ECG1

Mathématiques Appliquées

TP Python

V. Goyheneche ❖ Lycée Louis Thuillier ❖ 2026 – 2027

Table des matières

Introduction

1. Présentation de ce TP

a) Niveau attendu

Le niveau en programmation que vous avez chacun en abordant l'ECG présente sûrement de grands écarts, en fonction de votre pratique au lycée, de votre goût personnel pour l'algorithmique et l'informatique en général... **À chacun de progresser à son niveau !**
Il n'est pas nécessaire d'être un expert pour obtenir des points aux concours, mais une connaissance fine des principes de la programmation peut être un vrai plus pour se démarquer.

Ainsi, chaque TP comportera des exercices d'approfondissement, destinés à ceux d'entre vous qui sont le plus en avance. Si le contenu d'un TP vous semble facile, n'hésitez pas à traiter rapidement les exercices d'application pour attaquer ceux qui demandent plus de réflexion.

En particulier la première partie sera, pour certains d'entre vous, des rappels de notions vues au lycée :

- environnement de programmation (EduPython, Pyzo)
- gestion et types des variables
- utilisation des boucles et instructions conditionnelles : `if`, `for`, `while`
- utilisation d'une `fonction` Python
- calculer l'image d'un nombre par une fonction donnée, calculer le terme d'une suite

Même s'il s'agit (en principe) de rappels, ces notions sont fondamentales et ne sont pas à négliger ! Elles seront indispensables dans la plupart des exercices donnés en DS, puis en concours.

b) Au programme en ECG1

Le programme d'ECG1 à proprement parler commence à la partie II. Il est lié au programme de mathématiques et traite les thèmes suivants :

- utilisation de listes ;
- statistiques, analyse de données ;
- approximation numérique ;
- graphes (utilisation de matrices) ;
- simulation de phénomènes aléatoires.

c) Et en ECG2 ?

L'an prochain, le travail en `Python` se fera dans la continuité des TP de première année. En lien avec le programme, deux nouvelles notions seront travaillées (chaînes de Markov et intervalles de confiance).

S'y ajoutera l'étude des *bases de données* avec le langage SQL, dont le but est d'organiser, de stocker et d'interroger de gros fichiers de données.

d) Bibliothèques

Les `bibliothèques` utilisées en `Python`, conformément au programme, sont :

- `numpy` pour les calculs mathématiques
- `numpy.linalg` pour les opérations sur les matrices
- `numpy.random` pour simuler le hasard
- `matplotlib.pyplot` pour tracer des courbes et graphiques
- `pandas` pour les statistiques

Tous les exercices des TP seront réalisés à partir de ces 5 bibliothèques, aucune autre n'est nécessaire. Voir en annexe la liste des commandes figurant au programme.

À noter. En informatique, le hors-programme est accepté au concours. Autrement dit : si vous connaissez certaines commandes qui donnent le résultat recherché, votre réponse sera validée (même si elles ne figurent pas dans le programme officiel).
Attention : ce n'est pas valable pour les mathématiques. Une méthode hors programme ne sera pas valorisée.

2. Python, qu'est-ce que c'est ?

a) Les débuts

Le langage `Python` a été créé en 1989 par Guido van Rossum, un informaticien néerlandais. À l'origine, il a développé `Python` pour améliorer le langage de programmation `ABC`, qui était utilisé principalement pour l'éducation.

Guido van Rossum avait pour objectif de créer un langage de programmation facile à lire, à écrire et à maintenir, tout en étant puissant et flexible. Il a baptisé son nouveau langage de programmation « Python » en hommage à la troupe de comédiens britanniques Monty Python, qu'il admirait.

b) Utilisation

Le langage `Python` est diffusé en Open Source. Il est donc en particulier téléchargeable gratuitement. Par ailleurs, de nombreux modules ont été développés, ce qui permet de réaliser des tâches variées.

Il est multi-plateformes : on peut tout aussi bien l'utiliser sur Windows, MacOS ou Linux.

Il est principalement utilisé dans les domaines de la *Data Science* (traitement des données) et du *Machine Learning* (intelligence artificielle), mais aussi pour la création d'applications ou de sites web.

Il est relativement facile à apprendre car sa syntaxe est proche de l'anglais courant. Par contre ce n'est pas le langage le plus efficace en vitesse de calcul. On utilise d'autres langages pour des tâches plus complexes.

À titre de comparaison, voici le même algorithme (un calcul de racine carrée) implémenté dans 3 langages différents.

En Python :

```python
import math
# Demande un nombre à l'utilisateur
nombre = float(input("Entrez un nombre: "))

# Vérifie si le nombre est positif
if nombre < 0:
    print("Le nombre entré est négatif.")
else:
    # Calcul de la racine carrée
    racine = math.sqrt(nombre)
    # Affiche le résultat
    print("La racine carrée de",nombre, "est ",
    racine)
```

En Javascript :

```javascript
// Demande un nombre à l'utilisateur
let nombre = parseFloat(prompt("Entrez un nombre
    : "));

// Vérifie si le nombre est positif
if (nombre < 0) {
    alert("Le nombre entré est négatif.");
} else {
    // Calcul de la racine carrée
    let racine = Math.sqrt(nombre);
    // Affiche le résultat
    alert(`La racine carrée de ${nombre} est ${
    racine}`);
}
```

En C :

```c
#include <stdio.h>
#include <math.h>

int main() {
    float nombre, racine;

    // Demande un nombre à l'utilisateur
    printf("Entrez un nombre: ");
    scanf("%f", &nombre);

    // Vérifie si le nombre est positif
    if (nombre < 0) {
        printf("Le nombre entré est négatif.\n")
    ;
    } else {
        // Calcul de la racine carrée
        racine = sqrt(nombre);
        // Affiche le résultat
        printf("La racine carrée de %.2f est %.2
    f\n", nombre, racine);
    }

    return 0;
}
```

c) Pour en savoir plus

Si vous voulez aller au-delà du programme d'ECG, il existe de nombreux tutos pour découvrir d'autres fonctionnalités de Python.

Voici quelques ressources, évidemment cette liste est loin d'être exhaustive!

https://python.doctor/page-cours-python-avance-documentation-francaise
https://zestedesavoir.com/tutoriels/954/notions-de-python-avancees/

3. Commencer à programmer

a) Installer Python

Je vous conseille fortement d'installer Python sur votre ordinateur personnel, si vous en possédez un. Cela vous permettra de retravailler certains exercices réalisés en classe, et de tester les programmes demandés en DM.

→ Sous Windows : le plus simple est d'installer Édu Python, c'est ce logiciel qui sera utilisé au lycée. Aller sur https://edupython.tuxfamily.org

→ Sous macOS : on peut utiliser l'application Pyzo. Attention, il faut d'abord installer le langage Python sur le lien suivant : https://www.python.org/downloads/mac-osx/
puis installer Pyzo sur la page suivante : https://pyzo.org/start.html

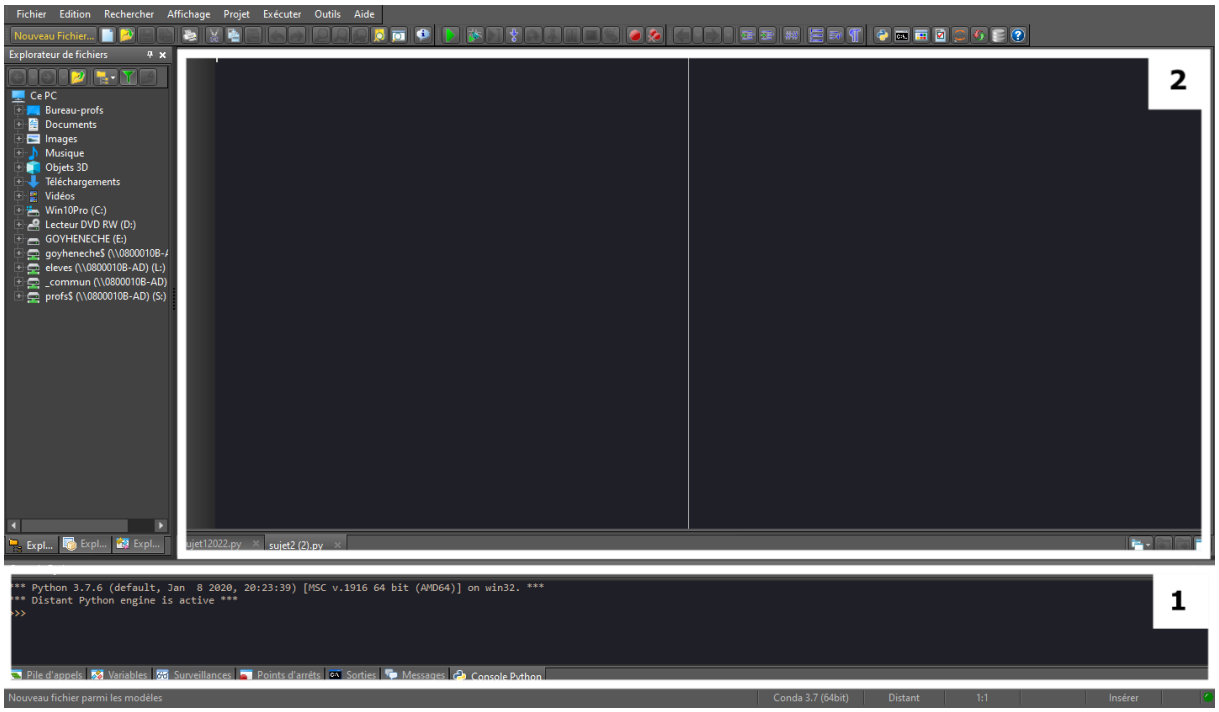
D'autres solutions, permettant d'écrire des programmes directement en ligne :
https://www.lelivrescolaire.fr/outils/console-python
https://replit.com/languages/python3

b) Environnement Python

Les logiciels ci-dessus possèdent une interface avec deux éléments principaux :

→ 1 la *console* (*shell* en anglais) : elle sert de calculatrice, affiche les résultats des programmes et les erreurs. On l'utilise quand on a besoin d'écrire de courtes instructions d'une ou deux lignes.
Pour exécuter une ligne tapée dans la console, on appuie simplement sur `entrée`

→ 2 l'*éditeur*, qui permet d'écrire des programmes. Ceux-ci peuvent être sauvegardés et modifiés.
Pour exécuter un programme écrit dans l'éditeur, on clique sur la flèche verte, ou on utilise le raccourci `Ctrl + E`



Les tailles de ces deux éléments peuvent être adaptées : pensez à agrandir la console si vous effectuez plusieurs calculs successifs dessus.

Vocabulaire. Dans la console apparaît la commande >>>, qu'on appelle l'**invite**. Celle-ci signifie que Python est prêt à exécuter les instructions que vous lui donnez.
Si l'invite n'apparaît pas, inutile d'écrire, c'est que Python est encore occupé.

c) Algorithme / programme

On confond souvent ces deux termes. Leur signification est en effet assez proche, avec quelques nuances.

On appelle *algorithme* une méthode pour résoudre un problème. Par exemple vous connaissez l'algorithme permettant de déterminer les racines d'un polynôme de la forme ax^2+bx+c : on suit les étapes suivantes

1. calculer $\Delta = b^2 - 4ac$
2. — si $\Delta > 0$, déterminer $x_1 = \dfrac{-b-\sqrt{\Delta}}{2a}$ et $x_2 = \dfrac{-b+\sqrt{\Delta}}{2a}$.
   — si $\Delta = 0$, déterminer $x_0 = \dfrac{-b}{2a}$.
   — si $\Delta < 0$, il n'y a aucune racine réelle.

Si chacun de nous comprend bien ces instructions, un ordinateur ne saura pas les interpréter. Il faut donc écrire un *programme*, autrement dit une traduction dans un langage qu'un ordinateur comprend.

Dans un exercice de programmation, on suit souvent ces deux étapes :

1. Je traduis le problème en *algorithme* : je réfléchis aux méthodes mathématiques pour le résoudre (soit de tête, soit sur papier)
2. Je transcris ma méthode sous forme d'un *programme*, que Python saura comprendre.

✔ Ne pas négliger le travail sur papier ! La programmation commence souvent au brouillon, ce qui permet de mettre ses idées au clair. Inutile de commencer à coder si on n'a aucune idée des étapes à suivre dans la programmation.

Première partie

Manipulations de base

TP 1

Introduction à Python par l'exemple : variables, types, affectations

Pour commencer nous allons voir comment Python manipule les différentes données et peut les enregistrer.

1. Créer / modifier une variable

Code (Comprendre le signe =)

```
x = 3+2
```

Cette instruction est interprétée par Python en 2 étapes **successives** :

1. il calcule l'expression à droite du symbole =
2. il enregistre le résultat dans la variable écrite à gauche

Exercice 1 : Quelles sont les opérations valides ?

```
x - 3 = 2
```

```
2y = 10
```

```
x = x+1
```

```
y = 5
```

```
y2 = 10
```

```
xz = 45/7
```

Exercice 2 : Compléter le tableau avec les valeurs prises par les variables à chaque étape. On suppose que les instructions de la première colonne sont saisies successivement dans la console.
Attention : l'une de ces lignes produit une erreur !

code	x	y	a	b	c
`x = 3`					
`y = -6`					
`a = x+y`					
`b = a + x`					
`a = b+c`					
`c = 2*x + a`					
`a = a+b`					
`x = y`					
`y = x`					

2. Types de variables

Mémo 1

À chaque fois que Python enregistre une variable, il lui affecte un *type*. Par exemple pour un nombre, il peut considérer que c'est un entier ou que c'est un réel : selon ce choix, il pourra effectuer différentes actions.
Voici les principaux types utilisés en Python

float
Nombre « flottant »
Pour simplifier : réel
Utilisé pour la plupart des calculs
Affiché avec un point décimal

int
Nombre entier
Utilisé pour les répétitions (boucles)

str
« string » : chaîne de caractères
Pour simplifier : texte
Utilisé pour les affichages, Python n'interprète pas ce qui est écrit
Affiché entre guillemets

bool
« Booléen »
Pour simplifier : vrai/faux
Valeur logique qui prend uniquement `True` ou `False` comme valeur
Utilisé pour les tests logiques (`if`, `while`)

Exercice 3 : Quels sont les types attribués à ces variables? On suppose que ces instructions sont données à la suite, colonne par colonne.

```
x = 3
y = 5.5
z = 3.0
```

```
a = 6,5
b = x+y
c = x+5
```

```
x = b
d = 'bonjour'
e = 'x'
```

```
f = y*2
g = '23'
```

À retenir :

Exercice 4 : Donner la valeur et le type des variables à la fin de ces programmes.

```
x = 5.5
y = 8
y = 5*y - 2.5
x = x*y
```

```
a = 2
b = 3*a
a = b**2
b = a/10
```

```
a = "tic"
b = "tac"
b = a+b
c = 3*b
```

```
x = (5<7)
y = (-7<-10)
z = (-1>12)
```

```
t = 3
c = t/2
z = c*2
```

```
a = 3
b = 2.5
c = a+b
a = c - b
d = a + b*2
c = int(b+3)
```

TP 2

Prise en main de Python

1. Opérations et fonctions usuelles

Mémo 1 (Nombres décimaux et opérations de base)

✔ Le point et la virgule : Python utilise la notation anglo-saxonne. Ainsi,
- le point sépare les parties entières et décimales, par exemple `34.56`
- la virgule sépare deux nombres distincts. En Python, `34,56` est interprété comme une liste (plus précisément un `tuple`) composée de deux nombres : 34 et 56

✔ Les opérations de base (tous ces symboles figurent sur le pavé numérique) :
- addition : `+`
- différence : `-`
- produit : `*`
- quotient : `/`
- exposant : `**`

Exercice 1 : A l'aide de la console, calculer les nombres suivants. Ne simplifiez pas les calculs à la main, saisissez les expressions entières.

$(2+3^6)^4$ $\quad \dfrac{3+5}{2\times 6}$ $\quad \dfrac{3{,}2}{10}$ $\quad 16^{1/2}$ $\quad 2+3\times 4^{5+2}$

Un certain nombre de commandes Python nécessitent l'importation de *bibliothèques* (aussi appelées *librairies*) pour être utilisées. Pour utiliser les fonctions usuelles, il va falloir utiliser la bibliothèque **numpy** via l'instruction `import numpy as np`
→ `import numpy` signifie qu'on indique à Python d'importer la bibliothèque
→ `as np` signifie qu'on demande à Python d'utiliser le raccourci `np` au lieu de réécrire `numpy` à chaque utilisation.

On peut aussi utiliser la commande `from numpy import *`. Dans ce cas, on n'a pas besoin de donner le préfixe `np` pour utiliser ses commandes.
Tous les exercices de ce TP utiliseront la syntaxe `import numpy as np`, qui est utilisée lors des concours.

Mémo 2 (Fonctions usuelles)

Pour utiliser les fonctions mathématiques usuelles,

✔ on tape une fois la commande : `import numpy as np`

✔ puis on utilise la commande :

- `np.exp(...)` pour la fonction exponentielle
- `np.log(...)` pour la fonction logarithme népérien
- `np.abs(...)` pour la fonction valeur absolue
- `np.floor(...)` pour la fonction partie entière

Utilisation : `np.exp(5)` ; `np.log(1)`...

Mémo 3 (π et e)

Pour faire appel à la valeur de π, on écrira `np.pi` et pour faire appel à la valeur de e, on écrira `np.e`

 ici il s'agit de **nombres** et non de fonctions, donc à utiliser sans parenthèse.

Code (Dans la console)

Certaines des instructions suivantes vont renvoyer des erreurs. À vous de les repérer sur feuille. Vérifiez ensuite en saisissant les commandes dans la console.

```
>>> import numpy as np  >>> np.log(2)          >>> np.floor(-6.4)     >>> np.e()
>>> exp(5)              >>> np.abs(5)          >>> np.exp             >>> np.pi
>>> np.exp(5)           >>> np.abs(-3.2)       >>> pi                >>> np.e
>>> log(2)             >>> np.floor(4.5)      >>> e
```

Exercice 2 : A l'aide de la console, calculer les nombres suivants :

$\dfrac{1+\pi}{3}$ $\qquad \sqrt{9-e}$ $\qquad \ln(2-e^5)$ $\qquad \dfrac{12\pi+\mathrm{e}^4}{\sqrt{1+50^2}}$ $\qquad \dfrac{\sqrt{9-e}}{(2+3^6)^4}$

2. Les variables

Une variable est une « boîte » qui porte un nom et dans laquelle est rangé un « objet ». Le plus souvent, il s'agit d'une valeur numérique mais on peut aussi y stocker des chaînes de caractères ou un tableau. Le but de ces boîtes est de stocker temporairement des résultats intermédiaires et de les rappeler plus tard quand on en aura besoin. Pour définir une variable, Python a besoin de deux éléments : son nom et son expression (ce qu'elle vaut).

Le **nom d'une variable** répond à quelques règles :

1. c'est une séquence de lettres et de chiffres qui doit toujours commencer par une lettre,
2. il n'y a pas de lettres accentuées, espaces, caractères spéciaux SAUF le tiret long _.

L'expression peut prendre beaucoup de formes différentes : un entier, un flottant (nombre non entier), un calcul, un calcul qui utilise d'autres variables définies précédemment, une liste, un tableau, une fonction, une chaîne de caractères...

Mémo 4 (Variable)

La syntaxe Python pour définir une variable est : `nom = expression`
Exemple : `x = 5+3` Cette instruction est exécutée dans l'ordre suivant :

1. Python calcule (si nécessaire) toute l'expression située à droite : ici il calcule $5+3$ et trouve 8
   L'expression doit donc être calculable ! En particulier toutes les valeurs utilisées doivent être connues.
2. Il affecte le résultat à la variable de gauche (il peut éventuellement y avoir plusieurs variables, affectées simultanément). Donc ici il enregistre 8 dans la variable `x`.

 à partir de là, Python retient uniquement que `x` vaut 8. L'opération utilisée `5+3` est oubliée.

Code (Dans la console)

À faire sur papier, puis vérifier sur la console. Écrivez les variables qui sont définies et les valeurs qu'elles contiennent. Repérez les expressions qui vont produire des erreurs.
Vérifiez ensuite sur la console. Prenez le temps de lire le message affiché en cas d'erreur (Python vous explique ce qui ne va pas, en particulier sur la dernière ligne rouge).

```
>>> a = 3
>>> 3 = a
>>> b = 5
```

```
>>> c = a+b
>>> c
>>> a = a+1
```

```
>>> a
>>> a+1 = a
>>> d,e = 9,10
```

```
>>> d
>>> e
```

Exercice 3 (Cherchez l'erreur !) **:** Les instructions suivantes comportent chacune une erreur et ne pourront pas être exécutées par Python. À vous de les trouver !

Code

```
>>> x1, y1 = 3
```

```
>>> b = b+3
```

```
>>> 2x = 5
```

```
>>> z + 3 = 8
```

```
>>> 5 = c
```

Mémo 5 (Types de variables)

Suivant ce que représente la variable, elle ne sera pas stockée de la même manière dans la mémoire de l'ordinateur et les opérations possibles seront différentes.
Quelques types de variables en Python :

- ✔ `int` désigne les entiers.
- ✔ `float` désigne les flottants (nombres non entiers, souvent interprétés comme les réels).
- ✔ `bool` désigne les booléens. Ils ne peuvent prendre que deux valeurs : `True` ou `False`.
- ✔ `str` (abréviation de *string*) désigne les chaînes de caractères.

Autrement dit, suivant le type de variables, les opérations peuvent ne pas avoir le même sens et sont parfois impossibles (on ne divise pas les chaînes de caractères par exemple.)
Il existe une commande qui permet de connaître le type d'une variable, il s'agit de la commande `type`. En cas d'incohérences, Python affiche généralement `TypeError`.

Mémo 6 (Chaînes de caractères)

→ on saisit une chaîne de caractères (type `str` pour *string*) en notant le texte entre apostrophes, guillemets, triples apostrophes ou triples guillemets.

→ on peut concaténer ("recoller") deux chaînes en utilisant le symbole +, et concaténer plusieurs fois une chaîne avec elle-même en utilisant le symbole *.

Code (Dans la console)

Noter sur papier les valeurs enregistrées dans les variables `a` et `b`, puis les textes affichés.

```
>>> a,b = "l'informatique," , '''je dis : "c'est super!"'''
>>> a
>>> b
>>> b+a
>>> a+b
>>> 2*a + b
>>> 5*(a+2*b)
>>> a = '3'
>>> b = 4*a
>>> b
```

3. Commandes entrées et sorties

La commande `print` permet d'afficher la valeur d'une variable, du texte, ou les deux. Elle est particulièrement utile quand on utilise l'éditeur pour programmer.

Mémo 7 (La commande de sortie `print`)

La commande print permet d'afficher un message ou le contenu d'une variable.

✔ `print(x)` affiche le **contenu** de la variable `x` (quel que soit son type),

✔ `print("texte")` affiche le texte écrit entre guillemets.

