NSI 1ere - Programmation

présentation de Thonny

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

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

Dialogue avec l'interpréteur

L'*invite de commande* (ou prompt) `>>>` attend une instruction. Une instuction 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

Facilités de dialogue

- utilisation historique via la flèche haut
- complétion automatique via la touche TAB (sera revue avec l'éditeur)
- possibilité de «nettoyer» le shell via clik droit option `clear`

L'éditeur

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

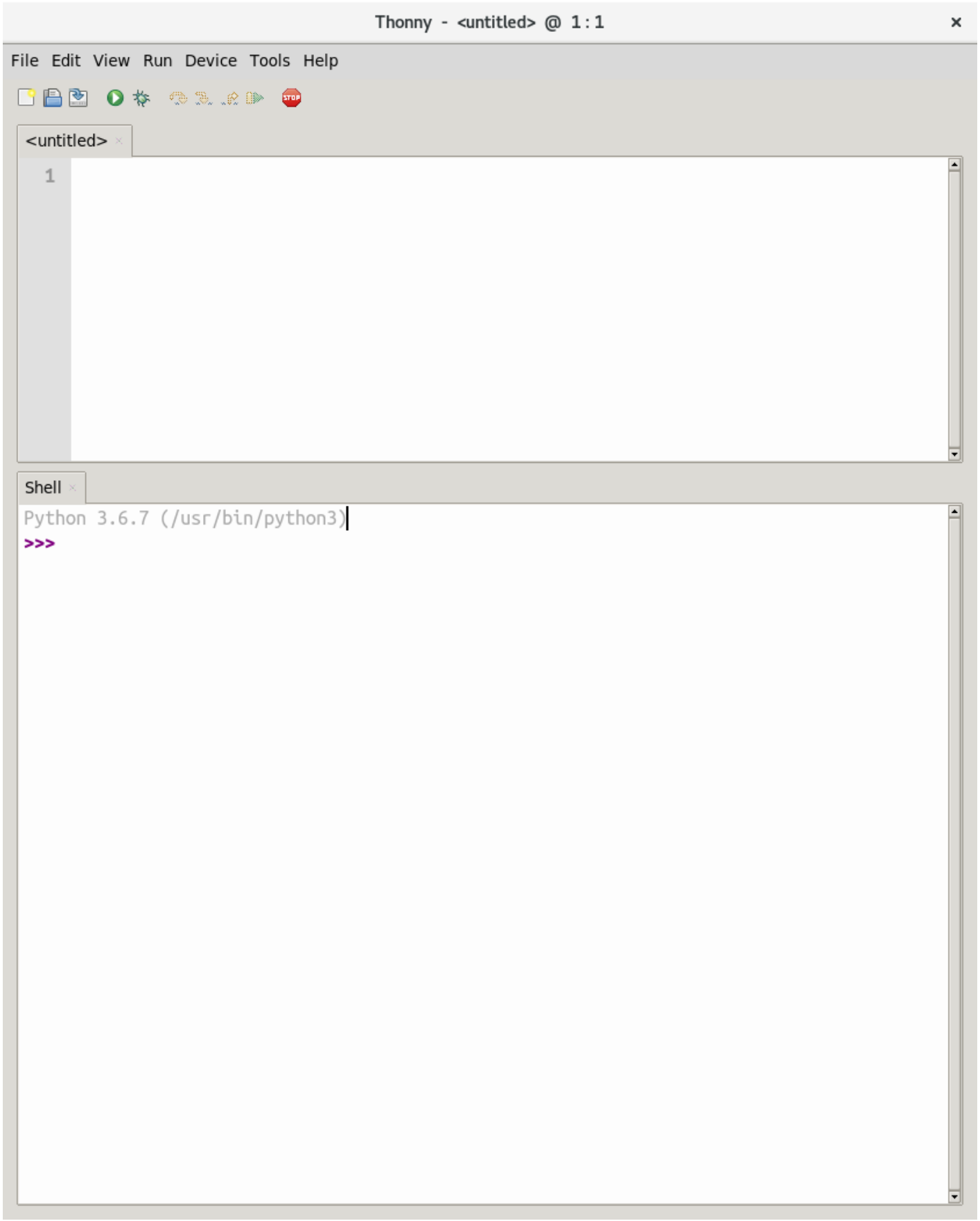


Figure 1: vue sur Thonny

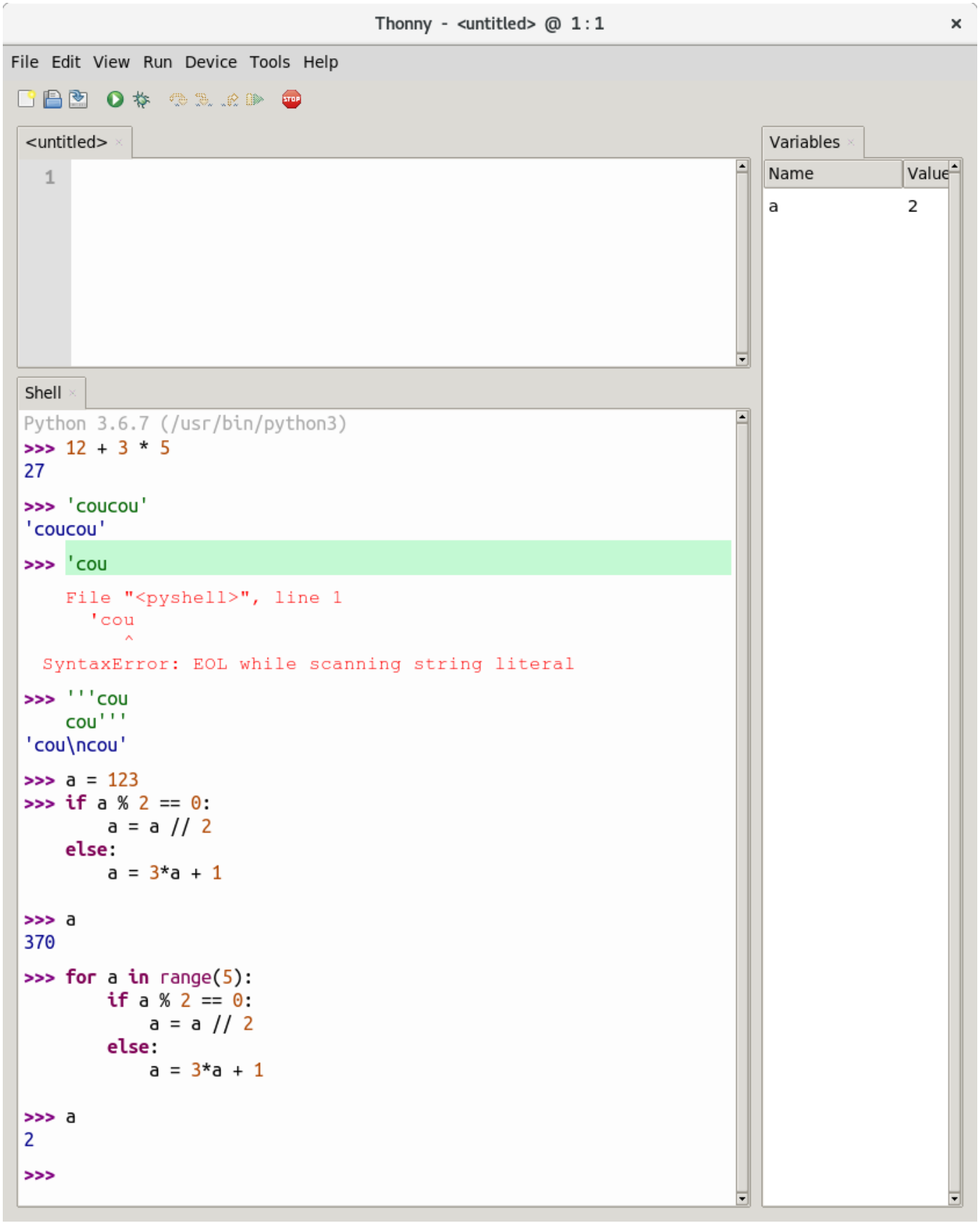


Figure 2: Thonny : utilisation du Shell

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 litérales (couleurs distinctes pour nombres et chaînes de caractères)
  - coloration des régions marqués par un délimiteur ouvert mais non fermé (chaines 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.

