Base de l'informatique 1

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

- 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.

```
>>> 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
```
SHELL

- 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.

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

```
>>>
```
SHELL

- Une variable est une boîte qui contient une valeur.

```
>>> 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
```
SHELL

- 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.

```
>>> 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
```
SHELL

- Une variable est une boîte qui contient une valeur.

```
>>> 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
```
SHELL

- 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.

```
>>> 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
>>>
```
SHELL

- Une variable est une boîte qui contient une valeur.

```
>>> 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
```
SHELL

- 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.

```
>>> 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
```
SHELL

- Une variable est une boîte qui contient une valeur.

```
>>> 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
```
SHELL

- 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.

```
>>> 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
```
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

- 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
```

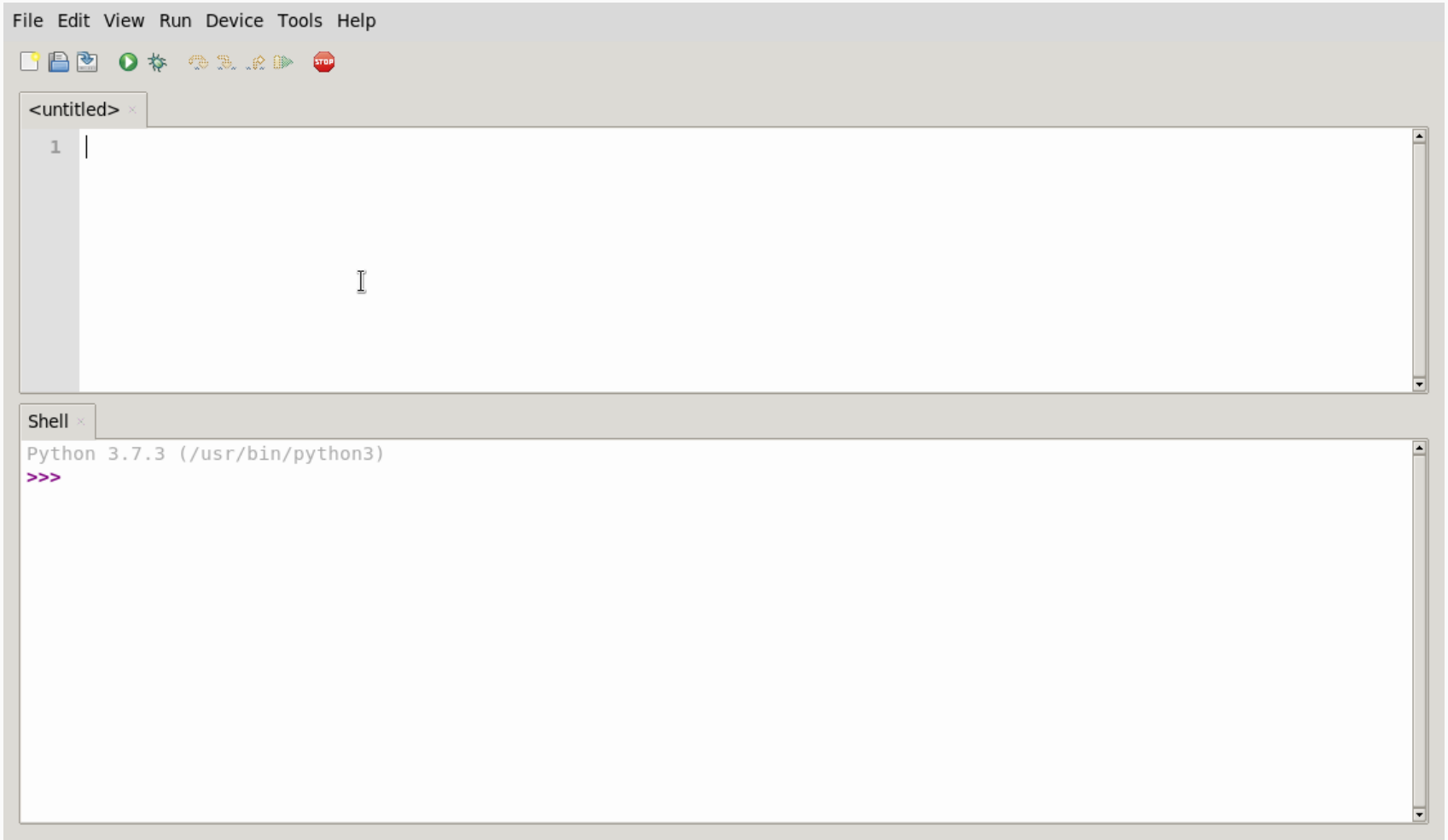
- Il ne faut pas confondre les instructions et les expressions. Une instruction est exécutée. Une expression est calculée.
  Exemple : a vaut 2 et b vaut 3.

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

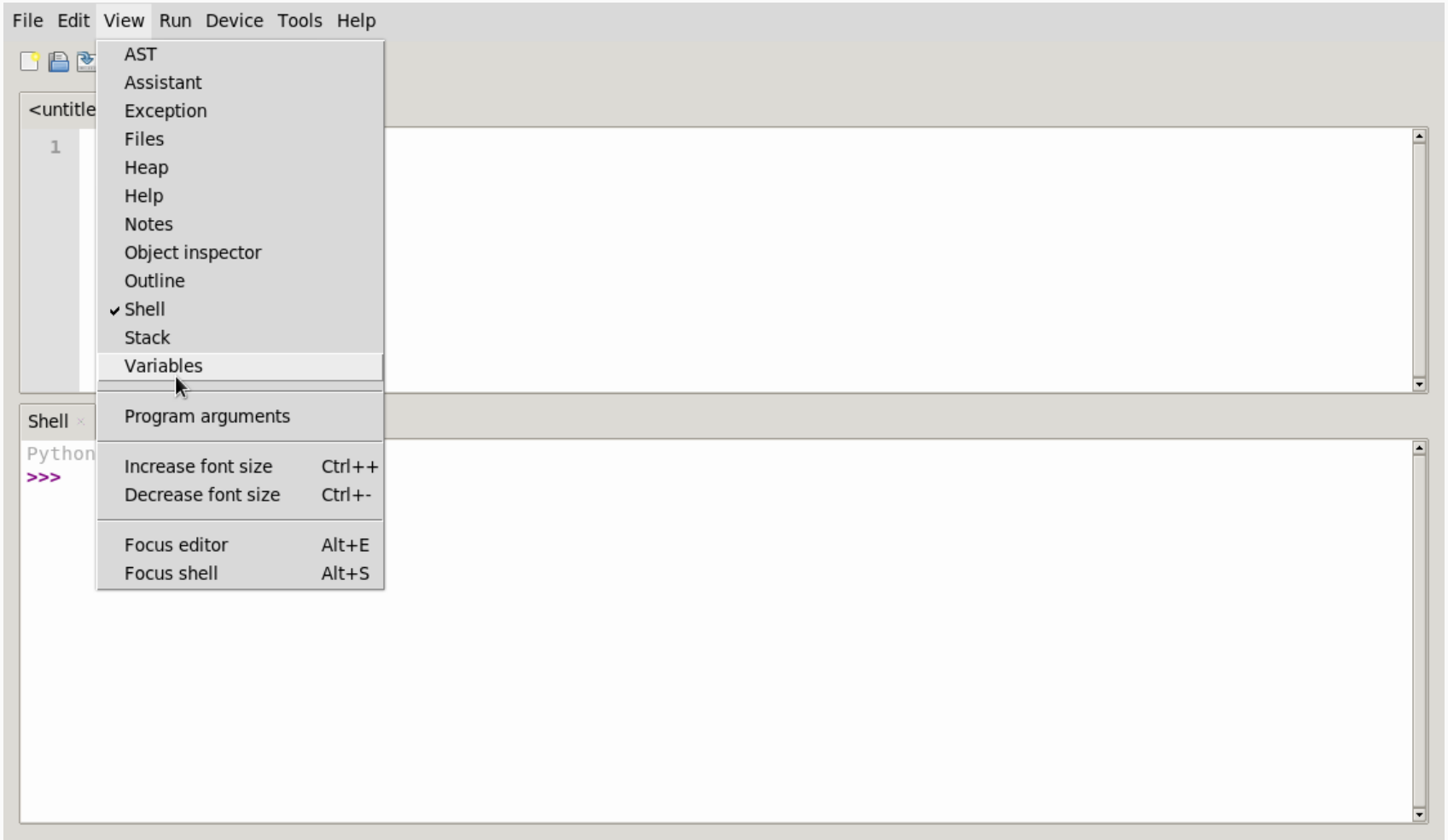
- Une variable peut changer de valeur.

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
```

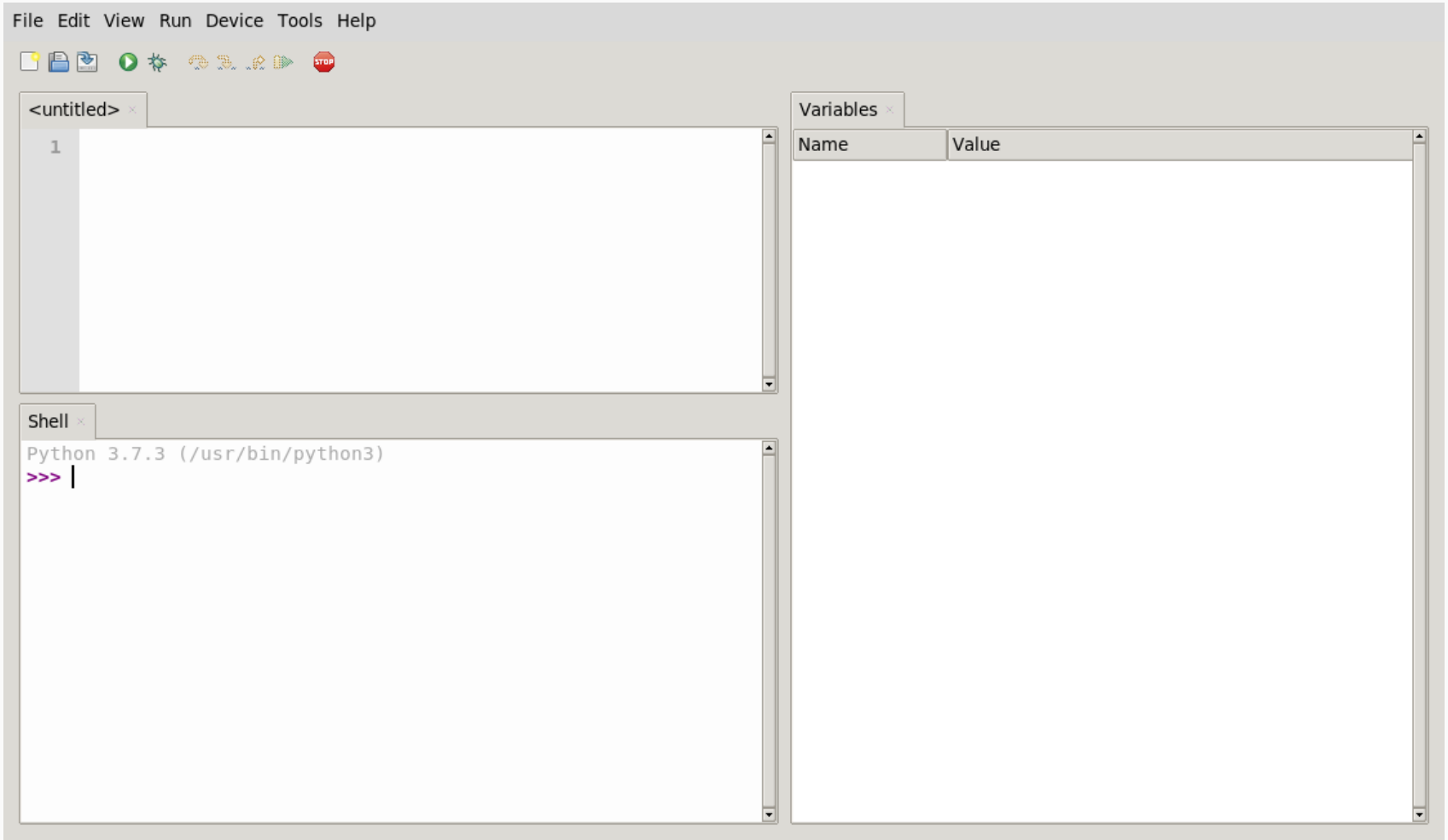
- Si nécessaire on ouvre l'onglet Variables (menu View)



