Formation Python

Elie Dumas-Lefebvre

Institut des Sciences de la Mer de Rimouski

elie.dumas-lefebvre.1@ulaval.ca

20 juin 2017

Histoire de Python

Pourquoi utiliser Python ?

- Language élégant
- Syntaxe rigoureuse (indentation, conventions, etc.)
- Logiciel libre, gratuit et portable
- Plusieurs librairies disponibles pour le traitement de données scientifiques et océanographiques (Numpy, Scipy, Matplotlib, netCDF4)
- Communauté très active (Stack Overflow)

Note : La version Python utilisée pour cette présentation est la version 3.6

Déclaration de variables

```
# Voici un premier commentaire
toto = 1 # En voila un deuxième
         # Et pourquoi pas un troisième ?

titi = 3.1415

tata = '#Ceci n'est pas un commentaire'
```

Déclaration de variables

Bilan :

- toto est un entier
- titi est un réel
- tata est une chaine de caractères

Nombres et opérations mathématiques de base

- interaction entre les réels et les entiers
- +, -, /, //, %, *, **

Exemples :

```
>>> 3 + 4
7
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4.0
5.0
>>> 5 ** 2 # 2 est l'exposant de 5
25
>>> 8 / 5  # La division retourne toujours un réel
1.6
```

Exemples (suite) :

```
>>> 8 // 5    # La division entière arrondi à
1             # la baisse la division normale

>>> 8 // 5.0 # La division entière effectuée
1.0           # avec un réel retournera un réel

>>> 8 % 5     # L'opérateur modulo donne le
3             # reste de la division entière
```

Chaines de caractères

Exemples :

```
>>> 'ISMER'
'ISMER'
>>> 'L\'annexe A' == "L'annexe A"
True
>>> '"L\'annexe A", dit-il'
'"L\'annexe A", dit-il'
>>> print('"L\'annexe A", dit-il')
"L'annexe A", dit-il
```

Caractères spéciaux

- \n : nouvelle ligne
- \r : retour de chariot
- \t : tabulation horizontale

Interprétation

```
>>> print('La première ligne.\nLa deuxième.')
La première ligne.
La deuxième.
>>> print(r'La première ligne.\nLa deuxième')
'La première ligne.\nLa deuxième'
```

Concaténation

Exemples :

```
>>> 3 * 'la' + 'si'
'lalalasi'
>>> 3 * 'la' 'si'
'lasilasilasi'
>>> 'la' 'si' 'do'
'lasido'
>>> note_1 = 'do'
>>> note_1 + 're'
'dore'
```

Indexation et tranches

Matlab

- Séquences indexées de 1 à n
- Les tranches se font avec des "(i :j)"

Python

- Séquences indexées de 0 à n-1
- Les tranches se font avec des "[i :j]"

Exemples :

```
>>> mot = 'bonjour'
>>> mot[0]
b
>>> mot[6]
r
>>> mot[-1]
r
>>> mot[-7]
b
>>> mot[:3]
bon
>>>mot[1:4] # Caractères étant compris entre
onj          # l'indice 1 (inclus) et
             # l'indice 4 (exclus)
>>> len(mot)
7
```

Immuabilité des chaines de caractères :

```
>>> mot[:3] = 'ba'
TypeError:'str' object does not
support item assignment

# Changer bonjour pour bajour
>>> nouveau_mot = 'ba' + mot[3:]
>>> print(nouveau_mot)
'bajour'
```

Défi !

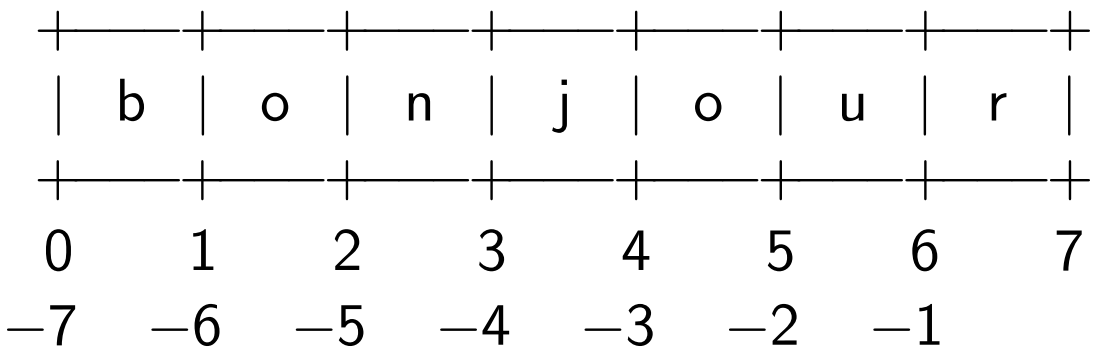
- Créer une chaine de caractère de longueur paire (e.g. 'Python')
- inverser la première moitié avec la deuxième

