



---

# Python

## Programmation objet



Macademia

Animé par Abdellatif Bedjbedj



# Le langage Python

Pour commencer ...

# Que savez-vous de Python ?

macademia

# Python ... en quelques mots

- ▷ Simple.
- ▷ Gratuit.
- ▷ Portable.
- ▷ Multithread.
- ▷ Dynamique.
- ▷ Extensible.
- ▷ Orienté objet.



# Ils utilisent Python



Instagram



NETFLIX



# Python face aux autres langages

↔	#	Logo	Nom
=	1		JavaScript
=	2		Python
=	3		Java
=	4		PHP
▲	5		C++
▼	6		C#
=	7		Ruby
=	8		TypeScript
=	9		C
=	10		Swift

RedMonk Index juin 2020

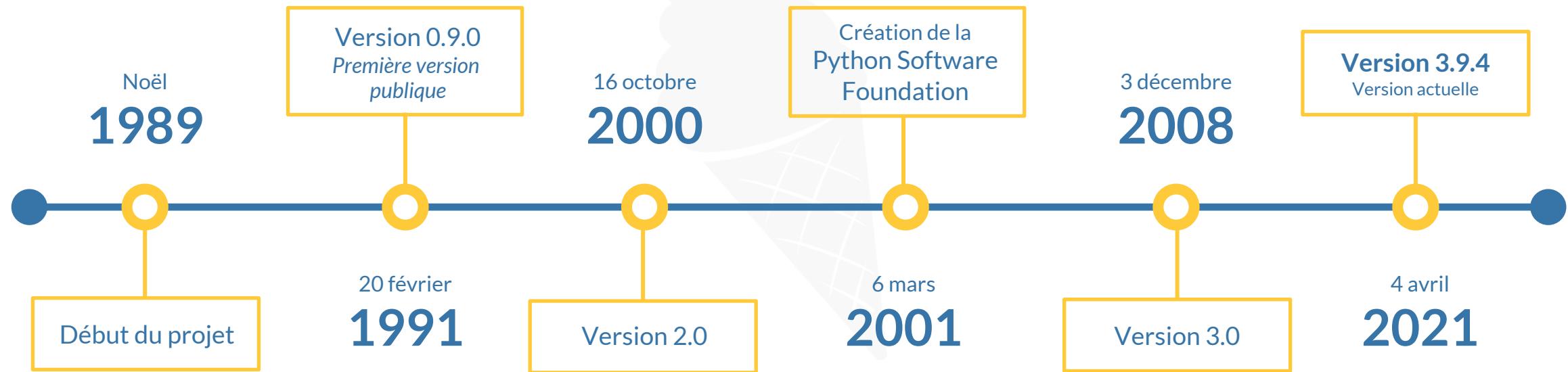
# Un peu d'histoire ...

- ▷ Crée en 1989 par **Guido van Rossum** (Pays-Bas, photo) durant les vacances de Noël.
  - › *Dictateur bienveillant à vie par intérim* de Python ... jusqu'en 2018.
- ▷ Nommé en hommage aux *Monty Python*.
- ▷ Inspiré des langages ABC, Modula-3 et C.



Interview de Guido van Rossum pour *Le Monde* (25/07/2018) : <https://lemonde.fr/2JWI8SJ>

# Historique des versions et dates importantes



# Fin du support Python 2

Depuis le 1<sup>er</sup> janvier 2020 ...

**Fin du support officiel**

**Python 2**

**La migration est très fortement conseillée !**

Rappel : pas de rétrocompatibilité entre Python 2 et Python 3

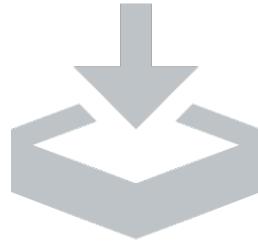
# Documentation

- ▷ Site officiel
  - › <https://www.python.org/>
- ▷ Documentation officielle
  - › <https://docs.python.org/fr/3>
- ▷ Téléchargement
  - › <https://www.python.org/downloads>
- ▷ Tutoriel
  - › <https://docs.python.org/fr/3/tutorial>

# Un langage interprété

- ▷ Python est un langage **interprété** ( $\neq$  compilé).
- ▷ L'interpréteur Python « traduit » les instructions une par une.
- ▷ Pas de fichier binaire généré.
- ▷ L'exécution d'un programme interprété est généralement plus lente qu'un programme compilé.

Macademia



# Installation et prise en main

# Installation de Python

Rendez-vous sur la page

<https://www.python.org/downloads/>

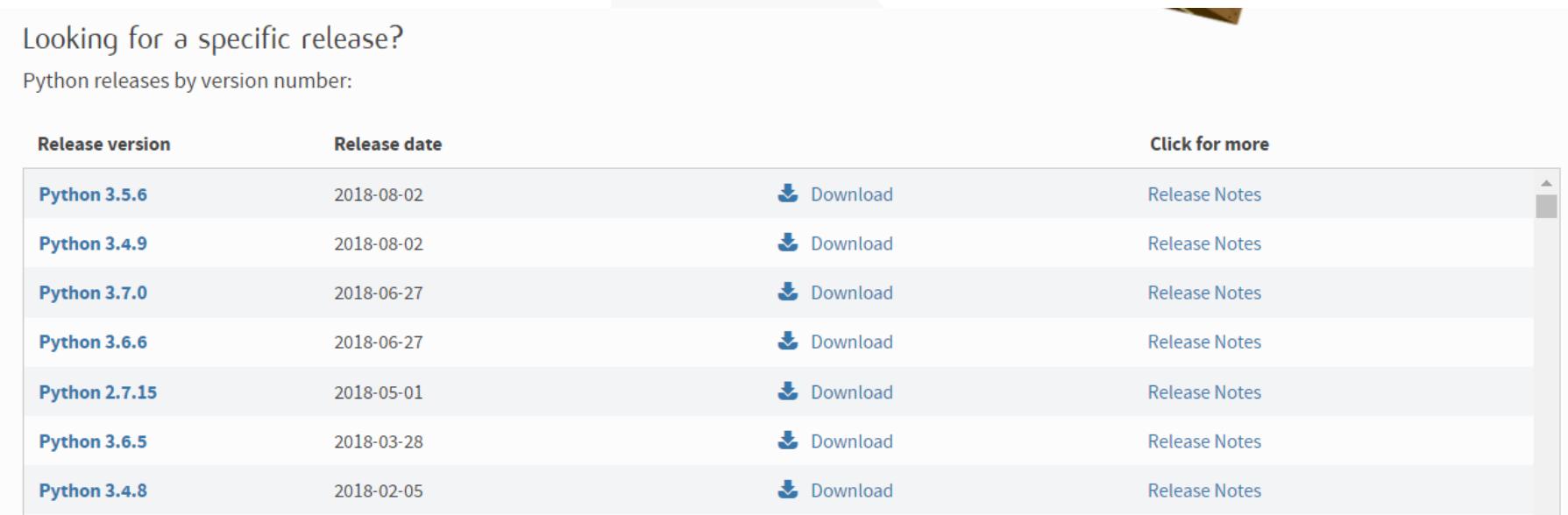
Cliquez sur le bouton

Download Python 3.x.x

correspondant à la dernière version

# Installation de Python

Plus bas dans la page, vous avez également la possibilité de télécharger une release spécifique :



The screenshot shows a table titled "Python releases by version number". The columns are "Release version", "Release date", "Click for more", and "Release Notes". The table lists the following releases:

Release version	Release date	Click for more	Release Notes
<a href="#">Python 3.5.6</a>	2018-08-02	 Download	<a href="#">Release Notes</a>
<a href="#">Python 3.4.9</a>	2018-08-02	 Download	<a href="#">Release Notes</a>
<a href="#">Python 3.7.0</a>	2018-06-27	 Download	<a href="#">Release Notes</a>
<a href="#">Python 3.6.6</a>	2018-06-27	 Download	<a href="#">Release Notes</a>
<a href="#">Python 2.7.15</a>	2018-05-01	 Download	<a href="#">Release Notes</a>
<a href="#">Python 3.6.5</a>	2018-03-28	 Download	<a href="#">Release Notes</a>
<a href="#">Python 3.4.8</a>	2018-02-05	 Download	<a href="#">Release Notes</a>

Notez que les « anciennes » versions de Python sont toujours mises à jour.

# Installation de Python

3. Lancez le fichier d'installation « **python-3.x.x.exe** »



4. Cliquez sur « **Install Now** » (notez bien le chemin de destination).

# Installation de Python

- ▷ Si cela n'a pas été fait lors de l'installation, il faut ajouter Python à la variable d'environnement PATH.
- ▷ Tester la réussite de l'installation en lançant la commande :

```
$ python --version
```

Dans un invite de commande Windows (ou terminal Unix)

# Les différents modes de programmation

- ▷ Il est possible de programmer de deux manières en Python :



En mode **classique**



En mode **interactif**

# Le mode classique

- ▷ Méthode de programmation la plus largement utilisée.
- ▷ Les instructions sont écrites à la suite dans un fichier **.py**
- ▷ Écrire des programmes réutilisables.
- ▷ Possibilité d'utiliser un **IDE**.

macademia

# Le mode interactif

- ▷ Il est possible de tester des « morceaux » de code grâce au mode interactif.
- ▷ Chaque instruction est exécutée de manière **unitaire**.
- ▷ Le code n'est pas stocké : la mémoire est détruite à la fin de l'exécution de l'interpréteur.

macademia

# Le mode interactif

L'accès au mode interactif se fait en lançant la commande

```
C:> python
```

Dans un invite de commande  
Windows

```
$ python3
```

Dans un terminal  
Unix

macademia

# Le mode interactif

- ▷ Le prompt Python est matérialisé par 3 chevrons ( >>> )

```
C:\Users\Abdel> python
Python 3.8.2 (v3.7.0:1bf9cc5093, Apr 17 2020, 04:06:47) [MSC v.1914 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> a = 5
>>> b = 6
>>> c = a + b
>>> print(c)
11
>>> exit(0)

C:\Users\Abdel>
```

# Les avantages d'utiliser un IDE

- ▷ Fournit un éditeur de texte.
- ▷ Coloration syntaxique, autocomplétion, détection d'erreurs, ...
- ▷ Prise en charge de l'interpréteur Python.
- ▷ Fournit généralement une console pour le mode interactif.
- ▷ Eclipse (PyDev), Spyder, Thonny, PyCharm, ...
  - › Pour nos cas pratiques, nous utiliserons PyCharm.

# Installation de PyCharm

1. Rendez-vous sur la page <https://www.jetbrains.com/pycharm/download>
2. Sélectionnez la version « Community » et cliquez sur « Download ».

## Download PyCharm

Windows

macOS

Linux

### Professional

Full-featured IDE  
for Python & Web  
development

DOWNLOAD

Free trial

### Community

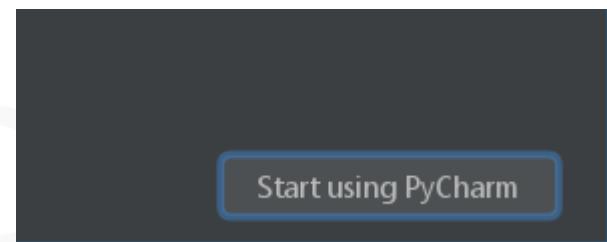
Lightweight IDE  
for Python & Scientific  
development

DOWNLOAD

Free, open-source

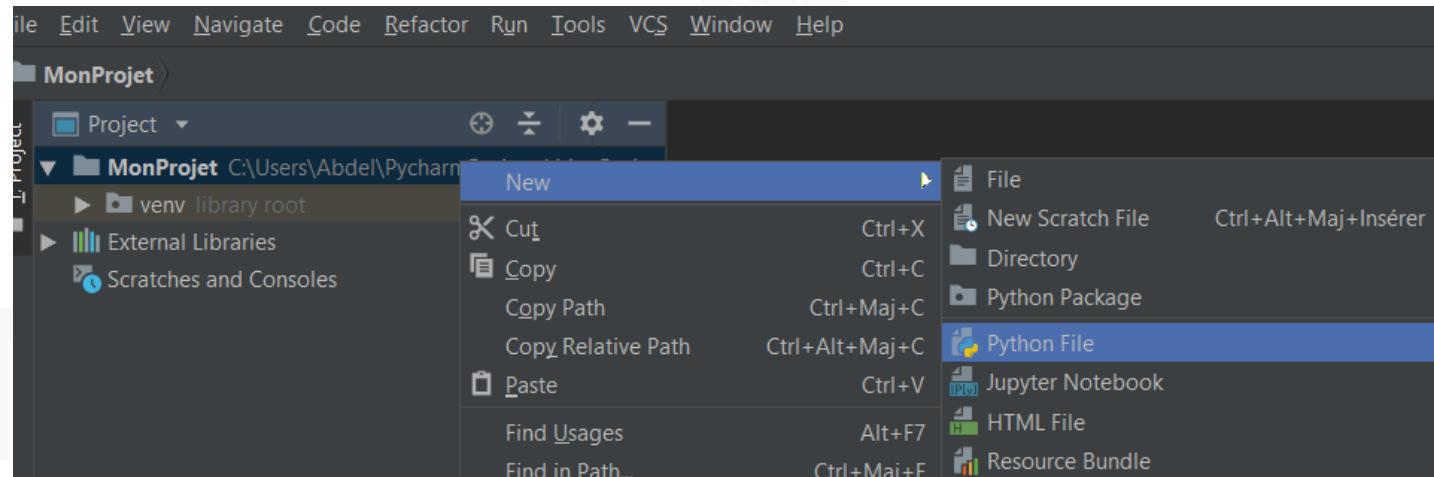
# Installation de PyCharm

3. Lancez le fichier d'installation « **pycharm-community-20yy.x.exe** »
4. Next, next, next, next, ... finish.
5. Lancez **JetBrains PyCharm Community Edition**.
6. Sélectionnez un thème, puis cliquez sur « Start using PyCharm ».



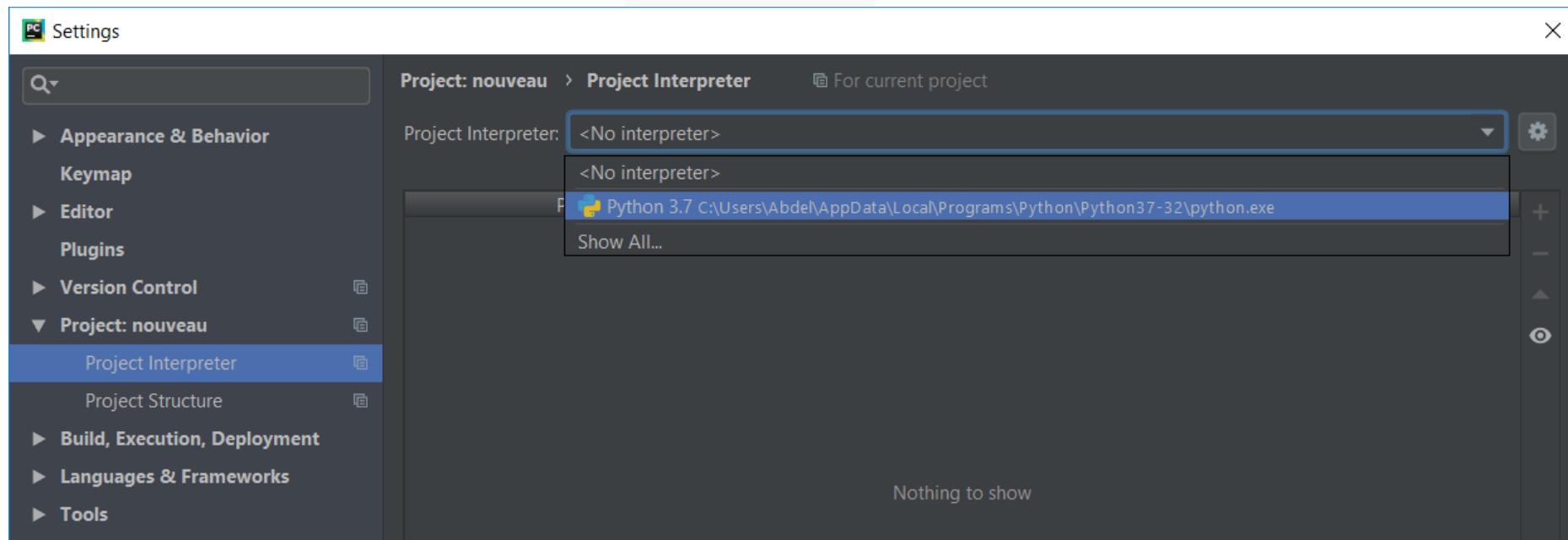
# Installation de PyCharm

7. Créez un nouveau projet. Indiquez le nom du projet et le chemin de répertoire désiré.
8. Une fois le projet créé, créez votre premier fichier Python en faisant un clic droit sur le nom du projet dans la vue à gauche.



# Installation de PyCharm

9. Si l'interpréteur n'est pas configuré par défaut, il faut le définir dans « File > Settings > Project: xxxx > Project Interpreter »





# Syntaxe

# Les instructions

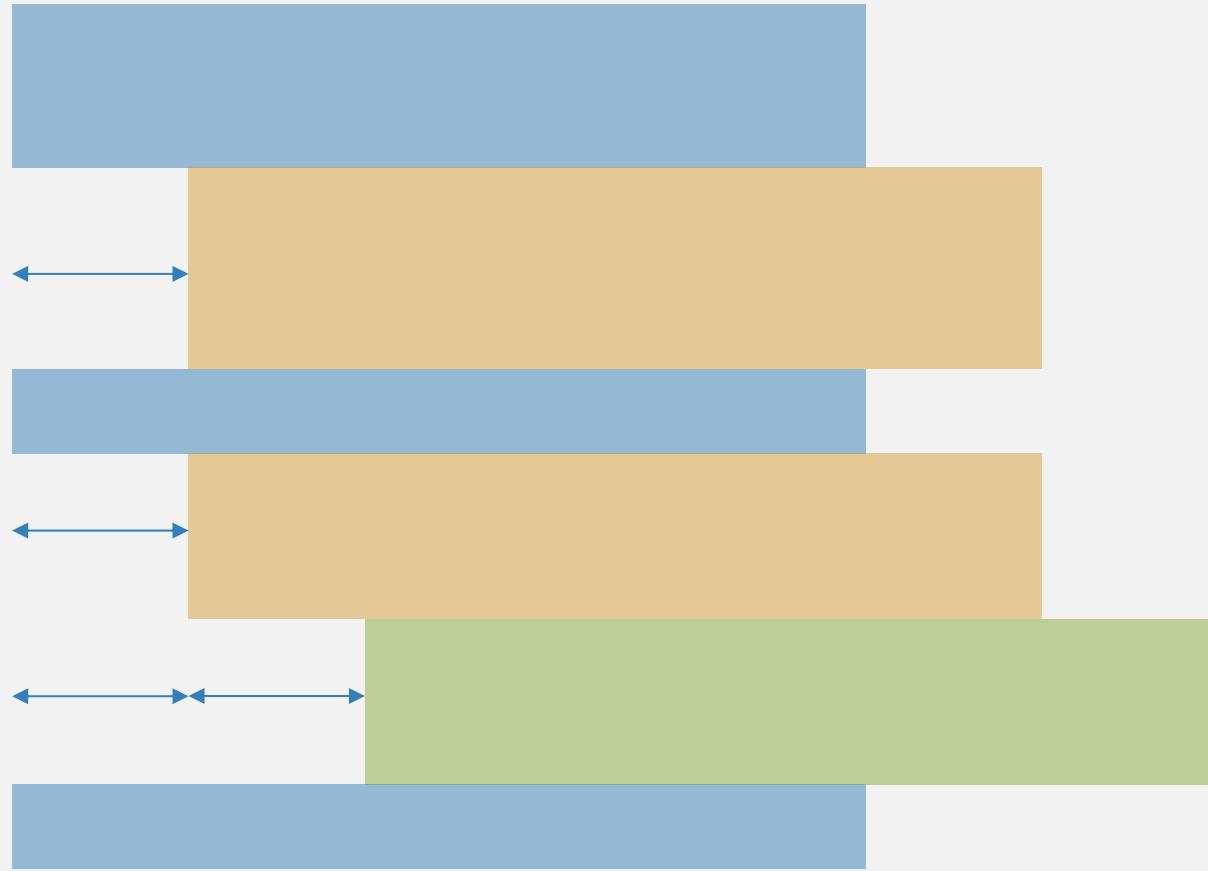
- ▷ En Python, la fin d'une instruction est matérialisée par le caractère « **retour chariot** ».
  - › Chaque ligne correspond à une instruction.
- ▷ Si une instruction est trop longue, il est possible de l'écrire sur plusieurs lignes avec le caractère \ (antislash) à la fin de chaque ligne.
- ▷ Il est possible mais **déconseillé** d'écrire plusieurs instructions sur la même ligne.

Macademia

# L'indentation

- ▷ Il n'existe pas de caractère tels que les accolades ou les crochets pour définir des **blocs d'instructions** en Python.
  - ▷ L'indentation est donc **primordiale** et **obligatoire** pour définir des blocs d'instruction.
  - ▷ Lorsque l'on **entre** dans un bloc, on **ajoute** un niveau d'indentation.  
Lorsque l'on **sort** du bloc, on **réduit** un niveau d'indentation.
- ⓘ La convention PEP 8 préconise une tabulation ou 4 espaces.**

# L'indentation



# Les commentaires

- ▷ En Python, les commentaires sont notés avec le symbole **#** (croisillon).
- ▷ Il est possible d'écrire un commentaire à la suite d'une instruction.
- ▷ Les commentaires sur plusieurs lignes peuvent être écrits avec des **"""** (trois quotes, simples ou doubles), au début et à la fin du commentaire.

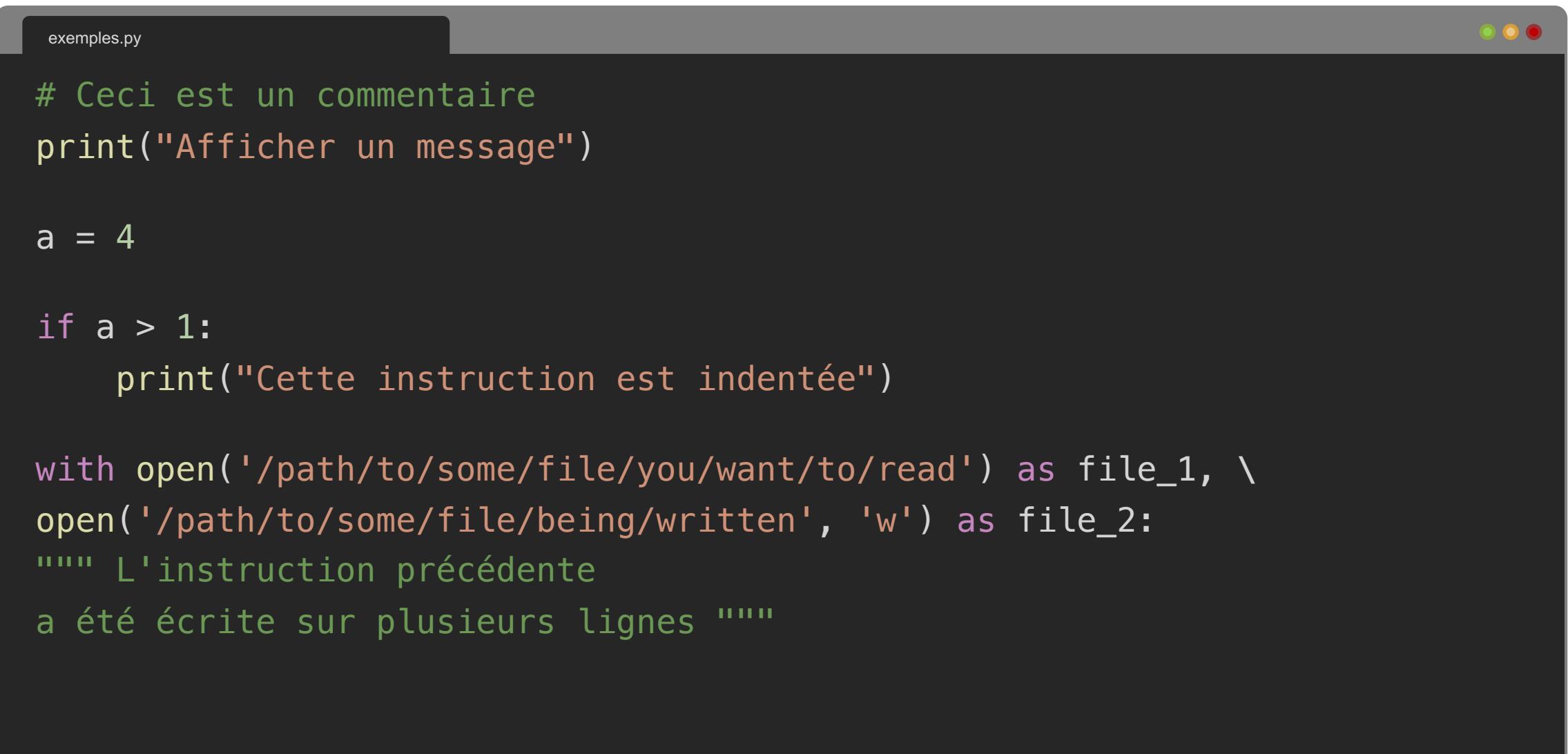
macademia

# Des outils pratiques pour débuter

- ▷ On utilise **print()** pour afficher un message dans la console.
- ▷ On utilise **input()** pour demander à l'utilisateur d'entrer une information au clavier dans la console.

A large, semi-transparent watermark of the word "macademia" in a lowercase, sans-serif font, centered on the slide.

