Guide de survie de Python pour le calcul scientifique

François Ducrot

27 décembre 2018

Table des matières

1 Qu'est-ce que Python ?

Python est un langage de programmation, qui existe en deux branches : la branche 2.x (actuellement en 2.7, et qui n'est plus censée évoluer) et la branche 3.x (en version 3.6 au moment de l'écriture de ce texte). Dans cette introduction, nous nous intéresserons spécifiquement à la branche 3.

1.1 Les scripts python

Voici un exemple de fichier écrit en Python :

somme_euler.py

```
1 #Calcul de la somme d'Euler
2
3 def somme(n):
4     s=0
5     for k in range(1,n+1):
6         s=s+1/k**2
7     return s
8
9 print(somme(1000000))
```

Quand on tape dans un terminal `python somme_euler.py`, l'interpréteur python exécute ligne à ligne le fichier (constitué ici d'une définition de fonction, et de l'affichage d'un résultat obtenu par cette fonction) et affiche le résultat demandé.

1.2 Structure d'un script python

En examinant le script précédent, on voit tout de suite quelques aspects de la structure :

Un bloc, comme celui constitué des lignes 3-7 est constitué d'une déclaration (ligne 3), terminée par un symbole "deux points", et d'un corps de bloc (lignes 4-7), qui est entièrement déterminé par l'indentation . A l'intérieur de ce bloc, les lignes 5-6 constituent un bloc dont le corps (ligne 6) est à son tour indenté.
Il est vital que toutes les indentations d'un même bloc logique aient la même taille, sous peine d'une erreur d'exécution. La bonne pratique veut que toutes les indentations utilisées dans un programme soient de même taille (en général quatre espaces). Un traitement de texte adapté à Python doit savoir gérer correctement les indentations, généralement réalisées par la touche <Tab>.

On peut (et on doit) mettre des commentaires ; ceux-ci sont précédés du symbole #.

1.3 La console python

Quand, dans une fenêtre de terminal, on lance la commande `python`, on se retrouve devant une console de commande, dont l'invite est >>>. Chaque commande python que l'on tape est évaluée au moment où on tape la touche <Enter>. Les différentes variables qui auront été initialisées sont stockées en mémoire pour le temps de la session, comme on le voit dans l'exemple suivant :

```
[ducrot@pc-ducrot ~]$ python
Python 3.6.0 |Anaconda custom (64-bit)|  on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> b = a ** 2
>>> print(a+b)
6
>>>
```

On peut aussi utiliser une console plus évoluée, en tapant `ipython` :

```
In [1]: a = 2

In [2]: b = a**2

In [3]: a+b

   6

In [4]: run somme_euler.py

   1.64493306684877
```

Par rapport à la console de base, `ipython` apporte entre autres une aide, une complétion automatique (avec la touche <Tab>), et la possibilité d'évaluer un script à l'intérieur de la session en cours (comme `somme_euler.py`, dans l'exemple précédent).

Par ailleurs, on dispose dans `ipython` d'un certain nombre de commandes, dites « commandes magiques », toutes préfixées par le symbole %, qui permettent d'interagir avec le système. Citons par exemple :

- changer de répertoire : `%cd nouveau_repertoire`
- lister le contenu du répertoire courant : `%ls`
- revenir à une console vide : `%clear`
- se mettre en mode d'affichage graphique de matplotlib `%matplotlib` (sera vu plus loin)

Dans la suite de ce document, les expérimentations seront effectuées avec `ipython`.

On notera que lorsqu'on tape une commande dans la ligne de commande de la console, on obtient le résultat de la console (ainsi, quand on a tapé `a+b`, on a eu la réponse `6` ; en revanche, si on veut qu'un script affiche au cours de son exécution des valeurs de calculs, il faut lui dire explictement : `print(a+b)` .

1.4 Charger des modules

Python est livré par défaut avec une multitude de modules permettant d'effecuer beaucoup de tâches, mais ceux-ci ne sont pas automatiquement chargés au démarrage :

```
In [5]: pi
```

```
NameError: name 'pi' is not defined
```

Importons la constante `pi` contenue dans le module `math` :

```
In [6]: from math import pi ; print(pi)
```

```
3.141592653589793
```

On peut aussi importer l'ensemble du module `math` et utiliser ses fonctions :

```
In [7]: import math ; print(math.cos(0))
```

```
1.0
```

On peut enfin importer toutes les fonctions du module `math` :

```
In [8]: from math import * ; print(e**pi)
```

```
23.140692632779263
```

On accède ainsi à toutes les fonctions et constantes du module `math` sans avoir besoin de les préfixer par le nom du module. C'est pratique, mais dangereux, car on ne sait pas précisément ce qui a été importé.

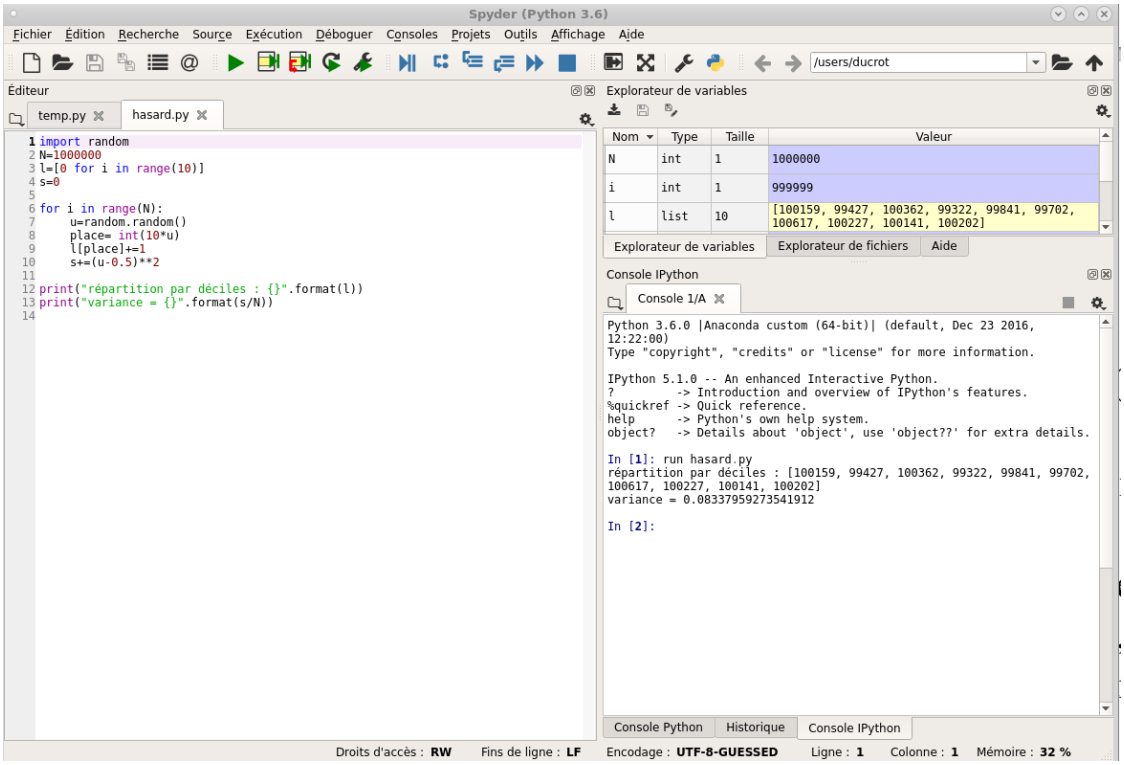
Dans la suite, on va dans un premier temps s'attacher à comprendre le fonctionnement de Python sans faire appel à des modules spécifiques, puis on regardera certains modules adaptés au calcul scientifique :

— `numpy` pour les tableaux de nombres réels,
— `matplotib` pour la visualisation des graphiques.

