UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en Python

Cours 1. Variables, fonctions et conditions

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

Avant de me contacter

- La réponse est-elle sur moodle ?
- sur `https://upinfo.univ-cotedazur.fr/~obaldellon/python` ?
- sur le site de la licence info : `https://upinfo.univ-cotedazur.fr`
- Puis-je attendre la fin d'un cours en amphi ?

Écrire à l'enseignant

- Sur mon email `@univ-cotedazur.fr` (surtout pas sur moodle)
- avec votre adresse étudiante `@etu.univ-cotedazur.fr`
- Objet : clair et précis
- Rappel du nom du cours, de votre groupe
- Rappel de votre nom (signature suffisante)
- Si la question vous concerne (choix d'IP, emploi du temps, lettre de recommandation) : numéro d'étudiant !

Exemple

```
Objet : TP Python annulé à cause des grèves ?

Bonjour Monsieur,

Je suis étudiant du groupe A2 du cours de Python. Le TP
du mercredi 29 juillet à 10h est-il annulé à cause des
grèves ?

Cordialement,

Nicolas Bourbaki (22314159)
```

- Écrivez des messages brefs qui vont à l'essentiel !

Organisation

- CM, TD & TP
  - 9 séances de cours magistral.
  - 9 séances de 2h de TD
  - 9 séances de 2h de TP

Organisation

- **CM, TD & TP**
  - 9 séances de cours magistral.
  - 9 séances de 2h de TD
  - 9 séances de 2h de TP
- **Évaluation**
  - un partiel en cours de semestre $\approx 50\%$.
  - un examen terminal à la fin du semestre $\approx 50\%$

- **CM, TD & TP**
  - 9 séances de cours magistral.
  - 9 séances de 2h de TD
  - 9 séances de 2h de TP
- **Évaluation**
  - un partiel en cours de semestre ≈ 50%.
  - un examen terminal à la fin du semestre ≈ 50%
  - des interros surprises (peut-être… c'est une surprise)
  - un projet bonus et facultatif

- **CM, TD & TP**
  - 9 séances de cours magistral.
  - 9 séances de 2h de TD
  - 9 séances de 2h de TP
- **Évaluation**
  - un partiel en cours de semestre ≈ 50%.
  - un examen terminal à la fin du semestre ≈ 50%
  - des interros surprises (peut-être… c'est une surprise)
  - un projet bonus et facultatif
- **En plus de vos notes manuscrites, vous trouverez sur mon site web :**
  - Transparents des cours magistraux (avec ou sans animations).
    - Version avec animations pour relire et retravailler le cours chez soi.
    - Version sans animations pour retrouver rapidement une information.
  - Sujets et corrigés des derniers exercices de TP/TD.

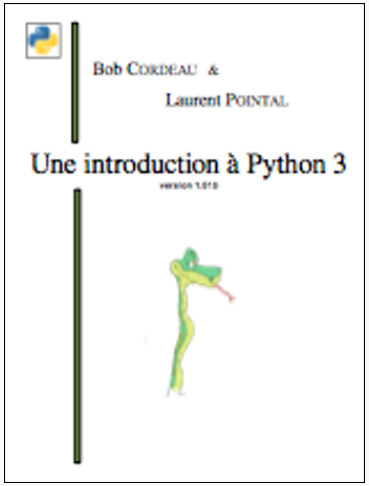
- **CM, TD & TP**
  - 9 séances de cours magistral.
  - 9 séances de 2h de TD
  - 9 séances de 2h de TP
- **Évaluation**
  - un partiel en cours de semestre ≈ 50%.
  - un examen terminal à la fin du semestre ≈ 50%
  - des interros surprises (peut-être… c'est une surprise)
  - un projet bonus et facultatif
- **En plus de vos notes manuscrites, vous trouverez sur mon site web :**
  - Transparents des cours magistraux (avec ou sans animations).
    - Version avec animations pour relire et retravailler le cours chez soi.
    - Version sans animations pour retrouver rapidement une information.
  - Sujets et corrigés des derniers exercices de TP/TD.
    - pour avoir les corrections des premiers exercices : venez en TD/TP !

Quelques références pour ce cours

- La documentation officielle Python : `http://docs.python.org/py3k`

- Livre gratuit en ligne

- Poly. IUT Orsay



- Livre de mon prédecesseur

- De très nombreuses autres références en ligne ou à la BU

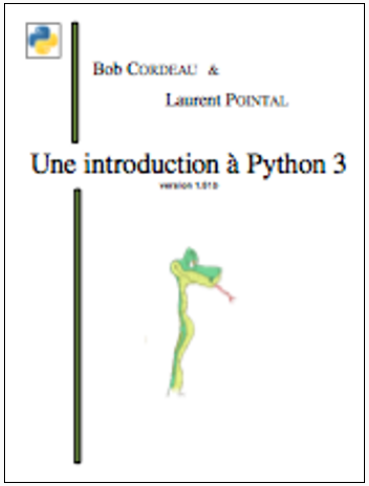
- En particulier le livre de Jean-Paul Roy (à l'origine de cette UE) est à la BU.

Quelques références pour ce cours

- La documentation officielle Python : `http://docs.python.org/py3k`

- Livre gratuit en ligne

- Poly. IUT Orsay



- Livre de mon prédecesseur

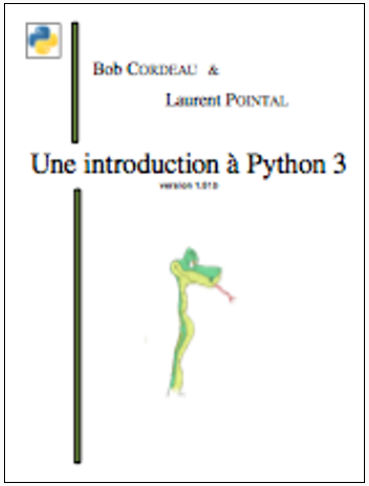
- De très nombreuses autres références en ligne ou à la BU
- En particulier le livre de Jean-Paul Roy (à l'origine de cette UE) est à la BU.
- Le mieux reste mon cours :)

Quelques références pour ce cours

- La documentation officielle Python : `http://docs.python.org/py3k`

- Livre gratuit en ligne

- Poly. IUT Orsay



- Livre de mon prédecesseur

- De très nombreuses autres références en ligne ou à la BU
- En particulier le livre de Jean-Paul Roy (à l'origine de cette UE) est à la BU.
- Le mieux reste mon cours :)
  - Pas dans l'absolue, mais pour préparer cette UE !

Algorithmique

- Comment résoudre un problème ?
- Branche des mathématiques (feuilles blanches et crayon)
- Ce problème peut ensuite être résolu par :
  - Un humain : chercher un mot dans le dictionnaire, poser une addition
  - Une machine : modifier une image, décider de votre orientation (parcoursup)

Algorithmique

- Comment résoudre un problème ?
- Branche des mathématiques (feuilles blanches et crayon)
- Ce problème peut ensuite être résolu par :
  - Un humain : chercher un mot dans le dictionnaire, poser une addition
  - Une machine : modifier une image, décider de votre orientation (parcoursup)

Programmation

- Donner des consignes (par exemple un algorithme) à une machine
- On utilise une langage artificiel (C, **Python**, OCaml, Java, etc)
- Tout ce que fait un ordinateur est décrit sous forme de texte.
  - Les fichiers contenant ces textes s'appelle le code source
- Exemples de tâche : lire une vidéo, faire un calcul, modifier une image

C'est un cours de programmation avec Python version 3.

- Créé par le Néerlandais Guido van Rossum en **1989**.
- **Langage de haut niveau** :
  - bas niveau : proche du fonctionnement de la machine
  - haut niveau : proche du raisonnement humain
- **Langage multi-paradigmes** : impératif, fonctionnel, orienté objets.
- **Nombreuses bibliothèques**.
- **Libre** : code source disponible et modifiable, gratuit.
- **Attention** Python 2 est un **autre langage**. Très proche, certes.

Objectif du cours

- Ce n'est pas un cours de Python !

Objectif du cours

- Ce n'est pas un cours de Python !
- C'est un cours de programmation impérative.
  - `while` `if`, fonction et variable
  - Tout le reste n'est que raccourcis.
  - Python n'est qu'un outils

Objectif du cours

- Ce n'est pas un cours de Python !
- C'est un cours de programmation impérative.
  - `while` `if`, fonction et variable
  - Tout le reste n'est que raccourcis.
  - Python n'est qu'un outils
- Apprendre à programmer ce n'est pas apprendre plein d'astuces
  - fonction prédéfinie « `max(liste)` »
  - méthode « `liste.sort()` »
  - mots-clé « `x in liste` »
  - construction syntaxique « `[f(x) for x in L]` »

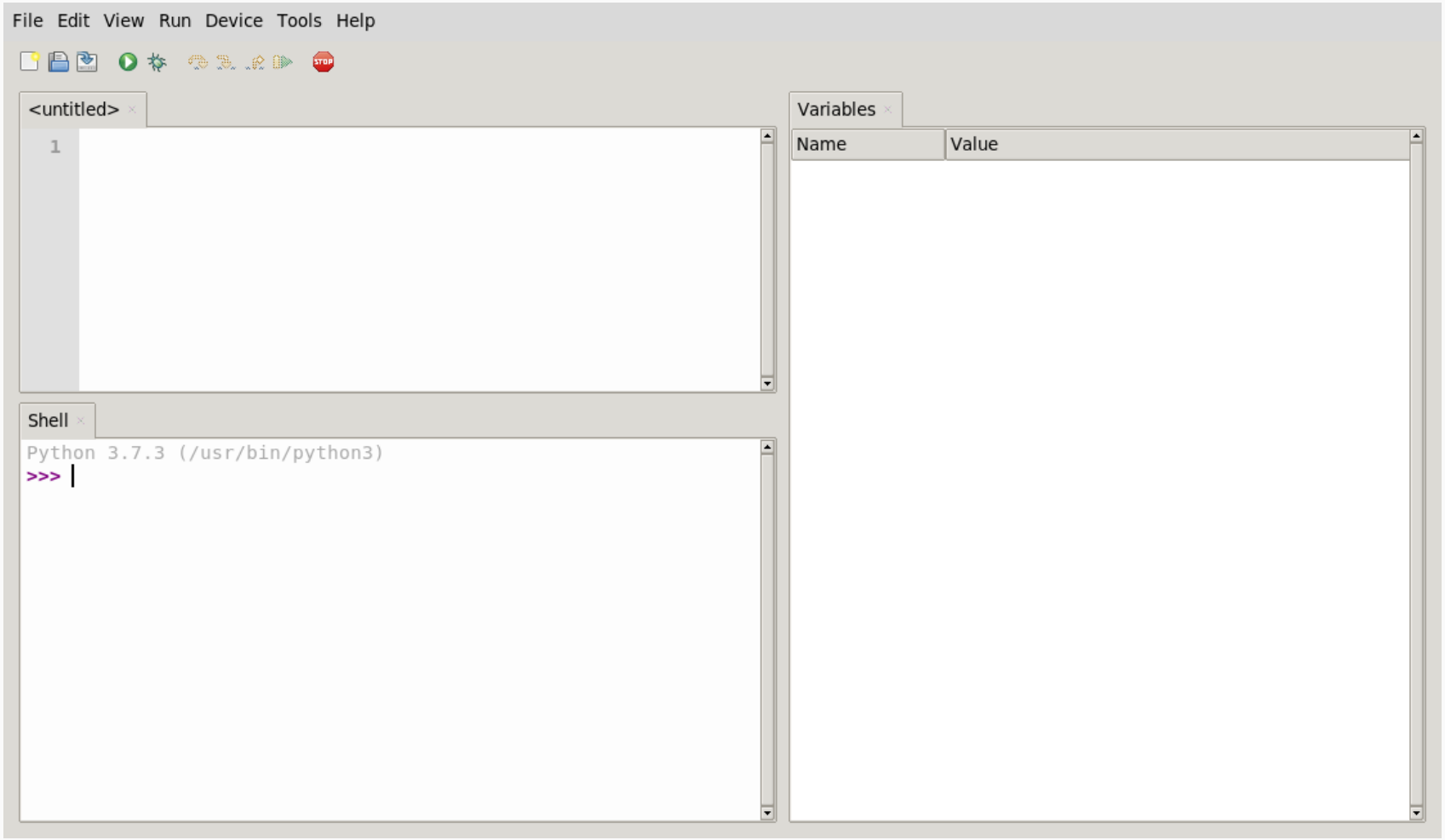
Objectif du cours

- Ce n'est pas un cours de Python !
- C'est un cours de programmation impérative.
  - `while` `if`, fonction et variable
  - Tout le reste n'est que raccourcis.
  - Python n'est qu'un outils
- Apprendre à programmer ce n'est pas apprendre plein d'astuces
  - fonction prédéfinie « `max(liste)` »
  - méthode « `liste.sort()` »
  - mots-clé « `x in liste` »
  - construction syntaxique « `[f(x) for x in L]` »
- Pourquoi se focaliser sur les bases :
  - Permet d'apprendre rapidement d'autres langages
  - Permet d'apprendre à résoudre des problèmes plus complexes
  - Permet de programmer plus efficacement
  - Permet de comprendre que l'informatique ce n'est pas magique…

Objectif du cours

- Ce n'est pas un cours de Python !
- C'est un cours de programmation impérative.
  - `while` `if`, fonction et variable
  - Tout le reste n'est que raccourcis.
  - Python n'est qu'un outils
- Apprendre à programmer ce n'est pas apprendre plein d'astuces
  - fonction prédéfinie « `max(liste)` »
  - méthode « `liste.sort()` »
  - mots-clé « `x in liste` »
  - construction syntaxique « `[f(x) for x in L]` »
- Pourquoi se focaliser sur les bases :
  - Permet d'apprendre rapidement d'autres langages
  - Permet d'apprendre à résoudre des problèmes plus complexes
  - Permet de programmer plus efficacement
  - Permet de comprendre que l'informatique ce n'est pas magique… c'est juste vachement chouette !

À installer au plus vite sur votre ordinateur via le site `https://thonny.org`



Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

Session interactive

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL

```
>>>
```

Session interactive

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
```

Session interactive

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL

```
>>> 1+1
2
>>>
```

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
```

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
12
>>>
```

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
12
>>> 2**10
```

Session interactive

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL

```
>>> 1+1
2
>>> 123//10
12
>>> 2**10
1024
```

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
12
>>> 2**10
1024
```

- Une opération mal utilisée peut provoquer une erreur.

SHELL
```
>>> 5//0 # Bouh, la vilaine division par zéro
```

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
12
>>> 2**10
1024
```

- Une opération mal utilisée peut provoquer une erreur.

SHELL
```
>>> 5//0 # Bouh, la vilaine division par zéro
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
12
>>> 2**10
1024
```

- Une opération mal utilisée peut provoquer une erreur.

SHELL
```
>>> 5//0 # Bouh, la vilaine division par zéro
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- Le code après # est un commentaire, non pris en compte par Python.

Python a une console (ou shell, ou toplevel) avec laquelle on peut interagir. On peut lui soumettre un calcul comme à une calculatrice.

SHELL
```
>>> 1+1
2
>>> 123//10
12
>>> 2**10
1024
```

- Une opération mal utilisée peut provoquer une erreur.

SHELL
```
>>> 5//0 # Bouh, la vilaine division par zéro
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError:  integer division or modulo by zero
```

- Le code après # est un commentaire, non pris en compte par Python.
- Lisez-bien les messages d'erreurs ! Il sont souvent assez clairs.

Opérations de base sur les entiers :

- Les opérations classiques : `+` `-` `*`
- Le quotient et le reste de la division euclidienne `//` et `%`
- La division décimale `/`
- `a**b` calcule la puissance a^b

Opérations de base sur les entiers :

- Les opérations classiques : `+` `-` `*`
- Le quotient et le reste de la division euclidienne `//` et `%`
- La division décimale `/`
- `a**b` calcule la puissance a^b

En mathématique, il y a deux opérations de divisions :

- La division décimale, 11÷2=5,5
- La division euclidienne (`//` et `%`), 11 divisé par 2 donne 5 reste 1

On définit la division euclidienne de `a` par `b` avec `a` et `b` entier et `b`$\neq 0$:

- `a//b` est le quotient et `a%b` est le reste (a = quotient × b + reste)

On définit la division euclidienne de `a` par `b` avec `a` et `b` entier et `b`$\neq 0$:

- `a//b` est le quotient et `a%b` est le reste (a = quotient × b + reste)
- `a == (a//b) * b + (a%b)`

On définit la division euclidienne de `a` par `b` avec `a` et `b` entier et `b`$\neq 0$:

- `a//b` est le quotient et `a%b` est le reste (a = quotient × b + reste)
- `a == (a//b) * b + (a%b)`
- `123=(123//10)*10 + (123%10) = 12*10 + 3`

On définit la division euclidienne de `a` par `b` avec `a` et `b` entier et $b \neq 0$:

- `a//b` est le quotient et `a%b` est le reste (a = quotient × b + reste)
- `a == (a//b) * b + (a%b)`
- `123=(123//10)*10 + (123%10) = 12*10 + 3`

Application du modulo : à connaître par cœur !

- `a%b` se lit a modulo `b`
- `a` est un multiple de `b` si et seulement si `a%b=0`.
- en particulier, `n` est pair si et seulement si `n%2=0`

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

SHELL
```
>>>
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

SHELL
```
>>> 15 * 60 + 43
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

SHELL
```
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

```shell
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 2
Sachant que 1024 minutes se sont écoulées depuis minuit, le cours de programmation impérative est-il terminé ?

```shell
>>>
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

```shell
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 2
Sachant que 1024 minutes se sont écoulées depuis minuit, le cours de programmation impérative est-il terminé ?

```shell
>>> 1024//60
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

```shell
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 2
Sachant que 1024 minutes se sont écoulées depuis minuit, le cours de programmation impérative est-il terminé ?

```shell
>>> 1024//60
17
>>>
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

```shell
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 2
Sachant que 1024 minutes se sont écoulées depuis minuit, le cours de programmation impérative est-il terminé ?

```shell
>>> 1024//60
17
>>> 1024%60
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

```shell
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 2
Sachant que 1024 minutes se sont écoulées depuis minuit, le cours de programmation impérative est-il terminé ?

```shell
>>> 1024//60
17
>>> 1024%60
4
```

Problème 1
Il est 15h43, combien de minutes se sont-elles écoulées depuis minuit ?

SHELL
```
>>> 15 * 60 + 43
943
```

15 heures et 43 minutes : 943 minutes

Problème 2
Sachant que 1024 minutes se sont écoulées depuis minuit, le cours de programmation impérative est-il terminé ?

SHELL
```
>>> 1024//60
17
>>> 1024%60
4
```

1024 minutes = 17 heures et 4 minutes

Opérations et priorités

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+` `-`	`*` `//` `%` `/`	`**`

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>>
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>>
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>> 1-2+5
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>> 1-2+5
4
>>>
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>> 1-2+5
4
>>> (1-2)+5
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>> 1-2+5
4
>>> (1-2)+5
4
>>>
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>> 1-2+5
4
>>> (1-2)+5
4
>>> 1-(2+5)
```

- Ordre de priorité des opérateurs.

moins prioritaire	plus prioritaire	très prioritaire
`+ -`	`* // % /`	`**`

- En cas de doute, on peut utiliser des parenthèses inutiles.

SHELL
```
>>> 5 - 8 + 4 * 2 ** 3 # Attention à la priorité !
29
>>> (5 - 8) + (4 * (2 ** 3))
29
```

- Remarque : associativité à gauche pour les opérations de même priorité.

SHELL
```
>>> 1-2+5
4
>>> (1-2)+5
4
>>> 1-(2+5)
-6
```

Taille des entiers

- La taille des entiers n'est limitée que par la mémoire de la machine.

SHELL
```
>>>
```

- La taille des entiers n'est limitée que par la mémoire de la machine.

SHELL
```
>>> 87412192161147838437159141 9 ** 10
```

- La taille des entiers n'est limitée que par la mémoire de la machine.

SHELL
```
>>> 87412192161147838437159141 9 ** 10
260447454985660585245180084757921014509252146181223003455270
044550605023694309556482234857745408080127776885578607664767
325495386096105286268494775055260671252317663749619931275002
342815836733596266389781703659821356427940431697254607426485
624871372306773584664397463801
```

- On dit (abusivement) que le calcul entier a une précision infinie ou que le calcul entier est exact.

- Cette propriété est intéressante pour les problèmes de cryptologie où on manipule de grands nombres entiers.

- La taille des entiers n'est limitée que par la mémoire de la machine.

SHELL
```
>>> 87412192161147838437159141 ** 10
260447454985660585245180084757921014509252146181223003455270
044550605023694309556482234857745408080127776885578607664767
325495386096105286268494775055260671252317663749619931275002
342815836733596266389781703659821356427940431697254607426485
624871372306773584664397463801
```

- On dit (abusivement) que le calcul entier a une précision infinie ou que le calcul entier est exact.

- Cette propriété est intéressante pour les problèmes de cryptologie où on manipule de grands nombres entiers.

- Ne marche pas avec les nombres avec virgule (flottant)

SHELL
```
>>> 87412192161147838437159141.9 ** 10
```

- La taille des entiers n'est limitée que par la mémoire de la machine.

SHELL
```
>>> 87412192161147838437159141 ** 10
260447454985660585245180084757921014509252146181223003455270
044550605023694309556482234857745408080127776885578607664767
325495386096105286268494775055260671252317663749619931275002
342815836733596266389781703659821356427940431697254607426485
624871372306773584664397463801
```

- On dit (abusivement) que le calcul entier a une précision infinie ou que le calcul entier est exact.

- Cette propriété est intéressante pour les problèmes de cryptologie où on manipule de grands nombres entiers.

