MINISTÈRE DE L'ENSEIGNEMENT
SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

Université de Carthage

Institut Préparatoire aux Études
d'Ingénieur de Nabeul

INFORMATIQUE

Programmation et Simulation numérique

avec le langage Python

Travaux pratiques • 1ère ANNÉE MP-PT-PC

Manuel à l'usage des étudiants des classes préparatoires

Année 2022-2023

1ère Année MP/PT/PC

Anis SAIED

Professeur d'Informatique
IPEIN - Université de Carthage

```
def savoir():
print("Apprendre")
print("Comprendre")
print("Réussir")
```

Table des Matières

Introduction

Ce fascicule constitue le support officiel des Travaux Pratiques d'informatique pour les étudiants en première année des classes préparatoires (MP-PT-PC). Il a été conçu pour accompagner l'étudiant dans sa maîtrise de l'algorithmique et de la programmation Python, outils fondamentaux pour sa future carrière d'ingénieur.

Organisation de l'année

Le programme est structuré pour un apprentissage régulier : les étudiants vont travailler sur **un TP par semaine tout au long de l'année universitaire**. Ce rythme hebdomadaire de 22 séances permet une assimilation progressive, garantissant une transition fluide entre la logique algorithmique et les méthodes numériques.

Progression Pédagogique

Le contenu est organisé selon une difficulté croissante :

1. **Bases et Structures de Données (Semaines 1 à 8) :** Syntaxe, logique conditionnelle et types de données fondamentaux.
2. **Approfondissement (Semaines 9 à 14) :** Fonctions, récursivité, gestion des exceptions et complexité algorithmique.
3. **Calcul Scientifique (Semaines 15 à 22) :** Analyse numérique avec `NumPy`, visualisation de données avec `Matplotlib` et résolution d'équations différentielles.

Version Numérique

Dans une démarche d'accessibilité, la version numérique de ce fascicule, ainsi que des ressources complémentaires et les corrigés, sont disponibles en ligne sur :

`https://anis-saied.com`

Bon travail à tous pour cette année de formation.

TP N°1: À la découverte de Python

Lors de cette séance nous allons faire connaissance avec le langage Python. La version utilisée sera Python 3. Nous n'utiliserons pas directement Python ; nous passerons par un environnement de développement intégré (en anglais IDE : interactive development environnement) basique appelé IDLE.
Pour lancer l'« Interpréteur Python » : Menu démarrer , tapez IDLE.
Dans cet interpréteur, le « shell », vous pouvez directement taper apèrs le symbole >>> (le prompt) des opérations ou des instructions, qui seront effectuées immédiatement.

Exercice 1: Utilisation de la console comme calculatrice

1. À l'aide de la console, calculer : $54+23$ $24-7$ 123×87 $34/5$
2. À quoi correspondent les opérations //,% et $**$?
3. Peut-on indiquer plusieurs calculs différents sur une même ligne ? Si oui, quel(s) séparateur(s) mettre ?

Exercice 2: À la découverte des Variables

Une variable est destinée à stocker une valeur en mémoire, elle possède un nom permettant l'accès à cette place en mémoire. La valeur d'une variable peut changer au cours du temps. L'action de donner une valeur à une variable s'appelle « l'affectation », et s'effectue avec le signe = .

1. Créer une variable x de valeur 3. Peut-on écrire l'égalité d'affectation dans le sens qu'on veut ?
2. Créer une variable y égale à $7\times x$, et afficher sa valeur. Modifier la valeur de x. Quel est l'effet de cette modification sur la valeur de y ?
3. À l'aide de la fonction `help`, comprendre ce que fait la fonction `id`, et la tester sur la variable `x`. De même pour la fonction `type`.
4. Rajouter 1 à x. Quel est l'effet sur le type et l'identifiant ? Même question en rajoutant 0.5.
5. Comprendre l'effet sur x des opérations +=, -=, *=, /=.
6. Échanger le contenu de deux variables x et y.

Exercice 3: Comment importer un module

1. Essayer de calculer $sin(1)$.
   Certains outils sont disponibles dans des « modules » spécialisés. C'est le cas d'un certain nombre de fonctions mathématiques usuelles, disponibles dans le module `math`. Il y a 3 façons de faire appel à une fonction définie dans un module. Nous les étudions, de la moins coûteuse à la plus coûteuse.
2. Pour pouvoir faire appel à des fonctions d'un module par exemple `math`, on utilise l'instruction `import math`. On peut alors utiliser la fonction `sinus` par exemple en écrivant `math.sin(x)`.
   - Déterminer $sin(1)$.
   - Calculer $\sqrt{1+ln^2(\pi)}$. (utilisez `help` pour lister les fonctions disponibles dans le module `math`).
3. Si on fait appel un grand nombre de fois à une fonction précise, on peut l'importer, ce qui évite par la suite de devoir faire référence au module `math` à chaque utilisation. Cela se fait (pour le `sinus`) avec l'instruction `from math import sin`.
   - Calculer $sin(1)+sin(2)+sin(3)+sin(4)+sin(5)+sin(6)$.
   - Calculer $\sqrt{1+\sqrt{1\sqrt{1+\sqrt{1+\sqrt{2}}}}}$.
4. Enfin, si on fait appel à un grand nombre de fonctions du module, on peut les importer toutes, en écrivant `from math import *`.
   - Tester l'égalité $cos(2\pi)=cos^2(\pi)+sin^2(\pi)$.

Script Python: Les programmes écrits dans l'interpréteur d'IDLE ne peuvent pas être sauvegardés : ce n'est pas très pratique si vous souhaitez relancer une même série de calculs avec des valeurs différentes pour les variables. Il s'agit donc d'écrire dans un fichier une succession d'instructions qui ne seront effectuées que lorsque nous lancerons l'exécution du programme.
Pour écrire et sauvegarder des programmes, la procédure générale est la suivante :

1. Pour créer un nouveau programme (script) Python, utiliser le menu file / New File.
2. Enregistrer-sous votre script sous le dossier D:/votre_groupe/numéro_tp/numéro_exercice.py
   Exemple : D:/mp1/tp1/ex1.py
   Remarque : lorsque vous saisissez le nom du fichier «ex1.py» , n'oubliez pas de spécifier l'extension du fichier «.py», cela permet à l'éditeur IDLE de mieux visiualiser votre code source.
3. Pour lancer l'exécution de votre programme, utiliser le menu Run / Run Module (F5).

Exercice 4: Premier programme

Nous quittons maintenant la console, pour écrire notre premier programme.
Écrire un programme affichant *Bonjour*.

Exercice 5: La commande `print`

On souhaite rédiger un programme élémentaire qui calcule le prix d'une commande de livres. Les trois valeurs suivantes sont représentées par les variables :

nbr = entier désignant le nombre de livres commandés

prix = prix unitaire d'un livre

reduc = pourcentage de réduction (compris entre 0 et 100) dont bénéficie le client

Le programme devra afficher le montant de la facture.

1. Trouver une expression algébrique représentant le montant de la facture en fonction des variables `nbr`, `prix` et `reduc`.
2. Ouvrir un nouveau fichier que vous sauvegardez sous le nom `factureLivres.py`.
3. Affecter les variables comme indiqué ci-dessus dans le cas d'une commande de 27 livres dont le prix unitaire est de $30,50\,dt$ pour un client bénéficiant de 5% de réduction.
4. Afficher le résultat dans l'interpréteur. Pour afficher un résultat, on peut faire appel à la fonction `print`. Pour afficher une phrase, vous pouvez, par exemple, utiliser la syntaxe : `print('Le montant de la facture est de',m,'dt')`. Vous afficherez la valeur tronquée de m à deux chiffres après la virgule (utilisez `round`). Pour en savoir plus sur la commande round, tapez `help(round)`.
5. Ajouter les commentaires nécessaires pour mieux expliquer le rôle de votre programme.
   Remarque : Pour faire figurer des commentaires dans un script, on les rédige précédés d'un symbole #. Ils apparaissent ainsi dans le script sans être interprétés.

Exercice 6: La commande `input`

La commande input interrompt l'exécution des instructions et attend que l'utilisateur réalise une entrée. Elle récupère les entrées sous forme d'une chaîne de caractères.
Ci-dessous, la commande `input` récupère l'age de l'utilisateur sous la forme d'une chaîne de caractères:

```
age = input('Quel âge avez-vous ?')
type(age) → str
```

Les commandes `int` et `float` convertissent une chaine de caractères respectivement en un entier ou un flottant. Et inversement, la commande `str` convertit un nombre en une chaîne de caractères.

```
age = input('Quel âge avez-vous ?')
age = int(age)
type(age) → int
```

Reprendre l'exercice 5 en utilisant la fonction `input`. Votre programme interroge l'utilisateur au sujet du nombre de livres, puis lui affiche le montant de sa facture.

TP N°2: Les structures conditionnelles

Principe et syntaxe en python

Une `instruction conditionnelle` permet d'exécuter des instructions seulement sous une certaine condition. En python, on utilise les mots clefs `if, else et elif` qui consistent à tester si une affirmation est vraie ou non. L'instruction `if` permet d'exécuter un programme `P1` ou un autre `P2` selon qu'une condition `C` est vrai où non. La syntaxe est la suivante :

```
if C:
    P1
else:
    P2
```

Attention à bien indenter les instructions des programmes `P1` et `P2` . `Indenter` signifie décaler vers la droite de n (généralement 4) espaces. Ce n'est pas décoratif, mais cela a un sens pour python, car l'indentation permet de regrouper des instructions par `blocs`. Un `bloc` est un groupe d'instructions consécutives ayant la même indentation.

Exemple: Le programme suivant met le maximum de a et b dans la variable z et l'imprime à l'écran.

```
a = int(input("a = "))
b = int(input("b = "))
if a < b:
    z = b
else:
    z = a
print(z)
```

Rappel: l'instruction `r = input(s)` affiche la chaîne s, attend que l'utilisateur tape une chaîne de caractères au clavier terminée par la touche `Entrée`, puis met cette chaîne de caractères dans la variable r.

Remarque: On peut imbriquer les `if` (et donc ajouter les indentations des blocs), par exemple pour calculer le maximum m de a; b et c :

```
a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))
if a < b:
    if b < c :
        m = c
    else:
        m = b
else: # ici on est dans le cas a >= b
    if a < c:
        m = c
    else:
        m = a
print(m)
```

Remarque: En Python, la structures conditionnelle `if` possède une forme généralisée dont chaque `else: if condition:` est remplacée par `elif condition:` . Par exemple :

```
if a < b:
    if b < c :
        m = c
    else:
        m = b
elif a < c: #  ici on est dans le cas b <= a < c
    m = c
else: #  ici on est dans le cas b <= c <= a
    m = a
print(m)
```

Remarques

- Une `condition` est une expression logique de type `booléen` (dont son évaluation retourne soit `True` soit `False`). Tous les nombres différents de zéro et toutes les structures de données non vides sont évalués à `True`.
- Dans le cas où une condition est vérifiée et on ne souhaite rien faire, on utilisera le mot clé `pass`. Cette instruction est certes inutile mais sachez que vous êtes obligé de mettre au moins une instruction aprés une condition (après chaque utilisation de deux points :).
- Pour enchainer les tests, on peut remplacer `if a < b and b < c :` par `if a < b < c :`
- Il est possible que dans votre code vous ayez besoin de faire une condition dans une condition. Dans ce cas, il faut faire très attention aux indentations (décalage) : si on les modifient le retour du programme peut être différent.

Exercices

Pour écrire et sauvegarder des programmes, la procédure générale est la suivante :

1. Pour créer un nouveau programme (script) Python, utiliser le menu file / New File.
2. Enregistrer-sous votre script sous le dossier D:/votre_groupe/nom_prénom/num_tp/num_exercice.py
   Exemple : D:/M1/Ali BEN Ahmed/tp2/ex1.py
   Remarque : lorsque vous saisissez le nom du fichier «ex1.py» , n'oubliez pas de spécifier l'extension du fichier «.py», cela permet à l'éditeur IDLE de mieux visiualiser votre code source.
3. Pour lancer l'exécution de votre programme, utiliser le menu Run / Run Module (F5).

Attention Il n'est pas possible d'ouvrir un fichier dans IDLE en double-cliquant sur ce fichier depuis l'explorateur de fichier. Pour ouvrir un script, il faut toujours lancer IDLE et utiliser la commande « Ouvrir... » du menu Fichier.
Il faudra bien sûr enregistrer votre script très régulièrement pour ne pas tout perdre en cas de problème informatique.

Exercice 1

1. Créer un fichier intitulé `test_poids`. Définir deux variables de type flottants `taille` et `poids` affectées avec votre taille et votre poids.
2. L'indice de masse corporelle (abrégé IMC, égal à $\frac{poids}{taille^2}$) d'un individu donne une indication sur sa santé. Écrire un script qui calcule l'IMC que vous stockerez dans une variable `imc`, et indiquera par un message à l'écran si l'individu est en surpoids (IMC > 25) ou en sous-poids (IMC < 18).
3. Tester pour 3 individus : 1,80 m et 50 kg, puis 1,50 m et 100 kg, puis 1,75 m et 70 kg.

Exercice 2: Équation du second degré

On considère une équation du second degré de la forme $ax^2+bx+c=0$, dont les trois coefficients seront supposés être des entiers.

1. Écrire un script qui, en fonction des valeurs attribuées à a, b et c, indique automatiquement *le nombre de solutions* de l'équation ainsi que *leurs valeurs numériques.* On travaillera dans le corps des complexes, ce qui veut dire que si le discriminant est négatif on donnera des solutions complexes.
2. On initialisera les valeurs dans le script par l'affectation des trois coefficients. Par exemple :

```
a,b,c = 1,2,-3
[ suite du script ]
```

3. On testera le programme avec $(a,b,c) = (1,2,3)$ puis $(a,b,c) = (1,2,1)$ et enfin $(a,b,c) = (2,2,1)$.

TP N°3: Les structures itératives

Les *structures de contrôle itératives* ou les *boucles* permettent d'exéecuter plusieurs fois une ou plusieurs instructions. Python propose deux types de structures itératives : `for` et `while`.

1 La boucle `for ... in range(...):`

La boucle `for` est une boucle *inconditionnelle* sert à exécuter une ou plusieurs instructions un *nombre connu* de fois. En python, la syntaxe est la suivante :

```
for compteur in range(debut, fin, pas):
    bloc instructions
```

range retourne une séquence de nombres à partir de `debut` jusqu'à `fin` (exclu) : $[debut, fin[$

debut c'est la première valeur de l'intervalle crée par le fonction `range`, sert à initialiser la variable `compteur`. Par défaut : `debut = 0`

fin sert à donner une borne supérieure exclue de l'intervalle crée par le fonction `range` et dont la variable `compteur` ne doit pas y arriver.

pas C'est la valeur à ajouter à la variable `compteur` à chaque itération. Par défaut le `pas = 1`. On peut lui affecter une valeur négative, dans ce cas le parcours se fait dans le sens inverse, et par conséquant `debut` doit être supérieure à `fin`.

for sert à exécuter un bloc d'instructions un certain nombre de fois selon cette démarche :

1. Intialise la variable `compteur` par la valeur `debut`
2. Exécute son *corps* (le bloc d'instructions après le ":")
3. Incrémente la variable `compteur` par la valeur `pas`
4. Si la variable `compteur` est toujours strictement inférieur (ou strictement supérieur si le `pas` est négatif) à la valeur `fin` alors ré-itérer à partir du point 2, sinon la boucle s'arrête.

2 La boucle `while ... :`

La boucle `while` est une boucle *conditionnelle* sert à exécuter une ou plusieurs instructions un *nombre inconnu* de fois. La boucle `while` permet d'exécuter des instructions seulement sous une certaine condition. En Python ,la syntaxe est la suivante :

```
while condition:
    bloc instructions
```

condition Expression logique, aide la boucle `while` à décider si elle va exécuter son `corps` ou non.

while sert à exécuter un bloc d'instructions un certain nombre de fois selon cette démarche :

1. Evalue la `condition`
2. Si l'évaluation égale à `True`, alors elle exécute son *corps* (le bloc d'instructions après le ":") et ré-itérer à partir du point 1
3. Si l'évaluation égale à `False`, alors la boucle s'arrête.

3 Remarques

On peut utiliser les commandes suivantes avec les deux boucles `for` et `while` :

break On utilise l'instruction `break` lorsque l'on désire sortir d'une boucle itérative.
Lorsqu'on utilise `break` dans une boucle imbriquée, elle permet d'arrêter une seule boucle itérative à la fois (la boucle dont elle appartient à son corps).

continue On utilise l'instruction `continue` dans une boucle itérative lorsqu'on désire arrêter l'exécution de l'itération en cours et passer à l'itération suivante.

else On ajoute la clause `else` lorsqu'on désire exécuter un autre bloc d'inscturctions le cas où la boucle n'est pas interompue par `break`.

Exemple :

```
#Exemple 1
for i in  range (5):
    if i % 2 == 0:
        continue # ne pas afficher les nombres pairs
    print(i)

#Exemple 2
while True:
    n = eval(input("n = "))
    if n > 0:
        break # arrêter la boucle while
else: # ce bloc ne sera pas exécuté si la boucle while est arrêtée avec la commande break
    print(n)
```

4 Exercices

Exercice 1: La boucle `for`

1. Afficher tous les multiples de 7 inférieurs à 100.
2. Donner la somme des carrés de tous les entiers de 1 à 1000.
3. A l'aide d'une boucle `for` et de la commande `print(arg1,arg2,...)`, écrire un programme qui affiche sur 10 lignes la table des carrés des 10 premiers entiers non-nuls.
4. Écrire un programme qui calcule et affiche $\sum_{k=p}^{n^3} \frac{1}{k}$,où p et n sont deux entiers choisis par l'utilisateur.
5. On considère les deux suites récurrentes de *Mycielski* définies par : $m_1 = 2$; $m_k = 2m_{k-1} + 1$; $c_1 = 1$; $c_k = 3c_{k-1} + m_{k-1}$. Ecrire un script Python nommém *mycielski.py* , qui permet de saisir un entier n strictement positif , puis calcule et affiche le n^{eme} terme de la suite c_n

Exercice 2: La boucle `while`

1. Écrire un programme demandant un entier N, et qui affiche à l'écran tous les diviseurs de N. Combien 540 a-t-il de diviseurs ?
2. Écrire un programme demandant un entier naturel N (supérieur à 2) et qui affiche à l'écran son plus petit diviseur (supérieur à 2).

Exercice 3

On appelle *persistance* d'un nombre, le nombre d'itérations (répétitions) nécessaires pour le réduire à un seul chiffre, en multipliant tous ceux qui le composent et en recommençant avec le résultat obtenu.

Ainsi la persistance de 6788 est 6 car :
$6 \times 7 \times 8 \times 8 = 2688 \rightarrow 1^{ere} itration \mid 2 \times 6 \times 8 \times 8 = 768 \rightarrow 2^{eme} itration \mid 7 \times 6 \times 8 = 336 \rightarrow 3^{eme} itration$
$3 \times 3 \times 6 = 54 \rightarrow 4^{eme} itration \mid 5 \times 4 = 20 \rightarrow 5^{eme} itration \mid 2 \times 0 = 0 \rightarrow 6^{eme} itration$

Ecrire un script Python nommé *persistance.py* permettant de calculer et afficher la persistance d'un entier n strictement positif saisi au clavier.

