Polycopié de Python

Gabriel Dauphin

March 23, 2026

Contents

Le document présente un certain nombre d'explications, d'instructions, de questions et d'exercices. Ces instructions sont à exécuter de manière à les comprendre. Les questions sont des applications directes des instructions et ne devraient pas poser de problèmes. Les exercices peuvent être réalisés aussi avec un petit nombre d'instructions. Ces instructions sont a testés d'abord en ligne. Pour les exercices, il est important de sauvegarder les instructions que vous proposez dans un fichier texte afin de les montrer quand un enseignant passe vous voir. Le fichier texte peut être un script écrit avec l'éditeur Matlab/Octave.

L'annexe de ce document présente des idées permettant d'idées pour comprendre pourquoi une instruction ou un programme ne fonctionne pas. Il présente aussi les instructions à mettre pour utiliser les toolbox sous Octave, ce qui permet maintenant de réaliser presque tout le TP de traitement numérique du signal avec Octave.

Chapter 1

Opérations sur les nombres et les vecteurs

1.1 Première utilisation de Python

1.1.1 Diverses techniques pour exécuter une commande en Python `print, help, import`

Sous Windows, on ouvre une fenêtre de commande avec les touches `Win+R` suivi de `cmd`. Dans la suite du document, j'appelle cette fenêtre de commande un terminal.

Sous Linux [1], on ouvre un terminal par exemple en écrivant `term` dans la fenêtre activité.

On peut lancer l'interpréteur de Python en écrivant `python` (ou `python3`). L'instruction suivante permet d'afficher le terme `Bonjour`.

```
>>>print('Bonjour')
```

>>> est ici une notation pour indiquer que ce sont des commandes à exécuter dans l'interpréteur Python. La fonction `print` ne fonctionne pas sans les parenthèses.

`'Bonjour'` est une chaîne de caractères, ce qui s'appelle un `str` pour string. Et ce `str` peut aussi s'écrire `"Bonjour"`.

```
>>>print("Bonjour")
```

On peut fermer l'interpréteur avec `exit()` ou bien `Ctr+Z+Return`.

Remarquez que dans un terminal sous Windows ou Linux, ou dans l'interpréteur Python, il est possible de rappeler cette commande en tapant le début de la commande et en appuyant sur la flèche supérieure.

Il est aussi possible d'utiliser directement le terminal sous Windows ou Linux pour exécuter cette commande

```
python -c print('Bonjour')
```

Remarquez qu'ici on ne peut remplacer les apostrophes par des guillemets.

On peut aussi exécuter un script appelé ici `ex.py` contenant la commande `print('Bonjour')`. Sous Windows, on peut fabriquer ce script avec cette commande

```
echo print('Bonjour') > ex.py
```

On peut lire son contenu avec `type ex.py`
On peut lire, écrire et modifier son contenu avec n'importe quel éditeur de texte par exemple avec la commande `notepad ex.py &` En pratique, j'utilise l'éditeur de texte `Notepad++`.

Sous Linux, on peut aussi

```
echo 'print("Bonjour")' > ex.py
```

On peut lire son contenu avec `cat ex.py`
On peut lire, écrire et modifier son contenu avec n'importe quel éditeur de texte par exemple avec la commande `nano ex.py` avec ou sans `&` suivant qu'on est sur l'ordinateur ou en ligne de commande via `ssh`. J'utilise parfois l'éditeur `gedit`.

Ce script `ex.py` peut être exécuté en tant que script Python avec

[1] Il existe de nombreuses environnement graphiques disponibles sous Linux, je considère ici Gnome qui est utilisé par Ubuntu.

```
python ex.py
```

On peut ouvrir l'interpréteur Python juste après avoir exécuté le script

```
python -i ex.py
```

On peut aussi voir ce script comme un module et dans l'interpréteur, écrire

```
>>>import ex
```

La commande `help` permet d'avoir des informations sur le module `ex.py` depuis l'interpréteur Python

```
>>>help(ex)
```

Cette commande ne fonctionne pas sans les **parenthèses**. Cette commande permet d'afficher des explications lorsque celles-ci sont précédés de trois guillemets et terminés par trois guillemets. Mais cette commande ne fonctionne pas si juste avant on n'a pas importé le module `ex.py`.

