Les scripts Python

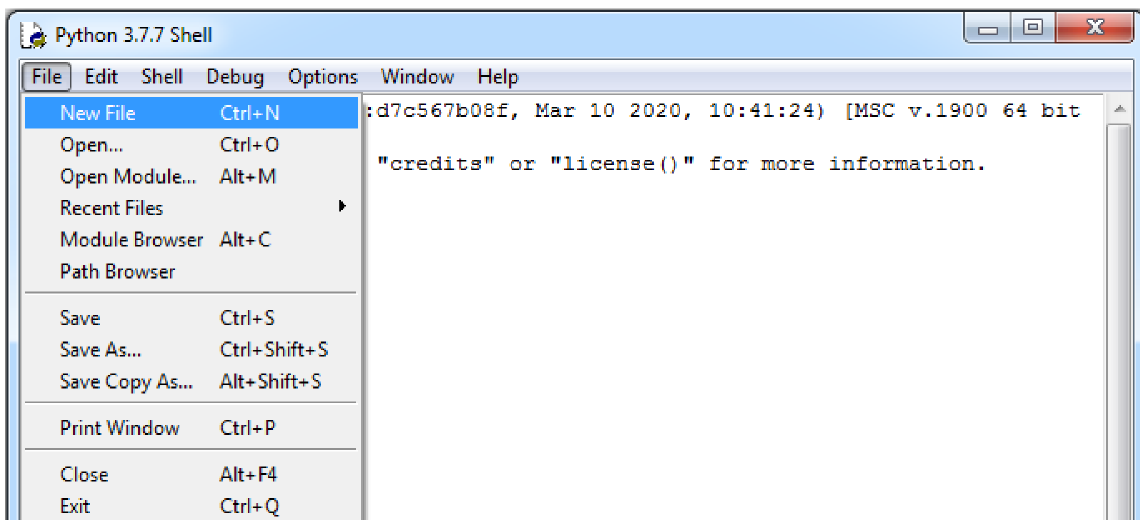
En mode interactif, les lignes d'instructions ne sont plus accessibles une fois exécutées. Mais il est bien-sûr possible d'écrire et de conserver un programme (un script), à l'aide d'un éditeur, pour pouvoir l'exécuter à loisir ou pour le modifier ultérieurement.

Il existe de nombreux éditeurs de scripts Python qui intègre également un interpréteur pour exécuter les programmes. C'est ce qu'on appelle un IDE ou Environnement de développement (Integrated Development Environment).

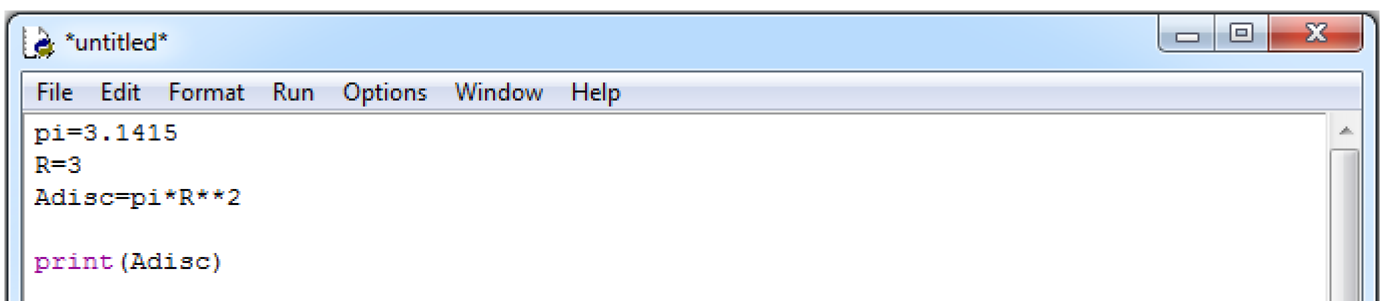
C'est un environnement de programmation complet qui se présente sous la forme d'une application. Il se compose généralement d'un éditeur de code, d'un interpréteur, d'un débogueur… On peut citer **Pycharm**, **Spider**…

Mais pour une initiation à la programmation en Python, l'utilisation de l'interpréteur **IDLE** qui permet également d'éditer des scripts est tout à fait adapter :

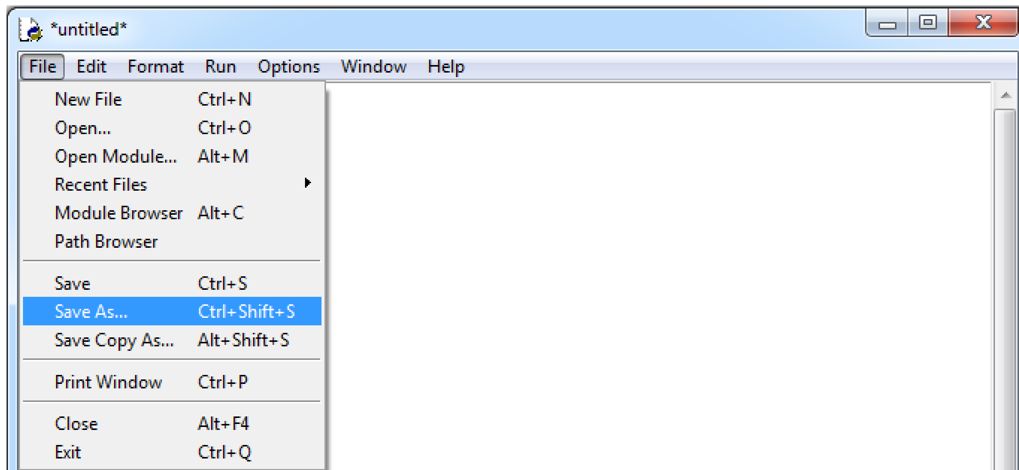
- Dans la fenêtre Python Shell (celle du mode interactif), sélectionnez "**New File**" dans le menu "**File**" :



- Une nouvelle fenêtre s'ouvre alors. C'est dans cette fenêtre que l'on va écrire notre premier programme (calcul de la surface d'un disque de rayon R) :



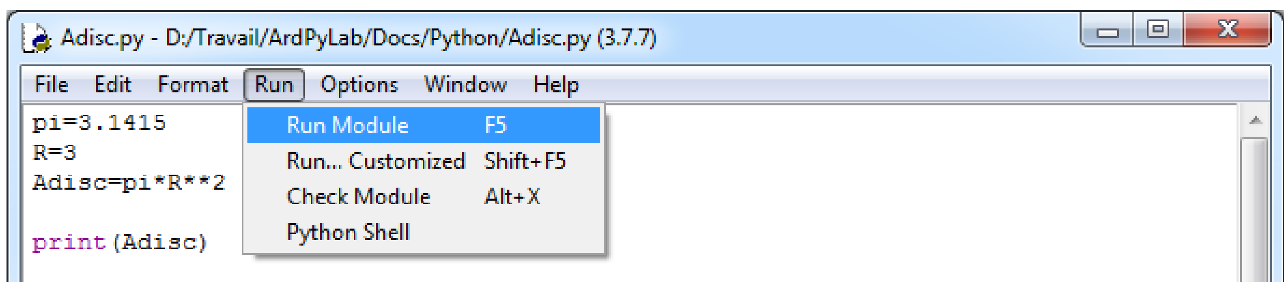
- Enregistrement du programme : Sélectionnez ”**Save as”** dans le menu ”**File”** :



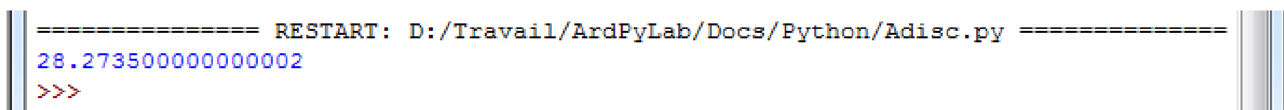
Une fenêtre d’enregistrement s’ouvre alors. Choisissez dans l’arborescence le dossier dans lequel vous voulez enregistrer le programme, puis dans le champ d’enregistrement du fichier saisissez le nom du programme suivi de l’extension ”.py”, puis cliquez sur enregistrer.

- Pour ouvrir un programme python précédemment enregistré, il suffira de sélectionner ”**Open**” dans le menu ”**File**” de l’environnement IDLE puis de chercher dans vos dossiers le fichier python à ouvrir.

- Exécution du programme : Pour exécuter le programme, il suffit de sélectionner ”**Run Module**” dans le menu ”**Run**” (si une modification du script a été effectuée, on vous proposera d’enregistrer le script modifié avant de l’exécuter).



Le programme s’exécute dans la fenêtre Python shell :



On pourra alors modifier notre premier programme, en demandant par exemple à l’utilisateur de saisir le rayon du disque. Les modifications seront enregistrées et le programme ré-exécuté autant de fois que vous souhaitez.

2.1 Structure des scripts Python

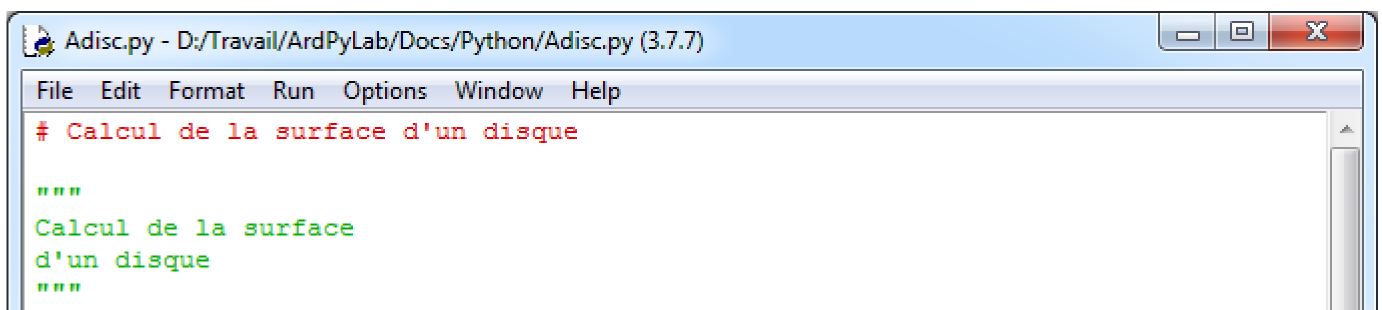
Un script Python est formée d'une suite d'instructions exécutées de haut en bas du script.

. Les instructions

Chaque instruction s'écrit sur une ligne, il n'y a pas de séparateur d'instruction. Si une ligne est trop grande, le caractère "****" permet de passer à la ligne suivante.

On utilise le caractère "**#** "pour insérer des commentaires dans un programme. Les commentaires vont du caractère "**#**" jusqu'à la fin de la ligne.

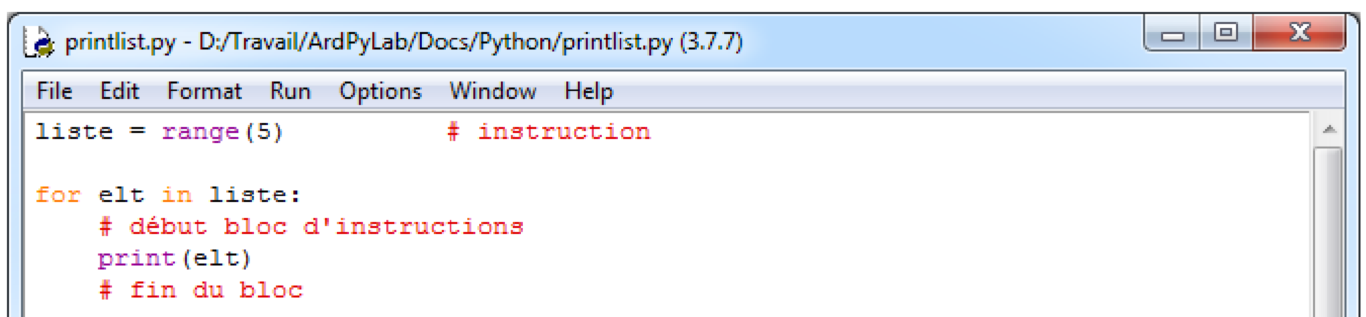
Il n'existe pas de commentaires en bloc comme en C **(/* ... */**). Mais il est possible d'utiliser des triples guillemets doubles ou simples (**'''**) avant et après le bloc de commentaires comme pour déclarer une variable chaine de caractères sur plusieurs lignes. Le bloc n'étant pas rattaché à une variable, il sera ignoré par l'interpréteur.
Sous IDLE, le plus simple est cependant de sélectionner le bloc de commentaires et de le déclarer comme tel avec "**Format/comment out region**".



. Les blocs d'instructions

Les blocs d'instructions sont matérialisés par des indentations (plus de **{** et **}** comme en C !).

Le caractère "**:**" sert à introduire les blocs.



. Les entrées-sorties

L'utilisateur a généralement besoin d'interagir avec le programme. En l'absence d'interface graphique, en mode "console" (Python shell), on doit pouvoir saisir ou entrer des informations, ce qui est fait depuis une lecture au clavier. Inversement, on doit pouvoir afficher ou sortir des informations, ce qui correspond généralement à une écriture sur l'écran (dans la console).

- Les entrées

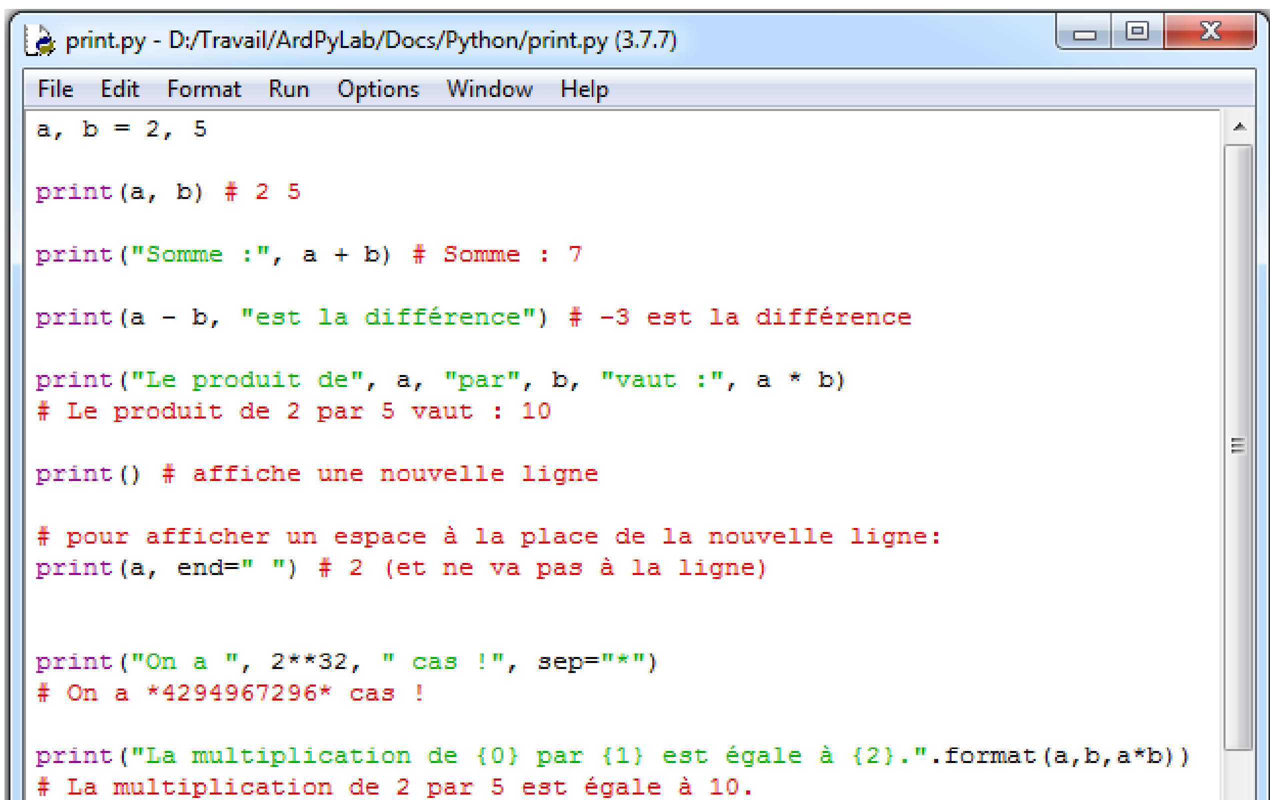
Il s'agit de réaliser une saisie dans la console Python. Pour cela, on utilise la fonction "**input()**" qui interrompt le programme, affiche une éventuelle invite et attend que l'utilisateur entre une donnée et la valide par la touche "**Entrée** " .

La fonction "**input()**" effectue toujours une saisie en mode texte (la saisie est une chaine – type '**str**') dont on peut ensuite changer le type par une conversion.

```
>>> nb = input("veuillez saisir un nombre:")
veuillez saisir un nombre:45
>>> print(type(nb))
<class 'str'>

>>> ch = input("veuillez saisir une chaine de caractères:")
veuillez saisir une chaine de caractères:azert123
>>> print(type(ch))
<class 'str'>
```

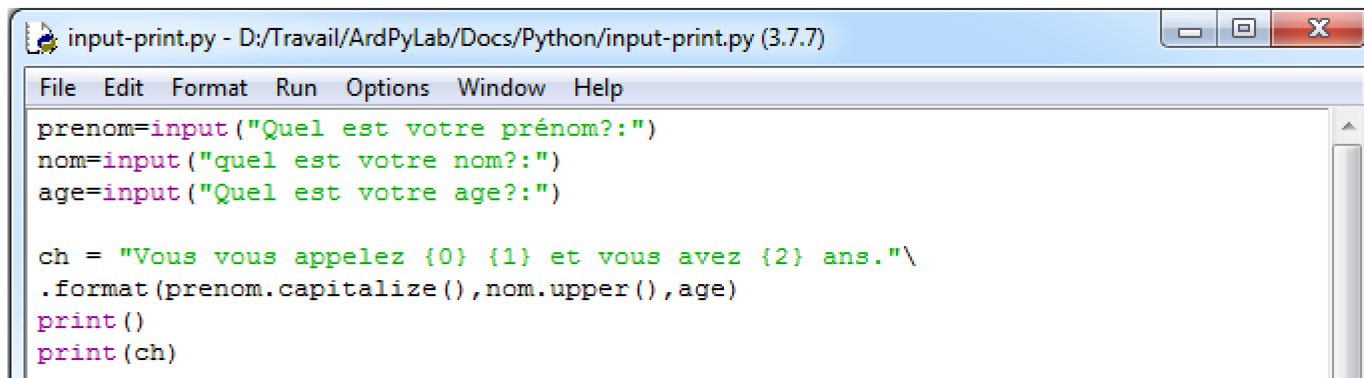
- Les sorties

En mode interactif, Python lit, évalue et affiche, mais la fonction "**print()**" reste indispensable aux affichages dans les scripts :



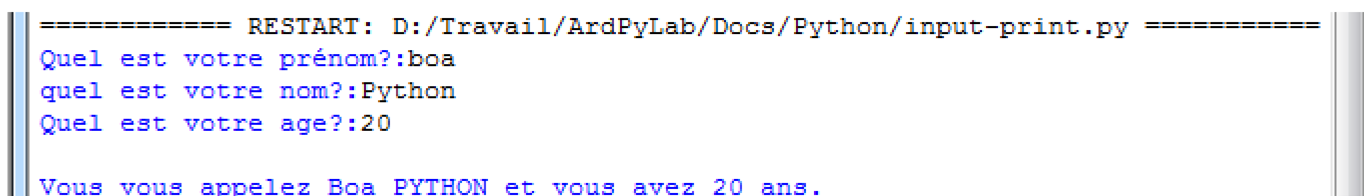
```
print.py - D:/Travail/ArdPyLab/Docs/Python/print.py (3.7.7)
File  Edit  Format  Run  Options  Window  Help
a, b = 2, 5

print(a, b) # 2 5

print("Somme :", a + b) # Somme : 7

print(a - b, "est la différence") # -3 est la différence

print("Le produit de", a, "par", b, "vaut :", a * b)
# Le produit de 2 par 5 vaut : 10

print() # affiche une nouvelle ligne

# pour afficher un espace à la place de la nouvelle ligne:
print(a, end=" ") # 2 (et ne va pas à la ligne)


print("On a ", 2**32, " cas !", sep="*")
# On a *4294967296* cas !

print("La multiplication de {0} par {1} est égale à {2}.".format(a,b,a*b))
# La multiplication de 2 par 5 est égale à 10.
```

Exemple : Le programme ci-dessous demande le nom, le prénom et l'age de l'utilisateur et affiche les données après formatage.




```
prenom=input("Quel est votre prénom?:")
nom=input("quel est votre nom?:")
age=input("Quel est votre age?:")

ch = "Vous vous appelez {0} {1} et vous avez {2} ans."\
.format(prenom.capitalize(),nom.upper(),age)
print()
print(ch)
```

Résultat dans la fenêtre Python shell :



```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/input-print.py ===========
Quel est votre prénom?:boa
quel est votre nom?:Python
Quel est votre age?:20

Vous vous appelez Boa PYTHON et vous avez 20 ans.
```

. Les conversions

Il existe un certain nombre de fonctions permettant de convertir les données d'un type à l'autre.
La fonction "**type()**" permet de récupérer le type de la donnée sous forme d'une chaîne.

Fonction	**Description**	**Exemple**
ord	retourne la valeur ASCII d'un caractère	ord('A')
chr	retourne le caractère à partir de sa valeur ASCII	char(65)
str	convertit en chaîne	str(10), str([10,20])
int	interprète la chaîne en entier	int('45')
int long	interprète la chaîne en entier long	long('56857657695476')
float	interprète la chaîne en flottant	float('23.56')

Autre conversions :

. Conversion binaire

La fonction "**bin()**" permet de convertir un nombre binaire en chaine de caractères :

```
>>> bin(204)
'0b11001100'
```

Pour convertir une chaine de caractères représentant un nombre binaire (base 2), on utilise la fonction **int()** en précisant la base :

```
>>> int('11001100',2)
204
```

Le préfixe 0b n'est pas obligatoire et peut être supprimé.

. **Conversion hexadécimale**

La fonction "**hex()**" permet de convertir un nombre hexadécimal en chaine de caractères et la fonction **int()** en précisant la base 16 fait la conversion inverse:

```
>>> hex(32)
'0x20'
>>> int('0x20',16)
32
```

. Les structures de contrôles

- Condition **if** :

L'instruction **if** ("si" en français), utilisée avec un opérateur logique de comparaison, permet de tester si une condition est vraie.

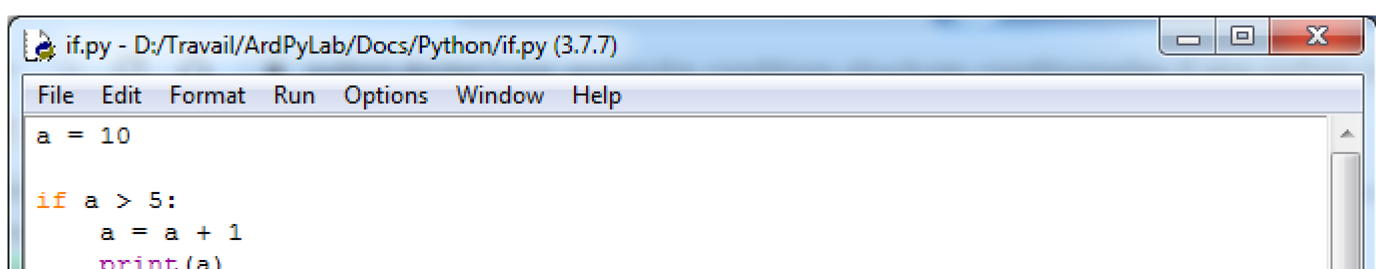
Le format d'un test **if** est le suivant :

```
if  uneVariable > 50 :
    # Instructions  (attention aux tabulations !)
```

Dans cet exemple, le programme va tester si la variable **uneVariable** est supérieure à 50. Si c'est le cas, le programme va réaliser une action particulière. Autrement dit, si l'état du test est vrai, le bloc d'instructions (ligne d'instructions de même indentation) après le caractère "**:**" est exécuté.

Exemple :

Une valeur est donnée à une variable et si cette valeur est supérieure à 5, alors celle-ci est incrémentée de 1 et affichée.



Résultat dans la fenêtre Python Shell :

```
================ RESTART: D:/Travail/ArdPyLab/Docs/Python/if.py ================
11
>>>
```

Rappel des opérateurs logiques de comparaison :

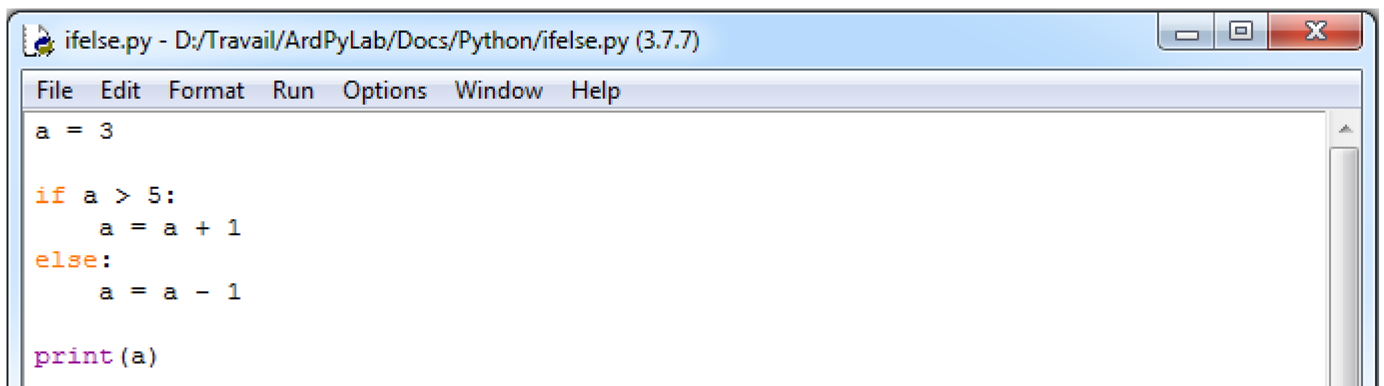
- x == y est vrai quand x est égal à y,
- x != y est vrai quand x est différent de y,
- x > y est vrai quand x est strictement supérieur à y,
- x < y est vrai quand x est strictement inférieur à y,
- x >= y est vrai quand x est supérieur ou égal à y,
- x <= y est vrai quand x est inférieur ou égal à y.

