2.4 Les modules – Les packages

Nous avons vu que dans un script Python (fichier avec une extension **.py**), il était possible de définir des fonctions réutilisables afin d'éviter les répétitions de code.
Jusqu'à présent, dans les scripts que nous avons étudiés, les fonctions étaient toujours définies au début du script.
Il est cependant possible d'effectuer la définition des fonctions dans un fichier séparé pour ensuite utiliser les fonctions dans un script principal.

. Les modules

Un programme Python est donc généralement composé de plusieurs fichiers sources avec une extension .**py**, appelés **modules** indépendants les uns des autres et pouvant être réutilisés dans d'autres programmes.

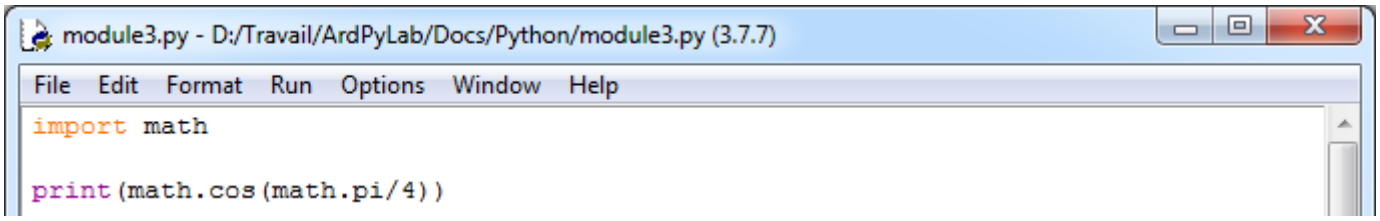
Un module rassemble les fonctions utilisées par le programme principal dans un même fichier situé dans le répertoire d'exécution du programme (pour les modules personnels).

Pour pouvoir utiliser les fonctions du module, le programme principal doit les importer.

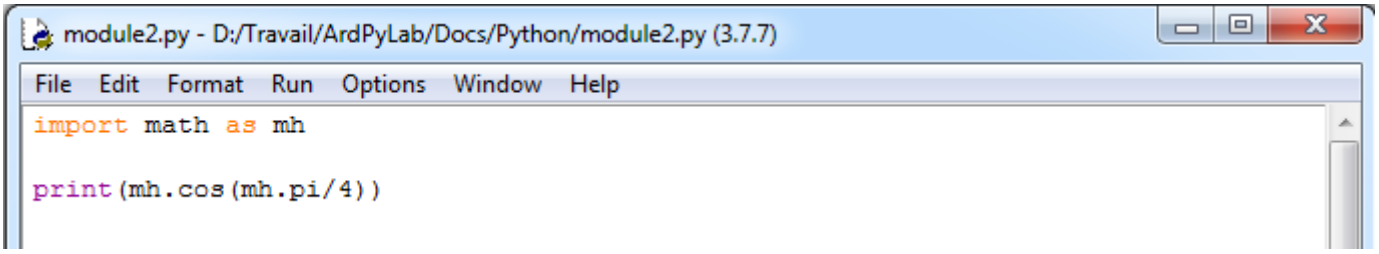
Deux syntaxes sont possibles pour l'importation d'un module :

. la commande **import nom_module** importe la totalité des objets du module.

Exemple : import math



Le module peut être importé sous un autre nom, un alias. On utilise alors cet alias pour appeler les fonctions :



. la commande **from nom_module import obj1, obj2**... n'importe que les objets **obj1**, **obj2**... du module :

Exemple: from math import pi, sin, log

Dans cet exemple, seules les fonctions **cos** et **pi** du module math sont importées :

module4.py - D:/Travail/ArdPyLab/Docs/Python/module4.py (3.7.7)

File Edit Format Run Options Window Help

```
from math import cos, pi

print(cos(pi/4))
```

Ou toutes les fonctions du module :

module.py - D:/Travail/ArdPyLab/Docs/Python/module.py (3.7.7)

File Edit Format Run Options Window Help

```
from math import *

print(cos(pi/4))
```

(Le caractère **'*'** permet d'importer toutes les fonctions du module.)

Exemple :

Reprenons le programme d'affichage de la table de multiplication en créant un module appelé **table.py** contenant les fonctions du programme.

table.py - D:/Travail/ArdPyLab/Docs/Python/table.py (3.7.7)

File Edit Format Run Options Window Help

```
#Fonction affichage table de multiplication

def table(nb=1, max=10):
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1

# Fonction saisie d'un nombre

def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre
```

Ln: 20 Col: 17

Le programme principal importe toutes les fonctions du module table et les utilise :

table_main.py - D:/Travail/ArdPyLab/Docs/Python/table_main.py (3.7.7)

File Edit Format Run Options Window Help

```
from table import *

nombre=saisie("Veuillez saisir le nombre à multiplier:")
multimax=saisie("Veuillez saisir la valeur maximale du multiplicateur:")

table(nombre,multimax)
```

Ln: 6 Col: 22

Résultat dans la fenêtre Python Shell :

```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/table_main.py ============
Veuillez saisir le nombre à multiplier:5
Veuillez saisir la valeur maximale du multiplicateur:9
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
```

. Les packages

Quand on a un grand nombre de modules, il est intéressant de les organiser dans des dossiers. Un dossier qui rassemble des modules est appelé un **package** (paquets en français). Ce dossier doit contenir un fichier nommé **__init__.py** vide ou décrivant l'arborescence du package. Le fichier **__init__.py** même vide est essentiel pour que Python considère les dossiers le contenant comme des paquets.

Exemple :

Toujours avec programme d'affichage de la table de multiplication, nous allons créer un module **table.py** contenant la fonction d'affichage de la table :

table.py - D:\Travail\ArdPyLab\Docs\Python\fonctions\table.py (3.7.7)

File Edit Format Run Options Window Help

```
#Fonction affichage table de multiplication

def table(nb=1, max=10):
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Et un module **saisie.py** contenant la fonction de saisie d'un nombre :

saisie.py - D:\Travail\ArdPyLab\Docs\Python\fonctions\saisie.py (3.7.7)

File Edit Format Run Options Window Help

```
# Fonction saisie d'un nombre

def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre
```

Ces deux modules ainsi que le fichier **__init__.py** sont situés dans un dossier nommé **fonctions**. Dans ce cas simple, le fichier **__init__.py** peut être vide, mais il peut également décrire l'arborescence du dossier **fonctions** :

init.py - D:\Travail\ArdPyLab\Docs\Python\fonctions_init_.py (3.7.7)

File Edit Format Run Options Window Help

```
fonctions/
    _init_.py
    saisie.py
    table.py
```

Le dossier **fonctions** est situé dans le répertoire d'exécution du programme principal qui va importer la fonction **table** du module **table.py** et la fonction **saisie** du module **saisie.py** :

table_main2.py - D:/Travail/ArdPyLab/Docs/Python/table_main2.py (3.7.7)

File Edit Format Run Options Window Help

```
# importation des modules du package

from fonctions.table import table  # ou from fonctions.table import *
from fonctions.saisie import saisie # ou from fonctions.table import *

# appel des fonctions

nombre=saisie("Veuillez saisir le nombre à multiplier:")
multimax=saisie("Veuillez saisir la valeur maximale du multiplicateur:")

table(nombre,multimax)
```

Résultat dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/table_main2.py ==========
Veuillez saisir le nombre à multiplier:5
Veuillez saisir la valeur maximale du multiplicateur:5
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
```

