



Programmation en Python

CESI - INFAL59
Pierre-Loïc Bayart



Plan du cours

1. Découverte du langage
2. La programmation orientée objet avec Python
3. Les bibliothèques en Python
4. Fonctionnalités avancées



Découverte du langage



Plan du chapitre - Découverte du langage

1. Généralités sur Python
2. Les listes, tuples et dictionnaires
3. Structures conditionnelles et boucles
4. Les modules
5. Les exceptions

Chapitre

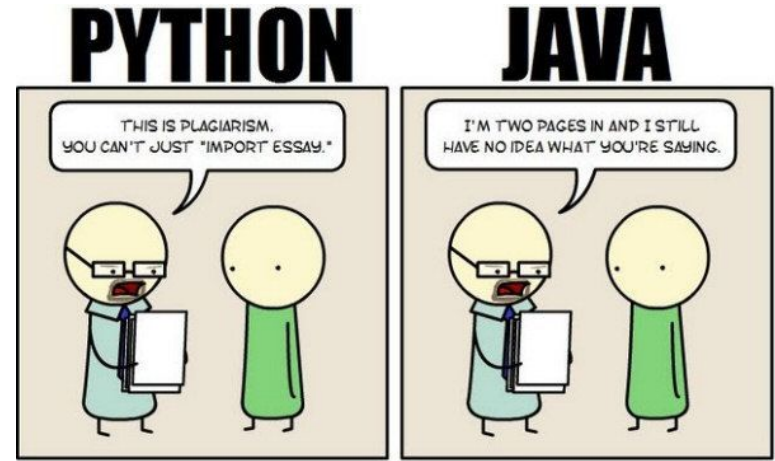
Découverte du langage

Généralités sur Python

Généralités sur Python

Python en quelques mots :

- Langage Interprété
- Typé fort dynamiquement
- Généraliste (script, data science, intelligence artificielle...)
- Open source
- Créé par Guido van Rossum en décembre 1989
- Développement du langage et animation de la communauté réalisés par la [Python Software Foundation \(PSF\)](https://www.python.org/psf/)



Généralités sur Python

Un mot de plus sur les **origines** du nom **Python** : il vient du nom de la troupe d'humoristes anglais **Monty Python Flying Circus** que Guido van Rossum apprécie beaucoup.



Généralités sur Python

Quelques concurrents de Python

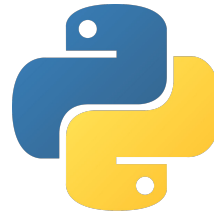


VS



Généralités sur Python

Différents types d'implémentation de Python : **CPython**, Jython, PyPy, IronPython



Généralités sur Python

La version courante de Python est la **3.8**
(en date de juin 2020)

Il existe deux versions majeures de Python : Python 2 et Python 3. Ces deux versions ne sont pas compatibles entre-elles. Depuis le 1er janvier 2020, la version 2 n'est plus mise à jour.

Généralités sur Python

La communauté qui développe le langage Python publie régulièrement des Python Enhancement Proposal (PEP) pour des propositions d'amélioration, d'ajout de fonctionnalités ou de recommandations sur l'usage de Python.

Les trois premières PEP à lire pour débuter en Python :

- [PEP20](#) (The Zen of Python)
- [PEP8](#) (Style Guide for Python Code)
- [PEP257](#) (Docstring Conventions)

Généralités sur Python

Le Zen du Python (PEP 20)
regroupe 19 principes qui
permettent de
comprendre la
philosophie du langage
Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
Namespaces are one honking great idea -- let's do more of those!
```

Généralités sur Python

La [PEP 8](#) fournit des bonnes pratiques pour rédiger du code Python de qualité :

- Pas plus de 79 caractères par ligne de code
- Une ligne par import
- Ordre des imports : librairie standard, librairies tierces, imports locaux
- ...

Des librairies externes permettent de vérifier le respect de la PEP 8 : pep8, pylint, flake8 ...

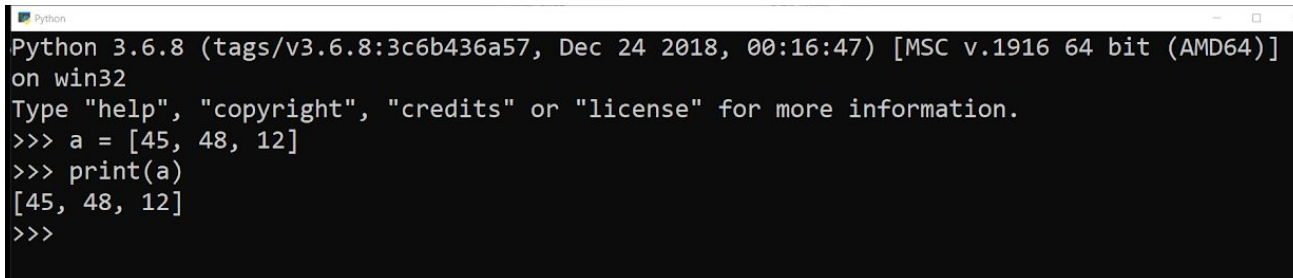
Généralités sur Python

La PEP 257 précise le fonctionnement des docstrings. La chaîne de caractères est ajoutée à l'attribut `__doc__` du module, de la classe, de la fonction ou de la méthode.

Les docstring sont utilisées lorsqu'on demande de l'aide sur un objet (avec *help()*)

Généralités sur Python

L'interpréteur de commandes Python permet de voir le fonctionnement des commandes. Son inconvénient est que les commandes ne sont pas sauvegardées.

A screenshot of a Windows command prompt window titled "Python". The window shows the Python 3.6.8 version information and the architecture (MSC v.1916 64 bit (AMD64)) on a win32 system. It prompts the user to type "help", "copyright", "credits", or "license" for more information. The user has entered a list assignment and a print statement, which has been executed successfully, displaying the list [45, 48, 12].

```
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = [45, 48, 12]
>>> print(a)
[45, 48, 12]
>>>
```

Généralités sur Python

Un fichier de code Python est un fichier texte avec l'extension **.py**

Pour écrire du code Python, plusieurs solutions sont utilisables :

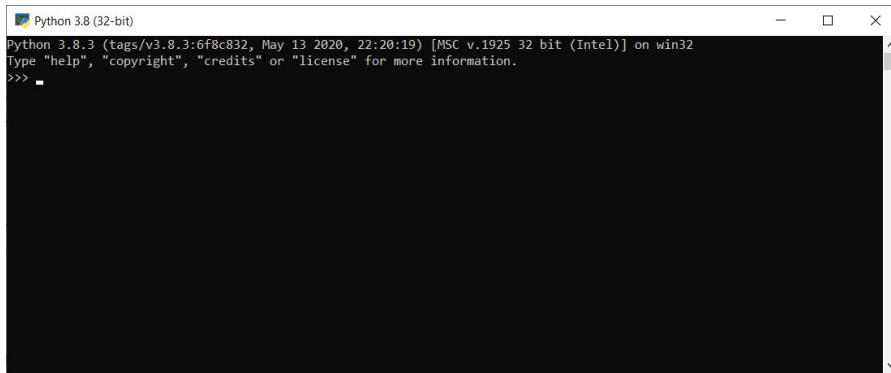
- Éditeur de texte : Vim, Emacs, Sublime Text, ...
- IDE : PyCharm, Visual Studio Code, Eclipse, Spyder (data science), ...
- Notebook (pour la data science) : Jupyter Notebook, Azure Notebooks, Google Colab, ...

Généralités sur Python

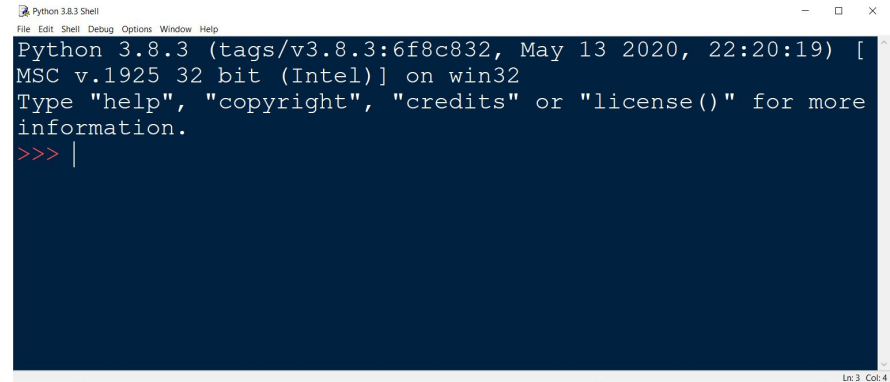
Pour installer Python, il faut l'installer à partir du site officiel :

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

Une fois l'installation finie, on a accès à l'interpréteur Python et à un IDE de base nommé **Idle**.



```
Python 3.8 (32-bit)
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> _
```



```
Python 3.8.3 Shell
File Edit Shell Debug Options Window Help
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [
MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> |
```

Généralités sur Python

Les commentaires en Python sont définis le caractère # (tout ce qui suit ce caractère n'est pas interprété par Python)

```
# Exemple de commentaires  
entier = 4 # Affectation d'une variable  
texte = "# ceci n'est pas un commentaire"
```

La logique des blocs d'instructions est gérée par l'indentation. Elle doit être cohérente pour tout le code (la PEP8 préconise 4 espaces)

Généralités sur Python

En Python, l'**affectation** des variables se fait à l'aide du symbole =

```
>>> var = 4
>>> print(var)
4
>>> var = "test"
>>> print(var)
test
```

Généralités sur Python

La vérification d'**égalité en valeur** s'effectue grâce au symbole **==**. La **différence en valeur** est vérifiée avec le symbole **!=**

```
>>> var = 4
>>> print(var == 4)
True
>>> var = 4
>>> print(var != 4)
False
>>> var = var == 3
>>> print(var)
False
```

Généralités sur Python

La vérification d'**égalité en référence** (même objet) s'effectue grâce au mot-clé ***is***. La **différence en référence** est vérifiée avec le mot-clé ***is not***

```
>>> var_1 = [45, 12]
>>> var_2 = [45, 12]
>>> print(var_1 == var_2)
True
>>> print(var_1 is var_2)
False
```

Généralités sur Python

Les différents calculs possibles nativement en Python : division (/), multiplication (*), addition (+), soustraction (-), division entière (//), modulo (%), puissance (**)

```
>>> result = 4 + 5
>>> print(result)
9
>>> result = 4 * 5
>>> print(result)
20
>>> result = 42 % 5
>>> print(result)
2
```

Généralités sur Python

De nombreux modules de la bibliothèque standard ajoutent des fonctionnalités mathématiques à Python : *math*, *cmath*, *numbers* (classes abstraites), *decimal*, *fractions*, *random*, *statistics*

```
>>> import math
>>> print(math.pi)
3.141592653589793
>>> from fractions import Fraction
>>> print(Fraction(16, -10))
-8/5
```

Généralités sur Python - chaînes de caractères

Les chaînes de caractères en Python sont des éléments **itérables** et **non modifiables**. Elles sont définies par les symboles `"` ou `'` en début et fin de chaîne. Si ces symboles apparaissent à l'intérieur de la chaîne de caractères, il faut les **échapper** à l'aide du symbole `\`

Pour définir des chaînes de caractères sur **plusieurs lignes**, il faut utiliser les symboles `"""` ou `'''` en début et fin de chaîne.

```
>>> texte_1 = 'C\'est 1\'heure'
>>> texte_2 = "C'est 1'heure"
>>> print(texte_1, texte_2)
C'est 1'heure C'est 1'heure
```


Généralités sur Python - chaînes de caractères

Les chaînes de caractères possèdent différentes **méthodes**. On peut par exemple séparer les éléments d'une chaîne de caractères dans une liste grâce à la méthode *str.split()*.