- Condition **if / else** :

L'instruction **if / else** (si/sinon en français) permet un meilleur contrôle du déroulement du programme que la simple instruction if, en permettant de grouper plusieurs tests ensemble.

```
if  var > 10:
    #action A
else:
    #action B
```

Exemple :



Résultat dans la fenêtre Python Shell :

```
===== RESTART: C:\Users\Olivier\Docs\Travail\ArdPyLab\Docs\Python\ifelse.py ====
2
>>>
```

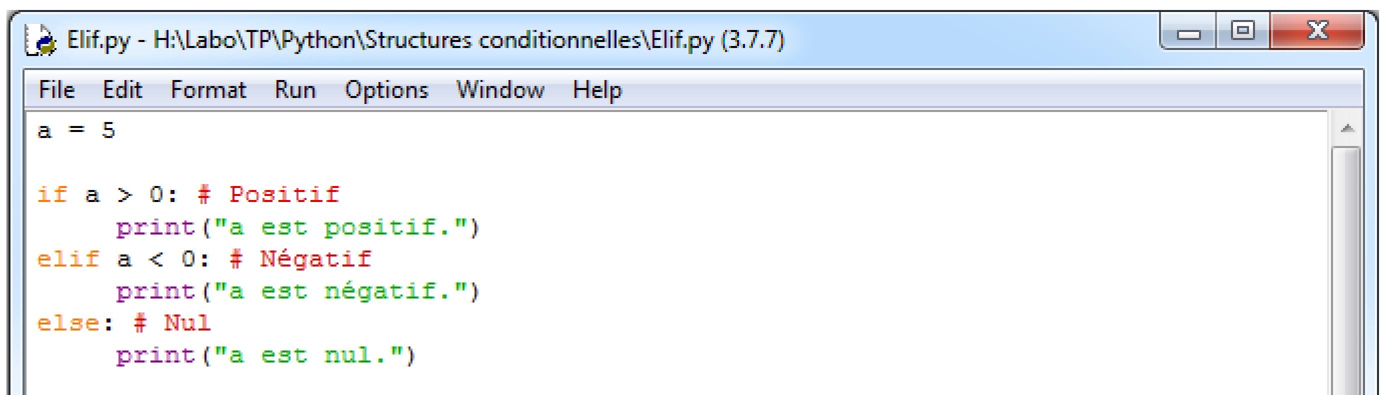
- Condition **elif** :

L’instruction **else** peut contenir un autre test **if**, et donc des tests multiples, mutuellement exclusifs peuvent être réalisés en même temps. On peut alors utiliser le mot clé **elif** à la place de **else : if ....**

Chaque test sera réalisé après le suivant jusqu'à ce qu'un test VRAI soit rencontré.
Quand une condition vraie est rencontrée, les instructions associées sont réalisées, puis le programme continue son exécution à la ligne suivant l'ensemble de la construction **if/elif**.
Si aucun test n'est VRAI, le bloc d'instructions par défaut **else** est exécuté, s’il est présent, déterminant ainsi le comportement par défaut.

Un bloc **elif** peut être utilisé avec ou sans bloc de conclusion **else**.
Un nombre illimité de branches **elif** est autorisé.

Exemple :



- **AND / OR**

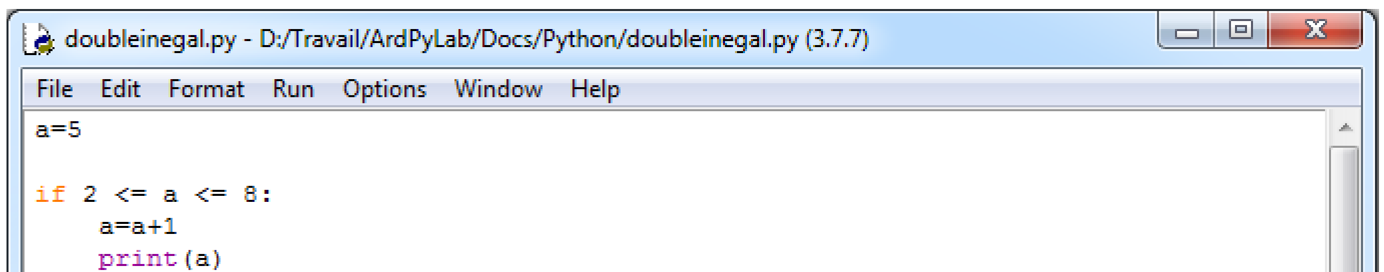
Il est possible d'affiner une condition avec les mots clé **AND** qui signifie " ET " et **OR** qui signifie " OU ".
Ces opérateurs peuvent être utilisés à l'intérieur de la condition d'une instruction if pour associer plusieurs conditions à tester.

```
if var >= 5  and var <= 10 :  # est VRAI seulement si var appartient à l’intervalle [5;10]
    # bloc d’instructions
else:
    # bloc d’instructions
```

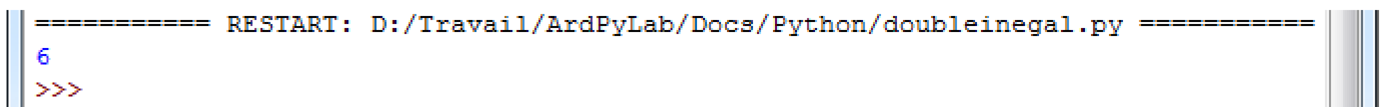
```
if var1 > 0 or var2 > 0 : # est vrai si var1 supérieur à 0 ou si var2 supérieur à 0
    # bloc d'instructions
else:
    # bloc d'instructions
```

Remarque :

Python permet aussi l'enchaînement des comparaisons à l'aide d'une double inégalité.



Résultat dans la fenêtre Python Shell :



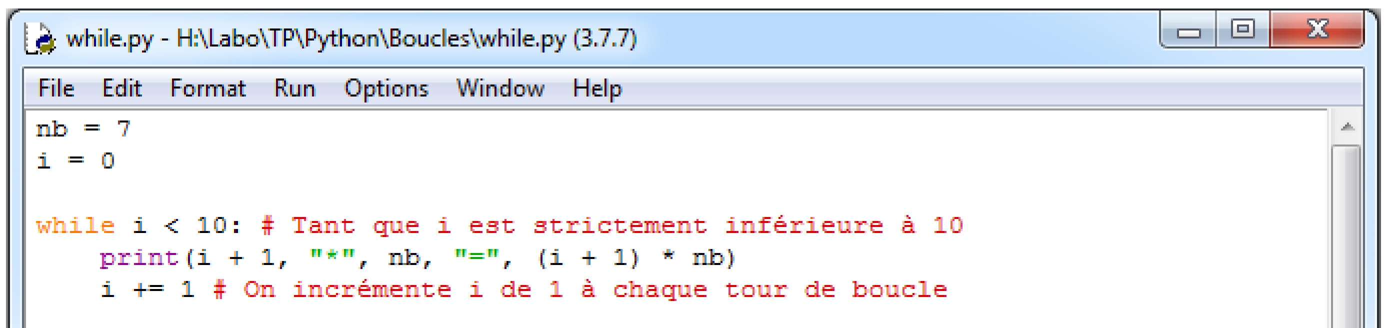
- les structures itératives :

. boucles **While**

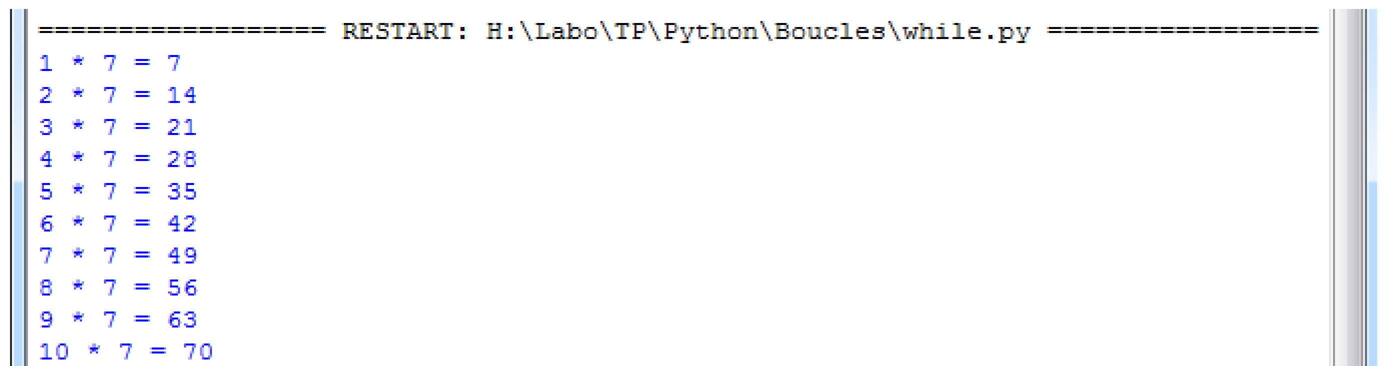
Les boucles while ("tant que" en anglais) bouclent sans fin, et indéfiniment, jusqu'à ce que la condition ou l'expression testée devienne fausse.
Quelque chose doit modifier la variable testée, sinon la boucle while ne se terminera jamais. C'est généralement une variable incrémentée dans le bloc d'instructions de la boucle.

```
var = 0
while var < 10 :  # tant que la variable est inférieur à 10
    # fait quelque chose 10 fois de suite...
    var += 1 # incrémente la variable
```

Exemple : Affichage d'une table de multiplication



Résultat dans la fenêtre Python Shell :



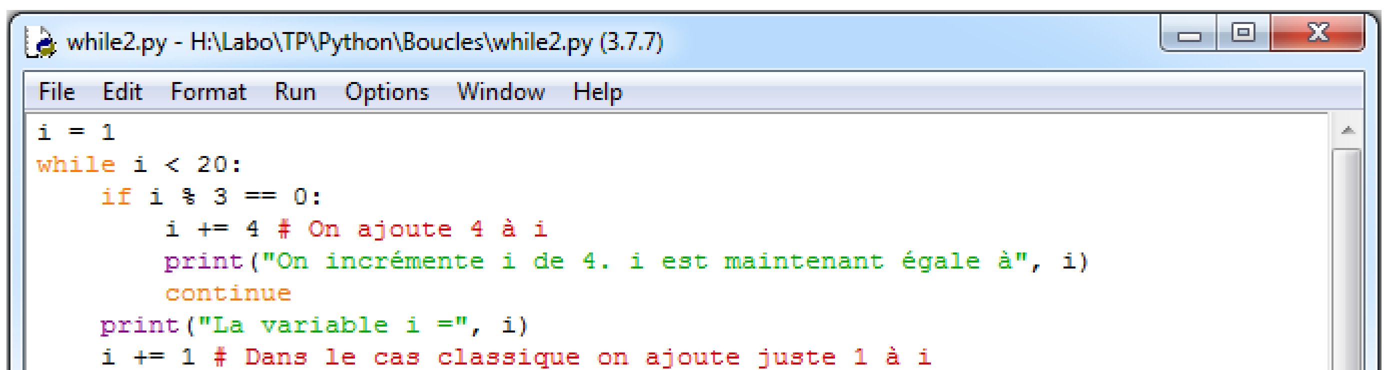
Remarques :

Il existe des mots clés qui permettent d'effectuer une rupture dans la boucle itérative :

. **continue** (continue directement à la prochaine itération de la boucle, la boucle est court-circuitée)

Exemple :

Dans l'exemple suivant, tant que la variable i est inférieure à 20, celle-ci est incrémentée de 1 jusqu'à ce qu'elle soit égale à un multiple de 3. Elle est alors incrémentée de 4 et la boucle normale est reprise à l'aide de l'instruction **continue**.



Résultat dans la fenêtre Python Shell :

```
================= RESTART: H:\Labo\TP\Python\Boucles\while2.py =================
La variable i = 1
La variable i = 2
On incrémente i de 4. i est maintenant égale à 7
La variable i = 7
La variable i = 8
On incrémente i de 4. i est maintenant égale à 13
La variable i = 13
La variable i = 14
On incrémente i de 4. i est maintenant égale à 19
La variable i = 19
```

. **break** (sort de la boucle en cours)

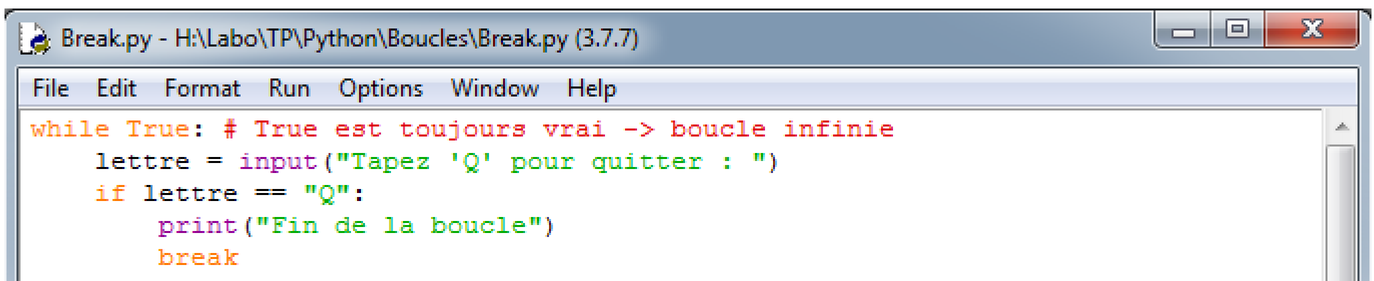
. **pass** (instruction vide – ne fait rien)

Il est parfois pratique d'utiliser une boucle **while** infinie (dont la condition est toujours vraie), et d'utiliser les ruptures de séquences.

```
while True:
    # bloc d'instructions
    if condition : break
```

Exemple :

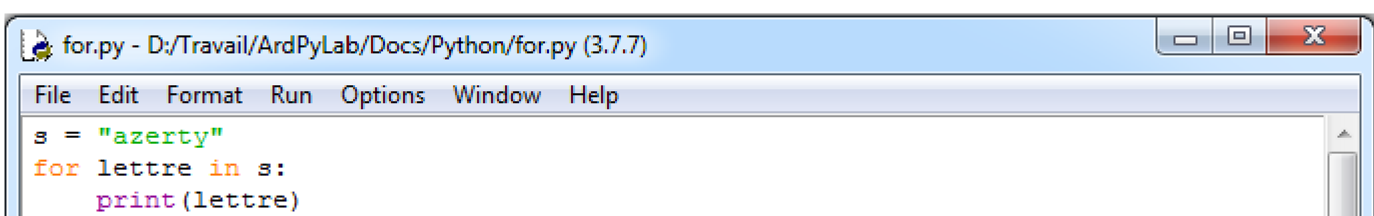
Dans cet exemple, on demande à l'utilisateur de saisir la lettre 'Q' pour sortir de la boucle.



. boucles **for**

L'utilisation principale de l'instruction **for** est de parcourir un itérable, c'est-à-dire un conteneur que l'on peut parcourir élément par élément, dans l'ordre ou non, suivant son type :

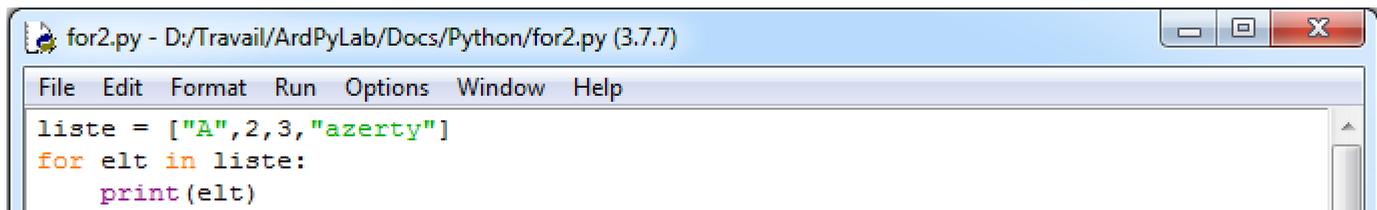
- Parcours d'une chaine de caractères :



Résultat dans la fenêtre Python Shell :

```
================ RESTART: D:/Travail/ArdPyLab/Docs/Python/for.py ===============
a
z
e
r
t
y
```

- Parcours d'une liste :



Résultat dans la fenêtre Python Shell :

```
=============== RESTART: D:/Travail/ArdPyLab/Docs/Python/for2.py ===============
A
2
3
azerty
```

L'instruction **for** peut également être est utilisée pour répéter l'exécution d'un bloc d'instructions à l'aide de la fonction **range()** déjà vu lors de la présentation des listes :

```
for i in range(10):
    # bloc d'instructions
```

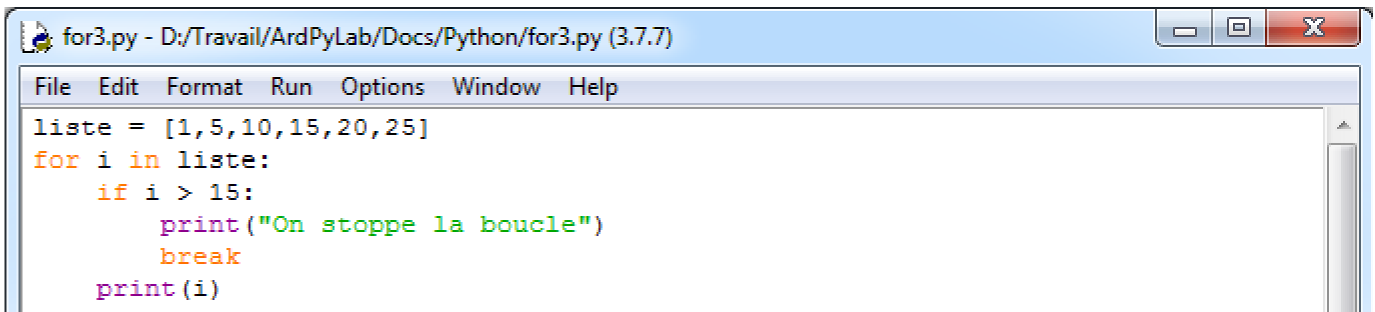
Dans cet exemple, le bloc d'instructions sera exécuté 10 fois.

Remarques:

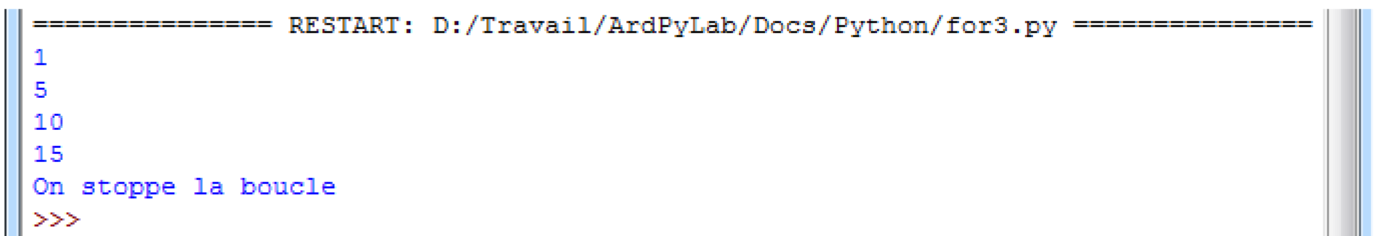
- Une boucle **for** peut également être arrêtée avec l'instruction **break**

Exemple :

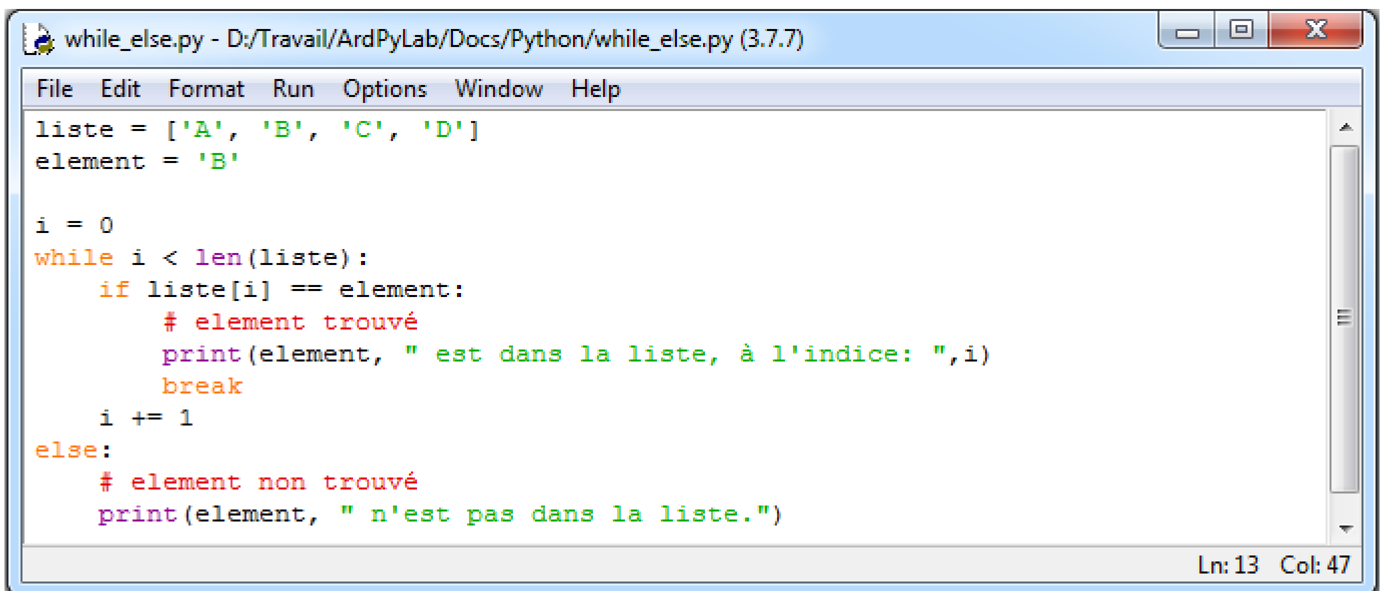
Dans cet exemple, la boucle **for** parcours les éléments d'une liste de nombres. Si le nombre lu est supérieur à 15, la boucle est stoppée.



Résultat dans la fenêtre Python Shell :

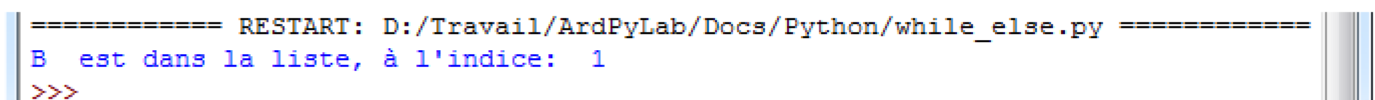


- Les boucles **while** et **for** peuvent posséder une clause **else** qui ne s'exécute que si la boucle se termine normalement, c'est-à-dire sans interruption avec l'instruction **break** :



Dans l'exemple ci-dessus, on parcourt une liste à l'aide d'une boucle **while** pour savoir si la variable **element** est dans la liste. Si la variable est trouvée, la boucle est stoppée avec une instruction break :

Résultat dans la fenêtre Python Shell :

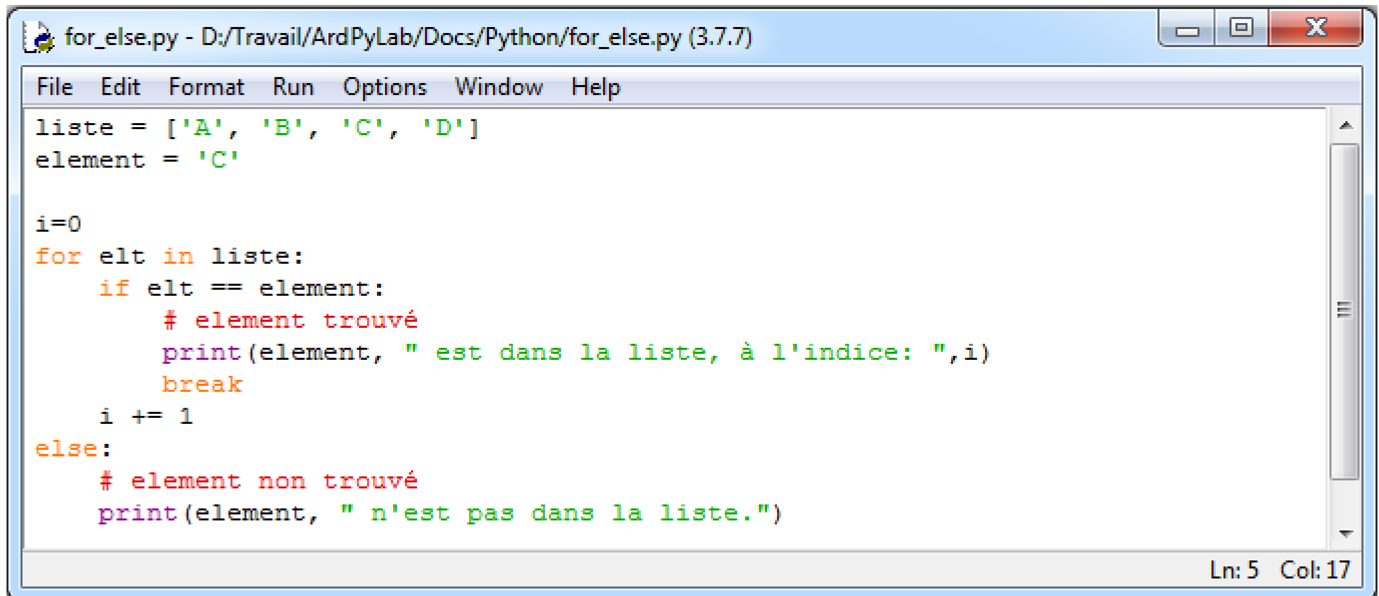


, sinon on affiche que la variable n'est pas dans la liste :

<u>Résultat dans la fenêtre Python Shell avec modification de la variable **element = ‘E’**</u> :

```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/while_else.py ============
E  n'est pas dans la liste.
>>>
```

Il en est de même avec une boucle **for** :



<u>Résultats dans la fenêtre Python Shell</u> :

```
============= RESTART: D:/Travail/ArdPyLab/Docs/Python/for_else.py =============
C  est dans la liste, à l'indice:  2
>>>
```

avec modification de la variable **element = ‘E’** :

```
============= RESTART: D:/Travail/ArdPyLab/Docs/Python/for_else.py =============
E  n'est pas dans la liste.
>>>
```

. Les exceptions – Gestion des erreurs dans les scripts

Lorsqu'une instruction d'un script ne se déroule pas correctement (par exemple, une division par zéro), une **exception est levée** ce qui interrompt le contexte d'exécution, pour revenir à un environnement d'exécution supérieur, jusqu'à celui gérant cette exception.

