L2-S3 : UE Méthodes numériques

SEANCE 2

Structuration du code, Représentations graphiques

13 septembre 2024

Structurer un code

Un programme doit pouvoir être **lu et compris**, et éventuellement pouvoir être exécuté...Un programme doit pouvoir être **réutilisé et étendu par n'importe quel utilisateur** : vos collègues demain, vous-même dans six mois. Pour **maximiser la lisibilité** :

- on insère autant de **commentaires** que nécessaire (et même plus...)
- on adopte la structure suivante **dans chaque cellule**

```
#chargement des librairies (quand il y en a)
import numpy as np
from scipy.optimize import curve_fit

### definition des fonctions ###############
### independantes du corps du programme ###

def fonction1(arguments):
    """description de fonction1"""
    ...
    return resultat

def fonction2(arguments):
    """description de fonction2"""
    ...
    return resultat

### corps du programme ####################
#chargement des donnees et parametres
...

#traitement des donnees par les fonctions
...

#affichage des resultats (print, plot)
...
```

Note 1 : chaque bloc peut parfois être enregistré dans des cellules différentes du notebook pour la lisibilité.

Note 2 : les fonctions peuvent être des fonctions au sens mathématiques ($x \to y$) ou plus largement réaliser un certain nombre d'opérations informatiques / numériques, parfois sans `return` explicite.

Fonctions

Fonction : une suite d'instructions qui permet de faire réaliser à l'ordinateur une tâche bien spécifique. Elles permettent

- de segmenter le problème global en plusieurs sous-tâches simples
- que le corps du programme principal soit le plus court et simple possible
- d'augmenter la clarté du programme, la rapidité pour le débugguer et le développer vers des besoins plus complexes.
- de définir des morceaux de code réutilisables dans d'autres cellules ou dans d'autres programmes

Plus précisément, à partir d'un ensemble d'**arguments** en entrée, une fonction produit un ensemble de **résultats** en sortie.

Structure générale :

```
1 def ma_fonction(arguments):
2     ...
3     instructions
4     ...
5     return resultat
```

Fonctions

Note : pour les fonctions simples (une seule ligne d'instructions) :

```
ma_fonction = lambda arguments : resultat
```

Par exemple, les deux fonctions suivantes sont parfaitement équivalentes :

```
def add(x, y):
  return x + y
```

```
add = lambda x, y: x + y
```

Fonctions

- Le mot clé qui introduit la définition de la fonction est **def**, suivi du nom de la fonction complété par des **parenthèses** et le symbole **double points**.
- Les parenthèses contiennent les **arguments** de la fonction, mais peuvent parfaitement rester vides si la fonction ne nécessite pas d'argument.
- Puis les instructions sont obligatoirement écrites avec une **indentation**.
- L'instruction **return** renvoie le résultat de la fonction.

Une fois définie, la fonction peut être exécutée normalement comme une simple ligne d'instruction.

In [1]:
```
1 add(1, 1)
```

Out[1]:
```
2
```

Fonction à une variable

Exemple 1 : définissons une fonction parabolique $f(x) = 2x^2 + 3x + 4$, et évaluons-la en $x = -2.45$:

In [2]:
```
def parabole(x):
    f = 2*x*x+3*x+4
    return f

parabole(-2.45)
```

Out[2]:
```
8.655
```

Exemple 2 : Le mot clé **return** n'est pas obligatoire lorsque la fonction ne retourne aucun résultat, par exemple lorsque le résultat obtenu par la fonction est simplement écrit à l'écran avec l'instruction **print**.

In [3]:
```
def conversion_hms(secondes):
    h = secondes // 3600   # // div. euclid.
    m = (secondes  - h*3600) // 60
    s = secondes - h*3600 - m*60
    print(h,' heures ',m,' minutes ',s,'
    secondes')
conversion_hms(12345)
```

Fonction à une variable

Les variables définies à l'intérieur d'une fonction sont **locales**, c'est-à-dire que l'ordinateur ne connaît la variable qu'au sein de cette fonction. Elimine une importante source de bugs...

Exemple 3 : Conversion euro/dollar avec définition du facteur de conversion à l'intérieur de la fonction :

In [4]:
```
def conversion_euro_to_dollar(euros):
    facteur = 1.27
    dollars = euros * facteur
    print(euros,' euros = ',dollars,' dollars')
    return dollars

conversion_euro_to_dollar(30)
```