TP N°4: Les listes

Une *liste* est une séquence ordonnée et modifiable d'objets homogènes ou hétérogènes séparés par des virgules et délimités par des crochets [ ]. Elle supporte la répétition et appartient au type `list`.

1. Les valeurs des listes peuvent être de types différents.
2. Les listes sont *indexées* à partir de 0. On accède à l'élément d'indice i d'une liste L par la syntaxe $L[i]$. Noter également que $L[-1]$ désigne le dernier élément de L, $L[-2]$ l'avant-dernier, etc.
3. Les listes sont *modifiables* : `L[i] = x` remplace l'élément d'indice i de L par x.

Création de listes

```
L1 = [] ; L2 = list() # listes vides
L3 = [1,2,3]
L4 = ["a" , 2.5] # types hétérogènes
L5 = [L3, L4] # liste de listes
L6 = list(range(0,1000,200)) # créer une liste à partir d'un intervalle
L7 = [for i in range(0,1000,200)] #Définition par compréhension
L8 = [i ** 2 for i in range(1, 11) if i % 2 == 0]#Définition par compréhension avec une condition
L9 = [[ i for i in range(11)] for j in  range(11)]#Définition par compréhension: liste de listes
```

Exercice 1

1. Créer la liste `liste = [2, 0, 5, [7, 6, 17], 4]`, remplacer le 0 par 8 puis remplacer le 17 par 22.
2. Construire par compréhension la liste des entiers multiples de 7 compris entre 1 et 100.

Fonctions et méthodes associées aux listes

```
L = [2, 0, 5, [7, 6, 17], 4] # liste initiale
L.pop() # permet d'enlever (et de récupérer) le dernier élément d'une liste
L.append(8) # permet d'ajouter un élément à la fin d'une liste.
L.insert(3,15) # Ajouter un élément à une position donnée dans une liste
len(L) # renvoie le nombre d'éléments d'une liste.
del (L) # Supprimer définitivement la liste
L1,L2 = [1, 2, 3] , [4, 5, 6]
L1.extend([1, 2, 3]) # Agrandir la liste L1 par la liste L2
L1 + L2 # concaténer deux listes
[1] * 5 # concaténer plusieurs copies d'une liste avec l'opérateur *.
L1 == L2 # Comparer le contenu de deux listes
2 in [1, 2, 3] # tester avec in et not in si un élément appartient ou non à une liste.
max(L), min(L), sum(L) # Retourner le maximum, le minimum et la somme des éléments d'une liste
L = [2, 1, 4, 3]
sorted(L) # renvoie une copie de L triée. L n'est pas modifiée
L.sort() # ne renvoie rien. L est triée.
```

Exercice 2

1. Construire, à partie d'une nombre entier `n`, la liste `L` composée de ses chiffres.
   Exemple : `n = 1234` → `L = [1,2,3,4]`
2. Ecrire une instruction qui modifie chacune des listes préremplies pour parvenir à la liste `L=[1,2,3,4,5]` (plusieurs réponses possibles):
   a) `L=[1,2,3]` b) `L=[-1,0,1,2,3,4,5]` c) `L=[1,2,3,3.5,4,5]` d) `L=[5,4,3,2,1]`
3. Ecrire un programme python qui permet de saisir une liste de 5 entiers strictement positifs. Afficher la liste triée en ordre descendant.

Itération sur les listes

```
for x in  L: # On peut itérer directement sur une liste  : x représente un élément de L
    print(x)

for i in range(len(L)): # i représente un indice variant de 0 à len(L) - 1
    print(L[i])
```

Exercice 3

1. Ecrire un programme python qui permet de saisir une liste de n nombres et renvoie le produit de ses éléments.
2. Ecrire un programme python qui à partir d'une liste d'entiers L, renvoie un liste de deux listes, la première sous-liste contenant les éléments pairs de L et la seconde ses éléments impairs.
   Exemple : `[1, 2, 4, 7, 3, 2, 1]` → `[ [2, 4, 2], [1, 7, 3, 1] ]`

Slicing

Le *slicing* (découpage en tranches en français) permet de récupérer ou de modifier un morceau d'une liste : `L[i:j]` représente la sous-liste de `L` formée des éléments dont les indices sont compris entre `i` et `j-1`. On peut spécifier un `pas` égal à `k` en écrivant `L[i:j:k]`.

```
L = [2, 8, 1, 3, 5]
L[2:4] # éléments d'indices 2 et 3
L[2:] # éléments d'indices 2 et plus (jusqu'à la fin de la liste)
L[:2] # éléments d'indices strictement inférieurs à 2
L[:] # toute la liste (copie de la liste)
L[1:5:2] # éléments d'indices 1 et 3
L[::-1] # toute la liste mais à l'envers
```

Exercice 4

1. Ecrire un programme python qui permet de saisir une liste de n nombres et renvoie le produit de ses éléments.
2. Ecrire un programme python qui à partir d'une liste d'entiers L, renvoie un liste de deux listes, la première sous-liste contenant les éléments pairs de L et la seconde ses éléments impairs.
   Exemple : `[1, 2, 4, 7, 3, 2, 1]` → `[ [2, 4, 2], [1, 7, 3, 1] ]`

Copier une liste

```
L1 = [1, 2, 3] # on crée en mémoire l'objet [1, 2, 3] et on lui associe le nom L1
L2 = L1; id(L1); id(L2) # on crée une copie de L1. L1 et L2 pointent vers le même objet.
L1[0] = 4; L1 ; L2 # mais si on modifie un élément de L1, L2 est modifiée aussi
L3 = L1[:] ; id(L3) # on crée un nouvel objet en mémoire. L3 pointe vers le nouvel objet créé.
L4 = [1,2,[3,4]]
L5 = L4[:] ; id(L4[2]); id(L5[2]) # copier la référence de la sous liste L4[2]
L5[0] = 5; L5[2][0] = 55; L5; L4
import copy
L6 = copy.copy(L4) # copie superficielle (shallow copy) : ne crée pas une copie de sous listes
L6[0] = 6; L6[2][0] = 66; L6; L4
L7 = copy.deepcopy(L4) # copie approfondie : crée une copie indépendante de sous listes
L7[0] = 7; L7[2][0] = 77; L7; L4
```

Exercice 5

Ecrire un programme python qui à partir d'une liste d'entiers `L` et deux objets `x1` et `x2`, renvoie une copie de `L` où on a remplacé toutes les occurrences de `x1` par `x2`.
Exemple : `x1 = 4; x2 = 1; L = [4, 5, 3, 4, 2, 1]` → `copie = [1, 5, 3, 1, 2, 1]`

TP N°5: Les tuples

Définition

Un tuple ou *n-uplet* en Python est une *séquence ordonnée* et *non modifiable* d'éléments éventuellement *hétérogènes* séparés par des virgules et entourés de parenthèses (Les parenthèses ne sont pas obligatoires).

- Le type *tuple* est un type de données similaire au type `list`. La seule différence avec les listes est que les tuples sont *non-modifiables* (on ne peut pas modifier un élément). Utiliser un tuple à la place d'une liste revient à avoir une assertion implicite que les données sont constantes.
- Le *tuple vide* (appelé aussi *0-uplet*) se note `( )`.
- Un tuple contenant 1 élément `x` se note `(x,)` et non `(x)` qui désigne juste `x`.
- Un tuple à deux éléments (couple) x et y se note `(x,y)` (ou `(x, y,)`).
- On peut convertir une liste en tuple avec la fonction `tuple()` ainsi on a :
  `tuple ( [ 4 2 , 17 , 7 3 ] ) == (42 , 17 , 73)`

Étant donné un tuple t, on peut, avec la même syntaxe que pour les listes :

- accéder à son k – $i^{\text{ème}}$ élément avec la syntaxe `t[k]` (comme pour les listes, la numérotation commence à partir de 0 comme pour les listes,
- désigner les éléments en comptant à partir de la fin en prenant des valeurs de k négatives),
- calculer son nombre d'éléments avec `len(t)`,
- et effectuer du `tranchage` (`slicing`).

Création

```
t1 = tuple() # créer un tuple vide
t2 = (1, "ok", "non") # créer un tuple non vide
t3 = tuple([1,2,3]) # créer un tuple à partir d'un itérable (liste)
t4 = tuple(range(0,1000,200)) # créer un tuple à partir d'un itérable (intervalle)
t5 = tuple(x*x for x in range(20) if x % 3 == 0) # créer un tuple en compréhension
t6 = tuple('aeiouy') # ('o', 'i', 'e', 'a', 'y', 'u')
t7 = 1, 2, 3 # Créer un tuple sans utiliser les parenthèses
t8 = 1, # n'oubliez pas d'y ajouter une virgule, sinon ce n'est pas un tuple.
```

Remarques :

- Les parenthèses ne sont pas obligatoires mais facilite la lisibilité du code (rappelons que la force de python est sa simplicité de lecture).
- Un tuple a une longueur fixée dès sa création (on ne peut pas lui ajouter un élément sans recréer le tuple)

Fonctions et méthodes associées aux tuples

```
t = (2, 0, 5, [7, 6, 17], 4) # tuple initiale
len(t) # renvoie le nombre d'éléments d'un tuple.
del(t) # Supprimer définitivement le tuple
t1,t2 = (1, 2, 3) , (4, 5, 6)
t1 + t2 # renvoie un tuple (nouvelle séquence) résulat  de concaténation de t1 et t2
t1 * 3 # concaténer plusieurs copies du tuple t1 avec l'opérateur *.
(1,) * 3 # utile pour l'initialisation
t1 == t2 # Comparer le contenu de deux tuples
2 in t1 # tester avec in et not in si un élément appartient ou non à un tuple.
max(t1), min(t1), sum(t1) # Retourner le maximum, le minimum et la somme des éléments d'un tuple
t = (2, 1, 4, 3)
L = sorted(t) # renvoie une liste triée dont les éléments sont ceux du tuple t.
t[1] = 3 # TypeError: 'tuple' object does not support item assignment
a,b = (1,2) # affectation multiple
```

Exercice 1

1. On donne le tuple suivant : `tup=(1,3,5,'Hilbert',(12,'ipt','génial'),4.6)`
   Écrire les lignes de code qui affiche :
   ```
   Hilbert
   le dernier élément du tuple
   (3,5)
   'ipt','génial'
   ```
2. Tapez les commandes Python permattant de lire un nombre réel, séparer les parties avant et après le point décimal et les afficher sous forme de tuple. Par exemple : pour 65.21 on doit afficher `(65,21)`

Slicing

Le *slicing* (découpage en tranches en français) permet de récupérer un morceau d'un tuple :
`t[i:j]` représente le sous-tuple de `t` formé des éléments dont les indices sont compris entre `i` et `j-1`.
On peut spécifier un pas égal à `k` en écrivant `t[i:j:k]`.

```
t = (2, 8, 1, 3, 5)
t[2:4] # éléments d'indices 2 et 3
t[2:] # éléments d'indices 2 et plus (jusqu'à la fin de la liste)
t[:2] # éléments d'indices strictement inférieurs à 2
t[:] # tous les éléments de t
t[1:5:2] # éléments d'indices 1 et 3
t[::-1] # tout le tuple mais à l'envers
```

Itération sur les tuples

```
for x in  t: # On peut itérer directement sur un tuple  : x représente un élément de t
    print(x)

for i in range(len(t)): # i représente un indice variant de 0 à len(t) - 1
    print(t[i])
```

Exercice 2

1. Construire, à partie d'une nombre entier `n`, le tuple `t1` composée de ses chiffres.
   Exemple : `n = 1234` $\rightarrow$ `t1 = (1,2,3,4)`
2. Écrire un programme python qui permet de saisir un tuple `t2` de 5 entiers strictement positifs. Afficher le tuple trié en ordre descendant.
3. Écrire un programme python qui permet de saisir un tuple `t3` de n nombres et affiche le produit de ses éléments.
4. Écrire un programme python qui à partir d'une tuple d'entiers `t4`, affiche un couple (2-uplet) de tuples, le premier sous-tuple contenant les éléments pairs de `t` et le second ses éléments impairs.
   Exemple : `t4 = (1, 2, 4, 7, 3, 2, 1)` $\rightarrow$ `( (2, 4, 2), (1, 7, 3, 1) )`
5. Écrire un script python qui, étant donné un tuple `t`, renvoie le nombre maximum de zéros consécutifs dans le tuple.
   Par exemple : pour `t=(1,0,2,0,0,4)` le programme renvoie 2.
6. Écrire un script python qui teste si deux tuples ont au moins un élément commun.
7. Écrire un script python qui, étant donné un tuple `t`, crée un tuple de tous les sous-tuples possibles.
   Par exemple :
   Les sous-tuples de `(1, 2, 3)` sont : `((), (1), (2), (3), (1, 2), (2, 3), (1, 2, 3))`.

TP N°6: Les chaînes de caractères

Une *chaîne de caractères* est une suite ou *séquence ordonnée* de caractères Unicode (lettres et symboles) entre simple ou double côtes *non modifiable*. Elle appartient au type `str`. Elles supportent le test d'appartenance, la concaténation, la répétition, l'accès par indice, par tranche, la taille,... mais pas de changement de ses élémentch. On peut convertir une chaîne en une liste et la modifier.

Création

```
ch1 = "Bonjour" # Créer une chaîne de caractères
ch2 = "".join(['v','é','l','o']) # Créer une chaîne de caractères 'vélo' à partir d'une liste
str(['v','é','l','o']) # retourne "['v', 'é', 'l', 'o']"
L = list ("vélo") # Convertir une chaine en une liste ['v','é','l','o']
ch3 = "Aujourd'hui" # chaîne contenant une apostrophe
ch4 = 'Aujourd\'hui'
ch5 = """Première ligne
Deuxième ligne"""  # utiliser """ ou  ''' : pour encadrer une chaîne définie sur plusieurs lignes
ch6 = 'Première ligne\nDeuxième ligne' # utiliser \n pour faire un retour à la ligne
ch7 = "a" * 3 # utiliser l'opérateur * pour répéter une chaîne
ch8 = "IP" + "EIN" # utiliser l'opérateur + pour concaténer une chaîne
```

Remarques :

- `ord('c')` : retourne le code ASCII de 'c'. Exemple : `ord('m')` → `109`
- `chr(code ASCII)` : retourne le caractère correspondant. Exemple : `chr(109)` → `'m'`
- On peut afficher une chaine dans un format prédéfini à l'aide de la fonction `format()`.
  Exemple : `"Bonjour M. {}".format('Ali')` → `"Bonjour M. Ali"`

Exercice 1:

1. Transformer la chaîne de caractère "3.14" en un flottant.
2. En utilisant l'écriture formatée, affichez la phrase «`Bonjour M. Ali`» en utilisant les variables `temps` et `prenom` dont les valeurs sont respectivement "jour", "Ali".

Fonctions et méthodes associées aux chaînes

```
ch = "Bonjour"  # chaîne initiale
ch.index('n')   # renvoie l'index de la première occurrence d'une sous-chaîne;
                # renvoie erreur si non trouvée.
ch.find("jour") # renvoie l'index de la première occurrence d'une sous-chaîne;
                # renvoie -1 si non trouvé.
ch.count('o')   # compte le nombre de sous-chaîne "o" dans la chaîne ch
ch.split('o')   # diviser la chaîne ch en une liste basée sur un délimiteur "o".
ch.replace("jour","soir") # Remplace une sous-chaîne par une autre
ch.upper()      # Changer la casse (mettre la chaîne en Majuscule)  : BONJOUR
ch.lower()      # Changer la casse (mettre la chaîne en miniscule)  : bonjour
ch.capitalize() # Mettre la première lettre en Majuscule            : Bonjour
ch.isalpha()    # Vérifier si la chaîne ne contient uniquement des caractères alphabétiques
ch.isdigit()    # Vérifier si la chaîne ne contient uniquement des caractères numériques
ch.isalnum()    # Vérifier si la chaîne ne contient uniquement des caractères alphanumériques
ch.isspace()    # Vérifier si la chaîne ne contient uniquement des caractères espaces
len(s)          # renvoie le nombre de caractères d'une chaîne
"b" in ch       # Tester avec in et not in si un caractère appartient ou non à une chaîne.
ch == "bonsoir" # Comparer le contenu de deux chaînes
```

Exercice 2

1. Écrire un script qui détermine si une chaîne contient ou non le caractère « e », puis s'il existe, compter son nombre d'occurrences et l'enlever de la chaîne de caractères.

2. Écrire un script qui détermine si une chaîne est palindrome ou non. On rappelle qu'une chaîne de caractères est dite palindrome, si elle se lit de la même manière dans les deux sens. Exemple: non, "12321", "elle".
3. Écrire un script qui détermine si une chaîne de caractères `ch` contient seulement des caractères chiffres.
4. Soit la chaîne de caractères `a='1.0 3.14 7 8.4 0.0'`. Écrire un script qui, à partir de `a`, et en utilisant `float()` et `split()`, construit la liste de réels `[1.0, 3.14, 7.0, 8.4, 0.0]`.

Itération sur les chaînes

```
for x in  S: # On peut itérer directement sur une chaîne  : x représente un élément de S
    print(x)

for i in range(len(S)): # i représente un indice variant de 0 à len(L) - 1
    print(S[i]) # TypeError: 'set' object does not support indexing
```

Exercice 3

1. Écrire un script qui génère une chaîne composée des chiffres de 0 à 9.
2. Écrire un script qui génère une chaîne composée des caractères alphabétiques de *a* à *z*.
3. Écrire un script qui recopie une chaîne (dans un nouvelle variable), en insérant des astérisques entre les caractères. Ainsi par exemple, « Docteur » devra devenir « d*o*c*t*e*u*r »
4. Écrire un script qui recopie une chaîne (dans une nouvelle variable) en l'inversant. Ainsi par exemple, « python » deviendra « nohtyp ».
5. En partant de l'exercice précédent, Écrire un script qui détermine si une chaîne de caractères est un palindrome (c'est-à-dire une chaîne qui peut se lire indifféremment dans les deux sens), comme par exemple « radar » ou « s.o.s »
6. Écrire un script Python qui génère une chaîne de longueur *n* choisie, constituée de caractères tirés au hasard parmi les lettres majuscules. Essayer de deviner cette chaîne.
   Vous pouvez utiliser le module Python `random`.

```
import random
random.randint(1,10)     # renvoie un nombre entier aléatoire entre 1 et 10
```

Exercice 4

1. Générer la liste de tous les *bigrammes* (chaine de deux lettres) possibles avec l'alphabet français. Votre script donnera ce résultat: `bigrammes = ["AA", "AB", "AC", "AD", ..., "BA", "BB", "BC", ..., "ZY", "ZZ"]`
2. Soit la séquence *nucléotidique* ch suivante: ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGC. Déterminer la lettre de l'alphabet la plus fréquente dans la chaîne de caractères ch.
3. Faites un programme qui répertorie dans une liste de tuples tous les mots de 2 lettres qui existent dans une séquence nucléotidique ainsi que leur nombre d'occurrences puis qui les affiche à l'écran. Votre script donnera ce résultat: `[("AC",12),("AB",41),("GA",10), ...]`.