```
>>> texte = "Une chaine de caractères"
>>> print(texte.split())
['Une', 'chaine', 'de', 'caractères']
>>> print(texte.split("c"))
['Une ', 'haine de ', 'ara', 'tères']
```

Généralités sur Python - ressources complémentaires

Ressource texte :

- Site du langage Python : <https://www.python.org/>
- Documentation **officielle** de Python : <https://docs.python.org/fr/3/tutorial/appetite.html>
- Wikipédia sur Python : [https://fr.wikipedia.org/wiki/Python \(langage\)](https://fr.wikipedia.org/wiki/Python_(langage))

Ressource video :

- The Story of Python, by Its Creator, Guido van Rossum :
<https://www.youtube.com/watch?v=J0Aq44Pze-w>

Chapitre

Découverte du langage

Les listes, tuples et dictionnaires

Les listes, tuples et dictionnaires - généralités

	Séquence	Élément simple
Modifiable (mutable)	Listes (list) Dictionnaires (dict) Ensembles (set)	
Non modifiable (immutable)	Chaînes de caractères (str) Tuples (tuple) Ensembles immuables (frozenset)	Entiers (int) Nombres à virgule flottante (float) Booléens (bool)

Les listes, tuples et dictionnaires - list

Les listes sont des structures ordonnées pour stocker des données de différents types. Elles sont muables et itérables

Utilité des listes en Python :

- Regrouper plusieurs données de différents types
- Récupérer des données par leur indice dans la liste (commence à 0)

Inconvénient :

- Peu efficace au niveau de la gestion de la mémoire (stockage de données de différents types)

Les listes, tuples et dictionnaires - list

Il y a deux façons de créer une liste vide :

```
# Création de listes vides  
liste_vide_1 = []  
liste_vide_2 = list()
```

On peut aussi créer une liste directement avec des valeurs :

```
# Création d'une liste avec des valeurs  
liste = [1, 2.3, "rze", True, [23, -5.3, None], False]
```

Les listes, tuples et dictionnaires - list

Les listes peuvent être indicées avec des indices positifs (commençant à 0)

```
# Indichage des listes
>>> liste = [45, 13, 15, 8]
>>> print(liste[0])
45
```

Les listes peuvent être indicées avec des indices négatifs (commençant à -1)

```
# Indichage des listes
>>> liste = [45, 13, 15, 8]
>>> print(liste[-2])
15
```

Les listes, tuples et dictionnaires - list

Les listes imbriquées peuvent aussi être indicées :

```
# Indichage des listes imbriquées
>>> liste = [45, 13, 15, [8, 5, [1, 5]], [4, 8, 12]]
>>> print(liste[3][-1][1])
5
```


Les listes, tuples et dictionnaires - list

Les listes peuvent être découpées (slicing) de différentes manières

```
# Découpage des listes
>>> liste = [45, 13, 15, 8, 48, 12, 10, 13]
>>> print(liste[1:3])
[13, 15]
>>> print(liste[-4:-2])
[48, 12]
>>> print(liste[2:-2])
[15, 8, 48, 12]
>>> print(liste[-5:5])
[8, 48]
>>> print(liste[-5:2])
[]
```

Les listes, tuples et dictionnaires - list

Les listes peuvent être découpées (slicing) avec un pas variable (positif ou négatif)

```
# Découpage des listes
>>> liste = [45, 13, 15, 8, 48, 12, 10, 13]
>>> print(liste[1:6:2])
[13, 8, 12]
>>> print(liste[-6:-2:2])
[15, 48]
>>> print(liste[5:2:-1])
[12, 48, 8]
>>> print(liste[:3:2])
[45, 15]
>>> print(liste[4::2])
[48, 10]
```

Les listes, tuples et dictionnaires - list

Les listes étant modifiables, on peut modifier des éléments de la liste

```
# Modification des éléments d'une liste
>>> liste_1 = [14, 45, 56]
>>> liste_1[1] = "a"
>>> print(liste_1, id(liste_1))
[14, 'a', 56] 19814056
>>> liste_1[1:] = "b", "c"
>>> print(liste_1, id(liste_1))
[14, 'b', 'c'] 19814056
```

Les listes, tuples et dictionnaires - list

On peut connaître le nombre d'éléments dans la liste grâce à la fonction *len()*

```
# Calcul du nombre d'éléments dans la liste
>>> ma_liste = [4, 45, 56]
>>> print(len(ma_liste))
3
```

Les listes, tuples et dictionnaires - list

On peut savoir si un élément est présent dans la liste avec le mot-clé *in*



Cette syntaxe est
très utile pour les
conditions

```
# Vérification de la présence d'un élément dans la liste
>>> ma_liste = ["a", 45, 56]
>>> print("a" in ma_liste)
True
>>> print(-12 in ma_liste)
False
```

Les listes, tuples et dictionnaires - list

Certaines opérations mathématiques (+ et *) fonctionnent avec les listes :

- L'addition : le signe + effectue la concaténation de deux listes. On ne peut pas concaténer une liste avec un entier ou une chaîne de caractères de cette façon. La notation condensée += fonctionne aussi pour les listes

```
# Concaténation de listes
>>> ma_liste_1 = [23, 15, 2]
>>> ma_liste_2 = [-12, -56]
>>> print(ma_liste_1 + ma_liste_2)
[23, 15, 2, -12, -56]
>>> ma_liste_2 += ma_liste_1
>>> print(ma_liste_2)
[-12, -56, 23, 15, 2]
>>> print(ma_liste_1 + 5)
Traceback (most recent call last):
  File "d:code_exemples.py", line 117, in <module>
    print(ma_liste_1 + 5)
TypeError: can only concatenate list (not "int") to list
```

Les listes, tuples et dictionnaires - list

- La multiplication : le signe `*` effectue la concaténation de la même liste plusieurs fois. On ne peut pas multiplier deux listes entre-elles de cette façon. La notation condensée `*=` fonctionne aussi pour les listes

```
# Multiplication d'une liste par un entier
>>> print([5, 2]*3)
[5, 2, 5, 2, 5, 2]
>>> print(2*[-5, "a"])
[-5, 'a', -5, 'a']
>>> print([56, 5]*[45, 12])
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 127, in <module>
    print([56, 5]*[45, 12])
TypeError: can't multiply sequence by non-int of type 'list'
```

Les listes, tuples et dictionnaires - list

Il est possible de comparer deux listes :

- Pour vérifier leur égalité en valeur, on utilise la syntaxe `==`. Pour vérifier si deux listes sont le même objet, on utilise le mot-clé `is`.

```
# Vérification de l'égalité de deux listes
>>> ma_liste_1 = [2, 15]
>>> ma_liste_2 = [2, 15]
>>> print(ma_liste_1 == ma_liste_2)
True
>>> print(ma_liste_1 is ma_liste_2)
False
>>> ma_liste_3 = ma_liste_1
>>> print(ma_liste_3 is ma_liste_1, ma_liste_3 == ma_liste_1)
True True
>>> ma_liste_3 += [-56]
>>> print(ma_liste_1)
[2, 15, -56]
```


Les listes, tuples et dictionnaires - list

- On peut aussi comparer des listes avec des inégalités (>, >=, <, <=). La comparaison s'effectue élément par élément. Il n'est pas possible de comparer des listes avec un mélange d'entiers et de chaînes de caractères

```
# Comparaison de listes
>>> print([2, 3, 45]>[1, 18])
True
>>> print(["c", "g"]>["a", "z"])
True
>>> print(["b", 5]>["a", 1])
True
>>> print(["b", 5]>[1, "a"])
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 155, in <module>
    print(["b", 5]>[1, "a"])
TypeError: '>' not supported between instances of 'str' and 'int'
```

Les listes, tuples et dictionnaires - list

On peut inverser l'ordre des éléments d'une liste à l'aide de la fonction *reversed()* (renvoie un itérateur)

```
# Inversion de l'ordre des éléments d'une liste
>>> ma_liste = [45, 12, 3]
>>> print(list(reversed(ma_liste)))
[3, 12, 45]
```

Les listes, tuples et dictionnaires - list

On peut trier les éléments d'une liste à l'aide de la fonction `sorted()` (renvoie une liste)

```
# Trie des éléments de la liste avec la fonction sorted()
>>> ma_liste = [45, 12, 30, -30, 45, 16]
>>> print(sorted(ma_liste), ma_liste)
[-30, 12, 16, 30, 45, 45] [45, 12, 30, -30, 45, 16]
>>> print(sorted(ma_liste, key=lambda x: str(x)[1]))
[30, 12, -30, 45, 45, 16]
>>> print(sorted(ma_liste, reverse=True))
[45, 45, 30, 16, 12, -30]
```

Les listes, tuples et dictionnaires - list

On peut supprimer des éléments avec le mot-clé *del*

```
# Suppression d'éléments
>>> ma_liste = list(range(1, 11))
>>> print(ma_liste)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> del ma_liste[-1]
>>> print(ma_liste)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del ma_liste[:3]
>>> print(ma_liste)
[4, 5, 6, 7, 8, 9]
>>> del ma_liste
>>> print(ma_liste)
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 178, in <module>
    print(ma_liste)
NameError: name 'ma_liste' is not defined
```

Les listes, tuples et dictionnaires - list

Les listes possèdent différentes méthodes. On peut les regrouper en 5 grands types :

- Méthodes d'ajout : *list.append(x)*, *list.extend(iterable)*, *list.insert(i, x)*
- Méthodes de suppression : *list.remove(x)*, *list.pop([i])*, *list.clear()*
- Méthodes d'information : *list.index(x[, start[, end]])*, *list.count(x)*
- Méthodes de modification de l'ordre : *list.sort(key=None, reverse=False)*, *list.reverse()*
- Méthode de copie : *list.copy()*



La fonction native `dir()` permet de connaître tous les attributs et méthodes d'un objet

```
>>> print(dir(list))
['_add_', '_class_', '_contains_', '_delattr_', '_delitem_', '_dir_',
'_doc_', '_eq_', '_format_', '_ge_', '_getattribute_', '_getitem_',
'_gt_', '_hash_', '_iadd_', '_imul_', '_init_', '_init_subclass_',
'_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_',
'_reduce_', '_reduce_ex_', '_repr_', '_reversed_', '_rmul_',
'_setattr_', '_setitem_', '_sizeof_', '_str_', '_subclasshook_',
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 3 méthodes d'ajout : *list.append(x)*, *list.extend(iterable)*, *list.insert(i, x)*

- *list.append(x)* : cette méthode ajoute un élément à la fin de la liste

```
# Ajout d'un élément à la fin de la liste
>>> ma_liste = [4, 45, 56]
>>> ma_liste.append(12)
>>> print(ma_liste)
[4, 45, 56, 12]
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 3 méthodes d'ajout : *list.append(x)*, *list.extend(iterable)*, *list.insert(i, x)*

- *list.extend(iterable)* : cette méthode ajoute un groupe de plusieurs éléments (itérable) à la fin de la liste



Fonctionne
comme la
concaténation
avec +

```
# Ajout d'un groupe d'éléments à la fin de la liste
>>> ma_liste_1 = [4, 45, 56]
>>> ma_liste_2 = [2, 89]
>>> ma_liste_1.extend(ma_liste_2)
>>> print(ma_liste_1)
[4, 45, 56, 2, 89]
>>> print(ma_liste_2)
[2, 89]
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 3 méthodes d'ajout : *list.append(x)*, *list.extend(iterable)*, *list.insert(i, x)*

- *list.insert(i, x)* : insère un élément x à la position i dans la liste (position après insertion)

```
# Ajout d'un élément x à la position i
>>> ma_liste = [1, False, 45]
>>> ma_liste.insert(1, None)
>>> print(ma_liste)
[1, None, False, 45]
```


Les listes, tuples et dictionnaires - list

Les listes possèdent 3 méthodes de suppression : *list.remove(x)*, *list.pop([i])*, *list.clear()*

- *list.remove(x)* : supprime l'élément x de la liste




Si la valeur
n'existe pas,
Python lève une
ValueError

```
# Suppression de l'élément x
>>> ma_liste = [45, 56, 2]
>>> ma_liste.remove(56)
>>> print(ma_liste)
[45, 2]
>>> ma_liste.remove(12)
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 188, in <module>
    ma_liste.remove(12)
ValueError: list.remove(x): x not in list
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 3 méthodes de suppression : *list.remove(x)*, *list.pop([i])*, *list.clear()*

- *list.pop([i])* : supprime l'élément en position i dans le liste et renvoie sa valeur



A la différence de *del*, *list.pop([i])* renvoie la valeur supprimée

```
# Suppression de l'élément en position i
>>> ma_liste = [5, 45, 59, -23]
>>> print(ma_liste.pop(1))
45
>>> print(ma_liste)
[5, 59, -23]
>>> print(ma_liste.pop())
-23
>>> print(ma_liste.pop(12))
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 201, in <module>
    print(ma_liste.pop(12))
IndexError: pop index out of range
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 3 méthodes de suppression : *list.remove(x)*, *list.pop([i])*, *list.clear()*

- *list.clear()* : permet de supprimer tous les éléments d'une liste



La syntaxe
ma_liste.clear()
est équivalente
à *del ma_liste[:]*,
ma_liste[:] = [] et
*ma_liste *= 0*

```
# Suppression de tous les éléments de la liste
>>> ma_liste = [45, 12, -3]
>>> print(ma_liste, id(ma_liste))
[45, 12, -3] 51153480
>>> ma_liste.clear()
>>> print(ma_liste, id(ma_liste))
[] 51153480
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 2 méthodes d'information sur les éléments : *list.index(x[, start[, end]])*, *list.count(x)*

- *list.index(x[, start[, end]])* : retourne l'index de la première occurrence de l'élément x (entre les éléments à l'indice *start* et *end* s'ils sont précisés)

```
# Index de la première occurrence
>>> ma_liste = [45, 12, 13, 56, 12, 13]
>>> print(ma_liste.index(56), ma_liste.index(56, 2), ma_liste.index(56, 2, 4))
3 3 3
>>> print(ma_liste.index(56, 4, 5))
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 220, in <module>
    print(ma_liste.index(56, 4, 5))
ValueError: 56 is not in list
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 2 méthodes d'information sur les éléments : *list.index(x[, start[, end]])*, *list.count(x)*

- *list.count(x)* : retourne le nombre d'occurrences de l'élément x (entre les éléments à l'indice *start* et *end* s'ils sont précisés)


```
# Compte le nombre d'occurrences dans la liste
ma_liste = [45, 12, 13, 56, 12, 13]
print(ma_liste.count(12), ma_liste.count(56),
      ma_liste.count(55))
2 1 0
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 2 méthodes de modification de l'ordre des éléments :

list.sort(key=None, reverse=False), list.reverse()

- *list.sort(key=None, reverse=False)* : trie en place les éléments de la liste par ordre croissant



Il est impossible de trier des données qui ne sont pas comparables (*int* et *None* par exemple)