Out[4]:
```
30  euros =  38.1  dollars
38.1
```

In [5]:
```
print(facteur)
```

Out[5]:
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'facteur' is not defined
```

Fonction à une variable

N'importe quel type de variable peut être argument d'une fonction.

Par exemple une liste :

In [6]:
```
def somme(liste):
    tot = 0
    for element in liste :
        tot = tot + element
    return tot

somme([2,-3.5,7.12,99])
```

Out[6]:
```
104.62
```

ou beaucoup mieux : un numpy array !!!

In [7]:
```
import numpy as np

def somme(tableau):
    return tableau.sum()

somme(np.array([2,-3.5,7.12,99]))
```

104.62

Fonction à une variable

Important : lorsqu'on appelle la fonction `somme` avec un argument (ici np.array([2, -3.5, 7.12, 99]), alors on recopie les valeurs de cet argument dans la variable tableau, utilisée en interne par la fonction `somme`. Le nom de la variable fournie en argument lorsqu'on appelle `somme` est donc indépendant de tableau.
Il arrive cependant qus le progammeur choisisse d'appeler cette variable aussi tableau : on a alors deux variables différentes comportant les mêmes valeurs et portant le même nom, dont l'une n'est connue qu'en interne de la fonction. Cela ne pose donc pas de problème hormis un peu de confusion...

In [8]:
```
1 def somme(tableau):
2     return tableau.sum()
3
4 tableau = np.array([2,-3.5,7.12,99])
5 somme(tableau)
```

Out[8]:
```
104.62
```

Fonction à une variable

ou une chaîne de caractères :

In [9]:
```
def is_email(chaine):
    if '@' in chaine :
        print(chaine,"est une adresse email")
    else :
        print(chaine,"n'est pas une adresse email")

is_email("www.moncourspython.fr")
```

Out[9]:
```
www.moncourspython.fr n'est pas une adresse email
```

In [10]:
```
is_email("prenom.nom@moncourspython.fr")
```

Out[10]:
```
prenom.nom@moncourspython.fr est une adresse email
```

Fonction à plusieurs variables

Une fonction peut admettre plusieurs arguments, de types différents, séparés par des virgules.

Exemple : Convertir une heure en seconde :

In [11]:
```
def conversion_secondes(h,m,s):
    secondes = h*3600 + m*60 + s
    print(h,'h',m,'m',s,'s = ',secondes,'
    secondes')

conversion_secondes(3,25,45)
```

Out[11]:
```
3 h 25 m 45 s =  12345 secondes
```

Fonction à plusieurs variables

Certains des arguments peuvent être facultatifs, si on leur donne une **valeur par défaut** avec le signe $=$.

Exemple : Calcul d'un angle de réfraction par la troisième loi de Descartes, souvent le milieu 1 est l'air d'indice $n_1 = 1$.

In [12]:
```
def angle_refraction(theta1,n2,n1=1):
    theta2 = np.arcsin( n1*np.sin(theta1)/n2 )
    return theta2

angle_refraction(np.pi/4,1.33) # n1=1 par
    defaut
```

Out[12]:
```
0.5605584137424605 2
```

In [13]:
```
angle_refraction(np.pi/4,1.33,1.5)   # n1=1.5
```

Out[13]:
```
0.9231215703612742 9
```

mais attention, les **arguments facultatifs sont toujours placés après** les arguments obligatoires !

Fonction à plusieurs variables

Bien distinguer les arguments facultatifs les uns des autres lorsqu'il y en a plusieurs :
Syntaxe recommandée : spécifier *nom du paramètre facultatif=valeur :*

In [14]:
```
1 angle_refraction(np.pi/4,1.33, n1=1.5)
```

Note : pour les fonctions numériques, les angles sont définis en radians !

Fonction à plusieurs variables

Les **arguments facultatifs** peuvent être intéressants pour faire passer des variables booléennes qui changent le comportement de la fonction, par exemple si elle doit imprimée ou non le résultat à l'écran :

In [15]:
```
def conversion_euro_to_dollar(euros,verbose=
    False):
    # verbose signifie verbeux anglais
    facteur = 1.27
    dollars = euros * facteur
    if(verbose): print(euros,' euros = ',
    dollars,' dollars')
    return dollars

conversion_euro_to_dollar(30, verbose = True)
```

Out[15]:
```
30  euros =  38.1  dollars
38.1
```

In [16]:
```
conversion_euro_to_dollar(30)
```

Out[16]:
```
38.1
```

Fonction retournant plusieurs variables

Une fonction peut retourner plusieurs variables à la fois par le mot clé **return**. Dans ce cas, la fonction renvoie une liste des variables (ici des float) de sortie.

Exemple : rotation d'un angle dans un espace euclidien à 2D

In [17]:
```
def rotation(x,y,theta):
  #fonction rotation
    u = x*np.cos(theta) - y*np.sin(theta)
    v = x*np.sin(theta) + y*np.cos(theta)
    return u,v