Pour afficher le contenu de plusieurs variables ou mixer des messages et des variables, on sépare avec des virgules. Par exemple, `print("valeur=",x)` affichera le message texte `valeur=`, suivi du contenu de la variable `x`.

Code

```
>>> a = 5        >>> print(a)        >>> print("a")        >>> print("la valeur de a est ", a)
```

Mémo 8 (La commande d'entrée `input`)

La commande `input` permet de demander à l'utilisateur l'entrée d'une valeur, qui sera stockée dans une variable pour être utilisée.
Par défaut, Python considère l'entrée comme une chaîne (type `str`). On doit préciser `float` ou `int` si on attend un nombre.

```
nom_variable = type_variable(input("message utilisateur"))
```

Par exemple, la commande `x = int(input("donner un nombre : "))` demande à l'utilisateur une valeur entière qui sera stockée dans la variable notée `x`.

Exercice 4 (Dans la console) **:**

1. Que fait ce programme ?

```
>>> age = int(input('Quel est ton age?'))
>>> prochain = age + 1
>>> print("l'an prochain tu auras " , prochain, " ans")
```

2. Recopiez ces instructions en supprimant le `int` dans la première ligne. Que se passe-t-il ?

3. En s'inspirant de ce programme, écrire un programme qui demande l'âge de l'utilisateur puis donne son année de naissance.

Mémo 9 (Utiliser l'éditeur)

Pour un programme long, ou qui doit être utilisé plusieurs fois, il est plus efficace de taper le code dans l'éditeur.

1. Saisir l'ensemble du code dans l'éditeur
2. Cliquer sur le triangle vert pour exécuter toutes les lignes d'un coup

 On remarque que Python **ne lit pas les modifications** au fur et à mesure. Il en prend connaissance uniquement quand on clique sur le triangle vert « exécuter ».

Exercice 5 (Dans l'éditeur) **:** Écrire un script qui demande l'entrée d'un réel x et d'un entier n et qui calcule et affiche x^n.

Exercice 6 (Dans l'éditeur) **:**

1. Écrire un script qui demande à l'utilisateur de saisir 3 nombres successivement. Le script calculera ensuite la moyenne de ces nombres et affichera le résultat.
2. Modifier le script pour que le résultat s'affiche sous la forme suivante (si par exemple on a saisi les nombres 2, 3 et 7) :
   `La moyenne de 2, 3 et 7 est 4.`

4. Approfondissement

Exercice 7 (La conjecture de Syracuse) **:** *Prérequis : utiliser la condition* `if`.
Indication : la commande `%` *détermine le reste d'une division. Par exemple,* `17%2` *renvoie* `1`

Faire une recherche sur la *conjecture de Syracuse*. Celle-ci repose sur le calcul d'une suite (u_n), où le premier terme u_0 est un entier naturel choisi arbitrairement.

Voici quelques idées pour tester cette conjecture grâce à Python.

— Définir $u_0 = 15$ puis afficher les 20 premiers termes de la suite.
— Afficher tous les termes de la suite jusqu'à tomber sur la séquence $1, 2, 4$.
— Demander la saisie du premier terme u_0 au clavier.
— Afficher la durée de vol (autrement dit, le nombre de termes calculés jusqu'à trouver la séquence $1, 2, 4$)
— Chercher un premier terme u_0 dont la durée de vol soit supérieure à 20 (ou à 100, ou à 1000...)

Exercice 8 : *Prérequis : utiliser une boucle* `while`.

On souhaite déterminer les prix TTC d'une liste de produit. Écrire un programme qui demande un prix HT puis calcule et affiche la valeur TTC de ce produit. On répétera ces calculs tant que l'utilisateur saisit de nouveaux prix. Le programme s'arrêtera si l'utilisateur saisit la valeur 0.

TP 3

Instructions conditionnelles : if ... elif ... else

En algorithmique, on est très souvent amené à effectuer des opérations sous certaines conditions : par exemple si l'utilisateur répond « oui », faire telle action. La *condition* peut être **vraie** (alors on effectue l'action demandée) ou **fausse**.

On effectuera donc un (ou plusieurs) test pour savoir si la condition est vérifiée et appliquer l'instruction correspondante. Les tests sont repérés par *si* ou *sinon* (en Python : `if` / `else`).

1. L'essentiel

Mémo 1 (La commande `if`)

On appelle « instruction conditionnelle » la structure suivante :

Code

```
if condition :
    instruction1
    instruction2
    ...
```

Ici, Python vérifie si la `condition` indiquée est vraie :

- si elle est vraie, il effectue les instructions données en-dessous et qui sont **indentées** (c'est-à-dire décalées vers la droite) puis passe à la suite du programme
- si elle est fausse, il ne lit pas les instructions indentées, et passe à la suite du programme.

Dans tous les cas, le programme se poursuit avec les instructions qui apparaissent en dessous, sans décalage.

 Ne pas oublier les `:` après la condition, sinon Python vous indiquera le problème par un message d'erreur.

Exercice 1 : Que vont afficher ces deux programmes ?

Code (Dans l'éditeur)

```
age = 15
if age < 18 :
    print("Vous êtes mineur")
    print("aujourd'hui")
print("Fin du programme")
```

Code (Dans l'éditeur)

```
age = 21
if age < 18 :
    print("Vous êtes mineur")
    print("aujourd'hui")
print("Fin du programme")
```

Mémo 2 (Commandes `else` et `elif`)

— La commande `else :` peut se placer après une commande `if`. Les instructions qui en dépendent seront exécutées dans le cas où la condition du « if » n'est pas vérifiée.

— La commande `elif ... :` est utilisée pour tester plusieurs conditions à la suite. On l'utilise entre `if` et `else`.

```
if condition :
    bloc d instructions 1
else :
    bloc d instructions 2
```

→ On lit la condition du `if`
→ Si elle est vraie on suit les instructions du bloc 1
→ Si elle est fausse on suit les instructions du bloc 2
Dans tous les cas, on poursuit le programme avec les lignes non indentées qui suivent.

```
if condition :
    bloc d instructions 1
elif condition :
    bloc d instruction 2
elif condition :
    bloc d instruction 3
...
else :
    bloc d instructions n
```

→ On lit la condition du `if`
→ Si c'est faux, on regarde le premier elif.
→ Si c'est encore faux, on regarde le suivant.
→ Si aucune condition n'est vraie, on exécute else.
Dès qu'une condition est vraie, Python exécute le bloc correspondant et s'arrête (il ne lit pas les autres `elif` / `else`, mais poursuit le programme si d'autres instructions suivent).

Mémo 3 (Les conditions)

Les instructions `if` et `elif` contiennent des conditions : `Python` les *évalue* en déterminant si elles sont vraies ou fausses.
Pour écrire un test, on utilisera donc :

1. des variables, fonctions, constantes,
2. des opérateurs de comparaison,

Opérateurs	=	≠	<	>	≤	≥
Commande Python	==	!=	<	>	<=	>=

3. des connecteurs logiques

Connecteurs logiques	et	ou	non
Commande Python	`and`	`or`	`not`

Lors d'un test, Python renvoie un booléen : `False` si le test est faux et `True` si le test est vrai.

2. Exercices

Exercice 2 (Tests logiques) **:** Notez au papier la valeur de chacun de ces tests, `True` ou `False`.
Vérifiez ensuite en les saisissant dans la console.

Code (Dans la console)

```
>>> 1 == 1
>>> 1 == 2
>>> 1 <= 2
>>> 2 <= 1
```

```
>>> a = 2
>>> a < 3
>>> 2*a != 4
>>> (1<2) and (2<3)
```

```
>>> (1<2) and (2>3)
>>> (1<2) or (2>3)
>>> not 1==2
>>> not 1>3
```

```
>>> 1 < 3 < 5
>>> 1 < 3 > 2
>>> 1 <= 2 < 7
>>> 1 < 2 < -3
```

Exercice 3 :

1. Quelle est la différence entre ces deux programmes ?

Code (Dans l'éditeur)

```
x = float(input("donner un réel "))
if x>1 :
    print(x," est supérieur à 1")
    y = x+1
print(y)
```

Code (Dans l'éditeur)

```
x = float(input("donner un réel "))
if x>1 :
    print(x," est supérieur à 1")
y = x+1
print(y)
```

2. Que se passe-t-il si on remplace la première ligne par `x = input("donner un entier ")` (en supprimant la commande `float`) ?

Exercice 4 : Que vont afficher les programmes suivants ?

```
x = 7
if x > 0:
    print("A")
if x > 5:
    print("B")
```

```
x = 10
y = 5
if x > 5:
    if y > 10:
        print("A")
    else:
        print("B")
else:
    print("C")
```

Exercice 5 : Dans l'éditeur, écrire un programme Python qui demande à l'utilisateur de saisir un nombre réel x et qui affiche « Le nombre x est compris entre -1 et 2 » si c'est le cas.

Exercice 6 : À quoi correspond la valeur affichée par ce programme ?

Code (Dans l'éditeur)

```
x = float(input("Saisir un réel "))
if x >= 0 :
    y = x
else :
    y = -x
print(y)
```

Exercice 7 : Dans l'éditeur, écrire un programme qui effectue les instructions suivantes :

— il demande à l'utilisateur de saisir un nombre a, puis un nombre b

— il affiche le maximum entre ces deux nombres.

Exercice 8 : On considère le code suivant :

Code (Dans l'éditeur)

```
x = float(input("Saisir un nombre réel : "))
if x <= 0 :
    print("Réponse A")
elif x < 1 :
    print("Réponse B")
else :
    print("Réponse C")
```

Quel texte sera affiché si on saisit au clavier le nombre 0,2 ? Le nombre 1 ? Le nombre 0 ? Le nombre -3 ? Le nombre 4 ? Vérifiez en recopiant ce code et en testant les valeurs données.

Exercice 9 : Écrire un programme qui affiche les commentaires suivants en fonction de la température T :

$T < 0$	"Froid"
$0 \leq T < 10$	"Frais"
$10 \leq T < 15$	"Doux"
$15 \leq T$	"Chaud"

Exercice 10 : Écrire un programme Python qui demande à l'utilisateur de saisir un nombre entier. Il affichera ensuite, selon sa valeur : « négatif », « positif » ou « nul ».

3. Approfondissement

Exercice 11 : Écrire un programme qui demande la saisie au clavier des coefficients d'un polynôme du second degré a, b et c, calcule le discriminant et donne les deux racines si le discriminant est strictement positif, l'unique racine si le discriminant est nul et affiche « il n'y a pas de racine » si le discriminant est strictement négatif.

Exercice 12 (Année bissextile) **:** Demander une année à l'utilisateur puis afficher si elle est ou non bissextile.
Les années bissextiles suivent les règles suivantes :

- les années divisibles par 4 sont bissextiles
- exception : les années divisibles par 100 ne sont pas bissextiles
- exception : les années divisibles par 400 sont bissextiles

Ces règles visent à être au plus proche de la durée réelle d'une année, qui est environ 365,2422 jours.
Commande utile : la commande `%` donne le résultat de la division euclidienne de deux nombres. Par exemple `2026%4` donne 2, donc 2026 n'est pas multiple de 4.

Exercice 13 (Triangles possibles) **:** Demander à l'utilisateur de saisir 3 longueurs a, b et c, puis afficher si on peut

construire un triangle dont les longueurs des côtés sont a, b et c.
Rappel : un tel triangle existe si on a $a+b>c$, $a+c>b$ et $b+c>a$

Exercice 14 (Tri) **:** Demander à l'utilisateur de saisir 3 nombres, puis afficher ces nombres en ordre croissant.

Par exemple si l'utilisateur saisit `2`, `5` puis `1`, le programme affichera `1 2 5`.

Exercice 15 (Cinéma) **:** Voici les tarifs proposés par un cinéma :

Condition	Prix
tarif normal	10 €
moins de 14 ans	5 €
étudiant	6 €

Attention : les étudiants de plus de 30 ans paient 10 €.

Écrire un programme qui demande l'âge de l'utilisateur puis s'il est étudiant (réponse attendue oui / non). Il affichera ensuite le tarif de son billet de cinéma.

TP 4

Boucles `for`

Lorsque l'on souhaite répéter un certain nombre de fois une série d'instructions, il existe évidemment un moyen plus rapide que de recopier la commande à chaque fois. Il s'agit des **structures répétitives**. Il en existe deux types différents. Nous allons voir dans ce TP celle où le nombre de répétitions est fixé à l'avance : c'est ce qu'on appelle la structure `for`.

1. L'essentiel

Mémo 1 (Boucle for)

La syntaxe Python d'une boucle for est la suivante :

Code

```
for i in valeurs:
    instruction1
    instruction2
    ...
```

Les valeurs possibles sont les suivantes :

- ✔ `range(m,n)` est la liste des entiers de m à $n-1$,
- ✔ `range(n)` est la liste des entiers de 0 à $n-1$. (S'il n'y a pas de m, c'est comme s'il valait 0.)
- ✔ `range(m,n,pas)` est :
  - la liste des entiers m, $m+p$, $m+2p$ inférieurs à n si $m<n$ et si le pas est positif.
  - on peut aussi donner un pas négatif, ce qui donnera une suite décroissante.


On retient que Python commence par défaut à 0, et ne prend pas en compte la dernière valeur.
On retient que `range(a,b)` = entiers de l'intervalle $[a,b[$ et `range(b)` = entiers de l'intervalle $[0,b[$

2. Exercices

Exercice 1 : Que font ces codes ?

Code (Dans l'éditeur)

```
for i in range(6):
    print(2**i)
```

Code (Dans l'éditeur)

```
for i in range(4,2):
    print(2**i)
```

Code (Dans l'éditeur)

```
resultat = 1
for i in range(2,6):
    resultat = resultat*i
print(resultat)
```

Exercice 2 :

1. Écrire un code qui permet d'afficher les nombres entiers de 1 à 10.
2. Modifier le code précédent pour qu'il affiche les nombres entiers allant de 5 à 61.
3. Écrire un code qui permet d'afficher les nombres impairs compris entre 1 et 10.
4. Écrire un code qui demande d'entrer un entier naturel n puis affiche tous les entiers impairs compris entre 1 et n.

Exercice 3 : Écrire un programme qui demande à l'utilisateur de rentrer deux nombres : a un réel et n un entier naturel et qui calcule a^n à l'aide d'une boucle `for`. (*Rappel :* $a^n = \underbrace{a \times a \times \ldots \times a}_{n \text{ fois}}$.)

Exercice 4 (Application aux sommes) **:** Pour cet exercice, on oublie les formules vues en cours et on effectue les sommes terme par terme.

1. Écrire un programme qui demande une valeur de n et calcule la somme des entiers de 1 à n.
2. Écrire un programme qui demande une valeur de n et calcule la somme des entiers au carré de 1 à n.

3. Approfondissement : boucles for imbriquées

Code (Dans l'éditeur)

```
for i in range(1,11):
    for j in range(1,11):
        print(i*j)
```

Code (Dans l'éditeur)

```
for i in range(1,11):
    for j in range(1,i+1):
        print(i*j)
```

Que font ces programmes ?

Exercice 5 :

1. Écrire un programme qui demande à l'utilisateur de renseigner un entier n et qui affiche un carré d'étoiles de taille n. Par exemple si l'utilisateur tape 5, le programme affichera :

```
*****
*****
*****
*****
*****
```

2. Même chose, mais avec un triangle de hauteur n :

```
*
**
***
****
*****
```

4. Approfondissement

Exercice 6 : Écrire un programme qui demande une taille n, affiche un tapis de $n+1$ lignes sur $n+1$ colonnes traversé par une diagonale.

Exemple (pour une taille de 10) :

```
+-----------+
|########## |
|######### #|
|######## ##|
|####### ###|
|###### ####|
|##### #####|
|#### ######|
|### #######|
|## ########|
|# #########|
| ##########|
+-----------+
```

Remarque : le tapis généré est de taille $n+1$ et non n afin de gérer plus facilement le cas "0"

```
+-+
| |
+-+
```

On peut aussi varier les plaisirs, par exemple en affichant un joli triangle isocèle :

```
###########
 #########
  #######
   #####
    ###
     #
```

TP 5

Boucles `while`

1. L'essentiel

Mémo 1 (Boucle `while`)

La boucle `while` (« tant que » en français) permet de répéter une instruction. Elle est utilisée quand on ne sait pas à l'avance combien de fois il faut la répéter, mais qu'on connaît une condition permettant son arrêt.

La structure Python des boucles while est la suivante :

```
while condition :
    instruction1
    instruction2
    ...
```

Ici Python vérifie si la condition contenue dans le `while` est vraie. Si c'est le cas, elle suit les instructions qui sont indentées.
À la différence du `if`, Python revient ensuite à la condition et recommence si elle est toujours vérifiée.

 On retient que le `while` donne une condition pour **continuer**.
Parfois l'énoncé donne une condition d'arrêt, par exemple pour la consigne « donner le plus petit entier i tel que $\ln(i) > 10$ » on utilisera `while np.log(i) <= 10`

Remarque : Que donnent les codes suivants ?

```
i = 0
while i<10 :
    print(i)
    i = i+1
print("Fin")
```

```
i = 0
while i<10 :
    print(i)
i = i+1
print("Fin")
```

2. Exercices

Exercice 1 (Division par 2) **:** Écrire un programme qui demande à l'utilisateur de saisir un nombre n entier naturel non nul et qui divise ce nombre par 2 tant qu'il est supérieur ou égal à 1. Puis ce programme affiche le résultat obtenu.

Par exemple, si l'utilisateur saisit $n = 10$, le programme calcule : $10/2 = 5$, puis $5/2 = 2.5$ puis $2.5/2 = 1.25$ puis $1.25/2 = 0.625$ et s'arrête automatiquement car ce nombre est inférieur strictement à 1.

Exercice 2 (Un utilisateur tête en l'air) **:** Écrire un programme qui demande à l'utilisateur de saisir un nombre n positif strictement supérieur à 1. Tant que l'utilisateur se trompe et saisit un nombre inférieur ou égal à 1, le programme demande à l'utilisateur de ressaisir un nombre supérieur strictement à 1.

Exercice 3 (Avec du ln) **:** Écrire un programme qui demande à l'utilisateur un réel A et affiche le premier entier n tel que $\ln(n) > A$ et la valeur de $\ln(n)$.

Exercice 4 (Avec des factorielles) **:** Écrire un programme qui détermine le plus petit entier n tel que $n! > 10^6$.

Exercice 5 : Pour tout $n \in \mathbb{N}^*$, on considère $S_n = \sum_{k=1}^{n} \frac{1}{k}$.

Écrire un programme qui renvoie le plus petit entier n tel que $S_n > A$ pour A donné par l'utilisateur.

Exercice 6 : Écrire un programme qui permet de trouver le plus grand entier inférieur ou égal à un nombre positif saisi par l'utilisateur.
Par exemple si l'utilisateur saisit 6,5, le programme retourne 6. Si l'utilisateur saisit 11,2, le programme retourne 11…

Le but n'est pas d'utiliser la fonction `np.floor` mais une boucle `while`.

3. Approfondissement

Exercice 7 : L'utilisateur donne un entier supérieur à 1 et le programme affiche, s'il y en a, tous ses diviseurs propres sans répétition ainsi que leur nombre. S'il n'y en a pas, il indique qu'il est premier.

Par exemple :

```
Entrez un entier strictement positif : 12
Diviseurs propres sans répétition de 12 : 2 3 4 6 (soit 4 diviseurs propres)
Entrez un entier strictement positif : 13
Diviseurs propres sans répétition de 13 : aucun ! Il est premier
```

Exercice 8 : Une légende de l'Inde ancienne raconte que le jeu d'échecs a été inventé par un vieux sage, que son roi voulut remercier en lui affirmant qu'il lui accorderait n'importe quel cadeau en récompense. Le vieux sage demanda qu'on lui fournisse simplement un peu de riz pour ses vieux jours et plus précisément un nombre de grains de riz suffisant pour que l'on puisse en déposer 1 seul sur la première case du jeu qu'il venait d'inventer, deux sur la suivante, quatre sur la troisième et ainsi de suite jusqu'à la 64^{e} case.

Écrivez un programme Python qui affiche le nombre de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières :

1. le nombre exact de grains (nombre entier)
2. le nombre de grains en notation scientifique (nombre réel)

Exercice 9 : Le programme doit demander à l'utilisateur de saisir une suite d'entiers strictement positifs. Il compte :

1. le nombre total de valeurs saisies
2. le nombre de valeurs supérieures à 100
3. la moyenne des valeurs

Le programme s'arrête si l'utilisateur saisit la valeur 0.

Remarque : il n'est pas nécessaire d'enregistrer la totalité des valeurs saisies.

TP 6

Les fonctions

Quand une même suite d'instructions est amenée à être utilisée plusieurs fois, il peut être utile de définir une *fonction*, qui sera appelée à chaque fois que les instructions doivent être exécutées.

Par exemple si on veut calculer le coefficient binomial $\binom{10}{6}$, on a besoin de calculer 10!, 6! et 4!. Au lieu de recopier 3 fois des instructions similaires, on peut les écrire sous la forme d'une *fonction*.

1. L'essentiel

Mémo 1

Pour définir une fonction avec Python, il faut respecter la syntaxe suivante.

```
def nom_fonction(variables) :
    instructions
    return(resultat)
```

On notera en particulier que :

- ✔ `nom_fonction` est le nom qui sera utilisé pour appeler la fonction,
- ✔ `variables` est la liste des variables (séparées par des virgules) qui seront utilisées lors de l'instruction de la fonction,
- ✔ `return(resultat)` permet de renvoyer un résultat.

Remarque. `return` peut également être placé au milieu du programme (par exemple après une instruction conditionnelle). Dans ce cas, la fonction s'arrête dès que cette instruction `return` est exécutée.


On écrit toujours les fonctions **dans l'éditeur**, le but est de pouvoir ensuite les utiliser dans la console.

Le code qui suit permet de définir la fonction $f(x) = \dfrac{1}{2+x}$.
On l'écrit dans l'**éditeur**.

```
def f(x):
    return (1/(2+x))
```

A présent, calculer les valeurs suivantes puis les taper dans la **console** pour vérifier vos calculs (certaines vont donner des erreurs, à vous de comprendre pourquoi).

```
>>> f(0)
>>> f(2)
>>> f(-2)
>>> f(x)
```

Remarque : Jusqu'à présent, les fonctions considérées « se ressemblaient » au sens suivant : l'espace de départ était $\mathbb{R}$ (ou un sous-ensemble de $\mathbb{R}$) et l'espace d'arrivée était $\mathbb{R}$ (ou un sous-ensemble de $\mathbb{R}$). En réalité, rien ne nous interdit de changer d'espace de départ et/ou d'arrivée. Par exemple, la fonction suivante est celle qui à la température ambiante T associe un commentaire météorologique (i.e qui à un réel associe une chaîne de caractère).

```
def meteo(T):
    if T < 0 :
        return("Attention, risque de verglas")
    elif T < 10 :
        return("Temps frais.")
    else :
        return("Temps clément.")
```

Quel est l'espace de départ de cette fonction ? Quel est l'espace d'arrivée de cette fonction ? Tester cette fonction dans la console pour quelques valeurs de T : par exemple `meteo(7.5)`, `meteo(-5)`, etc.