```
# Trie les éléments de la liste
>>> ma_liste = [45, 12, 18, 56, 12, 13]
>>> ma_liste.sort()
>>> print(ma_liste)
[12, 12, 13, 18, 45, 56]
>>> ma_liste.sort(reverse=True)
>>> print(ma_liste)
[56, 45, 18, 13, 12, 12]
>>> ma_liste.sort(key=lambda x: str(x)[1])
>>> print(ma_liste)
[12, 12, 13, 45, 56, 18]
```

Les listes, tuples et dictionnaires - list

Les listes possèdent 2 méthodes de modification de l'ordre des éléments :
list.sort(key=None, reverse=False), *list.reverse()*

- *list.reverse()* : inverse l'ordre des éléments de la liste en place (à la différence de la fonction *reversed()* qui renvoie une nouvelle liste)

```
# Inverse l'ordre des éléments de la liste
>>> ma_liste = [45, 12, 18, 56, 12, 13]
>>> ma_liste.reverse()
>>> print(ma_liste)
[13, 12, 56, 18, 12, 45]
```

Les listes, tuples et dictionnaires - list

Les listes possèdent une méthode de copie :

- *list.copy()* : renvoie une copie de la liste



La syntaxe
`m = ma_liste.copy()`
est équivalente à
`m = ma_liste[:]`

```
# Copie de la liste
ma_liste_1 = [45, 12, 18, 56, 12, 13]
ma_liste_2 = ma_liste_1.copy()
ma_liste_1.pop()
print(ma_liste_1, ma_liste_2)
[45, 12, 18, 56, 12] [45, 12, 18, 56, 12, 13]
```


Les listes, tuples et dictionnaires - tuple

Les tuples sont des structures souples pour stocker des données de différents types comme les listes.

A la différence des listes, ils sont **non modifiables**. Ils possèdent donc moins de méthodes que les listes. Ils utilisent aussi moins de mémoire que les listes.

Le découpage (slicing) et l'indexage des tuples fonctionnent comme pour les listes. Toutes les fonctions ou les syntaxes qui ne modifient pas les listes fonctionnent aussi sur les tuples (*len()*, *reversed()*, *sorted()*, *in*, *+*, ***, *>*, *>=*, *<*, *<=*, *==*, *is*)

Les listes, tuples et dictionnaires - tuple

Il existe différentes syntaxes pour créer des tuples :

```
# Création de tuples
>>> tuple_1 = (1, 56, 2)
>>> tuple_2 = (23,)
>>> tuple_3 = 15, 56, "zr"
>>> tuple_4 = tuple([45, None, "reze"])
>>> print(type(tuple_1), type(tuple_2), type(tuple_3), type(tuple_4))
<class 'tuple'> <class 'tuple'> <class 'tuple'> <class 'tuple'>
>>> pas_tuple = (5)
>>> print(type(pas_tuple))
<class 'int'>
```

Les listes, tuples et dictionnaires - tuple

Comme les tuples sont **immuables**, ils ne possèdent que deux méthodes (qui fonctionnent de la même façon que pour les listes) :

- `tuple.count(x)` : retourne le nombre d'occurrences de la valeur x dans le tuple
- `tuple.index(x[, start[, end]])` : retourne l'indice de la première occurrence de la valeur x dans le tuple et lève une exception si la valeur n'est pas trouvée (start et end permettent de rechercher dans une sous-partie)



La fonction native `dir()` permet de connaître tous les attributs et méthodes d'un objet

```
>>> print(dir(tuple))
['_add_', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'count', 'index']
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires sont des objets modifiables qui peuvent contenir n'importe quel type d'objet comme valeur (par contre, les clés doivent être hashables). A la différence des listes, les valeurs sont récupérées grâce à des clés et non pas à des indices.

Utilité des dictionnaires en Python :

- Rapidité pour trouver une valeur à partir d'une clé
- Peut contenir tout type d'objets comme valeur

Inconvénient :

- Pas de notion d'indices comme avec les listes et les tuples
- Les clés doivent être hashables

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent différentes méthodes. On peut les regrouper en 5 grands types :

- Méthodes de modification : *dict.setdefault(key[, default_value]), dict.update([other])*
- Méthodes de suppression : *dict.clear(), dict.pop(key[, default]), dict.popitem()*
- Méthodes de récupération de données : *dict.get(key, default = None), dict.items(), dict.keys(), dict.values()*
- Méthode de création : *dict.fromkeys(keys, value = None)*
- Méthode de copie : *dict.copy()*

```
print(dir(dict))
['_class_', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
 'setdefault', 'update', 'values']
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 2 méthodes de modification : *dict.setdefault(key[, default_value])*, *dict.update([other])*

- *dict.setdefault(key[, default_value])* : renvoie la valeur de clé si la clé existe sinon la clé est ajoutée avec comme valeur *default_value* (si spécifiée sinon *None*)

```
>>> dico = {"a":1, "b":2}
>>> print(dico.setdefault("b"), dico)
2 {'a': 1, 'b': 2}
>>> print(dico.setdefault("c"), dico)
None {'a': 1, 'b': 2, 'c': None}
>>> print(dico.setdefault("d", 4), dico)
4 {'a': 1, 'b': 2, 'c': None, 'd': 4}
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 2 méthodes de modification : *dict.setdefault(key[, default_value])*, *dict.update([other])*

- *dict.update([other])* : si la clé du dictionnaire *other* est présent dans le dictionnaire d'origine, la valeur est mise à jour sinon la clé et la valeur sont ajoutées au dictionnaire d'origine

```
>>> dico_1 = {"a":1, "b":2}
>>> dico_2 = {"b":3}
>>> dico_1.update(dico_2)
>>> print(dico_1, dico_2)
{'a': 1, 'b': 3} {'b': 3}
>>> dico_3 = {"c":4}
>>> dico_1.update(dico_3)
>>> print(dico_1, dico_3)
{'a': 1, 'b': 3, 'c': 4} {'c': 4}
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 3 méthodes de suppression : *dict.clear()*, *dict.pop(key[, default])*, *dict.popitem()*

- *dict.clear()* : supprime tous les éléments du dictionnaire

```
>>> dico_1 = {"a":1, "b":2}
>>> print(dico_1, id(dico_1))
{'a': 1, 'b': 2} 2352802644688
>>> dico_1.clear()
>>> print(dico_1, id(dico_1))
{} 2352802644688
```


Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 3 méthodes de suppression : *dict.clear()*, *dict.pop(key[, default])*, *dict.popitem()*

- *dict.pop(key[, default])* : supprime l'élément qui correspond à la clé passée en paramètre et renvoie sa valeur. Si la clé n'existe pas, la valeur *default* est renvoyée (si elle est spécifiée, sinon *KeyError*)

```
>>> dico_1 = {"a":1, "b":2}
>>> print(dico_1.pop("b"), dico_1)
2 {'a': 1}
>>> print(dico_1.pop("b", "Clé non présente"), dico_1)
Clé non présente {'a': 1}
>>> print(dico_1.pop("b"), dico_1)
Traceback (most recent call last):
  File "d:/module.py", line 10, in <module>
    print(dico_1.pop("b"), dico_1)
KeyError: 'b'
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 3 méthodes de suppression : *dict.clear()*, *dict.pop(key[, default])*, *dict.popitem()*

- *dict.popitem()* : supprime le dernier élément ajouté dans le dictionnaire (pour les versions de Python antérieures à 3.7, la suppression est effectuée au hasard)

```
>>> dico_1 = {"a":1, "b":2, "c":3}
>>> value = dico_1.popitem()
>>> print(dico_1, value)
{'a': 1, 'b': 2} ('c', 3)
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 4 méthodes de récupération de données :
dict.get(key, default = None), *dict.items()*, *dict.keys()*, *dict.values()*

- *dict.get(key, default = None)* : retourne la valeur de la clé *key* si elle existe sinon retourne la valeur *default*

```
>>> dico_1 = {"a":1, "b":2, "c":3}
>>> print(dico_1.get("b"))
2
>>> print(dico_1.get("d"))
None
>>> print(dico_1.get("e", "Pas de clé e"))
Pas de clé e
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 4 méthodes de récupération de données :
dict.get(key, default = None), dict.items(), dict.keys(), dict.values()

- *dict.items()* : renvoie un objet contenant les clés et la valeurs du dictionnaire

```
>>> dico_1 = {"a":1, "b":2, "c":3}
>>> print(dico_1.items())
dict_items([('a', 1), ('b', 2), ('c', 3)])
>>> for key, val in dico_1.items():
...     print(f"{key} : {val}")
a : 1
b : 2
c : 3
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 4 méthodes de récupération de données :
dict.get(key, default = None), dict.items(), dict.keys(), dict.values()

- *dict.keys()* : renvoie les clés du dictionnaire

```
>>> dico_1 = {"a":1, "b":2, "c":3}
>>> print(dico_1.keys())
dict_keys(['a', 'b', 'c'])
>>> for key in dico_1.keys():
...     print(key)
a
b
c
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent 4 méthodes de récupération de données :
dict.get(key, default = None), dict.items(), dict.keys(), dict.values()

- *dict.values()* : renvoie les valeurs du dictionnaire

```
>>> dico_1 = {"a":1, "b":2, "c":3}
>>> print(dico_1.values())
dict_values([1, 2, 3])
>>> for val in dico_1.values():
...     print(val)
1
2
3
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent une méthode de création : *dict.fromkeys(keys, value = None)*

- *dict.fromkeys(keys, value = None)* : crée un dictionnaire à partir de la séquence de clés passée en paramètre

```
>>> keys = ("a", "b", "c")
>>> dico = dict.fromkeys(keys)
>>> print(dico)
{'a': None, 'b': None, 'c': None}
>>> dico = dict.fromkeys(keys, 100)
>>> print(dico)
{'a': 100, 'b': 100, 'c': 100}
```

Les listes, tuples et dictionnaires - dict

Les dictionnaires possèdent une méthode de copie : *dict.copy()*

- *dict.copy()* : crée une copie en dur des éléments du dictionnaire

```
>>> dico_1 = {"a":1, "b":2, "c":3}
>>> dico_2 = dico_1
>>> dico_3 = dico_1.copy()
>>> dico_1["d"] = 4
>>> print(dico_1, dico_2, dico_3)
{'a': 1, 'b': 2, 'c': 3, 'd': 4} {'a': 1, 'b': 2, 'c': 3, 'd': 4} {'a': 1, 'b': 2, 'c': 3}
```


Les listes, tuples et dictionnaires - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/tutorial/datastructures.html>

Ressource video :

- Socratica - Python Lists | | Python Tutorial | | Learn Python Programming :
<https://www.youtube.com/watch?v=ohCDWZgNIU0>
- Socratica - Python Dictionaries | | Python Tutorial | | Learn Python Programming :
<https://www.youtube.com/watch?v=XCcpzWs-CI4>
- Socratica - Python Tuples | | Python Tutorial | | Learn Python Programming :
<https://www.youtube.com/watch?v=NI26dqhs2Rk>

Chapitre

Découverte du langage

Structures conditionnelles et boucles

Structures conditionnelles et boucles - for

A la différence de beaucoup d'autres langages informatiques, Python itère sur les éléments d'une séquence pour une boucle *for* (pas d'indice nécessaire)

```
>>> ma_liste = [45, 48, 12, 13]
>>> for number in ma_liste:
...     print(number)
45
48
12
13
```

Structures conditionnelles et boucles - for

Si les indices sont nécessaires à la logique de la boucle, on peut utiliser la fonction native de Python *enumerate()*

```
>>> ma_liste = [45, 48, 12, 13]
>>> for i, number in enumerate(ma_liste):
...     print(i, number)
0 45
1 48
2 12
3 13
```



L'indigage des listes en Python commence à 0

Structures conditionnelles et boucles - while

Comme dans d'autres langages, la boucle *while* en Python permet de répéter un bout de code tant qu'une condition est respectée

```
# Boucle while en Python
>>> i = 0
>>> while i<5:
...     print(i)
...     i += 1
0
1
2
3
4
```

Structures conditionnelles et boucles - if

Les structures conditionnelles sont construites avec les mots réservés *if*, *elif* et *else* en Python (le bloc conditionnel doit au moins contenir le mot réservé *if*)



Il n'existe pas de *Switch* en Python (à la différence de Java ou C++). On peut le construire à partir d'un dictionnaire ou de plusieurs *elif*

```
# Condition
>>> a = 5
>>> if 0<=a<=2:
...     print("a compris entre 0 et 2")
... elif 2<=a<=5:
...     print("a compris entre 2 et 5")
... else:
...     print("a supérieur à 5")
a compris entre 2 et 5
```

Structures conditionnelles et boucles - break

Pour arrêter une boucle *for* ou *while* avant la fin, on peut utiliser le mot-clé *break*



En cas de boucles imbriquées, seule la boucle à laquelle appartient le mot-clé *break* est arrêtée

```
# Boucle while en Python avec break
>>> i = 0
>>> while True:
...     print(i)
...     i += 1
...     if i>2:
...         break
0
1
2
```

Structures conditionnelles et boucles - continue

Le mot-clé *continue* permet d'arrêter l'itération en cours et de continuer à la suivante dans une boucle *for* ou *while*