u,v = rotation(2,3,0.1) # renvoie 2 variables
print(u,v)
```

Out[17]:
```
1.6905080806155672    3.184679329127734
```

In [18]:
```
Aprime = rotation(2,3,0.1) # renvoie une liste
print(Aprime)
```

Out[18]:
```
(1.6905080806155672, 3.184679329127734)
```

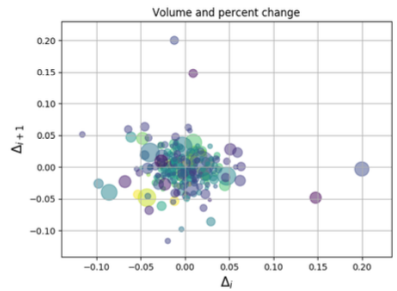
Graphiques : matplotlib.pyplot

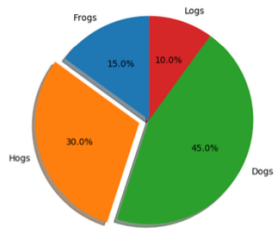
matplotlib : module permettant de tracer des graphiques.
pyplot : sous module pour les graphiques simples.

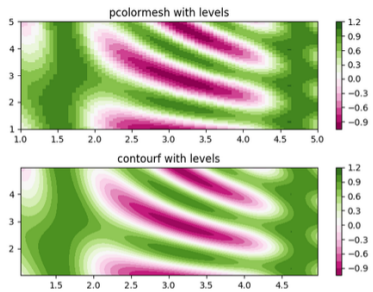
Discrétisation : tracer numérique $\rightarrow$ définir une gamme [`xmin` ; `xmax`] et un pas `dx`, puis calcul des $y_i = f(x_i)$ correspondant.

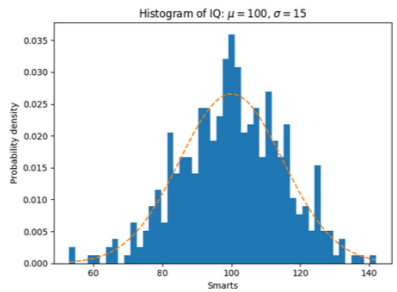
Graphiques : matplotlib.pyplot

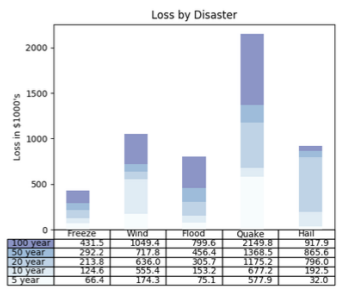
De nombreuses fonctions de **pyplot** permettent des tracés :
`https://matplotlib.org/tutorials/introductory/sample_plots.html`

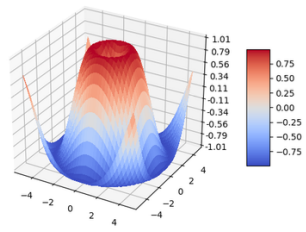










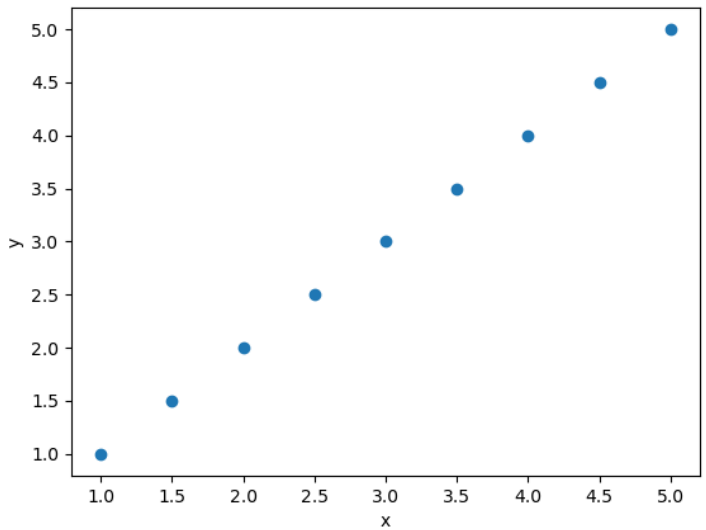


Note : ne pas apprendre les fonctions par coeur. Connaître leur nom et **les retrouver sur votre moteur de recherche favori** !

Diagramme de points : plt.scatter(x,y)

Exemple : “scatter plot” de la fonction identité sur [1 ;5] par pas de 0.5.