2 Comment travailler en Python ?

On peut envisager différentes façon de travailler :

— En éditant un fichier `mon_fichier.py` avec un éditeur de texte adapté à Python (`emacs`, `geany`, `atom`, `kate`,...) et en l'exécutant par une commande `python mon_fichier.py`.
— En combinant l'exécution de commandes dans la console `ipython` et l'édition de fichiers `*.py`.
— Avec un environnement de développement intégré (IDE), qui réunit dans une même fenêtre une console `ipython`, un éditeur de texte, et d'autres outils (explorateur de fichiers, explorateur de variables, outil de visualisation, fenêtre de déboggage, fenêtre d'historique,...). Dans une optique de calcul scientifique, on choisira ici d'utiliser `spyder` :



2.1 Installer Python sur son ordinateur

Le plus simple pour avoir une version de Python pleinement fonctionnelle est d'installer une distribution complète de Python . On vous invite ici à utiliser la distribution Anaconda, téléchargeable sur le site https://www.anaconda.com/downloads, et qui fournit tous les modules scientifiques. Elle est disponible pour les trois plateformes Windows, MacOS et GNU/Linux ; choisir la version python 3 adaptée à votre système.

3 Objets, variables et affectation

3.1 Objets et types

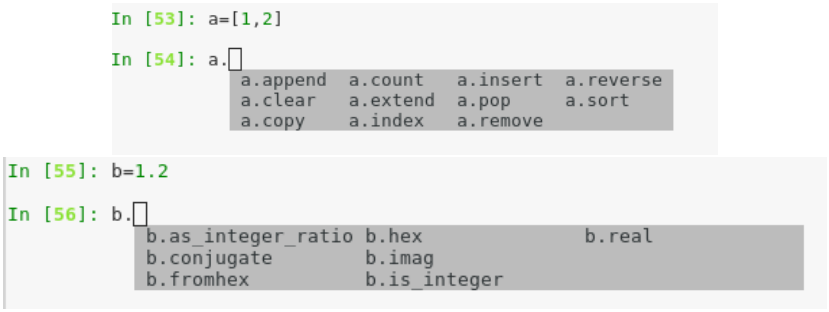
Dans Python , tout est objet. Un objet est un espace de mémoire contenant un certain nombre d'informations. Un objet a un type, qui décrit la façon dont l'objet est stocké en mémoire, et les opérations qu'on peut lui appliquer. Le type de l'objet est déterminé dynamiquement par Python et n'a pas à être défini par l'utilisateur.

```
In [9]: type(1), type(1.), type(1+5j), type("azerty"), type(True), type(None),
           type([1,2]), type((1,2)), type({1,2}), type({'a':1,'z':3})
```

```
(int, float, complex, str, bool, NoneType, list, tuple, set, dict)
```

Chaque objet a un identifiant indiquant l'adresse mémoire occupée par cet objet. On accède à l'identifiant d'un objet par la commande `id(objet)` ; même si cette commande sert peu dans la pratique de la programmation, elle permet de bien comprendre comment fonctionne Python , comme on le verra dans la suite.

Chaque objet dispose d'un certain nombre de méthodes qu'on peut lui appliquer, qui dépendent du type de l'objet. Dans les deux copies d'écrans, on a créé une liste et un nombre flottant, et on a demandé à `ipython` de proposer les méthodes qu'ont peut appliquer aux objets :



3.2 Affectation

Une affectation est effectuée par une commande

`variable = cible`

Cette opération calcule la valeur du terme de droite et crée un lien (pointeur) du nom `variable` vers l'objet résultant du calcul du terme de droite.

On peut efffectuer des affectations multiples comme par exemple :

`a, b, c = 1, 2, 3`

ou encore

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

Dans les affectations suivantes, les variables pointent vers le même objet :

In [10]:
```
a=1; b=1; id(1), id(a),  id(b)
```

```
(140569174792448, 140569174792448, 140569174792448)
```

Alors que dans les affectations suivantes, qui travaillent sur des listes, c'est plus compliqué :

In [11]:
```
a=[1,2]; b=[1,2]; c=a; id(a),  id(b), id(c)
```

```
(140568913711304, 140568942481928, 140568913711304)
```

Ici `a` et `b` ont la même valeur, mais ne pointent pas vers le même objet, ce qu'on peut vérifier :

In [12]:
```
a == b , a is b
```

```
(True, False)
```

3.3 Objets modifiables ou non modifiables

Certains objets sont dits non modifiables. Ainsi dès qu'on cherche à modifier une variable `a` de type nombre, l'objet sous-jacent n'est plus le même, comme on le vérifie en regardant `id(a)` avant et après l'opération.

En revanche une liste est modifiable :

In [13]:
```
a=[1,2] ; a, id(a)
```

```
([1, 2, 5], 140568913700936)
```

In [14]:
```
a.append(5) ; a, id(a)
```

```
([1, 2, 5], 140568913700936)
```

On verra plus loin que cette notion d'objets modifiables est importante quand on effectue des affectations.

4 Les différents types élémentaires de données

4.1 Nombres et booléens

On dispose

— de nombres entiers de taille arbitraire :

In [15]:
```
a=145698745632145698745632145698745632 1 ; a**2, type(a)
```

```
(21228124478780695328381540162532284286313236355926846032453167378828550 41,int)
```

Notons que le quotient et le reste de la division euclidienne de `a` par `b` se calculent par `a//b` et `a%b`

— de nombres réels ou complexes en virgule flottante, représentés sur 64 bits

In [16]:
```
a=1.2 ; b= int(a) ; c=b/2 ; d = 2+3j; a**2, type(a), b, type(b),
    c, type(c) , d, type(d)
```

```
(1.44, float, 1, int, 0.5, float, (2+3j), complex)
```

On notera que les nombres imaginaires s'écrivent sous la forme d'un réel auquel est accolée la lettre `j`. Ainsi le complexe i se code `1j`.

— de constantes booléennes `True` et `False`

In [17]:
```
a = True ; b = False ; type(a), a and b , a or b , not (a and b)
```

```
(bool, False, True, True)
```

— Signalons enfin l'objet un peu particulier `None`. Si vous écrivez une fonction qui ne rend rien en sortie, sa valeur de sortie existera néanmoins, et vaudra `None`.

4.2 Listes

Une liste est une suite ordonnées de termes, non nécessairement de mêmes types, séparés par des virgules et encadrés par des crochets :

In [18]:
```
a = [1,2,3] ; b = [5,6,'blabla',[8,7]]; c=[] ; print(len(a), len(b), len(c),
   a+b)
```

```
3 4 0 [1, 2, 3, 5, 6, 'blabla', [8, 7]]
```

La fonction `len(a)` retourne la longueur de la liste `a` (même chose plus loin avec les tuples et les chaînes de caractères).

La liste vide correspond à `[]`, et la concaténation est obtenue par +.

Partons d'une chaîne (on utilise ici le générateur `range`, qui sera détaillé plus loin) :

In [19]:
```
a = list(range(3,12)); print(a)
```

```
[3, 4, 5, 6, 7, 8, 9, 10, 11]
```

On peut en extraire des tranches (slice) :

In [20]:
```
print(a[0], a[2], a[1:3], a[5:], a[0::3], a[-1:-5:-1])
```

```
3 5 [4, 5] [8, 9, 10, 11] [3, 6, 9] [11, 10, 9, 8]
```

On notera que les objets indexables (listes, tuples, chaînes de caractères) de Python sont indexés à partir de 0, contrairement à ce qui se passe dans dans certains autres langages. De plus, les éléments peuvent aussi être repérés par des indices négatifs, en partant de la fin ; ainsi `a[-1]` et `a[-2]` sont les dernier et avant-dernier termes de la liste `a`.

On peut affecter à une tranche une nouvelle valeur, et ainsi modifier la liste initiale :

In [21]:
```
a[0::3] = ['x','y','z'] ; print(a)
```

```
['x', 4, 5, 'y', 7, 8, 'z', 10, 11]
```

Une liste est donc un objet modifiable ; on peut modifier une partie de la liste, mais la liste est toujours le même objet.