Par défaut, l'environnement supérieur est le shell de commande depuis lequel l'interpréteur Python a été lancé, et le comportement de gestion par défaut est d'afficher l'exception :

```
============= RESTART: D:/Travail/ArdPyLab/Docs/Python/exception.py ============
Traceback (most recent call last):
  File "D:/Travail/ArdPyLab/Docs/Python/exception.py", line 2, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

Pour gérer l'exception, et éviter la fin du programme, il faut utiliser la structure **try** et **except** :

```
try:

        # bloc d'instructions susceptibles d'échouer

except:

        # bloc d'instructions à faire en cas d'échec
```

Toutes les exceptions levées par Python sont des instances de sous-classe de la classe **Exception**.

La hiérarchie des sous-classes offre plusieurs exceptions standard, comme **ValueError** (exception levée quand on tente par exemple de convertir en nombre une chaine de caractères ne représentant pas un nombre) ou **ZeroDivisionError** (quand on tente de diviser un nombre par zéro).

Il est possible de compléter la structure **try except** avec un bloc **else** et un bloc **finally**. Les instructions du bloc **else** ne sont exécutées qu'en l'absence d'erreur et les instructions du bloc **finally** sont toujours effectuées quel que soient les erreurs rencontrées lors de l'exécution du bloc **try** ou en l'absence d'erreur.

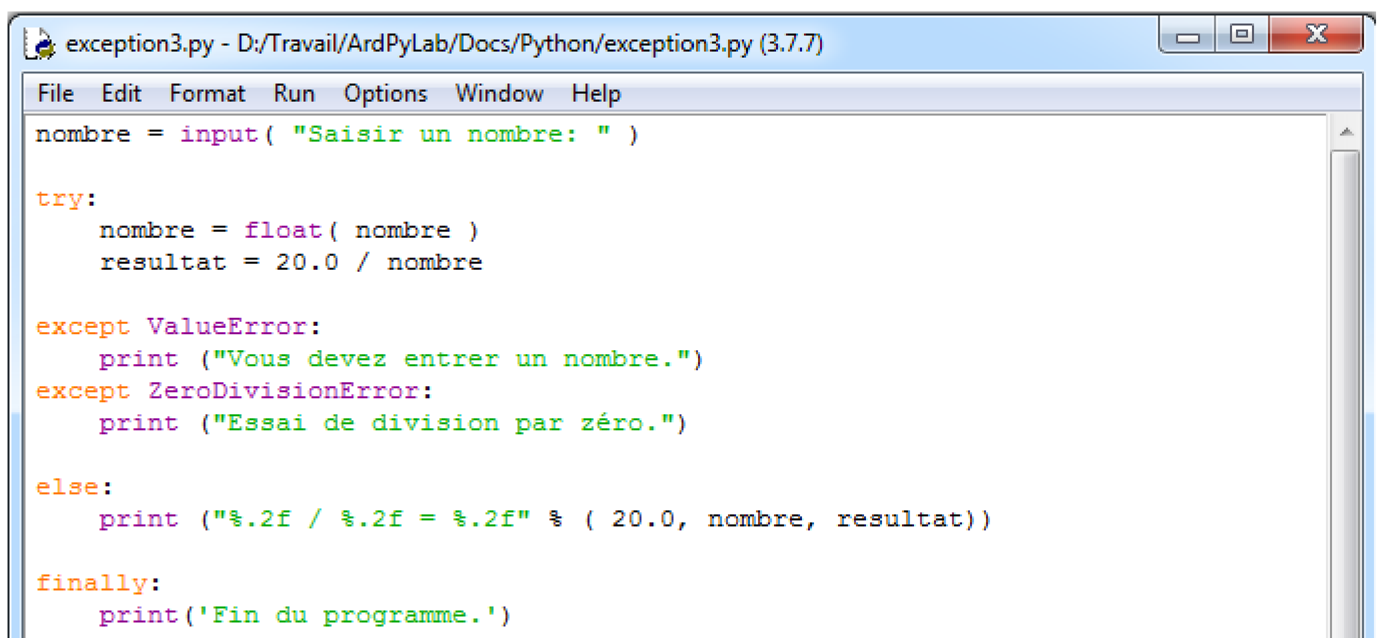
La syntaxe complète d'une exception est alors :

```
try:

        ... # séquence normale d'exécution

except exception_1:
```

```
        ... # traitement de l'exception 1
except exception_2:
        ... # traitement de l'exception 2
else:
        ... # bloc d’instructions exécutées en l'absence d'erreur
finally:
        ... # bloc d’instructions toujours exécutées
```

<u>Exemple</u> :

Ce programme demande à l’utilisateur de saisir un nombre et tente de le convertir en flottant et de faire une division avec ce nombre :

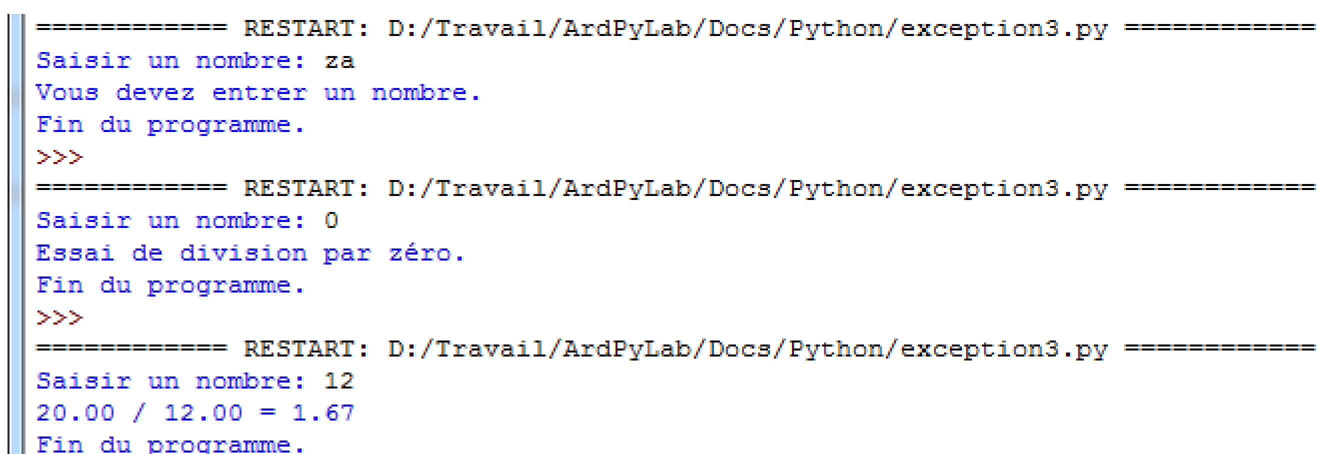


exception3.py - D:/Travail/ArdPyLab/Docs/Python/exception3.py (3.7.7)

File Edit Format Run Options Window Help

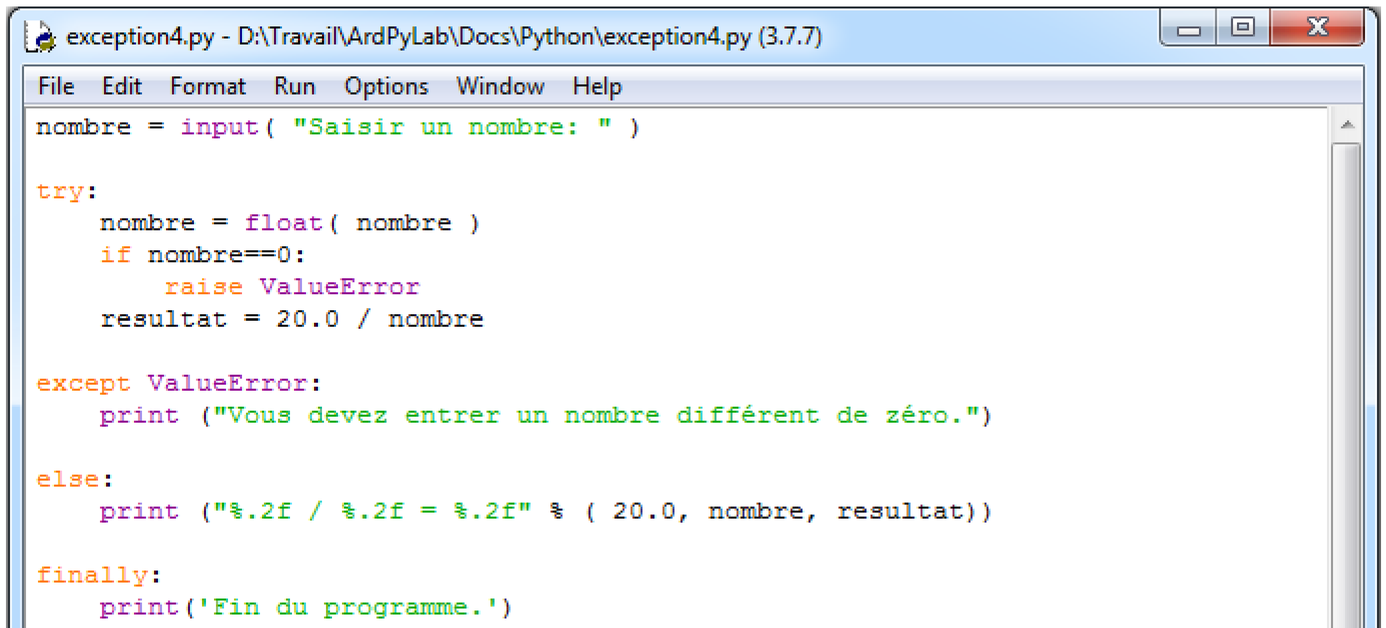
```python
nombre = input( "Saisir un nombre: " )

try:
    nombre = float( nombre )
    resultat = 20.0 / nombre

except ValueError:
    print ("Vous devez entrer un nombre.")
except ZeroDivisionError:
    print ("Essai de division par zéro.")

else:
    print ("%.2f / %.2f = %.2f" % ( 20.0, nombre, resultat))

finally:
    print('Fin du programme.')
```

<u>Résultats dans la fenêtre Python Shell</u> :



```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/exception3.py ============
Saisir un nombre: za
Vous devez entrer un nombre.
Fin du programme.
>>>
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/exception3.py ============
Saisir un nombre: 0
Essai de division par zéro.
Fin du programme.
>>>
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/exception3.py ============
Saisir un nombre: 12
20.00 / 12.00 = 1.67
Fin du programme.
```

L'instruction **raise** permet de lever volontairement une exception. Ainsi le script du programme précédent devient :

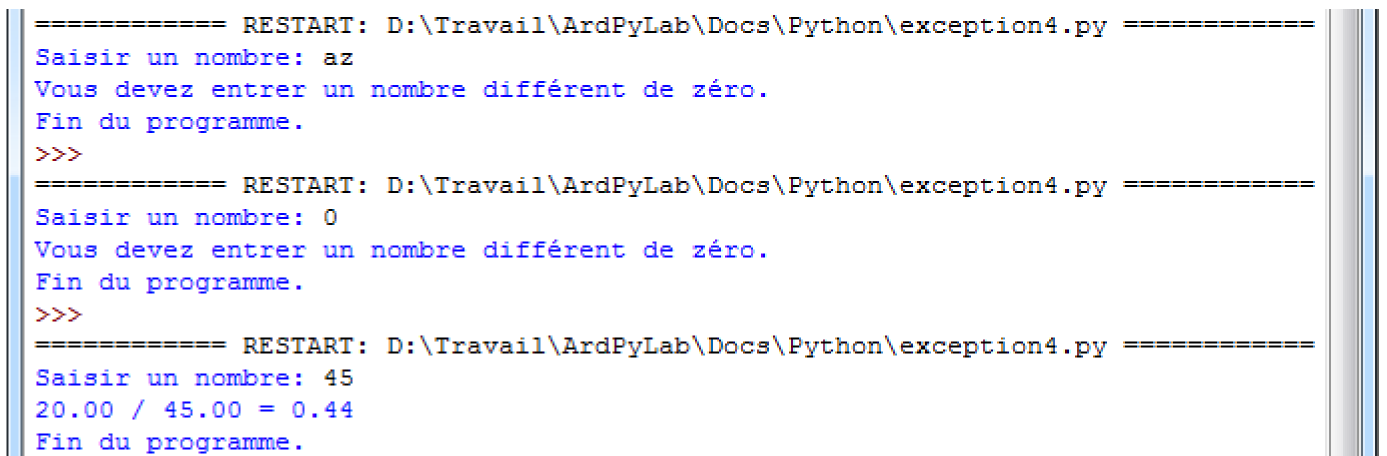


exception4.py - D:\Travail\ArdPyLab\Docs\Python\exception4.py (3.7.7)

File Edit Format Run Options Window Help

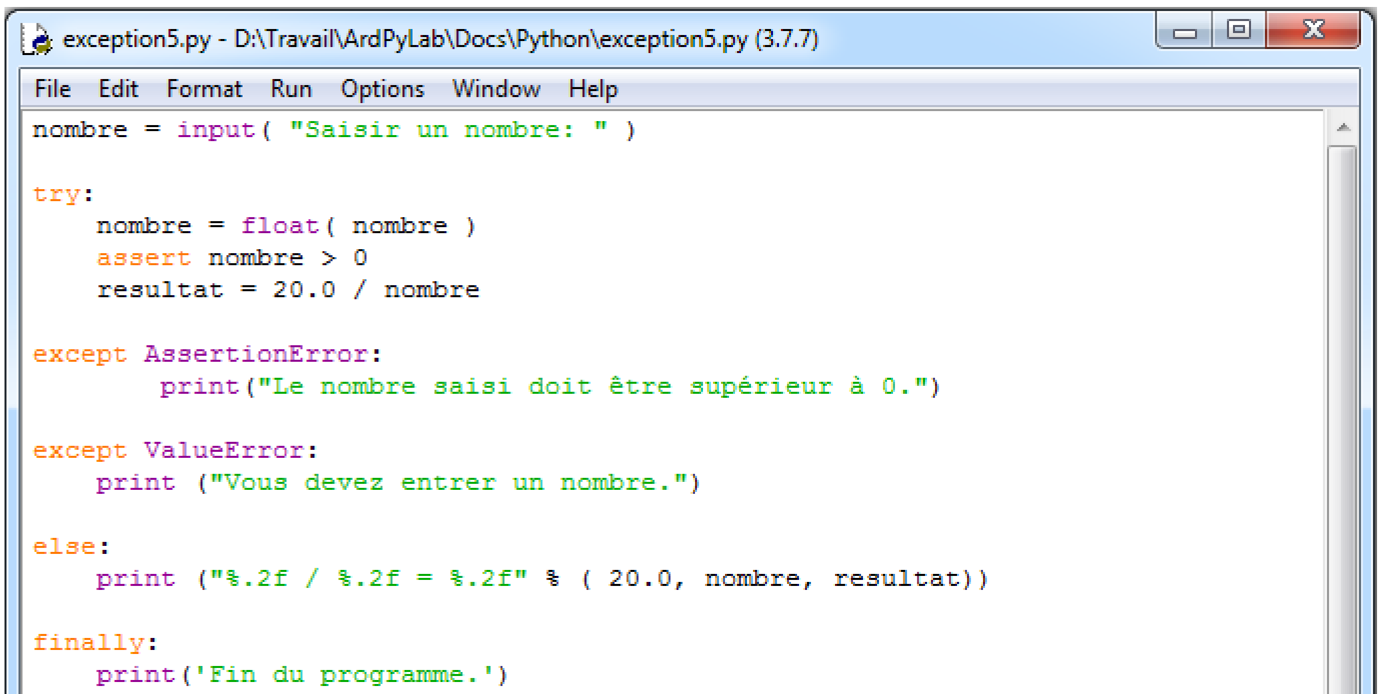
```
nombre = input( "Saisir un nombre: " )

try:
    nombre = float( nombre )
    if nombre==0:
        raise ValueError
    resultat = 20.0 / nombre

except ValueError:
    print ("Vous devez entrer un nombre différent de zéro.")

else:
    print ("%.2f / %.2f = %.2f" % ( 20.0, nombre, resultat))

finally:
    print('Fin du programme.')
```

Après la conversion de la chaine saisie au clavier en nombre, un test est effectué sur le nombre, si celui-ci est égale à 0, l'exception **ValueError** est levée à l'aide de l'instruction **raise**. Et bien-sûr si la chaine ne peut pas être convertie l'exception **ValueError** est également levée (c'est aussi le bloc d'exception du bloc **try**) :



```
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception4.py ============
Saisir un nombre: az
Vous devez entrer un nombre différent de zéro.
Fin du programme.
>>>
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception4.py ============
Saisir un nombre: 0
Vous devez entrer un nombre différent de zéro.
Fin du programme.
>>>
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception4.py ============
Saisir un nombre: 45
20.00 / 45.00 = 0.44
Fin du programme.
```

Il est également possible de lever une exception avec l'instruction **assert**. Cette instruction va tester la condition mise juste après, et si elle est fausse, va lever une exception de type **AssertionError,** ce qui donne pour notre exemple :

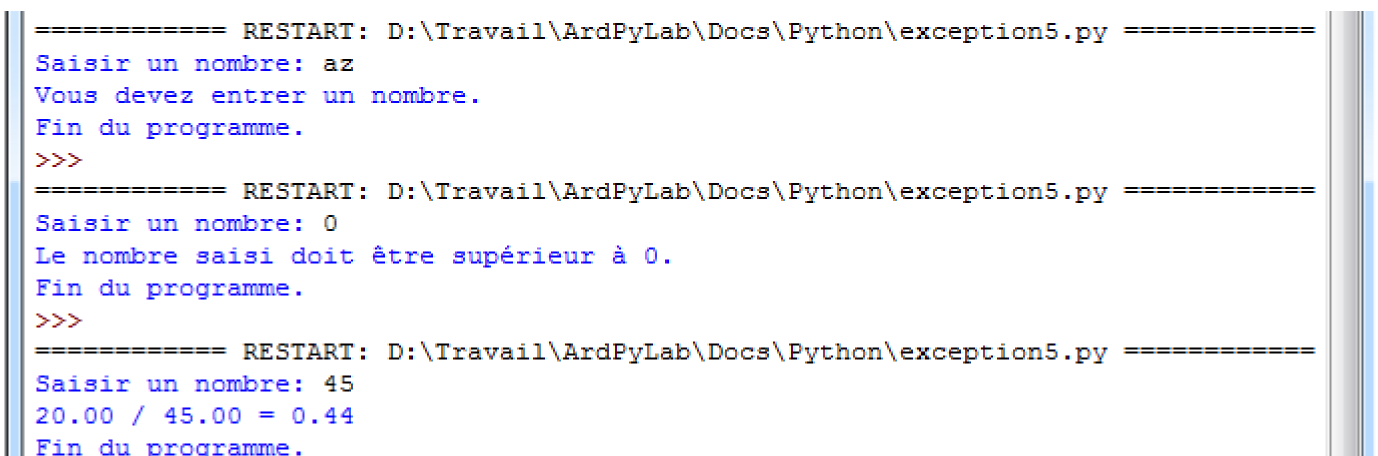


exception5.py - D:\Travail\ArdPyLab\Docs\Python\exception5.py (3.7.7)

File Edit Format Run Options Window Help

```
nombre = input( "Saisir un nombre: " )

try:
    nombre = float( nombre )
    assert nombre > 0
    resultat = 20.0 / nombre

except AssertionError:
        print("Le nombre saisi doit être supérieur à 0.")

except ValueError:
    print ("Vous devez entrer un nombre.")

else:
    print ("%.2f / %.2f = %.2f" % ( 20.0, nombre, resultat))

finally:
    print('Fin du programme.')
```

<u>Résultats dans la fenêtre Python Shell</u> :



```
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception5.py ============
Saisir un nombre: az
Vous devez entrer un nombre.
Fin du programme.
>>>
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception5.py ============
Saisir un nombre: 0
Le nombre saisi doit être supérieur à 0.
Fin du programme.
>>>
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception5.py ============
Saisir un nombre: 45
20.00 / 45.00 = 0.44
Fin du programme.
```

La structure **Try … Except** est donc très utile pour tout ce qui est vérification des saisies au clavier. Mais contrairement aux scripts précédents, il est préférable que le programme ne s'arrête pas si la saisie au clavier ne satisfait pas au programme.

On utilisera pour cela une boucle **while**, afin de redemander à l'utilisateur une saisie au clavier si la précédente n'est pas adéquate, comme dans l'exemple suivant :

Dans ce programme, on demande à l'utilisateur de saisir un nombre supérieur à 0 et on vérifie si c'est un nombre premier.

exception2.py - D:\Travail\ArdPyLab\Docs\Python\exception2.py (3.7.7)

File Edit Format Run Options Window Help

```python
nombresaisi = False

while nombresaisi == False:

    nombre = input("Saisissez un nombre : ")

    try:
        nombresaisi = True
        nombre = int(nombre)
        assert nombre > 0

        i = 2
        while i < nombre and nombre % i != 0:
            i = i + 1

        if i == nombre:
            print("Le nombre", nombre, "est premier.")
        else:
            print("Ce n'est pas un nombre premier.")

        break

    except AssertionError:
        print("Le nombre saisi est inférieur ou égal à 0.")

    except:
        print("vous n'avez pas saisi un nombre!")

    finally:
        nombresaisi = False
```

Résultats dans la fenêtre Python shell :

```
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception2.py ============
Saisissez un nombre : az
vous n'avez pas saisi un nombre!
Saisissez un nombre : 0
Le nombre saisi est inférieur ou égal à 0.
Saisissez un nombre : 15
Ce n'est pas un nombre premier.
>>>
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception2.py ============
Saisissez un nombre : 13
Le nombre 13 est premier.
>>>
```

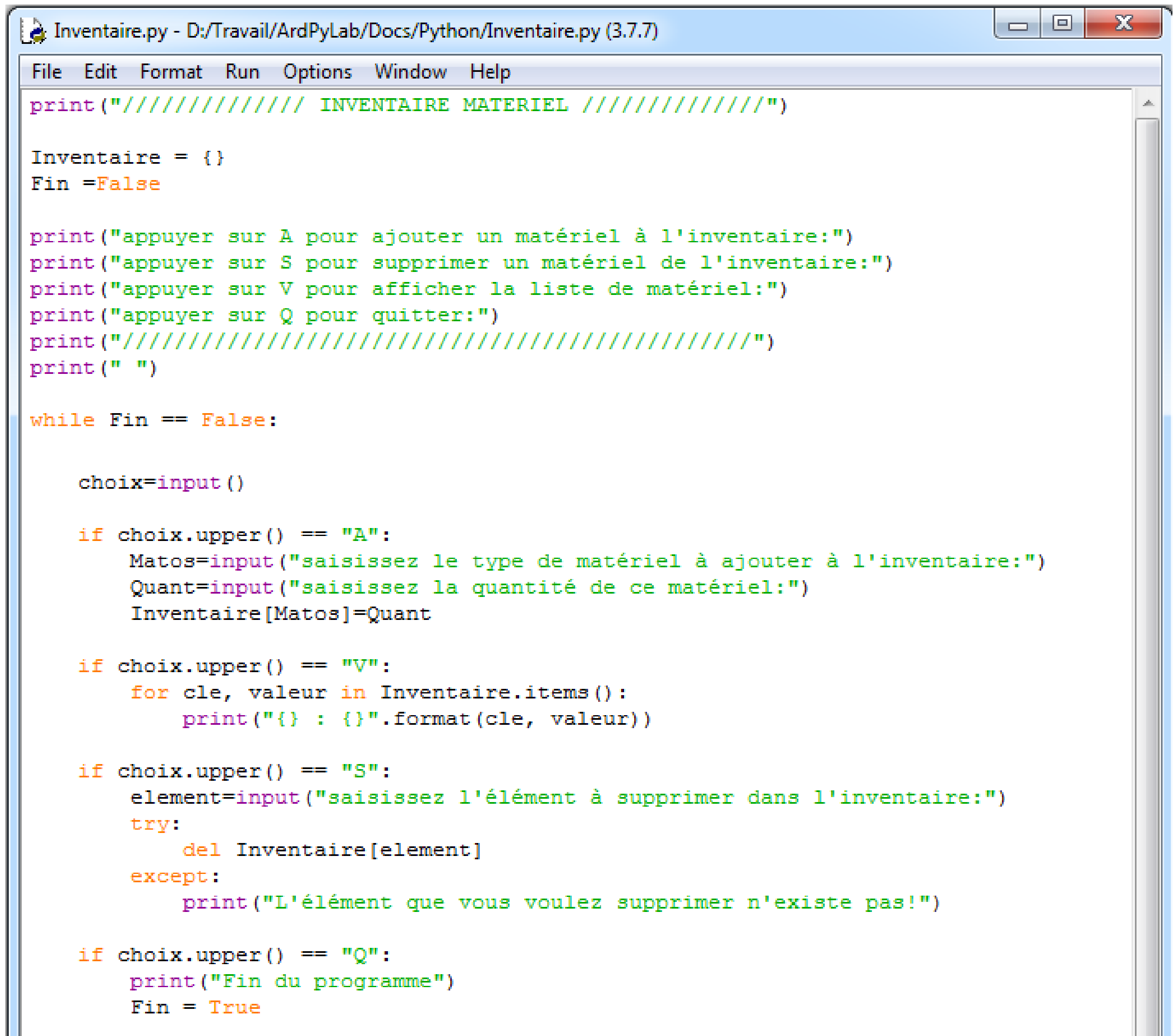
Remarque :

L'instruction **break** du bloc **try** permet, en l'absence d'erreur, de sortir de la boucle **while** et de finir le programme.

. Synthèse structure des scripts Python

Les affectations de variables, les structures conditionnelles et itératives, les gestions d'erreurs sont la base des scripts Python.

Voici un script qui résume tout ce qui a été vu jusqu'à présent. Dans ce programme, l'utilisateur va créer un inventaire de matériel visualisable et modifiable.