Une démarche possible

```
>>> ma_chaine = 'Python'
>>> ma_chaine_rev = ma_chaine[3:] + ma_chaine[:3]

>>> ma_chaine_rev
'honPyt'
```

Représentation générale d'une séquence Python :



Listes

```
>>> ma_liste = [1, 2, 3, 4]
>>> print(ma_liste)
[1, 2, 3, 4]
```

Indexation et tranches

```
>>> ma_liste[0]
1
>>> ma_liste[1:3]
[2, 3, 4]
>>> ma_liste[-1]
4
>>> len(ma_liste)
4
>>> min(ma_liste)
1
>>> max(ma_liste)
4
```

Copie d'une liste vs pointeur

```
# Création d'une copie indépendante
>>> copie_de_ma_liste = ma_liste[:]
>>> copie_de_ma_liste
[1, 2, 3, 4]

>>> ma_liste[0] = 5

>>> ma_liste
[5, 2, 3, 4]
>>> copie_de_ma_liste
[1, 2, 3, 4]
```

```
#Création d'un pointeur
>>> copie_de_ma_liste = ma_liste
>>> copie_de_ma_liste
[1, 2, 3, 4]

>>> ma_liste[0] = 5

>>> ma_liste
[5, 2, 3, 4]
>>> copie_de_ma_liste
[5, 2, 3, 4]
```

Mutabilité de la liste

```
>>> fibonacci = [1, 1, 2, 3, 6, 8] # Il y a
                                   # une erreur !

>>> fibonacci[4] = 5 # On remplace la mauvaise
                     # valeur par la bonne

>>> print(fibonacci)
[1, 1, 2, 3, 5, 8]
```

Note : Une liste peut contenir plusieurs objets de différents types.

Liste mixte

```
>>> liste_mixte = [1, 2, 3, 'a', 'b', 'c']

>>> liste_mixte[3:] = ['A', 'B', 'C']
>>> print(liste_mixte)
[1, 2, 3, 'A', 'B', 'C']

>>> del liste_mixte[3:]
>>> print(liste_mixte)
[1, 2, 3]
```

Concaténation

```
>>> ma_liste = [1, 2, 3, 4]
>>> DLC      = [5, 6, 7, 8]

>>> ma_liste = ma_liste + DLC
>>> print(ma_liste)
[1, 2, 3, 4, 5, 6, 7, 8] # Concaténation directe
```

Méthode append

```
>>> ma_liste = [1, 2, 3, 4]

>>> ma_liste.append(DLC)
>>> print(ma_liste)
[1, 2, 3, 4, [5, 6, 7, 8]] # Ajout de l'argument
                           # de la fonction append
                           # à la fin
```

Démarche alternative avec append

```
>>> ma_liste = ma_liste[:4]

>>> for i in DLC:
...     ma_liste.append(i)
...
>>> print(ma_liste)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Méthode extend

```
>>> ma_liste = [1, 2, 3, 4]

>>> ma_liste.extend(DLC)
>>> print(ma_liste)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Tuples

- Syntaxe semblable à la liste
- Accepte tout type d'éléments
- Immuable

```
>>> mon_tuple = (1, 2, 3) # Tuple d'entiers
>>> type(mon_tuple)
<type 'tuple'>

>>> t[0] = 4 # Immuabilité
TypeError: 'tuple' object does not support
item assignment

# Un tuple peut être hétérogène
>>> mon_autre_tuple = ([1, 2, 3], 'toto')
```

Méthodes utiles : Applicables à tout type de séquence

list(s)	convertir s en une liste
tuple(s)	convertir s en un tuple
len(s)	retourne la longueur de s
max(s)	retourne la valeur maximale de s
min(s)	retourne la valeur maximale de s

Méthodes utiles : Applicables aux listes

s.append(x)	ajoute l'élément x à la fin de s
s.extend(t)	ajouter la séquence t à la fin de s
s.index(x)	retourne l'indice de l'élément x dans s
s.insert(i, x)	insère l'élément x à l'indice i dans s

Conversion de types

float(x)	convertir x en un float
int(x)	convertir x en un entier
str(x)	convertir x en une chaine de caractères
type(x)	obtenir le type de x

Défi !

