CHAPITRE I
Rappels Python

Vous trouverez les documents, cours TP, corrections et annales sur le site :
https ://padlet.com/jbyvernault/ipt-decours-mp-psi-ggkcw07mqi1rv3xf

1 Le langage Python.

1.1 Lien entre langage et algorithme.

Un algorithme est un enchaînement d'instruction souvent répétitif qui permet de résoudre un problème, souvent numérique mais que l'on utilise aussi en géométrie dans les programmes de construction, en cuisine pour la réalisation d'une recette, dans les jeux de construction en suivant le schéma d'assemblage.
On utilise un ordinateur qui manipule essentiellement des nombres et... ne fait que ce qu'on lui demande. On utilise ses deux points forts :

- sa mémoire (qui permet de faire des opérations avec de nombreuses valeurs)
- sa capacité à répéter rapidement sans erreur ou presque et sans se lasser... les même opérations.

On traduit alors l'algorithme sous la forme d'un langage qui permet à l'homme de communiquer avec la machine. On parle alors de programme et nous utiliserons le langage **Python** [1] qui est un langage structuré (toute référence aux *Monty Python* [2] ne serait pas fortuite).

1.2 Démarrer Python.

Le **Python** est un langage sophistiqué dit orienté objet (voir plus loin).
Pour tous les langages, les instructions sont :

- des séquences(enchainement d'instructions successives),
- des tests (« si... alors... sinon »),
- des boucles (répétition d'une séquence),
- des instructions d'entrée de données,
- des instructions de sorties (à l'écran, sur un automate, ...)

Pour programmer en **Python**, le plus simple est d'utiliser un **IDLE** (*Integrated DeveLopment Environment*, environnement de développement intégré) qui possède :

- un éditeur de texte, pour charger, taper, enregistrer ses programmes ;
- un terminal, pour taper directement ses commandes ;
- un *débogueur*, pour compiler (vérifier) le code et l'exécuter.

Remarque. Dans toute la suite, le code qui est tapé dans le terminal sera précédé (comme à l'écran) de « >>> ».
Le résultat de l'exécution des instructions tapées dans le terminal sont entre deux lignes commençant par « >>> ».

1.3 Programmer en Python

En **Python**, l'écriture d'un programme tient compte de la mise en page : une série d'instructions à exécuter en bloc sera placée sur un même alignement ; le début du bloc étant signalé à la ligne précédente par des « : » finaux.

1. https://www.python.org
2. http://pythonline.com/

Remarque. Comme tout langage évolué, le **Python** permet l'ajout de commentaires à la fin des lignes de programme. Ils sont précédés du symbole `#`. Attention, tout ce qui se trouve après ce symbole est en commentaire et ne sera donc pas lu par la machine lors de l'exécution.
L'ajout de commentaires placés après le symbole `#` permettent de facilement analyser, relire, comprendre et donc modifier le programme.

1.4 Les bases du Python.

1.4.1 Variables.

Une variable désigne une zone mémoire à laquelle on affecte une valeur (numérique, caractère, chaîne de caractères, ...) à l'aide de la commande `=`.
Sous **Python**, les noms de variables doivent en outre obéir à quelques règles simples :

- un nom de variable est une séquence de lettres (a ... z, A ... Z) et de chiffres (0 ... 9), qui doit toujours commencer par une lettre,
- seules les lettres ordinaires sont autorisées ; les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, , etc. sont interdits, à l'exception du caractère _ (souligné),
- la casse est significative (les caractères majuscules et minuscules sont distingués) ; ainsi `Ipt`, `ipt` et `IPT` sont donc des variables différentes.

L'habitude veut que l'on écrive les noms de variables en caractères minuscules (y compris la première lettre) ; il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans `uneVariableDontJAimeBienLeNom`, par exemple.
En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les « mots réservés » utilisés par le langage.

Remarque. Les noms commençant par une majuscule ne sont pas interdits, mais l'usage veut qu'on le réserve plutôt aux variables qui désignent des classes (le concept de classe sera abordé plus loin dans ces notes).

On peut alors effectuer des opérations simples et modifier cette variable comme on le souhaite :

```
>>> x=3
>>> 2*x
6
>>> x=5-1
>>> x
4
>>> x=x+3 # la nouvelle valeur de la variable x est 3 de plus que l'ancienne valeur
>>> x
7
>>> 4+2 # ceci étant un commentaire, le calcul 6+3 n'apparait pas
6
>>> x="ceci est un texte"
>>> x
'ceci est un texte'
>>>
```

Remarque. Dans tout le texte, une espace dans une chaîne de caractères est codée par `" "`.

1.4.2 Entrées et sortie de données.

L'affichage à l'écran se fait à l'aide de l'instruction `print()` ; il est possible d'afficher le résultat d'un calcul, un texte ou une variable.

```
print(3+2)
print(3*2)
print("un texte")
print("un texte avec l'apostrophe")
print('un "texte entre guillemets".')
```

Pour faire saisir des valeurs à l'utilisateur, on utilise la commande `input()`, qui interroge l'utilisateur et fait une « pause » en attendant l'entrée d'une donnée.

```
a=input()
print(a)
```

ou dans le terminal :

```
>>> a=int(input()) # on saisit 25
>>> a
'25'
>>>
```

On peut aussi, pour plus de clarté, ajouter du texte dans la commande `input()`. Qui apparaitra au moment où le programme s'exécute.

```
>>> a=int(input("Entrez la valeur de a : "))
Entrez la valeur de a : 3
>>> a
'3'
```

Remarque. Lors de l'utilisation de la commande `input()`, la variable est stockée comme chaîne de caractères. Il s'agit donc d'un texte !

```
>>> a+2
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Il faut donc « typer » les variables.

Pour voir afficher la valeur ou le contenu d'une grandeur (exclusivement au milieu d'un programme) et voir s'afficher le résultat attendu on utilise la commande `print()` :

```
>>>print(a)
3
>>>print("coucou")
coucou
```

1.5 Les nombres.

On peut travailler avec des entiers (très précis et bien pratique) ou des flottants (nombres à virgule flottante puisque l'on peut déplacer la virgule en modifiant la puissance de 10 de la notation scientifique).

1.5.1 Les entiers.

Les entiers sont les bons amis des ordinateurs, leur manipulations (additions, multiplications, puissances, quotients, ...) sont sans erreurs d'arrondi et préférables. Les entiers, de type `int` ne sont limités en taille que par la mémoire de la machine !!

Exemple. Si les calculs sont effectués sur 16 bits (c'est-à-dire 16 interrupteurs prenant les valeurs 0 ou 1), on obtient des nombres de 0 à $2^{16} - 1 = 65535$: la première moitié de 0 à 32767 pour les positifs et la deuxième pour les négatifs (-1 correspond à 65535 car son suivant est 0). Les positifs ont leur premier bit nul. En pratique, on se limite aux codages sur 32 bits ou 64 bits.

Pour les entiers relatifs, c'est généralement la représentation en complément à 2 qui est utilisée :

$$\forall i : a_i \in \{0,1\} \quad 0a_n \dots a_0 \text{ représente } \sum_{k=0}^{n} a_k 2^k$$

$$\forall i : a_i \in \{0,1\} \quad 1a_n \dots a_0 \text{ représente } -2^n + \sum_{k=0}^{n} a_k 2^k$$

1.5.2 Les flottants.

Les flottants, de type `float`, sont presque toujours des valeurs approchées et codés en binaire (sauf dans le cas de nombres dyadiques du type $\sum_{k\in J\subset\mathbb{Z}} a_k 2^k$). Ils sont codés en binaire et de la forme : $s \times m \times 2^e$ où :

- s est codé sur un bit, le 63^{e} (0 pour les positifs et 1 pour les négatifs) ;
- m, la **mantisse** est codée sur 52 bits, du numéro 0 au numéro 51 ;
- $e + 1\,023$, l'exposant *décalé* est codé sur 11 bits, du numéro 52 au numéro 63, on trouve donc e entre -1 023 et 1 024 mais ces deux valeurs extrêmes sont réservées pour 0 et ∞. L'exposant e se situe donc entre -1 022 et 1 023

A noter, que les nombre décimaux (par exemple 0.1) ne sont pas forcément dyadique et leur représentation est donc une approximation qui peut générer des errurs. La mantisse est nécessairement un nombre dont le chiffre avant la virgule est un 1 !! Donc il n'est pas représenté, ce qui fait gagner un bit pour la représentation. Quelques petits problème de précision :

```
>>>2+2**-54-2
>>>0.0
```

L'addition de flottants n'est pas associative !!
En **Python**, pour faire la différence entre un flottant et un entier, il suffit d'écrire l'entier sous forme décimale avec la virgule . :

```
>>> type(1)
<class 'int'>
>>> type(1.)
<class 'float'>
>>>
```

1.5.3 Saisie d'un nombre.

Pour forcer la saisie d'un entier on utilise `int(input())` et `float(input())` pour des flottants.

```
>>> n=int(input()) # on saisit 25
>>> n
25
>>> n=int(input()) # on saisit 2.5
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2.5'
>>> x=float(input("saisir un réel :")) # on saisit 2
>>> type(x)
<class 'float'>
>>> x
2.0
>>>
```

1.5.4 Opérations numériques.

L'addition `+`, la multiplication `*`, la soustraction et le changement de signe `-`, la division réelle `/`, le parenthésage `(...)` et l'exponentiation `a**b` pour a^b, s'emploient normalement et produisent les résultats que l'on attend en respectant la priorité des opérations.
On peut ajouter aux opérations de base la valeur absolue `abs(...)`, la troncature à l'unité `int(...)` et l'arrondi à 10^{-n} `round(...,n)`.
Il faut compléter cette liste par les fonctions d'entiers à valeurs entières que sont le quotient `//` et le reste positif de la division euclidienne `%`.

Remarque. La machine respecte la priorité des opérations.

Exemple.