2. Exercices

Exercice 1 : Définir la fonction $f(x) = \dfrac{\sqrt{1+e^x}}{1+x^2}$ avec Python. En déduire une valeur approchée de $f(1)$ et de $f(2)$. *La valeur e n'est pas prédéfinie en Python, il faut utiliser la bibliothèque `numpy`, voir TP 2.*

Exercice 2 : Redéfinir la fonction valeur absolue avec Python (sans utiliser `np.abs`).

Exercice 3 : Créer une fonction qui, à une note sur 20, associe automatiquement un commentaire de votre choix. On raisonnera par tranches de 5 : notes de 0 à 5, notes de 5 à 10, notes de 10 à 15 et notes de 15 à 20.

Exercice 4 : Écrire une fonction appelée `majeur` qui renvoie `True` ou `False` en fonction de l'âge.

Exercice 5 : Écrire une fonction qui prend n en entrée et renvoie la somme des entiers de 0 à n.

Exercice 6 : Écrire une fonction qui prend en entrée un réel a et qui renvoie l'entier ayant la plus petite puissance de 3 supérieure ou égale à a.

Exercice 7 : Définir la fonction $f(x) = x^4$ de deux manières différentes : la première utilisera le symbole `**` . La deuxième utilisera une boucle for.

3. Un peu de théorie : variable locale / globale

Quand on définit une fonction, Python « oublie » toutes les variables déjà définies. Il n'utilise alors que les variables données en paramètre, et celles définies à l'intérieur de la fonction. On dit que ces variables sont *locales*, elles n'existent qu'à l'intérieur de la fonction.
À l'inverse, les variables définies dans le corps du programme (à l'extérieur des fonctions) sont appelées variables *globales*, elles peuvent être utilisées en dehors des fonctions tout au long du programme.

Exercice 8 : On écrit le programme ci-contre dans l'éditeur. Quels affichages pourront être exécutés ? Lesquels renverront une erreur ?

```
x = 3
y = 2
print(x)

def somme(a,b) :
    print(x)
    print(a)
    s = a + b
    return s

print(x)
print(a)
print(s)

s = somme(x,y)
print(s)
```

4. Approfondissement

Exercice 9 : 1. Écrire une fonction `Factorielle` qui prend en paramètre un entier naturel n et renvoie $n!$.

2. Écrire une fonction `CoefBin` qui prend en paramètre deux entiers n et p et renvoie le coefficient binomial $\binom{n}{p}$

Comme toujours, on peut peaufiner en vérifiant que les valeurs saisies soient valables (entiers naturels, et $0 \leq p \leq n$).

Exercice 10 : Écrire un programme qui approxime par défaut la valeur de la constante mathématique e, pour n assez grand, en utilisant la formule :

$$e \approx \sum_{i=0}^{n} \frac{1}{i!}$$

Pour cela, définissez la fonction factorielle et, dans votre programme principal, saisissez l'ordre n et affichez l'approximation correspondante de e.

Optimisation. Pour gagner en efficacité, évitez de recalculer chaque factorielle à partir de zéro, mais utilisez la factorielle précédente.

TP 7

Calculs des termes d'une suite

Le but du TP suivant est de calculer le n-ième terme d'une suite, qu'elle soit définie par une relation explicite ou une relation de récurrence.

1. L'essentiel

Mémo 1 (Suite définie par une relation explicite)

Lorsque $(u_n)_{n\in\mathbb{N}}$ est définie par une relation explicite, c'est-à-dire par une relation du type :

$$\forall n \in \mathbb{N}, \quad u_n = f(n),$$

pour une certaine fonction f, on peut calculer le n-ième terme de cette suite en définissant une fonction prenant comme argument d'entrée n et retournant $f(n)$.

```
def nom_suite(n):
    return ....
```

En remplaçant les pointillés par la formule permettant de calculer u_n.

Exemple : Considérons la suite définie par : $\forall n \in \mathbb{N}$, $u_n = \dfrac{3n^2}{1+2n^2}$. Les fonctions python suivantes permettent de calculer n'importe quel terme de la suite :

```
def suite(n):
    return 3*n**2 / (1 + 2*n**2)
```

```
def suite(n):
    u = 3*n**2 / (1 + 2*n**2)
    return u
```

Taper `suite(10)` dans la console pour connaître u_{10} et `suite(57)` pour connaître u_{57}.

Mémo 2 (Suite définie par une relation de récurrence d'ordre 1)

Lorsque la suite est définie par une relation de récurrence :

$$\begin{cases} u_{n+1} = f(u_n), & \forall n \in \mathbb{N}, \\ u_0 = a \in \mathbb{R}, \end{cases}$$

alors pour calculer u_n, il nous faut connaître tous les termes précédents de u_0 à u_{n-1}.
Pour cela, on initialise avec le premier terme de la suite puis on utilise une boucle for pour calculer les termes suivants de la suite.

```
def nom_suite(n) :
    y = a
    for i in range(1,n+1) :
        y = f(y) # relation de recurrence
    return(y)
```

Où on remplace `a` par la valeur u_0 et `f` par la formule adaptée (celle donnée par la relation de récurrence).

Exemple : Considérons, par exemple, la suite définie par :

$$\begin{cases} u_{n+1} = u_n + 3, & \forall n \in \mathbb{N}, \\ u_0 = 1. \end{cases}$$

```
def suite_u(n) :
    y = 1 # initialisation
    for i in range(1,n+1):
        y = y + 3 # formule de recurrence
    return(y)
```

Taper `suite_u(2)` dans la console pour connaître u_2 et `suite_u(10)` pour connaître u_{10}.

Mémo 3 (Suite récurrente d'ordre 2)

Le principe est le même qu'à l'ordre 1 avec quelques difficultés en plus :

- on initialise deux valeurs u_0 et u_1, stockées dans 2 variables, par exemple `a` et `b`.
- à chaque étape `a` représente u_n et `b` représente u_{n+1}
- on calcule u_{n+2} à l'aide de la relation de récurrence. On le stocke dans une variable `c`.
- on met à jour les variables pour passer à l'itération suivante.

Par exemple pour la suite (u_n) définie par $u_0 = 3$, $u_1 = 1$ et $u_{n+2} = 4u_n - u_{n+1}$, $\forall n \in \mathbb{N}$

```
def suite(n) :
    a = 3 # u_0
    b = 1 # u_1
    for i in range(2,n+1) :
        c = 4*a-b # calcul de u_(n+2)
        a = b # mise à jour des variables
        b = c
    return b
```

 on utilise `range(2,n+1)` car la première application de la formule calcule u_2, et on veut poursuivre jusque u_n.

 pour aller plus loin : on peut gérer les cas $n = 0$ et $n = 1$ en testant en début de programme ces valeurs de n (avec un `if` pour renvoyer directement le résultat. Le programme présenté ici n'est valable que pour $n \geq 1$.
On peut aussi remplacer les 3 lignes de la boucle `for` par une double affectation `a,b = b, 4*a-b`. Dans ce cas la variable `c` est inutile.

2. Exercices


Avant de commencer ces exercices, on détermine s'il faut ou non une boucle `for` :

— Si je peux calculer u_{100} directement (par exemple en saisissant une formule dans la calculatrice), alors on calcule aussi directement en Python. Pas de boucle `for`

— Si je ne peux pas calculer u_{100} directement (parce qu'il faudrait calculer u_1 puis u_2 ... puis u_{99}), alors j'ai besoin d'une boucle `for` en Python.

Exercice 1 :

1. Créer une fonction Python qui permette de calculer n'importe quel terme de la suite définie sur $\mathbb{N}$ par : $\forall n \in \mathbb{N}$, $u_n = \dfrac{e^n}{\sqrt{n}+\ln(2+n)}$.
   Application : calculer u_{10}, u_{2000}.
2. Faire de même avec la suite (v_n) définie sur $\mathbb{N}$ par $v_n = \ln\left(\sqrt{n+1}-\sqrt{n}\right)$.

Exercice 2 : On considère la suite :

$$\begin{cases} u_{n+1} = u_n^2 + u_n, & \forall n \in \mathbb{N}, \\ u_0 = 0,5. \end{cases}$$

1. Créer une fonction `suite_u` qui prend comme argument d'entrée n et qui retourne la valeur de u_n.
2. Mettre la valeur initiale de la suite en argument d'entrée de la fonction. Votre fonction prendra donc deux arguments en entrée : n et a. Ainsi pour calculer le 10-ième terme avec $u_0 = 0,5$, on tapera `suite_u(10,0.5)`

Exercice 3 : Soit $(u_n)_n$ la suite définie par $\begin{cases} u_0 = 0 ; u_1 = 1 \\ \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n \end{cases}$

1. En utilisant une boucle for et la relation de récurrence qui définit (u_n), construire une fonction qui calcule u_n pour $n \geq 2$ un entier naturel saisi par l'utilisateur.
2. Améliorer ce programme pour que le programme permette à l'utilisateur de changer les valeurs de u_0 et de u_1.

Pour tester votre programme : $u_{10} = 55$.

Exercice 4 : On considère la suite (u_n) définie par : $u_n = \exp(3n-2), \forall n \in \mathbb{N}$.

Écrire un programme qui prend en paramètre un entier n et renvoie la valeur de u_n.

Exercice 5 : Faire de même avec la suite :

$$\begin{cases} v_{n+1} = \dfrac{1}{1+|\ln(v_n)|}, & \forall n \in \mathbb{N}, \\ v_0 = a \in \mathbb{R}. \end{cases}$$

n et a seront les deux paramètres d'entrées de la fonction.

1. Créer une fonction `suite_v` qui prend comme argument d'entrée n et qui retourne la valeur de v_n.
2. Mettre la valeur initiale de la suite en argument d'entrée de la fonction. Votre fonction prendra donc deux arguments en entrée : n et a. Ainsi pour calculer le 10-ième terme avec $v_0 = 0,5$, on tapera `suite_v(10,0.5)` .

Exercice 6 : On considère la suite (u_n) définie par : $\begin{cases} u_0 = 3 ; u_1 = -5 \\ \forall n \in \mathbb{N},\ u_{n+2} = 2u_{n+1}^2 - u_n^3 \end{cases}$.

Écrire un programme qui prend en paramètre un entier n et renvoie la valeur de u_n.

3. Recherche de seuils

Un exercice classique consiste à déterminer le premier rang pour lequel une suite dépasse une valeur donnée. Cela sous-entend qu'on a déjà fait l'étude mathématique pour s'assurer que ce seuil sera bien atteint.

Exercice 7 :

```
def suite(n) :
    u = 1
    for i in range(n) :
        u = u + n
    return u

i = 0
while suite(i) < 10**6 :
    i = i+1
print(i)
```

1. Reconnaitre mathématiquement la suite (u_n) définie par la fonction `suite`.
2. À quoi correspond le résultat affiché en fin de programme ?

Exercice 8 : On considère la suite (u_n) définie par $u_{n+1} = 5u_n + 3$ et $u_0 = 2$

1. Écrire une fonction Python qui prend en paramètre un entier n et renvoie la valeur u_n correspondant.
2. Écrire une fonction Python d'en-tête `seuil(M)` qui prend en paramètre un entier M et renvoie le premier entier n tel que $u_n > M$.

4. Approfondissement : utiliser la récursivité

Cette méthode n'est pas indispensable. Pour ceux qui sont à l'aise et qui comprennent bien le fonctionnement des algorithmes, elle permet d'écrire des programmes plus légers, et d'alléger l'utilisation des conditions initiales.

Le principe d'une fonction **récursive** est qu'elle s'auto-appelle. Dans le renvoi, on utilise la fonction elle-même, mais appliquée à une autre valeur. Cela peut être très efficace pour les suites définies par une relation de récurrence d'ordre 1 ou d'ordre 2.

Mémo 4 (Suite récurrente d'ordre 1)

On reprend la suite définie par

$$\begin{cases} u_{n+1} = f(u_n), \quad \forall n \in \mathbb{N}, \\ u_0 = 1, \end{cases}$$

Dans ce cas, on renvoie directement la valeur $u_0 = a$ si le rang est $n = 0$. Dans les autres cas, on applique la formule de récurrence directement dans le `return`. Ici encore on remplace le `f` indiqué dans ce programme par la formule de récurrence.

```
def nom_suite(n):
    if n == 0 :
        return 1  # valeur u_0
    else:
        return f(nom_suite(n-1)) # relation de recurrence
```

Exemple :

Considérons, par exemple, la suite définie par :

$$\begin{cases} u_{n+1} = u_n + 3, \quad \forall n \in \mathbb{N}, \\ u_0 = 1. \end{cases}$$

Reproduire le code ci-contre et vérifier que vous obtenez bien les mêmes résultats que précédemment.

```
def suite_v(n) :
    if n == 0 :
        return 1
    else :
        return suite_v(n-1) + 3
```

Exemple : Reprenez les autres exercices pour les implémenter sous forme d'une fonction récursive (suite récurrente linéaire d'ordre 2, recherche de seuil)

5. Approfondissement : exercices

Exercice 9 : On considère la suite (u_n) définie par

$$u_n = 3n - 2, \forall n \in \mathbb{N}^*$$

1. Écrire une fonction `suite1` qui prend en paramètre un entier `n` et renvoie la valeur u_n
2. Écrire une fonction `premiers_termes1` qui prend en paramètre un entier `n` et construit une liste contenant les n premiers termes de la suite (u_n)

Le résultat attendu est le suivant :

```
>>> premiers_termes(10)
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28]
```

Exercice 10 : 1. Même consigne avec la suite (u_n) définie par

$$u_0 = 2 \text{ et } u_{n+1} = 3u_n - 2 \text{ pour } n \in \mathbb{N}$$

Le résultat attendu est le suivant :

```
>>> premiers_termes2(10)
[2, 4, 10, 28, 82, 244, 730, 2188, 6562, 19684]
```

2. Écrire un programme `seuil` qui prend en paramètre un réel `s` et renvoie le plus petit entier n tel que $u_n > s$.
Le résultat attendu est le suivant :

```
>>> seuil(100)
5
```

Exercice 11 : Même consigne avec la suite (u_n) définie par

$$u_0 = 2, u_1 = -1 \text{ et } u_{n+2} = 3u_{n+1} - 2u_n \text{ pour } n \in \mathbb{N}$$

et en créant la liste des termes d'indice pair inférieur ou égal à n (on commencera au rang 2 pour éviter les deux premières valeurs qui sont plus compliquées à définir).

Le résultat attendu est le suivant :

```
>>> premiers_termes3(10)
[-7, -43, -187, -763, -3067]
```

Exercice 12 : En mathématiques, la suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.

Écrire une fonction qui prend en paramètre un entier n, puis affiche les n premiers termes de la suite.

TP 8

Soigner son code

Voici quelques indications pour rendre votre code plus lisible. Gardez en mémoire qu'un programme informatique doit être assez clair pour être utilisé et relu par une autre personne.

Cette fiche ne vous apprendra pas de nouvelle méthode de programmation, mais vous permettra de rendre votre code plus lisible pour les autres, et pour vous aussi.

Mémo 1 (Soyez efficace !)

Quelques conseils et astuces qui vous permettront de gagner en efficacité dans votre travail sur ordinateur.

— Utilisez le **pavé numérique** du clavier pour saisir les chiffres et opérations. Obligez-vous à le faire : les débuts seront peut-être pénibles, mais vous gagnerez en efficacité sur le long terme.

— Dans la console, les **flèches** « haut » et « bas » permettent de faire défiler les instructions précédentes. À utiliser pour éviter de saisir deux fois le même texte (que l'on peut ensuite modifier).

— Utilisez les **suggestions** de noms de variables et fonctions. Quand vous commencez à saisir un nom de variable / fonction déjà définie, EduPython (ou Pyzo) vous propose les noms qui correspondent. Validez votre choix par une tabulation.

Mémo 2 (Commentez !)

Les commentaires permettent d'expliciter ce que va réaliser votre programme. Python propose deux façons de les saisir :

— le croisillon #, qui permet d'ajouter un commentaire en fin de ligne

— les triples guillemets """, à placer au début et en fin d'un paragraphe de commentaires

```
"""
ce programme calcule la factorielle d un nombre
apres avoir demande de le saisir au clavier
"""
n = int(input('saisir un entier positif : '))
fact = 1  # initialisation pour le calcul de la factorielle
for i in range (1,n+1) :
    fact = fact*i
print(fact)  # affichage du resultat
```

Mémo 3 (Testez, testez et re-testez !)

Pensez à multiplier les tests pour vérifier votre programme. S'il doit fonctionner en saisissant un réel, faites des tests :

- avec des entiers
- avec des décimaux
- avec des positifs / négatifs
- avec 0
- avec tout autre nombre qui vous semble pertinent

Vérifiez en même temps que votre boucle soit bien définie : si elle commence à 0 au lieu de 1 vous risquez d'avoir une valeur en trop.
Pensez aussi à faire quelques calculs à la main pour vérifier que les réponses de votre programme soient cohérentes.

Mémo 4 (Nommez clairement)

Utilisez des noms de variables **explicites**. Évitez d'appeler `x` tout nombre à calculer. Utilisez plutôt `fact` pour une factorielle, `maxi` pour un maximum,...
De même pour plus de lisibilité, on peut nommer une constante, par exemple

```
LONGUEUR = 10
```

Il est interdit d'utiliser les espaces dans les noms de variables, mais on peut utiliser l'*underscore* (tiret long) :

```
coef_a = 2
coef_b = 5
```

Pour les plus rigoureux, quelques recommandations pour les noms de variables :

- utilisez des majuscules et *underscores* pour les noms de constantes
- utilisez des minuscules et *underscores* pour les noms de variables et de fonctions
- utilisez des noms courts pour les boucles

Par exemple on écrira :

```
def age(naissance) :
    ANNEE = 2022
    age = ANNEE - naissance
    return age
```

Mémo 5 (Aérez !)

Quelques recommandations qui rendront votre code plus lisible.

→ Mettre des espaces dans une ligne pour aérer les calculs, par exemple de chaque côté des signes = ou +, ainsi qu'avant les commentaires. Comparez les deux écritures suivantes :

```
nbvaleur=2+3*8-5#sans espaces
nb_valeur = 2 + 3*8 - 5  # avec espaces
```

De même, ajoutez une espace avant la demande d'un nombre au clavier (l'affichage sera plus lisible sur la console).

→ Ajoutez des lignes vides entre plusieurs fonctions

Mémo 6 (Utilisez les fonctions pour structurer)

Si vous voulez copier/coller un bout de code, utilisez plutôt une fonction. Cela évitera les répétitions. De plus, si vous avez besoin de faire une modification, vous serez sûr de ne rien oublier
Chaque fonction doit avoir une **responsabilité unique**. Par exemple, au lieu d'écrire une fonction qui calcule à la fois une factorielle et une puissance, séparez ces deux actions dans des fonctions séparées.

Mémo 7 (Lisez et comprenez les erreurs)

Python affiche les messages d'erreurs, en précisant de quel endroit l'erreur semble provenir :

```
Traceback (most recent call last):
  File "/Users/valeriegoyheneche/Dropbox/ECG/TP Python/TP 5/TP5_ex1.py", line 22, in <m
odule>
    y = 1 / x
ZeroDivisionError: division by zero
```

Ici il est indiqué :

- l'endroit où l'erreur a été repérée : ligne 22 (attention, la correction devra parfois être effectuée sur une autre ligne : ici c'est la variable x qui ne prend pas la bonne valeur)
- la ligne 22 en totalité
- l'erreur rencontrée : une division par 0.

On distingue 3 types d'erreurs :

1. **Erreur de syntaxe.** Le code source du programme est mal formé. Python ne peut donc pas comprendre ce que vous avez programmé.
   - → oubli des deux points après avoir écrit un « if »
   - → ajout d'une condition dans un « else »
   - → `IndentationError` : l'indentation du code n'est pas correcte
   - → `TabError` : si on indente parfois avec une tabulation, parfois avec des espaces
2. **Erreur d'exécution.** Le programme est syntaxiquement correct, mais conduit à un calcul interdit.
   - → `ZeroDivisionError` : division par zéro
   - → `TypeError` : une opération est utilisée avec une variable d'un mauvais type
   - → `ImportError` : un package n'a pas pu être importé (vérifiez si son nom est bien écrit)
   - → `NameError` : une variable ou fonction n'a pas pu être trouvée (vérifiez ici aussi si son nom est bien écrit)
   - → `IndexError` : mauvais indice dans une liste (ou un `range`)
3. **Erreur logique.** Le programme s'exécute sans problème mais ne produit pas le résultat attendu. Aucun message d'erreur ne s'affiche. La seule solution : tester et re-tester !

Pensez à vérifier vos **parenthèses**. Une parenthèse oubliée peut créer différentes erreurs. Quand le curseur est placé après une parenthèse, celle-ci apparaît en couleur, ainsi que la parenthèse qui la complète. Si elle est rouge, c'est qu'il manque une parenthèse.

```
x  = (1 * (-3-2)
if x > 0 :
    print x
```

```
    x  = (1 * (-3-2)
           ^
SyntaxError: '(' was never closed
```

Mémo 8 (Pour aller plus loin)

Testez la commande suivante, pour accéder au Python Zen

```
import this
```

Pour plus de détails, vous pouvez consulter des informations sur le guide PEP 8, qui indique quel style utiliser en Python. Ci-dessous le guide complet et un résumé en français.

https://peps.python.org/pep-0008/

https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/

Deuxième partie

Listes

TP 9

Créations de listes

1. L'essentiel

Une *liste* est un ensemble ordonné (chaque élément a un numéro d'ordre) d'éléments dont les types peuvent être hétérogènes (par exemple des `int` et des `float`). On peut définir une liste de plusieurs manières.

Mémo 1 (Définition en extension)

Pour créer une liste *en extension*, on écrit ses éléments entre crochets et séparés par des virgules.

```
liste_1 = [2,3,5,7]  # une liste d'entiers
liste_2 = [1,'a',[1,2,3]]  # les objets ne sont pas nécessairement du même type
liste_3 = [ ]   # liste vide
```

Mémo 2 (Concaténation)

1. On peut concaténer 2 listes `L` et `M` avec la commande `+` (comme pour le type `string`) : `L+M`
2. On peut concaténer `a` fois (`a` entier) une même liste `L` avec la commande `*` (là aussi comme pour le type `string`) : `a*L`.

```
liste_1 + liste_2
3*liste_1
```

Mémo 3 (Création avec une boucle)

On peut créer une liste avec une boucle en concaténant chaque nouveau terme à la liste :
→ on initialise en créant une liste vide
→ dans la boucle, on ajoute les termes un à un.

```
def liste(n):
    L=[ ]
    for i in range(1,n+1):
       L=L+[2*i]
    return L
```

Mémo 4 (Liste en compréhension)

Une liste peut être écrite en compréhension, c'est-à-dire qu'elle est définie à l'aide d'une propriété.
La structure est la suivante :
`[formule_a_calculer for i in range(...)]`

```
[i for i in range(10)]
[i for i in range(1,40,2)]
[3*p+1 for p in range(20)]
L=[1,2,3,4]
[i**3 for i in L]
```

2. Exercices

Exercice 1 : On veut créer la liste des multiples de 3 inférieurs à 20.