- Ne marche pas avec les nombres avec virgule (flottant)

SHELL
```
>>> 87412192161147838437159141.9 ** 10
2.6044745498566053e+259
```

- Ce qui signifie $2,6 \times 10^{259}$

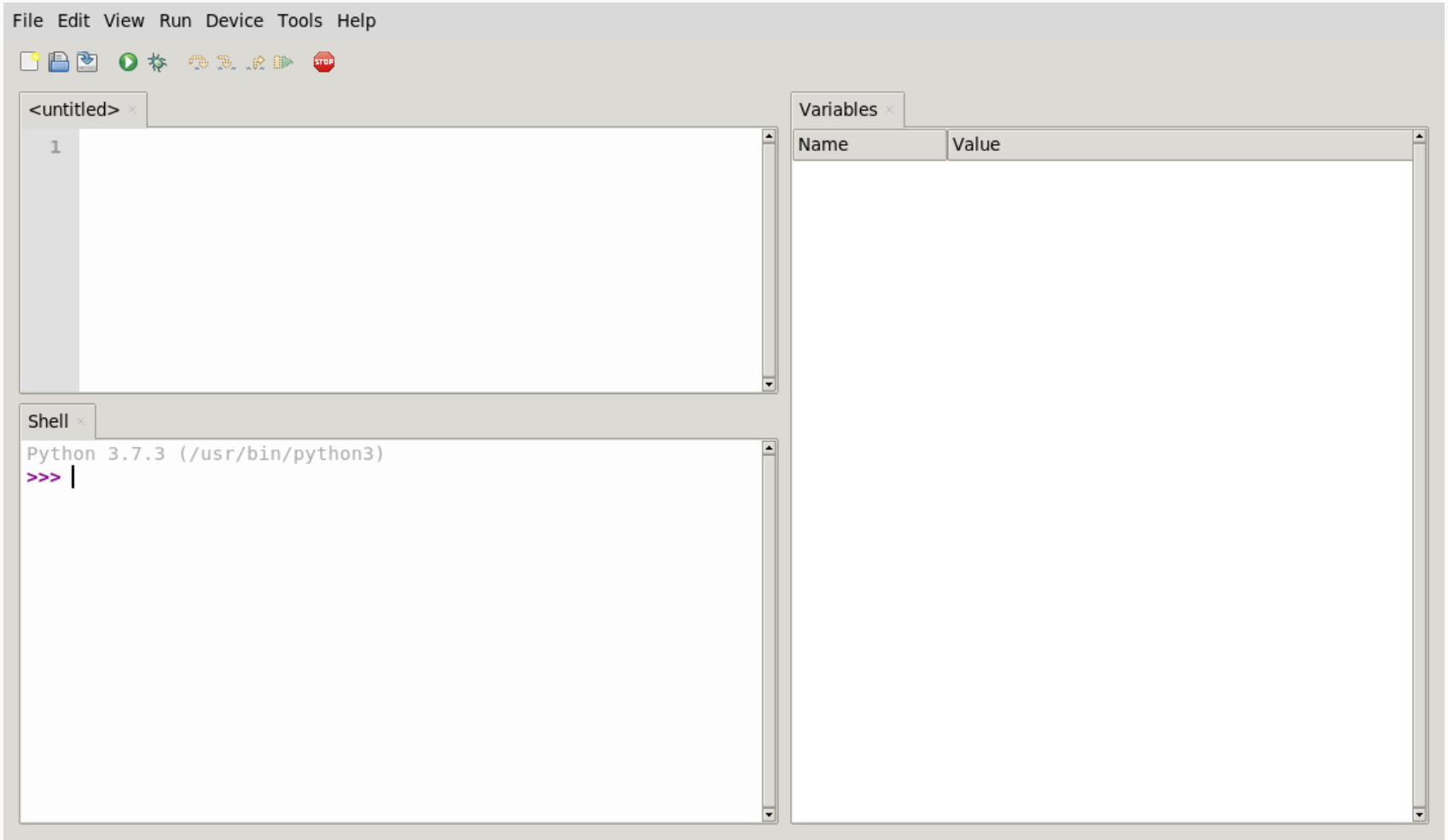
Essayez de répondre aux questions suivantes pour voir si vous avez compris. La réponse est donnée afin que vous puissiez vérifier tout seul.

- Calculer $\frac{2 \times 3}{2 + 1}$ Réponse :
- Calculer $2 \times \frac{3}{2} + 1$ Réponse :
- Calculer $2^{(3^4)}$ Réponse :

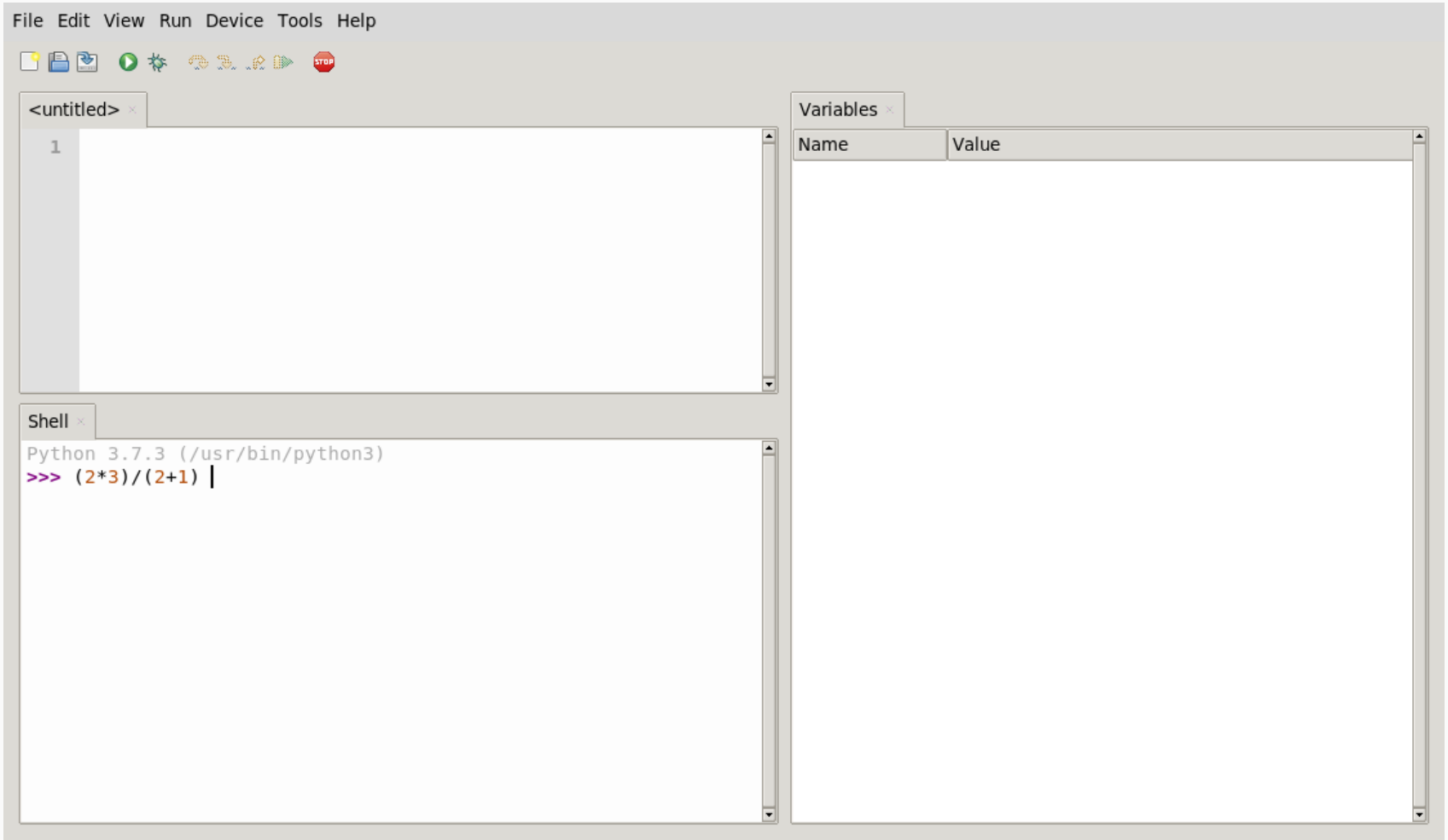
Essayez de répondre aux questions suivantes pour voir si vous avez compris. La réponse est donnée afin que vous puissiez vérifier tout seul.

- Calculer $\frac{2 \times 3}{2 + 1}$ Réponse : `2.0` (et non `2`)
- Calculer $2 \times \frac{3}{2} + 1$ Réponse : `4.0`
- Calculer $2^{(3^4)}$ Réponse : `241785163922925834941235 2`

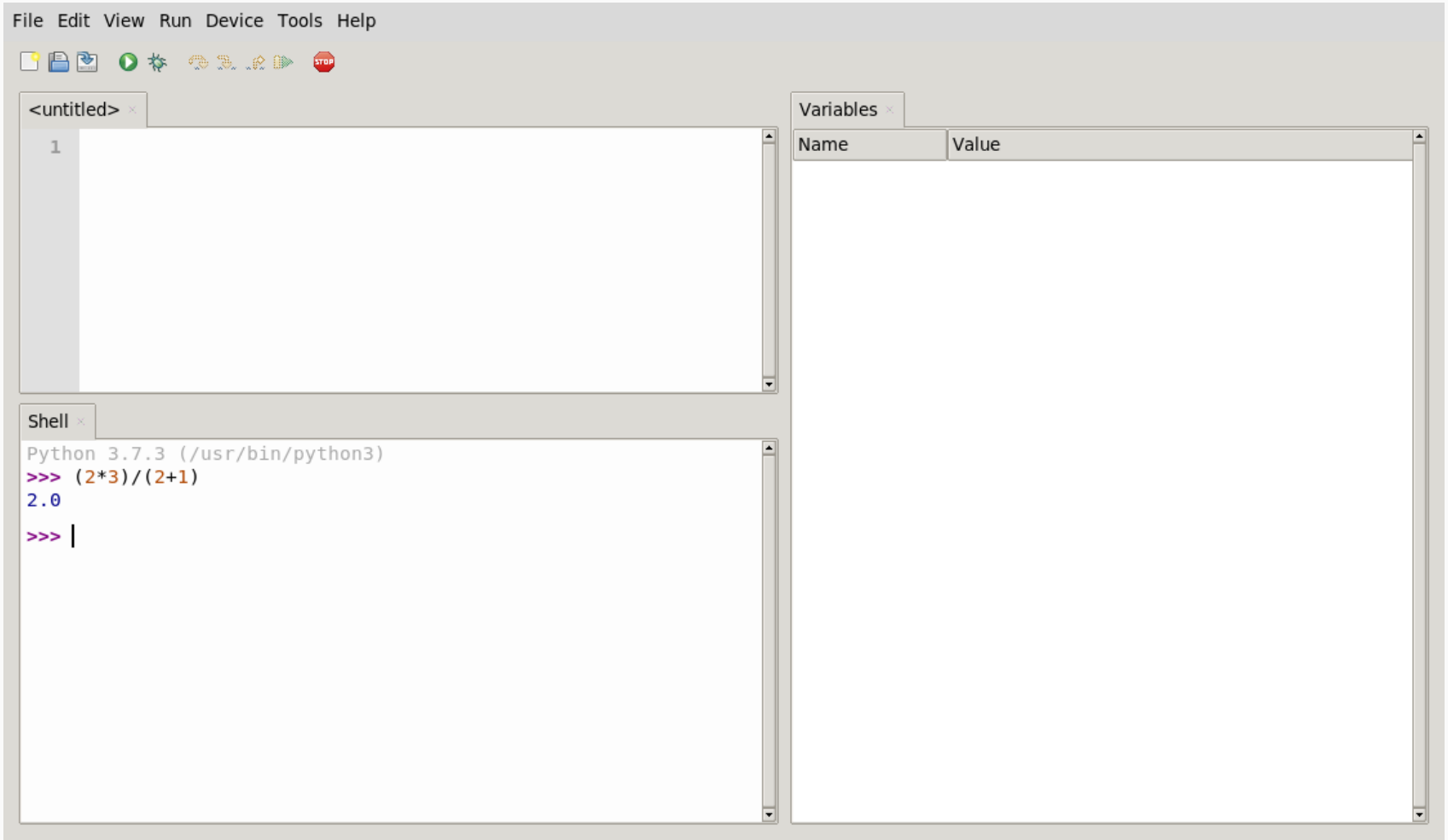
- On se place dans la fenêtre en bas à gauche Shell



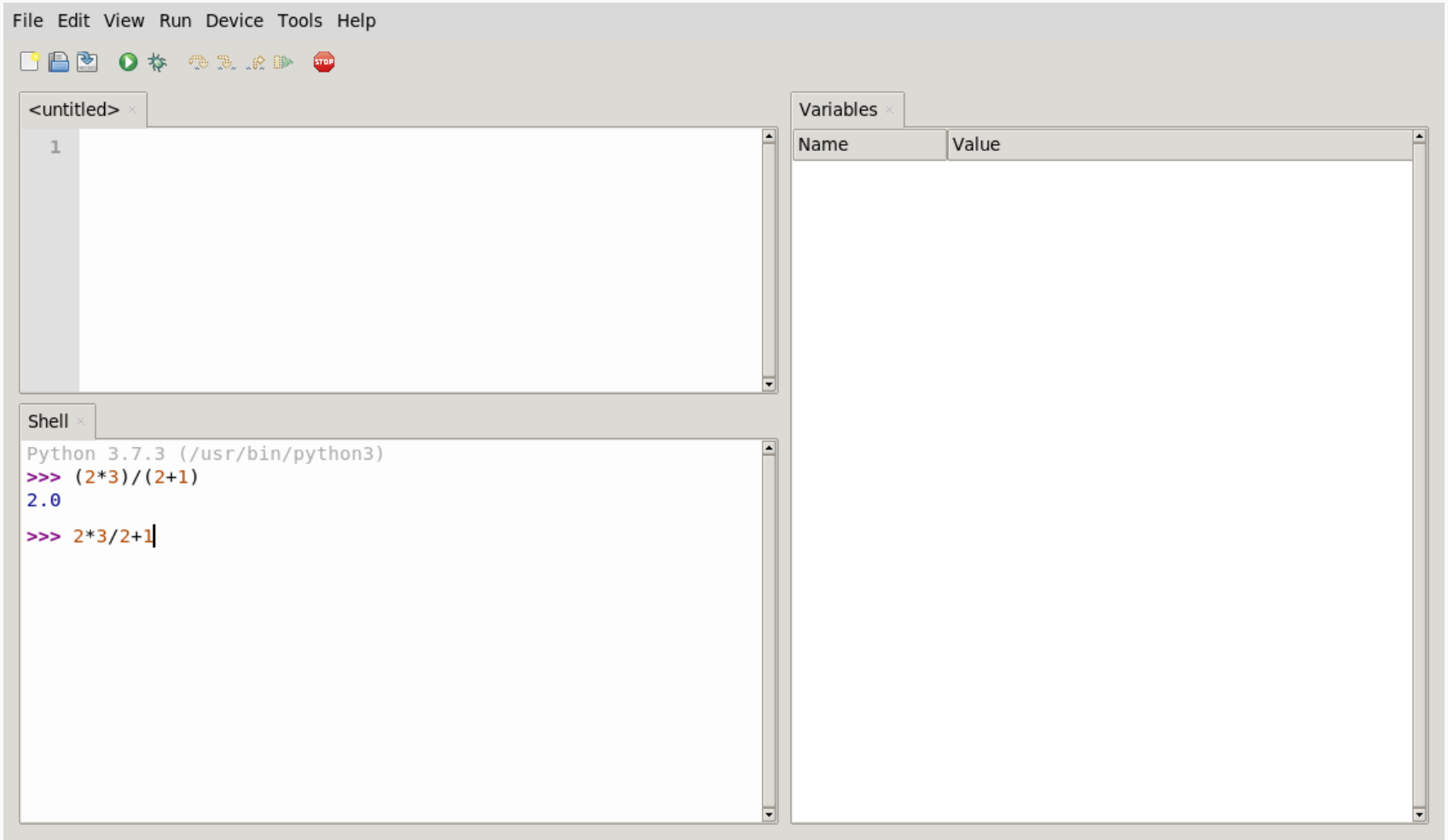
- On se place dans la fenêtre en bas à gauche Shell
- On écrit le calcul…



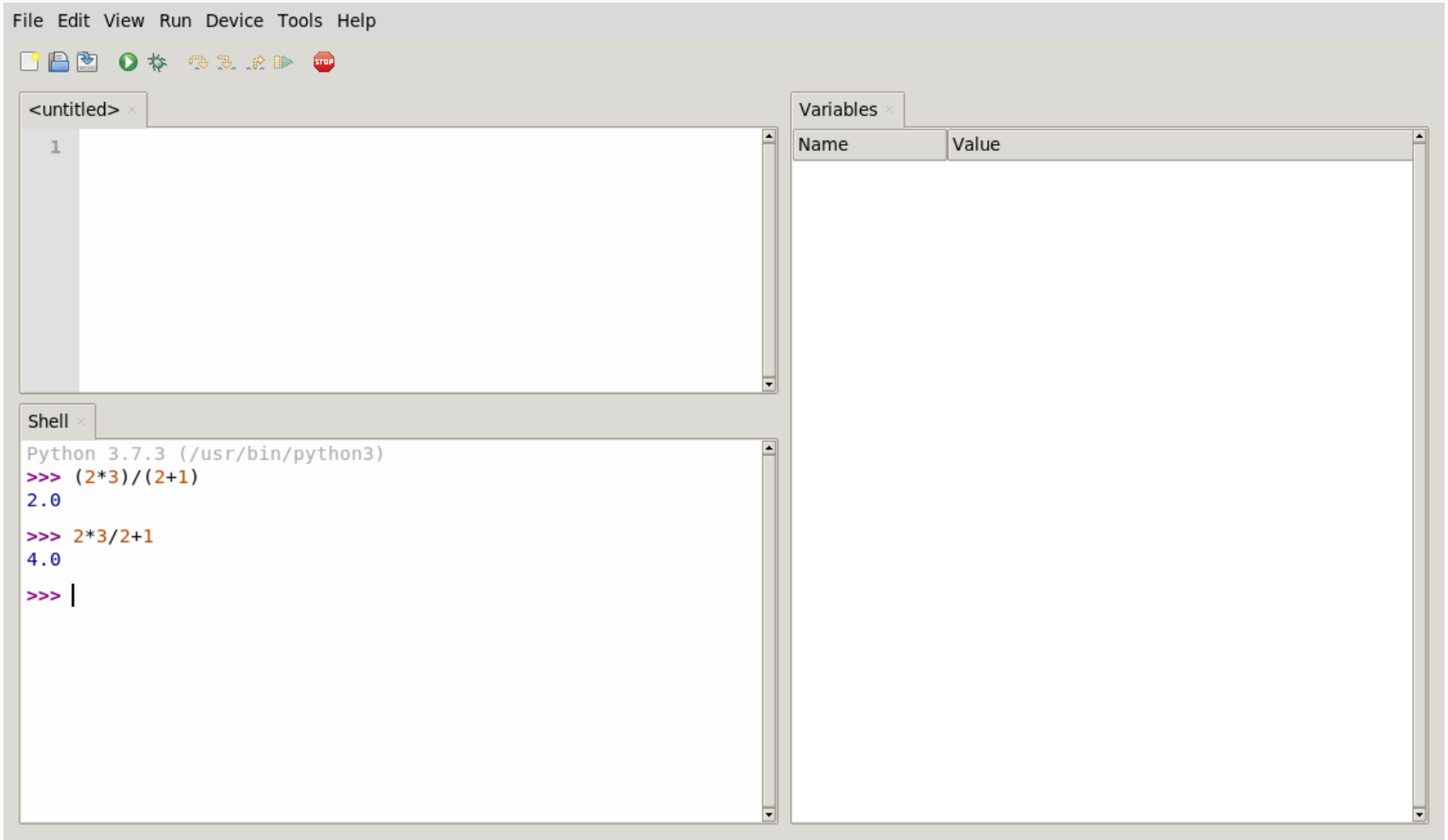
- On se place dans la fenêtre en bas à gauche Shell
- On écrit le calcul… puis on appuie sur la touche Entrée ↵



