nouveau graphe).
- Il ne faut pas générer une liste de tous les graphes possibles, puis les tester dans un second temps. Il faut en générer un puis le tester avant de passer au suivant.
- Dès que tu as trouvé 3 amis (ou 3 étrangers) c'est gagné ! Stoppe immédiatement la boucle quitte à utiliser l'instruction `break` et passe au graphe suivant.
- Tu peux seulement tester les graphes qui correspondent à p entre 0 et $2^N/2$ (car pour les p suivants cela revient à échanger les segments verts en rouges et inversement).

Avec ces conseils voici les temps de calcul auxquels tu peux t'attendre :

Nombre de sommets	Nombre de graphes	Temps de calcul approximatif
$n = 6$	32 768	< 1 seconde
$n = 7$	2 097 152	< 1 minute
$n = 8$	268 435 456	< 1 heure
$n = 9$	68 719 476 736	< 10 jours

2. Il existe un énoncé plus difficile. Il s'agit de trouver à partir de quelle taille n un graphe contient toujours ou bien 4 amis ou bien 3 étrangers. Être 4 amis signifie que deux à deux ils sont reliés par un segment vert, comme ci-dessous :



- (a) Trouve des graphes à $n = 6$ (puis $n = 7$) sommets qui ne vérifient pas cet énoncé.
- (b) En cherchant un peu avec la machine trouve des graphes à 8 sommets qui ne vérifient pas cet énoncé.
- (c) Prouve que n'importe quel graphe ayant 9 sommets contient 4 amis ou bien 3 étrangers !
- Indications.* Il faut tester tous les graphes correspondants aux entiers p compris entre 0 et $2^N = 2^{\frac{8 \times 9}{2}} = 68\,719\,476\,736$. Le temps total de calcul est d'environ 20 jours ! Tu peux partager les calculs entre plusieurs ordinateurs : un ordinateur fait les calculs pour $0 \leq p \leq 1\,000\,000$, un deuxième ordinateur pour $1\,000\,001 \leq p \leq 2\,000\,000$,...
3. • Il existe des raisonnements pour pouvoir démontrer à la main que pour $n = 6$ il y a toujours 3 amis ou 3 étrangers. Cherche un tel raisonnement ! Avec un peu plus d'efforts, on prouve aussi que c'est $n = 9$ qui répond au problème des 4 amis/3 étrangers.
- On sait prouver qu'il faut $n = 18$ sommets pour avoir toujours 4 amis ou 4 étrangers.
- Par contre personne dans le monde ne sait quelle est la valeur du plus petit n pour le problème des 5 amis/5 étrangers !

Le bitcoin est une monnaie dématérialisée et décentralisée. Elle repose sur deux principes informatiques : la cryptographie à clé publique et la preuve de travail. Pour comprendre ce second principe, tu vas créer un modèle simple de bitcoin.

Activité 1 (Preuve de travail).

Objectifs : comprendre ce qu'est une preuve de travail sur un modèle simple. Cette activité est indépendante du reste de la fiche. L'idée est de trouver un problème difficile à résoudre mais facile à vérifier. Comme les sudokus par exemple : il suffit de dix secondes pour vérifier qu'une grille est remplie correctement, par contre il a fallu plus de dix minutes pour le résoudre.

Le problème mathématique est le suivant : on te donne un nombre premier p et un entier y ; tu dois trouver un entier x tel que :

$$x^2 = y \pmod{p}$$

Autrement dit x est une racine carrée de y modulo p . Attention, il n'existe pas toujours de solution pour x .

Exemples.

- Pour $p = 13$ et $y = 10$, alors une solution est $x = 6$. Effet $x^2 = 6^2 = 36$. Et $36 = 2 \times 13 + 10$ donc $x^2 = 10 \pmod{13}$.
- La solution n'est pas forcément unique. Par exemple, vérifie que $x = 7$ est aussi solution.
- Il n'y a pas toujours de solution, par exemple pour $p = 13$ et $y = 5$, aucun entier x n'est solution.
- Exercice : pour $p = 13$, trouve à la main deux solutions x au problème $x^2 = 9 \pmod{13}$; trouve une solution x au problème $x^2 = 3 \pmod{13}$.

L'entier p est fixé pour toute l'activité. Pour les exemples faciles on prendra $p = 101$, pour les exemples moyens $p = 15\,486\,869$ et pour les exemples difficiles $p = 2\,276\,856\,017$. Plus l'entier p est grand, plus la preuve de travail est difficile à obtenir.

1. **Vérification (facile).** Écris une fonction `verification(x, y)` qui renvoie « vrai » si x est bien solution du problème $x^2 = y \pmod{p}$ et « faux » sinon.
Vérifie que $x = 6\,543\,210$ est solution lorsque $y = 8\,371\,779$ et $p = 15\,486\,869$. Affiche le temps de calcul nécessaire à cette vérification. (Voir le cours plus bas.)
2. **Recherche de solution (difficile).** Pour trouver une solution x , il n'y a pas vraiment d'autres choix pour nous que de tester tous les x en commençant par $x = 0$, $x = 1 \dots$. Programme une fonction `racine(y)` qui renvoie une solution x du problème pour y donné (ou `None` s'il n'y a pas de solution).
 - Pour $p = 101$ et $y = 17$, trouve x tel que $x^2 = y \pmod{p}$.
 - Pour $p = 15\,486\,869$ et $y = 8\,371\,779$, tu dois retrouver le x de la première question. Combien de temps a pris la recherche ?
 - Pour $p = 15\,486\,869$ et $y = 13\,017\,204$, trouve x tel que $x^2 = y \pmod{p}$.

Conclusion : nous avons trouvé un problème difficile à résoudre, mais pour lequel il est facile de vérifier qu'une solution donnée convient. Pour des valeurs de p plus grandes, la recherche d'une solution x peut être beaucoup trop longue et ne pas aboutir. Nous allons voir comment on peut ajuster la difficulté du problème.

3. Au lieu de chercher une solution exacte à notre problème $x^2 = y \pmod{p}$, qui est équivalent à $x^2 - y \pmod{p} = 0$. On cherche une solution approchée, c'est-à-dire qui vérifie :

$$x^2 - y \pmod{p} \leq m.$$

Par exemple si $m = 5$, alors on peut avoir (modulo p) : $x^2 - y = 0$, $x^2 - y = 1$, $x^2 - y = 2, \dots$, $x^2 - y = 5$.

Programme une fonction `racine_approchee(y, marge)` qui trouve une solution approchée à notre problème $x^2 = y \pmod{p}$.

Combien de temps faut-il pour trouver un solution au problème approché lorsque $p = 15\,486\,869$, $y = 8\,371\,779$ et $m = 20$? Choisis un nombre premier p assez grand et trouve une marge d'erreur m de sorte que trouver une solution au problème approché nécessite environ entre 30 et 60 secondes de calculs (pour un y quelconque).

Voici des exemples de nombres premiers que tu peux utiliser pour tes essais :

101 1097 10651 100109 1300573 15486869 179426321 2276856017 32416187567

Cours 1 (Chronomètre).

Le module `time` permet de chronométrer le temps d'exécution mais aussi de savoir la date et l'heure (voir aussi le module `timeit` expliqué dans la fiche « Arithmétique – Tant que – I »). Voici un petit script pour mesurer le temps de calcul d'une instruction.

```
import time

debut_chrono = time.time()

time.sleep(2)

fin_chrono = time.time()

total_chrono = fin_chrono - debut_chrono
print("Temps d'exécution (en secondes) :", total_chrono)
```

Explications.

- Le module s'appelle `time`.
- La fonction qui renvoie le temps s'appelle aussi `time()`, on l'appelle donc par `time.time()` !
- La fonction `time()` renvoie un nombre qui ne nous intéresse pas. Ce qui nous intéresse c'est l'écart entre deux valeurs de cette fonction.
- `debut_chrono = time.time()` c'est comme lancer un chronomètre.
- `fin_chrono = time.time()` c'est comme l'arrêter.
- `total_chrono = fin_chrono - debut_chrono` est la durée totale qu'a pris notre script en secondes.
- La fonction `time.sleep(duree)` fait une pause d'une certaine durée de secondes dans le programme.

Cours 2 (Bitcoin et blockchain).

La monnaie *bitcoin* est une monnaie dématérialisée. Les transactions sont enregistrées dans un grand livre de compte appelé *blockchain*. Nous allons construire un modèle (très simplifié) d'un tel livre de compte.

Imaginons un groupe d'amis qui souhaitent partager les dépenses du groupe de façon la plus simple possible. Au départ tout le monde dispose de 1000 *bitcoins* et on note au fur et à mesure les dépenses et les recettes de chacun.

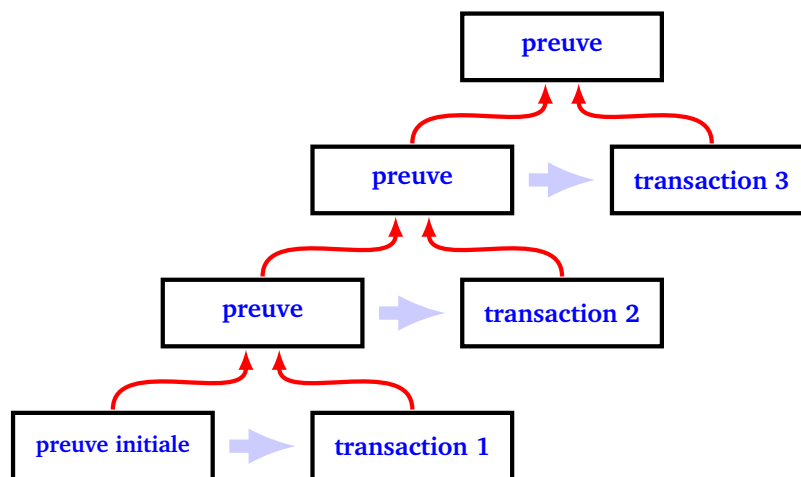
On note sur le livre de compte la liste des dépenses/recettes, par exemple :

- « Amir a dépensé 100 *bitcoins* »
- « Barbara a reçu 45 *bitcoins* »
- etc.

Il suffit de parcourir tout le livre pour savoir combien chacun a reçu ou dépensé depuis le début.

Pour éviter que quelqu'un ne vienne truquer le livre de compte, après chaque transaction on ajoute dans le livre une certification construite à partir d'une preuve de travail. Voici ce que l'on écrit dans le livre :

1. On commence par une preuve de travail quelconque. Pour nous ce sera $[0, 0, 0, 0, 0, 0]$.
2. On écrit la première transaction (par exemple "Amir -100").
3. On calcule et on écrit dans le livre une preuve de travail, qui va servir de certificat. C'est une liste (par exemple $[56, 42, 10, 98, 2, 34]$) obtenue après beaucoup de calculs prenant en compte la transaction précédente et la précédente preuve de travail.
4. À chaque nouvelle transaction (par exemple "Barbara +45"), quelqu'un calcule une preuve de travail pour la dernière transaction associée à la précédente preuve. On écrit la transaction, puis la preuve de travail.



La preuve de travail que l'on calcule dépend de la transaction précédente mais aussi de la preuve de travail précédente, qui elle-même dépend des données antérieures... Ainsi, chaque nouvelle preuve de travail dépend en fait de tout ce qui a été écrit depuis le début (même si le calcul ne porte explicitement que sur les deux dernières entrées).

Quelqu'un qui voudrait truquer une transaction passée devrait recalculer toutes les preuves de travail qui viennent après. Ceci n'est pas possible pour une personne seule : il y a plusieurs preuves de travail à calculer et chaque preuve demande beaucoup de calculs.

Activité 2 (Outils pour les listes).

Objectifs : construire des fonctions utiles pour manipuler les listes d'entiers dans les activités suivantes.

Nos listes sont des listes d'entiers compris entre 0 et 99.

1. Programme une fonction `addition(liste1, liste2)` qui additionne terme à terme et modulo 100, les éléments de deux listes de même longueur. Par exemple `addition([1, 2, 3, 4, 5, 6], [1, 1, 1, 98, 98, 98])` renvoie `[2, 3, 4, 2, 3, 4]`.

2. On va chercher des listes qui commencent par des zéros (ou bien des zéros puis des nombres assez petits). Une liste `liste` est plus petite que la liste `liste_max` si chaque élément de `liste` est inférieur ou égal à chaque élément de même rang de `liste_max`.

Par exemple la liste `[0, 0, 1, 2, 3, 4]` est plus petite que la liste `[0, 0, 5]`. Ce n'est pas le cas de la liste `[0, 10, 0, 1, 1]`. Autre exemple : être plus petit que la liste `[0, 0, 0]` cela signifie commencer par trois zéros. Être plus petit que la liste `[0, 0, 1]` signifie commencer par `[0, 0, 0]` ou `[0, 0, 1]`. Programme une fonction `est_plus_petit(liste, liste_max)` qui renvoie « vrai » lorsque `liste` est plus petite que `liste_max`.

3. On aura besoin de transformer une phrase en une liste de nombres. De plus, on va découper nos listes en blocs de taille N (avec $N = 6$), on rajoute donc des zéros en début de liste afin qu'elle soit de longueur un multiple de N .

Écris une fonction `phrase_vers_liste(phrase)` qui convertit une chaîne de caractères en une liste d'entiers entre 0 et 99 et si besoin rajoute des zéros devant afin que la liste ait la bonne taille.

La formule à utiliser pour convertir un caractère en un entier strictement inférieur à 100 est :

$$\text{ord}(c) \% 100$$

Par exemple : si `phrase = "Vive moi !"` alors la fonction renvoie :

`[0, 0, 86, 5, 18, 1, 32, 9, 11, 5, 32, 33]`

Le caractère "i" à pour code ASCII/unicode 105 donc, modulo 100, le nombre est 5. Note que la fonction ajoute deux 0 en début de liste afin d'avoir une longueur qui est un multiple de $N = 6$.

Activité 3 (Fonction de hachage).

Objectifs : créer une fonction de hachage. À partir d'un long message nous calculons une courte empreinte. Il est difficile de trouver deux messages différents ayant la même empreinte.

Dans cette activité, notre message est une liste d'entiers (entre 0 et 99) de longueur un multiple quelconque de $N = 6$, nous le transformons en une liste de longueur $N = 6$: son empreinte (ou *hash*). Voici des exemples de ce que va faire notre fonction de hachage :

- la liste `[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]` a pour empreinte :
`[10, 0, 58, 28, 0, 90]`
- la liste `[1, 1, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]` a pour empreinte :
`[25, 14, 29, 1, 19, 6]`

L'idée est de mélanger les nombres par bloc de $N = 6$ entiers, puis de combiner ce bloc au suivant et de recommencer, jusqu'à obtenir un seul bloc.

1. Un tour.

Pour un bloc `[$b_0, b_1, b_2, b_3, b_4, b_5$]` de taille $N = 6$, faire un tour consiste à faire les opérations suivantes :

- (a) On additionne certains entiers :

$$[b'_0, b'_1, b'_2, b'_3, b'_4, b'_5] = [b_0, b_1 + b_0, b_2, b_3 + b_2, b_4, b_5 + b_4]$$

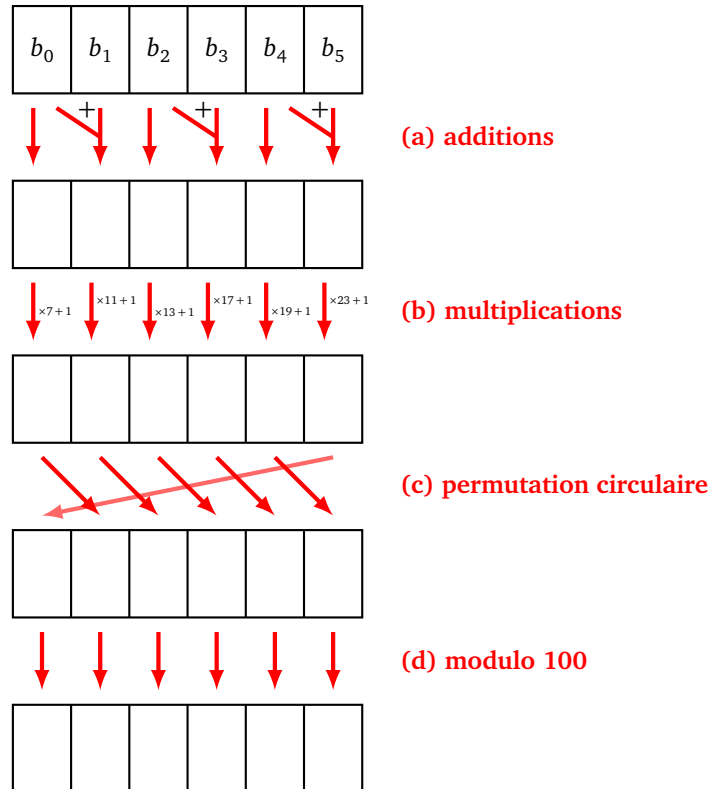
(b) On multiplie ces entiers par des nombres premiers (dans l'ordre 7, 11, 13, 17, 19, 23) et on rajoute 1 :

$$[b_0'', b_1'', b_2'', b_3'', b_4'', b_5''] = [7 \times b_0' + 1, 11 \times b_1' + 1, 13 \times b_2' + 1, 17 \times b_3' + 1, 19 \times b_4' + 1, 23 \times b_5' + 1]$$

(c) On effectue une permutation circulaire (le dernier passe devant) :

$$[b_0''', b_1''', b_2''', b_3''', b_4''', b_5'''] = [b_5'', b_0'', b_1'', b_2'', b_3'', b_4'']$$

(d) On réduit chaque entier modulo 100 afin d'obtenir des entiers entre 0 et 99.