Considérant le contenu suivant pour `ex.py`

```
"""Ce programme permet d'illustrer les commandes print et help.
"""
print('Bonjour')
```

Et les commandes suivantes dans l'interpréteur Python

```
>>>import ex
Bonjour
>>>help(ex)
Help on module ex:

NAME
    ex - Ce programme permet d'illustrer les commandes print et help.

FILE
    c:\users\invite\ex.py
```

Il est possible de commenter une ligne ou une portion de ligne en utilisant le caractère # (c'est ce qui suit ce caractère qui n'est pas pris en compte par Python).

```
>>> print("Bonjour")#ceci est faux
```

Si le caractère # se trouve entre des guillemets, il est considéré comme un caractère quelconque.

Il est possible de scinder une longue ligne sur plusieurs lignes avec le caractère \.

```
>>> print("1 2 3\
>>>   4 5 6\
>>>   7 8 9")
```

Il est possible de mettre plusieurs instructions sur la même ligne en les séparant avec un point virgule.

```
>>> print("abc"); print("bac")
```

Exercice 1 *Essayez de prévoir le résultat d'exécution du programme suivant appelé* `ex.py`

```
"""Ce programme permet\
   d'illustrer les commandes\
   print et help.
"""
import ex
help(ex)
print('Bonjour')
```

Vérifiez en exécutant ce script que votre prédiction se confirme.

1.1.2 Calculs sur les scalaires et utilisation de fonctions `def`,`return`,`int`,`float`,`str`,`type`,*définition, appel, argument*

On peut utiliser l'interpréteur pour faire des calculs

```
>>>2+3*5**2
77
```

Remarquez que la multiplication est prioritaire par rapport à l'addition et que la puissance est notée avec deux étoiles.

Mais si on met cette instruction dans un script alors le résultat n'est pas affiché à moins d'utiliser `print`

```
print(2+3*5**2)
```

permet d'afficher 77

Python définit de très très nombreux types d'objets. `type` permet de trouver le type de l'objet défini. Connaître le type d'un objet est essentiel parce que c'est en fonction de cela que des commandes existent ou non et ont une certaine façon de fonctionner.

```
>>>type("abc")
<class 'str'>
>>>type(6)
<class 'int'>
>>>type(5.0)
<class 'float'>
```

La division / transforme un `int` en `float`. On peut garder un entier en utilisant //.

```
>>> 2/3
0.6666666666666666
>>> 2//3
0
```

On définit une variable de type `int` ou `float` suivant que la valeur affectée est un entier ou un réel.

```
>>> a = 3
>>> print(type(a))
<class 'int'>
>>> a = 3.0
>>> print(type(a))
<class 'float'>
```

On peut définir une fonction avec la commande `def` et `return` en utilisant l'interpréteur.

```
>>> def exemple():
>>>   return "abc"
>>>
>>> print(exemple())
abc
```

Attention cette fonction ne fonctionne pas si on ne met pas deux ou quatre espaces à la deuxième ligne, autrement dit si on ne respecte pas l'indentation.

Dans l'exemple ci-dessus, on appelle définition de la fonction les deux premières lignes, c'est le code qui explique ce que fait la fonction. On appelle appel de la fonction la dernière ligne, c'est le code qui appelle la fonction. Remarquez que pour la définition et pour l'appel, il faut mettre des parenthèses. C'est en fait la même raison qui fait que précédemment pour `help` et `print`, il faut des parenthèses. On appelle argument le ou les paramètres qui sont dans ces parenthèses.