```
>>> 2*3
6
>>> 2*3.
6.0
>>> 2.*3
6.0
>>> 2.*3.
6.0
>>> 2/3
0.6666666666666666
>>> 3/2
1.5
>>> 2/3
0.6666666666666666
>>> 25//7 # car 25=3*7+4
3
>>> 25%7
4
>>> 4.0//2.0
2.0
>>> 4.0//3.0 # tiens, il s'agit de la
        troncature de la division !!!
1.0
>>> int(4.2)
4
>>> int(-4.2) # il ne s'agit donc pas de la
         partie entière !!!
-4
>>> round(4.2)
4
>>> round(-4.2)
-4
>>> round(-4.7)
-5
>>> round(4.7)
5
>>>
```

```
>>> 2**3
8
>>> 2.**3 # pour jouer un peu avec les
        entiers et les flottants
8.0
>>> 2**3.
8.0
>>> -1**5
-1
>>> -1**-2
-1.0
>>> -2**-2
-0.25
>>> (-2)**-2 # attention aux parenthèses
0.25
>>> 2**3+1
9
>>> 2**(3+1) # attention aux parenthèses (
        je l'ai déjà dit, non ?)
16
>>> 0**2
0
>>> 1**0
1
>>> 0**0 # et un résultat curieux, un !
1
>>> 4**0.5 # la racine d'un carre entier n'
        est pas un entier...
2.0
>>> (-1)**0.5 # en voila une drôle d'idée !
(6.123233995736766e-17+1j)
>>> type((-1)**0.5) # on s'attend un peu à
        la réponse... non ?
<class 'complex'>
>>> abs(1+1j)
1.4142135623730951
>>> 1+j # le langage Python est très
        exigeant...
Traceback (most recent call last):
  File "<interactive input>", line 1, in <
        module>
NameError: name 'j' is not defined
>>> 1+1j #... quant a la syntaxe.
(1+1j)
>>>
```

Attention aux priorités opératoires qui peuvent conduire à des résultats aberrants suivant la précision des calculs :

```
>>> 2**-54
5.551115123125783e-17
>>> 2-2+2**-54
5.551115123125783e-17
>>> 2+2**-54-2
0.0
>>> 2**-54+2-2
0.0
>>>
```

Remarque. Les résultats des différents calculs peuvent dépendre des types valeurs de départ, en effet la somme, le produit ou le quotient d'un complexe par une autre complexe ou un flottant ou un entier renverra un complexe ; de même la somme, le produit ou le quotient d'un flottant par un autre flottant ou un entier renverra un flottant.

Remarque. Avec les puissances, il est facile de déterminer que le domaine des flottants :

```
>>> 10.**308
1e+308
>>> 10.**309
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
OverflowError: (34, 'Result too large')
>>> 1e+308
1e+308
>>> 1e+309
inf
>>> 1.7976931348623157e+308
1.7976931348623157e+308
>>> 1.79769313486231579e+308
1.7976931348623157e+308
>>> 1.7976931348623158e+308
1.7976931348623157e+308
>>> 1.797693134862316e+308
inf
>>> 1e-323
1e-323
>>> 1e-324
0.0
>>>
```

Remarque. La fonction factorielle et les combinaisons ne sont pas prédéfinies.

Remarque. Pour calculer $\min(x, y)$, on peut utiliser sans faire de test : `((x+y)/2-abs(y-x))/2`. En effet, si x et y sont vus comme les extrémités d'un segment, le minimum est celui qui se trouve à la moitié de la longueur du segment avant le milieu de ce segment. De même, pour calculer $\max(x, y)$, on peut utiliser sans faire de test : `((x+y)/2+abs(y-x))/2`.

1.6 Les variables booléennes.

Il s'agit des variables `True` et `False` dont les principaux opérateurs sont `and` (multiplication logique), `or` (addition logique) et `not` (inversion logique) :

```
>>> not True
False
>>> (True or False) and False
False
```

```
>>> True or False and False # par convention, et par soucis de cohérence, le "and" est
          prioritaire sur le "or"
True
>>> True or (False and False)
True
>>>
```

Ou pour vérifier si une relation (d'égalité ==, de non égalité !=, d'ordre <, >, <=, >=) est vraie :

```
>>> 1==3-2
True
>>> 2.!=2 # un flottant peut être égal à un entier
False
```

2 Outils de programmation.

2.1 Types de variables complexes.

2.1.1 Les listes.

Une liste de taille n est un n-uplet (une suite, un vecteur) numérique ordonné. Concrètement, il s'agit d'une adresse à laquelle est attachée une suite de valeurs numérotées de 0 à $n-1$; sa longueur étant n. Plusieurs opérations et fonctions peuvent être appliquées sur les listes : la longueur `len(...)`, la concaténation `+` et la suppression de valeurs `del(...)`

```
>>> L=[2,3,4,7]
>>> L[1] # la valeur d'indice 1 de la liste... donc la deuxième valeur
3
>>> len(L)# la longueur de la liste
4
>>> [3,4]+L # on parle de concaténation
[3,4,2,3,4,7]
>>> L=2*L
>>> L
[2,3,4,7,2,3,4,7]
>>> del L[2:5] # efface de la liste les valeurs d'indices 2 inclus à 5 exclu
>>> L
[2, 3, 3, 4, 7]
>>> L=[1,2,3,6,7]
>>> L[1:4] # sous liste de L d'indices de 1 inclus à 4 exclu
[2,3,6]
>>> L=[1,2,3,4,5]
>>> L=L[0:2]+L[3:5] # On retire la valeur d'indice 2
>>> L=[1,2,4,5]
>>> 2 in L
True
>>> 7 in L
False
>>> a,b,c,d=L
>>> a
1
>>> d
5
>>> L[0]=3 # Modifier la première valeur de la liste
>>>L
[3,2,4,5]
```

Remarque. Attention on ne manipule pas des listes comme les autres variables :

```
>>> a=2
>>> b=a
>>> a=a+2
>>> b # la valeur de b n'a pas changé
2
>>> a=[1,2]
>>> b=a
>>> a[0]=3
>>> b # la variable b suit les changements de a
[3, 2]
>>> a=[1]
>>> b # mais pas tous....
[3,2]
>>> a=[1,2] # heureusement, on peut s'en sortir
>>> b=list(a)
>>> a[1]=3
>>> a
[1, 3]
>>> b
[1, 2]
>>>
```

Pour construire une liste de termes non affectés, on utilise `None` :

```
>>> L=5*[None] # On répète par concaténation 5 fois [None]
>>> L
[None, None, None, None, None]
```

Attention les liste ne se copies pas naturelement :

```
>>>A=[1,2,3]
>>>B=A
>>>A[2]=45
>>>A,B
        ([1, 2, 45], [1, 2, 45])
```

On utilisera `B.copy()` :

```
>>>B=A.copy()
>>>A[2]=-56
>>>A
Out[15]: [1, 2, -56]
>>>B
Out[16]: [1, 2, 45]
```

2.1.2 Les tuples.

C'est un peu la même chose mais en plus rigide :

```
>>> L=(1,2,3)
>>> L[1]
2
>>> L[1]=3 # on ne peut pas la modifer
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> M=list(L)
>>> M
[1, 2, 3]
>>> M[1]=3
>>> M
[1, 3, 3]
>>> L=tuple(M)
>>> L
(1, 3, 3)
>>> L=(1,2,9)
>>>a,b,c=L
>>>c
Out[22]: 9
```

2.1.3 Les chaînes.

Les chaines se comporte comme des listes mais portent sur du texte, c'est une liste de caractères délimitée par "..." ou par '...'. On peut effectuer les mêmes opérations que sur les listes.

```
>>> s='bonjour'
>>> s[2]
'n'
>>> s='\''
>>> s
"'"
>>> s="\""
>>> s
'"'
>>> s="un texte qui comporte 'une citation' entre apostrophes"+' et "une autre" entre
        guillemets.'
>>> s
'un texte qui comporte \'une citation\' entre apostrophes et "une autre" entre
        guillemets.'
>>>
```

2.1.4 Les dictionnaires.

Un dictionnaire c'est un peu comme une liste mais indicé par des clés :

```
>>> mon_dictionnaire={"mot de passe"‘‘:123,"identifiant":"Bozo"}# "123" et "Bozo" sont
        indexé par "mot de passe" et "identifiant"
>>> mon_dictionnaire[mot de passe]
123
>>> mon_dictionnaire["mot de passe"]=456 # change le mot de passe
>>> mon_dictionnaire['mot de passe']
456
```

On peut utiliser ce qu'on veut comme clé, une chaine, un tuple, ... sauf une liste Pour effacer une clé, on utilise del

```
>>> del mon_dictionnaire["mot de passe"]
```

2.1.5 Les tableaux .

Un tableau se définie comme une liste de listes, plus précisément comme une liste de lignes.

Exemple. La matrice : $L = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ est `L=[[1,2],[3,4]]`. En effet :

```
>>> L=[[1,2],[3,4]]
>>> L[0][1] # coefficient 1ère ligne, 2ème colonne
2
>>> L[0] # deuxième ligne
[1,2]
>>>L[:][1] # deuxième colonne
[2,4]
```

Remarque. Comme un tableau est une liste de listes, on retrouve tous les travers des listes :

```
>>> v=[1,2]
>>> L=[v,v] # Les colonnes de L sont défini par une même adresse v
>>> L
[[1,2],[1,2]]
>>> v[0]=2 # On modifie une valeur de v
>>> L
[[2,2],[2,2]] # On modifie toutes les colonnes de L
```

Copier une matrice, ou créer une matrice, demande donc un programme spécifique ou .... utiliser une commande toute faites :

```
>>>A=L.copy()
```

2.2 Numpy et les matrices

Dans le module numpy on peut créer des matrices avec lesquelles on peut faire les opérations algébriques classique mais pas de concaténation :

```
>>>import numpy as np
>>> A=np.array([[1,2,3],[2,4,5],[1,2,5]])# est une matrice 3 3
>>>2*A
array([[ 2,  4,  6],
       [ 4,  8, 10],
       [ 2,  4, 10]])
>>>B=np.zeros((3,3)) # creer une matrice 3 3 de zéros
>>>B
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> B[0,1]=3 # affecter 3 à ligne 1 colonne 2
>>>B
array([[ 0.,  3.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>>B+3*A
array([[  3.,   9.,   9.],
       [  6.,  12.,  15.],
       [  3.,   6.,  15.]])
>>>A[1]
array([2, 4, 5])
>>>A[0,1]
```

```
2
>>>A[:,1] # colonne 2
array([2, 4, 2])
>>>np.dot(A,B) # produit AB
array([[ 0.,  3.,  0.],
       [ 0.,  6.,  0.],
       [ 0.,  3.,  0.]])
```

2.3 Pile

Une pile est un moyen d'ordonner le stockage de données en ... empilant celles ci. En gros, on classe les données par ordre d'apparition (ou d'utilisation) comme on peut le faire sur un traitement de texte en utilisant la touche *undo*... C'est le principe d'une file d'attente : comme quand des fichiers attendent bien sagement d'être imprimés, dans leur ordre d'arrivée !
On va alors apprendre à créer des piles, vider des piles empiler et dépiler...