# Quelques exemples



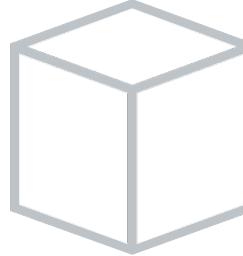
A screenshot of a code editor window titled "exemples.py". The code is written in Python and demonstrates various syntax elements:

```
# Ceci est un commentaire
print("Afficher un message")

a = 4

if a > 1:
    print("Cette instruction est indentée")

with open('/path/to/some/file/you/want/to/read') as file_1, \
open('/path/to/some/file/being/written', 'w') as file_2:
    """ L'instruction précédente
    a été écrite sur plusieurs lignes """
```



# Les variables et les types

# Les variables

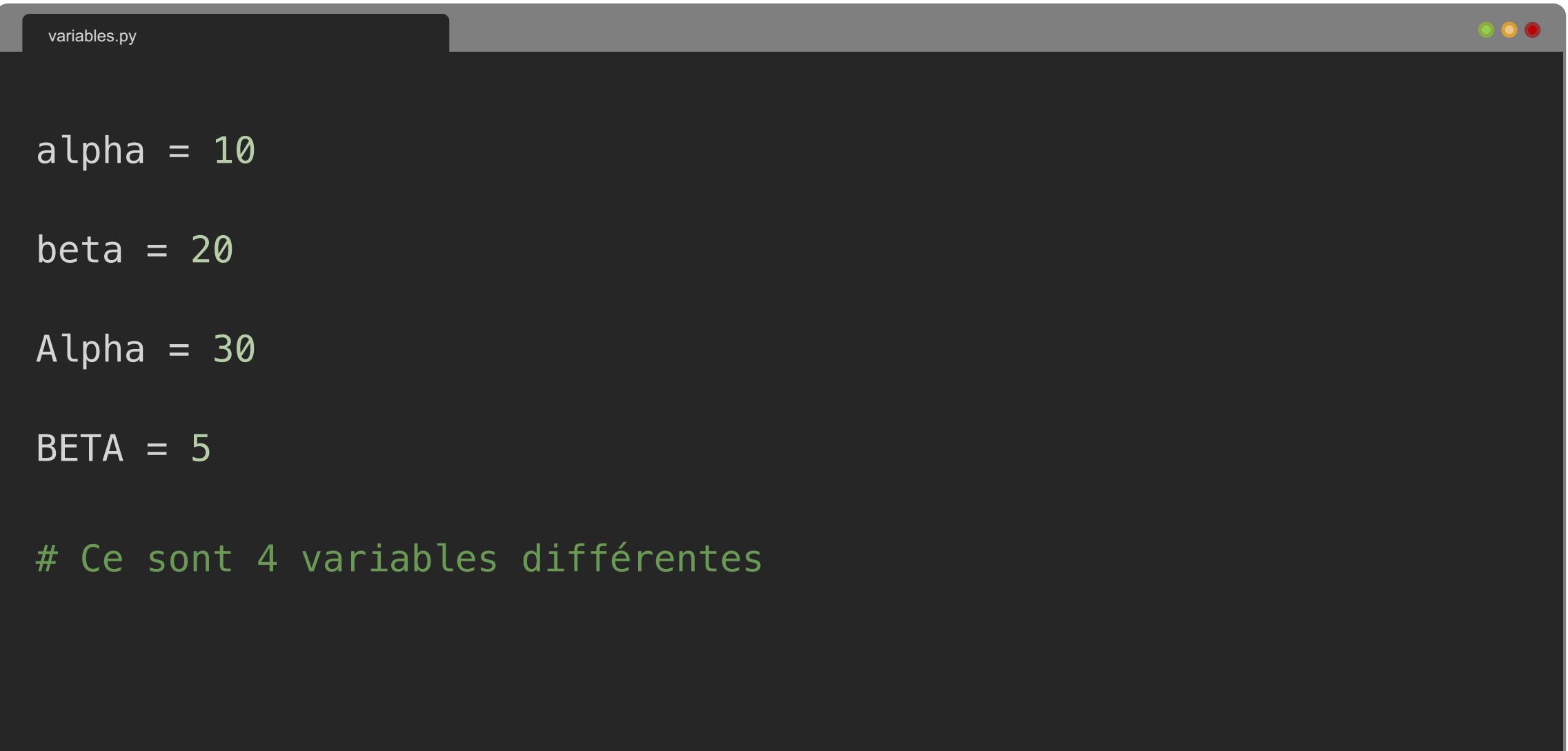
- ▷ Une variable permet de stocker une information.
  - ▷ Le nom est sensible à la casse.
  - ▷ Le nom doit comporter uniquement des caractères alphanumériques ou des \_ (underscore).
  - ▷ Le nom ne peut pas commencer par un chiffre.
- ⓘ La convention PEP 8 préconise la notation\_snake\_case.**

# Affectation

- ▷ On affecte une valeur grâce au symbole **=** (égal).
- ▷ Une variable est **initialisée** lors de son affectation.
  - › Il n'y a pas de déclaration de variable.
- ▷ Le typage est **implicite** et **dynamique**.
- ▷ On supprime une variable grâce à **`del nom_variable`**.

macademia

# Création de variables



```
variables.py
```

```
alpha = 10
beta = 20
Alpha = 30
BETA = 5
# Ce sont 4 variables différentes
```

# Les types de variables

- ▷ En Python, les types de variables de base sont peu nombreux.
- ▷ Il existe des **types simples** : des nombres, des chaînes, etc.
- ▷ Il existe également des types *complexes* appelés « **séquences** », qui peuvent contenir d'autres variables de type simple ou de type complexe.
- ▷ Toutes les variables sont des **objets**.
- ▷ La liste suivante n'est pas exhaustive.

# Les nombres entiers

- ▷ N'importe quel nombre entier, qu'il soit positif ou négatif.
- ▷ En Python, le type entier est **int**.

```
alpha = 10
```

```
beta = -176371091736301
```

macademia

# Les nombres flottants

- ▷ N'importe quel nombre décimal.
- ▷ En Python, le type flottant est **float**.

```
pi = 3.14159
```

```
gamma = -10.0
```

**Le séparateur n'est pas la virgule mais le point.**

# Les booléens

- ▷ Une variable qui n'a que deux états possibles : **vrai** ou **faux**.
- ▷ En Python, le type booléen est **bool**.

```
delta = True
```

```
delta = False
```

macademia

# Les chaînes de caractères

- ▷ Une chaîne de caractères est matérialisée par des **quotes** (simples ou doubles).
- ▷ En Python, le type chaîne de caractères est **str**.
- ▷ Le caractère d'échappement est \ (antislash).

```
omega = "Bonjour à toutes et à tous !"  
epsilon = 'Attention à l\'apostrophe'  
sigma = """
```

# En résumé ...

Type	Définition	Exemple
<b>int</b>	Nombre entier	<b>12</b>
<b>float</b>	Nombre décimal	<b>6.35</b>
<b>bool</b>	Booléen	<b>True</b>
<b>str</b>	Chaîne de caractères	<b>"Coucou"</b>

# Opérations sur les variables

operations.py



```
# Addition  
a + b  
  
# Soustraction  
a - b  
  
# Multiplication  
a * b  
  
# Division  
a / b  
  
# Quotient de la division euclidienne  
a // b
```

# Opérations sur les variables

operations.py



```
# Modulo (reste de la division euclidienne)
a % b

# Puissance (ab)
a ** b

# Incrémentation (valable pour tous les opérateurs)
a += 5

# Concaténation (chaînes de caractères)
chaine = "Hello " + "world"
```

# Connaitre le type d'une variable

```
type_variable.py

omega = "Bonjour à toutes et à tous !"

print(type(omega))

print(type(5) == int)

print(isinstance(5, int))

print(isinstance(5, (int, float)))
```

```
<class 'str'>
True
True
True
```

# Conversion de types

- ▷ En Python, la conversion des types (cast) est **explicite**.

Par exemple :

```
age = 18  
phrase = "J'ai " + age + " ans"
```

provoque une erreur !

# Conversion de types

- ▶ Pour convertir une variable, on indique simplement **le type de destination** avec le nom de la variable entre parenthèses.

```
age = 18  
phrase = "J'ai " + str(age) + " ans"
```

- ▶ La valeur de la variable doit être **compatible** avec le type de destination.
  - › On ne peut convertir un **str** en **int** que si la chaîne est un nombre.

# Astuce pour print()

- ▷ Afin d'afficher des variables de types différents avec **print()**, on peut les séparer grâce à une **virgule**.
- ▷ Cela évite de faire la conversion des types.

```
age = 18  
print("J'ai", age, "ans")
```

macademia

# Les *fstrings*

- ▷ Les *fstrings* permettent de **faciliter la concaténation**.
  - › Il faut préfixer la chaîne de caractère par la lettre **f**.
- ▷ Utilisation des « **{ ... }** » (accolades) dans les chaînes de caractères.
- ▷ L'expression se trouvant dans les accolades est **évaluée**.
  - › Permet notamment d'insérer des noms de variables.
  - › Le cast se fait alors automatiquement.

```
age = 18
phrase = f"J'ai {age} ans"
```

# Les séquences

- ▷ Les séquences sont des variables permettant de stocker un **ensemble de valeurs**.
- ▷ Elles peuvent contenir des éléments **de différents types**.
- ▷ Il existe trois principaux types de séquences :
  - › les **listes**,
  - › les **tuples**,
  - › les **dictionnaires**.

# Les listes

- ▷ Les listes sont des tableaux dynamiques.
- ▷ Les listes (type `list`) sont créées grâce à des [ ] (crochets).
- ▷ Possibilité d'ajouter, modifier, supprimer un élément de la liste.
- ▷ Chaque élément possède un indice qui permet d'y accéder.
- ▷ **Le premier indice d'une liste est 0.**

# Les listes

```
listes.py
```

```
liste1 = [1, 4, 9, 2, 0, -7]

liste2 = [5.3, "chien", 8, "lapin"]

liste3 = [[0, -3], [5, 3, 9]]

liste4 = [10] * 5
# Equivaut à [10, 10, 10, 10, 10]
```

# Accéder aux éléments d'une liste

```
listes.py

liste1 = [1, 4, 9, 2, 0, -7]
liste2 = [5.3, "chien", 8, "lapin"]

print(liste1[0])
# 1

print(liste2[3])
# 'lapin'

print(liste1[-3])
# 2

liste1[1] = 22
# [1, 22, 9, 2, 0, -7]
```

# Accéder aux éléments d'une liste : slicing

```
listes.py

liste1 = [1, 4, 9, 2, 0, -7]
liste2 = [5.3, "chien", 8, "lapin"]

print(liste1[1:3])
# [4, 9]

print(liste2[2:])
# [8, 'lapin']

print(liste1[:5])
# [1, 4, 9, 2, 0]

print(liste1[1:-2])
# [4, 9, 2]
```

# Accéder aux éléments d'une chaîne de caractères

```
chaine.py  
chaine = "Python"  
  
print(chaine[1])  
# 'y'  
  
print(chaine[2:])  
# 'thon'  
  
chaine[3] = 'z'
```

```
Traceback (most recent call last):  
  File "<input>", line 9, in chaine.py  
TypeError: 'str' object does not support item assignment
```

# Opérations sur les listes

- ▷ *Liste.append(element)*
  - › Ajouter un élément à la fin de la liste.
- ▷ *Liste.clear()*
  - › Supprimer tous les éléments de la liste.
- ▷ *Liste.copy()*
  - › Renvoie une copie de la liste.
- ▷ *Liste.count(element)*
  - › Retourne le nombre d'occurrences de l'élément indiqué.
- ▷ *Liste.index(element)*
  - › Retourne l'indice de la première occurrence de l'élément indiqué.

# Opérations sur les listes

- ▷ *Liste.insert(position, element)*
  - › Insérer un élément à une position précise.
- ▷ *Liste.pop(position)*
  - › Retirer l'élément correspondant à l'indice indiqué.
- ▷ *Liste.remove(element)*
  - › Retirer la première occurrence de l'élément indiqué.
- ▷ *Liste.reverse()*
  - › Inverser la liste.
- ▷ *Liste.sort()*
  - › Trier la liste (options possibles).

# Opérations sur les listes

- ▷ *Liste1.extend(Liste2)*
  - › Ajouter les éléments de la liste 2 à la fin de la liste 1.
- ▷ *Liste1 + Liste2*
  - › Concaténation de deux listes.
- ▷ *len(Liste)*
  - › Renvoie la taille de la liste.
- ▷ *element in liste*
  - › Tester si un élément est présent dans une liste.



# Les ranges

- ▷ Les ranges permettent de générer une liste de nombres entiers.
- ▷ La liste est générée selon 1, 2 ou 3 paramètres :
  - › fin (obligatoire),
  - › début,
  - › pas.
- ▷ L'ordre des paramètres est le suivant : `range(debut, fin, pas)`

Le type de la liste générée ne sera pas `list`, mais `range`.  
**ⓘ Il est possible de les convertir en `list` facilement.**

# Les ranges

```
ranges.py

print(list(range(5)))
# [0, 1, 2, 3, 4]

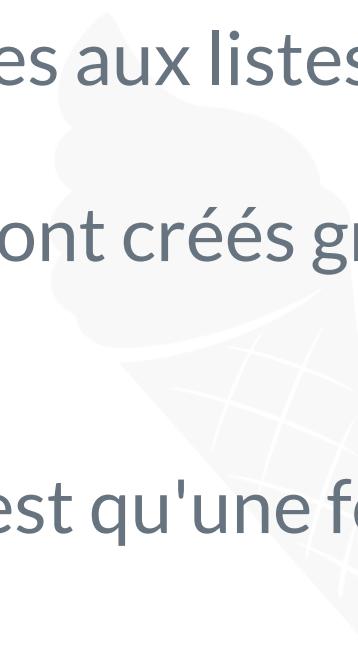
print(list(range(1, 5)))
# [1, 2, 3, 4]

print(list(range(1, 6, 2)))
# [1, 3, 5]

print(list(range(10, 0, -1)))
# [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

# Les tuples

- ▷ Les tuples sont semblables aux listes.
- ▷ Les tuples (type **tuple**) sont créés grâce à des () (parenthèses, mais non obligatoires).
- ▷ La principale différence est qu'une fois construits, on **ne peut pas les modifier**.



# Les tuples

```
tuples.py

tuple1 = (1, 2, 3)

print(tuple1)
# (1, 2, 3)

print(tuple1[1])
# 2

tuple2 = 'a', 'b'

print(tuple2)
# ('a', 'b')
```

# Les tuples

```
tuples.py

tuple3 = (1, 3, ('a', 'b'), 4)

print(tuple3[2][1])
# 'b'

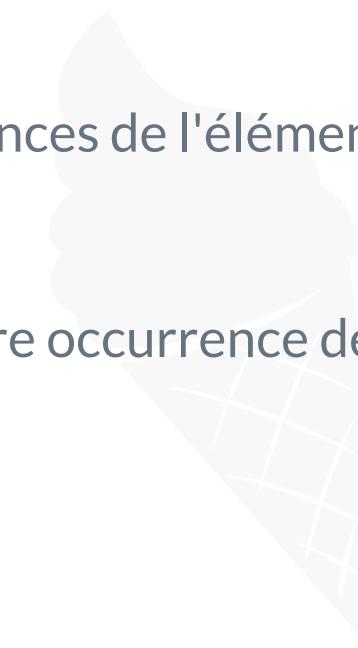
print((1, 2, 3, 4)[2:])
# (3, 4)

tuple1[0] = 5
```

```
Traceback (most recent call last):
  File "<input>", line 9, in tuples.py
TypeError: 'tuple' object does not support item assignment
```

# Opérations sur les tuples

- ▷ ***tpl.count(element)***
  - › Retourne le nombre d'occurrences de l'élément indiqué.
  
- ▷ ***tpl.index(element)***
  - › Retourne l'indice de la première occurrence de l'élément indiqué.



# macademia

# Les dictionnaires

- ▷ Les dictionnaires sont des ensembles de **clés-valeurs**.
- ▷ Les dictionnaires (type **dict**) sont créés grâce à des { } (accolades).
- ▷ Pas de notion d'ordre : chaque élément est identifié par sa clé.
- ▷ Par conséquent, chaque clé est **unique**.
- ▷ Possibilité d'ajouter, modifier, supprimer un élément.

# Les dictionnaires

```
dictionnaires.py

dico = {'alpha': 1, 'beta': 2, 'gamma': 3, 'delta': 4}

print(dico['beta'])
# 2

dico['alpha'] = 'un'

print(dico['alpha'])
# un

dico['epsilon'] = 5

print(dico)
# {'alpha': 'un', 'beta': 2, 'gamma': 3, 'delta': 4, 'epsilon': 5}
```

# Opérations sur les dictionnaires

- ▷ ***dico.clear()***
  - › Supprimer tous les éléments du dictionnaire.
- ▷ ***dico.copy()***
  - › Retourne une copie du dictionnaire.
- ▷ ***dict.fromkeys(Liste)***
  - › Crée un nouveau dictionnaire à partir des clés indiquées dans la liste (les valeurs sont vides).
- ▷ ***dico.get(cle)***
  - › Renvoie la valeur associée à la clé.

# Opérations sur les dictionnaires

- ▷ ***dico.items()***
  - › Renvoie le dictionnaire sous forme de liste de tuples.
- ▷ ***dico.keys()***
  - › Renvoie la liste des clés.
- ▷ ***dico.values()***
  - › Renvoie la liste des valeurs.
- ▷ ***dico.setdefault(*cle*, *valeur*)***
  - › Ajoute un couple de clé-valeur uniquement s'il n'est pas déjà présent dans le dictionnaire.



# Opérations sur les dictionnaires

## ▷ `dico.pop(cle)`

- › Renvoie la valeur associée à la clé et supprime le couple de clé-valeur.

## ▷ `dico.popitem()`

- › Renvoie le dernier couple de clé-valeur ajouté et le supprime.

## ▷ `dico1.update(dico2)`

- › Ajoute les couples de `dico2` à `dico1` si la clé n'est pas présente dans `dico1`, et met à jour la valeur dans `dico1` si la clé est déjà présente.

## ▷ `cle in dico`

- › Vérifie si la clé est présente dans le dictionnaire.

# En résumé ...

Type	Définition	Exemple
<b>list</b>	Une liste de valeurs	<code>[5, -3, 6, "hello", 4.1, 0, 3]</code>
<b>tuple</b>	Une liste non-modifiable	<code>(3, 2, -8, (0, 1, 3), 5)</code>
<b>dict</b>	Des couples de clés-valeurs	<code>{'nom' : 'Jean', 'age' : 38}</code>

# Affectation par référence et par copie

`var2 = var1`

- ▷ En Python, l'affectation se fait **par référence**.
  - › La référence mémoire de `var1` la même pour `var2`.
- ▷ Seuls les types **mutables** sont concernés.
  - › Pseudo copie pour les `int`, `str`, `tuple`, etc.

# Affectation de types immuables

```
a = 'toto'

print(a)
# toto

b = a
print(b)
# toto

b = 'tata'

print(a)
# toto
print(b)
# tata
```

# Affectation de types mutables

```
affectations.py

liste1 = [5, 6, 1, 3, 7, 4, 3, 4, 0, 2]

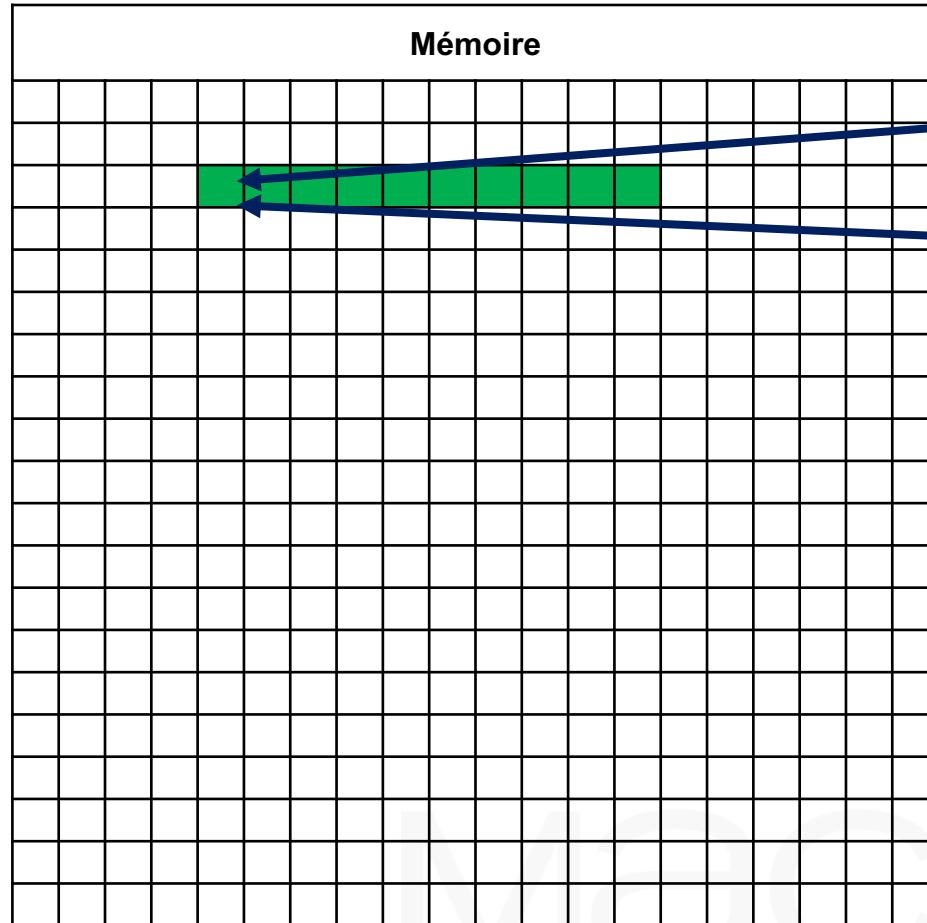
liste2 = liste1

print(liste2)
# [5, 6, 1, 3, 7, 4, 3, 4, 0, 2]

liste1[1] = 'a'
liste2.pop()

print(liste1)
# [5, 'a', 1, 3, 7, 4, 3, 4, 0]
print(liste2)
# [5, 'a', 1, 3, 7, 4, 3, 4, 0]
```

# Affectation par référence - que s'est-il passé ?

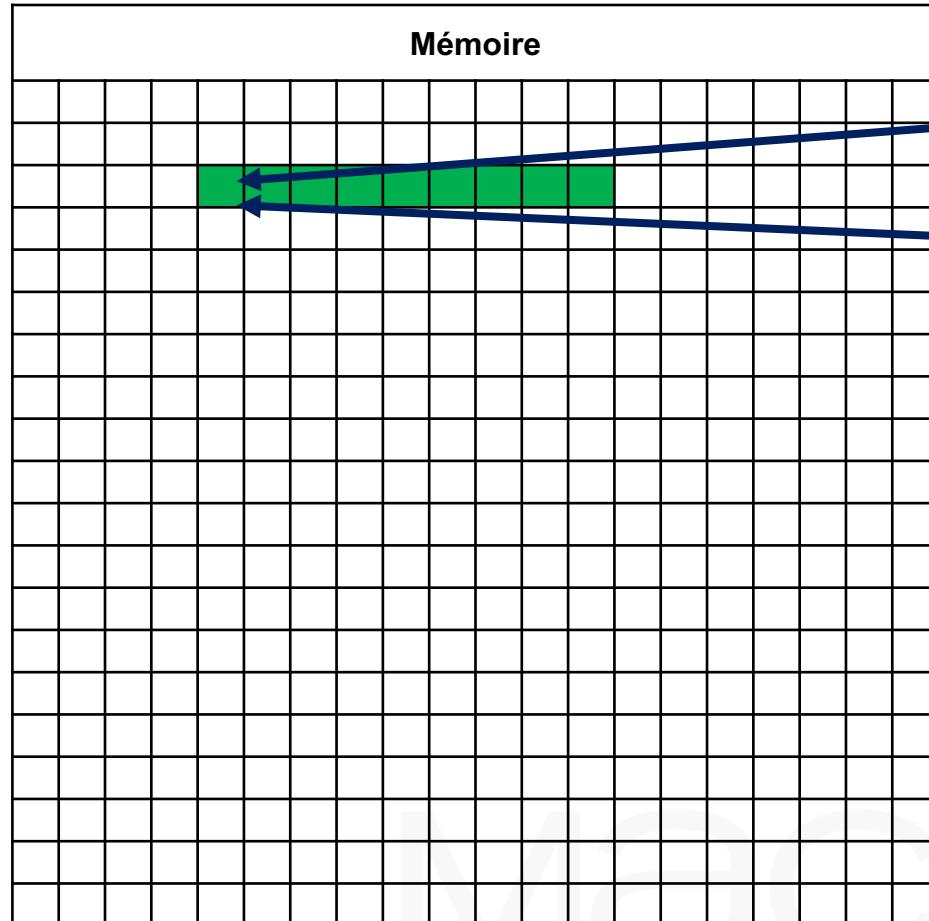


liste1

liste2

```
liste1 = [5, 6, 1, 3, 7, 4, 3, 4, 0, 2]  
liste2 = liste1
```

# Affectation par référence - que s'est-il passé ?