Partant du bloc $[0, 1, 2, 3, 4, 5]$, on a donc successivement :

(a) additions : $[0, 1, 2, 5, 4, 9]$

(b) multiplications : $[7 \times 0 + 1, 11 \times 1 + 1, 13 \times 2 + 1, 17 \times 5 + 1, 19 \times 4 + 1, 23 \times 9 + 1] = [1, 12, 27, 86, 77, 208]$

(c) permutation : $[208, 1, 12, 27, 86, 77]$

(d) réduction modulo 100 : $[8, 1, 12, 27, 86, 77]$

Programme une telle fonction `un_tour(bloc)` qui renvoie la transformation du bloc après ces opérations. Vérifie que le bloc $[1, 1, 2, 3, 4, 5]$ est transformé en $[8, 8, 23, 27, 86, 77]$.

2. Dix tours.

Pour bien mélanger chaque bloc, programme une fonction `dix_tours(bloc)` qui itère dix fois les opérations précédentes. Après 10 tours :

- le bloc $[0, 1, 2, 3, 4, 5]$ devient $[98, 95, 86, 55, 66, 75]$,
- le bloc $[1, 1, 2, 3, 4, 5]$ devient $[18, 74, 4, 42, 77, 42]$.

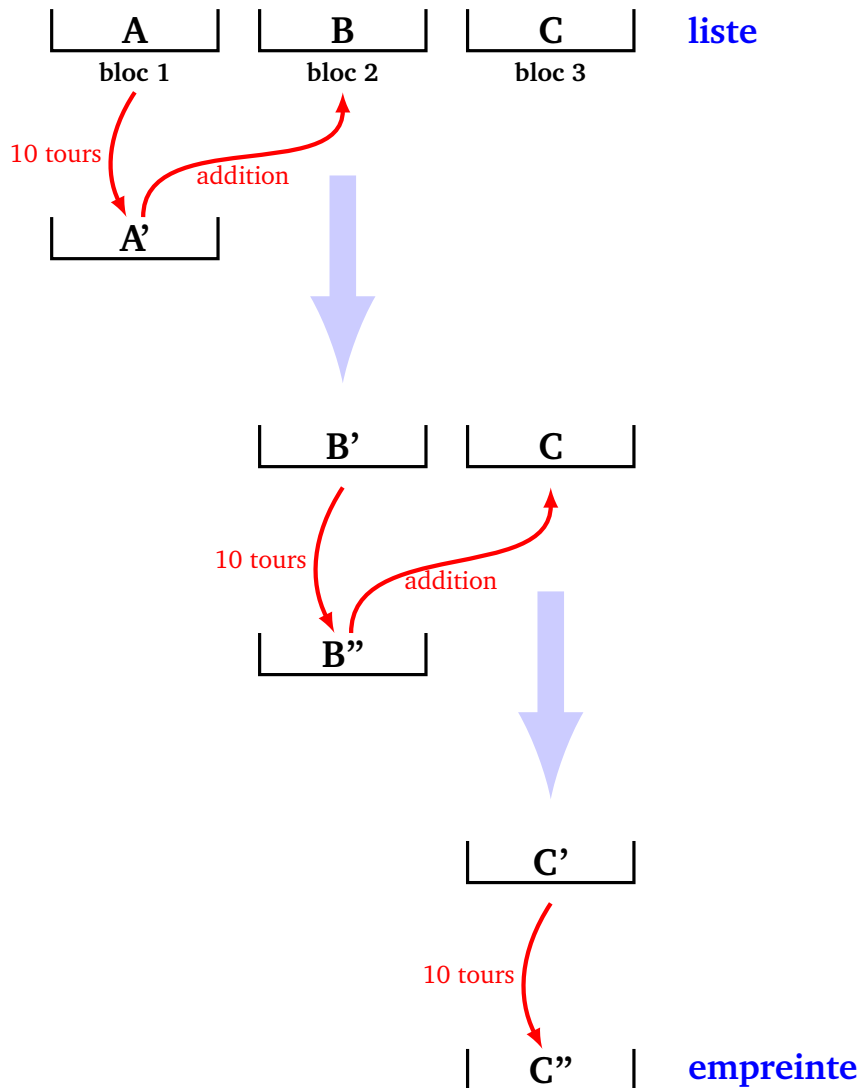
Deux blocs proches sont transformés en deux blocs très différents !

3. Hachage d'une liste.

Partant d'une liste de longueur un multiple de $N = 6$, on la découpe en blocs de longueur 6 et on calcule l'empreinte de cette liste selon l'algorithme suivant :

- On extrait le premier bloc de la liste, on effectue 10 tours de mélange.
- On ajoute terme à terme (et modulo 100), le résultat de ce mélange au second bloc.
- On recommence en partant du nouveau second bloc.
- Lorsqu'il ne reste plus qu'un bloc, on effectue 10 tours de mélange, le résultat est l'empreinte de la liste.

Voici le schéma d'une situation avec trois blocs : dans un premier temps il y a trois blocs (A,B,C) ; dans un second temps il ne reste plus que deux blocs (B' et C) ; à la fin il ne reste qu'un bloc (C'') : c'est l'empreinte !



Exemple avec la liste [0, 1, 2, 3, 4, 5, 1, 1, 1, 1, 1, 10, 10, 10, 10, 10, 10].

- Le premier bloc est [0, 1, 2, 3, 4, 5], son mélange à 10 tours est [98, 95, 86, 55, 66, 75].
- On ajoute ce mélange au second bloc [1, 1, 1, 1, 1, 1] (voir la fonction `addition()` de l'activité 2).
- La liste restante est maintenant [99, 96, 87, 56, 67, 76, 10, 10, 10, 10, 10, 10].
- On recommence. Le nouveau premier bloc est [99, 96, 87, 56, 67, 76], son mélange à 10 tours vaut [60, 82, 12, 94, 6, 80], on l'ajoute au dernier bloc [10, 10, 10, 10, 10, 10] pour obtenir (modulo 100) [70, 92, 22, 4, 16, 90].
- On effectue un dernier mélange à 10 tours pour obtenir l'empreinte : [77, 91, 5, 91, 89, 99].

Programme une fonction `hachage(liste)` qui renvoie l'empreinte d'une liste. Teste-la sur les exemples donnés au début de l'activité.

Activité 4 (Preuve de travail - Minage).

Objectifs : construire un mécanisme de preuve de travail à l'aide de notre fonction de hachage.

On va construire un problème compliqué à résoudre, pour lequel, si quelqu'un nous donne la solution, alors il est facile de vérifier qu'elle convient.

Problème à résoudre. On nous donne une liste, il s'agit de trouver un bloc tel que, lorsque qu'on le rajoute à la liste, cela produit un hachage commençant par des zéros. Plus précisément étant donné une liste `liste` et un objectif maximal `Max`, il s'agit de trouver un bloc preuve qui, concaténé à la liste puis haché, est plus petit que la liste `Max`, c'est-à-dire :

`hachage(liste + preuve) plus petit que Max`

La liste est de longueur quelconque (un multiple de $N = 6$), la preuve est un bloc de longueur N , l'objectif est de trouver une liste commençant par des 0 (voir l'activité 2).

Par exemple : soit la `liste = [0,1,2,3,4,5]` et `Max = [0,0,7]`. Quel bloc preuve puis-je concaténer à `liste` pour résoudre mon problème ?

- `preuve = [12,3,24,72,47,77]` convient car concaténé à notre liste cela donne `[0,1,2,3,4,5,12,3,24,72,47,77]` et le hachage de toute cette liste vaut `[0,0,5,47,44,71]` qui commence par `[0,0,5]` plus petit que l'objectif.
- `preuve = [0,0,2,0,61,2]` convient aussi car après concaténation on a `[0,1,2,3,4,5,0,0,2,0,61,2]` dont le hachage donne `[0,0,3,12,58,92]`.
- `[97,49,93,87,89,47]` ne convient pas, car après concaténation puis hachage on obtient `[0,0,8,28,6,60]` qui est plus grand que l'objectif voulu.

1. Vérification (facile).

Programme une fonction `verification_preuve_de_travail(liste,preuve)` qui renvoie vrai si la solution `preuve` proposée convient pour `liste`. Utilise la fonction `est_plus_petit()` de l'activité 2.

2. Recherche de solution (difficile).

Programme une fonction `preuve_de_travail(liste)` qui cherche un bloc preuve solution à notre problème pour la liste donnée.

Indications.

- La méthode la plus simple est de prendre un bloc `preuve` de nombres au hasard et de recommencer jusqu'à trouver une solution.
- Tu peux aussi tester systématiquement tous les blocs en commençant avec `[0,0,0,0,0,0]`, puis `[0,0,0,0,0,1]`... et t'arrêter au premier qui convient.
- Tu ajustes la difficulté du problème en changeant l'objectif : facile avec `Max = [0,0,50]`, moyen avec `Max = [0,0,5]`, difficile avec `Max = [0,0,0]`, trop difficile avec `Max = [0,0,0,0]`.
- Comme il existe plusieurs solutions, tu n'obtiens pas nécessairement la même solution à chaque recherche.

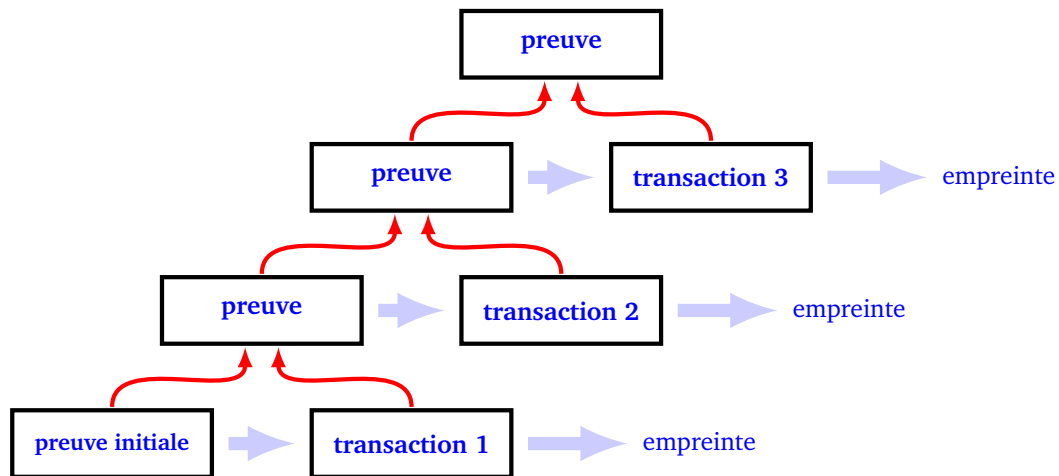
3. Temps de calcul.

Compare le temps de calcul d'une simple vérification par rapport au temps de recherche d'une solution. Choisis l'objectif `Max` de sorte que la recherche d'une preuve de travail nécessite environ entre 30 et 60 secondes de calculs.

Pour le *bitcoin*, ceux qui calculent des preuves de travail sont appelés les *mineurs*. Le premier qui trouve une preuve gagne une récompense. La difficulté du problème est ajustée de sorte que le temps de calcul mis par le gagnant (parmi l'ensemble de tous les mineurs) pour trouver une solution, soit d'environ 10 minutes.

Activité 5 (Tes bitcoins).

Objectifs : créer un livre de compte (appelé blockchain pour le bitcoin) qui enregistre toutes les transactions, ce registre est public et certifié. Il est pratiquement impossible d'y falsifier une transaction déjà inscrite.

**1. Initialisation et ajout d'une transaction.**

(a) Initialise une variable globale `Livre` qui est une liste et contient au départ une preuve nulle :
`Livre = [[0,0,0,0,0,0]]`.

(b) Une *transaction* est une chaîne de caractères comprenant un nom et la somme à ajouter (ou à retrancher) à son compte. Par exemple "Abel +25" ou "Barbara -45".

Programme une fonction `ajout_transaction(transaction)` qui ajoute la chaîne de caractères `transaction` à la liste `Livre`. Par exemple après l'initialisation `ajout_transaction("Camille +100")`, `Livre` vaut `[[0,0,0,0,0,0], "Camille +100"]`. Attention, pour pouvoir modifier `Livre` il faut commencer la fonction par : `global Livre`.

2. Dès qu'une transaction est ajoutée, il faut calculer et ajouter au livre de compte une preuve de travail. Programme une fonction `minage()`, sans paramètre, qui ajoute une preuve de travail au livre.

Voici comment faire :

- On prend la dernière transaction `transaction`, on la transforme en une liste d'entiers par la fonction `phrase_vers_liste()` de l'activité 2.
- On prend la preuve de travail `prec_preuve` située juste avant cette transaction.
- On forme la liste `liste` composée d'abord des éléments de `prec_preuve`, puis des éléments de la liste d'entiers obtenue en convertissant la chaîne `transaction`.
- On calcule une preuve de travail de cette liste.
- On ajoute cette preuve au livre de compte.

Par exemple si le livre se termine par :

`[3,1,4,1,5,9], "Abel +35"`

alors après calcul de la preuve de travail le livre se termine par exemple par :

`[3,1,4,1,5,9], "Abel +35", [32,17,37,73,52,90]`

On rappelle que la preuve de travail n'est pas unique et qu'en plus elle dépend de l'objectif `Max`.

3. Une seule personne à la fois ajoute une preuve de travail. Par contre tout le monde a la possibilité de vérifier que la preuve proposée est correcte (et devrait le faire). Écris une fonction `verification_livre()`, sans paramètre, qui vérifie que la dernière preuve ajoutée au `Livre` est valide.

4. Écris un livre de compte qui correspond aux données suivantes :

- On prend $\text{Max} = [0, 0, 5]$ et au départ $\text{Livre} = [[0, 0, 0, 0, 0, 0]]$.
- "Alfred -100" (Alfred doit 100 *bitcoins*).
- Barnabé en reçoit 150.
- Chloé gagne 35 *bitcoins*.

Conclusion : imaginons que Alfred veuille tricher, il veut changer le livre de compte afin de recevoir 100 *bitcoins* au lieu d'en devoir 100. Il doit donc changer la transaction le concernant en "Alfred +100" mais il doit alors recalculer une nouvelle preuve de travail ce qui est compliqué, surtout il doit aussi recalculer la preuve de la transaction de Barnabé et aussi celle de la transaction de Chloé !

Quelqu'un qui veut modifier une transaction doit modifier toutes les preuves de travail suivantes. Si chaque preuve demande suffisamment de temps de calcul ceci est impossible. Pour le *bitcoin* chaque preuve demande énormément de calculs (trop pour une personne seule) et une nouvelle preuve est à calculer toutes les 10 minutes. Il est donc impossible pour une personne de modifier une transaction passée.

L'autre aspect du *bitcoin* que nous n'avons pas abordé, c'est de s'assurer de l'identité de chaque personne impliquée, afin que personne ne puisse récupérer l'argent d'un autre. Ceci est rendu possible grâce à la cryptographie à clé privée/clé publique (système RSA). Chaque compte est identifié par une clé publique (deux très grands entiers), ce qui garantit l'anonymat. Mais surtout, seul celui qui possède la clé privée du compte (un grand entier) peut accéder à ses *bitcoins*.

Constructions aléatoires

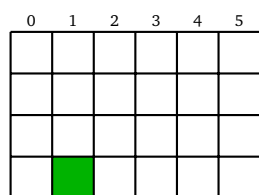
Tu vas programmer deux méthodes pour construire des figures qui ressemblent à des algues ou des coraux. Chaque figure est formée de petits blocs lancés au hasard et qui se collent les uns aux autres.



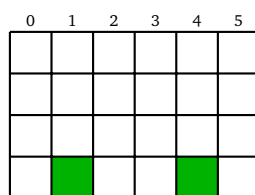
Cours 1 (Chutes de blocs).

On fait tomber des blocs carrés dans une grille, sur le principe du jeu « Puissance 4 » : après avoir choisi une colonne, un bloc tombe du haut vers le bas. Les blocs se posent sur le bas de la grille ou sur des autres blocs ou à côté d'autres blocs. Il y a une grosse différence avec le jeu « Puissance 4 », ici les blocs sont « collants », c'est-à-dire qu'un bloc reste collé dès qu'il rencontre un voisin à gauche ou à droite.

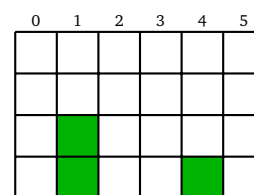
Voici un exemple de lancer de blocs :



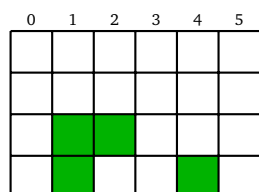
1. Bloc lancé en col. 1



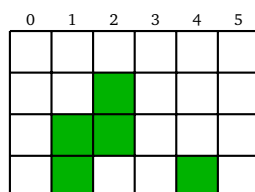
2. Bloc lancé en col. 4



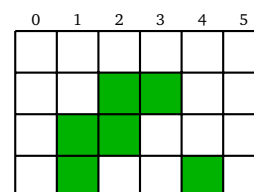
3. Bloc lancé en col. 1



4. Bloc lancé en col. 2



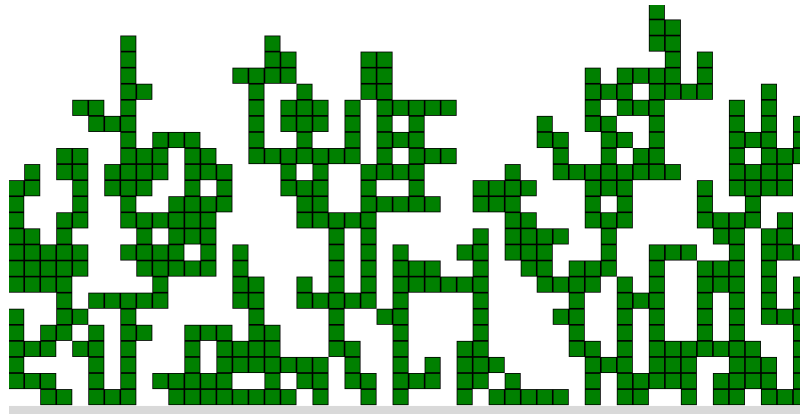
5. Bloc lancé en col. 2



6. Bloc lancé en col. 3

Par exemple à l'étape 4, le bloc lancé dans la colonne numéro 2 ne descend pas jusqu'en bas mais reste « scotché » à son voisin, il se retrouve donc définitivement suspendu.