```
>>>[] # creer une pile
>>>p.pop() # retire  le dernier terme de la liste et le retourne
>>>p.append(v) # ajoute v comme dernier terme dans la liste
>>>len(p) # retourne la taille de la pile
>>>p[-1]# retourne le sommet de la pile.
```

On peut aussi travailler avec des piles finies :

```
>>>P=[None]*5 #créer une pile finies
```

On peut dépiler et empiler dans une autre pile :

```
P2=P2.append(P1.pop())
```

Plus généralement... tout dépiler :

```
def depiler_empiler(P,Q):
""" Depile P et l'empiler dans Q"""
   while len(P)>0:
        Q=Q.append(P.pop())
```

On peut dépiler et empiler à gauche....

```
def depiler_gauche(P):
      Q=P[0]
      del P[0]
      return Q
```

Sauf qu'en python c'est une opération couteuse. Pour ça on importe `deque()` du module `collections` :

```
>>>from collections import deque
>>>A=deque()
>>>A
      deque([])
>>>B=deque([1,2,3])
>>>B
       deque([1, 2, 3])
>>>B.popleft()
     1
>>>B
     deque([2,3])
>>>B.appendleft('a')
```

```
>>>B
      deque(['a', 2, 3])
```

2.3.1 Passage d'un type à l'autre.

Tout ce qui suit est à utiliser avec la plus grande prudence, si vous manipulez ces commandes, les auteurs de ce cours nieront vous avoir informé de leurs existences...

```
>>> int(12.5)
12
>>> int(2)
2
>>> float(2.5)
2.5
>>> str(2.5)
'2.5'
>>> float(2)
2.0
>>> str(2)
'2'
>>> float('2')
2.0
>>> float('2.5')
2.5
```

```
>>> int('2')
2
>>> list('2.5')
['2', '.', '5']
>>> tuple('2.5')
('2', '.', '5')
>>> tuple([1,2])
(1, 2)
>>> list((2,5))
[2, 5]
>>> str([2,5])
'[2, 5]'
>>> str((2,5))
'(2, 5)'
>>>
```

2.4 Les tests.

La structure de test « if-then-else » est très simplifiée en **Python**, ce qui mérite une attention particulière.
On utilise : pour signaler le début d'un bloc d'instructions qui est indenté par rapport à la ligne qui l'appelle :

```
if delta<0:
    print("pas de solution")
else:
    if delta=0:
        print(-b/2/a)
    else:
        ...
        ...
```

que l'on peut écrire :

```
if delta<0:
    print("pas de solution")
elif delta=0:
    print(-b/2/a)
else:
    ...
    ...
```

car `elif` est la contraction de `else: if`

2.5 Les boucles.

Il s'agit de structures qui permettent de répéter autant de fois qu'on le veut ou tant qu'une condition est vérifiée (ou pas, avec la négation) une série d'instructions.

2.5.1 Boucle finie.

Ici, on connaît le nombre de répétitions. Appelons `i` l'indice correspondant au numéro de la répétition. La variable `i` appartient à un itérable numérique (liste, chaîne, n-uplet, ...).
Plus généralement, on utilise la commande `range(n)` qui donne les n premiers entiers naturels (donc numérotés de 0 à $n-1$) ou, plus précisément, la commande `range(m,n,p)` ou `m` est la valeur de départ (optionnelle, qui vaut 0 si omise), `n` la valeur finale (obligatoire !) et `p` le pas (qui vaut 1 si omis).

```
for i in range(3):
    print(i)
for i in range(3,13,3):
    print(i)
for i in range(100,90,-1):
    print(i)
```

renvoie :

0 1 2

3 6 9 12

100 99 98 97 96 95 94 93 92 91

On peut créer une liste :

```
>>> x=[i**2 for i in range(10)]
>>> x
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

ou sommer les termes d'une liste :

```
>>> sum(x[i] for i in range(4))
14
>>>
```

2.5.2 Boucle conditionnelle.

Il s'agit ici de répéter une suite d'instruction tant qu'une condition est vérifiée... ce qui peut (la machine faisant une confiance aveugle à son programmeur tout-puissant) mal tourner ou pour être plus exact tourner indéfiniment !! Il faut donc choisir une grandeur testée si possible décroissante et éviter les tests d'égalité à cause des approximation de calcul pour leur préférer des encadrements de plus en plus restreints.

Exemple. Pour obtenir le plus petit entier n tel que $2^n \geqslant 10\,000$:

```
p=1
while p<10000:
    p=p*2 # on calcul les 2 puissance n
    n=n+1 # n augmente de 1 à chaque boucle et on recommence
print(n) # on affiche le plus petit entier tel que 2 puissance n dépasse 10000
```

Pour vérifier si le calcul fonctionne, il est bon de chercher un invariant (une grandeur qui ne bouge pas). On peut aussi prévoir d'interrompre une boucle (pour éviter les ennuis) en utilisant la commande `break`.

3 Les fonctions

Dans les langages sophistiqués, une fonction est un sous-programme (suite d'instructions indépendante du reste) qui, quand il est appelé, associe à des valeurs (les paramètres) un résultat obtenu par l'exécution associée à ce

sous-programme [3].
La mise en forme se présente ainsi :

```
def nomdelafonction(x1,...,xn): # x1, ..., xn sont les paramètres de la fonction
    ...
    ...
    ... # série d'instructions
  # la fin de l'indentation signale la fin de la définition de la fonction
... # ici démarre le corps du programme principal
...
...
```

Tout ce qui se trouve à l'intérieur d'une fonction est traité en « local » contrairement aux variables du corps du programme qui sont des **variables globales**.

Exemple. Le programme :

```
def f(x): # la variable x présente ici est
          locale dans la fonction f
    x=x+2 # les modifications n'auront pas
            d'influence sur la valeur de
             x à l'extérieur
    f=x
    return f

x=3 # la variable x dans le corps du
        programme est "globale"
print(x)
print(f(x))
print(x)
```

renvoie :

```
*** Remote Interpreter Reinitialized  ***
>>>
3
5
3
>>>
```

La valeur de la variable `x` n'a pas changé avec l'exécution de la fonction `f`.

Si l'on veut utiliser et manipuler une variable globale dans une fonction, il faut la déclarer comme telle dans le corps de la fonction.

Exemple. Le programme :

```
def f(x): # la variable x présente ici est
          locale dans la fonction f
    global y # la variable y est une
               variable globale, il faut la
                définir à l'extérieur de la
                fonction
    y=x+2 # les modifications auront une
            influence sur la valeur de y
             à l'extérieur
    f=y
    return f

y=3 # la variable x dans le corps du
        programme est "globale"
print(y)
print(f(y))
print(y)
```

renvoie :

```
*** Remote Interpreter Reinitialized  ***
>>>
3
5
5
>>>
```

La valeur de la variable `y` a bien changé avec l'exécution de la fonction `f`.

Les fonctions étant des sous-programmes, on peut, évidemment, appeler et définir des fonctions dans les fonctions. Les fonctions permettent de découper un programme en petits sous-programmes appelés plusieurs fois.

Exemple.

3. ce résultat peut ne pas être numérique : tracé de courbes, changement de valeurs, modification d'une table...

En utilisant :

```
def moyenne(x,y): # calcule de x puissance
          n
    return (x+y)/2

x=float(input("1ère valeur :"))
y=float(input("2ème valeur :"))
print("moyenne des valeurs : ",moyenne(x,y)
          )
```

avec les valeurs 4 et 3, on obtient :

```
*** Remote Interpreter Reinitialized  ***
>>>
moyenne des valeurs :  3.5
>>>
```

Si on veut retourner plusieurs valeurs (comme pour `print`) on indique `return(x,y)`.
Le langage **Python** se lisant de gauche à droite, il faut faire attention à la composition de fonctions.

Exemple.

```
>>> a=3
>>> def g(x):
...     global a
...     a=x+1
...     return a
...
>>> def somme(x,y):
...     return(x+y)
...
>>> somme(a,g(a))
7
>>> somme(g(a),a)
10
>>>
```

Une fonction peut être un paramètre d'une autre fonction.

Exemple.

```
>>> def somme_fonction(f,n):
...     s=0
...     for i in range(n+1):
...         s=s+f(i) # calcul de la somme des f(i) pour i allant de 0 à n
...     return(s)
...
>>> def carre(x):
...     return(x*x)
...
>>> somme_fonction(carre,10)
385
>>>
```

Il est possible de ne pas définir la fonction paramètre, en utilisant la commande `lambda`.

Exemple.

```
>>> somme_fonction(lambda x:2*x*x-x+1,10)
726
>>>
```

Remarque. On appellera **procédure**, toute fonction qui ne retourne pas de valeur mais peut, éventuellement, modifier le contenu de certaines variables.

Les conditions d'utilisation de la fonction peuvent être controler à l'aide la fonction `assert`

```
def fonction( x,...):
    assert  x !=0 # condition à verifier
    """ explications utilisation """"
```

La commande `help(fonction)` renverra les modalités indiquée dans `"""explications utilisations"""`.