In [22]:
```
a = [1,2,3] ; print(a , id(a))
```

```
[1, 2, 3] 139851268344136
```

In [23]:
```
a[2] = 7 ;  print(a , id(a))
```

```
[1, 2, 7] 139851268344136
```

In [24]:
```
a=[7,8,9,10] ; a.append(11) ; print(a) ; a.insert(2,17) ; print(a)
```

```
[7, 8, 9, 10, 11]
[7, 8, 17, 9, 10, 11]
```

In [25]:
```
x = a.pop(1) ; print("x =", x," , ", "a =", a)
```

```
x = 8  ,  a = [7, 17, 9, 10, 11]
```

In [26]:
```
a.remove(10) ; print(a)
```

```
[7, 17, 9, 11]
```

In [27]:
```
a.reverse() ; print(a)
```

```
[11, 9, 17, 7]
```

In [28]:
```
b = sorted(a) ; print(b)
```

```
[7, 9, 11, 17]
```

4.3 Tuples

Un tuple est la même chose qu'une liste, mais c'est un objet non modifiable. Cela se représente par une suite de termes, séparés par des virgules et encadrés par des parenthèses. Le fait d'être non modifiable entraine que la représentation en machine d'un tuple est plus efficace que celle d'une liste. Par ailleurs, un tuple peut être utilisé comme clé d'un dictionnaire alors qu'une liste ne peut pas l'être. Donc, quand on n'envisage pas de modifier en place une liste d'éléments, il est préférable de la représenter par un tuple que par une liste.

In [29]:
```
a=(1,2,3,4); print(a, type(a))
```

```
(1, 2, 3, 4), tuple
```

4.4 Chaînes de caractères

Une chaîne de caractère est une expression encadrée par des apostrophes ou des guillemets.

In [30]:
```
a = "Salut les amis !" ; print(len(a), a[0], a[15])
```

```
16 S !
```

On peut extraire le i-ème caractère de `a` (on numérote à partir de zéro) par `a[i]`.
On ne peut pas modifier un élément d'une chaîne :

In [31]:
```
a[0]="s"
```

```
TypeError: 'str' object does not support item assignment
```

Autrement dit, une chaîne, comme un tuple, n'est pas un objet modifiable.

4.5 Ensembles

Un ensemble sous Python est une structure de données qui vérifie les mêmes propriétés que la structure mathématique usuelle : ses objets sont uniques, et ne sont pas ordonnés. Il est représenté par une suite d'objets, éventuellement de différents types, séparés par des virgules ; cette suite est encadrée par des accolades.

In [32]:
```
a={3,2,3,5,'bla','bla'} ; print(a , type a, id(a))
```

```
{2, 3, 5, 'bla'} set 140340462134440
```

Puisque les éléments ne sont pas ordonnés, on ne peut pas accéder à un élément déterminé par un indice :

In [33]:
```
a[1]
```

```
TypeError: 'set' object does not support indexing
```

Les ensembles sont des objets modifiables :

In [34]:
```
a.remove('bla') ; a.add('toto') ; print(a, id(a))
```

```
{2, 3, 5, 'toto'}, 140340462134440
```

On peut effectuer sur les ensembles les opérations ensemblistes usuelles : `union`, `difference`, `intersection`, `symmetric_difference`, etc, qu'on utilise avec la syntaxe :

In [35]:
```
a.union({5,7})
```

```
{2, 3, 5, 7, 'toto'}
```

Notons que pour définir l'ensemble vide, la commande n'est pas `{}` (qui crée un dictionnaire), mais `set()`.

4.6 Dictionnaires

Un dictionnaire est une suite d'associations clé :valeur, qui est défini par la syntaxe suivante :

In [36]:
```
dico={"mirza":"chien", "felix":"chat", "medor":"chien", 1:"nombre"}
```

In [37]:
```
dico["felix"], dico[1]
```

```
('chat', 'nombre')
```

Mais si on appelle une clé inexistente, on a droit à un message d'erreur :

In [38]:
```
dico[2]
```

```
KeyError: 2
```

On peut modifier un dictionnaire en rajoutant une entrée :

In [39]:
```
dico["ninja"]="tortue"
```

ou en suprimant une entrée (la fonction pop rend en même temps la valeur qui a été enlevée) :

In [40]:
```
dico.pop(1)
```

```
'nombre'
```

In [41]:
```
dico
```

```
{'felix': 'chat', 'medor': 'chien', 'mirza': 'chien', 'ninja':'tortue'}
```

On remarquera que l'ordre des entrées n'est pas celui qu'on a donné : un dictionnaire n'est pas une structure ordonnée.
On peut aussi accéder à l'ensemble des composants d'un dictionnaire sous forme de listes :

In [42]:
```
dico.items()
```

```
dict_items([('mirza', 'chien'), ('felix', 'chat'), ('medor', 'chien'),
    ('ninja', 'tortue')])
```

In [43]:
```
dico.keys()
```

```
dict_keys(['mirza', 'felix', 'medor', 'ninja'])
```

In [44]:
```
cles = list(dico.keys()) ; valeurs =  list(dico.values())
print(cles, valeurs)
```

```
['felix', 'medor', 'mirza', 'ninja'] ['chat', 'chien', 'chien', 'tortue']
```

Les clés d'un dictionnaire ne peuvent pas être des objets modifiables, donc en particulier pas des listes ou des ensembles, mais peuvent être des tuples : on voit là un des intérêts de la notion de tuple.

4.7 D'un type à un autre

Lors d'une lecture de données, la donnée entrée est souvent une chaîne de caractère, et on a besoin de la transformer en un objet du type souhaité. Voici un exemple :

In [45]:
```
a = "123" ; print(a, a+a, type(a))    # a est une chaine
```

```
'123' '123123' str
```

In [46]:
```
b = int(a) ; print(b, b+b, type(b))   # b est un entier
```

```
123 246 int
```

In [47]:
```
c = float(a) ; print(c, type(c))  # c est un nombre en vigule flottante
```

```
123.0 float
```

On peut transformer une chaine de caractères en la liste de ses caractères :

In [48]:
```
n = "azerty" ; l = list(n) ; print(l, type(l))
```

```
['a', 'z', 'e', 'r', 't', 'y'] list
```

Naviguons maintenant un peu entre listes, tuples et ensembles :

In [49]:
```
liste = [1,9,8,1,7,9,9] ; nuplet=tuple(liste)
ensemble = set(liste) ; liste2 = list(ensemble)
print(liste, nuplet, ensemble, liste2)
```

```
[1, 9, 8, 1, 7, 9, 9] (1, 9, 8, 1, 7, 9, 9) {8, 1, 9, 7} [8, 1, 9, 7]
```

On voit sur cet exemple qu'on ne maitrise pas du tout la façon dont sont ordonnés les éléments dans un ensemble.

4.8 Affectation et copie

Il faut passer un peu de temps pour bien comprendre ce que fait une affectation. Effectuons quelques expérimentations :

In [50]:
```
a = 1 ; b = 1; c = a  ; id(a), id(b), id(c)
```

```
(139707911960832, 139707911960832, 139707911960832)
```

Ici les variables a, b, c pointent vers le même objet (l'entier 1).

In [51]:
```
a = 2  ; id(a), id(b), id(c)
```

```
(139707911960864, 139707911960832, 139707911960832)
```

La variable a pointe maintenant vers un nouvel élément (l'entier 2), alors que b, c pointent toujours vers le même objet.

In [52]:
```
a = [1,2,3]; b = a ; c = [1, 2, 3]  ; id(a), id(b), id(c)
```

```
(139707678849544, 139707678849544, 139707678848776)
```

Ici les variables a et b pointent vers le même objet, alors que la variable c pointe vers un autre objet, même s'il a la même valeur.

In [53]:
```
b[0] = 5 ; id(a), id(b), a , b, c, a == c
```

```
(139707678849544, 139707678849544, [5, 2, 3], [5, 2, 3], [1, 2, 3], True)
```

Quand on modifie l'objet vers lequel pointe b, cela modifie donc la valeur de a ; en revanche c n'est pas affecté. Le comportement observé sur cet exemple est lié au fait que les listes sont des objets modifiables. On peut souvent avoir envie d'effectuer une copie d'une liste, de façon à ce qu'elle soit ensuite entièrement indépendante de son original :

In [54]:
```
d = a[:] ; print(id(a) ; id(d) ; a is d)
```

```
139707678849544 139707651074312 False
```

Maintenant a et d ne pointent plus vers le même objet et on peut les modifier indépendamment l'une de l'autre.