Exercice 5

La *distance de Hamming* entre deux mots est une notion utilisée dans de nombreux domaines (télécommunications, traitement du signal, . . .). Elle est définie, pour deux mots de même longueur, comme le nombre de positions où les deux mots ont un caractère différent. Écrire un programme Python qui calcule la distance de Hamming entre deux mots lorsqu'ils ont la même longueur, et qui renvoie -1 sinon. Par exemple:
- Pour `ch1="aaba"` et `ch2="aaha"` la distance de Hamming = 1,
- Pour `ch1="stylo"` et `ch2="bouteille"` la distance de Hamming = -1.

TP N°7: Les ensembles

Un *ensemble* en Python est une collection modifiable d'objets *sans répétition* et *sans ordre* (donc sans numérotage). Il appartient au au type `set`. Il est créé par la séquence de ses éléments (valeurs), encadrées par { et }, ou en utilisant le constructeur `set` appliqué sur un itérable.

- Les valeurs des ensembles peuvent être de types différents.
- Les ensembles ne sont pas *indexées*.
- Les ensembles ne sont *mutables*, on peut ajouter et supprimer des éléments, mais on ne peut pas écrie s[i] = valeur, puisque un ensemble n'est pas une séquence.
- Les emplois basiques sont le test d'appartenance et l'élimination des entrée dupliquées.

Création

```
s1 = set() # ensemble vide
s2 = {"a" , 2.5 , 5} # créer un ensemble avec des valeurs de types hétérogènes
s3 = set([1,2,3]) # créer un ensemble à partir d'un itérable (ensemble)
s4 = set(range(0,1000,200)) # créer un ensemble à partir d'un itérable (intervalle)
s5 = {x*x for x in range(20) if x % 3 == 0} # créer un ensemble en compréhension
s6 = set('aeiouy') # {'o', 'i', 'e', 'a', 'y', 'u'}
```

Remarques :

- L'ensemble vide se note `set()` et non { } qui crée un *dictionnaire*

Exercice 1

Écrire un programme qui, à partir d'une chaîne de caractères `s`, crée un ensemble `lettres` formée des lettres utilisées dans la chaîne `s`.

Fonctions et méthodes associées aux ensembles

```
s = {2, 0, 5, 4} # ensemble initial
s.add(8) # Permet d'ajouter un élément à un ensemble.
s.remove(5) # Supprimer un élément donné d'un ensemble
s.pop() # Permet d'obtenir un élément (lequel ?) d'un ensemble et de le supprimer en même temps.
s.clear() # Vider un ensemble
len(s) # renvoie le nombre d'éléments (cardinal) d'un ensemble
2 in s # Tester avec in et not in si un élément appartient ou non à un ensemble.
s == {1,2} # Comparer le contenu de deux ensembles
max(s), min(s), sum(s) # Retourner le maximum, le minimum et la somme des éléments d'un ensemble
```

Exercice 2

1. Soit `L` une liste. Supprimez, à l'aide d'une seule instruction simple, tous les doublons de `L`. Calculer le nombre d'éléments distincts de `L`.
2. Reprendre la question précédente où cette fois `L` est une liste de chaînes de caractères ASCII (pas de lettres accentuées). On considère qu'une chaîne *« doublonne»* avec une autre si elle sont identiques sans tenir compte de la casse (par exemple 'Everest' et 'eVereSt' sont identiques). Supprimez tous les doublons de `L`.

Opérations ensembensembles

Les diverses opérations ensembensembles, définies dans le cours de mathématiques, sont réalisables avec des `set`. Supposons, par exemple, que l'on ait défini les deux ensembles `S1` et `S2` suivants :

```
>>> S1 = set([1, 2, 3])
>>> S2 = set([0, 1])
```

Union `S1.union(S2)` ou bien `S1 | S2`

Intersection `S1.intersection(S2)` ou bien `S1 & S2`

Différence `S1.difference(S2)` ou bien `S1 - S2`

Différence symétrique `S1.symmetric_difference(S2)` ou bien `S1 ^ S2`

Inclusion `S1.issubset(S2)` ou bien `S2.issuperset(S1)` : tester si S1 est inclus dans S2

Exercice 3

1. Définir deux ensembles (sets), `X = {'a', 'b', 'c', 'd'}` et `Y = {'s', 'b', 'd'}`
2. Affichez les résultats suivants:
   - Les ensembles initiaux ;
   - Le test d'appartenance de l'élément $'c'$ à X ;
   - Le test d'appartenance de l'élément $'a'$ à Y ;
   - Les ensembles $X - Y$ et $Y - X$;
   - L'ensemble $X \cup Y$ (union) ;
   - L'ensemble $X \cap Y$ (intersection).

Exercice 4

1. Écrire un programme python qui permet d'utiliser un ensemble pour vérifier si un mot contient des voyelles.
2. Soient `E` et `F` deux ensembles. Écrire un programme python qui calcul le produit cartésien de `E` et `F`, noté $E \times F$, formé de l'ensemble des couples (x,y) où $x \in E$ et $y \in F$.

Itération sur les ensembles

```
for x in  S: # On peut itérer directement sur une ensemble  : x représente un élément de S
    print(x)

for i in range(len(S)): # i varie de 0 à len(S) - 1
    print(S[i]) # TypeError: 'set' object does not support indexing
```

Remarque : Un ensemble n'est pas une séquence. On peut pas accéder à ses éléments par indices.

Exercice 5

1. Écrire un programme python qui permet de saisir un ensemble de n nombres et affiche le produit de ses éléments.
2. Écrire un programme python qui permet d'énumérer un ensemble de n nombres et crée la liste de sous ensembles énumérés.
3. Calculer le nombre de listes à k éléments dans d'un ensemble à n éléments.
   Rappelons que les listes sont ordonnées : `[1,2,3]` $\neq$ `[1,3,2]`
4. Écrire un programme python qui à partir d'une liste d'entiers `L`, crée une liste de deux ensembles, le premier sous-ensemble contenant les éléments pairs de L et le second ses éléments impairs.
   Exemple : `[1, 2, 4, 7, 3, 2, 1]` $\rightarrow$ `[ {4, 2}, {7, 3, 1} ]`

TP N°8: Les dictionnaires

Un *dictionnaire* en python est un type de données *non ordonné*, *mutable* et appartenant au type *dict*. Le dictionnaire est une sorte de liste mais au lieu d'utiliser des index, on utilise des *clés*, c'est à dire des valeurs autres que numériques.

Il permet de stocker des couples « **clé** : **valeur** ».

- Les **clés** pouvaient être des entiers, des chaînes et «quelques autres types non-mutables» , les tuples sont un de ces types. Toutes les clés doivent être distinctes.
- Les **valeurs** peuvent être de n'importe quel type Python. Toutes les valeurs possibles peuvent être stockés dans un dictionnaire.

Création

```
d1 = dict(); d2 = {} # dictionnaires vides
d3 = {"un":"one", "deux":"two", "trois":"three"}
d4 = {"p1": (1, 2.5) , "p2": (5.,7.8)} # créer un dictionnaire non vide
L=[(1, 2.5) , (5.,7.8)]
d5 = dict(L) # créer un dictionnaire à partir d'une liste de couples : d5 = {1: 2.5, 5.0: 7.8}
```

Remarques :

- Le dictionnaire vide se note `dict()` ou `{ }`.
- Les clés de dictionnaire doivent être non-mutables. Les tuples sont non-mutables mais si vous avez un tuple contenant des listes, il est considéré comme mutable et n'est pas utilisable comme clé de dictionnaire. Seuls les tuples de chaînes, de nombres ou d'autres tuples utilisable comme clé peuvent être utilisés comme clé de dictionnaire.

Fonctions et méthodes associées aux dictionnaires

```
d1 = {"un":"one", "deux":"two", "trois":"three"} # dictionnaire initial
d1["un"] ; d1.get("un")         # Récupérer la valeur de la clé "un"
d2 = {"etudiants" : [ { "nom" : "Ali" , "age" : 20} , {"nom" : "Ahmed" , "age" : 21 } ],
      "matieres" : { "Informatique" : {"coeff" : 4,  "nb_heures" : 2 } ,
                     "Français"     : {"coeff" : 4,  "nb_heures" : 2 } ,
                     "Analyse"      : {"coeff" : 10, "nb_heures" : 6 }
                   }
     }
d2["etudiants"][0]["nom"] # "Ali"
d2["matieres"]["Informatique"]["coeff"] # 4
d1["quatre"] = "four"      # ajouter/modifier(si la clé existe) le couple "quatre":"four"
d1.__setitem__("cinq","five")
d3 = {"six":"six", "sept":'seven'}
d1.update(d3)        # ajouter un dictionnaire d2 à un dictionnaire existant d.
d1.pop("trois")       # retourne la valeur "three" associée à la clé "trois"
                     # et supprime le couple "trois":"three" du dictionnaire.
del d1["quatre"]      # Supprimer le couple de clé "quatre"
len(d1)               # renvoie le nombre d'éléments d'un dictionnaire
"deux" in d1          # Tester avec in et not in si une clé appartient ou non à un dictionnaire.
d1 == d3              # Comparer le contenu de deux dictionnaires avec ==  et !=
d1.clear()            # Vider le dictionnaire d
```

Remarques :

- Si l'on demande la valeur associée à une clé inexistante, une exception `KeyError` est levée.
- Il est aussi possible de demander si la clé existe, avec l'opérateur `in` ou via la méthode `has_key()`. Exemple : `'onze' in d`, `d.has_key('onze')`

Exercice 1

1. Créer le dictionnaire d2 proposé précédemment.
2. Y ajouter un nouveau étudiant dont le nom est "Sami" et l'age est 19 ans.
3. Quelle est la taille du dictionnaire des matières.
4. Afficher les matières dont le coefficient égal à 10.

Itération sur les dictionnaires

Un dictionnaire étant un objet itérable, on peut itérer dessus. Par défaut, on itère sur les clés, mais on peut parcourir un dictionnaire avec les méthodes `.keys()`, `.values()` ou `.items()` qui renvoient une liste constituée des *clefs*, des *valeurs*, ou de *tuples (clef,valeur)*.

```
for k in d: # Par défaut, on itère sur les clés,  k représente une clé
    print("clé :", k, " --- valeur :", d[k])

for k in  d.keys(): # k représente une clé
    print(d[k])

for v in d.values(): # v représente une valeur
    print(v)

for k,v in d.items(): # items() : retourne des tuples (k = clé, v = valeur)
    print(k,v)
```

Exercice 2

Écrivez un programme python qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques.
Exemple : `{"un":"one", "deux":"two"}` → `{"one":"un", "two":"deux"}`

Exercice 3

Écrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, dans lequel vous mémoriserez les noms d'une série de copains, leurs âges et leurs tailles.
Votre script devra comporter deux parties : la première pour le remplissage du dictionnaire, et la seconde pour sa consultation.

- Dans la partie de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur. Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de type réel).
- La partie de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple (âge, taille) correspondant.
  Le résultat devra être une ligne de texte bien formatée, telle par exemple :
  `« Nom :  Ali ben Ahmed - âge :  15 ans - taille :  1.74 m ».`
  Pour obtenir ce résultat, utilisez la méthode `format()` pour le formatage des chaînes de caractères.

Exercice 4

1. Vous avez à votre disposition un texte quelconque. Écrivez un script qui analyse ce texte et enregistre dans un dictionnaire les occurrences de chacune des lettres de l'alphabet dans ce texte .
2. Modifiez le script ci-dessus afin qu'il mémorise dans un dictionnaire l'emplacement exact de chacun des mots (compté en nombre de caractères à partir du début).
   Lorsqu'un même mot apparaît plusieurs fois, tous ses emplacements doivent être mémorisés : chaque valeur de votre dictionnaire doit donc être une liste d'emplacements.

TP N°9: Les fonctions (1/2)

Une *fonction* est un bloc d'instructions qui possède un *nom*, reçoit un certain nombre de *paramètres* et renvoie un *résultat*. L'usage des fonctions permet de décomposer les tâches complexes en tâches simples plus abordables, d'éviter la répétition et de rendre le code réutilisable à travers l'importation de fichiers de fonctions.

Déclaration et syntaxe

Une fonction Python a la syntaxe suivante :

```
def nom_fonction([param1, param2,..]):
    ''' documentation '''
        corps de fonction
        [return valeur]
```

def : sert à déclarer un objet de type `function`.

Les paramètres : on ne précise jamais le type des paramètres (notion de *typage dynamique*).

La documentation : est optionnelle mais fortement recommandée (entre triple côtes/guillemets).

Le corps de la fonction : doit être limité par les *indentations*.

return : permet d'arrêter l'exécution d'une fonction et de renvoyer un résultat.

```
def somme (a,b) :
    ''' calculer la somme de a et b '''
        s = a + b
        return s
```

Une fois déclarée, la fonction peut être appelée par son nom suivi par ses paramètres

```
r = somme (1,2)
```

Remarques :

- Si on n'a pas encore désigné de quoi est chargée notre fonction, on peut écrire l'instruction `pass` au niveau du corps de la fonction, ce qui permet d'avoir un bloc d'instructions vide.
- Quand la fonction ne programme pas d'instructions de type `return`, Python retourne tout de même la valeur `None` (un objet vide) ayant comme type `NoneType`. Le mot clé `return` peut figurer dans plusieurs endroits au sein d'une fonction.

Exercice 1

Écrire les fonctions Python suivantes :

- `fact(n)` : calcul et retourne le factoriel d'un entier n passé en paramètre.
- `somme(L)` : calcul et retourne la somme des éléments d'une liste `L` passée en paramètre.
- `diviseurs(n)` : qui retourne la liste des diviseurs propres d'un entier n passée en paramètre (n n'est pas un diviseur propre de lui-même).
- `amis(n,m)` : teste si deux entiers n et m sont amis (deux entiers sont dits amis si chacun est égal à la somme des diviseurs propres de l'autre).

Passage de paramètres

Lors d'un appel à une fonction, Python fait une substitution (replacement) des paramètres formels par des paramètres effectifs. Python utilise deux modes de passage de paramètres:

1. Les types non mutables (int, float, str, complex, bool, tuple...) sont *passés par valeur*.
2. Les types mutables (listes, ensembles, dictionnaires ...) imitent le mode de *passage par variable*.

```python
# Exemple 1 : passage par valeur
def incrementer(n): # n : paramètre formel
    n = n + 1
k = 0
incrementer(k) # k : paramètre effectif
print(k) # k = 0

# Exemple 2 : passage par variable
def ajouter(L,x): # L : paramètre formel
    ''' ajouter à la fin d'une liste L un objet x '''
    L.append(x)
L = [1,2,3]
ajouter(L,4) # L : objet mutable, donc passé par variable
print(L) # L = [1,2,3,4]
```

Paramètres avec valeurs par défaut

Une fonction peut être déclarée avec des paramètres ayant une valeur de départ. Si l'utilisateur omet l'un des paramètres au moment de l'appel, la valeur par défaut sera prise en considération.

```python
def afficher_point(x=0,y=0):
    print('(x,y)=({},{})'.format(x,y))
afficher_point(); afficher_point(1,2); afficher_point(1); afficher_point(y=2)
```

Variables locales et globales

Les variables locales : Toutes les variables de la liste des paramètres d'une fonction, et toutes les variables créées dans une fonction sont des variables locales de la fonction. Les changements des variables locales faites lorsque la fonction est en cours d'exécution n'ont aucun effet sur toutes les variables en dehors de la fonction.
Les variables locales : Toutes les variables du programme principal, et toutes les variables créées dans une fonction avec le mot clé `global` sont des variables globales de la fonction. Toute variable globale est accessible et modifiable par la fonction où elle a été déclarée, par les autres fonctions qui la déclarent comme global et en dehors de toutes les fonctions.

Remarques:

- La *fonction imbriquée (interne)* est une fonction déclarée dans une autre fonction et ne peut être appelée que par la fonction englobante ou par des fonctions imbriquées dans la même fonction englobante.
- `if __name__ == '__main__'` : permet de séparer le corps du programme pricipal des définitions de fonctions, permettant une utilisation en module.

Exercice 2: Écrire en Python les fonctions suivantes

- `maximum(x,y,z)` : retourne le maximum de trois entiers passés en paramètres.
- `estpremier(n)`: vérifie si un entier n passé en paramètre est premier ou non.

Exercice 3

Deux nombres premiers sont *jumeaux* si leur différence est égale à 2. Par exemple, 5 et 7 sont jumeaux, 11 et 13 aussi. Écrire une fonction Python `jumeaux(n)` qui permet de trouver tous les couples de nombres premiers jumeaux compris entre 1 et n.

Exercice 4

La conjecture de Goldbach s'énonce ainsi : Tout nombre entier pair supérieur 3 peut s'écrire comme la somme de deux nombres premiers. Écrire une fonction Python `verif_Goldbach(n)` qui vérifie la conjecture de Goldbach pour tout entier naturel inférieur 1000.

TP N°9: Les fonctions (2/2)

Utilisations de fonctions

En Python, une fonction est une valeur comme une autre, c'est-à-dire qu'elle peut être *passée en argument* (1), *renvoyée comme résultat* (2) ou encore *stockée dans une variable* (3).

```
1 def g():
2     return 4
3 def f(fonction):
4     return fonction() + 1
5 print(f(g))
```

Listing 1: fonction passée en argument

```
1 def f():
2     def g():
3         return 4
4     return g
5 print(f()())
```

Listing 2: fonction renvoyée comme résultat

```
1 def g():
2     return 4
3
4 f = g()
5 print(f())
```

Listing 3: fonction stockée dans une variable

Remarques:

- On peut assigner une fonction à une variable comme tout autre objet python.
  Exemple: `afficher=print; afficher("bonjour")`

Exercice 1

Écrire les fonctions Python suivantes :

- `carre(n)` : calcule et retourne le carré d'un entier n passé en paramètre.
- `somme_fonction(f, n)` : calcule et retourne la somme $\sum_{i=0}^{n} f(i)$ pour une fonction f qui retourne le carré du paramètre n=10.

La fonction lambda()

Pour optimiser le code de l'exercice 1, on peut même éviter de nommer la fonction *carré(n)*, puisqu'elle est réduite à une simple expression, cela nécessiterait d'y incorporer un concept supplémentaire : les *expressions lambda*.
Une expression lambda est une *fonction anonyme* s'écrit: `lambda x: e` qui désigne la fonction $x \to e(x)$ où `e` est une expression pouvant comporter la variable `x`.
Dans L'exercice 1, on peut, sans définir la fonction `carré`, appeler la fonction `somme_fonction` plus simplement comme suit :

```
1 somme_fonction(lambda x : x**x , 10)
```

Remarques:

- En Python, une expression lambda est une fonction
  - anonyme (n'a pas de nom),
  - qu'on peut appliquer "à la volée" dans une expression.
  - et dont l'expression retournée ne doit pas contenir d'instructions composées (pas d'affectation, pas de boucle, etc.)
- Les fonctions lambda sont réservées à des situations relativement simples.

Exemple :

```
1 somme = lambda x,y : x + y # la somme de ses deux arguments
2 print(somme(3,4))
3 f = lambda  : 1/2  # fonction lambda sans arguments
4 print(f()) # 0.5
```

Exercice 2

Écrire les fonctions Python suivantes : ...

Les fonctions anonymes permettent d'appliquer deux commandes intéressantes sur les listes : *map* et *filter*.