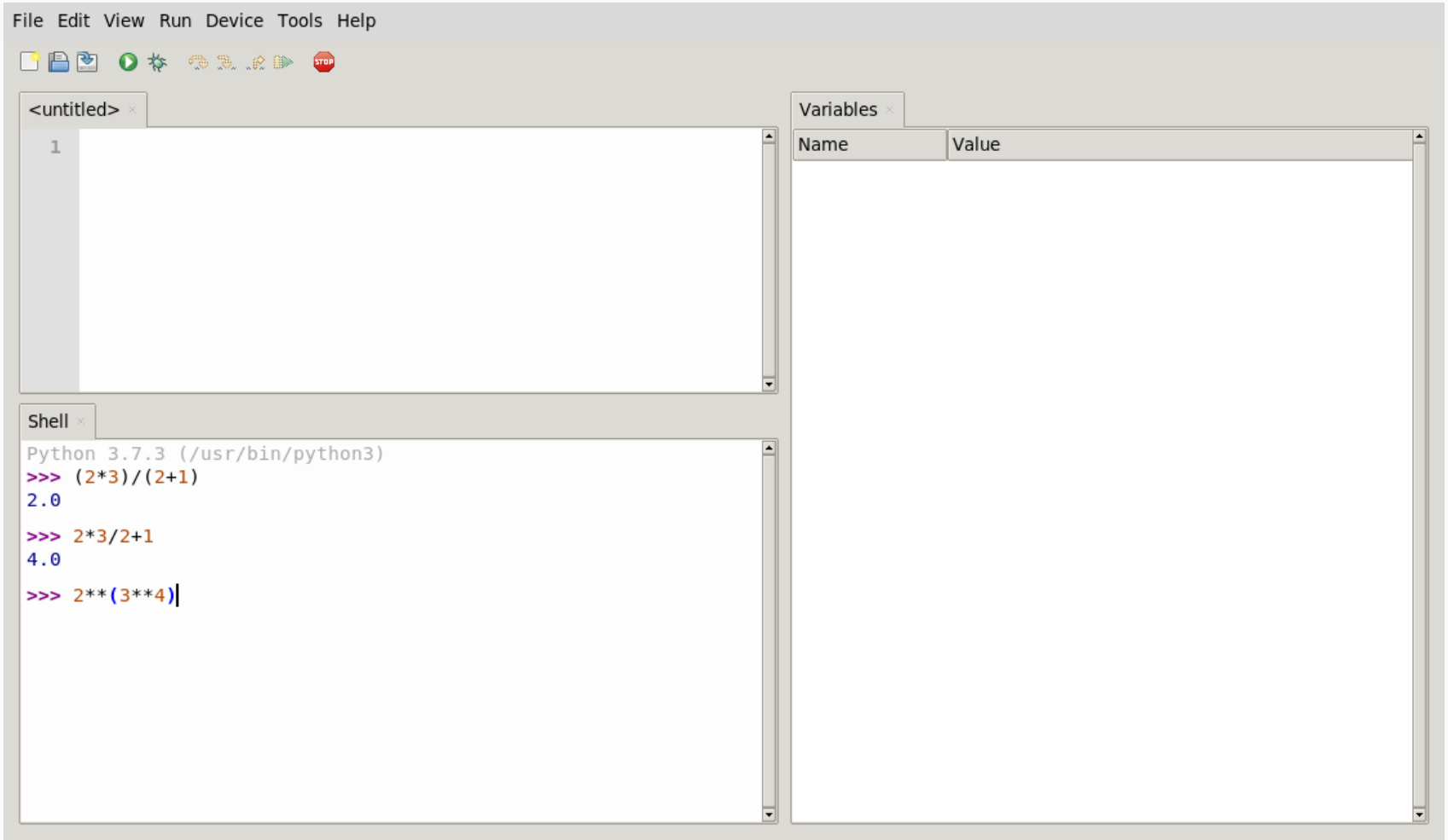
- On se place dans la fenêtre en bas à gauche Shell
- On écrit le calcul… puis on appuie sur la touche Entrée ↵



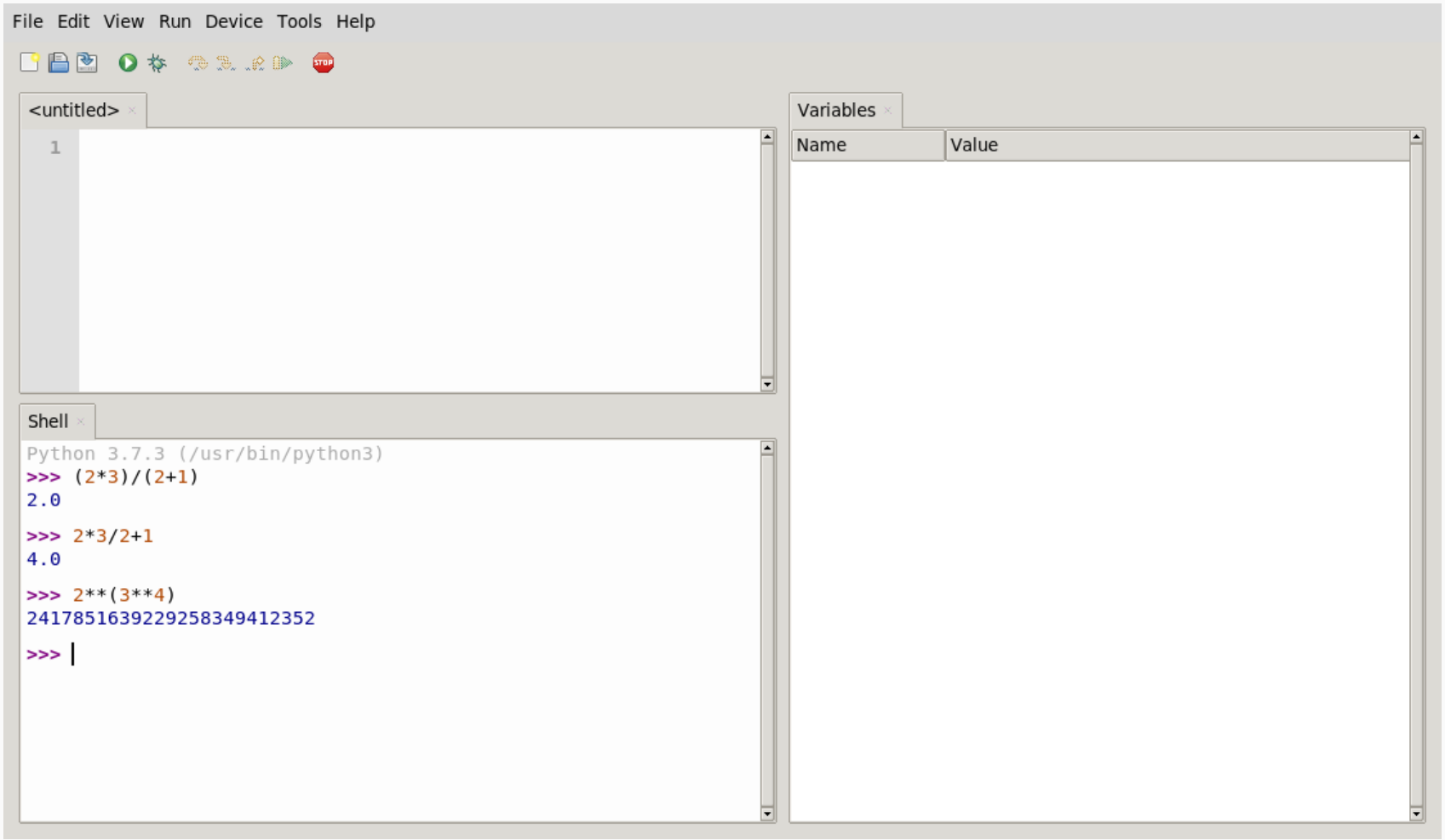
- On se place dans la fenêtre en bas à gauche Shell
- On écrit le calcul… puis on appuie sur la touche Entrée ⏎



- On se place dans la fenêtre en bas à gauche Shell
- On écrit le calcul… puis on appuie sur la touche Entrée ↵



- On se place dans la fenêtre en bas à gauche Shell
- On écrit le calcul… puis on appuie sur la touche Entrée ↵



Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

- Une variable est une boîte qui contient une valeur.

```
>>>
```
SHELL

- Une variable est une boîte qui contient une valeur.

```shell
>>> a = 2 # lire : a prend la valeur 2
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

```shell
>>> a = 2 # lire : a prend la valeur 2
>>>
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>>
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>>
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

SHELL
```
>>>
```

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

SHELL
```
>>> 1+2 = a
```

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

SHELL
```
>>> 1+2 = a
  File "<console>", line 1
    1+2 = a
    ^^^
SyntaxError:  cannot assign to
expression here. Maybe you meant
'==' instead of '='?
```

SHELL
```
>>>
```

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

SHELL
```
>>> 1+2 = a
  File "<console>", line 1
    1+2 = a
    ^^^
SyntaxError:  cannot assign to
expression here. Maybe you meant
'==' instead of '='?
```

SHELL
```
>>> a=1+2
```

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

SHELL
```
>>> 1+2 = a
  File "<console>", line 1
    1+2 = a
    ^^^
SyntaxError:  cannot assign to
expression here. Maybe you meant
'==' instead of '='?
```

SHELL
```
>>> a=1+2
>>>
```

- Une variable est une boîte qui contient une valeur.

SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

SHELL
```
>>> 1+2 = a
  File "<console>", line 1
    1+2 = a
    ^^^
SyntaxError:  cannot assign to
expression here. Maybe you meant
'==' instead of '='?
```

SHELL
```
>>> a=1+2
>>> a
```

- Une variable est une boîte qui contient une valeur.

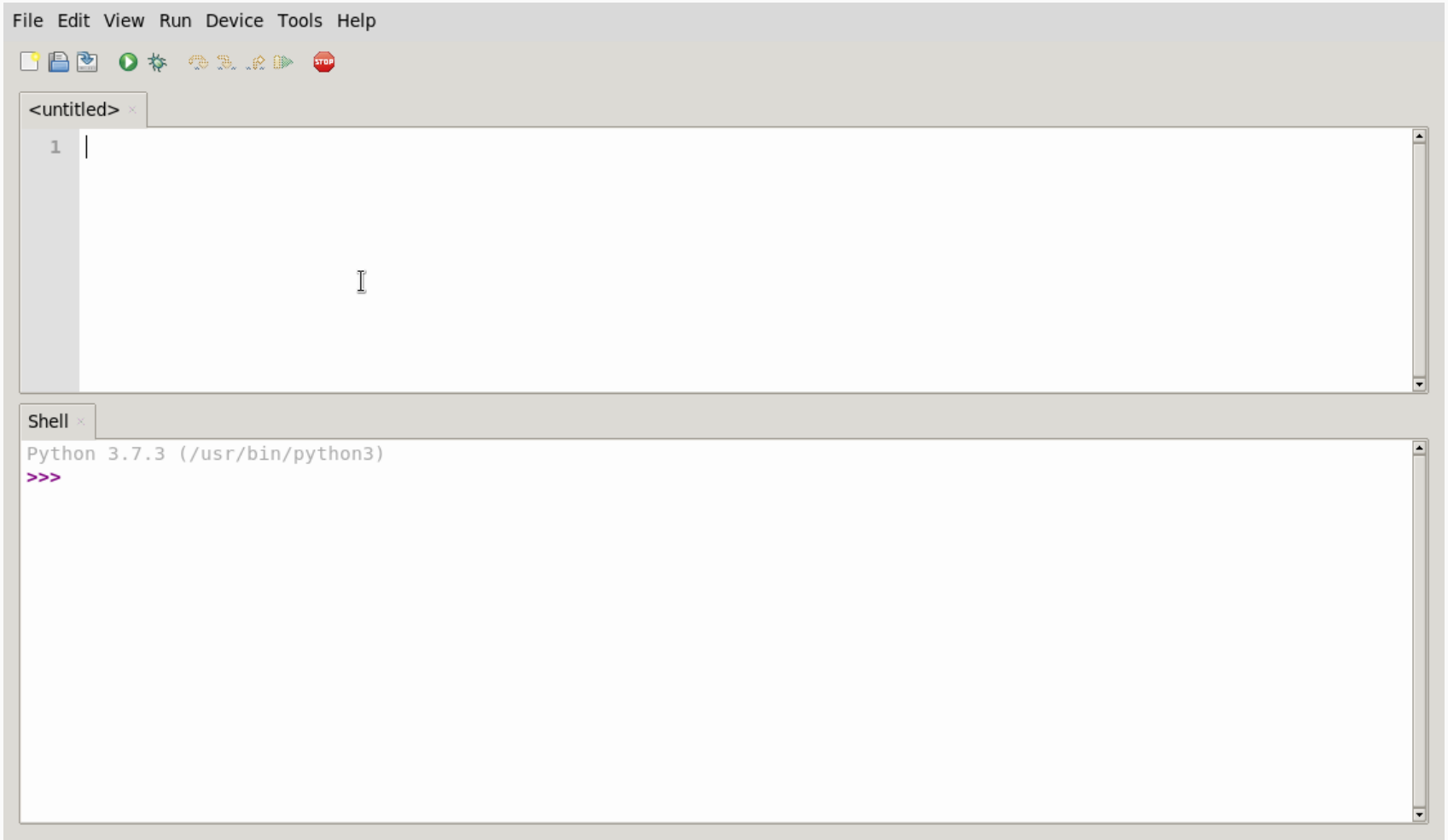
SHELL
```
>>> a = 2 # lire : a prend la valeur 2
>>> p = 10 # p prend la valeur 10
>>> c = a ** p # c prend pour valeur celle de a^p
>>> c
1024
```

- Ce symbole = n'est pas l'égalité mathématique. C'est l'instruction d'affectation. Il faut écrire le nom de la variable affectée à gauche.

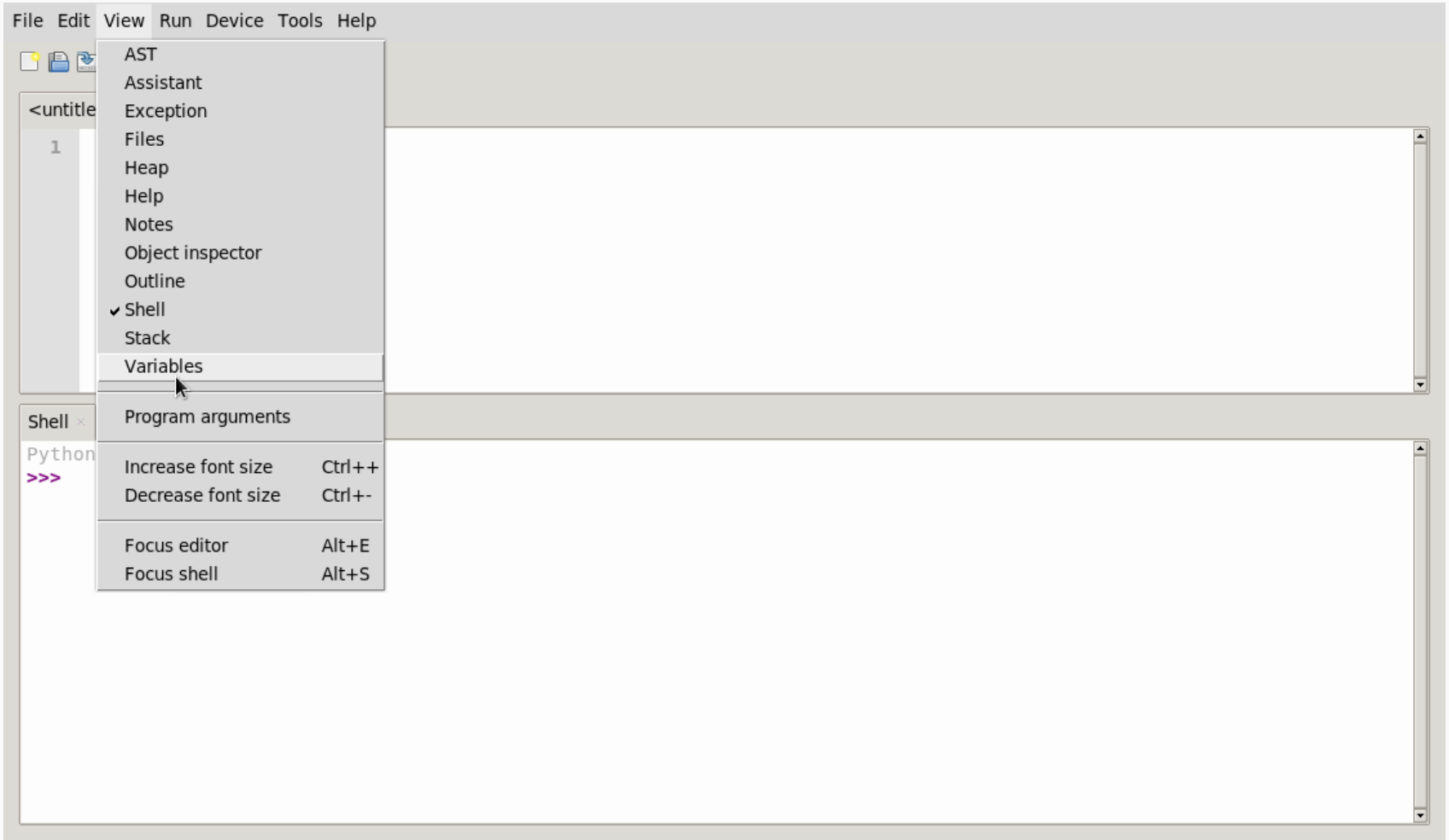
SHELL
```
>>> 1+2 = a
  File "<console>", line 1
    1+2 = a
    ^^^
SyntaxError:  cannot assign to
expression here. Maybe you meant
'==' instead of '='?
```

SHELL
```
>>> a=1+2
>>> a
3
```

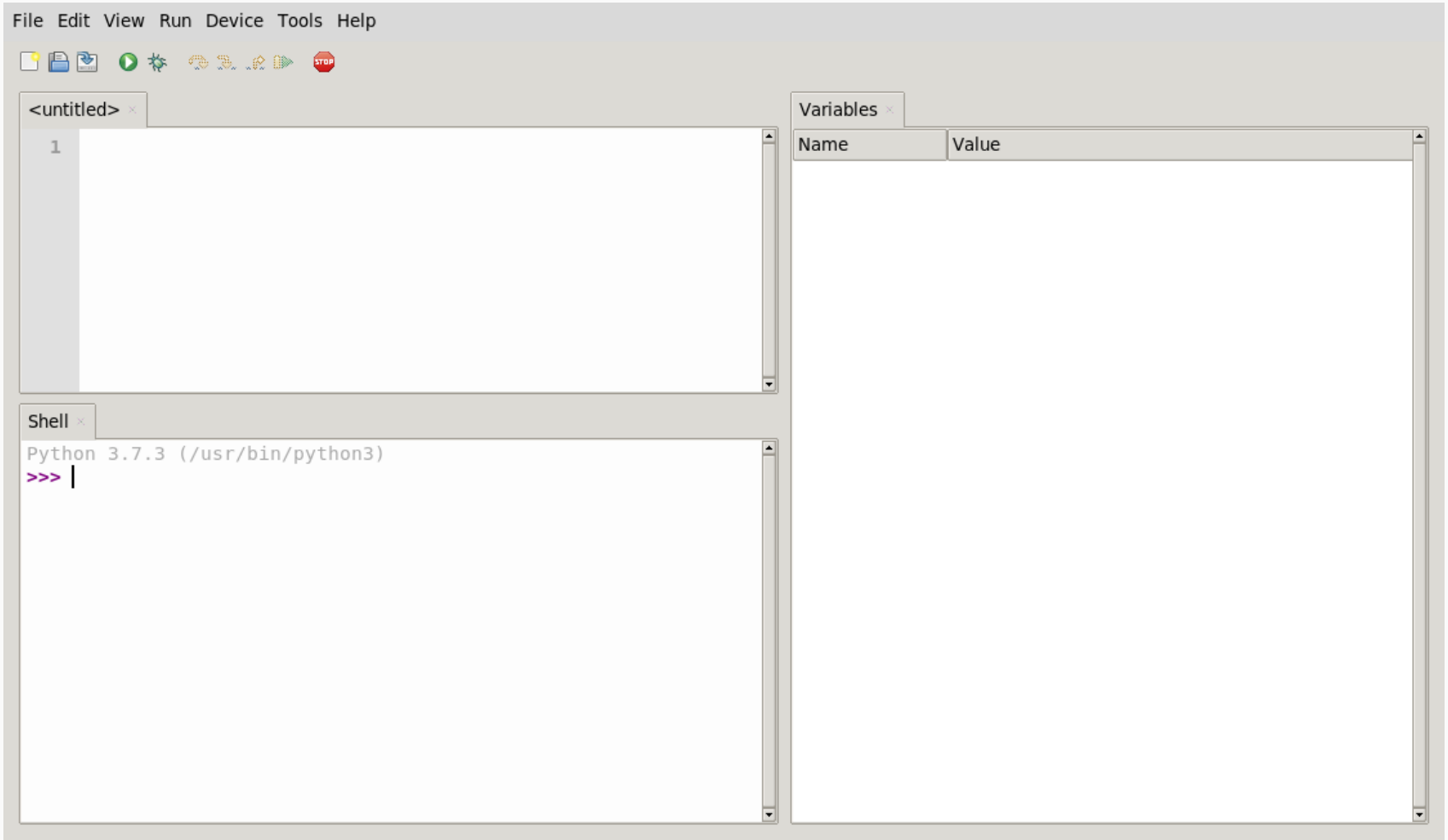
- Si nécessaire on ouvre l'onglet Variables (menu View)



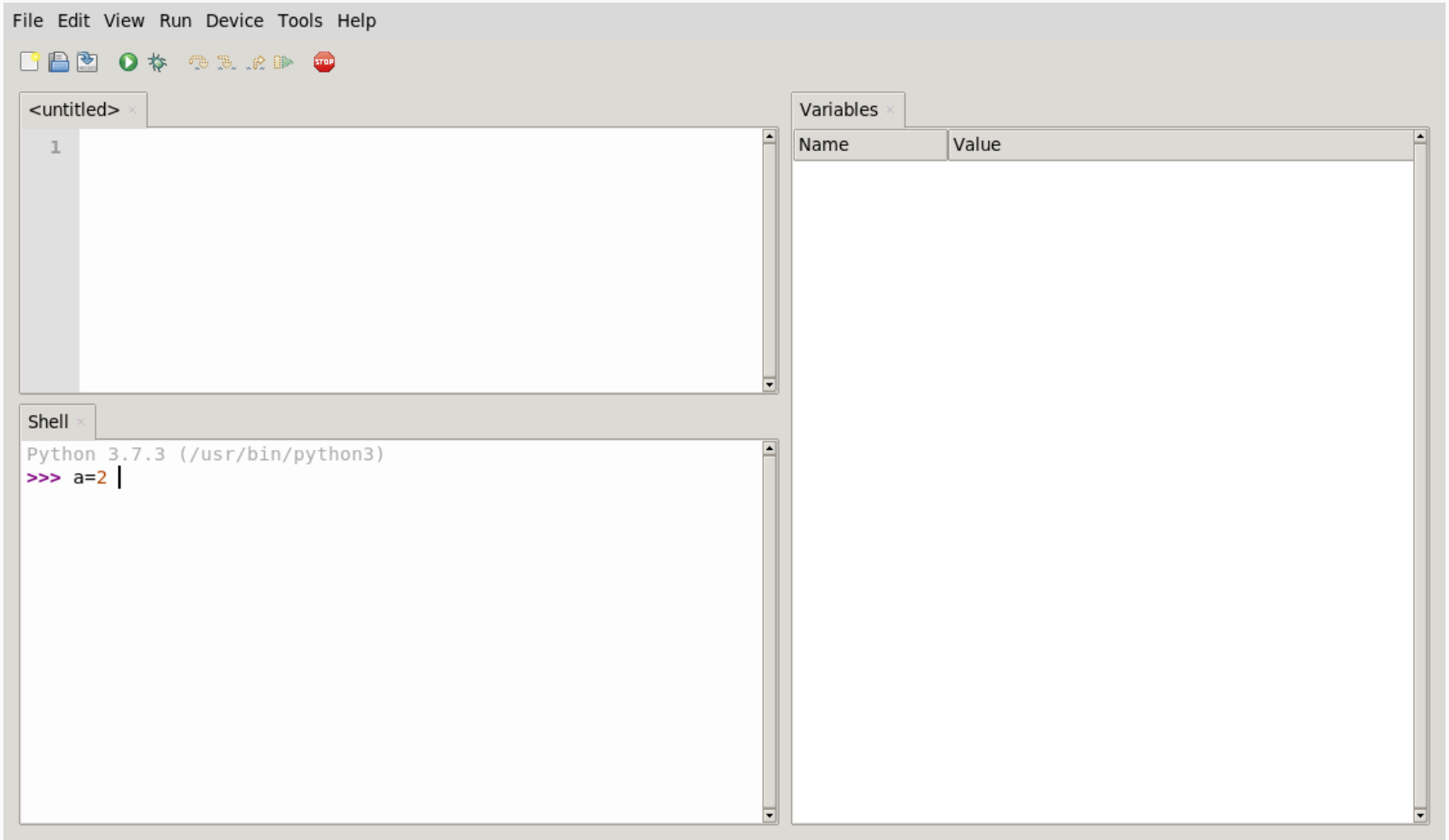
- Si nécessaire on ouvre l'onglet Variables (menu View)