On peut maintenant mettre cette fonction dans un module que l'on appelle ici `ex.py`.

```
def exemple():
  return "abc"
```

On peut alors exécuter cette fonction soit dans un interpréteur Python avec

```
>>> import ex
>>> ex.exemple()
abc
```

Remarquez que si vous modifiez le fichier `ex.py` après avoir fait `import`, ces modifications ne sont pas prises en compte. Une première façon de prendre en compte ces modifications consiste à refermer l'interpréteur Python et en ouvrir un autre. Une autre façon consiste à utiliser la fonction `reload` du module `importlib`.

```
>>> import importlib
>>> importlib.reload(ex)
```

On peut aussi exécuter ce script dans un terminal avec

```
python -c "import ex; print(ex.exemple())"
abc
```

Pour exécuter cette fonction à partir d'un terminal, on pourrait être tenté de faire comme dans l'exercice 1. La solution consiste à placer à la fin du module, des instructions indiquant les commandes à exécuter, et ces instructions suivent un test sur la variable appelée `__name__`. Quand cette variable vaut `__main__`, cela veut dire que le script est appelé dans un terminal. Quand le script est appelé avec la commande `import`, alors cette variable vaut le nom du script, ici `ex`.

Pour réaliser l'expérience, je met l'instruction suivante dans le script `ex.py`

```
print(__name__)
```

J'exécute dans un terminal,

```
python ex.py
__main__
```

J'exécute ensuite dans l'interpréteur Python

```
>>> import ex
ex
```

La solution pour exécuter une fonction d'un module dans un terminal est illustré avec le script suivant

```
def exemple():
  return "abc"

if __name__ == "__main__":
  print(exemple())
```

Exercice 2 *L'objectif est de réaliser une fonction appelée* `add` *qui ajoute deux nombres et renvoie le résultat.*

1. *Réalisez cette fonction dans l'interpréteur Python et utilisez la dans l'interpréteur.*
2. *Ecrivez un module dans lequel vous mettez cette fonction. Utilisez ce module dans l'interpréteur.*
3. *Modifiez le module créé pour que l'utilisation de cette fonction et l'affichage du résultat se fasse depuis un appel en ligne de commande dans un terminal.*

1.1.3 Sélection du répertoire, os,getcwd,chdir,mkdir,rmdir,*répertoire courant*

Les programmes et les données que l'on utilise dans une simulation se trouvent physiquement sur le disque dur d'un serveur ou d'un ordinateur, c'est le système d'exploitation qui les gère au travers de répertoire. Quand on lance la commande `python` depuis un terminal, il y a un répertoire courant, c'est en fonction de celui-ci que la commande va être effectué. Par exemple la commande `python ex.py` ne fonctionne pas si le répertoire courant ne contient pas le fichier `ex.py`. On distingue le chemin absolu du chemin relatif. Le premier spécifie un répertoire indépendamment du répertoire courant. Le deuxième spécifie la façon dont on accède à un répertoire à partir du répertoire courant en utilisant `./` pour le répertoire courant et `../` pour le répertoire parent.

Dans un terminal sous Windows, on a les commandes suivantes.

- La commande `pwd` permet de trouver le chemin absolu du répertoire courant.
- La commande `cd` suivi d'un espace et du nom d'un ou plusieurs répertoires séparés par \ permet de changer de répertoire.
- Une lettre associée à un disque dur suivi de deux points permet de changer de disque dur.
- `mkdir` créé un repertoire.
- `del` détruit un répertoire ou un fichier.

En pratique le plus simple pour se placer dans un répertoire donné, j'utilise l'Explorateur de fichier (`Win+E`), je sélectionne le répertoire et je copie le nom complet de ce répertoire en cliquant avec la touche droite de la souris dans l'onglet en haut au milieu et en sélectionnant *copier l'adresse*, puis en collant cette adresse dans la fenêtre de commande juste après `cd`. L'action de copier du texte se fait avec `Ctr+C` et l'action de coller du texte se fait avec `Ctr+V`. Remarquez que sous Windows, les noms de répertoires utilisent un \ en ligne de commande et deux \ lorsque le nom est enregistré sur un fichier ou dans une chaîne de caractères. Dans les faits, utiliser un / fonctionne aussi sous Windows et n'a pas besoin d'être doublé et fonctionne aussi sous Linux, c'est de fait ce que j'utilise. Quand un nom comporte des espaces, on le met entre guillemets.

Dans un terminal sous Linux, on a les commandes suivantes.