```python
print("////////////// INVENTAIRE MATERIEL //////////////")

Inventaire = {}
Fin =False

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(" ")

while Fin == False:

    choix=input()

    if choix.upper() == "A":
        Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
        Quant=input("saisissez la quantité de ce matériel:")
        Inventaire[Matos]=Quant

    if choix.upper() == "V":
        for cle, valeur in Inventaire.items():
            print("{} : {}".format(cle, valeur))

    if choix.upper() == "S":
        element=input("saisissez l'élément à supprimer dans l'inventaire:")
        try:
            del Inventaire[element]
        except:
            print("L'élément que vous voulez supprimer n'existe pas!")

    if choix.upper() == "Q":
        print("Fin du programme")
        Fin = True
```

Résultats dans la fenêtre Python Shell :

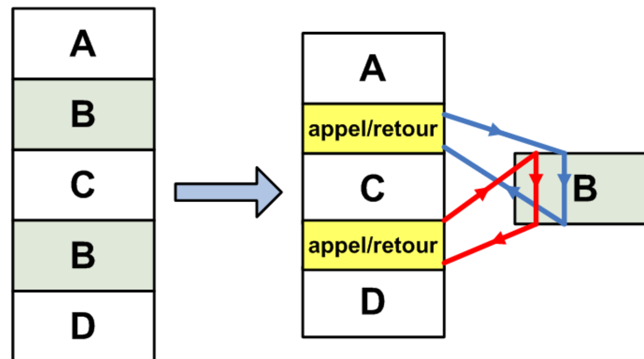
```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/Inventaire.py ============
////////////// INVENTAIRE MATERIEL //////////////
appuyer sur A pour ajouter un matériel à l'inventaire:
appuyer sur S pour supprimer un matériel de l'inventaire:
appuyer sur V pour afficher la liste de matériel:
appuyer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

A
saisissez le type de matériel à ajouter à l'inventaire:Bécher
saisissez la quantité de ce matériel:20
a
saisissez le type de matériel à ajouter à l'inventaire:Eprouvette
saisissez la quantité de ce matériel:15
v
Bécher : 20
Eprouvette : 15
s
saisissez l'élément à supprimer dans l'inventaire:Bécher
v
Eprouvette : 15
a
saisissez le type de matériel à ajouter à l'inventaire:Erlenmeyer
saisissez la quantité de ce matériel:10
v
Eprouvette : 15
Erlenmeyer : 10
s
saisissez l'élément à supprimer dans l'inventaire:bécher
L'élément que vous voulez supprimer n'existe pas!
q
Fin du programme
```

2.2 Les fonctions

Une fonction est un bloc d'instructions que l'on peut appeler à tout endroit d'un programme. Elle est particulièrement utile quand une tâche doit être réalisée plusieurs fois par un programme avec seulement des paramètres différents.

Nous avons déjà vu diverses fonctions prédéfinies : **print()**, **input()**, **range()**, **len()**…

Mais, on peut également créer ses propres fonctions afin d'éviter les répétitions de code et permettre une réutilisation :



La définition d'une fonction se fait à l'aide du mot clé **def** :

```
def ma_fonction():
    # bloc d'instructions
```

Il est possible de définir des paramètres à la fonction. Ce sont les arguments de la fonction :

def maFonction(x,y,z)

Lors de l'appel de la fonction dans le programme principal, chaque paramètre de l'appel correspond dans l'ordre à chaque argument de la définition de la fonction. La correspondance se fait par affectation :

définition

```
def maFonction(x, y, z):
```

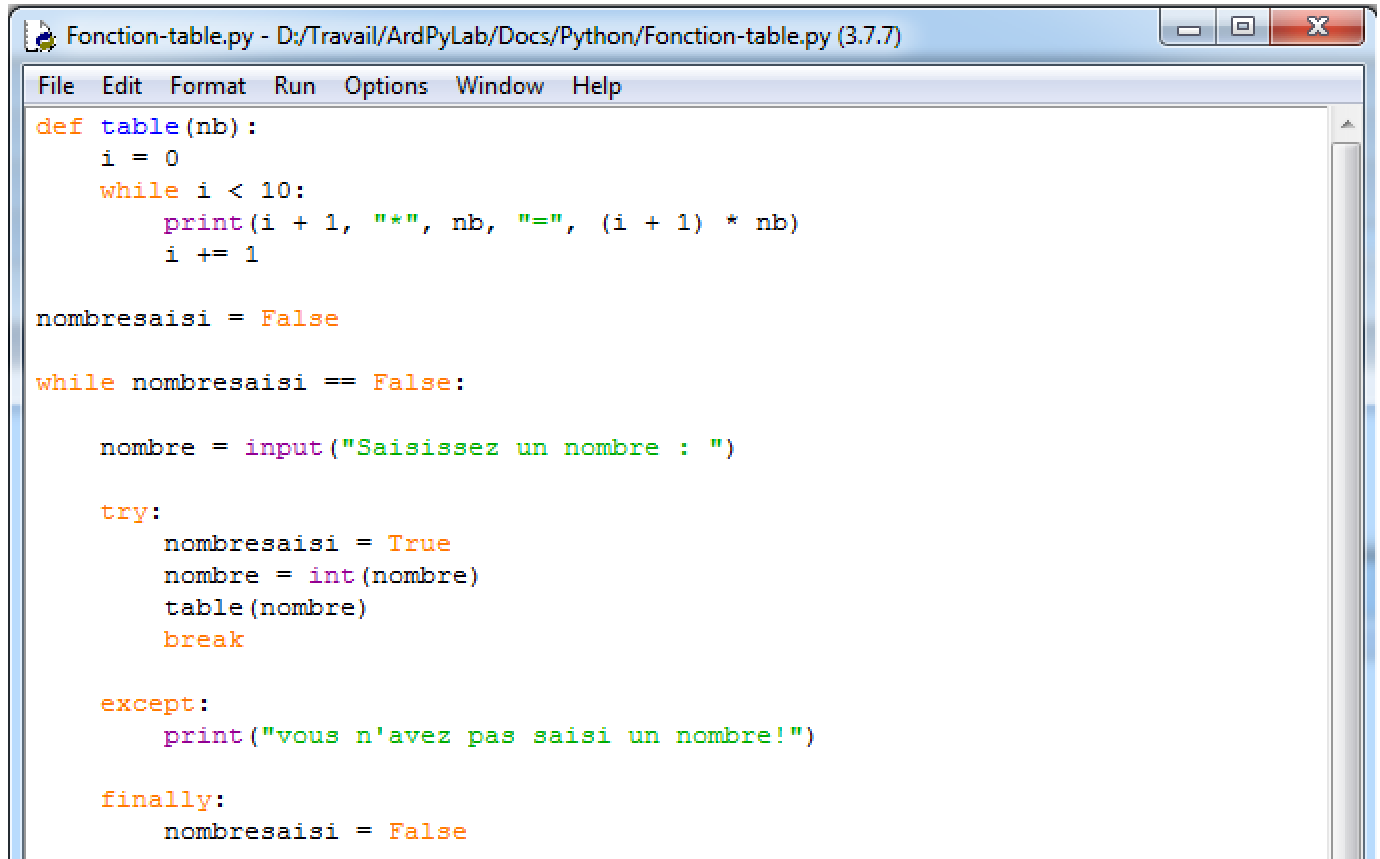
x = 7
y = 'k'
z = 2.718

appel

```
maFonction(7, 'k', 2.718):
```

Exemple :

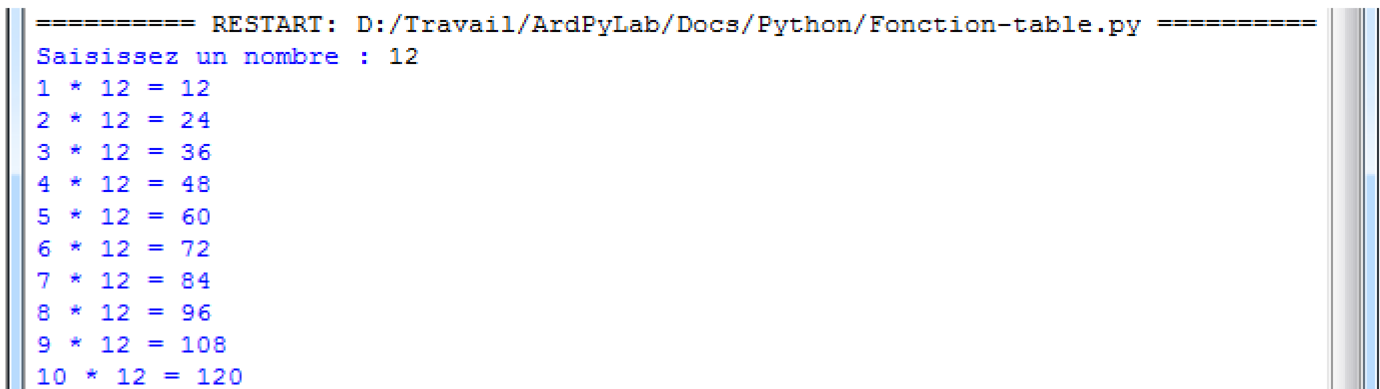
Dans le programme suivant, La fonction **table** permet d'afficher la table de multiplication d'un nombre. L'argument de la fonction **table** étant ce nombre :



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

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

nombresaisi = False

while nombresaisi == False:

    nombre = input("Saisissez un nombre : ")

    try:
        nombresaisi = True
        nombre = int(nombre)
        table(nombre)
        break

    except:
        print("vous n'avez pas saisi un nombre!")

    finally:
        nombresaisi = False
```

Résultats dans la fenêtre Python Shell :



```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/Fonction-table.py ==========
Saisissez un nombre : 12
1 * 12 = 12
2 * 12 = 24
3 * 12 = 36
4 * 12 = 48
5 * 12 = 60
6 * 12 = 72
7 * 12 = 84
8 * 12 = 96
9 * 12 = 108
10 * 12 = 120
```

Les paramètres de la fonction peuvent être nommés et recevoir des valeurs par défaut. Ils peuvent ainsi être donnés dans le désordre et/ou pas en totalité.

Exemple :

Ajoutons à la fonction **table**, l'argument **max**=10 correspondant à la valeur maximale du multiplicateur et donnons une valeur par défaut au nombre à multiplier :

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

L'appel de la fonction pourra se faire ainsi :

. **table(nombre)** : la valeur par défaut de max est utilisée

. **table(nombre1, nombre2)** avec **nombre1** le nombre à multiplier et **nombre2** la valeur maximale du multiplicateur

. **table(nombre1, max=nombre2)**

. **table(nb=nombre1, max=nombre2)**

. **table(max=nombre2, nb=nombre1)**

. table() : les valeurs par défaut de nb et max sont utilisées

. Fonctions avec return

Les fonctions peuvent retourner une ou plusieurs données à l'aide du mot clé **return.** A noter qu'une fonction sans **return**, est plutôt appelée procédure.

Exemple :

De façon à pouvoir utiliser la nouvelle fonction **table**, nous allons modifier le programme d'affichage des tables de multiplication en créant une fonction permettant de saisir un nombre et qui retourne ce 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
```

Le seul argument de la fonction est le message à afficher lors de la demande de saisie du nombre à l'aide de la fonction **input()**.

Si l'entrée clavier ne correspond pas à un nombre, l'utilisateur en est informé et la demande de saisi d'un nombre est affichée de nouveau.

Si l'entrée clavier est valide, le nombre saisi est retourné par la fonction.

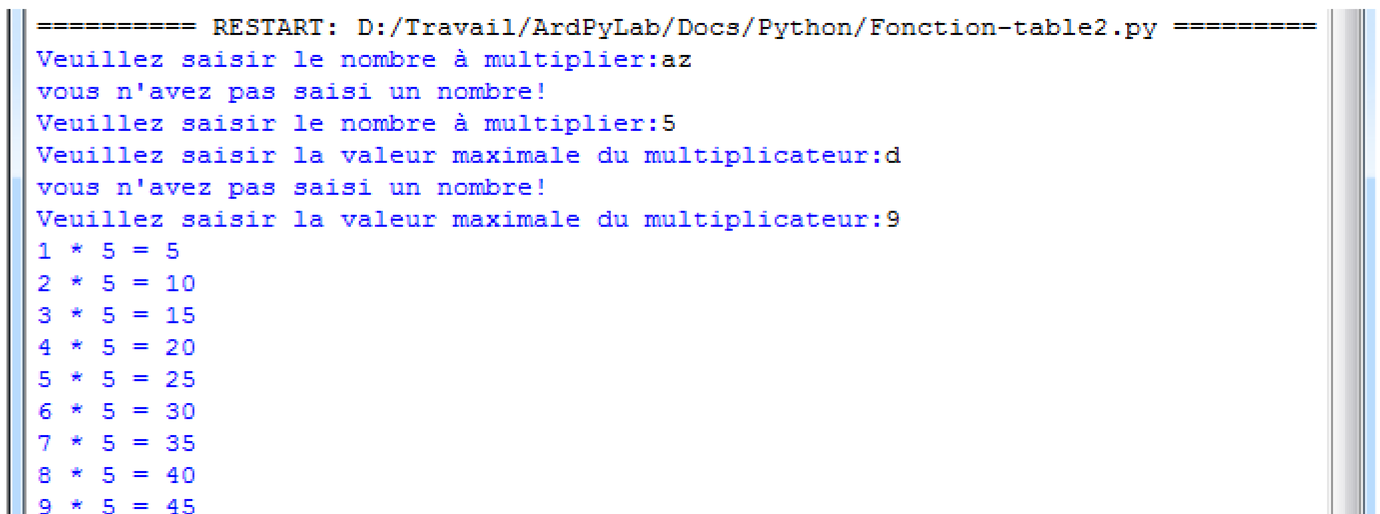
Le programme d’affichage de la table de multiplication souhaitée est alors :

Fonction-table2.py - D:/Travail/ArdPyLab/Docs/Python/Fonction-table2.py (3.7.7)

File Edit Format Run Options Window Help

```python
#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

# Appels des fonctions

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

table(nombre,multimax)
```

Résultats dans la fenêtre Python Shell :



```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/Fonction-table2.py =========
Veuillez saisir le nombre à multiplier:az
vous n'avez pas saisi un nombre!
Veuillez saisir le nombre à multiplier:5
Veuillez saisir la valeur maximale du multiplicateur:d
vous n'avez pas saisi un nombre!
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
```

Remarque :

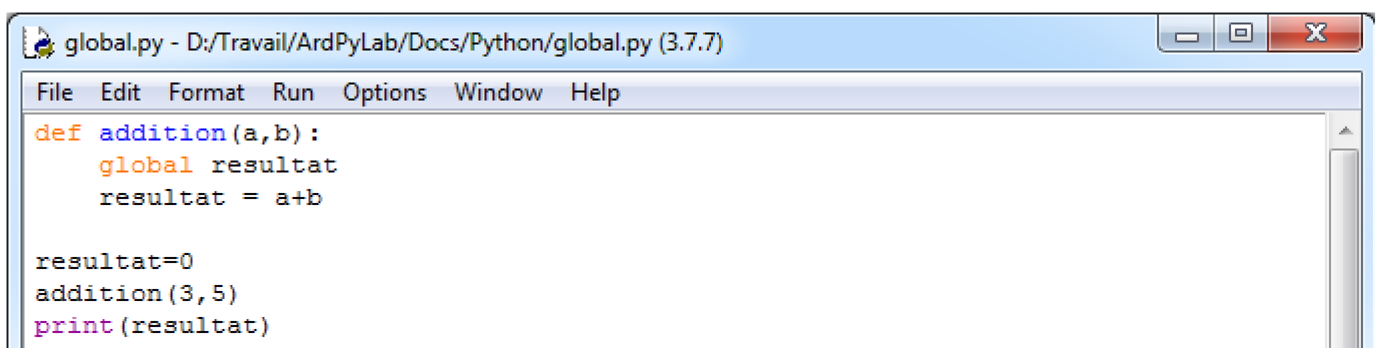
Les variables à l'intérieur du corps d'une fonction ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales**.

Une variable locale peut avoir le même nom qu'une variable du programme principal mais elle reste néanmoins indépendante.

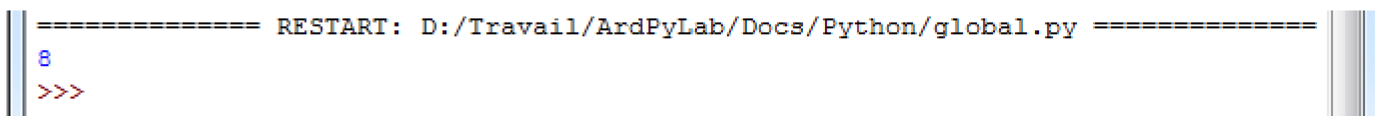
Le contenu des variables locales est inaccessible depuis l'extérieur de la fonction.

Les variables définies à l'extérieur d'une fonction sont des **variables globales**. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

Pour modifier une variable globale au sein d'une fonction, il faut utiliser l'instruction **global**. Cette instruction permet d'indiquer quelles sont les variables à traiter globalement :



Résultats dans la fenêtre Python Shell :



. Fonctions lambda

Pour des fonctions très courtes, on peut utiliser des fonctions anonymes, connues aussi sous le nom de fonctions lambda.

Prenons l'exemple d'une fonction retournant la valeur de l'addition de deux nombres :

```
def addition(a,b):
    return a+b
```

On peut écrire cette fonction en une seule ligne, avec le mot clé **lambda** :

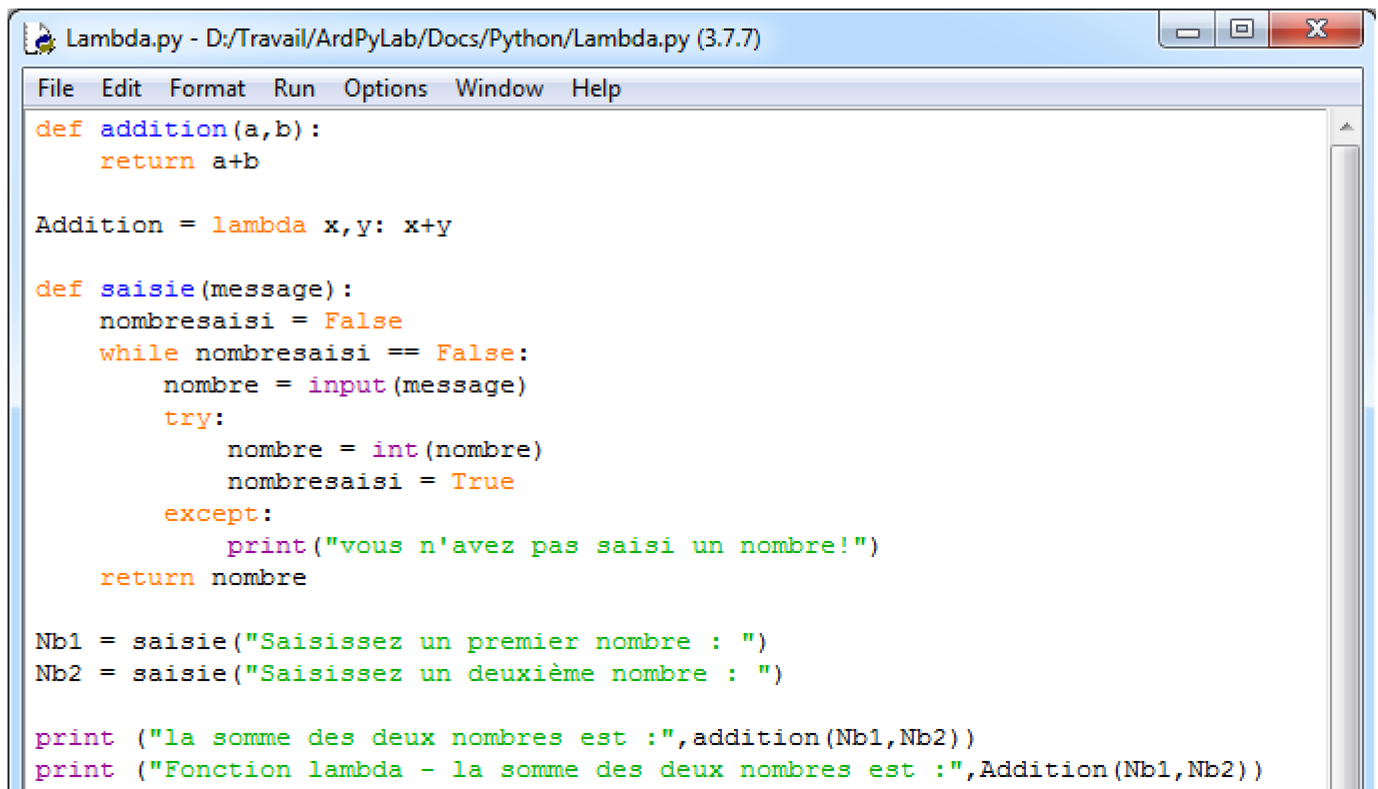
```
Addition = lambda x,y: x+y
```

A noter cependant qu'avec les fonctions lambda :

. On ne peut les écrire que sur une seule ligne.

. On ne peut pas avoir plus d'une instruction dans la fonction.

Voici le programme permettant d'afficher le résultat de l'addition :

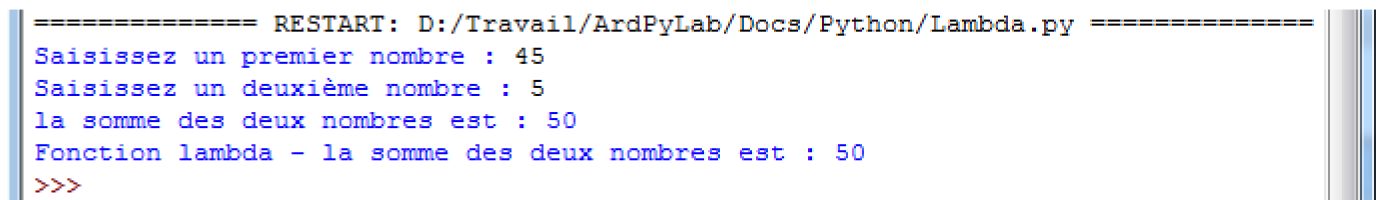


Lambda.py - D:/Travail/ArdPyLab/Docs/Python/Lambda.py (3.7.7)

File Edit Format Run Options Window Help

```python
def addition(a,b):
    return a+b

Addition = lambda x,y: x+y

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

Nb1 = saisie("Saisissez un premier nombre : ")
Nb2 = saisie("Saisissez un deuxième nombre : ")

print ("la somme des deux nombres est :",addition(Nb1,Nb2))
print ("Fonction lambda - la somme des deux nombres est :",Addition(Nb1,Nb2))
```

<u>Résultats dans la fenêtre Python Shell</u> :