La fonction map()

Une Exemple : ajouter 1 à tous les éléments d'une liste `L`.

```
f = lambda x : x + 1
L = [1,2,3]
L = list(map(f,L)) # L = [2, 3, 4]
```

Remarques:

- La fonction `map()` peut être appliquée à plus d'une liste. Les listes doivent avoir la même longueur. Exemple : `map(f, L1, L2, L3)`
- `map()` appliquera sa fonction lambda pour les éléments des listes d'arguments.
- `map()` applique d'abord la fonction `f` aux éléments avec l'indice 0, puis aux éléments d'indice 1 et ainsi de suite jusqu'à ce que le dernier indice est atteint.

Exemple :

```
L1 = [1,2,3]; L2 = [1,2,3]
L = list(map(lambda x,y : x + y, L1,L2)) # L = [2, 4, 6]
```

Exercice 3

Écrire les fonctions Python suivantes :

- `fact(n)` : calcul et retourne le factoriel d'un entier n passé en paramètre.

La fonction filter()

La fonction filter offre un moyen élégant de filtrer tous les éléments d'une séquence.
Syntaxe : `filter(fonction, sequence)`
La fonction `filter` a besoin de deux arguments :

Une fonction : comme premier argument, qui retourne une valeur booléenne, soit `True` soit `False`. Cette fonction sera appliquée à tous les éléments de la sequence (deuxième argument de `filter`).

Une sequence: c'est le deuxième argument de `filter`. Seulement les éléments inclus dans cette séquence où la fonction renvoie `True` seront produites par le résultat du `filter`.

Exemple : supprimer tous les multiples de 2 d'une liste d'entiers `L`.

```
f = lambda x : x % 2
L = list(range(20))
L = list(filter(f,L)) # L = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Remarques:

- La fonction `map()` peut être appliquée à plus d'une liste. Les listes doivent avoir la même longueur. Exemple : `map(f, L1, L2, L3)`

Exercice 4

- Filtrer d'abord les nombre impairs puis les nombres pairs des éléments de la séquence des n premiers nombres de Fibonacci.

TP N°10: Les exceptions

Les exceptions

Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution. Les erreurs détectées durant l'exécution sont appelées des *exceptions*. Lorsqu'une *erreur d'exécution* survient, Python lève une exception : il stoppe le programme et affiche le message d'erreur (la pile d'appels et le type d'*exception*). Exemple :

```
>>> 10 * (1/0)
ZeroDivisionError: integer division or modulo by zero
```

Les exceptions peuvent être de différents types. Les exceptions les plus courantes sont :

- Accéder à une clé non-existante d'un dictionnaire déclenche une exception `KeyError`.
- Un problème de conversion déclenche une exception `ValueError`.
- Appeler une méthode non-existante déclenche une exception `AttributeError`.
- Référencer une variable non-existante déclenche une exception `NameError`.
- Mélanger les types de données sans conversion déclenche une exception `TypeError`.

La liste complète des exceptions : https://docs.python.org/3/library/exceptions.htmlexception-hierarchy

Déclencher une exception

L'instruction `raise` permet au programmeur de déclencher une exception spécifique. L'instruction `raise` prend deux arguments : le type de l'exception et une chaîne de caractère qui décrit l'exception. Exemple :

```
age = input('Votre age : ')
if age < 0 :
    raise ValueError, "age invalide" # lever une exception de type ValueError
```

L'instruction `raise` est bien pratique lorsque vous souhaitez stopper l'exécution d'un programme si une variable ne contient pas la bonne valeur ou si une condition non vérifiée.

Gestion des exceptions

Il est possible d'écrire des programmes qui prennent en charge certaines exceptions. Gérer une exception permet d'intercepter une erreur pour éviter un arrêt du programme.

La bloc try-except

L'exemple suivant demande une saisie à l'utilisateur jusqu'à ce qu'un entier valide ait été entré.

```
while True:
    try:
        x = int(input("Saisir un entier: "))
        break # arrêt de la boucle while
    except ValueError:
        print("Erreur : Conversion en entier impossible !")
```

Remarques :

- Une instruction `try` peut comporter plusieurs clauses `except`, pour permettre la prise en charge de différentes exceptions. Mais un seul gestionnaire, au plus, sera exécuté. Exemple :

```
L = [1,2,3]
try:
    i = int(input(" i = "))
    x,y = L[i],L[i+1]
    print(x / y)
except ValueError:
```

```
        print ("Indice saisi non numérique")
    except IndexError:
        print ("Indice saisi dépasse la taille de la liste")
    except ZeroDivisionError:
        print ("Division par zéro!")
```

- Une même clause `except` peut citer plusieurs exceptions sous la forme d'un tuple entre parenthèses. Exemple :

```
except ValueError, IndexError, ZeroDivisionError :
    print ("Erreur")
```

- La dernière clause `except` peut omettre le(s) nom(s) d'exception(s), pour servir de joker. Elle peut aussi être utilisée pour afficher un message d'erreur avant de relever l'exception. Exemple :

```
except:
    print("Erreur inattendue")
    raise # relever l'exception : Autoriser python à gérer lui-même l'exception
```

Les clauses optionnelles pour l'instruction try

L'instruction `try - except` a également deux clauses optionnelles :

1. La clause `else` qui, lorsqu'elle est présente, doit suivre toutes les clauses `except`. Elle est utile pour le code qui doit être exécuté lorsqu'aucune exception n'a été levée par la clause `try`.
2. La clause `finally` est toujours exécutée avant de quitter l'instruction `try`, qu'une exception ait été déclenchée ou non.

Exemple:

```
try:
    result = x / y
except ZeroDivisionError:
    print ("Division par zéro!")
else:  # exécutée si aucune exception n'a été levée
    print ("Le résultat est = ", result)
finally:  # exécutée toujours, qu'une exception ait été déclenchée ou non
    print ("Exécution de la clause finally ")
```

Exercice 1

1. Créer une fonction Python `saisie()` qui saisie et retourne un entier strictement positif. Ajouter un bloc `try-except` qui gère une exception de type `ValueError`, ce qui pourrait se produire si l'utilisateur saisie une valeur non numérique. Imprimer un message d'erreur si cela se produit.
2. Créer une fonction Python `creer_liste(n)` qui crée et retourne une liste contenant les n premiers entiers premiers et demande de l'utilisateur un indice i pour afficher le i-ème nombre premier. Ajouter un bloc try-except qui gère une exception de type `IndexError`, ce qui pourrait se produire si l'index fourni dépasse la longueur de la liste. Imprimer un message d'erreur si cela se produit.
3. Écrire une fonction Python `ajouter_elt(d,L,x)` qui ajoute un élément x à une liste L dans un dictionnaire d des listes. Utiliser un bloc `try-except` qui gère une éventuelle exception de type `KeyError` possible si la liste avec le nom fourni n'existe pas encore dans le dictionnaire. Votre fonction doit inclure les clauses `else` et `finally` dans le bloc `try-except`.

TP N°11 : Les fonctions récursives

Définition

La *récursivité*, ce n'est pas autre chose que la récurrence mathématique transposée dans la programmation. Par exemple, le calcul d'un terme de la suite suite U définie par récurrence par

$$U_0 = 2, \forall n \in \mathbb{N}^*, U_n = \frac{1}{2}(U_{n-1} + \frac{3}{U_{n-1}})$$

se fera par une fonction récursive :

```
def u(n):
    if n == 0 :
        return 2
    else:
        res = (u(n-1) + 3/u(n-1)) / 2
        return res
for k in range (10) :
    print(u(k))
```

Remarques :

- La fonction u est récursive parce que dans sa définition même, elle est utilisée.
- On distinguera l'appel principal de u des appels récursifs :
  - L'*appel principal* se fait lorsque u n'est pas encore en cours d'exécution.
  - Les *appels récursifs* sont les appels provoqués par l'exécution de u.
- Une fonction récursive peut avoir un ou plusieurs *cas de bases* et un ou plusieurs *cas de propagation*

  Le cas de base C'est un cas particulier pour lequel la valeur à retourner ne nécessite pas d'appel à la fonction u. Sans sa présence, la fonction ne peut pas se terminer (garantie l'arrêt du programme).

  Le cas de propagation C'est lui qui contient l'appel récursif et dont l'un ou plusieurs des arguments qui lui sont transmises doivent converger et arriver à la condition d'arrêt ($n = 0$).

Exercice 1

```
def  f(a,b):
    if b == 1 :
        return a
    else:
        return a + f(a,b-1)
print(f(3,5))
```

1. Quel est le résultat affiché par ce programme ? Donner le schéma d'exécution de l'appel f(3,5).
2. Quel est le cas de base dans cette fonction récursive ?
3. Qu'est ce qui garantit dans les appels récursifs que le programme finira par s'arrêter ?
4. Que retourne `f(a,b)` (a et b étant des entiers naturels non nuls) ?

Remarques :

- A chaque nouvel appel récursif, des nouvelles variables sont crées qui portent les mêmes noms mais pas les mêmes valeurs
- L'ensemble des variables avec leurs valeurs de chaque appel récursif composent le *contexte* de l'appel récursif principal

Exercice 2

Définir les fonctions récursives suivantes :

1. `fact(n)` : la fonction factorielle, définie par $fact(0) = 1, \forall n \in \mathbb{N}^*, fact(n) = nfact(n-1)$
2. `somme(n)` : calcule la somme des n premiers entiers.
3. `syracuse(n)` : calcule et affiche les termes de la suite de Syracuse. Rappelons que selon la conjecture de Syracuse, toute itération de cette fonction à partir d'un entier naturel non-nul aboutit à la valeur 1.

$$C_n = \begin{cases} n/2 & \text{si n est pair} \\ 3n+1 & \text{sinon} \end{cases}$$

Exercice 3

1. Écrire une fonction récursive `somme_termes` qui prend en argument une liste. Elle renvoie une liste de même longueur dont la i-ème élément est la somme des i premiers éléments de la liste.
   Par exemple : `somme_termes([1,4,3,-1])` renvoie `[1,5,8,7]`
2. Écrire une fonction récursive `somme_rec` qui renvoie la somme d'une liste passée en argument.

Exercice 4

Écrire une fonction récursive permettant de calculer des termes de la suite U_n définie par :
$U_0 = U_1 = U_2 = 1$ et $\forall n \geq 3$, $U_{n+3} = U_n + U_{n+1} + U_{n+2}$

TP N°12 : Les méthodes de tri

Les méthodes de tri sont évidemment fondamentales. On dispose de nombreux algorithmes qui se distinguent par leurs complexités en temps et en place mémoire. Nous allons illustrer quelques-uns de ces algorithmes avec des listes (tableaux) d'entiers (ou de flottants).

En Python, on peut trier une liste à l'aide de la méthode *sort*. Si `a` est une liste d'entiers ou de flottants, `a.sort()` modifie la liste en la liste triée. Une fois triée, elle peut servir pour d'autres traitements comme :

— L'insertion d'une nouvelle donnée à sa bonne position en prenant en considération l'ordre établi ;

— Les algorithmes dichotomiques qui utilisent les listes triées pour accélérer la recherche ...

On distingue deux grandes catégories de tri :

1. Les *Tris lents* (tri par sélection, à bulles, par insertion...)
2. Les *Tris rapides* (tri par fusion, quicksort,...)

1 Un premier exemple de tri simple : le tri à bulles

Le principe de l'algorithme de *tri à bulles* est de faire « remonter » le plus grand élément à l'extrémité droite de la liste (comme une bulle dans un verre d'eau), puis de recommencer sur le reste de la liste.
L'algorithme parcourt la liste, et compare les éléments successifs. Lorsque deux éléments successifs ne sont pas dans le bon ordre, ils sont échangés. Après chaque parcours complet de la liste, l'algorithme recommence l'opération. On donne une version en pseudo-code de cet algorithme :

Algorithme 1 : tri_bulle(liste)

```
n ← taille(liste) ;
pour i=0 à n-2 faire
    i = n-i-1                          ▷ ainsi, i va varier de n − 1 à 1 ;
    pour j=0 à i-1 faire
        si liste[j] > liste[j+1] alors
            Échanger liste[ j ] et liste[ j+1 ] ;
        sinon
    fin
fin
```

Exercice 1

1. Implémenter le pseudo-code précédent en python et vérifier son fonctionnement.
2. Mesurer le temps d'exécution pour des listes d'éléments tirés de tailles comprises entre 100 et 5000, et variant par pas de 100. Récupérer les tailles et les temps d'exécution dans deux listes et afficher le graphe du temps d'exécution en fonction de la taille de la liste à trier.

2 Tri par sélection

Le *tri par sélection* est un algorithme de tri par comparaison. Le principe de cet algorithme est le suivant :

— Commencer à chercher l'indice du minimum de la liste `L` ;

— Permuter le 1er élément de la liste avec l'élément minimum trouvé ;

— Chercher l'indice du minimum des éléments de la sous liste `L[1 :]` et le permuter avec le 2ème élément ;

— Continuer ce principe jusqu'à ce que la liste devienne triée.

D'une manière générale, on échange l'élément à la position `i` avec le minimum de la sous liste `L[i+1 :]`.
Exemple : L=[4,5,1,-6,2]

- Etape N°1 : Le minimum de `L` est -6, on le permute avec 4 : L=[-6,5,1,4,2] ;
- Etape N°2 : Le minimum de [5,1,4,2] est 1, on le permute avec 5 : L=[-6,1, 5,4,2]
- Etape N°3 : Le minimum de [5,4,2] est 2, on le permute avec 5 : L=[-6,1,2,4,5]
- Etape N°4 : Le minimum de [4,5] est 4, il est à sa bonne place. L est bien triée.

Exercice 2

1. Créer une fonction `tri_selection(L)` qui trie la liste `L` selon le principe de tri par sélection.
2. Modifier la fonction `tri_selection(L)`, pour qu'elle accepte les deux sens de tri (croissant et décroissant).
3. Créer une version récursive de la fonction `tri_selection(L)`.

3 Tri par insertion

Dans le *tri par insertion*, on place chaque nouvel arrivant dans le tableau à sa position, dans une zone déjà triée selon le principe suivant :

— On considère une liste ou un tableau dont les premiers éléments d'indices $0,1,\cdots,i-1$ sont déjà triés ;

— On cherche à placer l'élément d'indice i à sa bonne place parmi les éléments déjà triés, pour cela on procède à le comparer aux précédents jusqu'à ce que la place de $T[i]$ soit trouvée.

Exercice 3

1. Écrire une fonction `position_insertion(L,i)` prenant en entrée une liste `L` et un indice `i` , tel que $i \in [1, len(L)]$ et retournant la position où on doit insérer `L[i]` dans `L[:i]` supposée déjà triée.
2. Créer une fonction `tri_insertion(L)` qui trie la liste `L` selon le principe de tri par insertion.
3. Quelle est la complexité de ce type de tri ?

4 Tri rapide (quicksort)

Le *tri rapide* est un algorithme de tri fondé sur la méthode de conception *diviser pour régner*. Pour trier une liste `L`, le principe de ce tri est le suivant :

— Si `L` est vide ou admet un seul élément, elle est triée (c'est notre critère d'arrêt) ;

— Sinon, On choisit un élément `e` $\in$ `L`, quelconque, que l'on retire de `L` ;

— On construit deux sous listes :
- `Li` formée des éléments inférieurs à `e` ;
- `Ls` formée des éléments plus grands que `e` ;

— Le résultat est la concaténation de `Li` récursivement triée, de `[e]`, et de `Ls` qui est aussi récursivement triée.

Exercice 4

1. Créer une fonction `tri_rapide(L)` qui trie la liste `L` selon le principe de tri rapide.
2. Quelle est la complexité de ce type de tri ?

Exercice 5

On veut trier une liste d'une manière décroissante en utilisant le principe de *tri par comptage*. On va supposer que la liste ne contient que des éléments distincts. L'algorithme compte pour chaque élément de la liste `L` le nombre d'éléments qui lui sont supérieurs. On utilise une deuxième liste `A` pour insérer les éléments de `L` chacun à sa bonne position comme le montre l'exemple suivant.
Soit `L` la liste à trier : `L=[4,3,6]`

— Pour 4 : il a 1 seul élément qui lui est supérieur ; il sera inséré à la position 1

— Pour 3 : il a 2 éléments qui lui sont supérieurs ; il sera inséré à la position 2

— Pour 6 : il a 0 élément qui lui est supérieur ; il sera inséré à la position 0

— La fonction retourne la valeur de la liste `A = [6,4,3]`.

1. Écrire une fonction `remplir_distinct(L,n)` qui remplit une liste avec n entiers tous distincts.
2. Écrire une fonction `compter(L,x)` qui compte le nombre d'éléments d'une liste `L` qui sont supérieurs strictement à un entier `x` passé en paramètre.
3. Écrire une fonction `tri_comptage(L)` qui trie une liste `L` en se basant sur le principe décrit en haut.

TP N°13 : Les méthodes de recherche

Les algorithmes de recherche sont très utiles en informatique. Leur rôle est de :

- Vérifier l'existence d'un ou de plusieurs éléments, satisfaisant une condition donnée, dans un tableau ;
- Chercher le nombre d'occurrences d'un élément ;
- Chercher la ou les positions d'un élément donné dans un tableau ; etc.

Problématique : On considère une liste `L` et un objet `e`. On souhaite tester l'appartenance de `e` à `L`.
Des primitives de Python réalisent ce travail (`e in L`), mais notre objectif est de comprendre comment on les réalise.

Il existe essentiellement deux stratégies de recherche : la *recherche séquentielle* ou encore *linéaire* et la *recherche dichotomique*.

1 Recherche séquentielle

La recherche séquentielle d'un élément dans un tableau consiste à parcourir le tableau séquentiellement jusqu'à trouver l'élément recherché ou atteindre la fin du tableau. La recherche peut se faire sur un tableau trié ou non, la différence réside dans le parcours du tableau et dans la condition d'arrêt de la recherche.

Principe :

1. Parcourir la liste.
2. Comparer chaque élément parcouru à l'élément recherché.
3. Si l'élément est trouvé, quitter la boucle et renvoyer `True` ou l'indice de l'élément (plus précisément de la 1ère occurrence).
4. Arrivé à la fin de la liste, quitter la boucle et renvoyer `False` ou `None` (aucune occurrence).

Implémentation en Python