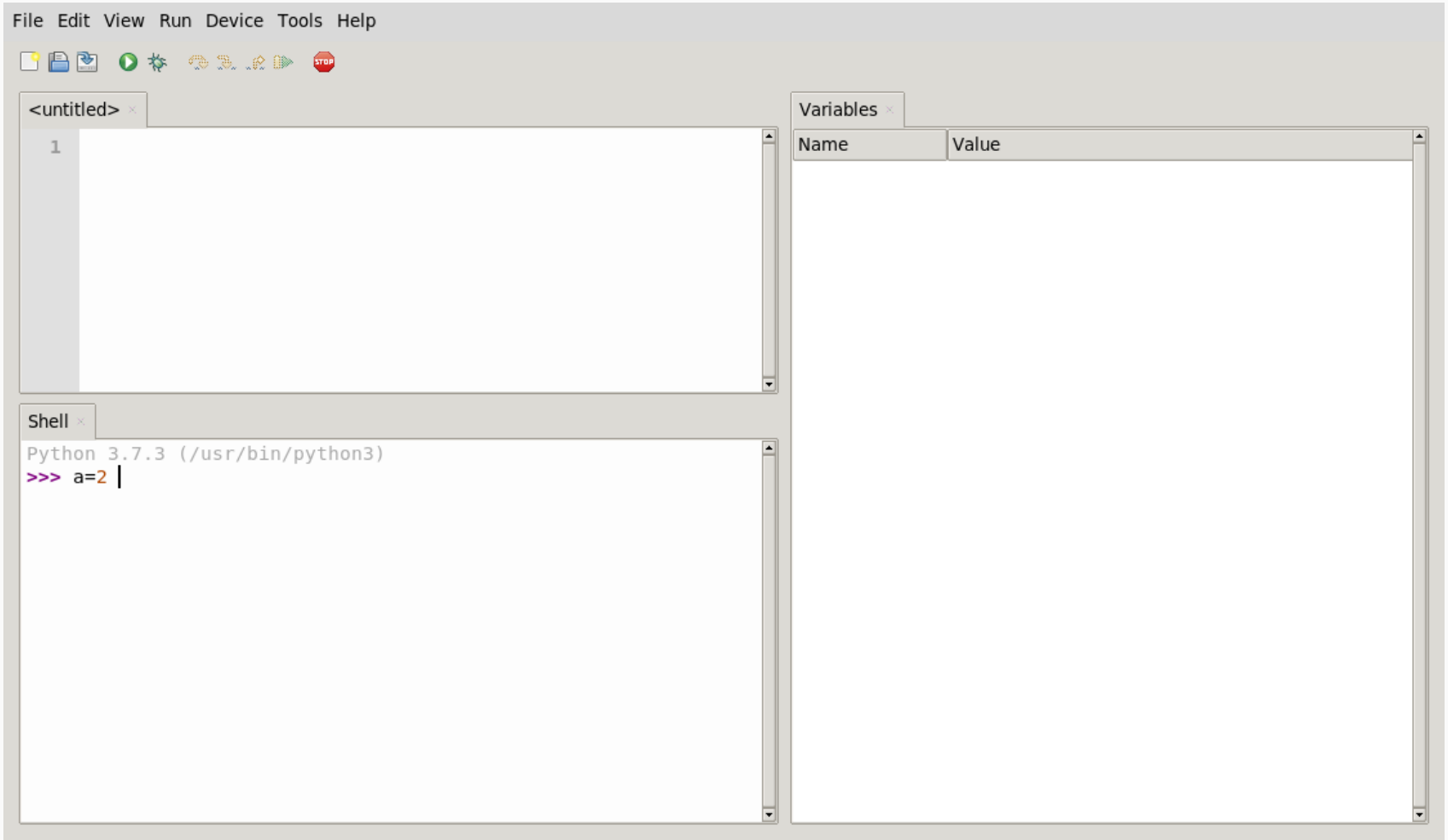
- Si nécessaire on ouvre l'onglet Variables (menu View)



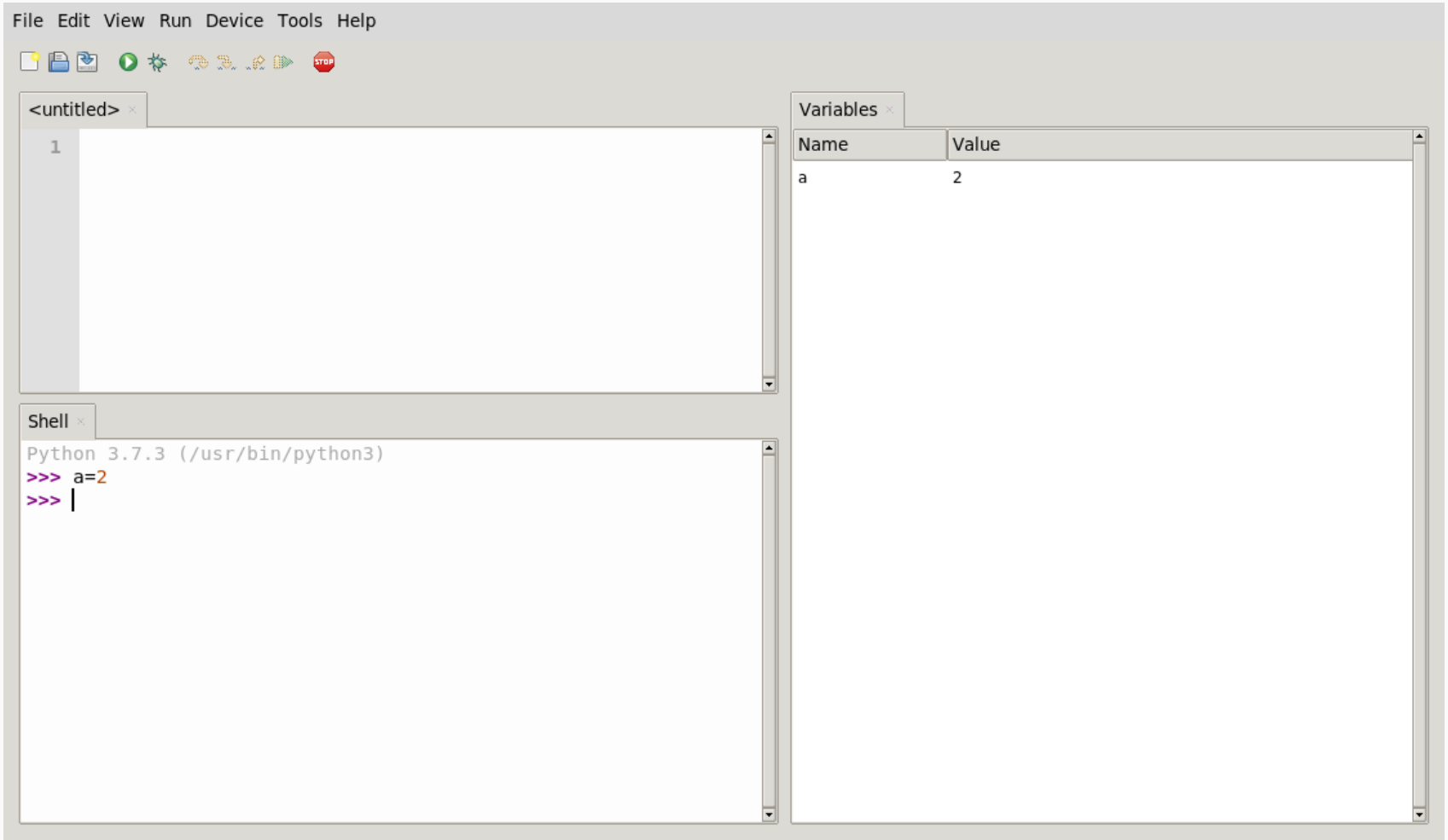
- Si nécessaire on ouvre l'onglet Variables (menu View)



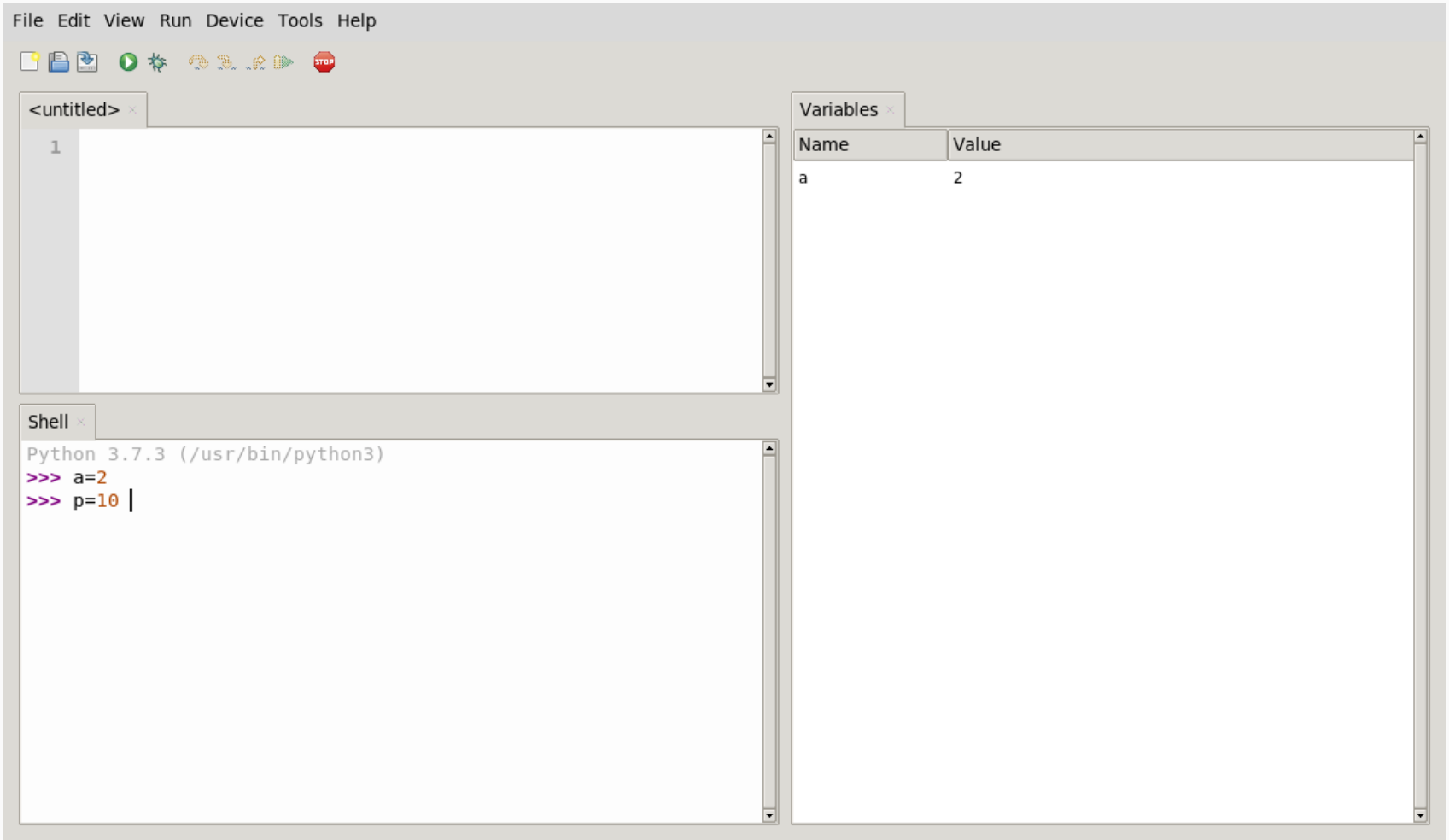
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell



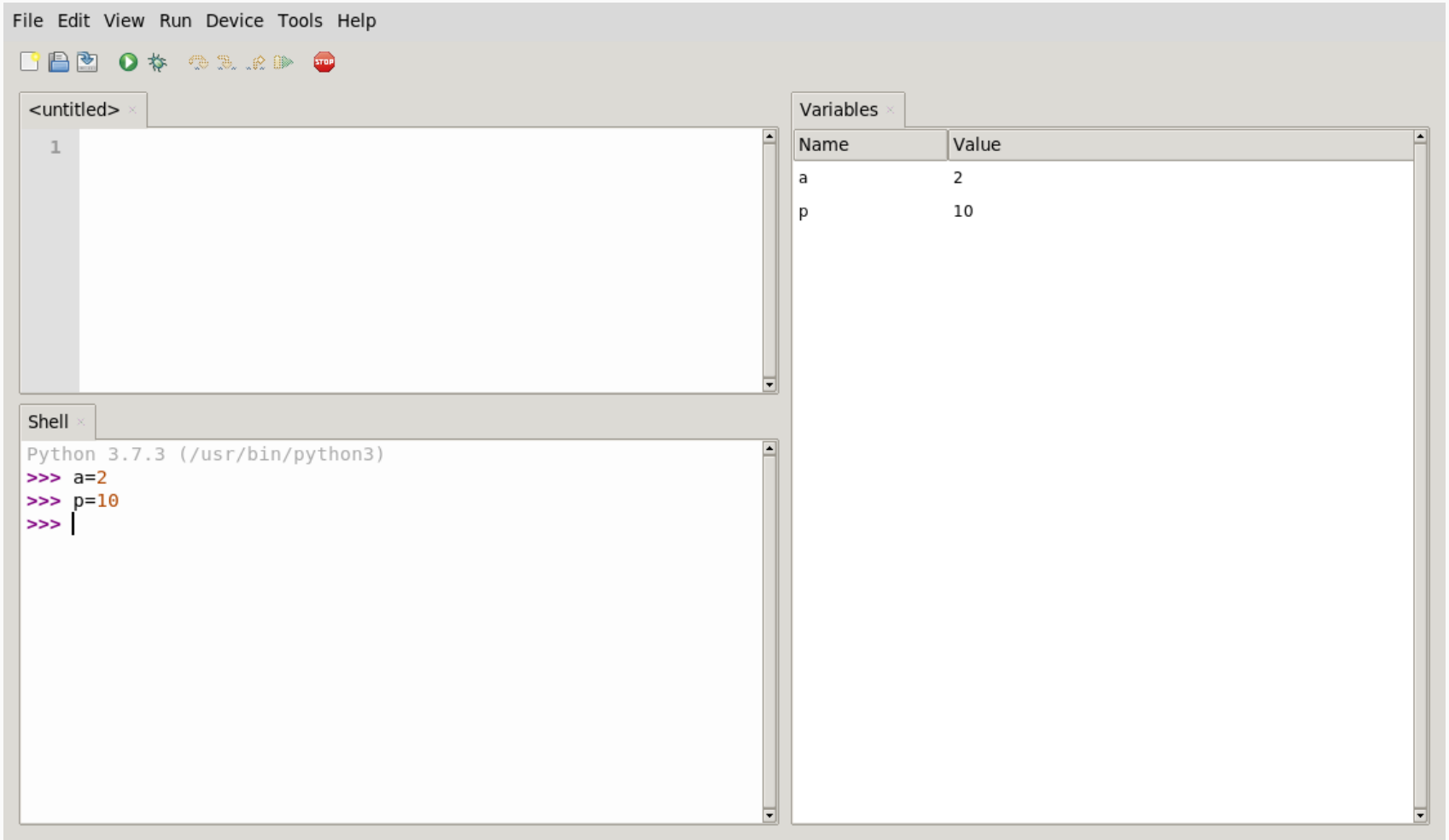
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell



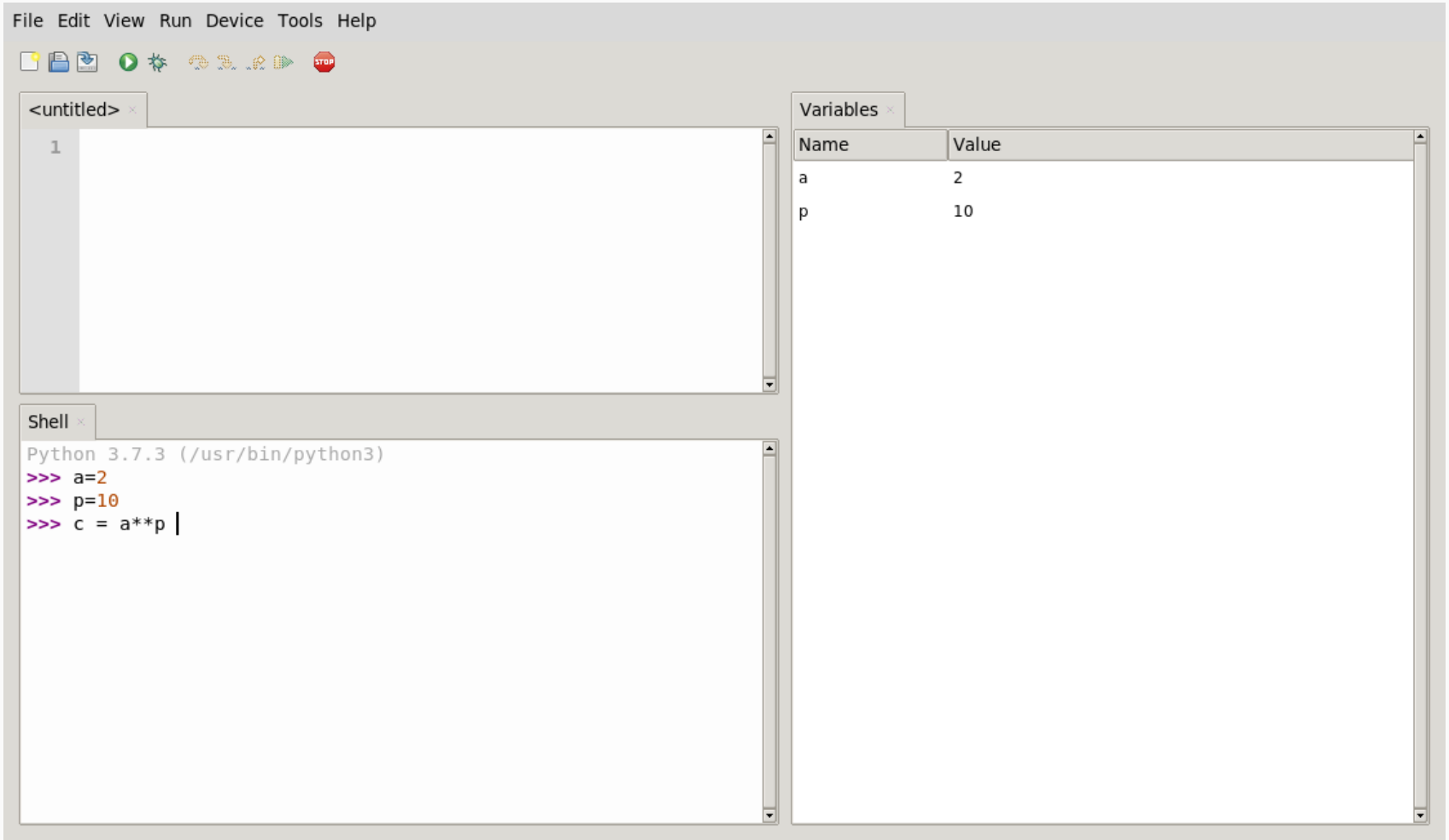
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



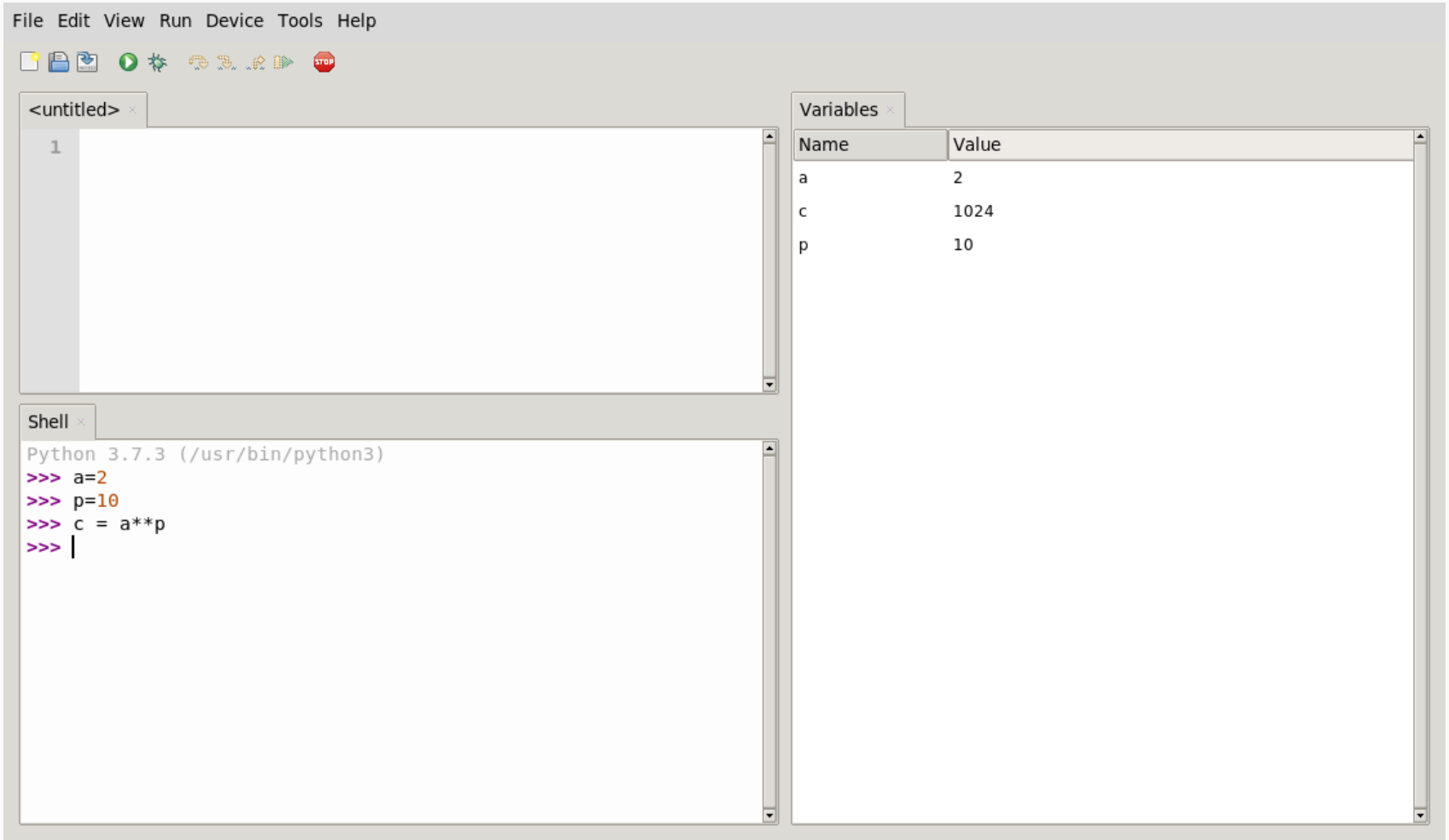
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



