Mise à niveau en Python

December 30, 2021

1 Le langage python

1.1 Installation

On utilisera Python 3. Anaconda permet de gérer facilement les différentes versions de Python ainsi que les principaux modules, notamment pour le calcul scientifique. Anaconda Navigator offre une interface graphique pour gérer ces installations (via les environnements, mais l'environnement de base existant contient déjà tout ce qu'il faut pour MNI) et pour lancer votre éditeur de code.

Pour éditer des fichiers Python, vous avez notamment le choix entre les solutions gratuites de références suivantes: * Spyder: un environnement de développement particulièrement utile pour le calcul scientifique, avec une interface similaire à celles d'Octave et Matlab; l'installation de Spyder se fait facile depuis Anaconda Navigator. * Microsoft Visual Studio Code: un éditeur assez léger et pratique, avec notamment un débugueur intégré; le téléchargement et l'installation se font à partir du site internet ou depuis Anaconda Navigator. * PyCharm: un environnement de développement intégré avec de nombreuses fonctionnalités (au prix de plus de lenteur); le téléchargement et l'installation se font à partir du site internet ou depuis Anaconda Navigator.

Dans les salles de TP ou via la virtualisation, Anaconda est déjà installé. Vous pouvez lancer Anaconda Navigator depuis le menu des programmes.

Sur votre ordinateur personnel, vous pouvez installer Anaconda sur tous les OS.

1.2 Les objets de base

[1]:
```
a = 10  # un entier
b = -1.75643  # un flottant
c = 'hello "toi"'  # une chaîne de caractères
d = "coucou l'ami"  # une autre chaîne, on peut utiliser '...' ou "..." au choix
print(a, type(a))
print(b, type(b))
print(c, type(c))
print(d, type(d))
```

```
10 <class 'int'>
-1.75643 <class 'float'>
hello "toi" <class 'str'>
coucou l'ami <class 'str'>
```

[2]:
```
# Les chaînes formatées
print('il suffit de mettre f avant les guillemets et des accolades pour insérer␣
 ↪des valeurs de variables')
s = f'{c}, Ceci est une chaine formatée {a}, {b}, {b:.2f} (avec un format: deux␣
 ↪décimales)'
print(s)
```

```
il suffit de mettre f avant les guillemets et des accolades pour insérer des
valeurs de variables
hello "toi", Ceci est une chaine formatée 10, -1.75643, -1.76 (avec un format:
deux décimales)
```

1.3 Les collections (tuples, listes, dictionnaires, ensembles)

Les collections permettent de réunir plusieurs objets dans un même objet. On utilise les crochets pour accéder à un objet dans une collection. Plusieurs types de collections existent, avec des propriétés différentes.

Les tuples: * les types des objets peuvent être différents * création avec des parenthèses (ou rien du tout) * pas de modification possible * indexation par des entiers à partir de 0

[3]:
```
mon_tuple = (1, 2, 'coucou', 3.5)
un_autre_tuple = 'hello', 5
print(mon_tuple, type(mon_tuple))
print(un_autre_tuple)
print(mon_tuple[0], mon_tuple[2])
```

```
(1, 2, 'coucou', 3.5) <class 'tuple'>
('hello', 5)
1 coucou
```

[4]:
```
mon_tuple[0] = 0  # Génère une erreur, pas de modification possible dans un␣
 ↪tuple
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-4-6430c656d38e> in <module>
----> 1 mon_tuple[0] = 0  # Génère une erreur, pas de modification possible␣
↪dans un tuple

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

Les listes: * les types des objets peuvent être différents * création avec des crochets * modification possible * indexation par des entiers à partir de 0

[5]:
```
ma_liste = [1, 2, 'coucou', 3.5]
print(ma_liste, type(ma_liste))
ma_liste[3] = 5
print(ma_liste)
ma_liste.append(10)  # ajout d'un élément
print(ma_liste)
print(ma_liste[1:3])  # [i:j] sous-liste entre les indices i inclus et j exclus
print(ma_liste[:3])  # [:j] sous-liste jusqu'à l'indice j exclus
print(ma_liste[1:])  # [i:] sous-liste à partir de l'indice i inclus
print(ma_liste[-1], ma_liste[-2], ma_liste[:-2])  # indice négatif = indice en␣
 ↪partant de la fin
```

```
[1, 2, 'coucou', 3.5] <class 'list'>
[1, 2, 'coucou', 5]
[1, 2, 'coucou', 5, 10]
[2, 'coucou']
[1, 2, 'coucou']
[2, 'coucou', 5, 10]
10 5 [1, 2, 'coucou']
```

Les dictionnaires: * les types des objets peuvent être différents * création avec des accolades * modification possible * indexation par des *clés* (keys)

[6]:
```
mon_dict = {'une clé': 10, 'another key': 'une valeur', 5: "5 est une clé",␣
 ↪('une clé bizarre', 5): -1}
print(mon_dict, type(mon_dict))
print(mon_dict['une clé'])
print(mon_dict[5])
print(mon_dict['another key'])
print(mon_dict['une clé bizarre', 5])
print("Les clés:", mon_dict.keys())
print("Les valeurs:", mon_dict.values())
print("Les paires (clé, valeur):", mon_dict.items())
```

```
{'une clé': 10, 'another key': 'une valeur', 5: '5 est une clé', ('une clé
bizarre', 5): -1} <class 'dict'>
10
5 est une clé
une valeur
-1
Les clés: dict_keys(['une clé', 'another key', 5, ('une clé bizarre', 5)])
Les valeurs: dict_values([10, 'une valeur', '5 est une clé', -1])
Les paires (clé, valeur): dict_items([('une clé', 10), ('another key', 'une
valeur'), (5, '5 est une clé'), (('une clé bizarre', 5), -1)])
```

[ ]:

1.4 Les structures de contrôle (for, while, if)

Indentation obligatoire (recommandation: 4 espaces pour chaque indentation).

[7]:
```
for i in range(10):
    print(i)
for x in [1, 4, -2]:
    print('x =', x)
```

```
0
1
2
3
4
5
6
7
8
9
x = 1
x = 4
x = -2
```

[8]:
```
i = 10
while i > 0:
    print(i)
    i -= 2
```

```
10
8
6
4
2
```

[9]:
```
for i in range(10):
    if i == 3:
        print('trois')
    elif i > 5:
        print(i, '>5')
    else:
        print(i)
```

```
0
1
2
trois
```

```
4
5
6 >5
7 >5
8 >5
9 >5
```

1.5 Les fonctions

[10]:
```python
def f(x):
    return x + 1, x * 2

def g(x, y):
    if x > y:
        return x
    else:
        return y

def h(x):
    print(x)
```

[11]:
```python
x = f(3)
a, b = f(3)
print(x)
print(a)
print(b)
print(g(5, 10))
h(1)  #  La fonction affiche
print(h(1))  # La fonction affiche, puis renvoie None (pas de return), qui␣
 ↪s'affiche avec le print
```

```
(4, 6)
4
6
10
1
1
None
```

[12]:
```python
# Le nommage des arguments lors de l'appel à des fonctions permet de ne pas␣
 ↪respecter l'ordre des arguments
def div(numerateur, denominateur):
    return numerateur / denominateur

print(div(1, 2))
print(div(numerateur=1, denominateur=2))
print(div(denominateur=2, numerateur=1))
```

```
0.5
0.5
0.5
```

[13]:
```python
# Les valeurs par défaut des arguments
def incremente(x, n=1):
    return x + n
print(incremente(3, 4))
print(incremente(3, 1))
print(incremente(3))
```