```
1 def recherche_seq(L,x):
2     n =len(T)
3     for i in range(n):
4         if L[i] == x:
5             return i
6     return None
```

Complexité : le temps d'exécution de ce programme est proportionnel à la taille de la liste, on dit que la complexité de calcul est linéaire, on la note : $O(n)$ où n est la longueur de la liste.

Exercice 1

1. Écrire une fonction qui renvoie l'indice de la première occurrence de l'élément maximal d'une liste d'entiers. Évaluer la complexité de cette fonction.
2. Écrire une fonction qui renvoie les deux plus grands éléments d'une liste d'entiers. On supposera que la liste est de longueur au moins 2 ; en revanche, on veillera à ne la parcourir qu'une seule fois.

Exercice 2: Recherche d'un mot (une séquence) dans un texte (tableau)

Un problème classique en informatique consiste à rechercher, non pas une seule valeur, mais une séquence de valeurs dans un tableau. Cela revient à chercher une occurrence d'un tableau dans un autre ou, pour les chaînes de caractères, une occurrence d'un mot dans un texte.

1. Écrire une fonction `recherche_motif(m, t)` qui, étant donnés deux tableaux m et t, détermine la position de la première occurrence de m dans t, si elle existe, et qui renvoie `None` sinon.
   Exemple : pour les tableaux `m=[1,2,3]` et `t=[2,1,4,1,2,6,1,2,3,7]`, la fonction renvoie 6.
2. Modifier la fonction `recherche_motif(motif, texte)` pour qu'elle retourne toutes les occurrences d'une chaîne de caractères `motif` de longueur m dans une chaîne de caractères `texte` de longueur n. On supposera $m \leq n$.

Existe-t-il un algorithme plus efficace que la recherche naïve ?
OUI si la liste est triée. C'est la *recherche dichotomique*.

2 Recherche dichotomique

L'algorithme que nous proposons ici suppose que la liste dans laquelle on recherche un élément est **triée** en ordre croissant. Quand la liste (tableau) est triée, la recherche d'un élément dans une liste peut être réalisée de manière plus efficace en appliquant le principe *"diviser pour régner"*.

L'idée est que l'on compare l'élément cherché au terme du milieu de la liste, ce qui conduit, tant qu'on ne le trouve pas, à le rechercher soit dans la première partie soit dans la seconde partie de la liste.

Principe :

1. Parcourir la liste.
2. Comparer chaque élément parcouru à l'élément recherché.
3. Si l'élément est trouvé, quitter la boucle et renvoyer `True` ou l'indice de l'élément (plus précisément de la 1ère occurrence).
4. Arrivé à la fin de la liste, quitter la boucle et renvoyer `False` ou `None` (aucune occurrence).

Implémentation en Python

```
def rech_dicho(x,T):
    a, b = 0, len(T)-1
    while (a <= b):
        m = (a+b)//2
        if x > T[m]:
            a = m + 1
        elif x < T[m]:
            b = m - 1
        else : # ici, x == T[m], on retoune la position m
            return m
    return None
```

Complexité : La complexité de la recherche dichotomique est $O(log_2(n))$ où n est la taille du tableau.

Exercice 3

1. Proposer une fonction itérative `recherche_dicho(x,L)` qui renvoie le booléeen indiquant la présence (`True`) ou non (`False`) de x dans L par dichotomie.
2. Vérifiez votre code en cherchant 13, puis 2 et enfin 20 dans `[1,3,5,7,8,13,14,17,19]`.
3. Écrire une fonction `rechDicho1(t,n,x)` qui effectue une recherche dichotomique de x (entier) parmi les n premiers éléments d'un Tableau t trié et qui renvoie l'indice d'une occurrence (pas forcément la première) de x. La fonction renvoie -1 en cas de recherche infructueuse.
4. Exécuter votre fonction sur les données suivantes : `n=7`, `t=[1,7,8,9,12,15,15,22,30,31]` et `x=15`.
5. Écrire une fonction `rechDicho2(t,n,x)` version récursive de `rechDicho1(t,n,x)`.

TP N°14 : Les fichiers

Objectif : Apprendre à lire et écrire des fichiers texte via Python.
Avant de commencer : Créez un dossier *TP_14* dans « *D :\votre_classe\votre_nom_prenom* » dans lequel vous placerez tous vos scripts et les fichiers que vous créerez durant ce TP. Créer dans ce dossier le fichier texte nommé « test.txt » contenant quelques lignes de texte .

1 Définition

Un *ficher* est généralement le moyen utilisé pour sauvegarder les données de vos programmes pour les réutiliser plus tard. Python permet d'interagir avec des fichiers (écriture, lecture). On s'intéresse aujourd'hui aux fichiers contenant du texte.

2 Lecture de fichier

En programmation, la lecture d'un fichier se fait en trois temps :

L'ouverture du fichier : se réalise grâce à la fonction *open* qui a deux paramètres : le nom de fichier et le mode de traitement du fichier ("r" pour la lecture d'un fichier).

La lecture du contenu du fichier : L'intégralité d'un fichier peut être récupérée dans une unique chaîne de caractères grâce à la méthode *read*.

La fermeture du fichier : s'effectue grâce à la méthode *close*. Elle peut s'effectuer même si le fichier n'a pas été lu.

Exemple :

```
f = open("test.txt","r") # ouverture
texte = f.read() # récupération du texte dans une variable
print(texte) # affichage du texte récupéré
f.close() # fermeture
```

Remarques :

- La fonction `open` retourne un *descripteur* de fichiers qui est un objet particulier. Ce descripteur doit être récupéré lors de l'appel (par exemple dans une variable `f`) pour pouvoir être manipulé par la suite.
- Une seconde possibilité de lecture consiste à indiquer en paramètre de la fonction `read()` le nombre de caractères qu'on désire lire. Exemple : `ch = f.read(5)`
- Deux autres méthodes permettant de lire un fichier texte ligne par ligne : `readline()` et `readlines()`.

Exercice 1

1. Que renvoie la méthode `read()` lorsque la fin du fichier est atteinte ?
2. Décrire le fonctionnement des commandes `f.readline()`, `readline(n)` et `readlines()` ?
3. Écrivez une fonction `lecture` qui lit le fichier « test.txt » ligne par ligne et affiche son contenu.
4. Chaque ligne de texte se termine par la séquence de caractères `\n` qui s'appelle *séquence d'échappement* et désigne un passage à la ligne. Adapter la fonction `lecture` pour ne pas afficher ces séquences `\n`.
5. Modifiez la fonction `lecture` pour afficher le contenu d'un fichier dont le nom est donné en paramètre. Validez cette nouvelle fonction avec le fichier « test.txt ».
6. Essayez cette fonction avec un autre fichier texte.
7. Essayez cette fonction avec un nom de fichier qui n'existe pas. Modifiez la fonction `lecture` de manière à afficher "Le fichier <nom du fichier> n'existe pas".

3 Écriture de fichier

Comme pour la lecture, l'écriture d'un fichier se fait en trois temps :

L'ouverture du fichier : la valeur du paramètre "mode d'ouverture" est "w" ou "a" au lieu de "r".

L'écriture du contenu du fichier : se fait via la méthode `write()` en mettant en paramètre la chaîne de caractères à écrire dans le fichier.

La fermeture du fichier : est obligatoire pour enregistrer les modifications.

```
f␣=␣open("test_2.txt","w")␣#␣ouverture
f.write("Bonjour␣")
f.write("Ahmed\n")␣#␣première␣ligne
f.write("tu␣apprends␣Python␣!")␣#␣deuxième␣ligne
f.close()␣#␣fermeture␣et␣enregistrement
```

Remarques :

- Le mode d'ouverture "w" pour "write" permet d'écrire dans un fichier. Avec ce mode d'ouverture, si le fichier n'existe pas il sera créé, et s'il existe son ancien contenu sera écrasé avant de commencer l'écriture.
- Le mode d'ouverture "a" pour "append" permet d'ajouter des informations dans un fichier. Si le fichier n'existe pas, il sera créé, et s'il existe l'écriture commence juste à la fin du fichier.
- L'écriture du fichier n'est pas faite au moment de l'usage de la fonction `write()` mais au moment de la fermeture avec `close()`.

Exercice 2

1. Écrivez une fonction `ecriture` qui écrit dans un fichier ce qui est saisi au clavier jusqu'à ce qu'une ligne vide soit saisie.
2. Testez votre fonction et observez en particulier que toutes les lignes saisies sont mises bout à bout dans le fichier.
3. Améliorer votre fonction `ecriture` de manière à ce que chaque ligne saisie soit sur une ligne dans le fichier écrit.
4. Écrivez une fonction `ajout` qui Ajoute quelques lignes dans un fichier (vérifiez que cela est bien réalisé).

4 Compléments : gestion de chemins d'accès

Un fichier n'est pas nécessairement dans le dossier de travail. Il peut s'avérer nécessaire de préciser sa localisation. Les chemins *absolus* et *relatifs* sont deux moyens de décrire le chemin menant à des fichiers ou répertoires.

Une première solution consiste à spécifier le *chemin absolu* (chemin précis) du fichier qu'on désire manipuler. Pour ce la, on décrit la suite des répertoires menant au fichier à partir du nom de volume (C :, D :).

```
f = open("D:/votre_classe/votre_nom_prenom/tp_14/test.txt","r")#1 slash «/» ou 2 antislashs «\\»
```

Une deuxième solution consiste savoir dans quel dossier travaille l'interpréteur. Au lancement de l'interpréteur, le *répertoire de travail courant* est celui dans lequel se trouve l'exécutable de l'interpréteur ou celui d'où vous lancez votre programme.

```
import os                # os : operating system
print(os.getcwd())       # Afficher le répertoire courant (cwd : Current Working Directory)
```

Cela peut permettre d'utiliser des *chemins relatifs* (relatifs à l'endroit où travaille Python). Donc il est possible de changer de répertoire de travail de l'interpréteur.

```
os.chdir("D:/votre_classe/votre_nom_prenom/tp_14") # chdir : change directory
f = open("test.txt","r") # accéder directement à votre fichier
```

Exercice 3

1. Écrivez une fonction `copier(fichier)` qui recopie un fichier texte. Le programme demandera à l'utilisateur le nom (avec chemin d'accès) de fichier et le recopier dans le même emplacement.
2. Écrire une fonction `comparer` qui compare les contenus de deux fichiers et signale la première différence rencontrée.
3. A partir de deux fichiers préexistants A et B, construire un fichier C qui contienne alternativement un élément de A, un élément de B, un élément de A, et ainsi de suite, jusqu'à atteindre la fin de l'un des deux fichiers originaux. Compléter ensuite C avec les éléments restants sur l'autre.

Exercice 4

Écrire un script python qui permet de définir et d'appeler les fonctions suivantes :

1. Créer une fonction `saisie_listes(n,m)` qui saisie et retourne n listes chacune de m entiers positifs. Tester cette fonction pour n=10 et m=10.
2. Créer une fonction `remplir(fichier, listes)` qui enregistre les listes dans un fichier texte dont le chemin est passé en paramètres. Tester cette fonction pour enregistrer les listes crées dans la question 1 dans un fichier nommé «fichier1.txt».
   Exemple :
   La liste `[[12, 5], [23, 4], [25, 1]]` sera enregistrée dans le fichier text sous le format suivant :
   12,5
   23,4
   25,1
3. Créer une fonction `trier_listes_fichier(fichier_source, fichier_distination)` qui permet de lire les listes d'entiers à partir de `fichier_source`, trier les entiers de chaque liste dans l'ordre croissant et enregistrer les résultats dans le `fichier_distination`. Tester cette fonction avec
   `fichier_source` : «fichier1.txt» et `fichier_distination` : «fichier2.txt».
4. Créer une fonction `trier_lignes_fichier(fichier_source, fichier_distination)` qui permet de lire les listes d'entiers à partir de `fichier_source`, trier les listes d'entiers dans l'ordre croissant selon le premier entier de chaque liste et enregistrer les résultats dans le `fichier_distination`. Tester cette fonction avec `fichier_source` : «fichier2.txt» et `fichier_distination` : «fichier3.txt».
5. Créer une fonction `recherche(n,fichier)` qui retourne les numéros de lignes et numéros de colonnes de chaque occurrence de l'entier n dans le fichier passé en paramètres ou bien elle retourne `None` si l'entier n'existe pas. Tester cette fonction sur le fichier «fichier3.txt» pour un entier n saisi au clavier.

Exercice 5

1. Écrivez un script qui cherche un mot dans un fichier texte et affiche, pour chaque occurrence trouvée, le numéro de la ligne contenant ce mot et la position du mot dans cette ligne. Le programme demandera à l'utilisateur le nom (avec chemin d'accès) de fichier et le mot à chercher.
2. Modifier le script précédent pour qu'il puisse chercher un mot dans une liste de fichiers d'un répertoire donné. Le programme demandera le nom du répertoire (avec chemin d'accès), le mot à chercher et affichera dans ce cas : le nom du fichier, le numéro de ligne et la position de chaque occurrence du mot cherché.
3. Modifier le script précédent pour qu'il puisse chercher un mot dans une liste de fichiers d'un répertoire donné et récursivement dans les fichiers de ses sous-répertoires.

TP N°15 : Le module incontournable du calcul scientifique : Numpy
Les vecteurs

Ce TP a pour objectif de présenter l'essentiel du module Numpy de Python grâce à des exemples à saisir dans la console (sous IDLE) et quelques exercices d'applications.

Présentation du module Numpy :

Le module numpy est la boîte à outils indispensable pour faire du calcul scientifique avec Python. Pour modéliser les vecteurs, les matrices, et plus généralement les tableaux à n dimensions, numpy fournit le type ndarray (*N dimensional array*). Il y a des différences majeures avec les listes (resp. les listes de listes) qui pourraient elles aussi nous servir à représenter des vecteurs (resp. des matrices) :

- Les tableaux numpy sont homogènes, c'est-à-dire constitués d'éléments du même type. On trouvera donc des tableaux d'entiers, des tableaux de flottants, des tableaux de chaînes de caractères, etc.
- La taille des tableaux numpy est fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme le ferait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ce module n'est pas fourni avec la distribution Python de base, il doit être installé à partir du site officel :
`http://docs.scipy.org/doc/numpy-1.10.1/user/install.html`
Le module numpy propose plusieurs sous-modules intéressants :

- `numpy.linalg` : un module d'algèbre linéaire basique,
- `numpy.random` : un module pour les générateurs aléatoires,

On charge traditionnellement l'ensemble du module numpy en le renommant np : `import numpy as np`
Pour ne pas surcharger l'espace des noms et risquer la présence d'homonymes, on ne charge pas un module par l'instruction `from module import *`

Création des tableaux

Création avec np.array

On définit souvent un tableau numpy à partir d'une liste (ou lise de listes), en utilisant la fonction array du module numpy. La fonction array agit comme un mécanisme de conversion de type 'list' vers'numpy.ndarray' qui est le type commun à tous les tableaux numpy.

```
a = np.array( [1, 2, 3] ) # créer un tableau à partie d'une liste
print(type(a))  # <class 'numpy.ndarray'>
print(a.dtype) # int64
b = a.tolist() #créer un objet b de type list, copie de l'objet a de type ndarray
```

Remarques :

— L'attribut dtype (abréviation de *data type*) de a, 'int64' qui indique le type commun à tous les éléments du tableau, ici des entiers codés sur 64 bits, c'est-à-dire quatre octets ou *bytes*.

— La méthode tolist d'un tableau numpy le transforme en la liste (éventuellement une liste de listes) de ses éléments. Attention, ce c'est pas une fonction du module numpy. On n'écrira donc pas `np.tolist(a)` mais `a.tolist()`.

Création avec des fonctions spéciales

La fonction `arange(d, f, h, dtype=...)` crée des vecteurs de *valeurs régulièrement espacées* dans l'intervalle `[d, f[` avec un pas de h.

```
t0 = np.arange(0, 10, 2) # début=0, fin(exclue)=10 , pas=2 -> t0 = [0 2 4 6 8]
t1 = np.arange(3) # tableau de 3 entiers -> t1 = [0 1 2]
t2 = np.arange(3.) # tableau de 3 réels -> t2 = [ 0.  1.  2.]
t3 = np.array([2]*5); #concaténation de 5 exemplaires de la liste [2] -> t3 = [2 2 2 2 2]
```

La fonction `linspace(a, b, n)` retourne un vecteur de n valeurs régulièrement échelonnées du segment `[a, b]` (donc ici les extrémités sont incluses). Par défaut, $n = 50$. Si $b < a$, les valeurs sont obtenues dans l'ordre décroissant. Ces fonctions sont en particulier utilisées pour le tracé des fonctions. Exemple : `b = np.linspace(0, 10, 5)`

Exercice 1

Construire un objet de type numpy.ndarray

1. dont les termes sont les multiples de 3 compris entre 0 et 27 ;
2. de taille égale à 100 dont les extrémités sont 0 et 27 ;

Lecture de valeurs dans un vecteur : « slicing »

La lecture d'éléments d'un tableau `numpy` procède par coupes (« slices » en anglais). La syntaxe est la suivante :

`t[indice_début(inclu) : indice_fin(exclu) : incrément]`

```
v = np.array(range(0,160,10))
v[0] # le 1er élément : compte à partir du début
v[-v.size] # le 1er élément : on compte à partir de la fin
v[-1] # le dernier élément
v[v.size-1] # le dernier élément
```

Quelques coupes de valeurs consécutives dans l'ordre des indices croissants :

```
v[4:11] # de v[4] à v[10], donc 11-4 = 7 éléments consécutifs
v[:] # la totalité du vecteur v
v[::-1] # la totalité du vecteur v mis à l'envers (incrément = -1)
v[:6:2] # (incrément = 2)
```

Accès aux données dans un ordre quelconque :
Si *a* est un tableau, et si *I* est un tableau d'indices (éventuellement une liste ou un tuple), alors *a[I]* renvoie le tableau (de même format que le tableau *I*) formé des éléments *a[i]*, où *i* est dans *I*.

```
v = np.array(range(0,160,10))
a = v[[6,2,1,3]]
indices = np.array([[5,2,1],[3,8,7]])
b = v[indices]
np.take(v,[6, 1, 5, 0]) # renvoie le tableau des valeurs v[6], v[1], v[5], v[0]
```

Écriture de valeurs dans un vecteur

Les tableaux `numpy` sont mutables, on peut donc modifier les valeurs qu'ils contiennent sans redéfinir l'objet lui-même. Si *a* est un vecteur `numpy`, l'instruction `a[n] = x` écrit la valeur `x` en position `n`. La valeur `x` écrite en position `n` du vecteur `a` doit a priori être compatible avec le « `data type` » de `a`.

```
a = np.arange(10); a[7] = 21 # écrit la valeur 21 en position 7
```

On peut écrire plusieurs valeurs simultanément dans un vecteur, en utilisant une syntaxe du genre `a[coupe ]=...`. Il faut simplement veiller à ce que le membre de droite (la source) soit « `array-like` » (une liste, un tuple, un tableau) et que la coupe spécifiée *a* (donc la cible) soient de même taille. Ainsi, les instructions suivantes ont le même effet : `a[4:7] = [111,222,333]` et `a[4:7] = (111,222,333)` ou encore `a[4:7] = np.array([111,222,333])` .

On peut écrire plusieurs valeurs simultanément dans un vecteur dans un ordre quelconque, en utilisant une syntaxe du genre `a[I]=V`, avec `I` un tableau d'indices et `V` un tableau de valeurs à insérer dans le tableau `a`.

```
a[[7, 2, 5, 3]] = (7777, 2222, 5555, 3333) # on modifie a[7], a[2], a[5] et a[3]
a.put([6,1,5,2],[66,11,55,22]) # place 66 en position 6, 11 en position 1, etc.
```

Exercice 2

Définir les objets suivants : `L1 = [1,2,3]` et `L2 = [4,5,6]`

1. Créer les deux vecteurs `v1` et `v2` à partir de `L1` et `L2`.
2. Écrire une fonction `prod_scal(v1,v2)` prenant en entrée deux vecteurs de même taille `v1` et `v2` et renvoyant leur produit scalaire. Donner sa complexité.
3. En déduire une fonction `ortho(v1,v2)` renvoyant un booléen indiquant si les vecteurs sont orthogonaux.
4. Écrire une fonction `prod_vec(v1,v2)` prenant en entrée deux vecteurs de taille 3 et renvoyant leur produit vectoriel. On pourra utiliser la commande `np.zeros(n)` créant un vecteur de taille *n* rempli de 0.

TP N°16 : Le module incontournable du calcul scientifique : Numpy
Les matrices

On importera la bibliothèque Numpy au moyen de la commande : `import numpy as np`

Création des tableaux bi-dimensionnels

Création avec np.array

On définit souvent un tableau numpy à partir d'une lise de listes, en utilisant la fonction array du module numpy. La fonction array agit comme un mécanisme de conversion de type 'list' vers 'numpy.ndarray' qui est le type commun à tous les tableaux numpy.