```
============== RESTART: D:/Travail/ArdPyLab/Docs/Python/Lambda.py ==============
Saisissez un premier nombre : 45
Saisissez un deuxième nombre : 5
la somme des deux nombres est : 50
Fonction lambda - la somme des deux nombres est : 50
>>>
```

2.3 Les fichiers

Pour sauvegarder des données, il peut être intéressant de les stocker dans des fichiers qu'il sera possible de lire ultérieurement et éventuellement modifier.

En premier, il faut ouvrir ou créer un fichier avec la fonction **open()**. Cette fonction prend en premier paramètre le chemin du fichier et en second paramètre le type d'ouverture :

fichier = open("chemin", "type d'ouverture")

On peut également préciser l'encodage du fichier :

fichier = open("chemin", "type d'ouverture", encoding="utf-8")

(UTF-8 est un encodage universel qui réunit les caractères utilisés par toutes les langues)

Les types d'ouvertures sont :

. **"r"**, pour une ouverture en lecture (READ).

. **"w"**, pour une ouverture en écriture (WRITE), à chaque ouverture le contenu du fichier est écrasé. Si le fichier n'existe pas python le crée.

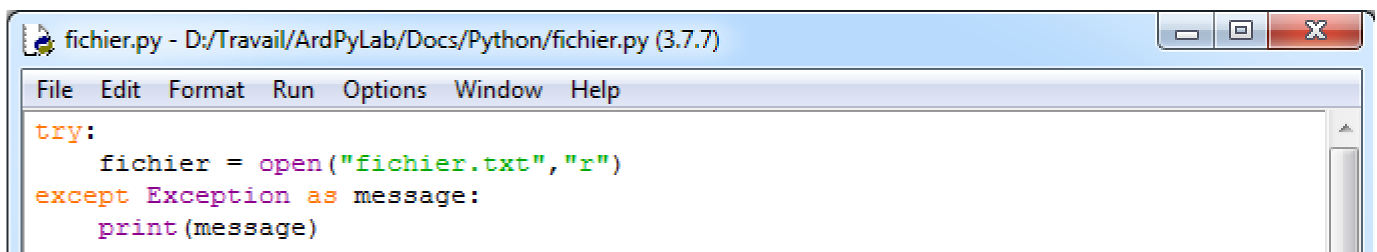
. **"a"**, pour une ouverture en mode ajout à la fin du fichier (APPEND). Si le fichier n'existe pas python le crée.

. **"b"**, pour une ouverture en mode binaire.

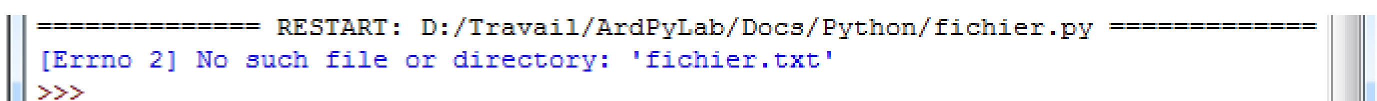
. **"t"**, pour une ouverture en mode texte.

. **"x"**, crée un nouveau fichier et l'ouvre pour écriture.

Pour vérifier que l'ouverture du fichier se fait correctement, il faut traiter une exception de type Exception (elle peut fournir une description de l'erreur) :



Résultat dans la fenêtre Python Shell :

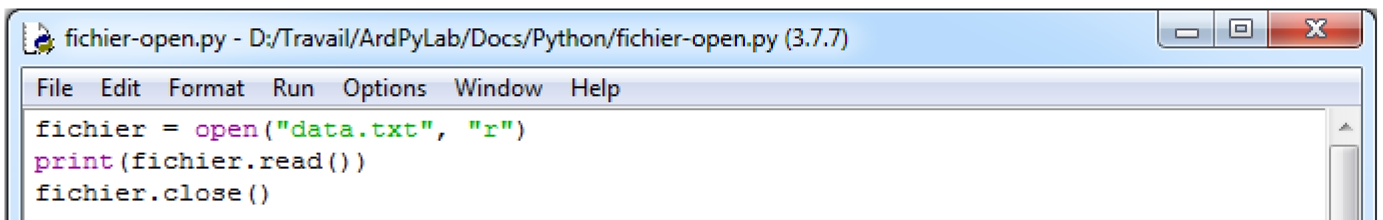


Après ouverture du fichier et une fois les instructions sur le fichier terminées, il faut le fermer. Pour cela, on utilise la méthode **close() :**

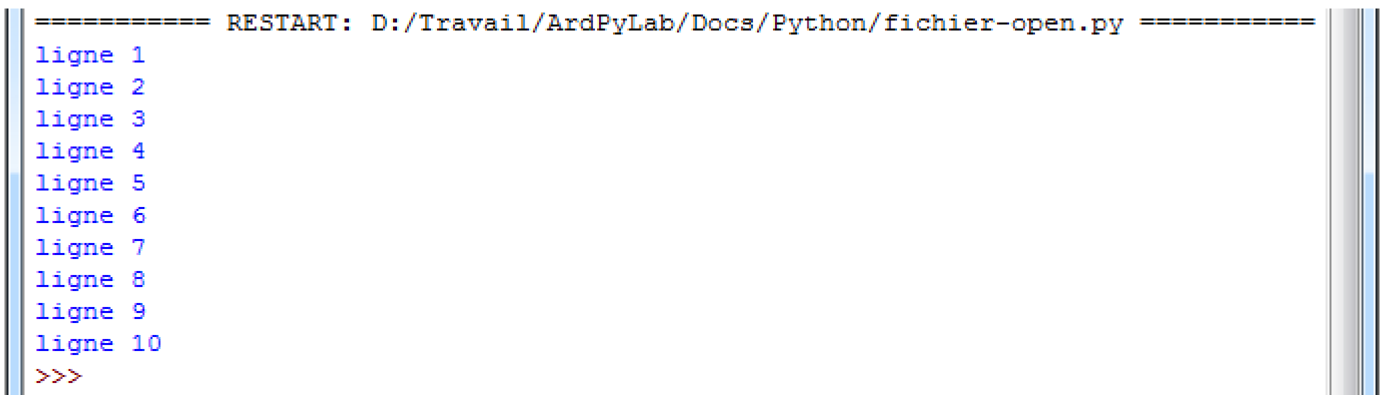
fichier.close()

. Lecture d'un fichier

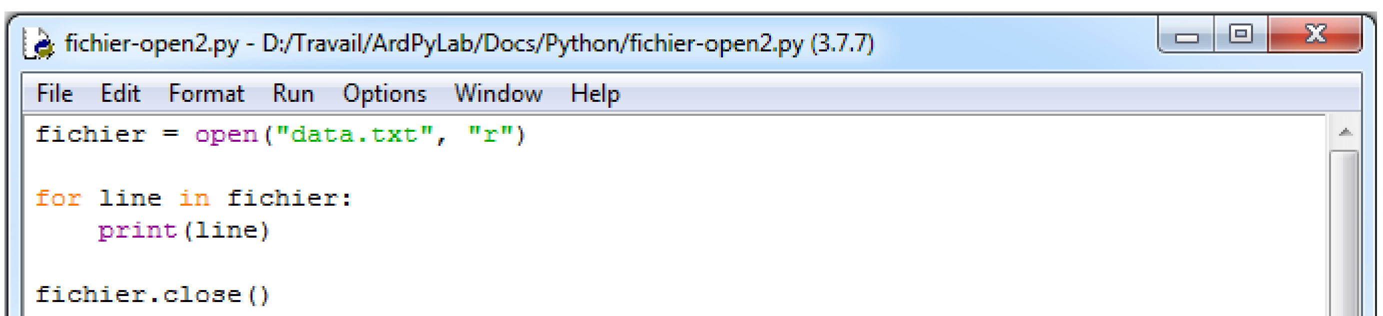
- Pour afficher tout le contenu d'un fichier, on peut utiliser la méthode **read** sur l'objet **fichier** :



Résultat dans la fenêtre Python Shell :

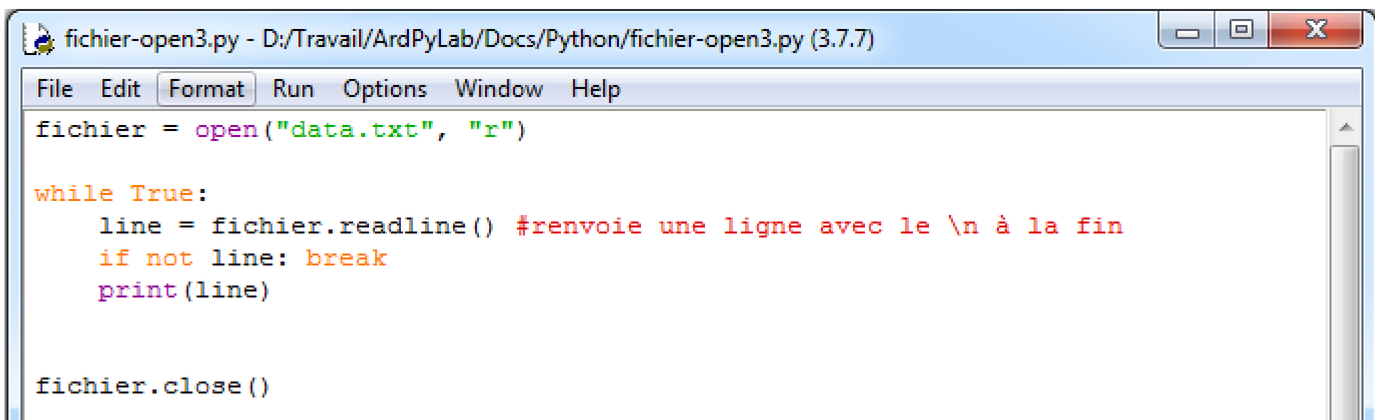


L'objet de type **file** est un objet qui peut se comporter comme une liste, ce qui permet d'utiliser également une boucle **for** :



Attention, lorsque l'on récupère une ligne d'un fichier, c'est une chaîne de caractères qui se termine par le caractère '**\n**' de fin de ligne.

Ou bien simplement, la méthode **readline()** :

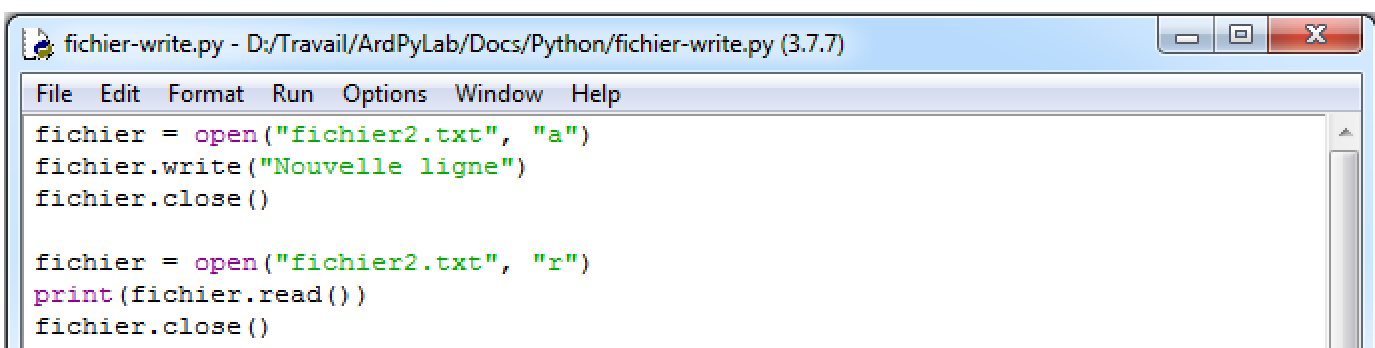

```
fichier = open("data.txt", "r")

while True:
    line = fichier.readline() #renvoie une ligne avec le \n à la fin
    if not line: break
    print(line)


fichier.close()
```

Ces deux façons de lire un fichier donnent cet affichage dans la fenêtre Python Shell :

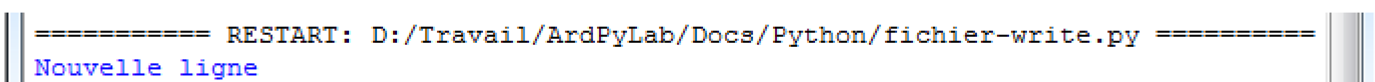
```
=========== RESTART: D:/Travail/ArdPyLab/Docs/Python/fichier-open3.py ==========
ligne 1

ligne 2

ligne 3

ligne 4

ligne 5

ligne 6

ligne 7

ligne 8

ligne 9

ligne 10
>>>
```

. Ecrire dans un fichier

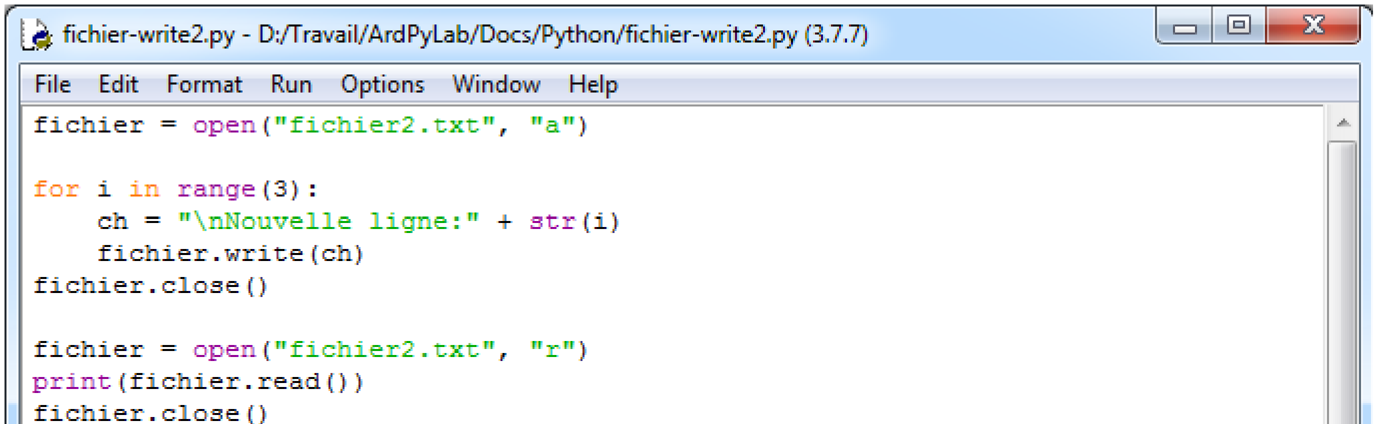
Pour écrire dans un fichier, il faut au préalable l'ouvrir ou le créer en mode **"w"**, **"a"** ou **"x"**.


```
fichier = open("fichier2.txt", "a")
fichier.write("Nouvelle ligne")
fichier.close()

fichier = open("fichier2.txt", "r")
print(fichier.read())
fichier.close()
```

Résultat dans la fenêtre Python Shell :

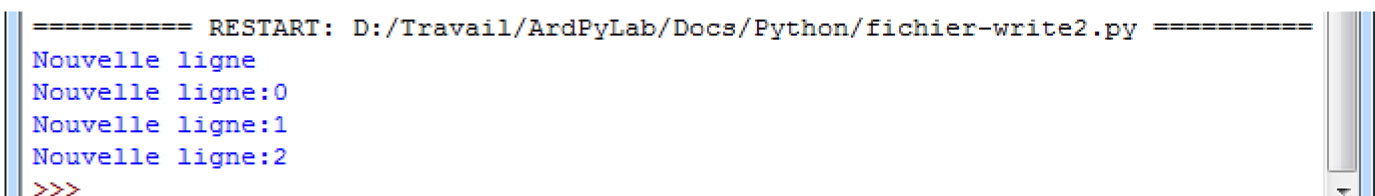

```
=========== RESTART: D:/Travail/ArdPyLab/Docs/Python/fichier-write.py ==========
Nouvelle ligne
```

A noter que pour le monde d'ouverture **"a"**, pour écrire à la ligne, on peut utiliser le saut de ligne \n :



```
fichier = open("fichier2.txt", "a")

for i in range(3):
    ch = "\nNouvelle ligne:" + str(i)
    fichier.write(ch)
fichier.close()

fichier = open("fichier2.txt", "r")
print(fichier.read())
fichier.close()
```

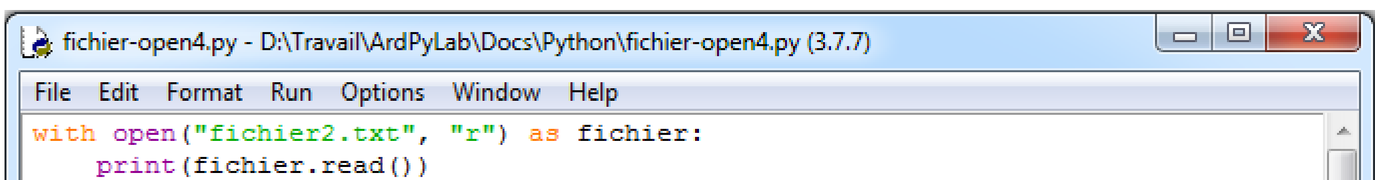
Résultat dans la fenêtre Python Shell :



```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/fichier-write2.py ==========
Nouvelle ligne
Nouvelle ligne:0
Nouvelle ligne:1
Nouvelle ligne:2
>>>
```

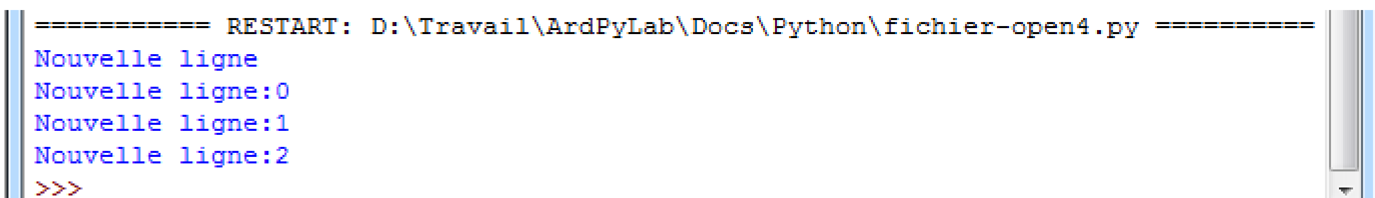
. Autre méthode d'ouverture de fichiers

Une ouverture de fichiers avec le mot clé **with** est également possible et cette méthode présente l'avantage de ne pas être obligé de fermer le fichier après traitement par le programme.



```
with open("fichier2.txt", "r") as fichier:
    print(fichier.read())
```

Résultat dans la fenêtre Python Shell :



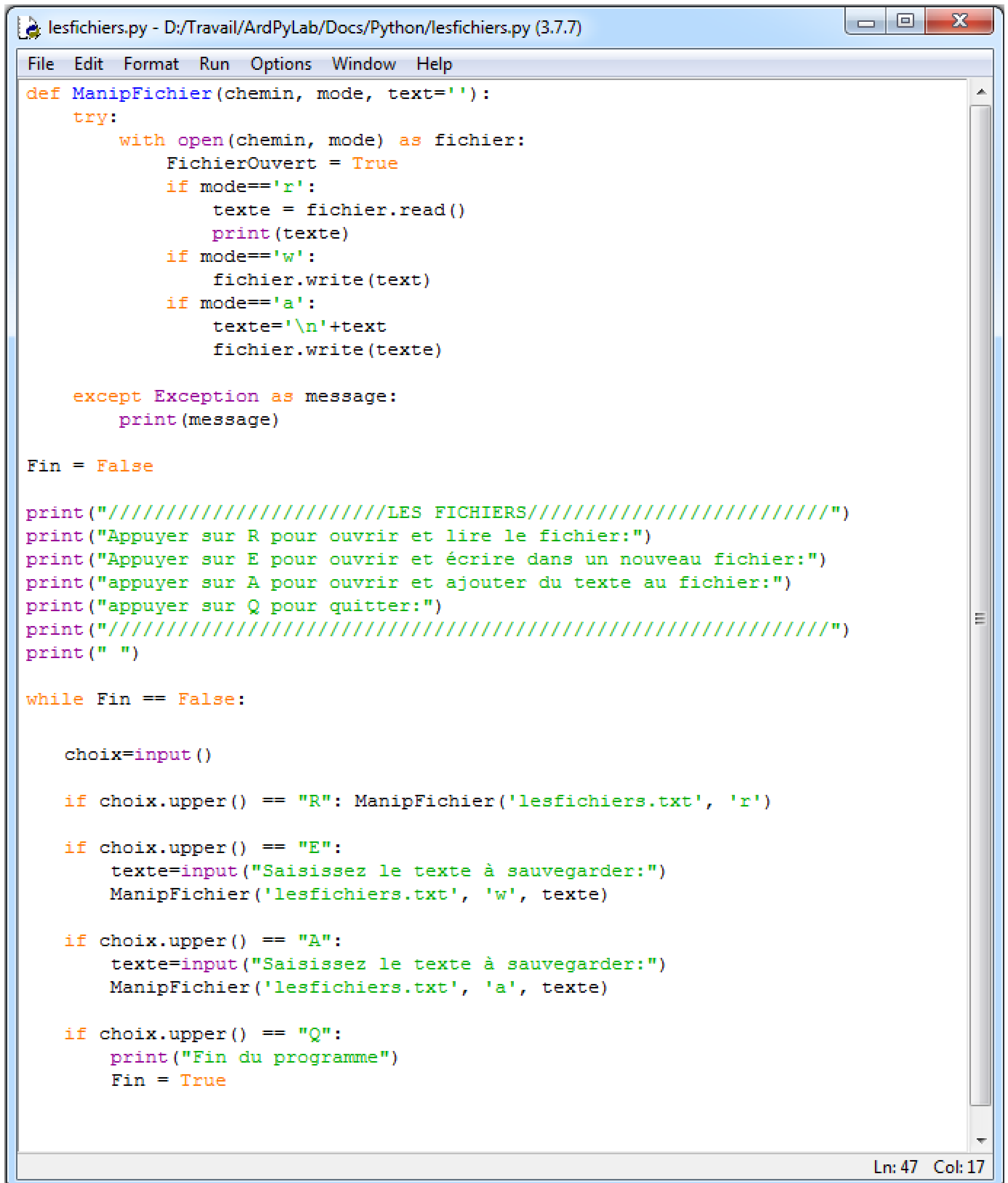
```
=========== RESTART: D:\Travail\ArdPyLab\Docs\Python\fichier-open4.py ==========
Nouvelle ligne
Nouvelle ligne:0
Nouvelle ligne:1
Nouvelle ligne:2
>>>
```

Remarque :

Si l'on ne précise pas l'emplacement où l'on veut créer un fichier, celui-ci sera créé dans le répertoire courant (en général, dossier du script python).

. Synthèse sur la manipulation des fichiers

Le script ci-dessous résume les principales manipulations qu'il est possible d'effectuer avec un fichier :



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

File Edit Format Run Options Window Help

```
def ManipFichier(chemin, mode, text=''):
    try:
        with open(chemin, mode) as fichier:
            FichierOuvert = True
            if mode=='r':
                texte = fichier.read()
                print(texte)
            if mode=='w':
                fichier.write(text)
            if mode=='a':
                texte='\n'+text
                fichier.write(texte)

    except Exception as message:
        print(message)

Fin = False

print("////////////////////////LES FICHIERS/////////////////////////")
print("Appuyer sur R pour ouvrir et lire le fichier:")
print("Appuyer sur E pour ouvrir et écrire dans un nouveau fichier:")
print("appuyer sur A pour ouvrir et ajouter du texte au fichier:")
print("appuyer sur Q pour quitter:")
print("//////////////////////////////////////////////////////////////")
print(" ")

while Fin == False:

   choix=input()

   if choix.upper() == "R": ManipFichier('lesfichiers.txt', 'r')

   if choix.upper() == "E":
       texte=input("Saisissez le texte à sauvegarder:")
       ManipFichier('lesfichiers.txt', 'w', texte)

   if choix.upper() == "A":
       texte=input("Saisissez le texte à sauvegarder:")
       ManipFichier('lesfichiers.txt', 'a', texte)

   if choix.upper() == "Q":
       print("Fin du programme")
       Fin = True
```

Ln: 47 Col: 17

Les taches effectuées par la fonction **ManipFichier()** dépendent des arguments de la fonction. En effet, après avoir vérifié que l'ouverture du fichier, dont le chemin est en

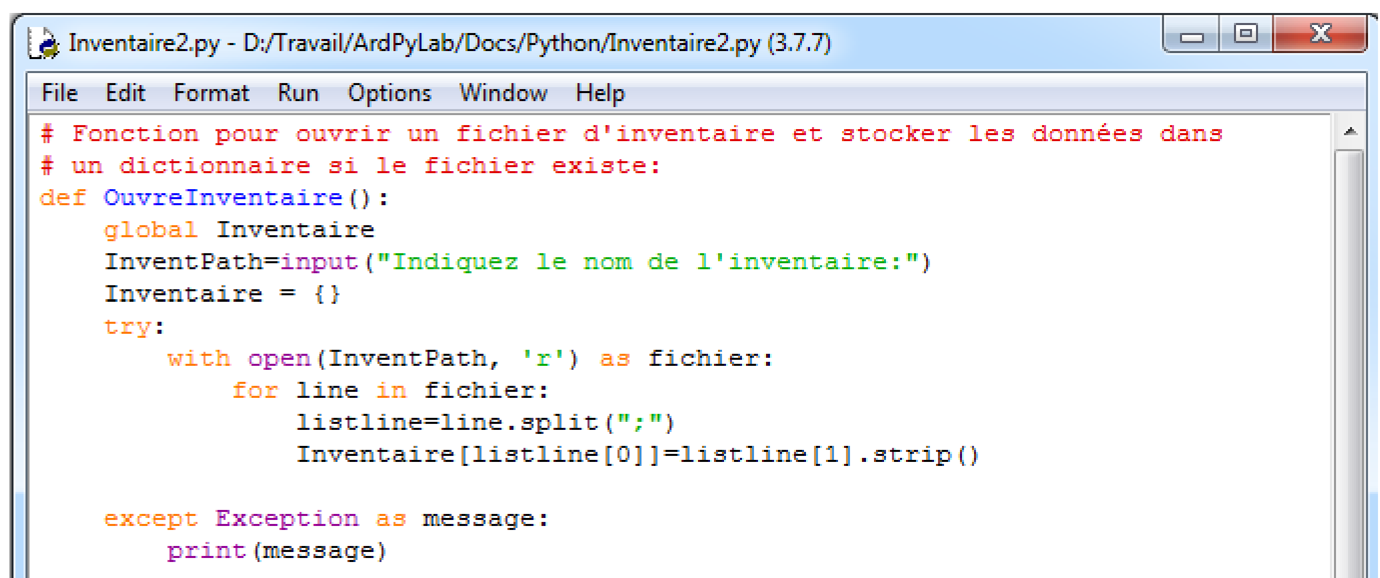
argument, se fait correctement (structure **try except**), le fichier est soit lu, soit créé et mis à jour, soit ouvert et mis à jour, en fonction des valeurs des arguments **mode** et **text**.

Résultats dans la fenêtre Python Shell :

```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/lesfichiers.py ===========
////////////////////////LES FICHIERS////////////////////////
Appuyer sur R pour ouvrir et lire le fichier:
Appuyer sur E pour ouvrir et écrire dans un nouveau fichier:
appuyer sur A pour ouvrir et ajouter du texte au fichier:
appuyer sur Q pour quitter:
//////////////////////////////////////////////////////////////

r
[Errno 2] No such file or directory: 'lesfichiers.txt'
e
Saisissez le texte à sauvegarder:ligne1
r
ligne1
a
Saisissez le texte à sauvegarder:ligne2
r
ligne1
ligne2
e
Saisissez le texte à sauvegarder:new
r
new
q
Fin du programme
```

. Exemple d'application

Maintenant que nous savons sauvegarder des données et définir des fonctions, nous allons pouvoir modifier le programme de création d'inventaire de façon à pouvoir enregistrer les modifications qui lui sont apportées à l'aide de fonctions définies pour chaque action sur l'inventaire.