1. Créer cette liste en extension
2. Créer cette liste en compréhension
3. Créer cette liste à l'aide d'une boucle

Exercice 2 : Soit (u_n) la suite définie par $\forall n \in \mathbb{N}$, $u_n = 2^n$. Écrire une fonction qui prend n en argument et renvoie la liste des termes $u_0, u_1, \dots, u_n$.

Exercice 3 : A l'aide d'une écriture par compréhension, écrire la liste des entiers naturels impairs inférieurs à 50.

Exercice 4 : Soit (u_n) la suite définie par $u_0 = 2$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \dfrac{u_n}{1+u_n}$. Écrire une fonction qui prend n en argument et renvoie la liste des termes $u_0, u_1, \dots, u_n$.

Exercice 5 : A l'aide d'une écriture par compréhension, écrire la liste des images de l'ensemble $[\![1;20]\!]$ par la fonction $f : x \mapsto \dfrac{e^x}{x+1}$.

Exercice 6 : On considère la fonction $f(x) = \ln(x)$ définie sur $\mathbb{R}_+^*$.

1. Créer la liste formée de $f(1)$, $f(2)$, ..., $f(10)$
2. Écrire une fonction en `Python` qui prend en paramètre un entier n et renvoie la liste des valeurs $f(k)$ pour $1 \leq k \leq n$.

3. D'autres commandes utiles

Mémo 5 (Création avec range)

La commande `range` parcourt une liste, mais sans la créer. Il est possible de le faire avec la fonction `list` :

1. `list(range(a,b))` affiche la liste des entiers entre a et $b-1$.
2. `list(range(a,b,m))` affiche la liste des entiers entre a et $b-1$ avec un pas de m

Exercice 7 :

1. Créer la liste des entiers de 1 à 10.
2. Créer la liste des entiers pairs entre 1 et 10.

TP 10

Commandes sur les listes

1. L'essentiel

Mémo 1 (Extraction d'une liste)

1. `L[i]` correspond à l'élément d'indice i de la liste en sachant que la numérotation commence à 0.
2. `L[i:j]` est la liste constituée des éléments de L, de l'indice i (inclus) à j (exclu).


On retient que le premier élément s'obtient avec `L[0]`, le 5e élément s'obtient avec `L[4]`.

Code (Dans la console)

```
>>> L=[i**2 for i in range (11)]
>>> L[3]
>>> L[0]
>>> L[-1]
>>> L[-2]
>>> L[2:7]
```

Mémo 2 (Longueur d'une liste)

`len(L)` donne la longueur de la liste `L` (son nombre d'éléments).

```
>>> L = [5,6,8,1]
>>> len(L)
```

Mémo 3 (Commandes pratiques)

1. `L[i]=a` remplace l'élément en position `i` par `a`.
2. `L.append(a)` ou `L=L+[a]` ajoute l'élément a à la fin de la liste `L`.
3. `a in L` permet de tester si `a` est présent dans la liste `L` (le résultat est un booléen, c'est-à-dire `True` ou `False`).
4. `L.count(a)` compte le nombre de fois où l'élément a apparaît dans la liste `L`.

```
>>> L[2]=7
>>> L
>>> L.append(15)
>>> del L[2]
>>> 7 in L
>>> L.count(7)
>>> -5 in L
```

2. Exercices

Exercice 1 (Recherche d'un élément) **:** Créer de deux façons des fonctions `Recherche` et `Recherche2` qui prennent en entrée une liste `L` et un élément `a` et renvoient `True` si `a` est dans la liste `L` et `False` sinon. L'une des deux méthodes utilisera la commande in, l'autre non.

Exercice 2 (Recherche du maximum et du second maximum d'une liste) **:**

1. Créer une fonction `max1` qui prend en entrée une liste de réels `L` et renvoie la valeur maximale de cette liste.
2. Créer une fonction `max2` qui prend en entrée une liste de réels `L` et renvoie la valeur maximale de cette liste et l'indice du maximum (on prendra le premier indice si la valeur maximale apparaît plusieurs fois).
3. Écrire une fonction `max3` qui donne le premier et le second maximum.

Exercice 3 (Statistiques) **:**

1. Créer une fonction `moyenne` qui prend en entrée une liste de réels `L` et renvoie la moyenne des valeurs de la liste.
2. L'utiliser pour créer une fonction `variance` qui prend en entrée `L` et renvoie la variance des valeurs de la liste.

3. D'autres commandes utiles

Mémo 4 (Extraction d'une liste)

1. `L[i:j:k]` est la liste constituée des éléments de la liste de l'indice i (inclus) à l'indice j (exclu) avec un pas de k.
2. `L[i:]` est la liste obtenue en considérant tous les éléments de L à partir de l'indice i (inclus).
3. `L[:j]` est la liste constituée des éléments de L de l'indice 0 (inclus) à l'indice j (exclu).
4. `L[-i:]` est la liste constituée des i derniers éléments de L
5. `L[:-j]` est la liste constituée des éléments de L privée des j derniers éléments.

Mémo 5 (Supprimer des éléments d'une liste)

1. `del L[i]` supprime l'élément en position `i` de la liste `L`
2. `del L[i:j]` supprime les éléments de la position `i` (incluse) à la position `j` (exclue) de la liste `L`

4. Approfondissement

Exercice 4 : Écrire une fonction qui prend en paramètre une liste `L` et échange le premier et le dernier élément de cette liste. On créera une nouvelle liste `L_modif` avec ces deux éléments inversée.

```
>>> L = [1,2,8,6,4,-1,2,3]

>>> echange(L)
[3, 2, 8, 6, 4, -1, 2, 1]
```

Exercice 5 : Écrire une fonction `parite` qui prend en paramètres une liste `L` et renvoie deux listes `(L_even, L_odd)` où `L_even` est composée des éléments de `L` d'indices pairs et `L_odd` est composée par les éléments d'indices impairs.

```
>>> L = [1,2,8,6,4,-1,2,3]

>>> parite(L)
([1, 8, 4, 2], [2, 6, -1, 3])
```

On pourra utiliser l'opération `%` qui renvoie le reste de la division euclidienne : par exemple `20%3` renvoie 2.

Exercice 6 :

1. Écrire une fonction qui demande à l'utilisateur de saisir 10 nombres et renvoie la liste de ces 10 nombres.
2. Écrire une fonction qui renvoie la plus grande valeur et la plus petite valeur de cette liste.

TP 11

Algorithmes autour des listes

Nous présentons ici quelques algorithmes classiques sur les listes.

1. Calcul de moyenne

Pour calculer la moyenne des éléments d'une liste L contenant des valeurs numériques, il suffit d'additionner les éléments de la liste et de diviser par le nombre d'éléments de cette liste.

```
def moyenne(L):
    somme = ....
    for e in L :
       somme = somme + ....
    return somme/....
```

2. Algorithmes de recherche

a) Recherche du minimum

Dans le TP précédent, nous avions déjà vu comment déterminer le maximum d'une liste L, c'est la même idée pour le minimum.
Le principe est d'initialiser une variable avec la première valeur de la liste puis de parcourir la liste et de conserver dans cette variable le plus petit élément.

```
def minimum(L):
    min = ...
    for e in L :
       if e < min:
          min = ....
    return min
```

b) Recherche séquentielle du nombre d'occurrence

La recherche d'occurrence consiste à trouver combien de fois un élément a apparaît dans une liste L. Il existe une commande toute prête pour cela dans Python, la commande `count` mais ici nous allons détailler le programme pour voir comment il fonctionne.
Le principe est de parcourir la liste et d'*incrémenter* un compteur à chaque fois que l'on trouve l'élément cherché (incrémenter = augmenter de 1).

Tester le code avec la liste $L = [1,2,2,5,6,2,8,4,5]$ et l'élément 2.

```
def occurrence(L,a):
    occ = ...
    for e in L:
        if e == ...:
            occ = occ + ...
    return occ
```

3. Algorithmes de tri

a) Algorithme de tri par sélection du minimum

Le tri par sélection consiste à parcourir toute la liste pour trouver l'élément le plus petit. Une fois celui-ci trouvé, on échange sa place avec le premier élément de la liste avant de recommencer en parcourant la liste à partir du second élément et ainsi de suite.

Exercice 1 : Appliquer cet algorithme sur papier avec la liste $L = [4,2,5,6,3,9,8]$.

```
def tri_minimum(L):
    for i in range(len(L)-1):
        a = L[i]
        indice_min = i
        for j in range(i+1,len(L)):  # la liste L a déjà été triée jusqu'au rang i-1
            if L[indice_min]>L[j]:
                indice_min = ...   # indice du minimum
        L[i] = ......
        L[indice_min] = .....
    return L
```

Tester le code avec la liste précédente.

b) Algorithme de tri par insertion

Le tri par insertion est généralement le tri que l'on utilise pour trier des cartes lorsqu'on ramasse un jeu.

- ✔ On laisse le premier élément de la liste que l'on place en position 0.
- ✔ On prend le deuxième élément de la liste (celui en position 1). S'il est plus grand que l'élément en position 0, alors on le laisse à la place ; sinon, on échange les deux nombres.
- ✔ On considère ensuite le troisième élément (en position 2). S'il est supérieur au nombre en position 1, alors on le laisse à sa place. Sinon, s'il est inférieur au nombre en position 1 et supérieur au nombre en position 0, on échange les nombres situés en positions 1 et 2. Sinon, on place ce troisième élément en début de liste.

Exercice 2 : Vérifier à la main la compréhension de cet algorithme avec la liste $L = [11,6,9,17,5,23,2]$.

```
def tri_insertion(L):
    for i in range(1,len(L)):
        k = L[i]
        j = i-1
        while j >=... and k <... :
            L[j+1] = ...
            j = j-1
        L[j+1] = .....
    return L
```

4. Recherche dans un tableau trié

Maintenant que nous savons trier un tableau, nous pouvons nous poser la question de savoir si l'on ne pourrait pas trouver un algorithme plus efficace, que celui qui consiste à passer tous les termes d'une liste en revue, pour trouver une occurrence dans une liste. En effet, si cette valeur se trouve en fin de liste, nous allons perdre beaucoup de temps (car il faudra parcourir presque tout le tableau), alors même que le tableau est déjà trié et qu'il y a donc un ordre dans les données.

Nous allons donc utiliser cet ordre pour effectuer ce que l'on appelle une recherche dichotomique :

1. Nous allons prendre le milieu du tableau et vérifier si la valeur à cet endroit est supérieure ou inférieure à ce que l'on cherche :
   (a) Si le milieu est plus petit : on cherche à droite
   (b) Si le milieu est plus grand : on cherche à gauche
2. Le tableau est alors divisé en deux et on ne conserve que la partie où est susceptible de se trouver la valeur recherchée.
3. On recommence l'opération en divisant à nouveau le demi-tableau jusqu'à tomber sur la bonne valeur.

Exercice 3 : Vérifiez que vous avez compris la méthode en l'appliquant à la main à la liste $L = [1,2,5,9,10,15,18,19,23,25]$ avec l'élément 2.

```
def occurrence_triée(L,x):
    vide="pas d'occurrence"
    a = 0
    b = len(L)-1
    while a <= b:
        m = ....
        valeur_m = L[m]
        if valeur_m==x:
            return ...
        elif valeur_m<x:
            a = ...
        else:
            b = ....
    return vide
```

Tester le code avec la liste $L = [1,2,5,9,10,15,18,19,23,25]$ et l'élément 2.

5. Un peu de théorie : complexité d'un algorithme

En programmation, certains algorithmes autour des listes vont revenir très souvent : recherche dans une liste, tri d'une liste, nombre d'occurrence dans une liste, etc. On cherchera toujours à trouver les méthodes les plus efficaces pour résoudre ces problèmes, à la fois du point de vue de l'écriture du code que de son exécution.

En informatique, la notion « complexité » d'un algorithme, indique grossièrement le coût en termes de temps d'exécution de l'algorithme.

En considérant que l'ensemble de données de départ possède n éléments, la complexité d'un algorithme va être donnée en fonction de ce nombre n.

Par exemple, s'il faut parcourir une fois toute la liste, on dira que la complexité est en n et donc qu'elle est linéaire. Par exemple, passer en revue une liste de 1000 éléments nécessitera 1000 opérations.

Si l'algorithme doit imbriquer deux boucles et parcourir les n éléments de la liste pour chaque boucle, alors la complexité sera en n^2. Par exemple pour une liste de 1000 éléments, on effectuera 1 000 000 opérations.

⟶ En informatique, on peut calculer la complexité d'un programme (hors programme en ECG), cela permet de l'optimiser pour que le temps de calcul soit réduit. Mieux vaut une complexité de l'ordre de n que de l'ordre de n^2 ou n^3. Sur de gros calculs, le gain de temps peut être énorme.

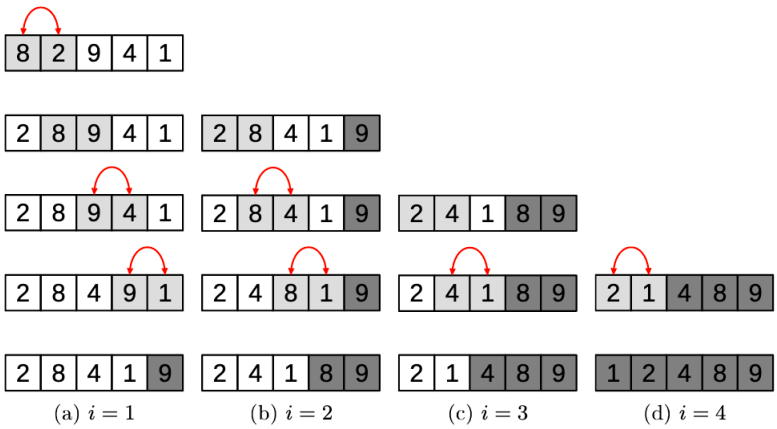
6. Approfondissement

Voici une autre méthode de tri (qui ne figure pas au programme) : le tri à bulles.

Le principe de cet algorithme est de parcourir la liste en comparant deux éléments consécutifs. S'ils ne sont pas dans le bon ordre, on les échange.

Après un premier passage, le plus grand élément doit être placé en fin de liste. On recommence un nouveau passage en s'arrêtant à l'avant-dernier élément, et ainsi de suite.

Un exemple :



(a) $i = 1$ (b) $i = 2$ (c) $i = 3$ (d) $i = 4$

→ Coder cet algorithme en Python.

TP 12

Algorithmes gloutons

1. L'essentiel

Avant de commencer :
Pour une présentation rapide du principe d'un algorithme glouton, commencez par consulter la vidéo suivante : https://www.youtube.com/watch?v=ypeLe6JtZCU

À retenir :
Un algorithme glouton (greedy algorithm) est un algorithme qui suit le principe de faire, étape par étape, un choix optimal local. Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur le sous-problème restant à résoudre sans jamais remettre en question les choix précédents. La méthode gloutonne consiste à choisir des solutions locales optimales d'un problème dans le but d'obtenir une solution optimale globale au problème.

2. Le problème du rendu de monnaie

Vous êtes commerçant et devez rendre de la monnaie à vos clients de façon optimale, c'est-à-dire avec le nombre minimal de pièces et de billets.

- ✔ On suppose que les clients ne vous donnent que des sommes entières en euros (pas de centimes pour simplifier).
- ✔ Les valeurs des pièces et billets à votre disposition sont : 1, 2, 5, 10, 20, 50, 100 et 200. On suppose que vous avez autant d'exemplaires que nécessaire de chaque pièce et billet.
- ✔ Dans la suite, afin de simplifier, nous désignerons par « pièces » à la fois les pièces et les billets.

Pour minimiser le nombre de pièces à rendre, on adopte la stratégie suivante :

- ✔ On commence par rendre la pièce de plus grande valeur possible.
- ✔ On déduit (retranche) cette valeur de la somme à rendre.
- ✔ On recommence, jusqu'à obtenir une somme à rendre nulle.

En procédant ainsi, on se rend compte que l'on résout le problème étape par étape et qu'un choix optimal est fait à chaque étape (la pièce de plus grande valeur). Cette stratégie entre donc bien dans la catégorie des algorithmes gloutons.

Exercice 1 : Appliquer l'algorithme avec les pièces de 1, 2, 5, 10, 20, 50, 100 et 200 euros et 158€ à rendre.

→ `somme_a_rendre` est un entier qui contient la somme à rendre
→ `pieces` est une liste qui contient la liste des pièces que l'on peut utiliser (rangées en ordre croissant).
→ `lst_pieces` est créée dans ce programme : c'est une liste à compléter avec les pièces à rendre.

```
def pieces_a_rendre(somme_a_rendre, pieces):
    lst_pieces = ...
    i = len(pieces) - 1
    while somme_a_rendre ....:
        if somme_a_rendre < pieces[i]:
            i =...
        else:
            lst_pieces.append(..........)
            somme_a_rendre =..............
    return ............
```

3. Allocation de salles

Le problème est le suivant. On suppose que l'on a n demandes de réservation pour une salle avec pour chacune d'entre elles l'heure de début et l'heure de fin (on supposera que l'on n'a pas besoin de période de battement entre deux réservations). Bien sûr, à un instant donné, la salle ne peut être réservée qu'une fois ! On cherche à donner satisfaction au maximum de demandes.

L'idée de l'algorithme est la suivante :

- ✔ On choisit la demande qui se termine en premier.
- ✔ On rejette toutes les demandes qui commencent avant la fin de la première demande choisie.
- ✔ On retourne à la première étape.

Exercice 2 : Voici 8 réservations avec les créneaux de début et de fin : (3;8) ; (0;6) ; (1;4) ; (6;8) ; (4;7) ; (5;11) ; (7;9) ; (9;10). Comment donner satisfaction à un maximum de personnes ?

On note les créneaux demandés sous forme de liste de la manière suivante : `[heure_début,heure_fin]`

L'ensemble des créneaux formera donc une liste de listes.

On commence par trier les créneaux par ordre de fin croissante. Pour cela, on réutilise un algorithme vu au TP 11.

```
def tri_minimum(L):
    for i in range(len(L)-1):
        a=L[i]
        min=i
        for j in range(i+1,len(L)):
            if L[min][1]>L[j][1]:
                min=j
        L[i]=L[min]
        L[min]=a
    return L
```

La seule modification a lieu à la ligne 6 du code. Il faut indiquer à l'algorithme de comparer les heures de fin. Ces dernières se trouvent en position 1 (rappel, la numérotation des listes commence à 0) des sous-listes.

Tester le code avec la liste de créneaux précédents.

```
def planning(tab_intervalles):
    nb_intervalles = len(tab_intervalles)
    tab_intervalles = tri_minimum(tab_intervalles)
    tab_planning = ..........
    j = 0
    for i in range(1,nb_intervalles):
        if tab_intervalles[i][...] >= tab_intervalles[j][...]:
            tab_planning.append(tab_intervalles[i])
            j = ...
    return tab_planning
```

Tester le code avec la liste de créneaux précédents.

4. Exercices

Exercice 3 (Regrouper des fichiers sur des clés USB) **:**

On dispose de fichiers de tailles différentes, exprimées en Go, et de clés USB ayant toutes la même capacité C.

Par exemple :

```
fichiers = [4, 8, 1, 4, 2, 7, 3, 5]
C = 10
```

On veut répartir les fichiers dans un nombre aussi petit que possible de clés USB.

On utilise l'algorithme glouton suivant :

- on trie d'abord les fichiers par taille décroissante ;
- on prend ensuite les fichiers dans cet ordre ;
- pour chaque fichier, on le place dans la première clé USB où il reste assez de place ;
- si aucune clé USB déjà utilisée ne peut contenir ce fichier, on crée une nouvelle clé.

1. Appliquer cet algorithme à la main sur l'exemple présenté ci-dessus.
   On indiquera les clés USB obtenues à chaque étape.
2. Expliquer pourquoi cet algorithme est un algorithme glouton.
3. Compléter le programme suivant. La fonction doit renvoyer une liste de clés USB, chaque clé étant elle-même une liste contenant les tailles des fichiers placés dessus.
   On supposera dans un premier temps que la liste `fichiers` est déjà triée par ordre décroissant.

```
def cles_usb(fichiers, C):
    cles = []

    for fichier in fichiers:
    place = False # indique que le fichier n?est pas encore enregistré
        for cle in cles :
        # on essaie d?enregistrer le fichier dans la cle
             if place == False and sum(cle) + fichier <= C:
                cle.append(...)
                place = ... # le fichier est enregistré

        if place == ...:
        # aucune clé n'a assez de place
            cles.append([fichier]) # on ajoute une nouvelle clé

    return ...
```

4. Tester la fonction avec la liste donnée.
5. Afficher le nombre de clés USB utilisées.
6. Donner un exemple où cette stratégie n'est pas forcément optimale.

Exercice 4 (Minimiser le nombre d'arrêts à une station-service) **:**

Une voiture doit parcourir une distance totale D.

Son réservoir plein lui permet de parcourir au maximum R kilomètres. On connaît les positions des stations-service sur la route, mesurées en kilomètres depuis le point de départ.

Par exemple :

```
D = 950
R = 400
stations = [200, 375, 550, 750]
```

La voiture part du kilomètre 0 avec le plein et veut arriver au kilomètre D.

On veut s'arrêter le moins souvent possible.

La stratégie gloutonne consiste à aller, à chaque étape, jusqu'à la station la plus éloignée encore atteignable.

1. Appliquer cette stratégie à la main avec les données suivantes : $D = 950$, $R = 400$ et `stations = [200, 375, 550, 750]`.
2. Écrire une fonction `stations_choisies(D, R, stations)` qui renvoie la liste des stations où la voiture doit s'arrêter.
3. Si le trajet est impossible, la fonction devra renvoyer la chaîne de caractères `"impossible"`.
4. Tester la fonction avec :

   ```
   D = 950
   R = 400
   stations = [200, 375, 550, 750]
   ```

5. Tester ensuite avec :

   ```
   D = 950
   R = 300
   stations = [200, 375, 550, 750]
   ```

   Que constate-t-on ?
6. Modifier la fonction pour qu'elle renvoie seulement le nombre minimal d'arrêts nécessaires.
7. Expliquer pourquoi la stratégie gloutonne semble naturelle dans ce problème.