```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #initialise les donnees
6  x = np.arange(1,5.5,0.5)  #5.5 exclu
7  y = x                     #identite
8
9  #trace le graphique
10 fig = plt.figure() #nouvelle fenetre
11 plt.scatter(x,y)   #ajout de y=f(x)
12 plt.xlabel("x")    #titre abscisses
13 plt.ylabel("y")    #titre ordonnees
14 plt.show()         #affiche a l'ecran
```

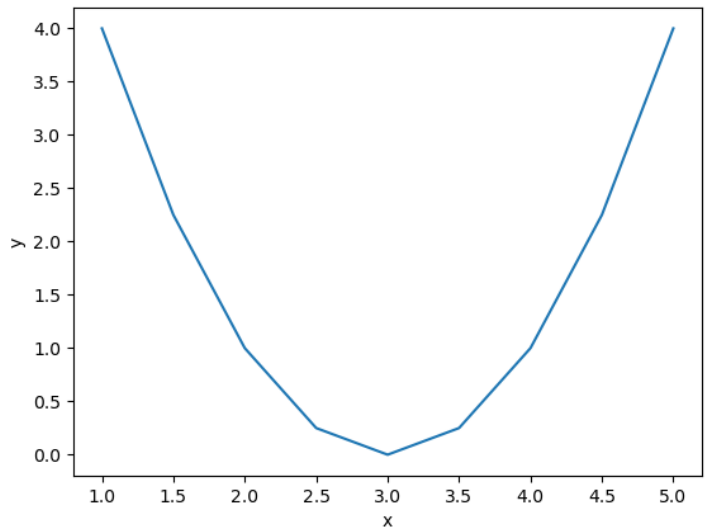


Tracé d'une courbe : plt.plot(x,y)

plt.plot(x,y, options) : fonction de tracé de base de `pyplot`. Permet de tracer des points, des segments par interpolation linéaire entre ces points.

Exemple : parabole centrée sur 3 sur [1 ;5] par pas de 0.5.

```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #initialise les donnees
6  x = np.arange(1,5.5,0.5) #5.5 exclu
7  y = (x-3.)**2            #parabole
8
9  #trace le graphique
10 fig = plt.figure() #nouvelle fenetre
11 plt.plot(x,y)      #segments y=f(x)
12 plt.xlabel("x")    #titre abscisses
13 plt.ylabel("y")    #titre ordonnees
14 plt.show()         #affiche a l'ecran
```



Par défaut : tracé de segments de droite, de couleur déterminée automatiquement.

Tracé d'une courbe : plt.plot(x,y)

Nombreuses options possibles pour la fonction plot, dont :

- **color** : fixe la couleur de la courbe, à choisir parmi 'green', 'blue', 'red', 'black',... ou 'g','b','r','k',...
- **linestyle** : fixe le style du trait, à choisir parmi '-' (trait continu), '- -' (tireté), ' :' (pointillé) ou '' (aucun trait)
- **marker** : choisit le style des points parmi '.' (point), 'o' (rond), '*' (étoile), etc
- **label** : donne un nom au tracé, utile pour construire une légende

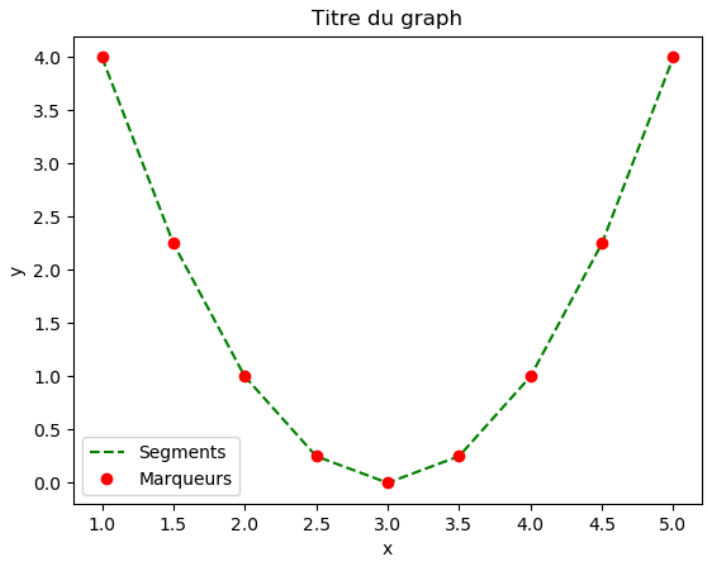
On peut **superposer** plusieurs graphiques en répétant l'instruction **plot**. La construction de **légende** liste automatiquement les **label** via la commande `plt.legend()`.

On peut ajouter un **titre au graphique** via `plt.title('Titre')`.

Tracé d'une courbe : plt.plot(x,y)

Exemple (on attend tous ces détails dans vos graphiques !).

```
#charge les librairies
import matplotlib.pyplot as plt
import numpy as np

#initialise les donnees
x = np.arange(1,5.5,0.5)  #5.5 exclu
y = (x-3.)**2             #parabole