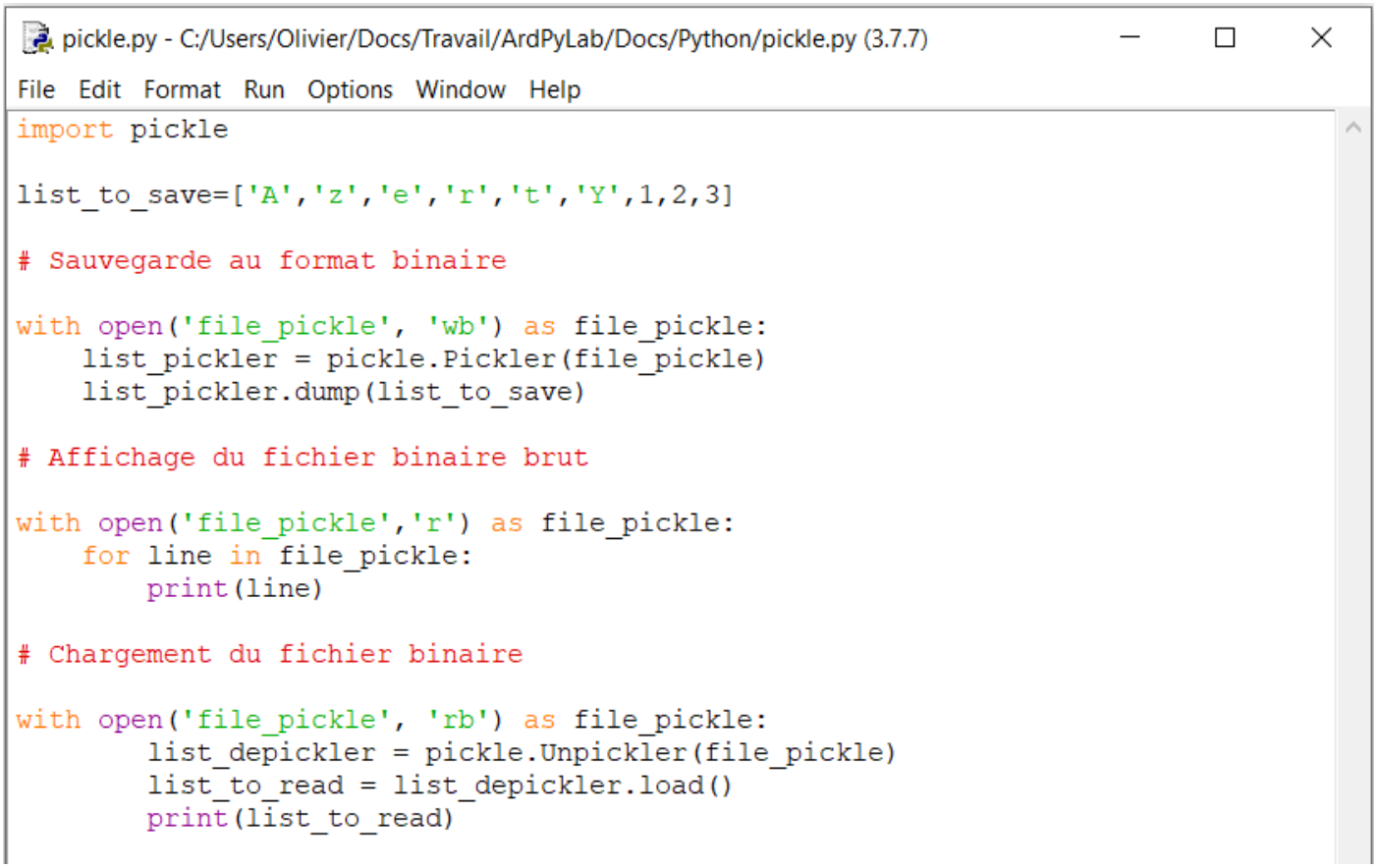
. La bibliothèque standard de Python

Par défaut, Python dispose de nombreux packages et modules intégrés à sa bibliothèque standard. On peut citer :

. **random** : fonctions permettant de travailler avec des valeurs aléatoires
. **math** : toutes les fonctions utiles pour les opérations mathématiques (cosinus,sinus,exp,etc.)
. **sys** : fonctions systèmes
. **os** : fonctions permettant d'interagir avec le système d'exploitation
. **time** : fonctions permettant de travailler avec le temps
. **tkinter** : interface graphique
…

Par exemple, le module **pickle** permet de sauvegarder dans un fichier au format binaire, n'importe quel objet Python (une liste, un dictionnaire, un tuple, etc…). C'est une alternative intéressante à la sauvegarde des données dans un fichier texte.

Le script ci-dessous enregistre une liste appelée **list_to_save** dans un fichier binaire nommé **file_pickle**, puis ouvre le fichier en mode lecture normal et l'affiche dans la fenêtre Python Shell pour montrer qu'il s'agit bien d'un fichier binaire, et enfin charge le fichier en mode lecture binaire et l'affiche dans la fenêtre Python Shell :



Déroulement du programme

- Sauvegarde au format binaire :

Le fichier **file_pickle** est d'abord ouvert ou créé en mode écriture binaire **('wb')**. Avec la méthode **Pickler** du module **pickle,** on crée l'objet **list_pickler** rattaché au fichier binaire. La méthode **dump** appliquée à cet objet permet d'enregistrer au format binaire la liste **list_to_save** dans le fichier **file_pickle**.

- Affichage du fichier binaire brut :

Le fichier est ouvert en mode lecture normal **('r')** et affiché dans la fenêtre Python Shell.

- Chargement du fichier binaire :

Le fichier **file_pickle** est d'abord ouvert en mode lecture binaire **('rb')**. Avec la méthode **Unpickler** du module **pickle,** on crée l'objet **list_depickler** rattaché au fichier binaire ouvert. La méthode **load** appliquée à cet objet permet d'affecter la liste sauvegardée en binaire à la liste **list_to_read** qui est ensuite affichée.

Résultats dans la fenêtre Python Shell

```
===== RESTART: C:/Users/Olivier/Docs/Travail/ArdPyLab/Docs/Python/pickle.py ====
€]q (X  AqX  zq X  eqX  rqX  tqX  YqKK Ke.
['A', 'z', 'e', 'r', 't', 'Y', 1, 2, 3]
>>>
```

Remarque :

Pour la méthode **dump**, le fichier doit toujours être ouvert en mode '**wb**' afin d'écraser le contenu précédent si le fichier existe déjà.

Exemple d'application :

Nous allons appliquer cette méthode de sauvegarde à notre programme de gestion d'inventaire.

Dans le script suivant, des fonctions d'ouverture et de sauvegarde au format binaire d'un dictionnaire nommé **inventaire** sont définies et utilisées pour afficher l'inventaire et enregistrer les modifications qui lui sont apportées (ajout ou suppression de lignes).

Invent-pickle.py - C:/Users/Olivier/Docs/Travail/ArdPyLab/Docs/Python/Invent-pickle.py (3.7.7)

File Edit Format Run Options Window Help

```
import pickle

# Fonction pour ouvrir le fichier d'inventaire et stocker les données dans
# un dictionnaire si le fichier existe:
def OuvreInventaire():
    global Inventaire
    Inventaire = {}
    try:
        with open('fileInvent', 'rb') as fichierInvent:
            Invent_depickler = pickle.Unpickler(fichierInvent)
            Inventaire = Invent_depickler.load()
    except Exception as message:
        print(message)

#Fonction pour sauvegarder l'inventaire en cours:
def SaveInventaire():
    global Inventaire
    with open('fileInvent', 'wb') as fichierInvent:
        Invent_pickler = pickle.Pickler(fichierInvent)
        Invent_pickler.dump(Inventaire)

# Fonction pour ajouter une ligne à l'inventaire ouvert ou initialement vide
# et enregistrer les modifications:
def AjoutMatos():
    global Inventaire
    Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
    Quant=input("saisissez la quantité de ce matériel:")
    Inventaire[Matos]=Quant
    SaveInventaire()

#Fonction pour effacer une ligne de l'inventaire en cours
# et enregistrer les modifications:
def EffaceMatos():
    global Inventaire
    element=input("saisissez l'élément à supprimer dans l'inventaire:")
    try:
        del Inventaire[element]
        SaveInventaire()
    except:
        print("L'élément que vous voulez supprimer n'existe pas!")

#Fonction pour afficher l'inventaire en cours:
def ReadInventaire():
    global Inventaire
    OuvreInventaire()
    for cle, valeur in Inventaire.items():
        print("{} : {}".format(cle, valeur))


# initialisation des variables:
Inventaire = {}
Fin =False

print("////////////////// INVENTAIRE MATERIEL /////////////////")
print("appuyer sur A pour ajouter un matériel à l'inventaire.")
print("appuyer sur S pour supprimer un matériel de l'inventaire.")
print("appuyer sur V pour afficher la liste de matériel.")
print("appuyer sur Q pour quitter.")
print("////////////////////////////////////////////////////////")
print(" ")

# programme principal:
while Fin == False:

    # attente d'un choix:
    choix=input()

    # Action en fonction de l'entrée clavier:
    if choix.upper() == "A": AjoutMatos()
    if choix.upper() == "V": ReadInventaire()
    if choix.upper() == "S": EffaceMatos()
    if choix.upper() == "Q":
        print("Fin du programme")
        Fin = True
```

Résultats dans la fenêtre Python Shell :

```
= RESTART: C:/Users/Olivier/Docs/Travail/ArdPyLab/Docs/Python/Invent-pickle.py =
////////////////// INVENTAIRE MATERIEL //////////////////
appuyer sur A pour ajouter un matériel à l'inventaire.
appuyer sur S pour supprimer un matériel de l'inventaire.
appuyer sur V pour afficher la liste de matériel.
appuyer sur Q pour quitter.
////////////////////////////////////////////////////////

v
[Errno 2] No such file or directory: 'fileInvent'
a
saisissez le type de matériel à ajouter à l'inventaire:pipette
saisissez la quantité de ce matériel:20
v
pipette : 20
a
saisissez le type de matériel à ajouter à l'inventaire:becher
saisissez la quantité de ce matériel:10
v
pipette : 20
becher : 10
s
saisissez l'élément à supprimer dans l'inventaire:becher
v
pipette : 20
q
Fin du programme
>>>
```

. Installation de bibliothèques (package) Python

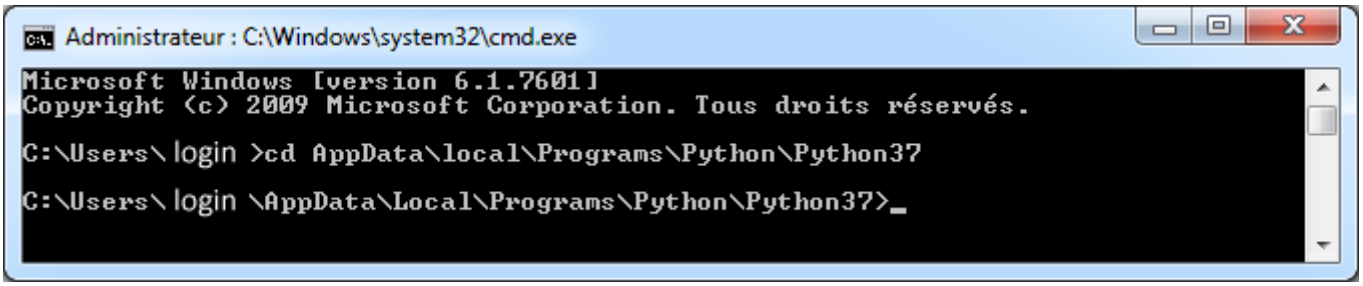
Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines.
Le site pypi.python.org/pypi (The Python Package Index) recense des milliers de modules et de packages !

Nous allons nous intéresser plus particulièrement aux bibliothèques **numpy** et **matplotlib**.