3.1 Fonctions récursives

La récursivité est un mode naturel de programmation d'un algorithme : très puissant et gourmand en mémoire. Des algorithmes définis par récurrence, a priori compliqués à programmer deviennent très simple (comme on le verra avec les exemples qui suivent). Il consiste a appeler dans la fonction la même fonction appliquer sur des données de taille inférieur (c'est une récurrence descendante). L'ordinateur "descend" jusqu'à obtenir une valeur et remonte progressivement pour construire la réponse. Ainsi l'étude de la complexité s'obtient souvent par une relation de récurrence : c'est magique ! Attention suivant le choix de l'écriture on peut aussi dangereusement augmenter les calcules. On remarquera que le choix récursif ne permet pas d'accélérer un algorithme mais permet de le formaliser et de le mettre en place très simplement.
Par exemple une suite récurrente s'écrit très naturellement :

```
>>> def u(n):
        if n==1:
          return 1
        if n==0:
          return 1
        else:
          return u(n-1)+u(n-2)
```

La structure globale :

```
def fonction(A,n):
    if n=n0:
        return .... # initialisation f(A,n0)
    B=f(A,n-1)
     .........
     return .... # renvoie f(A,n)
```

4 Les modules.

Contrairement aux fonctions déjà présentées, beaucoup d'autres ne sont pas directement implémentées dans le langage **Python**. Il faut, donc, appeler des « modules » qui les contiennent à l'aide de la commande `import`. Nous utiliserons principalement, les modules :

- `math` qui contient toutes les fonctions et les constantes mathématiques usuelles
- `cmath` qui contient toutes les fonctions nécessaires aux calculs sur $\mathbb{C}$
- `random` pour générer des nombres aléatoires (en fait, pseudo-aléatoires [4]
- `fractions` pour faire des calculs exacts avec des fractions
- `scipy`, module qui comprend les modules suivant :
  - `numpy` pour le calcul scientifique
  - `matplotlib.pyplot` puissant outil pour tracer et visualiser des données
- `PIL.Image` pour manipuler les fichiers d'images.

Pour importer une bibliothèque, on utilise la commande `import nom_du_module` et pour utilise la commande : `nom_du_module.nom_de_la_commande`. Il s'agit de programmation objet [5] : dans la « classe » `module`, on appelle l' « attribut » `commande`.

4. une séquence de nombres présentant certaines propriétés du hasard
5. voir section ??, page ??

Exemple.

```
>>> import math
>>> math.pi
3.141592653589793
>>> import random
>>> random.randint(1,6) # retourne un nombre "aléatoire" entre 1 et 6
4
>>> import matplotlib.pyplot
>>> import numpy
>>> t = numpy.arange(0., 5., 0.2)
>>> matplotlib.pyplot.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^') # pour t en
          tirets rouges, t^2 en carrés bleus et t^3 en triangle verts
[<matplotlib.lines.Line2D object at 0x046132D0>,
 <matplotlib.lines.Line2D object at 0x04613470>,
 <matplotlib.lines.Line2D object at 0x04613830>]
>>> matplotlib.pyplot.show()
>>>
```

4.1 Calcul matriciel

On importe les modules `numpy` pour le calcul matriciel et `numpy.alg` pour le calcul algébrique (résolution de système)

```
>>> import numpy as np
import numpy.linalg as alg
```

Pour créer une matrice :

```
A=np.array([...,....],[....])
```

On peut aussi transformer une liste en matrice

```
A=np.array([[ i +j for i in range(1,n)]for j in range(1;n)])
```

Parmi les opérations utiles :

```
A.shape() # renvoie la dimension de la matrice
n,p=np.shape(A)
np.dot(A,B)
A.dot(B) # pour multiplier deux matrices
np.transpose(A) # la transposée
A.T # transpose aussi
np.trace(A) # la trace
```

Les matrice usuelle pour en construire d'autres :

```
np.zeros((n,p)) # une matrice de 0 de taille (n,p)
np.ones((n,p) # une matrice de 1 de taille (n,p)
np.eye(n) # une matrice identité de taille (n,n)
np.diag([d_1,...d_n]) # renvoie une matrice diagonale de diagonale d_1,...d_n
```

Les listes et arrangements :

```
np.arange(a,b,delta) # une liste de nombre entre a et b avec un pas de delta
np.linspace(a,b,N) # une liste de N nombres réparti régulièrement entre a et b
rang(n) # une liste d'entier entre 0 et n-1
range(p,n) # une liste d'entier entre 1 et n-1
range(p,n,k)# une liste d'entier de p â n-1 avec un pas de k
```

Pour la lecture et l'utilisation des matrices on fera attention à la nuimérotation qui commence en 0 :

```
A[1, 0] # terme de la deuxième ligne, première colonne 3
A[0, :] # première ligne sous forme de tableau à 1 dimension array([1, 2])
A[0, :].shape
(2,)
A[0:1, :] # première ligne sous forme de matrice ligne array([[1, 2]])
A[0:1, :].shape
(1, 2)
A[:, 1] # deuxième colonne sous forme de tableau à 1 dimension array([2, 4, 6])
A[:, 1:2] # deuxième colonne sous forme de matrice colonne
array([[2],[4],[6]])
A[1:3, 0:2] # sous-matrice lignes 2 et 3, colonnes 1 et 2
```

Dans le module `numpy.linalg` :

```
alg.inv(A) # renvoie l'inverse de A
alg.eigvals(A) # renvoie les valeurs propres de A
alg.eig(A)  # renvoie les vecteurs propres de A
alg.solve(A,b) # résout AX=b
np.vdot(u,v) # produit scalaire canonique
```

Opération sur les liste ou les matrices :

```
np.sum(A) # somme des éléments de A
np.min()# min des éléments de A
np.max()# max des éléments de A
np.mean()# moyenne des éléments de A
np.cumsum()# somme partielle des éléments de A
np.median()# médiane des éléments de A
np.var()# variance des éléments de A
np.std()# écart type des éléments de A
```

On peut aussi le faire pour chaque colonne ou chaque ligne :

```
np.sum(A,0) # par colonne
np.sum(A,1) # par ligne
```

4.2 Représentation graphique

Importer le modul `pyplot` :

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

Choisir les axes :

```
plt.axis('equal') # repère orthonormé
plt.axis([x_min,x_max,y_min,y_max]) # taille de la fenetre
plt.grid() # creer un cadrillage
plt.show() # affiche le graphe
plt.clf() # efface le graphe
```

Pour tracer il faut la liste de abscisses $X = [x_1, \dots x_n]$ et celle des ordonnée $Y = [f!x_1), \dots, f(y_n)]$

```
X=np.arange(x_1,x_n,pas)
X=np.linspace(x_1,x_n,n)
Y=[f(x) for x in X] # creer la liste des abscisses à partir de celle des ordonnée

# on peut transformer f en la vectorisant et la faire fonctionner sur les listes
```

```
f=np.vectorize(f)
Y=f(X)
```

Remarquons que les fonctions numpy sont directement vectorisées.
Pour tracer en deux dimension :

```
plt.plot(X,Y)
plt.show()
# en cpmplément:
plt.plot(x,y,color='*',linestyle='*',marker='*') # les couleurs: 'g: vert, r: rouge, b:
            bleu
# le type de ligne: linestyle: '-' ligne continue,'--' ligne discontinue, ';' pointillé
           .   Le style des points:  marker:'+', ','o','v'
```

Les fonctions usuelles :

```
np.exp, np.log,np.sin,np.cos,no.sqrt,np.abs,np.floor
np.e, np.pi
```

Fonctions de deux variables et représentation en trois dimensions :

```
from mpl_toolkits.mplot3d import Axes3D # importer la fonction Axes3D
ax=Axes3D(plt.figure())
X=np.arange(....)
Y=np.arange(....)
X,Y=np.meshgrid(X,Y) # creer une grille
Z=f(X,Y) # ou vectoriser la fonction si, necessaire
ax.plot_surface(X,Y,Z)
plt.show()


plt.contour(X,Y,Z,[c_1,...c_n]) # les ligne f(x,y)=c_i
ax.quiver(X,Y,Z,,u,v,w))# tracer le gradien
plt.quiver(X,Y,u,v) # u=partial_1 f(X,Y), v=partial_2f(X,Y)
```

Pour les diagramme en barre et histogramme :

```
plt.bar(X,Y,e) # abscisse X et ordonnée Y épaissaur e
plt.bar([1,2,3,4,5,6,7,8],[1,2,3,4,5,1,2,3],0.1)
plt.hist(X, bins=N)# X la série statistique et N la liste des classes ou le nb de
           classe
plt.hist(5*rd.random(100),bins=[0,1,2,3,4,5])
```

4.3 Probabilité

Pour simuler du hasard et les lois classiques :

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

Les lois discrètes :

```
rd.randint(a,b,n) # une liste de n VA independantes de loi uniforme discrète sur (a,b
           -1)
rd.randint(a,b,(n,p)) # un tableau de n*p VA independantes de loi uniforme discrète sur
            (a,b-1)
rd.binomiale(n,p,N) #une liste de n VA independantes de loi binomiale de paramètre (n,p
           )
rd.binomiale(n,p,(N,P))  # un tableau de n*p VA independantes
```

```
rd.geometric(p,N) #une liste de n VA independantes de loi geometrique de parametre p
d.geometric(p,(N,P))  # un tableau de n*p VA independantes
rd.poisson(lambda,N) #une liste de n VA independantes de loi Poisson de parametre
           lambda
rd.poisson(lambda,(N,P))  # un tableau de n*p VA independantes
```

4.4 Analyse numérique

On importe le modul `optimize` et `integrate` :

```
import scipy.optimize as resol
import scipy.integrate as integr
import numpy as np
```

Pour résoudre une èquation $f(x) = 0$ on peut utiliser la commande `fsolve` qui prend une condition initiale et nous renvoie une solution (qui peut dépendre du choix de x_0) :

```
resol.fsolve(f,x_0)
```

5 La gestion des fichiers sous Python.

Il est utile de pouvoir ouvrir, modifier, créer, écrire, réécrire un fichier pour récupérer des données en grand nombre ou pour traiter une image, un son, ...

5.1 Les répertoires.

Les commandes qui suivent doivent être lancées après `import os`.

- savoir quel est le répertoire courant : `os.getcwd()`
- connaître le contenu du répertoire courant : `os.system('truc')` où `truc` est la commande système de l'O.S. [6] sous lequel **Python** est exécuté [7].
- changer de répertoire courant : `os.chdir('chemin_du_repertoire')`

5.2 Les fichiers.

Pour ouvrir un fichier, il faut utiliser la commande `open`, qui renvoie une « interface-fichier » à stocker dans une variable, dont la syntaxe est la suivante :

```
open(fichier,'mode','code')
```

où `mode` et `code` sont optionnels [8]. Si le fichier `fichier` n'est pas dans le répertoire courant, il faut indiquer explicitement le chemin d'accès.