```
7
4
4
```

1.6 Pour aller plus loin

Des aspects importants à voir rapidement après avoir pris connaissance des bases

- voir la notion de type mutable et immutable
- Python est un langage objet, voir comment créer et manipuler des classes

2 Calcul scientifique: `numpy`

Le module `numpy` fournit en premier lieu la structure de tableau numpy nd-array, qui est particulièrement adaptée pour le calcul scientifique (vecteurs, matrices, et plus généralement tableaux à n dimensions).

[14]:
```python
import numpy as np  # On importe le module numpy et on le renomme np
```

2.1 Créations de tableaux numpy

Les possibilités de création sont nombreuses: tableaux vides ou initialisés avec une valeur, conversion depuis une liste ou un tuple, etc.

[15]:
```python
# un vecteur de 10 zéros
vecteur_de_zeros = np.zeros(10)
print(vecteur_de_zeros)
print(type(vecteur_de_zeros))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<class 'numpy.ndarray'>
```

[16]:
```python
# une matrice 5x3 initalisée à un
matrice_de_uns = np.ones([5, 3])
print(matrice_de_uns)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
```

```
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

[17]:
```
# Des séquences
print(np.arange(10))  # Un argument n: 0, 1, ..., n-1
print(np.arange(3, 15))  # Deux arguments n, m (début, fin): n, n+1, ..., m-1
print(np.arange(5, -5, -2))  # Trois arguments n, m, p (début, fin, pas): n,␣
 ↪n+p, n+2p, ...
print(np.linspace(0, 1, 11))  # n=11 valeurs régulières entre 2 limites 0 et 1
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 3  4  5  6  7  8  9 10 11 12 13 14]
[ 5  3  1 -1 -3]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

[18]:
```
# Conversion depuis une liste: ici, une liste de listes pour obtenir une matrice
un_autre_tableau = np.array([[1, 3, 5], [-1, 6, 2]])
print(un_autre_tableau)
```

```
[[ 1  3  5]
 [-1  6  2]]
```

[19]:
```
# Matrice identité
identity = np.eye(4)
print(identity)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

[20]:
```
# un tableau vide en dimension 3 (ses coefficients sont arbitraires)
vide = np.empty([2, 3, 4])
print(vide)
```

```
[[[-2.31584178e+077 -2.31584178e+077  6.91691904e-323  0.00000000e+000]
  [ 0.00000000e+000  0.00000000e+000  0.00000000e+000  0.00000000e+000]
  [ 0.00000000e+000  0.00000000e+000  0.00000000e+000  0.00000000e+000]]

 [[ 0.00000000e+000  0.00000000e+000  0.00000000e+000  3.50786609e-322]
  [-1.49457796e-154 -2.31584178e+077  2.96439388e-323  0.00000000e+000]
  [ 0.00000000e+000  0.00000000e+000 -2.31584178e+077 -2.31584178e+077]]]
```

[21]:
```
# Nombre de dimensions: ndim
print(vecteur_de_zeros.ndim)
print(matrice_de_uns.ndim)
print(un_autre_tableau.ndim)
```

```
print(identity.ndim)
print(vide.ndim)
```

```
1
2
2
2
3
```

[22]:
```
# Dimension: shape
print(vecteur_de_zeros.shape)
print(matrice_de_uns.shape)
print(un_autre_tableau.shape)
print(identity.shape)
print(vide.shape)

n_rows = matrice_de_uns.shape[0]
n_cols = matrice_de_uns.shape[1]
print(f'Il y a {n_rows} lignes et {n_cols} dans la matrice de uns.')
n_rows, n_cols = matrice_de_uns.shape  # Équivalent (.shape renvoie un tuple)
print(f'Il y a {n_rows} lignes et {n_cols} dans la matrice de uns.')
```

```
(10,)
(5, 3)
(2, 3)
(4, 4)
(2, 3, 4)
Il y a 5 lignes et 3 dans la matrice de uns.
Il y a 5 lignes et 3 dans la matrice de uns.
```

[23]:
```
# Nombre d'élements: size (le produit des dimensions)
print(vecteur_de_zeros.size)
print(matrice_de_uns.size)
print(un_autre_tableau.size)
print(identity.size)
print(vide.size)
```

```
10
15
6
16
24
```

[24]:
```
# Redimensionner
print(un_autre_tableau)
print(un_autre_tableau.shape)
```

```
print('Une autre matrice')
a = un_autre_tableau.reshape(3,2)
print('Dimensions:', a.shape)
print(a)

print('Un vecteur')
a = un_autre_tableau.reshape(6)
print('Dimensions:', a.shape)
print(a)

print('Une matrice colonne')
a = un_autre_tableau.reshape(6, 1)
print('Dimensions:', a.shape)
print(a)

print('Une matrice ligne')
a = un_autre_tableau.reshape(1, 6)
print('Dimensions:', a.shape)
print(a)
```

```
[[ 1  3  5]
 [-1  6  2]]
(2, 3)
Une autre matrice
Dimensions: (3, 2)
[[ 1  3]
 [ 5 -1]
 [ 6  2]]
Un vecteur
Dimensions: (6,)
[ 1  3  5 -1  6  2]
Une matrice colonne
Dimensions: (6, 1)
[[ 1]
 [ 3]
 [ 5]
 [-1]
 [ 6]
 [ 2]]
Une matrice ligne
Dimensions: (1, 6)
[[ 1  3  5 -1  6  2]]
```

[25]:
```
# Astuce: on peut utiliser -1 pour une des dimensions dans reshape
# pour laisser numpy calculer automatiquement cette dimension
# car il n'y a pas d'ambiguité (le nombre total d'élément reste le même)
print('Une autre matrice')
```

```
a = un_autre_tableau.reshape(3,-1)
print('Dimensions:', a.shape)
print(a)

print('Une autre matrice')
a = un_autre_tableau.reshape(-1,2)
print('Dimensions:', a.shape)
print(a)

print('Un vecteur')
a = un_autre_tableau.reshape(-1)
print('Dimensions:', a.shape)
print(a)

print('Une matrice colonne')
a = un_autre_tableau.reshape(-1, 1)
print('Dimensions:', a.shape)
print(a)

print('Une matrice ligne')
a = un_autre_tableau.reshape(1, -1)
print('Dimensions:', a.shape)
print(a)
```

```
Une autre matrice
Dimensions: (3, 2)
[[ 1  3]
 [ 5 -1]
 [ 6  2]]
Une autre matrice
Dimensions: (3, 2)
[[ 1  3]
 [ 5 -1]
 [ 6  2]]
Un vecteur
Dimensions: (6,)
[ 1  3  5 -1  6  2]
Une matrice colonne
Dimensions: (6, 1)
[[ 1]
 [ 3]
 [ 5]
 [-1]
 [ 6]
 [ 2]]
Une matrice ligne
Dimensions: (1, 6)
```

```
[[ 1  3  5 -1  6  2]]
```

[26]:
```
# Le dtype (data type): les nd-array ne peuvent contenir que des éléments d'un même type
print('Par défaut, le dtype est flottant, comme dans le cas du vecteur de zeros')
vecteur_de_zeros = np.zeros(10)
print(vecteur_de_zeros)
print(vecteur_de_zeros.dtype)
print('On peut spécifier le dtype')
vecteur_de_zeros = np.zeros(10, dtype=int)
print(vecteur_de_zeros)
print(vecteur_de_zeros.dtype)
print('On peut convertir le dtype')
vecteur_de_faux = vecteur_de_zeros.astype(bool)
print(vecteur_de_faux)
print(vecteur_de_faux.dtype)
```

```
Par défaut, le dtype est flottant, comme dans le cas du vecteur de zeros
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
float64
On peut spécifier le dtype
[0 0 0 0 0 0 0 0 0 0]
int64
On peut convertir le dtype
[False False False False False False False False False False]
bool
```

[27]:
```
print('Le dtype peut être estimé automatiquement')
un_autre_tableau = np.array([[1, 3, 5], [-1, 6, 2]])
print(un_autre_tableau)
print(un_autre_tableau.dtype)
print('À nouveau, on peut le spécifier')
un_autre_tableau = np.array([[1, 3, 5], [-1, 6, 2]], dtype=float)
print(un_autre_tableau)
print(un_autre_tableau.dtype)
```

```
Le dtype peut être estimé automatiquement
[[ 1  3  5]
 [-1  6  2]]
int64
À nouveau, on peut le spécifier
[[ 1.  3.  5.]
 [-1.  6.  2.]]
float64
```