On pourra commencer par ajouter l'arrivée à la liste des stations :

```
stations = stations + [D]
```

On pourra ensuite utiliser l'idée suivante : tant que l'arrivée n'est pas atteinte, on cherche la station la plus éloignée accessible depuis la position actuelle.

```
def stations_choisies(D, R, stations):
    stations = stations + [D]
    arrets = []
    position = 0
    i = 0

    while position + R < D:
        derniere_station = position

        while i < len(stations) and stations[i] <= position + R:
            derniere_station = stations[i]
            i = i + 1
        if derniere_station == position:
            return "impossible"
        if derniere_station != D:
            arrets.append(derniere_station)

        position = derniere_station
    return arrets
```

Exercice 5 (Rendu de monnaie avec stock limité) **:** Un commerçant doit rendre une somme entière S en euros.

Il dispose des pièces et billets suivants, mais son stock est limité :

```
pieces = [1, 2, 5, 10, 20, 50]
stock = [10, 5, 3, 4, 2, 1]
```

Cela signifie qu'il dispose de 10 pièces de 1 euro, de 5 pièces de 2 euros, de 3 billets de 5 euros, etc.

On veut appliquer une stratégie gloutonne : à chaque étape, on utilise la plus grande pièce ou le plus grand billet encore disponible, tant que cela ne dépasse pas la somme restante à rendre.

1. Appliquer à la main cette stratégie pour rendre $S = 87$ euros.
2. Écrire une fonction `rendu_limite(S, pieces, stock)` qui applique cette stratégie gloutonne et renvoie la liste des pièces et billets utilisés.
3. Si la somme ne peut pas être rendue exactement, la fonction devra renvoyer la chaîne de caractères `"impossible"`.
4. Tester la fonction avec les valeurs $S = 87$, $S = 126$ et $S = 194$.
5. Modifier la fonction pour qu'elle renvoie également le nombre total de pièces et billets utilisés.
6. Donner un exemple de somme S qui ne peut pas être rendue exactement avec ce stock.
7. Expliquer pourquoi le stock limité peut empêcher de rendre une somme, même lorsque la valeur totale des pièces et billets disponibles est suffisante.

On pourra partir du squelette suivant :

```
def rendu_limite(S, pieces, stock):
    rendu = []
    reste = S

    i = len(pieces) - 1

    while i >= 0:
        while reste >= pieces[i] and stock[i] > 0:
            ...
            ...
            ...
        i = ...

    if reste == 0:
        return rendu
    else:
        return "impossible"
```

Troisième partie

Représentations graphiques

TP 13

Représentations graphiques de fonctions

1. L'essentiel

Dans ce TP on utilisera les librairies `numpy` pour construire des vecteurs et `pyplot` pour tracer les courbes. On les importe avec les commandes :

```
import numpy as np
import matplotlib.pyplot as plt
```

Mémo 1 (`pyplot` : principe général)

- On construit deux vecteurs `x` et `y` qui contiennent respectivement les abscisses et ordonnées des points du graphique
- `plt.plot(x,y)` permet de tracer la figure qui relie tous ces points
- `plt.show()` permet de voir le graphique

Dessinez sur papier ce que produit le programme suivant :

```
x = np.array([0,2,3,4,4,3,4,2,2,0])
y = np.array([0,0,1,0,2,3,4,4,2,0])
plt.plot(x,y)
plt.show()
```

Mémo 2 (Tracé du graphe d'une fonction $y = f(x)$)

Étude d'un exemple : représentation de la fonction $f(x) = x^2$ sur $[-2;3]$

```
import numpy as np
import matplotlib.pyplot as plt

def f(x) :
    return x**2

x = np.linspace(-2,3,100)
y = f(x)

plt.plot(x,y)
plt.show()
```

On note ici que

— la fonction $f(x) = x^2$ est définie sous forme d'une fonction `Python`. On peut facilement la modifier si on veut tracer une autre fonction

— la commande `np.linspace(-2,3,100)` crée un vecteur qui comporte 100 valeurs réparties entre -2 et 3.

— le vecteur `y` des ordonnées est directement obtenu à l'aide de la fonction `f`. C'est l'avantage d'utiliser des vecteurs plutôt que des listes.

2. Exercices

Exercice 1 :

1. On veut tracer la fonction inverse entre 0,5 et 10.
   (a) Créer en `Python` la fonction f définie sur $\mathbb{R}^*$ par $f(x) = 1/x$.
   (b) Créer `x1` un vecteur contenant 10 points entre 0,5 et 10.
   (c) Créer `y1` comme l'image du vecteur `x1`.
   (d) Tracer la courbe de f et la faire afficher.
2. Pour améliorer le tracé, on va ajouter des points au vecteur `x`.
   (a) Dans le programme précédent, ajouter un vecteur `x2` affichant 100 points entre 0,5 et 10 et un vecteur `y2` image du vecteur `x2`. Tracer à nouveau la courbe de f.
   (b) Deux courbes sont affichées, que constatez-vous ?

Exercice 2 : On s'intéresse à la fonction exponentielle.

1. Tracer sa représentation graphique sur $[-1,1]$
2. Déterminer l'équation de sa tangente au point d'abscisse 0.
3. Tracer cette tangente sur le même graphique.

Exercice 3 (HEC/ESSEC 2023) **:** On considère la fonction γ définie sur $\mathbb{R}$ par

$$\gamma(t) = \begin{cases} 1 & \text{si } t < 0 \\ 0 & \text{si } t > 1 \\ 1 - 3t^2 + 2t^3 & \text{si } t \in [0,1] \end{cases}$$

1. Écrire une fonction Python `gamma(t)` qui calcule et renvoie la valeur $\gamma(t)$, t étant donné.
2. Utiliser la fonction précédente pour écrire un script qui affiche le graphe de γ sur le segment $[-1,2]$ dans un repère.
   Attention : on ne peut pas directement calculer le vecteur `y` avec la commande `y = f(x)`. En effet les conditions `if / else` contenues dans la fonction ne peuvent pas s'appliquer à un vecteur. Demander de l'aide si besoin pour trouver une autre méthode.

3. D'autres commandes utiles

Mémo 3 (Options de `plt.plot`)

Pour améliorer le rendu et la lisibilité du graphique, on peut ajouter des options. On les place entre guillemets après les ordonnées : par exemple `plt.plot(x,y,'b:+')` pour avoir une fonction représentée par des points en forme de croix de couleur bleue reliés par des pointillés.

couleur	b (bleu), c (cyan), g (vert), k (noir), m (magenta), r (rouge), w (blanc), y (jaune)
type de trait	- (ligne continue), – (ligne discontinue), : (pointillés), -. (trait mixte)
type de marqueur	+ (croix), * (astérisque), o (cercle), d (losange), s (carré), x (croix)

Mémo 4 (Commenter les courbes)

On donne la légende à l'aide des commandes suivantes :

1. `plt.plot(x1,y1,'options',label='nom_de_la_courbe')` permet de définir une légende pour la courbe.
2. `plt.legend()` affiche la légende définie par label='nom' dans `plt.plot`
3. `plt.xlabel("nom_axe_des_abscisses")` permet de nommer l'axe des abscisses.
4. `plt.ylabel("nom_axe_des_ordonnées")` permet de nommer l'axe des ordonnées.
5. `plt.title('nom du graphique')` donne un titre au graphique.
6. `plt.xlim(-10,10)` permet de tracer un graphique dont les abscisses sont comprises entre -10 et 10. On utilise de même `plt.ylim`
7. `plt.grid()` permet d'afficher une grille

TP 14

Représentations graphiques de suites

Le principe général est le même que pour tracer des fonctions, mais deux éléments sont à prendre en compte :

— en abscisse on a besoin uniquement de nombres entiers, donc la commande `np.linspace` n'est pas adaptée

— la boucle `for` n'est pas compatible avec les vecteurs, donc on ne peut pas directement calculer `y=f(x)` dès qu'on a une relation de récurrence

1. L'essentiel

Mémo 1 (Tracé du graphe d'une suite (u_n))

Étude d'un exemple : représentation des 10 premiers termes de la suite définie par $u_0 = 3$ et $u_{n+1} = \dfrac{u_n}{u_n + 2}$

Code

```
import numpy as np
import matplotlib.pyplot as plt

def u(n) :
    u = 3
    for i in range(n) :
        u = u/(u+2)
    return u

x = np.arange(10)
y = np.zeros(10)
for i in range(10) :
    y[i] = u(i)

plt.plot(x,y,'+')
plt.show()
```

On note ici que

— la suite (u_n) est définie sous forme d'une fonction `Python`. On peut facilement la modifier si on veut tracer une autre suite

— la commande `np.arange(10)` crée un vecteur qui comporte les entiers de 0 à 9.

— pour calculer y on commence par créer un vecteur ne contenant que des 0. Puis on les remplace à l'aide de la boucle `for` par les valeurs u_n.

— dans la commande `plt.plot(x,y,'+')`, on précise avec `'+'` de tracer des points : ils ne seront pas reliés, ce qui est cohérent avec une représentation de suite.

2. Exercices

Exercice 1 : Soit (u_n) la suite définie par $u_0 = 0,5$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = \frac{1}{3}(4-u_n)^2$

1. Représenter graphiquement les 100 premiers termes.
2. Conjecturer la convergence de la suite.

Exercice 2 : Soit (u_n) la suite définie sur $\mathbb{N}$ par $u_0 = 3$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = \frac{u_n}{2-u_n}$.

1. Représenter graphiquement les 20 premiers termes et conjecturer la limite de la suite.
2. Écrire un programme qui permette de déterminer le premier rang pour lequel $|u_n| < 10^{-4}$

Exercice 3 : Soit (u_n) la suite définie par $u_n = \sum_{k=1}^{n} \frac{1}{k^2}$.

1. Représenter graphiquement les 50 premiers termes.
2. Sur le même graphe, tracer la droite d'équation $y = \frac{\pi^2}{6}$. Qu'en pensez-vous ?

3. Approfondissement

Exercice 4 :

1. (a) Représenter graphiquement la suite définie par $u_0 = 0$, $u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$ $\forall n \in \mathbb{N}$.
   (b) Représenter graphiquement la suite définie par $v_n = \frac{u_{n+1}}{u_n}$ pour $n \geq 1$ (avec (u_n) la suite définie à la question précédente). Que remarque-t-on ?
2. Représenter graphiquement la suite définie par $u_0 = -2$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = 2u_n - n$

Exercice 5 : On considère la suite (u_n) définie par $u_0 = 0$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = \frac{u_n}{2} + \frac{1}{2}$.

1. Représenter graphiquement les 50 premiers termes de la suite (u_n).
2. Graphiquement, conjecturer :
   (a) Les variations de la suite
   (b) Un encadrement
   (c) Sa limite
3. Démontrer successivement les 3 points ci-dessus.
4. Recommencer en considérant cette fois $u_0 = 2$.

Exercice 6 : On s'intéresse à la suite (u_n) définie par $u_0 \in \mathbb{R}$ et pour tout $n \in \mathbb{N}$,

$$u_{n+1} = u_n^2 - u_n + 1$$

On souhaite déterminer le comportement de cette suite pour différentes valeurs de u_0.

Faites des tests en faisant varier la valeur u_0 choisie. Conjecturer, selon cette valeur, le sens de variation et la limite de la suite.

Démontrez ensuite les résultats obtenus !

Quatrième partie

Matrices et utilisations

TP 15

Matrices

Dans ce TP nous aurons besoin des librairies `numpy` et `linalg`. On les importe avec les commandes suivantes :

```
import numpy as np
import numpy.linalg as al
```

1. L'essentiel

Pour définir des matrices en Python on utilise un type de variable `array`, autrement dit un tableau de nombres. Il faudra différencier les opérations de Python sur les tableaux (effectuées coefficient par coefficient) et les opérations matricielles utilisées en mathématiques.

Mémo 1 (Définir une matrice)

✔ En saisissant tous ses coefficients : `np.array([[1,2,3],[4,5,6]])` pour la matrice $\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}$.


Chaque ligne est entourée de crochets, et une autre paire de crochets encadre la matrice.
S'il s'agit d'une matrice ligne, on peut simplifier l'écriture : `np.array([1,2,3])`

✔ Matrices usuelles :

— `np.zeros([2,3])` pour la matrice nulle $0_{2,3}$

— `np.ones([2,3])` pour la matrice de taille $(2,3)$ qui ne contient que des 1

— `np.eye(2,3)` pour la matrice « identité » de taille $(2,3)$

Les syntaxes `np.zeros((2,3))` et `np.ones((2,3))` sont également correctes.
On peut utiliser `np.eye(5)` pour la matrice I_5.

✔ En modifiant une matrice définie avec les formules ci-dessus : la commande `M[i,j]` permet d'accéder à l'élément de la ligne `i` et la colonne `j` dans la matrice `M`.
Celui-ci est considéré comme une variable, on peut par exemple le modifier avec `M[2,4] = -5`.


`Python` numérote les lignes et colonnes en partant de 0. Par exemple dans la matrice $M=\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}$:
- l'instruction `M[1,2]` renvoie ...
- on accède à la valeur 2 de cette matrice avec `M[...,...]`

Mémo 2 (Opérations matricielles)

Pour effectuer les opérations mathématiques sur les matrices, on utilise les commandes suivantes :

— pour la somme ou la soustraction de matrice : `M + P` et `M - P`

— pour la multiplication par un scalaire : `3*M`, on peut aussi diviser par un scalaire `M / 3`

— pour le produit matriciel : `np.dot(M,P)`

— pour calculer une puissance d'une matrice : `al.matrix_power(M,n)` donnera M^n

— pour déterminer l'inverse d'une matrice : `al.inv(M)`

On suppose ici que toutes ces opérations sont possibles ; dans le cas contraire `Python` affichera un message d'erreur.

 La plupart des opérations sont effectuées coefficient par coefficient (on se souvient que `Python` considère nos matrices comme des tableaux de nombres). Par exemple `M*P` ne calcule pas le produit matriciel. Testez-le sur un exemple.
Testez également les calculs `M/P`, puis `M>P` et `M**P`.
Le calcul `M+1` s'effectue aussi sur tous les coefficients : on ajoute `1` à chaque coefficient de `M`. On se souvient que cette écriture n'est pas valide en mathématiques.

Mémo 3 (Dimensions d'une matrice)

La commande `np.shape(M)` renvoie les dimensions de la matrice `M` : son nombre de lignes et son nombre de colonnes.
Par exemple : `n, p = np.shape(M)` peut être utile si on ne connait pas à l'avance la taille de la matrice utilisée.

2. Exercices

Exercice 1 : Définir en Python, avec la méthode la plus efficace, les matrices suivantes :

$$A = \begin{pmatrix} 2 & 2 & -6 & 4 \\ 3 & 1 & 0 & 1 \\ 0 & 2 & -3 & -1 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & -5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad C = \begin{pmatrix} 4 & 1 & 1 & 1 & 1 \\ 1 & 4 & 1 & 1 & 1 \\ 1 & 1 & 4 & 1 & 1 \\ 1 & 1 & 1 & 4 & 1 \\ 1 & 1 & 1 & 1 & 4 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad D = 0_{3,7}$$

Exercice 2 : Quelle est la matrice produite par ce programme ?

```
M = np.ones([4,5])
M[2,3] = 7
M[0,2] = -4
M[3,3] = 0
```

Exercice 3 : Que renvoie la fonction suivante ?

```
import numpy as np
def diag(n):
   A = np.zeros([n,n])
   for i in range(n):
      A[i,i] = i
   return A
```

Exercice 4 : Traduire mathématiquement les résultats donnés par ce programme :

Code

```
>>> import numpy as np
>>> import numpy.linalg as al
>>> A = np.array([[1,0,2],[2,1,1],[-1,2,0]])
>>> B = al.matrix_power(A,3)-2*al.matrix_power(A,2)+A-8*np.eye(3)
>>> B
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Exercice 5 : Soit A la matrice définie par : $A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

1. Saisir la matrice A en Python puis calculer A^2, A^3, A^4...
2. Conjecturer une expression explicite pour A^n, puis la prouver par récurrence.

Exercice 6 (Puissances et inverse d'une matrice) **:** On considère les matrices $N = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ et $A = I_3 + N$.

1. Définir les matrices N, I_3 et A en `Python`.
2. Calculer N^2, N^3 et N^4. Que constate-ton ?
3. Calculer les matrices A^n pour $n = 1,2,3,4,5$. On pourra utiliser `al.matrix_power`.
4. Conjecturer une formule pour A^n. On pourra la donner en fonction de I_3, N et N^2.
5. En utilisant le fait que $N^3 = 0_3$, proposer une expression de A^{-1}, puis vérifier avec `Python`.

Exercice 7 (Puissances et polynôme annulateur) **:** On considère la matrice $A = \begin{pmatrix} 0 & -2 \\ 1 & 3 \end{pmatrix}$.

1. Définir la matrice A en `Python`.
2. Calculer la matrice $A^2 - 3A + 2I_2$ en `Python`.
3. Compléter la fonction suivante, qui calcule A^n sans utiliser `al.matrix_power`

```
def puissance_A(n) :
    A = ...
    I = np.eye(2)

    if n == 0 :
        return ...
    if n == 1 :
        return ...

    P = A
    for k in range(1,n+1) :
        P = ...
    return P
```

Vérifier votre programme en comparant par exemple `puissance_A(10)` avec `al.matrix_power(A,10)`.

3. D'autres commandes utiles

Mémo 4 (Accès aux lignes et aux colonnes)

— `M[i]` permet d'accéder à la ligne `i` de la matrice (pour l'utiliser ou pour la modifier).

— `M[:,j]` permet d'accéder à la colonne `j` de la matrice.

On se souvient que `Python` numérote lignes et colonnes en commençant à `0`.

Mémo 5 (D'autres fonctions matricielles)

— `np.transpose(M)` renvoie la transposée de la matrice `M`

— On peut appliquer une fonction sur une matrice : testez par exemple `np.exp(M)`. Cela concerne aussi les fonctions que vous avez vous-même définies (avec `def` / `return`) : on peut appliquer `f(M)` si `f` est une fonction déjà programmée et compatible avec les tableaux (donc sans condition `if` ou boucle `for` par exemple).

4. Exercices d'approfondissement

Exercice 8 :

1. Écrire une fonction **matprod** qui effectue le produit de deux matrices lorsque c'est possible et vous signale l'erreur dans le cas contraire.
2. Écrire une fonction **mattrace** qui calcule la trace d'une matrice carrée, c'est-à-dire la somme des coefficients diagonaux de cette matrice.
3. Écrire une fonction **danstab** qui teste si un élément `x` appartient à une matrice `A`.

Exercice 9 : Créer une fonction qui prend en paramètre un entier `n` et renvoie la matrice $n \times n$ suivante :

$$\begin{pmatrix} 1 & 1 & 0 & \dots & 0 \\ 1 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & 0 & 0 & \ddots & 1 \\ 1 & 0 & 0 & \dots & 0 \end{pmatrix}.$$

Exercice 10 : Quelle matrice est créée par cette fonction ?

```
import numpy as np
def mystere(n) :
   P=np.zeros([n,n])
   print(P)
   for i in range(n):
      P[i,i]=1
      P[i,0]=1
   print(P)
   for i in range(2, n):
       for j in range(1, n):
          P[i,j]=P[i-1,j-1]+P[i-1,j]
   return P
```

Exercice 11 (Puissances réduites par un polynôme annulateur) **:**

On considère la matrice :

$$C = \begin{pmatrix} 0 & 0 & -2 \\ 1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

1. Définir la matrice C en Python.
2. Vérifier avec Python que : $C^3 - 2C^2 - C + 2I_3 = 0$.
3. En déduire que, pour tout $n \geq 3$, $C^n = 2C^{n-1} + C^{n-2} - 2C^{n-3}$.
4. Écrire une fonction `puissance_C(n)` qui calcule C^n en utilisant cette relation de récurrence.
5. Tester cette fonction pour $n = 5$, $n = 10$, puis comparer avec la commande `al.matrix_power`.
6. Déduire une expression de C^{-1} en fonction de I_3, C et C^2.

Exercice 12 (Matrice involutive) **:**

On considère la matrice : $S = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$.

1. Définir la matrice S en Python.
2. Calculer S^2. Que peut-on dire de S ?
3. On pose : $A = 2I_3 + S$.
4. Calculer A^n pour $n = 1, 2, 3, 4, 5$.
5. On cherche à écrire : $A^n = \alpha_n I_3 + \beta_n S$.
   Montrer que les coefficients vérifient une relation de récurrence de la forme :

$$\begin{cases} \alpha_{n+1} = 2\alpha_n + \beta_n, \\ \beta_{n+1} = \alpha_n + 2\beta_n. \end{cases}$$

6. Écrire une fonction `coefficients(n)` qui renvoie α_n et β_n.
7. Écrire une fonction `puissance_A(n)` qui renvoie A^n sans utiliser `al.matrix_power`.
8. Déterminer A^{-1} sous la forme :

$$A^{-1} = xI_3 + yS.$$

   Puis vérifier avec Python.

TP 16

Graphes

1. L'essentiel

Ici nous n'allons pas apprendre de nouvelles commandes. Le but de ce TP est d'utiliser les outils déjà connus en Python pour représenter des graphes.

Il faut donc **modéliser**, choisir une méthode pour représenter un graphe avec des outils Python. Deux représentations sont proposées en ECG.

a) Avec une liste d'adjacence

Mémo 1 (Liste d'adjacence)

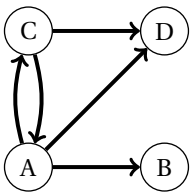
On considère un graphe d'ordre n. On peut implémenter le graphe par sa liste d'adjacence de la manière suivante :