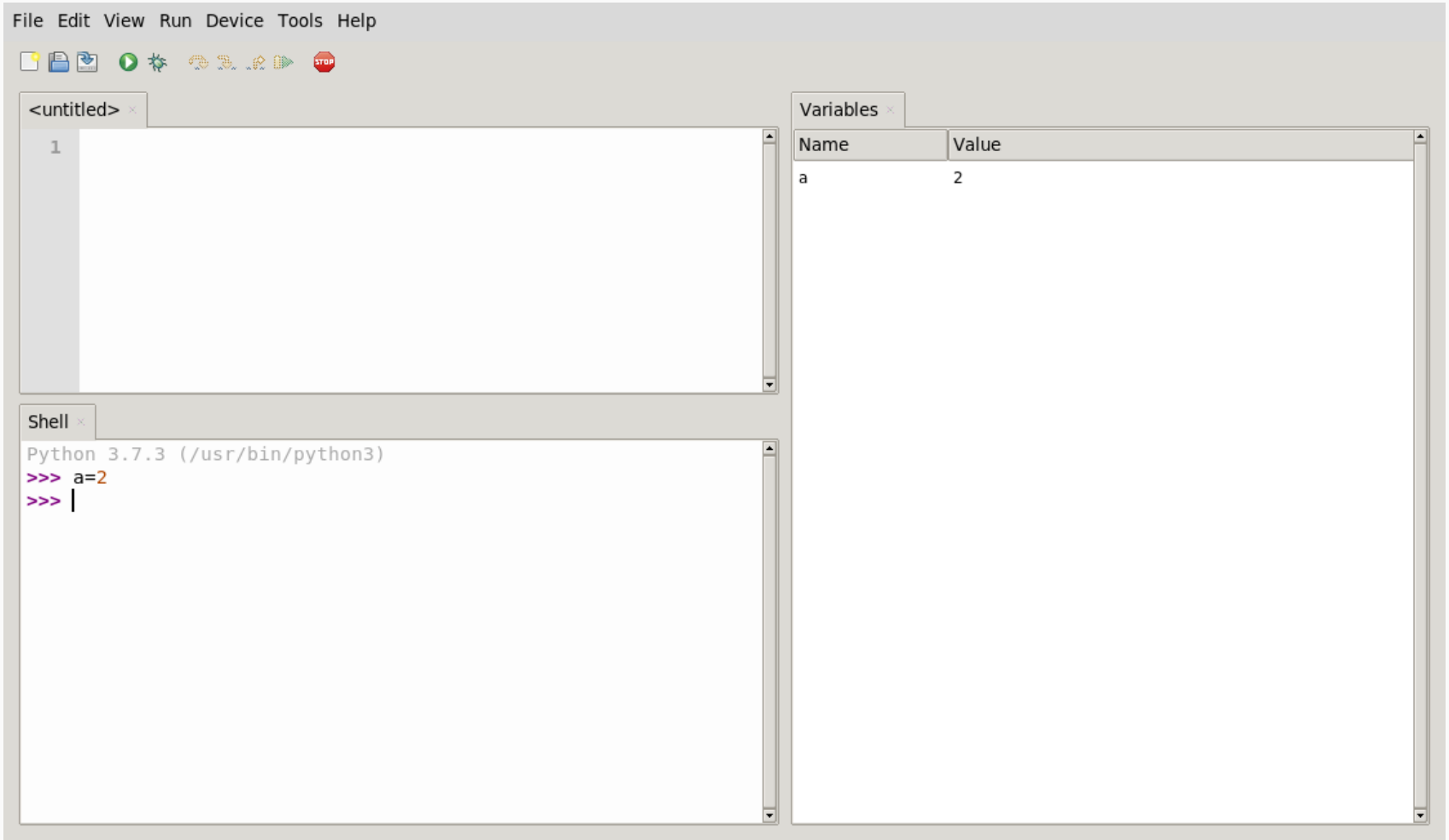
- Si nécessaire on ouvre l'onglet Variables (menu View)



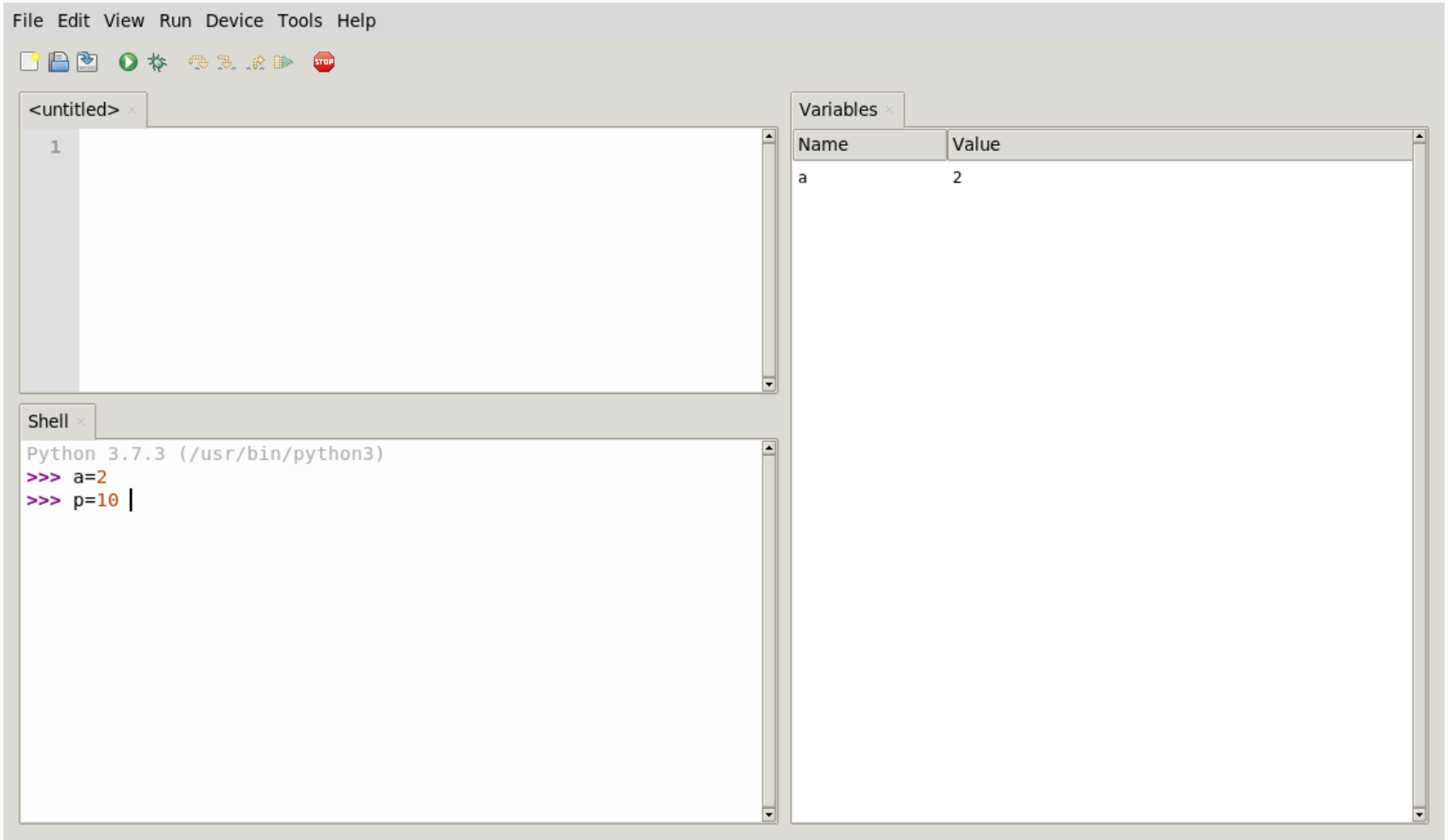
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell



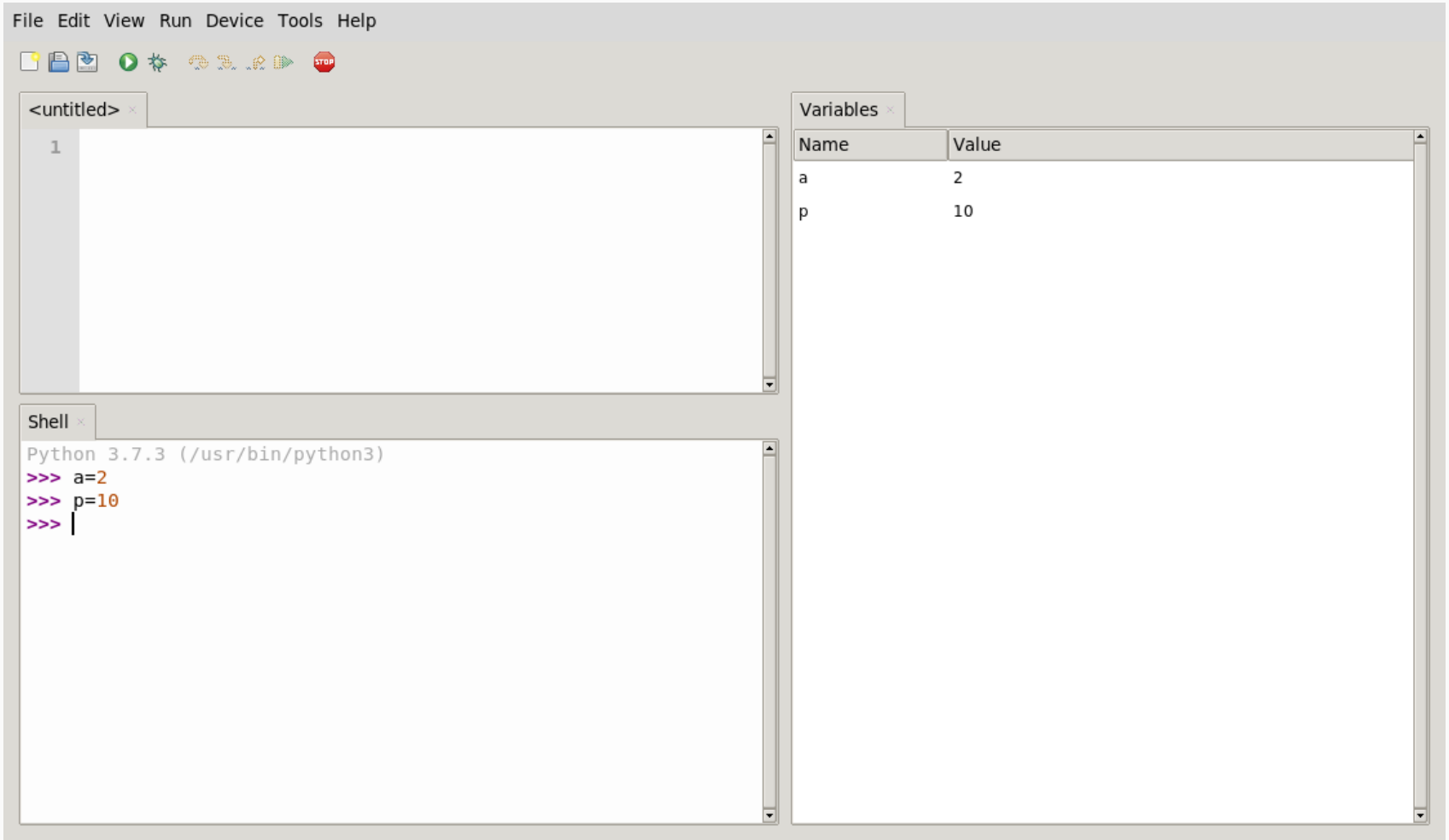
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell



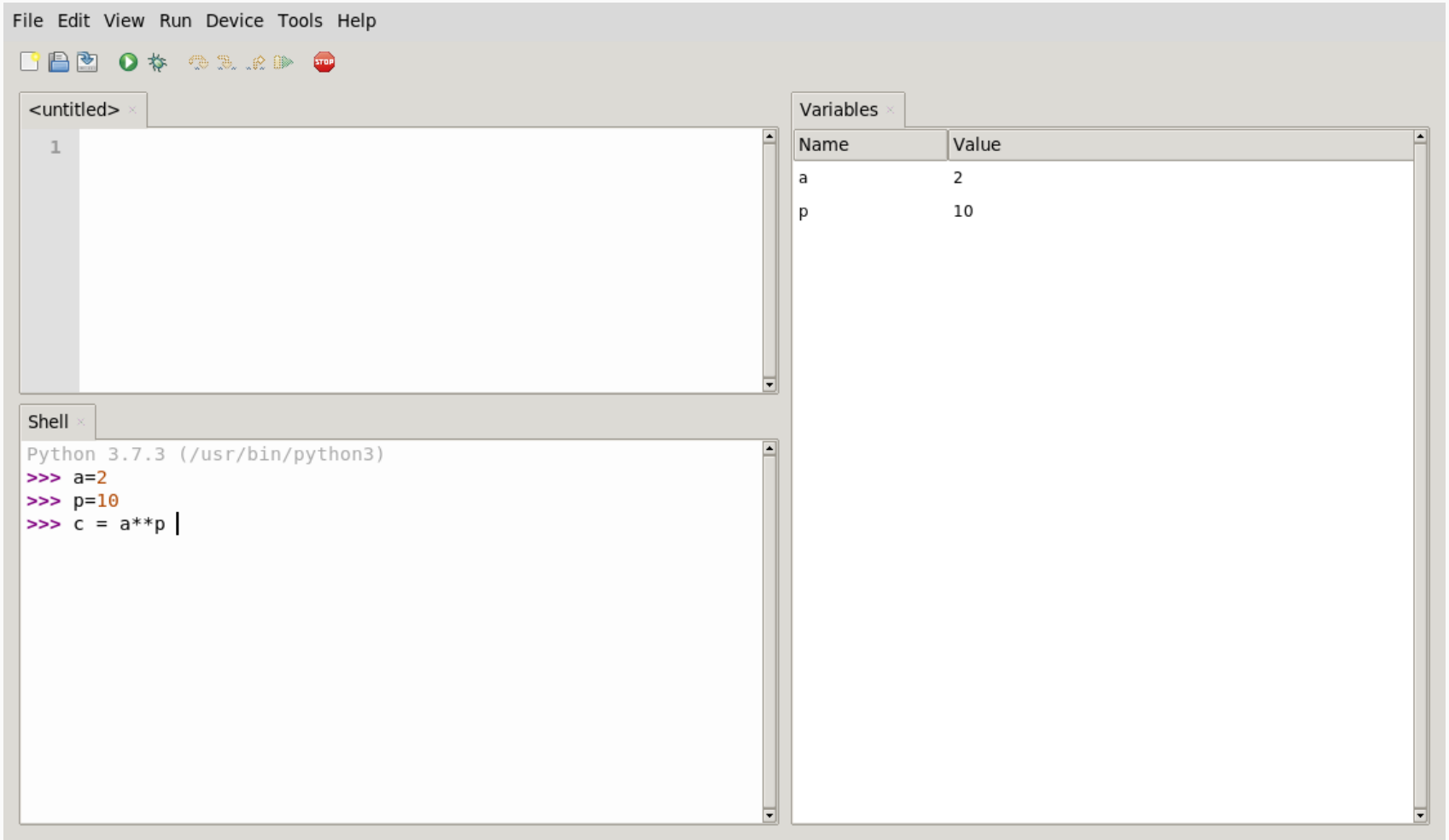
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



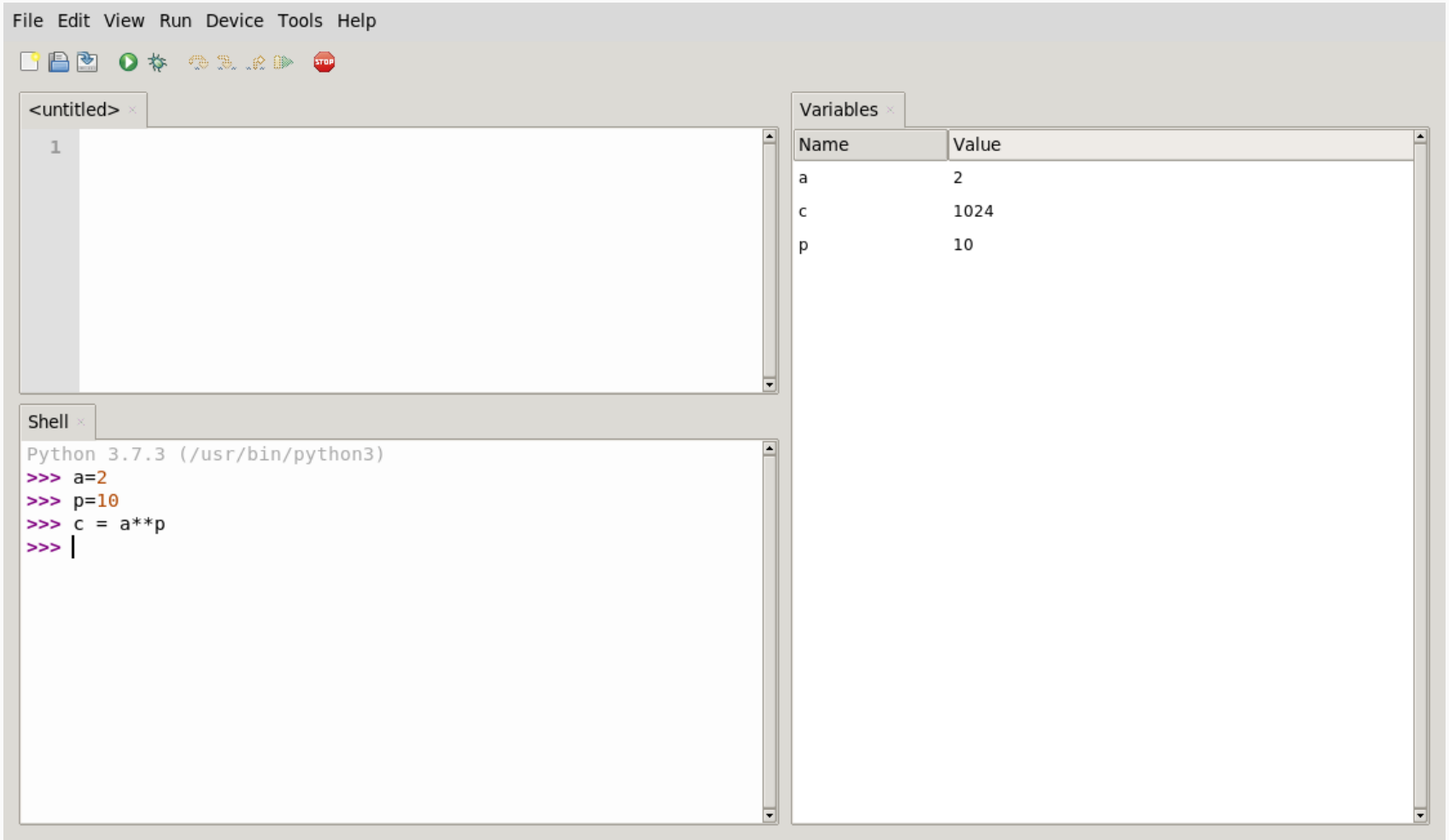
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