```
# Boucle for en Python avec continue
>>> for i in range(5):
...     print(f"Début itération {i}")
...     if 1<=i<=3:
...         continue
...     print(f"Fin itération {i}")
Début itération 0
Fin itération 0
Début itération 1
Début itération 2
Début itération 3
Début itération 4
Fin itération 4
```


Structures conditionnelles et boucles - pass

Le mot-clé *pass* ne fait rien. Il est utilisé lorsqu'une instruction est nécessaire pour que la syntaxe soit correcte

```
# Boucle for qui ne fait rien
>>> for i in range(10):
File "d:/plbayart/code_exemples.py", line 177
    for i in range(10):
    ^
IndentationError: expected an indented block
>>> for i in range(10):
...     pass
```

Structures conditionnelles et boucles - compréhension

Les compréhensions de listes permettent de créer des listes de manière très concise

```
# Compréhension de liste simple
>>> liste_comp_1 = [i for i in range(10)]
>>> print(liste_comp_1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# Équivalent de la compréhension de liste avec une boucle for
>>> liste_comp_2 = []
>>> for i in range(10):
...     liste_comp_2.append(i)
>>> print(liste_comp_2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Structures conditionnelles et boucles - compréhension

On peut aussi ajouter des conditions dans les compréhensions de listes pour sélectionner des valeurs

```
# Compréhension de listes avec condition
>>> liste_comp_1 = [i for i in range(10) if i%2==0]
>>> print(liste_comp_1)
[0, 2, 4, 6, 8]
```

```
# Équivalent de la compréhension de liste avec une boucle for
>>> liste_comp_2 = []
>>> for i in range(10):
...     if i%2==0:
...         liste_comp_2.append(i)
>>> print(liste_comp_2)
[0, 2, 4, 6, 8]
```

Structures conditionnelles et boucles - compréhension

On peut aussi ajouter des conditions dans les compréhensions de listes pour avoir des valeurs différentes suivant les conditions

```
# Compréhension de listes avec condition
>>> liste_comp_1 = [i if i%2==0 else "impair" for i in range(10) ]
>>> print(liste_comp_1)
[0, 'impair', 2, 'impair', 4, 'impair', 6, 'impair', 8, 'impair']
```

```
# Équivalent de la compréhension de liste avec une boucle for
>>> liste_comp_2 = []
>>> for i in range(10):
...     if i%2==0:
...         liste_comp_2.append(i)
...     else:
...         liste_comp_2.append( "impair")
>>> print(liste_comp_2)
[0, 'impair', 2, 'impair', 4, 'impair', 6, 'impair', 8, 'impair']
```

Structures conditionnelles et boucles - compréhension

On peut créer des compréhensions de listes imbriquées

```
# Compréhension de listes imbriquées (exemple avec la
transposée d'une matrice)
>>> matrice = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> liste_comp_1 = [[ligne[i] for ligne in matrice] for i
in range(4)]
>>> print(liste_comp_1)
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
# Équivalent de la compréhension de liste avec deux boucles for
>>> liste_comp_2 = []
>>> for i in range(4):
...     ligne_temp = []
...     for ligne in matrice:
...         ligne_temp.append(ligne[i])
...     liste_comp_2.append(ligne_temp)
>>> print(liste_comp_2)
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Structures conditionnelles et boucles - compréhension

Il est possible d'écrire des compréhensions de listes, de dictionnaires, d'ensembles (set) et de générateurs

```
# Compréhension de listes
>>> liste_comp = [i for i in range(10) if i%2==0]
>>> print(liste_comp)
[0, 2, 4, 6, 8]
# Compréhension de dictionnaires
>>> dict_comp = {i: "pair" if i%2==0 else "impair" for i in range(5)}
>>> print(dict_comp)
{0: 'pair', 1: 'impair', 2: 'pair', 3: 'impair', 4: 'pair'}
# Compréhension d'ensembles
>>> set_comp = {"pair" if i%2==0 else "impair" for i in range(10)}
>>> print(set_comp)
{'pair', 'impair'}
# Compréhension de générateurs
>>> gen_comp = (i if i%2==0 else "impair" for i in range(10))
>>> print(list(gen_comp))
[0, 'impair', 2, 'impair', 4, 'impair', 6, 'impair', 8, 'impair']
```

Structures conditionnelles et boucles - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python :
 - <https://docs.python.org/3/tutorial/controlflow.html>
 - https://docs.python.org/3/reference/compound_stmts.html#if
- Trey Hunner - List Comprehensions and Generator Expressions :
<https://pycon2018.trey.io/index.html>

Ressource video :

- Socratica - If, Then, Else in Python || Python Tutorial || Learn Python Programming (ENG - 06:52) : https://www.youtube.com/watch?v=f4KOjWS_KZs
- Socratica - List Comprehension || Python Tutorial || Learn Python Programming (ENG - 07:42) : <https://www.youtube.com/watch?v=AhSvKGTh28Q>

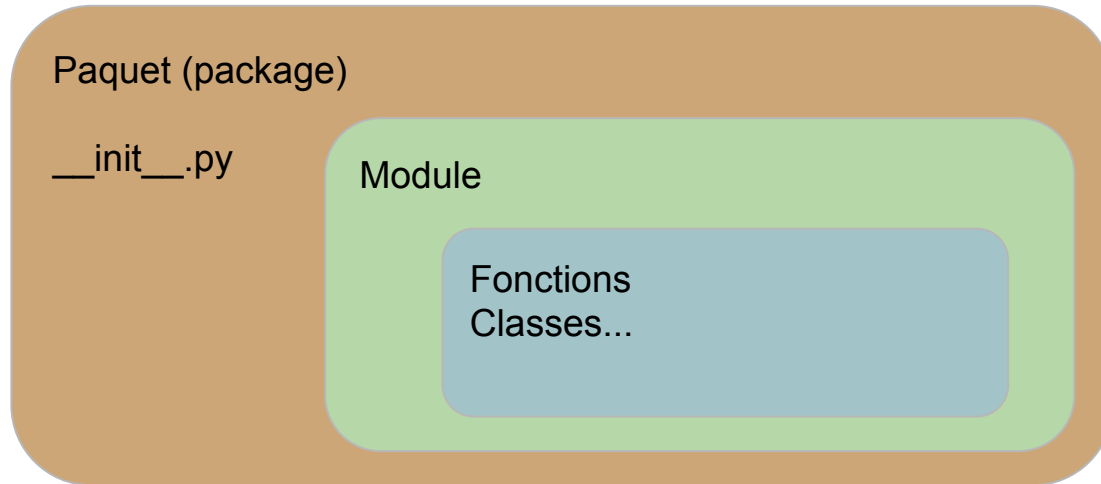
Chapitre

Découverte du langage

Les modules

Les modules - généralités

Pour organiser du code Python, on peut créer des paquets (dossier contenant des fichiers python et un fichier `__init__.py`) et des modules (fichier python avec l'extension `".py"`)



Arborescence :

```
paquet_1
    __init__.py
    module1.py
    module2.py
Paquet_2
    __init__.py
    module3.py
```

Les modules - imports

Pour utiliser des modules ou des paquets dans du code Python, il faut d'abord les importer. Pour cela, on utilise le mot-clé *import* suivi du nom du module ou du paquet. Les imports se font de préférence en tout début de fichier.



La PEP8 préconise d'importer d'abord les modules de la librairie standard, puis ceux des librairies externes et enfin ceux internes liés au projet

```
import math
import numpy as np
import ma_librairie
```

Les modules - imports

Il est possible de n'importer qu'une partie d'un module. Pour cela, on utilise la syntaxe suivante : *from* <<module/paquet>> *import* <<élément à importer>>



Il est déconseillé d'utiliser la syntaxe *from* <<module/paquet>> *import* * car il y a un risque d'écraser des fonctions déjà existantes

```
>>> from math import pi, sin
>>> print(pi)
3.141592653589793
```

Les modules - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/tutorial/modules.html>

Ressource video :

- Corey Schafer - Python Tutorial for Beginners 9: Import Modules and Exploring The Standard Library (ENG - 21:56) : <https://www.youtube.com/watch?v=CqvZ3vGoGs0&t=190s>

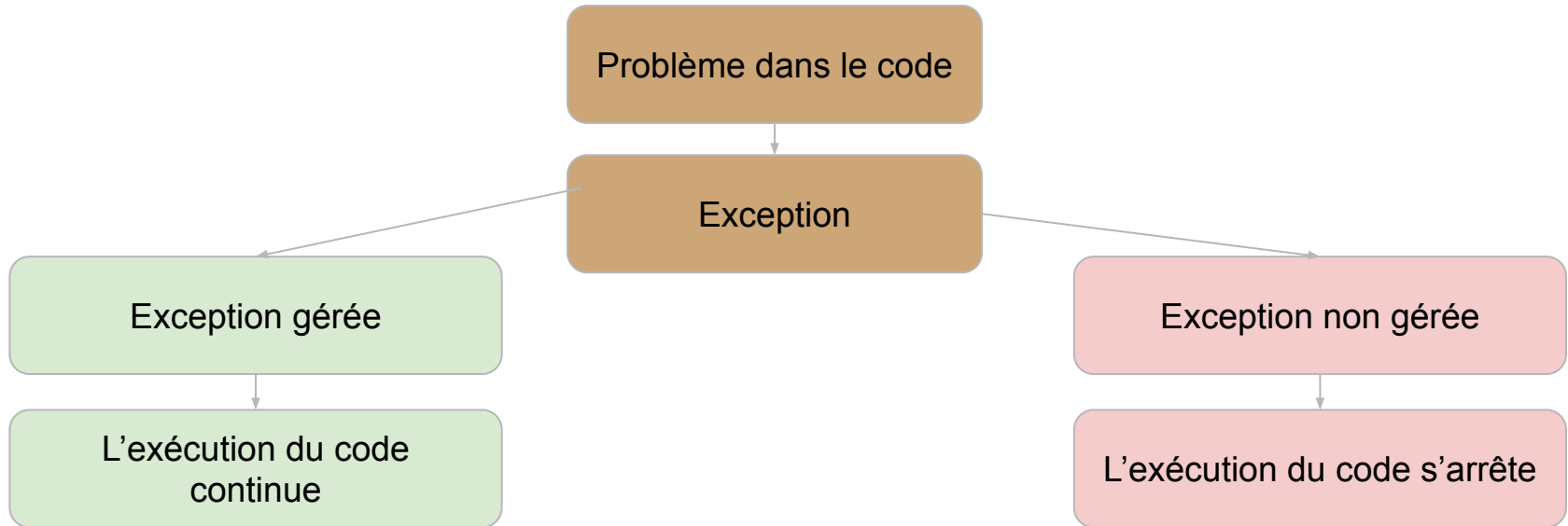
Chapitre

Découverte du langage

Les exceptions

Les exceptions - généralités

Lorsque l'interpréteur Python rencontre une ligne de code qu'il ne peut pas interpréter correctement (nom de variable avec une erreur, division par zéro...), il lève une exception : c'est-à-dire qu'il indique qu'il y a un problème. Ensuite, il y a deux cas de figure : soit l'exception est gérée soit elle ne l'est pas (dans ce cas l'interpréteur s'arrête).



Les exceptions - types d'exceptions

Il existe deux grands types d'erreurs en Python :

- Les erreurs de syntaxe

Elles sont détectées lors de la première étape (le **parsing** du code en bytecode) avant l'exécution du code. Elles doivent être corrigées ; elle ne peuvent pas être gérées

- Les exceptions

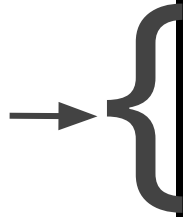
Elles sont détectées lors de l'exécution du code. Il en existe différents types. Elle peut être gérée (pour ne pas arrêter l'exécution du code)

Toutes les exceptions en Python sont des classes qui sont héritées de **BaseException** ou d'une de ces classes enfants.

Les exceptions - messages d'erreur

Les messages d'erreur en Python contiennent plusieurs parties :

Ensemble des parties du code où l'erreur s'est propagée



```
Traceback (most recent call last):  
  File "d:\module1.py", line 9, in <module>  
    print_hello(300)  
  File "d:\module1.py", line 4, in print_hello  
    if module.pair(nombre):  
  File "d:\module.py", line 6, in pair  
    raise ValueError("Le nombre est trop grand")  
ValueError: Le nombre est trop grand
```

Endroit du code qui a appelé le code qui a généré l'erreur



Endroit du code où l'erreur est générée



Nom de l'exception générée



Détails sur l'exception générée



Les exceptions - erreurs de syntaxe

Pour les erreurs de syntaxe, Python indique à l'aide du symbole `^` où se situe l'erreur. Cette indication est approximative : il faut regarder aussi les lignes qui précèdent ou qui suivent l'erreur. Dans l'exemple, l'erreur de syntaxe vient de la ligne 7 où la virgule est manquante.

```
5 | dico = {
6 |     "a" : 1,
7 |     "b" : 2
8 |     "c" : 3
9 | }
File "d:/module.py", line 8
    "c" : 3
      ^
SyntaxError: invalid syntax
```

Hierarchie de classes
de **SyntaxError** :

```
BaseException
+-- Exception
+-- SyntaxError
```

Les exceptions - erreurs de syntaxe

Python ne permet pas de mélanger les niveaux d'indentation : ils doivent toujours correspondre à un même nombre d'espaces ou de tabulations (mais pas un mélange des deux). Dans ce cas, Python renvoie une **IndentationError**.