```
G=[[liste_sommets_adjacents_sommet_1],...,[liste_sommets_adjacents_sommet_n]]
```

Mémo 2 (Matrice d'adjacence)

Pour rappel, la construction d'une matrice se fait avec la commande `np.array` et les matrices se rentrent ligne par ligne.
Vous pouvez aussi utiliser `np.zeros([n,p])`, `np.ones([n,p])` ou `np.eye(n,p)`

On considère le graphe orienté ci-contre.



Avec une liste d'adjacence on décrit ce graphe par
`G=[['B','C','D'],[],['A','D'],[]]`

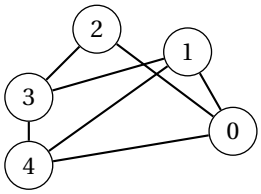
Avec une matrice d'adjacence, on décrit ce graphe par
`M = np.array([[0,1,1,1],[0,0,0,0],[1,0,0,1],[0,0,0,0]])`

2. Exercices

a) Définir un graphe

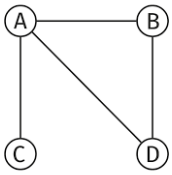
Exercice 1 : Représenter le graphe défini par la liste d'adjacence suivante :
`G=[[],['A','B'],['B','D','E'],['A','C'],['B','C','D']]`

Exercice 2 :

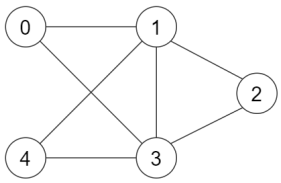


1. Implémenter ce graphe avec sa liste d'adjacence.
2. Implémenter ce graphe par sa matrice d'adjacence.

Exercice 3 : Reprendre les questions de l'exercice précédent avec le graphe suivant.



Exercice 4 : On considère le graphe G suivant.



1. Écrire la matrice d'adjacence M du graphe avec Python.
2. Calculer M^4. Quel est le nombre de chaînes de longueur 4 reliant les sommets 1 et 2 ?

b) Utiliser un graphe

Exercice 5 :
1. On considère dans cette partie que les graphes sont représentés par une liste d'adjacence.
   (a) Écrire une fonction Python qui prend en argument un graphe G et renvoie son nombre d'arêtes.
   (b) Écrire une fonction Python qui prend en argument un graphe G et renvoie une liste contenant le degré de chaque sommet.
2. Reprendre les questions précédentes pour un graphe représenté par une matrice d'adjacence.

Exercice 6 : Écrire une fonction Python `est_complet(M)` qui prend en entrée la matrice d'adjacence d'un graphe, et renvoie `True` si le graphe est complet et `False` sinon.

3. Approfondissement

Exercice 7 (Graphe eulérien) **:**

1. Écrire une fonction `est_connexe(M)` qui prend en entrée la matrice d'adjacence d'un graphe non orienté, vérifie qu'il l'est bien et renvoie `True` ou `False` selon s'il est connexe ou non.
2. Écrire une fonction `est_eulerien(M)` qui prend en entrée la matrice d'adjacence d'un graphe non orienté, vérifie qu'il est connexe et renvoie `True` ou `False` selon s'il est eulérien ou non.
3. On veut écrire la fonction `cycle(M)` qui prend en entrée la matrice d'adjacence d'un graphe non orienté et renvoie un cycle eulérien si le graphe est eulérien.

   Algorithme : Voici un algorithme qui construit un cycle Eulérien c d'un graphe eulérien :

- choisir un sommet s arbitraire, et former la chaîne $c = \{s\}$. (On peut par exemple prendre le plus petit sommet.)
- Tant que le dernier sommet de c possède une arête incidente a qui n'appartient pas à c, ajouter à la chaîne c l'arête a en ajoutant son sommet extrémité.
- si toutes les arêtes de G sont dans c alors l'algorithme est terminé et c est un cycle Eulérien.
- sinon il faut considérer le graphe G' constitué des arêtes qui ne sont pas dans c, choisir un sommet s' qui soit à la fois dans c et dans G' et construire dans G' un cycle c' d'origine s' avec le même algorithme. Enfin, il faut insérer c' dans c et retourner à l'étape précédente.

Remarque : Pour connaître les arêtes restantes, on enlèvera les 1 dans la matrice d'adjacence dès que l'arête sera utilisée.

Explications du programme

✔ La fonction `trouve_suivant(i,P)` prend en argument un sommet `i` et la matrice d'adjacence `P` (modifiée au fur et à mesure). Elle doit renvoyer le premier sommet j adjacent à i.

✔ La fonction `trouve_insertion(c,P)` prend en argument une chaîne `c` et la matrice d'adjacence `P` (modifiée au fur et à mesure). Elle cherche la position du premier sommet de la chaîne pour lequel certaines arêtes adjacentes n'ont pas encore été parcourues.

✔ La fonction `ajout_boucle(c,P)` prend en argument une chaîne `c` et la matrice d'adjacence `P` (modifiée au fur et à mesure). On commence par chercher la position du premier sommet de la chaîne pour lequel certaines arêtes adjacentes n'ont pas encore été parcourues puis on regarde sa valeur. Tant qu'on n'a pas créé de boucle (autrement dit, tant qu'on n'est pas revenu à ce sommet), on ajoute au fur et à mesure des arêtes adjacentes. On modifie enfin la chaîne que l'on avait en lui intercalant la boucle `b` que l'on vient de construire.

✔ La fonction `cycle(M)` prend en argument la matrice d'adjacence `M` du graphe. Elle renvoie un message d'erreur si le cycle n'est pas eulérien et un cycle s'il l'est.

```
import numpy as np

def trouve_suivant(i,P):
    n,n=np.shape(P)
    for j in range(n):
        if .........:
            return(j)

def trouve_insertion(c,P):
    for i in range(.....):
        if sum(......)>0:
            return(...)

def ajout_boucle(c,P):
    s.....................
    i=....
    j=-1
    b=[]
    while j!=c[s]:
        j=..................
        P[i,j]=...
        P[j,i]=...
        b.append(j)
        i=....
    c[s+1:s+1]=...    #pas besoin de return

def cycle(M):
    if est_eulérien(M).........:
        return(..................)
    else:
        P=np.copy(M) #pour modifier P sans modifier M
        c=[0]
        while ............:
            ajout_boucle(c,P)
        return(c)
```

TP 17

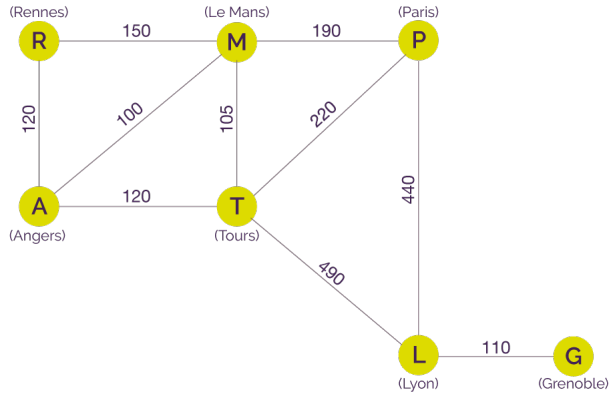
Graphes pondérés et algorithme de Dijkstra

On s'intéresse ici à un type de graphe, appelé *graphe pondéré*. L'idée est d'ajouter un *poids* sur chaque arête, une valeur qui lui est affectée. Cela peut par exemple représenter les distances entre différentes villes.
On se pose maintenant la question de chercher le chemin le plus **court** entre deux sommets donnés, donc celui qui a le poids le plus petit. Une méthode efficace est d'utiliser l'algorithme de Dijkstra. Celui-ci n'est pas simple à aborder, je vous propose un premier aperçu sur la vidéo suivante.
https://www.youtube.com/watch?v=XGmpZJOGSPc

1. Graphes pondérés et recherche du plus court chemin

On s'intéresse au réseau routier suivant, modélisé par un graphe. Les nombres sur chaque arête représentent les distances routières entre les villes.

On veut déterminer le plus court chemin pour aller de Rennes à Grenoble.



Mémo 1

Un **graphe pondéré** est un graphe sur lequel chaque arête est associée à un réel positif, appelé le **poids** de cette arête.

Les graphes pondérés peuvent être utilisés dans différents domaines :

— en probabilités : sur un arbre de probabilités, les poids représentent les probabilités

— en optimisation : les poids peuvent représenter des distances, des coûts ou des durées qu'on cherche à minimiser.

— autres utilisations : représenter un réseau de canalisations d'eau (les poids représentent les diamètres de tuyaux) ou une connexion internet

Mémo 2

On considère un graphe pondéré non orienté et connexe.

— La **longueur** d'un chemin est la somme des poids des arêtes qui le composent.

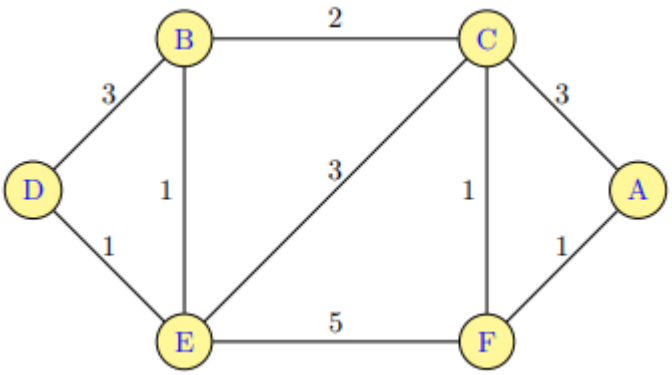
— Le **plus court chemin** entre deux sommets est le chemin de longueur minimale entre ces deux sommets.

2. Algorithme de Dijkstra

L'algorithme de Dijkstra (prononciation proche de *Dekstra*), du nom de son inventeur l'informaticien néerlandais Edsger Dijkstra, a été publié en 1959. Il permet de déterminer un plus court chemin entre deux sommets d'un graphe pondéré. Nous allons le présenter par un exemple.

Principe de l'algorithme pour trouver la distance minimale de D à A.
L'idée est de tenir à jour une liste des distances courantes des sommets du graphe par rapport au sommet initial. La distance courante est la distance minimale trouvée jusqu'à présent, qui pourra donc diminuer au fur et à mesure.

- Au départ, la distance minimale des sommets est par convention ∞. Celle du sommet de départ est mise à 0.
- À chaque étape :
  — On choisit le sommet s ayant la plus petite distance courante (parmi les sommets disponibles) ;
  — Pour chacun des voisins v de s, on calcule $d = \text{distance}_{\text{courante}}(s) + \text{poids}(s, v)$ (distance jusqu'à v en passant par s).
  Si $d < \text{distance}_{\text{courante}}(v)$, c'est qu'on a trouvé une meilleure distance courante pour v. On la modifie donc.
- On s'arrête quand on choisit le sommet final. Sa distance courante est la distance minimale cherchée.



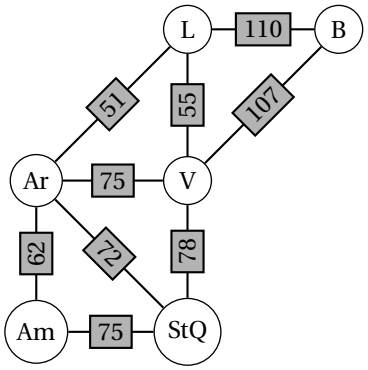
Conclusion : la distance minimale de D à A est

Remarque. Ce tableau permet de trouver la distance minimale, mais pas d'obtenir le chemin à parcourir... ce qui est d'un intérêt très limité en pratique !

Reprenons donc en indiquant à côté de la distance courante le *sommet père*, c'est-à-dire le sommet précédent qui a donné cette distance courante.

Conclusion : la distance minimale de D à A est
Le chemin optimal est...

Exercice 1 : Reprenez l'algorithme de Dijkstra sur le graphe suivant. On cherche les plus courts chemins partant d'Amiens.
Les villes représentées sont : Lille, Bruxelles, Arras, Valenciennes, Amiens et Saint-Quentin. Les distances sont données en kilomètres par la route.



3. Implémentation en Python

L'algorithme de Dijkstra consiste en la recherche des plus courts chemins menant d'un sommet s à chaque autre sommet d'un graphe pondéré G. Toutes les arêtes de G sont supposées de poids positif.

Pour tout sommet t, on notera $\delta(t)$ la longueur du plus court chemin menant de s à t.

a) Principe général

Avant de déterminer les plus courts chemins entre s et tout autre sommet v, on commence par déterminer la longueur de chacun de ces plus courts chemins i.e. la fonction δ.

Pour ce faire, on utilise un attribut $d[v]$ du sommet v qui contient le poids du supposé plus court chemin menant de s à v. Cet attribut est corrigé au fur et à mesure de l'algorithme de telle sorte à contenir $\delta(v)$ à la fin de l'exécution. Évidemment, cet attribut est, à chaque étape de l'algorithme, un majorant du poids d'un plus court chemin menant de s à v. Autrement dit :

$$\text{Pour tout sommet } v,\ \delta(v) \leq d[v]$$

Mise à jour de l'attribut $d[v]$

1. **Initialisation** : On commence avec $d(s) = 0$ et pour tout sommet $v \neq s$, $d(v) = +\infty$.

2. **Sélection d'un sommet** u : À chaque étape de l'algorithme, on sélectionne le sommet u qui vérifie les propriétés suivantes :
   - u n'a pas encore été sélectionné,
   - l'attribut $d[u]$ est le plus faible (parmi tous les sommets non encore sélectionnés).

   On parle alors d'algorithme glouton puisqu'on sélectionne à chaque étape la sous-solution optimale i.e. le sommet réalisant la meilleure distance.
3. **Mise à jour des distances** : On effectue un test permettant de savoir s'il est possible, en passant par u, d'améliorer le plus court chemin jusqu'à v. A-t-on
   $$d[u] + g_{u,v} < d[v] \text{ ?}$$
   - Si non, on ne fait rien.
   - Si oui, cela signifie que l'on a trouvé un chemin de plus petite taille pour aller de s à v en passant par le sommet u. On effectue la mise à jour
   $$d[v] = d[u] + g_{u,v}$$
4. **Fin de l'algorithme** Une fois tous les sommets visités, on a trouvé tous les plus courts chemins de s à tous les sommets v.

b) Application de l'algorithme

Voici une proposition pour implémenter cet algorithme en `Python`. On suppose ici que le programme prend deux éléments en paramètre :

- `G` est une matrice carrée : c'est la matrice d'adjacence du graphe pondéré. Ici un coefficient `G[i,j]` représente le poids de l'arête reliant le sommet `i` au sommet `j`. En l'absence d'arête le coefficient sera ∞, obtenu avec `np.inf`.
  On note donc que pour ce graphe pondéré, la matrice utilisée ne correspond pas à celle vue en cours, qui ne comporte que des 0 *et des* 1.
- `depart` est un entier : il représente le numéro du sommet de départ. On suppose que la numérotation des sommets commence à 0, ce qui correspond à la façon des lire les coefficients des matrices par `Python`.

```
import numpy as np

def dijkstraDist(G, depart):
    # On récupère le nombre de sommets N du graphe
    n,n=np.shape(G)

    # Initialisation du tableau des plus courts chemins avec des infinis
    # partout sauf pour depart
    # Le booléen est là pour savoir si le sommet a déjà été sélectionné
    pcc=[]
    for i in range(n):
        pcc.append([np.inf,False,None])
    pcc[depart]=[0,True,depart]

    # Initialisation du sommet u et de la distance au départ
    sommet_u=depart
    dist_u=0

    # On compte le nombre de sommets sélectionnés
    cpt=0
    while cpt!=n-1:
        # À chaque étape, la solution optimale doit être conservée
        # (pour sélection du sommet correspondant à l'étape suivante)
        minimum=np.inf

        # Mise à jour de toutes les distances
        for v in range(n):
            # Si le sommet v n'a pas encore été sélectionné
            if pcc[v][1] == False:
                # Test longueur jusqu'à v chemin en passant par u et màj distance
                if dist_u + G[sommet_u,v] < pcc[v][0] :
                    pcc[v][0]=dist_u + G[sommet_u,v]
                    pcc[v][2]=sommet_u
                # Mise à jour de la solution minimale à cette étape
                if pcc[v][0]<minimum :
                    minimum=pcc[v][0]
                    prochain_sommet_select=v
```

```
        # On a traité complètement un sommet
        cpt=cpt+1

        #Le sommet à traiter est sélectionné et d[u] est mis à jour
        sommet_u=prochain_sommet_select
        pcc[sommet_u][1]=True
        dist_u=pcc[sommet_u][0]

    return pcc
```

c) Exhiber les plus courts chemins

Maintenant que l'on a trouvé les distances des plus courts chemins de s à tous les autres sommets, il nous reste à reconstituer le chemin.

Ainsi, si à une étape, $d[v]$ a été modifié, il faut se rappeler de quel sommet u provient cette modification. L'idée étant alors que l'arête (u, v) sera une arête du plus court chemin.

```
Inf = np.inf
G = np.array([ [0,3,1,Inf,Inf,Inf], [3,0,1,2,Inf,Inf], [1,1,0,3,5,Inf], [Inf,2,3,0,1,3], [Inf,Inf,5,1,0,1], [Inf,Inf,Inf,3,1,0] ])

def dijkstraPCC(G, depart, arrivee):
    pcc = dijkstraDist(G, depart)
    # Reconstitution du plus court chemin
    chemin = [arrivee]
    # On reconstitue le plus court chemin d'arrivee vers depart
    # remis dans l'ordre
    ville = arrivee
    while ville != depart:
        ville = pcc[ville][2]
        chemin=[ville]+chemin
    return(chemin)
```

Cinquième partie

Statistiques

TP 18

Statistiques : manipulations de données

Nous allons voir dans ce TP comment manipuler des tableaux de données en Python. Les données seront issues d'un fichier au format CSV.

Les bibliothèques utiles pour ce TP sont `pandas` (pour la manipulation de statistiques) et `numpy` (pour les calculs).

```
import pandas as pd
import numpy as np
```

Les commandes de la librairie `pandas` ne sont pas exigibles (il ne faut pas les connaitre par cœur, mais il faut les comprendre pour pouvoir les interpréter).

1. L'essentiel

Mémo 1 (Le format CSV)

On utilise le format CSV (comma-separated values) pour stocker des données brutes de façon légère. Ce format est particulièrement efficace quand il faut stocker un nombre important d'informations sous forme de tableau (statistiques INSEE, données Parcoursup...)
Le principe est d'enregistrer dans un fichier les valeurs en les séparant par `;` ou `,`. Ces caractères seront interprétés comme des sauts de colonnes par Python, Excel, LibreOffice...

Fichier *notes.csv*

```
Maths;ESH;Anglais
12;17;14
8;12;9
18;15;13
9;11;17
```

Rendu dans un tableur

Maths	ESH	Anglais
12	17	14
8	12	9
18	15	13
9	11	17

Pour créer un fichier au format CSV :

- On saisit le texte directement dans un fichier : soit sur un bloc-note, soit en créant un nouveau fichier sur Edupython (pensez dans ce cas à supprimer la première ligne qui indique votre nom et la date de création).
- Dans ce fichier, on utilise un *séparateur* pour séparer chaque colonne, le plus souvent un caractère `,` ou `;`
- On enregistre le fichier sous la forme `fichier.csv` (il est indispensable de préciser l'extension csv pour qu'il soit bien lu). Ce fichier doit être placé dans le même dossier que vos documents Python.

Mémo 2 (Lire un fichier CSV en Python)

En Python on va importer le fichier et l'enregistrer dans une variable pour pouvoir le manipuler. On utilise la syntaxe suivante :

```
T = pd.read_csv('fichier.csv', sep=';')
```

— `'fichier.csv'` à adapter selon le nom choisi pour votre fichier

— `sep=','` permet de préciser le séparateur utilisé dans votre fichier (mettre `,` ou `;` selon le caractère choisi).

Mémo 3 (Manipuler un tableau de données)

On suppose qu'un tableau de données est stocké dans une variable `T`. On suppose que ses colonnes se nomment `colonne1`, `colonne2`, ...
Les librairies `pandas` et `numpy` permettent de faire les manipulations suivantes :

— `T.describe()` affiche un résumé de tous les indicateurs qui suivent.

— `T.count()` renvoie le nombre de valeur par colonne

— `T.head()` affiche les 5 premières lignes du tableau

— `np.shape(T)` renvoie le nombre de lignes et de colonnes du tableau.

— `T.mean()` affiche les moyennes de chaque colonne du tableau.

— `T.median()` affiche les médianes de chaque colonne du tableau.

— `T.var()` affiche les variances.

— `T.std()` affiche les écart-types

On peut aussi utiliser la librairie `numpy` pour les 4 dernières commandes avec `np.mean(T,0)` / `np.median(T,0)` / `np.var(T,0)` et `np.std(T,0)`.
Dans ce cas on donne le paramètre `0` pour que ces valeurs soient calculées colonne par colonne (sinon `numpy` calcule les indicateurs sur l'ensemble du tableau).

Remarque : La commande `std` de `pandas` n'indique pas l'écart-type que nous avons défini en cours mais l'écart-type *sans biais*.

En notant s_x l'écart-type, l'écart-type sans biais est $\overline{s}_x = \sqrt{\dfrac{n}{n-1}}s_x$.

Pour obtenir l'écart-type de la série (celui défini en cours), on utilise la commande `A.std(ddof=0)`. S'il y a 1 à la place du 0, c'est l'écart-type sans biais que l'on retrouve.

C'est le même « problème » pour la variance.

Quelle différence entre écart-type et écart-type sans biais?
En statistiques descriptives, lorsque les données étudiées forment **toute** la série, on utilise l?écart-type avec division par n en calculant $\sqrt{\dfrac{1}{n}\sum(x_i = \overline{x})^2}$.
Lorsque les données ne sont qu'un **échantillon** destiné à estimer une population plus grande, on utilise souvent une version corrigée, avec division par $n-1$, appelée écart-type sans biais en calculant $\sqrt{\dfrac{1}{n-1}\sum(x_i = \overline{x})^2}$. Cela permet de compenser le fait que sur un échantillon l'écart-type est sous-estimé.

2. Exercices

Exercice 1 : On s'intéresse aux taux de croissance, de chômage et d'inflation de 2016 à 2025.

Année	Croissance	Chômage	Inflation
2016	0,009	0,101	0,002
2017	0,021	0,094	0,010
2018	0,016	0,091	0,019
2019	0,020	0,085	0,011
2020	-0,074	0,081	0,005
2021	0,069	0,079	0,016
2022	0,027	0,073	0,052
2023	0,016	0,074	0,049
2024	0,015	0,074	0,020
2025	0,008	0,077	0,009

1. Enregistrer le tableau ci-dessus dans un fichier `donnees1.csv`.
2. Importer ce fichier sur Python dans une variable notée T.
3. Afficher les premières lignes du tableau T puis ses dimensions.
4. Tester la commande `describe` sur le tableau. Quels sont les quartiles pour le taux de chômage ?
5. Quel est le taux de chômage moyen au cours de ces années ?
6. Quel est l'écart-type de la croissance sur ces années ?

Exercice 2 : Sur cahier de prépa, télécharger un fichier CSV, au choix, dans le dossier « TP Statistiques ». Les fichiers proposés sont :

- `reservations_logement_touristique.csv` qui présente des données fictives de réservation d'un logement touristique.
- `Buteurs_Coupe_Du_Monde_2026.csv` qui présente les statistiques de plusieurs buteurs lors des matchs de la coupe du monde de football 2026.
- `indicateurs_pays_2007.csv` qui présente les indicatifs économiques des différents pays (données issues de Gapminder)

Téléchargez le fichier de votre choix. Prenez le temps de comprendre sa structure et les données qui y figurent. À vous ensuite de choisir quels indicateurs vous voulez déterminer sur cette série de données.

3. D'autres commandes utiles : Trier les données d'une série statistique

Mémo 4 (Commande `sort_values` de `pandas`)

`A.sort_values('NomCol',ascending=True)` donne une table où les données sont triées dans l'ordre croissant de la colonne `NomCol` (remplacer `True` par `False` pour l'ordre décroissant).
On note que cette commande ne modifie par directement le tableau, mais elle en produit un nouveau, qu'on peut simplement afficher, ou l'enregistrer, par exemple `B = A.sort_values('NomCol',ascending=True)`

Exercice 3 : Reprendre le tableau ci-dessus et trier les données par ordre croissant des taux d'inflation.

TP 19

Statistiques : représenter les données

Les bibliothèques utiles pour ce TP sont `pandas` (pour la manipulation de statistiques), `numpy` (pour les calculs) et `pyplot` pour les représentations graphiques.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Dans ce TP, nous nous intéresserons aux représentations graphiques des données statistiques. Avant de commencer, une commande permettant d'accéder à une colonne de données.

Mémo 1 (Extraire une colonne)

Si une série de données est stockée dans un tableau `A`, alors on peut en extraire une colonne en précisant son nom : `A['nom_colonne']`

Exercice 1 :

1. Réimporter le fichier de données `donnees1` du précédent TP dans Python.
2. Définir la variable `C` qui contient la liste des taux de chômage ; puis afficher sa moyenne et sa médiane.

1. Taux de chômage, inflation et croissance

Mémo 2 (Boîte à moustaches)

La commande `plt.boxplot(serie_de_donnees)` crée la boîte à moustaches des données. Pour obtenir l'affichage, on utilise toujours `plt.show()`.

Exercice 2 :

1. Utiliser la commande `pandas` `describe` pour l'étude des séries statistiques des taux de chômage, croissance et inflation.
2. Tracer la boîte à moustaches pour chacune des séries.
3. En vous aidant du TP représentation graphique des fonctions, tracer la courbe de l'évolution du taux de chômage, de croissance et d'inflation en fonction de l'année sur un même graphique.

2. Durée de fonctionnement d'un produit

Mémo 3 (Diagramme en bâtons)

La commande `plt.bar(x,n)` trace le diagramme en bâtons d'une série statistique où `x` contient les valeurs de la série et `n` les effectifs.

Exercice 3 : Une entreprise fabrique des produits. Elle effectue des tests sur un échantillon pour mesurer la durée de fonctionnement (arrondie en centaines d'heures). Les résultats sont donnés ci-dessous :

Durée	1	2	3	4	5	6
Effectifs	20	15	15	15	10	4

Tracer le diagramme en bâtons de la série.

3. Étude de la population par communes en 2023

Exercice 4 :

1. Taper le lien `https://www.insee.fr/fr/statistiques/4989724?sommaire=4989761` pour récupérer la population par commune au format csv et l'enregistrer dans un fichier nommé `Communes`.
2. A l'aide de la commande `describe`, étudier la série statistique.
3. Tracer sa boîte à moustaches.

Mémo 4 (Créer un histogramme)

Un histogramme est une représentation d'une série statistique avec en abscisse les données regroupées en intervalles et en ordonnées, l'effectif des données de cet intervalle.
La commande `plt.hist(donnees,bins=valeurs,density=True)` suivie de `plt.show()` affiche l'histogramme pour lequel `donnees` représente les valeurs de la série statistique et `valeurs` représente les bornes des classes de données qui nous intéressent. On pourra utiliser `np.linspace` ou `np.arange` pour créer les classes.
La commande `density=True` permet de normaliser les aires de telle sorte que la somme soit égale à 1 . Pour des classes de même taille, on retrouve les fréquences d'apparition.

Exercice 5 :

1. Tracer l'histogramme de la population en 2023, en répartissant les données en 100 classes.
2. Comme vous pouvez le constater, la majorité de la population réside dans des communes de moins de 10000 habitants. Retracer l'histogramme en ne vous intéressant qu'à ces communes.

Remarque. Avec la librairie Pandas, on peut ajouter une condition logique pour ne prendre en compte que les valeurs répondant à un critère donné. Par exemple, si on a chargé la liste des communes dans une variable `Com`, alors on peut utiliser la commande suivante pour extraire les lignes du département Somme