2.2 Opérations élémentaires sur les nd-array

[28]:
```
# Transposée
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
print(a.T)
```

```
[[1 2 3]
 [4 5 6]]
[[1 4]
 [2 5]
 [3 6]]
```

[29]:
```
# Addition, multiplication, puissance avec un scalaire (coefficient par␣
 ↪coefficient)
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
print(a + 1)
print(a * -2)
print(a ** 2)
```

```
[[1 2 3]
 [4 5 6]]
[[2 3 4]
 [5 6 7]]
[[ -2  -4  -6]
 [ -8 -10 -12]]
[[ 1  4  9]
 [16 25 36]]
```

[30]:
```
# Addition, multiplication, puissance entre 2 matrices (coefficient par␣
 ↪coefficient)
a = np.array([[1.1, 2, 3], [4, 5, 6]])
b = np.array([[-0.5, -2, -3], [1, 2, 3]])
print(b)
print(a + b)
print(a * b)
print(a ** b)
```

```
[[-0.5 -2.  -3. ]
 [ 1.   2.   3. ]]
[[0.6 0.  0. ]
 [5.  7.  9. ]]
[[-0.55 -4.   -9.  ]
 [ 4.   10.   18.  ]]
[[9.53462589e-01 2.50000000e-01 3.70370370e-02]
 [4.00000000e+00 2.50000000e+01 2.16000000e+02]]
```

[31]:
```
# Multiplication matricielle: @
a = np.array([[1, 2], [3, 4], [5, 6]])
b = np.array([[7, 8, 9, 10], [11, 12, 13, 14]])
c = a @ b
print(a)
print('de dimensions', a.shape)
print(b)
print('de dimensions', b.shape)
print('Produit matriciel')
print(c)
print('de dimensions', c.shape)
```

```
[[1 2]
 [3 4]
 [5 6]]
de dimensions (3, 2)
[[ 7  8  9 10]
 [11 12 13 14]]
de dimensions (2, 4)
Produit matriciel
[[ 29  32  35  38]
 [ 65  72  79  86]
 [101 112 123 134]]
de dimensions (3, 4)
```

[32]:
```
# Attention, un tableau 1D est un vecteur, pas une matrice colonne
A = np.array([[1, 2], [3, 4], [5, 6]])
u = np.array([-1, 2, 1])
v = np.array([-1, 2])
print('matrice a de dimensions', a.shape)
print(A)
print('vecteur v de dimensions', v.shape)
print(v)
print('vecteur u de dimensions', u.shape)
print(u)
print('Produit matrice vecteur A @ v')
print(A @ v)
print('Produit vecteur matrice u @ A')
print(u @ A)
print('Produit scalaire <u,u> (plusieurs façons de faire)')
print(u @ u)
print(np.inner(v, v))
print('Produit vectoriel u^v (plusieurs façons de faire)')
print(v.reshape(-1,1) @ u.reshape(1,-1))  # Conversion en matrice colonne et␣
↪matrice ligne
print(v.reshape(-1,1) @ u.reshape(-1,1).T)  # Conversion en matrices colonnes␣
↪et transposition
```

```
print(np.outer(v, u))
print('Mais pas (erreur)')
print(v @ u.T)
```

```
matrice a de dimensions (3, 2)
[[1 2]
 [3 4]
 [5 6]]
vecteur v de dimensions (2,)
[-1  2]
vecteur u de dimensions (3,)
[-1  2  1]
Produit matrice vecteur A @ v
[3 5 7]
Produit vecteur matrice u @ A
[10 12]
Produit scalaire <u,u> (plusieurs façons de faire)
6
5
Produit vectoriel u^v (plusieurs façons de faire)
[[ 1 -2 -1]
 [-2  4  2]]
[[ 1 -2 -1]
 [-2  4  2]]
[[ 1 -2 -1]
 [-2  4  2]]
Mais pas (erreur)
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-32-3048d5be59f3> in <module>
 21 print(np.outer(v, u))
 22 print('Mais pas (erreur)')
---> 23 print(v @ u.T)


ValueError: matmul: Input operand 1 has a mismatch in its core dimension␣
↪0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 2)
```

[33]:
```
# Matrice diagonale à partir d'un vecteur
```

```
print("La fonction np.diag permet de passer d'un vecteur à une matrice␣
 ↪diagonale")
v = np.arange(1, 5)
print(v)
print(np.diag(v))
```

```
La fonction np.diag permet de passer d'un vecteur à une matrice diagonale
[1 2 3 4]
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

[34]:
```
# Diagonale d'une matrice
print("La même fonction np.diag permet de passer d'une matrice à sa diagonale␣
 ↪(vecteur)")
M = np.arange(9).reshape(3, 3)
print(M)
print('Diagonale:', np.diag(M))
```

```
La même fonction np.diag permet de passer d'une matrice à sa diagonale (vecteur)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Diagonale: [0 4 8]
```

2.3 Indexation des nd-array

Voici quelques possibilités de base pour manipuler des nd-arrays. Voir la documentation de numpy pour plus de détails.

[35]:
```
A = np.arange(20).reshape(4, 5)
print(A)
```

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

[36]:
```
# Manipuler un coefficient
print(A[2, 3])
A[2, 3] = -1
print(A)
```

```
13
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
```

```
 [10 11 12 -1 14]
 [15 16 17 18 19]]
```

[37]:
```
# Extraire la i-ième ligne ou la j-ième colonne
i = 0
j = 1
print(A[i, :])
print(A[:, j])
```

```
[0 1 2 3 4]
[ 1  6 11 16]
```

[38]:
```
# Extraire un bloc : lignes 1 à 3-1=2, colonne 2 à fin
print(A[1:3, 2:])
```

```
[[ 7  8  9]
 [12 -1 14]]
```

[39]:
```
# Extraire un coefficient sur 2 sur la dernière ligne
print(A[-1, ::2])
```

```
[15 17 19]
```

[40]:
```
# Attention, les opérations ci-dessus sont appelées slices,
# le tableau obtenu n'est qu'une "vue" dans le tableau original
# qui est modifié si on change la vue
A = np.arange(20).reshape(4, 5)
print(A)
B = A[-2:, ::2]
print(B)
B[0,0] = 100
print(B)
print(A)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
[[10 12 14]
 [15 17 19]]
[[100  12  14]
 [ 15  17  19]]
[[  0   1   2   3   4]
 [  5   6   7   8   9]
 [100  11  12  13  14]
 [ 15  16  17  18  19]]
```

[41]:
```
# Indexation à partir de tableaux d'indices : extraire un vecteur de␣
↪coefficients
A = np.arange(20).reshape(4, 5)
print(A)
ind_row = (0, 0, 1, 1, 0)  # On peut utiliser de façon équivalente un tuple,␣
↪une liste, un nd-array
ind_col = (2, 3, 4, 4, 0)  # i_row et i_col doivent être de même longueur
print(A[ind_row, ind_col])
print('A[ind_row, ind_col] renvoie dans un tableau 1D les coefficients:')
for i in range(len(ind_row)):
    print(A[ind_row[i], ind_col[i]])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
[2 3 9 9 0]
A[ind_row, ind_col] renvoie dans un tableau 1D les coefficients:
2
3
9
9
0
```

[42]:
```
# Indexation à partir de masques booléens : extraire un vecteur de coefficients
m_bool = A > 10
print(A[m_bool])
```

```
[11 12 13 14 15 16 17 18 19]
```

[43]:
```
# Indexation à partir de masques booléens : extraire une sous-matrice
print(A)
m_row_bool = (False, True, False, True)  # un booléen pour chaque ligne
m_col_bool = (True, False, False, True, True)  # un booléen pour chaque colonne
print('Sélection des lignes')
print(A[m_row_bool, :])
print('Sélection des colonnes')
print(A[:, m_col_bool])
print('Sélection des lignes et des colonnes')
print(A[:, m_col_bool][m_row_bool, :])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
Sélection des lignes
[[ 5  6  7  8  9]
 [15 16 17 18 19]]
```