Répertoire de travail

NB lors de l'éxécution 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

=> 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

- **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 quelques variables définies dans la session en cours

demo_01.py

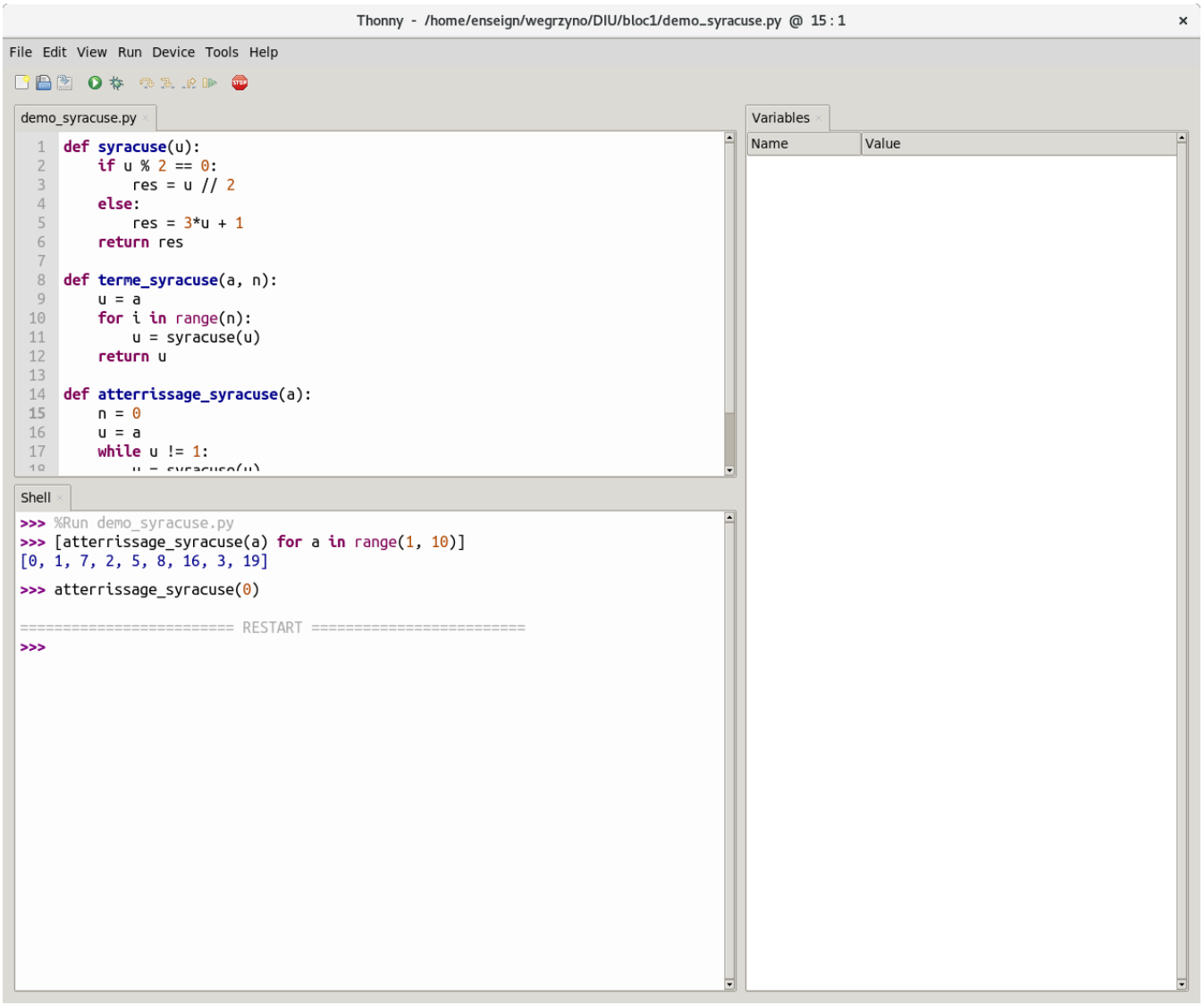
- **demo_01.py** dans le shell utiliser la fonction `syracuse`
- dans l'éditeur écrire le deuxième fonction `terme_syracuse` sans docstring. Profiter de la complétion pour écrire l'appel à `syracuse`.
- observer le surlignement des identificateurs identiques en plaçant le curseur sur l'un d'eux (par 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

- **demo_02.py** dans le shell utiliser la foncton `terme_syracuse` contruire 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.