```
Somme = Com[Com['CODDEP']=='80']
```

Sixième partie

Calcul numérique

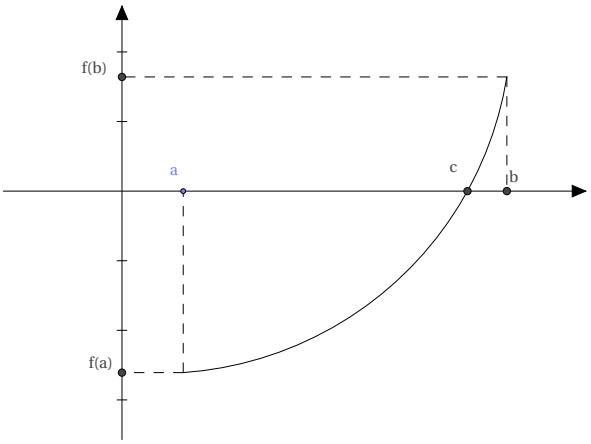
TP 20

Calcul approché d'une solution d'une équation $f(x) = 0$

1. Présentation du problème

On considère $f : \mathbb{R} \to \mathbb{R}$ une fonction. On souhaite résoudre l'équation : $f(x) = 0$. (E).

Lorsque f est une fonction assez « simple » telle qu'une fonction affine ou une fonction polynomiale de degré 2, la résolution de cette équation est possible : on connaît le nombre de solutions et leurs expressions.



Cependant, le plus souvent, il est impossible de résoudre une telle équation. En effet, le théorème des valeurs intermédiaires (ou le théorème de la bijection) donne l'existence (et l'unicité) d'une solution mais ne donne pas la ou les solution(s).

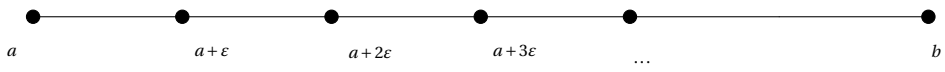
Toutefois, il existe des méthodes algorithmiques qui permettent d'approcher les solutions x de (E).

Dans tout ce TP, on travaillera dans le cas où la fonction f est continue et où l'équation $f(x) = 0$ admet une unique solution c dans un intervalle $[a; b]$ donné.

2. Méthode du balayage

On souhaite approcher la solution c de (E) avec une erreur d'au plus ε.

L'idée est de diviser le segment $[a; b]$ en sous-segments de largeur ε comme sur le graphe suivant :



Mémo 1 (Méthode balayage)

- ✔ Si $f(a)$ et $f(a+\varepsilon)$ sont de signe contraire, la solution $c \in [a, a+\varepsilon]$ d'après le TVI.
- ✔ Sinon, on continue en se déplaçant de ε : on compare alors les signes de $f(a+\varepsilon)$ et de $f(a+2\varepsilon)$ et ainsi de suite jusqu'à trouver deux termes de signes opposés.

Remarque : La méthode du balayage propose une approximation de la solution à ε près (longueur entre deux points consécutifs $a+k\varepsilon$ et $a+(k+1)\varepsilon$). Plus ε est petit et plus l'erreur commise est faible, mais plus le balayage risque d'être long ...

Exercice 1 : Soit

$$f(x) = x^2 - 2. \qquad (E_1)$$

On admet que f s'annule exactement une fois sur $[1,3]$

1. Définir la fonction f en Python.
2. Tracer la fonction f et la fonction nulle sur $[1,3]$ sur le même graphique.
3. Donner graphiquement une valeur approchée de la solution de l'équation.
4. Compléter la fonction qui permet d'approcher la solution de l'équation $f(x) = 0$ à ε près par la méthode du balayage.

```
def balayage(f,a,b,eps):
    s=a
    while ..... :
        if ..............:
            s=
        else :
            return ..........
    return ..................
```

5. Tester avec différentes valeurs de ε.

3. Méthode de la dichotomie

La méthode suivante est bien plus efficace que le balayage.

a) Description de la méthode

Mémo 2 (Méthode de la dichotomie)

La méthode est la suivante :

1. On divise l'intervalle $[a;b]$ en deux intervalles de mêmes tailles. Le milieu de cet intervalle est $m = \dfrac{a+b}{2}$. On obtient donc deux intervalles $[a;m]$ et $[m;b]$.
2. Si $f(a)f(m) < 0$ alors c est dans $[a;m]$ sinon c est dans l'intervalle $[m;b]$.
3. On garde l'intervalle contenant c, autrement dit :
   (a) si c est dans $[a;m]$, alors on effectue l'opération : $b \leftarrow m$.
   (b) sinon, on effectue l'opération $a \leftarrow m$.

On répète alors les étapes 1., 2. et 3. autant de fois que nécessaire pour se rapprocher de la solution. Le milieu du dernier intervalle est alors une approximation de la solution.

Remarque : Pour aller plus loin, on peut écrire un algorithme plus robuste en traitant le cas où $f(a)f(m) = 0$. Dans ce cas on a forcément $f(m) = 0$, donc on peut renvoyer directement la valeur m. Dans ce cas particulier, on a obtenu une solution exacte de l'équation.

Exercice 2 : Soit $g(x) = \ln(x) - 1$. On considère l'équation :

$$g(x) = 0. \qquad (E_2)$$

On admet que l'équation $g(x) = 0$ admet une unique solution dans $[1;4]$

1. Définir la fonction g en Python.
2. Tracer la fonction g et la fonction nulle sur $[1,4]$ sur un même graphe.
3. Donner graphiquement une valeur approchée de la solution de l'équation.
4. Compléter la fonction qui permet d'approcher la solution d'une équation $f(x) = 0$ par la méthode de dichotomie en connaissant le nombre de divisions n de l'intervalle $[a,b]$.

```
def dicho1(f,a,b,n):
    for k in range(1,...):
        m=.....
        if ............:
            ...=m
        else:
            ...=m
    return [a,b]
```

5. Tester le programme avec $n = 50$.
6. Compléter la fonction qui permet d'approcher la solution d'une équation $f(x) = 0$ sur $[a, b]$ par la méthode de dichotomie à ε près.

```
def dicho2(f,a,b,eps):
    while .......:
        m=.....
        if ............:
            ...=m
        else:
            ...=m
    return [a,b]
```

7. Tester le programme avec $\varepsilon = 0.001$.

Exercice 3 : Reprendre les questions de l'exercice précédent (à partir de la question 4) avec la fonction de l'exercice 1 : $f(x) = x^2 - 2$, sur $[1;3]$.

b) Approfondissement : erreur commise

A chaque étape, l'intervalle est divisé en deux et on sait que la solution c est dans cet intervalle. Si nous estimons la solution en prenant le milieu de l'intervalle, alors l'erreur commise est donc d'au plus la moitié de cet intervalle, autrement dit :

Étape :	Taille de l'intervalle	Erreur
0	$b-a$	$\frac{b-a}{2}$
1	$\frac{b-a}{2}$	$\frac{b-a}{2^2}$
2	$\frac{b-a}{2^2}$	$\frac{b-a}{2^3}$
3	$\frac{b-a}{2^3}$	$\frac{b-a}{2^4}$
$\vdots$	$\vdots$	
N	$\frac{b-a}{2^N}$	$\frac{b-a}{2^{N+1}}$
$\vdots$	$\vdots$	

A mesure que N grandit, il est clair que l'erreur commise tend vers 0. Par conséquent, si on veut approcher la solution c de (E) en commettant une erreur d'au plus ε, il suffit de répéter l'algorithme un certain nombre de fois. Nous savons qu'après N répétitions de l'algorithme, l'erreur commise est $\frac{b-a}{2^{N+1}}$. Fixons un $\varepsilon > 0$. Ce ε représente l'erreur maximale que l'on tolère. Au bout de combien d'itérations N aura-t-on une erreur inférieure ou égale à ε ? Pour le savoir, résolvons l'inéquation :

$$\begin{aligned}
\frac{b-a}{2^{N+1}} \leq \varepsilon &\iff 2^{N+1} \geq \frac{b-a}{\varepsilon} \\
&\iff (N+1)\ln 2 \geq \ln\left(\frac{b-a}{\varepsilon}\right) \\
&\iff N \geq \frac{\ln\left(\frac{b-a}{\varepsilon}\right)}{\ln 2} - 1.
\end{aligned}$$

Exercice 4 : On considère $a = 1$, $b = 5$. On veut une erreur d'au plus 0.001. Combien d'itérations faut-il pour obtenir au plus une telle erreur ?

Exercice 5 : On reprend la fonction de l'exercice 1 : $f(x) = x^2 - 2$, sur $[1;3]$. Combien d'itérations faut-il pour obtenir au plus une erreur de $0,001$?

4. Étude d'une suite dont les termes sont solutions d'une équation du type $f_n(x)$

On s'intéresse ici aux solutions de l'équation $\ln(x) = \dfrac{1}{x^n}$ pour $n \in \mathbb{N}$.

Afin d'observer l'existence de solutions, nous allons tracer les courbes des fonctions utilisées.

Exercice 6 :

1. Taper le programme suivant dans un éditeur de texte et l'exécuter :

```
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(1,4, 0.001)
plt.plot(x,np.log(x), label='ln(x)')
plt.plot(x,x**(-1), label='1/x')
plt.plot(x,x**(-2), label='1/x^2')
plt.plot(x,x**(-3), label='1/x^3')
plt.plot(x,x**(-20), label='1/x^20')
plt.plot(x,x**(-100), label='1/x^100')
plt.title('solution de l\'equation ln(x)=1/x**n')
plt.legend()
plt.show()
```

2. Conjecturer le nombre de solutions de l'équation $\ln(x) = \dfrac{1}{x^n}$ pour $n \in \mathbb{N}$.

   On appelle (α_n) la suite des solutions des équations $\ln(x) = \dfrac{1}{x^n}$.

   Quel semble être le sens de variation de la suite des solutions ? sa limite ?

Afin de chercher une valeur approchée de la solution de l'équation $\ln(x) = \dfrac{1}{x^n}$, on décide d'étudier $f_n : x \mapsto \ln(x) - \dfrac{1}{x^n}$ et de chercher la solution de l'équation $f_n(x) = 0$ pour tout $n \in \mathbb{N}$ fixé.

Exercice 7 :

1. Utiliser la fonction dichotomie, vue dans la section précédente, pour calculer un encadrement de la solution de $\ln(x) = \dfrac{1}{x^n}$ pour $n = 3$ avec une précision de $0,1$ puis de $0,01$, ... puis de 10^{-10}.

   On appliquera cette méthode sur $[1;\mathrm{e}]$ *(car l'étude mathématique montre que la solution appartient nécessairement à cet intervalle).*
2. Essayez différentes valeurs de n en suivant le même principe.

On souhaite vérifier les conjectures faites sur la suite des solutions.

Exercice 8 :

1. A l'aide d'une boucle for et de la fonction dichotomie, construire un tableau sol contenant les valeurs approchées de la solution x_n de l'équation $\ln(x) = \dfrac{1}{x^n}$ avec une précision de 0,00001 près pour $n \in [\![1;40]\!]$.
2. Dans la console, représenter graphiquement la suite (x_n) en fonction de n.
3. Cela rejoint-il la conjecture émise sur la monotonie de la suite des solutions ? et la conjecture sur sa convergence ?
4. La convergence semble-t-elle rapide ?

Exercice 9 (Pour ceux qui seront en avance) **:** En vous inspirant des exercices précédents, étudier la suite des solutions éventuelles de l'équation $f_n(x) = 0$, avec $f_n : x \mapsto x^n - nx + 1$.

Septième partie

Simuler le hasard

TP 21

L'aléatoire en Python

Dans de TP nous utiliserons la bibliothèque `random`. Elle s'importe avec la commande `import numpy.random as rd`

1. L'essentiel

Mémo 1 (Générer des nombres aléatoires)

Nous utiliserons les 2 commandes suivantes :

— `rd.random()` génère un nombre aléatoire entre 0 et 1 (plus précisément dans l'intervalle $[0;1[$)

— `rd.randint(a,b+1)` génère un entier entre a et b. Comme dans un `range`, on s'arrête avant la dernière valeur.

Mémo 2 (Simuler une expérience aléatoire dont la probabilité de succès est p)

On utilise le fait que
`rd.random()` est inférieur à p avec probabilité p.

Exemple : on lance une pièce truquée qui tombe sur Pile avec probabilité p. On simule cette expérience par le programme ci-contre.

```
alea = rd.random()
if alea < p :
    # succes
    print('Pile')
else :
    # echec
    print('Face')
```

Mémo 3 (Simuler une expérience aléatoire à n issues équiprobables)

On utilise la commande `rd.randint(1,n+1)` pour représenter les n issues possibles.
On peut avoir besoin de modéliser pour représenter chaque issue par un nombre entier

Exemple : une urne contient 10 boules noires, 5 boules blanches et 15 boules vertes. On souhaite simuler cette expérience par un programme.
On a 30 boules en tout : on modélise en les représentants par 30 entiers

→ les boules noires par les nombres de 1 à 10

→ les boules blanches par les nombres de 11 à 15

→ les boules vertes par les nombres de 16 à 30

```
alea = rd.randint(1,31)
# entier entre 1 et 30
if alea <= 10 :
    print('Noire')
elif alea <= 15 :
    print('Blanche')
else :
    print('Verte')
```

2. Exercices

Exercice 1 (Tirer à pile ou face) **:**
1. Avec la méthode de votre choix, écrire une fonction `Python` qui simule un lancer d'une pièce équilibrée. Elle renverra `Pile` ou `Face`.
2. Modifier cette fonction pour qu'elle suive les étapes suivantes :
   — elle demande au joueur de choisir entre Pile ou Face
   — elle simule le lancer et affiche la face obtenue
   — elle affiche « Gagné » si le joueur avait parié sur la bonne face, et « Perdu » sinon
3. Reprendre les questions ci-dessus dans le cas d'une pièce truquée qui tombe sur Pile avec une probabilité 0,7.

Exercice 2 (Choix d'étudiants) **:** Écrire une fonction en `Python` qui choisit aléatoirement un étudiant dans la classe ECG1 et annonce si l'étudiant choisi suit l'option Mathématiques appliquées ou Mathématiques approfondies.

Exercice 3 (Lancers de dés) **:** On veut simuler des lancers de dé, en utilisant un dé équilibré à 6 faces.
1. Écrire une fonction `Python` qui simule un lancer de dé. Elle renverra la valeur obtenue.
2. On effectue à présent n lancers. Modifier la fonction précédente pour qu'elle prenne le nombre n de lancers en paramètre. Elle renverra une liste formée des n nombres obtenus.

Exercice 4 (Tirages dans une urne) **:** On considère une urne qui contient 4 boules bleues et 9 boules rouges.
1. Écrire une fonction `Python` qui simule le tirage d'une boule dans cette urne. Elle renverra la réponse sous forme d'une chaîne de caractères (`Bleu` ou `Rouge`).
2. On effectue à présent n tirages successifs avec remise. Modifier la fonction précédente pour qu'elle prenne le nombre n de tirages en paramètre. Elle renverra une liste formée des n résultats du tirage.

Exercice 5 (Nombre de succès) **:**
1. On considère l'expérience suivante : on lance n fois une pièce équilibrée (avec $n \in \mathbb{N}^*$) et on veut compter le nombre de Pile obtenus.
   Écrire une fonction Python qui prend en paramètre un entier $n \geq 1$, simule cette expérience, et renvoie le nombre de succès.
2. Même question avec un lancer de dé. On comptera le nombre de 6 obtenus.

Exercice 6 (Interpréter) **:** Que fait le programme suivant ?

```
def simulPiece() :
    nb_pile = 0
    alea = 0
    while alea == 0 :
        alea = rd.randint(0,2)
        if alea == 0 :
            nb_pile = nb_pile + 1
    return nb_pile
```

3. D'autres commandes utiles

Mémo 4 (Générer une liste de nombres)

Les commandes `rd.random(n)` et `rd.randint(a,b+1,n)` génèrent un vecteur avec n nombres aléatoires (soit un réel entre 0 et 1, soit un entier entre `a` et `b`).
Cela permet de simuler une répétition de n expériences identiques.

Remarque (Aléatoire, vraiment ?) **:** À faire en même temps que votre voisin : saisir la commande `rd.seed(5)`, puis générer des nombres aléatoires. Que remarquez-vous ?

Vous pouvez recommencer en modifiant le paramètre saisi dans `rd.seed`.

4. Exercices d'approfondissement

Exercice 7 (Tirages de boules simultanés) **:**

1. Créer une fonction `Tirage1(n)` qui simule le tirage d'une boule dans une urne contenant n boules numérotées de 1 à n.
2. Créer une fonction `Tirage2(n)` qui simule le tirage sans remise de deux boules dans une urne contenant n boules numérotées de 1 à n, en considérant $n \geq 2$.
3. Créer une fonction `Tirage(n,p)` qui simule le tirage sans remise de p boules dans une urne contenant n boules numérotées de 1 à n, en considérant $n \geq p$

Exercice 8 (Rangs d'apparition) **:** On considère l'expérience suivante : on lance plusieurs fois un dé équilibré à 6 faces. On continue jusqu'à avoir obtenu 2 fois le nombre 5.

1. Écrire une fonction `Python` qui simule cette expérience. Elle renverra les numéros des 2 lancers ayant donné un 5.
2. Écrire une fonction `Python` qui répète 1000 fois cette expérience et renvoie la moyenne des rangs obtenus pour la seconde apparition du nombre 5.

TP 22

Variables aléatoires discrètes

1. L'essentiel

Mémo 1 (Lois usuelles)

Certaines lois usuelles sont définies dans le module `numpy.random`. On peut utiliser les commandes suivantes :

- `rd.randint(a,b+1)` pour une loi uniforme sur l'intervalle $[\![a,b]\!]$.
- `rd.binomial(n,p)` pour une loi binomiale de paramètre n et p.
- `rd.geometric(p)` pour une loi géométrique de paramètre p.
- `rd.poisson(x)` pour une loi de Poisson de paramètre x.

On peut par ailleurs ajouter un entier k en paramètre : dans ce cas la commande produit une « liste » de k réalisations de cette variable aléatoire (plus précisément, une variable de type `array`).
Par exemple, `rd.geometric(0.3,10)` donne une liste de 10 valeurs d'une loi géométrique.

```
>>> import numpy.random as rd
>>> rd.binomial(20,0.4)
11
>>> rd.binomial(20,0.4)
8
>>> rd.geometric(0.3)
1
>>> rd.geometric(0.3,10)
array([ 3,  2,  2,  1,  1, 10,  7,  5,  3,  2])
```

Si on veut simuler une variable aléatoire X qui ne suit pas une loi usuelle, on utilise les commandes `rd.random` ou `rd.randint` pour simuler l'expérience. Selon le résultat, on détermine la valeur de X correspondant à la situation étudiée.

Exemple : On considère le jeu suivant : on jette deux dés à 6 faces et on gagne une certaine somme selon le nombre de 6 obtenus :

- 0 € si aucun 6
- 5 € si un seul 6
- 10 € si un double 6.

On note X la variable aléatoire égale au gain du joueur.