- Créer une liste contenant 5 chiffres
- Append un premier tuple contenant sont min sont max
- Insérer une chaine de caractère au centre

Une démarche possible

```
>>> ma_liste = [1, 2.1, 4.6, 5.9, 10.3]
>>> info_tuple = (min(ma_liste), max(ma_liste))
>>> ch_carac = 'intru'

>>> ma_liste.append(info_tuple)
>>> ma_liste.insert(3, ch_carac)

>>> ma_liste
[1, 2.1, 4.6, 'intru', 5.9, 10.3, (1, 10.3)]
```

TABLE – Comparaison entre les opérateurs logiques Python vs Matlab

Énoncé logique	Python	Matlab
plus petit que	<	<
plus petit ou égal	<=	<=
plus grand que	>	>
plus grand ou égal	>=	>=
égal à	==	==
n'égale pas	!=	~=
et	and	&
ou	or	|
non	not	~

If, elif, else

```
>>>chiffre=int(input('Veuillez entrer un chiffre:'))
... Veuillez entrer un chiffre: 42

>>> if chiffre < 42 :
...     print('Le chiffre doit être plus grand')
...
... elif chifre > 42 :
...     print('Le chiffre doit être plus petit')
...
... else:
...     print('Voila la réponse a l'Univers')
...
Voila la réponse a l'Univers
```

Expression générale

```
if condition1:
    conséquence

elif condition2:
    conséquence

else condition3:
    conséquence
```

Boucles

Deux types de boucles :

- while
- for

Boucle while

```
while condition:
    conséquence
```

Exemple :

```
>>>ma_liste = [1, 2, 3, 4]
>>>i = 0
>>>while i < 2:
...     ans = ma_liste[i] + ma_liste[i+1]
...     i += 1
...     print(ans, end=',')
...
3, 5
```

Boucle for

De façon générale, on exprime la boucle for comme suit,

```
for i in s:
    conséquence
```

Exemple :

```
>>> nombres = [3, 4, 71, 121]
>>>
>>> for nombre in nombres:
...     if nombre < 0:
...         var = 'nombre négatif trouvé'
...         break
...     else:
...         var = 'aucun nombre négatif trouvé'
...
>>> print(var)
'aucun nombre négatif trouvé'
```

Fonction range

- range(start, stop, step)
  retourne une liste de nombres k où start $\leq$ k $<$ stop, et où k est incrémenté par la valeur step.

```
>>> print(range(5))
range(0, 5) # Les chiffres que contient
            # le range sont implicites

>>> print(type(range))
<class 'range'>

>>> for n in range(0, 10, 2):
...     print(n, end=',')
0, 2, 4, 6, 8

>>> for n in range(10, 0, -2):
...     print(n, end=',')
10, 8, 6, 4, 2
```

Note : Il est très important de finir chaque condition par un ' :'.

Lecture et écriture de fichiers : la fonction open

```
fichier = open('mon_fichier', 'w')
```

Modes d'ouverture de fichier

'r'	mode lecture (par défaut)
'w'	mode écriture (tout fichier possédant le même *filename* sera 'overwrité')
'a'	mode append, toutes nouvelles données écrites dans le fichier seront ajoutées à la fin du fichier
'r+'	mode lecture et écriture
'b'	mode binaire (peut être joint à n'importe qu'elle autre méthode)

Exemples lecture de fichiers :

```
import numpy as np  # Import de la librairie
                    # numpy sous le nom 'np'

# Télécharger les données d'un fichier binaire
file = open(binary_file, 'rb')

data = np.fromfile(file, dtype=float32)

file.close()

# Télécharger les données d'un fichier .txt
data = np.loadtxt('text_file.txt')
```

Exemples d'écriture de fichiers :

```
# Écriture dans un fichier .txt
np.savetxt('mes_donnees.txt', data)

# Sauvegarder un array dans un fichier binaire .npy
np.save('mes_donnees.npy', data)
```

Note : On peut peut utiliser la fonction `load` de numpy pour ouvrir les fichiers `.npy`

Déclaration d'une fonction

- `def`
- le nom de la fonction
- la liste de paramètres ente parenthèses
- le caractère " :"
- un bloc d'énoncés indenté

