

NSI 1ere - Programmation

présentation de Thonny

Thonny

Thonny est un *environnement de développement intégré* (IDE en anglais pour *Integrated Development Environment*) qui a été pensé pour les programmeurs débutant en Python.

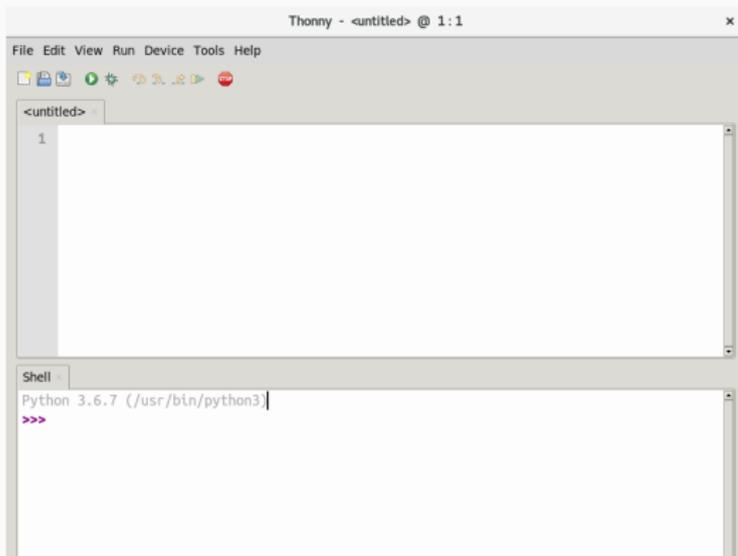
- Site officiel Thonny
- Wiki Wiki

Remarque distinguer **Python** qui est un *langage* et **Thonny** qui est un *environnement intégré de programmation en Python*.

Utilisation de base

Lorsqu'on lance Thonny la première fois, on découvre une fenêtre avec sa barre de menus usuels, une rangée de boutons, et deux panneaux correspondant à

- l'éditeur (onglet nommé <untitled>)
- l'interpréteur (onglet nommé Shell)



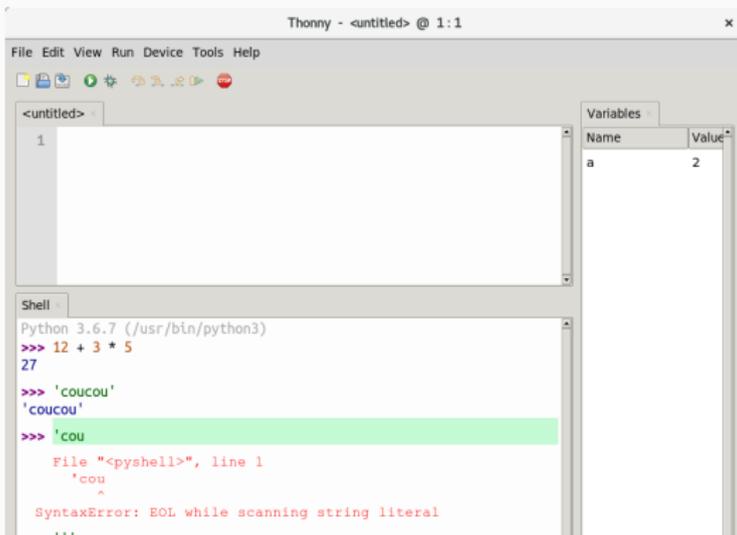
Le *shell* est la zone dans laquelle l'utilisateur interagit/dialogue avec l'interpréteur Python.

L'*invite de commande* (ou prompt) `>>>` attend une instruction. Une instruction doit être écrite sur une seule ligne sauf si

- elle est parenthésée (liste, tuples, dictionnaires, chaînes de caractères avec triple délimiteur)
- elle est composée : instructions conditionnelles, itérations

L'explorateur de variables

- faire apparaître une vue sur les variables définies par le programmeur dans la session courante : menu View/Variables
- un nouveau panneau (onglet Variables) apparaît sur la droite qui présente les variables en donnant leur nom et leur valeur
- dans le shell ajouter une définition d'une nouvelle variable, puis modifier sa valeur. observer dans la vue sur les variables