```
5 | for i in range(3):
6 |     print(i)
7 |     print(i+1)
File "d:/module.py", line 7
    print(i+1)
        ^
IndentationError: unindent does not match any outer
indentation level
```

Hierarchie de classes
de **IndentationError** :

```
BaseException
+-- Exception
    +-- SyntaxError
        +-- IndentationError
            +-- TabError
```

Les exceptions - les exceptions

Il existe différents types d'exceptions en Python. Voici les plus courantes lorsque vous débutez en Python :

- NameError
- TypeError
- ValueError
- IndexError
- KeyError
- AssertionError
- ZeroDivisionError
- FileNotFoundError
- ModuleNotFoundError



Les exceptions - NameError

Si vous essayez d'appeler une variable que vous n'avez définie au préalable, Python renvoie une exception **NameError** :

```
6 | var = 2
7 | print(var_1)
Traceback (most recent call last):
  File "d:/module.py", line 7, in <module>
    print(var_1)
NameError: name 'var_1' is not defined
```

Hierarchie de classes
de **NameError** :

```
BaseException
+-- Exception
    +-- NameError
        +-- UnboundLocalError
```



Les exceptions - UnboundLocalError

Si vous essayez d'appeler une variable qui n'existe pas dans l'environnement local, Python renvoie une exception **UnboundLocalError** :

```
ma_variable = 10
def ma_fonction():
    ma_variable += 2
    return ma_variable
print(ma_fonction())
```

Traceback (most recent call last):

File "d:/module.py", line 10, in <module>
 print(ma_fonction())

File "d:/module.py", line 8, in ma_fonction
 ma_variable += 2

UnboundLocalError: local variable 'ma_variable' referenced
before assignment

Hierarchie de classes de
UnboundLocalError :

```
BaseException
+-- Exception
+-- NameError
+-- UnboundLocalError
```



On peut résoudre
cette erreur en
utilisant le mot-clé
global pour la variable

Les exceptions - TypeError

Si vous essayez d'utiliser une opération ou une fonction avec un type d'objet qui n'est pas autorisé, Python renvoie une exception **TypeError** :

```
>>> print(3 + "a")
Traceback (most recent call last):
  File "d:/module.py", line 7, in <module>
    print(3 + "a")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Hierarchie de classes de
TypeError :

```
BaseException
+-- Exception
+-- TypeError
```



On peut résoudre
cette erreur, on peut
caster un des deux
éléments de l'addition

```
>>> print(str(3) + "a")
3a
>>> print(3 + int("a", 16))
13
```



Les exceptions - ValueError

Si vous essayez d'utiliser une opération ou une fonction avec le bon type mais avec une valeur inappropriée, Python renvoie une exception **ValueError** :

```
>>> print(int("b"))
Traceback (most recent call last):
  File "d:/module.py", line 7, in <module>
    print(int("b"))
ValueError: invalid literal for int() with base 10: 'b'

>>> print(int("b", 12))
11

>>> print(int(None))
Traceback (most recent call last):
  File "d:/module.py", line 14, in <module>
    print(int(None))
TypeError: int() argument must be a string, a bytes-like object
or a number, not 'NoneType'
```

Hierarchie de classes
de **ValueError** :

```
BaseException
+-- Exception
+-- ValueError
+-- UnicodeError
+-- UnicodeDecodeError
+-- UnicodeEncodeError
+-- UnicodeTranslateError
```


Les exceptions - IndexError

Si vous appelez un indice qui n'existe pas dans une liste, Python renvoie une exception **IndexError** :

```
>>> ma_liste = [2, "a", 56]
>>> print(ma_liste[5])
Traceback (most recent call last):
  File "d:/module.py", line 7, in <module>
    print(ma_liste[5])
IndexError: list index out of range
```

Hierarchie de classes
de **IndexError** :

```
BaseException
+-- Exception
    +-- LookupError
        +-- IndexError
        +-- KeyError
```




Les exceptions - KeyError

Si vous appelez une clé qui n'existe pas dans un dictionnaire, Python renvoie une exception **KeyError** :

```
>>> dico = {  
...     "a" : 2,  
...     "b" : 5,  
...     "c" : 7,  
... }  
>>> print(dico["d"])  
Traceback (most recent call last):  
  File "d:/module.py", line 11, in <module>  
    print(dico["d"])  
KeyError: 'd'
```

Hierarchie de classes
de **KeyError** :

```
BaseException  
+-- Exception  
    +-- LookupError  
        +-- IndexError  
            +-- KeyError
```



Les exceptions - AssertionError

Python permet de vérifier que certaines conditions sont respectées en particulier dans le cadre de **tests** ou de **débuggage**. On utilise dans ce cas le mot-clé ***assert***

```
>>> def multiplication(a,b):  
...     return a*b+1  
  
>>> assert multiplication(-2, 4)==-8  
Traceback (most recent call last):  
  File "d:/code_exemples.py", line 272, in <module>  
    assert multiplication(-2, 4)==-8  
AssertionError
```

Hierarchie de classes
de **AssertionError** :

```
BaseException  
+-- Exception  
   +-- AssertionError
```

Les exceptions - ZeroDivisionError

Si vous divisez un nombre par zéro, Python renvoie une exception **ZeroDivisionError** :

```
>>> print(8 / 0)
Traceback (most recent call last):
  File "d:/module.py", line 6, in <module>
    print(8 / 0)
ZeroDivisionError: division by zero
```

Hierarchie de classes
de **ZeroDivisionError** :

```
BaseException
+-- Exception
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
```

Les exceptions - FileNotFoundError

Si vous essayez d'ouvrir un fichier qui n'existe pas, Python renvoie une exception **FileNotFoundError** :

```
>>> with open('test.txt', 'r') as file:
...     print(file.read())
Traceback (most recent call last):
  File "d:/module.py", line 6, in <module>
    with open('test.txt', 'r') as file:
FileNotFoundError: [Errno 2] No such file or
directory: 'test.txt'
```

Hierarchie de classes
de **FileNotFoundError**
:

```
BaseException
+-- Exception
    +-- OSError
        +-- FileNotFoundError
        +-- ...
```



Les exceptions - ModuleNotFoundError

Si essayez d'importer un module qui n'existe pas dans l'environnement Python où vous exécutez le code, Python renvoie une exception **ModuleNotFoundError** :

```
>>> import numpy
Traceback (most recent call last):
  File "d:/module.py", line 6, in <module>
    import numpy
ModuleNotFoundError: No module named 'numpy'
```

Hierarchie de classes de
ModuleNotFoundError :

```
BaseException
+-- Exception
   +-- ImportError
      +-- ModuleNotFoundError
```



Les exceptions - gérer les exceptions

On peut gérer les exceptions pour ne pas arrêter l'exécution du code. Ceci est réalisé grâce aux blocs de code **try/except/else/finally**. A minima, il faut que les mots clés **try** et **except** soient présents.

```
try:
    print(1/0)
except TypeError:
    print("Erreur sur le type de donnée")
except ZeroDivisionError:
    print("Division par zéro")
except:
    print("Autre erreur")
else:
    print("Pas d'erreur")
finally:
    print("C'est fini")
```

```
Division par zéro
C'est fini
```



Les exceptions - lever des exceptions

Python permet de lever des exceptions grâce au mot clé **raise**.

```
>>> value = 1000
>>> if value > 100:
...     raise ValueError("La valeur est trop élevée")
```

```
Traceback (most recent call last):
  File "d:/module.py", line 8, in <module>
    raise ValueError("La valeur est trop élevée")
ValueError: La valeur est trop élevée
```



Les exceptions - créer des exceptions

En plus des exceptions de la librairie standard de Python, il est possible de créer des exceptions. On doit créer une classe qui hérite d'une exception de Python.

```
class SalaryNotInRangeError(Exception):
    def __init__(self, salary, message="Le salaire n'est
pas dans l'intervalle 5000-15000"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return f'{self.salary} -> {self.message}'
```

```
>>> salary = 2000
>>> if not 5000 < salary < 15000:
...     raise SalaryNotInRangeError(salary)

Traceback (most recent call last):
  File "d:/module.py", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: 2000 -> Le
salaire n'est pas dans l'intervalle 5000-15000
```


Les exceptions - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/library/exceptions.html>

Ressource video :

- Socratica - Exceptions in Python || Python Tutorial || Learn Python Programming (ENG - 09:01) : <https://www.youtube.com/watch?v=nICKrKGHSSk>
- Corey Schafer - Python Tutorial: Using Try/Except Blocks for Error Handling (ENG - 10:33) : https://www.youtube.com/watch?v=NIWwJbo-9_8
- PyCon 2017 - Amandine Lee Passing Exceptions 101 Paradigms in Error Handling PyCon 2017 (ENG - 31:15) : <https://www.youtube.com/watch?v=BMtJbrvwlmo>



La programmation orientée objet avec Python



Plan du chapitre - La programmation orientée objet avec Python

1. Classes
2. Méthodes et attributs
3. Héritages et polymorphisme
4. Encapsulation

Chapitre

La programmation orientée objet en Python

Classes

Classes - types de programmation

Python permet de programmer suivant différents types de programmation :

- Programmation impérative
- Programmation fonctionnelle
- Programmation procédurale
- Programmation orientée objet

Nous allons voir ces quatre types de programmation avec un même exemple : l'addition de nombres.

Classes - programmation impérative

En programmation impérative, les instructions sont exécutées les unes à la suite des autres. L'état des variables est modifié au fil des instructions.

```
liste = [45, 13, 125, 14]
somme = 0
for elt in liste:
    somme += elt
print(somme)
```

Classes - programmation fonctionnelle

En programmation fonctionnelle, les problèmes sont décomposés en un ensemble de fonctions. Ces fonctions ne doivent pas contenir d'états internes.



Il existe 3 modules de la librairie standard pour mettre en place la programmation fonctionnelle : *itertools*, *functools* et *operator*

```
from functools import reduce
liste = [45, 13, 125, 14]
print(reduce(lambda x, y: x + y, liste))
```

Classes - programmation procédurale

La programmation procédurale s'appuie sur des fonctions. Le problème à résoudre est découpé en une suite d'étapes qui sont réparties entre les différentes fonctions.

```
def somme(liste):  
    temp = 0  
    for elt in liste:  
        temp += elt  
    return temp  
  
liste = [45, 13, 125, 14]  
print(somme(liste))
```


Classes - programmation orientée objet

En programmation **orientée objet**, on manipule des ensembles d'objets. Ceux-ci possèdent un **état interne** et des **méthodes** qui interrogent ou modifient cet état d'une façon ou d'une autre.

```
liste = [45, 13, 125, 14]
class Nombre:
    def __init__(self, liste):
        self.liste = liste
    def somme(self):
        temp = 0
        for elt in self.liste:
            temp += elt
        return temp
nombre = Nombre(liste)
print(nombre.somme())
```

Classes - généralités

La syntaxe minimale pour une classe en Python est la suivante :

```
class MaClasse:  
    pass
```

Pour définir le nom de la classe, on utilise le mot-clé *class* (si vous essayez d'affecter une valeur à *class* vous aurez une *SyntaxError*). Le contenu de la classe est défini par l'indentation (dans l'exemple, on utilise le mot-clé *pass* pour créer une classe vide).

Classes - généralités

La PEP8 préconise de nommer les classes avec des majuscules au début de chaque mot qui compose le nom de la classe (sans underscore)

```
# Noms de classe respectant la PEP8
class MaClasse:
    pass

class UneAutreClasse:
    pass
```

```
# Noms de classe NE RESPECTANT PAS la PEP8
class ma_classe:
    pass

class Maclasse:
    pass
```



Classes - généralités

Une **classe** est un modèle à partir duquel on peut construire des **objets**. La création d'objets à partir d'une classe s'appelle l'**instanciation**. La fonction native de Python *isinstance(object, classinfo)* permet de savoir si un objet est une instance d'une classe donnée. En Python, l'instanciation s'effectue de la façon suivante :

```
# Création de la classe
>>> class MaClasse:
...     pass

# Instanciation de la classe
>>> ma_classe = MaClasse()
>>> print(isinstance(ma_classe, MaClasse))
True
```

Classes - généralités

La PEP8 préconise d'ajouter des docstrings pour toutes les classes et les méthodes.

```
# Docstring dans les classes
class MaClasse:
    """ Exemple de classe vide """

    def ma_methode(self):
        """ Exemple de methode """
        pass

help(MaClasse)
```

Help on class MaClasse in module __main__:

class MaClasse(builtins.object)

| Methods defined here:

| ma_methode(self)
| Exemple de methode

| -----
| Data descriptors defined here:

| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)

Classes - généralités

Les docstrings sont stockées dans l'attribut `__doc__` qui est présent dans la classe et dans l'instance.

```
# Docstring dans les classes
class MaClasse:
    """ Exemple de classe vide """

    def ma_methode(self):
        """ Exemple de methode """
        pass
```

```
>>> ma_classe = MaClasse()
>>> print(MaClasse.__doc__)
Exemple de classe vide
>>> print(MaClasse.ma_methode.__doc__)
Exemple de methode
>>> print(ma_classe.__doc__)
Exemple de classe vide
>>> print(ma_classe.ma_methode.__doc__)
Exemple de methode
```

Classes - généralités

Tout est objet en Python : cela veut dire que tout peut être affecté à une variable ou passé comme argument à une fonction. On peut donc affecter une classe à une variable puis instancier des objets à partir de cette variable