```
# Fonction pour ouvrir un fichier d'inventaire et stocker les données dans
# un dictionnaire si le fichier existe:
def OuvreInventaire():
    global Inventaire
    InventPath=input("Indiquez le nom de l'inventaire:")
    Inventaire = {}
    try:
        with open(InventPath, 'r') as fichier:
            for line in fichier:
                listline=line.split(";")
                Inventaire[listline[0]]=listline[1].strip()

    except Exception as message:
        print(message)
```

```python
# Fonction pour ajouter une ligne à l'inventaire ouvert ou initialement vide:
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

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

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

#Fonction pour sauvegarder l'inventaire en cours:
def SaveInventaire():
    global Inventaire
    InventPath=input("Indiquez le nom de sauvegarde de l'inventaire:")
    with open(InventPath, 'w') as fichier:
        for cle, valeur in Inventaire.items():
            fichier.write("{};{}".format(cle, valeur))
            fichier.write("\n")

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

print("////////////// INVENTAIRE MATERIEL //////////////")
print("appuyer sur O pour ouvrir un inventaire:")
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 E pour enregistrer l'inventaire:")
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() == "O": OuvreInventaire()
    if choix.upper() == "A": AjoutMatos()
    if choix.upper() == "V": ReadInventaire()
    if choix.upper() == "S": EffaceMatos()
    if choix.upper() == "E": SaveInventaire()
    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\Inventaire2.py ==
////////////// INVENTAIRE MATERIEL //////////////
appuyer sur O pour ouvrir un inventaire:
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 E pour enregistrer l'inventaire:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

o
Indiquez le nom de l'inventaire:invent
v
bécher : 10
eprouvette : 15
o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
eprouvette : 5
s
saisissez l'élément à supprimer dans l'inventaire:eprouvette
v
bécher : 10
erlenmeyer : 20
e
Indiquez le nom de sauvegarde de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
o
Indiquez le nom de l'inventaire:invent
v
bécher : 10
eprouvette : 15
o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
q
Fin du programme
>>>
```

```
== RESTART: C:\Users\Olivier\Docs\Travail\ArdPyLab\Docs\Python\Inventaire2.py ==
////////////// INVENTAIRE MATERIEL //////////////
appuyer sur O pour ouvrir un inventaire:
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 E pour enregistrer l'inventaire:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

o
Indiquez le nom de l'inventaire:invent2
a
saisissez le type de matériel à ajouter à l'inventaire:Pipette
saisissez la quantité de ce matériel:25
v
bécher : 10
erlenmeyer : 20
Pipette : 25
q
Fin du programme
```

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.

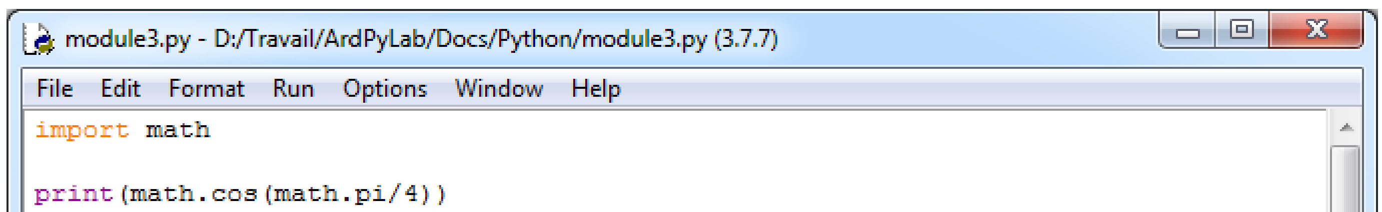
Un module rassemble les fonctions utilisées par le programme principal dans un même fichier situé dans le répertoire d'exécution du programme (pour les modules personnels).

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

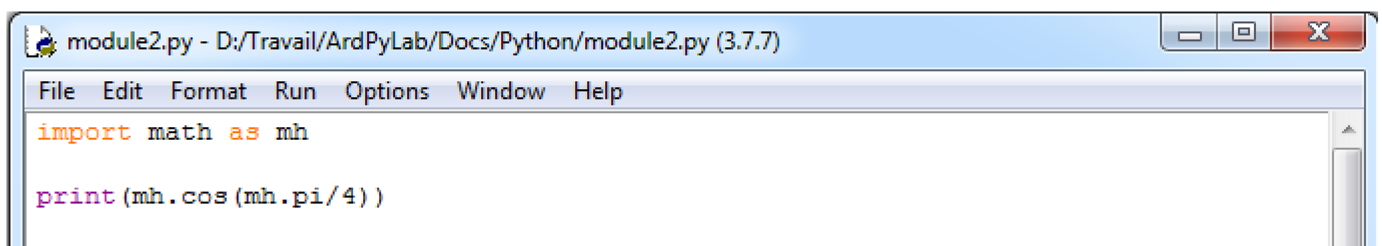
Deux syntaxes sont possibles pour l'importation d'un module :

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

Exemple : import math



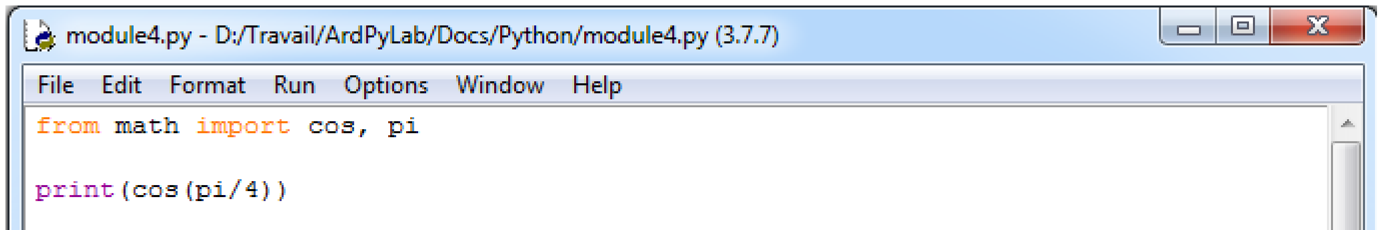
Le module peut être importé sous un autre nom, un alias. On utilise alors cet alias pour appeler les fonctions :



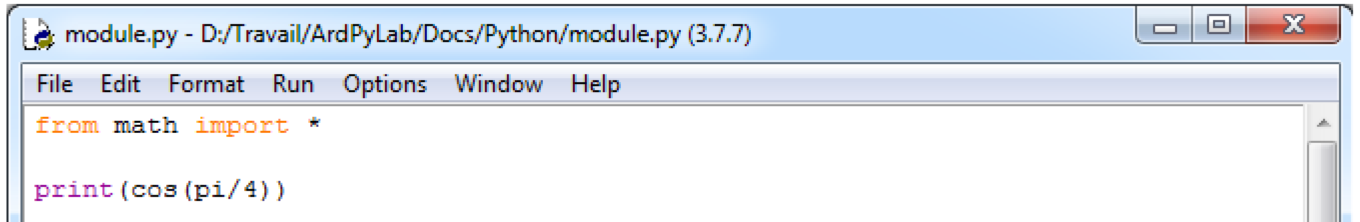
. la commande **from nom_module import obj1, obj2**... n'importe que les objets **obj1**, **obj2**... du module :

Exemple: from math import pi, sin, log

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



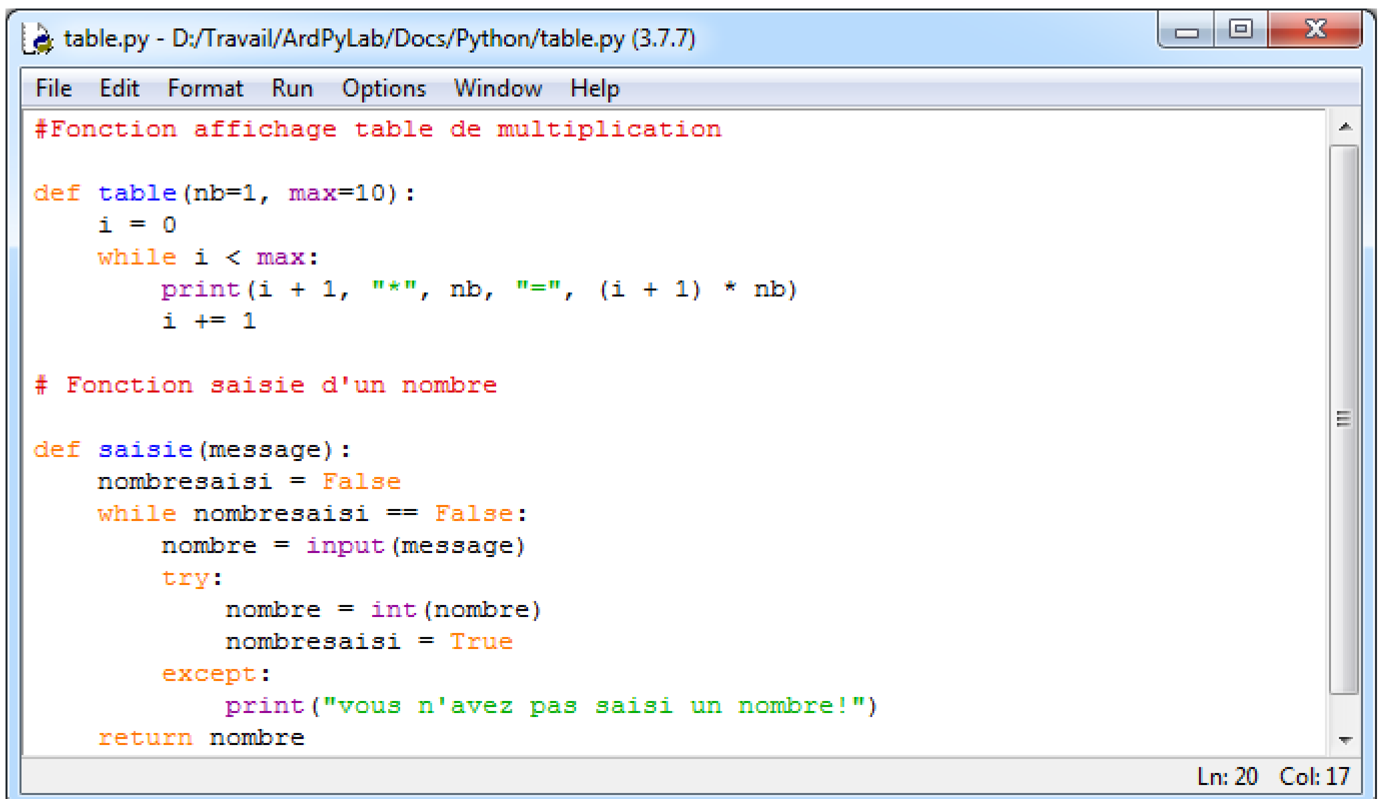
Ou toutes les fonctions du module :



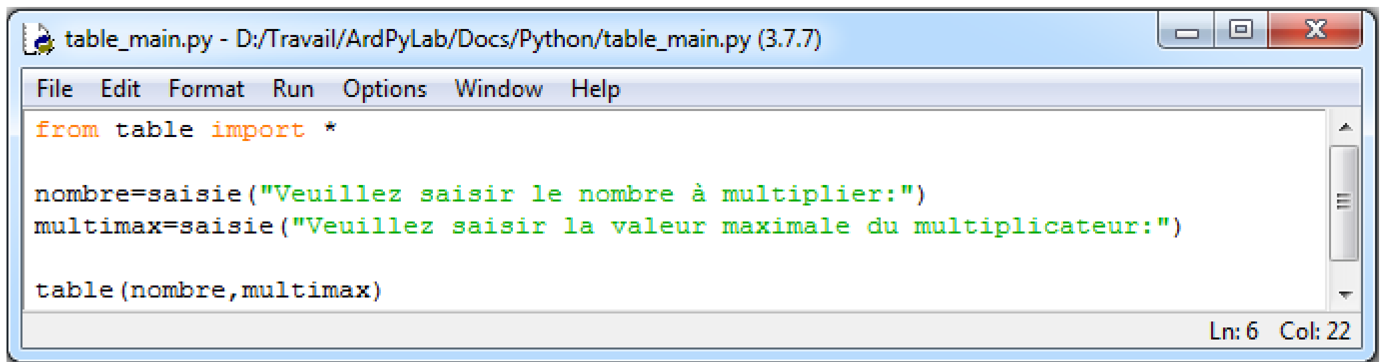
(Le caractère **'*'** permet d'importer toutes les fonctions du module.)

<u>Exemple</u> :

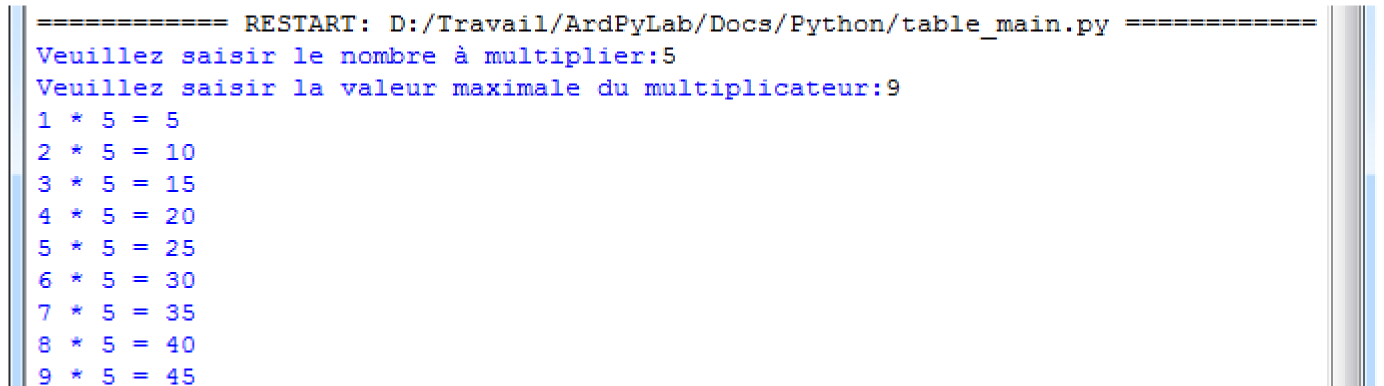
Reprenons le programme d'affichage de la table de multiplication en créant un module appelé **table.py** contenant les fonctions du programme.



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



Résultat dans la fenêtre Python Shell :

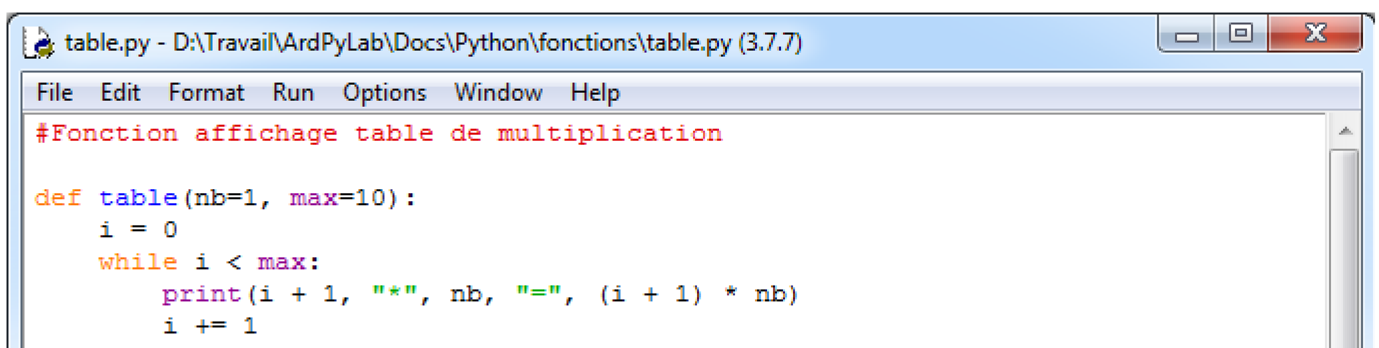


. Les packages

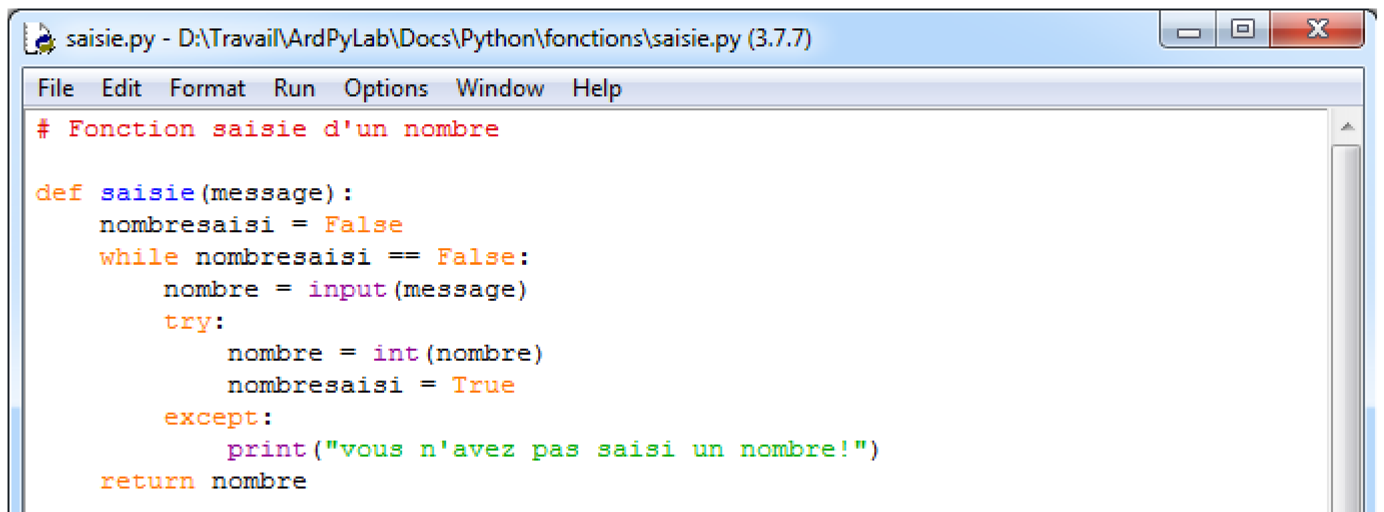
Quand on a un grand nombre de modules, il est intéressant de les organiser dans des dossiers. Un dossier qui rassemble des modules est appelé un **package** (paquets en français). Ce dossier doit contenir un fichier nommé **__init__.py** vide ou décrivant l'arborescence du package. Le fichier **__init__.py**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

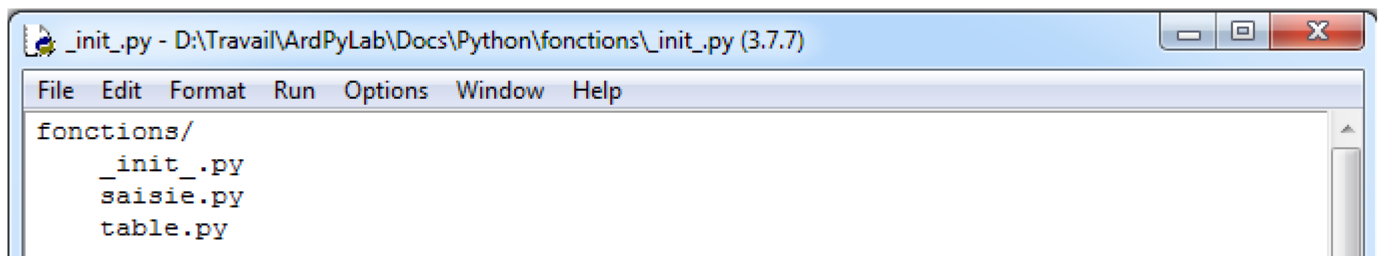


Et un module **saisie.py** contenant la fonction de saisie d'un nombre :



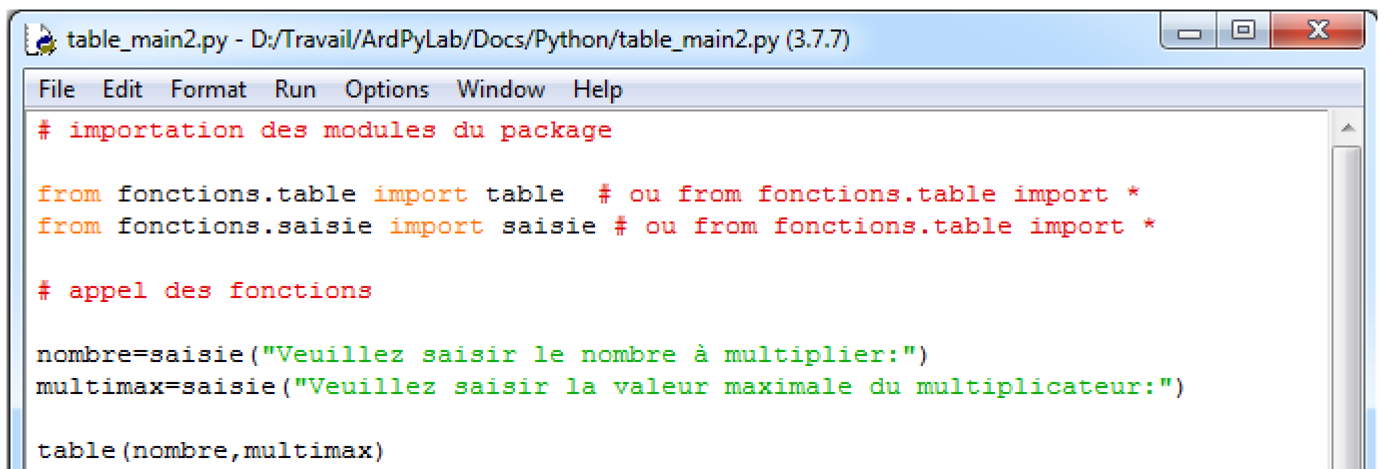
```
saisie.py - D:\Travail\ArdPyLab\Docs\Python\fonctions\saisie.py (3.7.7)
File Edit Format Run Options Window Help
# Fonction saisie d'un nombre

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** :



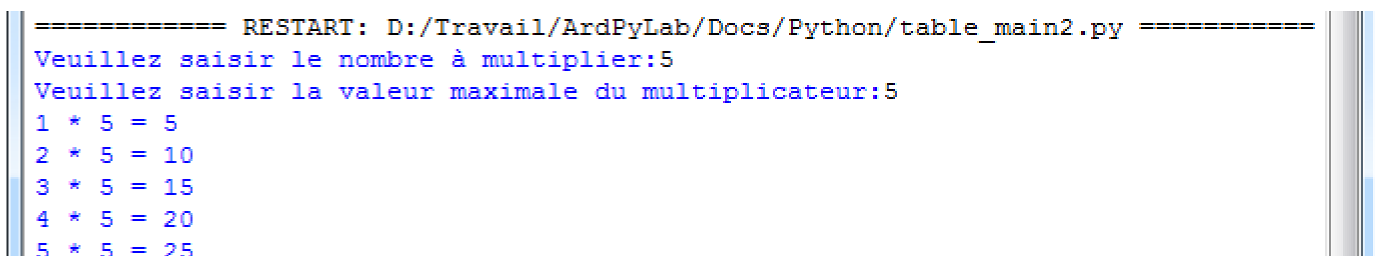
```
_init_.py - D:\Travail\ArdPyLab\Docs\Python\fonctions\_init_.py (3.7.7)
File Edit Format Run Options Window Help
fonctions/
    _init_.py
    saisie.py
    table.py
```

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)
```

<u>Résultat dans la fenêtre Python Shell</u> :



```
============ 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
```

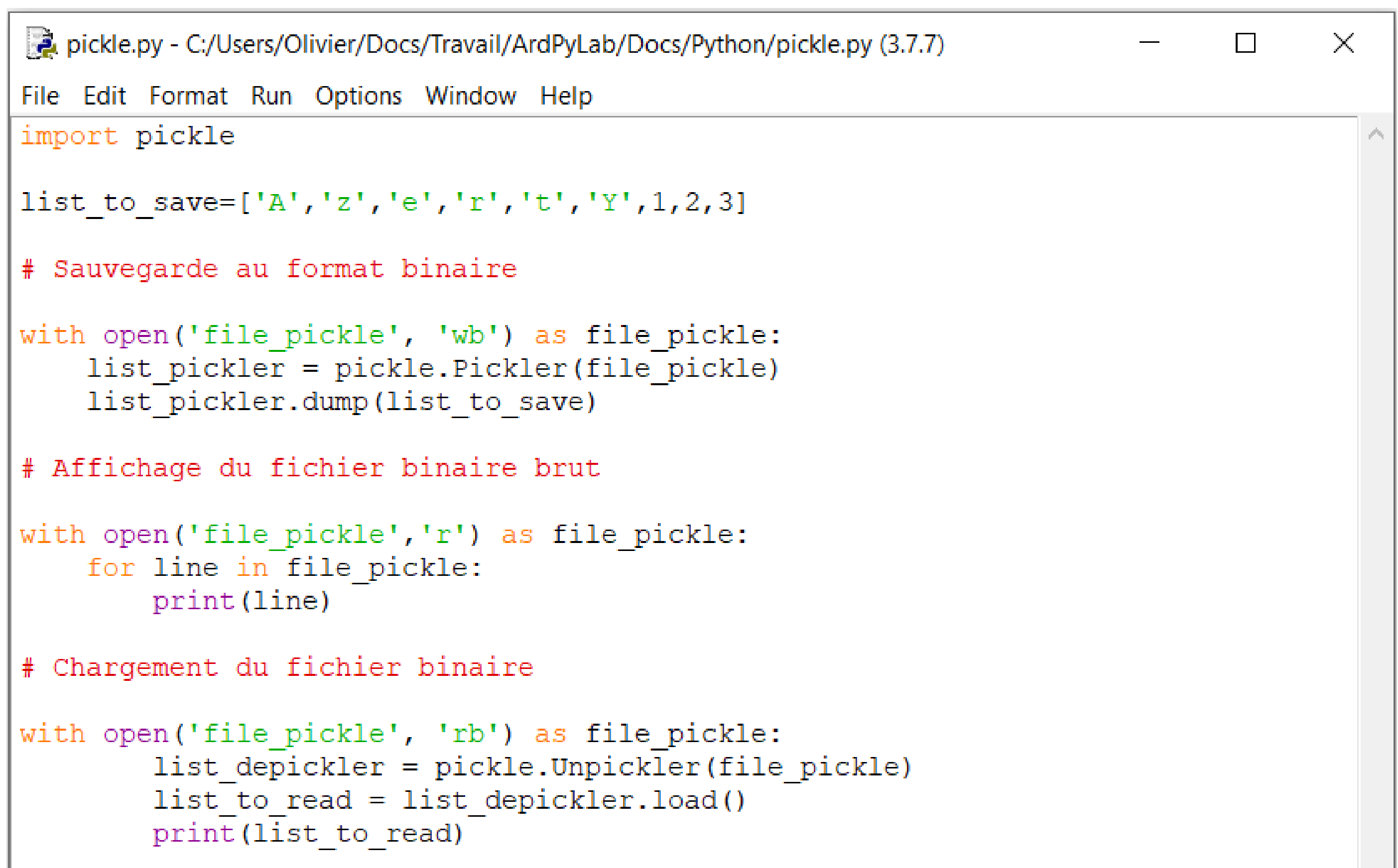
. La bibliothèque standard de Python

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

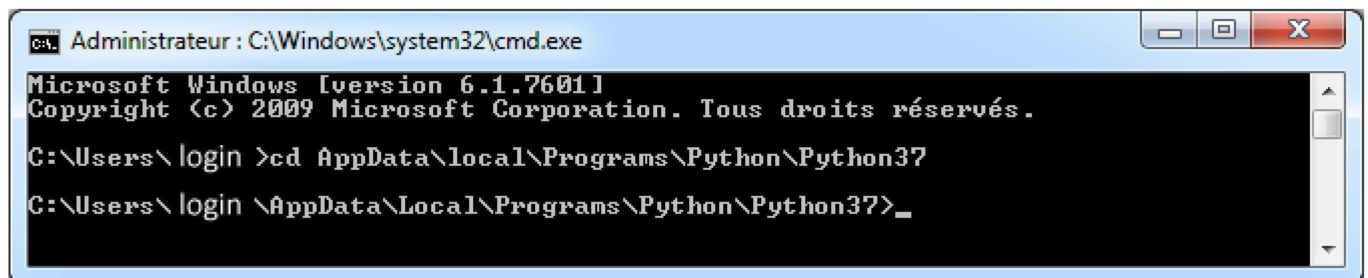
Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines.
Le site pypi.python.org/pypi (The Python Package Index) recense des milliers de modules et de packages !

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
. xn**	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)**.

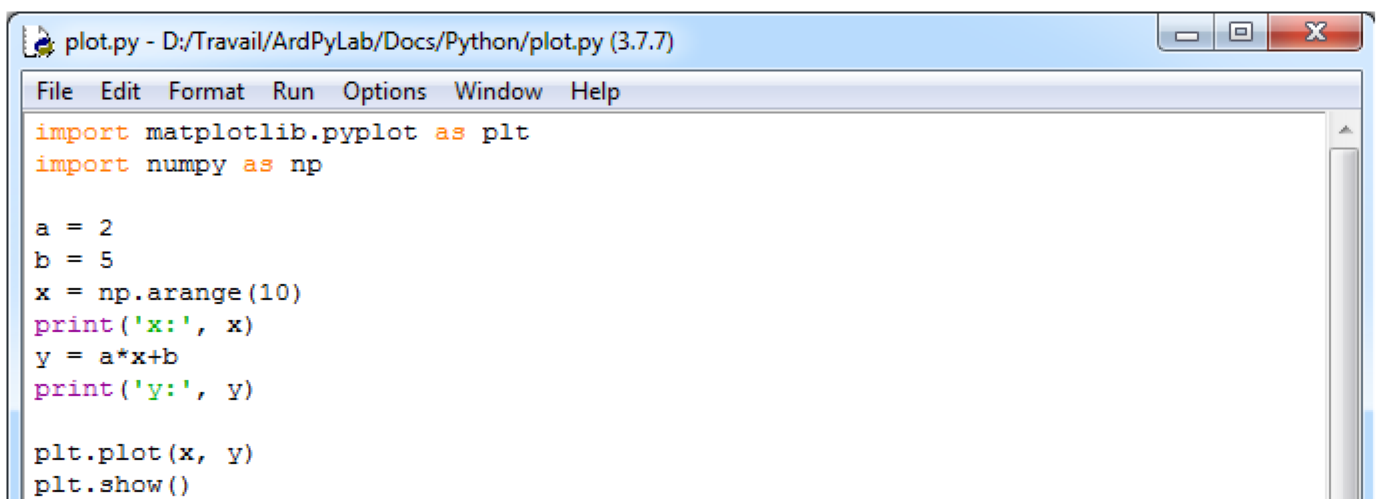
Cette caractéristique pourra être tracé à l'aide de la bibliothèque **matplolib**.

. 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**

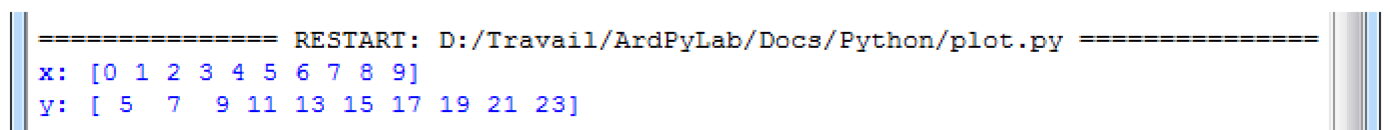



```
import matplotlib.pyplot as plt
import numpy as np

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)**).