Thonny - <untitled> @ 1:1

File Edit View Run Device Tools Help

<untitled>

```
1
```

Variables

Name	Value
a	2

Shell

```
Python 3.6.7 (/usr/bin/python3)
>>> 12 + 3 * 5
27
>>> 'coucou'
'coucou'
>>> 'cou
File "<pyshell>", line 1
'cou
^
SyntaxError: EOL while scanning string literal
```

L'éditeur permet la rédaction de *scripts* (fichiers contenant du code Python). Ces scripts peuvent être exécutés et utilisés dans le shell

Avantages de l'éditeur

- effectue une sauvegarde à chaque exécution (demande un nom de fichier si 1ère exécution) => toute modification est sauvegardée
- coloration syntaxique :
 - des mots clés du langage (def, if, for, while, True, False ...)
 - des constantes littérales (couleurs distinctes pour nombres et chaînes de caractères)
 - coloration des régions marqués par un délimiteur ouvert mais non fermé (chaînes de caractères, listes, tuples, dictionnaires, ...)

Avantages de l'éditeur

- indentation automatique lorsque nécessaire
- complétion automatique avec la touche TAB => favorise l'utilisation de noms longs pour les paresseux
- visualisation portée des variables

Exécution d'un script, plusieurs possibilités

- via le menu Run/Run current script
- via le bouton flèche verte
- via la touche F5

Si le script vient d'être créé (onglet nommé <untitled>), boîte de dialogue pour demander un nom de script.

NB lors de l'exécution d'un script, le répertoire de travail est celui dans lequel est sauvegardé le script. (Important si besoin d'accéder à des fichiers)

Remarque : si un script ne contient aucune instruction d'affichage (`print`), alors son exécution ne se traduit que par la mention `%Run nom_script.py` dans le shell et rien d'autre.

=> un script peut

- ne contenir que des définitions de variables, fonctions, ... et on utilise ces définitions dans le shell
- contenir des instructions d'affichage (ou d'input) qui seront immédiatement exécutées et laisseront des traces dans le shell (éviter leur abus en phase de développement de fcts)

Illustration de ces points avec suite Syracuse

cf fichier `demo_syracuse.py`

- **demo_00.py** commencer par écrire la première fonction `syracuse` sans docstring. Profiter de l'ouverture de la parenthèse des paramètres pour souligner la coloration syntaxique (en gris).
- exécuter => il faut donner un nom au script (qui change l'intitulé de l'onglet) => il ne se passe rien dans le shell.

ATTENTION pour la suite (docstring) il ne faut pas nommer le script du même nom qu'une des fonctions.

Noter que dans la vue sur les variables, les variables précédemment définies ont disparues, et un seul nom est défini :

```
syracuse de valeur <function syracuse at ...>
```

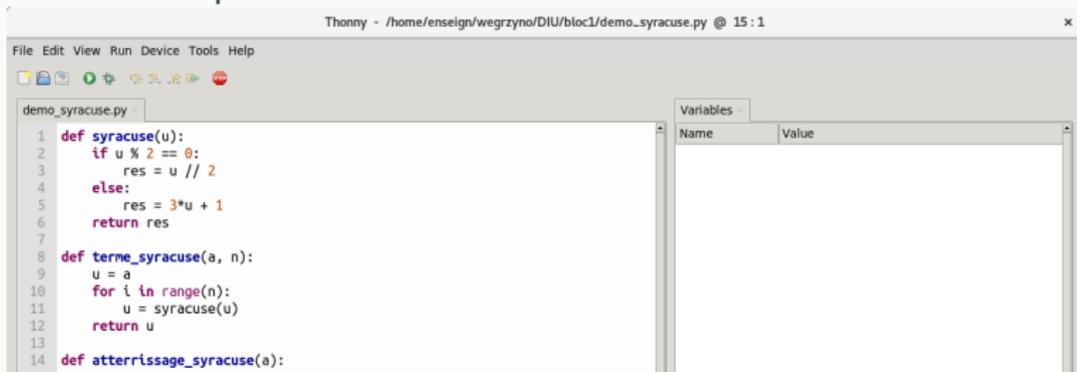
=> on travaille toujours dans un environnement de variables «propre» : celui des définitions du script plus éventuellement

- **demo_01.py** dans le shell utiliser la fonction `syracuse`
- dans l'éditeur écrire le deuxième fonction `terme_syracuse` sans docstring. Profiter de la complétion pour écrire l'appel à `syracuse`.
- observer le surlignement des identificateurs identiques en plaçant le curseur sur l'un d'eux (par exemple `syracuse`, `u`, `res`) => portée des variables
- exécuter (pas besoin de redonner un nom au script) => il ne se passe rien
- Observer la vue sur les variables