Exemple.

```
f=open('C:\\Users\\eleve\\Documents\\Python\\essai.py') #sous DOS/Windows
```

Les différents `mode` sont :

- `r` : ouverture en lecture seule (**r***ead only*)
- `w` : ouverture en écriture seule (**w***rite only*)
- `r+` : ouverture en mode lecture/écriture
- `x` : ouverture en mode création

6. *Operating System* ou système d'exploitation
7. ainsi, toutes les commandes systèmes peuvent être effectuées à partir de **Python**, effacement de fichiers, de répertoires, changement de répertoire, etc. : DANGER!!!!
8. valeurs par défaut : `rt` et `UTF-8`

— `a` : ouverture en mode **a***ppend* (écriture à la fin du fichier existant)
— `t` : mode **t**ext
— `b` : mode **b**inaire

5.2.1 Lecture.

On suppose ici que notre interface-fichier s'appelle `f`.
— `f.read(n)` :
  — lit le fichier jusqu'au caractère n (le premier caractère est 0 !!) et retourne ces n caractères sous forme de chaîne
  — en cas de répétition de l'instruction, la lecture se poursuit depuis la dernière position précédente
  — si l'argument est omis, tout le fichier est lu
  — s'il n'y a plus rien à lire, retourne `''`
— `f.tell()` : retourne la position courante
— `f.seek(n)` : se place à la position n.
— `f.readline()` :
  — retourne les caractères jusqu'au prochain `\n` qui marque une fin de ligne
  — en cas d'appel ultérieur, retourne la ligne suivante
  — s'il n'y a plus rien à lire, retourne `''`
— `f.readlines()` : retourne toutes les lignes du fichiers, chacune sous forme de chaîne

On peut aussi utiliser la syntaxe :

```
for ligne in f: # o\'u f=open(...)
    instruction
    ....
    ...
```

Remarque. Ne pas ouvrir un fichier binaire en mode texte car il y a un **énorme** risque de confusion dans les caractères de fin de ligne `\n`.

Remarque.
— `\n` est un saut de ligne
— `\t` est une tabulation
— `\b` est un « backspace »
— `\a` est un « bip »
— `\'` est un « ' », mais il ne ferme pas la chaine de caractères
— `\"` est un « " », mais il ne ferme pas la chaine de caractères
— `\\` est un « \ »

5.2.2 Écriture.

La commande `f.write('qqchose')` écrit à partir de la position courante la chaîne de caractère `'qqchose`.

Remarque. Il ne s'agit pas d'une insertion de texte mais d'un remplacement !!

```
>>> f=open('test.py','r+')
>>> f.read()
'ligne2\n'
>>> f.seek(1)
1
>>> f.write('!')
1
>>> f=open('test.py','r+')
>>> f.read()
'l!gne2\n'
>>>
```

5.2.3 Fermeture.

Il est préférable de fermer un fichier lorsqu'il n'est plus manipulé (cela évite de le modifier ou de le supprimer par mégarde) avec la commande : `f.close()`.

Remarque. La commande `f.closed()` retourne un booléen [9] qui indique si le fichier `f` est fermé ou non.

On peut aussi se servir de l'instruction `with ... as ...` qui fermera automatiquement le fichier après l'exécution des instructions.

5.3 Les fichiers images.

On peut représenter une image (noire et blanc ou couleur) grâce à une matrice ou chaque élément a_{ij} représente une case ou un pixel de cordonnée (i, j) qui est de couleur la valeur a_{ij} qui est un nombre entier entre 0 et 255 :

```
>>> import PIL.Image
>>> photo=PIL.Image.open('photo.png') # ouvre le fichier photo sous Python
>>> photo.show() # affiche l'image
>>> photo.size # indique la taille
(201,251)
```

Il faut, pour travailler sur une photo, d'abord la transformer en tableau (matrice). On utilise, pour cela, la bibliothèque `numpy` :

```
>>> import numpy
>>> phototab=numpy.array(photo) # Transforme la photo en tableau
>>> photo2=Image.fromarray(phototab) # Transforme un tableau en image
>>> photo2.save('photo.jpg') # Permet de sauvegarder une image sous un format au choix.
```

9. voir 1.6 page 6

6 Exemples et applications

6.1 Les listes

Exercice 6.1. Créer une liste de 50 entiers impairs successifs

Il est interessant de "stocker" les donners dans une liste. et en Python pour "ajouter" un terme dans une liste on concatène deux listes :

```
v=v+[u] #ajoute le terme u a la liste v
v.append(u) # la meme chose:
u=[1,2,3]

u=u+[75]

u
Out[3]: [1, 2, 3, 75]

u.append(1235)

u
Out[5]: [1, 2, 3, 75, 1235]
```

Dans ces exercices, on se propose de construire une liste contenant les n premières valeurs des suites suivantes :

Exercice 6.2. Soit pour tout $n \in \mathbb{N}^*$ la suite (u_n) de terme général :

$$u_n = \frac{n+1}{\sqrt{n^3 - n + 1}}.$$

Compléter le programme suivant, afin qu'il affiche la liste $[u_1, u_2 \dots u_n]$:

```
import numpy as np
n=int(input('n='))
u=[]
for i in range(n):
     ..............

print(u)
```

Exercice 6.3. Soit pour tout $n \in \mathbb{N}$ la suite (u_n) de terme général :

$$u_{n+1} = u_n - \ln(u_n)$$

Programmer une fonction `U(u0,n)` prenant pour paramètres u_0 et n et renvoyant la liste des $n+1$ premiers termes de (u_n) :

```
def U(u0,n):
   ................
   ................
   return u
```

Dans le module `numpy.linalg` :

```
alg.inv(A) # renvoie l'inverse de A
alg.eigvals(A) # renvoie les valeurs propres de A
alg.eig(A)  # renvoie les vecteurs propres de A
alg.solve(A,b) # resout AX=b
np.vdot(u,v) # produit scalaire canonique
```

Opération sur les liste ou les matrices :

```
np.sum(A) # somme des elements de A
np.min()# min des elements de A
np.max()# max des elements de A
np.mean()# moyenne des elements de A
np.cumsum()# somme partielle des elements de A
np.median()# mediane des elements de A
np.var()# variance des elements de A
np.std()# ecart type des elements de A
```

On peut aussi le faire pour chaque colonne ou chaque ligne :

```
np.sum(A,0) # par colonne
np.sum(A,1) # par ligne
```

Exercice 6.4. Construire la liste U des 100 premiers carrés, puis celle des sommes partielles des 100 premiers carrés.

Exercice 6.5. Déterminer la liste des $\frac{1}{n^2}$ pour les 50 premiers entiers non nul. Puis calculer une approximation de :

$$\sum_{k=1}^{50} \frac{1}{k^2}.$$

On ne peut pas non plus manipuler les listes (ou les tableaux) comme des matrices (les opération algébrique n'existent pas) il faut pour ça importer le module `numpy` et transformer les listes en matrices :

6.2 Matrices

Dans le module `numpy` on peut créer des matrices avec lesquelles on peut faire les opérations algébriques classique mais pas de concaténation, on peut tout simplement transformer une liste en matrice :
Pour la lecture et l'utilisation des matrice on fera attention à la numérotation qui commence en 0 :

```
A[1, 0] # terme de la deuxieme ligne, premiere colonne 3
A[0, :] # premiere ligne sous forme de tableau a 1 dimension array([1, 2])
A[0, :].shape
(2,)
A[0, :] # premiere ligne sous forme de matrice ligne array([[1, 2]])
A[0, :].shape
(1, 2)
A[:, 1] # deuxieme colonne sous forme de tableau a 1 dimension array([2, 4, 6])
A[:, 1:2]# deuxieme colonne sous forme de matrice colonne
A[:, :1]# premiere colonne colonne sous forme de matrice colonne
array([[2],[4],[6]])
A[[1,2], 1:3] # sous-matrice lignes 2 et 3, colonnes 1 et 2
```

On importe les modules `numpy` pour le calcul matriciel et `numpy.linalg` pour le calcul algébrique (résolution de système)

```
>>> import numpy as np
import numpy.linalg as alg
```

Pour créer une matrice :

```
A=np.array([[...,....],[....]])
```

On peut aussi transformer une liste en matrice

```
A=np.array([[ i +j for i in range(1,n)]for j in range(1;n)])
```

Mais attention on ne peut pas "copier directement" des listes ou des matrices, il faut utiliser `copy()` :

```
A=B.copy()# en oriente objet
A=np.copy(B) # en directe
```

Exercice 6.6. Créer une matrice A de $\mathcal{M}_{3,20}(\mathbb{R})$ dont la première ligne est constituée d'entiers pairs, la deuxième ligne d'entiers impairs et la troisième de multiples de trois. Créer la matrice B composée des deux dernières lignes de A. Extraire les trois dernières colonnes de A.

Exercice 6.7. Construire une matrice $\mathcal{M}_{30}(\mathbb{R})$ dont la première colonne et la dernière ligne ne sont composées que de 1, le reste étant nul.

Dans le module `linalg` :

```
alg.inv(A) # renvoie l'inverse de la matrice A
alg.rank(A) # renvoie le rang de la matrice A
alg.matrix_power(A,n) # renvoie A^n
```

Exercice 6.8. Soit

$$P = \begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ et } A = \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

on pose $B = PAP^{-1}$, vérifier que $Q(B) = 0$ avec $Q(X) = X^3 - 3X^2 - 4X + 12$

6.3 Structurer un programme

Pour structurer un programme il nous faut :

- Des paramètres d'entrée eventuelement avec la commande `input()`. :
- Déterminer la grandeur de sortie
- Afficher le résultat `print`

```
Donnee=input(" entrer la valeur")
 ..................
 print(Grandeur de sortie)
```

Pour le calcul de la grandeur étudiée, on peut :

- Utiliser la commande `for`, une boucle finie avec un début et une fin (on sait donc combien de boucles on va faire)

  ```
  for i in liste:
        ............
  ```

  Par exemple pour une liste de n terme :

  ```
  for i in range(n)
        ............
  ```

  Attention i prend les valeur de la liste range, c-a-d , des entier de 0 à $n-1$.
- Utiliser la commande `while`, une boucle conditionnelle (on ne sait pas quand combien de boucle on doit faire mais seulement à quelle condition on s'arrête)

  ```
  n=0 \\indice de comptage
  while relation:
       ..........
       n=n+1\\ compte le nombre de boucles
  ```

- Utiliser la commande `if`, un test (si c'est vrai alors).

```
if (A est vraie) :
    ......
else:
            ......
```

— La commande `elif` qui est la contraction de deux commande, `else` et `if` : if .condition : ..... elif condition : ..... else : .....