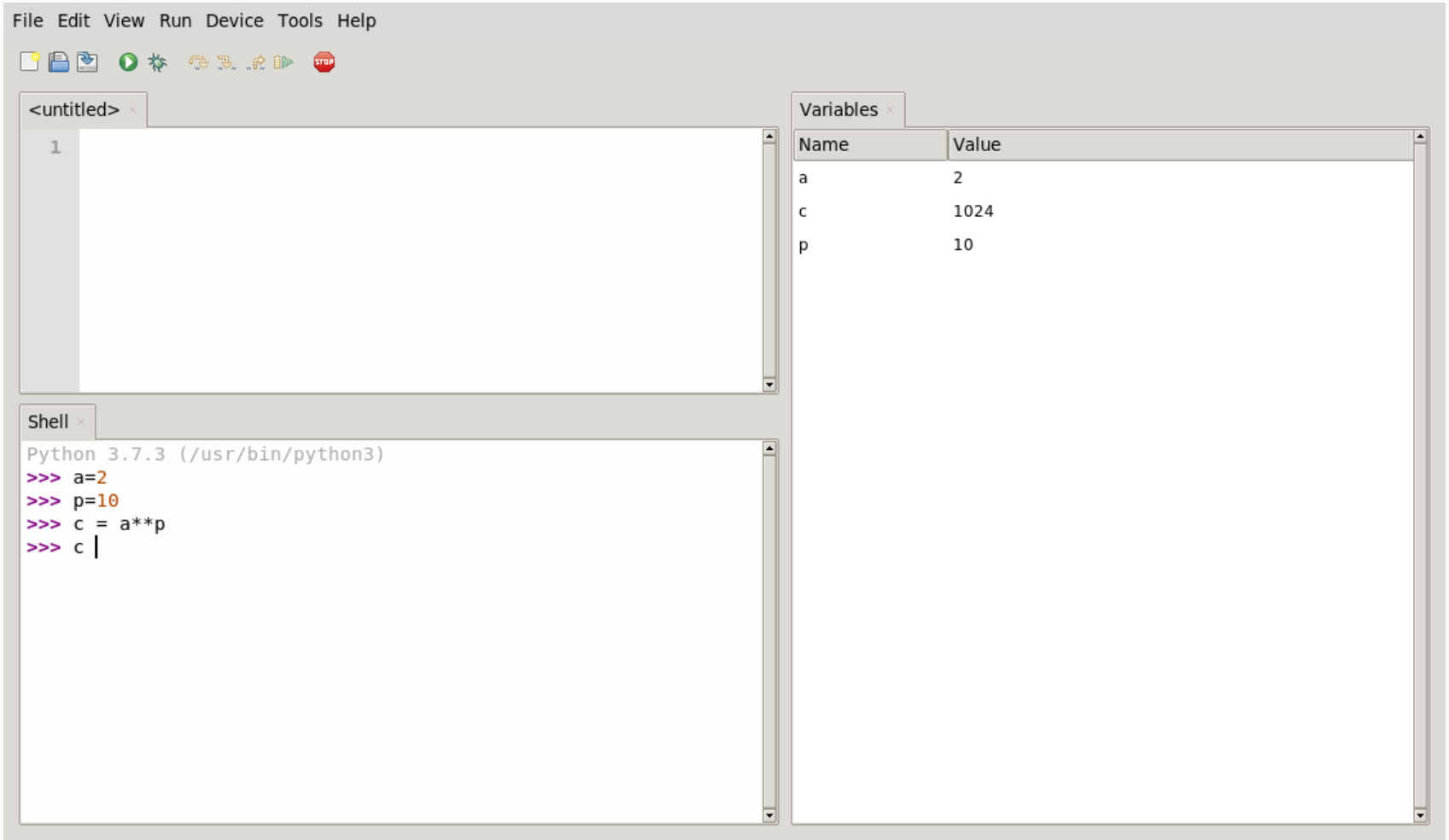
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



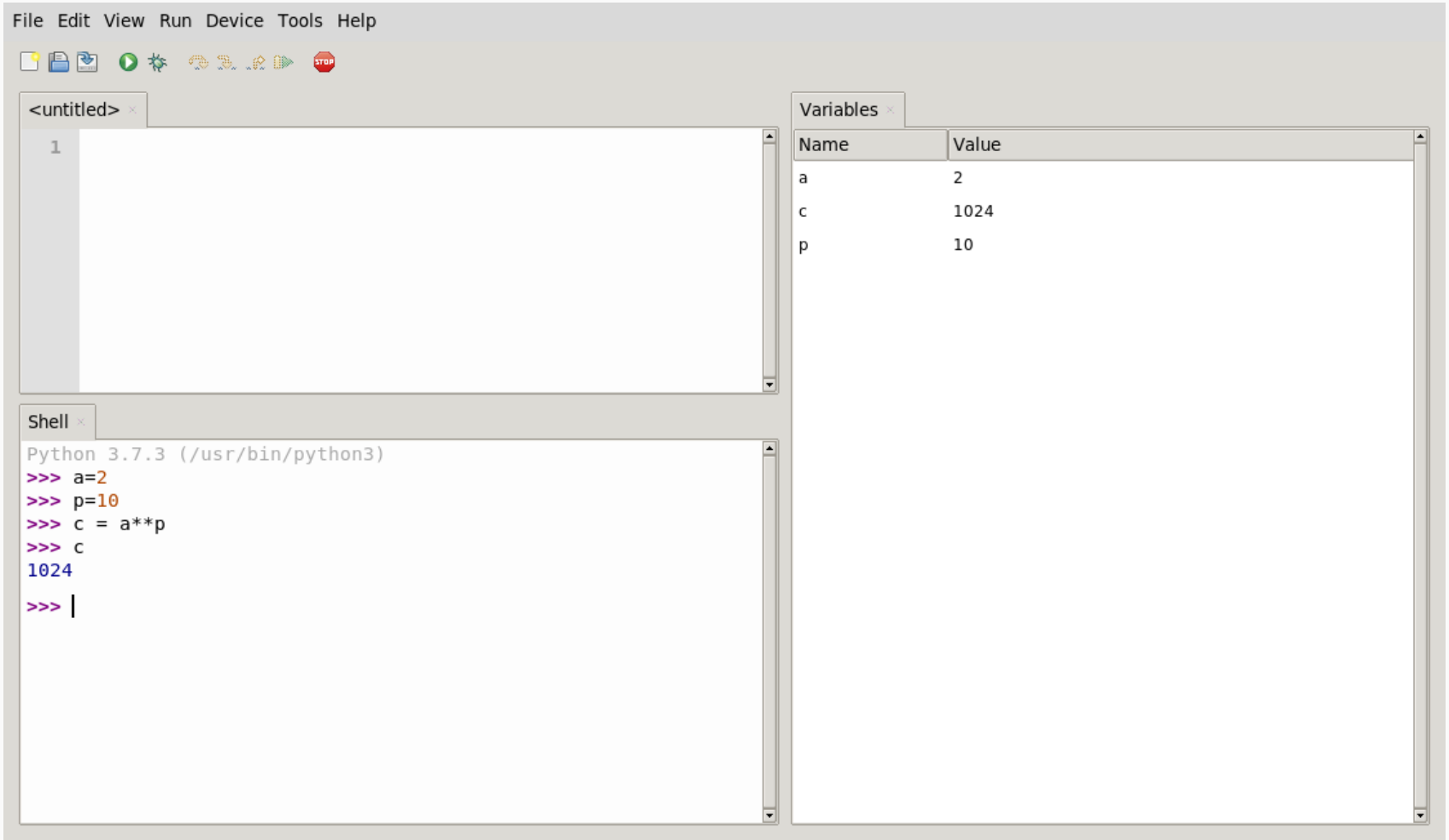
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



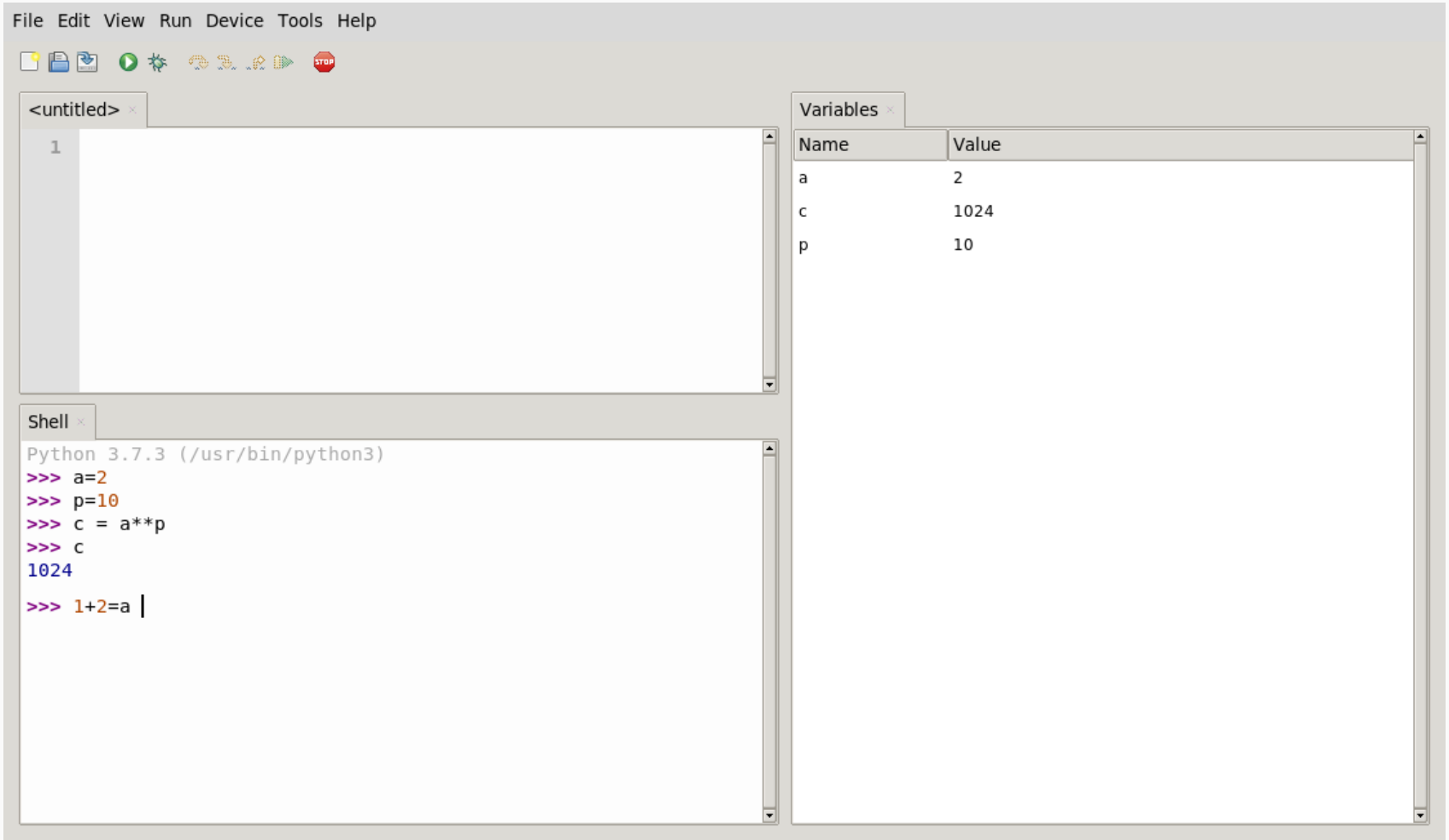
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



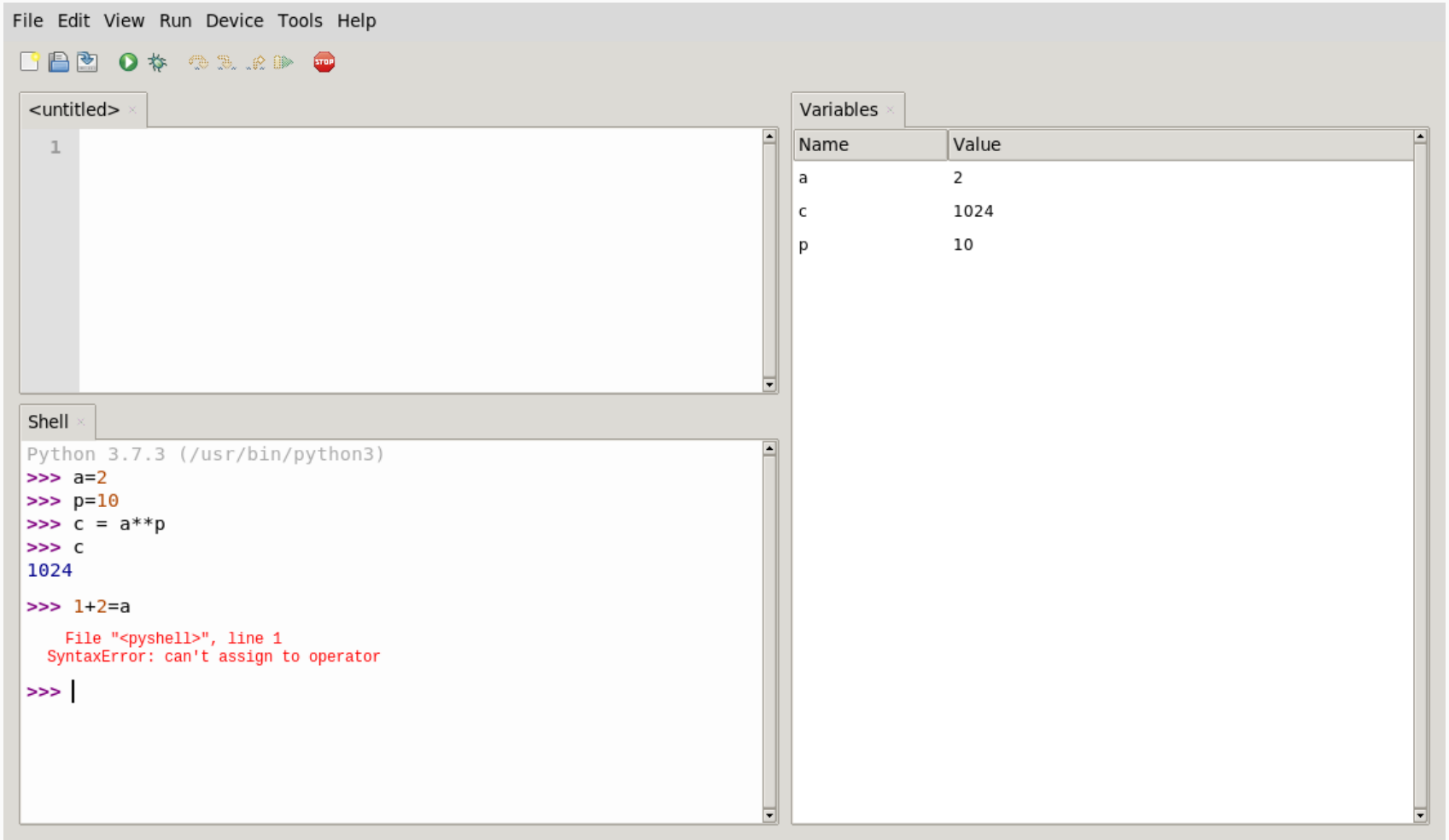
- Si nécessaire on ouvre l'onglet Variables (menu View)
- On continue à travailler dans le Shell
- mais on peut observer la mémoire (la valeur des variables) à droite



- 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



En Python chaque valeur a un type. Ce n'est pas le cas des variables.