#trace le graphique
fig = plt.figure()                #fenetre
plt.plot(x,y, linestyle='--',\
color='g', label = 'Segments') #ligne
plt.plot(x,y, marker='o', linestyle='',\
color='r', label = 'Marqueurs')#marqueurs
plt.legend()                      #legende
plt.xlabel("x")                   #titre x
plt.ylabel("y")                   #titre y
plt.title('Titre du graph')       #titre du graph
plt.show()                        #affiche
```

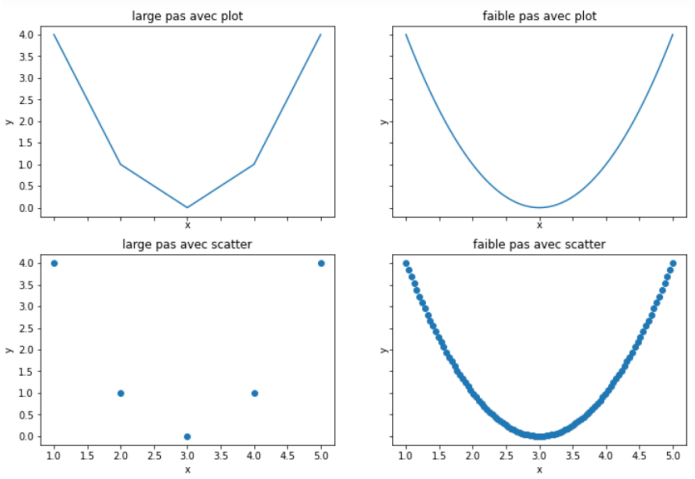


Note : a minima on attend des **titres sur les axes** et une **légende** avec labels associés si plusieurs courbes.

Pas de discrétisation

Utiliser un pas suffisamment faible pour les tracés.

```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #donnees a large pas (5 points)
6  x1 = np.linspace(1,5,5)  #5 inclus
7  y1 = (x1-3.)**2              #parabole
8
9  #donnees a faible pas (100 points)
10 x2 = np.linspace(1,5,100)  #5 inclus
11 y2 = (x2-3.)**2              #parabole
12
13 #Trace des graphiques
14 fig, ax = plt.subplots(2, 2, figsize=(12,8),
       sharex = True, sharey=True)
15 ax[0,0].plot(x1,y1)
16 ax[0,0].set_xlabel("x")
17 ax[0,0].set_ylabel("y")
18 ax[0,0].set_title("large pas avec plot")
19 ax[0,1].plot(x2,y2)
20 ax[0,1].set_xlabel("x")
21 ax[0,1].set_ylabel("y")
22 ax[0,1].set_title("faible pas avec plot")
23 ax[1,0].scatter(x1,y1)
24 ax[1,0].set_xlabel("x")
25 ax[1,0].set_ylabel("y")
26 ax[1,0].set_title("large pas avec scatter")
27 ax[1,1].scatter(x2,y2)
28 ax[1,1].set_xlabel("x")
29 ax[1,1].set_ylabel("y")
30 ax[1,1].set_title("faible pas avec scatter")
31 #affiche a l'ecran
32 plt.show()
```



Note : si plusieurs fenêtres, utiliser `plt.show()` **une seule fois**

Quelques fonctions utiles de pyplot

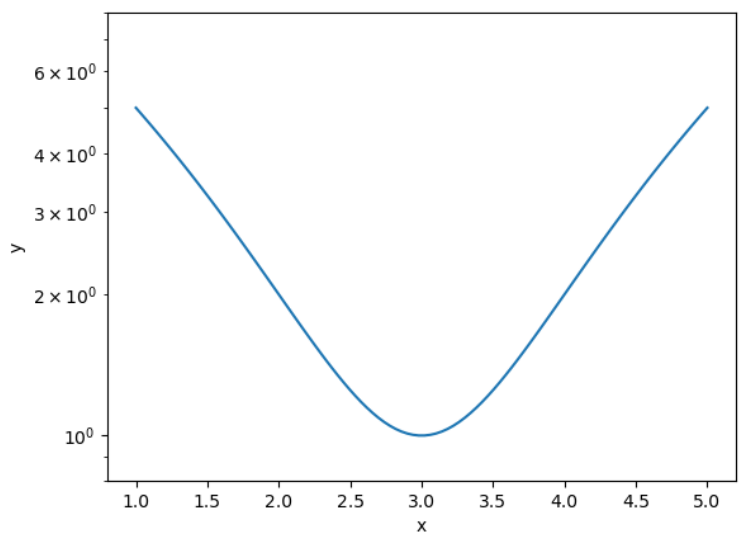
Par défaut, une fenêtre `pyplot` englobe tous les points. **(De)zoomer** :

- **plt.xlim(xmin,xmax)** : affiche entre xmin et xmax
- **plt.ylim(ymin,ymax)** : affiche entre ymin et ymax

Par défaut, échelle linéaire selon les deux axes. **Échelle log** :

- **plt.xscale('log')** : axe des abscisses en échelle logarithmique
- **plt.yscale('log')** : axe des ordonnées en échelle logarithmique