En effet, dans le domaine scientifique, la manipulation des données à travers des tableaux et le tracé de courbes caractéristiques est courante, et les bibliothèques **numpy** et **matplotlib** sont très utiles pour l'exploitation de ces données.

numpy et **matplotlib** ne font pas partie des bibliothèques standard de Python. Il faut donc ajouter ces bibliothèques à la distribution de Python.
A noter, que l'installation de **matplotlib** installe également la bibliothèque de calcul **numpy.**

Pour ajouter une bibliothèque à Python sous Windows, le plus simple est d'ouvrir une console de commandes (taper **cmd** dans la zone de recherche de Windows) et de se placer dans le dossier d'installation de Python :



(A adaptez bien-sûr à votre chemin d'installation, login et n° de version Python…)

Et de taper la commande suivante qui va installer la dernière version d'un package et de ses dépendances depuis le *Python Package Index* (**pip**) qui est l'outil d'installation de prédilection des bibliothèques (à partir de Python 3.4, il est inclus par défaut avec l'installateur de Python) :

```
python -m pip install package
```

Si un package est déjà installé, l'installer à nouveau n'aura aucun effet. La mise à jour de package existants doit être demandée explicitement :

```
python -m pip install --upgrade package
```

Donc, pour installer la bibliothèque **matplotlib**, il suffit de taper la ligne de commande suivante :

```
python -m pip install matplotlib
```

Et l'outil **pip** va télécharger sur le site de dépôts la librairie demandée ainsi que les dépendances nécessaires dont **numpy**.

. numpy

numPy (diminutif de numerical Python) est la bibliothèque indispensable pour le calcul scientifique avec Python.
Cette bibliothèque est utile pour manipuler des matrices ou tableaux multidimensionnels ainsi que les fonctions mathématiques opérant sur ces tableaux.

Voyons les bases de l'utilisation de numpy :

Il faut au départ importer le package numpy avec l'instruction recommandée suivante :

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

Toutes les fonctions de **NumPy** seront alors préfixées par **np**.

Le package **NumPy** permet la manipulation simple et efficace des tableaux en ajoutant à Python le type **array** similaire à une liste (type **list**). Mais contrairement aux listes, les tableaux **Numpy** ne peuvent contenir que des membres d'un seul type.

Pour créer un tableau Numpy, on peut convertir une liste avec la fonction **array()**:

```
>>> a= np.array([1, 2, 3, 4], int)
>>> a
array([1, 2, 3, 4])
>>> type(a)
<class 'numpy.ndarray'>
```

Le deuxième argument est optionnel et spécifie le type des éléments du tableau :

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([1., 4., 5., 8.])
```

Si dans la liste de départ, il y a des données de types différents, **Numpy** essaiera de les convertir toutes au type le plus général. Par exemple, les entiers **int** seront convertis en nombres à virgule flottante **float** :

```
>>> a = np.array([3.14, 4, 2, 3])
>>> a
array([3.14, 4.  , 2.  , 3.  ])
```

Un tableau peut être multidimensionnel ; ici 2 dimensions :

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

Comme pour les listes, on peut accéder aux éléments d'un tableau (attention, comme pour les listes, les indices des éléments commencent à zéro) :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a[0,2]
3
>>> a[1,1]
5
```

Et le slicing (découpage) extrait les tableaux :

```
>>> a = np.array([0,1,2,3,4,5,6,7,8,9])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [début:fin:pas]
array([2, 5, 8])
>>> a[2:8:3] # le dernier élément n'est pas inclus
array([2, 5])
>>> a[:5]    # le dernier élément n'est pas inclus
array([0, 1, 2, 3, 4])
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])  # le dernier élément est inclus
>>> a[2:9]
array([2, 3, 4, 5, 6, 7, 8]) # le dernier élément n'est pas inclus
```

Dans l'instruction **[début:fin:pas]**, deux des arguments peuvent être omis : par défaut l'indice de début vaut 0 (le 1er élément du tableau), et le pas vaut 1.

Un pas négatif inversera l'ordre du tableau :

```
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>> a[:2:-1]
array([9, 8, 7, 6, 5, 4, 3])
```

Et avec un début négatif, la lecture commence par la fin :

```
>>> a[-1::1]
array([9])
```

Pour un tableau bi-dimensionnel, on peut bien-sûr travailler avec les deux indices :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a[:1,:2]
array([[1, 2]])
>>> a[-1,:2]
array([4, 5])
```

Et on peut modifier les valeurs d'un tableau :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a[:1,:2]=7
>>> a
array([[7, 7, 3],
       [4, 5, 6]])
```

En fait, un tableau multidimensionnel est représenté par une liste de listes, et au final, un tableau bi-dimensionnel (lignes et colonnes) n'est rien d'autre qu'une liste de lignes, une ligne étant une liste de nombres.

On peut alors facilement créer un tableau bi-dimensionnel avec la fonction **range()** :

```
>>> a = np.array([range(i, i + 10) for i in [0, 10]])
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

Ce qui donne le tableau suivant :

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

Avec :

```
>>> a = np.array([range(i, i + 10) for i in [0, 10]])
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
>>> a[0]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1]
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> a[0,3]
3
>>> a[1,5]
15
```

Cependant, **Numpy** dispose de plusieurs fonctions pour créer directement des tableaux :

. Zeros()

```
# Un tableau bi-dimensionnel de taille 1x10, rempli d'entiers qui valent 0
>>> np.zeros(10, int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

. Ones()

```
# Un tableau bi-dimensionnel de taille 3x5, rempli de nombres à virgule flottante de valeur 1
>>> np.ones((3, 5), float)
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

. full()

```
# Un tableau bi-dimensionnel de taille 3x5, rempli de 2
>>> np.full((3, 5), 2)
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])
```

. arange()

```
# Un tableau bi-dimensionnel de taille 1x10, rempli d'une séquence linéaire d'entiers
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# Un tableau rempli d'une séquence linéaire de nombres à virgule flottante
>>> np.arange(2, 10, dtype=np.float)
array([2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(3.0)
array([0., 1., 2.])
# Un tableau rempli d'une séquence linéaire d'entiers par pas de 2
>>> np.arange(0, 20, 2)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
# Un tableau rempli d'une séquence linéaire de nombres à virgule flottante par pas de 0.1
>>> np.arange(2, 3, 0.1)
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

Remarques :

Dans l'instruction np.arange**(début:fin:pas)**, deux des arguments peuvent être omis :

. **début** : par défaut l'indice de début vaut 0 (le 1er élément du tableau)

. **pas** : par défaut le pas vaut 1.

Le dernier élément du tableau est l'argument **fin** auquel il faut retrancher le **pas**.

. linspace()

Comme il y quelques subtilités avec la fonction **arange()** quant au dernier élément, pour éviter tout problème, la fonction **linspace(premier,dernier,n)** renvoie un array commençant par **premier**, se terminant par **dernier** avec **n** éléments.

```
>>> np.linspace(1., 4., 6)
array([1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

. **reshape ()**

La fonction reshape() permet de redimensionner un tableau. Il faut cependant que le nombre d'éléments reste le même.

```
>>> a=np.arange(16)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> a=a.reshape(4,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a=a.reshape(2,8)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
```

NumPy dispose d'un grand nombre de fonctions mathématiques qui peuvent être appliquées directement à un tableau. Dans ce cas, la fonction est appliquée à chacun des éléments du tableau.

```
>>> x= np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y=2*x
>>> y
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
>>> x = np.linspace(-np.pi/2, np.pi/2, 3)
>>> x
array([-1.57079633,  0.        ,  1.57079633])
>>> y = np.sin(x)
>>> y
array([-1.,  0.,  1.])
```

Les fonctions mathématiques couramment utilisées sont :

. **numpy.sin(x)**	sinus
. **numpy.cos(x)**	cosinus
. **numpy.tan(x)**	tangente
. **numpy.arcsin(x)**	arcsinus
. **numpy.arccos(x)**	arccosinus
. **numpy.arctan(x)**	arctangente
. **x**n**	x à la puissance n, exemple : x**2
. **numpy.sqrt(x)**	racine carrée
. **numpy.exp(x)**	exponentielle
. **numpy.log(x)**	logarithme népérien
. **numpy.abs(x)**	valeur absolue

. **numpy.around(x,n)** arrondi à n décimales

On va donc pouvoir appliquer n'importe quelle fonction mathématique à un tableau de données **x** de façon à obtenir la caractéristique **y=f(x)**.

Cette caractéristique pourra être tracé à l'aide de la bibliothèque **matplolib**.

. matplotlib

matplotlib est une bibliothèque destinée à tracer et visualiser des données sous formes de graphiques.

Pour tracer des graphes **y=f(x)** avec les points reliés, où x et y sont fournis par des tableaux **numpy**, il faut importer le module **pyplot** de **matplotlib** (**numpy** étant aussi utilisé, il devra bien-sûr être également importé).

Nous allons commencer par un exemple simple, le tracé une droite d'équation : **y = ax+ b**

plot.py - D:/Travail/ArdPyLab/Docs/Python/plot.py (3.7.7)

File Edit Format Run Options Window Help

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

a = 2
b = 5
x = np.arange(10)
print('x:', x)
y = a*x+b
print('y:', y)

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

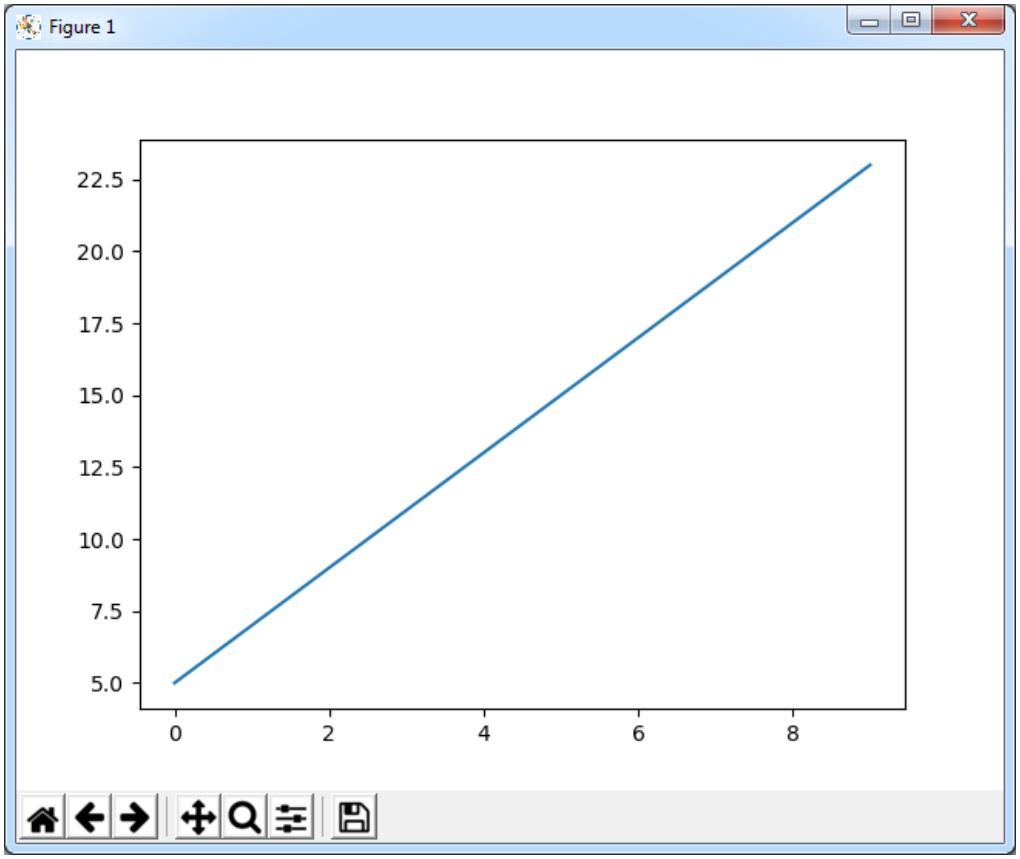
En premier, il faut créer un tableau numpy d'entiers, par exemple de 0 à 9, correspondant aux abscisses du graphe (**x=np.arange(10)**).

On applique la fonction de la droite à ce tableau pour obtenir un autre tableau numpy d'entiers correspondant aux ordonnées du graphes (**y= a*x + b**)

Résultats dans le fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/plot.py ==========
x: [0 1 2 3 4 5 6 7 8 9]
y: [ 5  7  9 11 13 15 17 19 21 23]
```