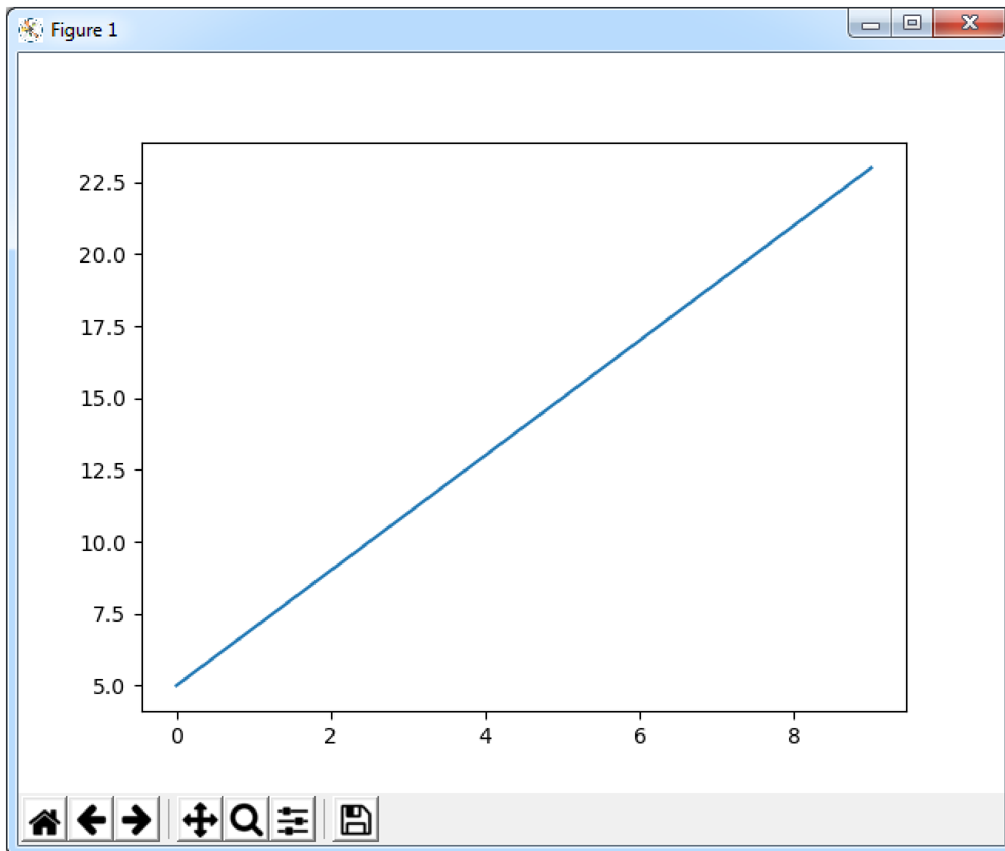
On applique la fonction de la droite à ce tableau pour obtenir un autre tableau numpy d'entiers correspondant aux ordonnées du graphes (**y= a*x + b**)

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

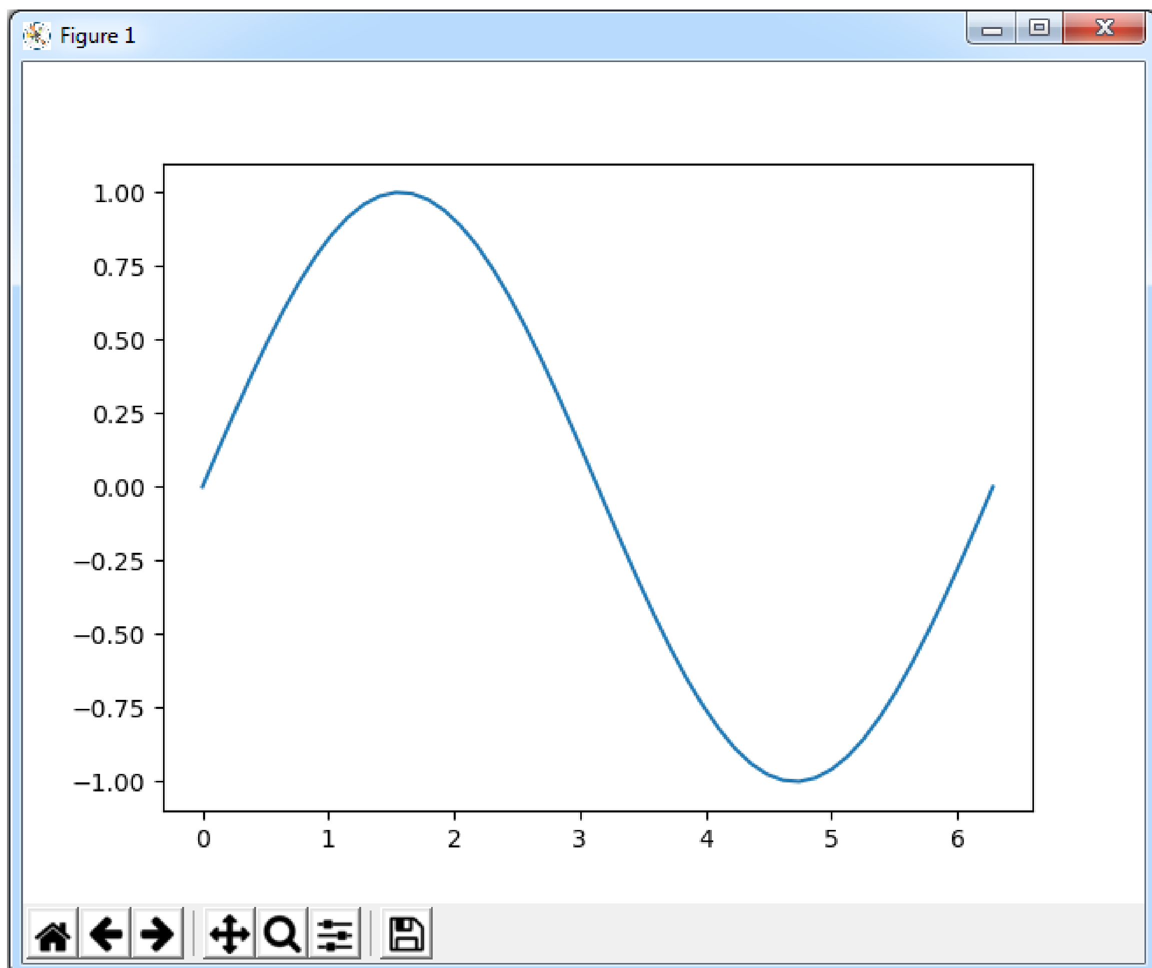
```
import matplotlib.pyplot as plt
import numpy as np

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 :

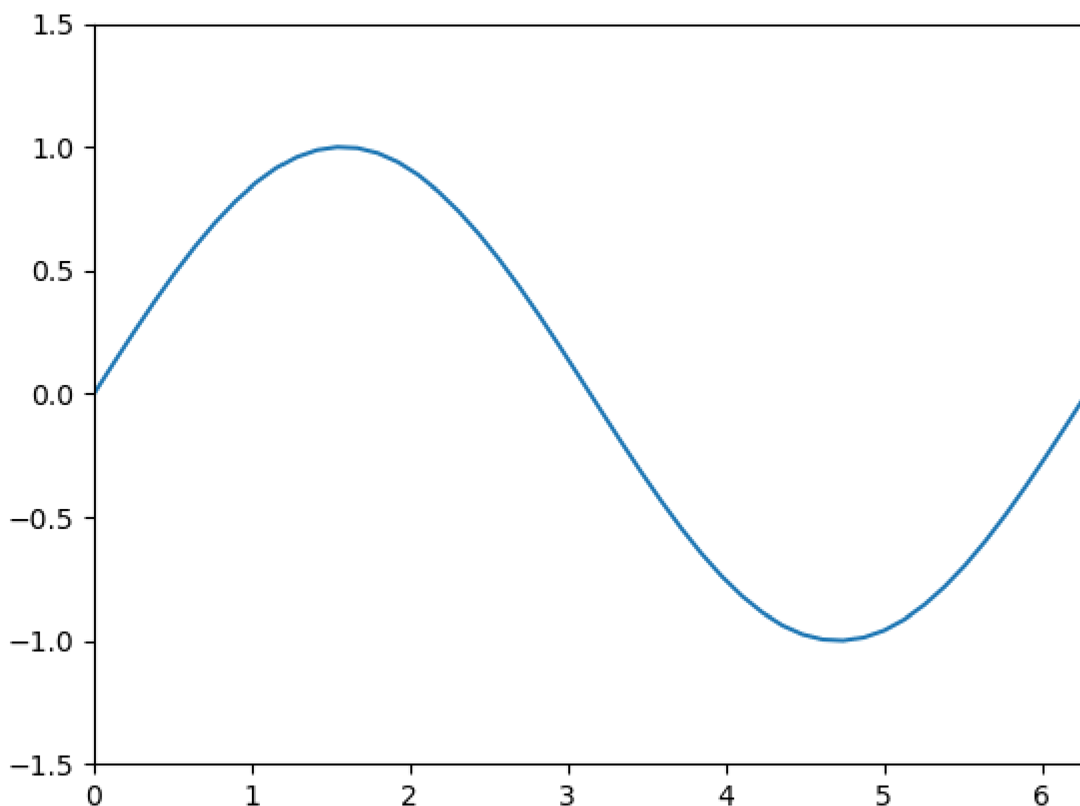
Il est possible fixer indépendamment les domaines des abscisses et des ordonnées en utilisant les fonctions **xlim()** et **ylim()**.

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")

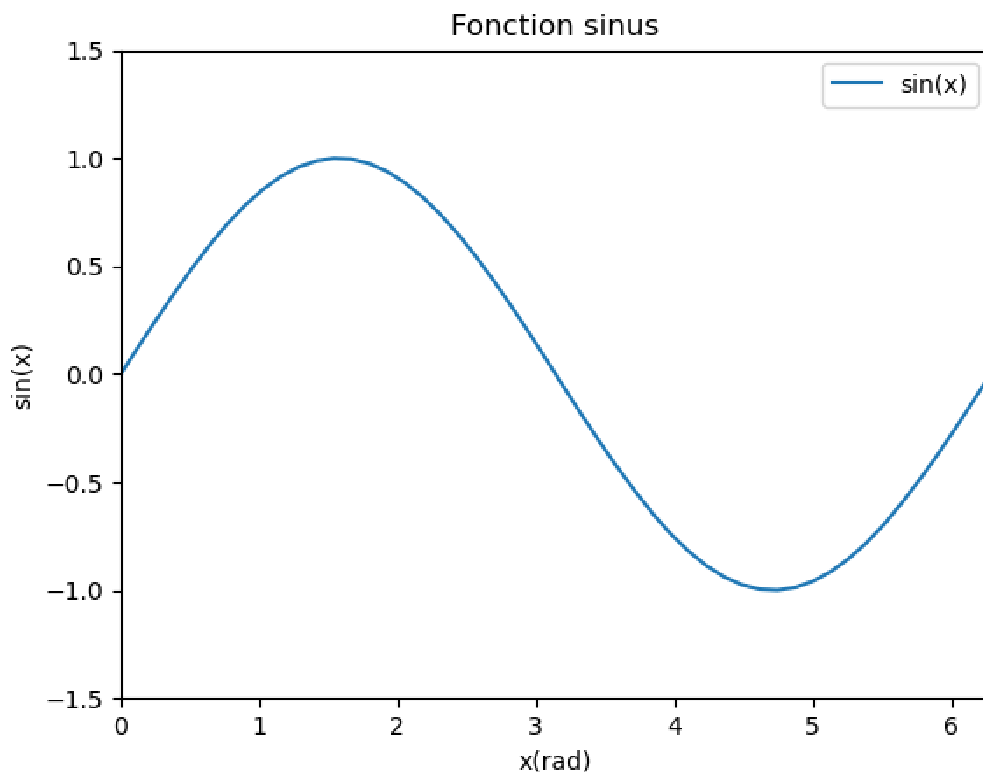
Exemple de mise en forme :

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)**.

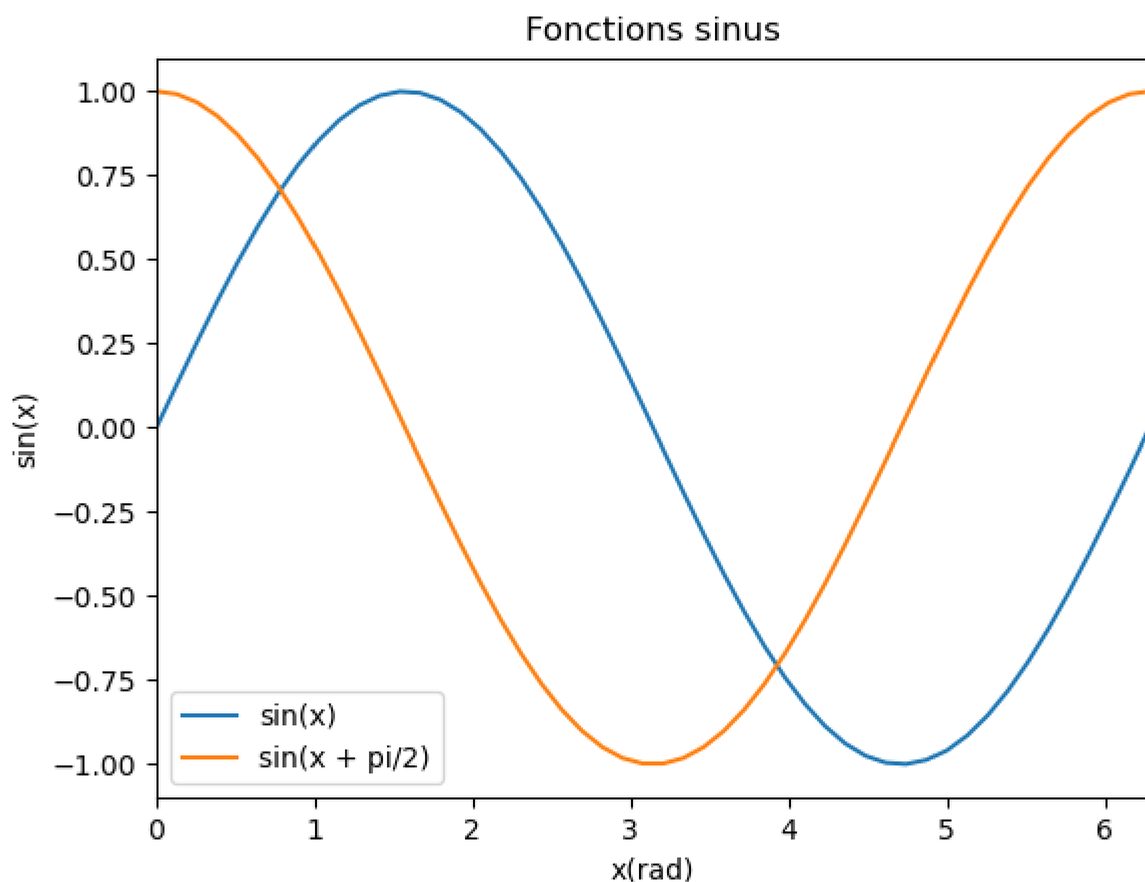
Exemple : Affichage de deux sinusoïdes avec un déphasage de π/2

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

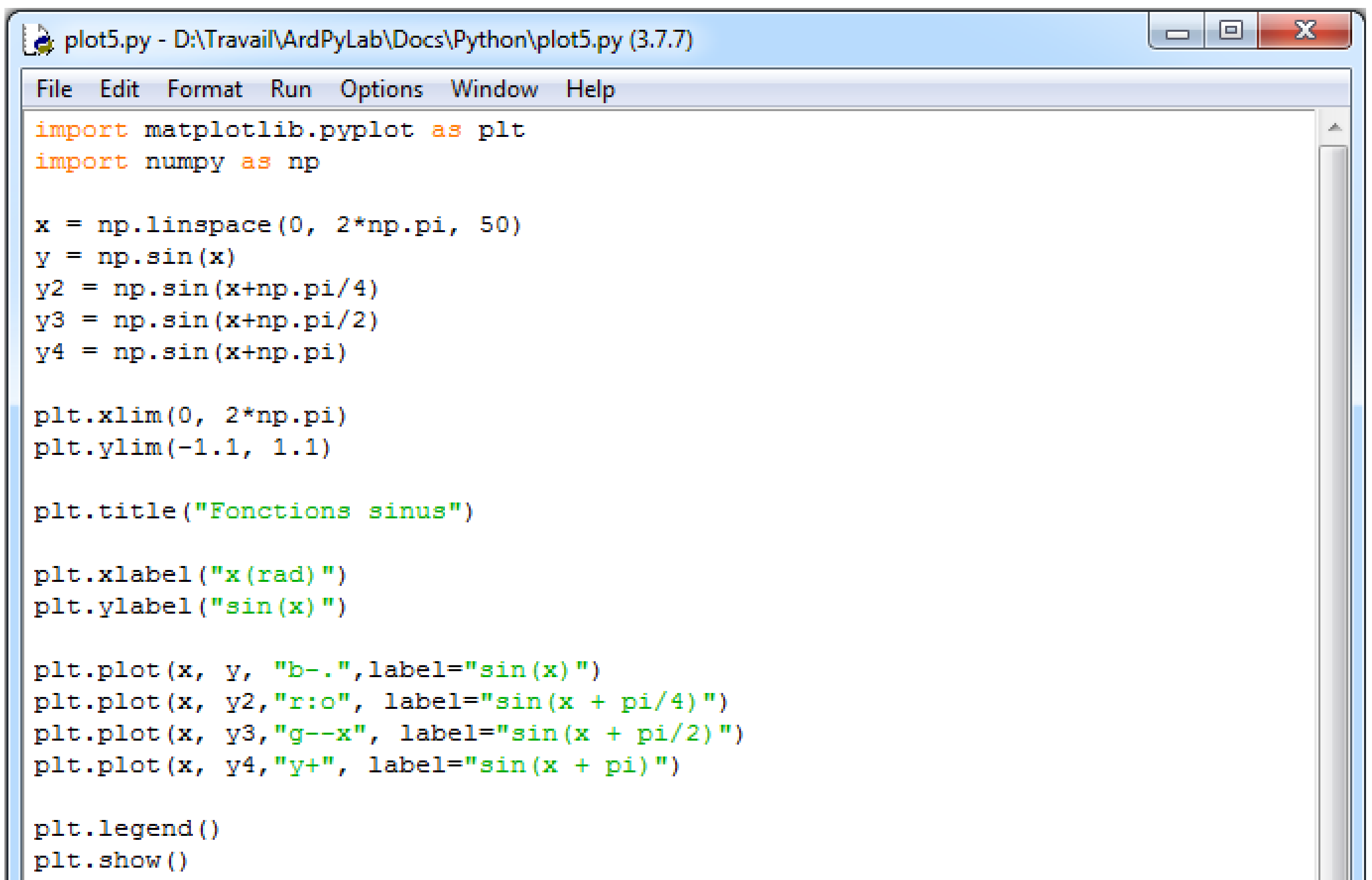
Les chaînes de caractères suivantes permettent de définir le symbole ("marker") :

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

Remarque :

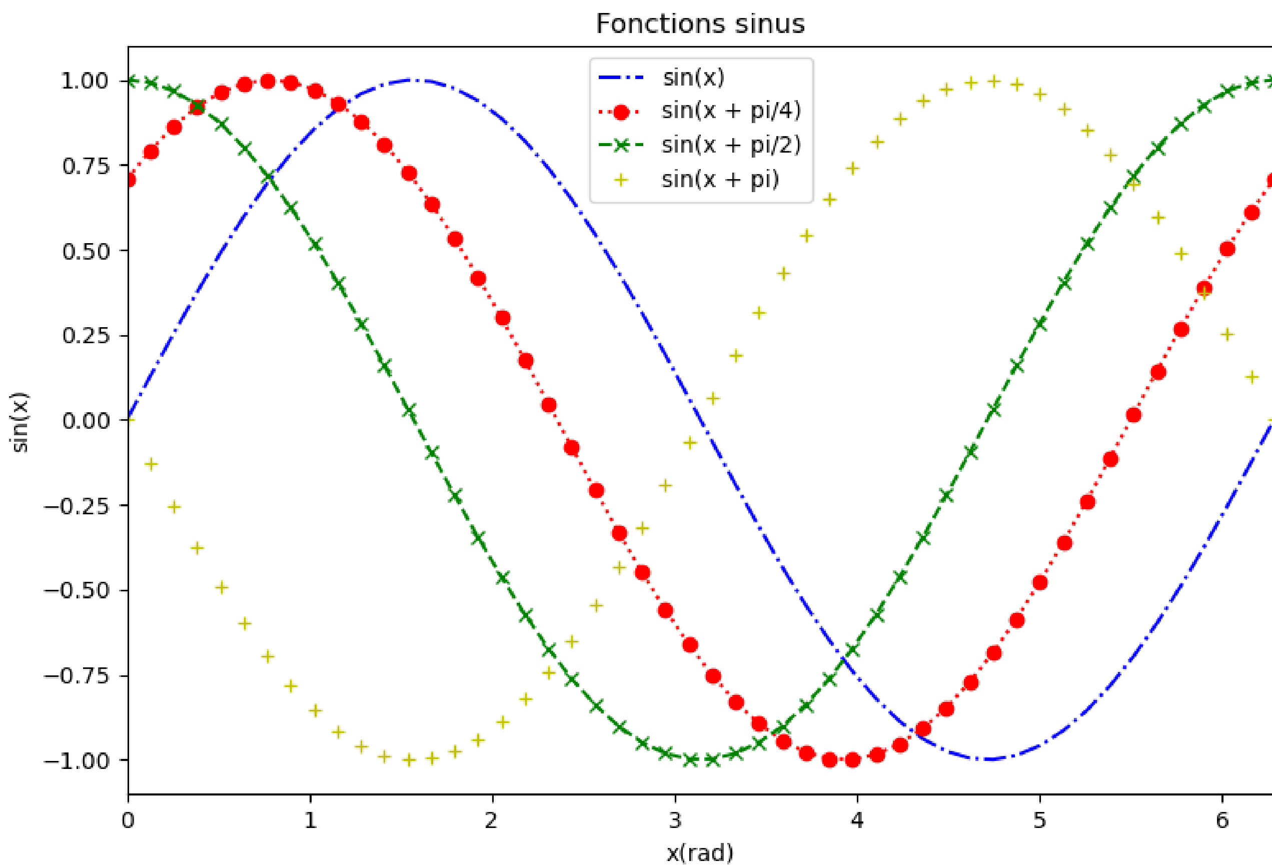
Pour une représentation graphique en nuage de points, il suffit d'indiquer un symbole sans préciser un style de ligne.

Exemples de styles de courbe :




```
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 :