```
liste1 = [5, 6, 1, 3, 7, 4, 3, 4, 0, 2]
liste2 = liste1
liste2.pop()
print('Liste 1 : ', liste1)
print('Liste 2 : ', liste2)
```

```
Liste 1 : [5, 6, 1, 3, 7, 4, 3, 4, 0]
Liste 2 : [5, 6, 1, 3, 7, 4, 3, 4, 0]
```

# La composition

- ▷ Il est possible d'affecter plusieurs valeurs à plusieurs variables en même temps.
- ▷ Pour cela on utilise des **tuples**.
- ▷ Très pratique pour les retours de fonction.

ⓘ Cette méthode peut poser des difficultés de lisibilité.

# La composition

```
composition.py

a, b, c = 1, 3, 'a'

print(a)
# 1

print(b)
# 3

print(c)
# 'a'
```

# Inverser la valeur de deux variables

composition.py

```
a = 5
b = 8

b, a = a, b

print(a)
# 8

print(b)
# 5
```

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Les structures de contrôle

# Les blocs d'instructions

- ▷ Un bloc d'instruction s'écrit comme cela :

*en-tête:*

*instruction 1*

*instruction 2*

*...*

*instruction n*

**❶ Pour rappel, l'indentation en Python est obligatoire.**

# Les conditions

conditions.py

```
if test:  
    ...  
elif test:  
    ...  
elif test:  
    ...  
elif test:  
    ...  
else:  
    ...
```



# Les opérateurs conditionnels

Opérateur	Signification	Exemple
<code>==</code>	Egal à	<code>a == b</code>
<code>!=</code>	Different de	<code>a != b</code>
<code>&lt;</code>	Strictement inférieur à	<code>a &lt; b</code>
<code>&lt;=</code>	Inférieur ou égal à	<code>a &lt;= b</code>
<code>&gt;</code>	Strictement supérieur à	<code>a &gt; b</code>
<code>&gt;=</code>	Supérieur ou égal à	<code>a &gt;= b</code>
<code>and</code>	La première condition ET la deuxième condition	<code>a &gt; b and c &lt; d</code>
<code>or</code>	La première condition OU la deuxième condition (inclusif)	<code>a &gt; b or c &lt; d</code>
<code>not</code>	L'inverse de la condition	<code>not a &gt; b</code>
<code>in</code>	La présence d'un élément dans un autre	<code>a in b</code>
<code>is</code>	L'égalité des adresses mémoire	<code>a is b</code>

# Les conditions

```
conditions.py

age = input("Quel âge avez-vous ?")
age = int(age) # Les inputs sont de type str

if age < 18:
    print("Vous êtes mineur")
elif age >= 18 and age < 120:
    print("Vous êtes majeur")
else:
    print("Vous êtes encore en vie !")
```

# Les conditions

```
conditions.py

age = input("Quel âge avez-vous ?")
age = int(age) # Les inputs sont de type str

if age < 18:
    print("Vous êtes mineur")
elif 18 <= age < 120:
    print("Vous êtes majeur")
else:
    print("Vous êtes encore en vie !")
```

# Les boucles *while*

boucles.py

```
while test:  
    instruction 1  
    instruction 2  
    ...  
    instruction n
```

# Les boucles *while*

```
boucles.py

i = 0

while i < 10:
    print(i)
    i += 1

personnes = ['Laurence', 'Thibaut', 'Muriel', 'Anthony']

i = 0

while personnes[i] != 'Muriel' and i < len(personnes):
    print('La personne est ', personnes[i])
    i += 1
```

# Les boucles *for*

boucles.py

```
for element in iterable:  
    instruction 1  
    instruction 2  
    ...  
    instruction n
```

# Les boucles *for*

```
boucles.py  
  
liste = ['a', 'b', 'c']  
  
for lettre in liste:  
    print(lettre)
```

```
a  
b  
c
```

# Instructions de branchement

- ▷ **break** : permet d'interrompre l'exécution d'une boucle (ou autre bloc d'instruction).
- ▷ **continue** : permet de passer au tour de boucle suivant.
- ▷ **return** : pour les retours de fonctions.
- ▷ **pass** : permet de passer à l'instruction suivante.

Macademia

# Instructions de branchement

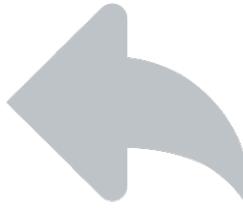
```
boucles.py

while True:
    # Cette condition n'a pas d'arrêt
    lettre = input("Tapez Q pour quitter : ")
    if lettre == "Q" :
        print("Fin de la boucle")
        break

i = 0
while i < 7:
    i += 1
    if i == 3:
        continue
    print("La variable i : ", i)
```

# En résumé

**Dans ce chapitre, nous avons vu ...**



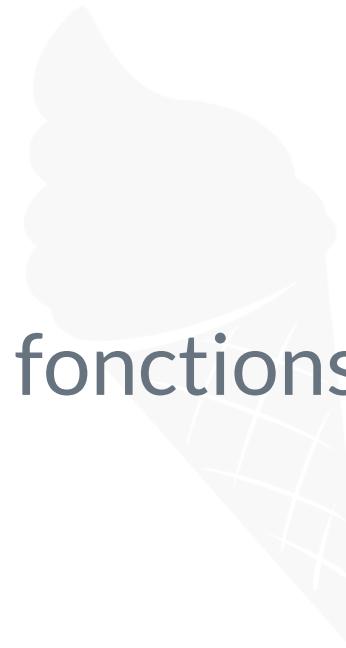
# Les fonctions

# Les fonctions

- ▷ Problématique : nous avons un morceau de code qui est utilisé à plusieurs reprises dans notre programme.
  - › Comment simplifier cela ?
- ▷ Plutôt que de recopier plusieurs fois les mêmes morceaux de code, on va écrire des **fonctions**, qui peuvent être appelées à tout moment.
- ▷ Permettent de **factoriser** le code et d'éviter les **redondances**.
- ▷ Utilisation du mot-clé **def**.

# Les fonctions

Pouvez-vous citer des fonctions que nous avons déjà vu ?



Macademia

# Les fonctions - comment ça marche ?

Appel de la fonction

fonction



Macademia

# Les fonctions - syntaxe

fonctions.py

```
def nom_fonction(param_1, param_2, param_n):  
    instruction 1  
    instruction 2  
    ...  
    instruction n
```

# Les fonctions *simples*

fonctions.py

```
def afficher_message_erreur():
    # Corps de la fonction
    print("Vous avez fait une erreur !")

afficher_message_erreur() # Appel à la fonction
```

You have made an error !

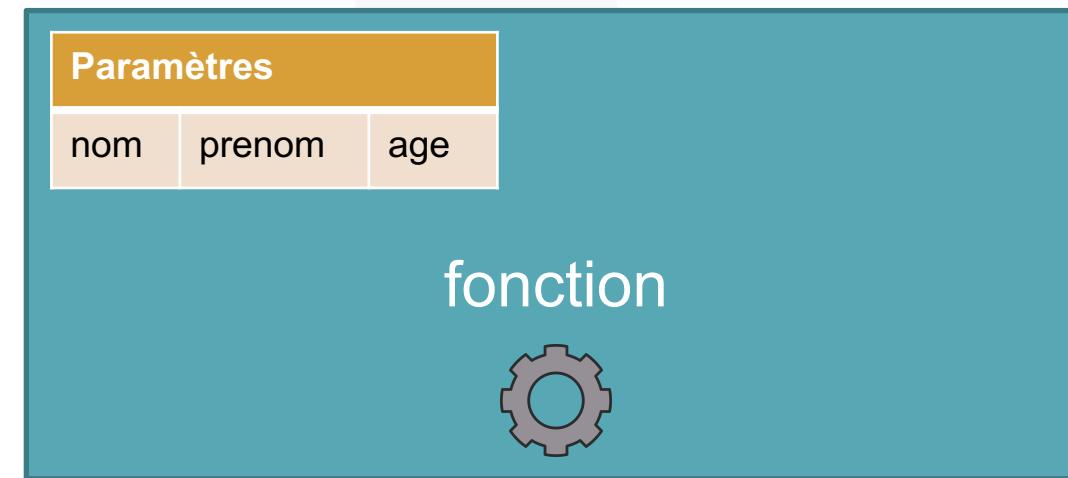
# Les paramètres

- ▷ On souhaite rendre notre fonction plus générique.
- ▷ Pour cela, on peut passer des **paramètres en argument** de la fonction (entre parenthèses).
- ▷ En Python, le passage des paramètres se fait :
  - › par copie pour les paramètres *simples* ;
  - › par référence pour les paramètres *complexes*.

**Attention, l'ordre des paramètres a une importance lors de l'appel.**

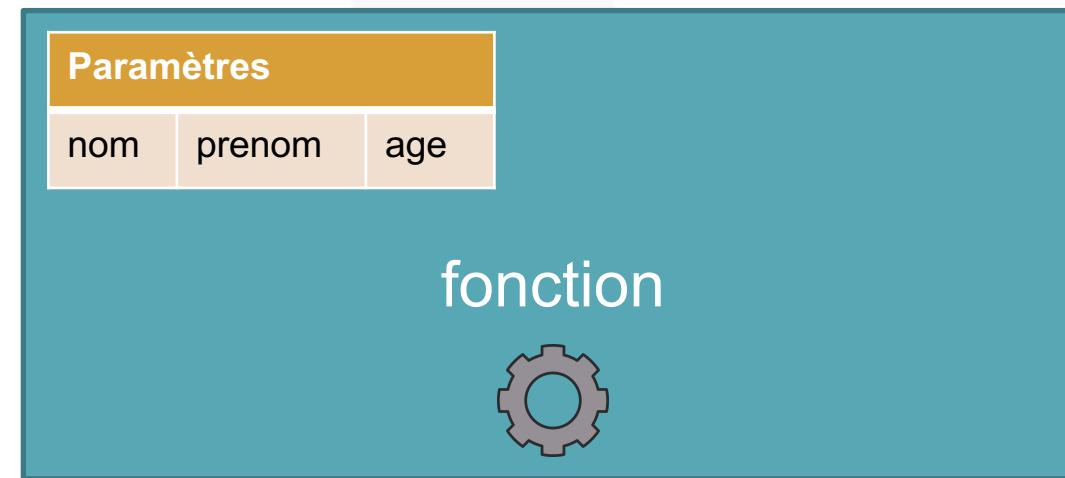
# Les paramètres - comment ça marche ?

Appel de la fonction	
nom	Doe
prenom	John
age	38



# Les paramètres - comment ça marche ?

Appel de la fonction	
nom	Personne
prenom	Jean
age	53



# Les fonction avec paramètres

fonctions.py

```
def afficher_infos_personne(nom, prenom, age):  
    print(f"You vous appelez {prenom} {nom}, vous avez {age} ans")  
  
afficher_infos_personne("Doe", "John", 38)  
afficher_infos_personne("Personne", "Jean", 53)
```

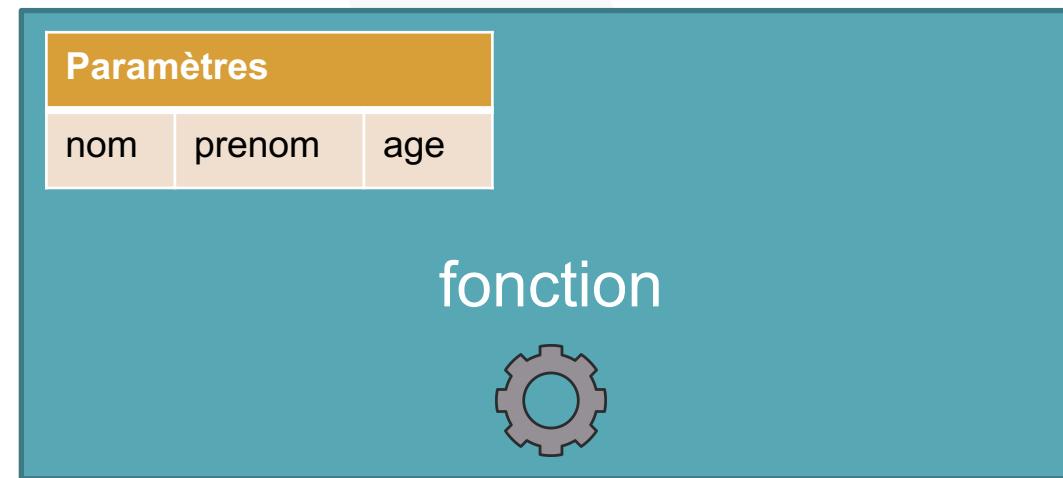
```
Vous vous appelez John Doe, vous avez 38 ans  
Vous vous appelez Jean Personne, vous avez 53 ans
```

# Les fonctions avec retour

- ▷ Une fonction est capable de retourner une valeur.
  - › Par exemple, le résultat d'un calcul.
- ▷ Utilisation du mot-clé **return**.
  - › Cette instruction de branchement doit être **la dernière instruction** de la fonction.
  - › Il est possible de faire des *return* différents selon une condition, mais il faut faire attention que tous les cas de figure soient couverts.
- ▷ Il n'existe pas de retour multiple : on ne retourne **qu'une valeur**.
  - › Par contre, on peut retourner des séquences.

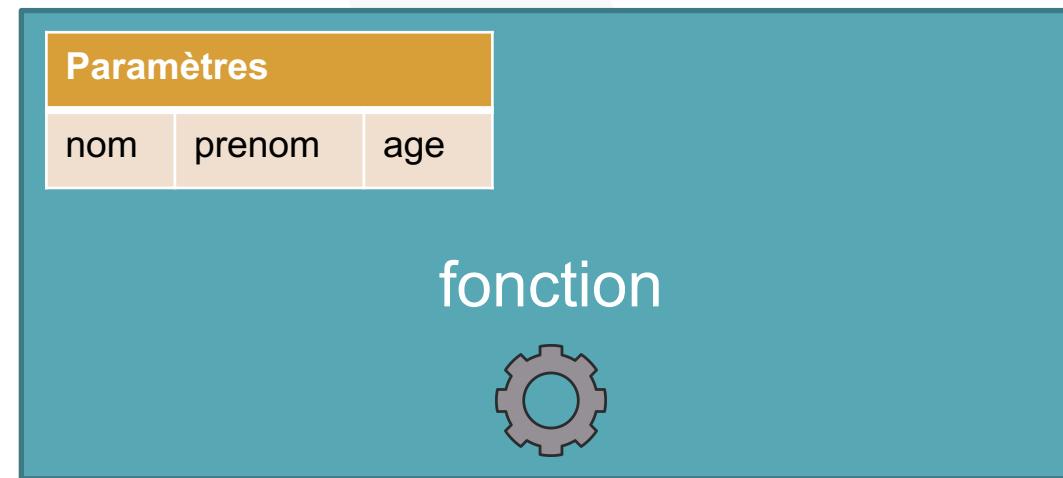
# Les fonctions avec retour

Appel de la fonction	
nom	Doe
prenom	John
age	38



# Les fonctions avec retour

Appel de la fonction	
nom	Personne
prenom	Jean
age	53



# Les fonctions avec retour

fonctions.py

```
def calcul_age_annee_prochaine(nom, prenom, age):  
    age_annee_prochaine = age + 1  
    return age_annee_prochaine  
  
age_john = calcul_age_annee_prochaine("Doe", "John", 38)  
age_jean = calcul_age_annee_prochaine("Personne", "Jean", 53)  
  
print(f"""L'année prochaine, John aura {age_john} ans  
et Jean aura {age_jean} ans""")
```

```
L'année prochaine, John aura 39 ans et Jean aura 54 ans
```

# Les fonctions avec retour

fonctions.py

```
def calcul_age_annee_prochaine(nom, prenom, age):
    # On peut simplifier ainsi
    return age + 1

age_john = calcul_age_annee_prochaine("Doe", "John", 38)
age_jean = calcul_age_annee_prochaine("Personne", "Jean", 53)

print(f"""L'année prochaine, John aura {age_john} ans
et Jean aura {age_jean} ans""")
```

```
L'année prochaine, John aura 39 ans et Jean aura 54 ans
```

# Portée des variables

- ▷ Comme vu dans l'exemple précédent, on peut créer des variables dans une fonction.
  - › À ne pas confondre avec les paramètres.
- ▷ La portée d'une variable se limite à la fonction dans lequel elle a été définie (portée locale).
  - › Une fonction n'a pas accès aux variables du bloc principal.
  - › On peut créer des variables de même nom dans des fonctions différentes.
- ▷ Le mot-clé **global** permet d'outrepasser cette règle.
  - › Les variables globales sont accessibles partout.

# Portée des variables - subtilité de Python

- ▷ Une variable *nait* au moment où on l'a définie, et *meurt* à la fin de la fonction dans laquelle elle a été définie (ou si elle a été détruite).
- ▷ **Ceci n'est pas vrai pour les blocs de type `if`, `while` ou `for`.**
- ▷ Une variable créée dans un bloc `if` sera accessible même une fois sorti du bloc `if`.
  - › C'est la même chose pour les boucles.

A large, semi-transparent watermark of the word "macadem" in a stylized, lowercase font, with a small "ia" at the end. It is positioned vertically in the center of the slide.

macadem

# Portée des variables

fonctions.py

```
def une_fonction():
    a = 3

def une_autre_fonction():
    global a
    a = 3

a = 2
une_fonction()
print(a)
une_autre_fonction()
print(a)
```

2

3

# Portée des variables - particularité de Python

```
fonctions.py

a = 5

if True:
    b = 8
    print(a)

print(b)
```

```
5
8
```

# Appel avec des étiquettes

- ▷ Précédemment nous avons vu que l'ordre des paramètres a une importance lors de l'appel.
- ▷ Il est possible de préciser le nom des paramètres lors de l'appel.
- ▷ Ainsi, l'ordre n'a plus besoin d'être respecté.

The Macademia logo is displayed as a large, semi-transparent watermark across the bottom of the slide. It consists of the word "macademia" in a lowercase, sans-serif font, where each letter is slightly tilted upwards and to the right. The letters are white, making them stand out against the darker background of the slide.

# Appel avec des étiquettes

fonctions.py

```
def afficher_infos_personne(nom, prenom, age):
    print(f"You vous appelez {prenom} {nom} et vous avez {age} ans")

afficher_infos_personne("Doe", "John", 38)
afficher_infos_personne(age=53, nom="Personne", prenom="Jean")
```

```
Vous vous appelez John Doe et vous avez 38 ans
Vous vous appelez Jean Personne et vous avez 53 ans
```

# Les paramètres par défaut

- ▷ Il est possible d'indiquer des valeurs par défaut aux paramètres.
- ▷ Si la valeur est renseignée lors de l'appel, la fonction utilisera celle-ci, sinon la fonction utilisera la valeur par défaut.
- ▷ Les paramètres ayant une valeur par défaut doivent être placés **après** ceux n'ayant pas de valeur par défaut.

macademia

# Les paramètres par défaut

fonctions.py

```
def calcul_aire(longueur, largeur=5):
    aire = longueur * largeur
    print(f'L'aire du rectangle est {aire} cm²')
```

```
calcul_aire(7, 4)
calcul_aire(10)
```

```
L'aire du rectangle est 28 cm²
L'aire du rectangle est 50 cm²
```

# Nombre variable de paramètres

- ▷ Il est possible de définir une fonction avec un nombre variable de paramètres.
- ▷ La liste des paramètres entrés lors de l'appel sont réunis dans un **tuple**.
- ▷ On utilise le symbole **\*** (astérisque) avant le nom du paramètre pour indiquer que sa taille est variable.

A large, semi-transparent watermark of the word "macadem" in a lowercase, sans-serif font, with a smaller "ia" at the end, centered on the slide.

# Nombre variable de paramètres

fonctions.py

```
def moyenne(*params):  
    print(sum(params) / len(params))
```

```
moyenne(1, 4, 6, 8)
```

```
moyenne(12, 18, 6, 15, 4, 11)
```

4.75

11

# Les paramètres nommés

- ▷ Il est possible de définir une fonction avec un nombre variable de paramètres et leur associer un nom.
- ▷ La liste des paramètres entrés lors de l'appel sont réunis dans un dictionnaire.
- ▷ On utilise le symbole **\*\*** (double astérisque) avant le nom du paramètre pour indiquer que sa taille est variable et que ses paramètres sont nommés.

# Les paramètres nommés

```
fonctions.py
```

```
def connexion(**kwargs):
    if 'ip' in kwargs:
        print('Connexion avec IP', kwargs['ip'])
    elif 'host' in kwargs:
        print('Connexion avec hostname', kwargs['host'])
    else:
        print('IP ou hostname manquant')

connexion(ip='127.0.0.1', user='abdel')
```

# Bonnes pratiques

- ▷ Les fonctions doivent être écrites **avant** le code principal.
- ▷ La convention PEP 8 préconise **2 sauts de lignes** entre les fonctions et le code principal.
- ▷ Une bonne pratique est de placer le code principal dans une fonction `main()`.
- ▷ Une autre bonne pratique est de précéder l'appel à la fonction `main()` par la condition suivante : `if __name__ == "__main__"`

# Bonnes pratiques

fonctions.py

```
def calcul_aire(longueur, largeur=5):
    aire = longueur * largeur
    print("L'aire du rectangle est", aire, "cm2")

if __name__ == "__main__":
    calcul_aire(7, 4)
    calcul_aire(10)
```

# Bonnes pratiques

fonctions.py

```
def calcul_aire(longueur, largeur=5):
    aire = longueur * largeur
    print("L'aire du rectangle est", aire, "cm2")

def main():
    calcul_aire(7, 4)
    calcul_aire(10)

if __name__ == "__main__":
    main()
```

# La docstring

- ▷ La docstring est un **commentaire** particulier qui décrit une fonction.
- ▷ Il s'agit d'une simple phrase écrite dans un commentaire multilignes (avec **3 quotes**).
- ▷ Une docstring bien rédigée permettra de **générer** plus facilement **la documentation** du projet.
- ▷ Il est conseillé de rédiger la docstring en anglais.

# Les expressions lambda

- ▷ Les **fonctions lambda** (ou fonctions anonymes) permettent de créer des fonctions ... qui ne portent pas de nom.
- ▷ Elles permettent de créer de petites fonctions **sans retour** grâce au mot-clé **lambda**.
- ▷ On les utilise comme une déclaration de variable.
- ▷ Particularités des lambdas en Python :
  - › On ne peut les écrire que sur une ligne.
  - › On ne peut pas avoir plus d'une instruction dans la fonction.

# Les expressions lambda

lambdas.py



```
# Fonction lambda créée dans la variable "moyenne"  
moyenne = lambda x, y: (x + y) / 2
```

```
# On appelle ensuite "moyenne" comme une fonction  
print(moyenne(7, 5))
```

# Les générateurs

- ▷ Souvenez-vous des itérateurs ...

```
for val in iterable:  
    ...
```

- ▷ Il est possible de créer ses propres *itérables* grâce aux **générateurs**.
- ▷ Les générateurs sont des fonctions.
- ▷ Les valeurs sont « renvoyées » grâce au mot-clé **yield**

# Les générateurs

generateurs.py

```
def generateur():
    yield "Salut"
    yield "Comment"
    yield "Ça va ?"

obj_gen = generateur()

for val in obj_gen:
    print(val)
```

```
Salut
Comment
Ça va ?
```

# Les générateurs

generateurs.py

```
def generateur():
    yield "Salut"
    yield "Comment"
    yield "Ça va ?"

obj_gen = generateur()

print(next(obj_gen))
print(next(obj_gen))
print(next(obj_gen))
print(next(obj_gen))
```

```
Salut
Comment
Ça va ?
Traceback (most recent call last):
  print(next(iterable))
StopIteration
```

# Les générateurs

generateurs.py

```
def generateur():
    yield "Salut"
    yield "Comment"
    yield "Ça va ?"
```

```
obj_gen = generateur()

print(type(obj_gen))
```

```
<class 'generator'>
```

# Les générateurs

generateurs.py

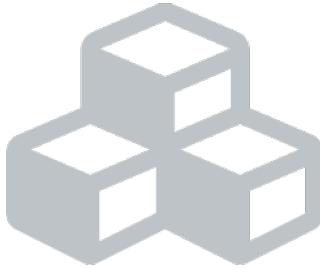
```
def generateur_bissextile(annee):
    while annee % 4 != 0:
        annee += 1

    while annee <= 2020:
        if annee % 4 == 0 and not annee % 100 == 0 or annee % 400 == 0:
            yield annee
        annee += 4

obj_gen = generateur_bissextile(1975)
```

# En résumé

**Dans ce chapitre, nous avons vu ...**



# L'approche Objet



Pouvez-vous citer des objets dans cette salle ?

Macademia

# Un objet ... qu'est-ce que c'est ?

Proposition de définition :

Toute entité modélisable à laquelle on peut associer un ensemble de caractéristiques et de comportements.

Macademia

# Un objet ... qu'est-ce que c'est ?

Toute **entité** modélisable à laquelle on peut associer un ensemble de **caractéristiques** et de **comportements**.

**Entité** : l'objet en lui-même, stocké dans une **variable**.

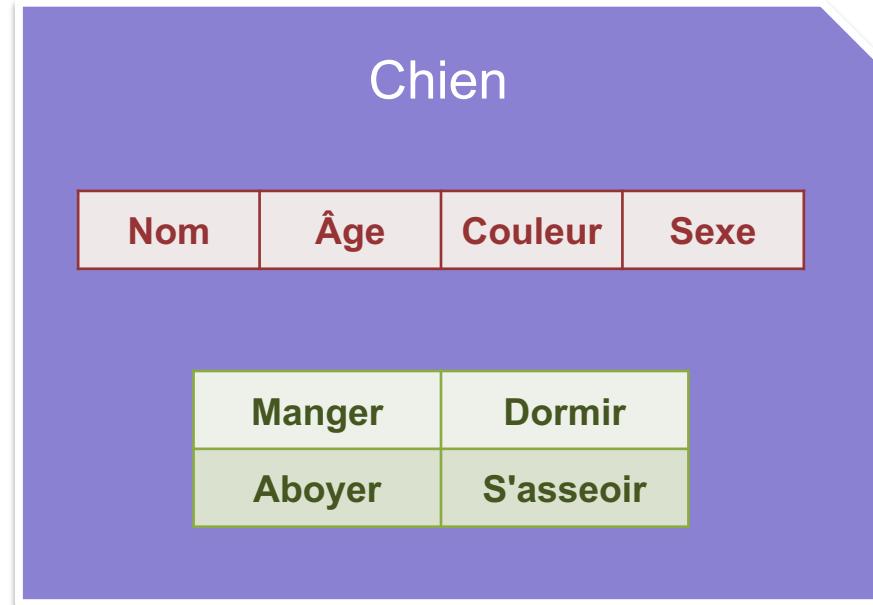
**Caractéristique** : représentée par une **variable**, donc d'autres objets.

**Comportement** : un ensemble d'instructions, donc une **fonction**.

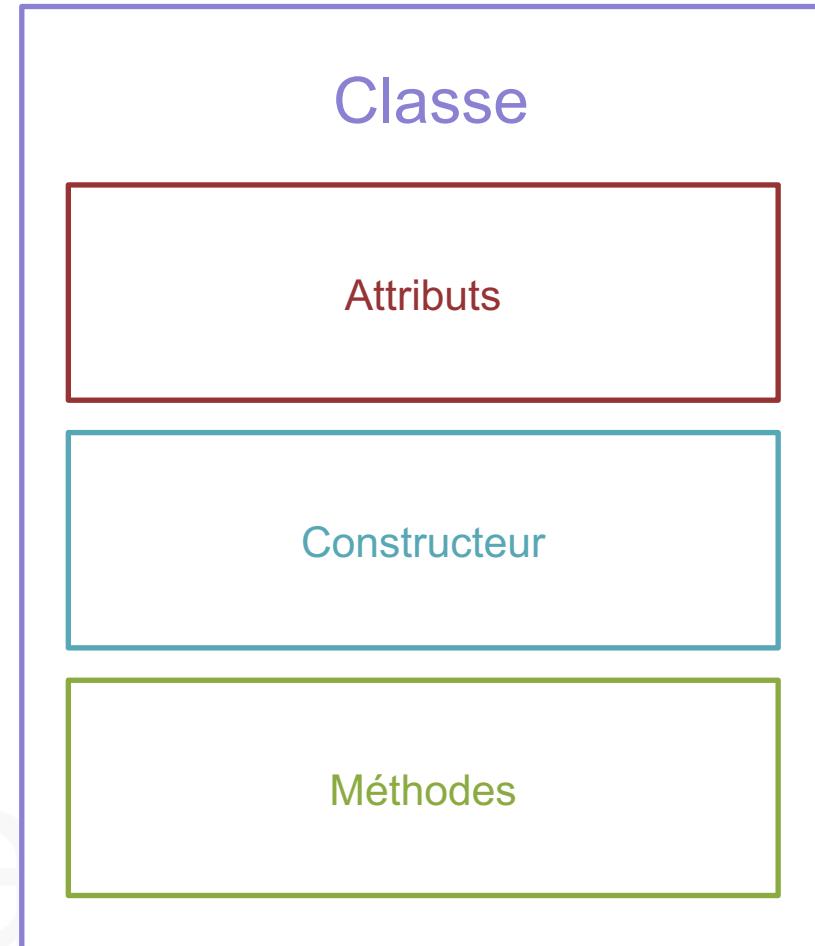
# Les classes

- ▷ Une classe permet de définir le *modèle* de l'objet.
  - › Elle permet de créer un nouveau **type d'objet**.
- ▷ Elle répertorie les caractéristiques des objets ainsi que leurs comportements.
- ▷ Un objet est appelé **instance** de classe. On peut créer plusieurs instances de classe.
- ▷ Une classe **ne contient pas** de données.

# Les classes



# Les classes



# Les attributs

- ▷ Les attributs décrivent les **caractéristiques**.
- ▷ Exemples :
  - › un nom,
  - › un âge,
  - › un parent,
  - › une date,
  - › une couleur,
  - › une organisation,
  - › etc.



macademia

# Les méthodes

- ▷ Les méthodes décrivent le **comportement**.
- ▷ Exemples :
  - › manger,
  - › dormir,
  - › écrire,
  - › se décrire,
  - › changer de nom,
  - › donner l'heure,
  - › etc.



Macademia

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Programmation objet avec Python

# Les classes - syntaxe

- ▷ Pour créer une classe, on utilise le mot-clé **class**.
  - ▷ On indique ensuite le nom de la classe.
    - › Il doit être le plus explicite possible.
  - ▷ Il est conseillé d'écrire une classe dans un fichier Python dédié, qui porte le même nom que la classe qu'il contient.
- ⓘ La convention PEP 8 préconise la CamelCase en majuscule.**

# Les classes - syntaxe

classes.py

```
class NomDeLaClasse:  
    instruction 1  
    instruction 2  
    ...  
    instruction n
```

# Les attributs

- ▷ Une classe définit des **attributs**, qui permettent de stocker les informations des objets.
- ▷ Toutes les instances d'une même classe possèdent **les mêmes attributs**, mais **leurs valeurs diffèrent** selon les instances.
  - › Pour rappel, chaque instance est unique.
- ▷ Lorsque l'on crée un objet, on renseigne **les valeurs** de ses attributs.
- ▷ On accède à un attribut avec le caractère « **.** » (point).

# Le constructeur

- ▷ Le constructeur est une fonction particulière qui est appelée **à la création d'un objet**.
- ▷ En Python, les constructeurs se notent **`__init__()`**.
- ▷ Ils prennent au minimum un paramètre : **self**.
- ▷ C'est dans le constructeur qu'on déclare les attributs d'une classe.

macademia

# Le constructeur et les attributs

```
classes.py

class Chien:
    """ DOCSTRING : Cette classe représente un chien """

    # Le constructeur
    def __init__(self, nom, age, couleur, sexe):
        # Définition des attributs
        self.nom = nom
        self.age = age
        self.couleur = couleur
        self.sexé = sexe
```

# Création d'une instance et accès aux attributs

classes.py

```
mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")
mon_deuxieme_chien = Chien("Laïka", 3, "Gris", "Femelle")

print(mon_chien.nom)
print(mon_deuxieme_chien.couleur)

mon_chien.age += 1
print(mon_chien.nom, "a", mon_chien.age, "ans")
```

Snoopy

Gris

Snoopy a 8 ans

# self

- ▷ Imaginons le cas de figure suivant ...