Le graphe est créé avec l'instruction **plt.plot(x,y)** et affiché avec **plt.show()** :



Selon le même principe, nous allons maintenant tracer une sinusoïde :

plot2.py - D:/Travail/ArdPyLab/Docs/Python/plot2.py (3.7.7)

File Edit Format Run Options Window Help

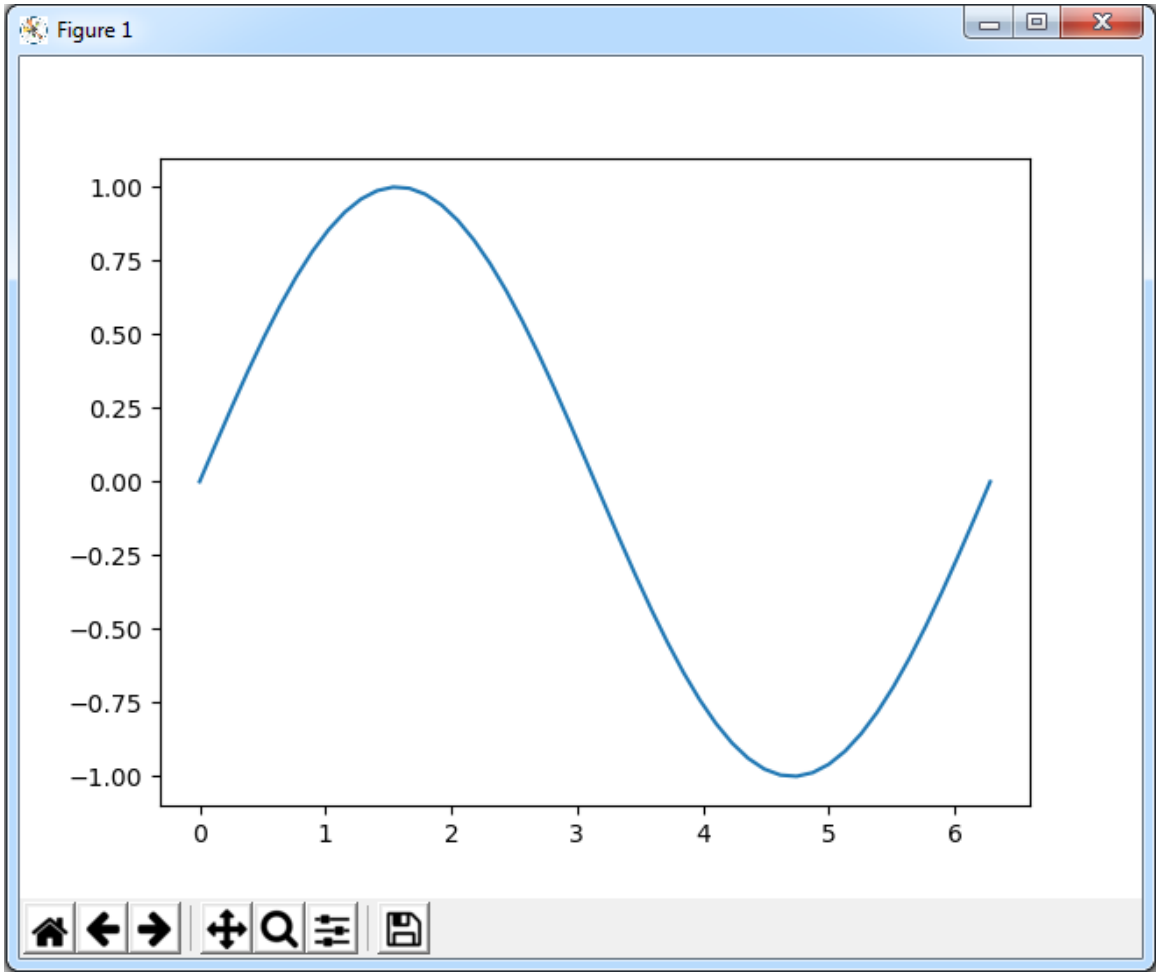
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
print('x:',x)
y = np.sin(x)
print('y:',y)

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

Dans cet exemple, le tableau des abscisses contient 50 éléments également espacés de 0 à 2π. On applique à ce tableau, la fonction sinus, pour obtenir le tableau des ordonnées de 50 éléments également (**y=np.sin(x)**).

Résultats dans le fenêtre Python Shell :

```
============== RESTART: D:/Travail/ArdPyLab/Docs/Python/plot2.py ==============
x: [0.         0.12822827 0.25645654 0.38468481 0.51291309 0.64114136
 0.76936963 0.8975979  1.02582617 1.15405444 1.28228272 1.41051099
 1.53873926 1.66696753 1.7951958  1.92342407 2.05165235 2.17988062
 2.30810889 2.43633716 2.56456543 2.6927937  2.82102197 2.94925025
 3.07747852 3.20570679 3.33393506 3.46216333 3.5903916  3.71861988
 3.84684815 3.97507642 4.10330469 4.23153296 4.35976123 4.48798951
 4.61621778 4.74444605 4.87267432 5.00090259 5.12913086 5.25735913
 5.38558741 5.51381568 5.64204395 5.77027222 5.89850049 6.02672876
 6.15495704 6.28318531]
y: [ 0.00000000e+00  1.27877162e-01  2.53654584e-01  3.75267005e-01
  4.90717552e-01  5.98110530e-01  6.95682551e-01  7.81831482e-01
  8.55142763e-01  9.14412623e-01  9.58667853e-01  9.87181783e-01
  9.99486216e-01  9.95379113e-01  9.74927912e-01  9.38468422e-01
  8.86599306e-01  8.20172255e-01  7.40277997e-01  6.48228395e-01
  5.45534901e-01  4.33883739e-01  3.15108218e-01  1.91158629e-01
  6.40702200e-02 -6.40702200e-02 -1.91158629e-01 -3.15108218e-01
 -4.33883739e-01 -5.45534901e-01 -6.48228395e-01 -7.40277997e-01
 -8.20172255e-01 -8.86599306e-01 -9.38468422e-01 -9.74927912e-01
 -9.95379113e-01 -9.99486216e-01 -9.87181783e-01 -9.58667853e-01
 -9.14412623e-01 -8.55142763e-01 -7.81831482e-01 -6.95682551e-01
 -5.98110530e-01 -4.90717552e-01 -3.75267005e-01 -2.53654584e-01
 -1.27877162e-01 -2.44929360e-16]
```

Le graphe est créé avec l'instruction **plt.plot(x,y)** et affiché avec **plt.show()** :



Les graphes ainsi obtenus peuvent être mis en forme :

- Domaine des axes des abscisses et des ordonnées :