In [55]:
```
d[1] = 7 ;  print(a , d)
```

```
[5, 2, 3] [5, 7, 3]
```

A vrai dire, ce procédé effectue une copie seulement avec une profondeur 1 (« shallow copy ») :

In [56]:
```
x = [[1]] ; y = x.copy() ; y[0][0]=2 ; print(x, y)
```

```
[[2]] [[2]]
```

Pour avoir deux copies vraiment indépendantes, il faut utiliser la fonction deepcopy du module copy :

In [57]:
```
import copy ; x = [[1]] ; y = copy.deepcopy(x) ; y[0][0]=2 ; print(x, y)
```

```
[[1]] [[2]]
```

5 Structure du code

Regardons un petit script, qui définit deux fonctions, l'une qui décide si un nombre est premier ou non, et l'autre qui parcourt les nombres entiers inférieurs à une borne donnée, et affiche quand ils sont entiers.

nombres_premiers.py

```
1  def premier(n):
2      if n == 2:
3          return True
4      d=2
5      while d**2 < n+1:
6          if n%d == 0:
7              return False
8          d = d+1
9      return True
10
11 def premiers(N):
12     for n in range(2,N+1):
13         if premier(n):
14             print(n,"est premier")
```

Ce script met en jeu :
- des définitions de fonctions,
- des boucles itératives `for` et `while`,
- des structures conditionnelles `if`.

5.1 Fonctions

Un bloc de définition de fonction suit la syntaxe :

```
def ma_fonction(mes_arguments):
    instruction_1
    instruction_2
```

Les arguments de la fonction sont séparés par des virgules, et une fonction peut éventuellement ne pas avoir d'arguments. Une fonction peut rendre une valeur en sortie ; dans ce cas, on met une ligne

```
    return valeur
```

L'instruction `return` arrête le déroulement de la fonction et retourne immédiatement la valeur indiquée. C'est ce qui est utilisé aux lignes 3, 7 et 9 de `nombres_premiers.py`.
Mais on peut aussi écrire une fonction qui affiche des résultats, ou exécute des actions, sans retourner explicitement de sortie (en réalité, elle retourne alors la valeur `None`). Regardons ainsi la suite de commandes

```
In [58]: def f(x):
             print(x**2)

In [59]: def g(x):
             return x**3

In [60]: f(2)
4

In [61]: g(2)
8

In [62]: f(g(2))
64

In [63]: g(f(2))
```

```
TypeError: unsupported operand type(s) for ** or pow(): 'NoneType' and 'int'
```

Ce message d'erreur est normal : `f(2)` affiche un résultat, mais rend la valeur `None` ; donc on ne peut pas calculer `g(f(2))`.

5.1.1 Documenter une fonction

On peut ajouter à une fonction un texte de commentaire, que les pythonistes appellent « docstring », et qui sera affiché dans l'aide. Ce texte peut être constitué de plusieurs lignes et est encadré par des triples guillemets « """ ».

In [64]:
```
def nombre_aleatoire():
    """
    Retourne un nombre aleatoire
    en utilisant  un algorithme secret
    """
    return 42

help(nombre_aleatoire)
```

```
nombre_aleatoire()
    Retourne un nombre aleatoire
    en utilisant  un algorithme secret
```

5.1.2 Les arguments d'une fonction

Le script suivant montre différentes façons de définir les arguments d'une fonction :

arguments-fonctions.py

```
1  # les arguments et leur nombre sont precises
2  def somme(x,y,z):
3      return x+y+z
4
5  #un nombre arbitraire d'arguments non nommes
6  def nombre_arguments(*arguments):
7      print('vous avez rentre', len(arguments), 'arguments')
8
9  #un nombre arbitraire d'arguments  nommes
10 def presentation(**dico_arguments):
11     nom = dico_arguments.get('nom', None)
12     prenom = dico_arguments.get('prenom', None)
13
14     if (nom and prenom):
15         print(f"mon nom est {nom}, {prenom} {nom}")
16     elif (prenom and not(nom)):
17         print(f"Mon prenom est {prenom}")
18     elif (not(prenom) and nom):
19         print(f"Mon nom est {nom}")
20     else:
21         print("Je ne suis personne")
```

— La fonction `somme` prend trois arguments, caractérisés par leur position.
— La fonction `nombre_arguments` prend un nombre arbitraire d'arguments et les stocke dans un n-uplet que j'ai appelé `arguments` (choix arbitraire) ; on utilise ensuite ce n-uplet comme on veut dans la fonction.
— La fonction `presentation` prend un nombre arbitraire d'arguments, sous la forme `cle = valeur`, dans un ordre quelconque, et les stocke dans un dictionnaire, appelé ici `dico_arguments` de manière arbitraire. Elle peut être appelée en particulier par
  — `presentation(nom='Bond', prenom='James')`
  — `presentation(prenom='James')`
  — `presentation(nom='Bond')`
  — `presentation()`

— `presentation(nom='Bond', prenom='James', age='35')` : ici l'âge n'est pas utilisé.

5.1.3 Les fonctions lambda

Il existe également une façon très compacte pour définir une fonction $x \mapsto f(x)$ est la notation lambda : `lambda x: f(x)`. Donnons un exemple :

In [65]:
```
def multiplie_par(a):
    return lambda x: a*x
f = multiplie_par(2)  # f est la fonction x->2*x
```

La fonction `multiplie_par()` prend un nombre a et rend la fonction $x \mapsto ax$.
L'utilisation des fonctions lambda est pratique quand on veut créer yune fonction auxiliaire à l'intérieur d'un programme. En dehors de ce cas d'utilisation, elle rend difficile la lecture du code python.

5.2 Tests et structures conditionnelles

Un test est une expression qui rend une valeur booléenne `True` ou `False` :

```
x == y     # test d'égalité
x < y      # inégalité stricte
x <= y     # inégalité large
x != y     # inégalité
x = y = z  # plusieurs égalités
(x = y) or ( z < t)
(x = y) and not ( z < t)
```

On utilise alors un test dans une structure conditionnelle :

```
if condition_booleenne:
    instruction1
elif autre_condition:
    instruction2
else:
    instruction3
```

Les parties `elif` et `else` sont optionnelles, mais forcément dans cet ordre, et il peut y avoir autant d'options `elif` que l'on veut.

5.3 Boucle Tant Que

La boucle `while` exécute une opération tant qu'une condition booléenne est vérifiée :

```
while condition_booleenne:
    instruction1
    instruction2
```

La condition booléenne est une expression qui vaut `True` ou `False`, et la boucle tournera tant que la condition aura la valeur `True`. C'est généralement le résultat d'un test. Mais la condition booléenne peut aussi être une constante `True`, et dans ce cas, la boucle tournera indéfiniment, à moins de tomber sur une instruction d'arrêt.
Dans les lignes 5-9 du script `nombres_premiers.py`, la boucle `while` tourne tant qu'elle ne rencontre pas une instruction `return`. Si elle arrive jusqu'à la fin, le script rencontre alors une expression `return True` et s'arrête également.

5.4 Boucle Pour et itération

La boucle pour parcourt itérativement toutes valeurs données dans l'argument. La syntaxe est

```
for indice in objet_iterable:
    instruction1
    instruction2
```

L'objet itérable qui apparait après le mot clé `in` est un objet qu'on peut parcourir itérativement. Il peut s'agir d'une liste, d'une chaîne de caractères, d'un objet générateur. Regardons quelques exemples :

In [66]:
```
for i in  [1,3,7]:
    print(i)
```

```
1
3
7
```

In [67]:
```
for i in "azerty":
    print(i+i)
```

```
aa
zz
ee
rr
tt
yy
```

In [68]:
```
a=range(3) ; print(a, type(a))
```

```
range(0, 3) <class 'range'>
```

In [69]:
```
for i in a:
    print(i**2)
```

```
0
1
4
```

L'objet `range` est ce qu'on appelle un générateur. À la différence d'une liste, qui est stockée entièrement en mémoire, un générateur calcule les valeurs successives juste au moment où on en a besoin. L'intérêt d'un tel objet est son efficacité, due à sa faible occupation de la mémoire.