```
def __init__(self, nom, age, couleur, sexe):  
    nom = nom  
    age = age  
    couleur = couleur  
    sexe = sexe
```

Que va-t-il se passer ?

# self

- ▷ Le mot-clé **self** permet de faire une **autoréférence**.
- ▷ Il permet d'expliciter que l'expression utilisée (nom de variable ou de méthode) appartient bien à la classe elle-même.
- ▷ On utilisera **toujours self** pour y faire référence.
- ▷ Comme expliqué précédemment, **self** est obligatoirement le premier paramètres de tous les constructeurs et méthodes.

# Les méthodes

- ▷ Une méthode est une **fonction** associée à une classe.
- ▷ Une méthode permet de modéliser un comportement, une action, une description, etc. liée à cette classe.
- ▷ Tout comme un constructeur, une méthode prend au moins un paramètre, **self**.
- ▷ Comme pour les attributs, on accède à une méthode avec le caractère « `.` » (point).

# Les méthodes

```
classes.py

class Chien:

    def __init__(self, nom, age, couleur, sexe):
        self.nom = nom
        self.age = age
        self.couleur = couleur
        self.sex = sexe

    def manger(self, aliment):
        print("Je mange", aliment)

    def dormir(self, heures):
        print("Je dors", heures, "heures")
```

# Appel de méthode

```
classes.py

mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")

mon_chien.manger("des croquettes")
mon_chien.dormir(6)

# Autre syntaxe
# Déconseillé, c'est juste pour comprendre le mécanisme
Chien.manger(mon_chien, "de la pâtée")
```

```
Je mange des croquettes
Je dors 6 heures
Je mange de la pâtée
```

# Encapsulation et propriétés

- ▷ Il n'y a pas de notion d'attributs publics ou privés en Python.
  - › Tous les attributs sont *publics*.
- ▷ Cela peut poser des problème des sécurité des données.
- ▷ La gestion de l'accès et la modification des attributs se fait grâce à des **propriétés**.
- ▷ Par convention, les attributs dont on souhaite restreindre l'accès / la modification commencent par un « `_` » (simple/double underscore).

# Encapsulation et propriétés - la fonction *property*

- ▷ On utilise la fonction **property()** dans la classe pour définir des propriétés à un attribut.
- ▷ On affecte cette fonction à l'attribut, **cette fois sans le « \_ »**.
- ▷ La fonction **property()** prend **4 paramètres**, tous optionnels :
  - › **get** : le nom de la fonction appelée lors de l'accès.
  - › **set** : le nom de la fonction appelée lors de la **modification**.
  - › **del** : le nom de la fonction appelée lors de la **suppression**.
  - › **help** : le nom de la fonction appelée lors de la demande d'aide.

# Encapsulation et propriétés - la fonction *property*

Appel avec 0 paramètre :

**attr = property()**

*L'attribut sera ni consultable, ni modifiable.*

macademia

# Encapsulation et propriétés - la fonction *property*

Appel avec 1 paramètre :

**attr = property(**get**)**

L'attribut sera **consultable**, mais pas modifiable.

macademia

# Encapsulation et propriétés - la fonction *property*

Appel avec 2 paramètres :

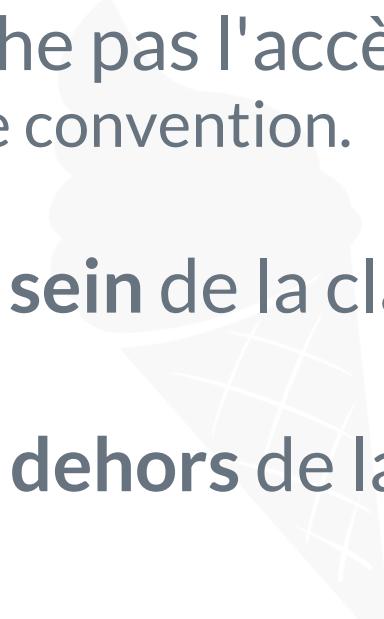
```
attr = property(get, set)
```

L'attribut sera *consultable* et *modifiable*.

Macademia

# Encapsulation et propriétés - la fonction *property*

- ▷ Le symbole « \_ » n'empêche pas l'accès à l'attribut.
  - › Il s'agit uniquement d'une convention.
- ▷ On accède à l'attribut **au sein de la classe avec le « \_ ».**
- ▷ On accède à l'attribut **en dehors de la classe sans le « \_ ».**

A large, semi-transparent watermark of the word "macadem" in a lowercase, sans-serif font, with a small "ia" at the end. It is positioned vertically in the center of the slide.

macadem

# Encapsulation et propriétés

proprietes.py

```
class Chien:

    def __init__(self, nom, age, couleur, sexe):
        self._nom = nom
        self.age = age
        self.couleur = couleur
        self.sex = sexe

    def _get_nom(self):
        return self._nom

    def _set_nom(self, nom):
        print(self._nom, "s'appellera désormais", nom)
        self._nom = nom

    nom = property(_get_nom, _set_nom)
```

# Encapsulation et propriétés

```
proprietes.py

mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")

print(mon_chien.nom)

mon_chien.age += 1
mon_chien.nom = "Snoop Dogg"

print(mon_chien.nom)
```

```
Snoopy
Snoopy s'appellera désormais Snoop Dogg
Snoop Dogg
```

# Les attributs de classe

- ▷ Nous avons vu qu'un attribut est associé à un objet.
- ▷ Il est cependant possible d'associer un attribut à une classe.
  - › Cet attribut sera partagé par toutes les instances de la classe.
- ▷ Pour cela, il faut déclarer l'attribut **avant le constructeur**.
- ▷ Lorsque l'on souhaite **accéder à cet attribut**, on n'utilisera pas le mot-clé **self** ou le nom de l'objet, mais **le nom de la classe**.

# Les attributs de classe

classes.py

```
class Chien:

    population = 0

    def __init__(self, nom, age, couleur, sexe):
        self._nom = nom
        self.age = age
        self.couleur = couleur
        self.sex = sexe
        Chien.population += 1
```

# Les attributs de classe

```
classes.py

mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")
mon_deuxieme_chien = Chien("Laïka", 3, "Gris", "Femelle")
mon_troisieme_chien = Chien("Bob", 2, "Noir", "Mâle")

print(Chien.population)
```

3

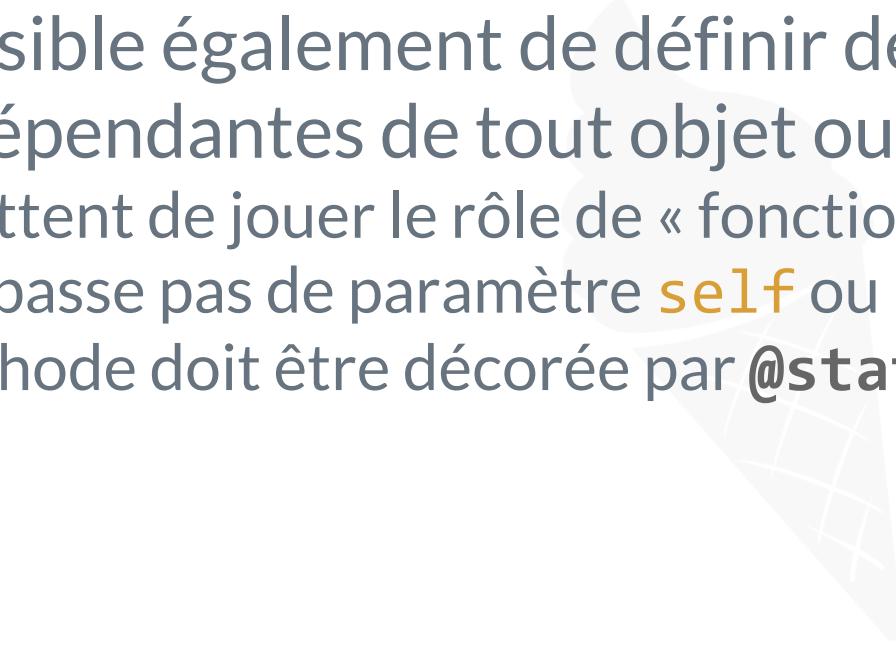
# Méthodes de classe

- ▷ Tout comme les attributs, il est possible de définir des **méthodes de classe**, qui seront associées à la classe, et non pas à ses instances.
  - › On ne passera pas `self` en premier paramètre, mais `cls`.
  - › La méthode doit être décorée par `@classmethod`.

macademia

# Méthodes statiques

- ▷ Il est possible également de définir des méthodes statiques, qui seront indépendantes de tout objet ou toute classe.
  - › Permettent de jouer le rôle de « fonctions utilitaires ».
  - › On ne passe pas de paramètre `self` ou `cls`.
  - › La méthode doit être décorée par `@staticmethod`.

A large, semi-transparent watermark of the word "macademia" in a bold, lowercase, sans-serif font, oriented vertically and centered on the slide.

macademia

# Méthodes de classes et statiques

```
classes.py

class MaClasse:

    def methode_normale(self):
        print("L'instance est :", self)

    @classmethod
    def methode_de_classe(cls):
        print("La classe est", cls)

    @staticmethod
    def methode_statique():
        print("Aucune référence à une instance ou une classe")
```

# Méthodes de classes et statiques

classes.py

```
mc = MaClasse()  
mc.methode_normale()  
  
MaClasse.methode_de_classe()  
  
MaClasse.methode_statique()
```

```
L'instance est : <__main__.MaClasse object at 0x009D05D0>  
La classe est <class '__main__.MaClasse'>  
Aucune référence à une instance ou une classe
```

# Introspection - outils pratiques

- ▷ Il existe un attribut spécial nommé **`__dict__`**, qui permet de lister les attributs d'un objet et ses valeurs **sous forme de dictionnaire**.
- ▷ Il existe également une fonction **`dir()`**, qui permet d'afficher tous les attributs et toutes les méthodes spéciales d'un objet passé en paramètre.
  - › Cette fonction affiche tous les attributs et toutes les méthodes, y compris les **méthodes spéciales**.

Macademia

# Introspection - outils pratiques

```
introspection.py

mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")

print(mon_chien.__dict__)
```

```
{'nom' : 'Snoopy', 'age' : 7, 'couleur' : 'Blanc', 'sexe' : 'Mâle'}
```

# Introspection - outils pratiques

```
introspection.py

mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")

print(dir(mon_chien))
```

```
['__Chien_nom__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_get_nom', '_set_nom', 'age', 'couleur', 'dormir', 'manger', 'nom',
 'population', 'sexe']
```

# Les méthodes spéciales

- ▷ Chaque classe possède un ensemble de méthodes spéciales définies dans le langage Python.
- ▷ Les méthodes spéciales s'écrivent comme ceci : **`__methode__()`**.
- ▷ Permettent de personnaliser certains comportements de la classe.
- ▷ La plus connue est la méthode **`__str__()`**, qui permet de convertir un objet en chaîne de caractères.
  - › Pour afficher ses informations dans un **`print()`** par exemple.

# La méthode `__str__()`

```
methodes_speciales.py

class Chien:

    def __init__(self, nom, age, couleur, sexe):
        self.nom = nom
        self.age = age
        self.couleur = couleur
        self.sexu = sexe

    def __str__(self):
        return f"{self.nom} est un(e) {self.sexu} de {self.age} ans de couleur {self.couleur}"
```

# La méthode `__str__()`

methodes\_speciales.py

```
mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle")  
  
print(mon_chien)
```

```
Snoopy est un(e) Mâle de 7 ans de couleur Blanc
```

# Le destructeur

- ▷ La méthode spéciale `__del__()` permet de définir le comportement d'un objet lorsque celui-ci est détruit.
  - › Lors de l'appel à l'opération `del`
- ▷ Cette méthode **ne supprime pas l'objet**, mais est appelée à la suppression de celui-ci.
- ▷ Elle est appelée sur tous les objets à la fin du programme s'ils n'ont pas été détruits avant.

A large, semi-transparent watermark of the word "macadem" in a stylized, lowercase font, with a small "ia" at the end. The letters have a slight shadow effect.

# Comparer des objets

- ▷ *objet.\_\_lt\_\_(self, autre)*
  - › Surcharge de l'opérateur <
- ▷ *objet.\_\_le\_\_(self, autre)*
  - › Surcharge de l'opérateur <=
- ▷ *objet.\_\_eq\_\_(self, autre)*
  - › Surcharge de l'opérateur ==
- ▷ *objet.\_\_ne\_\_(self, autre)*
  - › Surcharge de l'opérateur !=
- ▷ *objet.\_\_gt\_\_(self, autre)*
  - › Surcharge de l'opérateur >
- ▷ *objet.\_\_ge\_\_(self, autre)*
  - › Surcharge de l'opérateur >=



- ▷ Consulter la doc pour la liste complète :
  - › <https://docs.python.org/fr/3/reference/datamodel.html#special-method-names>

# En résumé

**Dans ce chapitre, nous avons vu ...**



# L'héritage

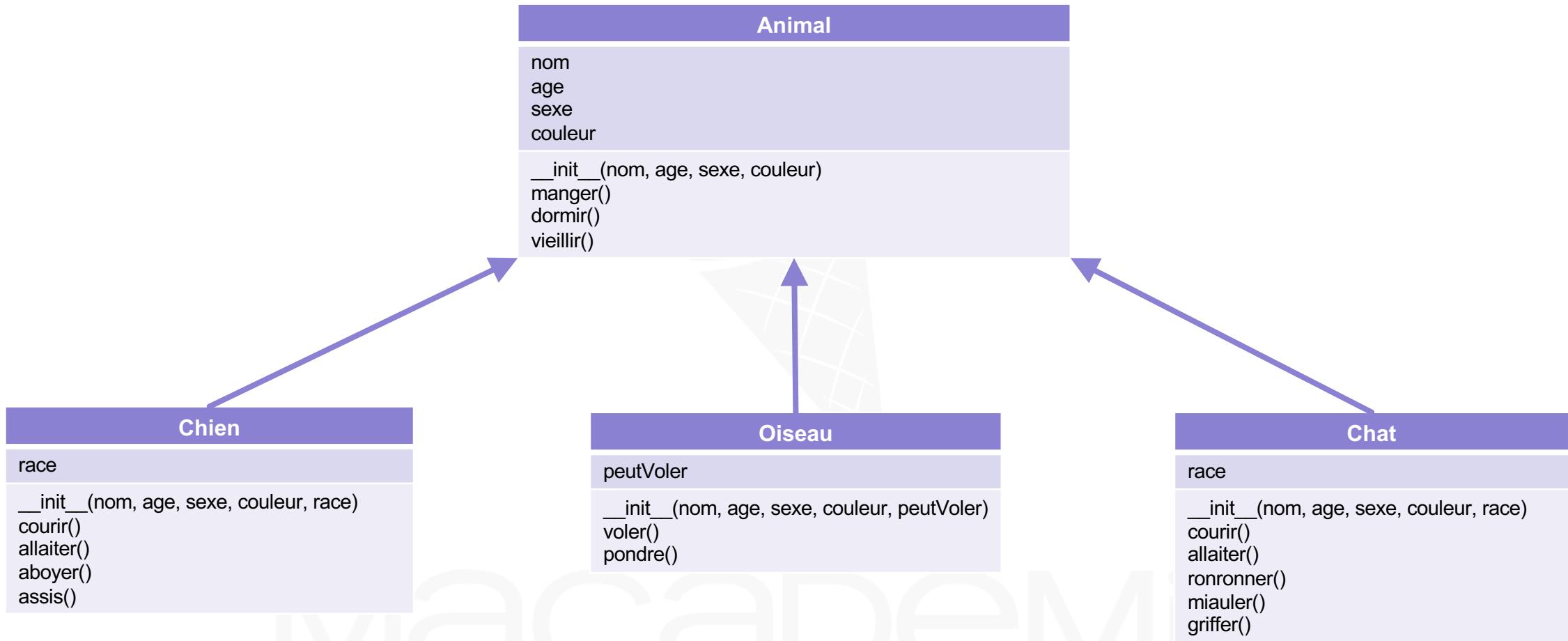
# L'héritage

- ▷ Une des notions les plus importantes de la programmation objet.
- ▷ Permet de créer des classes héritées (appelées aussi dérivées, filles) à partir d'une classe de base (appelée aussi superclasse, mère).
- ▷ La classe fille « hérite » **systématiquement** de tous les attributs, constructeur et méthodes de la classe mère.
- ▷ Une classe fille peut avoir ses propres attributs et méthodes, et peut **redéfinir** les méthodes de la classe mère.

# Héritage simple

- ▷ On parle d'héritage simple lorsqu'une classe n'hérite que d'une **seule classe**.
- ▷ On note le nom de la classe mère **entre parenthèses** après le nom de la classe fille.
- ▷ Si la classe mère possède des attributs, il faudra **appeler le constructeur de la classe mère dans celui de la classe fille**.
- ▷ Les méthodes redéfinies ont **la priorité sur celles de la classe mère**.

# Héritage simple



# Héritage simple

```
heritage.py
```

```
class Animal:  
    """ La classe mère """  
  
    def __init__(self, nom, age, couleur, sexe):  
        self.nom = nom  
        self.age = age  
        self.couleur = couleur  
        self.sexu = sexe  
  
    def manger(self, aliment):  
        print("Je mange", aliment)
```

# Héritage simple

heritage.py



```
class Chien(Animal):
    """ La classe fille """

    def __init__(self, nom, age, couleur, sexe, race):
        Animal.__init__(self, nom, age, couleur, sexe)
        self.race = race

    def aboyer(self):
        print("Ouaf ! Ouaf !")
```

# Héritage simple

heritage.py



```
class Chien(Animal):
    """ La classe fille """

    def __init__(self, nom, age, couleur, sexe, race):
        super().__init__(nom, age, couleur, sexe)
        self.race = race

    def aboyer(self):
        print("Ouaf ! Ouaf !")
```

# Héritage simple

```
heritage.py

mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle", "Beagle")

print(mon_chien.nom)                      # Attribut de Animal
print(mon_chien.race)

mon_chien.aboyer()
mon_chien.manger("des croquettes")      # Méthode de Animal
```

```
Snoopy
Beagle
Ouaf ! Ouaf !
Je mange des croquettes
```

# Redéfinition de méthode

```
heritage.py

class Chien(Animal):
    """ La classe fille """

    def __init__(self, nom, age, couleur, sexe, race):
        Animal.__init__(self, nom, age, couleur, sexe)
        self.race = race

    def aboyer(self):
        print("Ouaf ! Ouaf !")

    def manger(self, aliment):
        print("Je suis un chien et je mange", aliment)
```

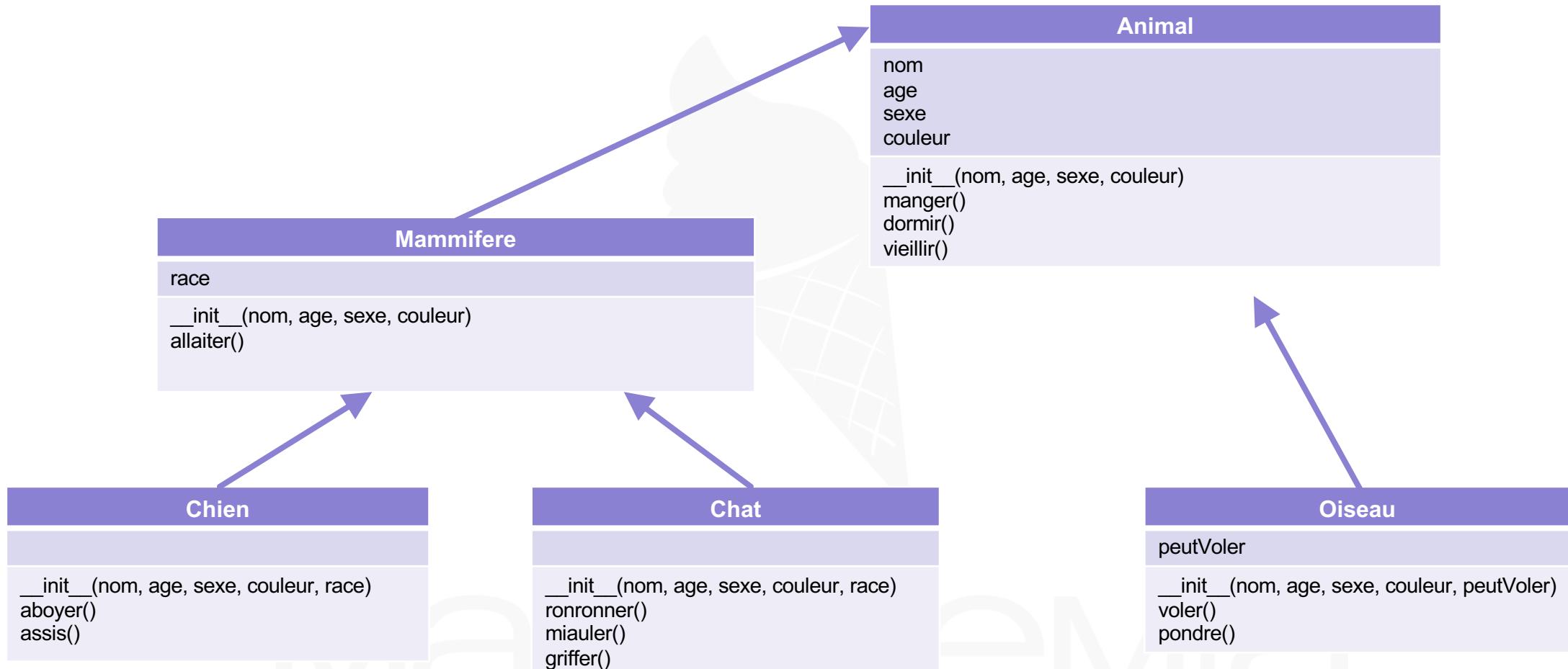
# Héritage simple

heritage.py

```
mon_chien = Chien("Snoopy", 7, "Blanc", "Mâle", "Beagle")  
  
# Méthode de Animal redéfinie dans Chien  
mon_chien.manger("des croquettes")
```

Je suis un chien et je mange des croquettes

# Héritage simple

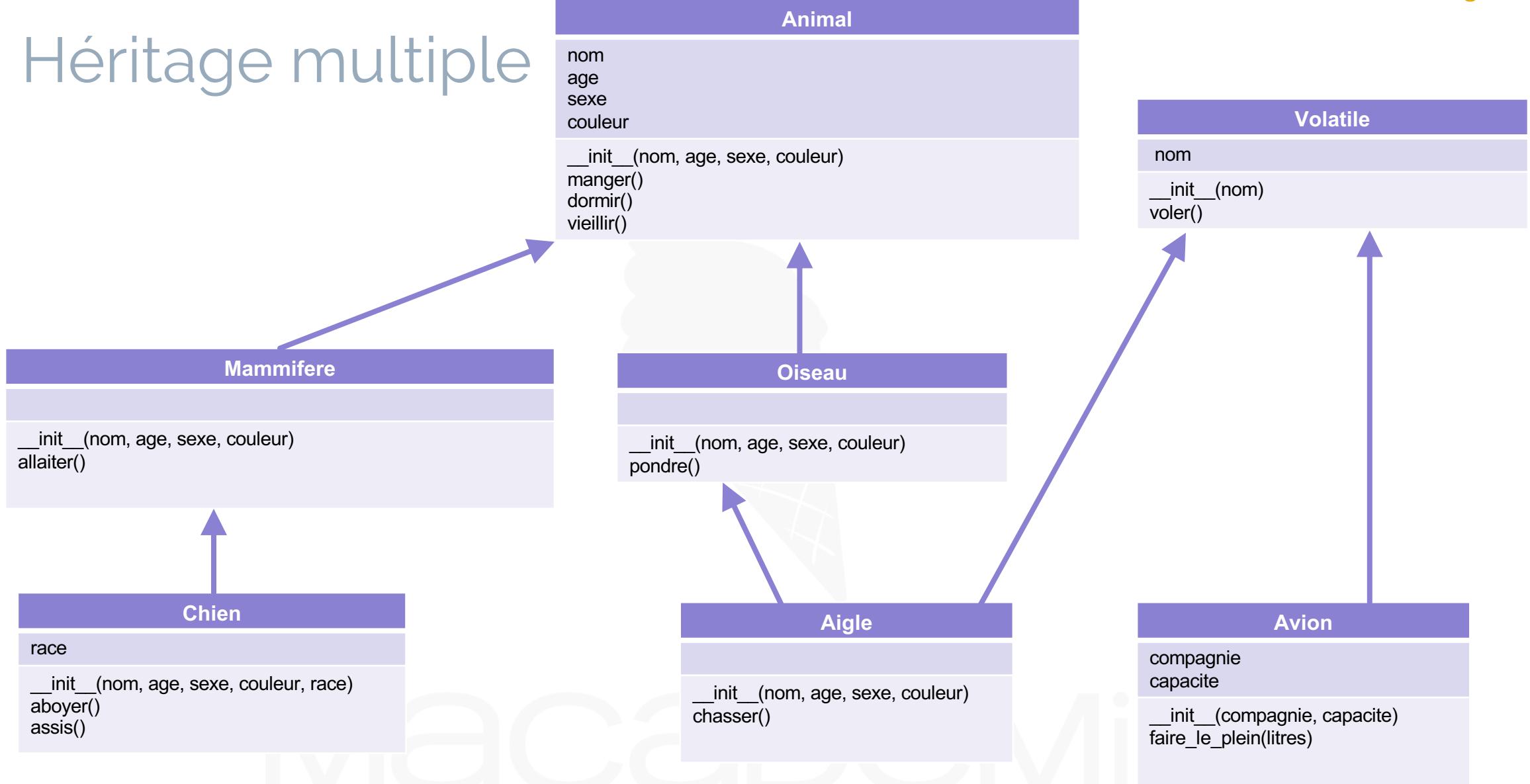


# Héritage multiple

- ▷ En Python, il est possible de mettre en place l'héritage multiple.
- ▷ On sépare les classes mères avec des **virgules**.
- ▷ Pratique pour les classes partageant des propriétés de plusieurs classes.
- ▷ L'ordre des classes mères a une importance.
  - › Si deux méthodes portent le même nom dans deux classes mères, c'est celle de la classe mère appelée en premier qui a la *priorité*.

# Héritage multiple

L'héritage



# Héritage multiple