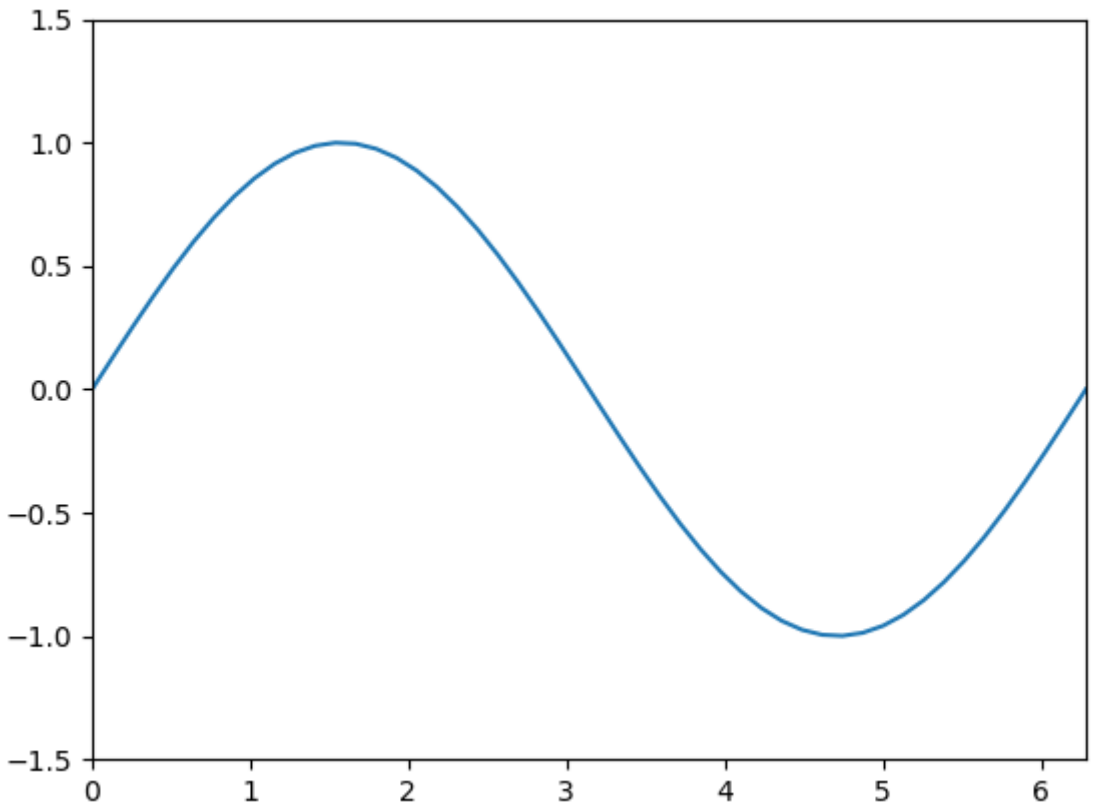
Il est possible fixer indépendamment les domaines des abscisses et des ordonnées en utilisant les fonctions **xlim()** et **ylim()**.

plot3.py - D:/Travail/ArdPyLab/Docs/Python/plot3.py (3.7.7)

File Edit Format Run Options Window Help

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

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.5, 1.5)

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

Ce qui donne :



- Ajout d'un titre :

On peut ajouter un titre grâce à l'instruction **title()** : **plt.title("titre")**

- Ajout d'une légende :

L'instruction **plt.legend()** ajoute la légende au graphe définie lors de la création du graphe :

plt.plot(x,y, "legende")

- Ajout d'étiquettes sur les axes :

Les fonctions **xlabel()** et **ylabel()** ajoutent respectivement des étiquettes sur les axes des abscisses et des ordonnées.

plt.xlabel("abscisses")

plt.ylabel("ordonnees")

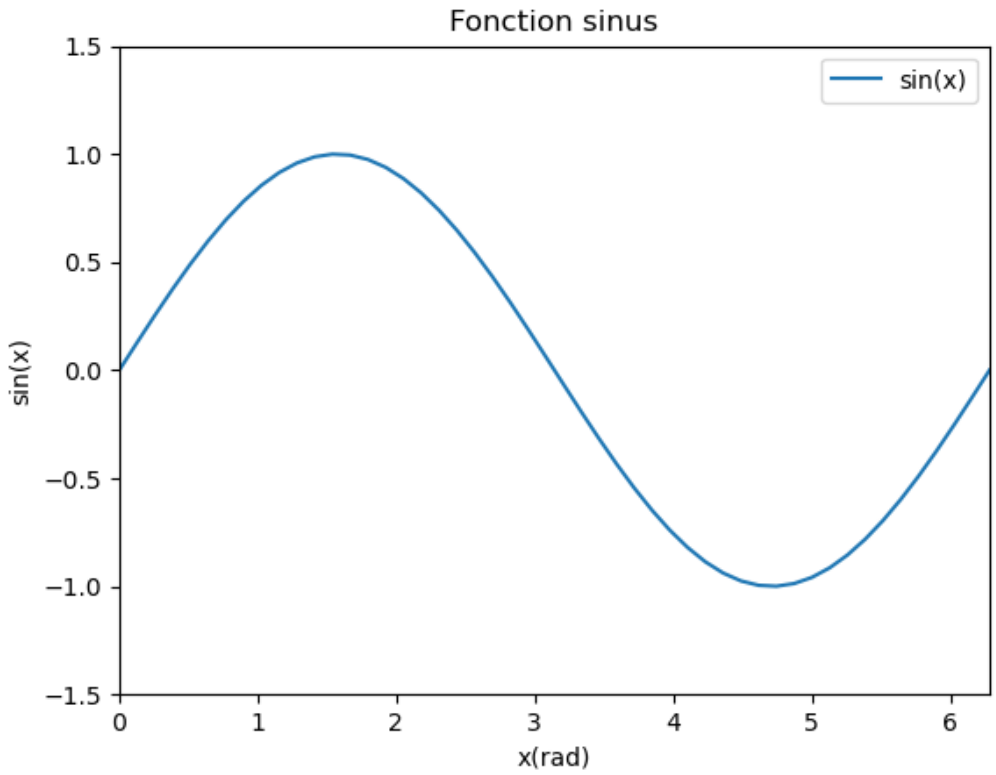
Exemple de mise en forme :

plot4.py - D:/Travail/ArdPyLab/Docs/Python/plot4.py (3.7.7)

File Edit Format Run Options Window Help

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

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.5, 1.5)

plt.title("Fonction sinus")

plt.xlabel("x(rad)")
plt.ylabel("sin(x)")

plt.plot(x, y, label="sin(x)")

plt.legend()
plt.show()
```

Ce qui donne :



Il est possible d'afficher plusieurs courbes sur le même graphe. Pour chaque courbe, il suffit de définir des tableaux **numpy** correspondant aux valeurs des ordonnées des caractéristiques **y=f(x)**.

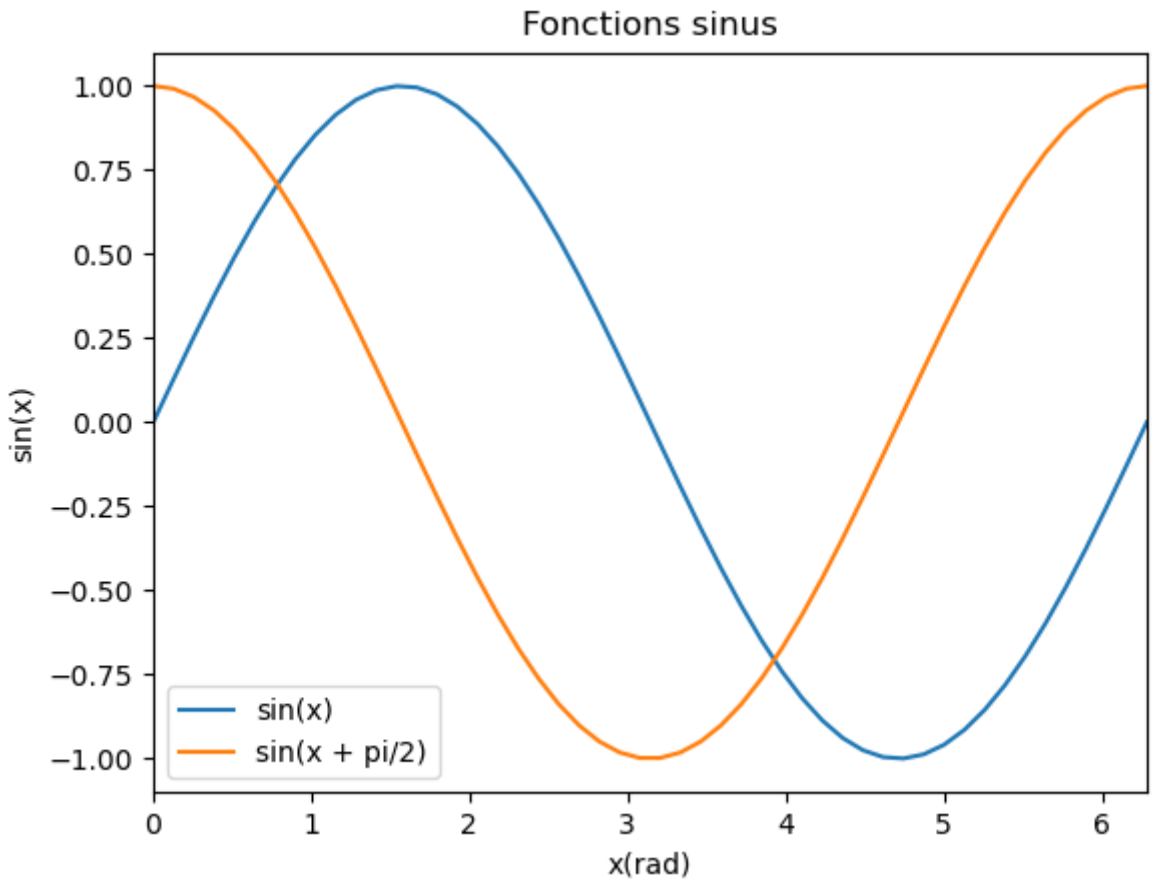
Exemple : Affichage de deux sinusoïdes avec un déphasage de π/2

plot5.py - D:/Travail/ArdPyLab/Docs/Python/plot5.py (3.7.7)

File Edit Format Run Options Window Help

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

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.sin(x+np.pi/2)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.1, 1.1)

plt.title("Fonctions sinus")

plt.xlabel("x(rad)")
plt.ylabel("sin(x)")

plt.plot(x, y, label="sin(x)")
plt.plot(x, y2, label="sin(x + pi/2)")

plt.legend()
plt.show()
```