```
import numpy.random as rd
```

```
def jeu() :
    alea = rd.randint(1,37)  # 36 résultats possibles sur les lancers
    if alea == 1  :
        # double 6
        return 10
    elif alea <= 12 :
        # 11 lancers avec un seul 6
        return 5
    else :
        return 0
```

Exemple (Écricome 2025) **:** Écrire une fonction `simulY(n, p)` simulant Y, avec $Y-1 \hookrightarrow \mathcal{B}(n-1,p)$, en utilisant uniquement `random` du module `numpy.random`.

```
import numpy.random as rd

def simulY(n,p) :
    #simulation de Z = Y-1
    # Z suit une loi binomiale (n-1,p)
    Z = 0
    for i in range(n-1) :
        alea = rd.random()
        if alea < p :
            # succes
            Z = Z+1
    # on a fini les experiences
    # il reste a determiner Y
    Y = Z+1
    return Y
```

Ici on simule la loi binomiale en simulant chacun des $n-1$ tirages (répétés dans la boucle `for`). Chacun a une proba de succès p, qu'on vérifie avec la commande `if alea < p`.
Si l'énoncé n'imposait pas d'utiliser uniquement `random`, on pourrait faire beaucoup plus rapide :

```
import numpy.random as rd

def simulY(n,p) :
    #simulation de Z = Y-1
    # Z suit une loi binomiale (n-1,p)
    Z = rd.binomial(n-1,p)
    Y = Z+1
    return Y

# Encore plus rapide :
def simulY(n,p) :
    return  rd.binomial(n-1,p) +1
```

2. Exercices

Exercice 1 : À traiter au choix dans le terminal ou dans l'éditeur.

1. Écrire un programme permettant de simuler une réalisation de loi binomiale de paramètre $n = 10$ et $p = 0.3$.
2. Écrire un programme permettant de simuler 1000 réalisations d'une loi géométrique de paramètre $p = 0.5$ puis les commandes permettant de calculer la moyenne et la variance de cet échantillon et les comparer à l'espérance et à la variance théorique.
3. Faire de même avec une loi de Poisson de paramètre $\lambda = 1$.

Exercice 2 (Somme de deux dés) **:** On lance deux dés équilibrés et on note S la variable aléatoire égale à la somme des deux nombres obtenus.

1. Écrire une fonction Python d'en-tête `def somme()` qui renvoie la valeur de S pour une réalisation de cette expérience.

2. Écrire une fonction Python d'en-tête `def simulation(n)` qui prend en paramètre un entier n, simule n réalisations de cette expérience et renvoie une liste des n valeurs obtenues.
3. Que fait le programme suivant ?

```
L = simulation(10000)
nb = L.count(7)
print(nb/10000)
```

4. Que fait le programme suivant ? On suppose la bibliothèque `numpy` déjà importée.

```
L = simulation(10000)
total = np.sum(L)
print(total/10000)
```

Exercice 3 (Premier six) **:** On lance un dé jusqu'à obtenir un 6.

On note X le nombre de lancers nécessaires pour obtenir le premier 6.

1. Écrire une fonction d'en-tête `def premier()` qui renvoie une réalisation de cette expérience.
2. Estimer $P(X \leq 3)$.
3. Estimer l'espérance de X.

Exercice 4 (Pile ou Face répétés) **:** On lance une pièce équilibrée 20 fois.

On note X le nombre de piles obtenus.

1. Simuler 10 000 réalisations de cette expérience (on donnera une liste contenant les 10000 résultats obtenus)
2. Estimer $P(X = 10)$.
3. Estimer $P(X \geq 15)$.

Exercice 5 (Nombre de mails reçus) **:** Une personne reçoit en moyenne 25 mails par jour.

On note X la variable aléatoire égale au nombre de mails reçus pour un jour donné. On suppose que X suit une loi de Poisson $\mathcal{P}(25)$.

1. Écrire un programme en Python qui calcule le nombre de mails reçus en un an. On commencera par créer une liste contenant 365 simulations de X, puis on donnera la somme de ces valeurs.
2. Estimer le nombre moyen de mails par jour obtenu sur cette simulation.

3. Approfondissement

Exercice 6 (Tirage sans remise) **:** Une urne contient 3 boules rouges et 7 boules bleues. On tire 5 boules sans remise.

On note X le nombre de boules rouges obtenues.

1. Simuler l'expérience avec une liste représentant l'urne.
   On peut par exemple définir la liste `urne = ["R", "R", "R", "B", "B", "B", "B", "B", "B", "B"]`. *On enlèvera de la liste, à chaque étape, la boule tirée.*
2. Estimer $P(X = 2)$.

Exercice 7 (Contrôle de réservation) **:** Une compagnie sait qu'en moyenne 5 % des passagers ne se présentent pas.
Un avion contient 180 places.
La compagnie vend 185 billets.

On note X le nombre de passagers qui se présentent (c'est-à-dire le nombre de passagers, sur 185, qui sont bien présents le jour du vol).

1. Modéliser X avec une loi binomiale.
2. Simuler 100 000 vols.

3. Estimer la probabilité que plus de 180 passagers se présentent.
4. Interpréter le risque de surbooking.

Exercice 8 (Tirages de boules) **:** Une urne contient :

— 3 boules rouges ;

— 2 boules bleues ;

— 1 boule verte.

On tire 2 boules successivement sans remise. Chaque partie coûte 5 €.

On obtient le gain suivant selon les boules obtenues :

— 10 € si les 2 boules sont de même couleur

— 2 € si les 2 boules sont de couleurs différentes.

On note X la variable aléatoire égale au gain algébrique obtenu (c'est-à-dire le gain réel, une fois déduite la somme misée).

1. Écrire une fonction en Python qui simule une réalisation de cette expérience.
2. Simuler 1000 réalisations et calculer le gain algébrique moyen obtenu. Ce jeu semble-t-il favorable au joueur ?

Exercice 9 (Représentation graphique - Loi uniforme) **:**

1. Écrire un programme pour simuler 100 fois une variable aléatoire $X \hookrightarrow \mathcal{U}(\llbracket -2,3 \rrbracket)$ et qui enregistre les résultats des simulations dans une liste.
2. Que fait le code suivant ?

Code

```
n=100
X=rd.randint(-2,4,n)
classes=np.arange(-2.5,4)
plt.hist(X,classes,rwidth=0.5,density=True)
plt.show()
```

3. Sur le graphique précédent, ajouter les fréquences théoriques obtenues.

Exercice 10 (Représentation graphique - Loi binomiale) **:**

1. Faire 1000 simulations de $X \hookrightarrow \mathcal{B}(5,1/2)$
2. Tracer l'histogramme correspondant à ces simulations.
3. Sur le graphique précédent, ajouter les fréquences théoriques obtenues.

Exercice 11 (Approximation d'une loi binomiale par une loi de Poisson) **:** Nous allons illustrer ici le fait que pour n assez grand on peut approximer une loi $\mathcal{B}\left(n, \dfrac{\lambda}{n}\right)$ par une loi de Poisson $\mathcal{P}(\lambda)$.

1. Créer un vecteur contenant 1000 réalisations de la loi $\mathcal{B}\left(100, \dfrac{5}{100}\right)$ et un autre vecteur contenant 1000 réalisations d'une loi $\mathcal{P}(5)$.
2. Représenter les deux histogrammes associés à ces simulations avec des couleurs différentes et des épaisseurs différentes pour les distinguer et commenter.

Huitième partie

Révisions

TP 23

Bilan du programme d'ECG1

Ce TP reprend tous les thèmes au programme cette année. N'oubliez pas à la lecture de l'énoncé de différencier si on vous demande de coder :

— une **fonction** Python : dans ce cas on commence par l'en-tête. Par exemple `def fonction(x) :` et on conclut (si besoin) par un `return`

— un **programme** Python : dans ce cas, les instructions peuvent être saisies dans le terminal directement. Il n'y a pas de paramètre : si on a besoin d'une valeur saisie par l'utilisateur c'est à l'aide de la commande `input`

Repérez les situations dans lesquelles une boucle est utile :

— si on sait à l'avance combien de répétitions : boucle `for`

— si on ne le sait pas : boucle `while`

— si un calcul direct suffit : pas de boucle !

Commencez par un calcul au papier pour voir si des répétitions sont nécessaires.

Gardez en tête que dans une succession de questions, on peut réutiliser les fonctions précédentes. Pensez à **recycler** les codes déjà écrits !

1. Les gammes

Exercice 1 (Fonctions) **:** On considère la fonction f définie sur $\mathbb{R}\setminus\{-1,1\}$ par $f(x)=\dfrac{e^x}{x^2-1}$

1. Écrire en `Python` une fonction qui prend en paramètre un réel x et renvoie la valeur $f(x)$.
2. Écrire un programme en `Python` qui demande à l'utilisateur de saisir un seuil `M` puis recherche le plus petit entier $x\geq 2$ tel que $f(x)>M$.

Exercice 2 (Suites) **:** Dans chaque cas, écrire une fonction en `Python` qui prend en paramètre un entier n et renvoie la valeur u_n, où la suite (u_n) est définie par

1. $u_n=\dfrac{2^n}{n+3}$
2. $u_0=5$ et $u_{n+1}=3u_n^2-2$
3. $u_0=1$, $u_1=2$ et $u_{n+2}=\sqrt{u_n u_{n+1}}$
4. $u_0=1$, $u_1=\dfrac{1+u_0^2}{1}$, $u_2=\dfrac{1+u_0^2+u_1^2}{2}$, …, $u_n=\dfrac{1+u_0^2+\cdots+u_{n-1}^2}{n}$ (question difficile !)

Variante : même principe, mais la fonction renvoie la liste des n premiers éléments de la suite.

Exercice 3 (Suites et moyennes) **:** En reprenant une suite de l'exercice précédent, écrire une fonction `moyenne(n)` qui prend en paramètre un entier n et renvoie la moyenne des n premiers termes de la suite (u_n)

Exercice 4 (Sommes) **:** Écrire une fonction qui prend en paramètre un entier n et renvoie la valeur d'une des sommes suivantes :

1. $\sum_{k=0}^{n} \sqrt{k}$
2. $\sum_{k=1}^{n} (3^k - 2^k)$
3. $\sum_{i=1}^{n} \left(\sum_{j=1}^{n} ij \right)$
4. $\sum_{i=1}^{n} \left(\sum_{j=1}^{i} ij \right)$

Exercice 5 (Représentations graphiques) **:** Représenter graphiquement la fonction f définie par $f(x) = \dfrac{e^x}{x^2+1}$ sur $[-5;5]$.

Exercice 6 (Suites et fonctions) **:** On définit la fonction $f : [0,1] \to \mathbb{R}$ définie par

$$f(x) = \begin{cases} 2x & \text{si } 0 \leq x \leq 1/2 \\ 2(1-x) & \text{sinon} \end{cases}$$

1. Écrire une fonction en `Python` qui prend en paramètre un réel x et renvoie la valeur $f(x)$. Pour aller plus loin, on pourra vérifier si $x \in D_f$, et renvoyer un message d'erreur si ce n'est pas le cas.
2. Tracer la courbe représentative de f.
3. On définit ensuite la suite (u_n) par :

   $$u_0 \in [0,1] \text{ et } \forall n \in \mathbb{N}, u_{n+1} = f(u_n)$$

   Écrire un programme qui prend comme arguments u_0 et n et renvoie les n premiers termes de (u_n), sous forme de liste.
4. (a) Tester le programme pour $u_0 = \dfrac{1}{2}$, $u_0 = \dfrac{1}{4}$, $u_0 = \dfrac{1}{8}$. Que se passe-t-il?

   Facultatif. Formuler une conjecture concernant le résultat pour $u_0 = \dfrac{1}{2^p}$, puis la prouver.

   (b) Tester ensuite $u_0 = \dfrac{2}{5}$, $u_0 = \dfrac{2}{7}$, $u_0 = \dfrac{2}{11}$. Que constate-t-on?

2. Algorithmique des listes

Exercice 7 (Recherches dans une liste) **:**
1. Écrire une fonction `existence(L,x)` qui prend en paramètre une liste `L` et un réel `x`, et qui renvoie `True` ou `False` selon que `x` appartienne ou non à la liste `L`.
2. Écrire une fonction `min(L)` qui prend en paramètre une liste `L` et renvoie sa plus petite valeur.
   Comment modifier ce programme pour qu'il renvoie la plus grande valeur de `L`?
3. Même principe, mais en renvoyant les deux plus petites valeurs. On peut pousser le travail en renvoyant un message d'erreur si tous les éléments de `L` sont identiques.

Exercice 8 (Moyenne) **:** Écrire une fonction `Moyenne(L)` qui prend en paramètre une liste L et renvoie la moyenne de ses éléments.

3. Statistiques

Exercice 9 (Manipulation de données) **:** On considère le fichier `notes.csv` ci-dessous.

```
Élève;Français;Mathématiques;SES
1;12;13;15
2;11;6;9
3;9;5;6
4;5;10;8
5;16;18;19
6;15;16;13
7;20;19;18
8;11;10;14
9;10;12;9
10;5;6;2
```

1. Charger ce fichier dans un tableau `T`
2. Afficher ses caractéristiques (moyennes, quartiles...)
3. Extraire les notes de mathématiques
4. Représenter par une boîte à moustaches les notes de mathématiques

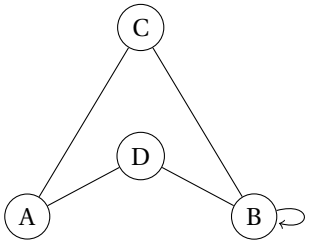
4. Approximation numérique

Exercice 10 (Recherche de racine) **:** On cherche à résoudre l'équation $x^3 - 3x + 1 = 0$

1. Expliquer pourquoi elle admet (au moins) une solution dans l'intervalle $[0, 1]$
2. Écrire une fonction qui prend en paramètre un réel x et renvoie la valeur $f(x) = x^3 - 3x + 1$.
3. On considère à présent un réel ε, que l'on notera `eps` en Python. On veut déterminer une valeur approchée à ε près de la solution de cette équation
   Écrire une fonction qui prend en paramètre `eps` et déterminer une valeur approchée de la solution :
   (a) par la méthode du balayage
   (b) par dichotomie

5. Graphes finis

Exercice 11 (Graphes et longueurs de chemins) **:** On considère le graphe suivant



1. Représenter ce graphe en `Python` (utiliser la méthode de votre choix)
2. Déterminer le nombre de chemins de longueur 6 reliant les sommets B et C.
3. Écrire une fonction Python qui prend en paramètre un entier n et renvoie la liste composée du nombre de chemins de longueur n allant du sommet B à chacun des autres sommets. Par exemple pour $n = 1$ on obtient la liste `[0,1,1,1]`

6. Simulation de phénomènes aléatoires

Exercice 12 (Simulations) **:** Écrire une fonction `Python` qui simule les expériences aléatoires suivantes. On pourra à chaque fois soit utiliser la fonction prédéfinie correspondant à la variable aléatoire donnée, soit utiliser uniquement les commandes `rd.random` ou `rd.randint`.

1. On pioche avec remise des boules numérotées de 1 à 20 jusqu'à obtenir la boule 17. On note X la variable aléatoire égale au numéro du tirage d'obtention de cette boule.
2. On pioche avec remise 20 boules dans une urne contenant 5 boules blanches, 10 boules rouges et 5 boules bleues. On note X le nombre de boules blanches obtenues.
3. Même question, dans le cas où l'urne contient n boules blanches (dans la fonction, ce nombre sera donné en paramètre).
4. On lance 10 fois un dé et on note X le numéro obtenu au dernier lancer.

Commandes à connaître

Voici un résumé des commandes qui figurent au programme d'ECG1.

1. Commandes générales

Opérations arithmétiques	
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`**`	exposant

Tests logiques	
`==`	est égal
`>`	supérieur
`<`	inférieur
`>=`	supérieur ou égal
`<=`	inférieur ou égal
`!=`	n'est pas égal

Opérateurs logiques	
`True`	Vrai
`False`	Faux
`and`	et
`or`	ou
`not`	négation

Structures de contrôle	
`=`	affectation d'une valeur à une variable
`if, elif, else`	instructions conditionnelles
`for`	boucle (nombre de répétitions connu)
`while`	boucle (nb de répétitions non connu)
`def fonction(n,p,...)`	définition d'une fonction (arguments entre parenthèses)
`return`	renvoie une variable ; arrêt de la fonction

Types	
`int`	les entiers
`float`	les nombres à virgule
`bool`	les booléens (Vrai / Faux)
`str`	les chaînes de caractères (string)
`np.array`	les objets matriciels

Import d'une bibliothèque
`from ... import *`
`import ... as ...`

2. Bibliothèque `matplotlib.pyplot`

Importation de la bibliothèque :
`import matplotlib.pyplot as plt`

Commandes pour tracer des graphiques / courbes	
`plt.plot(x,y)`	trace la courbe reliant les points dont les abscisses sont les éléments de `x` et les ordonnées ceux de `y`
`plt.show()`	afficher le graphique
`plt.hist(X,C)`	histogramme : `X` représente la série et `C` les extrémités des classes
`plt.bar(X,H)`	diagramme en bâtons : `X` représente la série et `H` les effectifs
`plt.boxplot(X)`	boîte à moustaches associée à la série `X`

3. Listes

Créer ou modifier une liste	
`L = [1,7,3,9]`	définit une liste en extension
`L = [i**2 for i in range(20)]`	Définir une liste en compréhension
`range(5)`	`[0,1,2,3,4]`
`range(2,5)`	`[2,3,4]`
`range(10,25,3)`	`[10,13,16,19,22]`
`L[5]`	Donne la valeur numéro 5 de la liste `L` (numérotation à partir de 0)
`L.append(5)`	ajoute 5 à la fin de la liste `L` (déjà définie)
`len(L)`	longueur de la liste `L`
`5 in L`	Teste si le nombre 5 apparaît dans la liste `L`. Renvoie `True` ou `False`
`del L[5]`	supprime l'élément numéro 5 dans la liste
`L.count(7)`	compte le nombre d'occurrences de la valeur 7 dans la liste `L`
`L1 + L2`	crée une liste qui contient les valeurs des listes `L1` puis `L2` (concaténation)
`L*5`	concatène 5 fois la liste `L`

4. Bibliothèque `numpy`

Créations de matrices	
`np.array([2,6,8])`	Crée le vecteur de composantes (2,6,8)
`np.array([[2,6,8],[1,2,3]])`	Crée la matrice dont les éléments sont donnés ligne par ligne
`np.zeros(5)`, `np.zeros([2,4])`	Crée un vecteur / une matrice de coordonnées nulles
`np.ones(5)`, `np.ones([2,4])`	Crée un vecteur / une matrice de coordonnées égales à 1.
`np.eye(4,5)`	Crée une « matrice identité »
`np.linspace(1,10,5)`	Vecteur composé de 5 valeurs entre 1 et 10 (inclus)
`np.arange(3, 15, 2)`	Vecteur des valeurs de 3 à 15 (exclu) avec un pas de 2
`np.shape(A)`	Taille (nb de lignes, nb colonnes) d'une matrice ou d'un tableau

Opérations sur les matrices	
`+,-,*, /,**`	opérations arithmétiques de base : coefficient par coefficient
`==,>,<, !=,>=,<=`	opérations logiques de base : coefficient par coefficient
`np.dot(A,B)`	produit matriciel
`np.transpose(A)`	transposée de la matrice

Opérations statistiques avec des matrices	
`np.sum(M)`	somme des éléments de `M`
`np.min(M)`	plus petit élément de `M`
`np.max(M)`	plus grand élément de `M`
`np.mean(M)`	moyenne des éléments de `M`
`np.cumsum(M)`	sommes cumulées de `M`
`np.median(M)`	médiane des éléments de `M`
`np.var(M)`	variance des éléments de `M`
`np.std(M)`	écart-type des éléments de `M`

Toutes ces opérations peuvent s'appliquer par ligne ou par colonne, par exemple `np.sum(M,0)` calcule la somme des éléments par colonne (pour le faire par ligne, on remplace `0` par `1`).

Fonctions mathématiques et nombres particuliers	
`np.exp(a)`	fonction exponentielle
`np.log(a)`	fonction logarithme népérien
`np.sqrt(a)`	fonction racine carrée
`np.abs(a)`	fonction valeur absolue
`np.floor(a)`	fonction partie entière
`np.e`	nombre e
`np.pi`	nombre π

Les fonctions ci-dessus peuvent s'appliquer à des nombres ou à des matrices : `np.exp(L)` renverra la matrice formée en calculant l'exponentielle coordonnée par coordonnée.
Ce principe s'applique si on calcule `f(L)` où `f` est une fonction Python écrite précédemment.

5. Bibliothèque `numpy.linalg`

Importation de la bibliothèque :
`import numpy.linalg as al`

Opérations sur les matrices	
`al.inv(M)`	Inverse de la matrice `M`
`al.matrix_rank(M)`	rang de la matrice `M` (*)
`al.matrix_power(M,n)`	puissance matricielle : renvoie M^n
`al.solve(A,b)`	renvoie la solution du système $AX = b$ (*)
`al.eig(M)`	renvoie les valeurs propres et vecteurs propres de la matrice `M` (*)

Les fonctions marquées d'un (*) sont au programme d'ECG2.

6. Bibliothèque `numpy.random`

Importation de la bibliothèque :
`import numpy.random as rd`

Générer des nombres aléatoires	
`rd.random()` `rd.random(5)`	renvoie un nombre aléatoire dans $[0,1[$ renvoie 5 nombres aléatoires dans $[0,1[$
`rd.randint(3,7)`	donne un entier aléatoire entre 3 et 6
`rd.binomial(6,0.2)`	loi binomiale de paramètre $(6;0{,}2)$
`rd.geometric(0.2)`	loi géométrique de paramètre 0,2
`rd.poisson(0.2)`	loi de Poisson de paramètre 0,2

Commandes à utiliser pour générer un nombre aléatoire, ou un vecteur / une matrice à coefficients aléatoires, par exemple

```
rd.binomial(10,0.2)
rd.binomial(10,0.2,100) # vecteur a 100 coordonnees
rd.binomial(10,0.2,[100,10]) # matrice 100 lignes et 10 colonnes
```

7. Bibliothèque `pandas`

Importation de la bibliothèque :
`import pandas as pd`

Les commandes de la librairie `pandas` ne sont pas exigibles (il ne faut pas les connaitre par cœur, mais il faut les comprendre pour pouvoir les interpréter).

Gérer un fichier de données	
`L = pd.read_csv("monfichier.csv")`	importe le fichier `monfichier.csv` et enregistre les données dans la variable `L`
`L.head()`	visualise les 5 premières lignes de la table `L`
`L.shape`	renvoie `n,p` le nombre de lignes et colonnes de la table
`L.describe()`	indicateurs statistiques de la table `L`
`L.mean()`	affiche la liste des moyennes
`L.std()`	affiche la liste des écarts-types
`L.median()`	affiche la liste des médianes
`L.count()`	nombre d'élément dans chaque colonne
`len(L)`	nombre de lignes dans le tableau `L`
`L.sort_values("col")`	affiche la table `L` classée selon les valeurs croissantes de la colonne `col`