5.5 Compréhension de listes

À partir du générateur `range`, on peut construire une liste par la fonction `list`

In [70]:
```
list(range(1,10))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

qui aurait aussi pu être obtenue par la syntaxe plus générale

In [71]:
```
a = [i for i in range(1,10)]
```

Cette notation reprend à peu de chose près l'écriture usuelle d'un ensemble par compréhension. Cela s'étend encore à une expression du genre

```
[ f(i) for i in objet_iterable if condition(i) ]
```

comme par exemple :

In [72]:
```
a=[ i**2 for i in range(1,20) if i%3 == 1 ]; a
```

```
[1, 16, 49, 100, 169, 256, 361]
```

6 Entrées-Sorties

6.1 Lire des entrées au clavier

```
a=input("entrer un nombre :\n")
```

6.2 Faire des sorties formatées à l'écran

On peut utiliser la commande `print()`, qui affiche à l'écran les différents arguments de la parenthèse, après les avoir évalués. Par exemple

In [73]:
```
k = 2 ; print("Le carre de",k,"est egal a",k**2)
```

```
Le carre de 2 est egal a 4
```

Cependant, la méthode recommandée depuis la version 3.6 de Python est d'utiliser le mécanisme des « f-strings »(chaînes formatées). Ainsi, le script

carres.py

```
1 for k in range(1,3):
2     print(f'Le carre de {k} est egal a {k**2}')
```

affiche

```
Le carre de 1 est egal a 1
Le carre de 2 est egal a 4
```

6.3 Lire et écrire dans un fichier

Pour stocker ou importer des données massives, on sera forcé de passer un fichier. Imaginons un fichier `animaux.txt` contenant trois lignes :

```
mirza
felix
medor
```

Pour accéder à ce fichier en lecture, on va l'ouvrir et lui associer un objet python :

```
f = open("animaux.txt","r")
```

On effectue ensuite des opérations de lecture :

```
contenu = f.read() ; print(contenu)
```

Après avoir effectué des opérations, il faut absolument fermer le fichier :

```
f.close()
```

On peut, et c'est recommandé, utiliser un bloc, défini par le mot clé `with`, qui ferme automatiquement le fichier à la sortie du bloc :

lecture.py

```
1 # Lit le fichier animaux.txt et l'imprime a l'ecran :
2
3 with open("animaux.txt","r") as f:
4     for s in f:
5         print(s)
```

Les modes de la fonction `open` sont `"r"` (lecture), `"w"` (écriture : efface le contenu du fichier si celui-ci existait), `"a"` (écriture à la suite : n'efface pas le contenu du fichier).

On aurait pu par exemple créer le fichier `animaux.txt` par le script :

ecriture.py

```
1 # ecrit le fichier animaux.txt :
2
3 liste = ["mirza", "felix", "medor"]
4 with open("animaux.txt","w") as f:
5     for nom in liste:
6         f.write(nom)
7         f.write("\n")  # pour le saut de ligne
```

On dispose d'un certain nombre de fonctions de lecture, qui avancent séquentiellement dans le fichier. Une fois qu'on est arrivé au bout, il n'y a plus qu'à le fermer. Voici quelques exemples :

- `f.read()` charge dans une unique chaîne de caractères : `'mirza\nfelix\nmedor\n\n'`.
- `f.readlines()` crée une liste de chaînes de caractères, une par ligne du fichier : `['mirza\n', 'felix\n', 'medor\n', '\n']`
- `f.readline()` crée un itérateur rendant à chaque itération une ligne du fichier ; on le parcourt donc avec une boucle `for`.

Et de même, pour un fichier ouvert en écriture :

- `f.write(chaine_de_caracteres)` écrit toute la chaîne de caractère dans le fichier. Si on fait plusieurs appels successifs à cette fonctions, les chaînes seront rajoutées bout à bout, d'où l'utilité de mettre des caractères `\n`.
- `f.writelines(liste)` écrit successivement chacun des termes de la liste.

7 Controler l'exécution d'un programme

7.1 Temps d'exécution

Quand on fait du calcul scientifique, il est important de mesurer l'efficacité du code qu'on écrit. Dans ce paragraphe, on va tester le temps d'exécution du code suivant, qui fabrique une liste de 100000 éléments, les permute de manière aléatoire, en utilisant la commande `shuffle` du module `random`, puis les réordonne :

melange_aleatoire.py

```
1 def test():
2     l = list(range(100000))
3     random.shuffle(l)
4     l.sort()
```

On peut tester la rapidité de ce programme dans l'interpréteur `ipython` :

```
In [74]: import random
         %timeit test()
```

```
73.9 ms per loop (mean  std. dev. of 7 runs, 10 loops each)
```

On peut utiliser des solutions plus élaborées faisant appel aux modules `time` ou `timeit`.

7.2 Les erreurs

Si le programme tombe sur une erreur, comme une division par zéro, il va s'arrêter en affichant un message d'erreur. On peut intercepter ce processus avec la structure `try/except`. La partie du code qui est dans le bloc `try` est effectuée si ce bloc ne rencontre pas d'erreur, si le bloc rencontre une erreur, le programme n'est pas arrêté, mais envoyé dans la partie `except`. Celle-ci effectue différentes opérations, qui peuvent dépendre du type d'erreur, si on a indiqué le code d'erreur correspondant.

division.py

```
1  #Effectue une division
2
3  numerateur = input("Entrez le numerateur : \n")
4  denominateur = input("Entrez le denominateur : \n")
5
6  try:
7      print ("valeur du quotient :", numerateur / denominateur)
8  except NameError:
9      print("Une des variables n'a pas ete definie.")
10 except TypeError:
11     print("Une des variables possede un type incompatible.")
12 except ZeroDivisionError:
13     print("Le denominateur est egal a 0.")
14 except:
15     print("Erreur non determinee.")
```

Le programme aurait aussi pu être suivi d'un bloc `finally:`, qui sera exécuté quels que soient les resultats precedents.

On peut aussi avoir envie de déclencher une erreur. On peut utiliser pour cela `raise` comme dans la fonction suivante :

```
def ma_fonction(param):
    if param not in (1, 2, 3):
        raise ValueError("'param' can only be either 1, 2 or 3")
    # reste du code
```

Lors de l'appel de `ma_fonction(5)`, elle déclenchera une erreur :

```
Traceback (most recent call last):
  File "<ipython-input-3-bcd6e8653c83>", line 1, in <module>
    votre_super_fonction(4)
  File "<ipython-input-2-46fc7cd18c42>", line 3, in ma_fonction
    raise ValueError("'param' can only be either 1, 2 or 3")
ValueError: 'param' can only be either 1, 2 or 3
```

8 Écrire son propre module

La façon la plus simple de créer un module est de fabriquer un fichier ayant une extension `.py`, comme dans l'exemple suivant :

module_exemple.py

```
"""
Un module d'exemple qui ne fait pas grand chose
mais qui le fait bien
"""

from math import acos, sqrt, pi

p = pi

def somme(x,y):
    """ calcule la somme de deux nombres """
    return x+y

def carre(x):
    return x*x

if __name__ == '__main__':
    x = y = 1
    alpha = acos(x/sqrt(somme(carre(x),carre(y))))
    print(alpha / p)
```

On peut maintenant importer le module et utiliser ses fonctions et ses constantes :

In [75]:
```
from module_exemple import *
print( somme(1,2) , p, sqrt(2) )
```

```
3  3.141592653589793  1.4142135623730951
```

On remarquera dans le fichier un dernier groupe `__name__ == '__main__':` . Ce groupe n'est pas exécuté lors de l'importation du module, mais si on lance le script :

In [76]:
```
run module_exemple.py
```