- La commande `pwd` permet de trouver le chemin absolu du répertoire courant.
- La commande `cd` suivi d'un espace et du nom d'un ou plusieurs répertoires séparés par / permet de changer de répertoire.
- `cd /` positionne le répertoire courant dans le répertoire de l'utilisateur.
- `mkdir` permet de créer un nouveau répertoire.
- `rm` permet de supprimer un répertoire ou un fichier.

En pratique le plus simple pour se placer dans un répertoire donné, j'utilise l'Explorateur de fichier, je créé ou je sélectionne le répertoire et je copie le nom complet de ce répertoire en cliquant avec la touche droite de la souris dans l'onglet en haut au milieu et en sélectionnant *copier l'adresse*, puis en collant cette adresse dans la fenêtre de commande juste après `cd`. L'action de copier du texte se fait avec `Ctr+Insert` et l'action de coller du texte se fait avec `Maj+Insert`.

Il est aussi possible de réaliser ces commandes dans l'interpréteur Python ou dans un script, mais en pratique je ne l'utilise pas. Ces fonctions sont dans le module `os`, ces deux lettres signifiant Operating System. Il s'agit d'abord de l'importer

```
>>> import os
```

puis de rappeler ce nom à chaque fois qu'on utilise une fonction contenue dans ce module.

- L'équivalent de `pwd` est `getcwd`

  ```
  >>> os.getcwd()
  ```

- L'équivalent de `pwd` est `chdir`

```
>>> os.chdir('../')
```

- `mkdir` existe dans `os` et permet de créer dans le répertoire courant un répertoire `rep`

```
>>> os.mkdir('rep')
```

- On supprime un répertoire appelé `rep` déjà vide et placé dans le répertoire courant.

```
>>> os.rmdir('rep')
```

- On peut lister les fichiers et répertoires contenus dans un répertoire appelé `rep`

```
>>> os.listdir('rep')
```

Remarquez que pour chacune des fonctions qui précèdent, il est possible de remplacer `'rep'` par le nom d'une variable de type `str` dans laquelle on met un nom de répertoire contenu dans le répertoire courant, un chemin absolu ou relatif au répertoire courant.

Lorsque le module auquel on veut accéder n'est pas dans le répertoire courant, il est aussi possible d'indiquer à l'interpréteur Python sa possible localisation ou de modifier la capacité de Python à trouver un module au sein d'un module. Cela se fait en important le module `sys`.

```
>>> import sys
```

`sys` contient une variable qui est une liste des répertoires parmi lesquels le module est recherché. Je présenterai la notion de liste ultérieurement. Mais ici j'utilise juste deux fonctions, `append` pour ajouter un élément, ici le répertoire où se trouve le module et `pop` pour supprimer le dernier élément rajouté. Ces modifications ne sont valables que tant que l'interpréteur est ouvert ou à l'intérieur du fichier associé au module. Toutes ces modifications n'impactent pas la lecture de données que je présenterai ultérieurement.

- Pour ajouter le répertoire appelé `rep` dans la liste des répertoires considérés pour trouver un module.

```
>>> sys.path.append('rep')
>>> import ex
```

- Pour supprimer le répertoire dernièrement ajouté à la liste des répertoires considérés pour trouver un module.

```
>>> sys.path.pop()
>>> import ex
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'ex'
```

Exercice 3 *Au sein du répertoire courant, créez deux répertoires contenant chacun un module. L'un des deux modules fait appel à l'autre. Faites en sorte de le faire fonctionner dans l'interpréteur et avec une commande en ligne.*

1.1.4 Tableaux

Le module `numpy` permet de définir des tableaux. Je propose d'appeler `np` ce module avec l'instruction suivante.

```
>>> import numpy as np
```

On peut définir un tableau à une dimension de plusieurs façons.