En Python, une classe est une instance de **type** (métaclass)

```
>>> class MaClasse:  
...     pass  
>>> var = MaClasse  
>>> objet = var()  
>>> print(type(var))  
<class 'type'>
```

Classes - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/tutorial/classes.html>

Ressource video :

- Socratica - Python Classes and Objects || Python Tutorial || Learn Python Programming (ENG - 10:32) : https://www.youtube.com/watch?v=apACNr7DC_s
- Corey Schafer - Python OOP Tutorial 1: Classes and Instances (ENG - 15:23) : <https://www.youtube.com/watch?v=ZDa-Z5jzLYM&list=PL-osiE80TeTsghluOqKhwlXsIBIdSeYtc>

Chapitre

La programmation orientée objet en Python

Méthodes et attributs

Méthodes et attributs - instance

Dans la définition d'une classe, l'instance est appelée *self* par convention. Dans l'exemple suivant, une variable d'instance est créée à l'initialisation de l'instance (*ma_variable_instance*) et une méthode d'instance est définie (*ma_methode*).

```
class MaClasse:

    def __init__(self, ma_variable):
        self.ma_variable_instance = ma_variable

    def ma_methode(self):
        return self.ma_variable_instance
```

Méthodes et attributs - instance

On peut créer des objets en instanciant la classe. Chaque variable d'instance contient une valeur propre à l'objet.

```
class MaClasse:

    def __init__(self, ma_variable):
        self.ma_variable_instance = ma_variable

    def ma_methode(self):
        return self.ma_variable_instance
```

```
# Instantiation de la classe
>>> ma_classe_1 = MaClasse(5)
>>> ma_classe_2 = MaClasse(10)
# Utilisation de la méthode d'instance
>>> print(ma_classe_1.ma_methode())
5
>>> print(ma_classe_2.ma_methode())
10
```

Méthodes et attributs - instance

La fonction native de Python *hasattr(object, name)* permet de savoir si une instance possède un certain attribut (renvoie un booléen)

```
>>> class MaClasse:
...     def __init__(self, ma_variable):
...         self.ma_variable_instance = ma_variable
>>> ma_classe = MaClasse(3)
>>> print(hasattr(ma_classe, 'ma_variable_instance'))
True
>>> print(hasattr(ma_classe, 'autre_variable'))
False
```

Méthodes et attributs - instance

La fonction native de Python `getattr(object, name[, default])` permet de récupérer la valeur d'un attribut

```
>>> class MaClasse:
...     def __init__(self, ma_variable):
...         self.ma_variable_instance = ma_variable
>>> ma_classe = MaClasse( 3)
>>> print(getattr(ma_classe, 'ma_variable_instance' ))
3
>>> print(getattr(ma_classe, 'autre_variable', 'aucun attribut'))
aucun attribut
>>> print(getattr(ma_classe, 'autre_variable' ))
Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 559, in <module>
    print(getattr(ma_classe, 'autre_variable' ))
AttributeError: 'MaClasse' object has no attribute 'autre_variable'
```

Méthodes et attributs - instance

Pour connaître tous les attributs d'instance d'un objet, on peut utiliser la fonction native de Python **vars()**. Elle renvoie l'attribut spécial **__dict__** de l'instance.

```
class MaClasse:
    def __init__(self, ma_variable):
        self.ma_variable_instance = ma_variable

maclasse = MaClasse(12)
print(vars(maclasse))
print(maclasse.__dict__)
```

```
{'ma_variable_instance': 12}
{'ma_variable_instance': 12}
```

Méthodes et attributs - instance

On peut ajouter des attributs d'instance après l'instanciation de l'objet.

```
class MaClasse:
    def __init__(self, ma_variable):
        self.ma_variable_instance = ma_variable

maclasse = MaClasse(12)
maclasse.ma_var = 48
print(vars(maclasse))
```

```
{'ma_variable_instance': 12, 'ma_var': 48}
```

Méthodes et attributs - attributs de classe

Pour avoir des données qui sont partagées entre les différentes instances de la classe, on peut utiliser des **attributs de classe**. Cela fonctionne comme une variable globale à toutes les instances de la classe.

Voici deux exemples de cas d'usage d'attributs de classe :

- Valeurs minimales ou maximales pour les attributs
- Constantes utilisées dans toutes les instances (pi par exemple)

Méthodes et attributs - attributs de classe

Pour créer des **attributs de classe**, on les définit à l'**intérieur de l'espace de noms de la classe** mais à l'extérieur des méthodes.

```
# Définition d'un attribut de classe
class MaClasse:
    attribut_classe = 0
```

Méthodes et attributs - attributs de classe

Les attributs de classes ont la même valeur pour toutes les instances de la classe.

```
# Définition d'un attribut de classe
>>> class MaClasse:
...     attribut_classe = 0

>>> ma_classe_1 = MaClasse()
>>> ma_classe_2 = MaClasse()
>>> MaClasse.attribut_classe = 2
>>> print(f"Attribut de la classe 1 : {ma_classe_1.attribut_classe}")
Attribut de la classe 1 : 2
>>> print(f"Attribut de la classe 2 : {ma_classe_2.attribut_classe}")
Attribut de la classe 2 : 2
```

Méthodes et attributs - attributs de classe

Les attributs de classe **ne sont pas modifiables** depuis l'**instance**, ils doivent être modifiés depuis la **classe**.

```
>>> class MaClasse:
...     attribut_classe = 0
>>> ma_classe_1 = MaClasse()
>>> ma_classe_2 = MaClasse()
>>> ma_classe_1.attribut_classe = 2
>>> print(f"Attribut de la classe 1 : {ma_classe_1.attribut_classe}")
Attribut de la classe 1 : 2
>>> print(f"Attribut de la classe 2 : {ma_classe_2.attribut_classe}")
Attribut de la classe 2 : 0
```

Méthodes et attributs - attributs de classe

Pour utiliser l'attribut de classe dans une méthode d'instance, il faut écrire *NomDeLaClasse.variable_de_classe*.

```
class MaClasse:
    attribut_classe = 0
    def modifier_attribut(self):
        MaClasse.attribut_classe += 1
ma_classe = MaClasse()
ma_classe.modifier_attribut()
print("Résultat : " + str(ma_classe.attribut_classe))
```

Résultat : 1

Méthodes et attributs - attributs de classe

Les fonctions natives de Python *hasattr(object, name)*, *getattr(object, name[, default])* et *setattr(object, name, value)* peuvent être utilisées de la même façon sur les attributs de classe que sur les attributs d'instance.

```
>>> class MaClasse:
...     attribut_classe = 0
...     def modifier_attribut(self):
...         MaClasse.attribut_classe += 1
>>> ma_classe = MaClasse()
>>> ma_classe.modifier_attribut()
>>> print(getattr(MaClasse, 'attribut_classe'))
1
>>> print(getattr(ma_classe, 'attribut_classe'))
1
```

Méthodes et attributs - méthodes de classe

Les méthodes de classes s'appliquent à la classe elle-même. Elles n'ont pas accès aux instances.

Elles sont créées en utilisant le décorateur **@classmethod**.

```
>>> class MaClasse:
...     attribut_classe = 0
...     @classmethod
...     def modifier_attribut(cls):
...         cls.attribut_classe += 1
>>> ma_classe = MaClasse()
>>> ma_classe.modifier_attribut()
>>> print(MaClasse.attribut_classe)
1
```

Méthodes et attributs - méthodes de classe

Un exemple d'utilisation d'une méthode de classe est la création d'un constructeur alternatif au constructeur `__init__(self)`.

```
class Date:
    def __init__(self, annee, mois, jour):
        self.annee, self.mois, self.jour = annee, mois, jour

    @classmethod
    def from_str(cls, datestr):
        parties = datestr.split("-")
        annee, mois, jour = int(parties[0]), int(parties[1]),
int(parties[2])
        return Date(annee, mois, jour)

dt = Date.from_str('2020-04-30')
print(dt.annee, dt.mois, dt.jour)
```

2020 4 30

Méthodes et attributs - méthodes statiques

Une méthode statique permet d'avoir une fonction indépendante de la classe et de l'instance à l'intérieur d'une classe. Elle est créée en utilisant le décorateur **@staticmethod**.

```
>>> class Somme:
...     @staticmethod
...     def ajouter(a, b):
...         return a + b
>>> somme = Somme()
>>> print(somme.ajouter(4, 5))
9
```


Classes - ressources complémentaires

Ressource video :

- The net Ninja - Python 3 Tutorial for Beginners #18 - Methods & Attributes (ENG - 09:25) :
<https://www.youtube.com/watch?v=LwFnF9XoEfM>
- Corey Schafer - Python OOP Tutorial 2: Class Variables (ENG - 11:40) :
<https://www.youtube.com/watch?v=BJ-VvGyQxho>
- Corey Schafer - Python OOP Tutorial 3: classmethods and staticmethods (ENG - 15:19) :
<https://www.youtube.com/watch?v=rq8cL2XMM5M>

Chapitre

La programmation orientée objet en Python

Héritages et polymorphisme

Héritages et polymorphisme - Héritage

L'**héritage** est une fonctionnalité de la **programmation orientée objet** qui permet de déclarer qu'une classe (appelée **classe fille**) sera modelée sur une autre (appelée **classe mère**).

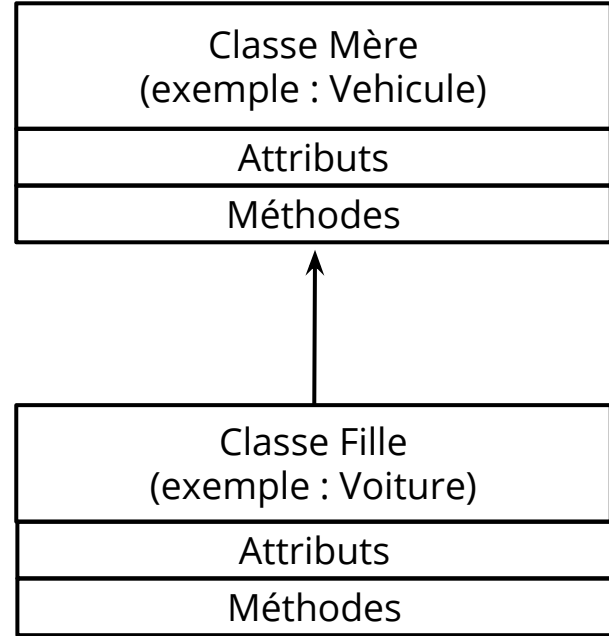
La classe **fille** a **accès** à tous les **attributs** et les **méthodes** de la classe **mère**. Elle peut en plus ajouter de nouveaux attributs ou méthodes et elle peut en plus **réécrire** les attributs et les méthodes de la classe mère.

Héritages et polymorphisme - Héritage

Dans l'exemple suivant, la classe **Voiture** hérite de la classe **Vehicule**.

```
# Héritage
class Vehicule:
    pass

class Voiture(Vehicule):
    pass
```



Héritages et polymorphisme - Héritage

La fonction native de Python *issubclass(class, classinfo)* permet de savoir si une classe hérite d'une autre classe.



Tout est objet en Python donc les classes sont aussi des objets (héritent de *object*)

```
# Héritage
>>> class Vehicule:
...     pass

>>> class Voiture(Vehicule):
...     pass

>>> print(issubclass(Voiture, Vehicule))
True
>>> print(issubclass(Voiture, object))
True
>>> print(issubclass(Voiture, int))
False
```

Héritages et polymorphisme - Héritage

La **classe fille** (dans l'exemple *Voiture*) hérite de toutes les méthodes et les attributs de la classe mère : elle hérite donc de la **méthode d'initialisation** `__init__()` et de l'attribut d'instance *numero*.

```
# Héritage
class Vehicule:

    def __init__(self, numero):
        self.numero = numero
        print(f"La voiture {self.numero} peut démarrer")

class Voiture(Vehicule):
    pass
```

```
>>> ma_voiture = Voiture(5)
La voiture 5 peut démarrer
>>> print(ma_voiture.numero)
5
```

Héritages et polymorphisme - Héritage

On peut réécrire la **méthode d'initialisation** de la **classe fille**. Si on veut aussi appeler la méthode d'initialisation de la classe mère, il faut le faire **explicitement**.

```
class Vehicule:

    def __init__(self, numero):
        self.numero = numero
        print(f"La voiture de la classe mère Vehicule {self.numero} peut démarrer")

class Voiture(Vehicule):

    def __init__(self, numero):
        Vehicule.__init__(self, numero)
        print(f"La voiture de la classe fille Voiture {self.numero} peut démarrer")
```

```
>>> ma_voiture = Voiture(5)
La voiture de la classe mère
Vehicule 5 peut démarrer
La voiture de la classe fille
Voiture 5 peut démarrer
```

Héritages et polymorphisme - Héritage

Pour appeler une méthode de la classe mère, on peut aussi utiliser le mot clé **super()** de Python.

```
class Vehicule:

    def __init__(self, numero):
        self.numero = numero
        print(f"La voiture de la classe mère Vehicule {self.numero} peut démarrer")

class Voiture(Vehicule):

    def __init__(self, numero):
        super().__init__(numero)
        print(f"La voiture de la classe fille Voiture {self.numero} peut démarrer")
```

```
>>> ma_voiture = Voiture(5)
La voiture de la classe mère
Vehicule 5 peut démarrer
La voiture de la classe fille
Voiture 5 peut démarrer
```