```
0.25000000000000006
```

9 Tableaux de nombres avec le module numpy

Une documentation précieuse sur l'aspect calcul scientifique se trouvent sur https://scipy.org/.

Le module numpy introduit un nouveau type ndarray de tableau multidimensionnel, dont toutes les coordonnées sont de même type (entiers, flottants sur un nombre de bits fixé, mais aussi booléens ou chaînes de caractères). L'utilisation de ces tableaux est beaucoup moins généraliste que les celle des listes usuelles de Python , mais du fait de sa spécificité, est algorithmiquement plus efficace. Numpy offre des fonctions spécifiques qui utilisent essentiellement la même syntaxe que les logiciels de calcul scientifiques comme Matlab/Scilab.

On fait ici le choix d'importer numpy par l'invocation :

In [77]:
```
import numpy as np
```

On crée ensuite des objets ndarray :

In [78]:
```
a=np.array([[1,2,3],[4,5,6]])  ;  a
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [79]:
```
type(a), a.shape, a.dtype
```

```
(numpy.ndarray, (2, 3), dtype('int64'))
```

Il s'agit d'un ndarray de taille 2×3, dont les éléments sont des entiers sur 64 bits.

On peut imposer le type d'un tableau ; par exemple, o,n veut un tableau en nombres flottants :

In [80]:
```
a = np.array([1,2,4],dtype=float) ; a
```

```
array([ 1.,  2.,  4.])
```

On peut envisager des tableaux en plus de deux dimensions :

In [81]:
```
a = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]]) ; a
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Le profil de a est ici (2,2,3). On voit à cette occasion qu'il devient vite nécessaire d'utiliser des moyens de construction plus puissants que l'entrée terme à terme.

On notera tout de suite qu'un tableau numpy est un objet modifiable (on le verra plus loin), mais que sa taille et le type d'objets dont il est constitué, et donc son emprise mémoire, sont déterminés à sa création, et ne peuvent pas être modifiés.

9.1 Quelques briques

In [82]:
```
a = np.arange(6) ; b = np.arange(0.5,2,0.3) ; a, b
```

```
(array([0, 1, 2, 3, 4, 5]), array([ 0.5,  0.8,  1.1,  1.4,  1.7]))
```

In [83]:
```
np.linspace(1,2,7)
```

```
array([ 1.        ,  1.16666667,  1.33333333,  1.5       ,  1.66666667,
        1.83333333,  2.        ])
```

In [84]:
```
np.zeros(5)
```

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

In [85]:
```
np.ones(5), np.full(4,3.14)
```

```
(array([ 1.,  1.,  1.,  1.,  1.]), array([ 3.14,  3.14,  3.14,  3.14]))
```

Pour fabriquer un tableau multidimensionnel, on peut construire un tableau unidimensionnel et le retailler ensuite.

In [86]:
```
a=np.arange(12) ; b=a.reshape(4,3); b
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [87]:
```
a=np.arange(12) ; b=a.reshape(2,3,2); b
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

9.2 Données extérieures à numpy

On peut transformer une liste ordinaire de PYTHON en tableau numpy :

In [88]:
```
lst = [1.414,1.732,3.14] ; a = np.asarray(lst,dtype=float) ; a
```

```
array([ 1.414,  1.732,  3.14])
```

Partons d'un fichier `vecteur.txt`, avec trois lignes contenant respectivement 1, 2, 3. On peut le lire et faire de son contenu un vecteur

In [89]:
```
a = np.loadtxt("vecteur.txt",dtype=float) ; a
```

```
array([ 1.,  2.,  3.])
```

De même, en partant d'un fichier

```
1,2,3
2,3,4
3,5,6
```

In [90]:
```
a = np.loadtxt("vecteur.txt",dtype=float,delimiter=",") ; a
```

```
array([[ 1.,  2.,  3.],
       [ 2.,  3.,  4.],
       [ 3.,  5.,  6.]])
```

et en sens inverse, pour obtenir un fichier texte avec le contenu du tableau `a`, et cette fois-ci séparé par des deux-points :

In [91]:
```
np.savetxt("sortie.txt",a,delimiter=":")
```

9.3 Extraction d'un tableau

Partons du tableau suivant, de type `'int64'` et de profil `(4,5)` :

In [92]:
```
tab = np.arange(20).reshape(4,5) ; tab
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

On accède à un élément du tableau en notant les indices avec des crochets (attention, on numérote à partir de 0) :

In [93]:
```
tab[1,1]
```

```
6
```

La ligne d'indice 1 :

In [94]:
```
tab[1,:]
```

```
array([5, 6, 7, 8, 9])
```

La colonne d'indice 2 (on notera que le résultat est un vecteur ligne) :

In [95]:
```
tab[:,2]
```

```
array([ 2,  7, 12, 17])
```

on peut aussi prendre des tranches plus compliquées, par exemple une ligne et une colonne sur deux :

In [96]:
```
tab[1:4:2,0:5:2]
```

```
array([[ 5,  7,  9],
       [15, 17, 19]])
```

Il faut bien comprendre que les tranches d'un tableau, telles que nous les avons extraites, ne sont pas des copies en mémoire,mais juste des vues d'un morceau du tableau. Elles vont nous permettre de modifier le tableau.

9.4 Modifier un tableau

Modifions des éléments du tableau précédent :

In [97]:
```
tab[1,1] = 3.14 ; tab[1,2] = 7  ; tab
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  3,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

La modification `tab[1,1] = 3.14` a été faite seulement modulo un passage à la partie entière, car le tableau est un tableau d'entiers, et ne peut pas accepter de valeurs réelles.

In [98]:
```
new = tab/2 ; new
```

```
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 2.5,  1.5,  3.5,  4. ,  4.5],
       [ 5. ,  5.5,  6. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5]])
```

La variable `new` pointe vers un nouvel objet dont le `dtype` est cette fois `float64`, mais `tab` est inchangé. Modifions maintenant une tranche :

In [99]:
```
tab[1:4:2,0:5:2] = 33 ; tab
```

```
array([[ 0,  1,  2,  3,  4],
       [33,  3, 33,  8, 33],
       [10, 11, 12, 13, 14],
       [33, 16, 33, 18, 33]])
```

9.5 Utilisation de masques

Un masque est un tableau booléen bâti sur un tableau donné ; par exemple pour notre `tab` préféré :

In [100]:
```
tab > 5
```

```
array([[False, False, False, False, False],
       [ True, False,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]], dtype=bool)
```

On peut utiliser un masque pour sélectionner une tranche et agir spécifiquement sur elle :

In [101]:
```
tab [ tab > 5 ] = -1
```

```
array([[ 0,  1,  2,  3,  4],
       [-1,  3, -1, -1, -1],
       [-1, -1, -1, -1, -1],
       [-1, -1, -1, -1, -1]])
```

9.6 Opérations sur les tableaux

Partons de deux tableaux de même profil (ici `(2,2)`) :

In [102]:
```
a = np.array([[1,2],[3,4]]) ; b = np.array([[4,5],[6,7]])
```

Opérations termes à termes

In [103]:
```
a + b , a*b , 2*a
```

Produit matriciel

In [104]:
```
a.dot(b)
```

```
array([[16, 19],
       [36, 43]])
```

Pour comprendre la notation : on applique la méthode `dot` de l'objet `a`, en prenant pour argument `b`.
On peut aussi concaténer des tableaux, suivant une direction (`axis`) :

In [105]:
```
append(a,b,axis=1)
```

```
array([[1, 2, 4, 5],
       [3, 4, 6, 7]])
```

Ici `axis=1` met les matrices côte à côte, mais `axis=0` les superposerait verticalement.
Un procédé pratique est la « propagation »ou « broadcasting ».

In [106]:
```
ligne = np.array([1,2,3]) ; colonne = np.array([[10],[20],[30]]) ;
  ligne + colonne
```

```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

`numpy` a propagé la ligne et la colonne en des matrices 3×3, de façon à pouvoir additionner les matrices ainsi obtenues.