```
a = np.array([[1, 2, 3],[2, 3, 4],[0, 1, 0]]) # créer une matrice à partie d'une liste de listes
print(type(a))  # <class 'numpy.ndarray'>
print(a.dtype) # int64
b = a.tolist() #créer un objet b de type list, copie de l'objet a de type ndarray
```

Création avec des fonctions spéciales

Découvrons quelques fonctions spéciales : np.ones, np.zeros, np.eye, np.diag,...
Rappelons que chez les booléens, False = 0 = None = « vide », et True = 1 = not None = « non vide » :

```
a = np.ones((3, 5)) # crée une matrice de 3 lignes × 5 colonnes, dtype = float (par défaut)
b = np.ones((3,5), dtype = np.int)
c = np.zeros((3, 5))
d = np.zeros((3, 5), dtype = np.bool)
e = np.eye(3) # Crée une matrice identité d'ordre n
g = np.eye(4,4)
h = np.diag([1,2,3])  # Crée une matrice diagonale
i = np.diag([1,2,3], k = 1) # essayer : -1, 2 (décalage de la diagonale)
A = np.ones((2,3))
B = np.zeros((2,3))
AB0 = np.concatenate((A,B), axis = 0) # crée une matrice par concaténation de B sous A
AB1 = np.concatenate((A,B), axis = 1) # crée une matrice par concaténation de B à droite de A
# np.column_stack :
v1 =np.array([1,2,3]); v2 =np.array([12,22,32]); v3 =np.array([13,23,33])
v = np.column_stack((v1,v2,v3)) # crée une matrice par empilement des vecteurs passés en arguments.
#Autres méthodes utiles
v.fill(1) # remplir le tableau v par une valeur constante.
a.fill(3.14) # remplir le tableau a par la valeur flottante 3.14
v.fill(3.14) # La valeur flottante 3.14 a été convertie en valeur entière 3 pour le remplissage
z = np.array([[1,2],[3,4]]*2) #tableau avec des répétitions
```

Tableaux répondant à une formule donnée

La fonction fromfunction permet de construire un tableau dont le terme général obéit à une formule donnée.

```
def f(i,j): return 10*i+j
t1 = np.fromfunction(f,(4,5)) # t1[i,j] = f(i,j)
t2 = np.fromfunction(f,(4,5), dtype = int) # On exige un tableau de type int
# Le même tableau peut être obtenu en convertissant une liste (liste de listes) "en compréhension"
t3 = np.array([[f(i,j) for j in range(5)] for i in range(4)])
```

Exemple : Matrice d'Hilbert, de terme général : $H[i,j] = \frac{1}{(i+j+1)}$ avec $i, j \geq 0$

```
t4 = np.fromfunction(lambda i,j: 1/(i+j+1),(4,4))
```

Exercice 1

Construire un objet de type numpy.ndarray

1. dont les termes sont les multiples de 3 compris entre 0 et 27 ;
2. de taille égale à 100 dont les extrémités sont 0 et 27 ;

Lecture de valeurs dans une matrice

Il est possible de sélectionner une sous matrice, en ne gardant que quelques lignes consécutives et/ou certaines colonnes consécutives. La spécification la plus générale d'une coupe d'un tableau M est :

```
M[ numero_ligne : numero_colonne ]
-----------------------------------------------------------------------------------------
M[ ligne_début(inclue) : ligne_fin(exclue) , colonne_début(inclue) : colonne_fin(exclue) ]
```

Si M est une matrice d'ordre $a * b$:

- `M[a,b]` : élément en position (a,b)
- `M[a] ou M[a,:]` : Vecteur ligne en position a
- `M[:,b]` : Vecteur colonne en position b
- `M[a:b,c:d]` : lignes de a à b-1, colonnes de c à d-1
- `M[:,:]` : une copie intégrale de M avec nouvelle référence
- `M[:a,:b]` : sous matrice les lignes du début jusqu'à a exclu ; les colonnes de début jusqu'à b exclu (Les a preimères lignes, les b premières colonnes).
- `M[::-1]` : On inverse l'ordre des lignes
- `M[:,::-1]` : On inverse l'ordre des colonnes
- `M[::2,::2]` : lignes et colonnes paires
- `M[1::2,1::2]` : lignes et colonnes impaires

```
c = np.array( range(24) )
c.shape = (3, 4, 2) ; print (c)
print(c[1,:,:])
print(c[1][0][2]) # troisième élément de la première colonne de la deuxième ligne du tableau c
```

Accès aux données dans un ordre quelconque :
Si m est une matrice, et si I est un tableau d'indices (éventuellement une liste ou un tuple), alors $m[I]$ renvoie le tableau (de même format que le tableau I) formé des éléments $m[i]$, où i est dans I.

```
m = np.arange(24).reshape(4,6); m
a = m[[3,1,0,1]] # on demande le tableau des lignes m[3] , m[1] , m[0] et m[1]
i = np.array([3,0,2]) # trois indices
m.take(i,axis=0) # lectures de lignes
m.take(i,axis=1) # lectures de colonnes
m.take(i) # lecture de m "aplati"
```

Écriture de valeurs dans une matrice

Exemples :

```
a = np.array(range(10))
a.shape = (2, 5) ; print(a)
a[0,0] = 7 # modifier un élément d'indice (0,0)
a[0,:] = np.zeros(5) # modifier la ligne d'indice 0
a[:,0] =np.ones(2) # modifier la colonne d'indice 0
b = np.concatenate((a,a), axis = 0) ;
b[2:4,2:4] = np.ones((2,2)) # modifier une sous-matrice
a[ a % 2 == 0 ] /= 2  ; print(a) # les éléments pairs de a sont divisés par 2
```

Quelques méthodes sur les tableaux

On peut utiliser les méthodes des objets (ou les fonctions associées du module `numpy`) : `a.max()`, `a.min()`, `a.sum()`, `a.prod()`, `a.mean()` (moyenne arithmétique), `np.apply_along_axis`.

```
a = np.random.randint(0, 10, (2,5)) ; print(a)
print(a.sum()) # ou encore np.sum(a)
a.sum(axis=0) # somme des lignes
```

```
a.sum(axis=1) # somme des colonnes
print(a.mean()) #moyenne arithmétique
print(a.mean(axis=0))
print(a.mean(axis=1))
m = np.arange(15).reshape(3,5); m
np.apply_along_axis(np.mean, 0, m) # moyenne lignes
np.apply_along_axis(np.mean, 1, m) # moyenne colonnes
print(a.max())
print(a.max(axis=0))
print(a.max(axis=1))
```

Exercice 2

1. Définir une matrice aléatoire a de flottants, de taille 50×50.
2. Compter le nombre de valeurs inférieures à 0.5.
3. Remplacer toutes les valeurs inférieures à 0.5 par 0, et celles strictement supérieures à 0.5 par 1.

TP N°17 : Le module incontournable du calcul scientifique : Numpy
Les fonctions universelles

On importera la bibliothèque Numpy au moyen de la commande : `import numpy as np`

Introduction

Une *fonction universelle* (le terme exact est « ufunc », abréviation de *universal function*) est une fonction qui peut s'appliquer terme à terme aux éléments d'un tableau.
Si f est une *ufunc* et si $a = [a_0, a_1, ..., a_{n-1}]$ est un tableau, alors $f(a)$ renvoie le tableau $[f(a_0), f(a_1), ..., f(a_{n-1})]$.
Les a_k peuvent être des tableaux, par exemple les lignes d'une matrice m. La fonction f s'applique alors récursivement aux éléments de m.
Un grand nombre de fonctions usuelles sont directement « universalisées » dans numpy. Les fonctions mathématiques gardent le même nom (préfixé par np si on a importé numpy par `import numpy as np`). Exemple : `np.exp(t)`
Si on effectue une opération arithmétique entre un tableau a et un scalaire x, tout se passe comme si l'opération arithmétique est effectuée entre x et chaque élément de tableau a.
Pour la documentation officielle en anglais, voir : http ://docs.scipy.org/doc/numpy/reference/ufuncs.html
Le mieux est de commencer par quelques opérations arithmétiques simples sur un vecteur ou une matrice.

Les opérations arithmétiques

Les opérations $+, *, /, ==$, etc. s'appliquent aux tableaux numpy, mais TERME à TERME.
L'expression `a * b` renvoie le tableau des produits terme à terme des éléments de a et b : elle ne désigne donc pas un produit matriciel au sens où on l'entend habituellement.

```
a = np.random.randint(0, 10, (2, 5))
-a ; negative(a) #tableaux des opposés
b = np.random.randint(1, 10, (2, 5)); print(b)
a + b ; np.add(a,b)  # additionne terme à terme les éléments de a et b
a - b ; np.subtract(a,b) #  soustrait terme à terme les éléments de b à ceux de a
a * b ; np.multiply(a,b) # multiplie terme à terme les éléments de a et b
a // b ; np.floor_divide(a,b) #  quotients (entiers) des divisions des éléments de a par ceux de b
a % b  ; np.mod(a,b) # donne les restes dans les divisions des éléments de a par ceux de b
a / b ; np.divide(a,b) # quotients (flottants) terme à terme des éléments de a par ceux de b
a ** b ; np.power(a,b) # élever les éléments de a à la puissance les éléments de b
a == b
```

Fonctions mathématiques usuelles

Toutes les fonctions mathématiques usuelles sont présentes sous forme universelle dans le module numpy (il faut bien penser à les préfixer par np).

- `log log10 log2` : logarithme népérien (resp. de base 10, de base 2).
- `degrees(t) (ou encore rad2deg)` : convertit les angles t de radians en degrés.
- `radians(t) (ou encore deg2rad)` : convertit les angles t de degrés en radians.
- fonctions trigonométriques : `sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh`

Exemple :

```
rad = np.array([pi/6, pi/4, pi/3, pi/2, pi])
deg = np.degrees(rad); # deg = [  30.,   45.,   60.,   90.,  180.]
```

Vectorisation d'une fonction

Pour qu'une fonction f puisse s'appliquer, terme à terme, à tous les éléments d'un ou de plusieurs tableaux (selon que f accepte un ou plusieurs arguments), il faut la *« vectoriser »*.
Exemple 1 : Considérons la fonction $f : x \longmapsto \dfrac{x + sin(x)}{x + cos(x)}$

```
def f(x): return (x+sin(x))/(x+cos(x))
a = np.arange(1,7)
f(a) # Tenter d'appliquer la fonction f à un tableau numpy conduit à une erreur
```

Il faut donc utiliser la fonction `vectorize` pour rendre f universelle.

```
vf = np.vectorize(f) # vf estt la version vectorisée de f
vf(a) # on peut applique f terme à terme aux éléments de a
(a+np.sin(a))/(a+np.cos(a)) # même résultat avec des opérations usuelles (toutes déjà vectorisées)
```

Exemple 2 :

```
def f(x,y):
        return 0 if x < y else y
vf = np.vectorize(f)
a = np.arange(1,7); a # array([1, 2, 3, 4, 5, 6])
b = np.array([4,1,8,7,2,9]); b # array([4, 1, 8, 7, 2, 9])
vf(a,b) # array([0, 1, 0, 0, 2, 0])
vf(b,a) # array([1, 0, 3, 4, 0, 6])
```

En continuant sur cet exemple, on voit bien que la version vectorisée de f agit comme une « *ufunc* » (fonction universelle).

```
vf(a,3) #array([0, 0, 3, 3, 3, 3])
vf(3,b) # array([0, 1, 0, 0, 2, 0])
vf(5,[[1,6,2],[3,4,8],[6,3,2]]) # array([[1, 0, 2],[3, 4, 0],[0, 3, 2]])
```

Exercice 1

1. Soit la fonction $h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$. Créer des tableaux xt et ht qui contiennent respectivement 41 valeurs reparties uniformement pour $x \in [-4,4]$ et l'évaluation de la fonction h sur ces valeurs.

TP N°18 : Le calcul matriciel

On importera la bibliothèque `Numpy` au moyen de la commande : `import numpy as np`

Introduction

Le calcul matriciel ...

Produits matriciels

Pour deux vecteurs $a = [a_1, ..., a_n]$ et $b = [b_1, ..., b_n]$, on calcule ici le produit scalaire réel $\sum_{k=1}^{n} a_k b_k$

Un vecteur v est considéré comme une ligne à gauche, et une colonne à droite.
Pour effectuer le produit matriciel : c'est la fonction `np.dot(a,b)` ou la méthode `a.dot(b)`.

```
A=np.array([[1, 2], [3, 4]])
B=np.array([[1, 1, 1], [2, 2, 2]])
np.dot(A,B) # array([[ 5,  5,  5],[11, 11, 11]])
np.dot(A,B).dot(np.ones((3,2))) # calcul du produit de plusieurs matrices
```

La fonction `vdot` permet de calculer le produit scalaire de deux vecteurs de R^n.

```
u=np.array([1, 2])
v=np.array([3, 4])
np.vdot(u, v) # 11
```

La fonction `cross` permet de calculer le produit vectoriel de deux vecteurs de R^3.

```
u=np.array([1, 0, 0])
v=np.array([0, 1, 0])
np.cross(u, v) # array([0, 0, 1])
```

Inversion de matrices, résolution de systèmes

```
A=np.array([[1, 1, 1], [2, 2, 2]])
np.transpose(A) # renvoie la transposée de A
A.T # renvoie aussi la transposée de A

import numpy.linalg as alg

alg.det(A) # le déterminant de A
a = np.array([[1,-2,3],[2,1,4],[5,-1,2]]);
b = np.array([14,12,13]);
x = alg.solve(a,b) # résout le système A·x = B. x = array([ 1., -2.,  3.]))
```

Normes matricielles et vectorielles

`linalg.norm(x)` calcule une norme d'un vecteur ou d'une matrice. $M = \sqrt{\sum |m_{i,j}|^2}$

```
m = np.arange(-10,10).reshape(5,4) # m une matrice de taille 5×4
x = np.linalg.norm(m) # x = 25.88435821108957
```

Exercice 1

1. Implémenter le pseudo-code précédent en python et vérifier son fonctionnement.

TP N°19 : Simulation d'expériences et traçage de courbes
Le module `Matplotlib`

Ce TP a pour objectif d'apprendre à utiliser le module `Matplotlib` de Python grâce à des exemples à saisir dans la console (sous IDLE) et quelques exercices d'applications.

Présentation

Le module `Matplotlib` est chargé de tracer les courbes. Dans ce TP on donne un bref aperçu des possibilités graphiques de `Matplotlib`. Il est possible de générer des graphiques à deux et à trois dimensions, et d'agir facilement sur les attributs du graphe (couleurs, type de ligne, axes, annotations...).
On importera la bibliothèque `Matplotlib` au moyen de la commande : `import matplotlib.pyplot as plt`

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

Les commandes présentées ci-dessous sont utiles notamment pour être introduites dans des scripts et automatiser la production de figure.

La fonction `plot()`

D'une manière générale les fonctions `plt.plot` attendent des tableaux de points du plan. Selon les options, ces points du plan sont reliés entre eux de façon ordonnée par des segments : le résultat est une courbe.
La fonction `plot(x,y)` permet de tracer une courbe reliant les points dont les **abscisses** sont données dans le vecteur **x** et les **ordonnées** dans le vecteur **y**. Commençons par la fonction *sinus* :

```
import matplotlib.pyplot as plt
import numpy as np
from math import pi
x= np.arange(0,2*pi,pi/16) # ou x= np.linspace(0,2*pi,100) :  échantillonner la variable x
plt.plot(x,np.sin(x),label="sin") # on utilise la fonction sinus de Numpy
plt.axis("equal") # Imposer une échelle identique sur les deux axes de coordonnées.
plt.xlim(-2,10) # Fixer l'intervalle de visualisation sur l'axe des abscisses.
plt.ylabel('fonction sinus')
plt.xlabel("l'axe des abcisses")
plt.title('Fonction sinus')
plt.savefig('ma_figure.png') # sauvegarder la figure (en .png ou .pdf)
plt.show() #Afficher l'image
plt.clf()
```

Si tout se passe bien, une fenêtre doit s'ouvrir avec la figure ci-dessus.

Exercice 1

1. Écrire un script qui demande à l'utilisateur d'introduire des nombres $x_1, y1, x_2, y_2, x_3, y_3$ et trace le triangle ABC avec $A(x_1, y_1), B(x_2, y_2)$ et $C(x_3, y_3)$.
2. Tracer la courbe représentative de $x \longmapsto cos(x)$ sur l'intervalle [-5, 5] à l'aide de la fonction `plot()`.
   Pour utiliser la fonction `plot()` de `matplotlib.pyplot`, on pourra exécuter les instructions :

   ```
   X = arange(-5,5,0.1)
   Y = cos(X)
   ```

Manipuler plusieurs graphiques

On peut utiliser plusieurs fenêtres graphiques en même temps, à condition de stoker les figures dans des variables :

```
t = np.linspace(0, 2*pi, 50)
fig1 = plt.figure()
plt.plot(t, np.cos(t), label="cos")
fig2 = plt.figure()
plt.plot(t, np.sin(t), label="sin")
plt.show() #Afficher toutes les figures
```

Une même fenêtre peut intégrer plusieurs graphes non superposés grâce à la commande `subplot(n, m, k)` qui permet de subdiviser la fenêtre en $n \times m$ cases, n lignes et m colonnes (comme pour les matrices).
Ces cases sont numérotées de gauche à droite et de haut en bas. La valeur de k permet de spécifier dans quelle case on désire faire un graphique.

```
plt.subplot(2, 1, 1)
plt.plot(t, np.cos(t))
plt.subplot(2, 1, 2)
plt.plot(t, np.sin(t))
plt.show() #Afficher toutes les figures
```

Il est possible de superposer deux courbes sur le même graphique :

```
plt.plot(t, np.cos(t), t,np.sin(t))
plt.show() #Afficher toutes les figures
```

Graphiques à 3 dimensions : Les courbes

Il est possible de tracer des courbes dans l'espace avec la librairie Axes3D. Par exemple une hélice :

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.plot(np.cos(t), np.sin(t), t, label="helice")
plt.show()
```

Autres options

Pour connaître toutes les options, le mieux est de se référer à la documentation de `Matplotlib`. Voyons ici quelques unes d'entre elles :

- **Ajouter un titre :** `plt.title("Mon titre")`
- **Légendes :** `plt.legend((g1, g2), ("ligne2","ligne 1"))`.
- **Épaisseur du trait :** `linewidth=flottant`
- **Style du trait :** se décide avec `linestyle` : '-' ligne continue, '–' tirets, '-.' points-tirets, ' :' pointillés, ...
- **Couleur du trait :** pour changer la couleur du tracé une lettre g vert (green), r rouge (red), k noir, b bleu, c cyan, m magenta, y jaune (yellow), w blanc (white).
- **Bornes :** spécifier un rectangle de représentation, ce qui permet un zoom, d'éviter les grandes valeurs des fonctions par exemple, se fait via la commande `plt.axis([xmin,xmax,ymin,ymax])`
- **Symboles :** mettre des symboles aux points tracés se fait via l'option marker. Les possibilités sont nombreuses parmi ['+' | '*' | ',' | '.' | '1' | '2' | '3' | '4' | '<' | '>' | 'D' | 'H' | 'ˆ' | '_' | 'd' | 'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|' | TICKUP | TICKDOWN | TICKLEFT | TICKRIGHT | 'None' | ' '].