Héritages et polymorphisme - Héritage

En Python, la **surcharge** de méthodes **n'existe pas** (comme en Java par exemple). Seul le **nom de la méthode** compte en Python même si la signature est différente.

```
class Vehicule:

    def __init__(self, numero):
        self.numero = numero
        print(f"La voiture de la classe mère Vehicule {self.numero} peut démarrer")

class Voiture(Vehicule):

    def __init__(self):
        print(f"La voiture de la classe fille Voiture peut démarrer")
```

```
>>> ma_voiture = Voiture()
La voiture de la classe fille
Voiture peut démarrer
>>> ma_voiture = Voiture(5)
Traceback (most recent call last):
  File "d:/module1.py", line 15, in
    <module>
        ma_voiture = Voiture(5)
TypeError: __init__() takes 1
positional argument but 2 were given
```

Héritages et polymorphisme - Héritage multiple

Python permet l'**héritage multiple**. Pour créer une classe qui hérite de plusieurs classes, il faut utiliser la syntaxe suivante :

```
# Héritage multiple
>>> class Vehicule:
...     pass
>>> class MoyenDeTransport:
...     pass
>>> class Voiture(Vehicule, MoyenDeTransport):
...     pass
>>> print(issubclass(Voiture, Vehicule))
True
>>> print(issubclass(Voiture, MoyenDeTransport))
True
```

Héritages et polymorphisme - Héritage multiple

Dans le cadre d'un héritage multiple, l'attribut `__mro__` (Method Resolution Order) permet de connaître l'ordre d'héritage des classes.

```
# Héritage multiple
>>> class Vehicule:
...     pass
>>> class MoyenDeTransport:
...     pass
>>> class Voiture(Vehicule, MoyenDeTransport):
...     pass
>>> print(Voiture.__mro__)
(<class '__main__.Voiture'>, <class '__main__.Vehicule'>, <class '__main__.MoyenDeTransport'>,
<class 'object'>)
```

Héritages et polymorphisme - Héritage multiple

Si les deux classes mères implémentent la même méthode, c'est l'ordre d'héritage qui va permettre de savoir quelle méthode va être utilisée.

```
# Héritage multiple
>>> class Vehicule:
...     def methode(self):
...         print("Méthode d'instance dans la classe Vehicule")
>>> class MoyenDeTransport:
...     def methode(self):
...         print("Méthode d'instance dans la classe MoyenDeTransport")
>>> class Voiture(Vehicule, MoyenDeTransport):
...     pass
>>> voiture = Voiture()
>>> voiture.methode()
Méthode d'instance dans la classe Vehicule
```

Héritages et polymorphisme - Classes abstraites

Une **classe abstraite** est une classe qui **ne peut pas être instanciée** : elle doit être héritée pour être instanciée. En Python, on utilise le module *abc*.

```
import abc

class Vehicule(abc.ABC):

    @abc.abstractmethod
    def rouler(self):
        pass

class Voiture(Vehicule):

    def __init__(self):
        print(f"La voiture de la classe fille Voiture peut démarrer")
```

```
>>> ma_voiture = Voiture()
Traceback (most recent call last):
  File "d:/module1.py", line 14, in
    <module>
        ma_voiture = Voiture()
TypeError: Can't instantiate
abstract class Voiture with abstract
methods rouler
```

Héritages et polymorphisme - Classes abstraites

Pour **supprimer l'exception**, il faut implémenter la méthode d'instance *rouler()* dans la classe fille *Voiture*.

```
import abc

class Vehicule(abc.ABC):

    @abc.abstractmethod
    def rouler(self):
        pass

class Voiture(Vehicule):

    def __init__(self):
        print(f"La voiture de la classe fille Voiture peut démarrer")

    def rouler(self):
        print("La voiture roule")
```

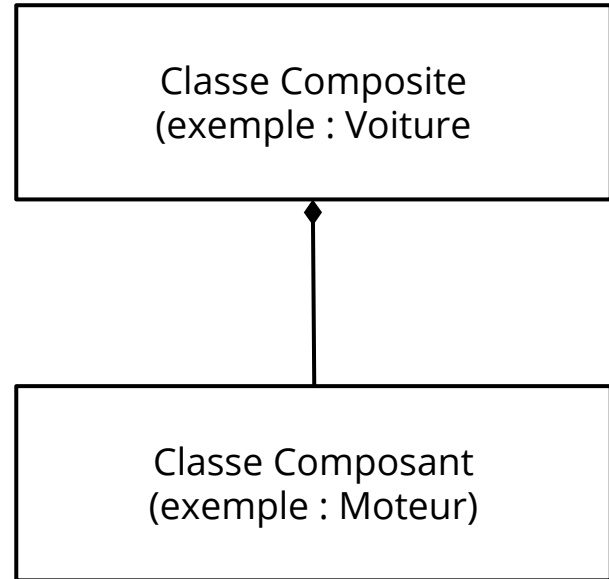
```
>>> ma_voiture = Voiture()
La voiture de la classe fille
Voiture peut démarrer
```

Héritages et polymorphisme - Composition

La **composition** permet d'intégrer une instance d'une classe à l'intérieur d'une autre classe

```
# Composition
class Moteur:
    pass

class Voiture:
    def __init__(self):
        self.moteur = Moteur()
```



Héritages et polymorphisme - polymorphisme

Le polymorphisme consiste à fournir une interface commune à des objets de types différents. Un exemple de polymorphisme en Python natif est l'utilisation du signe **+** : dans le cas des **entiers (int)**, ce signe effectue une **addition** et dans le cas des **chaînes de caractères (str)**, ce signe effectue une **concaténation**.

```
>>> print(5 + 10)
15
>>> print("hello " + "world")
hello world
```


Héritages et polymorphisme - polymorphisme

En arrière plan, lorsque l'on utilise le signe `+` en Python, la méthode spéciale `__add__()` est appelée. On peut donc mettre en place le polymorphisme pour le signe `+` dans des classes que l'on crée.

```
class MaClasse:

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

    def __add__(self, other):
        if self.budget + other.budget > 100:
            return MaClasse(0)
        else:
            return MaClasse(self.budget + other.budget)
```

```
>>> instance_1 = MaClasse(10)
>>> instance_2 = MaClasse(80)
>>> result = instance_1 + instance_2
>>> print(result.budget)
90
>>> result = result + instance_2
>>> print(result.budget)
0
```

Héritages et polymorphisme - polymorphisme

On peut aussi utiliser le polymorphisme sur les autres **méthodes de comparaison** en Python. Voici la liste de correspondances avec les méthodes spéciales :

- `==` appelle la méthode `__eq__()`
- `!=` appelle la méthode `__ne__()`
- `>=` appelle la méthode `__ge__()`
- `<=` appelle la méthode `__le__()`
- `>` appelle la méthode `__gt__()`
- `<` appelle la méthode `__lt__()`

Héritages et polymorphisme - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/tutorial/classes.html>

Ressource video :

- Corey Schafer - Python OOP Tutorial 4: Inheritance - Creating Subclasses (ENG - 19:39) : <https://www.youtube.com/watch?v=RSI87IqOXDE>
- Corey Schafer - Python OOP Tutorial 5: Special (Magic/Dunder) Methods (ENG - 13:49) : <https://www.youtube.com/watch?v=3ohzBxoFHAY&t=637s>

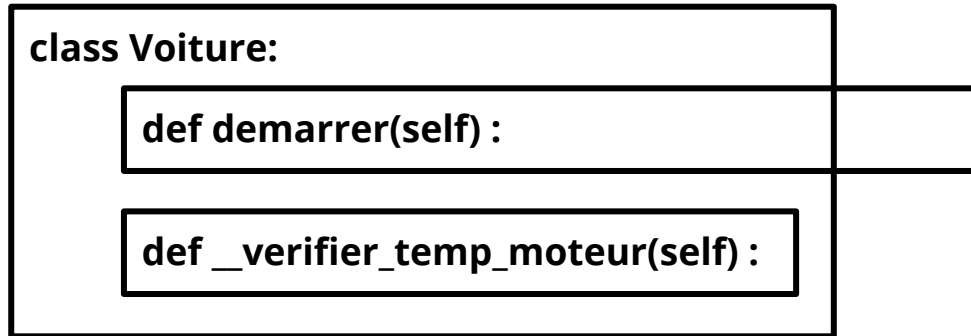
Chapitre

La programmation orientée objet en Python

Encapsulation

Encapsulation - généralités

L'encapsulation correspond au fait que les attributs et les méthodes sont **spécifiquement associés à l'objet** et **s'appliquent à l'objet lui-même** (et non à un autre objet).



Encapsulation - généralités

En programmation, il existe 3 types de **visibilités** pour les attributs et les méthodes :

- **Publique** : les attributs et les méthodes sont accessibles de partout
- **Protégée** : les attributs et les méthodes sont accessibles dans la classe où ils sont créés et dans les classes enfants
- **Privée** : les attributs et les méthodes ne sont accessibles que dans la classe où ils sont créés

Encapsulation - généralités

En Python, **tout est public**. Il est toujours possible d'accéder des attributs ou des méthodes protégés ou privés depuis l'extérieur d'une classe. C'est au développeur Python de savoir respecter les visibilités des attributs et des méthodes.

En Python, les attributs et les méthodes **protégés** commencent par `_` (**un underscore**).

En Python, les attributs et les méthodes **privés** commencent par `__` (**deux underscores**).

Encapsulation - exemple

Les attributs et les méthodes **protégés** sont accessibles depuis l'extérieur de l'objet (il n'y a pas d'exception)

```
# Attribut protégés
>>> class MaClasse:
...     def __init__(self):
...         self._attribut_protege = 5
...     def _methode_protegee(self):
...         self._attribut_protege += 10
>>> instance = MaClasse()
>>> print(instance._attribut_protege)
5
>>> instance._methode_protegee()
>>> print(instance._attribut_protege)
15
```


Encapsulation - exemple

Les attributs et les méthodes **privés** ne sont pas directement accessibles depuis l'extérieur de l'objet.

```
# Attribut privé
>>> class MaClasse:
...     def __init__(self):
...         self.__attribut_privé = 5
>>> instance = MaClasse()
>>> print(instance.__attribut_privé)
Traceback (most recent call last):
  File "d:/module1.py", line 6, in <module>
    print(instance.__attribut_privé)
AttributeError: 'MaClasse' object has no attribute '__attribut_privé'
```

Encapsulation - exemple

Mais les attributs et méthodes privés sont accessibles indirectement. Python utilise le **name mangling** pour les attributs et méthodes privés.

```
# Attribut privé
>>> class MaClasse:
...     def __init__(self):
...         self.__attribut_privé = 5
>>> ma_classe = MaClasse()
>>> print(vars(ma_classe))
{'_MaClasse__attribut_privé': 5}
>>> print(ma_classe._MaClasse__attribut_privé)
5
```

Encapsulation - propriétés

Il peut parfois être nécessaire d'accéder (grâce à des **accesseurs** / **getters**) ou de modifier (grâce à des **mutateurs** / **setters**) des attributs protégés ou privés depuis l'extérieur de l'objet en respectant le principe d'encapsulation.

Python facilite la mise en place des accesseurs et des mutateurs grâce aux **propriétés**.

Encapsulation - propriétés

Pour respecter le principe d'encapsulation, on accède aux attributs privés avec un **getter** et on le modifie avec un **setter**.

```
class MaClasse:

    def __init__(self, x):
        self.__x = x

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x
```

```
>>> instance = MaClasse(10)
>>> print(instance.get_x())
10
>>> instance.set_x(45)
>>> print(instance.get_x())
45
```

Encapsulation - propriétés

La classe **property** de la bibliothèque standard permet de simplifier la syntaxe.

```
class MaClasse:

    def __init__(self, x):
        self.__x = x

    def __get_x(self):
        return self.__x

    def __set_x(self, x):
        self.__x = x

    x = property(__get_x, __set_x)
```

```
>>> instance = MaClasse(10)
>>> print(instance.x)
10
>>> instance.x = 45
>>> print(instance.x)
45
```

Encapsulation - propriétés

On peut aussi utiliser la classe **property** avec des **décorateurs** :

```
class MaClasse:

    def __init__(self, x):
        self._x = x

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

```
>>> instance = MaClasse(10)
>>> print(instance.x)
10
>>> instance.x = 45
>>> print(instance.x)
45
>>> del instance.x
```

Encapsulation - ressources complémentaires

Ressource texte :

- OpenClassrooms - Découvrez la programmation orientée objet avec Python : <https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python/4313211-comprenez-lencapsulation>

Ressource video :

- ProgrammingKnowledge - Python Tutorial for Beginners 27 - Python Encapsulation (ENG - 11:35) : <https://www.youtube.com/watch?v=TFLo9m0jFEg>



Les bibliothèques en Python



Plan du chapitre - Les bibliothèques en Python

1. [La bibliothèque standard](#)
2. [Les bibliothèques tierces](#)

Chapitre

Les bibliothèques en Python

La bibliothèque standard

La bibliothèque standard - généralités

En python, les bibliothèques importées peuvent venir de trois sources différentes :

La bibliothèque standard :

Dans ce cas, il suffit d'ajouter l'import en début de fichier Python (pas de paquet à télécharger)

Les bibliothèques tierces :

Dans ce cas, il faut d'abord télécharger la bibliothèque à l'aide d'un gestionnaire de paquets (exemple : pip) et ensuite faire l'import en début de fichier Python

Vos propres modules :

Dans ce cas, il suffit d'ajouter l'import en début de fichier Python (le nom du module correspond au nom du fichier sans l'extension ".py")

La bibliothèque standard - généralités

Si vous importez un module d'une bibliothèque tierce que vous n'avez pas préalablement installée, Python renvoie une *ModuleNotFoundError* :