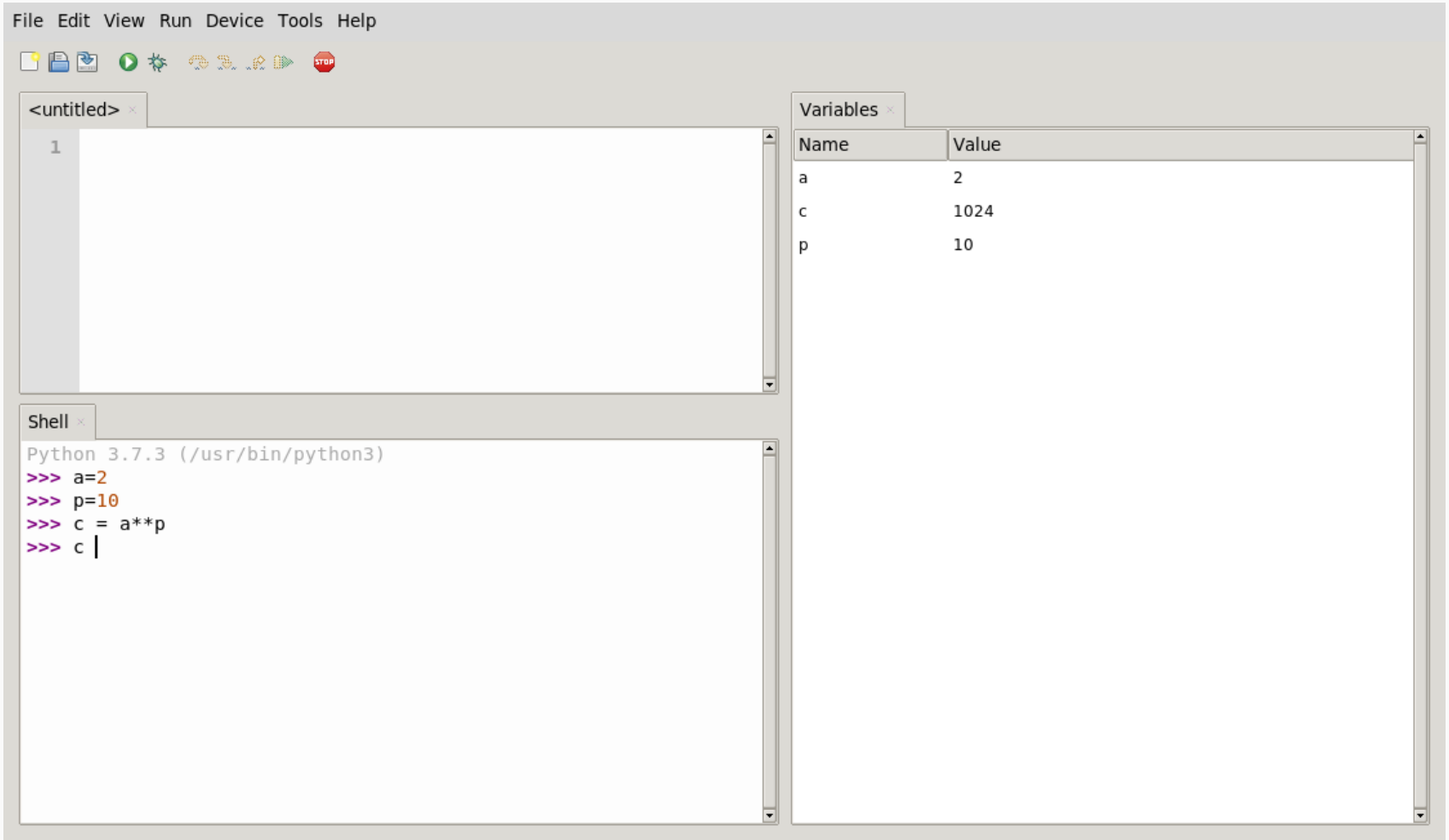
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



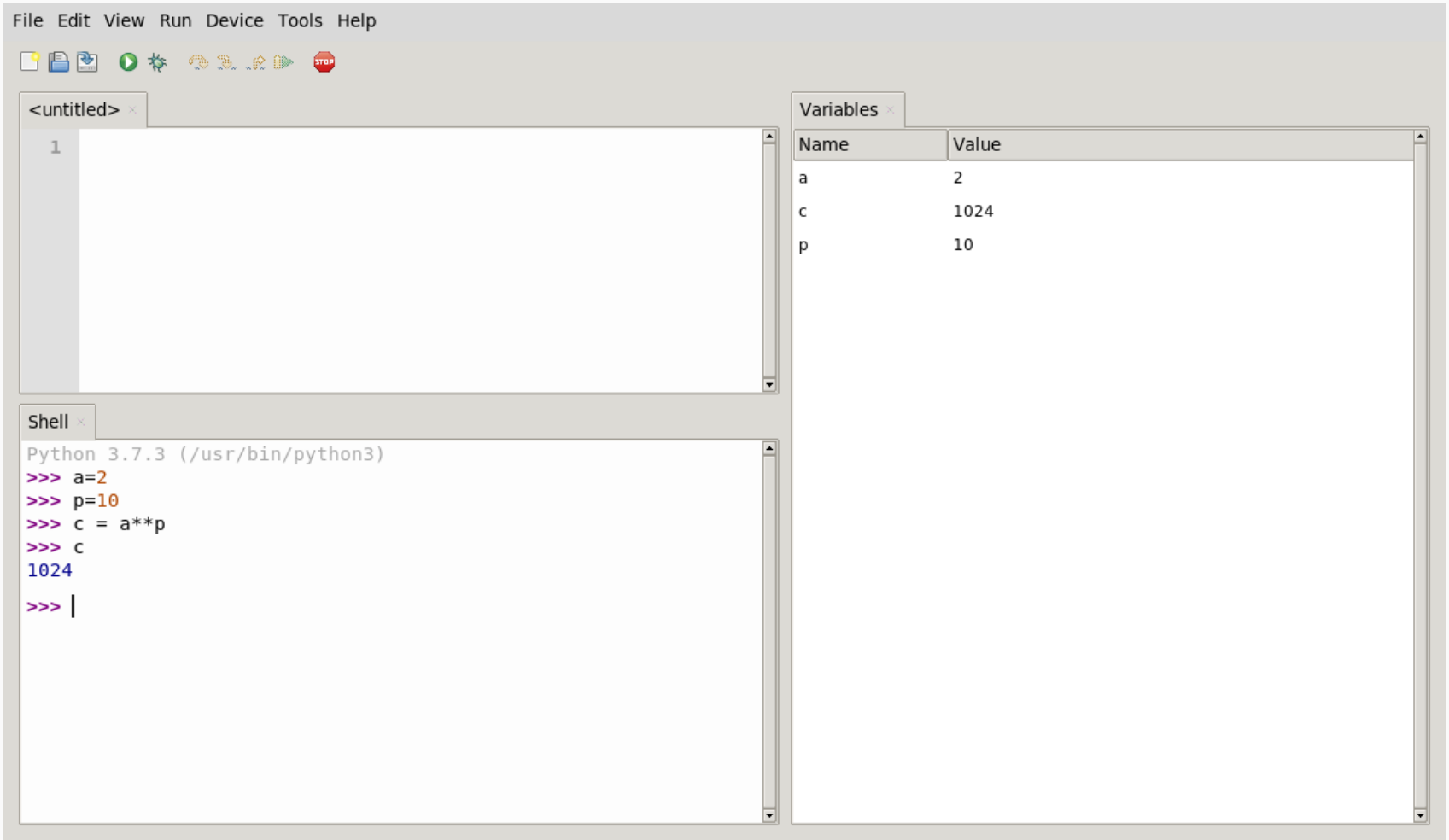
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



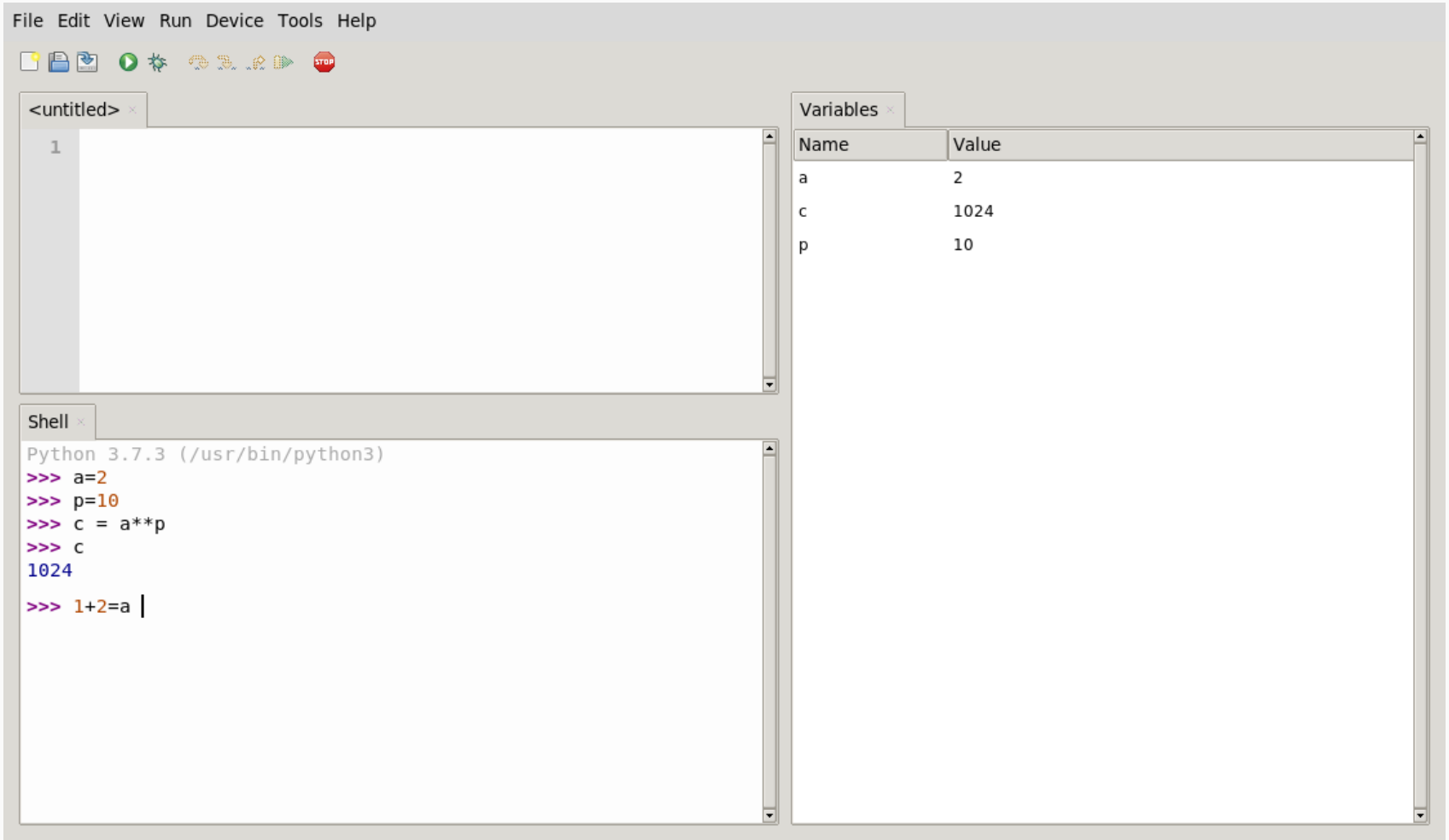
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



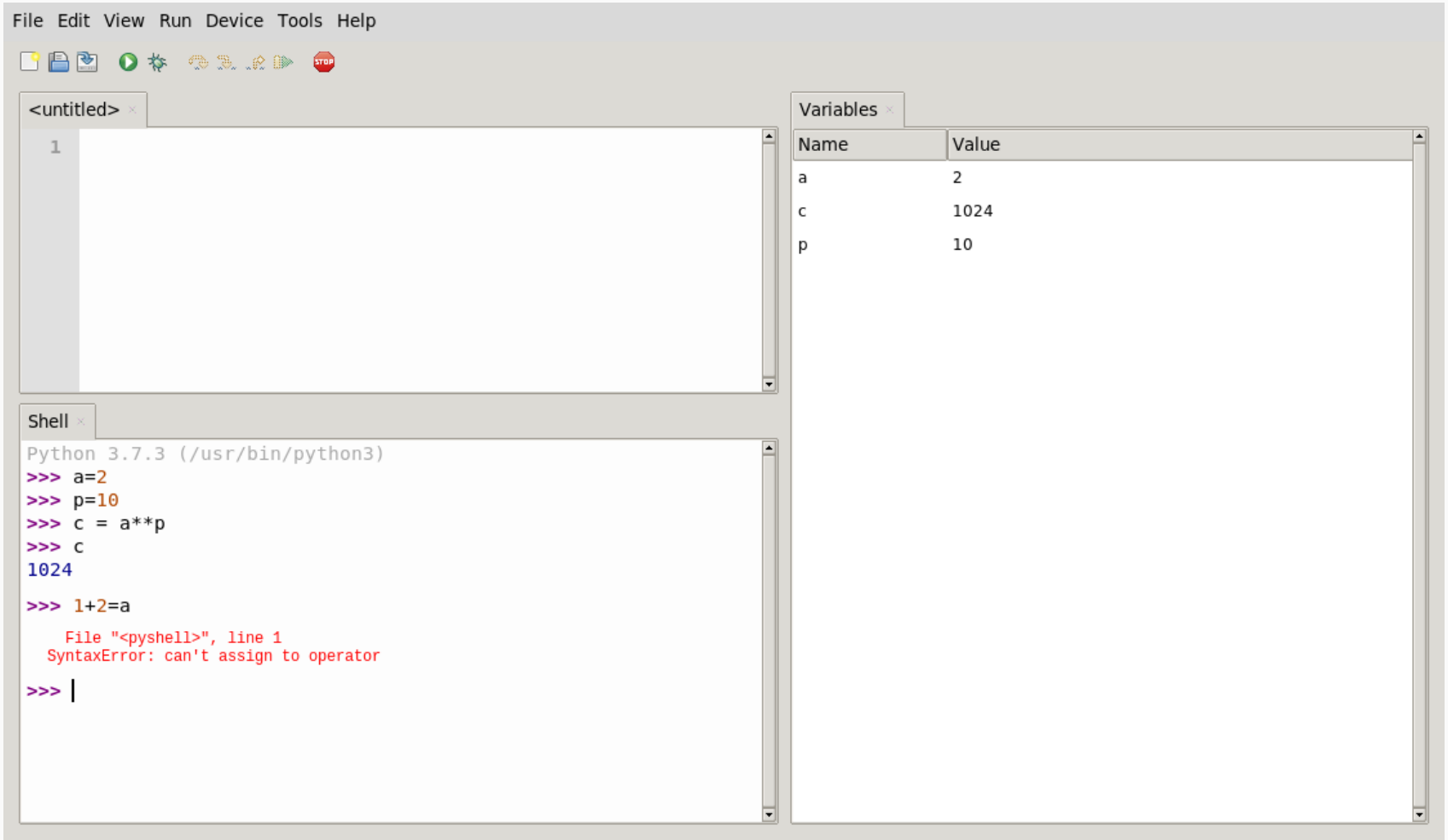
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

```
>>>
```
SHELL

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>>
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b # Instruction (modifie c sans résultat)
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b # Instruction (modifie c sans résultat)
>>>
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL

```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>>
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>>
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1  # Le résultat de a+1 est donné à a
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1  # Le résultat de a+1 est donné à a
>>>
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1  # Le résultat de a+1 est donné à a
>>> b
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1 # Le résultat de a+1 est donné à a
>>> b
2
>>>
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1 # Le résultat de a+1 est donné à a
>>> b
2
>>> a
```

- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

SHELL
```
>>> a*b    # expression (retourne un résultat)
6
>>> c=a+b  # Instruction (modifie c sans résultat)
>>> c      # expression (avec résultat)
5
```

- Une variable peut changer de valeur.

SHELL
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1  # Le résultat de a+1 est donné à a
>>> b
2
>>> a
3
```

On souhaite échanger les valeurs de deux variables : `a`↔`b`. Par exemple si « `a` vaut `5` » et « `b` vaut `4` », on souhaite avoir : « `a` vaut `4` » et « `b` vaut `5` ».

Méthode intuitive et FAUSSE !

SHELL
```
>>>
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>>
```

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
```

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>>
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL

```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>>
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>>
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> temp=a    # a:2 et b:3 et temp:2
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> temp=a    # a:2 et b:3 et temp:2
>>>
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> temp=a    # a:2 et b:3 et temp:2
>>> a=b       # a:3 et b:3 et temp:2
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> temp=a    # a:2 et b:3 et temp:2
>>> a=b       # a:3 et b:3 et temp:2
>>>
```

Échange de deux variables

On souhaite échanger les valeurs de deux variables : a↔b. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et FAUSSE !

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> a=b       # a:3 et b:3
>>> b=a       # a:3 et b:3 Cela ne marche pas !
```

Il faut une variable temporaire pour ne pas perdre la valeur initiale de a.

SHELL
```
>>> a=2 ; b=3 # a:2 et b:3
>>> temp=a    # a:2 et b:3 et temp:2
>>> a=b       # a:3 et b:3 et temp:2
>>> b=temp    # a:3 et b:2 et temp:2
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2   # instruction
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2   # instruction
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

SHELL
```
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs True et False s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs True et False s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>> a>0
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs `True` et `False` s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>> a>0
True
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs True et False s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>> a>0
True
>>> a+1<=4*a  # a+1 ≤ 4a
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs True et False s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>> a>0
True
>>> a+1<=4*a  # a+1 ≤ 4a
True
>>>
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs True et False s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>> a>0
True
>>> a+1<=4*a  # a+1 ≤ 4a
True
>>> (a+1>3)==False # F = F
```

- Le signe = correspond à l'affectation.
- Le signe == correspond à l'égalité mathématique.

SHELL
```
>>> a=2  # instruction
>>> a==4 # expression
False
>>> a==2 # expression
True
>>> True == False
False
```

Les valeurs True et False s'appellent des booléens (George Boole, Royaume Unie, 1815-1864)

Il existe d'autres opérateurs de comparaison : <, >, <=, >= et !=.

SHELL
```
>>> a<2
False
>>> a>=2 # a ≥ 2
True
>>> a!=2 #  a ≠ 2
False
```

SHELL
```
>>> a>0
True
>>> a+1<=4*a  # a+1 ≤ 4a
True
>>> (a+1>3)==False # F = F
True
```

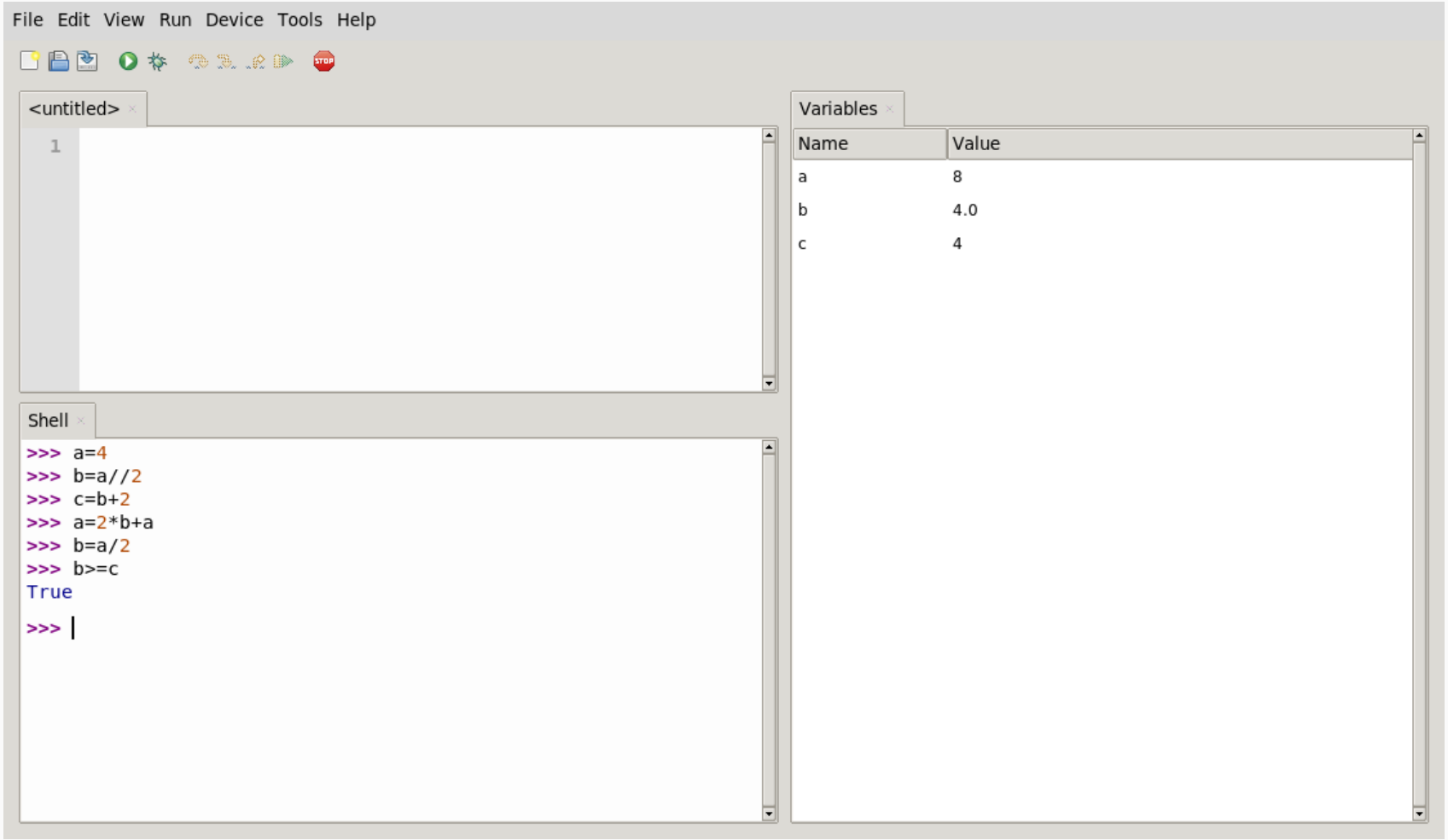
Que fait le code suivant ?

SCRIPT

```
a=4
b=a//2
c=b+2
a=2*b+a
b=a/2
b>=c
```

- Avec un papier et un crayon, décrivez le contenu et l'évolution de la mémoire étape par étape.
- Essayer avec Thonny, voir si vous aviez raison.

Voilà ce que vous êtes censé obtenir :



Évidemment, c'est à vous de l'exécuter pour voir l'évolution de la mémoire étape par étape.

Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

```
>>>
```
SHELL

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

```
>>> a = 2
```
SHELL

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL
```
>>> a = 2
>>>
```

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL
```
>>> a = 2
>>> print('Le carré de', a, 'vaut', a*a, "et 5.")
```

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL

```
>>> a = 2
>>> print('Le carré de', a, 'vaut', a*a, "et 5.")
Le carré de 2 vaut 4 et 5.
```

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL
```
>>> a = 2
>>> print('Le carré de', a, 'vaut', a*a, "et 5.")
Le carré de 2 vaut 4 et 5.
```

- L'appel de `print` ne produit aucun résultat.

SHELL
```
>>>
```

La commande d'affichage : `print`

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL
```
>>> a = 2
>>> print('Le carré de', a, 'vaut', a*a, "et 5.")
Le carré de 2 vaut 4 et 5.
```

- L'appel de `print` ne produit aucun résultat.

SHELL
```
>>> print(5) == None
```

La commande précédente affiche 5, puis fait le test d'égalité.

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL
```
>>> a = 2
>>> print('Le carré de', a, 'vaut', a*a, "et 5.")
Le carré de 2 vaut 4 et 5.
```

- L'appel de `print` ne produit aucun résultat.

SHELL
```
>>> print(5) == None
5
True
```

La commande précédente affiche 5, puis fait le test d'égalité.

- La fonction `print` permet d'afficher une suite d'expressions.

SHELL
```
>>> a = 2
>>> print('Le carré de', a, 'vaut', a*a, "et 5.")
Le carré de 2 vaut 4 et 5.
```

- L'appel de `print` ne produit aucun résultat.

SHELL
```
>>> print(5) == None
5
True
```

La commande précédente affiche 5, puis fait le test d'égalité.

- `None` est une valeur spéciale, qui signifie « rien ».

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>>
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>> a='bonjour !'
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>> a='bonjour !'
>>>
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>> a='bonjour !'
>>> a
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL

```
>>> a='bonjour !'
>>> a
'bonjour !'
>>>
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL

```
>>> a='bonjour !'
>>> a
'bonjour !'
>>> print('a contient :', a)
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL

```
>>> a='bonjour !'
>>> a
'bonjour !'
>>> print('a contient :', a)
a contient : bonjour !
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>> a='bonjour !'
>>> a
'bonjour !'
>>> print('a contient :', a)
a contient : bonjour !
```

La fonction `input(message)` affiche `message` puis demande à l'utilisateur d'entrer une chaîne de caractères.

SCRIPT
```
prénom = input('Quel est votre prénom ? ')
print('J'ai rencontré',prénom,'et il est gentil')
```

SCRIPT
```
Quel est votre prénom ?
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>> a='bonjour !'
>>> a
'bonjour !'
>>> print('a contient :', a)
a contient : bonjour !
```

La fonction `input(message)` affiche `message` puis demande à l'utilisateur d'entrer une chaîne de caractères.

SCRIPT
```
prénom = input('Quel est votre prénom ? ')
print('J'ai rencontré',prénom,'et il est gentil')
```

SCRIPT
```
Quel est votre prénom ?  Je m'appelle Olivier
```

- `'bonjour !'` est une chaîne de caractères
- une chaîne de caractère est un texte entouré de guillemets simples.
- On peut sauvegarder une chaîne de caractères dans une variable.

SHELL
```
>>> a='bonjour !'
>>> a
'bonjour !'
>>> print('a contient :', a)
a contient : bonjour !
```

La fonction `input(message)` affiche `message` puis demande à l'utilisateur d'entrer une chaîne de caractères.

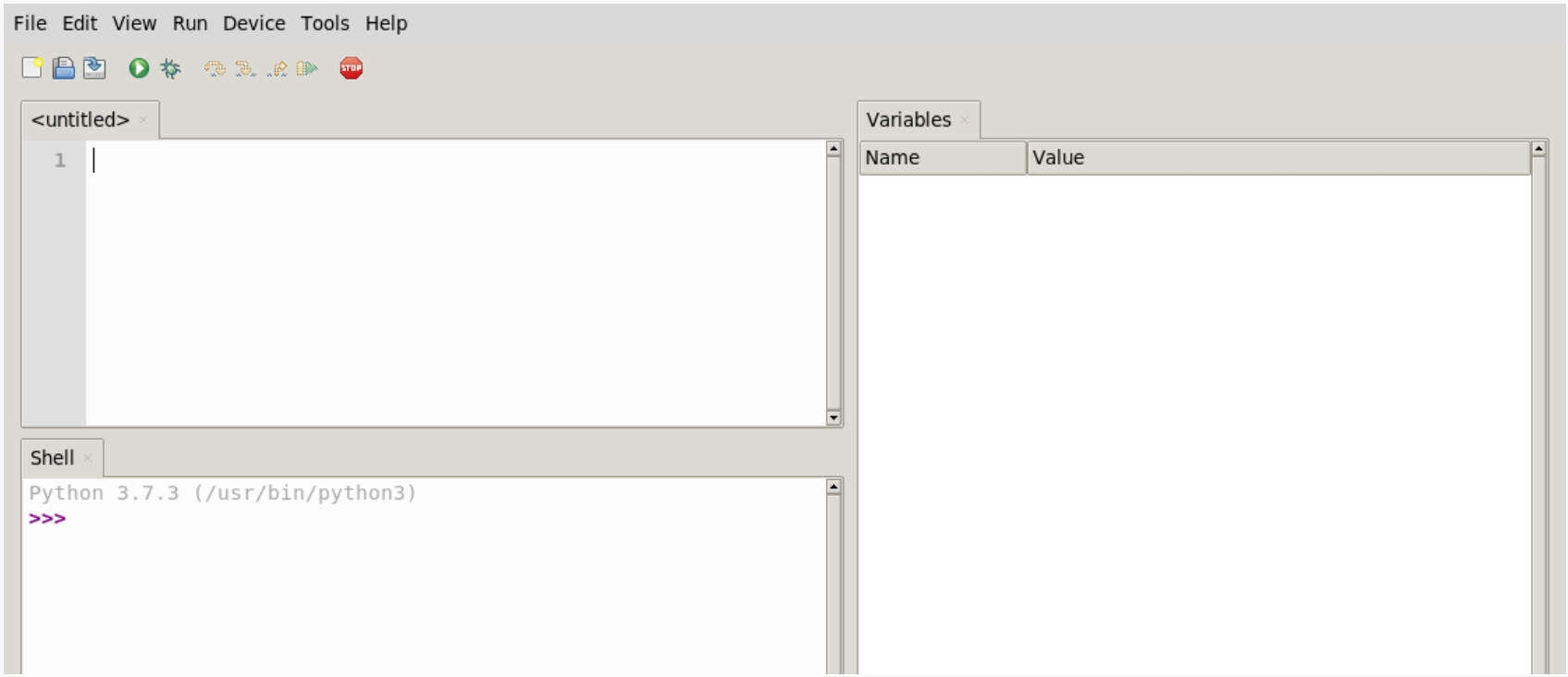
SCRIPT
```
prénom = input('Quel est votre prénom ? ')
print('J'ai rencontré',prénom,'et il est gentil')
```

SCRIPT
```
Quel est votre prénom ?  Je m'appelle Olivier
J'ai rencontré Je m'appelle Olivier et il est gentil
```

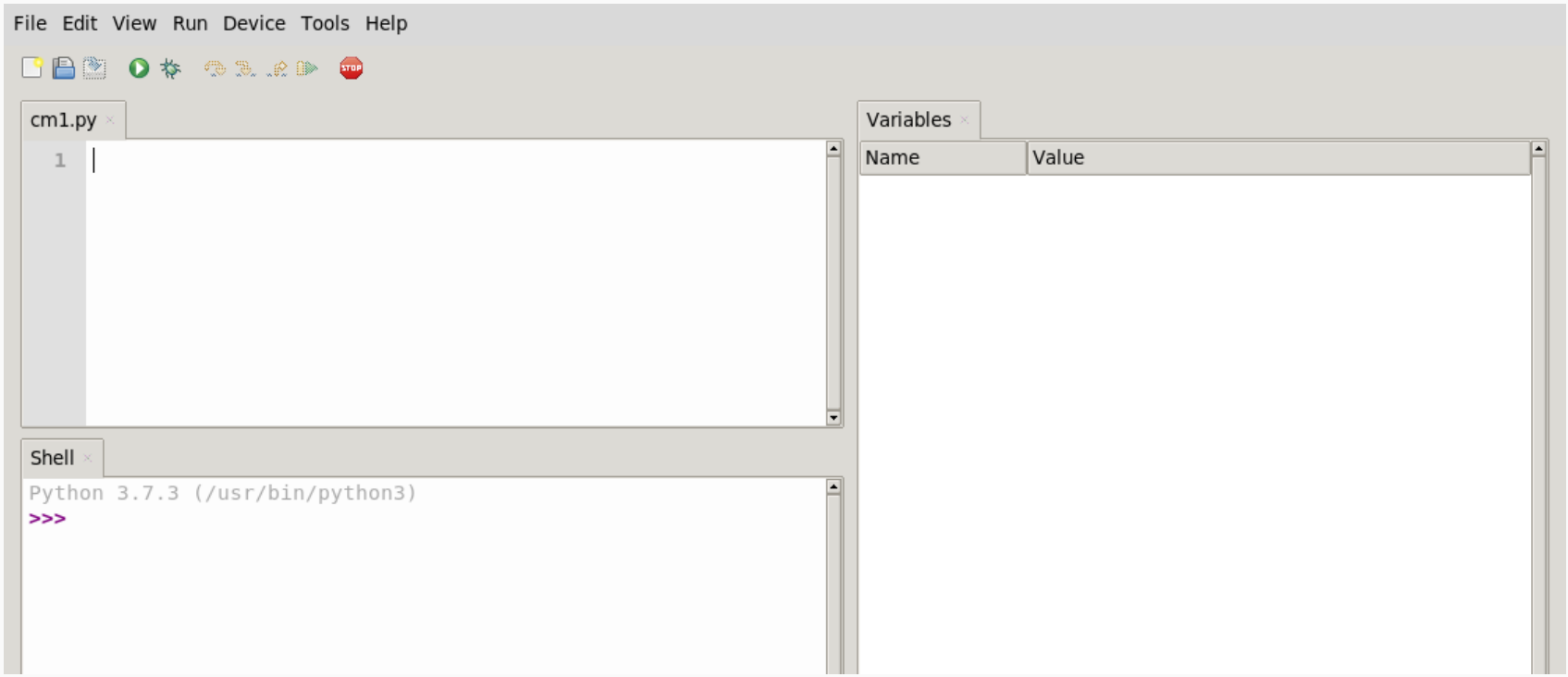
- Il est laborieux d'exécuter une à une les commandes dans le shell
  - et si l'on se trompe il faut tout recommencer !

- Il est laborieux d'exécuter une à une les commandes dans le shell
  - et si l'on se trompe il faut tout recommencer !
- Pour être plus efficace on va sauvegarder les commandes dans un fichier.
- un tel fichier est appelé script et son nom se termine par `.py`

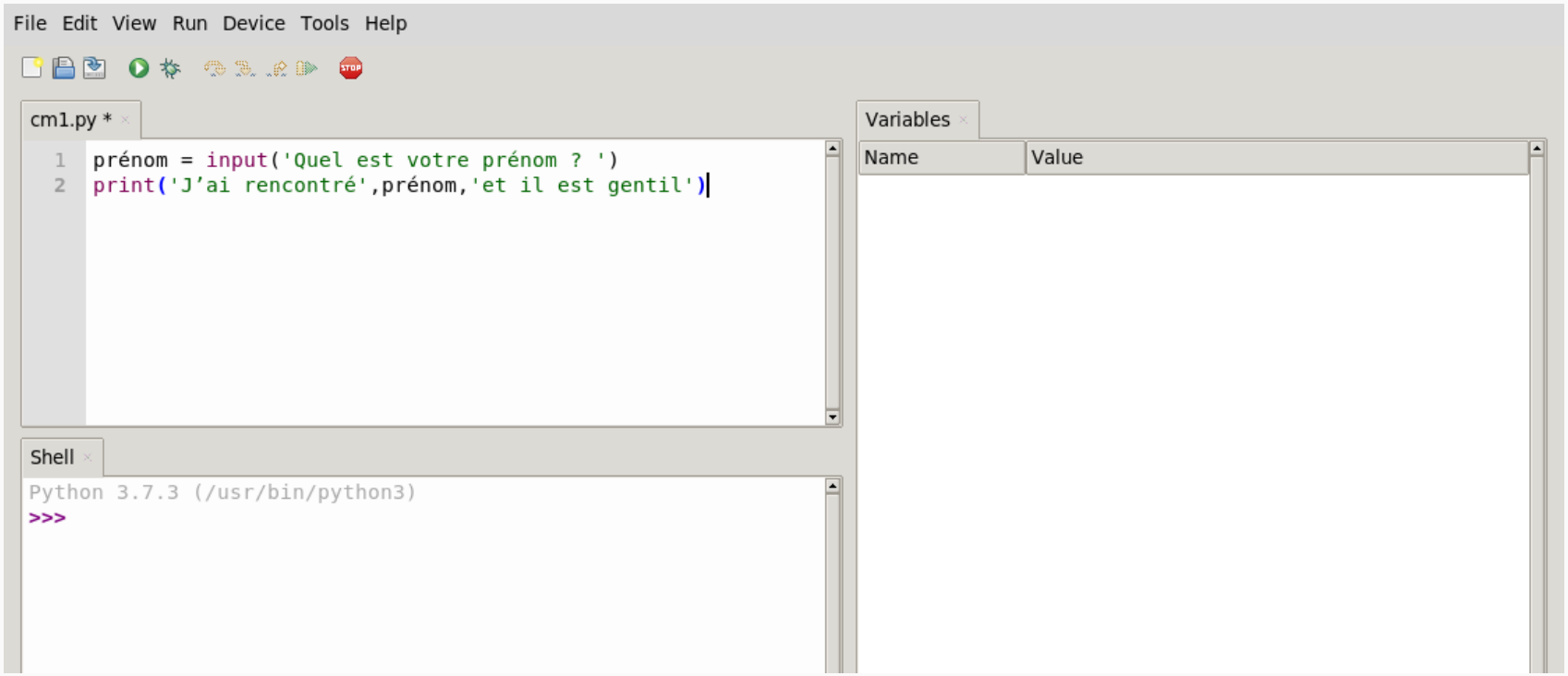
- On se place dans la fenêtre en haut à gauche.



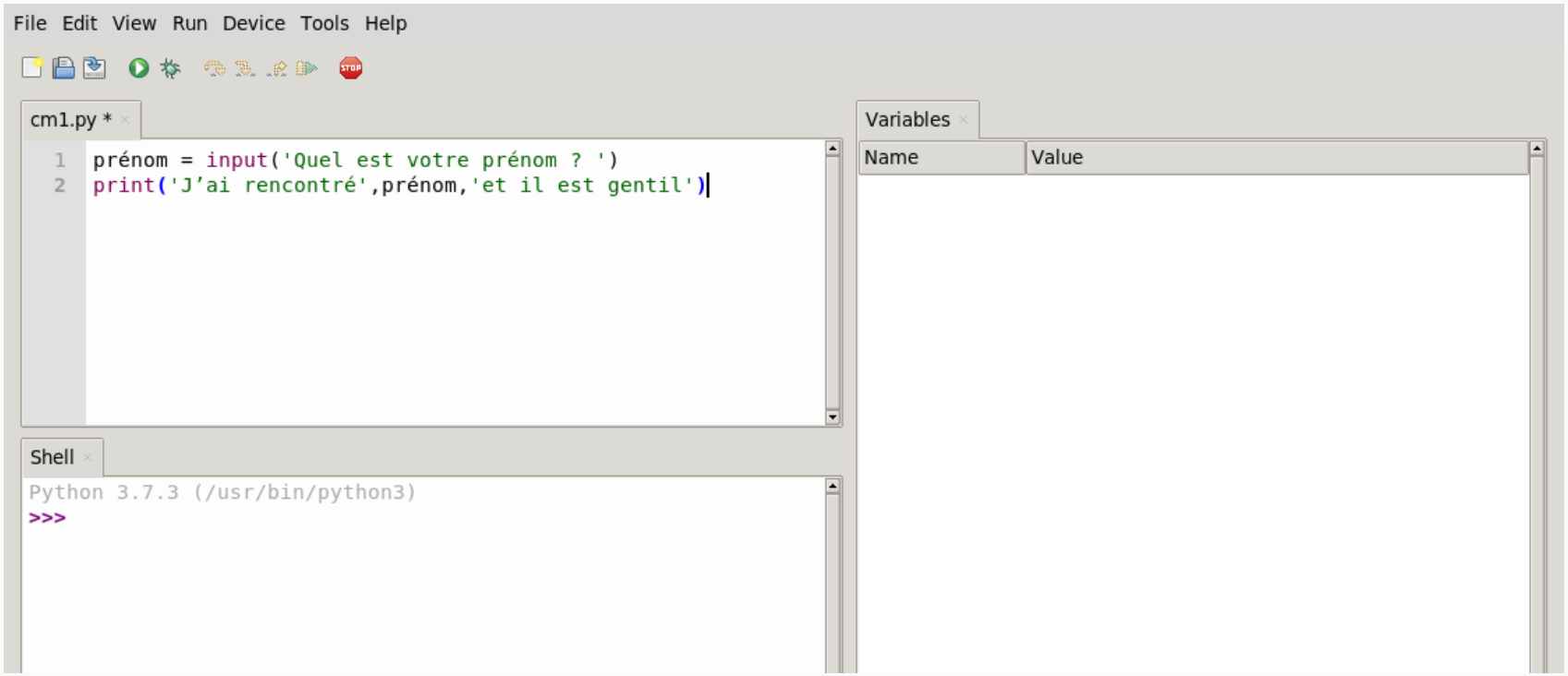
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S



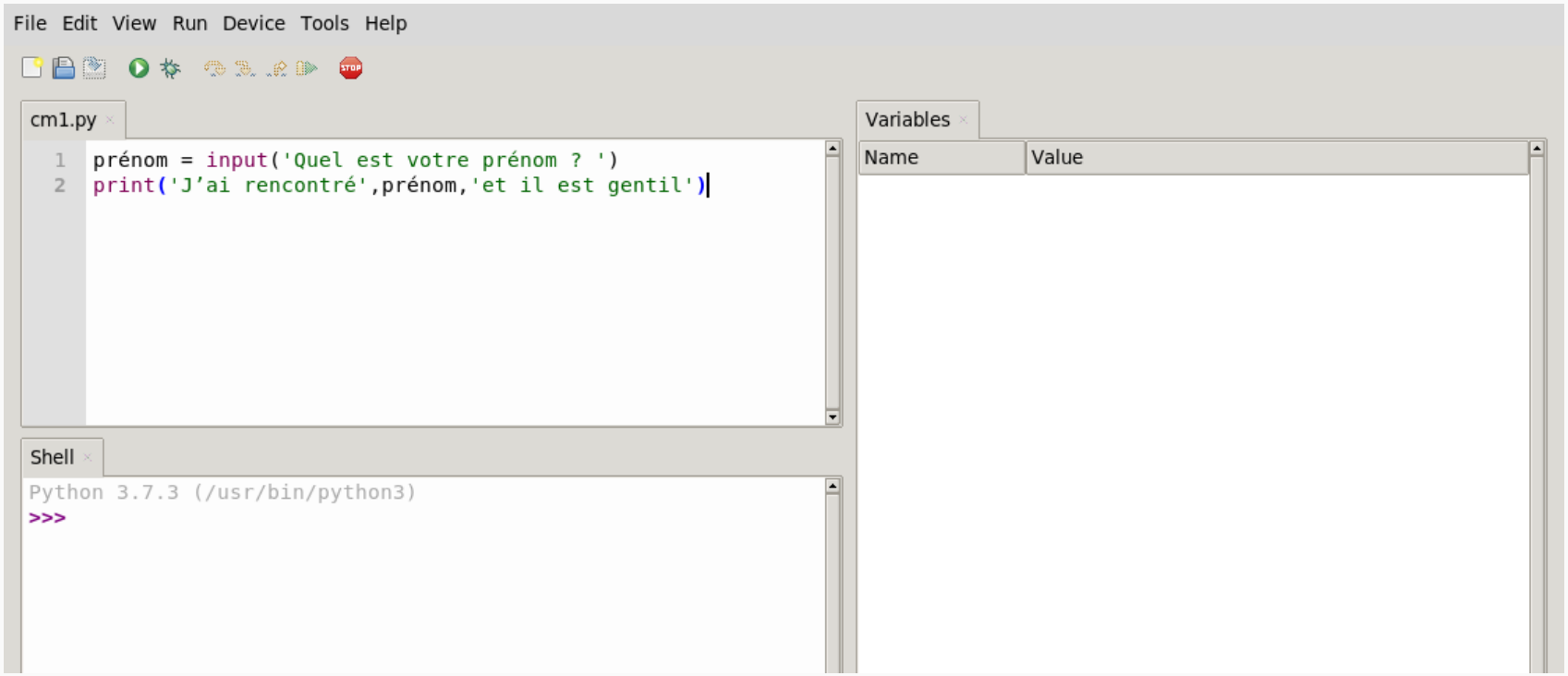
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script