```
heritage.py
```

```
class Aigle(Oiseau, Volatile):

    def __init__(self, nom, age, couleur, sexe):
        Oiseau.__init__(self, nom, age, couleur, sexe)
        Volatile.__init__(self, nom)

    def chasser(self):
        print(self.nom, "est à la recherche d'une proie !")
```

# Héritage multiple

```
heritage.py

mon_aigle = Aigle("Mèfi", 4, "Brun", "Femelle")

mon_aigle.chasser()
mon_aigle.pondre()      # Méthode de Oiseau
mon_aigle.voler()       # Méthode de Volatile
mon_aigle.dormir()      # Méthode de Animal
```

```
Mèfi est à la recherche d'une proie !
Cet oeuf fera une bonne omelette
I believe I can flyyyyy
zzzzzzzzzzzz
```

# Polymorphisme

- ▷ Derrière ce nom barbare se cache une notion très simple.
- ▷ Dans le cas où nous avons des objets de types différents mais qui héritent d'une même classe, alors on peut appeler une méthode de la classe mère, même si celle-ci a été redéfinie.
- ▷ Encore mieux, en Python, le polymorphisme s'applique, tant que **les types sont compatibles** et que les méthodes **ont la même signature**, même si les objets n'héritent pas d'une même classe.

# Polymorphisme

```
polymorphisme.py

class A:
    def ecrire(self):
        print("Bla bla bla")

class B:
    def ecrire(self):
        print("Tototototo")
```

# Polymorphisme

polymorphisme.py

```
a = A()
b = B()

liste = [a, b]

for x in liste:
    x.ecrire()
```

```
Bla bla bla
Tototototo
```

# Abstraction en Python

- ▷ Le point précédent sur le polymorphisme souligne le fait que le principe de classe abstraite en Python n'est pas nécessaire.
  - › Il n'existe donc pas.
- ▷ De même, il n'y a pas de notion d'interface en Python.
  - › À la place, nous avons l'héritage multiple et le polymorphisme.

macademia

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Les exceptions

# Le bloc try / except

- ▷ On exécute les instructions susceptibles de lever une exception dans le bloc **try**.
- ▷ En cas d'erreur ce sont les instructions du bloc **except** correspondant à l'exception levée qui seront exécutées.



Macademia

# Un exemple de bloc try / except

```
exceptions.py

denominateur = input("Entrez un dénominateur")

try:
    fraction = 1 / int(denominateur)
    print("Résultat = ", fraction)
except:
    print("Division par 0 !")
```

# Capturer une exception précise

- ▷ Il est possible de gérer **plusieurs exceptions** dans un même bloc `try`.
- ▷ Pour cela, on précisera autant de `except` que d'exceptions à gérer, en indiquant le **type de l'exception** potentiellement levée.

A large, semi-transparent watermark of the word "macadem" in a lowercase, sans-serif font, with a smaller "ia" at the end, centered on the slide.

macadem

# Capturer une exception précise

```
exceptions.py

denominateur = input("Entrez un dénominateur")

try:
    fraction = 1 / int(denominateur)
    print("Résultat = ", fraction)
except ZeroDivisionError:
    print("Division par 0 !")
except ValueError:
    print("La valeur entrée n'est pas un nombre")
```

# Le mot-clé *finally*

- ▷ Si un morceau de code doit être exécuté, qu'une exception se produise ou non, on le place dans un bloc **finally**, après le **except**.
- ▷ Le bloc **finally** est toujours exécuté, même si les blocs **try** ou **except** contiennent un **return**, le bloc **finally** sera exécuté avant.

A faint watermark of the Macademia logo, featuring a stylized graduation cap and a book, is visible in the background.

macademia

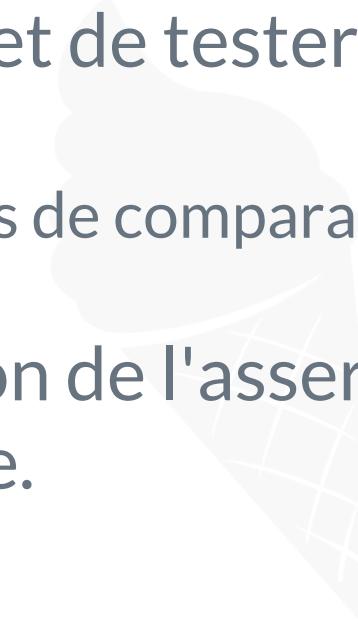
# Un exemple avec *finally*

```
exceptions.py

def fonction():
    try:
        print("On teste un truc")
        return True
    except:
        print("Oulala une erreur !")
        return False
    finally:
        print("Toujours exécuté")
```

# Les assertions

- ▷ Le mot-clé **assert** permet de tester la valeur d'une variable dans un bloc **try**.
  - › Utilisation des opérateurs de comparaison.
- ▷ Dans le cas où la condition de l'assertion n'est pas vérifiée, une **AssertionError** est levée.



# Les assertions

```
exceptions.py

annee = input("Quelle est votre année de naissance ?")

try:
    annee = int(annee)
    assert 1900 <= annee <= 2020
except AssertionError:
    print("Année de naissance incohérente")
except ValueError:
    print("Veuillez entrer un nombre")
```

# Lever une exception

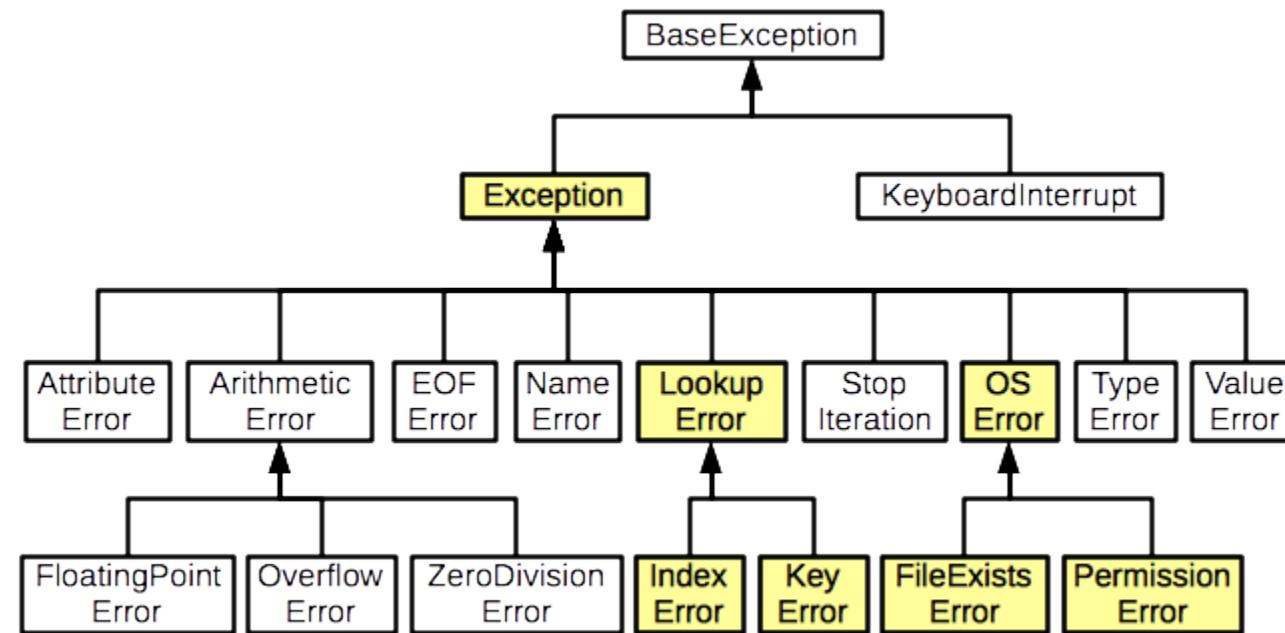
- ▷ Le mot-clé **raise** permet de lever une exception.
- ▷ Son utilisation devient intéressante à partir du moment où l'on crée nos propres classes d'exception.



Macademia

# Créer ses propres exceptions

- ▷ En Python, une exception est une instance d'une classe héritée de **BaseException**.



# Créer ses propres exceptions

- ▷ Afin de créer une classe d'exception, il faut donc **hériter** d'une classe d'exception existante.
  - › Généralement, on hérite de la classe **Exception**.
- ▷ L'exception devra au minimum afficher un message d'erreur.
- ▷ Il est conseillé de nommer sa classe avec le suffixe « **Exception** » ou « **Error** ».

macademia

# La classe d'exception personnalisée

exceptions.py

```
class AnneeNaissanceError(Exception):

    def __init__(self, annee):
        self.annee = annee

    def __str__():
        return f"{self.annee} n'est pas une année cohérente"
```

# Utilisation de *raise*

```
exceptions.py

class Personne:

    def __init__(self, nom, annee):
        if not 1900 <= annee <= 2020:
            raise AnneeNaissanceError(annee)
        self.nom = nom
        self.annee = annee

    def __str__(self):
        return + f"{self.nom} - {self.annee}"
```

# Lever une exception personnalisée

exceptions.py

```
p = Personne("John", 1875)
print(p)
```

```
Traceback (most recent call last):
  File "D:/Workspace/python/poo/exceptions.py", line 23, in <module>
    p = Personne("John", 1875)
  File "D:/Workspace/python/poo/exceptions.py", line 14, in __init__
    raise AnneeNaissanceError(annee)
__main__.AnneeNaissanceError: 1875 n'est pas une année cohérente
```

# Lever une exception personnalisée

exceptions.py

```
try:  
    pers = Personne("John", 1875)  
    print(pers)  
except AnneeNaissanceError as e:  
    print(e)
```

1875 n'est pas une année cohérente

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Les modules

# Les principes

- ▷ Python possède quelques fonctions *builtins*, c'est-à-dire qu'on peut les utiliser sans aucun prérequis.
  - › Ces fonctions sont assez limitées.
- ▷ Un module en Python représente **un fichier contenant un ensemble de fonctions**.
  - › Le nom du fichier est le nom du module.
- ▷ Il existe des modules prêts à l'emploi dans Python (StdLib).
- ▷ Il est possible de créer ses propres modules.

# Un module

```
fonctions.py

def ma_fonction():
    print("Ceci est ma fonction")

def mon_autre_fonction():
    return True

def ma_super_fonction(a, b):
    return a + b
```

# Import du module

quelconque.py



```
import fonctions

def toto():
    print("Je suis une fonction dans quelconque.py")

toto()
fonctions.ma_fonction()
```

Je suis une fonction dans quelconque.py  
Ceci est ma fonction

# Import du module

```
quelconque.py

import fonctions as fct

def toto():
    print("Je suis une fonction dans quelconque.py")
```

```
toto()
fct.ma_fonction()
```

```
Je suis une fonction dans quelconque.py
Ceci est ma fonction
```

# Import du module

quelconque.py



```
from fonctions import ma_fonction

def toto():
    print("Je suis une fonction dans quelconque.py")
```

```
toto()
ma_fonction()
```

```
Je suis une fonction dans quelconque.py
Ceci est ma fonction
```

# Import du module

quelconque.py



```
from fonctions import *

def toto():
    print("Je suis une fonction dans quelconque.py")

toto()
ma_fonction()
```

Je suis une fonction dans quelconque.py  
Ceci est ma fonction

# Import de package

- ▷ Un package n'est rien de plus qu'un **dossier** sur le disque.
- ▷ Un package doit contenir au minimum un fichier **`_init_.py`**
- ▷ Il est possible de n'importer qu'un seul fichier dans le package, grâce au caractère « `.` » (point).
  - › `from nom_package.nom_module import *`
- ▷ Il est possible d'importer plusieurs fichiers en même temps, regroupés dans un **package**.
  - › Pour cela, il faut créer une variable **`_all_`** dans le **`_init_.py`**

# Le fichier `__init__.py`

- ▷ Le code présent dans le fichier `__init__.py` est exécuté lors de l'import d'un package.

Exemple de fichier `__init__.py` :

```
__all__ = ["mon_module", "mon_autre_module", "mon_super_module"]

print("Le package 'mon_package' est importé")
```

# Import de package

```
# Importer toutes les fonctions de "mon_module"
from mon_package.mon_module import *

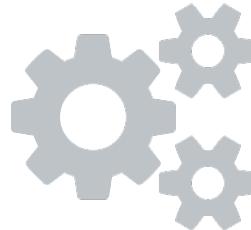
# Importer la fonction "ma_fonction()" de "mon_mondule"
from mon_package.mon_module import ma_fonction

# Importer mon_module / usage : mon_mondule.ma_fonction()
from mon_package import mon_module

# Importer tous les modules de mon_package (dans __init__.py)
from mon_package import *
```

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Modules de la librairie standard

# La librairie standard

- ▷ La **librairie** (ou *bibliothèque*) **standard** désigne l'ensemble des modules prêts à l'emploi.
  - › Ils sont fournis avec l'installation de Python.
- ▷ Il est conseillé de toujours utiliser la syntaxe « **import Lib** ».
- ▷ Il existe plus de 200 modules dans la StdLib.
- ▷ Les modules de la StdLib évoluent peu.

Documentation : <https://docs.python.org/fr/3/library/index.html>

# Arguments passés sur la ligne de commande

- ▷ Comme dans tout langage de script, il est possible de passer des valeurs en **argument** de la ligne de commande lors du lancement du programme.
- ▷ Les arguments sont récupérés dans une liste **sys.argv**, où l'argument n°0 est toujours le nom du programme.
- ▷ Il faut donc importer le module **sys**.

Macademia

# Arguments passés sur la ligne de commande

```
module_sys.py

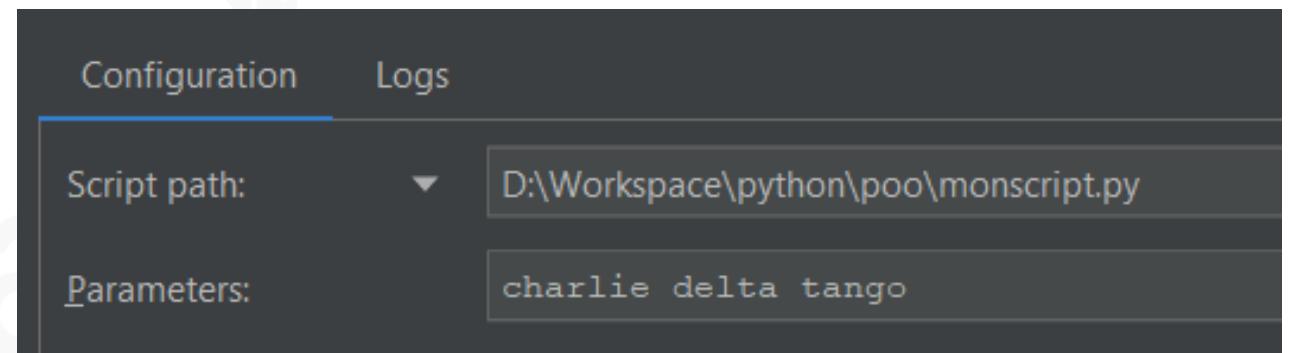
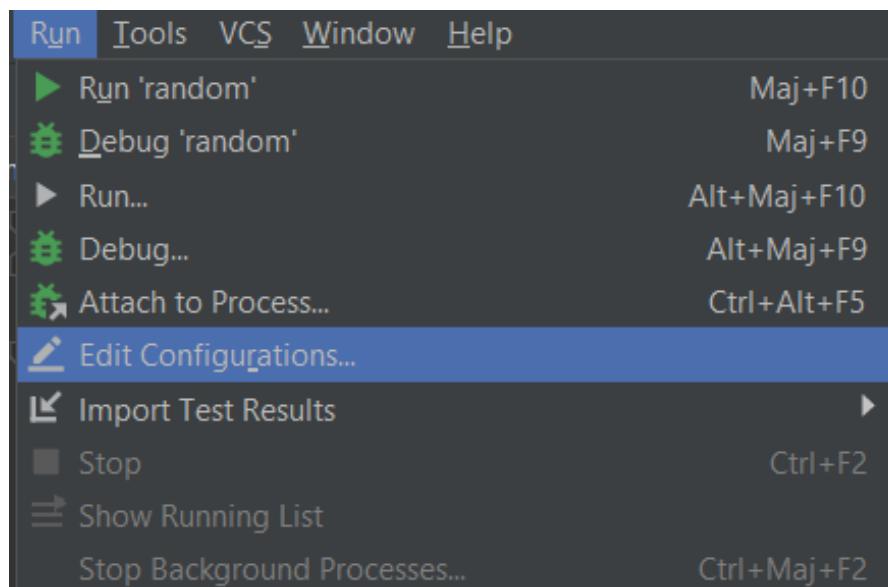
import sys

print("Nom du programme :", sys.argv[0])

for arg in sys.argv[1:]:
    print(arg)
```

# Passer un argument au programme sur PyCharm

- ▷ Dans **Run > Edit Configurations**, on indique les paramètres dans « Parameters ».



# Module *math* → opérations mathématiques

- ▷ Le module **math** contient quelques opérations mathématiques avancées.
- ▷ On retrouve des opérations comme la puissance ou la racine carrée, des opérations trigonométriques ou bien encore des opérations d'arrondi.



macademia

Documentation : <https://docs.python.org/fr/3/library/math.html>

# Module *math* → opérations mathématiques

- ▷ **math.pow(*nombre*, *puissance*)**
  - › Le nombre à la puissance indiquée.
  - › Même chose que l'opérateur \*\*, mais renvoie un float.
- ▷ **math.sqrt(*nombre*)**
  - › Racine carrée.
- ▷ **math.exp(*nombre*)**
  - › Exponentielle.
- ▷ **math.trunc(*nombre*)**
  - › Troncature.

# Module *math* → opérations mathématiques

module\_math.py



```
import math

print(math.pow(5, 3)) # 5 puissance 3
print(math.sqrt(144)) # Racine carrée de 144
print(math.exp(4)) # Exponentielle de 4
print(math.trunc(3.14159)) # Troncature de 3,14159
```

125.0

12.0

54.598150033144236

3

# Module *re* → expressions régulières

- ▷ Le module **re** permet de **chercher** des **expressions régulières** dans des chaînes de caractères.
- ▷ Cela permet, par exemple, de faire des vérifications sur le format attendu des chaînes de caractères.
- ▷ Il permet également de **remplacer** une chaîne de caractère par une autre selon une expression régulière.

Documentation : <https://docs.python.org/fr/3/library/re.html>

# Module *re* → expressions régulières

- ▷ **re.match(*expression*, *chaine*)**
  - › Recherche l'expression à partir du premier caractère de la chaîne.
- ▷ **re.search(*expression*, *chaine*)**
  - › Recherche l'expressions dans toute la chaîne.
- ▷ **re.sub(*expression*, *remplacement*, *chaine*)**
  - › Remplace l'expression par le texte de remplacement dans la chaîne.

# Module *re* → expressions régulières

module\_re.py



```
import re

print(re.search(r"thon", "Python"))
print(re.match(r"thon", "Python"))
print(re.search(r".*thon", "Sandwich au thon"))
print(re.search(r".*thon", "Sandwich au poulet"))

print(re.sub(r"thon", "fromage", "Sandwich au thon"))
```

```
<re.Match object; span=(2, 6), match='thon'>
None
<re.Match object; span=(0, 16), match='Sandwich au thon'>
None
Sandwich au fromage
```

# Gestion des fichiers

- ▷ La librairie standard permet de lire et d'écrire dans des fichiers.
- ▷ Il existe plusieurs modes d'accès.
- ▷ Pour des opérations avancées, voir le module **os**.
- ▷ Utilisation du bloc **with** pour gérer les erreurs et la fermeture du fichier à la fin de l'utilisation.

A large, semi-transparent watermark of the word "macadem" in a stylized, lowercase font, with a small "ia" at the end.

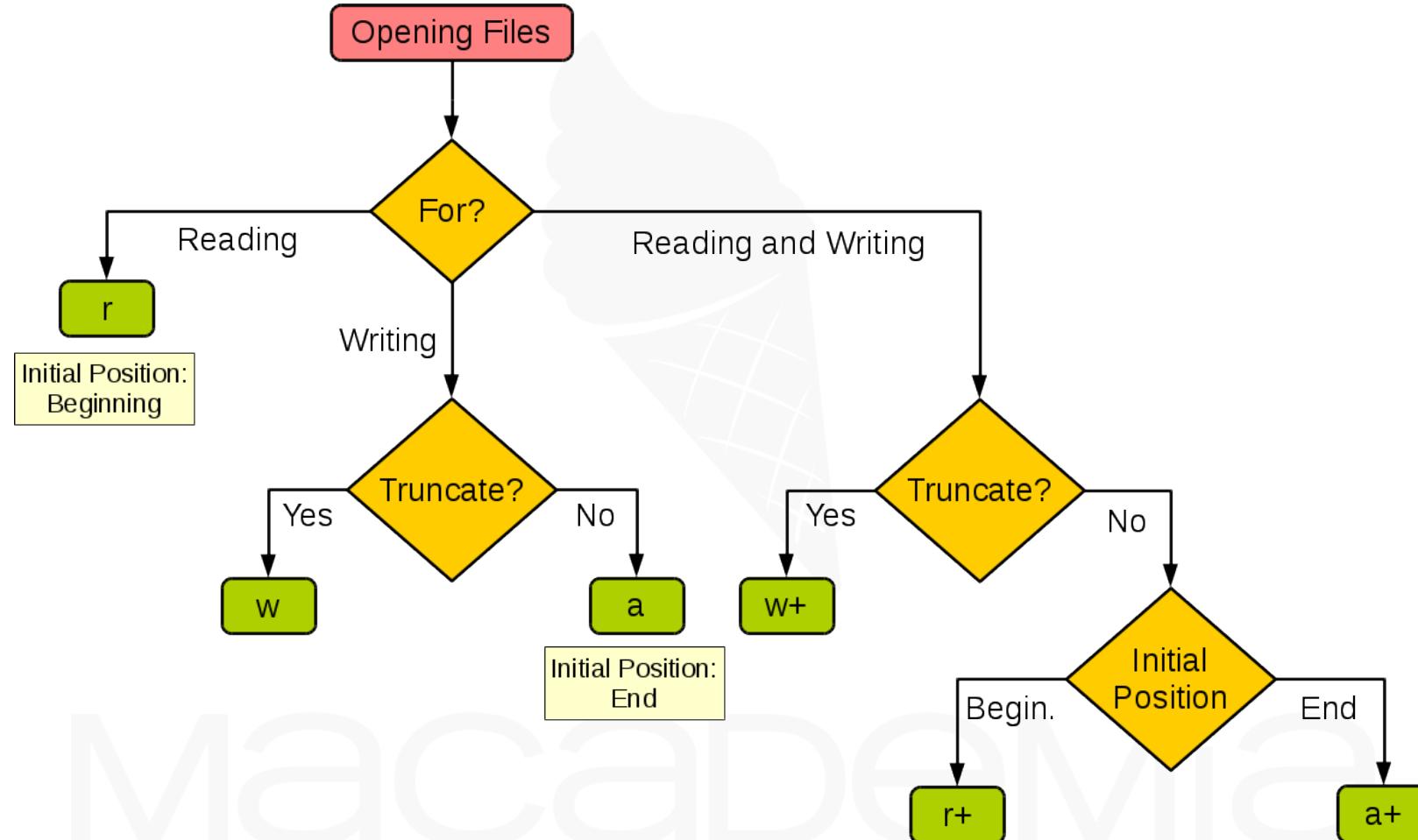
# Gestion des fichiers

- ▷ `with open(file, mode, encoding="utf8") as f:`
  - › Ouverture du fichier avec le mode indiqué.
  - › Retourne le fichier sous forme d'objet.
- ▷ `f.close()`
  - › Fermer le fichier. Inutile si on utilise `with`.
- ▷ `f.seek(position)`
  - › Déplacer le curseur à la position donnée.
- ▷ `f.tell()`
  - › Récupérer la position actuelle du curseur..

# Gestion des fichiers - les modes

Mode	Description
r	Lecture
r+	Lecture et écriture
w	Écriture : crée le fichier s'il n'existe pas, écrase le fichier lors de l'écriture
w+	Lecture et écriture : crée le fichier s'il n'existe pas, écrase le fichier lors de l'écriture
a	Écriture : crée le fichier s'il n'existe pas, écrit à partir de la fin du fichier
a+	Lecture et écriture : crée le fichier s'il n'existe pas, écrit à partir de la fin du fichier

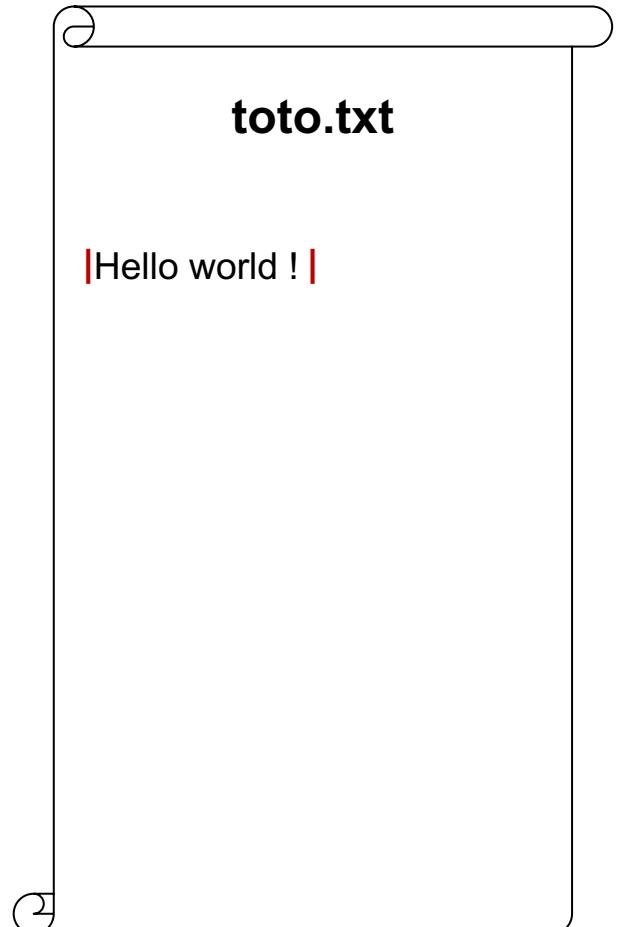
# Gestion des fichiers - les modes



# Gestions des fichiers

```
with open('toto.txt', 'w+') as f:  
    f.write("Hello world !")  
    f.seek(0)  
    print(f.read())
```