Exercice 2

1. Réaliser le graphe de la fonction $y(t) = v_0 t - \frac{1}{2}gt^2$ pour $v_0 = 10$, $g = 9.81$, et $t \in [0, 2\frac{v_0}{g}]$.
   Le label sur l'axe des x devra être "temps (s)" et le label sur l'axe des y est "hauteur (m)".
2. La fonction $f(x,t) = e^{-(x-3t)^2} sin(3\pi(x-t))$ décrit pour une valeur fixe de t une onde localisée en espace. Faire un programme qui visualise cette fonction comme une fonction de x dans l'intervalle $x \in [-4,4]$ pour $t = 0$.

TP N°20 : Méthodes de recherche des zéros d'une fonction

1 Introduction

De nombreux problèmes en chimie, mathématiques, physique et sciences de l'ingénieur nécessitent la résolution d'équations de type $h(x) = g(x)$ où h et g sont des fonctions et où x est l'inconnue.
Lorsque la résolution formelle de l'équation n'est pas simple, ou même impossible, on peut utiliser un résolution numérique. Pour cela on se ramène à une équation avec second membre nul : $g(x) - h(x) = 0$. On cherche donc à déterminer les zéros de la fonction f telle que $f(x) = g(x) - h(x)$.
Dans cette partie, nous allons voir comment évaluer numériquement une solution d'équation de la forme $f(x) = 0$. Deux méthodes vont être présentées, mais il en existe d'autres : La *méthode de dichotomie* et la *méthode de Newton*.

2 La méthode de dichotomie

2.1 Principe

Le cours de math nous apprend que si f est une fonction à valeurs réelles, continue sur un intervalle $[a,b] \subset \mathbb{R}$ telle que $f(a)$ et $f(b)$ sont de signes opposés, $f(a) \times f(b) < 0$, le théorème des valeurs intermédiaires nous assure que l'équation f admet une solution dans $[a,b]$. La démonstration de ce théorème nous donne un algorithme effectif pour déterminer des solutions approchées de telles équations.

2.2 Rappel : Théorème de la valeur intermédiaire

Soient f une fonction continue sur un intervalle I de $\mathbb{R}$, $(a,b) \in I^2$ tel que $a < b$ et $f(a) \times f(b) < 0$ et ϕ l'application de $[a,b]^2$ dans lui-même définie par :

$$\phi(a,b) = \begin{cases} (a, \frac{a+b}{2}) & \text{si } f(a)f(\frac{a+b}{2}) < 0 \\ (\frac{a+b}{2}, b) & \text{sinon} \end{cases}$$

Alors,

- Les deux suites $(a_n)_{n\in\mathbf{N}}$ et $(b_n)_{n\in\mathbf{N}}$ telles que

$$\begin{cases} (a_0, b_0) & = (a,b) \\ (a_{n+1}, b_{n+1}) & = \phi(a_n, b_n) \end{cases}$$

  sont correctement définies et l'on a pour tout $n \in \mathbb{N}$, $b_n - a_n = \frac{b-a}{2^n}$,
  avec $a_0 \leq a_n \leq a_{n+1} \leq b_{n+1} \leq b_n \leq b_0$;
- il existe $c \in I$, tel que $f(c) = 0$ et, pour tout $n \in \mathbb{N}, a_n \leq c \leq b_n$.

2.3 Présentation de la méthode par un exemple : valeur approchée de $\sqrt{2}$

Déterminer une valeur approchée de $\sqrt{2}$ à une précision $\epsilon = 10^{-7}$ (la précision est un nombre réel arbitrairement petit).
Rappel : Pour calculer la valeur approchée de $\sqrt{a}$ on doit résoudre l'équation $f(x) = x^2 - a = 0$, avec a réel 0 .

2.4 Méthode de travail et algorithme

1. On saisit l'expression de la fonction $f(x) = x^2 - a$ ainsi que les bornes a et b de l'intervalle d'étude et la précision p désirée.
2. On calcule la valeur $m = \frac{a+b}{2}$ puis on coupe l'intervalle en deux en fonction du signe du produit $f(a) \times f(b)$:
   - si $f(a) \times f(b) < 0$ alors on considère l'intervalle $[a;m]$.
   - si $f(a) \times f(b) > 0$ alors on considère l'intervalle $[m;b]$.
3. Si l'amplitude de l'intervalle est inférieure à p on arrête le processus sinon on calcule une nouvelle valeur de m.
4. A la fin du processus, les bornes de l'intervalle d'amplitude inférieure à p se trouvent dans le variables a et b ;

Algorithme

```
#Première implémentation de la dichotomie
def dicho (f,a,b,epsilon):
        u,v = min(a,b),max(a,b)
        while (v-u) > epsilon :
                m = (u + v)/2
                if  f(u) * f(m) < 0 :
                        v = m
                else:
                        u = m
        return u,v
```

2.5 Mise en Oeuvre

Tracer sur le graphe, pour chaque itération, les bornes des intervalles $[a_n, b_n]$.

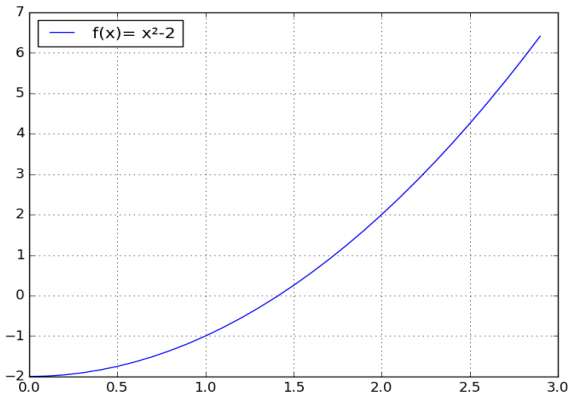


Figure 1 – Recherche du zéro d'une fonction $f(x) = x^2 - a = 0$ par dichotomie

Pour $[a,b] = [1,2]$ on aura :

- $f(1) = -1 < 0 < 2 = f(2)$, donc l'équation $f(x) = 0$ possède un solution dans $[1,2]$.
  La valeur médiane de cet intervalle est 1.5.
- $f(1) = -1 < 0 < f(1.5) = 0.25$, donc l'équation $f(x) = 0$ possède un solution dans $[1,1.5]$.
  La valeur médiane de cet intervalle est 1.25.
- $f(1.25) = -0.4675 < 0 < f(1.5)$, donc l'équation $f(x) = 0$ possède un solution dans $[1.25,1.5]$.
- ...
- $f(1.4140625) < 0 < f(1.41430664062) \simeq 0.000263273715973$, donc l'équation $f(x) = 0$ possède un solution dans $[1.4140625, 1.41430664062]$.

2.6 Exercices

Exercice 1

Soit $p \in \mathbb{N}$. Peut on evaluer le nombre n d'itérations qui permettront d'obtenir $b_n - a_n \leq 10^{-p}$?

Exercice 2

On choisit la fonction $f : x \longmapsto x^2 + 6x + 1$.

1. Calculer ses racines formellement
2. Déterminer des valeurs approchées à 10^{-12} près de chacune de ses racines avec votre programme $dico(f,a,b,epsilon)$. Avant de lancer le programme déterminer le nombre d'itérations prévisible. Ajouter un compteur pour vérifier que tout conforme aux prévisions.

Exercice 3

On choisit la fonction $g : x \longmapsto 10^{-8}x^2 - \frac{4}{5}x + 10^{-8}$

1. Préciser les racines formellement et en donner ensuite des valeurs approchées avec toute la précision possible.
2. Estimer en fonction de a et b, le nombre d'itérations provoquées par un appel `dicho(g,a , b, 10**(-12))`.
3. Lancer votre programme pour rechercher la racine comprise entre 0 et 1 avec une précision de 10^{-12};
4. Lancer votre programme pour rechercher la deuxième racine avec une précision de 10^{-12};
5. Si l'essai qui précède présente un anomalie, on ajoutera à *dico* l'affichage de u, $f(u)$, v, $f(v)$, m et $f(m)$ à chaque itération. Relancer le programme et expliquer le problème.

3 Méthode de Newton

On suppose, comme précédemment, que l'on dispose d'un intervalle $[a,b]$ sur lequel la fonction f est continue, strictement monotone et vérifie $f(a)f(b) < 0$. On sait alors que f admet un unique zéro r dans $]a,b[$. Ici, on suppose également que f est dérivable et que la dérivée de f ne s'annule pas.

On peut s'approcher de son zéro en « prenant la tangente ». La méthode de Newton aussi appelé **méthode de la tangente**.

3.1 Principe de la méthode de la tangente

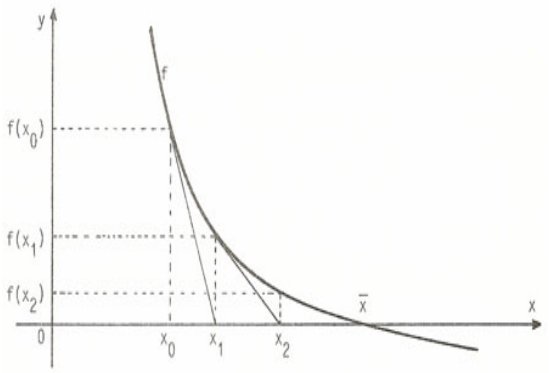


Figure 2 – Recherche du zéro d'une fonction par méthode de Newton

On commence par présenter le principe de la méthode sur la fonction f qui est présentée ci-dessus. Notons r le zéro de f dans l'intervalle considéré, x_0 un élément de cet intervalle.

On trace la tangente à la courbe représentative de f au point de la courbe d'abscisse x_0. Cette droite a une pente égale à $f'(x_0)$ et coupe l'axe des abscisses au point d'abscisse x_1. On remarque sur cet exemple que x_1 est plus proche que x_0 : x_1 est une meilleure approximation de r que x_0.

Exprimons x_1 en fonction de x_0, $f(x_0)$ et $f'(x_0)$:

- On sait que la tangente à la courbe de f au point d'abscisse x_0 a pour équation :
$$f(x) = f'(x_0)(x - x_0) + f(x_0)$$
- Par définition de x_1 :
$$0 = f'(x_0)(x_1 - x_0) + f(x_0)$$
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$
- On réitère l'opération précédente à partir de x_1, on obtient une seconde tangente qui coupe l'axe des abscisses au point d'abscisse x_2, qui est encore une meilleure approximation du zéro r.
De même à la $n^{\text{ème}}$ itération, on obtient
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On continue ce procédé jusqu'à ce que la différence entre deux approximations successives, x_n et x_{n+1}, soit très petite (inférieure à une précision e, que l'on choisit au départ). Le nombre x_{n+1} est alors une valeur approchée de r.
Cela revient à définir la suite donnée par la relation de récurrence :

$$U_{n+1} = U_n - \frac{f(U_n)}{f'(U_n)}$$

qui converge, sous certaines hypothèses, vers le zéro de f.

3.2 Exemple pratique

Calcul approché de $\sqrt{2}$. Pour obtenir une valeur approchée de 2, il suffit d'appliquer la méthode de Newton à la fonction f : $x \mapsto x^2 - 2$, en partant d'une valeur positive de x_0, par exemple $x_0 = 1$.

```
#Définition de la fonction f
def f(x):
        return x**2 - 2

#Définition de la dérivée de la fonction f
def fd(x):
        return 2*x

def newton(f, fprime, u0, eps = 1e-6):
    u = u0
    v = u - f(u) / fprime(u)
    k = 0
    while abs(v - u) > eps and k < 50:
        u = v # u = u(k)
        v = v - f(v) / fprime(v) # v = u(k+1)
        k += 1
    return v, k #k est le nombre d'itérations effectives.

#Initialisation des variables
e = 10**(-5) #précision
x0 = 1
r,n = newton(f, fd, x0, e)
print("Au bout de ",n," itérations, la valeur approchée obtenue est ",r)
```

L'exécution du script ci-dessus donne :
Au bout de 3 itérations, la valeur approchée obtenue est 1.4142135623746899
Si on change $x_0 = 6$ on trouve qu'au bout de 5 itérations, la valeur approchée obtenue est 1.4142135623732204

On remarque que plus x_0 est plus proche de la vraie valeur de la solution, plus le nombre d'itérations est petit, et donc plus le programme est rapide.

Exercice 4

Il s'agit de développer un programme Python de résolution numérique d'une équation par la méthode de Newton. On considère l'équation (E) suivante : $x - e\ sin(x) = 0$, il s'agit de déterminer la solution $\alpha \in [1.5;5]$ avec une précision $p = 10^{-4}$.
Travail démandé :

1. Écrire le programme Python permettant de résoudre numériquement l'équation (E).
2. Saisir l'expression de la fonction $f(x) = x - e\ sin(x)$ ainsi que les bornes a et b de l'intervalle d'étude et la précision p désirée.
3. Déterminer la solution $\alpha \in [1.5;5]$.

4 Déjà disponible en Python

Comme pour tous les algorithmes de ce chapitre, nous ne ferons que réimplémenter des fonctions déjà existantes en Python. Concernant les méthodes de résolution approchée d'équations, tout se trouve dans le module `scipy.optimize`, qui est lui-même un sous-module du (gros) module d'analyse numérique `scipy`. Il contient entre autres les fonctions suivantes :
Fonctions utiles du module `scipy.optimize` :

- `bisect(f,a,b)` : détermine une racine de f dans $[a,b]$ en effectuant une dichotomie.
- `newton(f,x0)` : détermine une racine par la méthode de Newton en partant de x_0

TP N°21 : Intégration numérique

1 Introduction

Dans cette partie, nous nous intéresserons aux algorithmes permettant le calcul numérique d'intégrales.

Le principe général commun aux méthodes que nous allons présenter est simple : découper l'intervalle d'intégration en petits morceaux, et approcher sur chacun de ces petits intervalles la courbe représentative de la fonction f par une courbe très simple pour laquelle le calcul d'aire est facile.

Étant donnés deux réels a et b tels que $a < b$ et une fonction f continue sur le segment $[a,b]$, on souhaite, à l'aide de suites, obtenir une valeur approchée de l'intégrale de f sur $[a,b]$.
On note $I(f) = \int_a^b f(t)dt$ l'intégrale à approcher.

Les valeurs de f peuvent provenir de la connaissance de la fonction, ou alors de la connaissance de valeurs de la fonction pour certaines valeurs de t (dans le cas, par exemple, de mesures régulières).
$\forall n \in \mathbb{N}^*$, on va partager le segment $[a,b]$ à l'aide d'une subdivision $(a_i)_{0\leq i\leq n}$ à pas constant égal à $\frac{b-a}{n}$ en posant

$$\forall i \in [0,n], a_i = a + i\frac{b-a}{n}$$

Sur Chaque segement de $[a_{i-1},a_i]$, on remplace f par une fonction dont on sait calculer l'intégrale.
Dans le cadre de notre cours, nous présentons deux méthodes :

- **La méthode des rectangles** où les fonctions utilisées pour approcher f sont constantes.
- **La méthode des trapèzes** où les fonctions utilisées pour approcher f sont affines.

2 La méthode des rectangles

Quoi de plus simple comme fonction qu'une fonction constante ? Et quoi de plus simple comme aire à calculer qu'une aire de rectangle ? La première méthode, que nous allons voir, consiste donc à approcher notre fonction sur chaque sous-intervalle par la fonction constate prenant la même valeur que f à gauche de l'intervalle.

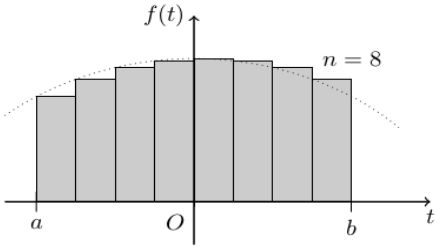
2.1 Définition de la suite associée à cette méthode

Sur Chaque segment de $[a_{i-1},a_i]$, on remplace f par la fonction constante de valeur $f(a_{i-1})$.

L'intégrale de f est alors approchée par la somme $R_n(f)$ des aires algébriques des rectangles de sommets $(a_{i-1},0)$, $(a_i,0)$, $(a_i,f(a_{i-1}))$ et $(a_{i-1},f(a_{i-1}))$.

Chaque rectangle a un coté de mesure (positive) $a_i - a_{i-1} = \frac{b-a}{n}$ et un autre coté de mesure algébrique $f(a_{i-1})$ de telle sorte que

$$\forall n \in \mathbb{N}^*, R_n(f) = \frac{b-a}{n}\sum_{i=0}^{n-1} f(x_i)$$



Intégration numérique : méthode de rectangles

2.2 Algorithme

```
def rectangle (f, a, b, n):
    """ valeur approchée de l'integrale de f sur [a,b]
    application de la methode des rectangles"""
    int_rect = 0
    h = (b-a)/n
    t = a
```

```
    for k in range(n):
        int_rect += h*f(t)
        t += h
    return int_rect
```

2.3 Exercices

Exercice 1

Écrire le script permettant de calculer numériquement $I = \int_1^5 x.ln(x)dx$

Solution : On utilise la méthode des rectangles

1. On découpe l'intervalle $[a,b] = [1,5]$ en N parties et on pose $h = \frac{b-a}{n}$.
   On obtient ainsi, $N+1$ points $t_i = a + i\frac{b-a}{N} = a + ih$ avec $i \in [0,N]$.
2. On construit N rectangles ;
   - L'aire du premier rectangle est égale à : $f(a) \times h = f(a + 0 \times h) \times h$
   - L'aire du second rectangle est égale à : $f(a+h) \times h = f(a + 1 \times h) \times h$
   - ...
   - L'aire du dernier (n^{eme}) rectangle est égale à : $f(a + (N-1) \times h) \times h$
3. On écrit la somme des aires :
$$\int_a^b f(x)dx \approx f(a).h + f(a+h).h + \ldots + f(a + (N-1).h).h$$
4. En simplifiant l'expression on obtient :
$$\int_a^b f(x)dx \approx h\left(f(a) + f(a+h) + \ldots + f(a + (N-1).h)\right) = h\sum_{i=0}^{N-1} f(a + i.h)$$
$$\int_a^b f(x)dx \approx h\sum_{i=0}^{N-1} f(t_i) \qquad avec \qquad t_i = a + ih$$
5. ```
   #On définit la fonction f
   def f(x):
           import numpy as np
           return x * np.log(x)

   #programme
   def rectangle (f, a, b, n):
       """ valeur approchée de l'integrale de f sur [a,b]
       application de la methode des rectangles"""
       int_rect = 0
       h = (b-a) / n
       t = a
       for k in range(n):
           int_rect += h * f(t)
           t += h
       return int_rect

   #Données
   a, b, n = 1, 5, 1000

   #Résultats numériques
   I=rectangle(f, a, b, n)
   print("Valeur approchée de I = ", I)
   ```
6. Valeur approchée de $I \approx 14.1018816722$
   ```

3 La méthode des trapèzes

Le principe général est exactement le même que celui de la méthode des rectangles, mais on approche cette fois-ci la courbe sur le segment $[a_i, a_{i+1}]$ par le segment de droite reliant les deux points de la courbe d'abscisses a_i et a_{i+1} , ce qui revient à calculer une somme d'aires de trapèzes pour approcher l'intégrale.