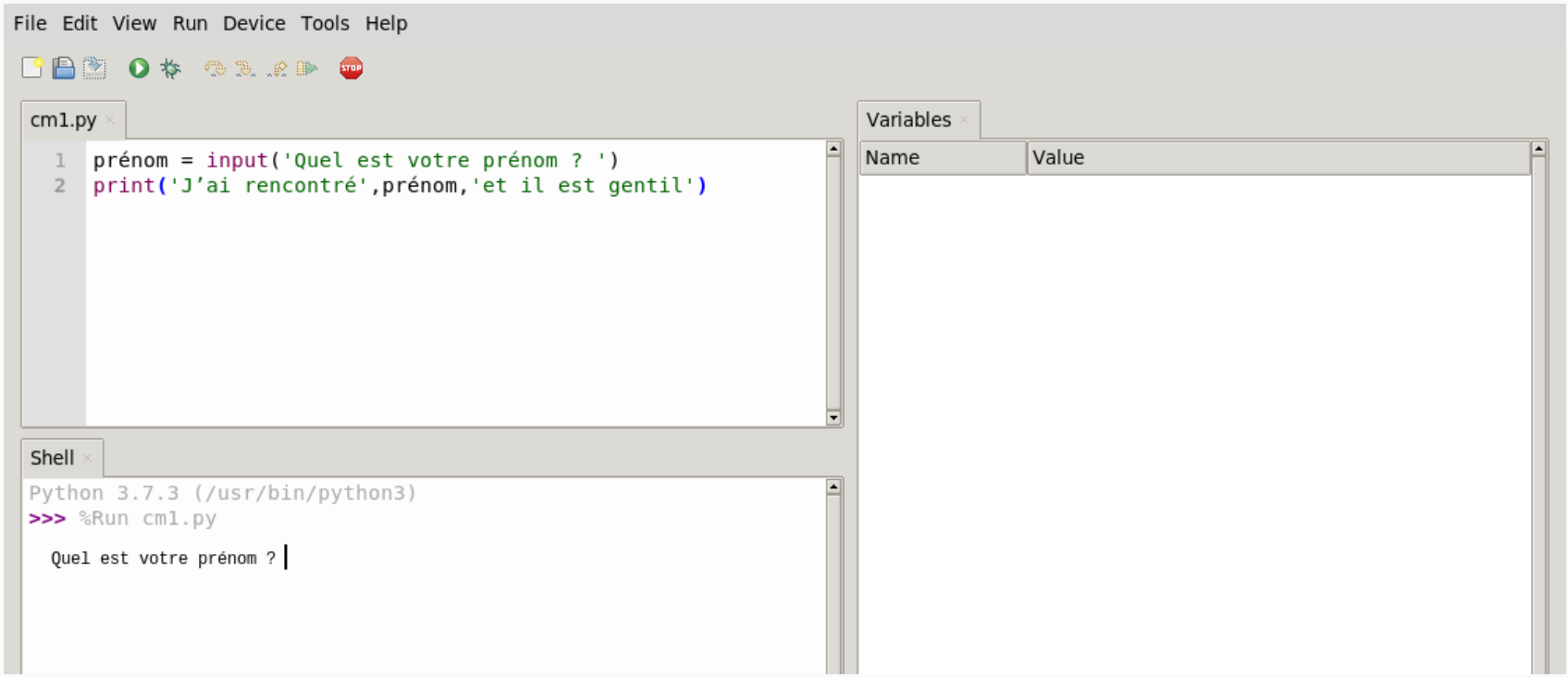
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder



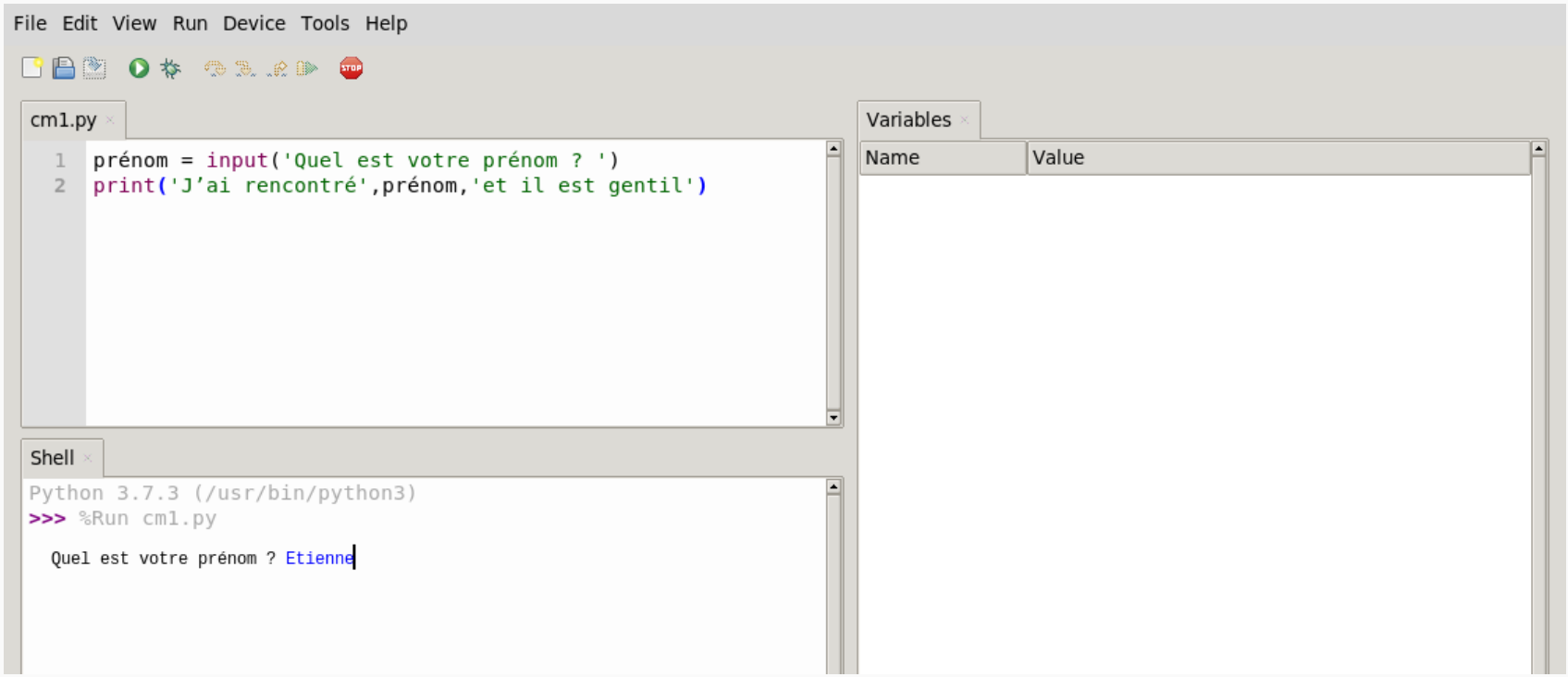
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît)



- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5



- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5



- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5

- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5
  - Puis on recommence :

- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5
  - Puis on recommence : ❶ on modifie le script

- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5
  - Puis on recommence : ❶ on modifie le script ❷ Ctrl + S

- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5
  - Puis on recommence : ❶ on modifie le script ❷ Ctrl + S ❸ F5

Les deux raccourcis les plus importants

- Ctrl + S : sauvegarder
- F5 : Exécuter le script

Mode Normal

- Ctrl + N : crée un nouveau script
- Ctrl + Z : annuler (si vous avez supprimé du code par accident)
- Ctrl + C : interrompt l'exécution (en cas de boucle infinie : cours 2)
- Ctrl + F2 : Relancer Python en vidant la mémoire

Mode Debug

- Ctrl + F5 : lancer le débuggage détaillé (ou ⇧ + F5 : débug. rapide)
- F7 : exécution pas à pas (si débuggage détaillé)
- F7 : exécution ligne par ligne (si débuggage rapide)
- F8 : exécuter le script jusqu'à la fin
- Ctrl + F8 : exécuter le script jusqu'au curseur

Créer un script `monscript.py` contenant les instructions suivantes

SCRIPT

```
a=4
b=a//2
c=b+2
print('c=',c)
a=2*b+a
b=a/2
print(b>=c)
```

- Exécutez-le
- Essayez les différents raccourcis clavier de la page précédente pour bien comprendre leurs utilités.

Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

L'instruction conditionnelle `if` permet d'exécuter une instruction seulement si une certaine condition est vérifiée.

SCRIPT

```
a = -2
if a > 0:
    print(a, 'est positif.') # Exécuté si a > 0
else:
    print(a, 'est négatif.') # Exécuté sinon
```

L'instruction conditionnelle `if` permet d'exécuter une instruction seulement si une certaine condition est vérifiée.

SCRIPT

```
a = -2
if a > 0:
    print(a, 'est positif.') # Exécuté si a > 0
else:
    print(a, 'est négatif.') # Exécuté sinon
```

SHELL

```
-2 est négatif.
```

Conditions : commande `if`

L'instruction conditionnelle `if` permet d'exécuter une instruction seulement si une certaine condition est vérifiée.

SCRIPT

```
a = -2
if a > 0:
    print(a, 'est positif.') # Exécuté si a > 0
else:
    print(a, 'est négatif.') # Exécuté sinon
```

SHELL

```
-2 est négatif.
```

- L'espace en début de ligne permet d'indiquer que l'on est dans le `if`
- Il s'agit de l'indentation qui s'obtient avec la touche tabulation ⇥

L'instruction conditionnelle `if` permet d'exécuter une instruction seulement si une certaine condition est vérifiée.

SCRIPT
```
a = -2
if a > 0:
    print(a, 'est positif.') # Exécuté si a > 0
else:
    print(a, 'est négatif.') # Exécuté sinon
```

SHELL
```
-2 est négatif.
```

- L'espace en début de ligne permet d'indiquer que l'on est dans le `if`
- Il s'agit de l'indentation qui s'obtient avec la touche tabulation ⇥

SCRIPT
```
if a > 0: # L'indentation est obligatoire !
print(a,'est positif')
```

Conditions : commande `if`

L'instruction conditionnelle `if` permet d'exécuter une instruction seulement si une certaine condition est vérifiée.

SCRIPT

```
a = -2
if a > 0:
    print(a, 'est positif.') # Exécuté si a > 0
else:
    print(a, 'est négatif.') # Exécuté sinon
```

SHELL

```
-2 est négatif.
```

- L'espace en début de ligne permet d'indiquer que l'on est dans le `if`
- Il s'agit de l'indentation qui s'obtient avec la touche tabulation ⇥

SCRIPT

```
if a > 0: # L'indentation est obligatoire !
print(a,'est positif')
```

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/olivier/python-2024/CM/C1/bug.py", line 3
    print(a,'est positif') # L'indentation est obligatoire !
    ^^^^^

IndentationError: expected an indented block after 'if'
statement on line 2
```

Conditions : Mot-clef `elif`

- On peut faire un `if` dans un autre `if`.

SCRIPT

```
a = 0
if a > 0:
    print('positif')
else:
    if a < 0:
        print('négatif') # deux tabulations !
    else:
        print('zéro')
```

- On peut faire un `if` dans un autre `if`.

SCRIPT

```
a = 0
if a > 0:
    print('positif')
else:
    if a < 0:
        print('négatif') # deux tabulations !
    else:
        print('zéro')
```

SHELL

```
zéro
```

- On peut faire un `if` dans un autre `if`.

SCRIPT
```
a = 0
if a > 0:
    print('positif')
else:
    if a < 0:
        print('négatif') # deux tabulations !
    else:
        print('zéro')
```

SHELL
```
zéro
```

- `else` + `if` peut se simplifier en `elif`

- On peut faire un `if` dans un autre `if`.

SCRIPT
```
a = 0
if a > 0:
    print('positif')
else:
    if a < 0:
        print('négatif') # deux tabulations !
    else:
        print('zéro')
```

SHELL
```
zéro
```

- `else` + `if` peut se simplifier en `elif`

SCRIPT
```
a = 0
if a > 0:
    print('positif')
elif a < 0:
    print('négatif')
else:
    print('zéro')
```

- On peut faire un `if` dans un autre `if`.

SCRIPT
```
a = 0
if a > 0:
    print('positif')
else:
    if a < 0:
        print('négatif') # deux tabulations !
    else:
        print('zéro')
```

SHELL
```
zéro
```

- `else` + `if` peut se simplifier en `elif`

SCRIPT
```
a = 0
if a > 0:
    print('positif')
elif a < 0:
    print('négatif')
else:
    print('zéro')
```

SHELL
```
zéro
```

- on peut utiliser le `if`, seul.
- on peut utiliser le `if` et le `else`.
- on peut utiliser le `if`, un ou plusieurs `elif` et le `else`.

SCRIPT

```
bloblo      # exécuté dans tous les cas
bloblo
if condition 1:
    blabla  # exécuté si la condition 1 est vraie
    blabla
elif condition 2:
    bleble  # exécuté si la condition 1 est fausse
    bleble  # et la condition 2 est vraie
elif condition 3:
    blublu  # exécuté si les conditions 1 et 2 sont fausses
    blublu  # et la condition 3 est vraie
else:
    blibli  # exécuté si les trois conditions sont fausses
    blibli
blybly      # exécuté dans tous les cas
blybly      # car il n'y a plus d'indentations
```

- Il n'y a jamais de conditions après le `else` !

Il y a trois opérateurs booléans :

-
-
-

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Il y a trois opérateurs booléans : négation

- La négation s'écrit `not`
-
-

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Il y a trois opérateurs booléans : négation ,« et logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
-

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL

```
>>>
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL
```
>>> not True
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL

```
>>> not True
False
>>>
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL

```
>>> not True
False
>>> (3>1) or (6>2)   # True or True
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL

```
>>> not True
False
>>> (3>1) or (6>2)   # True or True
True
>>>
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL

```
>>> not True
False
>>> (3>1) or (6>2)  # True or True
True
>>> True and not (1+1==2)  # True and False
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL

```
>>> not True
False
>>> (3>1) or (6>2)  # True or True
True
>>> True and not (1+1==2)  # True and False
False
```

Il y a trois opérateurs booléans : négation ,« et logique » , « ou logique »

- La négation s'écrit `not`
- `p and q` est vrai ⇔ p et q sont tous les deux vrais.
- `p or q` est faux ⇔ p et q sont tous les deux faux.

p	q	p and q	p or q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

SHELL
```
>>> not True
False
>>> (3>1) or (6>2)  # True or True
True
>>> True and not (1+1==2)  # True and False
False
```

Attention : le « ou logique » n'a pas le même sens qu'en français :

Mange ta soupe ou tu t'en prendras une !

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

```shell
>>>
```

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

SHELL
```
>>> x==3
```

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

SHELL
```
>>> x==3
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'x' is not defined
>>>
```

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

SHELL
```
>>> x==3
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'x' is not defined
>>> (1>0) or (x==3)
```

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

SHELL
```
>>> x==3
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'x' is not defined
>>> (1>0) or (x==3)
True
```

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

SHELL
```
>>> x==3
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'x' is not defined
>>> (1>0) or (x==3)
True
```

- `(x==3)` n'est pas évalué car « `True or` … » est toujours vrai.

- Lors d'un calcul booléen, les deux termes ne sont pas forcément calculés.

On parle d'évaluation paresseuse.

SHELL
```
>>> x==3
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'x' is not defined
>>> (1>0) or (x==3)
True
```

- `(x==3)` n'est pas évalué car « `True or` … » est toujours vrai.
- En pratique, on peut faire la même chose en mathématique.

$$0 \times \int_{log(2)}^{7\pi+3} \frac{3e^{4x\pi}}{\sqrt{13} \cdot cos(18)} \sum_{n=0}^{\infty} \frac{1}{n^2} dx = ?$$

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

SCRIPT

```
Donnez un nombre :
```

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

SCRIPT

```
Donnez un nombre :  27
```

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

SCRIPT

```
Donnez un nombre :  27
27 est impair
```

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

SCRIPT
```
Donnez un nombre :  27
27 est impair
```

SCRIPT
```
Donnez un nombre :
```

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

SCRIPT
```
Donnez un nombre :  27
27 est impair
```

SCRIPT
```
Donnez un nombre :  12
```

Écrire un script :

- qui demande à l'utilisateur un nombre `n` avec `input`
  - `input` renvoie une chaîne que l'on peut convertir en entier avec `int`
  - en effet on a : `int('23')==23`
  - il suffit donc de faire : `n=int(input('Message'))`
- qui affiche suivant la valeur de `n` :
  - `n est pair`
  - `n est impair`
- Bien sûr, on remplacera `n` par la valeur donnée par l'utilisateur.

Exemples d'utilisation

SCRIPT
```
Donnez un nombre : 27
27 est impair
```

SCRIPT
```
Donnez un nombre : 12
12 est pair
```

Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>>
```

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
```

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>>
```

- Documenté : https://docs.python.org/fr/3/library/math.html

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
```

- Documenté : https://docs.python.org/fr/3/library/math.html

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>>
```

- Documenté : https://docs.python.org/fr/3/library/math.html

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
```

- Documenté : https://docs.python.org/fr/3/library/math.html

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>>
```

- Documenté : https://docs.python.org/fr/3/library/math.html

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>> math.sqrt(2)
```

- Documenté : https://docs.python.org/fr/3/library/math.html

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>> math.sqrt(2)
1.4142135623730951
```

- Documenté : `https://docs.python.org/fr/3/library/math.html`
- Pour utiliser une fonction sans nom de module on l'importe directement.

SHELL
```
>>>
```

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

```
>>> min(abs(-3),10)
3
```
SHELL

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>> math.sqrt(2)
1.4142135623730951
```
SHELL

- Documenté : https://docs.python.org/fr/3/library/math.html
- Pour utiliser une fonction sans nom de module on l'importe directement.

```
>>> from math import sqrt
```
SHELL

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>> math.sqrt(2)
1.4142135623730951
```

- Documenté : https://docs.python.org/fr/3/library/math.html
- Pour utiliser une fonction sans nom de module on l'importe directement.

SHELL
```
>>> from math import sqrt
>>>
```

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>> math.sqrt(2)
1.4142135623730951
```

- Documenté : https://docs.python.org/fr/3/library/math.html
- Pour utiliser une fonction sans nom de module on l'importe directement.

SHELL
```
>>> from math import sqrt
>>> sqrt(25)
```

Les fonctions prédéfinies en Python

- Python définit aussi les fonctions `max`, `min`, `abs`, etc.

SHELL
```
>>> min(abs(-3),10)
3
```

- Les modules (`fractions`, `math`, etc) sont des « collections de fonctions ».
- Importer un module permet d'utiliser les fonctions du module

SHELL
```
>>> import math # Documentation : help('math')
>>> math.gcd(10,12)
2
>>> math.sqrt(2)
1.4142135623730951
```

- Documenté : https://docs.python.org/fr/3/library/math.html
- Pour utiliser une fonction sans nom de module on l'importe directement.

SHELL
```
>>> from math import sqrt
>>> sqrt(25)
5.0
```

- Pour définir la fonction $f : n \mapsto 2n + 1$.

SHELL
```
>>>
```

- Pour définir la fonction $f : n \mapsto 2n + 1$.

SHELL
```
>>> def f(n):

```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>>
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>>
```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>> f(2.5) # n n'est pas forcément entier
```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>> f(2.5) # n n'est pas forcément entier
6.0
>>>
```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>> f(2.5) # n n'est pas forcément entier
6.0
>>> f(2+2j) # j correspond au i des mathématiques
```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>> f(2.5) # n n'est pas forcément entier
6.0
>>> f(2+2j) # j correspond au i des mathématiques
(5+4j)
>>>
```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>> f(2.5) # n n'est pas forcément entier
6.0
>>> f(2+2j) # j correspond au i des mathématiques
(5+4j)
>>> f('Deux') #  le type doit être cohérent
```

- Pour définir la fonction $f : n \mapsto 2n+1$.

SHELL
```
>>> def f(n):
...     return 2*n+1 # Attention : indentation
>>> f(3)
7
```

- Le contenu de la fonction doit être indenté (tabulation)
- le mot-clef `return` définit le résultat de la fonction.

SHELL
```
>>> f(2.5) # n n'est pas forcément entier
6.0
>>> f(2+2j) # j correspond au i des mathématiques
(5+4j)
>>> f('Deux') #  le type doit être cohérent
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 1, in f
TypeError:  can only concatenate str (not "int") to str
```

Type des variables

Chaque variable a un type. Parmi les différents types possibles on trouve :

- `int` (entier) : 2 ; 5 ; -3
- `float` (décimaux) : 2.0 ; -5.3 ; .3
  - `.3` et `0.3` sont identiques
  - Le séparateur décimal est un point et non une virgule !
- `str` (chaîne de caractère) : `Coucou` ; `'++12à23'`
- `bool` (booléean) : `1>3` ou `True`
- `Nonetype` (instructions) : `None` ; `print('x=',3)`

Pour connaître le type, on peut utiliser la fonction `type`

```
>>>
```
SHELL

Type des variables

Chaque variable a un type. Parmi les différents types possibles on trouve :

- `int` (entier) : 2 ; 5 ; -3
- `float` (décimaux) : 2.0 ; -5.3 ; .3
  - `.3` et `0.3` sont identiques
  - Le séparateur décimal est un point et non une virgule !
- `str` (chaîne de caractère) : `Coucou` ; `'++12à23'`
- `bool` (booléean) : `1>3` ou `True`
- `Nonetype` (instructions) : `None` ; `print('x=',3)`

Pour connaître le type, on peut utiliser la fonction `type`

SHELL

```
>>> type(2)
```

Type des variables

Chaque variable a un type. Parmi les différents types possibles on trouve :

- `int` (entier) : 2 ; 5 ; -3
- `float` (décimaux) : 2.0 ; -5.3 ; .3
  - `.3` et `0.3` sont identiques
  - Le séparateur décimal est un point et non une virgule !
- `str` (chaîne de caractère) : `Coucou` ; `'++12à23'`
- `bool` (booléean) : `1>3` ou `True`
- `Nonetype` (instructions) : `None` ; `print('x=',3)`

Pour connaître le type, on peut utiliser la fonction `type`

SHELL

```
>>> type(2)
<class 'int'>
>>>
```

Type des variables

Chaque variable a un type. Parmi les différents types possibles on trouve :

- `int` (entier) : 2 ; 5 ; -3
- `float` (décimaux) : 2.0 ; -5.3 ; .3
  - `.3` et `0.3` sont identiques
  - Le séparateur décimal est un point et non une virgule !
- `str` (chaîne de caractère) : `Coucou` ; `'++12à23'`
- `bool` (booléean) : `1>3` ou `True`
- `Nonetype` (instructions) : `None` ; `print('x=',3)`

Pour connaître le type, on peut utiliser la fonction `type`

SHELL

```
>>> type(2)
<class 'int'>
>>> type(print('Vive le prof !'))
```

Chaque variable a un type. Parmi les différents types possibles on trouve :

- `int` (entier) : 2 ; 5 ; -3
- `float` (décimaux) : 2.0 ; -5.3 ; .3
  - `.3` et `0.3` sont identiques
  - Le séparateur décimal est un point et non une virgule !
- `str` (chaîne de caractère) : `Coucou` ; `'++12à23'`
- `bool` (booléean) : `1>3` ou `True`
- `Nonetype` (instructions) : `None` ; `print('x=',3)`

Pour connaître le type, on peut utiliser la fonction `type`

SHELL

```
>>> type(2)
<class 'int'>
>>> type(print('Vive le prof !'))
Vive le prof !
<class 'NoneType'>
```

SCRIPT

```
def absolue(n):
    if n>=0:
        return n # 2 indentations
    else:
        return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT

```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT
```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT
```
# Doublons les notes
def dn(x):
    if 2*x>20:
        return 20
    a=2*x
    return a
```

SHELL
```
>>>
```

SCRIPT
```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT
```
# Doublons les notes
def dn(x):
    if 2*x>20:
        return 20
    a=2*x
    return a
```

SHELL
```
>>> dn(4)
```

SCRIPT
```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT
```
# Doublons les notes
def dn(x):
    if 2*x>20:
        return 20
    a=2*x
    return a
```

SHELL
```
>>> dn(4)
8
>>>
```

SCRIPT
```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT
```
# Doublons les notes
def dn(x):
    if 2*x>20:
        return 20
    a=2*x
    return a
```

SHELL
```
>>> dn(4)
8
>>> dn(14)
```

- La dernière ligne ne sera pas exécutée si on est passé dans le `if` .

SCRIPT
```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- Les indentations (touche tab) sont **indispensables !**
- Les indentations du `if` et du `def` s'ajoutent.
- Le programme s'arrête lorsqu'il rencontre un `return`.