Exemple de fonction

```
>>> def exposant(x, y):
...     return x**y
...
>>>> exposant(2, 3)
8
```

Attribution de valeurs par défaut

```
# Bonne syntaxe
>>> def achat(item, prix, taxe = 0.15):
...         prix = prix*(1 + taxe)
...         print('{} : {} $'.format(item, prix))


# Mauvaise syntaxe
>>> def achat(item, taxe = 0.15, prix):
...         prix = prix*(1 + taxe)
...         print('{} : {} $'.format(item, prix))
SyntaxError: non-default argument follows
default argument
```

Exemple de fonction

```
>>> achat('pain', 2.50)
pain : 2.875 $

# Ordre des arguments non nécessaire
# s'ils sont nommés
>>> achat(taxe = 0.34, item = 'clou', prix = 0.15)
clou : 0.201 $
```

Valeur(s) de retour

```
>>> def exposant(x, y)
...     return x**y, y**x
...
>>> exposant(3, 4)
(81, 64)

>>> ans1, ans2 = exposant(3, 4)

>>> print(ans1, ans2)
81 64
```

Paramètres en nombres variables

```
>>> def sum(*s):
...     ans = 0
...     for n in s:
...         ans += n
...     return ans
...
>>> sum(1, 2, 3, 4), sum(1, 4)
(10, 5)
>>>
```

Variable globale vs variable locale

```
>>> def func1():
...     x = 13
...
>>> def func2():
...     global x
...     x = 13
...
>>> x = 8
>>> func1(); print(x) # x non modifié
8
>>> func2(); print(x) # x modifié
13
```

Démarche alternative

```
>>> def func1():
...     x = 13
...     return x
...
>>> x = 8
>>> x = func1(); print(x)
13
```

Importation

```
import numpy as np  # Importer de la bibliotheque
                    # numpy sous le nom np

from scipy.stats import mode # Importer la fonction
                             # mode de scipy.stats

from sympy import * # Importer tous les modules
                    # de la bibliothèque sympy
```

Numpy

- Outils pour intégrer du code C/C++ et du code Fortran
- Plusieurs fonctions mathématiques

array ou matrix ?

Réponse courte : `array`
Pourquoi ?

Array

Matlab

```
>>> a = [1,2,3;4,5,6]
a =
   1   2   3
   4   5   6
```

Python

```
>>> Import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(a)
[[1, 2, 3],
 [4, 5, 6]]
```

Dimensions

```
>>> print(a.shape) # L'attribut .shape d'un array
                   # est un tuple contenant
                   # (lignes, colonnes)
(2, 3)
>>> print(a.size)  # L'attribut .size est
                   # lignes * colonnes
6
>>> print(type(a))
<class 'numpy.ndarray'>
```

Attention : La fonction `size` de Matlab est l'équivalent de la fonction `shape` de Numpy

Bonne vs mauvaise déclaration d'un array

```
>>> a = np.array(1, 2, 3, 4, 5)
ValueError : only 2 key-word arguments accepted

>>> a = np.array([1, 2, 3, 4, 5]) # Bonne façon :)
```

Initialisation de matrices : Ones

```
>>> a = np.ones(5)
>>> print(a)
[1. 1. 1. 1. 1.]
>>> a.shape
(5,)
>>> print(a[1]) # Accès au deuxième élément
1.

>>> a = np.ones((5,1))
>>> print(a)
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

```
>>> a[1, 0] # Accès au deuxième élément
1.0
>>> a.shape
(5, 1)

>>> a = np.ones((1, 5))
>>> print(a)
[[1. 1. 1. 1. 1.]]
>>> a[0, 1] # Accès au deuxième élément
>>> a.shape
(1, 5)
```

Note : En remplacant `ones` par `zeros` dans l'exemple ci-dessus, on aurait obtenu une matrice contenant des 0.0.

Initialisation de matrices : arange

```
arange(start, stop, step)
```

```
>>> a = np.arange(1, 3, 0.5)
>>> a
[1.  1.5  2.  2.5]

>>> a = np.arange(1, 3.000001, 0.5)
>>> a
[1.  1.5  2.  2.5  3.]
```

Initialisation de matrices : linspace

```
linspace(start, stop, nelements)
```

```
>>> a = np.linspace(1, 3, 5)
>>> a
 [1.  1.5  2.  2.5  3.]
```

Opérations avec scalaires

```
# Vecteur ligne de dimensions (5,)
>>> a1 = np.arange(5)

>>> a1 - 2
array([-2, -1, 0, 1, 2])

>>> a1 * 2
array([0, 2, 4, 6, 8])

>>> a1 ** 3
array([0, 1, 8, 27, 64])
```

Opérations entre matrices

```
>>> a1 = np.ones((2, 2))

>>> a1
array([[1., 1.]
       [1., 1.]])