```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #donnees a faible pas (100 points)
6  x = np.linspace(1,5,100)  #5 inclus
7  y = 1+(x-3.)**2           #parabole
8
9  #trace le graphique
10 fig = plt.figure() #nouvelle fenetre
11 plt.plot(x,y)      #ligne y=f(x)
12 plt.xlabel("x")    #titre abscisses
13 plt.ylabel("y")    #titre ordonnees
14 plt.ylim(0.8,8)    #dezoom sur y
15 plt.yscale('log')  #echelle log sur y
16 plt.show()         #affiche
```



Tracé de mesures avec incertitudes : plt.errorbar()

Le module matplotlib contient aussi une fonction **errorbar()** pour tracer des points de mesures avec leurs **incertitudes**. Arguments :

- les abscisses et ordonnées des points de mesures (comme pour plot)
- les incertitudes sur les deux axes ou un seul (`xerr=`, `yerr=`)
- les autres options de tracé

Soit un fichier de données, 'data_RI.dat', dont les lignes indiquent la mesure d'un courant I traversant une résistance R alimentée par une tension $U = 5\,\mathrm{V}$ à ses bornes, pour différentes valeurs de R mesurées à l'ohmmètre (dont la notice indique une incertitude de 10%).

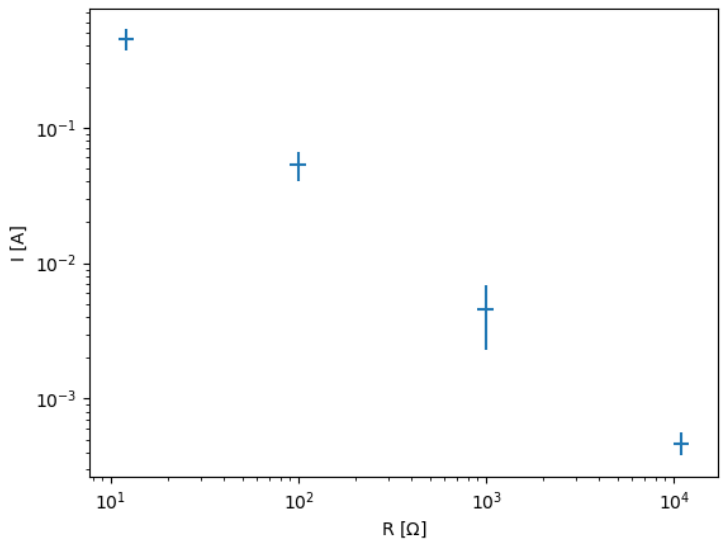
Contenu du fichier :

In [19]:
```
1 !cat data/data_RI.dat
```

Out[19]:
```
# R [ohms], R_err [ohms], I [A], I_err [A]
1.2099999999999964e+01 1.1999999999999956e
    +00 4.5000000000000111e-01
    8.0000000000000167e-02
1.0000000000000000e+02 1.0000000000000000e
    +01 5.2999999999999850e-02
    1.2999999999999940e-02
```

Tracé de mesures avec incertitudes : plt.errorbar()

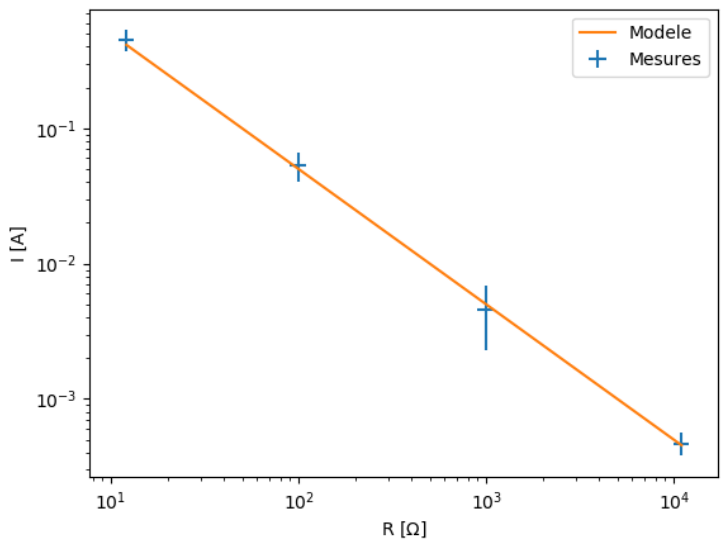
```
#charge les librairies
import matplotlib.pyplot as plt
import numpy as np

#charge les donnees depuis le fichier
data = np.loadtxt('data/data_RI.dat')
R = data[:, 0]    #1ere colonne
Rerr = data[:, 1] #2eme colonne
I = data[:, 2]    #3eme colonne
Ierr = data[:, 3] #4eme colonne

#trace le graphique
fig = plt.figure()          #nouvelle fenetre
plt.errorbar(R,I, xerr = Rerr, yerr = Ierr,\
linestyle = '')             #graph avec erreur
plt.xlabel(r'R [$\Omega$]') #titre abscisses
plt.ylabel('I [A]')         #titre ordonnees
plt.xscale('log')           #echelle log sur x
plt.yscale('log')           #echelle log sur y
plt.show()                  #affiche
```



Tracé de mesures avec incertitudes : plt.errorbar()

On peut bien entendu ajouter le modèle attendu : $I = U/R$