SCRIPT
```
# Doublons les notes
def dn(x):
    if 2*x>20:
        return 20
    a=2*x
    return a
```

SHELL
```
>>> dn(4)
8
>>> dn(14)
20
```

- La dernière ligne ne sera pas exécutée si on est passé dans le `if` .

Différence entre `print` et `return`

SCRIPT

```
def f(n):
    return 2*n+1
```

SCRIPT

```
def g(n):
    print(2*n+1)
```

SHELL

```
>>>
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.

Différence entre `print` et `return`

SCRIPT
```
def f(n):
    return 2*n+1
```

SCRIPT
```
def g(n):
    print(2*n+1)
```

SHELL
```
>>> g(3)
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.

Différence entre `print` et `return`

SCRIPT
```
def f(n):
    return 2*n+1
```

SCRIPT
```
def g(n):
    print(2*n+1)
```

SHELL
```
>>> g(3)
7
>>>
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.

Différence entre `print` et `return`

SCRIPT
```
def f(n):
    return 2*n+1
```

SCRIPT
```
def g(n):
    print(2*n+1)
```

SHELL
```
>>> g(3)
7
>>> 1 + f(3) # f(3) est un nombre qui vaut 7
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.

Différence entre `print` et `return`

SCRIPT
```
def f(n):
    return 2*n+1
```

SCRIPT
```
def g(n):
    print(2*n+1)
```

SHELL
```
>>> g(3)
7
>>> 1 + f(3) # f(3) est un nombre qui vaut 7
8
>>>
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.

Différence entre `print` et `return`

SCRIPT
```
def f(n):
    return 2*n+1
```

SCRIPT
```
def g(n):
    print(2*n+1)
```

SHELL
```
>>> g(3)
7
>>> 1 + f(3) # f(3) est un nombre qui vaut 7
8
>>> 1 + g(3) # g(3) n'est pas un nombre
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.

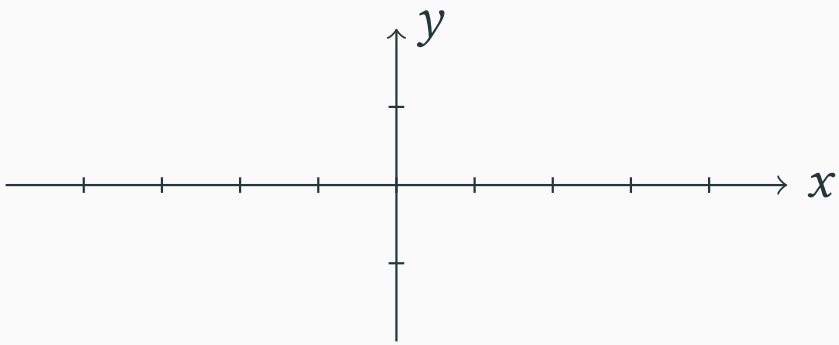
Différence entre `print` et `return`

SCRIPT
```
def f(n):
    return 2*n+1
```

SCRIPT
```
def g(n):
    print(2*n+1)
```

SHELL
```
>>> g(3)
7
>>> 1 + f(3) # f(3) est un nombre qui vaut 7
8
>>> 1 + g(3) # g(3) n'est pas un nombre
7
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unsupported operand type(s) for +: 'int' and
'NoneType'
```

- On peut faire plusieurs `print` dans une fonction.
- On peut faire seulement un `return` : c'est le résultat de la fonction.
  - On peut en faire plusieurs mais un seul sera exécuté.
  - Sans `return`, , la valeur de retour vaut `None`.



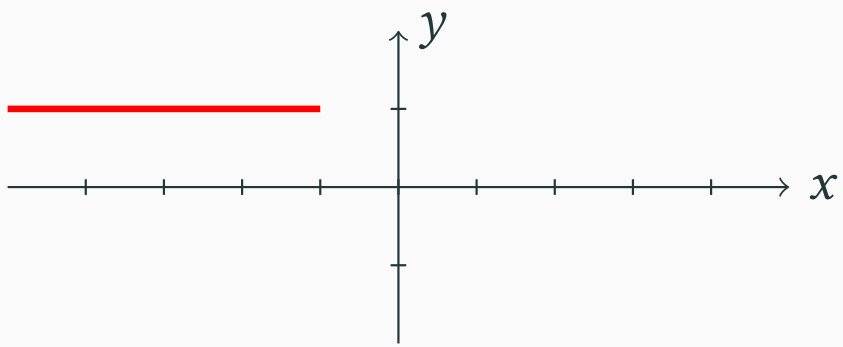
- Une fonction définie par morceaux.

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    elif x <= 1:
        return -x
    else:
        return -1
```

$\Leftrightarrow$

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    else :
        if x <= 1:
            return -x
        else:
            return -1
```

- Attention à bien indenter les blocs et sous-blocs



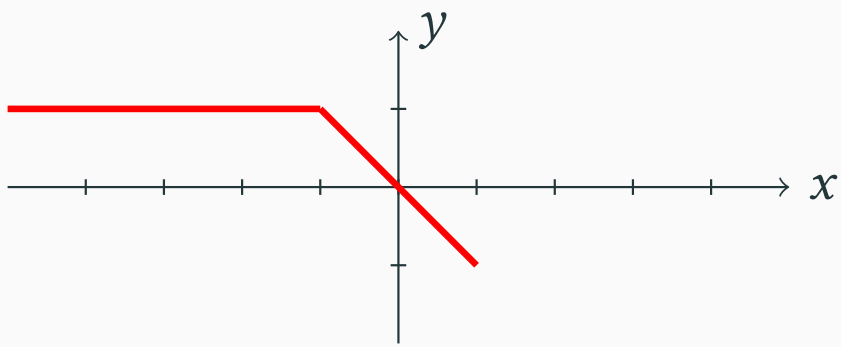
- Une fonction définie par morceaux.

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    elif x <= 1:
        return -x
    else:
        return -1
```

$\Leftrightarrow$

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    else :
        if x <= 1:
            return -x
        else:
            return -1
```

- Attention à bien indenter les blocs et sous-blocs



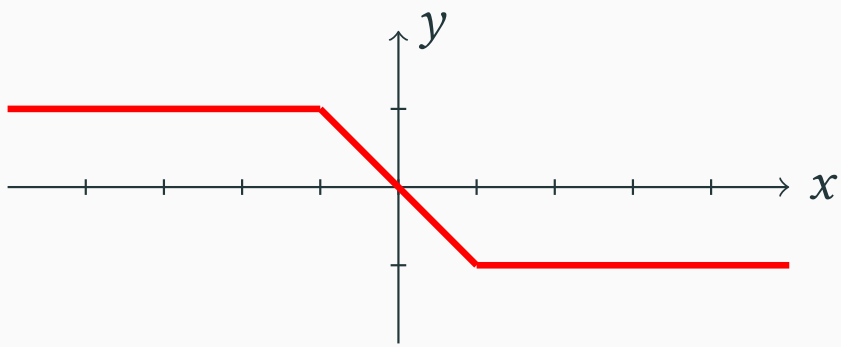
- Une fonction définie par morceaux.

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    elif x <= 1:
        return -x
    else:
        return -1
```

$\Leftrightarrow$

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    else :
        if x <= 1:
            return -x
        else:
            return -1
```

- Attention à bien indenter les blocs et sous-blocs



- Une fonction définie par morceaux.

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    elif x <= 1:
        return -x
    else:
        return -1
```

$\Leftrightarrow$

SCRIPT
```
def f(x):
    if x < -1:
        return 1
    else :
        if x <= 1:
            return -x
        else:
            return -1
```

- Attention à bien indenter les blocs et sous-blocs

SCRIPT

```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ?

SCRIPT
```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ? Testons la !

SCRIPT
```
print(plus_deux(10))
print(plus_deux(20))
```

SCRIPT
```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ? Testons la !

SCRIPT
```
print(plus_deux(10))
print(plus_deux(20))
```

SHELL
```
12
20
```

SCRIPT
```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ? Testons la !

SCRIPT
```
print(plus_deux(10))
print(plus_deux(20))
```

SHELL
```
12
20
```

- Pour l'instant aucuns soucis, mais testons avec des valeurs limites.

SCRIPT
```
print(plus_deux(10))
print(plus_deux(18.5))
print(plus_deux(20))
```

SCRIPT
```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ? Testons la !

SCRIPT
```
print(plus_deux(10))
print(plus_deux(20))
```

SHELL
```
12
20
```

- Pour l'instant aucuns soucis, mais testons avec des valeurs limites.

SCRIPT
```
print(plus_deux(10))
print(plus_deux(18.5))
print(plus_deux(20))
```

SHELL
```
12
20.5
20
```

SCRIPT
```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ? Testons la !

SCRIPT
```
print(plus_deux(10))
print(plus_deux(20))
```

SHELL
```
12
20
```

- Pour l'instant aucuns soucis, mais testons avec des valeurs limites.

SCRIPT
```
print(plus_deux(10))
print(plus_deux(18.5))
print(plus_deux(20))
```

SHELL
```
12
20.5
20
```

- Pour voir d'un coup d'œil si les tests ont fonctionné :

SCRIPT
```
print(plus_deux(10)==12)
print(plus_deux(18.5)==20)
print(plus_deux(20)==20)
```

SCRIPT
```
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20
```

- Comment savoir si notre fonction est correcte ? Testons la !

SCRIPT
```
print(plus_deux(10))
print(plus_deux(20))
```

SHELL
```
12
20
```

- Pour l'instant aucuns soucis, mais testons avec des valeurs limites.

SCRIPT
```
print(plus_deux(10))
print(plus_deux(18.5))
print(plus_deux(20))
```

SHELL
```
12
20.5
20
```

- Pour voir d'un coup d'œil si les tests ont fonctionné :

SCRIPT
```
print(plus_deux(10)==12)
print(plus_deux(18.5)==20)
print(plus_deux(20)==20)
```

SHELL
```
True
False
True
```

SCRIPT

```
1 def plus_deux(note):
2     if note < 19: # Jusqu'à 18 on peut ajouter 2
3         return note+2
4     else:
5         return 20
6
7 assert plus_deux(10) == 12
8 assert plus_deux(18.5) == 20
9 assert plus_deux(19) == 20
```

Tester ses fonctions : assert

SCRIPT

```python
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20

assert plus_deux(10) == 12
assert plus_deux(18.5) == 20
assert plus_deux(19) == 20
```

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/olivier/python-2024/CM/C1/cours1.py", line 8,
in <module>
    assert plus_deux(18.5) == 20
           ^^^^^^^^^^^^^^^^^^^^^

AssertionError
```

SCRIPT

```python
def plus_deux(note):
    if note < 19: # Jusqu'à 18 on peut ajouter 2
        return note+2
    else:
        return 20

assert plus_deux(10) == 12
assert plus_deux(18.5) == 20
assert plus_deux(19) == 20
```

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/olivier/python-2024/CM/C1/cours1.py", line 8,
in <module>
    assert plus_deux(18.5) == 20
           ^^^^^^^^^^^^^^^^^^^^^

AssertionError
```

- Comment corriger ?

SCRIPT

```python
def plus_deux(note):
    if note <= 18: # Jusqu'à 18 compris on peut ajouter 2
        return note+2
    else:
        return 20
```

- Dans l'idéal on écrit les tests avant d'écrire la fonction.

SCRIPT

```
1 def plus_deux(note):
2
3
4
5
6
7 assert plus_deux(10) == 12
8 assert plus_deux(18.5) == 20
9 assert plus_deux(19) == 20
```

- Dans l'idéal on écrit les tests avant d'écrire la fonction.
  - Permet de s'assurer que l'on sait ce que doit faire la fonction.

SCRIPT

```
1 def plus_deux(note):
2
3
4
5
6
7 assert plus_deux(10) == 12
8 assert plus_deux(18.5) == 20
9 assert plus_deux(19) == 20
```

- Dans l'idéal on écrit les tests avant d'écrire la fonction.
  - Permet de s'assurer que l'on sait ce que doit faire la fonction.
  - Permet de réfléchir aux cas limites et problématiques avant.

SCRIPT

```
1 def plus_deux(note):
2
3
4
5
6
7 assert plus_deux(10) == 12
8 assert plus_deux(18.5) == 20
9 assert plus_deux(19) == 20
```

- Dans l'idéal on écrit les tests avant d'écrire la fonction.
  - Permet de s'assurer que l'on sait ce que doit faire la fonction.
  - Permet de réfléchir aux cas limites et problématiques avant.

SCRIPT

```
def plus_deux(note):
    if note <= 18: # Jusqu'à 18 compris on peut ajouter 2
        return note+2
    else:
        return 20

assert plus_deux(10) == 12
assert plus_deux(18.5) == 20
assert plus_deux(19) == 20
```

- Dans l'idéal on écrit les tests avant d'écrire la fonction.
  - Permet de s'assurer que l'on sait ce que doit faire la fonction.
  - Permet de réfléchir aux cas limites et problématiques avant.

SCRIPT

```
1 def plus_deux(note):
2     if note <= 18: # Jusqu'à 18 compris on peut ajouter 2
3         return note+2
4     else:
5         return 20
6
7 assert plus_deux(10) == 12
8 assert plus_deux(18.5) == 20
9 assert plus_deux(19) == 20
```

- Lorsqu'on exécute le code :
  - Si tout est bon, rien ne s'affiche
  - Sinon, une erreur affiche le test échoué avec sa ligne.

- Dans l'idéal on écrit les tests avant d'écrire la fonction.
  - Permet de s'assurer que l'on sait ce que doit faire la fonction.
  - Permet de réfléchir aux cas limites et problématiques avant.

SCRIPT

```
1 def plus_deux(note):
2     if note <= 18: # Jusqu'à 18 compris on peut ajouter 2
3         return note+2
4     else:
5         return 20
6
7 assert plus_deux(10) == 12
8 assert plus_deux(18.5) == 20
9 assert plus_deux(19) == 20
```

- Lorsqu'on exécute le code :
  - Si tout est bon, rien ne s'affiche
  - Sinon, une erreur affiche le test échoué avec sa ligne.

- Ne marche qu'avec les fonctions qui retourne un résultat.
  - ne permet pas de tester lorsqu'il y a des `print`.

Sommaire

- Partie I. À propos
- Partie II. Entiers
- Partie III. Variables
- Partie IV. Écrire des scripts
- Partie V. Conditions
- Partie VI. Fonctions
- Partie VII. Exemples
- Partie VIII. Table des matières

On cherche à savoir si deux entiers n'ont que 1 comme diviseur commun

Encore un exemple de fonction

On cherche à savoir si deux entiers n'ont que 1 comme diviseur commun

SCRIPT

```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True
    else:
        return False
```

On cherche à savoir si deux entiers n'ont que 1 comme diviseur commun

SCRIPT
```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True
    else:
        return False
```

SCRIPT
```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True # Le programme se termine là
    return False # ou là
```

On cherche à savoir si deux entiers n'ont que 1 comme diviseur commun

SCRIPT
```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True
    else:
        return False
```

SCRIPT
```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True # Le programme se termine là
    return False # ou là
```

SCRIPT
```
def premiers_entre_eux(p,q):
    return gcd(p,q)==1
```

Encore un exemple de fonction

On cherche à savoir si deux entiers n'ont que 1 comme diviseur commun

SCRIPT
```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True
    else:
        return False
```

SCRIPT
```
def premiers_entre_eux(p,q):
    if gcd(p,q)==1:
        return True # Le programme se termine là
    return False # ou là
```

SCRIPT
```
def premiers_entre_eux(p,q):
    return gcd(p,q)==1
```

Mêmes résultats mais codés de façon plus ou moins élégante

- élégance ; c'est-à-dire efficacité et lisibilité

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT

```
from random import randint
```

- pour `a` $\leq$ `b`, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle [`a`, `b`]

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT

```
from random import randint
```

- pour `a` $\leq$ `b`, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle [`a`, `b`]
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour `a` $\leq$ `b`, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle [`a`, `b`]
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>>
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour `a` $\leq$ `b`, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle [`a`, `b`]
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>>
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>>
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>> jet35()
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>> jet35()
5
>>>
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>> jet35()
5
>>> # Parenthèses obligatoires !
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>> jet35()
5
>>> # Parenthèses obligatoires !
>>>
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a, b]$
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>> jet35()
5
>>> # Parenthèses obligatoires !
>>> jet35
```

Aléatoire : générer des entiers au hasard

- On peut importer la fonction `randint` du module `random` :

SCRIPT
```
from random import randint
```

- pour `a` $\leq$ `b`, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle [`a`, `b`]
- exemple : `2*randint(0,10)` renvoie un nombre pair aléatoire de $[0, 20]$
- Un fonction `jet35()` qui tire 3 ou 5 au hasard :

SCRIPT
```
def jet35():
    if randint(0,1)==0:
        return 3
    else:
        return 5
```

SHELL
```
>>> jet35()
5
>>> jet35()
3
>>> jet35()
5
>>> # Parenthèses obligatoires !
>>> jet35
<function jet35 at 0x7f73113932e0>
```

- On prend comme convention pile↔0 et face↔1

SCRIPT

```
from random import randint

pronostic = input('Pile ou face, humain ?')
lancer = randint(0,1)

if pronostic == 'pile' and lancer == 0:
    print('Gagné !')
elif pronostic == 'face' and lancer == 1:
    print('Gagné !')
else:
    print('Perdu…')
```

- On prend comme convention pile↔0 et face↔1

SCRIPT

```
from random import randint

pronostic = input('Pile ou face, humain ?')
lancer = randint(0,1)

if pronostic == 'pile' and lancer == 0:
    print('Gagné !')
elif pronostic == 'face' and lancer == 1:
    print('Gagné !')
else:
    print('Perdu…')
```

- Ce code fonctionne mais est peu élégant
  - Deux fois `print('Gagné !')`
  - Tests laborieux
  - On peut faire mieux !

Mettons les valeurs `'pile'` ou `'face'` directement dans la variable `lancer`

SCRIPT

```
from random import randint

pronostic = input('Pile ou face, humain ?')

if randint(0,1) == 0:
    lancer = 'pile'
else:
    lancer = 'face'

if pronostic == lancer:
    print('Gagné !')
else:
    print('Perdu…')
```

Mettons les valeurs `'pile'` ou `'face'` directement dans la variable `lancer`

SCRIPT

```
from random import randint

# Fonction auxiliaire pour rendre le code plus lisible
def pile_ou_face():
    if randint(0,1) == 0:
        return 'pile'
    else:
        return 'face'


pronostic = input('Pile ou face, humain ?')
lancer = pile_ou_face()

if pronostic == lancer:
    print('Gagné !')
else:
    print('Perdu…')
```

Merci pour votre attention
Questions

?

Cours 1 — Variables, fonctions et conditions

Partie 1. À propos

Communication

Exemple

Organisation

Quelques références pour ce cours

Qu'est-ce que la programmation ?

Python 3

Objectif du cours

Thonny : un éditeur pour Python

Partie 2. Entiers

Session interactive

Opérations sur les entiers

Divisions euclidiennes et modulos

Exemple : Heures et minutes

Opérations et priorités

Taille des entiers

Exercices

Correction

Partie 3. Variables

Affectation

Espionner les variables avec Thonny

Instructions et expressions

Échange de deux variables

Booléans (Vrai et Faux) et comparaisons

Exercices

Correction

Partie 4. Écrire des scripts

La commande d'affichage : `print`

Les chaînes de caractères

Les scripts python

Thonny : écrire des scripts

Thonny : raccourcis clavier

Exercices

Partie 5. Conditions

Conditions : commande `if`

Conditions : Mot-clef `elif`

Conditions : synthèse

Calculs booléans

Évaluation paresseuse

Exercices

Partie 6. Fonctions

Les fonctions prédéfinies en Python

Définir sa propre fonction

Type des variables

Un exemple : la fonction « valeur absolue »

Différence entre `print` et `return`

Autre exemple de fonctions

Tester ses fonctions

Tester ses fonctions : assert

Tester ses fonctions : bilan

Partie 7. Exemples

Encore un exemple de fonction

Aléatoire : générer des entiers au hasard

Pile ou face

Pile ou face, le retour !

Pile ou face, solution ultime

Partie 8. Table des matières