3.1 Définition de la suite associée à cette méthode

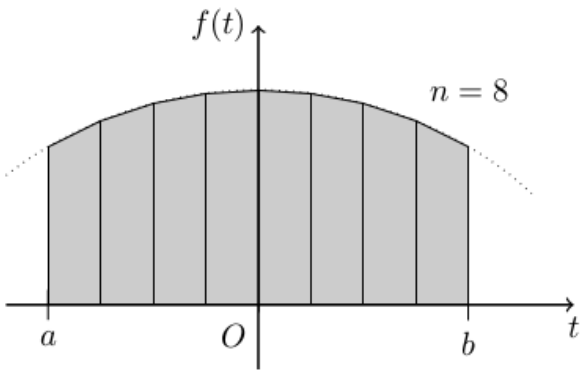
Sur chaque segment de $[a_{i-1}, a_i]$, on remplace f par la fonction affine qui coïncide avec f en a_{i-1} et en a_i.

L'intégrale de f est alors approchée par la somme $T_n(f)$ des aires algébriques des trapèzes de sommets $(a_{i-1}, 0)$, $(a_i, 0)$, $(a_i, f(a_i))$ et $(a_{i-1}, f(a_{i-1}))$.

Visuellement, l'impression est nettement meilleure que pour la méthode des rectangles.

Chaque trapèze a une aire algébrique qui vaut $\frac{b-a}{n}\frac{(f(a_i)+f(a_{i-1}))}{2}$ de telle sorte que

$$\forall n \in \mathbb{N}^*, T_n(f) = \frac{b-a}{2n}\sum_{i=1}^{n}(f(a_i)+f(a_{i-1}))$$



Intégration numérique : méthode de trapèzes

3.2 Algorithme

```python
def trapezes(f, a, b, n):
    h = (b - a) / n
    int_trapeze = 0.
    t = a
    for k in range(n):
        int_trapeze += h * (f(t) + f(t+h)) / 2
        t += h
    return int_trapeze
```

3.3 Exercices

Exercice 2

Écrire le script permettant de calculer numériquement $I = \int_1^5 x.ln(x)dx$

Solution : On utilise la méthode des trapèzes

1. On découpe l'intervalle $[a,b] = [1,5]$ en N parties et on pose $h = \frac{b-a}{n}$.
   On obtient ainsi, $N+1$ points $t_i = a + i\frac{b-a}{N} = a + ih$ avec $i \in [0, N]$.
2. On construit N trapèzes ;
   - L'aire du premier trapèze est égale à : $\frac{f(a)+f(a+h)}{2} \times h$
   - L'aire du second trapèze est égale à : $\frac{f(a+h)+f(a+2h)}{2} \times h$
   - ...
   - L'aire du dernier ($n^{\text{ème}}$) trapèze est égale à : $\frac{f(a+(N-1)h)+f(b)}{2} \times h$
3. On écrit la somme des aires :

$$\int_a^b f(x)dx \approx \frac{f(a)+f(a+h)}{2}.h + \frac{f(a+h)+f(a+2h)}{2}.h + \ldots + \frac{f(a+(N-1)h)+f(b)}{2}.h$$

4. En simplifiant l'expression on obtient :

$$\int_a^b f(x)dx \approx h \sum_{i=0}^{N-1} \left(\frac{f(t_i) + f(t_i + h)}{2} \right) \qquad avec \qquad t_i = a + ih$$

5.
```python
def trapezes(f, a, b, n):
    h = (b - a) / n
    int_trapeze = 0.
    t = a
    for k in range(n):
        int_trapeze += h * (f(t) + f(t+h)) / 2
        t += h
    return int_trapeze

#Résultats numériques
import numpy as np
f = lambda x : x * np.log(x)
a, b, n = 1, 5, 1000
print("I = ", trapezes(f, a, b, n))
```

6. Valeur approchée de $I \approx 14.1179760513$

4 Déjà disponible en Python

La fonction `scipy.integrate.trapz` permet de mettre en œuvre la méthode des trapèzes sans la programmer.
Si on peut/doit utiliser une fonction de bibliothèque, on privilégiera `scipy.integrate.quad`, plus rapide et performante que les méthodes du programme de notre classe.

TP N°22 : Résolution des équations différentielles du premier ordre

1 Introduction

Après les équations numériques, les *équations différentielles*. Dans ce cours, nous étudions les *équations différentielles linéaires du premier ordre (EDL1)*. Les équations différentielles ont un lien évident avec l'intégration numérique. C'est donc sans surprise que l'on va utiliser dans cette section des méthodes proches de celles de la précédente : découpage de l'intervalle de résolution en n morceaux, puis approximation de la solution sur chacun de ces intervalles.
C'est le principe de base de la *méthode d'Euler*, qui est la principale méthode de résolution numérique d'équations différentielles, sur laquelle nous allons concentrer tous nos efforts.
Dans la suite, nous allons étudier la méthode d'Euler, méthode numérique permettant d'obtenir une approximation de la solution d'un tel *problème de Cauchy*. Et on termine par des exemples d'utilisation de la fonction de résolution approchée d'équations différentielles *odeint*.

2 Définitions

Définition 1. Une **équation différentielle** est une équation qui relie une fonction f à une ou plusieurs de ses dérivées. Exemple : $f(x) = \alpha f'(x) + \beta f''(x)$.

Remarque 1. Pour simplifier l'écriture des équations différentielles, on remplace couramment $f(x)$ par y et $f'(x)$ par y'. Exemple : $f(x) = \alpha f'(x) + \beta$ devient $y = \alpha y' + \beta$.

Définition 2. Une **équation différentielle** est une équation dont l'inconnue est une fonction y (réelle ou complexe, même si nous traiterons surtout le cas réel dans ce chapitre), et faisant intervenir les dérivées successives $y', y'', ..., y^{(n)}$ de la fonction y. L'équation est dité **d'ordre** `n` si la dérivée d'ordre le plus élevé de y apparaissant dans l'équation est $y^{(n)}$.

Définition 3. Une équation différentielle est **linéaire** si elle s'écrit sous la forme :
$a_n y^{(n)} + a_{n-1} y^{(n-1)} + ... + a_1 y' + a_0 y = b$ où $a_0, a_1, ..., a_n$ et b sont des fonctions.

Définition 4. La fonction b est appelé le **second membre**. Si b est *nulle*, alors l'équation est dite **homogène** ou **sans second membre**. Si $a_0, a_1, ..., a_n$ sont des *constantes*, on parle d'équation différentielle linéaire à **coefficients constants**.

Définition 5. Une courbe intégrale associée à une équation différentielle est une courbe représentative de la fonction solution y de l'équation différentielle.

Remarque 2. Les équations différentielles considérées doivent être écrites sous la forme : $y' = F(y,t)$ où F est un fonction à deux variables y et t.
Par exemple, pour l'équation différentielle $y' + 2ty = 1$ on définit $F(y,t) = 1 - 2ty$ ce qui s'écrit avec *Python* :

```
def F(y, t):
        return 1-2*t*y
```

Remarque 3. Même si, comme c'est parfois le cas, on écrit l'équation sous la forme : $y'(t) + 2ty(t) = 1$ il ne faut surtout pas définir la fonction F en écrivant :

```
def F(y, t):
        return 1-2*t*y(t) #INCORRECT
```

Remarque 4. En mathématiques et en physique, l'habitude est plutôt de considérer les équations différentielles sous la forme $y' = F(t,y)$ et non $y' = F(y,t)$. Mais en Python, la fonction `odeint`, que nous verrons par la suite, utilise l'écriture $y' = F(y,t)$. Par souci de simplicité, nous nous conformerons à cette écriture dans ce cours.

Dans le cadre de notre cours on cherche à résoudre un problème numérique simple dit *problème de Cauchy*.

3 Cadre du problème de Cauchy

Un *problème de Cauchy* est un problème constitué d'une équation différentielle dont on recherche une solution vérifiant une certaine condition initiale. $\begin{cases} y' = F(t,y) \\ y(t_0) = y_0 \end{cases}$ où :

- y (est l'inconnue)est une fonction dérivable définie sur un intervalle I de $\mathbb{R}$;
- $t_0 \in I$;

On possède donc une équation différentielle et une condition initiale. Le théorème de Cauchy Lipschitz assure l'existence et l'unicité d'une solution à ce problème.

4 Méthode d'Euler

La méthode d'*Euler* est une procédure numérique pour résoudre par approximation des équations différentielles du premier ordre avec une condition initiale. C'est la plus simple des méthodes de résolution numérique des équations différentielles.

On considère une équation différentielle d'ordre 1 avec condition initiale (problème de Cauchy) : $\begin{cases} y' = F(y,t) \\ y(t_0) = y_0 \end{cases}$

On cherche à résoudre le problème de Cauchy sur un intervalle $I = [a,b]$.
Soit $n \in \mathbb{N}^*$, on subdivise l'intervalle $[a,b]$ en n petits segments de même longueur $h = \frac{b-a}{n}$.
On pose alors pour $0 \leq k \leq n$: $t_0 = a \quad t_1 = t_0 + h \quad t_2 = t_0 + 2h \quad ... \quad t_k = a + kh \quad ... \quad t_n = t_0 + nh = b$
On part de t_0 . On approche le petit morceau de courbe entre t_0 et t_1 par la tangente à la courbe au point d'abscisse t_0 .
La méthode d'Euler sert à déterminer approximativement les valeurs prises par y aux instants $t_0, t_1, ..., t_{n-1}$.
Nous utiliserons les notations suivantes :

- $y(t_0), y(t_1), ..., y(t_{n-1})$ sont les valeurs exactes (on connaît seulement $y(t_0)$) ;
- $y_0, y_1, ..., y_{n-1}$ sont les valeurs approchées fournies par la méthode d'Euler.

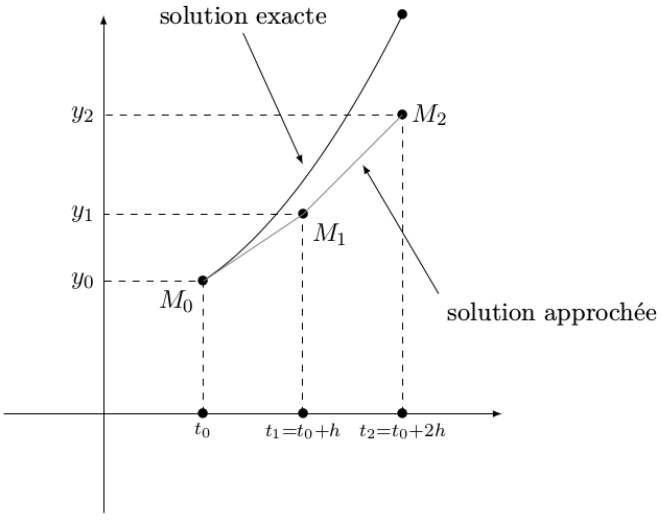
On a alors :

$$\begin{aligned} \frac{y(t_1) - y(t_0)}{h} &\approx y'(t_0) \\ &\approx F(y(t_0), t_0) \\ y(t_1) &\approx y_0 + hF(y(t_0), t_0) \end{aligned}$$

On pose : $y_1 = y_0 + hF(y_0, t_0)$. Le nombre y_1 est une approximation de la valeur exacte $y(t_1)$.
On recommence le procédé à partir du point $M_1(t_1, y_1)$...
De manière générale, on pose :

$$\forall k \in 0, \ldots, n-1, y_{k+1} = y_k + hF(y_k, t_k)$$

Ce qui amène à construire successivement les points $M_k(t_k, y_k)$.
La ligne polygonale reliant ces points est alors une approximation de la courbe représentative de la solution.



Cette méthode de résolution approchée, la seule à notre programme, s'appelle la méthode d'*Euler*.
Il existe de nombreuses autres méthodes qui peuvent être proposées dans le cadre d'un problème.

4.1 Comment programmer la méthode d'Euler?

Dans ce programme, on donne la valeur du pas h et on vérifie à chaque itération que $a + t_k h \leq b$.

```
def euler(F, a, b, y0, h=1e-2):
  """Fournit une solution approchée à l'équation y'=F(y,t), y(a)=y0
  sur l'intervalle [a,b] par la méthode d'Euler, le pas utilisé est h, par défaut 0.01
  La solution renvoyée est une liste de valeurs pour les_t et une liste de valeurs pour les_y"""
        y = y0
        t = a
        les_y = [y0] # la liste des valeurs renvoyées
        les_t = [a]
        while t+h <= b:
                #les_y = [y_0,...,y_k], les_t = [t_0,...,t_k]
                y += h * F(y, t)
                les_y.append(y)
                t += h
                les_t.append(t)
                # à la fin de la boucle, t contient a+nh ≈ b
        return les_t, les_y
```

Remarque 5. Si le nombre de subdivisions n nous est donné à la place de h, on pourra préférer écrire la fonction avec une boucle `while`, en utilisant la fonction `linspace` du module `numpy`.

4.2 Exemple Python

On considère l'équation $y' = y$ avec la condition initiale $y(0) = 1$. La solution est la fonction exponentielle, ce qui permet de comparer les résultats obtenus.
On va travailler sur l'intervalle $[0,1]$ en considérant tout d'abord un pas de 0.25 puis un pas de 0.1. La fonction exponentielle est représentée en pointillés pour comparer. F doit être définie comme une fonction de deux variables même si ici elle ne dépend que de y.

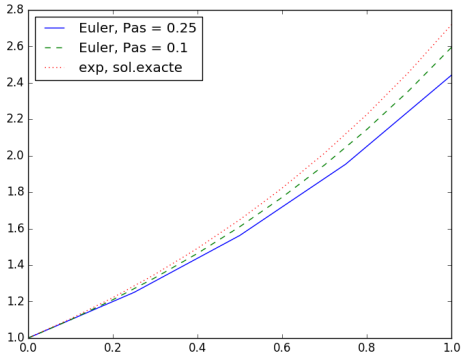
```
import numpy as np
import matplotlib.pyplot as plt
def F(y,t):
        return y

t,y = euler(F,0,1,1,0.25)
#ligne bleue continue
plt.plot(t,y,'b-',label='Euler, Pas = 0.25')

t,y = euler(F,0,1,1,0.1)
#ligne verte discontinue
plt.plot(t,y,'g--',label='Euler, Pas = 0.1')

t = np.linspace(0,1,11)
#ligne rouge pointillée
plt.plot(t,np.exp(t),'r:',label='exp,sol.exacte')

plt.legend(loc='upper left')
plt.show()
```



Exercice 1

On considère le *problème de Cauchy* : $\begin{cases} \dfrac{dy}{dt} = -ty + cos(t) \\ y(0) = 1 \end{cases}$
Représenter graphiquement la solution de ce problème sur l'intervalle $[0,5]$.

5 La fonction `odeint` du package `scipy.integrate`

La bibliothèque `scipy` fournit une fonction de résolution approchée et automatique des équations différentielles ordinaires. Il s'agit de la fonction `odeint`, acronyme de `ordinary differential equations integrating`. Pour pouvoir l'utiliser, il faut d'abord l'importer :

```
from scipy.integrate import odeint
```

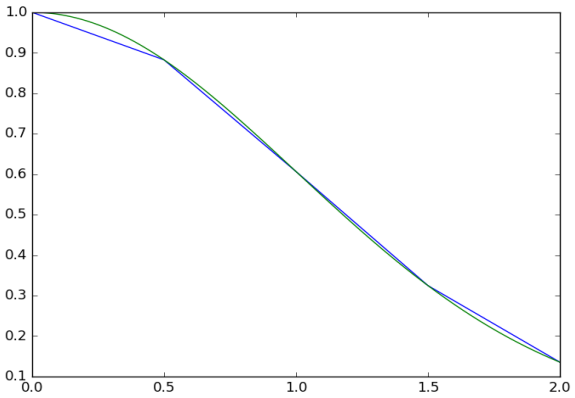
On appelle `odeint` sous la forme `odeint(F,y0,t)` où F est la fonction qui intervient dans l'équation différentielle, y_0 est la valeur initiale et t est le vecteur contenant les valeurs $[t_0 = a, t_1, ..., t_n = b]$.
Après avoir défini les objets de l'équation que l'on veut résoudre, on demande la résolution de l'équation différentielle au moyen de la commande suivante : `y = odeint (F , y0 , t )`
Le résultat y est un tableau des approximations de $y(t)$ correspondant aux différents instants du tableau t.

6 Exemples de résolution approchée d'ED1O.

Exemple 1. On considère l'équation différentielle d'ordre 1 avec condition initiale : $\begin{cases} \frac{dy}{dt} = -ty \\ y(0) = 1 \end{cases}$
On veut calculer des valeurs approchées de y entre 0 et 2 pour 5 valeurs régulièrement espacées.
Remarque : Le tracé est superposé avec celui de la fonction $t \longmapsto exp(-t^2/2)$.

```
#ED
def F(y,t):
        return -t*y
t=np.linspace(0,2,5)
y=odeint(F,1,t)

#exp
exp_t = np.linspace(0,t[4],1000)
exp_y = np.exp(-exp_t**2/2)
plt.plot(t,y,exp_t,exp_y)
plt.show()
```



Exemple 2. On veut résoudre sur l'intervalle [0, 5] le problème de Cauchy : $\begin{cases} y' + 2ty = 1 \\ y(0) = 1 \end{cases}$

```
from math import *
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
def F(y, t): # La fonction F associée à l'équation différentielle
    return 1-2*t*y
t = np.linspace(0,5,100) # Le vecteur t qui représente l'intervalle de temps considéré
y = odeint(F,1,t) # Le calcul du vecteur y avec la commande odeint
# Représentation graphique
plt.clf();plt.plot(t,y, 'b-');plt.xlabel("x"); plt.ylabel("y"); plt.show()
```

Exercice 2: Problème de cinétique chimique

On s'intéresse à une réaction chimique faisant disparaître un réactif A d'une solution. On suppose que la vitesse de disparition de A est proportionnelle à sa concentration (réaction d'ordre 1) : $\frac{d[A]}{dt} = -\alpha[A]$ où α est une constante s'exprimant en s^{-1} (la valeur $\frac{ln2}{\alpha}$ est un temps appelé temps de demi-réaction).
L'équation est bien de la forme $[A]' = F(t,[A])$ où $F : (t,y) \longmapsto -\alpha y$
On suppose $\alpha = 1s^{-1}$ et au temps $t = 0$, $[A] = 1mol/l$.
Tracer la courbe représentative de l'évolution de la vitesse de disparition de A jusqu'à $t = 6s$.

Quelques conseils pour la suite :

- **Ne craignez plus les erreurs :** Un programme qui ne marche pas du premier coup n'est pas un échec, c'est juste une étape de réflexion.
- **Gardez la curiosité :** Python est un outil incroyable. Ce que vous avez appris ici vous servira dans tous vos futurs projets, bien au-delà des concours.
- **Pratiquez régulièrement :** La logique de programmation est comme un muscle, elle s'entretient !

Si vous avez des questions ou besoin d'éclaircissements, n'hésitez pas à me contacter par e-mail :
anis_saied@hotmail.com

Toutes les ressources et les corrections détaillées sont disponibles sur :
https://anis-saied.com

Fin du manuel – IPEI Nabeul