Naviguer d'une fonction à l'autre

- 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 exeption 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.

Docstring

- 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

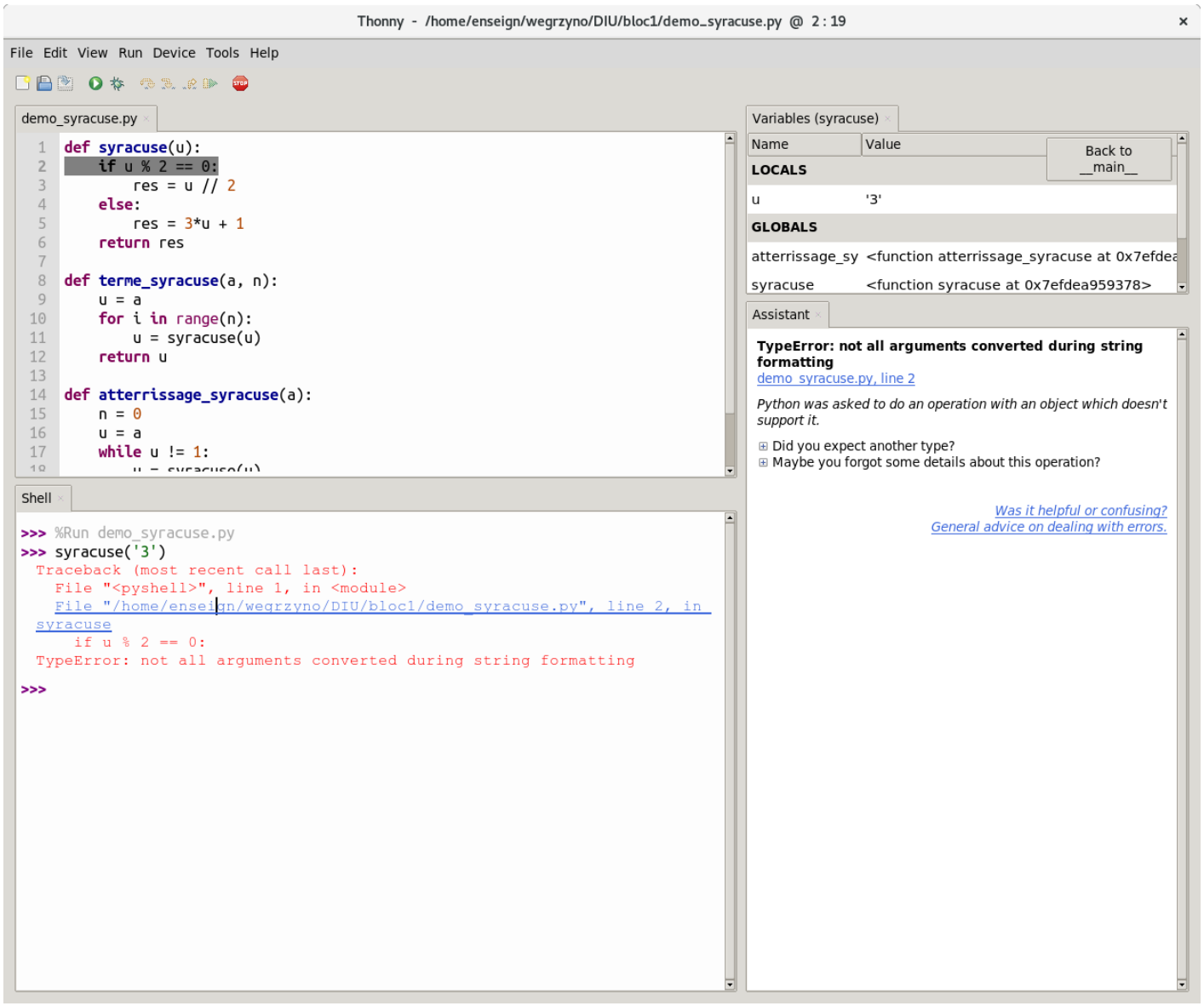


Figure 3: Thonny suivi d'un lien après déclenchement d'exception

demo_03.py

- **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`

demo_04.py

- **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):
    '''
    renvoie le terme suivant u d'une suite de Syracuse