9.7 Fonctions universelles

Les fonctions universelles (`ufunc`) sont des fonctions préprogrammées qui s'appliquent à l'ensemble d'un tableau :

- `np.sqrt(a)`, `np.exp(a)`, `np.sin(a)` calculent la racine carré, l'exponentielle, le sinus de chaque élément du tableau et rendent un tableau de même profil.
- `np.random.rand(4)` rend un tableau de longueur 4 de nombres pseudoaléatoires suivant la loi de répartition uniforme sur $[0, 1]$.
- `np.random.rand(4,5)` rend un tableau de profil 4×5 de nombres pseudoaléatoires suivant la loi normale.
- `mat.T` rend la matrice transposée de la matrice `mat`.
- Le module `np.linalg` propose un certain nombre de fonctions standard d'algèbre linéaire. Par exemple `np.linalg.det(mat)`, `np.linalg.eig(mat)` calculent le déterminant et des vecteurs propres de la matrice carrée `mat`.

10 Visualisation scientifique avec le module matplotlib

Le module `matplotlib` est dédié à la visualisation de diagrammes scientifiques. Pour explorer toutes ses possibilités , le plus simple est de visiter la galerie de matplotlib, qui présente des graphes et les codes qui les ont produits : https://matplotlib.org/gallery.html
L'utilisation de `matplotlib` se fera essentiellement au travers de la la console `ipython`.
Dans toute la suite, on suppose qu'on a au préalable importé `numpy` et le sous-module `pyplot` de `matplotlib`, chargé de la visualisation 2D /

In [107]:
```
import numpy as np
import matplotlib.pyplot as plt
```

10.1 Différentes façons d'utiliser la console ipython

Imaginons qu'on cherche à tracer le graphe de $f : [0, 1] \to \mathbb{R},\ x \mapsto x^2$. On va commencer par fabriquer une vecteur `x` de valeurs entre 0 et 1, le vecteur `y` des carrés de ces valeurs :

In [108]:
```
x = np.linspace(0,1,200) ; y = x*x
```

On a plusieurs façon d'afficher le graphe

10.1.1 Sans rien paramétrer dans ipython

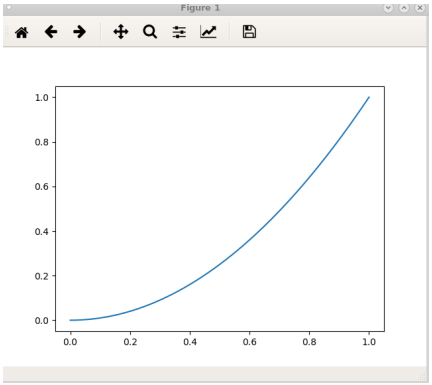
In [109]:
```
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa51c0f1710>]
```

Un objet dessin a été créé, mais pas affiché. Il faut donc l'afficher :

In [110]:
```
plt.show()
```

et on voit apparaitre une fenêtre :



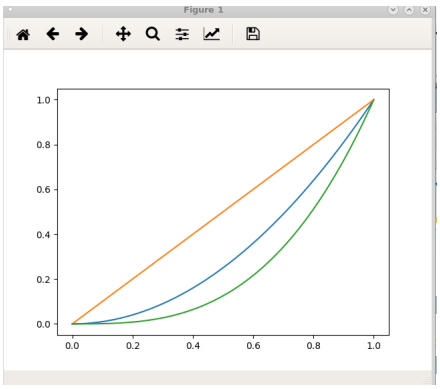
On n'a plus la main, et il faudra fermer la fenêtre graphique pour récupérer la main dans ipython.

10.1.2 En paramétrant ipython pour matplotlib

Au début de la session, on choisit le mode d'affichage `matplotlib` en entrant la commande %`mathplotlib`. Maintenant, dès qu'on lance la commande `plt.plot`, la fenêtre graphique s'ouvre, et python retrouve la main. On peut donc par exemple rentrer une deuxième commande

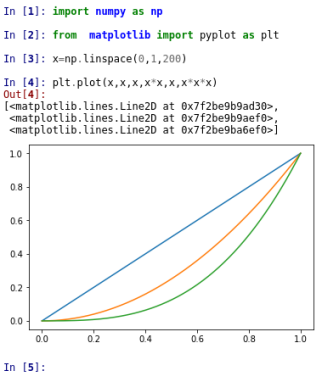
In [111]:
```
plt.plot(x,x,x,x*x*x)
```

et les nouveaux graphes de $x \mapsto x$ et $x \mapsto x^3$ se superposent au précédent :



10.1.3 A l'intérieur de Spyder

Dans l'IDE spyder, la console ipython est paramétrée de telle façon que les graphiques soient affichés à l'intérieur de la console (mode `inline`). Cela donne le rendu suivant :



C'est élégant, mais pas forcément très pratique. Si on veut revenir à un affichage `offline`, on peut entrer au début de la session la commande :
%`mathplotlib qt5`
Cela fonctionne sous inux ; sous Windows, cela dépend...

10.1.4 Sauvegarder les images dans des fichiers

Depuis la fenêtre graphique, on peut choisir par un menu d'enregistrer l'image dans un fichier ; le format du fichier de sortie sera déterminé par l'extension `.png`, `.pdf`, etc, choisie pour son nom.
Mais on peut aussi le faire grâce à une instruction :

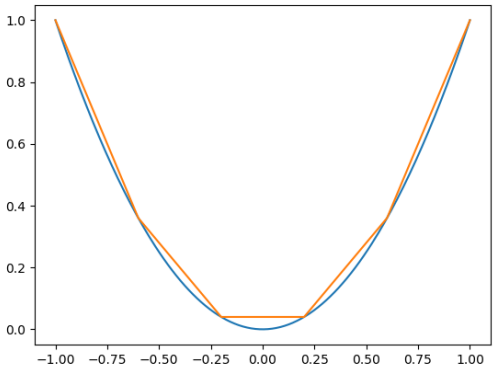
```
plt.savefig("fichier.png",dpi=300,orientation='portrait',transparent=True')
```

Seul le nom du fichier est nécessaire, les autres arguments n'étant à utiliser que si on veut paramétrer finement l'image de sortie (voir toutes les options dans la documentation)

10.2 Graphiques en 2D

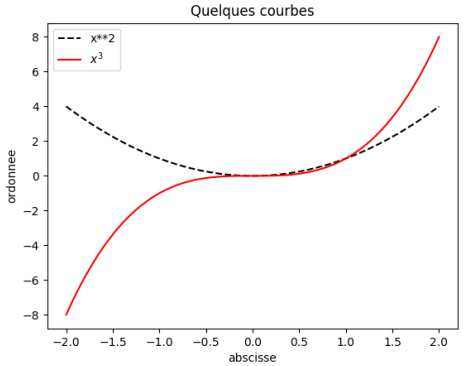
Pour tracer une courbe, on trace des points (x_i, y_i), reliés par des segments. Pour cela, on construit le vecteur `x` des x_i et le vecteur `y` des y_i, et on utilise la commande `plt.plot(x,y)`. On peut aussi conjuquer dans un même dessin le graphe de `y` en fonction de `x`, le graphe de `yy` en fonction de `xx`.