```
Sélection des colonnes
[[ 0  3  4]
 [ 5  8  9]
 [10 13 14]
 [15 18 19]]
Sélection des lignes et des colonnes
[[ 5  8  9]
 [15 18 19]]
```

2.4 Fonctions utiles d'algèbre linéaire

[44]:
```
# Norme d'un vecteur
v = np.array([1, 1, 0, -1, 1])
print('Vecteur:', v)
print('norme 2 par défaut:', np.linalg.norm(v))
print('norme 2:', np.linalg.norm(v, ord=2))
print('norme 1:', np.linalg.norm(v, ord=1))
print('norme infinie:', np.linalg.norm(v, ord=np.inf))
```

```
Vecteur: [ 1  1  0 -1  1]
norme 2 par défaut: 2.0
norme 2: 2.0
norme 1: 4.0
norme infinie: 1.0
```

[45]:
```
# Norme d'une matrice
M = np.eye(4)
print('Matrice:\n', M)
print('norme 2:', np.linalg.norm(M, ord=2))
print('norme Frobenius:', np.linalg.norm(M, ord='fro'))
```

```
Matrice:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
norme 2: 1.0
norme Frobenius: 2.0
```

[46]:
```
# Décomposition en valeurs propres (eigenvalues) : np.linalg.eig
A = np.arange(8).reshape(2,4)
B = A.T @ A

val_p, vec_p = np.linalg.eig(B)
print(B)
print('Valeurs propres:')
print(val_p)
print('Vecteurs propres:')
```

```
print(vec_p)
print('Test reconstruction:', np.linalg.norm(B - vec_p @ np.diag(val_p) @ vec_p.
 ↪T))
print('Test b.o.n.:', np.linalg.norm(np.eye(B.shape[0]) - vec_p.T @ vec_p))
```

```
[[16 20 24 28]
 [20 26 32 38]
 [24 32 40 48]
 [28 38 48 58]]
Valeurs propres:
[ 1.37675697e+02  2.32430274e+00  9.63869333e-15 -3.94441811e-16]
Vecteurs propres:
[[-0.32586834  0.77059057 -0.48893826 -0.02597241]
 [-0.43236661  0.33624265  0.46791983  0.44241892]
 [-0.53886488 -0.09810528  0.53097512 -0.80692063]
 [-0.64536315 -0.5324532  -0.50995669  0.39047411]]
Test reconstruction: 8.82475770564939e-14
Test b.o.n.: 0.5768070469442964
```

[47]:
```
# Décomposition en valeurs singulières (singular values) : np.linalg.svd

U, s, VT = np.linalg.svd(A)
print(A)
print('Valeurs singulières:')
print(s)
print('Vecteurs singuliers à gauche:')
print(U)
print('Vecteurs singuliers à droite:')
print(VT.T)
r = np.min(s.size)
print('Test carré des valeurs singulières', np.linalg.norm(np.sort(val_p)[-r:]␣
 ↪- np.sort(s) ** 2))
print('Test reconstruction:', np.linalg.norm(A - U[:, :r] @ np.diag(s) @ VT[:r,␣
 ↪:]))
print('Test b.o.n.:', np.linalg.norm(np.eye(A.shape[0]) - U.T @ U))
print('Test b.o.n.:', np.linalg.norm(np.eye(A.shape[1]) - VT @ VT.T))
```

```
[[0 1 2 3]
 [4 5 6 7]]
Valeurs singulières:
[11.73352876  1.52456641]
Vecteurs singuliers à gauche:
[[-0.29370412 -0.95589638]
 [-0.95589638  0.29370412]]
Vecteurs singuliers à droite:
[[-0.32586834  0.77059057 -0.38578674 -0.38880405]
 [-0.43236661  0.33624265  0.22458489  0.80595386]
```

```
 [-0.53886488 -0.09810528  0.70819044 -0.44549557]
 [-0.64536315 -0.5324532  -0.54698859  0.02834576]]
Test carré des valeurs singulières 5.709267256288867e-14
Test reconstruction: 4.58239956278683e-15
Test b.o.n.: 5.280011545101602e-16
Test b.o.n.: 7.093731533526306e-16
```

[48]:
```
# Décomposition de Cholesky: np.linalg.cholesky
A = np.random.randint(low=-1, high=0, size=(4, 4)) + 5 * np.eye(4)
print(A)
L = np.linalg.cholesky(A)
print(L)
print('Test reconstruction:', np.linalg.norm(A - L @ L.T))
```

```
[[ 4. -1. -1. -1.]
 [-1.  4. -1. -1.]
 [-1. -1.  4. -1.]
 [-1. -1. -1.  4.]]
[[ 2.          0.          0.          0.        ]
 [-0.5         1.93649167  0.          0.        ]
 [-0.5        -0.64549722  1.82574186  0.        ]
 [-0.5        -0.64549722 -0.91287093  1.58113883]]
Test reconstruction: 4.710277376051325e-16
```

[49]:
```
# Décomposition QR: np.linalg.qr
Q, R = np.linalg.qr(A)
print(Q)
print(R)
print('Test reconstruction:', np.linalg.norm(A - Q @ R))
print('Test b.o.n.:', np.linalg.norm(np.eye(Q.shape[0]) - Q.T @ Q))
```

```
[[-0.91766294 -0.06362848 -0.10482848  0.37796447]
 [ 0.22941573 -0.89079867 -0.10482848  0.37796447]
 [ 0.22941573  0.31814238 -0.83862787  0.37796447]
 [ 0.22941573  0.31814238  0.52414242  0.75592895]]
[[-4.35889894  1.3764944   1.3764944   1.3764944 ]
 [ 0.         -4.13585096  1.90885429  1.90885429]
 [ 0.          0.         -3.66899693  3.14485451]
 [ 0.          0.          0.          1.88982237]]
Test reconstruction: 2.214887977831648e-15
Test b.o.n.: 9.661444167943877e-16
```

3 Les entrées/sorties

[50]:
```
# Sauvegarder un seul tableau numpy: fichier .npy, fonction np.save
A = np.random.randn(5, 6)
print(A)

np.save('mon_fichier.npy', A)

data = np.load('mon_fichier.npy')
print(data)
```

```
[[-2.97454427e-01  1.65201241e+00  2.00026113e-01  6.35940736e-01
   1.57758194e-03  1.51643653e+00]
 [-1.06225929e+00 -4.51226676e-01 -5.96987526e-01  1.33377507e-01
   2.96722767e+00  3.16517977e-01]
 [ 5.26304843e-01  1.28968909e+00  4.77484650e-01  7.22107398e-01
  -4.29511170e-01 -7.64872467e-01]
 [-1.14464694e-01  8.24688105e-01  6.19013909e-01  4.83001570e-01
   5.97889995e-01  2.01274180e-02]
 [ 1.13332081e+00  4.15046932e-01  2.69828689e-01  4.82673234e-01
   1.51404733e+00 -8.53282083e-01]]
[[-2.97454427e-01  1.65201241e+00  2.00026113e-01  6.35940736e-01
   1.57758194e-03  1.51643653e+00]
 [-1.06225929e+00 -4.51226676e-01 -5.96987526e-01  1.33377507e-01
   2.96722767e+00  3.16517977e-01]
 [ 5.26304843e-01  1.28968909e+00  4.77484650e-01  7.22107398e-01
  -4.29511170e-01 -7.64872467e-01]
 [-1.14464694e-01  8.24688105e-01  6.19013909e-01  4.83001570e-01
   5.97889995e-01  2.01274180e-02]
 [ 1.13332081e+00  4.15046932e-01  2.69828689e-01  4.82673234e-01
   1.51404733e+00 -8.53282083e-01]]
```

[51]:
```
# Sauvegarder plusieurs tableaux numpy: fichier .npz, fonction np.savez
A = np.random.randint(10, size=(4, 5))
v = np.random.randint(10, size=(6))
print(A)
print(v)

np.savez('mon_fichier.npz', A=A, v=v, C=A)  # C=A pour sauvegarder A sous le␣
↪nom C

data = np.load('mon_fichier.npz')
# data est une sorte de dictionnaire dont les clés sont les noms des variables␣
↪sauvegardées

print(list(data.keys()))
for k in data.keys():
```

```
    print(k)
    print(data[k])
```

```
[[7 5 4 3 2]
 [1 5 3 1 4]
 [3 2 5 8 5]
 [9 0 9 1 7]]
[1 5 3 1 4 4]
['A', 'v', 'C']
A
[[7 5 4 3 2]
 [1 5 3 1 4]
 [3 2 5 8 5]
 [9 0 9 1 7]]
v
[1 5 3 1 4 4]
C
[[7 5 4 3 2]
 [1 5 3 1 4]
 [3 2 5 8 5]
 [9 0 9 1 7]]
```

[52]:
```
# Les fichiers .mat (Matlab/Octave)
from scipy.io import loadmat, savemat