```
Hello world !
```



# Gestions des fichiers

```
with open('toto.txt', 'a') as f:  
    f.write("\n")  
    f.write("It's a beautiful day")
```



# Fonctions de lecture

Toutes les opérations de lecture se font à partir de la position du curseur.

- ▷ ***f.read()***
  - › Lire le fichier en entier.
- ▷ ***f.read(*caractères*)***
  - › Lire un nombre de caractères.
- ▷ ***f.readline()***
  - › Lire la prochaine ligne (ne retourne rien si fin de fichier).
- ▷ ***f.readlines()***
  - › Retourne une liste de toutes les lignes du fichier.

# Fonctions de lecture

Toutes les opérations d'écriture se font à partir de la position du curseur.

- ▷ ***f.write(chaine)***
  - › Écrire la chaîne de caractères.
  
- ▷ ***f.writelines(Liste)***
  - › Écrire la liste de lignes.



macademia

# Module os → services du système d'exploitation

- ▷ Le module **os** permet d'effectuer des opérations ou récupérer des informations sur le système.
- ▷ Il permet aussi de se déplacer dans l'arborescence.
- ▷ Gestion avancée de l'accès aux fichiers.
- ▷ Le module **os** contient un sous-module **os.path**, pour la gestion des chemins de fichiers.

Documentation : <https://docs.python.org/fr/3/library/os.html>

# Module os → services du système d'exploitation

## ▷ `os.listdir(chemindossier)`

- › Récupérer la liste des éléments d'un dossier.

## ▷ `os.walk(chemindossier)`

- › Récupérer récursivement tous les sous-éléments d'un dossier (itérable).

## ▷ `os.mkdir(chemindossier)`

- › Créer le dernier dossier indiqué dans un chemin (erreur si mauvais chemin).

## ▷ `os.makedirs(chemindossier)`

- › Créer récursivement tous les dossiers d'un chemin s'ils n'existent pas.

## ▷ `os.remove(chemin)`

- › Supprimer le fichier / dossier indiqué.

# Module os → services du système d'exploitation

- ▷ **os.rename(*ancien, nouveau*)**
  - › Renommer le fichier / dossier indiqué.
- ▷ **os.getlogin()**
  - › Récupérer le nom de l'utilisateur courant connecté sur la machine.
- ▷ **os.getenv(*variable*)**
  - › Renvoie la valeur d'une variable d'environnement.
- ▷ **os.system(*commande*)**
  - › Exécute une commande dans l'invite de commande (ou terminal) et renvoie le résultat.
- ▷ **os.kill(*pid, sig*)**
  - › Envoie le signal *sig* au processus *pid*.

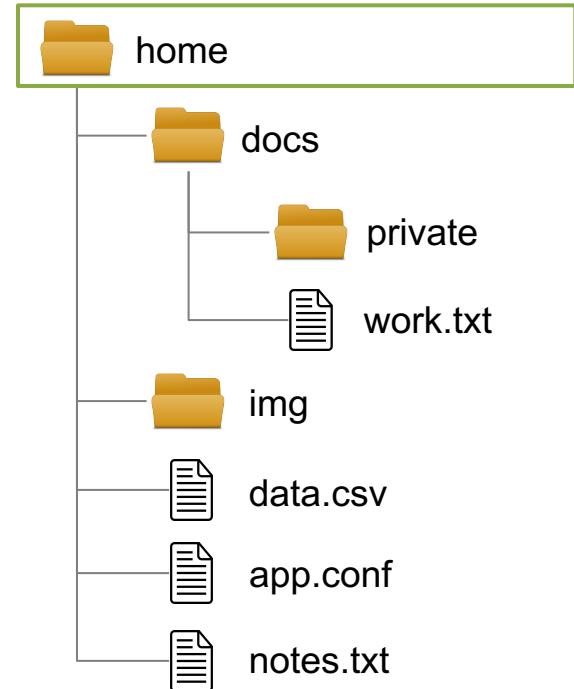
# Module os → services du système d'exploitation

- ▷ **os.path.exists(*chemin*)**
  - › Teste si le chemin indiqué existe.
- ▷ **os.path.isfile(*chemin*)**
  - › Indique si le chemin correspond à un fichier ou nom.
- ▷ **os.path.isdir(*chemin*)**
  - › Indique si le chemin correspond à un dossier ou nom.
- ▷ **os.path.getsize(*chemin*)**
  - › Renvoie la taille d'un fichier (en octets).
- ▷ **os.path.basename(*chemin*)**
  - › Renvoie le dernier élément du chemin.

# Module os → services du système d'exploitation

```
import os

os.listdir()
os.mkdir('docs/public/nouv_dossier')
os.makedirs('docs/public/nouv_dossier')
os.getlogin()
os.path.isfile('data.csv')
os.path.basename('docs/work.txt')
os.path.exists('docs/private/work.txt')
```



# Module os → gestion multi-OS ...

```
module_os.py

import os
import sys

def lister_repertoire():
    if sys.platform == "win32":
        # Système détecté : Windows
        print(os.system('dir'))
    elif sys.platform in ("linux", "darwin"):
        # Système détecté : Linux ou macOS
        print(os.system('ls -l'))
    else:
        print("Système d'exploitation non pris en charge")
```

# Module csv → traitement de fichiers CSV

- ▷ Le module **csv** permet de traiter ... des fichiers CSV (fichier de données séparées par une virgule).
- ▷ Fournit des fonctions pour lire et écrire des données tabulaires au format CSV.



macademia

Documentation : <https://docs.python.org/fr/3/library/csv.html>

# Module csv → traitement de fichiers CSV

- ▷ **csv.reader(*fichiercsv*, delimiter=',')**
  - › Lire le fichier CSV.
  
- ▷ **csv.writer(*fichiercsv*, delimiter=',')**
  - › Ecrire dans le fichier CSV.

A large, semi-transparent watermark of the word "macademia" in a bold, lowercase, sans-serif font.

# Module csv → traitement de fichiers CSV

```
module_csv.py

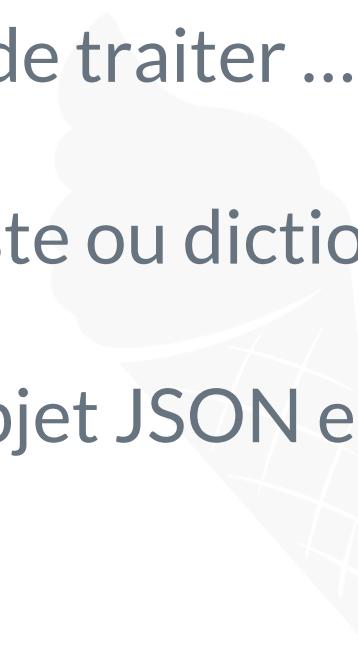
import csv

# Lecture
with open('hello.csv', 'r', encoding='UTF-8', newline='') as csvfile:
    lecture = csv.reader(csvfile, delimiter=',')
    for row in lecture :
        print(' '.join(row))

# Ecriture
with open('hello.csv', 'w', encoding='UTF-8', newline='') as csvfile:
    ecriture = csv.writer(csvfile, delimiter=',')
    ecriture.writerow(['Pomme', 'Poire', 'Pêche'])
```

# Module `json` → traitement de fichiers JSON

- ▷ Le module **json** permet de traiter ... des fichiers JSON.
- ▷ Permet d'encoder une liste ou dictionnaire en objet JSON.
- ▷ Permet de décoder un objet JSON en liste ou dictionnaire.



macademia

Documentation : <https://docs.python.org/fr/3/library/json.html>

# Module `json` → traitement de fichiers JSON

- ▷ `json.dumps(pythonObject)`
  - › Encoder une liste ou dictionnaire Python en JSON dans un str.
- ▷ `json.dump(pythonObject, fd)`
  - › Encoder une liste ou dictionnaire Python en JSON dans un fichier.
- ▷ `json.loads(jsonObject)`
  - › Décoder un JSON en liste ou dictionnaire Python.
- ▷ `json.load(fd)`
  - › Décoder un JSON en liste ou dictionnaire Python dans un fichier.

# Module *json* → traitement de fichiers JSON

```
import json

objet = json.dumps([1, 2, 3, {'4': 5, '6': 7}],
                   separators=(',', ':'),
                   sort_keys=True,
                   indent=4)

with open('toto.json', 'w+') as jsonfile:
    json.dump([1, 2, 3, {'4': 5, '6': 7}], jsonfile)

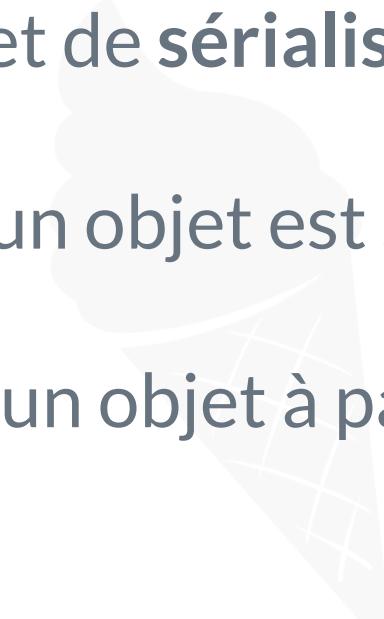
with open('toto.json', 'r+') as jsonfile:
    print(json.load(jsonfile))
```

# Module *json* → correspondance Python <→ JSON

Python	JSON
<code>dict</code>	<code>object</code>
<code>list, tuple</code>	<code>array</code>
<code>str</code>	<code>string</code>
<code>int, float</code>	<code>number</code>
<code>True</code>	<code>true</code>
<code>False</code>	<code>false</code>
<code>None</code>	<code>null</code>

# Module *pickle* → sérialiser des données

- ▷ Le module **pickle** permet de **sérialiser** des données.
- ▷ Cela signifie que l'état d'un objet est sauvegardé dans un fichier.
- ▷ Il est possible de recréer un objet à partir d'un fichier de sérialisation.



# macademia

Documentation : <https://docs.python.org/fr/3/library/pickle.html>

# Module *pickle* → sérialiser des données

- ▷ `pickle.dump(objet, open(nomfichier, 'wb'))`
  - › Sérialiser un objet dans un fichier.
  
- ▷ `objet = pickle.load(open(nomfichier, 'rb'))`
  - › Dé-sérialiser un objet contenu dans un fichier.

A faint watermark of the Macademia logo, featuring a stylized graduation cap and diploma, is positioned in the center of the slide.

Macademia

# Module *sqlite3* → gestion des bases de données

- ▷ Le module **sqlite3** permet de travailler avec le moteur SQLite.
- ▷ Nécessite une connexion à un fichier de stockage de base de données.
- ▷ Permet d'exécuter des requêtes SQL sur des tables.

Documentation : <https://docs.python.org/fr/3/library/sqlite3.html>

# Module *sqlite3* → gestion des bases de données

```
connexion_db.py

import sqlite3

# Etablissement de la connexion (fichier créé s'il n'existe pas)
connexion = sqlite3.connect('NomDuFichier.db')

# Récupérer le curseur
curseur = connexion.cursor()

# Création d'une table Personne
curseur.execute('''CREATE TABLE IF NOT EXISTS Personne (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nom TEXT,
    age INTEGER)'''')
```

<b>id</b>	<b>nom</b>	<b>age</b>
-----------	------------	------------

# Module *sqlite3* → gestion des bases de données

```
connexion_db.py

...
# Ajout d'une ligne dans la table Personne
curseur.execute('''INSERT INTO Personne(nom, age) VALUES ('Bob', 35)''')

# Valider les modifications
connexion.commit()

# Lire toutes les lignes de la table Personne
curseur.execute('''SELECT * FROM Personne''')
resultat = curseur.fetchone()

while resultat:
    print(resultat)
    resultat = curseur.fetchone()
```

<b>id</b>	<b>nom</b>	<b>age</b>
1	Bob	35

# Module *sqlite3* → gestion des bases de données

```
connexion_db.py

...
# Supprimer la table Personne
curseur.execute('''DROP TABLE Personne''')

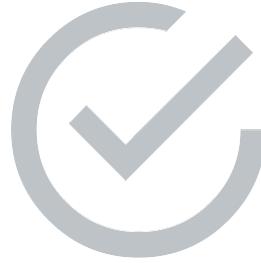
# Valider les modifications
connexion.commit()

# Déconnexion
connexion.close()
```

<b>id</b>	<b>nom</b>	<b>age</b>
1	Bob	35

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Qualité de code

# pip

- ▷ **pip** est un gestionnaire de paquets utilisé pour installer et gérer des paquets écrits en Python.
- ▷ De nombreux paquets peuvent être trouvés sur [PyPI](#).
- ▷ Nous avons besoin de pip pour installer des librairies externes.

ⓘ Il est conseillé de lancer les commandes pip en mode administrateur

# pip

▷ Installer un package :

```
$ pip install nom_du_package
```

▷ Mettre à jour un package :

```
$ pip install nom_du_package --upgrade
```

▷ Désinstaller un package :

```
$ pip uninstall nom_du_package
```

# Installation de pip

1. Aller à <https://bootstrap.pypa.io/get-pip.py> et copier le code.
2. Enregistrer le code dans un fichier `get-pip.py`
3. Dans un invite de commande, se déplacer à l'emplacement du fichier et lancer la commande **python get-pip.py**
4. Vérifier que l'installation a réussi.  
Successfully uninstalled pip-20.2.2

# pylint

- ▷ **pylint** est un outil de vérification de code et de qualité du code pour Python.
- ▷ Il utilise les recommandations officielles de style PEP 8.

```
***** Module chien
chien.py:15:30: C0326: No space allowed before :
    def manger(self, aliment) :
        ^ (bad-whitespace)
chien.py:18:29: C0326: No space allowed before :
    def dormir(self, heures) :
        ^ (bad-whitespace)
chien.py:38:37: C0326: No space allowed before :
    if choix.capitalize() == "O" :
        ^ (bad-whitespace)
chien.py:58:0: C0305: Trailing newlines (trailing-newlines)
chien.py:1:0: C0111: Missing module docstring (missing-docstring)
chien.py:11:4: R0913: Too many arguments (6/5) (too-many-arguments)
chien.py:15:4: C0111: Missing method docstring (missing-docstring)
chien.py:15:4: R0201: Method could be a function (no-self-use)
chien.py:18:4: C0111: Missing method docstring (missing-docstring)
```

# pylint

- ▷ Un outil paramétrable.
- ▷ Un outil personnalisable (possibilité d'ajouter des plugins).
- ▷ Permet à l'instar de PyChecker de vérifier le code Python :
  - › la taille des lignes ;
  - › les noms de variables / règles définies ;
  - › respect de Python Style ;
  - › etc.

Macademia

# Installation de pylint

1. Dans un invite de commande, lancer la commande

```
$ pip install pylint
```

2. Vérifier que pylint soit bien installé en lançant la commande

```
$ pylint --version
```

# Lancement de pylint

1. Dans un invite de commande, se déplacer dans le répertoire contenant le fichier à analyser.
2. Lancer la commande **pylint nom\_module.py**
3. Le résultat de l'analyse s'affiche, sous la forme suivante :  
MESSAGE\_TYPE: LINE\_NUM: [OBJECT:] MESSAGE
4. La dernière ligne attribue une note au code.  
L'objectif est bien sûr 10/10.

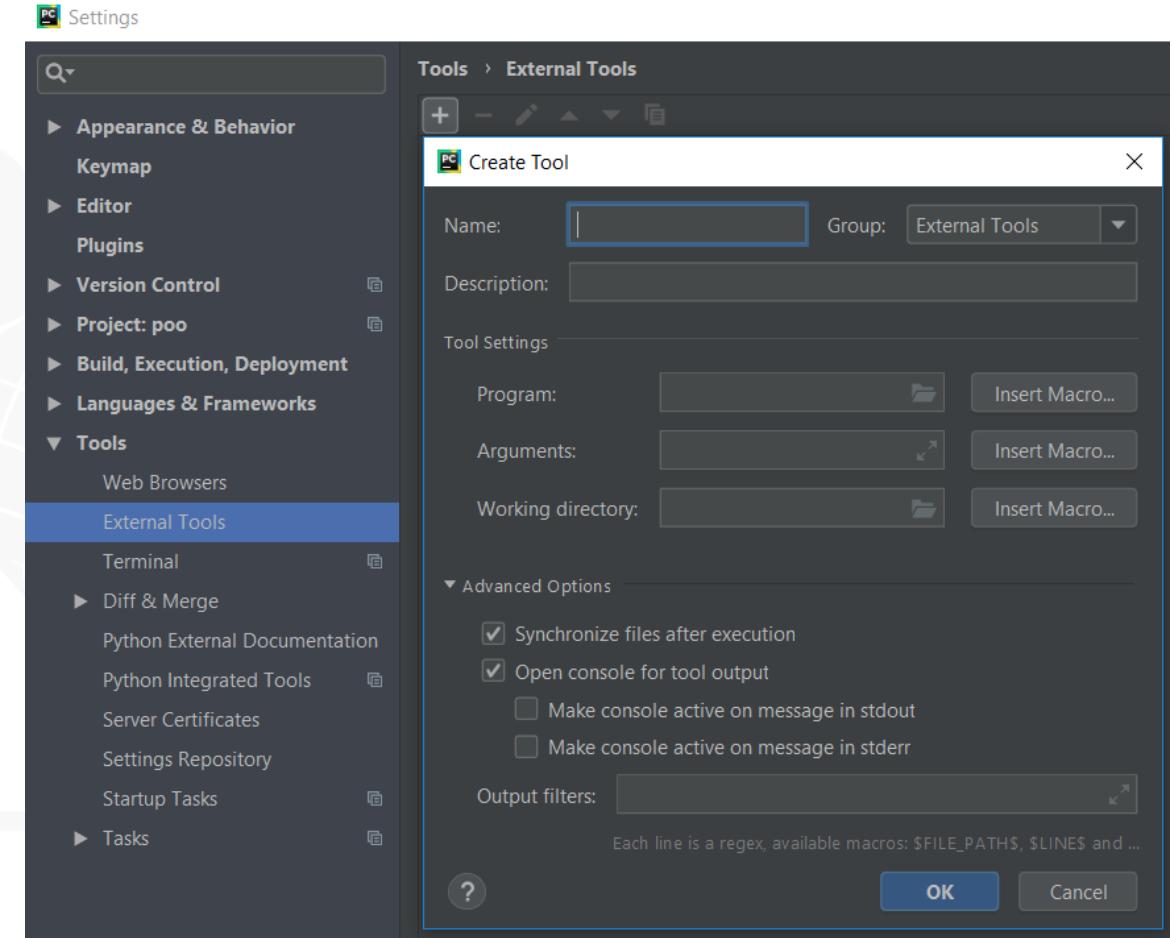
# Intégration de pylint dans PyCharm

- Pour retrouver l'emplacement où a été installé pylint, ouvrir l'invite de commande et taper

**where pylint** sur Windows  
**which pylint** sur Unix

- Dans PyCharm, aller dans **File > Settings > Tools > External Tools**.

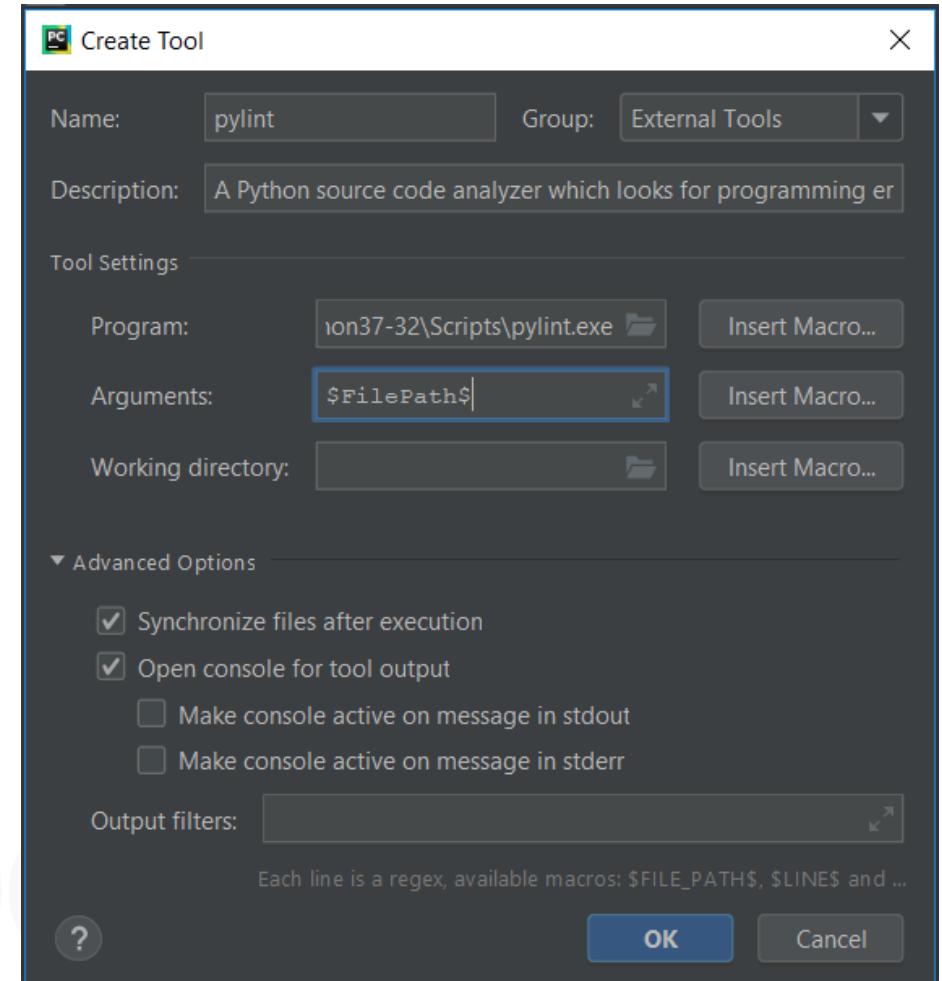
- Cliquer sur le « + ».



# Intégration de pylint dans PyCharm

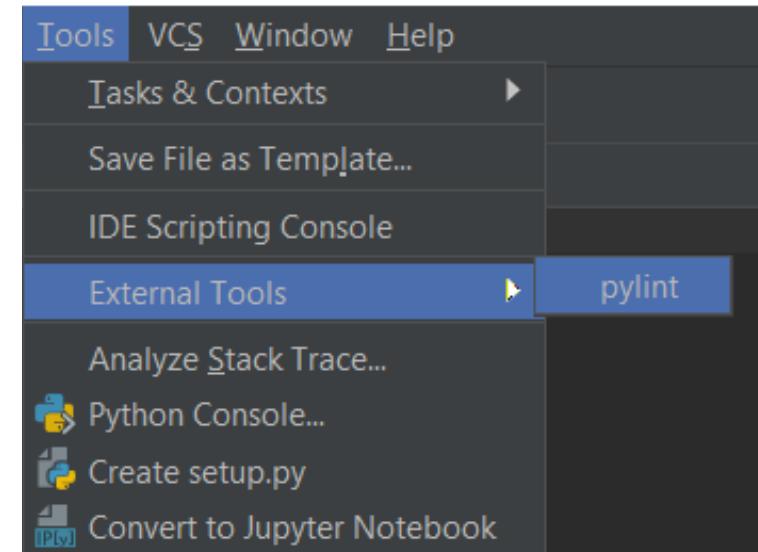
4. Dans la fenêtre « Create Tool », ces informations :

- **Name** : pylint
- **Description** : A Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells.
- **Program** : l'emplacement que vous avez récupéré.
- **Arguments** : \$FilePath\$



# Intégration de pylint dans PyCharm

5. Placez-vous dans le fichier sur lequel vous souhaitez lancer l'analyse.
6. Aller dans **Tools > External Tools > pylint**



# Formatage de code avec Black

- ▷ Le module externe **black** permet de **formater un fichier Python**.
- ▷ Se base sur les recommandations PEP 8.
- ▷ L'installation se fait grâce à la commande suivante :

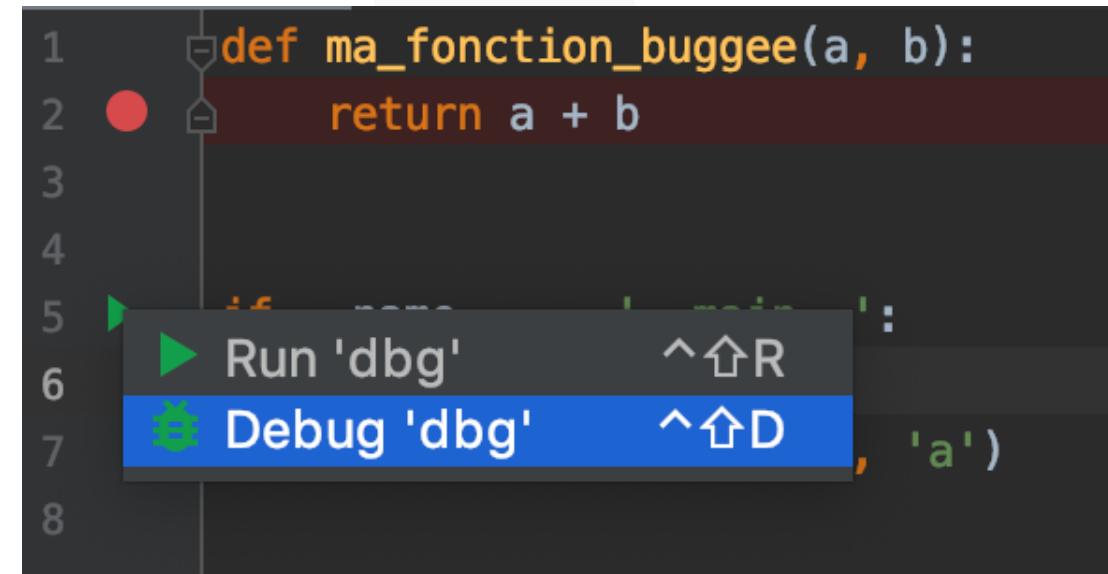
```
$ pip install black
```

- ▷ Il faut ensuite la commande :

```
$ black fichier_a_formater.py
```

Documentation : <https://black.readthedocs.io/en/stable/>

# Le debugger de PyCharm



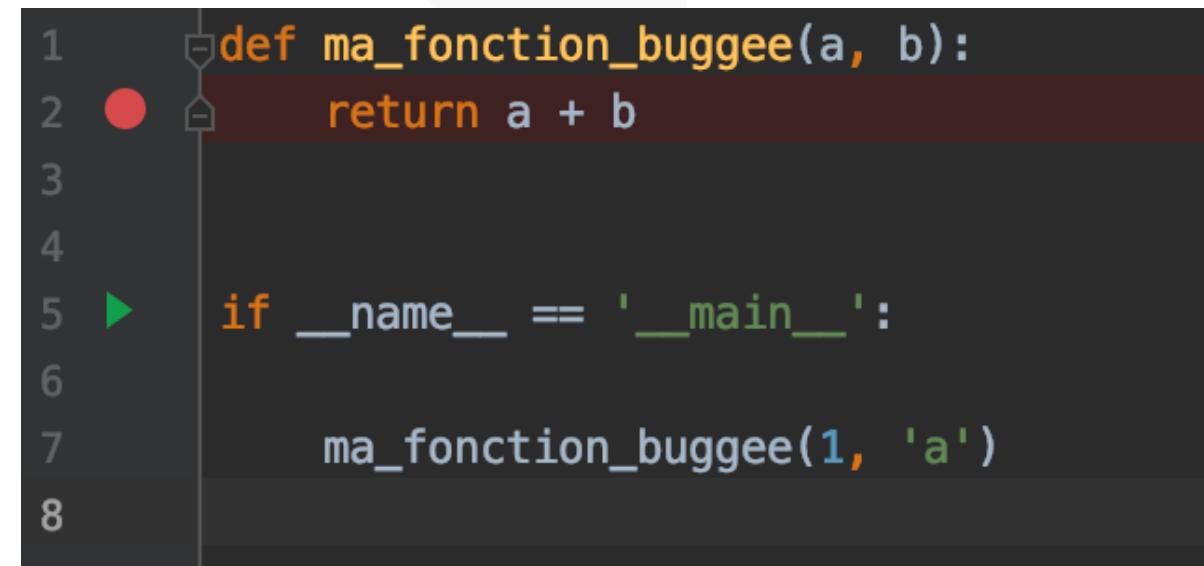
A screenshot of the PyCharm IDE interface. A code editor window shows a Python function definition:

```
1 def ma_fonction_buggee(a, b):
2     return a + b
3
4
5
6
7
8
```

The second line contains a red circular breakpoint marker. A context menu is open at the bottom of the editor, listing two options:

- Run 'dbg' (with keyboard shortcut ⌘R)
- Debug 'dbg' (with keyboard shortcut ⌘D, highlighted in blue)

# Le debugger de PyCharm : placer un point d'arrêt

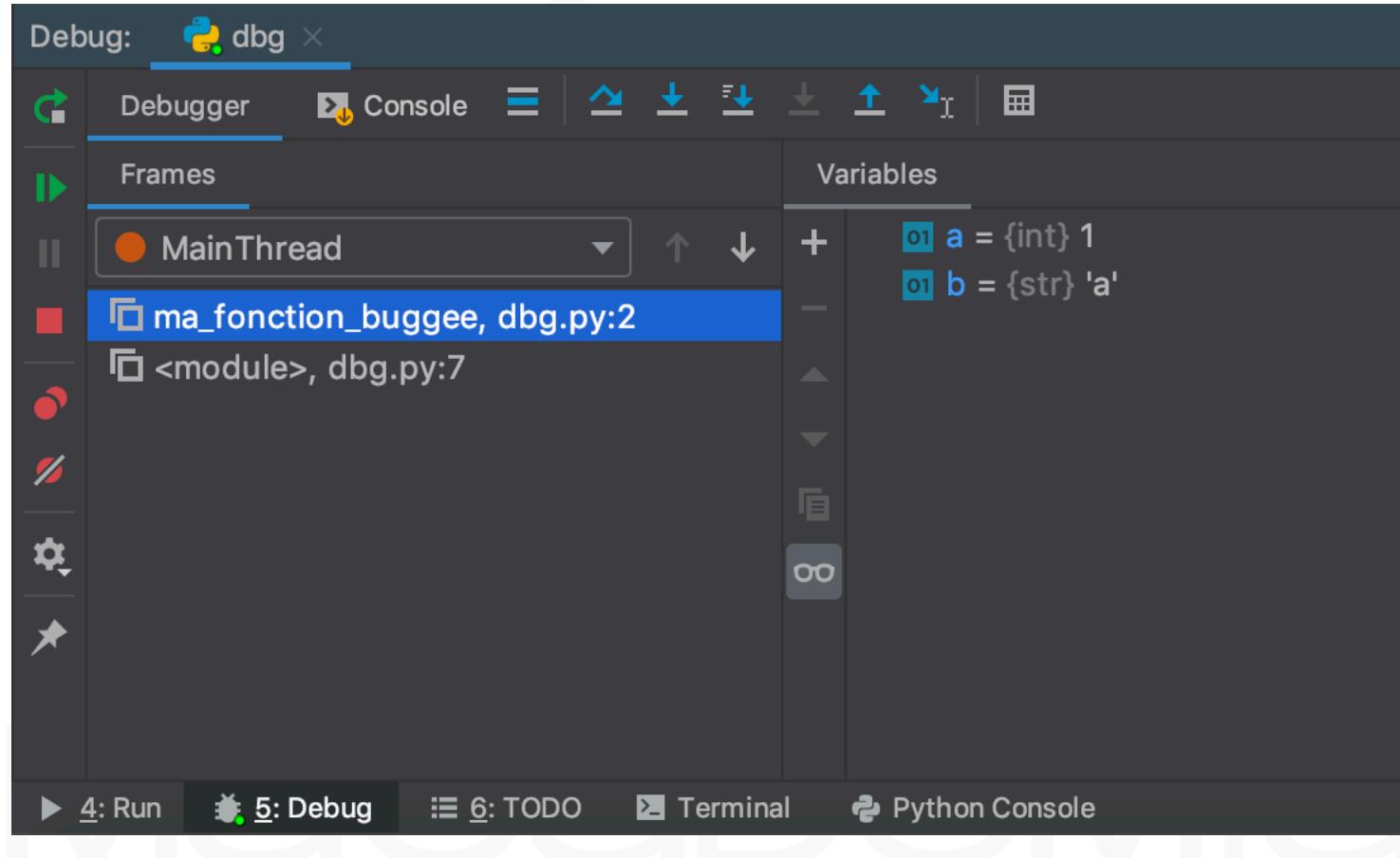


A screenshot of the PyCharm code editor in debug mode. The code is as follows:

```
1 def ma_fonction_buggee(a, b):
2     ● return a + b
3
4
5 ► if __name__ == '__main__':
6
7     ma_fonction_buggee(1, 'a')
8
```

The line `return a + b` is highlighted in red, indicating it is the current line of execution. A red dot at the start of line 2 indicates a breakpoint has been set there. A green triangle icon at the start of line 5 indicates the program is currently executing or about to execute that line.

# Le debugger de PyCharm : step over, step into



# Les tests unitaires

- ▷ Les tests sont **indispensables**.
- ▷ Utilisation du module **unittest** pour les tests unitaires.
- ▷ Comparaisons entre résultat attendu et résultat obtenu.
- ▷ Une classe de test hérite de **unittest.TestCase**
- ▷ Il est conseillé de nommer sa classe avec préfixe ou suffixe « **Test** ».

# Les méthodes d'assertion

Méthode	Test	Description
<code>assertEqual(a, b)</code>	<code>a == b</code>	Vérifie si a égal à b
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	Vérifie si a est différent de b
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	Vérifie si x vaut <i>True</i>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	Vérifie si x vaut <i>False</i>
<code>assertIs(a, b)</code>	<code>a is b</code>	Vérifie si a est b
<code>assert IsNot(a, b)</code>	<code>a is not b</code>	Vérifie si a n'est pas b
<code>assertIsNone(x)</code>	<code>x is None</code>	Vérifie si x a la valeur <i>None</i>
<code>assert IsNotNone(x)</code>	<code>x is not None</code>	Vérifie si x n'a pas la valeur <i>None</i>
<code>assertIn(a, b)</code>	<code>a in b</code>	Vérifie si a est présent dans b
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	Vérifie si a n'est pas présent dans b
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	Vérifie si a est une instance de b
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	Vérifie si a n'est pas une instance de b

# Une classe de test

```
test_operations.py
```

```
import unittest

class TestOperations(unittest.TestCase):

    def test_addition(self):
        resultat = 1 + 2
        self.assertEqual(resultat, 3)

    def test_soustraction(self):
        resultat = 6 - 1
        self.assertEqual(resultat, 5)
```

```
Ran 2 tests in 0.001s
OK
```

# En résumé

**Dans ce chapitre, nous avons vu ...**



# Les interfaces graphiques

# Principes de programmation des interfaces graphiques

- ▷ On parle également d'IHM ou de GUI.
- ▷ Une fenêtre est un assemblage d'éléments graphiques superposés.
- ▷ Les éléments graphiques ne sont rien de plus que des **objets**.

macademia

# Présentation de la bibliothèque Tkinter

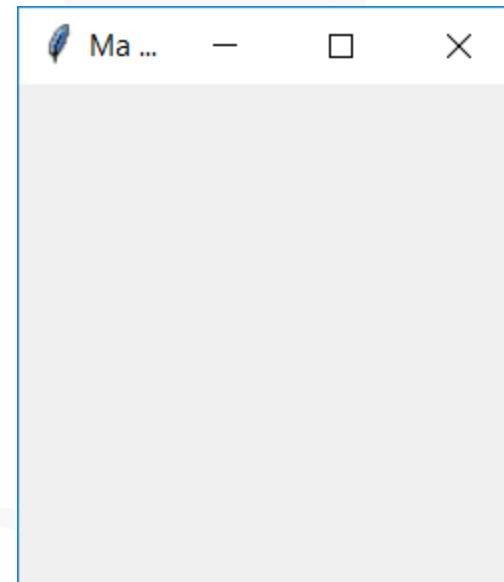
- ▷ **Tkinter** est un module intégré à la bibliothèque standard de Python.
  - › Il n'est pas maintenu directement par les développeurs de Python.
- ▷ Il s'agit d'une interface de la bibliothèque Tk, écrite pour **Tcl**.
- ▷ Permet de créer des interfaces graphiques via Python.
- ▷ Disponible sur Windows, macOS et la plupart des systèmes Unix.

Documentation officielle (anglais) : <https://docs.python.org/fr/3/library/tkinter.html>

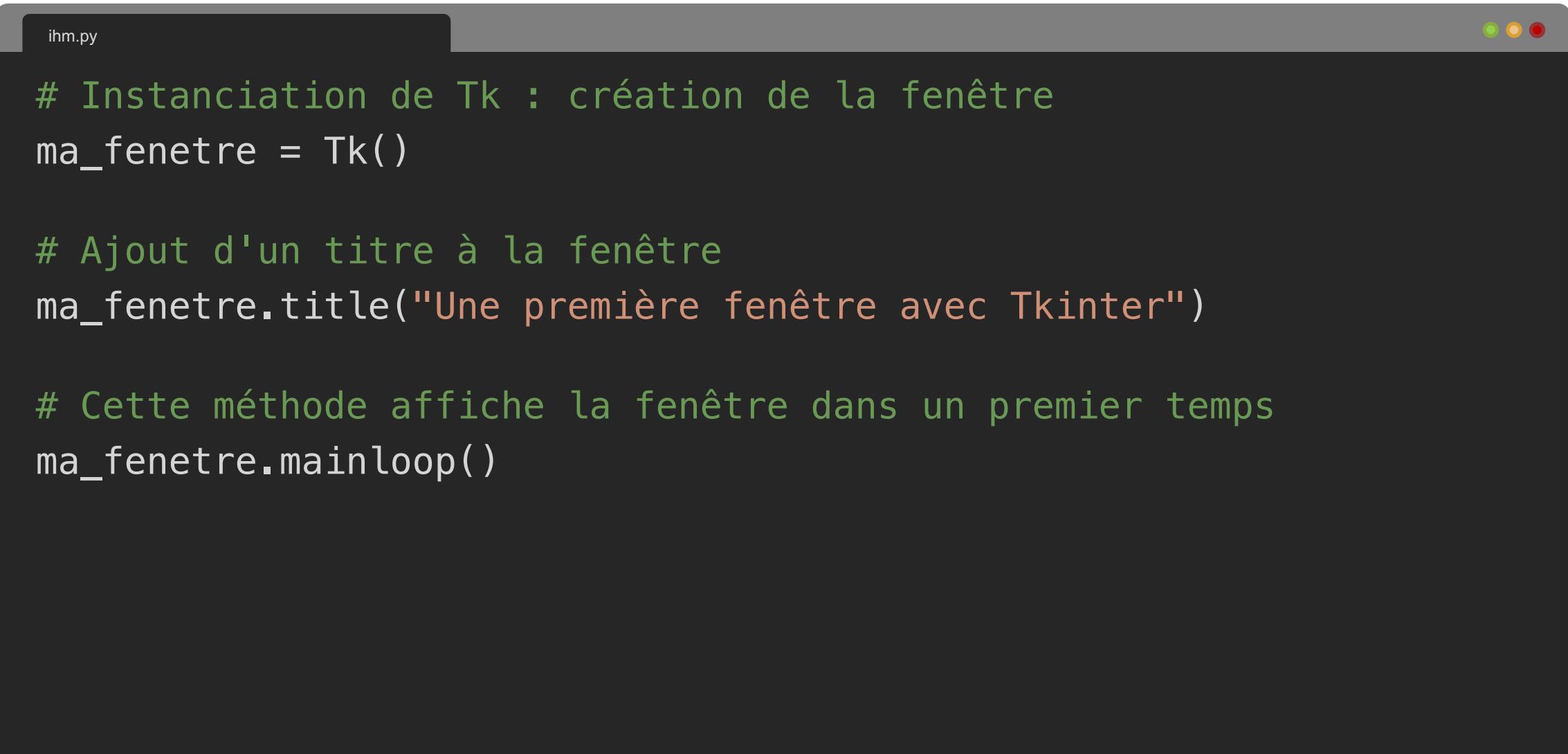
Documentation non-officielle (français) : <http://tkinter.fdex.eu/index.html>

# Une première fenêtre

- ▷ La création d'une fenêtre correspond à l'instanciation d'un objet Tk.



# Une première fenêtre



The screenshot shows a dark-themed code editor window titled "ihm.py". The code inside the window is as follows:

```
# Instanciation de Tk : création de la fenêtre
ma_fenetre = Tk()

# Ajout d'un titre à la fenêtre
ma_fenetre.title("Une première fenêtre avec Tkinter")

# Cette méthode affiche la fenêtre dans un premier temps
ma_fenetre.mainloop()
```

# Les widgets

- ▷ Les widgets sont les différents éléments graphiques.
- ▷ Lorsque l'on crée un widget, son premier paramètre est toujours le **paramètre parent** : c'est-à-dire le conteneur de ce widget.
  - › Par défaut, on indique la fenêtre dans laquelle le widget est contenu.
- ▷ Afin d'ajouter un widget à son parent, on appelle la méthode :
  - › **pack()** : pour placer le widget à la suite du précédent.
  - › **grid()** : pour placer le widget selon une grille.
  - › **place()** : pour placer le widget de manière absolue.

# Placement des widgets

ihm.py

```
# Placer le widget
mon_widget.pack()

# Placer le widget à la ligne 1 colonne 2
mon_widget.grid(row=1, column=2)

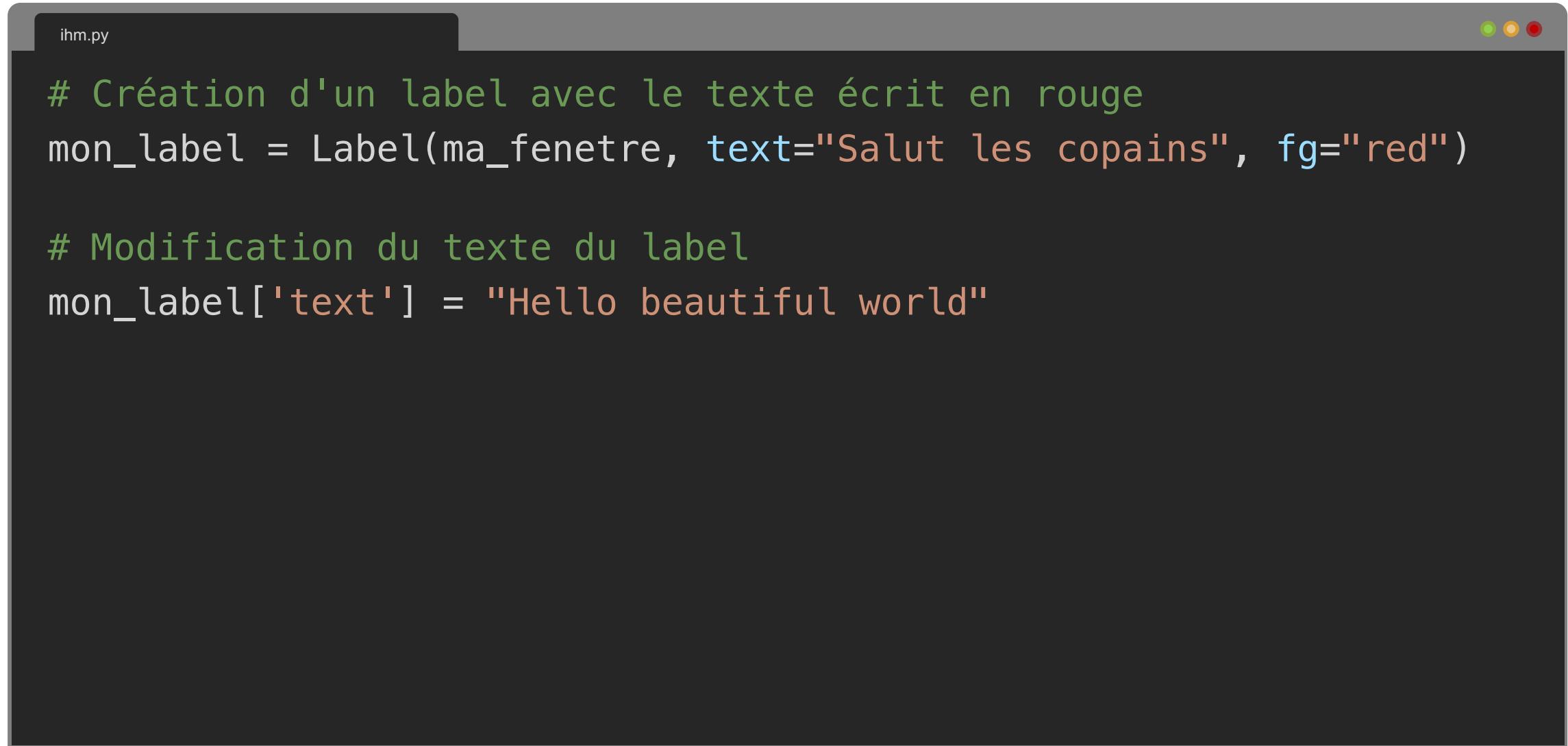
# Placer le widget à 120px horizontaux et 300px verticaux
# selon l'angle supérieur gauche
mon_widget.place(x=120, y=300)
```

# Label

- ▷ Un **Label** représente un simple **texte**.
- ▷ Il est possible de modifier le contenu du texte et son style.



# Label



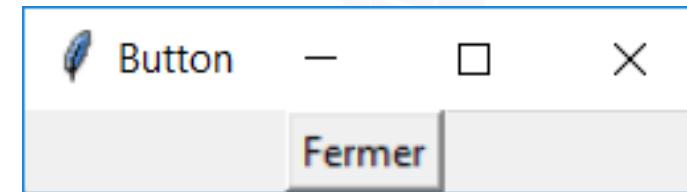
The screenshot shows a dark-themed code editor window titled "ihm.py". The code in the editor is as follows:

```
# Création d'un label avec le texte écrit en rouge
mon_label = Label(ma_fenetre, text="Salut les copains", fg="red")

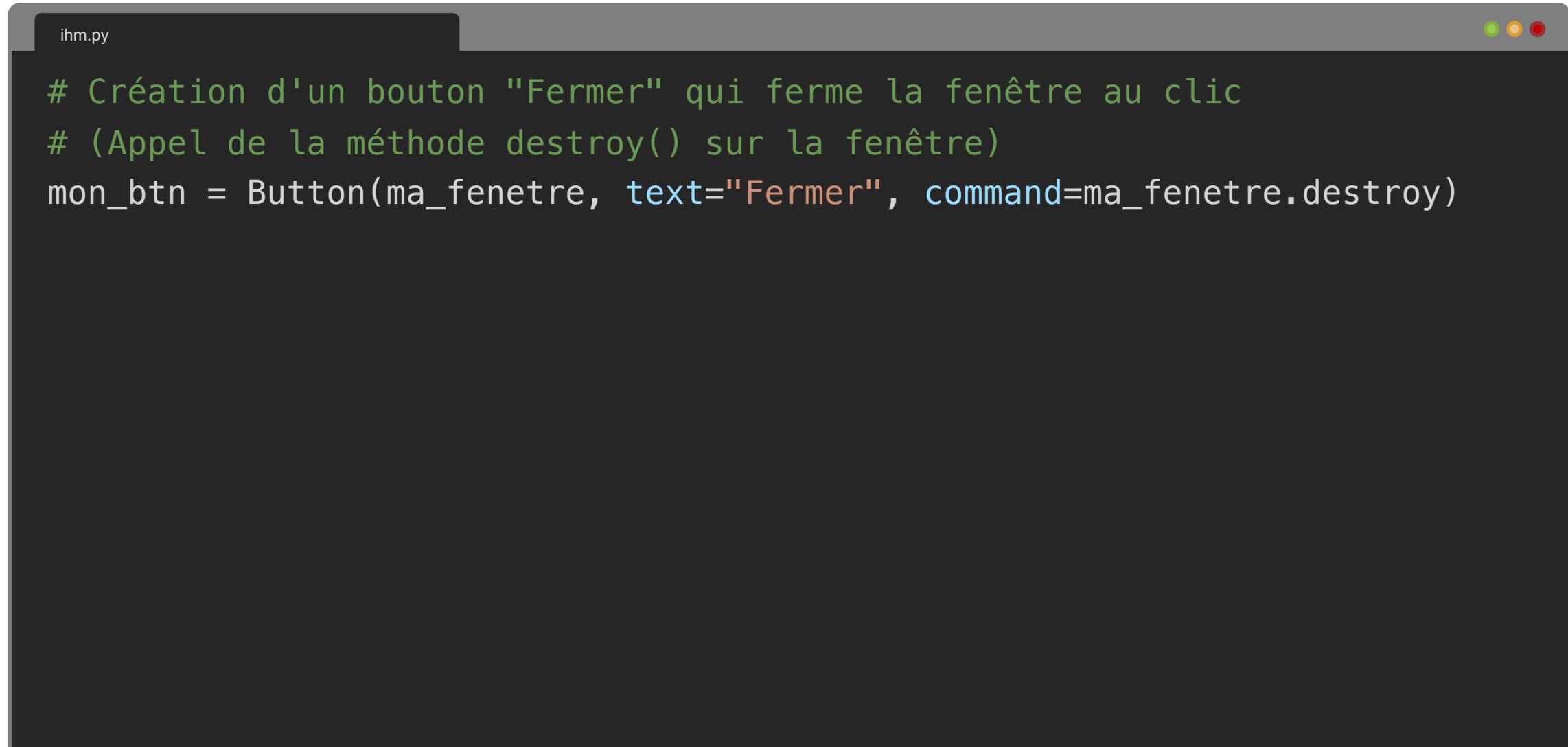
# Modification du texte du label
mon_label['text'] = "Hello beautiful world"
```

# Button

- ▷ Un **Button** représente un bouton d'action.
- ▷ Il est possible de lui associer un texte et une action.



# Button

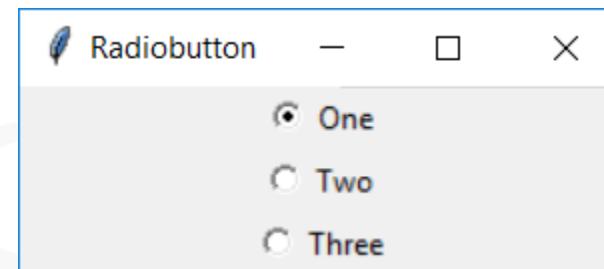


The screenshot shows a dark-themed code editor window titled "ihm.py". The code in the editor is:

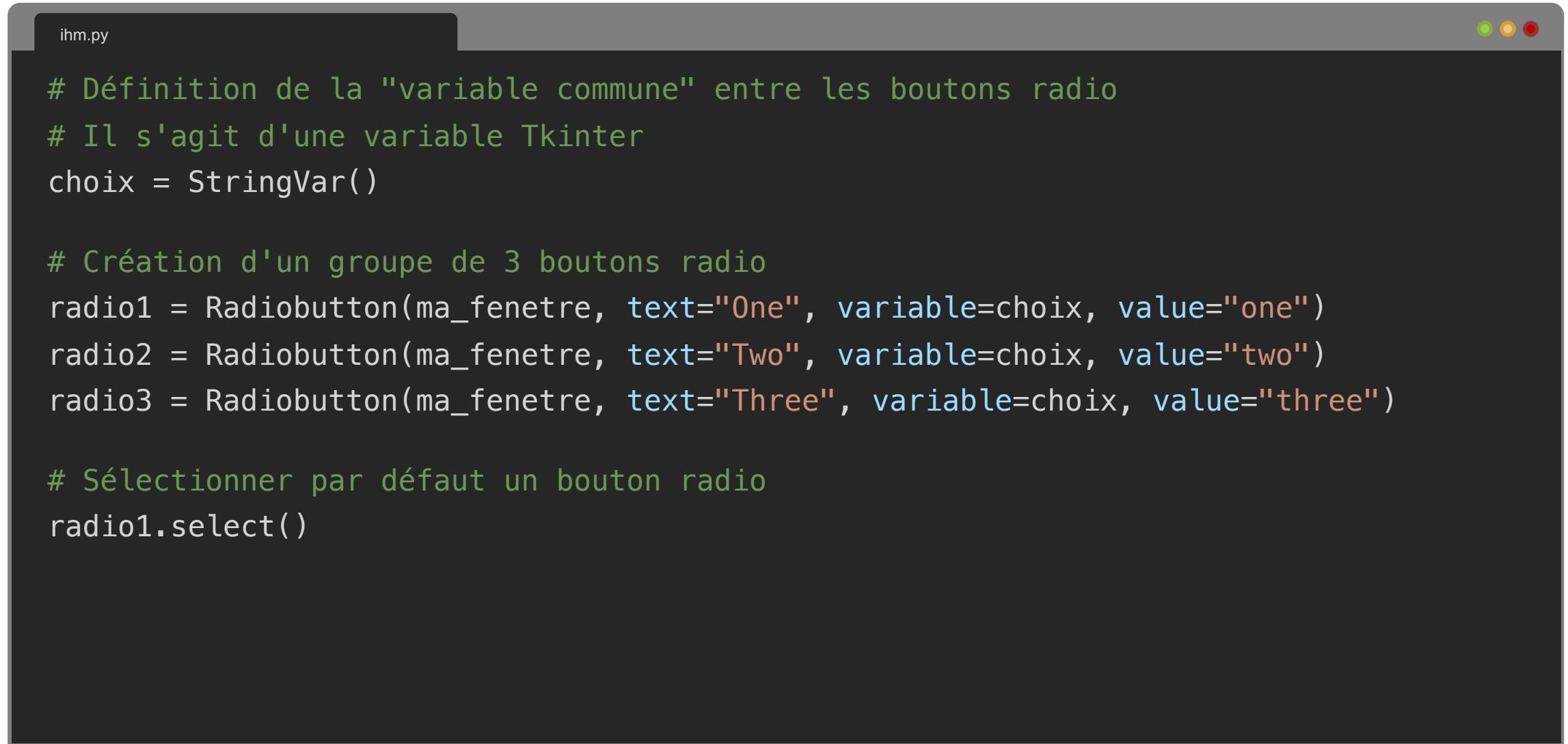
```
# Création d'un bouton "Fermer" qui ferme la fenêtre au clic
# (Appel de la méthode destroy() sur la fenêtre)
mon_btn = Button(ma_fenetre, text="Fermer", command=ma_fenetre.destroy)
```

# Radiobutton

- ▷ Un **Radiobutton** correspond à un **bouton radio**.
  - › Ils sont généralement associés en **groupe**.
  - › Lorsque l'on clique sur un bouton du groupe, les autres sont désélectionnés.
- ▷ Il est possible de leur associer un nom, une valeur, une action.
- ▷ Pour grouper les boutons radio, on utilise une variable commune.



# Radiobutton



ihm.py

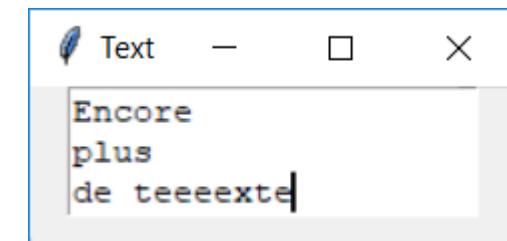
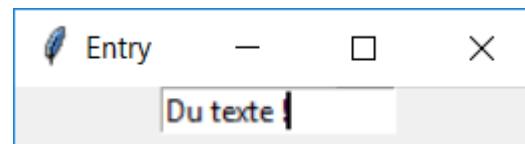
```
# Définition de la "variable commune" entre les boutons radio
# Il s'agit d'une variable Tkinter
choix = StringVar()