- **demo_02.py** dans le shell utiliser la fonction `terme_syracuse` construire une liste de valeurs pour $n=10$ et a compris entre 1 et 10
- écrire la troisième fonction `atterrissage_syracuse` sans docstring.
- exécuter une nouvelle fois
- essayer quelques appels à cette dernière fonction avec des termes initiaux strictement positifs

demo_02.py arrêter une boucle infinie

- essayer avec 0 => calcul infini => nécessité d'interrompre le calcul => deux solutions
 - usage du bouton panneau stop (en haut en rouge)
 - usage de la combinaison Ctrl+C au clavier les issues ne sont pas identiques :
 - le premier redémarre un nouvel interpréteur (et donc les variables sont perdues)
 - le second interrompt l'exécution de la commande en cours, et laisse intact l'environnement tel qu'il était au moment de l'interruption.



The screenshot shows the Thonny Python IDE interface. The title bar reads "Thonny - /home/enseign/wegrzyno/DIU/bloc1/demo_syracuse.py @ 15 : 1". The menu bar includes "File Edit View Run Device Tools Help". The toolbar contains icons for file operations, running, and stopping. The code editor displays the following Python code:

```
1 def syracuse(u):
2     if u % 2 == 0:
3         res = u // 2
4     else:
5         res = 3*u + 1
6     return res
7
8 def terme_syracuse(a, n):
9     u = a
10    for i in range(n):
11        u = syracuse(u)
12    return u
13
14 def atterrissage_syracuse(a):
```

On the right side, there is a "Variables" panel with a table structure:

Name	Value
------	-------

- aller dans le menu View/Outline : un nouvel onglet Outline apparait dans le panneau à côté des variables
- cliquer sur l'un ou l'autre des items qui y figurent permet de naviguer dans l'éditeur

Localiser une erreur

- dans le shell taper la commande `syracuse('3')` => cela se passe mal : plusieurs lignes rouges sont écrites ! Elles se lisent de bas en haut.
- dernière ligne : une exception est déclenchée portant le nom de `TypeError`
- ligne du dessus la ligne de code ayant déclenché l'exception
- ligne du dessus lien vers le fichier et la ligne de ce fichier contenant ce code
- cliquer sur le lien amène à la ligne de code responsable surlignée
- et dans la vue sur les variables, on visualise les variables locales et globales au moment du déclenchement de l'exception.

The screenshot shows the Thonny IDE interface. The main window displays a Python script named `demo_syracuse.py` with the following code:

```
1 def syracuse(u):
2     if u % 2 == 0:
3         res = u // 2
4     else:
5         res = 3*u + 1
6     return res
```

The line `if u % 2 == 0:` is highlighted in red, indicating an exception. The Variables window is open, showing the following table:

Name	Value
LOCALS	
u	'3'
GLOBALS	

The Variables window also includes a "Back to __main__" button.

- partir de la fonction `help` dans le shell sur quelques fcts prédéfinies (`abs`, `len`, `print`)
- essayer `help` avec les fcts du script `syracuse` => pas grand chose

- **demo_03.py** on ajoute une docstring dans syracuse

```
def syracuse(u):  
    '''  
    renvoie le terme suivant u d'une suite de Syracuse  
    '''
```

une *doctring* de fonction est une chaîne de caractères placée immédiatement après l'en-tête, qui s'étend généralement sur plusieurs lignes et donc qui est délimité par un triple ' ou un triple ".

- exécuter et utiliser la fonction `help` sur `syracuse`

- **demo04.py** on complète la doctring avec des informations de type des paramètres et de valeur renvoyée, en mentionnant les contraintes (ou conditions) d'utilisation (ici aucune) et en donnant quelques exemples (préparation aux doctests vus à la session prochaine)

```
def syracuse(u):
```

```
    '''
```

```
    retourne le terme suivant u d'une suite de Syracuse
```

```
    @param u: (int) entier quelconque
```

```
    @return: (int)
```

```
    >>> syracuse(0)
```