# Les variables à sauvegarder sont passées dans un dictionnaire
savemat(file_name='mon_fichier.mat', mdict={'A':A, 'v':v})

data = loadmat('mon_fichier.mat')
for k in data.keys():
    print(k)
    print(data[k])
    if isinstance(data[k], np.ndarray):
        print('Dimension', data[k].shape)
        if 1 in data[k].shape:
            print('Remarque: les tableaux numpy 1D sont transformés en vecteurs␣
 ↪lignes [1, n] pour convenir à Matlab.')
```

```
__header__
b'MATLAB 5.0 MAT-file Platform: posix, Created on: Thu Dec 30 19:08:58 2021'
__version__
1.0
__globals__
[]
A
[[7 5 4 3 2]
 [1 5 3 1 4]
 [3 2 5 8 5]
```

```
 [9 0 9 1 7]]
Dimension (4, 5)
v
[[1 5 3 1 4 4]]
Dimension (1, 6)
Remarque: les tableaux numpy 1D sont transformés en vecteurs lignes [1, n] pour
convenir à Matlab.
```

Tous les formats de fichiers usuels peuvent être manipulés en Python: csv, json, txt, etc. Le format `pickle` permet de sauvegarder des objets génériques en Python. Voir la documentation en ligne.

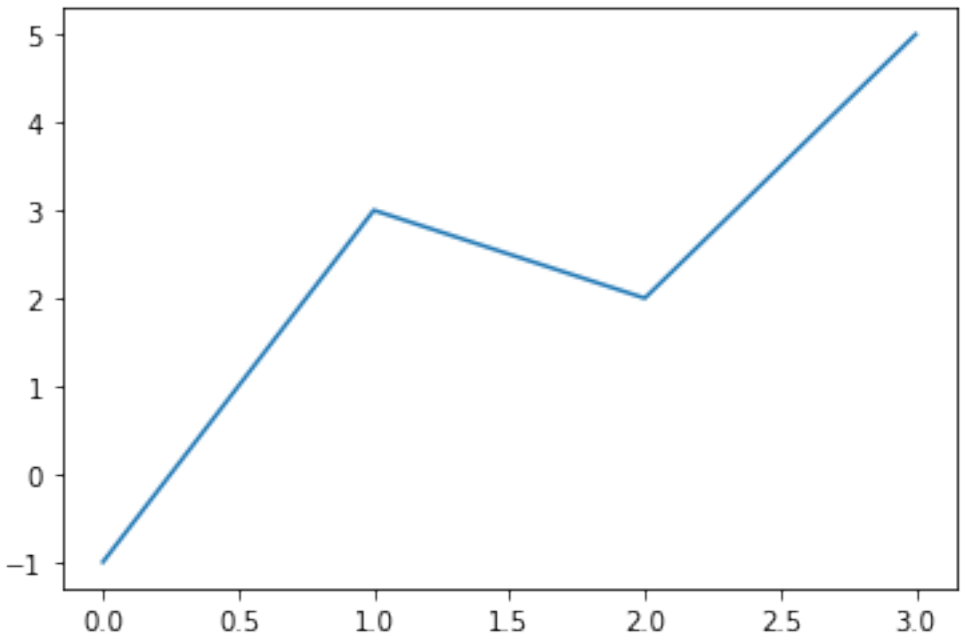
4 Les graphiques: `matplotlib`

[53]:
```
import matplotlib.pyplot as plt
```

4.1 Les tracés 2D

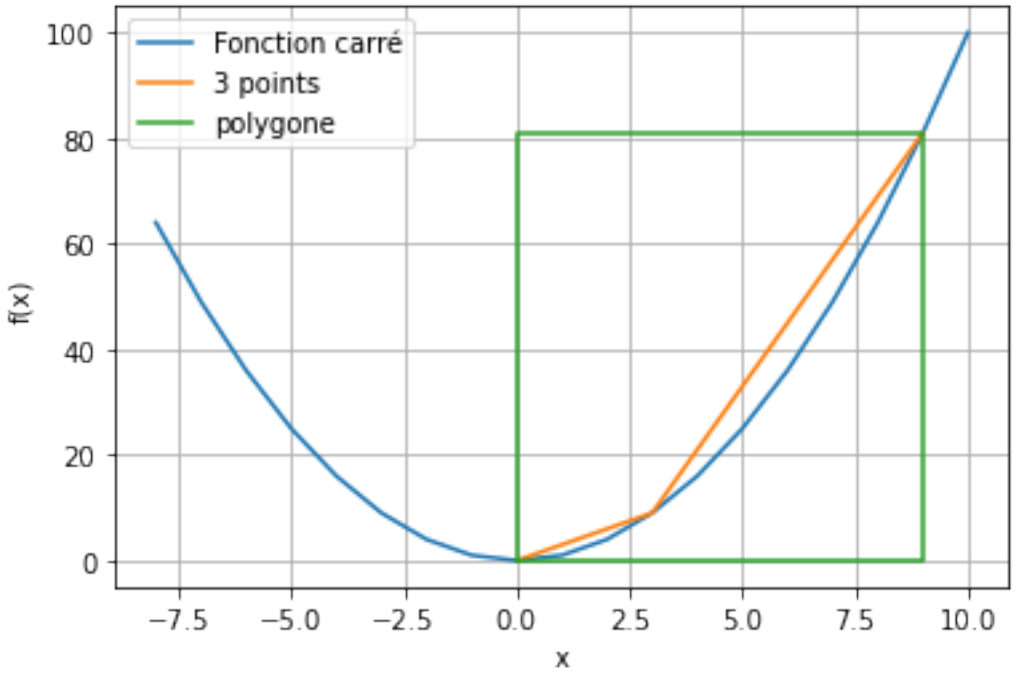
[54]:
```
# Le plus élémentaire
plt.plot([-1, 3, 2, 5])  # Abscisses automatiquement mises à 0, 1, 2, ...
```

[54]:
```
[<matplotlib.lines.Line2D at 0x1105541f0>]
```

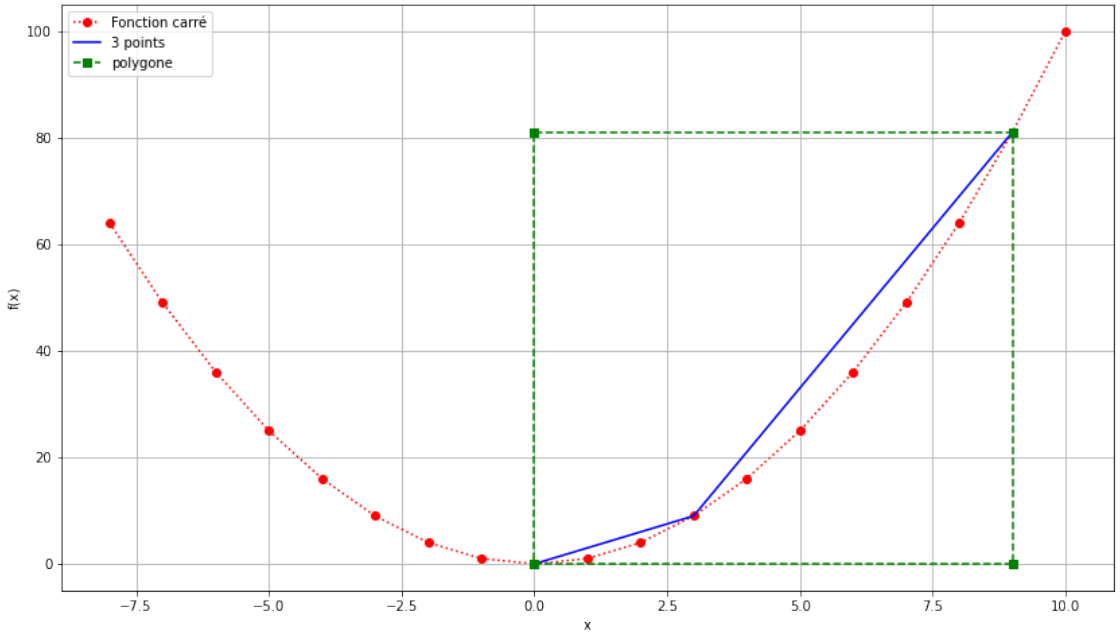


[55]:
```
# Gérer l'axe des abscisses, les noms des axes, la légende, ajouter une grille
x = np.arange(-8, 11)
y = x ** 2
```