# Création d'un groupe de 3 boutons radio
radio1 = Radiobutton(ma_fenetre, text="One", variable=choix, value="one")
radio2 = Radiobutton(ma_fenetre, text="Two", variable=choix, value="two")
radio3 = Radiobutton(ma_fenetre, text="Three", variable=choix, value="three")

# Sélectionner par défaut un bouton radio
radio1.select()
```

# Entry et Text

- ▷ Une **Entry** correspond à un **champ de texte** de formulaire.
- ▷ Un **Text** correspond à un champ adapté à un **long texte**.



# Entry et Text

```
ihm.py

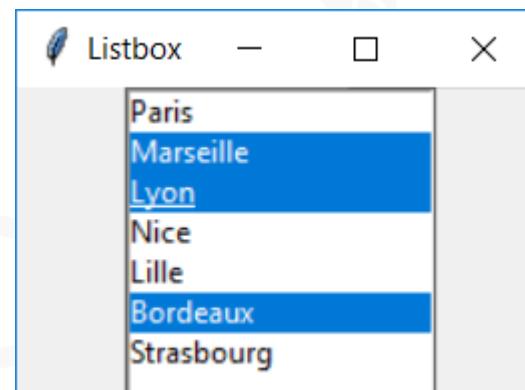
# Création de la variable qui contient le texte de l'Entry
texte = StringVar()

# Création de l'Entry (taille 15 caractères)
mon_entry = Entry(ma_fenetre, textvariable=texte, width=15)

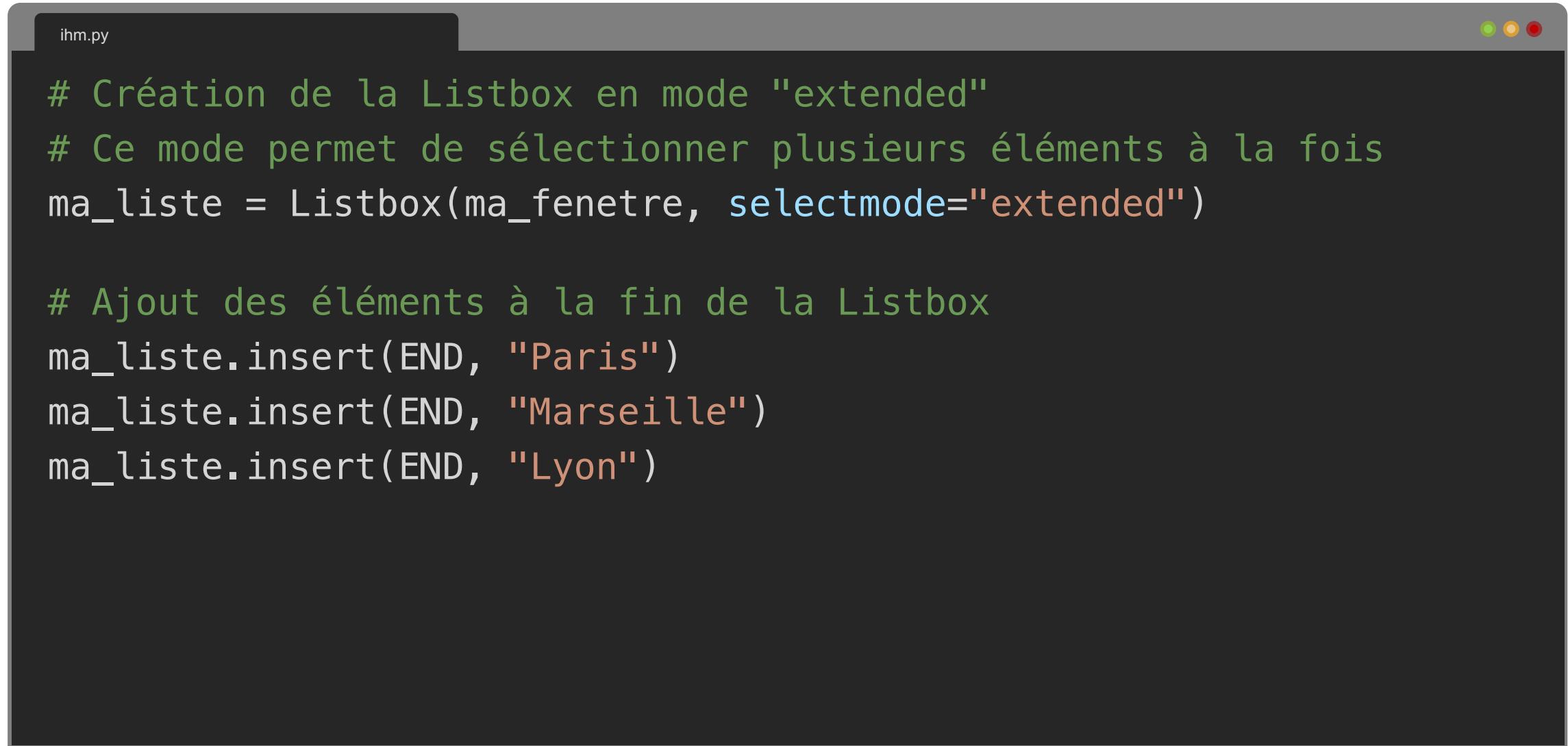
# Création du Text (taille 3 lignes de 20 caractères)
mon_text = Text(ma_fenetre, height=3, width=20)
```

# Listbox

- ▷ Une **ListBox** correspond à une **liste de valeurs**.
- ▷ On ajoute les éléments de la liste avec la méthode **insert()**
- ▷ Il est possible de déterminer le nombre d'élément que l'on peut sélectionner dans la liste grâce au paramètre **selectmode**.



# Listbox



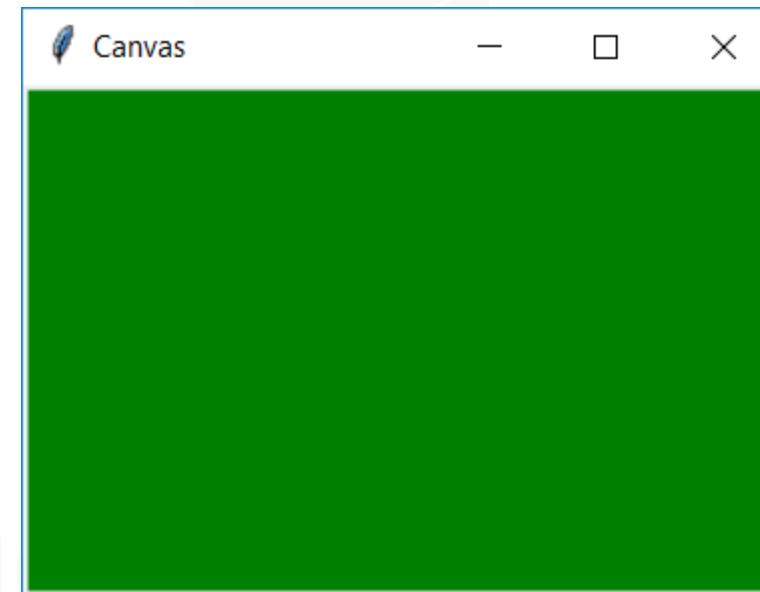
The screenshot shows a dark-themed code editor window titled "ihm.py". The code is written in Python and demonstrates how to create a Listbox with multiple selection mode and add items to it.

```
# Création de la Listbox en mode "extended"
# Ce mode permet de sélectionner plusieurs éléments à la fois
ma_liste = Listbox(ma_fenetre, selectmode="extended")

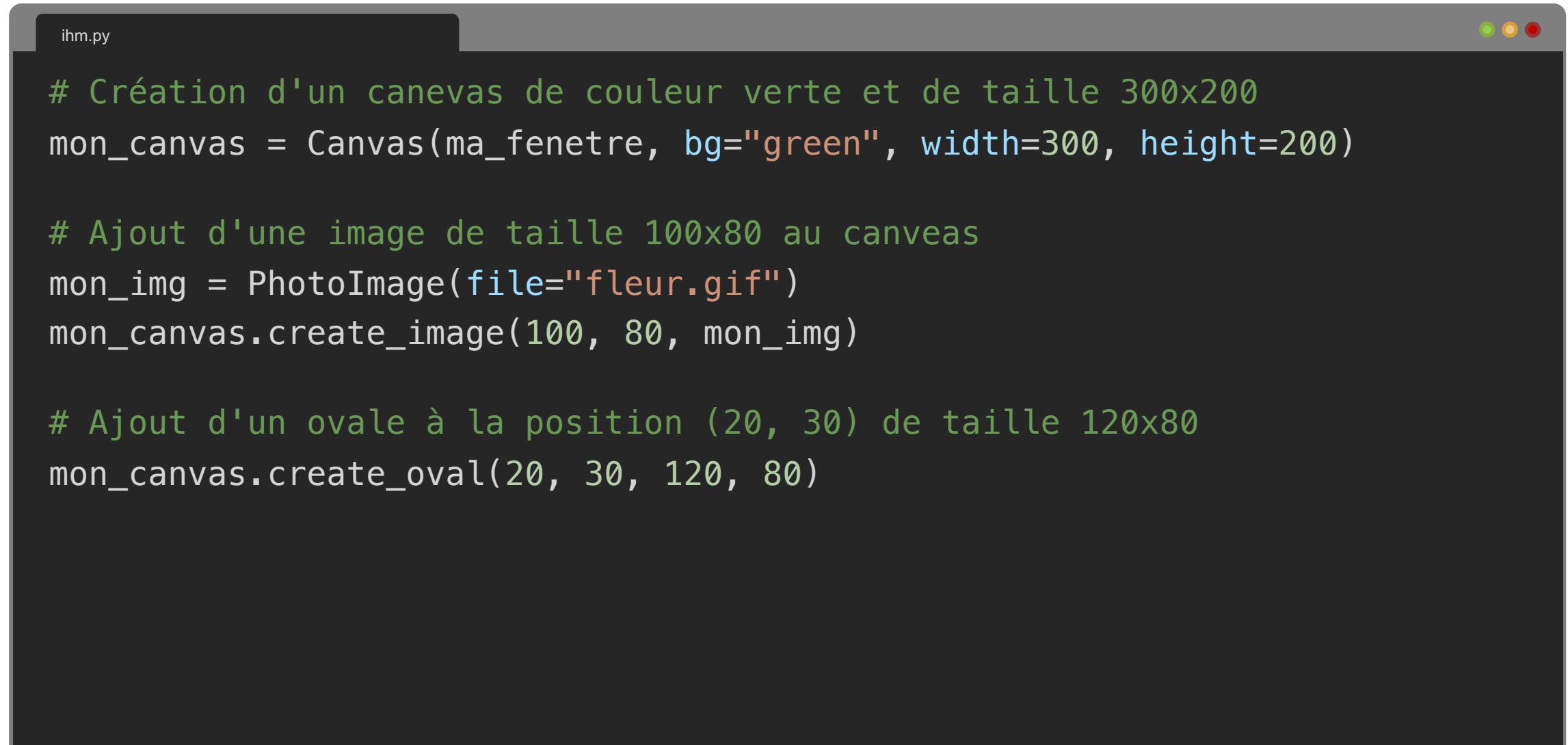
# Ajout des éléments à la fin de la Listbox
ma_liste.insert(END, "Paris")
ma_liste.insert(END, "Marseille")
ma_liste.insert(END, "Lyon")
```

# Canvas

- ▷ Un **Canvas** représente **surface rectangulaire délimitée**, dans laquelle on peut insérer divers dessins et images.



# Canvas



The screenshot shows a code editor window titled "ihm.py". The code uses the Tkinter library to create a green canvas, add a flower image, and draw an oval.

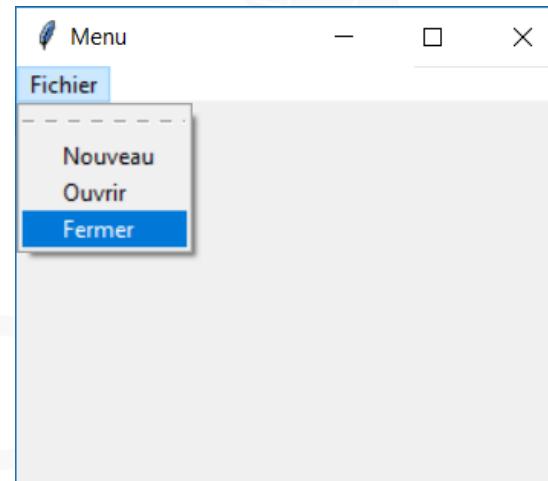
```
# Création d'un canevas de couleur verte et de taille 300x200
mon_canvas = Canvas(ma_fenetre, bg="green", width=300, height=200)

# Ajout d'une image de taille 100x80 au canveas
mon_img = PhotoImage(file="fleur.gif")
mon_canvas.create_image(100, 80, mon_img)

# Ajout d'un ovale à la position (20, 30) de taille 120x80
mon_canvas.create_oval(20, 30, 120, 80)
```

# Menu

- ▷ Un **Menu** correspond à une **barre de menus** et ses **sous-menus**.
- ▷ Il est possible de déplacer un menu de sa barre de menus (il se comporte alors comme une fenêtre).



# Menu

```
ihm.py
```

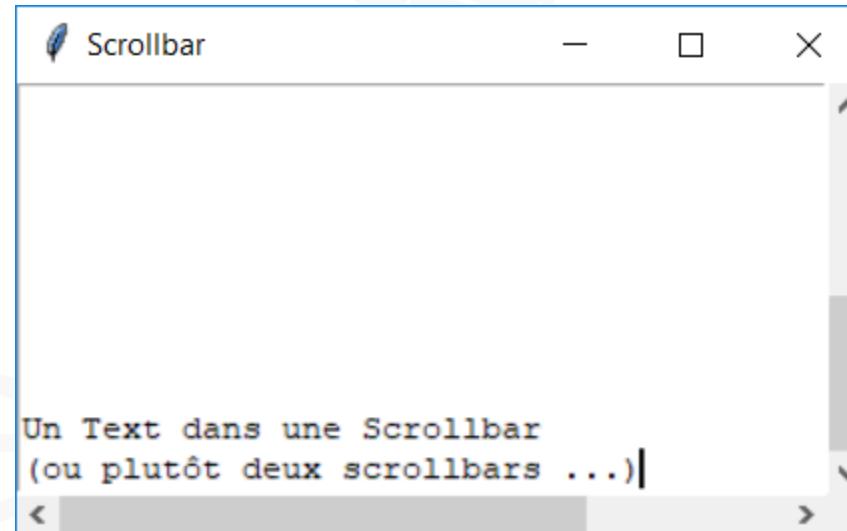
```
# Création de la barre de menu
barre_menu = Menu(ma_fenetre)

# Définition de la barre de menu en tant que barre de menu de la fenêtre
ma_fenetre['menu'] = barre_menu

# Création du sous-menu "Fichier"
sous_menu = Menu(barre_menu)
barre_menu.add_cascade(label='Fichier', menu=sous_menu)
sous_menu.add_command(label='Nouveau', command=nouveau)
sous_menu.add_command(label='Ouvrir', command=ouvrir)
sous_menu.add_command(label='Fermer', command=ma_fenetre.destroy)
```

# Scrollbar

- ▷ Une **Scrollbar** correspond à une barre de défilement.
- ▷ Elle peut être associée à d'autres widgets dont la taille peut être importante.



# Scrollbar

```
ihm.py

# Création d'une zone de texte de 10 lignes de 40 caractères
mon_text = Text(ma_fenetre, height=10, width=40)
mon_text.grid()

# Création de la scrollbar horizontale
defil_horiz = Scrollbar(ma_fenetre, orient='horizontal', command=mon_text.xview)
defil_horiz.grid(column=0, row=1, sticky='ew')

# Création de la scrollbar verticale
defil_vert = Scrollbar(ma_fenetre, orient='vertical', command=mon_text.yview)
defil_vert.grid(column=1, row=0, sticky='ns')

# Configuration de la scrollbar
mon_text['xscrollcommand'] = defil_horiz.set
mon_text['yscrollcommand'] = defil_vert.set
defil_horiz.config(command=mon_text.xview)
defil_vert.config(command=mon_text.yview)
```

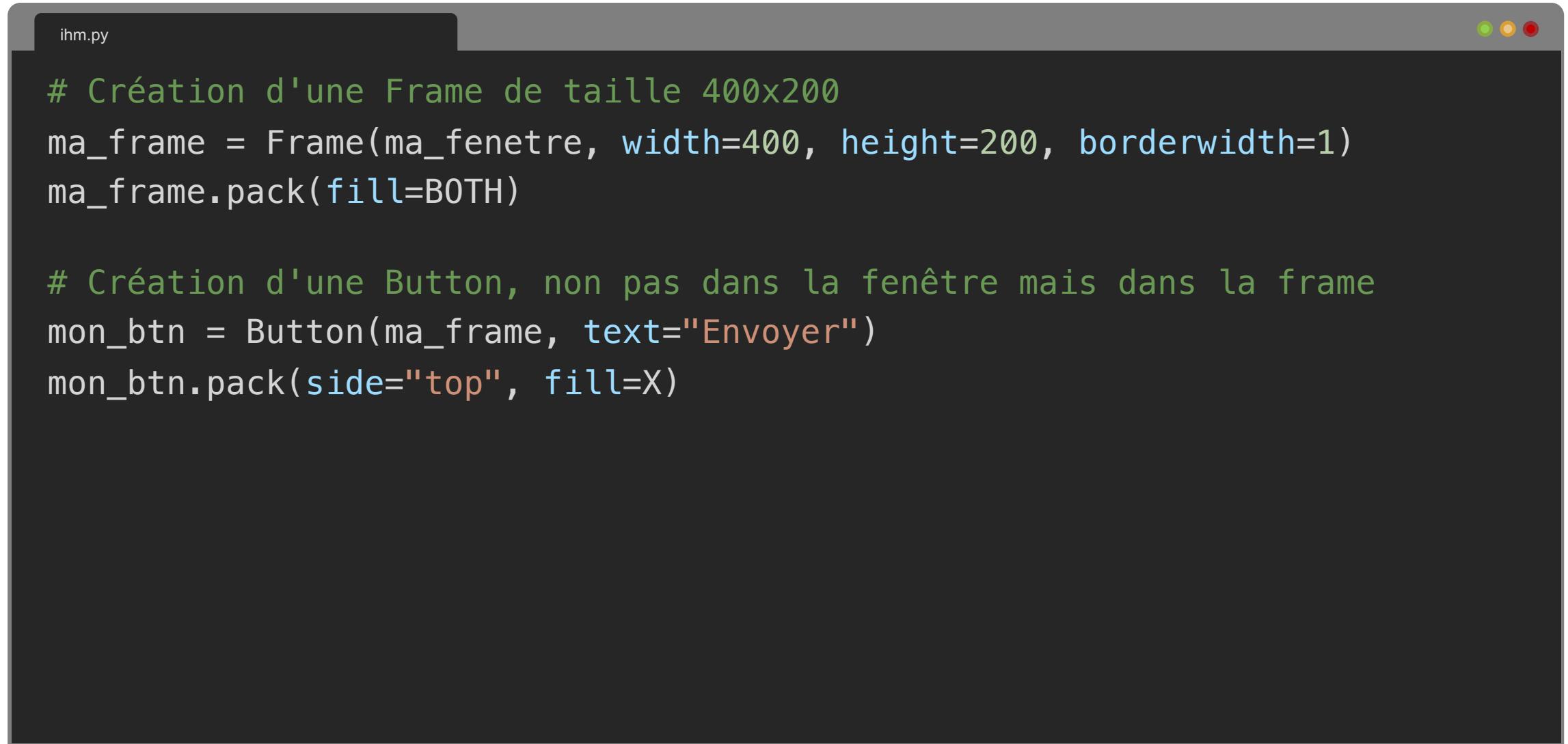
# Frame

- ▷ Une **Frame** est un cadre qui permet de contenir d'autres widgets.
- ▷ Si l'on souhaite placer des widgets dans une **Frame**, il faudra alors indiquer la **Frame** en tant que parent dans le constructeur du widget.



macademia

# Frame



The screenshot shows a code editor window with a dark theme. The file name "ihm.py" is visible in the top-left corner. In the top-right corner, there are three small colored circles (green, yellow, red) typically used for window control buttons. The code itself is written in Python and uses the Tkinter library to create a window with a frame and a button.

```
# Création d'une Frame de taille 400x200
ma_frame = Frame(ma_fenetre, width=400, height=200, borderwidth=1)
ma_frame.pack(fill=BOTH)

# Création d'une Button, non pas dans la fenêtre mais dans la frame
mon_btn = Button(ma_frame, text="Envoyer")
mon_btn.pack(side="top", fill=X)
```

# Les variables Tkinter

- ▷ Les classes **StringVar**, **IntVar**, **DoubleVar** et **BooleanVar** permettent de définir des *variables* Tkinter.
- ▷ Certains widgets (éléments de formulaire) prennent l'une de ces *variables* dans le paramètre **variable**.
- ▷ La méthode **get()** sur l'une des instances de ces *variables* permet d'obtenir la valeur du widget.
  - › Il faudra ensuite faire la conversion en **str** pour l'avoir sous forme de texte.

# Bonne pratique

- ▷ Une bonne pratique consiste à créer des classes dérivées des widgets existants afin de manipuler des éléments graphiques.
- ▷ Cela permet de « personnaliser » les widgets.
- ▷ Cela permettra également de créer des **méthodes sans paramètre** qui pourront être appelées en tant que *command* d'un widget.
  - › Ainsi, les méthodes pourront agir sur les attributs de la classe.

Macademia

# Les évènements

- ▷ Jusqu'ici, nous avions écrit des programmes, impératif ou objet, qui s'exécutaient dans un certain ordre et s'arrêtaient après la dernière instruction.
- ▷ Avec Tkinter, un programme ne s'interrompt pas tant que la fenêtre n'est pas fermée par l'utilisateur.
  - › Le processus d'exécution est modifié : **programmation évènementielle**.
- ▷ Nous allons désormais agir sur notre programme en déclenchant des actions via l'interface graphique.

# Les évènements

- ▷ À chaque widget est attaché par défaut un certain nombre de réponses automatiques à des événements.
- ▷ La boucle événementielle **mainloop()** les prend en charge et les répartit.

Macademia

# Des évènements ... oui mais lesquels ?

- ▷ Un évènement est une entité atomique qui nait de façon **spontanée** et **asynchrone**.
- ▷ Autrement dit, il peut s'agir d'un clic, d'un mouvement de la souris, d'une touche du clavier pressée, d'une seconde qui passe ... etc.
- ▷ Un évènement se traduit par un message qu'un objet envoie à un autre objet.
  - › Notion d'**émetteur** et de **récepteur**.

# Les command

- ▷ Sur les widgets, le paramètre `command` permet de définir l'action à réaliser, généralement au clic sur ce widget.
- ▷ Cette action correspond à une méthode, qui n'a aucun paramètre.
- ▷ On indique le nom de la méthode, sans parenthèse.
- ▷ Le fait de ne pas pouvoir passer de paramètre à la méthode d'action, incite à encapsuler les widgets dans une classe.
  - › Par exemple, une classe héritée de `Frame`.

# Les command

```
ihm.py
```

```
# Création du bouton, avec l'action "envoyer"
btn_go = Button(ma_fenetre, text="Go", command=envoyer)
btn_go.pack()

# Méthode appelée au clic sur le bouton
def envoyer():
    print("Envoi en cours ...")
```

# La méthode *bind()* et l'objet *event*

- ▷ La méthode **bind()** permet « d'attacher » à un widget une action déclenchée selon un évènement.
  - › Typiquement, un appui sur une certaine touche clavier.
- ▷ La méthode **bind()** prend deux paramètres :
  - › Le type d'évènement.
  - › La méthode à appeler.
- ▷ La méthode à appeler doit obligatoirement prendre un paramètre **event**.
  - › Il contient des informations sur l'évènement déclencheur.

# La méthode *bind()* et l'objet *event*

```
ihm.py

contenu = StringVar()

mon_entry = Entry(ma_fenetre, textvariable=contenu, width=15)
mon_entry.pack()
mon_entry.bind('<Return>', envoyer)

def envoyer(event):
    print(str(contenu.get()))
```

# Mixer une *command* et un *bind*

- ▷ **Problème** : les méthodes à passer au paramètre `command` ne doivent compoter aucun paramètre (excepté `self`), et les méthodes appelées par `bind()` doivent prendre un paramètre `event`.
  - › Il est donc *en théorie* impossible d'appeler une même méthode.
- ▷ **Solution** : on peut passer `event` en paramètre optionnel :

```
def envoyer(self, event=None):  
    # corps de la méthode  
    ...
```

# Récapitatif des widgets

Widget	Description
<b>Button</b>	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
<b>Canvas</b>	Un espace pour disposer divers éléments graphiques.
<b>Checkbutton</b>	Une case à cocher qui peut prendre deux états distincts. Un clic sur ce widget provoque le changement d'état.
<b>Entry</b>	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
<b>Frame</b>	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets.
<b>Label</b>	Un texte (ou libelle) quelconque (éventuellement une image).
<b>Listbox</b>	Une liste de choix proposée à l'utilisateur, généralement présentes dans une sorte de boîte.
<b>Menu</b>	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « pop up ».
<b>Menobutton</b>	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
<b>Message</b>	Variante du widget Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille.
<b>Radiobutton</b>	Cliquer sur un bouton radio donne la valeur correspondante à la variable, et vide tous les autres boutons radio.
<b>Scale</b>	Permet de faire varier de manière visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
<b>Scrollbar</b>	Ascenseur ou barre de défilement à utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
<b>Text</b>	Affichage de texte formaté. Permet aussi à l'utilisateur d'édition le texte affiché.
<b>Toplevel</b>	Une fenêtre affichée séparément, au premier plan.

# En résumé

**Dans ce chapitre, nous avons vu ...**

# Merci

## Place aux questions