Exercice 6.9. Proposer différents programmes effectuants les taches suivantes :

1. On propose d'entrer un nombre n et le programme renvoie la valeur de u_n avec
$$u_0 = 1 \quad u_{n+1} = \sqrt{u_n^2 + 1}.$$
2. On propose d'entrer un nombre M et le programme renvoie la première valeur de u_n dépassant M ainsi que n avec
$$u_0 = 1 \quad u_{n+1} = 2u_n + 1.$$
3. On lance un dé à 6 faces, on gagne un euro si on obtient 6, on perd un euro si on obtient 1 et rien sinon. Proposer un programme renvoyant le gain d'une partie.

L'objectif est, après avoir justifier la convergence, d'obtenir une approximation de u_n :

Exercice 6.10. 1. Soit $(u)_{n\in\mathbb{N}^*}$ la suite de terme général :
$$\begin{cases} u_{n+1} = 1 + \frac{u_n}{n+1} \\ u_1 > 0 \end{cases}$$

(a) Démontrer que u_n est strictement supérieur à 1 à partir d'un certain rang. Puis montrer que :
$$\forall n > 2 : \qquad u_n > 1 + \frac{1}{n}.$$

(b) En déduire que la suite de terme général u_n est décroissante à partir d'un certain rang et convergente. Déterminer sa limite.

2. Proposer un programme renvoyant la première valeur de n pour $u_1 = 10$ vérifiant $\mid u_n - 1 \mid \leq 10^{-5}$ en utilisant une boucle while.

6.4 Manipuler une fonctions

Une fonction est un sous programme, qui quand il est programmer, est garder en mémoire temporairement et est exécuté quand elle est appelé. Elle n'affiche rien si on ne la lance pas et s'applique sur les paramètre d'entrée. Elle peut être appelé dans n'importe quel programme et évite l'utilisation de la fonction `input`.

6.5 Fonction

```
def nom_fonction(parametre_entree):
                        """ aide et instruction"""# optionel
                        assert (condition necessaire) # optionel
                        ......................
                        [instruction_definir_parametre_sortie]
                        ...................
                        return( paramametres de sortie) # est optionel.
```

Exercice 6.11. Programmer la fonction `u(u0,n)` prenant comme paramètres un entier n et un réel u_0 et renvoyant la liste des termes $[u_0, \ldots, u_n]$ avec :
$$u_{n+1} = \sqrt{u_n^2 + 1}.$$

Exercice 6.12. Programmer la fonction `f(x,m)` prenant comme paramètres un entier m et un réel x et renvoyant la valeur $f_m(x)$ avec :

$$f_m(x) = \frac{3-m}{3-m\sqrt{x}}.$$

Exercice 6.13.

1. Ecrire un scripte renvoyant la matrice suivante :

```
A =

    1.    1.    0.    0.    0.    0.    0.    0.
    1.    0.    1.    0.    0.    0.    0.    0.
    1.    0.    0.    1.    0.    0.    0.    0.
    1.    0.    0.    0.    1.    0.    0.    0.
    1.    0.    0.    0.    0.    1.    0.    0.
    1.    0.    0.    0.    0.    0.    1.    0.
    1.    0.    0.    0.    0.    0.    0.    1.
    1.    1.    1.    1.    1.    1.    1.    1.
```

2. Programmer une fonction `matrice_n`, prenant pour paramètre n, renvoyant la matrice de $\mathcal{M}_n(\mathbb{R})$ généralisant la précédente.

Exercice 6.14.

1. Programmer la fonction `matrice_ab` prenant pour paramètres l'entier n, les réels a et b et renvoyant la matrice de $\mathcal{M}_n(\mathbb{R})$:

$$A = \begin{pmatrix} a & & (b) \\ & \ddots & \\ (b) & & a \end{pmatrix}$$

en complétant le squelette suivant :

```
def matrice_ab(a,b,n):
    ..................
    return A
```

Tester la fonction pour, $n = 10, a = 1$ et $b = 2$.

2. *Pas facile* En calculant $(A+(b-a)I)^2$, déterminer un polynôme annulant A et en déduire une CNS pour que A soit inversible et déterminer son inverse.
3. Programmer une fonction `inverse_matrice_ab` prenant pour paramètres a, b et n et renvoyant l'inverse de la matrice A.

Exercice 6.15. $\star$ Soit $A \in \mathcal{M}_n(\mathbb{R})$ vérifiant $a_{i,j} = \frac{1}{i+j}$.

1. Programmer une fonction `Hilbert_n` prenant pour paramètre l'entier n et renvoyant la matrice A
2. Extraire, en utilisant les commandes `Python` et la fonction `Hilbert_n`, les colonnes paires de cette matrice.

6.6 Représentation graphique

Importer le modul `pyplot` :

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

Choisir les axes :

```
plt.axis('equal') # repère orthonormé
plt.axis([x_min,x_max,y_min,y_max]) # taille de la fenetre
plt.grid() # creer un cadrillage
plt.show() # affiche le graphe
plt.clf() # efface le graphe
```

Pour tracer il faut la liste de abscisses $X = [x_1, \dots x_n]$ et celle des ordonnée $Y = [f!x_1), \dots, f(y_n)]$

```
X=np.arange(x_1,x_n,pas)
X=np.linspace(x_1,x_n,n)
Y=[f(x) for x in X] # creer la liste des abscisses à partir de celle des ordonnée

# on peut transformer f en la vectorisant et la faire fonctionner sur les listes
f=np.vectorize(f)
Y=f(X)
```

Remarquons que les fonctions `numpy` sont directement vectorisées.
Pour tracer en deux dimension :

```
plt.plot(X,Y)
plt.show()
# en cpmplément:
plt.plot(x,y,color='*',linestyle='*',marker='*')
# les couleurs: 'g: vert, r: rouge, b: bleu
# le type de ligne: linestyle: '-' ligne continue,
#'--' ligne discontinue, ';' pointillé.
# Le style des points:  marker:'+', ','o','v'
```

Les fonctions usuelles :

```
np.exp, np.log,np.sin,np.cos,no.sqrt,np.abs,np.floor
np.e, np.pi
```

On se propose de représenter graphiquement les séries de Riemann et de Bertrand :

Exercice 6.16. Proposer un programme (en adaptant les précédents) renvoyant la représentation graphique de la suite suivante :

$$u_n = \sum_{k=1}^{n} \frac{(\ln k)^\beta}{k^\alpha}$$

Dans un premier temps on choisira $\alpha = 1$ et $\beta = 1$. Puis on programmera une fonction prenant pour paramètres n, α et β ; renvoyant la liste des valeurs de (u_n) et on créera un programme qui donnera la représentation graphique de (u_n) en fonction de n, α et β.

Exercice 6.17. On se propose d'étudier, suivant les valeurs de $\alpha > 0$ la série de terme :

$$u_n = \sum_{k=1}^{n} \frac{1}{k^\alpha}$$

1. A quelle condition a t-on la convergence de la série ?
2. En utilisant la liste des n premiers entiers, compléter la fonction suivante, renvoyant la liste des n premier terme de la série.

```
def serieriemann(n,alpha):
    v=[(i+1).......  for i in range(n)]
    u=.............................
    return u
```

3. Proposer un script Python donnant la représentation graphique de la suite sur $[|1; 1000|]$, dans le cas $\alpha = 1$, $\alpha = 2$ et$\alpha = 1,5$

On prendra soin de formaliser les relations de récurrence sous forme matricielle avant de programmer sous `Python`

Exercice 6.18.

$$\begin{cases} u_{n+1} = \frac{1}{3}(u_n + v_n + z_n) \\ v_{n+1} = \frac{u_n}{2} + \frac{z_n}{2} \\ z_{n+1} = \frac{u_n}{3} + \frac{2v_n}{3} \end{cases}$$

1. Programmer une fonction `uvz(U,n)` prenant pour paramètres la liste $U = [u_0, v_0, z_0]$ et l'entier n ; renvoyant la liste $[u_n, v_n, z_n]$
2. Proposer un programme renvoyant la représentation graphique des suites (u_n),(v_n) et (z_n) pour $u_0 = 1$ $v_0 = 0$ et $z_0 = -3$ et $n \in [|0, 50|]$

Exercice 6.19. Proposer un programme Python renvoyant la représentation graphique des suites suivantes :

1. (u_n) définie par : $u_0 = 1; u_1 = -1$ et $\forall n \in \mathbb{N}, \quad u_{n+2} = 4u_{n+1} - 4u_n$.
2. la suite définie par :
$$\begin{cases} u_0 = 1 \\ u_{n+1} = 3u_n + 2 \end{cases}$$

6.7 Programmation récursive

La programmation récursive consiste à appeler une même fonction à l'intérieur d'elle même :
En recursif, une fonction s'appelle elle même :

```
def U(n,u0):
              if n==0:      // La condition initiale
                    return u0

              u=f(U(n-1,u0))   // la relation de recurrence
              return u
--> U(10,1)// retourne   u_10    ou u_(n+1)=f(u_n) et u_0=1
```

Exercice 6.20. ⋆⋆ En utilisant la relation :

$$\begin{cases} \forall n & \binom{n}{0} = 1 \\ \forall n & \binom{n}{n} = 1 \\ \forall p \leq n & \binom{n+1}{p} = \binom{n}{p} + \binom{n}{p-1} \end{cases}$$