```
plt.plot(x, y, label='Fonction carré')
plt.plot([0 , 3, 9], [0, 9, 81], label='3 points')
plt.plot([0, 9, 9, 0, 0], [0, 0, 81, 81, 0], label='polygone')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid()
```



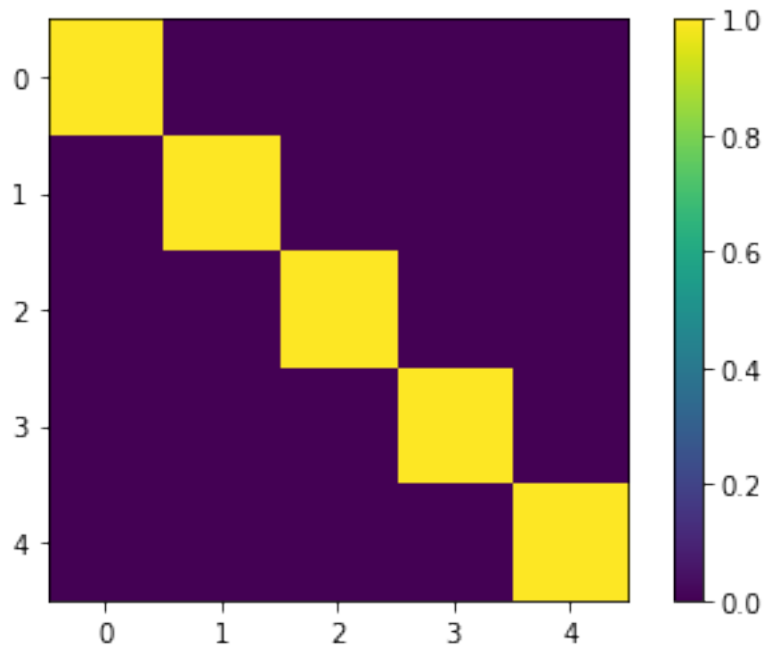
[56]:
```
# Personnaliser le style de ligne (:, --, etc.), les marqueurs (o, s, etc.),␣
 ↪les couleurs (b, g, r, etc.)
# Voir aussi toutes les options dans la doc de la fonction plot
plt.figure(figsize=(14, 8))  # Changer la taille de la figure
plt.plot(x, y, ':or', label='Fonction carré')
plt.plot([0 , 3, 9], [0, 9, 81], 'b', label='3 points', markersize=15)
plt.plot([0, 9, 9, 0, 0], [0, 0, 81, 81, 0], '--sg', label='polygone')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid()

# Sauvegarder dans un fichier image
plt.savefig('mon_graphique.pdf')
plt.savefig('mon_graphique.png')
```



4.2 Les images

[57]:
```
A = np.eye(5)
print(A)
plt.imshow(A)
plt.colorbar()
plt.savefig('figure_matrice.png')
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

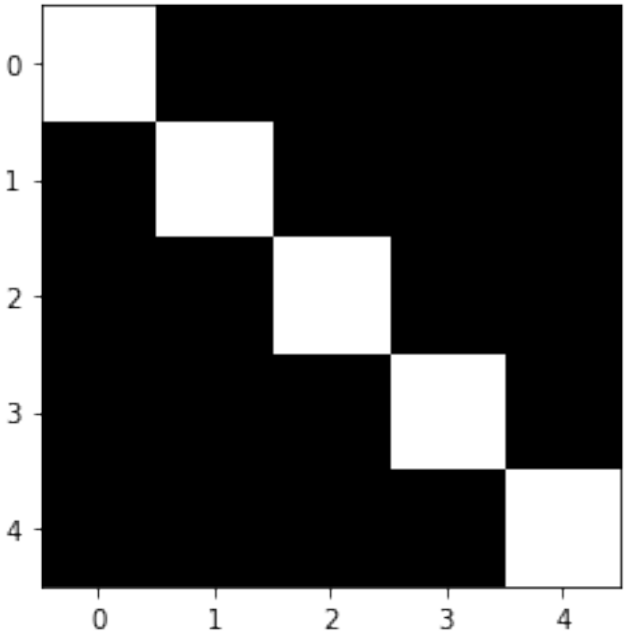


[58]:
```
# Sauvegarder une image
plt.imsave(fname='matrice.png', arr=A, cmap='gray')

# Chargement d'une image dans une matrice
B = plt.imread('matrice.png')
print(B.shape)
for canal in range(B.shape[2]):
    print(B[:, :, canal])

plt.imshow(B)
```

```
(5, 5, 4)
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
```

```
 [0. 0. 0. 0. 1.]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

[58]: <matplotlib.image.AxesImage at 0x110de87f0>

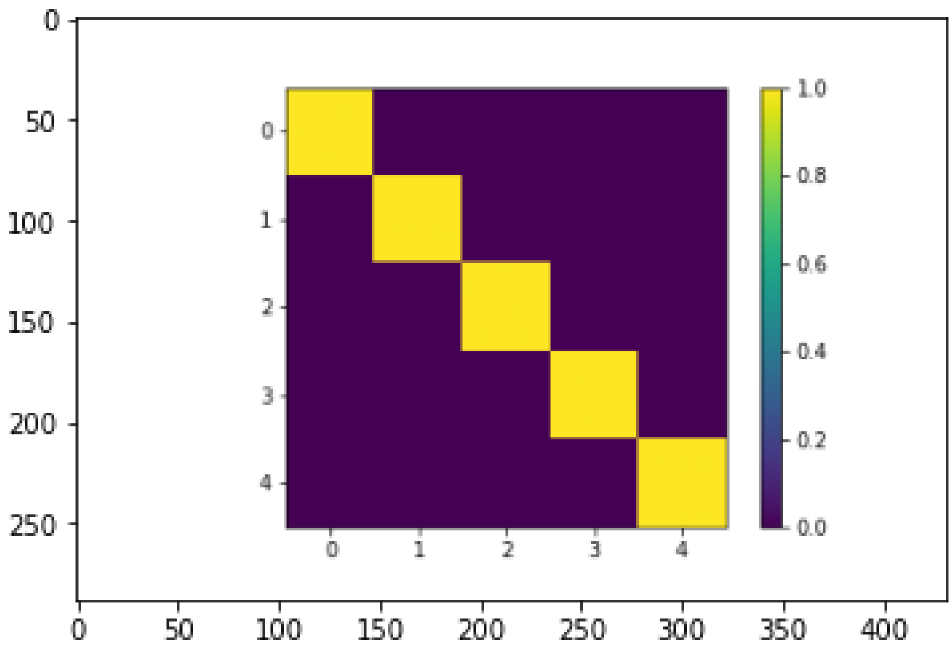


[59]:
```
# Attention à la différence entre imsave (sauvegarde d'un tableau comme image)␣
↪et savefig (sauvegarde d'une figure)
```

[60]:
```
C = plt.imread('figure_matrice.png')
print(C.shape)
plt.imshow(C)
```