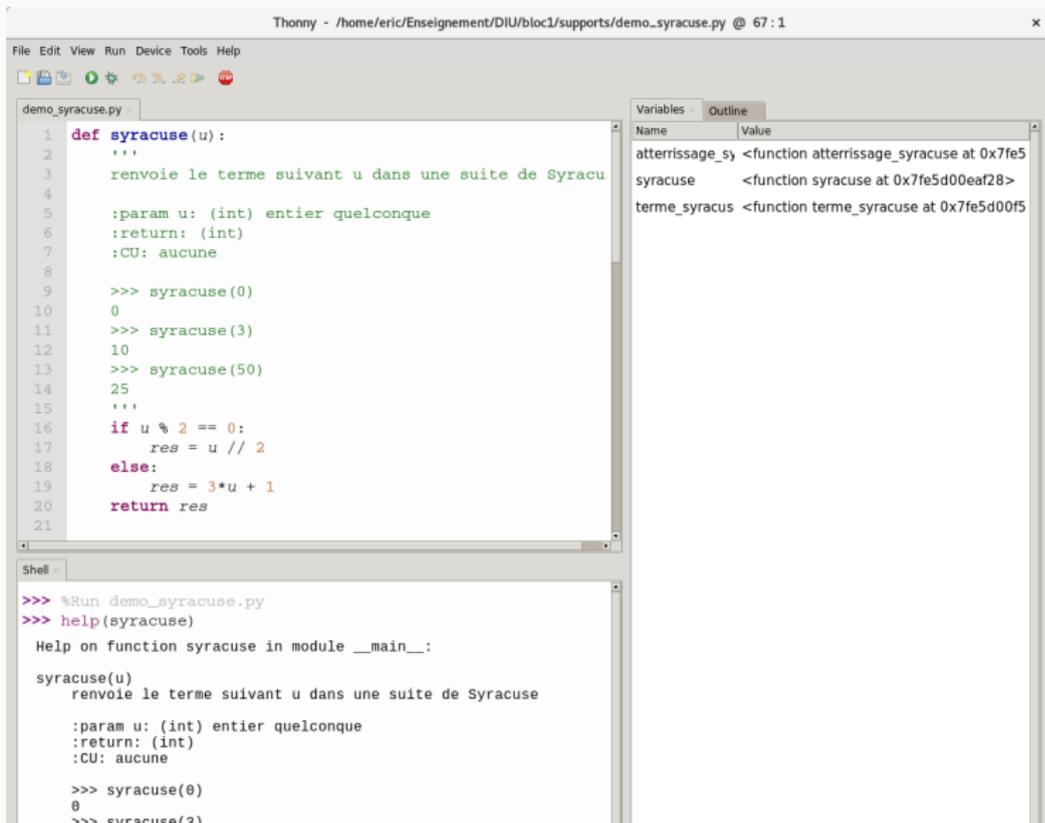
```
    0
```

```
    >>> syracuse(3)
```

```
    10
```

```
    '''
```

- demo05.py faire de même pour les deux autres fonctions



The screenshot shows a Python IDE window titled "Thonny - /home/eric/Enseignement/DIU/bloc1/supports/demo_syracuse.py @ 67:1". The main editor displays the source code for the 'syracuse' function, which calculates the next term in a Syracuse sequence. The function includes a docstring, a parameter description, and a loop that either divides the number by 2 if it's even or multiplies it by 3 and adds 1 if it's odd. Below the code, a shell window shows the execution of the script and the help command for the function.

```
1 def syracuse(u):
2     '''
3     renvoie le terme suivant u dans une suite de Syracu
4
5     :param u: (int) entier quelconque
6     :return: (int)
7     :CU: aucune
8
9     >>> syracuse(0)
10    0
11    >>> syracuse(3)
12    10
13    >>> syracuse(50)
14    25
15    '''
16    if u % 2 == 0:
17        res = u // 2
18    else:
19        res = 3*u + 1
20    return res
21
```

Shell

```
>>> %Run demo_syracuse.py
>>> help(syracuse)
Help on function syracuse in module __main__:
syracuse(u)
    renvoie le terme suivant u dans une suite de Syracuse
    :param u: (int) entier quelconque
    :return: (int)
    :CU: aucune
>>> syracuse(0)
0
>>> syracuse(2)
```

Name	Value
atterrissage_sy	<function atterrissage_syracuse at 0x7fe5d00eaf28>
syracuse	<function syracuse at 0x7fe5d00eaf28>
terme_syracus	<function terme_syracuse at 0x7fe5d00f5>