1. Programmer une fonction récursive `binome(n,p)`, prenant pour paramètres n et p et renvoyant le nombre $\binom{n}{p}$; en complétant le squelette suivant :

```
def binome(n,p):
    if (p==0) and (p==n):
          return 1


    b=..........
    return b
```

2. Améliorer le programme en proposant une fonction `tablebinome(n)` renvoyant la liste *bin* des listes des coefficients binomiaux, de telle sorte que `bin(n,p)` soit le coefficient $\binom{n}{p-1}$. On pourra compléter le programme suivant :

```
def tablebinome(n):
          if n==1:
               bin=[[1,1]]
               return bin

          bin=.......
          bin[n-1,0]=
```

```
          for i in range(1,n):
               bin[n-1,i]=.........
           end
           bin[n-1,n]=1
endfunction
```

7 Ce qu'il faut savoir

7.1 Factoriel

Calcul de $n!$.

```
>>> def factoriel(n)
""" calcule factoriel n"""
       if n==0:
          return 1
       else:
          return n*factoriel(n-1)
```

7.2 Puissance rapide

Un moyen d'accélérer le calcule de puissance repose sur le principe " diviser pour régner" :

```
>>> def puissance(x,n):
   if n==1:
            return x
   if n==0:
        return 1
   else:
        r=puissance(x,n//2)# n//2 est le quotien de n par 2
        if n% 2==0: # n est pair
            return r*r
        else:
            return x*r*r
```

7.3 Recherche dichotomique dans un tableau trié

Voici une variante d'un tri dichotomique pour un tableau trié.

```
>>> def recherche_dichotomique(x,a):
       m=len(a)//2
           if x==a[m]:
                      return True
           else:
                     if m==0:
                            return False
                     else:
                            if x>a[m]:
                                 return recherche_dichotomique(x,a[m:len(a)])
                            else:
                                      return recherche_dichotomique(x,a[0:m])


>>> recherche_dichotomique(2,[1,2,3,4,5])
True
```

Ici en posant n la longueur du tableau on trouve comme précédement une complexite en $O(\log n)$. On peut aussi proposer :

```
>>> def recherche_dichotomique(x,a):
       m=len(a)//2
           if x==a[m]:
                      return True
```

```
            else:
                    if m==0:
                            return False
                    else:
                            if x>a[m]:
                                    del a[:m]
                                return recherche_dichotomique(x,a)
                            else:
                                    del a[m:]
                                    return recherche_dichotomique(x,a)
```

7.4 Tris

Le tri insertion en moyenne $O(n^2)$:

```
def insertion(a,b):
    """ a une liste triée, on insert b dans a"""
    l=len(a)
    if b>=a[l-1]:
        return a+[b]
    if b<=a[0]:
        return [b]+a
    while b<a[l-1]and l>0:
        l=l-1
        return a[0:l]+[b]+a[l:]
def tri_insertion(a):
    l=len(a)
    if l<=1:
        return a
    L=tri_insertion(a[0:l-1])
    return insertion(L,a[l-1])
```

```
  >>>def tri_insertion2(a):
        if len(a)==1:
            return a
        for i in range(len(a)):
            if     a[i]>a[i+1]:
                   a[i],a[i+1]=a[i+1],a[i]
        max=a[len(a)-1]
        tri_insertion2(a[0:len(a)-1])
        return a.append(max)
```

Le tri rapide, au mieux en $0(n\ln(n))$:

```
 >>>import random
 >>> def partition(a):
    j=random.randint(0,len(a)-1)
    pivot=a[j]
    partieinf=[]
    partiesup=[]
    for k in range(len(a)):
        if  k!=j:
            if a[k]>pivot:
                partiesup=partiesup+[a[k]]
            else:
                partieinf=[a[k]]+partieinf
    return partiesup,partieinf,pivot
```

```
def tri_rapide(a):
    if len(a)<=2:
        return tri_insertion(a)
    if len(a)>2:
        sup,inf,pivot=partition(a)
        return   tri_rapide(sup)+[pivot]+tri_rapide(inf)
```

Le tri fusion, en moyenne en $0n\ln(n)$) :

```
>>> def indice(a,b):
    """ a une liste et b un nombre  l'indice de départ"""
    i=0
    while b>a[i]:
        i=i+1
    return i
def fusion(a,b):
    l=len(a)-1
    k=len(b)-1
    if a[0]>=b[k]:
        return b+a
    else:
        if b[0]>=a[l]:
            return a+b
        else:
            j=indice(a,b[0])
            return a[0:j]+[b[0]]+fusion(a[j:],b[1:])
def tri_fusion(a):
    if len(a)<4:
        return tri_insertion(a)
    else:
        n=len(a)//2
        c=tri_fusion(a[0:n])
        d=tri_fusion(a[n:])
        return fusion(c,d)
```

7.5 Newton

Voici un petit programme correspondant à l'algorithme de la sécante :

```
def newton(f,x0,df,n):
      def deriv(f,a,b):
            return (f(a)-f(b))/(a-b)
      if n==0:
            return x0
      if n==1:
            return x0-f(x0)/df
      x=newton(f,x0,df,n-1)
      y=newton(f,x0,df,n-2)
      return x-f(x)/deriv(f,x,y)
```

Ici la précision n'est pas explicite, on a juste imposé le nombre de boucle si on veut une précision ϵ il faut estimer n. On peut imposer la recherche de x_0 tant que $\mid f(x_k) \mid > \epsilon$ et on utilise une boucle conditionnelle while :

```
def newton(f,x0,df,epsilon):
      def deriv(f,a,b):
            return (f(a)-f(b))/(a-b)
```

```
        x1= x0-f(x0)/df
        while abs(f(x0))>epsilon:
             x0,x1=x1, x1-f(x1)/deriv(f,x0,x1)
        return x0
```

7.6 Dichotomie

Soit f continue sur $[a,b]$ si $f(a).f(b) < 0$ alors on pose $c = \frac{a+b}{2}$ et l'algorithme de dichotomie (ou de bissection) converge avec une vitesse $0(\frac{1}{2^n})$.

```
>>>def dichotomie(f,a,b,epsilon):
        assert f(a)*f(b)<=0
           if abs(a-b)>epsilon:
                  c=(a+b)/2
                  if f(c)*f(b)<0:
                      return dichotonmie(f,c,b,epsilon)
                  else:
                       return dichotomie(f,a,c,epsilon)
           return (a+b)/2
```

7.7 Euler

Pour résoudre l'équation différentiel du premier ordre :

$$y' = f(t,y).$$

Avec la méthode d'Euler explicite à un pas, on choisira le pas fixe, tel que $\Lambda\Delta T$ soit assez petit et surtout que ΛT ne soit pas trop grand(Λ est le coefficient de Lipschitz de f, $\Delta T = t_{n+1} - t_n = \frac{T}{N}$) et on construira les points de coordonnées (t_n, y_n) vérifiant :

$$\begin{cases} y_{n+1} = y_n + \Delta T f(t_n, y_n) \\ (y_0, t_0) = (y(0), 0) \\ t_n = \frac{nT}{N} \end{cases}$$

On se place d'abord dans le cadre des fonctions de $\mathbb{R}$ dans $\mathbb{R}$ et on programmera une fonction prenant les paramètres N, T, y_0 et F retournant un couple de listes (les listes X et Y correspondants aux coordonnées x_i, y_i).

```
  def euler(f,y0,T,N):
    y=[None]*(N+1)
    x=[i*T/N for i in range(N+1)]
    y[0]=y0
    for i in range(N):
        y[i+1]=y[i]+(T/N)*f(x[i],y[i])
    return x,y
```

Ou bien par concaténation :

```
def euler(f,y0,t0,T,N):
    t=[t0]
    y=[y0]
    for i in range(N)
         t.append(t[i]+((T-t0)/N))
         y.append(y[i]+((T-t0)/N)*f(t[i],y[i]))
    return t,y
```

Il peut être intéressant de représenter graphiquement ces données :

```
def graph_euler(f,y0,T,N):
    t,y=euler(f,y0,t0,T,N)
    import matplotlib.pyplot as pl
    pl.plot(t,y)
    pl.show()
```

Dans le cadre des équations d'ordre supérieur la méthode doit être étendue avec précaution car (à moins d'utiliser le mode *array* de *numpy*) les additions et multiplications sur les listes (ou les uplets) sont des concaténations.
Soit

$$\begin{cases} y'' = f(t, y, y') \\ y'(0) = a \\ y(0) = b \end{cases}$$

On pose $Y = (p, q) = (y', y)$ et $F : (t, Y) \to (f(t, q, p), p)$ et l'équation se ramène alors à :

$$\begin{cases} \frac{dY}{dt} = F(t, Y) \\ Y(0) = (a, b) \end{cases}$$

On adapte alors le programme précédent ainsi :

```
def euler_ordre2(f,y0,t0,T,N):
    DT=(T-t0)/N
    t=[i*DT for i in range(N+1)]
    y=[None]*(N+1)
    y[0]=y0

    for i in range(N):

        y[i+1]=[y[i][0]+DT*f(t[i],y[i])[0],y[i][1]+DT*f(t[i],y[i])[1]]
    return t,y
```

On l'applique alors à l'équation différentielle $y'' = y$ en définissant :

```
def f(t,y):
    return [y[1],y[0]]
```

On peut ainsi obtenir facilement le portrait de phase :

```
import matplotlib.pyplot as pl
t,y= euler_ordre2(f,y0,t0,T,N)
p,q=y
pl.pyplot(p,q)
pl.show()
```

On peut bien sûr le généraliser aux ordre supérieur :

```
def euler_ordrep(f,y0,t0,T,N):
    DT=(T-t0)/N
    p=len(y0)
    t=[i*DT for i in range(N+1)]
    y=[None]*(N+1)
    y[0]=y0

    for i in range(N):
        y[i+1]=[None]*p
        for j in range(p):
            y[i+1][j]=y[i][j]+DT*f(t[i],y[i])[j]
    return t,y
```

7.8 Integration

Dans le cas des rectangles on a poser :

$$\int_0^1 f(t)dt \simeq g(1).$$

Ce qui nous donne une approximation en $O(\frac{1}{n})$ et se programme très simplement de la façon suivante :

```
def int_rectangle(f,n,a,b):
    """calcule l'integrale de f entre a et b avec un pas de n"""
    assert a!=b

    S=0
    for i in range(n):
        xi=a+i*(b-a)/n
        S=f(xi)*(b-a)/n+S
    return S
```