>>> a1 * a1 # Multiplication élément par élément
array([[1., 1.]
       [1., 1.]])

>>> a1.dot(a1) # Multiplication matricielle
array([[2., 2.]
       [2., 2.]])
```

Opérations avec des fonctions

```
>>> a1 = np.arange(0, 2*np.pi + 1, np.pi/2)
>>> np.sin(a1)
array([0., 1., 1.22e-16, -1., -2.449e-16])

>>> a1 = np.arange(5)
>>> np.exp(a1)
array([1., 2.718, 7.389, 20.0855, 54.598])
```

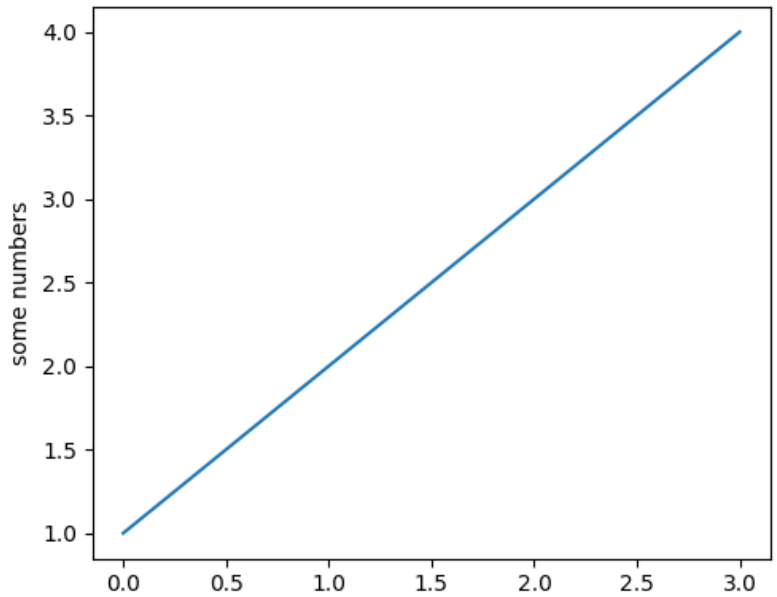
Site de référence : `https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html`

Pyplot

C'est quoi Pyplot ?

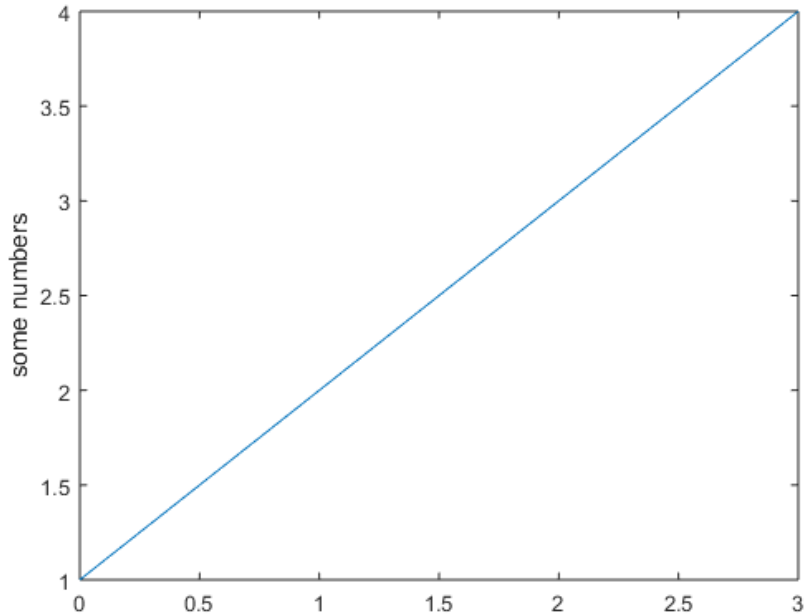
Python

```
import matplotlib.pyplot as plt
nums = np.arange(1,5)
plt.plot(nums)
plt.ylabel('some numbers')
plt.show()
```



Matlab