Ce qui donne :



-Style des courbes

Le style des courbes est modifiable en précisant la couleur, le style de ligne et les symboles des points ("marker") en ajoutant une chaîne de caractères à l'instruction de création du graphe, de la façon suivante :

plot(x, y, "couleur symbole style de ligne", label="label")

. couleurs

Les chaînes de caractères suivantes permettent de définir la couleur :

Chaîne	Couleur
b	Blue (bleu)
g	Green (vert)
r	Red (rouge)
c	Cyan
m	magenta
y	Yellow (jaune)
k	Black (noir)
w	White (blanc)

. Styles de ligne

Les chaînes de caractères suivantes permettent de définir le style de ligne :

Chaîne	Effet
-	ligne continue
--	tirets
:	ligne en pointillé
-.	tirets points

. Symboles

Les chaînes de caractères suivantes permettent de définir le symbole ("marker") :

Chaîne	Effet	Chaîne	Effet
.	point marker	s	square marker
,	pixel marker	p	pentagon marker
o	circle marker	*	star marker
v	triangle_down	h	hexagon1 marker
^	triangle_up marker	H	hexagon2 marker
<	triangle_left marker	+	plus marker
>	triangle_right	x	x marker
1	tri_down marker	D	diamond marker
2	tri_up marker	d	thin_diamond marker
3	tri_left marker	|	vline marker
4	tri_right marker	_	hline marker

Remarque :

Pour une représentation graphique en nuage de points, il suffit d'indiquer un symbole sans préciser un style de ligne.

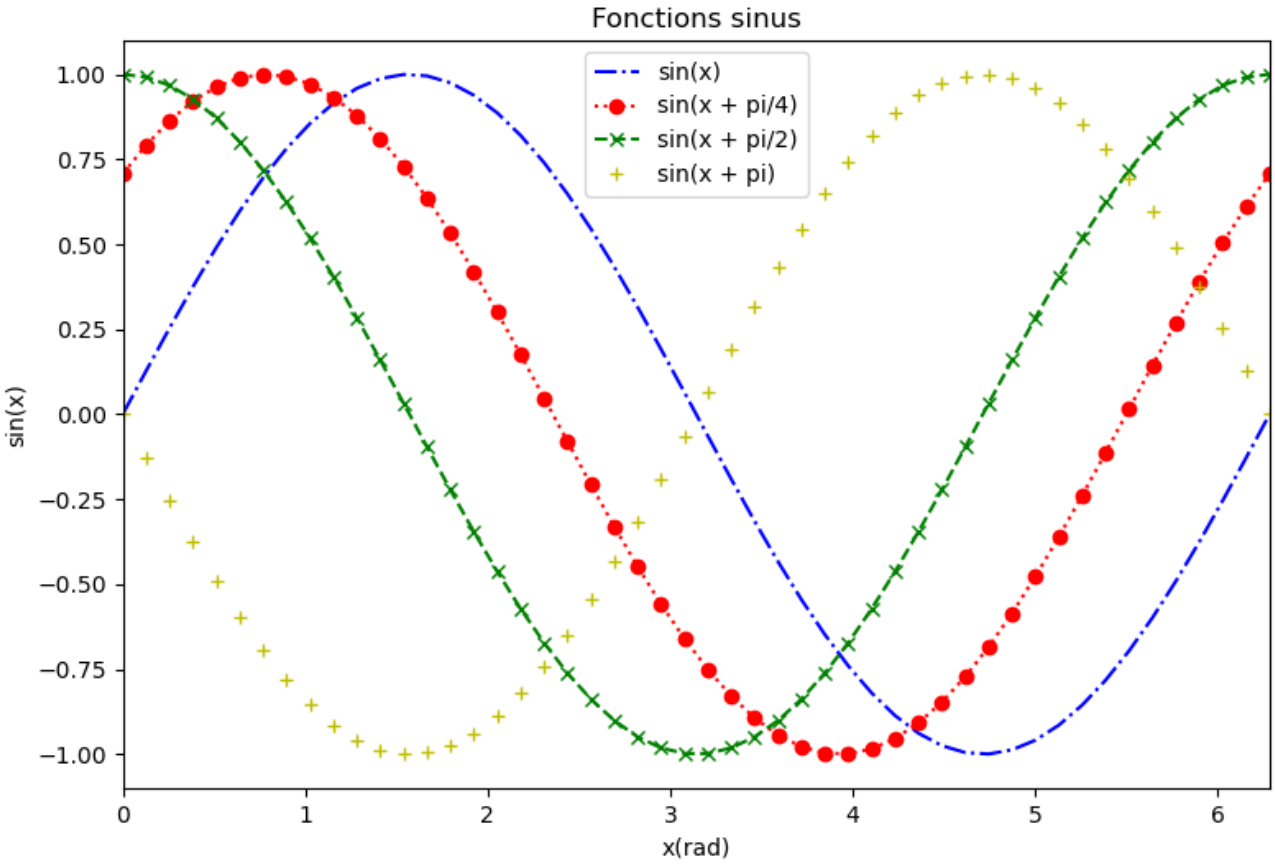
Exemples de styles de courbe :

plot5.py - D:\Travail\ArdPyLab\Docs\Python\plot5.py (3.7.7)

File Edit Format Run Options Window Help

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

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.sin(x+np.pi/4)
y3 = np.sin(x+np.pi/2)
y4 = np.sin(x+np.pi)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.1, 1.1)

plt.title("Fonctions sinus")

plt.xlabel("x(rad)")
plt.ylabel("sin(x)")

plt.plot(x, y, "b-.",label="sin(x)")
plt.plot(x, y2,"r:o", label="sin(x + pi/4)")
plt.plot(x, y3,"g--x", label="sin(x + pi/2)")
plt.plot(x, y4,"y+", label="sin(x + pi)")