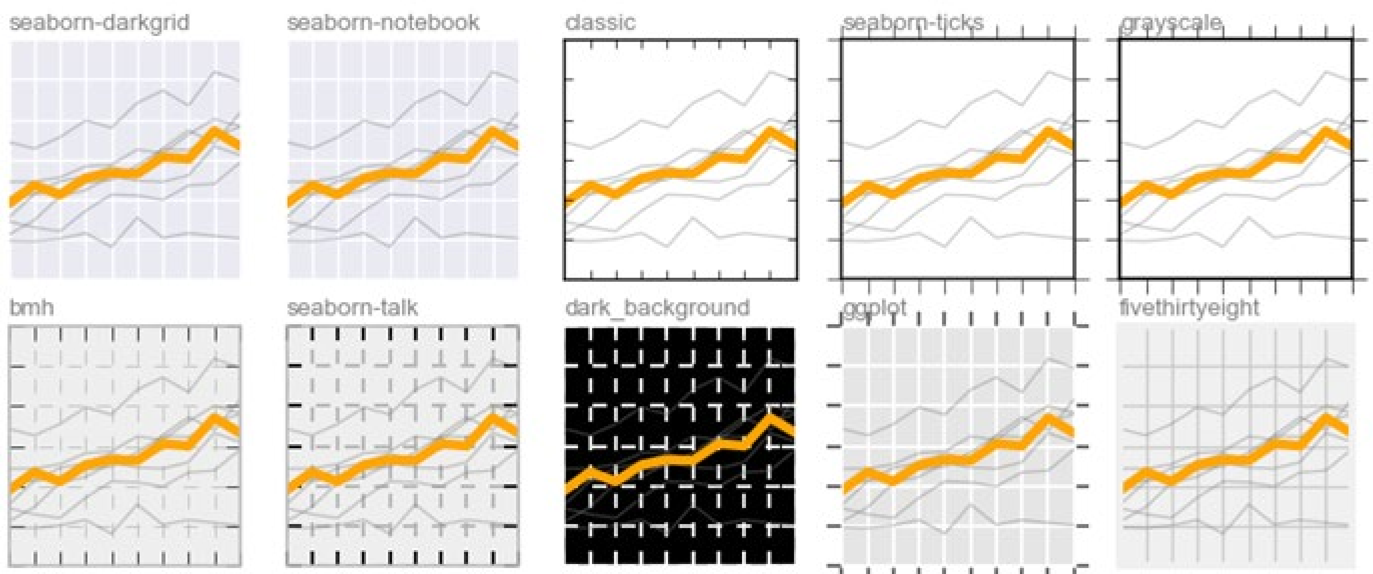


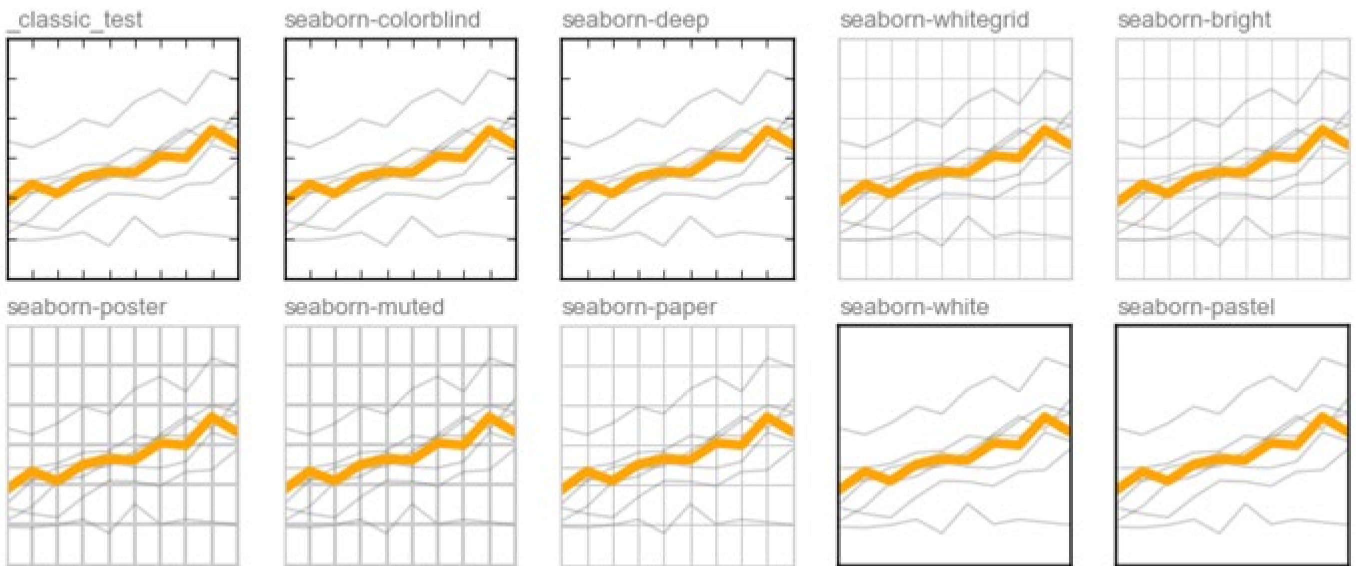
- <u>style de graphe</u> :

Le style du graphe est modifiable avec l'instruction :

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

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





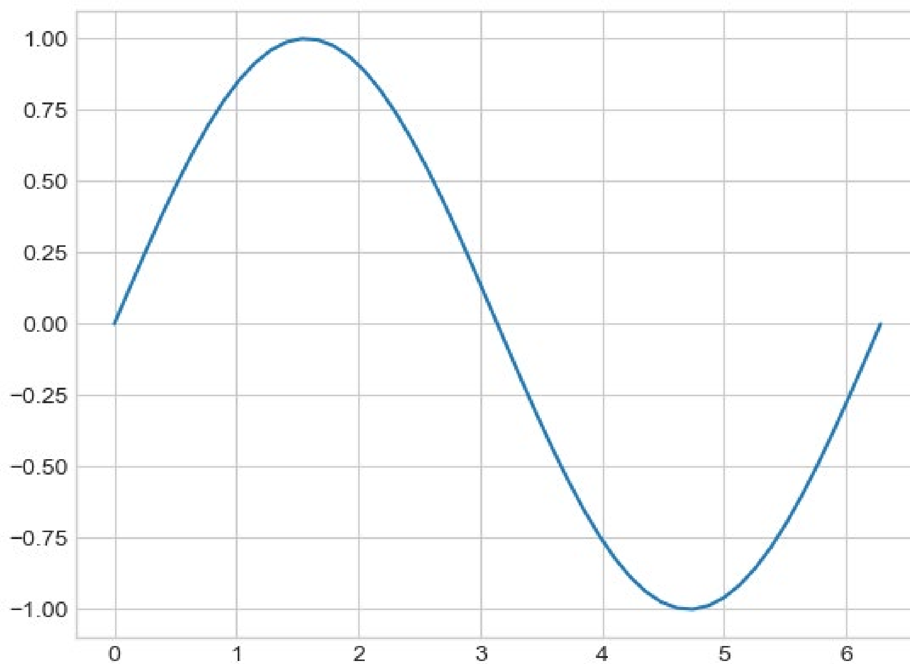
Exemple : Style ''seaborn-whitegrid''

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 :



- <u>Disposition et graphes multiples</u> :

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 :

subplot(211)

subplot(212)

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

ax2 = plt.subplot(122)

subplot(121) 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)

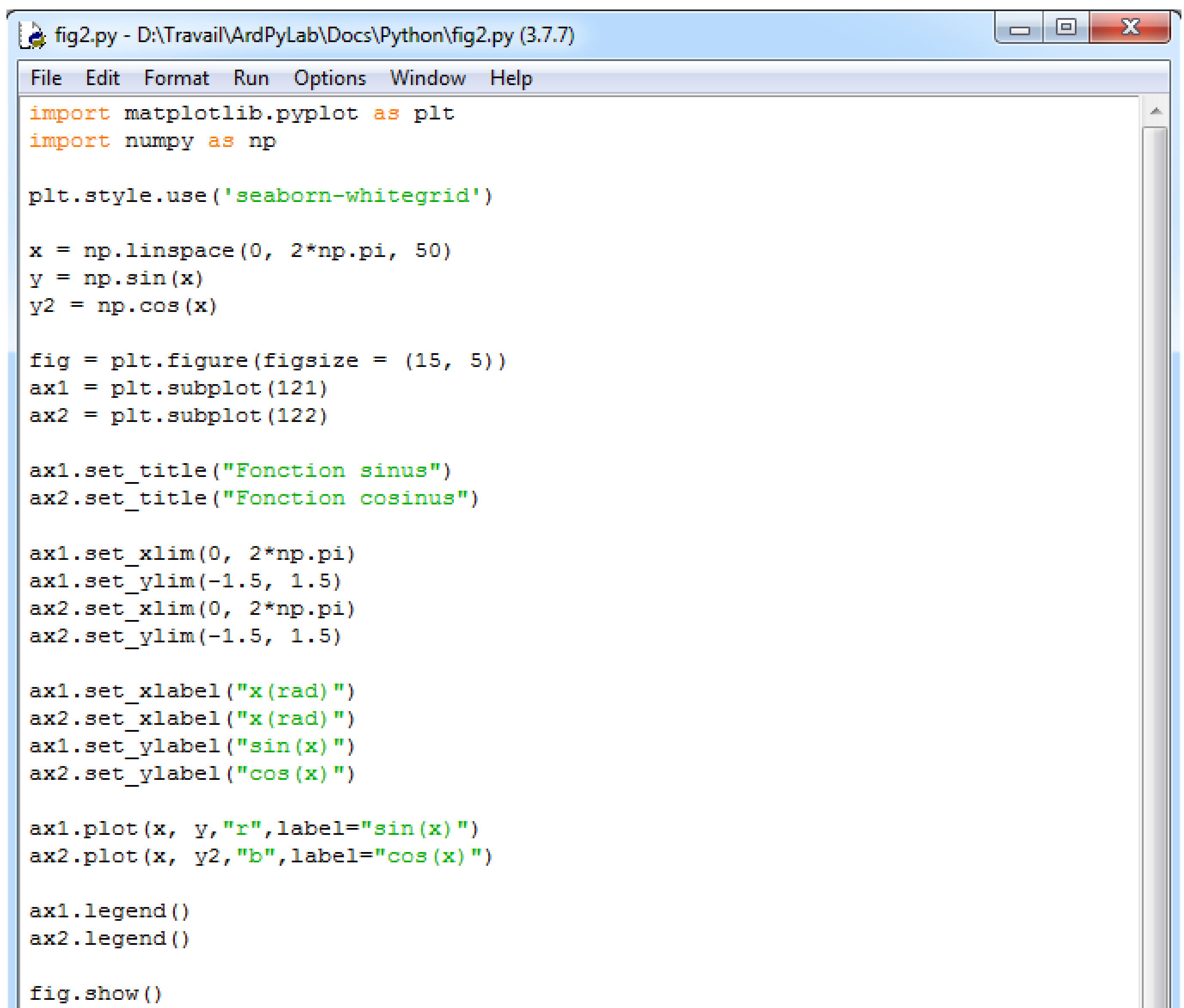
- Ajout d'étiquettes sur les axes :

  axe.set_xlabel("labelx")

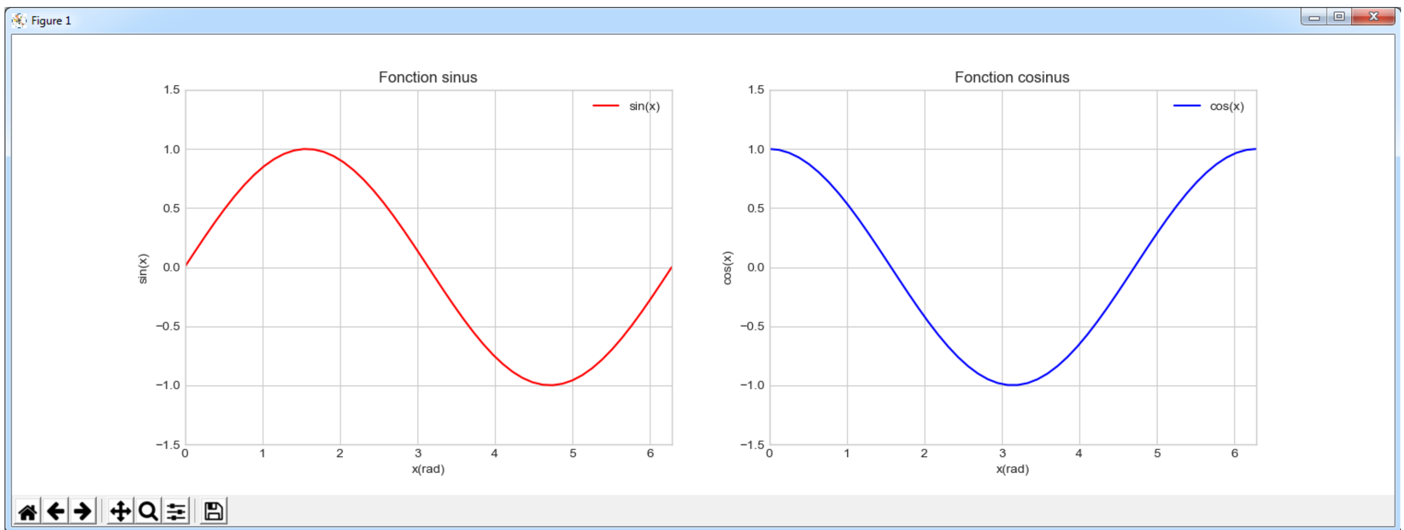
  axe.set_ylabel("labely")

Exemples :

. 2 graphes sur 1 ligne / 2 colonnes :



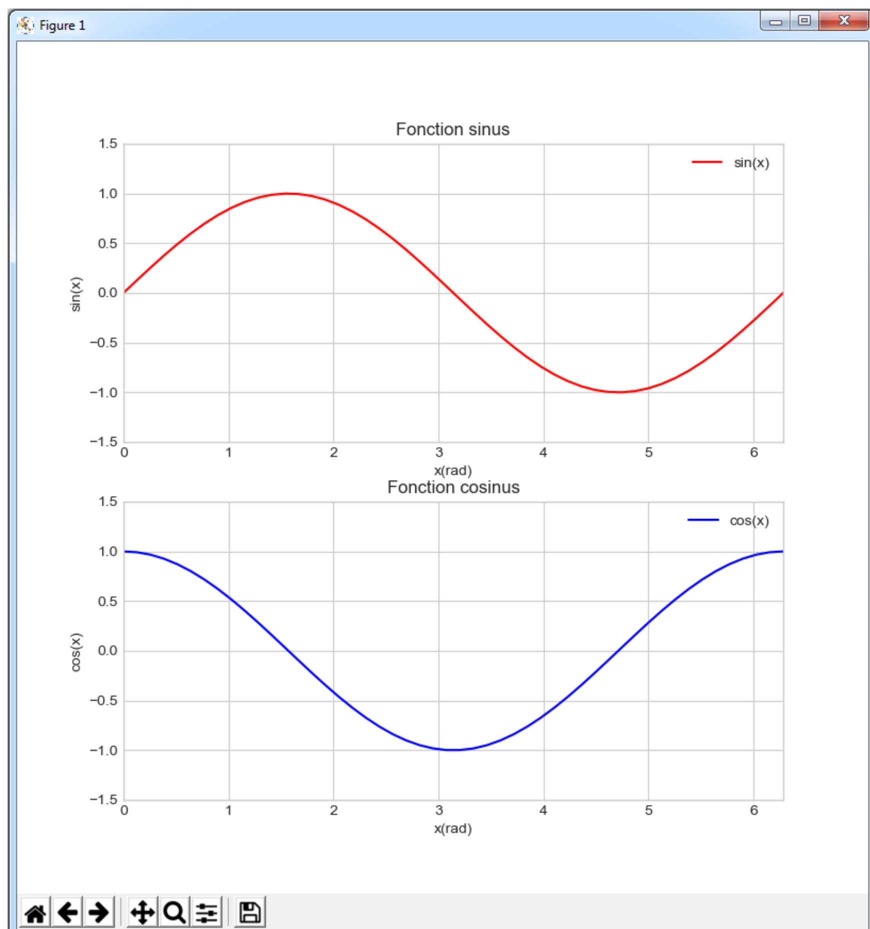

```python
import matplotlib.pyplot as plt
import numpy as np

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)
```

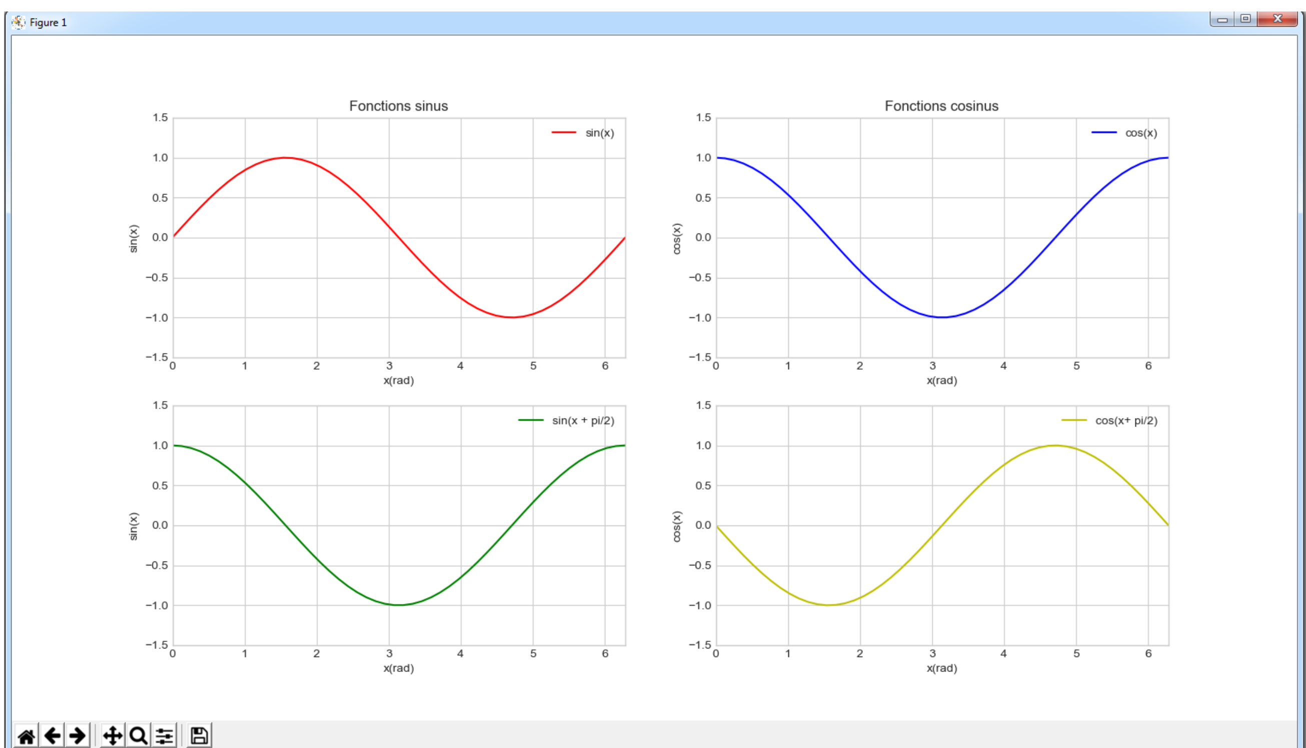


. 4 graphes sur 2 lignes / 2 colonnes :

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/

3.4.5 La programmation orientée objet (classes et objets)

La programmation orientée objet (ou POO en abrégé) est une autre manière de construire et d’organiser son code.

Python est un langage orienté objet, ce qui signifie que le langage tout entier est construit autour de la notion d’objets.

Par exemple, les types **str**, **int**, **list**,... sont des objets, les fonctions sont des objets, etc…

La programmation orientée objet repose sur le concept d’objets qui sont des blocs de code qui possède un ensemble de variables (appelées **attributs** en python) et de fonctions qui leur sont propres (appelées **méthodes** en python).

Les objets sont créés à partir de ”modèles” appelés **classes** qui sont également des ensembles de codes qui contiennent des variables et des fonctions. Les objets créés possèdent alors un même ensemble d’attributs et de méthodes que les classes, les attributs étant des variables accessibles depuis toute méthode de la classe où elles sont définies.

On dit qu’un objet est une instance d’une classe. On peut créer autant d'objets que l'on désire avec une classe.

. Les classes

Une classe est équivalente à un type de données et créer une nouvelle classe en Python correspond à définir un nouveau type de données.

Sans le savoir, nous avons déjà vu des classes :

- le type de donnée **str** est une classe dont l’objet **ch = ”chaine ”** est une instance et par exemple, la fonction **upper()** est une de ses méthodes,

- le type de donnée **list** est une classe dont l’objet **liste=[1,2,3,4]** est une instance et par exemple, la fonction **sort()** est une de ses méthodes,

- …

Pour créer une nouvelle classe Python, on utilise le mot clef **class** suivi du nom de la classe.

Nous allons créer une classe nommée **inventaire** à partir du programme d’inventaire déjà vu en tant qu’exemple d’application de la manipulation des fichiers.

```
class inventaire:
    def __init__(self):
        self.invent={}
        self.inventpath=""
        self.finprog=False
```

Les instructions ci-dessus crée une classe nommée **inventaire** dont les attributs sont :

. un dictionnaire nommé **self.invent** initialement vide,

. une chaine de caractères nommée **self.inventpath** correspondant au chemin de l’inventaire initialement vide,

. une variable booléenne nommée **self.finprog** initialisée en **False** afin de pouvoir mettre fin à l'exécution du programme.

La méthode **_init_(self)** dans laquelle les attributs sont définis est appelée lors de la création d'un objet à partir de la classe. Cette méthode est nommée un **constructeur**.

Pour créer un objet à partir de la classe **inventaire**, il suffira dans le programme principal d'appeler la classe à l'aide de l'instruction : **nom_objet = inventaire()**

La méthode **_init_(self)** est alors appelée et les variables du constructeur sont attribuées à l'objet créé.

La méthode prend en paramètre la variable **self**. Cette variable représente l'objet lui-même (d'où le nom **self**…) et c'est pourquoi les attributs sont de type : **self.variable** (car ce sont les variables de l'objet !).

De cette façon, il n'y a plus besoin de **variables globales** dans des fonctions du programme ayant besoin de modifier des variables externes à la fonction (les attributs **self.variable** étant des variables accessibles depuis toute méthode de la classe).

Après la définition des attributs, il faut définir les méthodes de la classe **inventaire** :

- Méthode pour ouvrir un inventaire :

```
def OuvreInventaire(self):
    self.InventPath=input("Indiquez le nom de l'inventaire:")
    self.invent = {}
    try:
        with open(self.InventPath, 'r') as fichier:
            for line in fichier:
                listline=line.split(";")
                self.invent[listline[0]]=listline[1].strip()

    except Exception as message:
        print(message)
```

Cette méthode demande à l'utilisateur le chemin d'un fichier d'inventaire (**self.InventPath**) et tente de l'ouvrir. Un message est affiché si le fichier n'existe pas, sinon l'inventaire est chargé dans le dictionnaire **self.invent**.

- Méthode pour ajouter un élément à l'inventaire :

```
def AjoutMatos(self):
    Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
    Quant=input("saisissez la quantité de ce matériel:")
    self.invent[Matos]=Quant
```

Cette méthode demande à l'utilisateur de saisir les données (nom et quantité) de l'élément à ajouter à l'inventaire et l'ajoute au dictionnaire **self.invent**.

- Méthode pour effacer un élément de l'inventaire :

```
    def EffaceMatos(self):
        element=input("saisissez l'élément à supprimer dans l'inventaire:")
        try:
            del self.invent[element]
        except:
            print("L'élément que vous voulez supprimer n'existe pas!")
```

Cette méthode demande à l'utilisateur de saisir le nom de l'élément à supprimer de l'inventaire et le supprime du dictionnaire **self.invent**.

- Méthode pour afficher l'inventaire :

```
    def ReadInventaire(self):
        for cle, valeur in self.invent.items():
            print("{} : {}".format(cle, valeur))
```

Cette méthode affiche la liste des éléments de l'inventaire **self.invent** dans la fenêtre Python shell.

- Méthode pour sauvegarder l'inventaire :

```
    def SaveInventaire(self):
        self.InventPath=input("Indiquez le nom de sauvegarde de l'inventaire:")
        with open(self.InventPath, 'w') as fichier:
            for cle, valeur in self.invent.items():
                fichier.write("{};{}".format(cle, valeur))
                fichier.write("\n")
```

Cette méthode demande à l'utilisateur de saisir le chemin d'enregistrement de l'inventaire (**self.InventPath**). L'inventaire est parcouru et sauvegarder dans le fichier ouvert.

- Méthode pour déterminer l'action à exécuter :

```
    def ChoixAction(self):
        print("////////////// INVENTAIRE MATERIEL //////////////")
        print("appuyer sur O pour ouvrir un inventaire:")
        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 E pour enregistrer l'inventaire:")
        print("appuyer sur Q pour quitter:")
        print("///////////////////////////////////////////////////")
        print(" ")

        while self.finprog == False:
            choix = " "
            while choix.upper()!="A" or "S" or "Q" or "P" or "O" or "E":
                choix=input()
```

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

Cette méthode affiche la liste des actions disponibles et demande en boucle à l'utilisateur de faire un choix.

Les attributs et les méthodes de la classe **inventaire** ont maintenant tous été définis. Le programme va créer un objet nommé **mon_inventaire** qui est une instance de la classe **inventaire** :

mon_inventaire=inventaire()

Puis on appelle la méthode **ActionChoix()** de l'objet créé pour lancer la boucle des actions :

mon_inventaire.ChoixAction()

Résultats dans le fenêtre Python Shell :

```
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\inventclass.py ===========
////////////// INVENTAIRE MATERIEL //////////////
appuyer sur O pour ouvrir un inventaire:
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 E pour enregistrer l'inventaire:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
a
saisissez le type de matériel à ajouter à l'inventaire:eprouvette
saisissez la quantité de ce matériel:5
e
Indiquez le nom de sauvegarde de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
eprouvette : 5
o
Indiquez le nom de l'inventaire:invent
v
bécher : 10
eprouvette : 15
o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
eprouvette : 5
q
Fin du programme
```

Voici Le programme complet avec la classe **inventaire** qui pourra bien-sûr se situé dans un module à part qui le cas échéant devra être importé avant utilisation :

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

File Edit Format Run Options Window Help

```
class inventaire:
    def __init__(self):
        self.invent={}
        self.inventpath=""
        self.finprog=False

    def OuvreInventaire(self):
        self.InventPath=input("Indiquez le nom de l'inventaire:")
        self.invent = {}
        try:
            with open(self.InventPath, 'r') as fichier:
                for line in fichier:
                    listline=line.split(";")
                    self.invent[listline[0]]=listline[1].strip()

        except Exception as message:
            print(message)

    def AjoutMatos(self):
        Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
        Quant=input("saisissez la quantité de ce matériel:")
        self.invent[Matos]=Quant

    def EffaceMatos(self):
        element=input("saisissez l'élément à supprimer dans l'inventaire:")
        try:
            del self.invent[element]
        except:
            print("L'élément que vous voulez supprimer n'existe pas!")

    def ReadInventaire(self):
        for cle, valeur in self.invent.items():
            print("{} : {}".format(cle, valeur))

    def SaveInventaire(self):
        self.InventPath=input("Indiquez le nom de sauvegarde de l'inventaire:")
        with open(self.InventPath, 'w') as fichier:
            for cle, valeur in self.invent.items():
                fichier.write("{};{}".format(cle, valeur))
                fichier.write("\n")
    def ChoixAction(self):
        print("////////////// INVENTAIRE MATERIEL //////////////")
        print("appuyer sur O pour ouvrir un inventaire:")
        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 E pour enregistrer l'inventaire:")
        print("appuyer sur Q pour quitter:")
        print("////////////////////////////////////////////////")
        print(" ")

        while self.finprog == False:
            choix = " "
            while choix.upper()!="A" or "S" or "Q" or "P" or "O" or "E":
                choix=input()

                # Action en fonction de l'entrée clavier:
                if choix.upper()=="O": self.OuvreInventaire()
                if choix.upper() == "A": self.AjoutMatos()
                if choix.upper() == "V": self.ReadInventaire()
                if choix.upper() == "S": self.EffaceMatos()
                if choix.upper()=="E": self.SaveInventaire()
                if choix.upper() == "Q":
                    print("Fin du programme")
                    self.finprog = True
                    break

mon_inventaire=inventaire()
mon_inventaire.ChoixAction()
```

Tout ce qui a été vu concernant la programmation en Python ne représente que des bases, mais cela va nous permettre d'aborder le thème suivant à savoir la communication entre un Arduino Uno et un programme Python à des fins de contrôle et d'exploitation de données.