```
>>> nums = linspace(1, 4, 4);
>>> xaxis = linspace(0, 3, 4);
>>> plot(xaxis, nums)
>>> ylabel('some numbers')
```



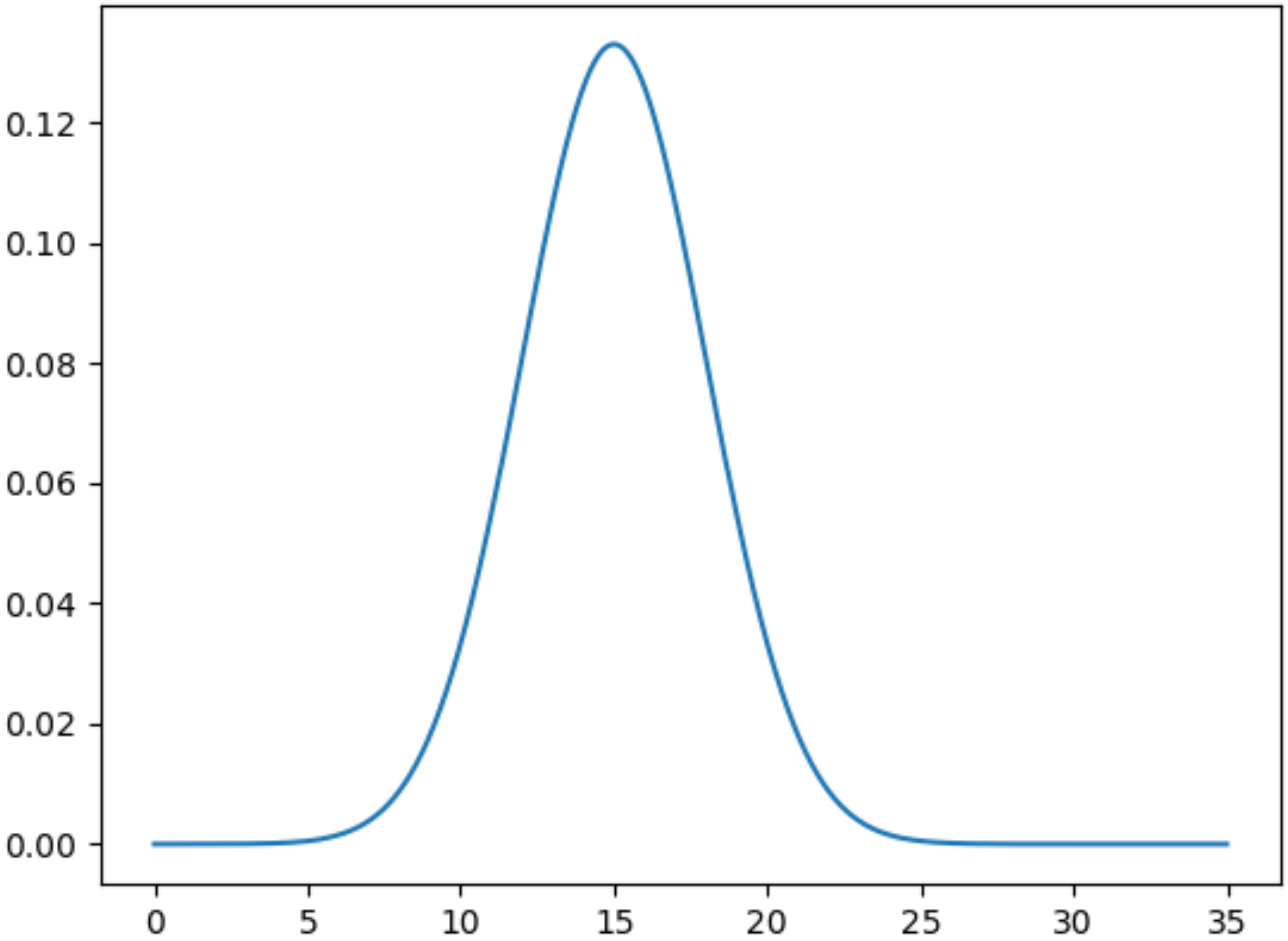
Exemple de mélange numpy/matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

def normal_pdf(x, mu, sigma):
    return np.exp((x - mu)**2/(2*sigma**2)) /
           (sigma*np.sqrt(2*np.pi))

x = np.arange(0, 35, 0.01)
y = normal_pdf(x, 15, 3)

plt.plot(x, y)
plt.show()
```


0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
5
10
15
20
25
30
35