```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #charge les donnees depuis le fichier
6  data = np.loadtxt('data/data_RI.dat')
7  R = data[:, 0]    #1ere colonne
8  Rerr = data[:, 1] #2eme colonne
9  I = data[:, 2]    #3eme colonne
10 Ierr = data[:, 3] #4eme colonne
11
12 #charge le modele
13 U = 5 #volts
14 Rplot = np.linspace(np.min(R),np.max(R),100)
15 Iplot = U/Rplot
16
17 #trace le graphique
18 fig = plt.figure()
19 plt.errorbar(R,I, xerr = Rerr, yerr = Ierr,\
20 linestyle = '',label='Mesures')
21 plt.plot(Rplot, Iplot,label='Modele') #modele
22 plt.legend()          #2 graphs => legende !
23 plt.xlabel(r'R [$\Omega$]')
24 plt.ylabel('I [A]')
25 plt.xscale('log')
26 plt.yscale('log')
27 plt.show()
```



Histogrammes : plt.hist()

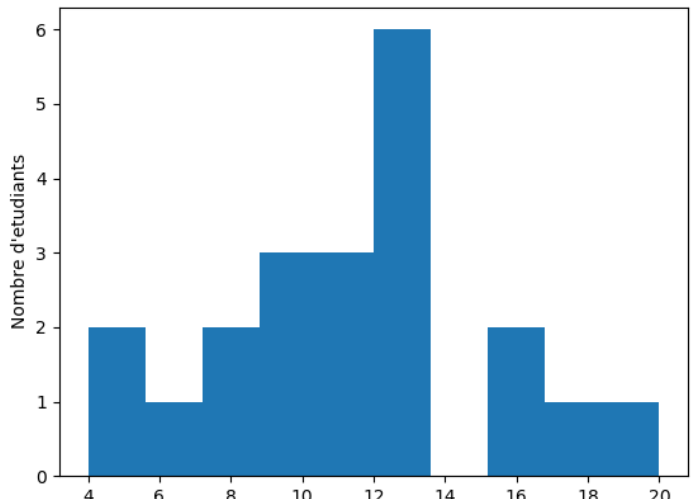
Histogramme : nombre d'occurrences d'une quantité dans un échantillon en fonction de la quantité, tracé avec la fonction **hist()**, prenant pour arguments :

- la série de données dont on veut former l'histogramme : **x**
- les intervalles ou le nombre d'intervalles désirés par l'argument **bins**

In [20]:
```
!cat data/data_L2.dat
```

Out[20]:
```
8.5 12.5 11 10 13.5 5 18 16.5 12 13 11 9 10 4
20 12.5 10.5 7 8.5 13.5 15.5
```

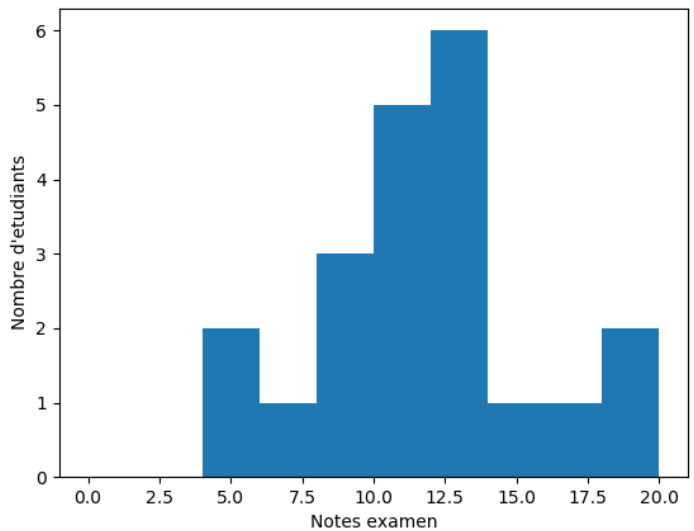
```
#charge les librairies
import matplotlib.pyplot as plt
import numpy as np

#charge les donnees
notes = np.loadtxt('data/data_L2.dat')