- Python a un typage dynamique (sources de bug)
- Bonne pratique : conserver le type lors durant la vie d'une variable.

En Python chaque valeur a un type. Ce n'est pas le cas des variables.

- Python a un typage dynamique (sources de bug)
- Bonne pratique : conserver le type lors durant la vie d'une variable.

SCRIPT

```
a = "cou" # str
a = 2 * a       # str

# Attention, on va changer le type de a.
# Ne faites pas ça chez vous !
# Cette cascade est reservée aux professionels !

a = 3           # int
a = 2 * a       # int
```

En python une variable est créée lors de sa première affectation.

SCRIPT

```
#Ici aucune variable n'existe
a=3
# Ici a existe mais b n'existe pas
b = 2*a
# Ici a et b sont définies
```

En python une variable est créée lors de sa première affectation.

SCRIPT
```
#Ici aucune variable n'existe
a=3
# Ici a existe mais b n'existe pas
b = 2*a
# Ici a et b sont définies
```

Attention à toujours initialiser vos variables !

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

En python une variable est créée lors de sa première affectation.

SCRIPT
```
#Ici aucune variable n'existe
a=3
# Ici a existe mais b n'existe pas
b = 2*a
# Ici a et b sont définies
```

Attention à toujours initialiser vos variables !

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

En python une variable est créée lors de sa première affectation.

SCRIPT
```
#Ici aucune variable n'existe
a=3
# Ici a existe mais b n'existe pas
b = 2*a
# Ici a et b sont définies
```

Attention à toujours initialiser vos variables !

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

Lors d'un appel de fonction, les variables associées aux paramètres sont crées et initialisées.

SCRIPT
```
def suivant(a):
    return a+1

suivant(4)
```

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

```
>>>
```

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

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

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

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

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

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

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

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

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

```
>>> a=2
>>> a
2
>>> que_vaut_a()
```

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

```
>>> a=2
>>> a
2
>>> que_vaut_a()
3
>>>
```

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

```
>>> a=2
>>> a
2
>>> que_vaut_a()
3
>>> a
```

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT

```
def que_vaut_a():
    a = 3
    return a
```

SHELL

```
>>> a=2
>>> a
2
>>> que_vaut_a()
3
>>> a
2
```

- Les variables d'une fonction sont propres à la fonction
  - on dit que les variables sont locales.
  - Les variables définies ailleurs ne sont pas utilisées.
  - (on peut, mais c'est à éviter : variables globales)

SCRIPT
```
def que_vaut_a():
    a = 3
    return a
```

SHELL
```
>>> a=2
>>> a
2
>>> que_vaut_a()
3
>>> a
2
```

- Un programme est organisé en fonction
  - chaque fonction peut-être comprise indépendamment des autres.
  - une fonction prend des paramètres…
  - … et renvoie un résultat.

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
```
>>>
```

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
>>>
```

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
>>>
```

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 !
```

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
```

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
```

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
>>>
```

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
```

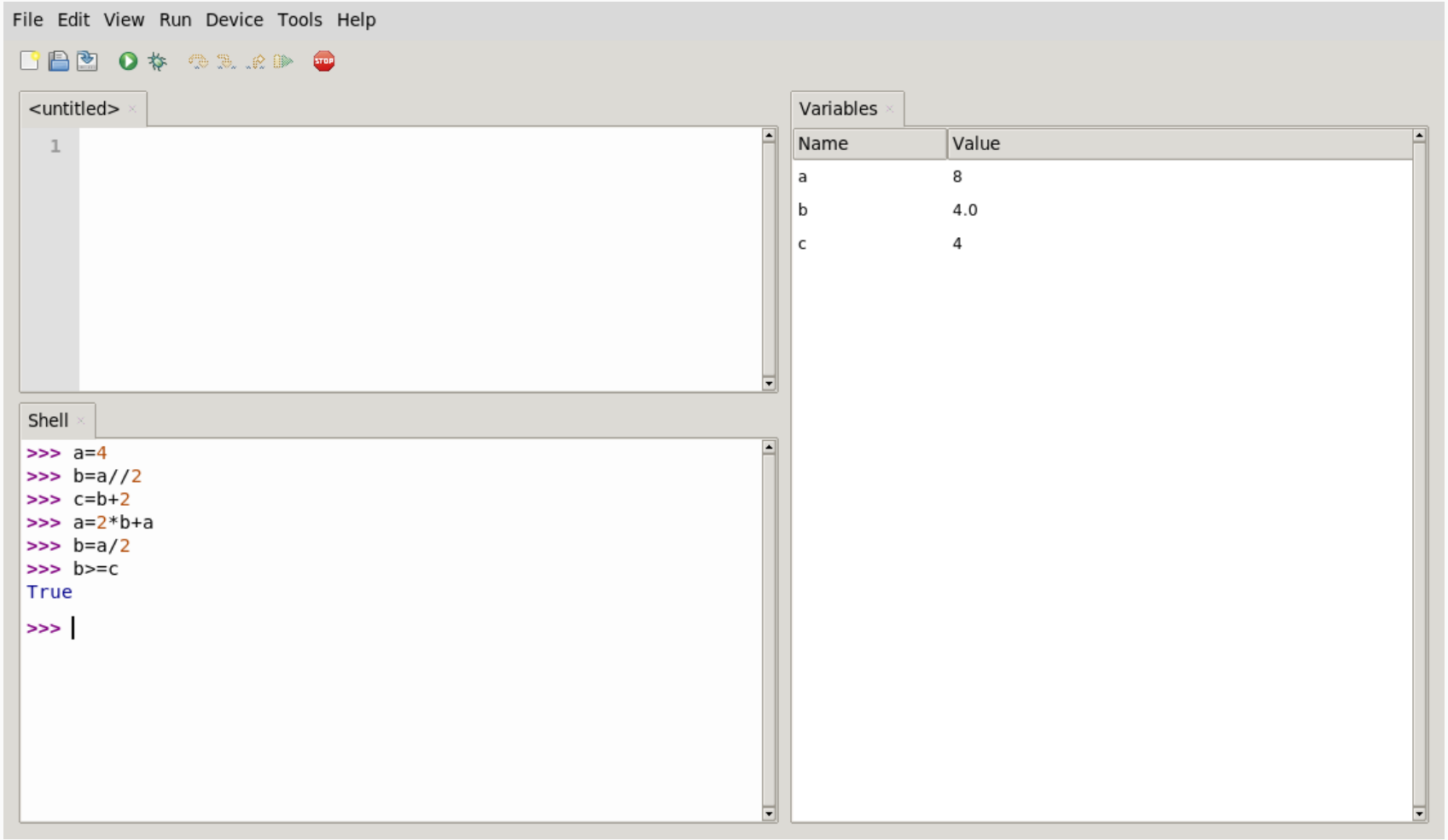
Que fait le code suivant ?

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

La commande d'affichage : `print`

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

```
>>>
```
SHELL

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

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

- 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
>>> 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 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.

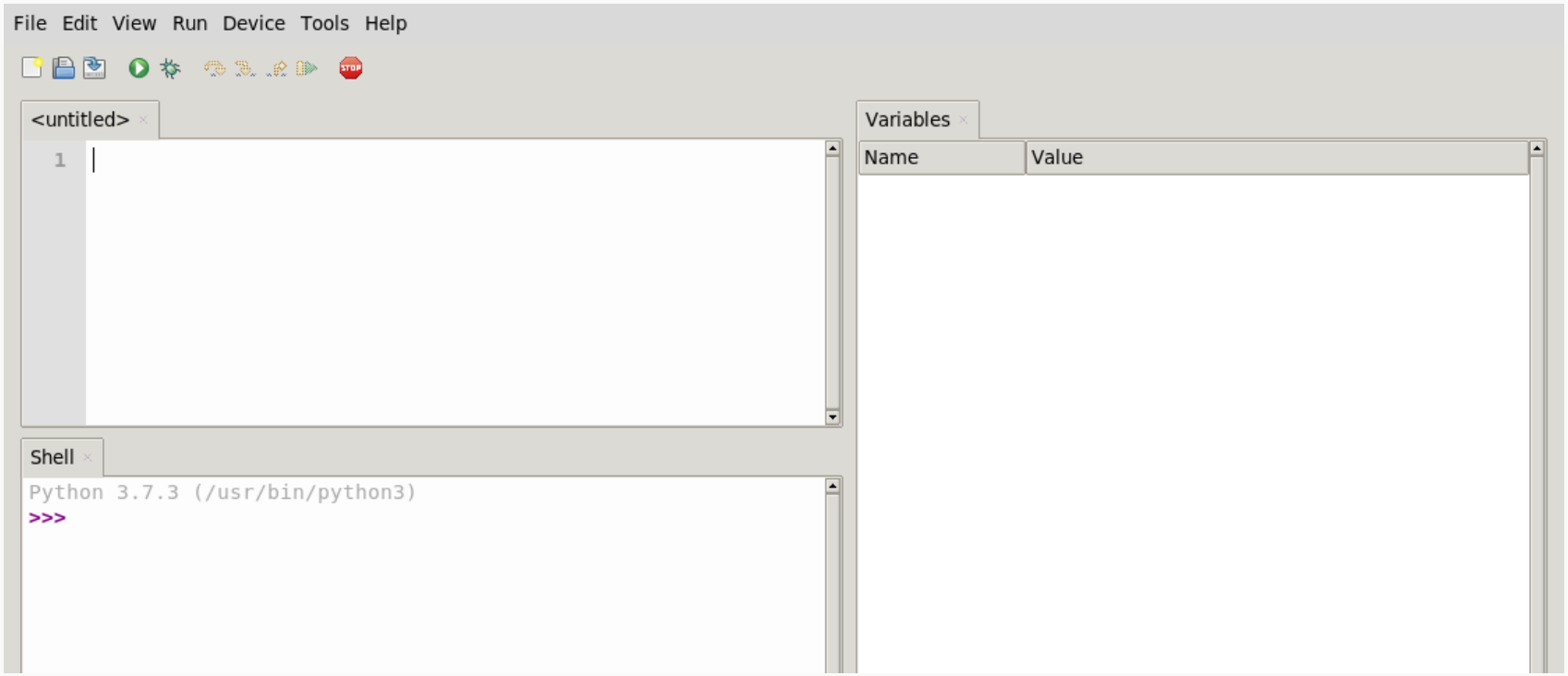
SCRIPT
```
prénom = input('Quel est votre prénom ? ')
print('J'ai rencontré',prénom,'et il est gentil')
```

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

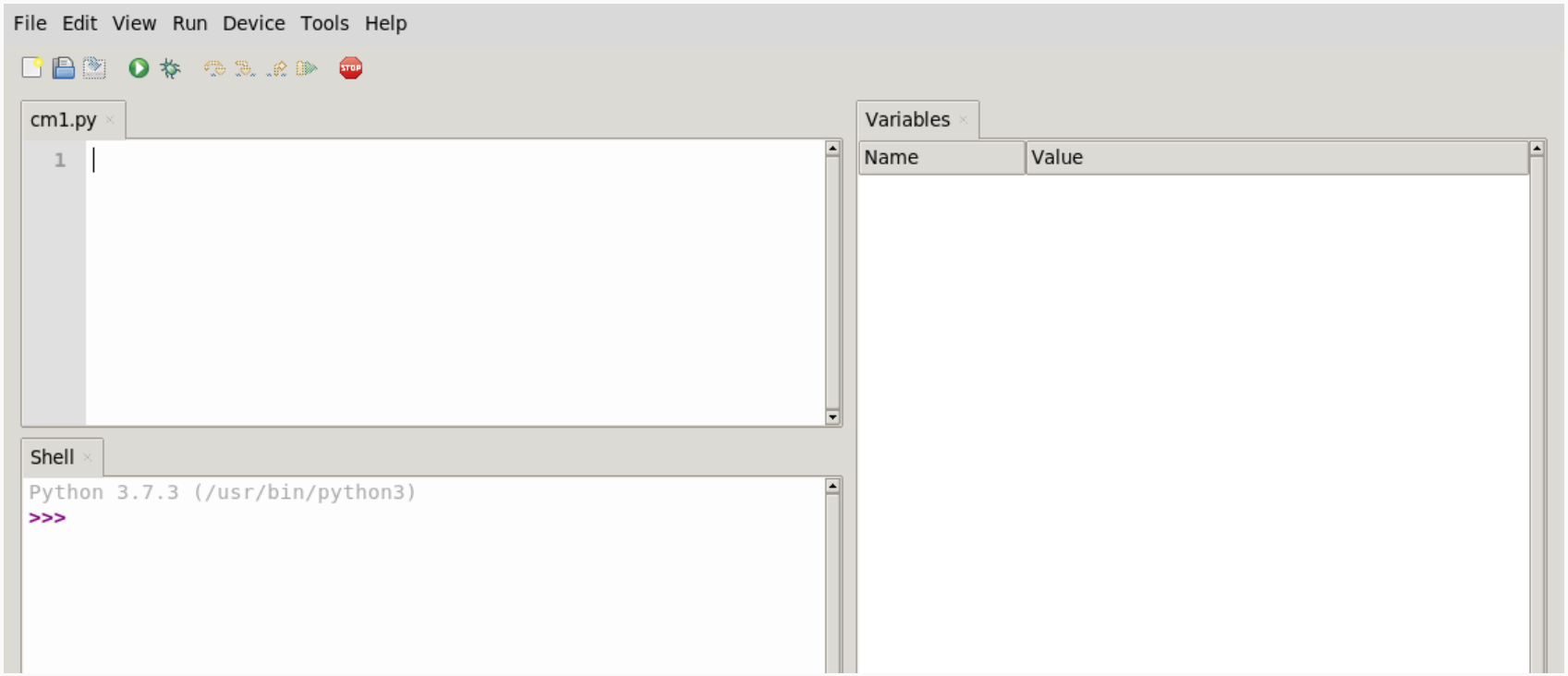
- Il est laborieux d'exécuter une à une les commandes dans le shell
  - et si l'on se trompe il faut tout recommencer !

- 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`

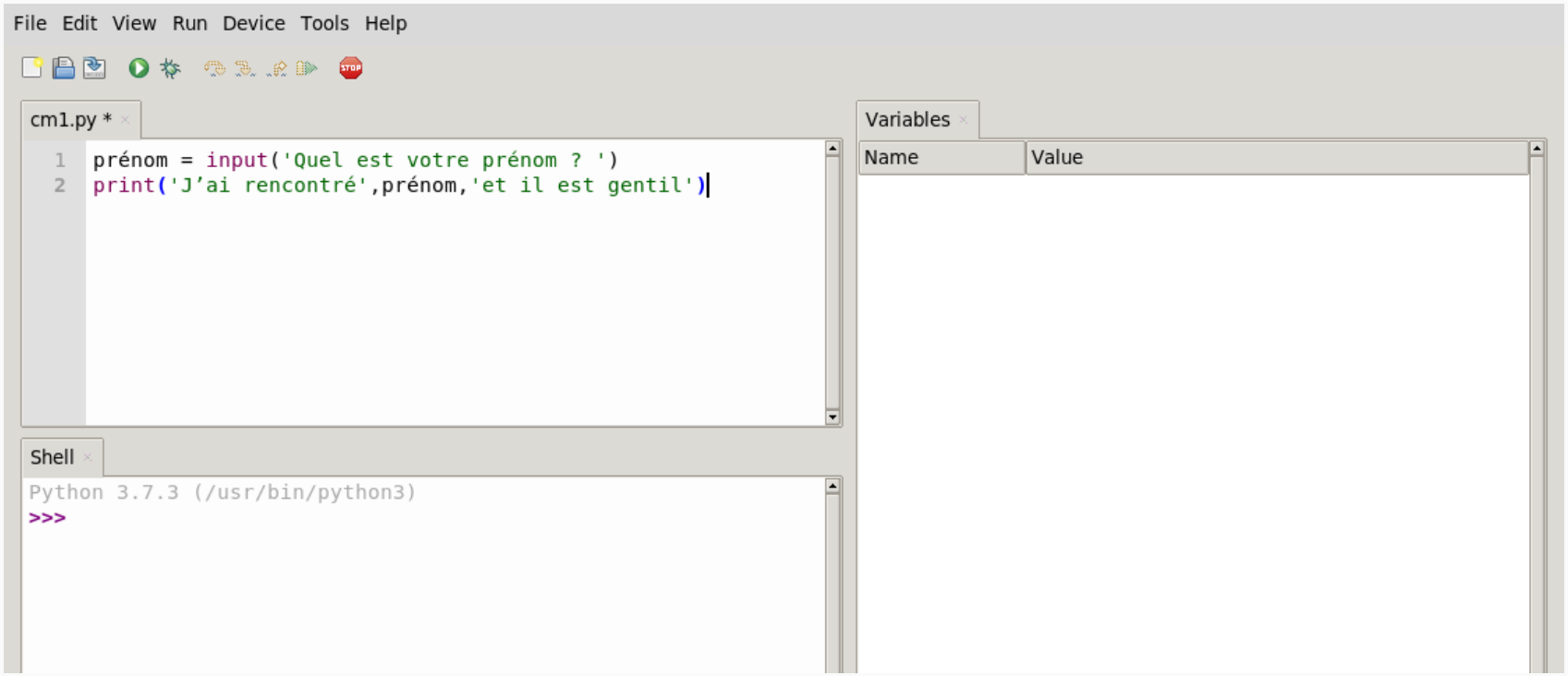
- On se place dans la fenêtre en haut à gauche.



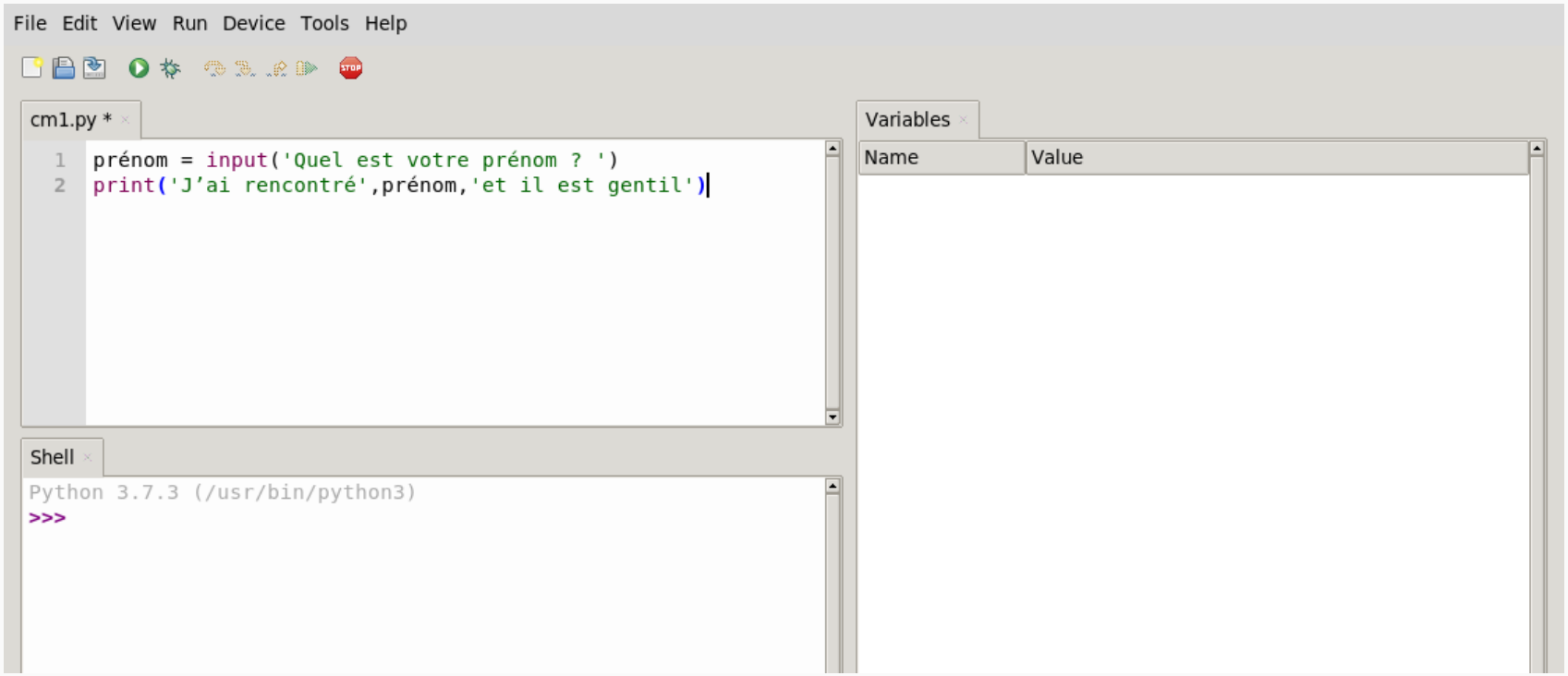
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S



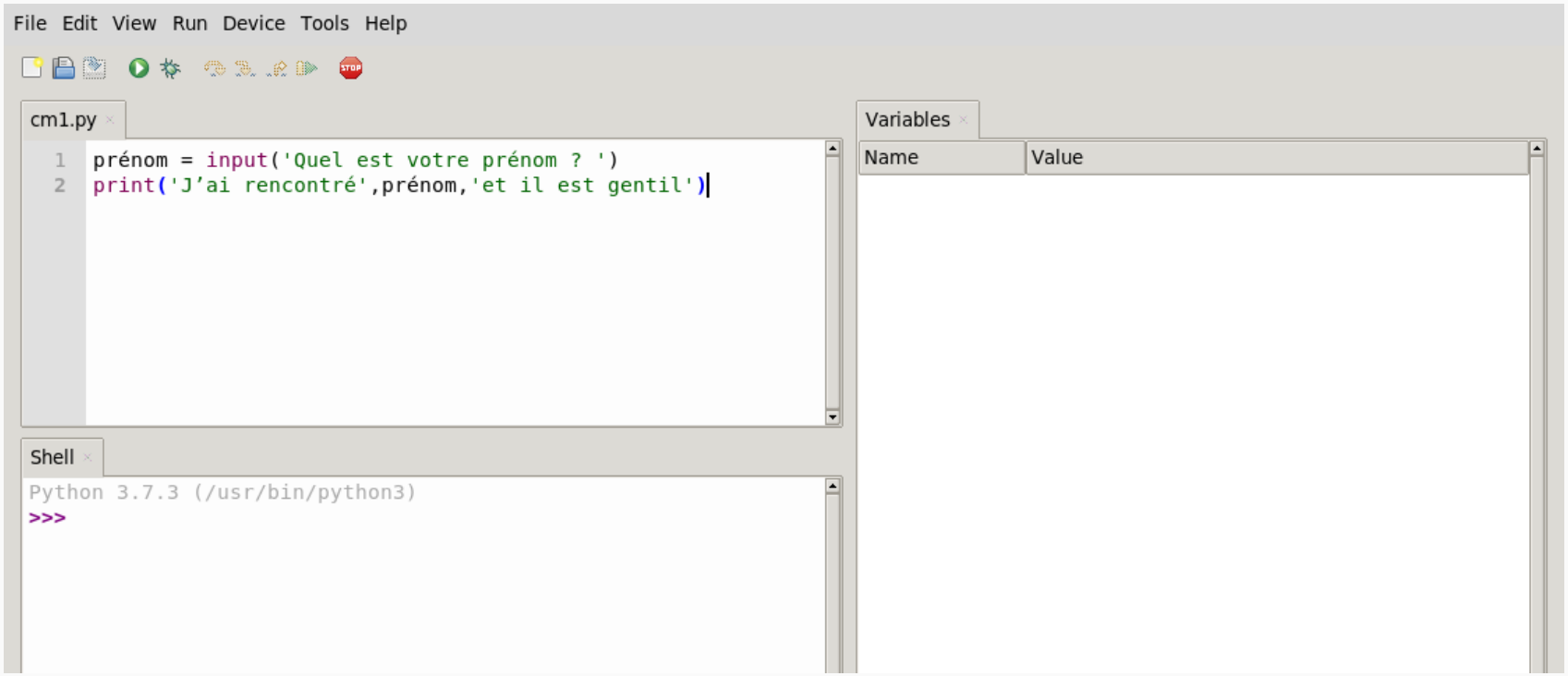
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script



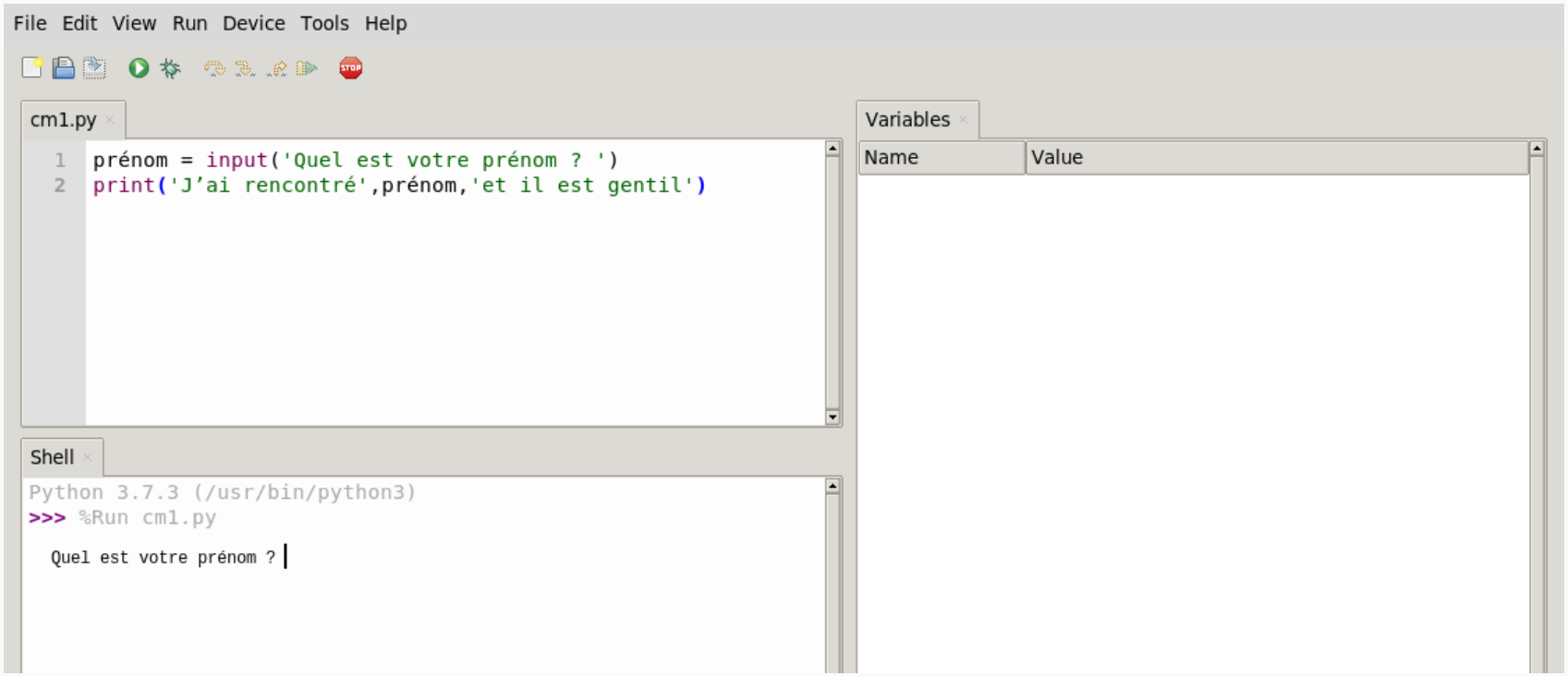
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder



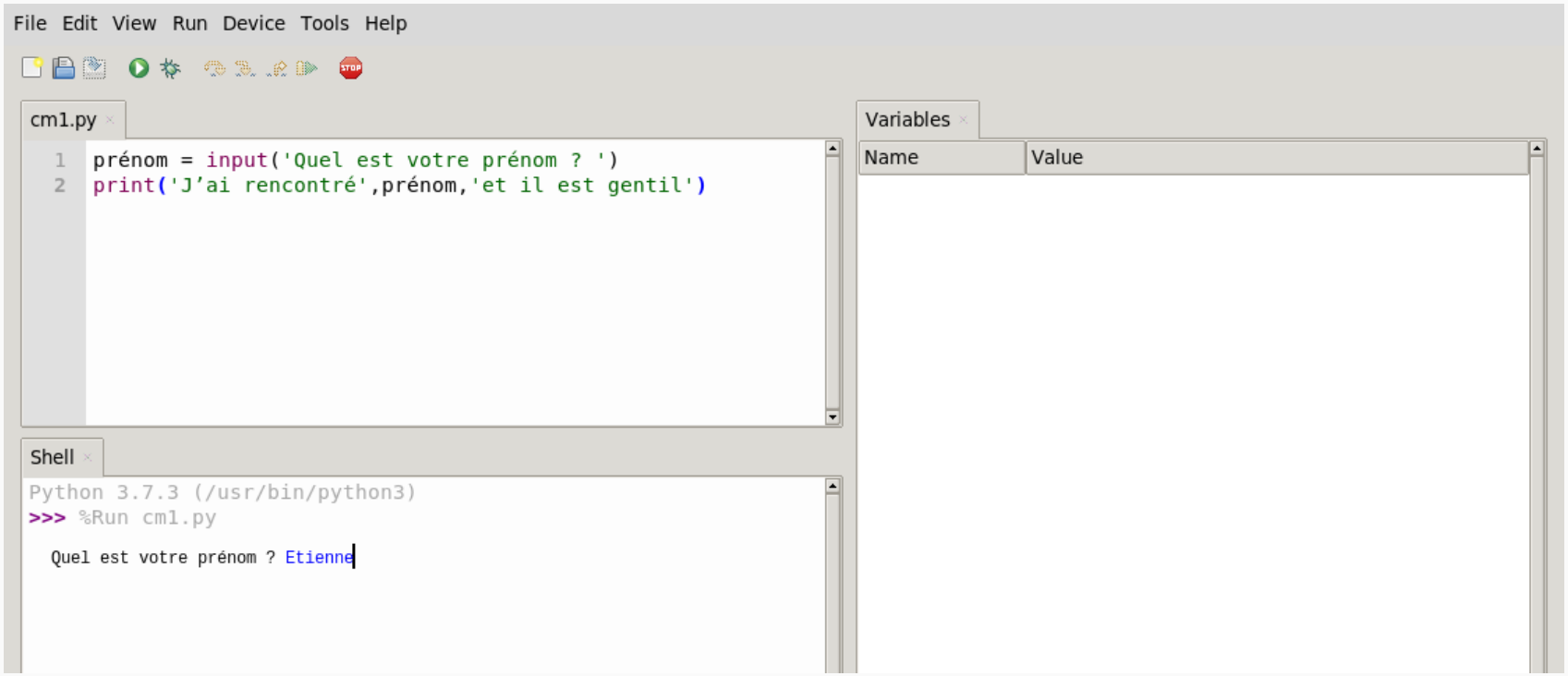
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît)



- 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

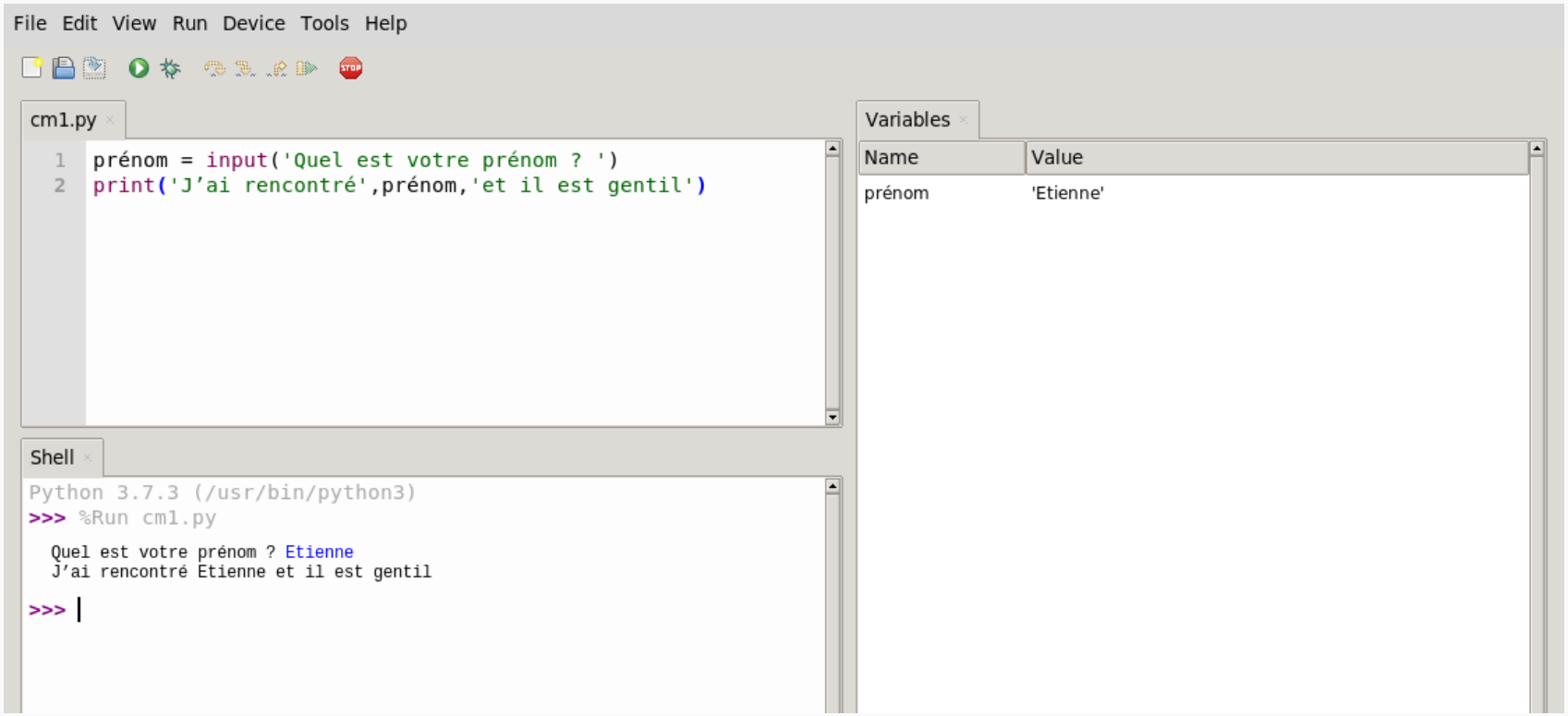


- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5

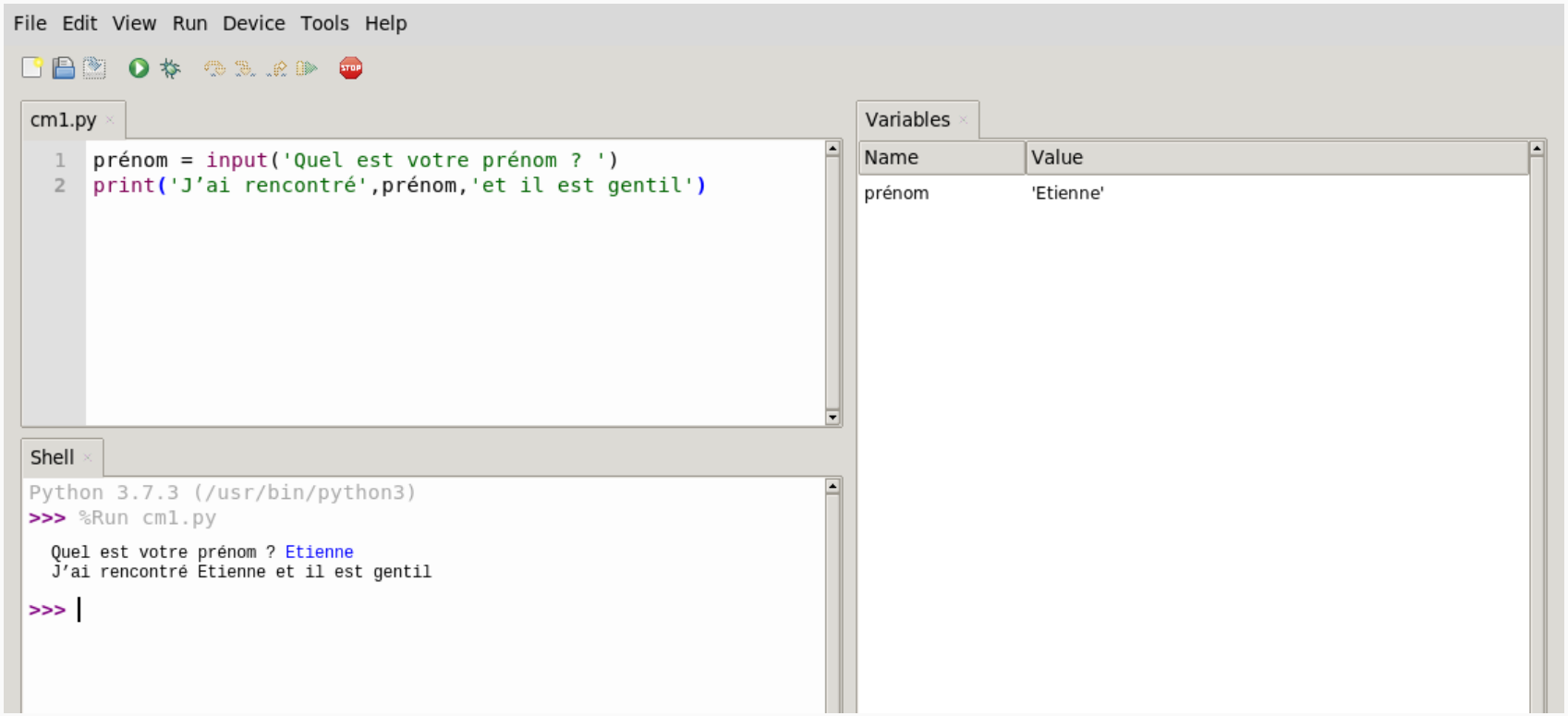
- On se place dans la fenêtre en haut à gauche.
  - Pour donner un nom au fichier script on appuie sur Ctrl + S
- On rédige notre script
  - L'étoile à côté du nom `cm1.py*` indique qu'il faut sauvegarder
- On sauvegarde Ctrl + S (l'* disparaît) puis on exécute F5
  - Puis on recommence :

- 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

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
```

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'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 "bug.py", line 3
    print(a,'est positif') # L'indentation est obligatoire !
    ^^^^^
IndentationError: expected an indented block after 'if'
statement on line 2
```

- 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` !

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

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

SHELL
```
>>>
```

- 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
```

- 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 ».

- 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

- 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
```
>>>
```

- 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
>>>
```

- 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)
```

- 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
```

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 caractères) : `Coucou` ; `'++12à23'`
- `bool` (booléen) : `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 caractères) : `Coucou` ; `'++12à23'`
- `bool` (booléen) : `1>3` ou `True`
- `Nonetype` (instructions) : `None` ; `print('x=',3)`

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

SHELL

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

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 caractères) : `Coucou` ; `'++12à23'`
- `bool` (booléen) : `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'>
>>>
```

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 caractères) : `Coucou` ; `'++12à23'`
- `bool` (booléen) : `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 caractères) : `Coucou` ; `'++12à23'`
- `bool` (booléen) : `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` .

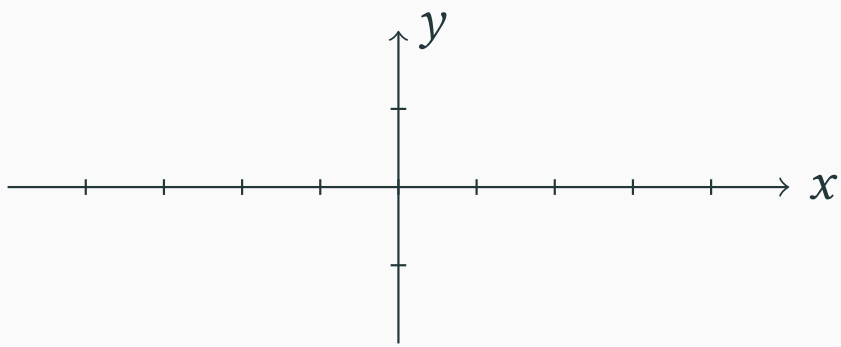
SCRIPT
```
def absolue(n):
⟶ if n>=0:
⟶ ⟶ return n # 2 indentations
⟶ else:
⟶ ⟶ return -n
```

- 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` .



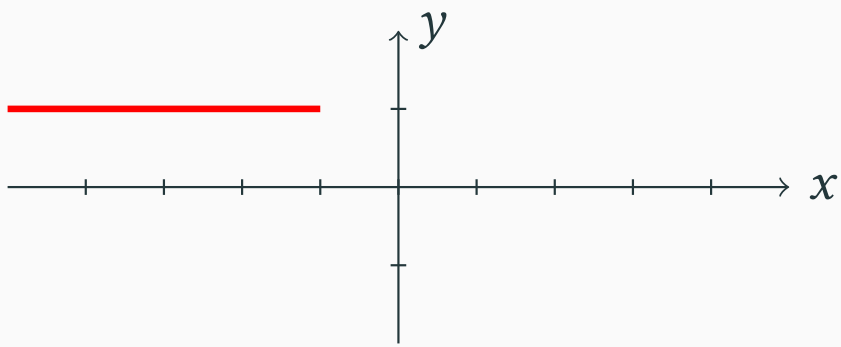
- Une fonction définie par morceaux.

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



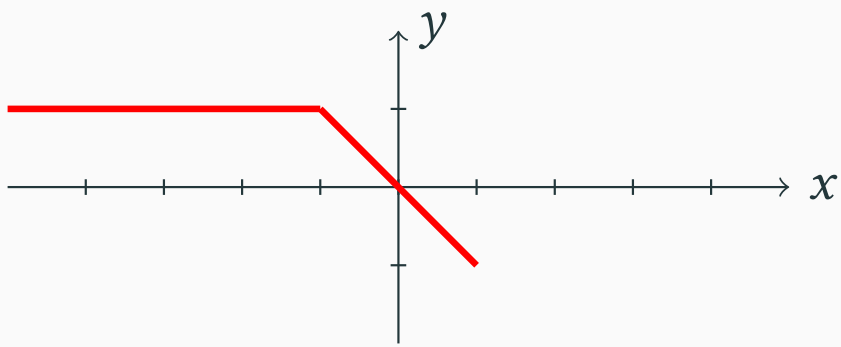
- Une fonction définie par morceaux.

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



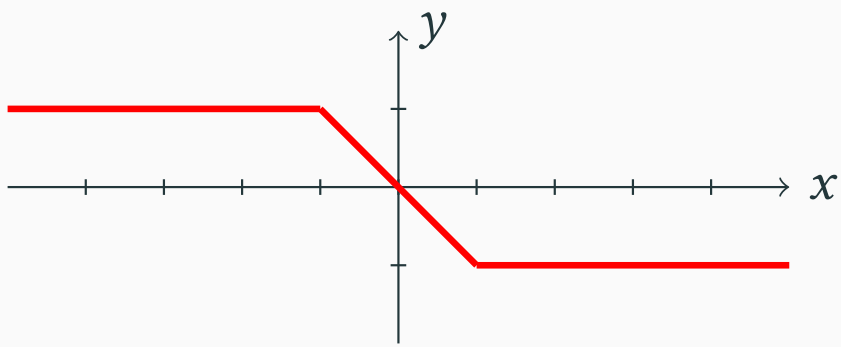
- Une fonction définie par morceaux.

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

Sommaire

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
```

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
```

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "cours1.py", line 8, in <module>
    assert plus_deux(18.5) == 20
           ^^^^^^^^^^^^^^^^^^^^^
AssertionError
```

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
```

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "cours1.py", line 8, in <module>
    assert plus_deux(18.5) == 20
           ^^^^^^^^^^^^^^^^^^^^^
AssertionError
```

- Comment corriger ?

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
```

- 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

```
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
```

- 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

```
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
```

- 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`.

Différence entre `print` et `return`

SCRIPT

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

SCRIPT

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

SHELL

```
>>>
```

Différence entre `print` et `return`

SCRIPT

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

SCRIPT

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

SHELL

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

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
>>>
```

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
```

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
>>>
```

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
```

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'
```

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`.

Honte aux fonctions impures !

- Attention : le code suivant est un exemple de **ce qu'il ne faut pas faire**

- Attention : le code suivant est un exemple de **ce qu'il ne faut pas faire**

SCRIPT

```
def fais_moi_un_gros_calcul_comme_un_cochon():
    chaîne = input("Donne moi un nombre : ")
    résultat = 2*int(chaîne) + 1
    print("Le résultat est", résultat)
```

- Attention : le code suivant est un exemple de **ce qu'il ne faut pas faire**

SCRIPT
```
def fais_moi_un_gros_calcul_comme_un_cochon():
    chaîne = input("Donne moi un nombre : ")
    résultat = 2*int(chaîne) + 1
    print("Le résultat est", résultat)
```

- Pourquoi ?

- Attention : le code suivant est un exemple de **ce qu'il ne faut pas faire**

SCRIPT
```
def fais_moi_un_gros_calcul_comme_un_cochon():
    chaîne = input("Donne moi un nombre : ")
    résultat = 2*int(chaîne) + 1
    print("Le résultat est", résultat)
```

- Pourquoi ?
  - Non réutilisable
  - Non testable
  - Mélange les calculs et les interactions
  - Nécessite de déranger l'utilisateur pour fonctionner

Comment organiser son code ?

- Un programme :
  - 99% de fonctions pures
  - 1% d'interactions avec l'utilisateurs

- Un programme :
  - 99% de fonctions pures
  - 1% d'interactions avec l'utilisateurs

SCRIPT

```
def lire_entier():
    chaîne = input("Donne moi un nombre : ")
    return int(chaîne)

def gros_calcul(n):
    return 2*n + 1

n = lire_entier() # entrée
résultat = gros_calcul(n) # cœur du programme
print("Le résultat est", résultat) # sortie
```

- Un programme :
  - 99% de fonctions pures
  - 1% d'interactions avec l'utilisateurs

SCRIPT

```
def lire_entier():
    chaîne = input("Donne moi un nombre : ")
    return int(chaîne)

def gros_calcul(n):
    return 2*n + 1

n = lire_entier() # entrée
résultat = gros_calcul(n) # cœur du programme
print("Le résultat est", résultat) # sortie
```

- La fonction `gros_calcul` :
  - peut être testée
  - peut-être réutilisée

- Un programme :
  - 99% de fonctions pures
  - 1% d'interactions avec l'utilisateurs

SCRIPT

```
def lire_entier():
    chaîne = input("Donne moi un nombre : ")
    return int(chaîne)

def gros_calcul(n):
    return 2*n + 1

n = lire_entier() # entrée
résultat = gros_calcul(n) # cœur du programme
print("Le résultat est", résultat) # sortie
```

- La fonction `gros_calcul` :
  - peut être testée
  - peut-être réutilisée

- Sauf demande explicite, interdiction d'utiliser `print` et `input`
  - `print` c'est le mal
  - `input` c'est le mal absolu

Sommaire

On cherche à savoir si deux entiers n'ont que 1 comme diviseur commun

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
```

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()
```

- 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
```

- 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 0x7f75c0dd2020>
```

- 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

Licence