plt.legend()
plt.show()
```

Ce qui donne :

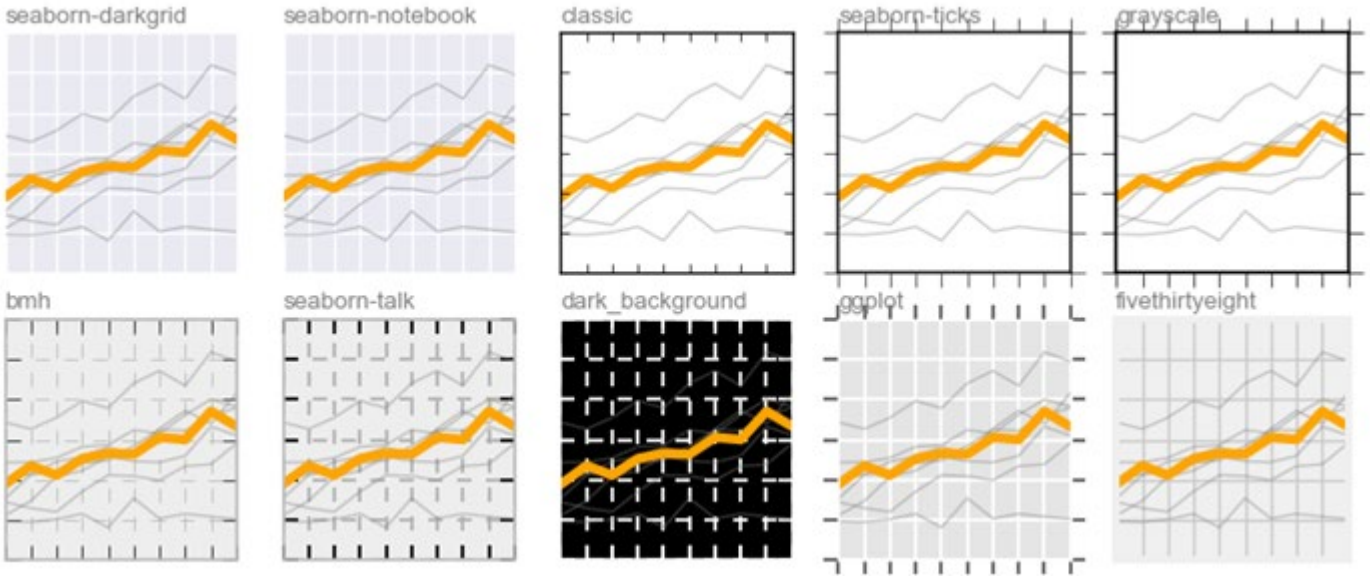


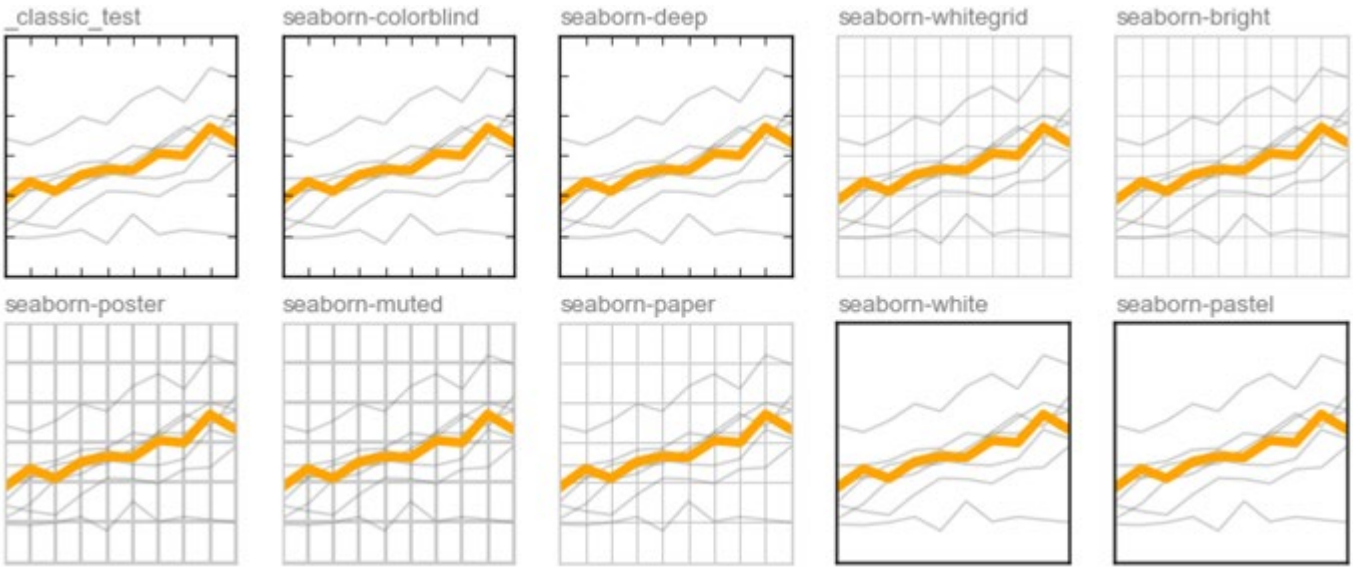
- <u>style de graphe</u> :

Le style du graphe est modifiable avec l'instruction :

pyplot.style.use('nom du style')

Voici quelques styles de graphes de **matplotlib** :





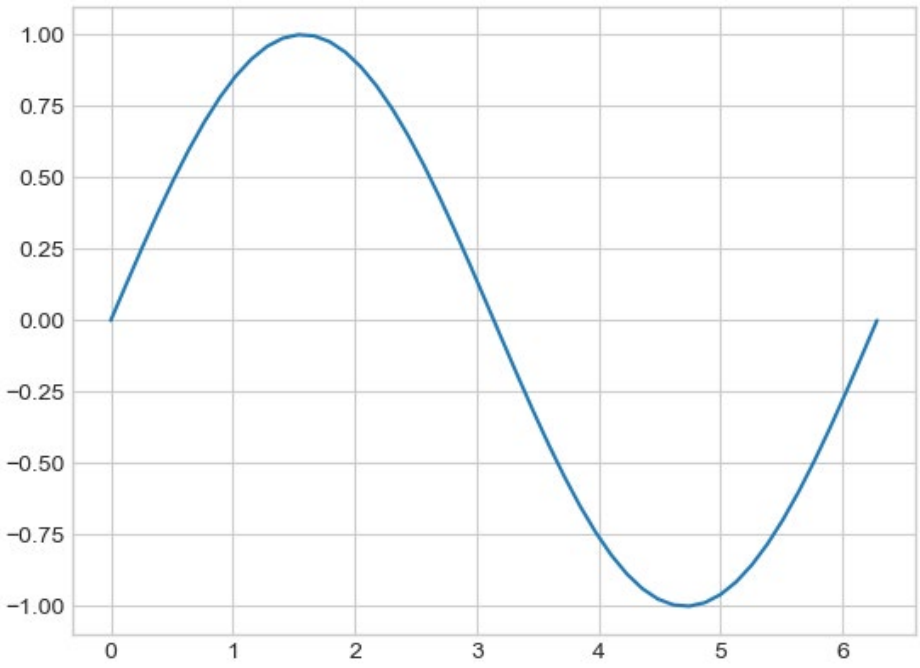
Exemple : Style ‘’seaborn-whitegrid’’

plot6.py - D:/Travail/ArdPyLab/Docs/Python/plot6.py (3.7.7)

File Edit Format Run Options Window Help

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

plt.style.use('seaborn-whitegrid')

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)


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

Ce qui donne :



- Disposition et graphes multiples :

Il est possible d'afficher plusieurs graphes sur la même figure en créant un objet **figure** dont on peut préciser la taille en pouce :

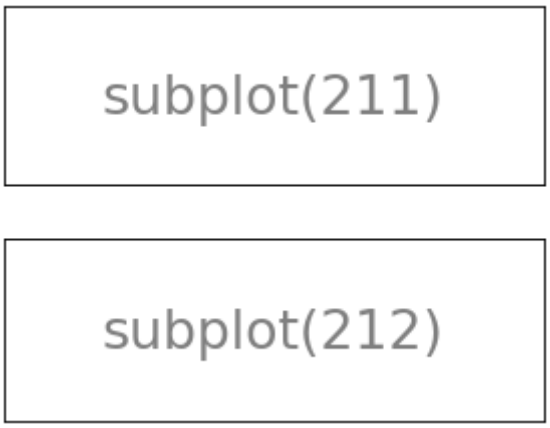
fig = pyplot.figure(figsize = (10, 10))

Les graphes dans cette figure sont modélisés par des objets **axes**. Pour créer un graphe dans la figure **fig,** on crée un objet **axe** à l'aide de la fonction **subplot()** en spécifiant le nombre de lignes et le nombre de colonnes dans la figure, ainsi que le numéro du graphe :

ax1 = pyplot.subplot(211)

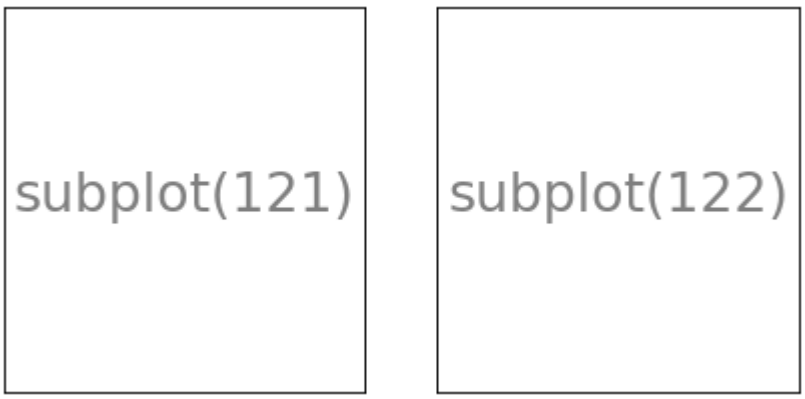
ax2 = pyplot.subplot(212)

Ces instructions vont créer 2 lignes et 1 colonne dans la figure, les 2 lignes contenant chacune 1 graphe :



Ou : **ax1 = plt.subplot(121)**

ax2 = plt.subplot(122)



1 ligne, 2 colonnes

Ou : **ax1 = plt.subplot(221)**

ax2 = plt.subplot(222)

ax3 = plt.subplot(223)

ax4 = plt.subplot(224)

2 lignes, 2 colonnes

Et ainsi de suite…

Les graphes sont créés avec la fonction **plot()** appliquée aux axes définis :

ax1.plot(x, y)

ax2.plot(x, y2)

…

et affichés avec la fonction **show()** appliquée à la figure :

fig.show()

La définition des styles de courbes reste le même :

axe.plot(x, y, "couleur symbole style de ligne", label="label")

et l'affichage de la légende se fait sur l'objet **axe** :

axe.legend()

La mise en forme des graphes (titre, échelles, étiquettes des axes) est cependant légèrement différent :

- Ajout d'un titre sur un objet **axe** :

axe.set_title("titre")

- Définition des échelles des axes :

axe.set_xlim(x_1, x_2)

axe.set_ylim(y_1, y_2)

- Ajout d'étiquettes sur les axes :

axe.set_xlabel("labelx")

axe.set_ylabel("labely")

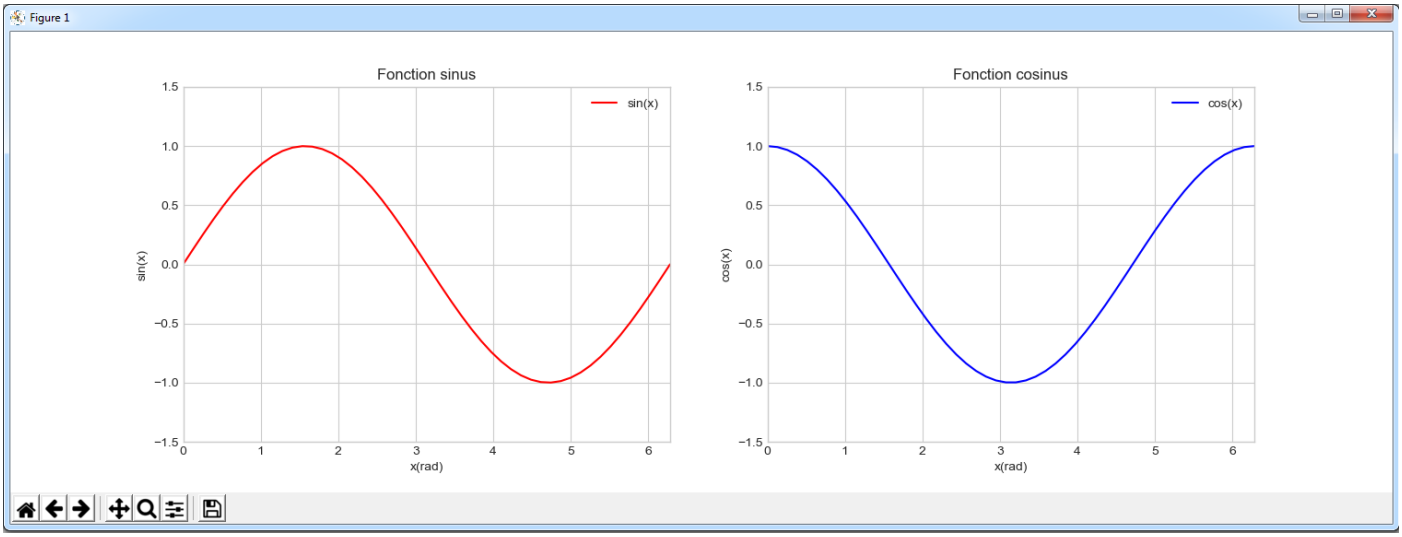
Exemples :

. 2 graphes sur 1 ligne / 2 colonnes :

fig2.py - D:\Travail\ArdPyLab\Docs\Python\fig2.py (3.7.7)

File Edit Format Run Options Window Help

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

plt.style.use('seaborn-whitegrid')

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.cos(x)

fig = plt.figure(figsize = (15, 5))
ax1 = plt.subplot(121)
ax2 = plt.subplot(122)

ax1.set_title("Fonction sinus")
ax2.set_title("Fonction cosinus")

ax1.set_xlim(0, 2*np.pi)
ax1.set_ylim(-1.5, 1.5)
ax2.set_xlim(0, 2*np.pi)
ax2.set_ylim(-1.5, 1.5)

ax1.set_xlabel("x(rad)")
ax2.set_xlabel("x(rad)")
ax1.set_ylabel("sin(x)")
ax2.set_ylabel("cos(x)")

ax1.plot(x, y,"r",label="sin(x)")
ax2.plot(x, y2,"b",label="cos(x)")

ax1.legend()
ax2.legend()

fig.show()
```



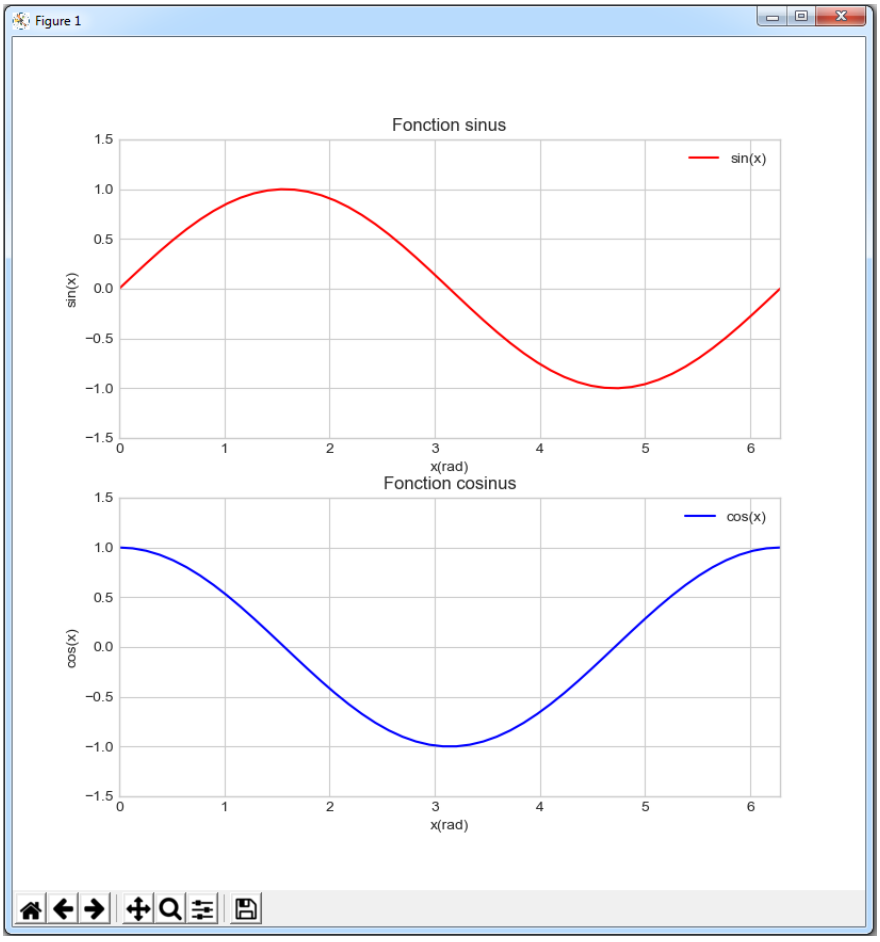
. 2 graphes sur 2 lignes / 1 colonne :

Le code est identique à celui de l'exemple précèdent seul la taille de la figure et la disposition des graphes changent :

fig = plt.figure(figsize = (8, 8))

ax1 = plt.subplot(211)

ax2 = plt.subplot(212)

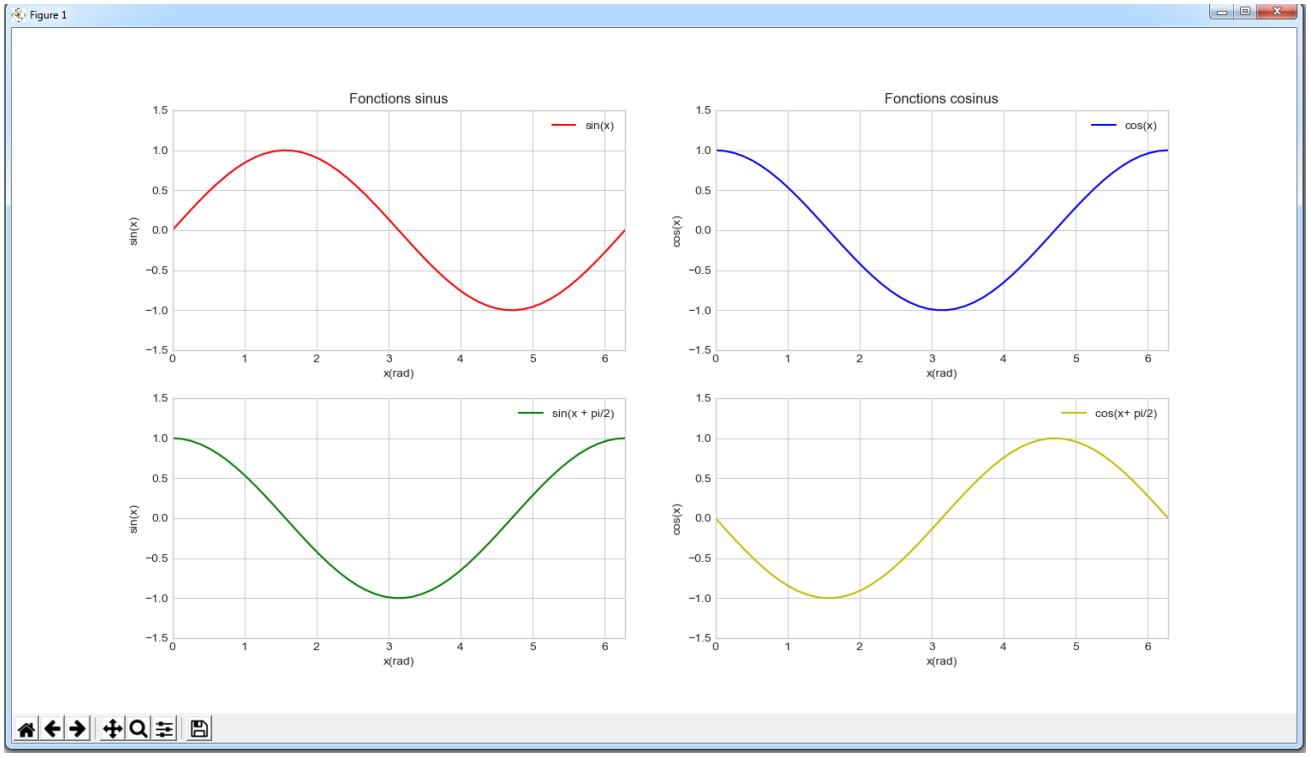


. 4 graphes sur 2 lignes / 2 colonnes :

fig3.py - D:/Travail/ArdPyLab/Docs/Python/fig3.py (3.7.7)

File Edit Format Run Options Window Help

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

plt.style.use('seaborn-whitegrid')

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.cos(x)
y3 = np.sin(x+np.pi/2)
y4 = np.cos(x+np.pi/2)

fig = plt.figure(figsize = (15, 8))
ax1 = plt.subplot(221)
ax2 = plt.subplot(222)
ax3 = plt.subplot(223)
ax4 = plt.subplot(224)

ax1.set_title("Fonctions sinus"), ax2.set_title("Fonctions cosinus")

ax1.set_xlim(0, 2*np.pi),ax1.set_ylim(-1.5, 1.5)
ax2.set_xlim(0, 2*np.pi),ax2.set_ylim(-1.5, 1.5)
ax3.set_xlim(0, 2*np.pi),ax3.set_ylim(-1.5, 1.5)
ax4.set_xlim(0, 2*np.pi),ax4.set_ylim(-1.5, 1.5)

ax1.set_xlabel("x(rad)"),ax2.set_xlabel("x(rad)")
ax3.set_xlabel("x(rad)"),ax4.set_xlabel("x(rad)")
ax1.set_ylabel("sin(x)"),ax3.set_ylabel("sin(x)")
ax2.set_ylabel("cos(x)"),ax4.set_ylabel("cos(x)")

ax1.plot(x, y,"r",label="sin(x)"),ax2.plot(x, y2,"b",label="cos(x)")
ax3.plot(x, y3,"g",label="sin(x + pi/2)"),ax4.plot(x, y4,"y",label="cos(x+ pi/2)")

ax1.legend(),ax2.legend(),ax3.legend(),ax4.legend()

fig.show()
```



Tout ce qui vient d'être vu n'est qu'une infime partie des possibilités que peut offrir **matplotlib**. Pour plus d'informations sur **matplotlib**, de nombreux tutoriels sont disponibles sur le site :

https://matplotlib.org/3.1.1/tutorials/