#charge les parametres
nombre_de_bins = 10

#trace le graphique
fig = plt.figure()
plt.hist(notes,bins=nombre_de_bins) # histogramme
plt.xlabel('Notes examen')
plt.ylabel("Nombre d'etudiants")
plt.show()
```



Histogrammes : plt.hist()

Plutôt que le nombre d'intervalles, **bins** peut aussi contenir les bornes des intervalles. Par exemple, 10 intervalles de largeur 2 allant de 0 à 20 :

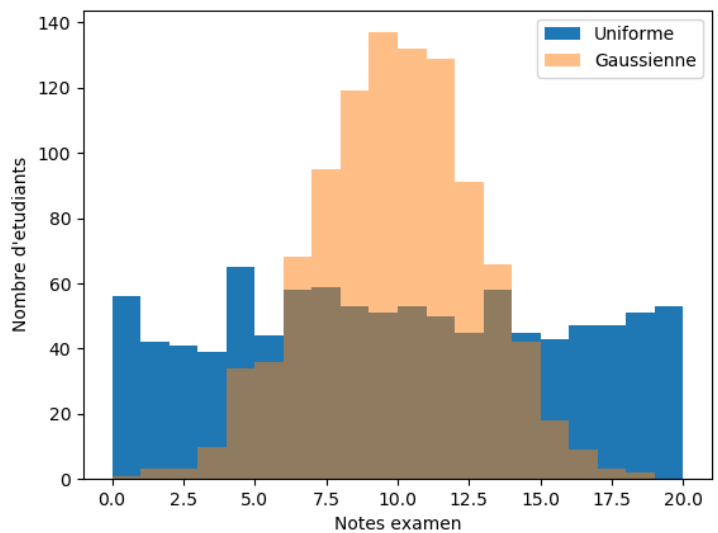
```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #charge les donnees
6  notes = np.loadtxt('data/data_L2.dat')
7
8  #charge les parametres
9  bornes_de_bins = np.linspace(0,20,11)
10
11 #trace le graphique
12 fig = plt.figure()
13 plt.hist(notes,bins=bornes_de_bins) #
       histogramme
14 plt.xlabel('Notes examen')
15 plt.ylabel("Nombre d'etudiants")
16 plt.show()
```



Note sur les nombres aléatoires

numpy possède une libraire **random** permettant de tirer des nombres pseudo-aléatoires suivant certaines distributions (voir la doc), e.g. uniforme (**np.random.uniform(xmin, xmax, size)**), exponentielle (**np.random.exponential(λ, size)**), gaussienne (**np.random.normal(μ, σ, size)**).

```
1  #charge les librairies
2  import matplotlib.pyplot as plt
3  import numpy as np
4  #initialise la graine du generateur
5  np.random.seed(123)
6  #donnees aleatoires uniformes
7  data1 = np.random.uniform(0,20,1000)
8  #donnees aleatoires gaussiennes
9  data2 = np.random.normal(10,3,1000)
10 #charge les parametres
11 bornes = np.linspace(0,20,21)
12 #trace le graphique
13 fig = plt.figure()
14 plt.hist(data1,bins=bornes, label="Uniforme")
15 plt.hist(data2,bins=bornes, label="Gaussienne"
       ,\
16 alpha=0.5) #alpha: transparence
17 plt.legend()
18 plt.xlabel('Notes examen')
19 plt.ylabel("Nombre d'etudiants")
20 plt.show()
```



Commentaires finaux : fonctions et numpy

Maximisez la compatibilité avec numpy. Préférez les fonctions de `numpy` (ensemble de valeurs) à celles de `math` (simple scalaire ⇒ boucle).

Mais...

Code qui fonctionne :

```python
#charge les librairies
import matplotlib.pyplot as plt
import numpy as np

def expo(x, L):
  """fonction exponentielle"""
  return np.exp(-x/L)

### corps du programme ###################
#parametre
L = 1
#variable a afficher
x = np.linspace(0,10,100)
y = expo(x, L)
#trace le graphique
fig = plt.figure()
plt.plot(x,y, label=f'L={L}')
plt.legend()
plt.xlabel('x')
plt.ylabel('y = exp(-x/L)')
plt.show()
```

Code qui plante, **pourquoi ?**

```python
#charge les librairies
import matplotlib.pyplot as plt
import numpy as np

def expo(x, L):
  """fonction exponentielle"""
  return np.exp(-x/L)

### corps du programme ###################
#parametre -> SEUL CHANGEMENT
L = np.arange(0.5,2.5,0.5)
#variable a afficher
x = np.linspace(0,10,100)
y = expo(x, L)
#trace le graphique
fig = plt.figure()
plt.plot(x,y, label=f'L={L}')
plt.legend()
plt.xlabel('x')
plt.ylabel('y = exp(-x/L)')
plt.show()
```

Commentaires finaux : fonctions et numpy

Solution à ce stade : boucle sur L pour le calcul et le tracé.

```
#charge les librairies
import matplotlib.pyplot as plt
import numpy as np

def expo(x, L):
  """fonction exponentielle"""
  return np.exp(-x/L)

### corps du programme ###################
#parametre
L = np.arange(0.5,2.5,0.5)
#variable a afficher
x = np.linspace(0,10,100)
y = [expo(x, Li) for Li in L]
#trace le graphique
fig = plt.figure()
for i, yi in enumerate(y):
  plt.plot(x,yi,label=f"L={L[i]}")
plt.legend()
plt.xlabel('x')
plt.ylabel('y = exp(-x/L)')
plt.show()
```