- Le tableau vaut 0 de type `float`, et est de longueur `S`, S étant une variable entière.

```
>>> A = np.zeros(shape=(S,),dtype=float)
```

- Le tableau vaut 1 de type `float`, et est de longueur `S`, S étant une variable entière.

```
>>> A = np.ones(shape=(S,),dtype=float)
```

- Le tableau est défini à partir d'une liste de valeurs particulières.

```
>>> A = np.array([1,2,3],dtype=float)
```

Remarquez que dans tous ces exemples, on aurait pu omettre le paramètre `dtype`, parce que par défaut, c'est un float, ou bien spécifier `int`.

C'est de la même façon qu'on peut définir un tableau à deux dimensions.

- Le tableau vaut 0, et est de taille `M`×`N`.

```
>>> A = np.zeros(shape=(M,N,))
```

- Le tableau vaut 1, et est de taille `M`×`N`.

```
>>> A = np.ones(shape=(M,N,))
```

- Le tableau est défini à partir d'une liste de valeurs particulières.

```
>>> A = np.array([[1,2,3],[2,4,6]])
```

  Remarquez qu'on ne peut pas utiliser l'espace pour séparer les valeurs.

À partir d'une variable `A`, on peut retrouver les informations suivantes.

- Son type

```
>>> type(A)
<class 'numpy.ndarray'>
```

- Quand il de type `numpy.ndarray`, la valeur de chaque composante

```
>>> A[0,1]
2
```

  Le premier indice, 0 indique la première ligne et le deuxième indice indique la deuxième colonne.

- Quand il de type `numpy.ndarray`, sa taille

```
>>> A.shape
(2,3,)
```

  Remarquez que le premier indice désigne le nombre de lignes et le deuxième le nombre de colonnes.

- Quand il de type `numpy.ndarray`, le type de ses composantes

```
>>> A.dtype
dtype('int64')
```

  Si `A` avait été définie comme composée de réels, il y aurait eu affiché `dtype('float64')` Je propose de considérer ces deux types comme `int` et `float` parce qu'on a bien

```
>>> A.dtype == int
True
```

- On peut afficher `A`

  ```
  print('A=',A)
  ```

  Dans cette instruction, la chaîne de caractère `A=` est d'abord affichée, puis le tableau `A`.

On peut additionner des tableaux de mêmes tailles.

```
>>> B = np.ones(shape=A.shape,dtype=float)
>>> A+B
array([[2., 3., 4.],
       [3., 5., 7.]])
```

On peut procéder à une multiplication termes à termes de deux matrices de mêmes tailles.

```
>>> A*A
array([[ 1.,  4.,  9.],
       [ 4., 16., 36.]])
```

On peut réarranger les termes d'un tableau, cela se fait en lisant les éléments de la matrice en ligne

```
>>> A.reshape((6,))
array([1., 2., 3., 2., 4., 6.])
>>> A.reshape((2,3))
array([[1., 2., 3.],
       [2., 4., 6.]])
>>> A.reshape((3,2))
array([[1., 2.],
       [3., 2.],
       [4., 6.]])
>>> A.reshape((6,1))
array([[1.],
 [2.],
 [3.],
 [2.],
 [4.],
 [6.]])
>>> A.reshape((1,6))
array([[1., 2., 3., 2., 4., 6.]])
```

On peut définir une matrice identité grâce à `diag`

```
>>> C = np.diag([1,1,1])
>>> C
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

Une particularité de Python est qu'on distingue le tableau à une dimension du vecteur ligne ou du vecteur colonne. Et en général, on n'utilise pas la notion de vecteur ligne ou colonne.

Le produit matriciel s'implémente avec @

```
>>> A@np.array([1,0,1])
array([4., 8.])
>>> np.array([1,1])@A
array([3., 6., 9.])
```

Ainsi le produit scalaire entre deux vecteurs s'implémente avec @

```
>>> x=np.array([1,1,1],dtype=float)
>>> x@x
np.float64(3.0)
```

Mais le produit d'un vecteur colonne avec un vecteur ligne s'implémente aussi avec @ à condition de convertir les tableaux à une dimension en un vecteur colonne et un vecteur ligne.

```
x.reshape((3,1))@x.reshape((1,3))
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Remarquez que `.reshape` est prioritaire par rapport à @. La raison est `.reshape` s'applique à un argument, tandis que @ s'applique à deux arguments.