```
(288, 432, 4)
```

[60]: <matplotlib.image.AxesImage at 0x110d94b20>



4.3 Les tracés 3D

[61]:
```
r = 8
x = np.linspace(-r, r, 2 * r + 1)
y = np.linspace(-r, r, 2 * r + 1)
print(x)
print(y)
X, Y = np.meshgrid(x, y)
print(X)
print(Y)
```

```
[-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
[-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
[[-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
```

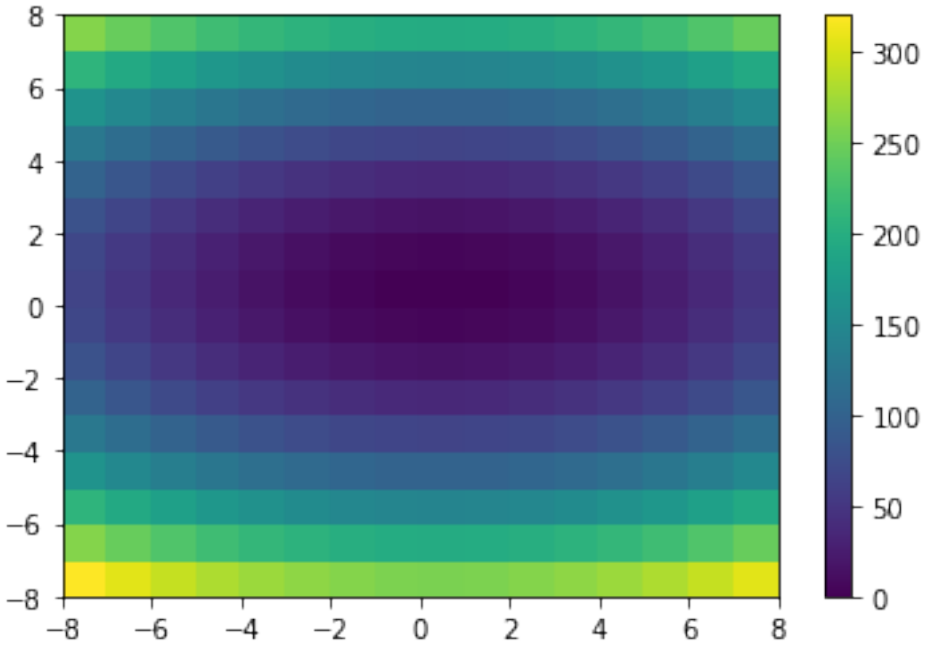
```
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [-8. -7. -6. -5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.  6.  7.  8.]]
[[-8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8. -8.]
 [-7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7. -7.]
 [-6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6. -6.]
 [-5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5.]
 [-4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4. -4.]
 [-3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3. -3.]
 [-2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2. -2.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
 [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
 [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
 [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
 [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
 [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
 [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]]
```

[62]:
```
def f(x, y):
    return x ** 2 + 4 * y ** 2
Z = f(X, Y)
print(Z.shape)
```