Le lancer aléatoire de centaines de blocs sur une grande grille produit de jolies formes géométriques ressemblant à des algues.



Activité 1 (Chutes de blocs).

Objectifs : programmer la chute des blocs (sans affichage graphique).

On modélise l'espace de travail par un tableau de n lignes et p colonnes. Au départ le tableau ne contient que des 0 ; ensuite la présence d'un bloc est représentée par 1.

Voici comment initialiser le tableau :

```
tableau = [[0 for j in range(p)] for i in range(n)]
```

On modifie le tableau par des instructions du type :

```
tableau[i][j] = 1
```

Voici un exemple de tableau (à gauche) pour représenter la situation graphique de droite (le bloc en haut à droite est en train de tomber).

$$\begin{array}{cccccc}
 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 0
 \end{array}$$

indice j

indice i

$j=0 \quad j=1 \quad j=2 \quad j=3 \quad j=4 \quad j=5$

$i=0$
 $i=1$
 $i=2$
 $i=3$

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=0$						
$i=1$						
$i=2$						
$i=3$						

Un tableau avec 5 blocs ($n = 4, p = 6$)

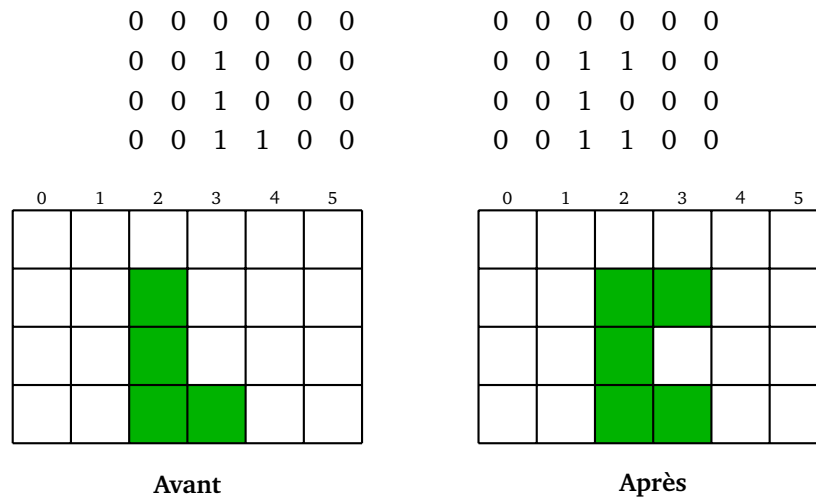
1. Programme une fonction `peut_tomber(i, j)` qui détermine si le bloc en position (i, j) peut descendre d'une case ou pas.

Voici les cas dans lesquels le bloc *ne peut pas* tomber :

- si le bloc est déjà sur la dernière ligne,
- s'il y a un bloc juste en dessous,
- s'il y a un bloc juste à droite ou juste à gauche.

2. Programme une fonction `faire_tomber_un_bloc(j)` qui fait tomber un bloc dans la colonne j jusqu'à ce qu'il ne puisse plus descendre. Cette fonction modifie les entrées du tableau.

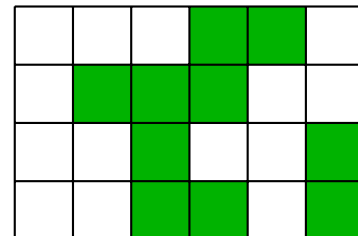
Par exemple, voici le tableau avant (à gauche) et après (à droite) avoir fait tomber un bloc dans la colonne $j = 3$.



3. Programme une fonction `faire_tomber_des_blocs(k)` qui lance k blocs un par un, en choisissant à chaque fois une colonne au hasard (c'est-à-dire un entier j avec $0 \leq j < p$).

Voici un exemple de tableau obtenu après avoir lancé 10 blocs :

0	0	0	1	1	0
0	1	1	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1

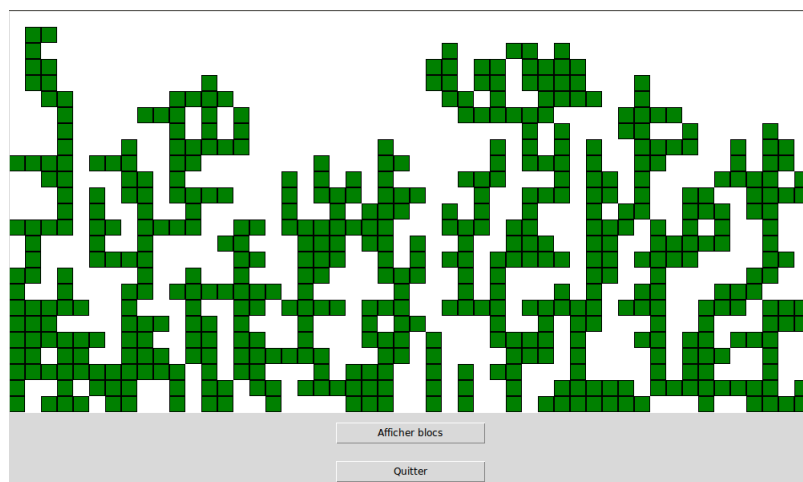


Lancer de 10 blocs

Activité 2 (Chutes de blocs (suite)).

Objectifs : programmer l'affichage graphique des blocs.

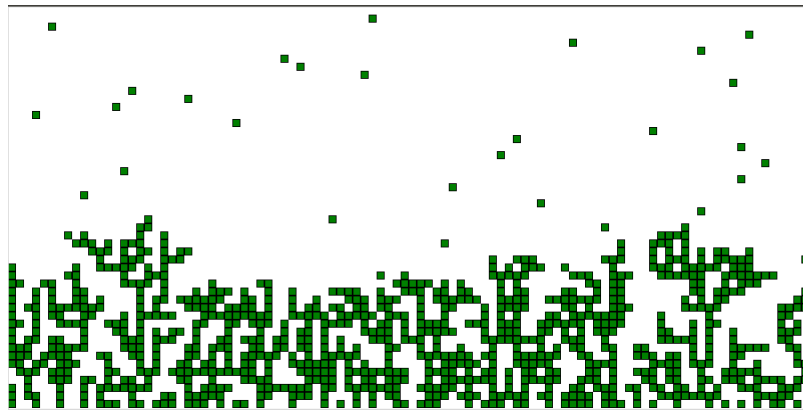
Affichage statique. Programme l'affichage graphique des blocs à partir d'un tableau.



Indications.

- Utilise le module `tkinter`, voir la fiche « Statistique – Visualisation de données ».
- Tu peux rajouter un bouton qui lance un bloc (ou plusieurs d'un coup).

Affichage dynamique (facultatif et difficile). Programme l'affichage des blocs qui tombent.

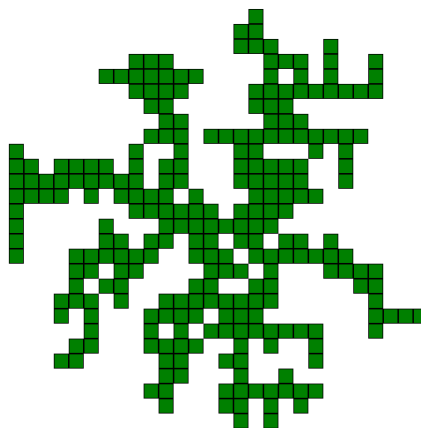


Indications.

- C'est beaucoup plus compliqué à programmer, mais très joli à voir !
- Pour le déplacement des blocs, inspire-toi du programme « Mouvement avec tkinter » à la fin de cette fiche.
- Pour faire une « pluie de blocs » de façon régulière (par exemple tous les dixièmes de secondes) : on fait descendre tous les blocs existant d'une case et on en fait apparaître un nouveau sur la ligne du haut.

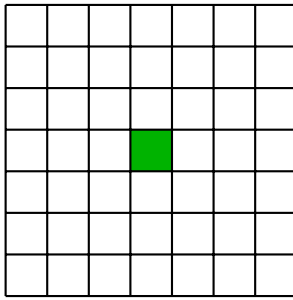
Cours 2 (Arbres browniens).

Voici une construction un peu différente, beaucoup plus longue à calculer, mais qui dessine aussi de jolies figures appelées « arbres browniens ».

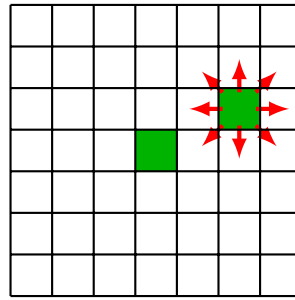


Le principe est le suivant :

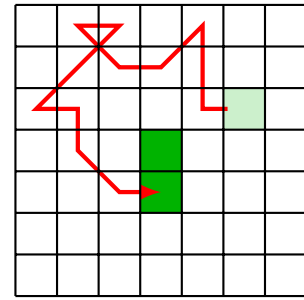
- On part d'une grille (il faut cette fois imaginer qu'elle est dessinée à plat sur une table). En son centre, on place un premier bloc fixe, le *germe*.
- On fait apparaître un bloc au hasard sur la grille. À chaque étape, ce bloc se déplace au hasard sur l'une des huit cases adjacentes, on parle d'un *mouvement brownien*.
- Dès que ce bloc touche un autre bloc par un côté, il s'y colle et ne bouge plus.
- Si le bloc sort de la grille, il se désintègre.
- Une fois le bloc collé ou désintégré, on relance alors un nouveau bloc depuis un point aléatoire de la grille.



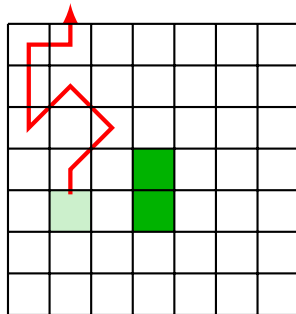
Le germe



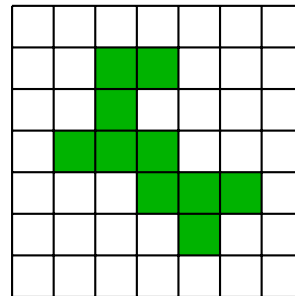
Un bloc et ses 8 mouvements possibles



Le mouvement aléatoire du bloc



Un bloc qui quitte la grille



10 blocs

On obtient petit à petit une sorte d'arbre qui ressemble à du corail. Les calculs sont très longs car beaucoup de blocs sortent de la grille ou mettent longtemps avant de se fixer (surtout au début). En plus, on ne peut lancer les blocs qu'un par un.

Activité 3 (Arbres browniens).

Objectifs : programmer la création d'un arbre brownien.

Première partie.

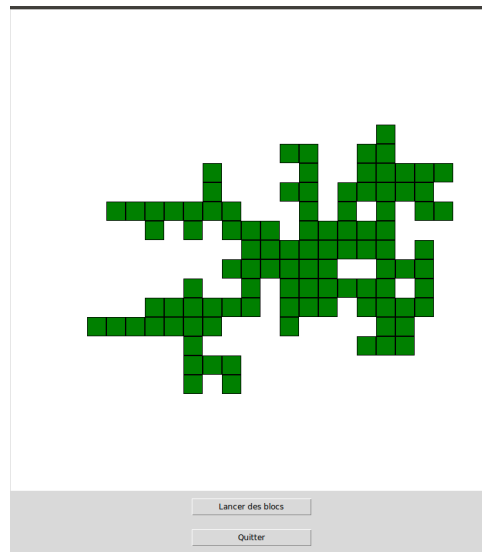
1. Modélise de nouveau l'espace de travail par un tableau de n lignes et p colonnes contenant des 0 ou des 1. Initialise toutes les valeurs à 0 sauf 1 au centre du tableau.
2. Programme une fonction `est_dedans(i, j)` qui détermine si la position (i, j) est bien dans la grille (sinon c'est que le bloc est en train de sortir).
3. Programme une fonction `est_libre(i, j)` qui détermine si le bloc en position (i, j) peut bouger (la fonction renvoie « vrai ») ou s'il est collé (la fonction renvoie « faux »).
4. Programme une fonction `lancer_un_bloc()`, sans paramètre, qui simule la création d'un bloc et son déplacement aléatoire, jusqu'à ce qu'il se colle ou qu'il quitte la grille.

Indications.

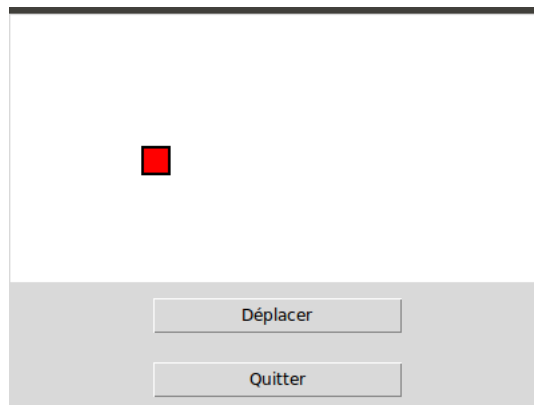
- Le bloc est créé à une position aléatoire (i, j) de la grille.
 - Tant que le bloc est dans la grille et libre de bouger :
 - tu choisis un déplacement horizontal en tirant au hasard un entier parmi $\{-1, 0, +1\}$, idem pour un déplacement vertical ;
 - tu déplaces le bloc selon la combinaison de ces deux mouvements.
 - Modifie alors le tableau.
5. Termine avec une fonction `lancer_des_blocs(k)` qui lance k blocs.

Seconde partie.

Programme l'affichage graphique à l'aide de `tkinter`. Tu peux ajouter un bouton qui lance 10 blocs d'un coup.

**Cours 3** (Mouvement avec « `tkinter` »).

Voici un programme qui fait se déplacer un petit carré et le faisant rebondir sur les bords de la fenêtre.



Voici les points principaux :

- Un objet `rect` est défini, c'est une variable globale, de même que ses coordonnées `x0`, `y0`.
- Cet objet est (un petit peu) déplacé par la fonction `deplacer()` qui décale le rectangle de `(dx, dy)`.
- Le point clé est que cette fonction sera exécutée une nouvelle fois après un court laps de temps. La commande :
`canvas.after(50,deplacer)`
demande une nouvelle exécution de la fonction `deplacer()` après un court délai (ici 50 millisecondes).
- La répétition de petits déplacements simule le mouvement.

```
from tkinter import *
```

```
Largeur = 400
```

```
Hauteur = 200
```

```

root = Tk()
canvas = Canvas(root, width=Largeur, height=Hauteur, background="white")
canvas.pack(fill="both", expand=True)

# Les coordonnées et la vitesse
x0, y0 = 100,100
dx = +5 # Vitesse horizontale
dy = +2 # Vitesse verticale

# Le rectangle à déplacer
rect = canvas.create_rectangle(x0,y0,x0+20,y0+20,width=2,fill="red")

# Fonction principale
def deplacer():
    global x0, y0, dx, dy

    x0 = x0 + dx # Nouvelle abscisse
    y0 = y0 + dy # Nouvelle ordonnée

    canvas.coords(rect,x0,y0,x0+20,y0+20) # Déplacement

    if x0 < 0 or x0 > Largeur:
        dx = -dx # Changement de sens horizontal
    if y0 < 0 or y0 > Hauteur:
        dy = -dy # Changement de sens vertical

    canvas.after(50,deplacer) # Appel après 50 millisecondes

    return

# Fonction pour le bouton
def action_deplacer():
    deplacer()
    return

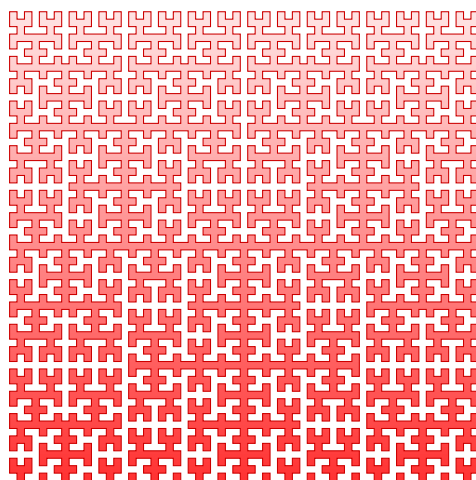
# Boutons
bouton_couleur = Button(root,text="Déplacer", width=20, command=action_deplacer)
bouton_couleur.pack(pady=10)

bouton_quitter = Button(root,text="Quitter", width=20, command=root.quit)
bouton_quitter.pack(side=BOTTOM, pady=10)

root.mainloop()

```

CINQUIÈME PARTIE

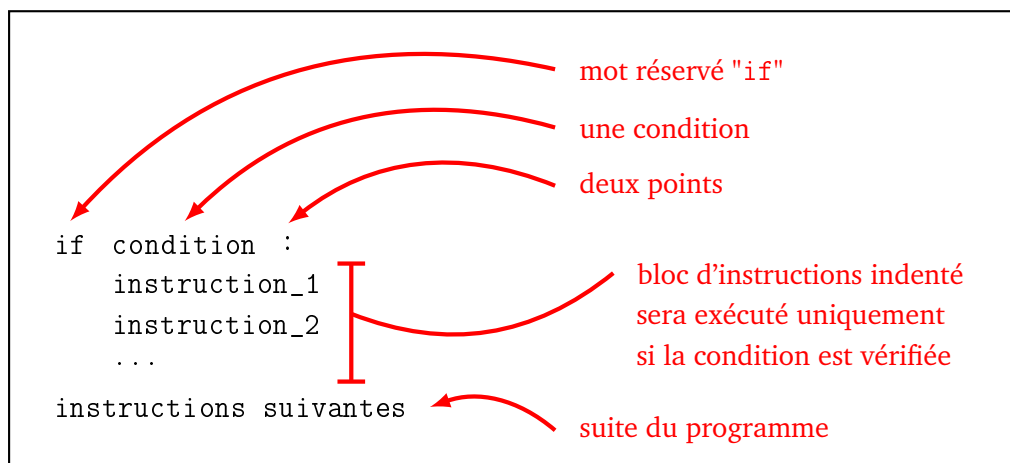


GUIDES

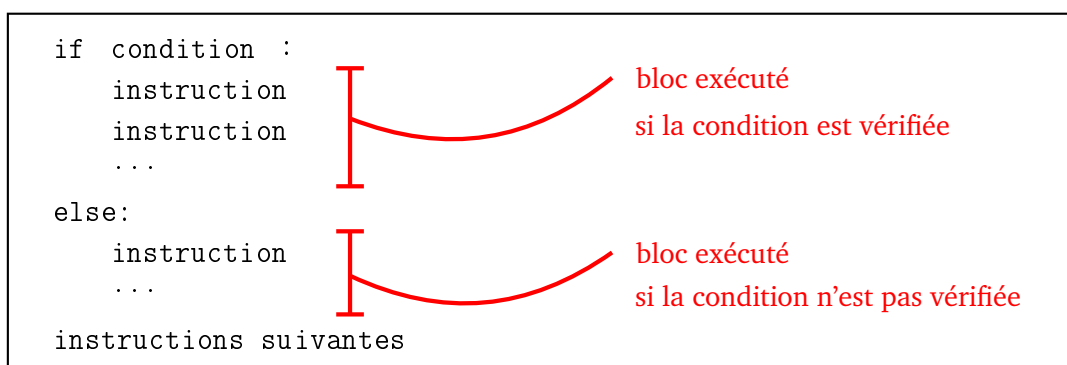
Guide de survie Python

1. Test et boucles

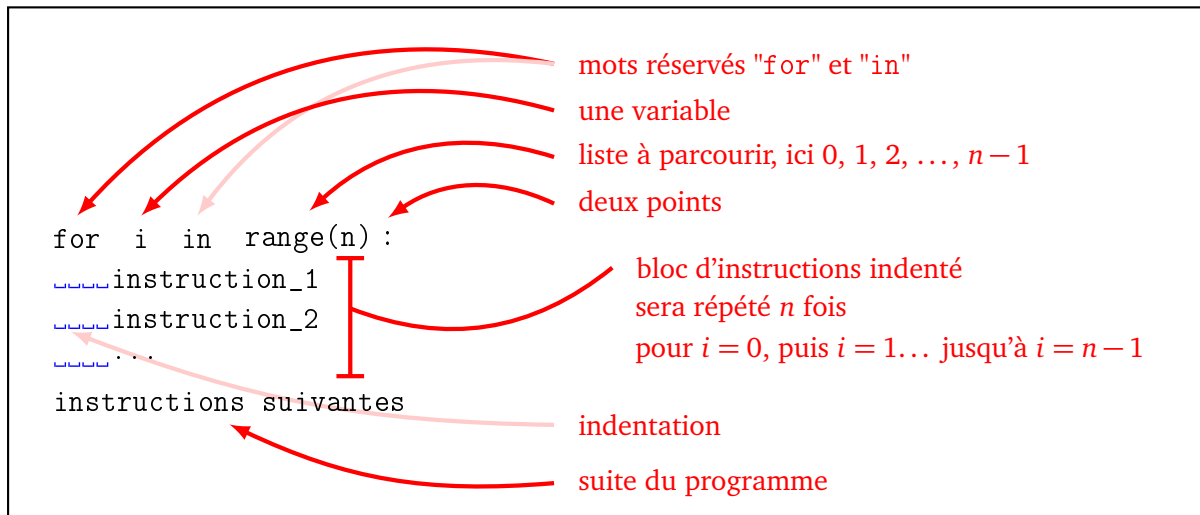
1.1. Si ... alors ...



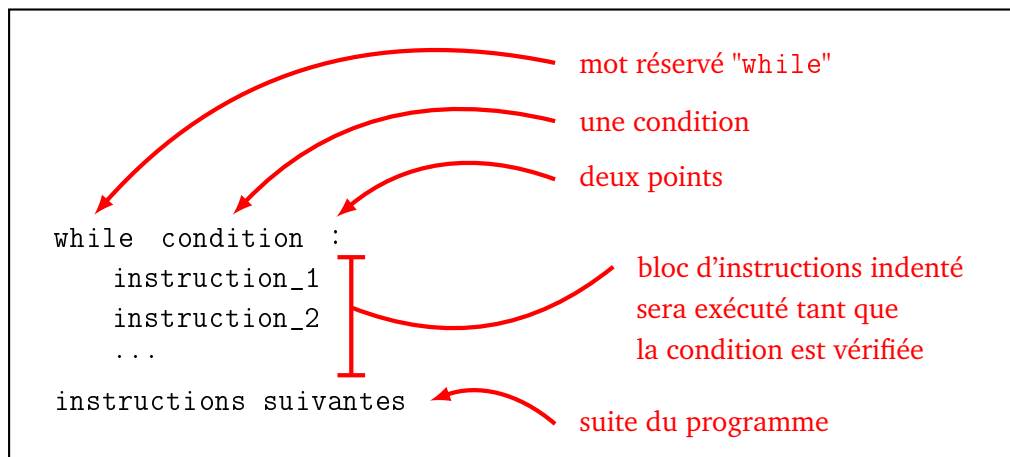
1.2. Si ... alors ... sinon ...



1.3. Boucle pour



1.4. Boucle tant que



1.5. Quitter une boucle

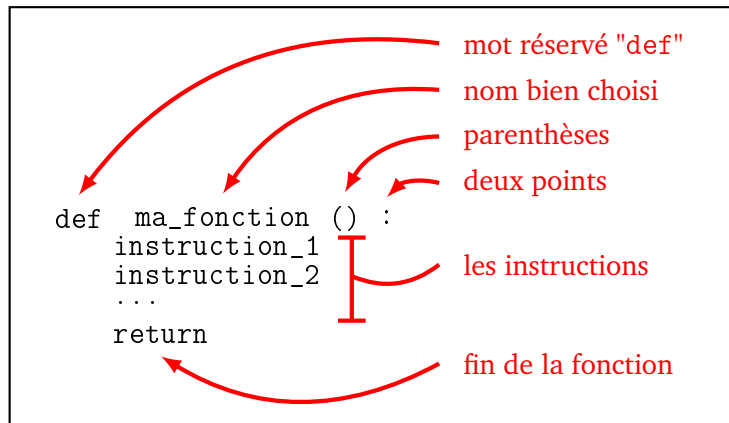
La commande Python pour quitter immédiatement une boucle « tant que » ou une boucle « pour » est l'instruction `break`.

2. Type de données

- `int` Entier. Exemples : 123 ou -15.
- `float` Nombre flottant (ou à virgule). Exemples : 4.56, -0.001, 6.022e23 (pour 6.022×10^{23}), 4e-3 (pour $0.004 = 4 \times 10^{-3}$).
- `str` Caractère ou chaîne de caractères. Exemples : 'Y', 'k', 'Hello', 'World !'.
- `bool` Booléen. True ou False.
- `list` Liste. Exemple : [1,2,3,4].

3. Définir des fonctions

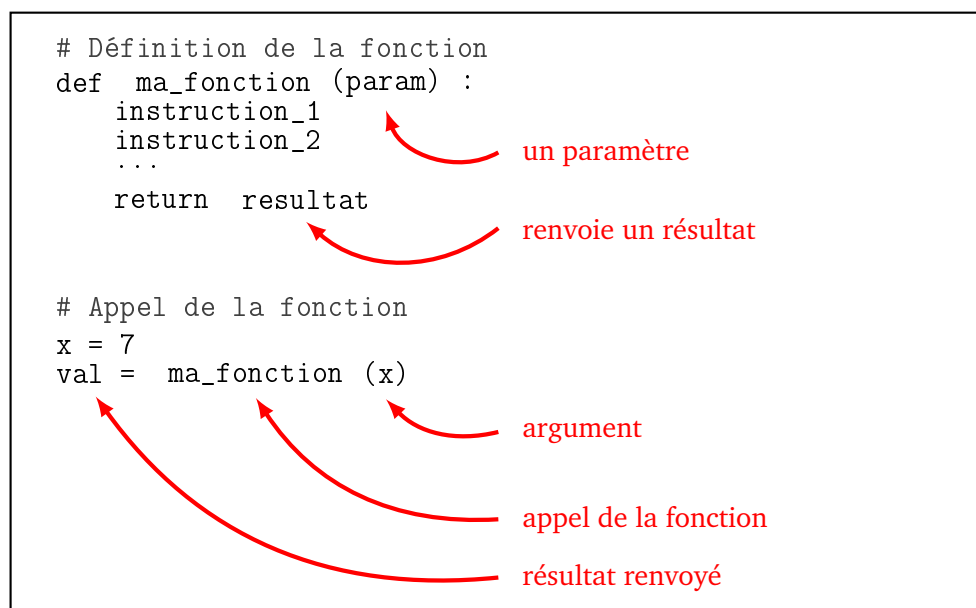
3.1. Définition d'une fonction



3.2. Fonction avec paramètre

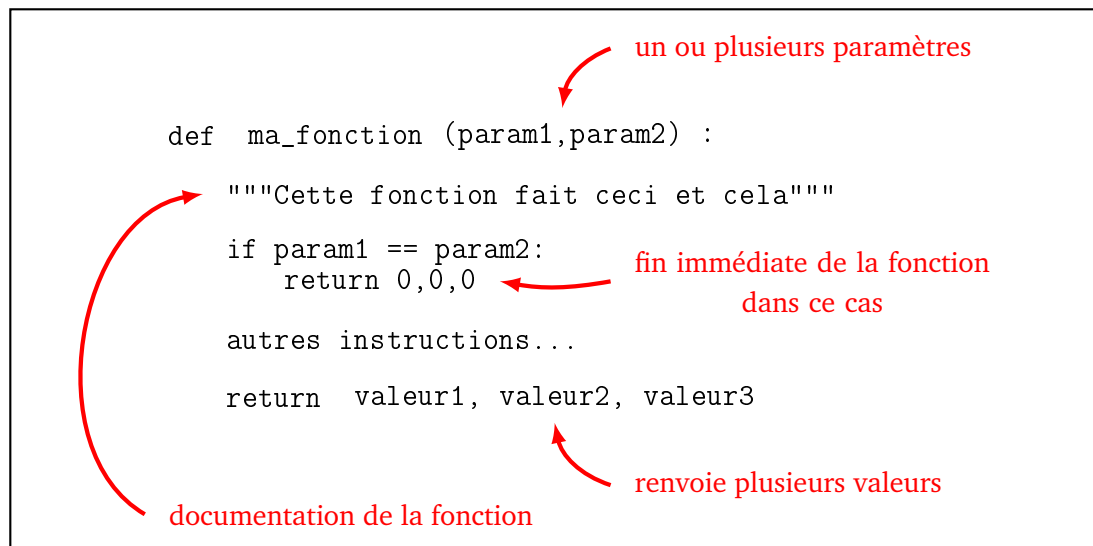
Les fonctions informatiques acquièrent tout leur potentiel avec :

- une *entrée*, qui regroupe des variables qui servent de *paramètres*,
- une *sortie*, qui est un résultat renvoyé par la fonction (et qui souvent dépendra des paramètres d'entrée).



3.3. Fonction avec plusieurs paramètres

Il peut y avoir plusieurs paramètres en entrée, il peut y avoir plusieurs résultats en sortie.



Voici un exemple d'une fonction avec deux paramètres et deux sorties.

```
def somme_produit(x,y):
    """ Calcule la somme et le produit de deux nombres. """
    S = x + y      # Somme
    P = x*y        # Produit
    return S, P    # Renvoie les résultats

# Appel de la fonction
som, prod = somme_produit(3,7) # Résultats
print("Somme :",som)          # Affichage
print("Produit :",prod)       # Affichage
```

- Très important ! Il ne faut pas confondre afficher et renvoyer une valeur. L'affichage (par la commande `print()`) affiche juste quelque chose à l'écran. La plupart des fonctions n'affichent rien, mais renvoient une valeur (ou plusieurs). C'est beaucoup plus utile car cette valeur peut être utilisée ailleurs dans le programme.
- Dès que le programme rencontre l'instruction `return`, la fonction s'arrête et renvoie le résultat. Il peut y avoir plusieurs fois l'instruction `return` dans une fonction mais une seule sera exécutée. On peut aussi ne pas mettre d'instruction `return` si la fonction ne renvoie rien.
- Dans les instructions d'une fonction, on peut bien sûr faire appel à d'autres fonctions !

3.4. Commentaires et docstring

- **Commentaire.** Tout ce qui suit le signe dièse `#` est un commentaire et est ignoré par Python. Par exemple :

```
# Boucle principale
while r != 0:    # Tant que le reste n'est pas nul
    r = r - 1    # Diminuer le reste
```

- **Docstring.** Tu peux décrire ce que fait une fonction en commençant par un *docstring*, c'est-à-dire une description en français, entourée par trois guillemets. Par exemple :

```
def produit(x,y):
    """ Calcule le produit de deux nombres
    Entrée : deux nombres x et y
    Sortie : le produit de x par y """
```

```
p = x * y
return p
```

3.5. Variable locale

Voici une fonction toute simple qui prend en entrée un nombre et renvoie le nombre augmenté de un.

```
def ma_fonction(x):
    x = x + 1
    return x
```

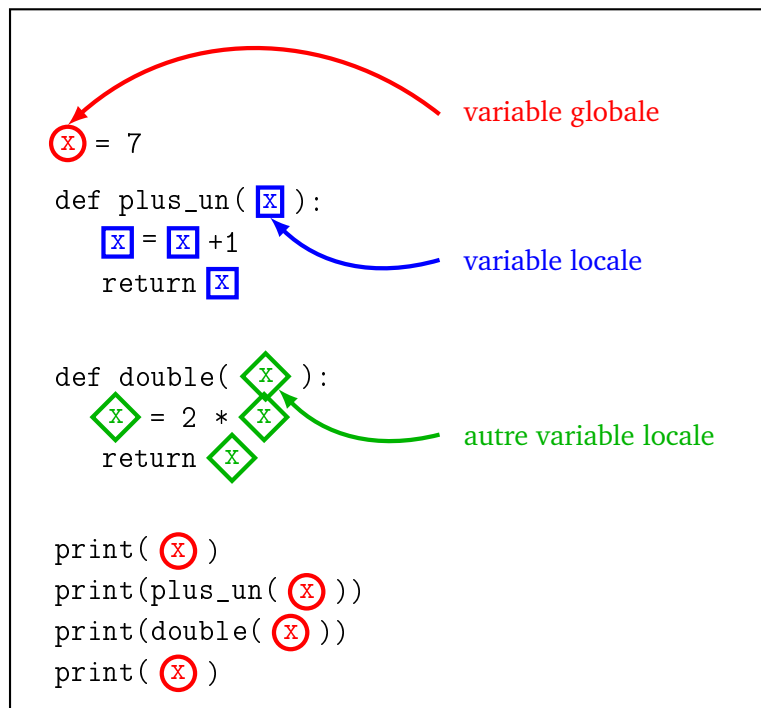
- Bien évidemment `ma_fonction(3)` renvoie 4.
- Si la valeur de `y` est 5, alors `ma_fonction(y)` renvoie 6. Mais attention, la valeur de `y` n'a pas changé, elle vaut toujours 5.
- Voici la situation problématique qu'il faut bien comprendre :

```
x = 7
print(ma_fonction(x))
print(x)
```

- La variable `x` est initialisée à 7.
- L'appel de la fonction `ma_fonction(x)` est donc la même chose que `ma_fonction(7)` et renvoie logiquement 8.
- Que vaut la variable `x` à la fin ? La variable `x` est inchangée et vaut toujours 7 ! Même s'il y a eu entre temps une instruction `x = x + 1`. Cette instruction a changé le `x` à l'intérieur de la fonction, mais pas le `x` en dehors de la fonction.

- Les variables définies à l'intérieur d'une fonction sont appelées **variables locales**. Elles n'existent pas en dehors de la fonction.
- Si une variable dans une fonction porte le même nom qu'une variable dans le programme (comme le `x` dans l'exemple ci-dessus), il y a deux variables distinctes ; la variable locale n'existant que dans la fonction.

Pour bien comprendre la portée des variables, tu peux colorier les variables globales d'une fonction en rouge, et les variables locales avec une couleur par fonction. Le petit programme suivant définit une fonction qui ajoute un, et une autre qui calcule le double.



Le programme affiche d'abord la valeur de `x`, donc 7, puis il ajoute un à 7, il affiche donc 8, puis il affiche le double de `x`, donc 14. La variable globale `x` n'a jamais changé, le dernier affichage de `x` est donc encore 7.

3.6. Variable globale

Une **variable globale** est une variable qui est définie pour l'ensemble du programme. Il n'est généralement pas recommandé d'utiliser de telles variables, mais cela peut être utile dans certains cas. Voyons un exemple. On déclare la variable globale, ici la constante de gravitation, en début de programme comme une variable classique :

```
gravitation = 9.81
```

La contenu de la variable `gravitation` est maintenant accessible partout. Par contre, si on souhaite changer la valeur de cette variable dans une fonction, il faut bien préciser à Python que l'on est conscient de modifier une variable globale !

Par exemple pour des calculs sur la Lune, il faut changer la constante de gravitation qui y est beaucoup plus faible.

```

def sur_la_lune():
    global gravitation    # Oui, je veux modifier cette variable globale !
    gravitation = 1.625   # Nouvelle valeur pour tout le programme
    ...

```

3.7. Arguments optionnels

Il est possible de donner des arguments optionnels. Voici comment définir une fonction (ici qui dessinerait un trait) en donnant des valeurs par défaut :

```
def tracer(longueur, epaisseur=5, couleur="blue"):
```

- La commande `tracer(100)` trace mon trait, et comme je n'ai précisé que la longueur, les arguments `epaisseur` et `couleur` prennent les valeurs par défaut (5 et bleu).
- La commande `tracer(100, epaisseur=10)` trace mon trait avec une nouvelle épaisseur (la couleur est celle par défaut).

- La commande `tracer(100, couleur="red")` trace mon trait avec une nouvelle couleur (l'épaisseur est celle par défaut).
- La commande `tracer(100, epaisseur=10, couleur="red")` trace mon trait avec une nouvelle épaisseur et une nouvelle couleur.
- Voici aussi ce que tu peux utiliser :
 - `tracer(100, 10, "red")` : ne pas préciser les noms des options si on fait attention à l'ordre.
 - `tracer(couleur="red", epaisseur=10, longueur=100)` : on peut nommer n'importe quelle variable ; les variables nommées peuvent être passées en paramètre dans n'importe quel ordre !

4. Modules

4.1. Utiliser un module

- `from math import *` Importe toutes les fonctions du module `math`. Pour pouvoir utiliser par exemple la fonction sinus par `sin(0)`. C'est la méthode la plus simple et c'est celle que nous utilisons dans ce livre.
- `import math` Permet d'utiliser les fonctions du module `math`. On a alors accès à la fonction sinus par `math.sin(0)`. C'est la méthode recommandée officiellement afin d'éviter les conflits entre les modules.

4.2. Principaux modules

- `math` contient les principales fonctions mathématiques.
- `random` simule le tirage au hasard.
- `turtle` la tortue Python, l'équivalent de *Scratch*.
- `matplotlib` permet de tracer des graphiques et visualiser des données.
- `tkinter` permet d'afficher des fenêtres graphiques.
- `time` pour l'heure, la date et chronométrer.
- `timeit` pour mesurer le temps d'exécution d'une fonction.

Il existe beaucoup d'autres modules !

5. Erreurs

5.1. Erreurs d'indentation

```
a = 3
b = 2
```

Python renvoie le message d'erreur *IndentationError* : *unexpected indent*. Il indique le numéro de ligne où se situe l'erreur d'indentation, il pointe même à l'aide du symbole « ^ » l'endroit exact de l'erreur.

5.2. Erreurs de syntaxe

- ```
while x >= 0
 x = x - 1
```

Python renvoie le message d'erreur *SyntaxError* : *invalid syntax* car il manque les deux points après la condition `while x >= 0` :

- `chaine = Coucou le monde` renvoie une erreur car il manque les guillemets pour définir la chaîne de caractères.

- `print("Coucou"` Python renvoie le message d'erreur *SyntaxError : unexpected EOF while parsing* car l'expression est mal parenthésée.
- `if val = 1:` Encore une erreur de syntaxe, car il faudrait écrire `if val == 1:`.

### 5.3. Erreurs de type

- **Entier**

```
n = 7.0
for i in range(n):
 print(i)
```

Python renvoie le message d'erreur *TypeError : 'float' object cannot be interpreted as an integer*. En effet 7.0 n'est pas un entier, mais un nombre flottant.

- **Nombre flottant**

```
x = "9"
sqrt(x)
```

Python renvoie le message d'erreur *TypeError : a float is required*, car "9" est une chaîne de caractères et pas un nombre.

- **Mauvais nombre d'arguments**

`gcd(12)` Python renvoie le message d'erreur *TypeError : gcd() takes exactly 2 arguments (1 given)* car la fonction `gcd()` du module `math` a besoin des deux arguments, comme par exemple `gcd(12, 18)`.

### 5.4. Erreurs de nom

- `if y != 0: y = y - 1` Python renvoie le message *NameError : name 'y' is not defined* si la variable `y` n'a pas encore été définie.
- Cette erreur peut aussi se produire si les minuscules/majuscules ne pas scrupuleusement respectées. `variable`, `Variable` et `VARIABLE` sont trois noms de variables différents.
- `x = sqrt(2)` Python renvoie le message *NameError : name 'sqrt' is not defined*, il faut importer le module `math` pour pouvoir utiliser la fonction `sqrt()`.
- **Fonction non encore définie**

```
produit(6,7)
```

```
def produit(a,b):
 return a*b
```

Renvoie une erreur *NameError : name 'produit' is not defined* car une fonction doit être définie avant d'être utilisée.

### 5.5. Exercice

Corrige le code ! Python doit afficher 7 5 9.

```
a == 7
if (a = 2) or (a >= 5)
 b = a - 2
 c = a + 2
else
b = a // 2
c = 2 * a
```



```
print(a b c)
```

## 5.6. Autres problèmes

Le programme se lance mais s'interrompt en cours de route ou bien ne fait pas ce que tu veux ? C'est là que les ennuis commencent, il faut décaféariser le code ! Il n'y a pas de solutions générales mais seulement quelques conseils :

- Un code propre, bien structuré, bien commenté, avec des noms de variables et de fonctions bien choisis est plus facile à relire.
- Teste ton algorithme à la main avec papier/crayon pour les cas faciles.
- N'hésite pas à afficher les valeurs des variables, pour voir leur évolution au cours du temps. Par exemple `print(i, liste[i])` dans une boucle.
- Une meilleure méthode pour inspecter le code est de visualiser les valeurs associées aux variables à l'aide des fonctionnalités *debug* de ton éditeur Python favori. Il est aussi possible de faire une exécution pas à pas.
- Est-ce que le programme fonctionne avec certaines valeurs et pas d'autres ? As-tu pensé aux cas extrêmes ? Est-ce que  $n$  est nul alors que ce n'est pas autorisé ? Est-ce que la liste est vide, alors que le programme ne gère pas ce cas ? etc.

Voici quelques exemples.

- Je veux afficher les carrés des entiers de 1 à 10. Le programme suivant ne renvoie pas d'erreur mais ne fait pas ce que je veux.

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

La boucle itère sur les entiers de 0 à 9. Il faut écrire `range(1, 11)`.

- Je veux afficher le dernier élément de ma liste.

```
liste = [1,2,3,4]
print(liste[4])
```

Python renvoie le message d'erreur *IndexError : list index out of range* car le dernier élément est celui de rang 3.

- Je veux faire un compte à rebours. Le programme suivant ne s'arrête jamais.

```
n = 10
while n != "0":
 n = n - 1
 print(n)
```

Avec une boucle « tant que » il faut prendre grand soin de bien écrire la condition et de vérifier qu'elle finit pas être fausse. Ici, elle est mal formulée, cela devrait être `while n != 0:`.

# Principales fonctions

## 1. Mathématiques

### Opérations classiques

- $a + b$ ,  $a - b$ ,  $a * b$  opérations classiques
- $a / b$  division « réelle » (renvoie un nombre flottant)
- $a // b$  quotient de la division euclidienne (renvoie un entier)
- $a \% b$  reste de la division euclidienne, appelé  $a$  modulo  $b$
- $\text{abs}(x)$  valeur absolue
- $x ** n$  puissance  $x^n$
- $4.56\text{e}12$  pour  $4.56 \times 10^{12}$

### Module « math »

L'usage d'autres fonctions mathématiques nécessite le recours au module `math` qui s'appelle par la commande :

```
from math import *
```

- $\text{sqrt}(x)$  racine carrée  $\sqrt{x}$
- $\cos(x)$ ,  $\sin(x)$ ,  $\tan(x)$  fonctions trigonométriques  $\cos x$ ,  $\sin x$ ,  $\tan x$  en radians
- `pi` valeur approchée de  $\pi = 3.14159265\dots$
- $\text{floor}(x)$  entier juste en-dessous de  $x$
- $\text{ceil}(x)$  entier juste au-dessus de  $x$
- $\text{gcd}(a, b)$  pgcd de  $a$  et de  $b$

### Module « random »

Le module `random` génère des nombres de façon pseudo-aléatoire. Il s'appelle par la commande :

```
from random import *
```

- $\text{random}()$  à chaque appel, renvoie un nombre flottant  $x$  au hasard vérifiant  $0 \leq x < 1$ .
- $\text{randint}(a, b)$  à chaque appel, renvoie un nombre entier  $n$  au hasard vérifiant  $a \leq n \leq b$ .
- $\text{choice}(liste)$  à chaque appel, tire au hasard un élément de la liste.
- $liste.\text{shuffle}()$  mélange la liste (la liste est modifiée).

### Écriture binaire

- $\text{bin}(n)$  renvoie l'écriture binaire de l'entier  $n$  sous la forme d'une chaîne. Exemple :  $\text{bin}(17)$  renvoie `'0b10001'`.
- Pour écrire directement un nombre en écriture binaire, il suffit d'écrire le nombre en commençant par `0b` (sans guillemets). Par exemple `0b11011` vaut 27.

## 2. Booléens

Un booléen est une donnée qui prend soit la valeur `True` (« vrai »), soit la valeur `False` (« faux »).

### Comparaisons

Les tests de comparaison suivants renvoient un booléen.

- `a == b` test d'égalité
- `a < b` test inférieur strict
- `a <= b` test inférieur large
- `a > b` ou `a >= b` test supérieur
- `a != b` test de non égalité

Ne pas confondre « `a = b` » (affectation) et « `a == b` » (test d'égalité).

### Opérations sur les booléens

- `P and Q` « et » logique
- `P or Q` « ou » logique
- `not P` négation

## 3. Chaînes de caractères I

### Chaînes

- `"A"` ou `'A'` un caractère
- `"Python"` ou `'Python'` une chaîne de caractères
- `len(chaine)` la longueur de la chaîne. Exemple : `len("Python")` renvoie 6.
- `chaine1 + chaine2` concaténation.  
Exemple : `"J aime bien" + "Python"` renvoie `"J aime bienPython"`.
- `chaine[i]` renvoie le *i*-ème caractère de `chaine` (la numérotation commence à 0).  
Exemple avec `chaine = "Python"`, `chaine[1]` vaut `"y"`. Voir le tableau ci-dessous.

| Lettre | P | y | t | h | o | n |
|--------|---|---|---|---|---|---|
| Rang   | 0 | 1 | 2 | 3 | 4 | 5 |

### Conversion nombre/chaîne

- **Chaîne.** `str(nombre)` convertit un nombre (entier ou flottant) en une chaîne. Exemples : `str(7)` renvoie la chaîne `"7"` ; `str(1.234)` renvoie la chaîne `"1.234"`.
- **Entier.** `int(chaine)` renvoie l'entier correspondant à la chaîne. Exemple `int("45")` renvoie l'entier 45.
- **Nombre flottant.** `float(chaine)` renvoie le nombre flottant correspondant à la chaîne. Exemple `float("3.14")` renvoie le nombre 3.14.

### Sous-chaînes

- `chaine[i:j]` renvoie la sous-chaîne des caractères de rang *i* à *j* – 1 de `chaine`.  
Exemple : avec `chaine = "Ceci est une chaine"`, `chaine[2:6]` renvoie `"ci e"`.
- `chaine[i:]` renvoie les caractères de rang *i* jusqu'à la fin de `chaine`.  
Exemple : `chaine[5:]` renvoie `"est une chaine"`.
- `chaine[:j]` renvoie les caractères du début jusqu'au rang *j* – 1 de `chaine`. Exemple : `chaine[:4]` renvoie `"Ceci"`.

### Mise en forme

La méthode `format()` permet de mettre en forme du texte ou des nombres. Cette fonction renvoie une chaîne de caractères.

- **Texte**

`Test` `Test` `Test`  
 — `'{:10}'.format('Test')` alignement à gauche (sur 10 caractères)  
 — `'{:>10}'.format('Test')` alignement à droite  
 — `'{:~10}'.format('Test')` centré

- **Entier**

`456` `456` `000456`  
 — `'{:d}'.format(456)` entier  
 — `'{:6d}'.format(456)` alignement à droite (sur 6 caractères)  
 — `'{:06d}'.format(456)` ajout de zéros non significatifs (sur 6 caractères)

- **Nombre flottant**

`3.141593` `3.14159265` `3.1416` `003.1416`  
 — `'{:f}'.format(3.141592653589793)` nombre flottant  
 — `'{: .8f}'.format(3.141592653589793)` 8 chiffres après la virgule  
 — `'{:8.4f}'.format(3.141592653589793)` sur 8 caractères avec 4 chiffres après la virgule  
 — `'{:08.4f}'.format(3.141592653589793)` ajout de zéros non significatifs

## 4. Chaînes de caractères II

### Encodage

- `chr(n)` renvoie le caractère associé au numéro de code ASCII/unicode *n*. Exemple : `chr(65)` renvoie "A" ; `chr(97)` renvoie "a".
- `ord(c)` renvoie le numéro de code ASCII/unicode associé au caractère *c*. Exemple : `ord("A")` renvoie 65 ; `ord("a")` renvoie 97.

Le début de la table des codes ASCII/unicode est donné ci-dessous.

|    |    |    |   |    |   |    |   |    |   |    |   |     |   |     |   |     |   |     |   |
|----|----|----|---|----|---|----|---|----|---|----|---|-----|---|-----|---|-----|---|-----|---|
| 33 | !  | 43 | + | 53 | 5 | 63 | ? | 73 | I | 83 | S | 93  | ] | 103 | g | 113 | q | 123 | { |
| 34 | "  | 44 | , | 54 | 6 | 64 | @ | 74 | J | 84 | T | 94  | ^ | 104 | h | 114 | r | 124 |   |
| 35 | #  | 45 | - | 55 | 7 | 65 | A | 75 | K | 85 | U | 95  | _ | 105 | i | 115 | s | 125 | } |
| 36 | \$ | 46 | . | 56 | 8 | 66 | B | 76 | L | 86 | V | 96  | ' | 106 | j | 116 | t | 126 | ~ |
| 37 | %  | 47 | / | 57 | 9 | 67 | C | 77 | M | 87 | W | 97  | a | 107 | k | 117 | u | 127 | - |
| 38 | &  | 48 | 0 | 58 | : | 68 | D | 78 | N | 88 | X | 98  | b | 108 | l | 118 | v |     |   |
| 39 | '  | 49 | 1 | 59 | ; | 69 | E | 79 | O | 89 | Y | 99  | c | 109 | m | 119 | w |     |   |
| 40 | (  | 50 | 2 | 60 | < | 70 | F | 80 | P | 90 | Z | 100 | d | 110 | n | 120 | x |     |   |
| 41 | )  | 51 | 3 | 61 | = | 71 | G | 81 | Q | 91 | [ | 101 | e | 111 | o | 121 | y |     |   |
| 42 | *  | 52 | 4 | 62 | > | 72 | H | 82 | R | 92 | \ | 102 | f | 112 | p | 122 | z |     |   |

### Majuscules/minuscules

- `chaine.upper()` renvoie une chaîne en majuscules.
- `chaine.lower()` renvoie une chaîne en minuscules.

**Chercher/remplacer**

- `sous_chaine in chaine` renvoie « vrai » ou « faux » si `sous_chaine` apparaît dans `chaine`.  
Exemple : `"PAS" in "ETRE OU NE PAS ETRE"` vaut `True`.
- `chaine.find(sous_chaine)` renvoie le rang auquel la sous-chaîne a été trouvée (et -1 sinon).  
Exemple : avec `chaine = "ABCDE"`, `chaine.find("CD")` renvoie 2.
- `chaine.replace(sous_chaine,nouv_sous_chaine)` remplace chaque occurrence de la sous-chaîne par la nouvelle sous-chaîne.  
Exemple : avec `chaine = "ABCDE"`, `chaine.replace("CD", "XY")` renvoie `"ABXYE"`.

**Séparer/regrouper**

- `chaine.split(separateur)` sépare la chaîne en une liste de sous-chaînes (par défaut le séparateur est l'espace).  
Exemples :  
— `"Etre ou ne pas etre.".split()` renvoie `['Etre', 'ou', 'ne', 'pas', 'etre.']`  
— `"12.5;17.5;18".split(";")` renvoie `['12.5', '17.5', '18']`
- `separateur.join(liste)` regroupe les sous-chaînes en une seule chaîne en ajoutant le séparateur entre chaque.  
Exemples :  
— `" ".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etreounepasetre.'` Il manque les espaces.  
— `" ".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre ou ne pas etre.'`  
C'est mieux lorsque le séparateur est une espace.  
— `"--".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre--ou--ne--pas--etre.'`

## 5. Listes I

**Construction d'une liste**

Exemples :

- `liste1 = [5,4,3,2,1]` une liste de 5 entiers.
- `liste2 = ["Vendredi", "Samedi", "Dimanche"]` une liste de 3 chaînes.
- `liste3 = []` la liste vide.
- `list(range(n))` liste des entiers de 0 à  $n-1$ .
- `list(range(a,b))` liste des entiers de  $a$  à  $b-1$ .
- `list(range(a,b,saut))` liste des entiers de  $a$  à  $b-1$ , avec un pas donné par l'entier `saut`.

**Accéder à un élément**

- `liste[i]` renvoie l'élément de la liste de rang  $i$ . Attention, le rang commence à 0.  
Exemple : `liste = ["A", "B", "C", "D", "E", "F"]` alors `liste[2]` renvoie `"C"`.

| Lettre | "A" | "B" | "C" | "D" | "E" | "F" |
|--------|-----|-----|-----|-----|-----|-----|
| Rang   | 0   | 1   | 2   | 3   | 4   | 5   |

- `liste[-1]` renvoie le dernier élément, `liste[-2]` renvoie l'avant-dernier élément...
- `liste.pop()` supprime le dernier élément de la liste et le renvoie (c'est l'opération « dépiler »).

**Ajouter un élément (ou plusieurs)**

- `liste.append(element)` ajoute l'élément à la fin de la liste. Exemple : si `liste = [5,6,7,8]` alors `liste.append(9)` rajoute 9 à la liste, `liste` vaut `[5,6,7,8,9]`.
- `nouv_liste = liste + [element]` fournit une nouvelle liste avec un élément en plus à la fin.  
Exemple : `[1,2,3,4] + [5]` vaut `[1,2,3,4,5]`.

- `[element] + liste` renvoie une liste où l'élément est ajouté au début. Exemple : `[5] + [1,2,3,4]` vaut `[5,1,2,3,4]`.
- `liste1 + liste2` concatène les deux listes. Exemple : avec `liste1 = [4,5,6]` et `liste2 = [7,8,9]` alors `liste1 + liste2` vaut `[4,5,6,7,8,9]`.

**Exemple de construction.** Voici comment construire la liste qui contient les premiers carrés :

```
liste_carres = [] # On part d'une liste vide
for i in range(10):
 liste_carres.append(i**2) # On ajoute un carré
```

À la fin `liste_carres` vaut :

`[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

### Parcourir une liste

- `len(liste)` renvoie la longueur de la liste. Exemple : `len([5,4,3,2,1])` renvoie 5.
- Parcourir simplement une liste (et ici afficher chaque élément) :

```
for element in liste:
 print(element)
```

- Parcourir une liste à l'aide du rang.

```
n = len(liste)
for i in range(n):
 print(i, liste[i])
```

## 6. Listes II

### Mathématiques

- `max(liste)` renvoie le plus grand élément. Exemple : `max([10,16,13,14])` renvoie 16.
- `min(liste)` renvoie le plus petit élément. Exemple : `min([10,16,13,14])` renvoie 10.
- `sum(liste)` renvoie la somme de tous les éléments. Exemple : `sum([10,16,13,14])` renvoie 53.

### Trancher des listes

- `liste[a:b]` renvoie la sous-liste des éléments du rang  $a$  au rang  $b - 1$ .
- `liste[a:]` renvoie la liste des éléments du rang  $a$  jusqu'à la fin.
- `liste[:b]` renvoie la liste des éléments du début jusqu'au rang  $b - 1$ .

| Lettre | "A" | "B" | "C" | "D" | "E" | "F" | "G" |
|--------|-----|-----|-----|-----|-----|-----|-----|
| Rang   | 0   | 1   | 2   | 3   | 4   | 5   | 6   |

Par exemple si `liste = ["A","B","C","D","E","F","G"]` alors :

- `liste[1:4]` renvoie `["B","C","D"]`.
- `liste[:2]` c'est comme `liste[0:2]` et renvoie `["A","B"]`.
- `liste[4:]` renvoie `["E","F","G"]`. C'est la même chose que `liste[4:n]` où  $n = \text{len}(\text{liste})$ .

### Trouver le rang d'un élément

- `liste.index(element)` renvoie la première position à laquelle l'élément a été trouvé. Exemple : avec `liste = [12, 30, 5, 9, 5, 21]`, `liste.index(5)` renvoie 2.

- Si on souhaite juste savoir si un élément appartient à une liste, alors l'instruction :

```
element in liste
```

renvoie True ou False. Exemple : avec `liste = [12, 30, 5, 9, 5, 21]`, «`9 in liste`» est vrai, alors que «`8 in liste`» est faux.

### Ordonner

- `sorted(liste)` renvoie la liste ordonnée des éléments.  
Exemple : `sorted([13, 11, 7, 4, 6, 8, 12, 6])` renvoie la liste `[4, 6, 6, 7, 8, 11, 12, 13]`.
- `liste.sort()` ne renvoie rien mais par contre la liste `liste` est maintenant ordonnée.

### Inverser une liste

Voici trois méthodes :

- `liste.reverse()` modifie la liste sur place ;
- `list(reversed(liste))` renvoie une nouvelle liste ;
- `liste[::-1]` renvoie une nouvelle liste.

### Supprimer un élément

Trois méthodes.

- `liste.remove(element)` supprime la première occurrence trouvée.  
Exemple : `liste = [2, 5, 3, 8, 5]`, la commande `liste.remove(5)` modifie la liste qui maintenant vaut `[2, 3, 8, 5]` (le premier 5 a disparu).
- `del liste[i]` supprime l'élément de rang *i* (la liste est modifiée).
- `element = liste.pop()` supprime le dernier élément de la liste et le renvoie. C'est l'opération « dépiler ».

### Liste par compréhension

- Partons d'une liste, par exemple `maliste = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1]`.
- `liste_doubles = [ 2*x for x in maliste ]` renvoie une liste qui contient les doubles des éléments de la liste `maliste`. C'est donc la liste `[2, 4, 6, 8, ...]`.
- `liste_carres = [ x**2 for x in maliste ]` renvoie la liste des carrés de éléments de la liste `maliste`. C'est donc la liste `[1, 4, 9, 16, ...]`.
- `liste_partielle = [x for x in maliste if x > 2]` extrait la liste composée des seuls éléments strictement supérieurs à 2. C'est donc la liste `[3, 4, 5, 6, 7, 6, 5, 4, 3]`.

### Liste de listes

Exemple :

```
tableau = [[2, 14, 5], [3, 5, 7], [15, 19, 4], [8, 6, 5]]
```

correspond au tableau :

|                 |              |                 |              |              |
|-----------------|--------------|-----------------|--------------|--------------|
|                 |              | indice <i>j</i> |              |              |
|                 |              | →               |              |              |
|                 |              | <i>j</i> = 0    | <i>j</i> = 1 | <i>j</i> = 2 |
| indice <i>i</i> | <i>i</i> = 0 | 2               | 14           | 5            |
|                 | <i>i</i> = 1 | 3               | 5            | 7            |
|                 | <i>i</i> = 2 | 15              | 19           | 4            |
|                 | <i>i</i> = 3 | 8               | 6            | 5            |

Alors `tableau[i]` renvoie la sous-liste de rang *i*, et `tableau[i][j]` renvoie l'élément situé dans la sous-liste de rang *i*, au rang *j* de cette sous-liste. Par exemple :

- `tableau[0]` renvoie la sous-liste `[2, 14, 5]`.
- `tableau[1]` renvoie la sous-liste `[3, 5, 7]`.
- `tableau[0][0]` renvoie l'entier 2.
- `tableau[0][1]` renvoie l'entier 14.
- `tableau[2][1]` renvoie l'entier 19.

Un tableau de  $n$  lignes et  $p$  colonnes.

- `tableau = [[0 for j in range(p)] for i in range(n)]` initialise un tableau et le remplit de 0.
- `tableau[i][j] = 1` modifie une valeur du tableau (celle à l'emplacement  $(i, j)$ ).

## 7. Entrée/sortie

### Affichage

- `print(chaine1, chaine2, chaine3, ...)` affiche des chaînes ou des objets. Exemple : `print("Valeur =", 14)` affiche `Valeur = 14`. Exemple : `print("Ligne 1 \n Ligne 2")` affiche sur deux lignes.
- **Séparateur.** `print(..., sep="...")` change le séparateur (par défaut le séparateur est le caractère espace). Exemple : `print("Bob", 17, 13, 16, sep="; ")` affiche `Bob; 17; 13; 16`.
- **Fin de ligne.** `print(..., end="...")` change le caractère placé à la fin (par défaut c'est le saut de ligne `\n`). Exemple `print(17, end="")` puis `print(89)` affiche 1789 sur une seule ligne.

### Entrée clavier

`input()` met le programme en pause et attend de l'utilisateur un message au clavier (qu'il termine en appuyant sur la touche « Entrée »). Le message est une chaîne de caractères.

Voici un petit programme qui demande le prénom et l'âge de l'utilisateur et affiche un message du style « Bonjour Kevin » puis « Tu es mineur/majeur » selon l'âge.

```
prenom = input("Comment t'appelles-tu ? ")
print("Bonjour", prenom)
```

```
age_chaine = input("Quel âge as-tu ? ")
age = int(age_chaine)
```

```
if age >= 18:
 print("Tu es majeur !")
else:
 print("Tu es mineur !")
```

## 8. Fichiers

### Commande

- `fic = open("mon_fichier.txt", "r")` ouverture en lecture ("`r`" = *read*).
- `fic = open("mon_fichier.txt", "w")` ouverture en écriture ("`w`" = *write*). Le fichier est créé s'il n'existe pas, s'il existait le contenu précédent est d'abord effacé.
- `fic = open("mon_fichier.txt", "a")` ouverture en écriture, les données seront écrites à la fin des données actuelles ("`a`" = *append*).
- `fic.write("une ligne")` écriture dans le fichier.
- `fic.read()` lit tout le fichier (voir plus bas pour autre méthode).
- `fic.readlines()` lit toutes les lignes (voir plus bas pour autre méthode).



- `fic.close()` fermeture du fichier.

### Écrire des lignes dans un fichier

```
fic = open("mon_fichier.txt", "w")
```

```
fic.write("Bonjour le monde\n")
```

```
ligne = "Coucou\n"
```

```
fic.write(ligne)
```

```
fic.close()
```

### Lire les lignes d'un fichier

```
fic = open("mon_fichier.txt", "r")
```

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

```
fic.close()
```

### Lire un fichier (méthode officielle)

```
with open("mon_fichier.txt", "r") as fic:
 for ligne in fic:
 print(ligne)
```

## 9. Tortue

Le module `turtle` s'appelle par la commande :

```
from turtle import *
```

### Principales commandes

- `forward(longueur)` avance de longueur pas
- `backward(longueur)` recule
- `right(angle)` tourne vers la droite selon l'angle donné en degrés
- `left(angle)` tourne vers la gauche
- `setheading(direction)` s'oriente dans une direction (0 = droite, 90 = haut, -90 = bas, 180 = gauche)
- `goto(x, y)` se déplace jusqu'au point (x, y)
- `setx(newx)` change la valeur de l'abscisse (déplacement horizontal)
- `sety(newy)` change la valeur de l'ordonnée (déplacement vertical)
- `down()` abaisse le stylo
- `up()` relève le stylo
- `width(epaisseur)` change l'épaisseur du trait
- `color(couleur)` change la couleur du trait : "red", "green", "blue", "orange", "purple", ...
- `position()` renvoie la position (x, y) de la tortue
- `heading()` renvoie la direction angle vers laquelle pointe la tortue

- `towards(x,y)` renvoie l'angle entre l'horizontale et le segment commençant à la tortue et finissant au point  $(x,y)$
- `speed("fastest")` vitesse maximale de déplacement
- `exitonclick()` termine le programme dès que l'on clique

### Plusieurs tortues

Voici un exemple de programme avec deux tortues.

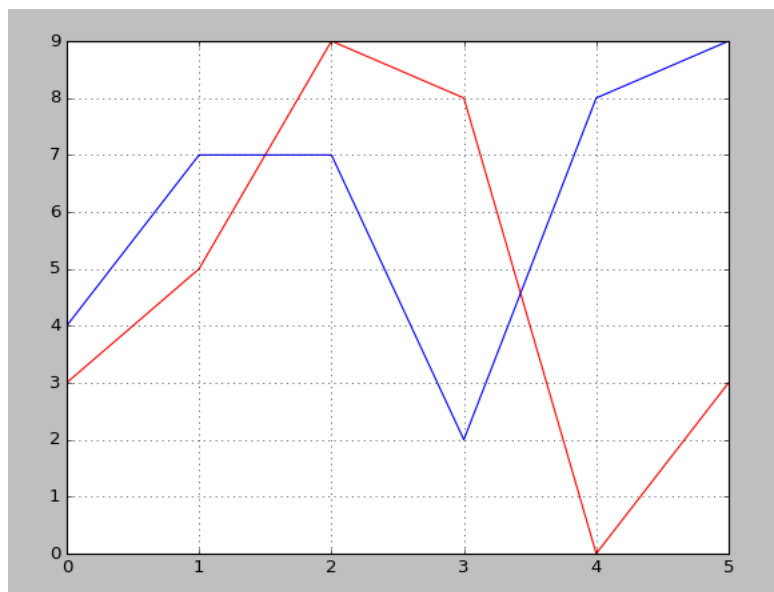
```
tortue1 = Turtle() # Avec un 'T' majuscule !
tortue2 = Turtle()

tortue1.color('red')
tortue2.color('blue')

tortue1.forward(100)
tortue2.left(90)
tortue2.forward(100)
```

## 10. Matplotlib

Avec le module `matplotlib` il est très facile de tracer une liste. Voici un exemple.



```
import matplotlib.pyplot as plt
```

```
liste1 = [3,5,9,8,0,3]
liste2 = [4,7,7,2,8,9]
```

```
plt.plot(liste1,color="red")
plt.plot(liste2,color="blue")
plt.grid()
plt.show()
```

### Principales fonctions.

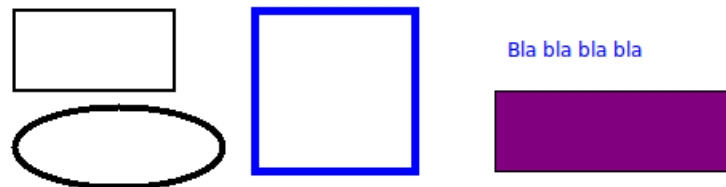
- `plt.plot(liste)` trace les points d'une liste (sous la forme  $(i, \ell_i)$ ) et les joint.

- `plt.plot(listex,listey)` trace les points d'une liste (sous la forme  $(x_i, y_i)$  où  $x_i$  parcourt la première liste et  $y_i$  la seconde).
- `plt.scatter(x,y,color='red',s=100)` affiche un point en  $(x, y)$  (d'une grosseur  $s$ ).
- `plt.grid()` trace une grille.
- `plt.show()` affiche tout.
- `plt.close()` termine le tracé.
- `plt.xlim(xmin,xmax)` définit l'intervalle des  $x$ .
- `plt.ylim(ymin,ymax)` définit l'intervalle des  $y$ .
- `plt.axis('equal')` impose un repère orthonormé.

## 11. Tkinter

### 11.1. Graphiques

Pour afficher ceci :



le code est :

```
Module tkinter
from tkinter import *

Fenêtre tkinter
root = Tk()

canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)

Un rectangle
canvas.create_rectangle(50,50,150,100,width=2)

Un rectangle à gros bords bleus
canvas.create_rectangle(200,50,300,150,width=5,outline="blue")

Un rectangle rempli de violet
canvas.create_rectangle(350,100,500,150,fill="purple")

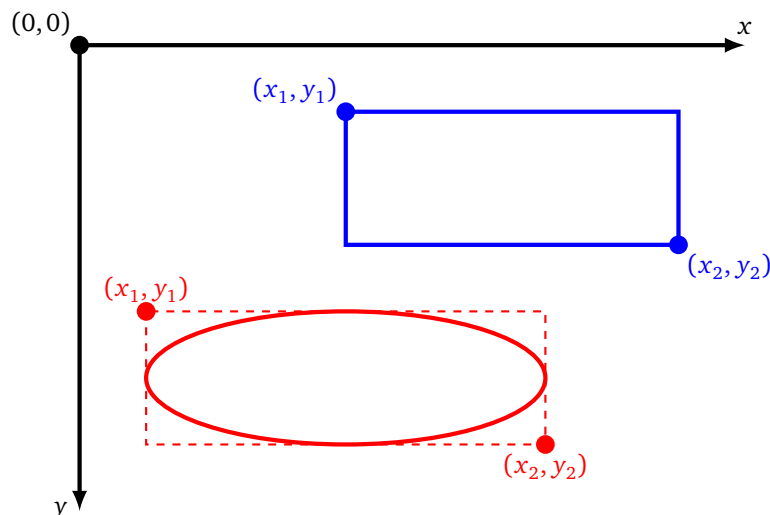
Un ovale
canvas.create_oval(50,110,180,160,width=4)

Du texte
canvas.create_text(400,75,text="Bla bla bla bla",fill="blue")
```

```
Ouverture de la fenêtre
root.mainloop()
```

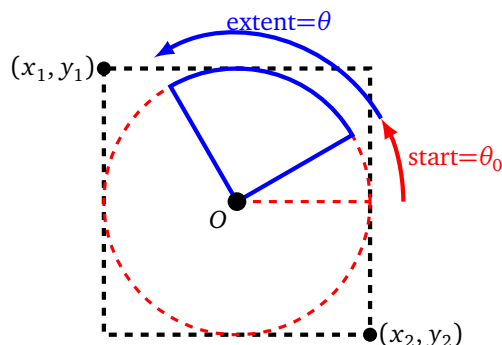
Quelques explications :

- Le module `tkinter` nous permet de définir des variables `root` et `canvas` qui définissent une fenêtre graphique (ici de largeur 800 et de hauteur 600 pixels). On décrit ensuite tout ce que l'on veut ajouter dans la fenêtre. Et enfin, la fenêtre est affichée par la commande `root.mainloop()` (tout à la fin).
- Attention ! Le repère graphique de la fenêtre a son axe des ordonnées dirigé vers le bas. L'origine  $(0,0)$  est le coin en haut à gauche (voir la figure ci-dessous).
- Commande pour tracer un rectangle : `create_rectangle(x1,y1,x2,y2)` ; il suffit de préciser les coordonnées  $(x_1, y_1)$  et  $(x_2, y_2)$  de deux sommets opposés. L'option `width` ajuste l'épaisseur du trait, `outline` définit la couleur de ce trait et `fill` définit la couleur de remplissage.
- Une ellipse est tracée par la commande `create_oval(x1,y1,x2,y2)`, où  $(x_1, y_1)$ ,  $(x_2, y_2)$  sont les coordonnées de deux sommets opposés d'un rectangle encadrant l'ellipse voulue (voir la figure). On obtient un cercle lorsque le rectangle correspondant est un carré.
- Le texte est affiché par la commande `canvas.create_text()` en précisant les coordonnées  $(x, y)$  du point à partir duquel on souhaite afficher le texte.



**Portion de cercle.** La fonction `create_arc()` n'est pas très intuitive. Il faut penser que l'on dessine un cercle, en précisant les coordonnées de deux sommets opposés d'un carré qui l'entoure, puis en précisant l'angle de début et l'angle du secteur (en degrés).

```
canvas.create_arc(x1,y1,x2,y2,start=debut_angle,extent=mon_angle)
```

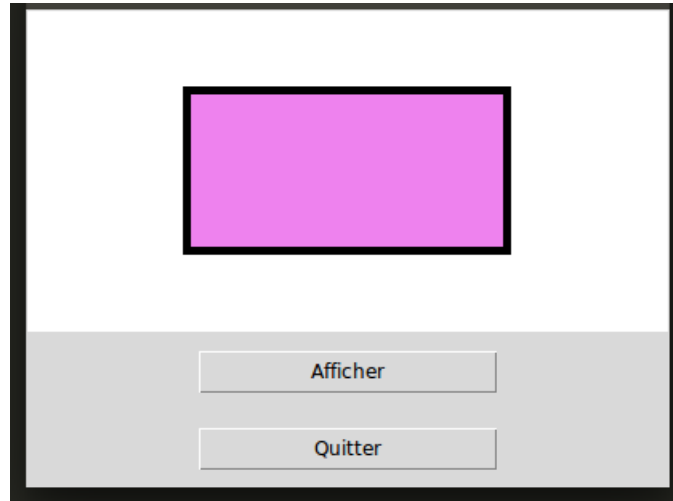


L'option `style=PIESLICE` affiche un secteur au lieu d'un arc.

## 11.2. Boutons

Il est plus ergonomique d'afficher des fenêtres dans lesquelles les actions sont exécutées en cliquant sur des boutons.

Voici un petit programme qui affiche une fenêtre avec deux boutons. Le premier bouton change la couleur du rectangle, le second termine le programme.



Le code est :

```
from tkinter import *
from random import *

root = Tk()
canvas = Canvas(root, width=400, height=200, background="white")
canvas.pack(fill="both", expand=True)

def action_bouton():
 canvas.delete("all") # Efface tout
 couleurs = ["red", "orange", "yellow", "green", "cyan", "blue", "violet", "purple"]
 coul = choice(couleurs) # Couleur au hasard
 canvas.create_rectangle(100, 50, 300, 150, width=5, fill=coul)
 return

bouton_couleur = Button(root, text="Afficher", width=20, command=action_bouton)
bouton_couleur.pack()

bouton_quitter = Button(root, text="Quitter", width=20, command=root.quit)
bouton_quitter.pack()

root.mainloop()
```

Quelques explications :

- On crée un bouton par la commande `Button`. L'option `text` personnalise le texte qui s'affiche sur le bouton. On ajoute le bouton créé à la fenêtre par la méthode `pack`.
- Le plus important est l'action associée au bouton ! C'est l'option `command` qui reçoit le nom de la fonction à exécuter lorsque le bouton est cliqué. Pour notre exemple `command=action_bouton` associe au clic sur le bouton un changement de couleur.

- Attention ! il faut donner le nom de la fonction sans parenthèses : `commande=ma_fonction` et pas `command=ma_fonction()`.
- Pour associer au bouton « Quitter » la fermeture du programme, l'argument est `command=root.quit`.
- La commande `canvas.delete("all")` efface tous les dessins de notre fenêtre graphique.

### 11.3. Texte

Voici comment afficher du texte avec Python et le module des fenêtres graphiques `tkinter`.

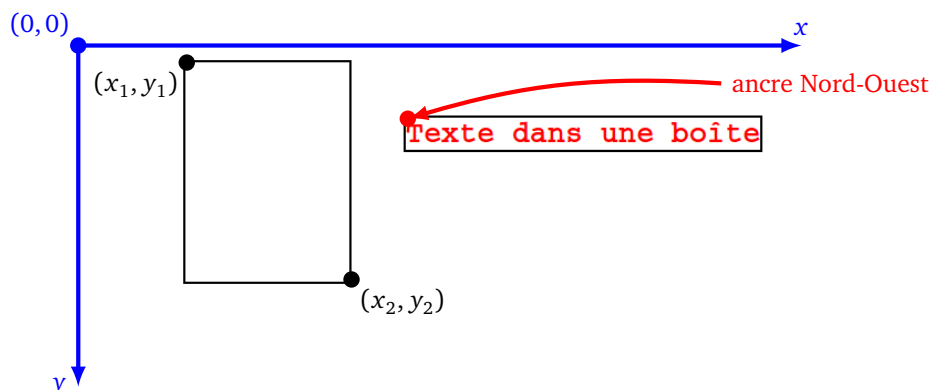
## Du texte avec Python !

Le code est :

```
from tkinter import *
from tkinter.font import Font
Fenêtre tkinter
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)
Fonte
mafonte = Font(family="Times", size=20)
Le texte
canvas.create_text(100,100, text="Du texte avec Python !",
anchor=NW, font=mafonte, fill="blue")
Ouverture de la fenêtre
root.mainloop()
```

Quelques explications :

- Le texte est affiché par la fonction `canvas.create_text()`. Il faut préciser les coordonnées  $(x, y)$  du point à partir duquel on souhaite afficher le texte.
- L'option `text` permet de passer la chaîne de caractères à afficher.
- L'option `anchor` permet de préciser le point d'ancrage du texte, `anchor=NW` signifie que la zone de texte est ancrée au point Nord-Ouest (NW) (voir la figure ci-dessous).
- L'option `fill` permet de préciser la couleur du texte.
- L'option `font` permet de définir la fonte (c'est-à-dire le style et la taille des caractères). Voici des exemples de fontes, à toi de les tester :
  - `Font(family="Times", size=20)`
  - `Font(family="Courier", size=16, weight="bold")` en **gras**
  - `Font(family="Helvetica", size=16, slant="italic")` en *italique*



## 11.4. Clic de souris

Voici un petit programme qui affiche une fenêtre graphique. Chaque fois que l'utilisateur clique (avec le bouton gauche de la souris) le programme affiche un petit carré (sur la fenêtre) et affiche « Clic à  $x = \dots$ ,  $y = \dots$  » (sur la console).

```
from tkinter import *

Fenêtre
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

Capture des clics de souris
def action_clic_souris(event):
 canvas.focus_set()
 x = event.x
 y = event.y
 canvas.create_rectangle(x,y,x+10,y+10,fill="red")
 print("Clic à x =",x,", y =",y)
 return

Association clic/action
canvas.bind("<Button-1>", action_clic_souris)

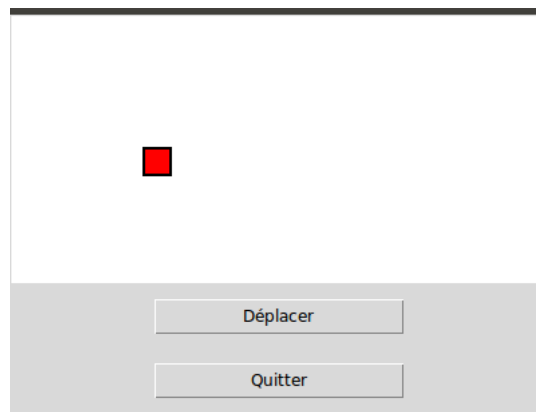
Lancement
root.mainloop()
```

Voici quelques explications :

- La création de la fenêtre est habituelle. Le programme se termine par le lancement de la fenêtre avec la commande `mainloop()`.
- Le premier point clé est d'associer un clic de souris à une action, c'est ce que fait la ligne  
`canvas.bind("<Button-1>", action_clic_souris)`  
 Chaque fois que le bouton gauche de la souris est cliqué, Python exécute la fonction `action_clic_souris`. Note qu'il n'y a pas de parenthèses pour l'appel à la fonction.
- Second point clé : la fonction `action_clic_souris` récupère les coordonnées du clic et ici ensuite fait deux choses : elle affiche un petit rectangle à l'endroit du clic et affiche dans la fenêtre du terminal les coordonnées  $(x, y)$ .
- Les coordonnées  $x$  et  $y$  sont exprimées en pixels ;  $(0, 0)$  désigne le coin en haut à gauche de la fenêtre (la zone délimitée par `canvas`).

## 11.5. Mouvement

Voici un programme qui fait se déplacer un petit carré en le faisant rebondir sur les bords de la fenêtre.



Voici les points principaux :

- Un objet `rect` est défini, c'est une variable globale, de même que ses coordonnées `x0`, `y0`.
- Cet objet est (un petit peu) déplacé par la fonction `deplacer()` qui décale le rectangle de  $(dx, dy)$ .
- Le point clé est que cette fonction sera exécutée une nouvelle fois après un court laps de temps. La commande :

```
canvas.after(50,deplacer)
```

demande une nouvelle exécution de la fonction `deplacer()` après un court délai (ici 50 millisecondes).

- La répétition de petits déplacements simule le mouvement.

```
from tkinter import *
```

```
Largeur = 400
```

```
Hauteur = 200
```

```
root = Tk()
```

```
canvas = Canvas(root, width=Largeur, height=Hauteur, background="white")
```

```
canvas.pack(fill="both", expand=True)
```

```
Les coordonnées et la vitesse
```

```
x0, y0 = 100,100
```

```
dx = +5 # Vitesse horizontale
```

```
dy = +2 # Vitesse verticale
```

```
Le rectangle à déplacer
```

```
rect = canvas.create_rectangle(x0,y0,x0+20,y0+20,width=2,fill="red")
```

```
Fonction principale
```

```
def deplacer():
```

```
 global x0, y0, dx, dy
```

```
 x0 = x0 + dx # Nouvelle abscisse
```

```
 y0 = y0 + dy # Nouvelle ordonnée
```

```
 canvas.coords(rect,x0,y0,x0+20,y0+20) # Déplacement
```

```
 if x0 < 0 or x0 > Largeur:
```

```
 dx = -dx # Changement de sens horizontal
```

```
 if y0 < 0 or y0 > Hauteur:
```

```
 dy = -dy # Changement de sens vertical
```



```
 canvas.after(50,deplacer) # Appel après 50 millisecondes

 return

Fonction pour le bouton
def action_deplacer():
 deplacer()
 return

Boutons
bouton_couleur = Button(root,text="Déplacer", width=20, command=action_deplacer)
bouton_couleur.pack(pady=10)

bouton_quitter = Button(root,text="Quitter", width=20, command=root.quit)
bouton_quitter.pack(side=BOTTOM, pady=10)

root.mainloop()
```

## Notes et références

Tu trouveras ici des commentaires et des indications de lectures sur chacune des activités.

### Ressources générales

- *Apprendre à programmer avec Python 3* de Gérard Swinnen, aux éditions Eyrolles. C'est un des livres de référence en français pour débiter. La seconde moitié du livre contient des notions beaucoup plus avancées. Le livre est disponible gratuitement en téléchargement, selon la licence *Creative Commons BY-NC-SA* : [inforef.be/swi/python.htm](http://inforef.be/swi/python.htm)
- La documentation officielle de Python contient des tutoriels et les explications de chacune des fonctions. Malheureusement certaines pages sont en anglais [docs.python.org/fr/3/](http://docs.python.org/fr/3/)
- *Wikipédia* est une source fiable pour en savoir plus sur certaines notions (principalement les projets) mais le niveau n'est pas toujours adapté à un lycéen.
- *Internet* et en particulier les forums ont bien souvent la réponse aux questions que tu te poses !
- Les plus fondus d'entre vous peuvent participer au *projet Euler* (en anglais) qui propose une liste d'énigmes mathématico-informatique. Accrochez-vous ! [projecteuler.net](http://projecteuler.net)

### 1. Premiers pas

L'apprentissage d'un langage de programmation peut être très difficile. C'est assez dur d'apprendre tout seul dans son coin. Il n'est pas rare de rester bloquer plusieurs heures pour une bête erreur de syntaxe. Il faut commencer modestement, ne pas hésiter à recopier du code déjà écrit par d'autres, être persévérant et demander de l'aide rapidement !

### 2. Tortue (Scratch avec Python)

L'idéal c'est de bien maîtriser *Scratch* avant de s'attaquer à Python ! Pour ceux qui ont zappé *Scratch*, il y a les activités *Scratch au collège*. Le livre et les vidéos sont disponibles ici :

[exo7.emath.fr](http://exo7.emath.fr) et [youtube.com/ScratchAuCollege](http://youtube.com/ScratchAuCollege)

Python propose un module *Tortue* qui fonctionne sur le même principe que le déplacement avec *Scratch*. Bien sûr au lieu de déplacer des blocs, il faut écrire le code ! C'est un bon exercice de transcrire en Python toutes les activités que tu sais faire avec *Scratch*.

### 3. Si ... alors ...

On attaque vraiment la programmation avec le test « si/sinon ». L'ordinateur agit donc d'une façon ou d'une autre selon la situation. Ce n'est plus un automate qui fait toujours la même chose. En plus avec l'entrée au clavier, on peut commencer à avoir des programmes interactifs.

Les erreurs de syntaxe classiques sont :

- oublier les deux points après `if condition:` ou bien `else:`,
- mal indenter les blocs.

Ces erreurs seront signalées par Python avec le numéro de la ligne où il y a un problème (normalement votre éditeur place le curseur sur la ligne fautive). Par contre si le programme se lance mais qu'il ne fait pas ce qu'il faut, c'est sûrement la condition qui est mal formulée. C'est plus compliqué de bien comprendre la condition qui convient : un peu de logique et de réflexion avec papier/crayon sont les bienvenues. Certains éditeurs Python permettent aussi une exécution « pas à pas » du programme.

On reviendra par la suite sur le « vrai » et le « faux ». Il existe aussi le test

```
if ... elif ... elif ... else ...
```

qui permet d'enchaîner les tests et qui n'est pas abordé ici. Bien comprendre `if ... else ...` est suffisant !

### 4. Fonctions

Assez rapidement il faut comprendre la structure d'un programme informatique : on décompose le programme en blocs de définitions et d'actions simples. On regroupe des actions simples dans des actions intermédiaires. Et à la fin le programme principal consiste juste à exécuter quelques bonnes fonctions.

Les arguments/paramètres d'une fonction sont d'un apprentissage délicat. On peut définir une fonction par `def fonc(x):` et l'appeler par `fonc(y)`. Le `x` correspond à une variable mathématique muette. En informatique, on préfère parler de la **portée** de la variable qui peut être locale ou globale.

Ce que l'on peut retenir, c'est que tout ce qui passe à l'intérieur d'une fonction n'est pas accessible en dehors de la fonction. Il faut juste utiliser ce que renvoie la fonction. Il faut s'interdire l'utilisation de `global` dans un premier temps.

Dernier point, il est vraiment important de commenter abondamment ton code et de bien expliquer ce que font tes fonctions. Si tu définis une fonction `fonc(x)` : prévois trois lignes de commentaires pour (a) dire ce que fait cette fonction, (b) dire quelle entrée est attendue (`x` doit être un entier ? un nombre flottant ? positif?...), (c) dire ce que renvoie la fonction.

Il faut aussi commenter les points principaux du code, donner des noms bien choisis aux variables. Tu seras bien content d'avoir un code lisible lorsque tu reliras ton code plus tard !

Un bon informaticien devrait vérifier que le paramètre passé en argument vérifie bien l'hypothèse attendue. Par exemple si `fonc(x)` est définie pour un entier `x` et que l'utilisateur transmet une chaîne de caractères, alors un gentil message d'avertissement devrait être envoyé à l'utilisateur sans faire échouer tout le programme ! Les commandes `assert`, `try/except` permettent de gérer ce genre de problème. Pour notre part, nous nous abstenons de ces vérifications en supposant que l'utilisateur/programmeur utilise les fonctions et les variables en bonne intelligence !

## 5. Arithmétique – Boucle tant que – I

Nous utiliserons uniquement Python3. Si vous ne savez pas quelle version vous avez, tapez `7/2` : Python3 renvoie 3.5 alors que Python2 renvoie 3 (dans la version 2, Python considérait que la division de deux entiers devait renvoyer un entier).

Avec Python3 c'est plus clair, `a / b` est la division habituelle (des nombres à virgules) alors que `a // b` est la division euclidienne entre deux entiers et renvoie le quotient.

## 6. Chaînes de caractères – Analyse d'un texte

Manipuler les chaînes permet de faire des activités sympas et quitter un peu le monde mathématique. Tu peux programmer des quiz, des programmes qui discutent avec l'utilisateur... Les chaînes de caractères sont surtout une bonne introduction à la notion de liste, qui est un outil essentiel par la suite.

## 7. Listes I

Python est particulièrement souple et agile pour l'utilisation des listes. Les « vieux » langages n'autorisaient souvent que les listes contenant un seul type d'élément, et pour parcourir une liste il fallait toujours procéder ainsi :

```
for i in range(len(liste)):
 print(liste[i])
```

Alors que :

```
for element in liste:
 print(element)
```

est beaucoup plus naturel !

Le tri d'une liste est une opération fondamentale en informatique. Imaginerait-on un dictionnaire contenant 60 000 mots, mais non classés par ordre alphabétique ? Ordonner une liste est une opération difficile, il existe beaucoup d'algorithmes de tri. L'algorithme du tri à bulles, présenté ici est l'un des plus simples. Programmer des algorithmes plus rapides au lycée est un beau challenge : il faut comprendre la récursivité et la notion de complexité.

## 8. Statistique – Visualisation de données

Si tous les lecteurs arrivent à programmer les calculs de somme, moyenne, écart-type, médiane... et leur visualisation, alors l'objectif de ce livre est atteint ! Cela prouve une bonne compréhension des outils mathématiques et informatiques de base.

Le module `tkinter` permet un affichage graphique. Pour commencer, il faut recopier les lignes d'un code qui fonctionne sans trop se poser de questions, puis l'adapter à ses besoins.

## 9. Fichiers

Les fichiers permettent de faire communiquer Python avec le monde extérieur : on peut par exemple récupérer un fichier de notes pour en calculer les moyennes et produire un bulletin pour chaque élève. Pour les plus avancés d'entre vous les fichiers sont une bonne occasion d'utiliser la gestion des erreurs avec `try/except`.

Les activités sur les images sont sympathiques et ces formats d'images seront utilisés par la suite, même si ce format a le gros désavantage de produire des fichiers de grande taille. Néanmoins ils sont standards et reconnus par les logiciels d'images (*Gimp* par exemple). Notez que certains logiciels écrivent des fichiers avec une seule donnée par ligne (ou bien toutes les données sur une seule ligne). C'est un bon exercice d'implémenter la lecture de tous les formats possibles !

## 10. Arithmétique – Boucle tant que – II

L'arithmétique en général et les nombres premiers en particulier ont une très grande importance en informatique. Ce sont eux qui sont à la base de la cryptographie moderne et assurent la sécurité des transactions sur internet.

Les algorithmes présentés ici sont bien sûr élémentaires. Il existe des techniques sophistiquées pour savoir si un nombre de plusieurs centaines de chiffres est premier ou pas en quelques secondes. Cependant, factoriser un entier de grande taille reste un problème difficile.

Un ordinateur montre sa puissance quand il manipule une grande quantité de nombres ou bien de très grands nombres. Avec l'arithmétique, on a les deux en même temps !

## 11. Binaire I

La première difficulté avec l'écriture binaire c'est de bien faire la différence entre un nombre et l'écriture du nombre. Nous sommes tellement habitués à l'écriture décimale que l'on a oublié son origine, 1234 c'est juste  $1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$ .

Comme les ordinateurs travaillent avec des 0 et 1 il faut être à l'aise avec l'écriture binaire. Le passage à l'écriture binaire n'est pas très difficile, il est tout de même préférable de faire quelques exemples à la main avant de s'attaquer à la programmation.

On utilisera l'écriture binaire pour d'autres problèmes sur le principe suivant : vous avez 4 interrupteurs alors 1.0.0.1 signifie que vous actionnez le premier et le dernier interrupteur et pas les autres.

## 12. Listes II

Les listes sont tellement utiles que Python possède toute une syntaxe efficace pour les gérer : le tranchage des listes et les listes par compréhension. On pourrait bien sûr s'en passer (c'est d'ailleurs le cas pour la plupart des autres langages) mais ce serait dommage. Nous aurons aussi besoin de listes de listes et en particulier de tableaux à deux dimensions pour afficher de belles images.

## 13. Binaire II

Il y a 10 sortes de personnes, celles qui comprennent le binaire et les autres !

## 14. Probabilités – Paradoxe de Parrondo

On attaque les projets avec un peu de probabilité et un joli paradoxe, surprenant (comme tous les paradoxes), mais en plus découvert récemment.

Cette activité est calquée sur l'article « Paradoxe de Parrondo » par Hélène Davaux (*La gazette des mathématiciens*, 2017).

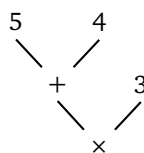
## 15. Chercher et remplacer

Chercher et remplacer sont deux actions tellement courantes que l'on ne se rend pas toujours compte de leur force. Tout d'abord c'est un bon exercice de programmer soi-même les opérations de recherche et de remplacement. Il existe une version sophistiquée de ces opérations qui s'appellent les expressions rationnelles (*regex*). C'est un langage à part entière, très puissant, mais un peu ésotérique.

Il y a aussi les mathématiques du « chercher/remplacer » ! Les activités proposées sont des exemples qui illustrent le théorème principal de l'article *A complete characterization of termination of  $0^p 1^q \rightarrow 1^r 0^s$* , par H. Zantema et A. Geser (AAECC, 2000) qui est introduit par Pierre Lescanne sur le site « Images de mathématiques » : [Est-ce que ça s'arrête ?](#)

## 16. Calculatrice polonaise – Piles

L'écriture polonaise (le nom exact est « écriture polonaise inverse ») est une autre façon d'écrire les opérations algébriques. Encore une fois, le plus dur c'est de s'adapter à ce changement d'écriture. Cela permet de voir les opérations (et leurs priorités) sous un nouveau jour. Une autre vision intéressante est de voir une opération sous la forme d'un d'arbre binaire :



qui représente  $(5 + 4) \times 3$  ou encore en notation polonaise  $5\ 4\ +\ 3\ \times$ .

On a essayé de repousser au maximum le changement d'une variable globale dans une fonction, mais ici il est naturel. L'utilisation de `global` est à éviter en général.

La notion de pile est une façon très simple de structurer et d'accéder aux données. C'est pourtant la bonne façon de gérer une expression avec parenthèses ! L'analogie avec la gare de triage devrait être éclairante. Nous retrouverons les piles dans l'activité sur les L-systèmes.

Une pile fonctionne sur le principe « dernier entré, premier sorti » (*fil* pour *first in, last out*). Une autre gestion des données possible est sur le principe « premier entré, premier sorti » (*fifo* pour *first in, first out*) comme dans une file d'attente.

## 17. Visualiseur de texte – Markdown

Le but de cette fiche est double : découvrir le *Markdown* qui est un langage très pratique pour formater un texte, mais aussi comprendre la justification d'un paragraphe.

Évidemment le langage *Markdown* possède davantage de balises que celles présentées ici. On pourrait poursuivre le projet en réalisant la justification avec des polices de tailles et de formes différentes, voir même créer un petit traitement de texte complet.

## 18. L-système

On retrouve le thème « chercher/remplacer » mais cette fois avec une vision géométrique. Les figures obtenues sont belles, faciles à programmer à l'aide de la tortue, mais le plus joli c'est de voir le tracé en direct des L-systèmes. Pour les L-systèmes définis par des expressions contenant des crochets on retrouve la notion de pile.

Les formules sont tirées du livre *The algorithmic beauty of plants*, par P. Prusinkiewicz et A. Lindenmayer (Springer-Verlag, 2004) en accès libre ici : [The algorithmic beauty of plants \(pdf\)](#).

Les illustrations qui débutent chaque partie de ce livre sont des itérations du L-système appelé la courbe de Hilbert et défini par :

```
depart = "X" regle1 = ("X","gYAdXAXdAYg") regle2 = ("Y","dXAgYAYgAXd")
```

## 19. Images dynamiques

Cette activité est calquée sur l'article « Images brouillées, images retrouvées » par Jean-Paul Delahaye et Philippe Mathieu (*Pour la Science*, 1997). Cet article s'intéresse en plus au calcul du nombre d'itérations qu'il faut avant de retrouver l'image de départ. La notion mathématique sous-jacente est celle de *permutation* : une transformation bijective d'un ensemble fini (ici l'ensemble des pixels) dans lui-même.

## 20. Jeu de la vie

Un grand classique de l'informatique amusante ! On trouvera sur internet des dizaines de sites avec des constructions aux propriétés incroyables et plein d'autres idées. Mais le plus fascinant reste que des règles extrêmement simples conduisent à des comportements complexes qui ressemblent à la vie et la mort des cellules.

## 21. Graphes et combinatoire de Ramsey

Les graphes sont des objets très courants en mathématique et en informatique. Le problème présenté ici est simple et amusant. Mais ce qu'il faut peut être retenir de cette fiche, c'est l'accroissement de la difficulté avec le nombre de sommets. Ici on effectue les calculs jusqu'à 6 sommets et on ne peut pas aller beaucoup plus loin avec notre méthode de vérification exhaustive.

Un grand mathématicien Paul Erdős a déclaré que si des extra-terrestres débarquaient sur Terre en menaçant de détruire notre planète sauf si on savait résoudre le problème des 5 amis/5 étrangers, alors en mobilisant tous les ordinateurs et les mathématiciens du monde on arriverait à s'en sortir (on sait que la réponse est entre 43 et 48 personnes). Par contre si les extra-terrestre nous demandaient de résoudre le problème pour 6 amis/6 étrangers alors le plus simple serait de se préparer à la guerre !

## 22. Bitcoin

Avant de placer toutes ses économies dans des *bitcoins* mieux vaut en comprendre le fonctionnement ! Les activités présentées ici ont pour but de présenter une version (très) simplifiée de la *blockchain* qui est à la base de cette monnaie virtuelle. Le principe de la *blockchain* et de la preuve de travail ne sont pas si compliqués, tu trouveras des explications plus détaillées dans les articles de Jean-Paul Delahaye parus dans la revue *Pour la science* :

- Bitcoin, la cryptomonnaie (2013) ([www.lifl.fr/~jdelahay/pls/2013/241.pdf](http://www.lifl.fr/~jdelahay/pls/2013/241.pdf))
- Les preuves de travail (2014) ([crystal.univ-lille.fr/~jdelahay/pls/2014/245.pdf](http://crystal.univ-lille.fr/~jdelahay/pls/2014/245.pdf))
- Du bitcoin à Ethereum : l'ordinateur-monde (2016) ([crystal.univ-lille.fr/~jdelahay/pls/2016/276.pdf](http://crystal.univ-lille.fr/~jdelahay/pls/2016/276.pdf))

## 23. Constructions aléatoires

Ces constructions aléatoires sont tout d'abord des récréations informatiques qui produisent des jolies figures, toutes ressemblantes mais toutes différentes. Mais elles font aussi l'objet de travaux mathématiques modernes et difficiles. Martin Hairer a obtenu la médaille Fields en 2014 pour l'étude de la frontière supérieure de nos blocs qui tombent, dont la forme est régie par une équation, appelée « équation KPZ ». Une bonne activité bonus serait de faire tomber des blocs du jeu *Tetris* à la place des petits carrés.



## Remerciements

Je remercie Stéphanie Bodin pour ses encouragements et pour avoir testé les activités des premières parties. Je remercie vivement Michel Bodin pour avoir testé toutes les premières activités et sa relecture. Merci à François Recher pour son enthousiasme et sa relecture attentive du livre. Merci à Éric Wegrzynowski pour sa relecture et ses idées pour les activités sur les images. Merci à Philippe Marquet pour sa relecture des premiers chapitres. Merci à Kroum Tzanev pour la maquette du livre.

Sur le site Exo7 vous pouvez récupérer les codes et fichiers sources.  
Les vidéos sont accessibles depuis la chaîne *Youtube* : « [Python au lycée](#) ».



Ce livre est diffusé sous la licence *Creative Commons – BY-NC-SA – 4.0 FR*.  
Sur le site Exo7 vous pouvez télécharger gratuitement le livre en couleur.

- $\pi$ , 5
- \*\***, 2
- +=**, 36
- =**, 3, 21
- #**, 3
- !=**, 21
- //**, 33
- <=**, 21
- ==**, 21
- >=**, 21
- %**, 33
  
- abs, 6
- ADN, 43
- affectation, 3, 21
- and, 21
- angle, 76
- append, 50
- argument, 27, 126
- ascii, 45
  
- bin, 85
- binaire, 82, 94, 149
- bitcoin, 154
- bits, 83
- blockchain, 156
- booléen, 21
- boucle
  - pour, 6, 42, 51
  - quitter, 80
  - tant que, 34
- bouton, 63
- break, 80
  
- caractère, 40
- carré magique, 91
- ceil, 5
- chaîne, 40, 69
  
- caractères, 149
- majuscule, 26
- regrouper, 109
- séparer, 109
- chercher, 101
- choice, 69
- chr, 45
- clic, 143
- close, 68
- codage des caractères, 45
- commentaire, 3
- concaténation, 26, 40, 51
- conjecture de Goldbach, 77
- cos, 5
- crible d'Ératosthène, 54
- csv, 70
  
- def, 24
- degrés, 76
- del, 53
- dépiler, 106, 130
- distance, 42, 74
- diviseur, 36
- division euclidienne, 2, 33
- docstring, 27
- down, 10
  
- écart-type, 60
- écriture
  - binaire, 82, 94, 149
  - décimale, 21, 82
  - polonaise, 111
- else, 18
- empiler, 106
- entrée, 19
- espérance, 98
- et, 21, 95

- eval, [117](#)
- expression rationnelle, [102](#)
- facteur premier, [54](#)
- False, [21](#)
- fenêtre, [60](#)
- fichier, [68](#)
- find, [101](#)
- flocon de Koch, [127](#)
- floor, [5](#)
- fonction, [24](#)
  - argument, [27](#)
  - argument optionnel, [126](#)
  - de hachage, [157](#)
  - docstring, [27](#)
  - paramètre, [24](#)
  - return, [27](#), [174](#)
- fonte, [119](#)
- for, [6](#), [42](#), [51](#)
- forward, [9](#)
- gcd, [5](#)
- global, [107](#)
- goto, [10](#)
- graphe, [145](#)
- graphique, [9](#), [55](#), [60](#)
- hasard, [19](#), [98](#)
- hash, [157](#)
- if, [18](#)
- image, [71](#), [133](#)
- import, [5](#)
- in, [6](#), [42](#), [88](#), [101](#)
- indentation, [6](#)
- index, [88](#), [101](#)
- input, [19](#)
- int, [19](#), [69](#)
- intérêts, [51](#)
- isdigit, [109](#)
- jeu de la vie, [139](#)
- join, [109](#)
- left, [9](#)
- len, [40](#), [51](#)
- list, [7](#), [149](#)
- liste, [50](#), [88](#)
  - ajouter, [50](#), [51](#)
  - de listes, [89](#)
  - fusionner, [51](#)
  - inverser, [52](#)
  - longueur, [51](#)
  - par compréhension, [88](#)
  - sous-liste, [52](#)
  - supprimer, [52](#)
  - trancher, [88](#)
  - trier, [53](#)
- lois de Morgan, [96](#)
- L-système, [125](#)
- majuscule, [26](#)
- markdown, [119](#)
- math, [5](#)
- matplotlib, [55](#)
- max, [59](#)
- médiane, [65](#)
- min, [59](#)
- minage, [160](#)
- module, [5](#)
  - math, [5](#)
  - matplotlib, [55](#)
  - random, [19](#)
  - re, [102](#)
  - time, [155](#)
  - timeit, [37](#)
  - tkinter, [60](#), [118](#), [143](#), [168](#)
  - turtle, [9](#)
- modulo, [2](#), [33](#)
- moyenne, [59](#), [66](#)
- nombre flottant, [2](#)
- nombre premier, [36](#), [54](#)
- non, [21](#), [95](#)
- None, [44](#)
- not, [21](#)
- not in, [42](#)
- open, [68](#)
- opération logique, [21](#), [94](#), [95](#)
- or, [21](#)
- ord, [45](#)
- ou, [21](#), [95](#)
- palindrome, [43](#), [94](#)
- paramètre, [24](#)
- pbm/pgm/ppm, [71](#), [135](#)
- pgcd, [5](#)
- pi, [5](#)
- pile, [130](#)
- plot, [55](#)

pop, [130](#)  
ppcm, [5](#)  
preuve de travail, [154](#)  
print, [3](#)  
puissance, [2](#)  
puissances de 2, [83](#)  
  
quartiles, [65](#)  
quotient, [33](#)  
  
racine carrée, [5](#), [79](#)  
radians, [76](#)  
randint, [19](#)  
random, [19](#)  
range, [7](#)  
regex, [102](#)  
remove, [52](#)  
remplacer, [101](#), [125](#)  
replace, [101](#)  
reste, [2](#), [33](#)  
return, [24](#), [27](#), [174](#)  
reverse/reversed, [52](#)  
right, [9](#)  
round, [5](#)  
rvb/rgb, [73](#), [74](#)  
  
shuffle, [91](#)  
si/alors, [18](#)  
sin, [5](#)  
sinon, [18](#)  
sort/sorted, [53](#)  
  
souris, [143](#)  
split, [109](#)  
sqrt, [5](#)  
str, [19](#), [40](#), [68](#)  
sum, [59](#)  
  
tableau, [89](#)  
temps de calcul, [37](#), [155](#)  
time, [155](#)  
timeit, [37](#)  
tir balistique, [56](#)  
tkinter, [60](#), [118](#), [143](#), [168](#)  
tortue, [9](#), [28](#), [125](#)  
tri, [53](#)  
triangle, [22](#)  
triangle de Sierpinski, [12](#), [128](#)  
True, [21](#)  
try/except, [81](#)  
turtle, [9](#)  
  
unicode, [45](#), [121](#)  
up, [10](#)  
upper, [26](#)  
  
valeur absolue, [6](#)  
variable, [3](#)  
    globale, [107](#)  
    locale, [31](#)  
variance, [60](#)  
  
while, [34](#)  
write, [68](#)