    :param u: (int) entier quelconque
    :return: (int)
    :CU: aucune

    >>> syracuse(0)
    0
    >>> syracuse(3)
    10
    >>> syracuse(50)
    25
    '''
```

demo_05.py

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

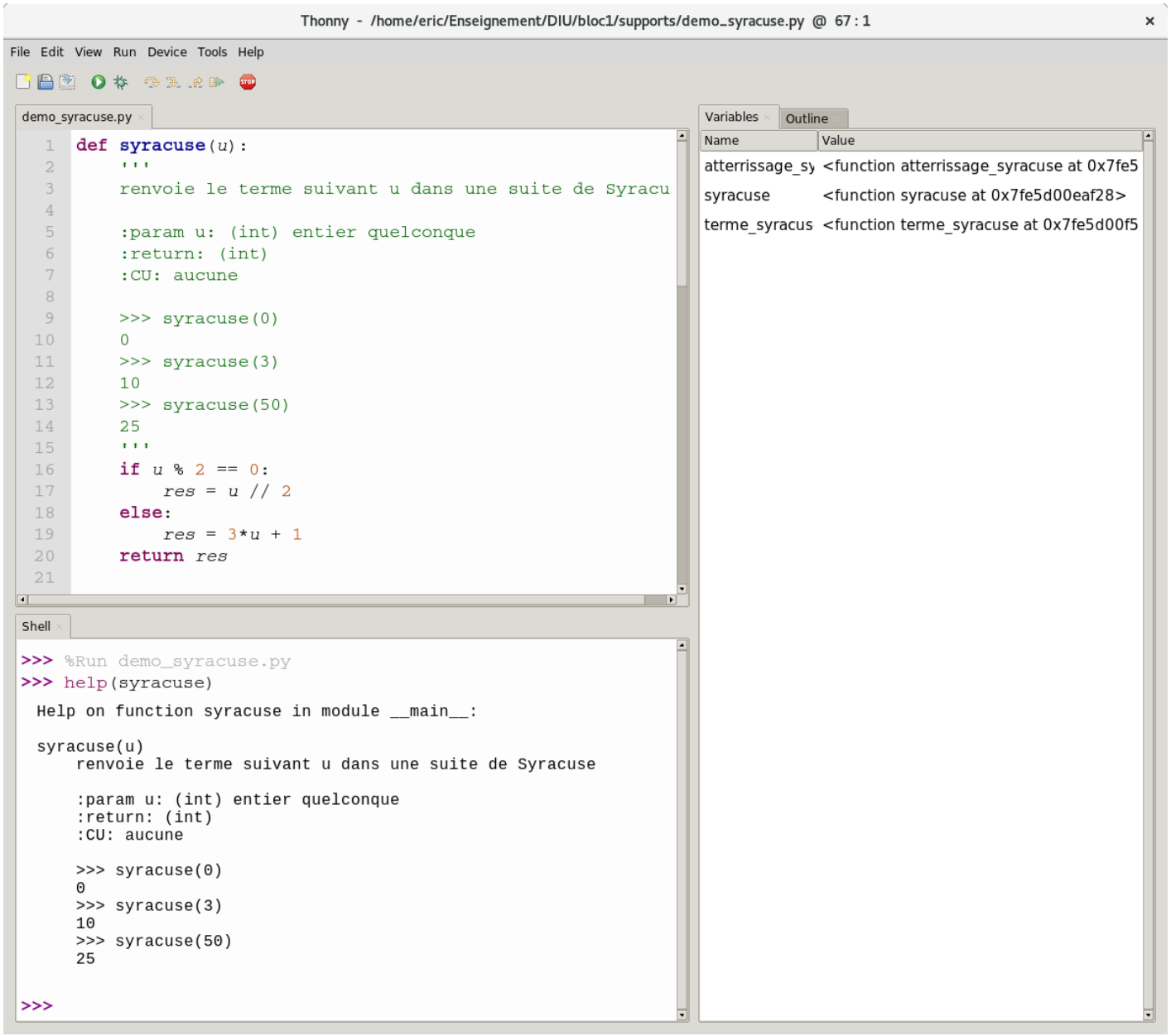


Figure 4: Docstring et fonction help