```
(17, 17)
```

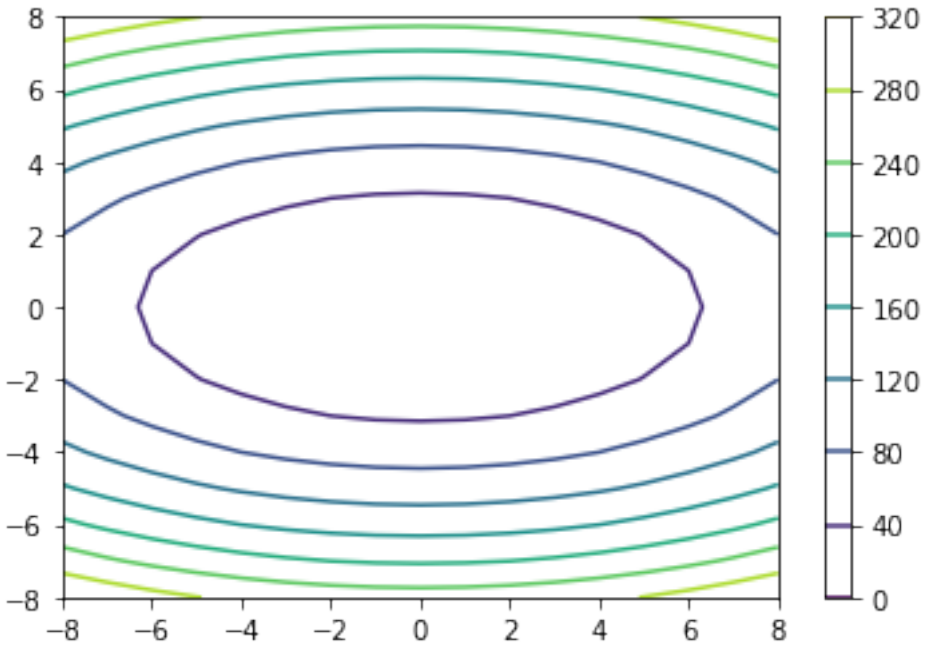
[63]:
```
plt.pcolormesh(X, Y, Z)
plt.colorbar()
```

[63]:
```
<matplotlib.colorbar.Colorbar at 0x1114d1820>
```



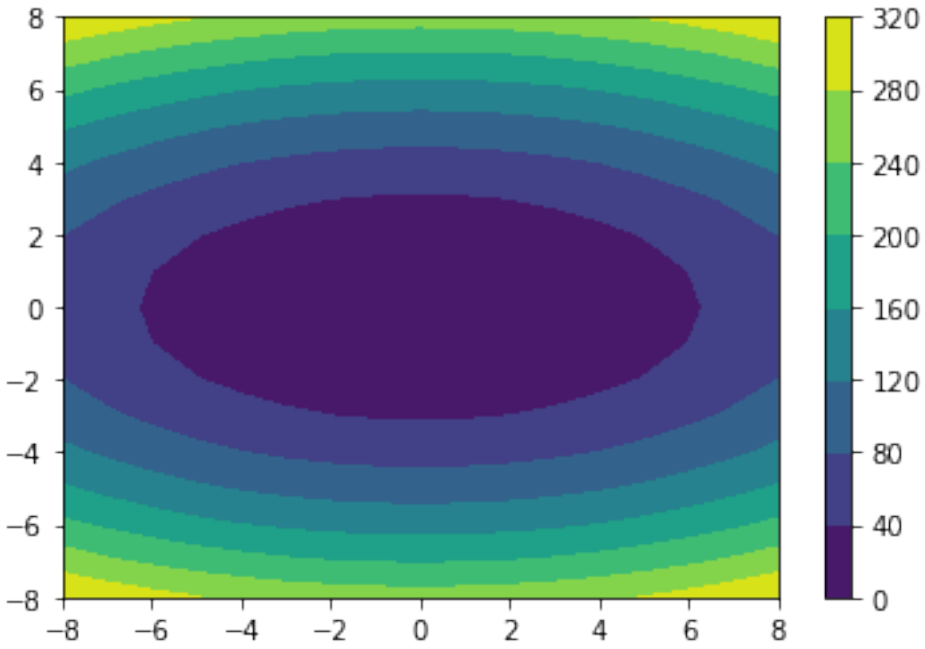
[64]:
```
plt.contour(X, Y, Z)
plt.colorbar()
```

[64]: <matplotlib.colorbar.Colorbar at 0x1115a6cd0>



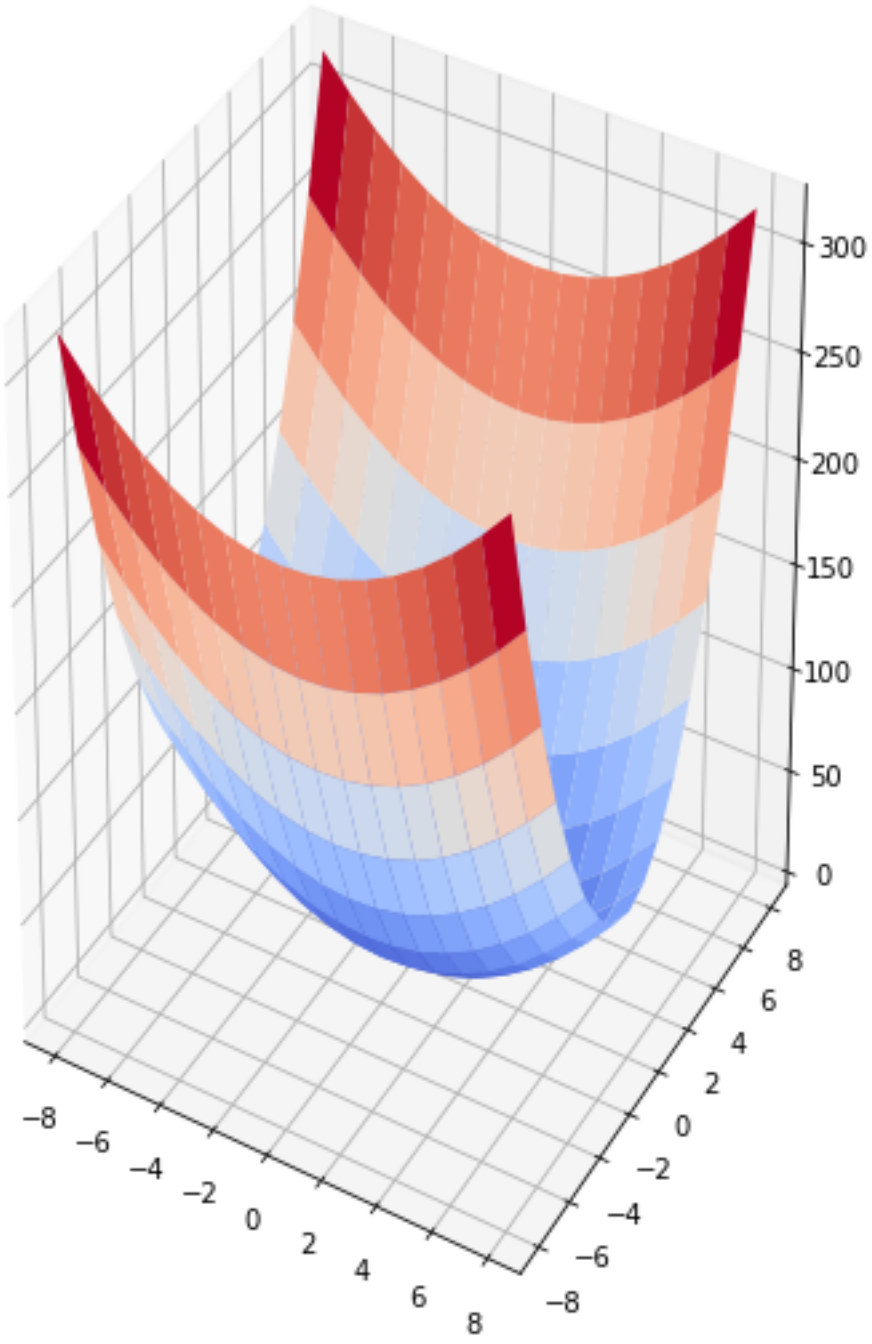
[65]:
```
plt.contourf(X, Y, Z)
plt.colorbar()
```

[65]: <matplotlib.colorbar.Colorbar at 0x1116bf670>



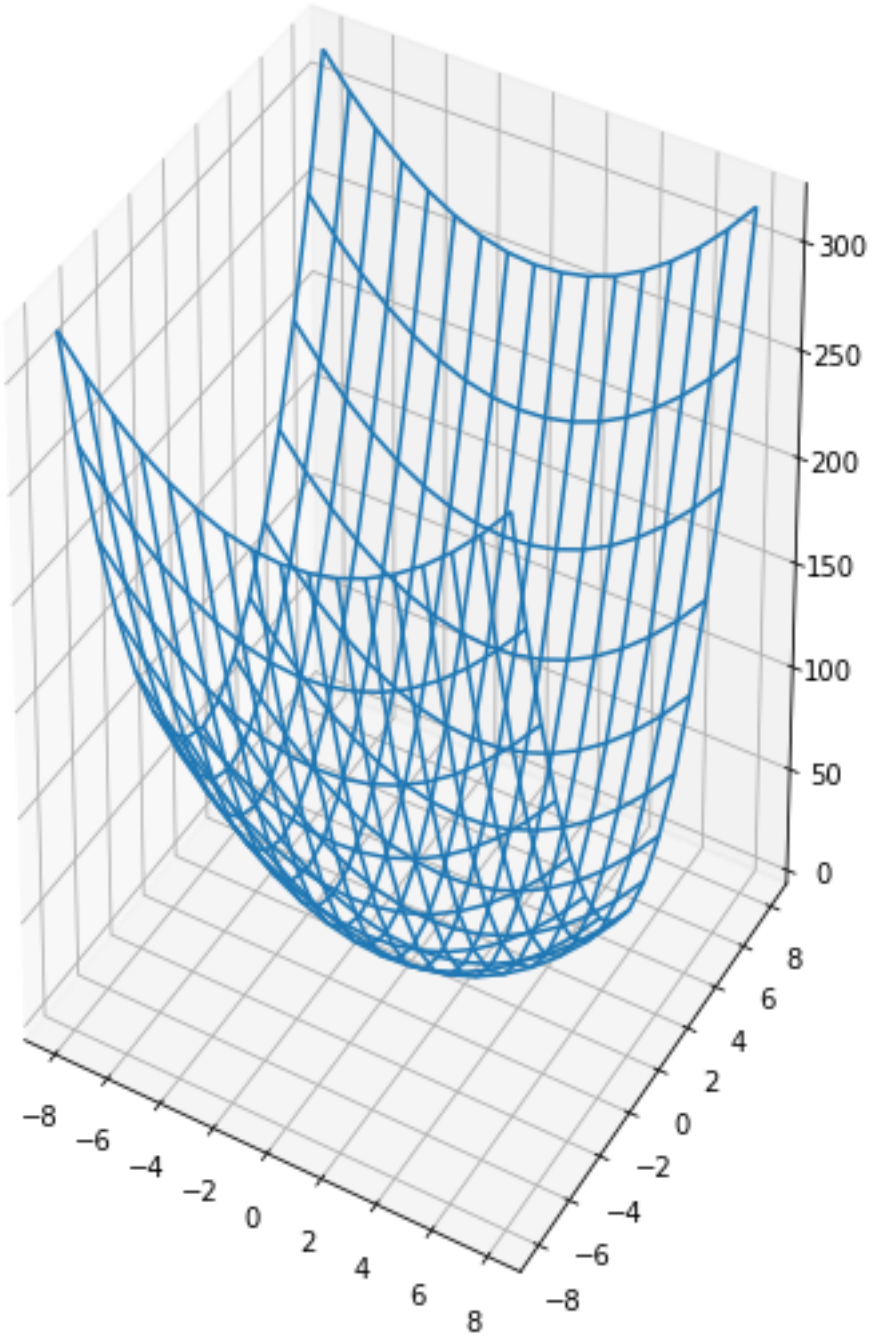
[66]:
```
from matplotlib import cm

fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)
```

[66]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x111793c70>



[67]:
```
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_wireframe(X, Y, Z)
```

[67]: <mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x111b62190>



Voir les tracés 3D dans la documentation de `matplotlib`

[ ]:

[ ]: