



UNIVERSITÉ
CÔTE D'AZUR

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 INGÉNIEURIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- 🍃 Partie I. Variables
- 🍃 Partie II. Écrire des scripts
- 🍃 Partie III. Conditions
- 🍃 Partie IV. Fonctions
- 🍃 Partie V. Bonnes pratiques
- 🍃 Partie VI. Exemples
- 🍃 Partie VII. Table des matières

- ▶ Une **variable** est une boîte qui contient une valeur.

```
>>>
```

SHELL

- ▶ Une **variable** est une boîte qui contient une valeur.

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

```
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**.

- ▶ Une **variable** est une boîte qui contient une valeur.

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

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**.

- ▶ Une **variable** est une boîte qui contient une valeur.

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

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**.

- ▶ Une **variable** est une boîte qui contient une valeur.

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

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**.

- ▶ 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 ap
```

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**.

- ▶ 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 ap
>>>
```

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**.

- ▶ 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 ap
>>> c
```

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**.

- ▶ 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 ap
>>> 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**.

>>>

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 ap
>>> 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
```

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 ap
>>> 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 ap
>>> 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 ap
>>> 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 ap
>>> 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 ap
>>> 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
  ~~~
```

SHELL

```
SyntaxError: cannot assign to
expression here. Maybe you meant
'==' instead of '='?
```

```
>>> 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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

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

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

```
>>> 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
```

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

```
>>> 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
>>>
```

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.

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

SHELL

- ▶ Une variable peut **changer de valeur**.

```
>>> 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
```

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.

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

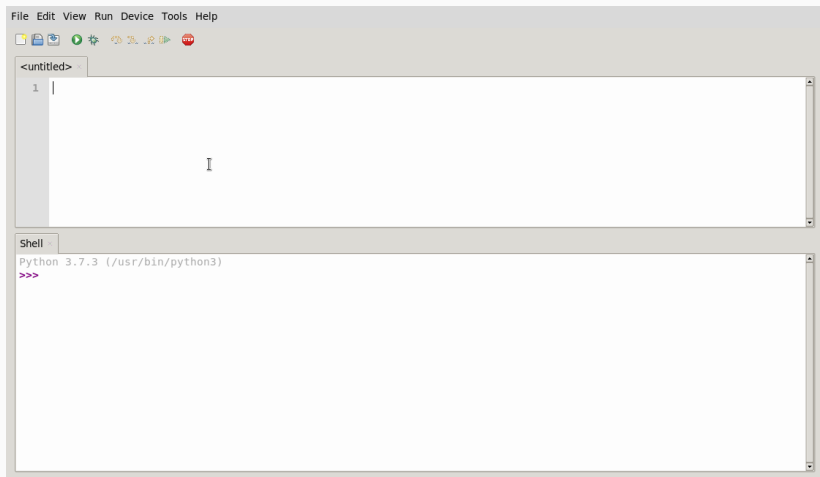
SHELL

- ▶ Une variable peut **changer de valeur**.

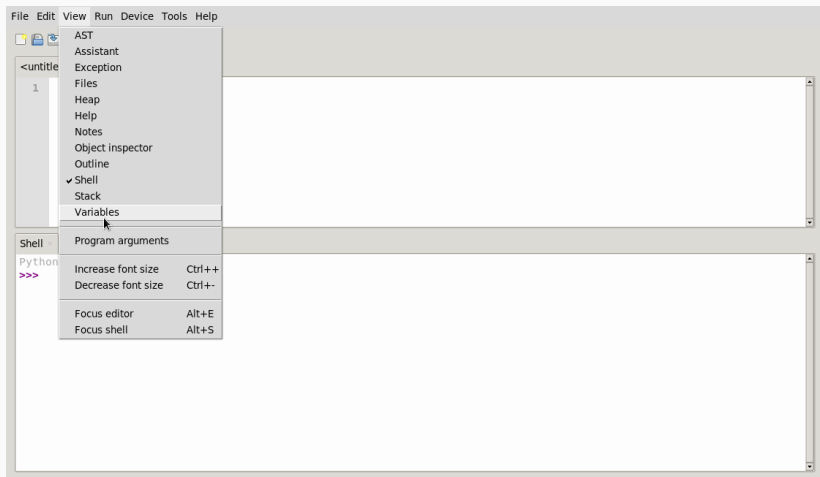
```
>>> a=2
>>> b=a    # Le résultat de a est donné à b
>>> a=a+1  # Le résultat de a+1 est donné à a
>>> b
2
>>> a
3
```

SHELL

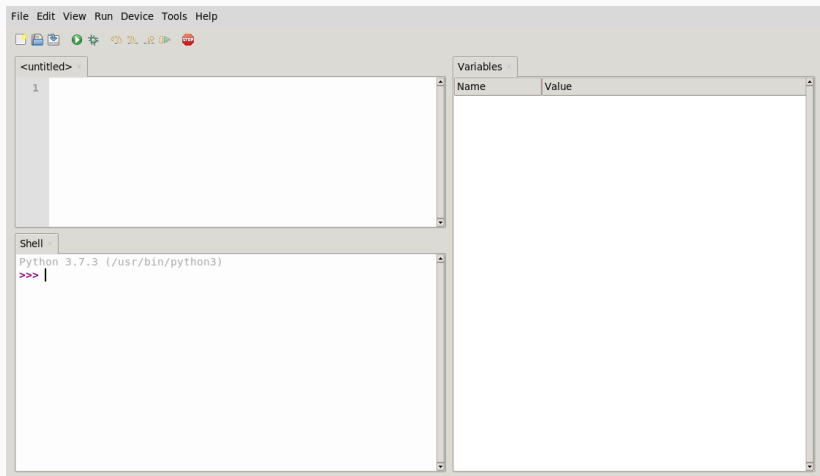
- ▶ Si nécessaire on ouvre l'onglet Variables (menu View)



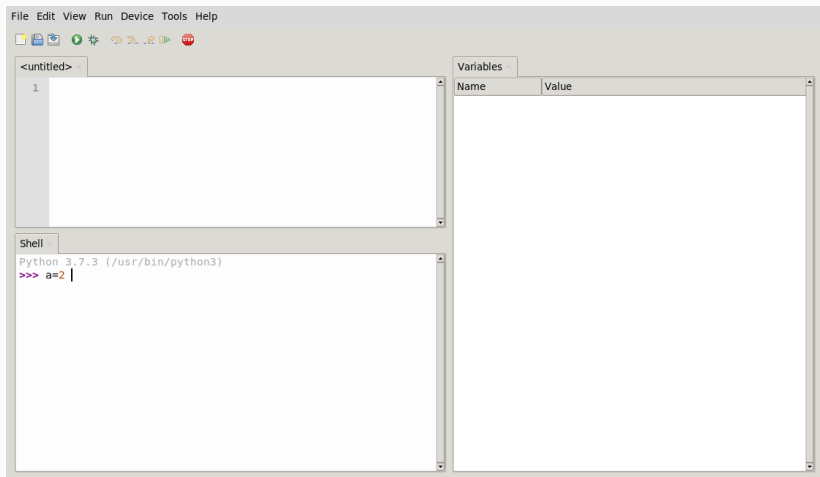
- ▶ Si nécessaire on ouvre l'onglet Variables (menu View)



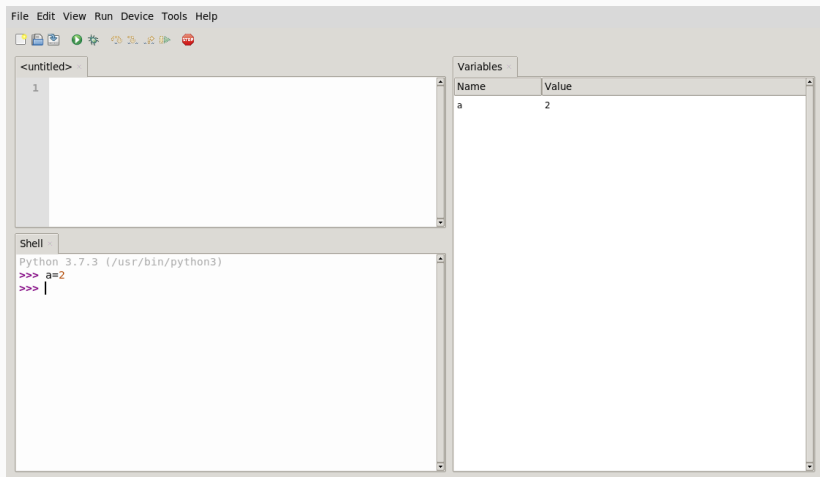
- ▶ Si nécessaire on ouvre l'onglet Variables (menu View)



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



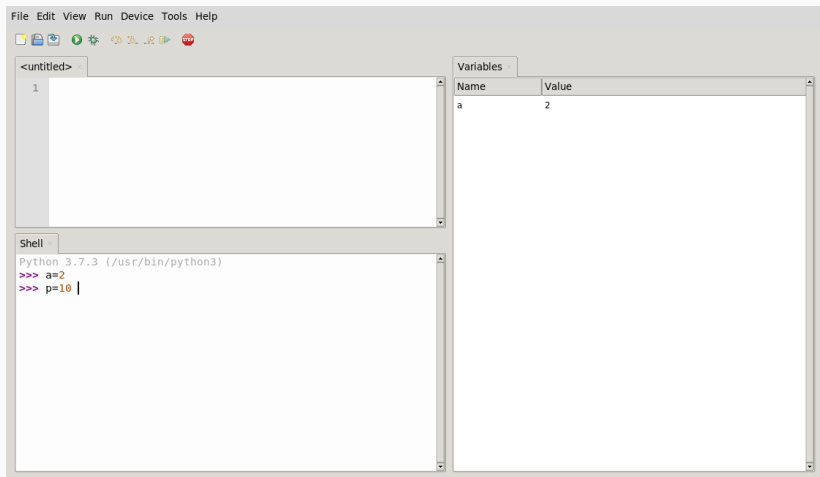
- ▶ Si nécessaire on ouvre l'onglet Variables (menu View)
- ▶ On continue à travailler dans le Shell



The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- <untitled>:** A code editor showing a single line of code: `1`.
- Shell:** A terminal window showing the Python 3.7.3 prompt and the command `a=2` being entered. The prompt is `Python 3.7.3 (/usr/bin/python3)`.
- Variables:** A panel displaying a table of current variables. The table has two columns: 'Name' and 'Value'. It shows one variable: `a` with a value of `2`.

- ▶ 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



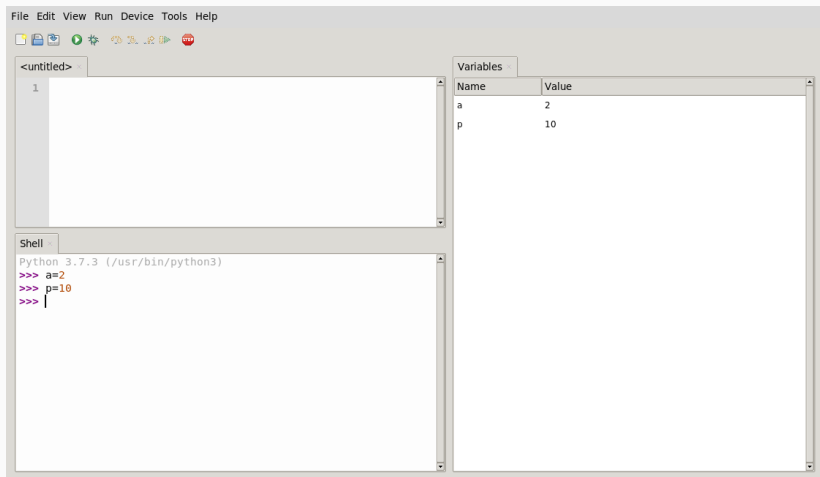
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- Code Editor:** A panel titled '<untitled>' containing a single line of code: `1`.
- Shell:** A panel titled 'Shell' showing the Python 3.7.3 shell prompt. The output is:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10 |
```
- Variables:** A panel titled 'Variables' displaying a table of current variables in memory:

Name	Value
a	2

- ▶ 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



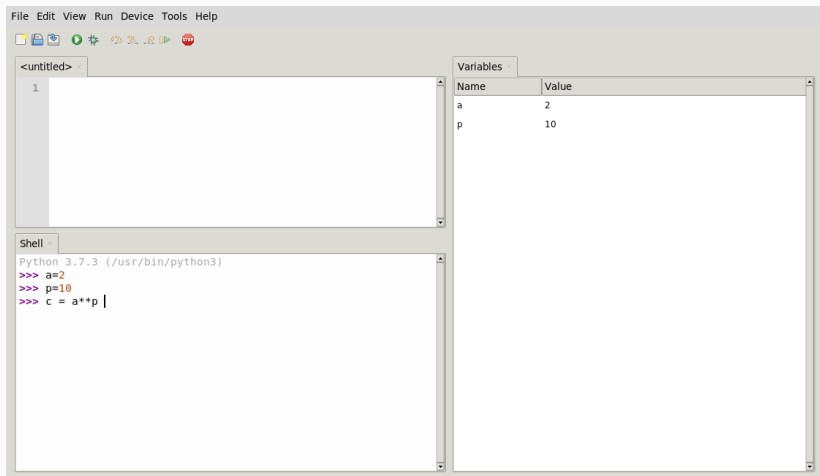
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main workspace is divided into three panels:

- Code Editor:** A window titled '<untitled>' containing a single line of code: `1`.
- Shell:** A window titled 'Shell' showing the Python 3.7.3 prompt. The input and output are:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> |
```
- Variables:** A window titled 'Variables' displaying a table of current variables in memory:

Name	Value
a	2
p	10

- ▶ 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



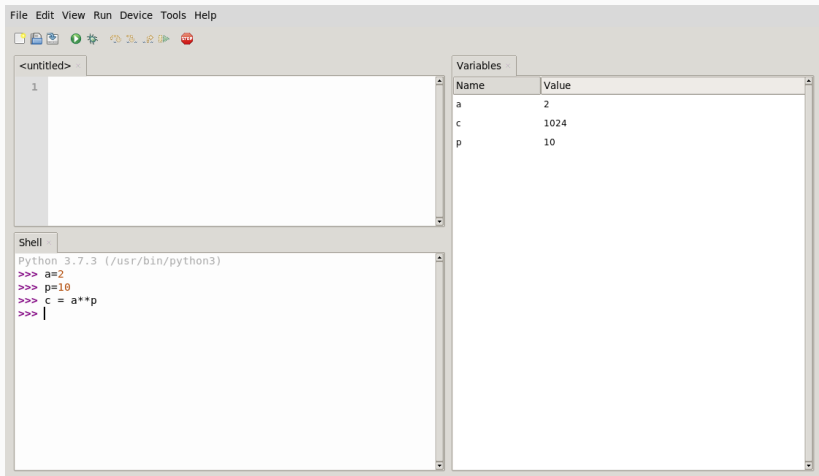
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- Code Editor:** A single line of code is visible: `1`.
- Shell:** A terminal window showing the execution of Python code:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> c = a**p |
```
- Variables:** A panel titled 'Variables' containing a table with two columns: 'Name' and 'Value'. The table lists the current state of variables:

Name	Value
a	2
p	10

- ▶ 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



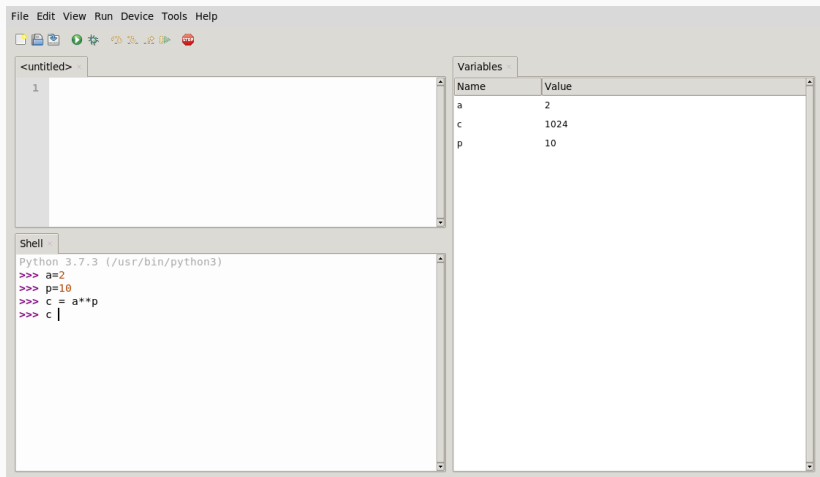
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- Code Editor:** A single line of code is visible: `1`.
- Shell:** A terminal window showing the execution of Python code:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> c = a**p
>>> |
```
- Variables:** A panel on the right side showing a table of current variables and their values:

Name	Value
a	2
c	1024
p	10

- ▶ 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



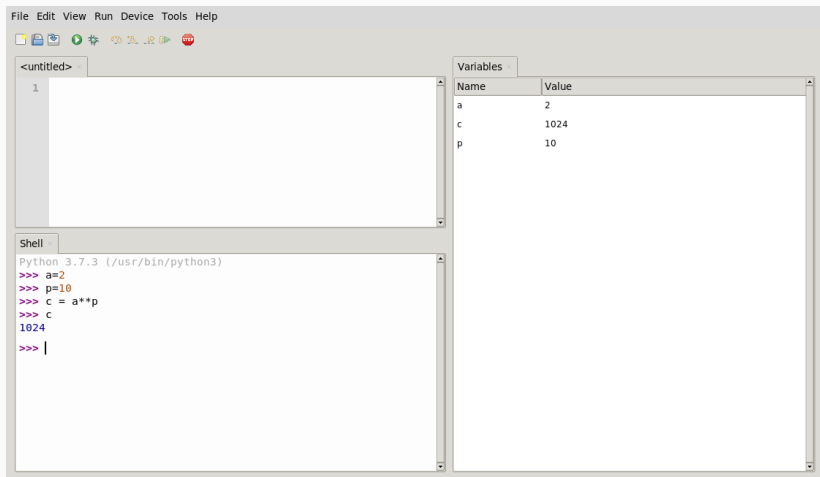
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main workspace is divided into three panels:

- Code Editor:** A single line of code is visible: `1`.
- Shell:** A terminal window showing the execution of Python code:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> c = a**p
>>> c |
```
- Variables:** A panel on the right side showing a table of current variables and their values:

Name	Value
a	2
c	1024
p	10

- ▶ 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



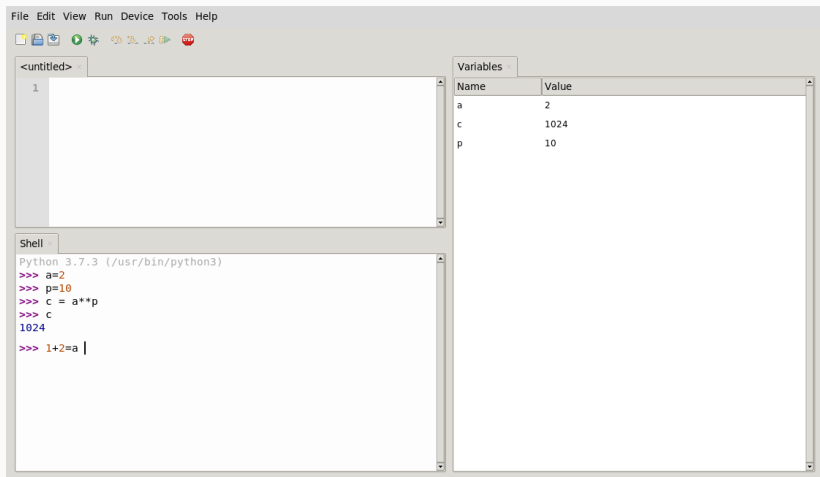
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- Code Editor:** Contains a single line of code: `1`.
- Shell:** Shows the execution of Python code:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> c = a**p
>>> c
1024
>>> |
```
- Variables:** A panel titled 'Variables' containing a table of current variables and their values:

Name	Value
a	2
c	1024
p	10

- ▶ 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



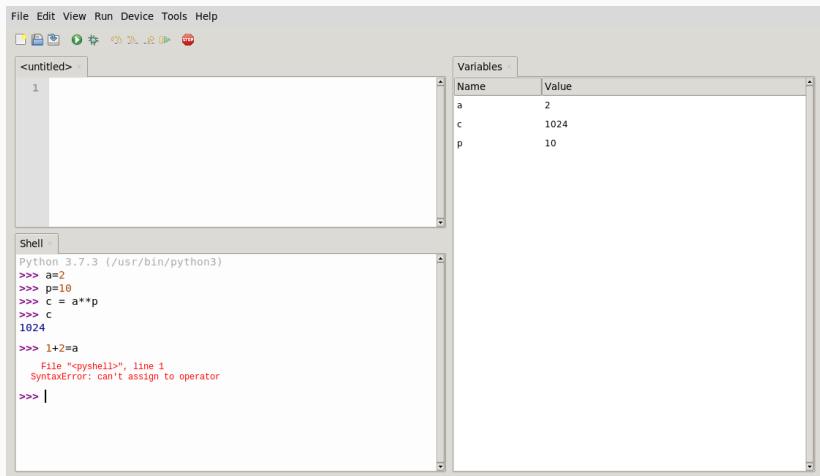
The screenshot shows the Thonny Python IDE interface. At the top is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- Code Editor:** A single line of code is visible: `1`.
- Shell:** A Python 3.7.3 shell window showing the following commands and output:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> c = a**p
>>> c
1024
>>> 1+2=a |
```
- Variables:** A panel titled 'Variables' displaying a table of current variables and their values:

Name	Value
a	2
c	1024
p	10

- ▶ 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



The screenshot shows the Thonny Python IDE interface. The top menu bar includes File, Edit, View, Run, Device, Tools, and Help. Below the menu bar is a toolbar with icons for file operations and execution. The main workspace is divided into three panels:

- <untitled>:** A code editor showing a single line of code: `1`.
- Shell:** A terminal window showing the execution of Python code:

```
Python 3.7.3 (/usr/bin/python3)
>>> a=2
>>> p=10
>>> c = a**p
>>> c
1024
>>> 1+2=a
File "c:\python37\python37\shell.py", line 1
SyntaxError: can't assign to operator
>>> |
```
- Variables:** A panel displaying a table of current variables and their values:

Name	Value
a	2
c	1024
p	10

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.

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

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

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

SCRIPT

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

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

SCRIPT

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

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

SCRIPT

Attention à toujours initialiser vos variables !

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

SHELL

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

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

SCRIPT

Attention à toujours initialiser vos variables !

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

SHELL

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

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

SCRIPT

Attention à toujours initialiser vos variables !

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

SHELL

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

```
def suivant(a):  
    return a+1  
  
suivant(4)
```

SCRIPT

- ▶ 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)

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

SCRIPT

```
>>>
```

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)

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

SCRIPT

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

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)

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

SCRIPT

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

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)

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

SCRIPT

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

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

SCRIPT

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

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)

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

SCRIPT

```
>>> a=2  
>>> a  
2  
>>> que_vaut_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)

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

SCRIPT

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

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)

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

SCRIPT

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

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

SCRIPT

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

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)

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

SCRIPT

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

SHELL

- ▶ 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 : $a \leftrightarrow b$. 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 : $a \leftrightarrow b$. Par exemple si « a vaut 5 » et « b vaut 4 », on souhaite avoir : « a vaut 4 » et « b vaut 5 ».

Méthode intuitive et **FAUSSE!**

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

SHELL

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

Méthode intuitive et **FAUSSE!**

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

SHELL

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

Méthode intuitive et **FAUSSE!**

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

SHELL

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

Méthode intuitive et **FAUSSE!**

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

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

```
>>>
```

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

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

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

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

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

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

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

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

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

```
>>> 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
```

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

```
>>> 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  
>>>
```

SHELL

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

Méthode intuitive et **FAUSSE!**

```
>>> 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 !
```

SHELL

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

```
>>> 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
```

SHELL

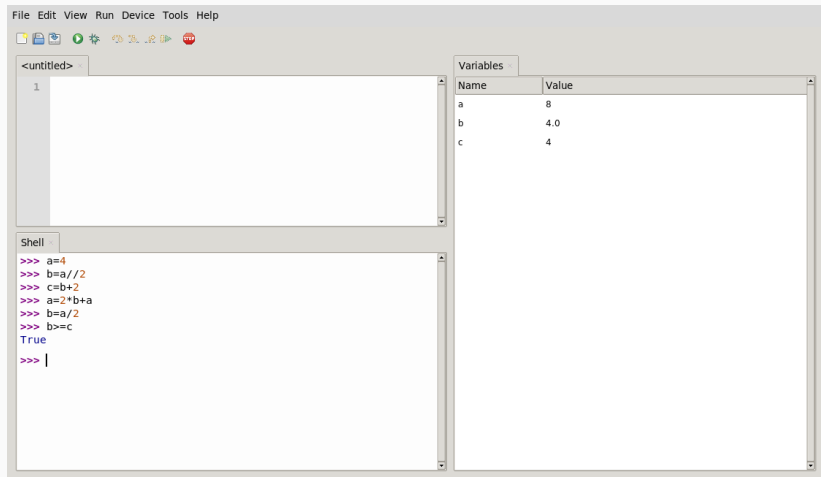
Que fait le code suivant ?

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

SCRIPT

- ▶ 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 :



The screenshot shows a Python IDE window with a menu bar (File, Edit, View, Run, Device, Tools, Help) and a toolbar. The main editor area contains a single line of code: `1`. Below the editor is a Shell window showing the execution of the code: `>>> a=4`, `>>> b=a//2`, `>>> c=b+2`, `>>> a=2*b+a`, `>>> b=a/2`, `>>> b>=c`, `True`, and `>>> |`. To the right of the Shell window is a Variables window with a table showing the current values of variables `a`, `b`, and `c`.

Name	Value
a	8
b	4.0
c	4

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

- 🍃 Partie I. Variables
- 🍃 Partie II. Écrire des scripts
- 🍃 Partie III. Conditions
- 🍃 Partie IV. Fonctions
- 🍃 Partie V. Bonnes pratiques
- 🍃 Partie VI. Exemples
- 🍃 Partie VII. Table des matières

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

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

SHELL

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

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

SHELL

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

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

SHELL

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

```
>>>
```

SHELL

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

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

SHELL

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

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

SHELL

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

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

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

SHELL

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

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

SHELL

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

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

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

SHELL

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

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

SHELL

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

SHELL

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

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

SCRIPT

```
Quel est votre prénom ?
```

SCRIPT

- ▶ '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.

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

SHELL

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

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

SCRIPT

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

SCRIPT

- ▶ '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.

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

SHELL

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

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

SCRIPT

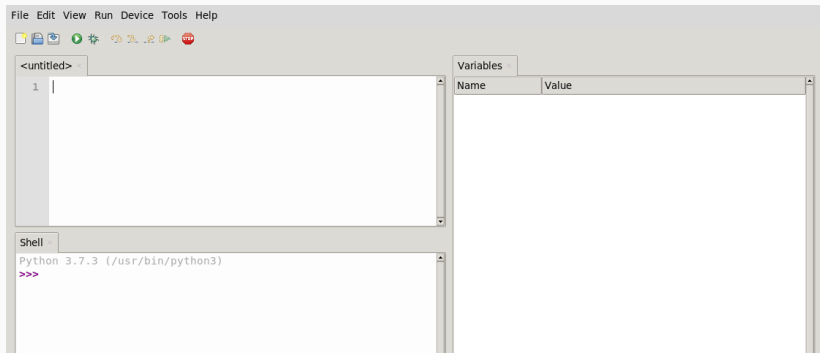
```
Quel est votre prénom ? Je m'appelle Olivier  
J'ai rencontré Je m'appelle Olivier et il est gentil
```

SCRIPT

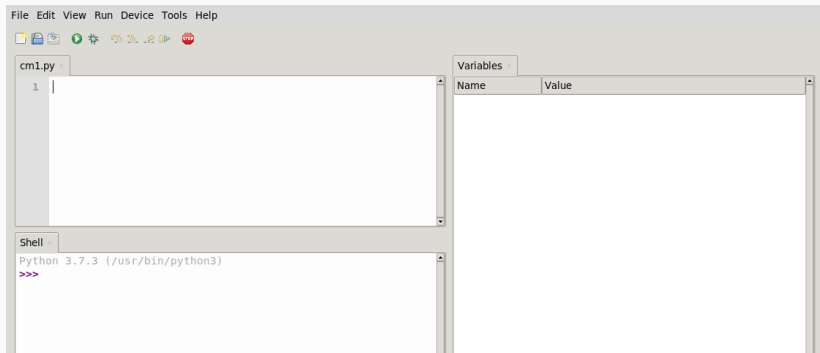
- ▶ Il est laborieux d'exécuter une à une les commandes dans le shell
 - et si l'on se trompe il faut tout recommencer !

- ▶ 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`

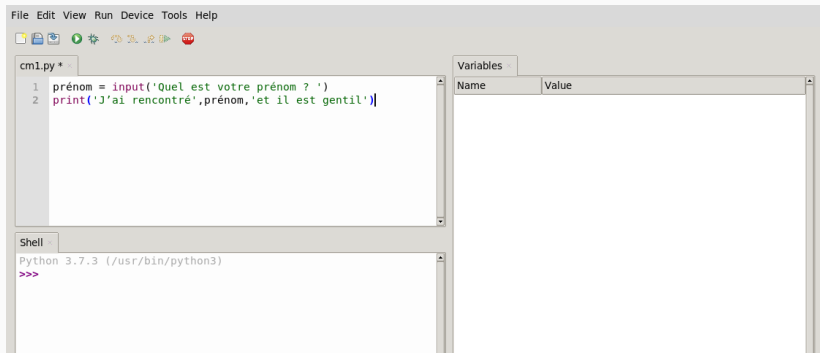
- ▶ On se place dans la fenêtre en haut à gauche.



- ▶ 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



The screenshot displays the Thonny IDE interface. The main editor window shows a Python script named 'cm1.py' with the following code:

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

Below the editor is a Shell window showing the Python 3.7.3 prompt:

```
Python 3.7.3 (/usr/bin/python3)
>>>
```

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

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

- ▶ 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

The screenshot displays the Thonny Python IDE interface. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main editor window shows a file named 'cm1.py' with the following code:

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

To the right of the editor is a 'Variables' panel with a table structure:

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

At the bottom of the IDE is a 'Shell' panel showing the Python 3.7.3 prompt:

```
Python 3.7.3 (/usr/bin/python3)
>>>
```

- ▶ 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)

The screenshot displays the Thonny Python IDE interface. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main editor window shows a file named 'cm1.py' with the following code:

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

Below the editor is a 'Shell' window showing the Python 3.7.3 prompt '>>>'. To the right of the editor is a 'Variables' window with a table structure:

Name	Value

- ▶ 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**

The screenshot shows the Thonny Python IDE interface. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main editor window displays a file named 'cm1.py' with the following code:

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

To the right of the editor is a 'Variables' panel with a table with two columns: 'Name' and 'Value'. The table is currently empty.

At the bottom of the IDE is a 'Shell' window. It shows the Python 3.7.3 prompt and the command to run the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
Quel est votre prénom ? |
```

- ▶ 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**

The screenshot shows the Thonny Python IDE interface. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main editor window displays a file named 'cm1.py' with the following code:

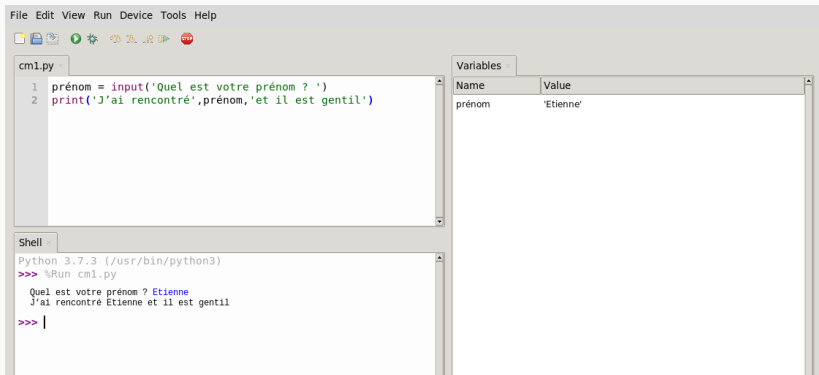
```
1 prénom = input('Quel est votre prénom ? ')
2 print('J'ai rencontré',prénom,'et il est gentil')
```

To the right of the editor is a 'Variables' panel with a table with two columns: 'Name' and 'Value'. The table is currently empty.

At the bottom of the IDE is a 'Shell' window. It shows the Python 3.7.3 prompt and the command to run the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
Quel est votre prénom ? Etienne
```

- ▶ 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**



The screenshot shows the Thonny Python IDE interface. The main editor window displays the following Python code in `cm1.py`:

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

Below the editor is the Shell window, which shows the execution of the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
    Quel est votre prénom ? Etienne
    J'ai rencontré Etienne et il est gentil
>>> |
```

On the right side of the IDE, the Variables window is open, showing a table with the following content:

Name	Value
prénom	'Etienne'

- ▶ 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 :

The screenshot shows the Thonny Python IDE interface. The main editor window displays the following Python code in `cm1.py`:

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

The Shell window shows the execution of the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
    Quel est votre prénom ? Etienne
    J'ai rencontré Etienne et il est gentil
>>> |
```

The Variables window on the right shows the current state of the program's memory:

Name	Value
prénom	'Etienne'

- ▶ 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 : **1** on modifie le script

The screenshot shows the Thonny Python IDE interface. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Run', 'Device', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main editor window displays a file named 'cm1.py' with the following Python code:

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

Below the editor is a 'Shell' window showing the execution of the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
    Quel est votre prénom ? Etienne
    J'ai rencontré Etienne et il est gentil
>>> |
```

On the right side of the IDE, there is a 'Variables' window showing the current state of variables:

Name	Value
prénom	'Etienne'

- ▶ 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**

The screenshot shows the Thonny Python IDE interface. At the top, there is a menu bar (File, Edit, View, Run, Device, Tools, Help) and a toolbar with icons for file operations and running code. The main editor window displays a Python script named `cm1.py` with the following code:

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

Below the editor is a Shell window showing the execution of the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
    Quel est votre prénom ? Etienne
    J'ai rencontré Etienne et il est gentil
>>> |
```

On the right side, the Variables window is open, showing a table of the current state of variables:

Name	Value
prénom	'Etienne'

- ▶ 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**

The screenshot shows the Thonny Python IDE interface. At the top, there is a menu bar (File, Edit, View, Run, Device, Tools, Help) and a toolbar with various icons. The main editor window displays a Python script named `cm1.py` with the following code:

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

Below the editor is the Shell window, which shows the execution of the script:

```
Python 3.7.3 (/usr/bin/python3)
>>> %Run cm1.py
    Quel est votre prénom ? Etienne
    J'ai rencontré Etienne et il est gentil
>>> |
```

On the right side of the IDE, there is a Variables window with a table showing the current state of variables:

Name	Value
prénom	'Etienne'

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 débogage détaillé (ou **⇧** + **F5** : debug. rapide)
- ▶ **F7** : exécution pas à pas (si débogage détaillé)
- ▶ **F7** : exécution ligne par ligne (si débogage 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

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

SCRIPT

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

- 🍃 Partie I. Variables
- 🍃 Partie II. Écrire des scripts
- 🍃 **Partie III. Conditions**
- 🍃 Partie IV. Fonctions
- 🍃 Partie V. Bonnes pratiques
- 🍃 Partie VI. Exemples
- 🍃 Partie VII. 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.

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

SCRIPT

```
-2 est négatif.
```

SHELL

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

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

SCRIPT

```
-2 est négatif.
```

SHELL

- ▶ 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.

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

SCRIPT

```
-2 est négatif.
```

SHELL

- ▶ 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 

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

SCRIPT


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

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

SCRIPT

```
-2 est négatif.
```

SHELL

- ▶ 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 

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

SCRIPT

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "bug.py", line 3
    print(a, 'est positif') # L'indentation est obligatoire !
    ~~~~~
```

SHELL

```
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`.

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

SCRIPT

zéro

SHELL

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

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

SCRIPT

zéro

SHELL

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

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

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

SCRIPT

zéro

SHELL

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

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

SCRIPT

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

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

SCRIPT

zéro

SHELL

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

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

SCRIPT

zéro

SHELL

- ▶ 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
             # et la condition 2 est vraie
elif condition 3:
    blublu   # exécuté si les conditions 1 et 2 sont fausses
             # 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

Donnez un nombre :

SCRIPT

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

Donnez un nombre : 27

SCRIPT

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

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

```
SCRIPT
```

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

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

SCRIPT

```
Donnez un nombre :
```

SCRIPT

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

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

`SCRIPT`

```
Donnez un nombre : 12
```

`SCRIPT`

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

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

`SCRIPT`

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

`SCRIPT`

- 🍃 Partie I. Variables
- 🍃 Partie II. Écrire des scripts
- 🍃 Partie III. Conditions
- 🍃 **Partie IV. Fonctions**
- 🍃 Partie V. Bonnes pratiques
- 🍃 Partie VI. Exemples
- 🍃 Partie VII. Table des matières

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

```
>>>
```

SHELL

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

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

SHELL

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

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

SHELL

- ▶ 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 ».

- ▶ 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

```
>>>
```

SHELL

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

- ▶ 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')
```

SHELL

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

- ▶ 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')
>>>
```

SHELL

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

- ▶ 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)
```

SHELL

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

- ▶ 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
>>>
```

SHELL

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

- ▶ 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)
```

SHELL

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

- ▶ 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.

```
>>>
```

SHELL

- ▶ 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

- ▶ 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

- ▶ 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
>>> sqrt(25)
```

SHELL

- ▶ 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
>>> sqrt(25)
5.0
```

SHELL

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

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`

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

SHELL

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`

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

SHELL

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`

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

SHELL

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`

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

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`.

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

SCRIPT

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

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

SCRIPT

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

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

SCRIPT

>>>

SHELL

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

SCRIPT

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

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

SCRIPT

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

SHELL

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

SCRIPT

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

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

SCRIPT

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

SHELL


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

SCRIPT

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

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

SCRIPT

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

SHELL

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

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

SCRIPT

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

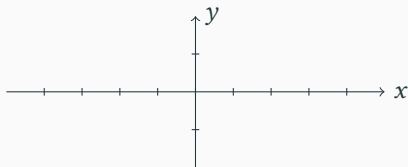
```
# Doublons les notes  
def dn(x):  
    if 2*x>20:  
        return 20  
    a=2*x  
    return a
```

SCRIPT

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

SHELL

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



- Une fonction définie par morceaux.

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

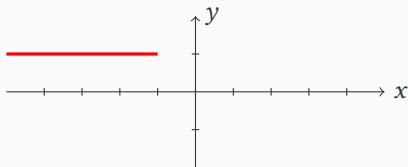
SCRIPT



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

SCRIPT

- Attention à bien indenter les blocs et sous-blocs



- Une fonction définie par morceaux.

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

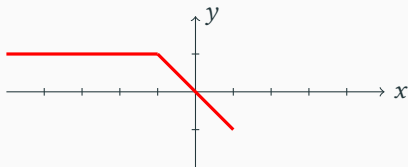
SCRIPT



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

SCRIPT

- Attention à bien indenter les blocs et sous-blocs



- Une fonction définie par morceaux.

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

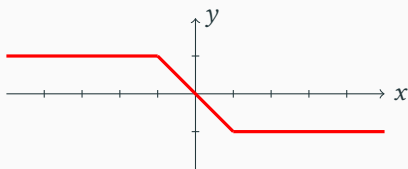
SCRIPT



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

SCRIPT

- Attention à bien indenter les blocs et sous-blocs



- Une fonction définie par morceaux.

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

SCRIPT



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

SCRIPT

- Attention à bien indenter les blocs et sous-blocs

- 🍃 Partie I. Variables
- 🍃 Partie II. Écrire des scripts
- 🍃 Partie III. Conditions
- 🍃 Partie IV. Fonctions
- 🍃 **Partie V. Bonnes pratiques**
- 🍃 Partie VI. Exemples
- 🍃 Partie VII. Table des matières

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 ?


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

SCRIPT

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

```
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
```

SCRIPT

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

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

SCRIPT

```
12  
20
```

SHELL

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

SCRIPT

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

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

SCRIPT

```
12  
20
```

SHELL

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

```
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
```

SCRIPT

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

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

SCRIPT

```
12  
20
```

SHELL

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

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

SCRIPT

```
12  
20.5  
20
```

SHELL

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

SCRIPT

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

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

SCRIPT

```
12  
20
```

SHELL

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

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

SCRIPT

```
12  
20.5  
20
```

SHELL

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

```
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
```

SCRIPT

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

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

SCRIPT

```
12  
20
```

SHELL

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

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

SCRIPT

```
12  
20.5  
20
```

SHELL

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

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

SCRIPT

```
True  
False  
True
```

SHELL

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.

```
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
```

SCRIPT

- ▶ 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.

```
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
```

SCRIPT

- ▶ 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.

```
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
```

SCRIPT

- ▶ 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
```

- ▶ 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.

```
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
```

SCRIPT

- ▶ 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.

```
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
```

SCRIPT

- ▶ 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`.

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

SCRIPT

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

SCRIPT

```
>>>
```

SHELL


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

SCRIPT

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

SCRIPT

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

SHELL

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

SCRIPT

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

SCRIPT

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

SHELL

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

SCRIPT

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

SCRIPT

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

SHELL

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

SCRIPT

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

SCRIPT

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

SHELL

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

SCRIPT

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

SCRIPT

```
>>> 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
```

SHELL

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

SCRIPT

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

SCRIPT

```
>>> 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'
```

SHELL

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

SCRIPT

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

SCRIPT

```
>>> 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'
```

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`.

- ▶ **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**

```
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)
```

SCRIPT

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

```
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)
```

SCRIPT

- ▶ Pourquoi?

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

```
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)
```

SCRIPT

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

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

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

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'utilisateur

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'utilisateur

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

- 🍃 Partie I. Variables
- 🍃 Partie II. Écrire des scripts
- 🍃 Partie III. Conditions
- 🍃 Partie IV. Fonctions
- 🍃 Partie V. Bonnes pratiques
- 🍃 **Partie VI. Exemples**
- 🍃 Partie VII. Table des matières

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

```
def premiers_entre_eux(p,q):  
    if gcd(p,q)==1:  
        return True  
    else:  
        return False
```

SCRIPT

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

```
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

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

```
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
```

SCRIPT

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

```
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
```

SCRIPT

Mêmes résultats mais codés de façon plus ou moins élégante

- élégance ; c'est-à-dire **efficacité** et **lisibilité**

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ pour $a \leq b$, `randint(a,b)` renvoie un entier aléatoire (pseudo-aléatoire) de l'intervalle $[a,b]$

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

>>>

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>>
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>>
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>> jet35()
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>> jet35()  
5  
>>>
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>> jet35()  
5  
>>> # Parenthèses obligatoires !
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>> jet35()  
5  
>>> # Parenthèses obligatoires !  
>>>
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>> jet35()  
5  
>>> # Parenthèses obligatoires !  
>>> jet35()
```

SHELL

- ▶ On peut importer la fonction `randint` du module `random` :

```
from random import randint
```

SCRIPT

- ▶ 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 $[0,20]$
- ▶ Une fonction `jet35()` qui tire 3 ou 5 au hasard :

```
def jet35():  
    if randint(0,1)==0:  
        return 3  
    else:  
        return 5
```

SCRIPT

```
>>> jet35()  
5  
>>> jet35()  
3  
>>> jet35()  
5  
>>> # Parenthèses obligatoires !  
>>> jet35  
<function jet35 at 0x7f75c0dd2020>
```

SHELL

- ▶ On prend comme convention pile↔0 et face↔1

```
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...')
```

SCRIPT

- ▶ On prend comme convention pile \leftrightarrow 0 et face \leftrightarrow 1

```
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...')
```

SCRIPT

- ▶ 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 I — Variables, fonctions et conditions

🍃 Partie I. Variables

Variables et affectations

Instructions et expressions

🐘 Espionner les variables avec Thonny

Variable et typage

Variables et initialisations

Variables et fonctions

Échange de deux variables

🐘 Exercices

🐘 Correction

🍃 Partie II. Écrire des scripts

La commande d'affichage : `print`

Les chaînes de caractères

Les scripts python

🐘 Thonny : écrire des scripts

🐘 Thonny : raccourcis clavier

🐘 Exercices

🍃 Partie III. Conditions

Conditions : commande `if`

Conditions : Mot-clef `elif`

Conditions : synthèse

🐘 Exercices

🍃 Partie IV. Fonctions

Les fonctions prédéfinies en Python

Type des variables

Un exemple : la fonction « valeur absolue »

Autre exemple de fonctions

🍃 Partie V. Bonnes pratiques

Tester ses fonctions

Tester ses fonctions : `assert`

Tester ses fonctions : bilan

Différence entre `print` et `return`

Honte aux fonctions impures!

Comment organiser son code?

🍃 Partie VI. 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 VII. Table des matières

- ▶ © 2024 — Olivier Baldellon
- ▶ Ce document est publié sous licence **CC-BY Attribution 4.0** 
- Vous êtes autorisé à :
 - ▶ **Partager** — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats pour toute utilisation, y compris commerciale.
 - ▶ **Adapter** — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.
- Selon les conditions suivantes :
 - ▶ **Attribution** — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.
- ▶ <https://creativecommons.org/licenses/by/4.0/deed.fr>