In [112]:
```
x = np.linspace(-1,1,200); xx = np.linspace(-1,1,6) ; y = x**2 ; yy = xx**2
plt.plot(x,y,xx,yy)
```



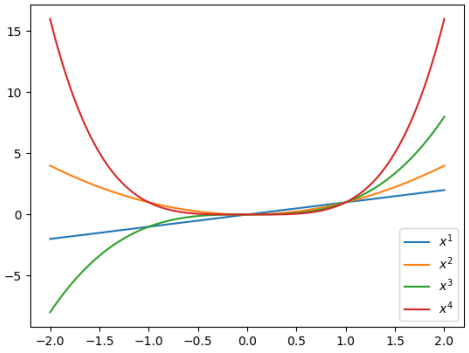
On peut aussi mettre quelques informations sur le graphe :

In [113]:
```
x = np.linspace(-2, 2, 100)
plt.plot(x,x*x, 'k--', label='x**2')
plt.plot(x, x*x*x, 'r', label=r'$x^3$')
plt.title('Quelques courbes')
plt.xlabel('abscisse')
plt.ylabel('ordonnee')
plt.legend()
```



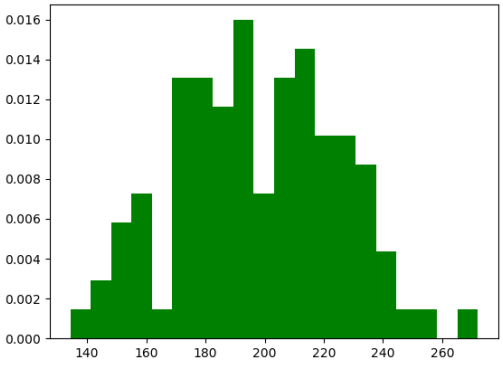
On voit ici que la deuxième légende est écrite TEX.
On peut aussi dessiner plusieurs courbes grâce à une boucle :

In [114]:
```
for i in range(1,5):
    plt.plot(x,x**i,label=f"$x^{i}$")
plt.legend()
```



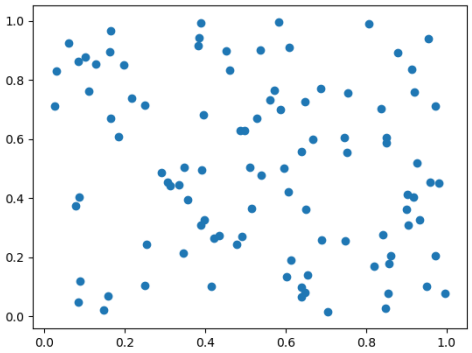
Un histogramme d'un échantillon de nombres pseudo-aléatoires (avec `plt.hist`) :

In [115]:
```
x = np.random.normal(200, 25, size=100)
plt.hist(x, 20, normed=1, histtype='stepfilled', facecolor='g')
```



Un nuage de points (avec `plt.scatter`) :

In [116]:
```
x = np.random.rand(100) ;  y = np.random.rand(100) ; plt.scatter(x,y)
```



10.3 La gestion de l'affichage dans matplotlib

Pour gérer précisément l'affichage dans matplotlib, il faut utiliser le modèle objet de `matplotlib`. Un affichage graphique est constitué d'une fenêtre (`figure`), d'un ou de plusieurs systèmes d'axes de coordonnées à l'intérieur de la fenêtre, et d'objets graphiques tracés dans les système d'axes. Chacun des composants de l'affichage a ses attributs que l'eon peut modifier indépendamment des autres. Regardons par exemple une suite d'instructions, qu'on gagnera à taper successivement dans une session interactive `ipython`, afin de voir apparaitre successivement les modifications :
Créons d'abord les objets à tracer :

In [117]:
```
x = np.linspace(0, 10, 500) ; y = np.sin(x) ; z = np.cos(x)
```

Ouvrons ensuite une fenêtre graphique :

In [118]:
```
fig = plt.figure(1)
```

Rajoutons lui un système d'axes par la commande `gca` (« get curent axes », qui récupère le système d'axes s'il existe, ou en crée un nouveau sinon) :

In [119]:
```
ax = fig.gca()
```

Créons enfin les graphes désirés dans le système d'axes

In [120]:
```
line1 = ax.plot(x, y, label="sinus") ; line2 = ax.plot(x, z, label="cosinus")
```

Mettons un titre :

In [121]:
```
ax.set_title("Fonctions sinus et cosinus")
```

Affichons les légendes

In [122]:
```
ax.legend()
```

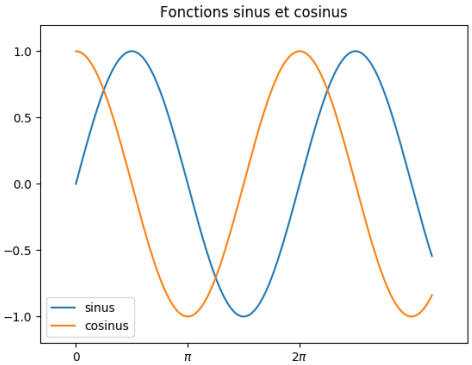
Étendons un peu les limites du graphique :

In [123]:
```
ax.set_xlim(-1,11) ; ax.set_ylim(-1.2,1.2)
```

Modifions les points de repère de l'axes des abscisses : (`xticks`)

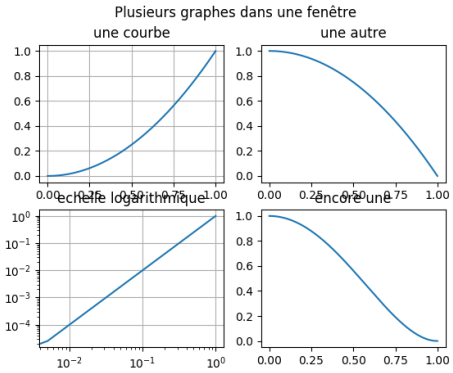
In [124]:
```
plt.xticks([0,np.pi,2*np.pi],[r'0',r'$\pi$',r'$2\pi$'])
```

Et voilà ce que cela donne :



Voici un exemple d'une fenêtre subdivisée en plusieurs sous-fenêtres :

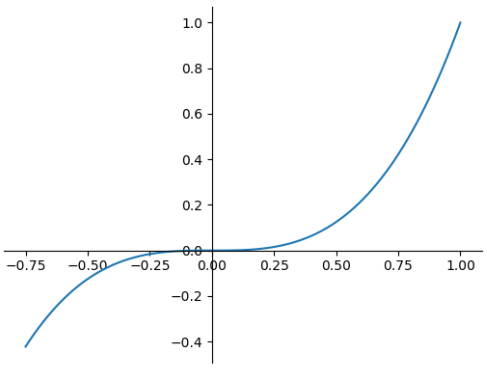
In [125]:
```
x = np.linspace(0,1,200) ; y = x**2
fig, axs = plt.subplots(2, 2)
fig.suptitle("Plusieurs graphes dans une fenetre")
axs[0,0].plot(x,y)
axs[0,0].grid()
axs[0,0].set_title("une courbe")
axs[0,1].plot(x,1-y)
axs[0,1].set_title("une autre")
ax = axs[1,0] ; ax.set_yscale('log') ; ax.set_xscale('log') ; ax.grid()
ax.plot(x,y) ; ax.set_title("echelle logarithmique")
axs[1,1].plot(x,(1-y)**2)
axs[1,1].set_title("encore une")
```



Un lecteur plus perfectionniste que l'auteur saura sans doute éviter certaines imperfections...

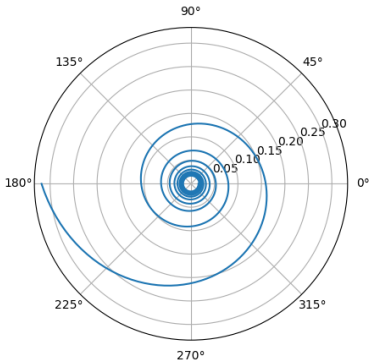
On peut aussi avoir envie de placer les axes comme un mathématicien le fait habituellement (effectuer les commandes une à une pour comprendre ce qui se passe) :

In [126]:
```
fig, ax = plt.subplots()
ax.spines['right'].set_color('none') # pas de trait a droite
ax.spines['top'].set_color('none') # pas de trait en haut
ax.spines['bottom'].set_position(('data',0)) # position de l'axe en  x=0
ax.spines['left'].set_position(('data',0))   # position de l'axe en  y=0
xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```



On peut aussi regarder une représentation en coordonnées polaires, en choisissant un système d'axes adapté :

In [127]:
```
theta=np.linspace(np.pi,20*np.pi,1000) ;  r = 1/theta
fig=plt.figure()
ax=fig.gca(projection='polar')
ax.plot(theta,r)
```


90°
45°
135°
180°
0°
225°
315°
270°
0.05
0.10
0.15
0.20
0.25
0.30