Dans le cas d'ordre supérieur, méthode du point milieux, de Simpson et autre....

```
 def approx_quad(g):
    return (g(0)+g(1))/2
def integration_quad(f,a,b,epsilon):
    n=1+int((b-a)/epsilon)
    integf=0
    for i in range(n):
        def g(t):
            return ((b-a)/n)*f(a+((t+i)*(b-a)/n)
        integf=integf+approx_quad(g)
    return integf
def approx_simpson(g):
    return (g(0)+g(1))/6+2*g(0.5)/3
def integration_simpson(f,a,b,epsilon):
    n=1+int((b-a)/epsilon)
    integf=0
    for i in range(n):
        def g(t):
            return ((b-a)/n)*f(a+((t+i)*(b-a)/n)
        integf=integf+approx_simpson(g)
    return integf
```

7.9 Piles

```
 >>>[] # creer une pile
 >>>p.pop() # retire  le dernier terme de la liste et le retourne
 >>>p.append(v) # ajoute v comme dernier terme dans la liste
 >>>len(p) # retourne la taille de la pile
 >>>p[-1]# retourne le sommet de la pile.
```

pour une file d'attente : premier arrivé, premier servi.

```
 def depiler_gauche(P):
       Q=P[0]
       del P[0]
       return Q
```

On peut aussi dépiler et empiler dans une autre pile :

```
P2=P2.append(P1.pop())
```

Plus généralement... tout dépiler :

```
def depiler_empiler(P,Q):
""" Depile P et l'empiler dans Q"""
   while len(P)>0:
        Q=Q.append(P.pop())
```

7.10 Odeint

Pour résoudre $y' = f(y,t)$ ou f est une fonction de plusieurs variables. On doit importer la bibliothèque *scipy.integrate* :

```
  >>> import scipy . integrate as *
  #ou bien
  >>> from spicy.integrate import odeint
```

Elle prend pour argument la fonction f (qu'on peut définir avec lambda) la condition initiale y_0 et une liste de temps $t = [t_0, ...t_i, ...t_n]$. Elle retourne alors la liste des approximation y_i :

```
  >>>odeint(lambda x,t:x,1,[0,0.1,0.2,0.3])
```

On peut créer un tableau de temps en utilisant :

```
  >>> [i/n for k in range(n+1)]
```

Dans le cadre de système et donc d'équation différentielle d'ordre p, la fonction retourne une liste de listes de taille $p-1$ (la liste des Y_i) et la condition initiale doit être une liste. Par exemple :

$$\begin{cases} y'' = y \\ y'(0) = 1 \\ y(0) = 1 \end{cases}$$

Se traduit par le système :

$$\begin{cases} \frac{dy_0}{dt} = y_1 \\ \frac{dy_1}{dt} = y_0 \\ y_1(0) = 1 \\ y_0(0) = 1 \end{cases}$$

```
 def f(y,t):
      return y[1],y[0]
t=[0.01*i for i in range(101)]
from scipy.integrate import odeint
Y=odeint(f,[1,1],t)
y=[Y[i][0] for i in range(101)] # ou aussi dans le mode array de numpy Y[:,0]
y1=[Y[i][1] for i in range(101)]# Y[:,1]
import matplotlib.pyplot as pl
pl.plot(t,y)# tracer de la courbe de y(t)
pl.plot(y,y1)# portrait de phase
pl.show()
```

Remarquons que si on utilise directement *plot* sur Y :

```
>>>Y=odeint(f,[1,1],t)
>>>pl.plot(Y)
>>>pl.show()
```

On obtient le tracé des points (i, y_i) et (i, y_i')(et non pas (t_i, y_i)

7.11 quad

Dans la bibliothèque scipy.integrate, on peut utiliser : $quad(f, a, b)$, que je vous invite à tester. Par exemple $\int_0^1 xdx$ donne bien :

```
>>> import scipy.integrate as intg

>>> intg.quad(lambda x:x,0,1)
>>> (0.5, 5.551115123125783e-15)
```

7.12 Newton et Dichotomie

```
 >>> Import scipy.optimize
 >>>scipy.optimize.newton(f,a,f')
 >>>scipy.optimize.bissect(f,a,b)
```

7.13 Numpy et matrices

```
  >>> from numpy import *
  >>> from scipy import linalg
  >>> A=array([ [1,1,1],[1,0,2],[1,1,0] ]) # On utilise le mode array de numpy
    >>> B=array([1,1,1]) # B étant définie en ligne
  >>> X=linalg.solve(A,B) # recoud AX=B.
   >>> X #   X ressort en  en ligne

  array([ 1.,  0.,  0.])
```

On peut aussi obtenir l'inverse d'une matrice et multiplier ces matrice dans le mode *array* de *numpy*

```
 >>> from numpy import *
 >>> M=array([ [1,1,1],[1,0,2],[1,1,0] ])
 >>> linalg.inv(M)
 >>> linalg.det(M) # détermine le déterminant.
 >>> dot(A,B) # multiplie A et B.

>>>import numpy as np
>>> A=np.array([[1,2,3],[2,4,5],[1,2,5]])# est une matrice 3 3
>>>2*A
array([[ 2,  4,  6],
       [ 4,  8, 10],
       [ 2,  4, 10]])
>>>B=np.zeros((3,3)) # creer une matrice 3 3 de zéros
>>>B
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> B[0,1]=3 # affecter 3 à ligne 1 colonne 2
>>>B
array([[ 0.,  3.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>>B+3*A
array([[  3.,   9.,   9.],
       [  6.,  12.,  15.],
```

```
       [  3.,   6.,  15.]])
>>>A[1]
array([2, 4, 5])
>>>A[0,1]
2
>>>A[:,1] # colonne 2
array([2, 4, 2])
>>>np.dpt(A,B) # produit AB
array([[ 0.,  3.,  0.],
       [ 0.,  6.,  0.],
       [ 0.,  3.,  0.]])
```

7.14 Tris

```
L.sort() # tri par ordre croissant
L.reverse() #  change l'ordre
min(L) # ressort le min
max(a) # je vous laisse deviner
L.insert(i,a)# insert le terme a en position i
del L[i] # efface le terme en position i
L[i:j] # extrait la sous liste d'indice i à j-1
L.remove(a)# retirer l'élément a de L
L.count(i)# indique le nombre de terme de L de valeur i
```

7.15 Probabilité

Pour simuler du hasard et les lois classiques :

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

Les lois discrètes :

```
rd.randint(a,b,n) # une liste de n VA independantes de loi uniforme discrète sur (a,b
           -1)
rd.randint(a,b,(n,p)) # un tableau de n*p VA independantes de loi uniforme discrète sur
            (a,b-1)
rd.binomiale(n,p,N) #une liste de n VA independantes de loi binomiale de paramètre (n,p
           )
rd.binomial(n,p,(N,P))  # un tableau de n*p VA independantes
rd.geometric(p,N) #une liste de n VA independantes de loi geometrique de parametre p
d.geometric(p,(N,P))  # un tableau de n*p VA independantes
rd.poisson(lambda,N) #une liste de n VA independantes de loi Poisson de parametre
           lambda
rd.poisson(lambda,(N,P))  # un tableau de n*p VA independantes
```

Les lois continues

```
rd.exponential(1/lambda, (a,b))# tableau de taille (a,b) de VA independantes
# de loi exponentielle paramatre lambda
rd.normal(mu, sigma,(a,b))# tableau de taille (a,b) de VA independantes
#loi normale esperance mu et ecart type sigma
rd.gamma(a,theta,(a,b))# tableau de taille (a,b) de VA independantes
# gamma de parametre a parametre thata=1
rd.uniform(a,b, (n,p))# tableau de taille (a,b) de VA independantes
# loi uniforme sur [a,b]
rd.random(n,p)# tableau de taille (,p) de VA independantes
```

```
rd.rand()# nombre aléatoire dans [0,1]
```

7.16 Représentation graphique

Importer le modul `pyplot` :

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

Choisir les axes :

```
plt.axis('equal') # repère orthonormé
plt.axis([x_min,x_max,y_min,y_max]) # taille de la fenetre
plt.grid() # creer un cadrillage
plt.show() # affiche le graphe
plt.clf() # efface le graphe
```

Pour tracer il faut la liste de abscisses $X = [x_1, \dots x_n]$ et celle des ordonnée $Y = [f!x_1), \dots, f(y_n)]$

```
X=np.arrange(x_1,x_n,pas)
X=np.linspace(x_1,x_n,n)
Y=[f(x) for x in X] # creer la liste des abscisses à partir de celle des ordonnée

# on peut transformer f en la vectorisant et la faire fonctionner sur les listes
f=np.vectorize(f)
Y=f(X)
```

Remarquons que les fonctions `numpy` sont directement vectorisées.
Pour tracer en deux dimension :

```
plt.plot(X,Y)
plt.show()
# en cpmplément:
plt.plot(x,y,color='*',linestyle='*',marker='*') # les couleurs: 'g: vert, r: rouge, b:
            bleu
# le type de ligne: linestyle: '-' ligne continue,'--' ligne discontinue, ';' pointillé
           .   Le style des points:  marker:'+', ','o','v'
```

Les fonctions usuelles :

```
np.exp, np.log,np.sin,np.cos,no.sqrt,np.abs,np.floor
np.e, np.pi
```

Fonctions de deux variables et représentation en trois dimensions :

```
from mpl_toolkits.mplot3d import Axes3D # importer la fonction Axes3D
ax=Axes3D(plt.figure())
X=np.arrange(....)
Y=np.arrange(....)
X,Y=np.meshgrid(X,Y) # creer une grille
Z=f(X,Y) # ou vectoriser la fonction si, necessaire
ax.plot_surface(X,Y,Z)
plt.show()


plt.contour(X,Y,Z,[c_1,...c_n]) # les ligne f(x,y)=c_i
ax.quiver(X,Y,Z,,u,v,w))# tracer le gradien
plt.quiver(X,Y,u,v) # u=partial_1 f(X,Y), v=partial_2f(X,Y)
```