```
import numpy

Traceback (most recent call last):
  File "d:/plbayart/code_exemples.py", line 1, in <module>
    import numpy
ModuleNotFoundError: No module named 'numpy'
```

La bibliothèque standard - généralités

L'import de vos propres bibliothèques se fait aussi avec le mot clé *import*. Les deux modules *module1* et *module2* sont dans le même dossier. On peut donc importer la fonction *print_hello()* du *module1* dans le *module2*.

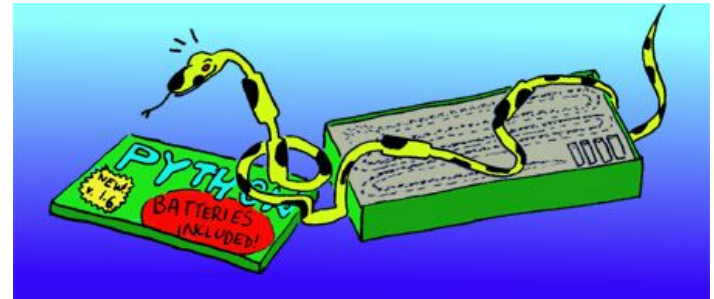
```
module1.py ×  
1 def print_hello():  
2     print("Hello")
```

```
module2.py ×  
1 import module1  
2  
3 module1.print_hello()
```

La bibliothèque standard - généralités

Pour avoir accès aux modules de la bibliothèque standard (il y en a plus de 200), il suffit de faire l'import du module au début du fichier Python. On va regarder plus en détails les modules suivants :

- *math* : calculs mathématiques
- *random* : génération de nombres pseudo-aléatoires
- *datetime* : gestion des dates et du temps
- *csv* : manipulation de fichiers csv
- *json* : manipulation de fichiers json



La bibliothèque standard - math

La librairie *math* de Python permet d'effectuer des calculs mathématiques (hors nombres complexes qui sont réalisés grâce à la librairie *cmath*) :

- Fonctions arithmétiques et de représentation (*ceil, floor, gcd ...*)
- Fonctions trigonométriques (*cos, sin, tan, asin ...*)
- Fonctions exponentielle et logarithme (*exp, log, pow, sqrt ...*)
- Fonctions angulaires (*degrees, radians*)
- Fonctions hyperboliques (*cosh, sinh, tanh, acosh, asinh, atanh*)
- Fonctions spéciales (*erf, gamma ...*)
- Constantes (*pi, e, tau, inf, nan*)

La bibliothèque standard - math

Quelques exemples d'utilisation du module *math* de la bibliothèque standard de Python :

```
>>> import math
>>> print(f"Le nombre pi vaut : {round(math.pi)}")
Le nombre pi vaut : 3
>>> print(f"Le cosinus de pi radians vaut : {round(math.cos(math.pi))}")
Le cosinus de pi radians vaut : -1
>>> print(f"Le sinus de pi radians vaut : {round(math.sin(math.pi))}")
Le sinus de pi radians vaut : 0
>>> print(f"La tangente de pi radians vaut : {round(math.tan(math.pi))}")
La tangente de pi radians vaut : 0
```


La bibliothèque standard - random

Le module *random* de Python permet de générer des nombres pseudo-aléatoires. Il est structurée de la façon suivante :

- Fonctions de gestion d'état (*seed()*, *getstate()*, *setstate()*, *getrandbits()*)
- Fonctions pour les entiers (*randrange()*, *randint()*)
- Fonctions pour les séquences (*choice()*, *choices()*, *shuffle()*, *sample()*)
- Distributions pour les nombres réels (*random()*, *uniform()* ...)
- Générateur alternatif (*Random()*, *SystemRandom()*)

! Le module *random* **ne doit pas** être utilisé pour des applications de sécurité. Pour ce cas, il faut utiliser le module *secrets*

La bibliothèque standard - random

Quelques exemples d'utilisation du module *random* :

```
>>> from random import seed, randint, choice, shuffle
# seed fixe la série de nombres aléatoires
>>> seed(42)
# randint renvoie un entier au hasard dans une plage de valeurs
>>> print(randint(0, 10))
10
# choice renvoie un élément choisi aléatoirement
>>> print(choice([45, 12, 16, 18]))
45
# shuffle mélange un ensemble d'éléments
>>> ma_liste = [45, 12, 16, 18]
>>> shuffle(ma_liste)
>>> print(ma_liste)
[18, 12, 16, 45]
```

La bibliothèque standard - datetime

Le module *datetime* permet de manipuler des dates et des temps en Python. Les principales classes de ce module sont :

- **class *date*** : pour gérer les dates basées sur le calendrier Grégorien
 - Attributs : *year*, *month* et *day*
- **class *time*** : pour gérer l'heure
 - Attributs : *hour*, *minute*, *second*, *microsecond* et *tzinfo*
- **class *datetime*** : pour gérer la date et l'heure
 - Attributs : *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond* et *tzinfo*
- **class *timedelta*** : pour gérer des différences entre deux dates et/ou heures
- **class *timezone*** : pour gérer l'offset par rapport au temps UTC

La bibliothèque standard - datetime

Quelques exemples d'utilisation du module *datetime* :

```
>>> from datetime import datetime, timedelta
>>> date_1 = datetime(2020, 6, 10, 10, 10)
>>> date_2 = datetime(2020, 7, 12, 5, 25)
>>> print(date_2 - date_1, type(date_2 - date_1))
31 days, 19:15:00 <class 'datetime.timedelta'>
>>> print(date_1 + timedelta(26), type(date_1 + timedelta(26)))
2020-07-06 10:10:00 <class 'datetime.datetime'>
```

La bibliothèque standard - csv

Le format **CSV (Comma Separated Values)** est un format très répandu dans l'importation et l'exportation de données. Le module `csv` de Python permet de lire et d'écrire des fichiers csv.

Les outils de lecture de fichiers csv sont la fonction `reader()` et la classe `DictReader`.

Les outils d'écriture de fichiers csv sont la fonction `writer()` et la classe `DictWriter`.

La bibliothèque standard - csv

Exemple d'utilisation du module csv pour la création d'un fichier csv :

```
import csv
with open('data.csv', 'w', newline='') as csvfile:
    fieldnames = ['prenom', 'note']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'prenom': 'Gabriel', 'note': 15})
    writer.writerow({'prenom': 'Louise', 'note': 12})
    writer.writerow({'prenom': 'Raphael', 'note': 18})
```



data - Bloc-notes

Fichier Edition Format

prenom,note

Gabriel,15

Louise,12

Raphael,18

La bibliothèque standard - csv

Exemple d'utilisation du module csv pour la lecture d'un fichier csv :

```
import csv
with open('data.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        for col, val in row.items():
            print(f"{col} : {val}")
```

```
prenom : Gabriel
note : 15
prenom : Louise
note : 12
prenom : Raphael
note : 18
```

La bibliothèque standard - json

Le format **JSON (JavaScript Object Notation)** est un format de données textuelles qui permet de représenter de l'information structurée. Le module json de Python contient des fonctions et de classes pour sérialiser des objets Python en json et désérialiser du json en objets Python.

- Sérialisation vers un fichier json : dump()
- Sérialisation vers une chaîne de caractères : dumps()
- Désérialisation depuis un fichier json : load()
- Désérialisation depuis une chaîne de caractères json : loads()

La bibliothèque standard - json

Quelques exemples d'utilisation du module *json* :

```
>>> import json
>>> data_json = json.dumps(['b', {'a': (True, None, 1.0, 2, False)}])
>>> print(data_json, type(data_json))
["b", {"a": [true, null, 1.0, 2, false]}] <class 'str'>
>>> data_python = json.loads('["b", {"a": [true, null, 1.0, 2, false]}]')
>>> print(data_python, type(data_python))
['b', {'a': [True, None, 1.0, 2, False]}] <class 'list'>
```

La bibliothèque standard - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/library/index.html>
- Survol de la bibliothèque standard : <https://docs.python.org/fr/3.8/tutorial/stdlib.html>

Ressource video :

- EuroPython Conference - Alessandro Molina - Python Standard Library, The Hidden Gems (ENG - 44:24) : <https://www.youtube.com/watch?v=fhn0p8uS788>

Chapitre

Les bibliothèques en Python

Les bibliothèques tierces

Les bibliothèques tierces - PyPI

Le Python Package Index (PyPI) est un serveur centralisé de paquets Python. Il permet de regrouper les modules et les paquets mis à disposition par la communauté ainsi que de gérer les versions.



Les bibliothèques tierces - pip

Pip est le logiciel en lignes de commandes qui permet de faire la connexion en votre ordinateur et les serveurs de PyPI afin d'installer, de mettre à jour ou de désinstaller des modules ou des paquets de bibliothèques tierces.

Pour installer la bibliothèque tierce Numpy, il faut taper la commande suivante dans l'invite de commande (ou PowerShell) :



En datascience,
c'est plutôt **conda**
qui est utilisé pour
gérer les
bibliothèques
tierces

```
C:\> pip install numpy
```

Les bibliothèques tierces - pip

La commande *pip freeze* permet de connaître toutes les **bibliothèques tierces** de l'environnement Python dans lequel vous êtes placé ainsi que les **numéros de versions** :



Dans l'exemple, seule la bibliothèque *Flask* a été installée. Toutes les autres bibliothèques sont des dépendances de *Flask*

```
PS D:\> pip freeze
click==7.1.2
Flask==1.1.2
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

Les bibliothèques tierces - pip

Pour transmettre la liste des bibliothèques tierces à utiliser pour un projet, on peut créer un fichier **requirements.txt** à partir de la commande *pip freeze*.

```
PS D:\> pip freeze > requirements.txt
```

On peut alors installer les bibliothèques tierces dans le nouvel environnement à l'aide de la commande suivante :

```
PS D:\> pip install -r requirements.txt
```

Les bibliothèques tierces - installation

Le problème d'utiliser directement **pip** est que l'installation des bibliothèques tierces s'effectue **directement** sur la distribution de Python qui est sur votre ordinateur. Si vous installez beaucoup de librairies différentes, il y a un risque de **conflit** entre différentes bibliothèques. De plus, il est parfois nécessaire de tester **différentes versions** de bibliothèques.

Les **environnement virtuels** permettent de régler ce problème. Il s'agit d'une copie de Python qui permet d'installer un environnement séparé avec toutes les bibliothèques dédiées à votre projet.

Les bibliothèques tierces - installation

Virtualenv est une bibliothèque qui permet de gérer les environnements virtuels (dans la bibliothèque standard, il existe aussi **venv**).

Pour simplifier la gestion des environnements virtuels, il existe une bibliothèque tierce qui regroupe **pip** en **virtualenv** : il s'agit de **pipenv**.

La commande suivante permet de créer un environnement virtuel (s'il n'existe pas encore) et d'y installer une librairie (dans cet exemple *numpy*) :

```
PS D:\> pipenv install numpy
```

Les bibliothèques tierces - installation

Une fois la commande **pipenv install** utilisée, deux fichiers de dépendances sont créés (équivalent au fichier *requirements.txt* de *pip*) :

- *Pipfile* : liste les dépendances et la version de Python à utiliser
- *Pipfile.lock* : liste les dépendances et les versions qui sont utilisées en fonction des requis du fichier *Pipfile*

Les bibliothèques tierces - installation

Pour utiliser un environnement virtuel, il faut l'activer grâce à la commande suivante :

```
PS D:\> pipenv shell
```

Toutes les commandes exécutées suite à cette commande seront interprétées avec Python et les bibliothèques tierces contenus dans l'environnement virtuel. Pour sortir de l'environnement virtuel, on utilise la commande suivante :

```
PS D:\> exit
```

Les bibliothèques tierces - installation

Pour installer les bibliothèques issues d'un fichier **Pipfile** (ou d'un fichier *requirements.txt* s'il a été créé à partir de *pip*) dans un environnement virtuel, on utilise la commande suivante dans le dossier où se situe le fichier de dépendances :

```
PS D:\> pipenv install
```

Les bibliothèques tierces - installation

Le langage Python étant beaucoup utilisé dans le domaine de la **data science**, voici quelques une des principales bibliothèques utilisées :

- Numpy
- Pandas
- Scipy
- Scikit-learn
- Keras

On verra par la suite des bibliothèques pour des **applications web** et pour des **tests**.

Les bibliothèques tierces - ressources complémentaires

Ressource texte :

- Documentation de pipenv : <https://pipenv-fork.readthedocs.io/en/latest/>

Ressource video :

- Corey Schafer - Python Tutorial: Pipenv - Easily Manage Packages and Virtual Environments (ENG - 32:28) : <https://www.youtube.com/watch?v=zDYL22QNiWk>

Fonctionnalités avancées

Plan du chapitre - Fonctionnalités avancées

1. Tests du code
2. Débugueur
3. Interfaces graphiques
4. Base de données

Chapitre Fonctionnalités avancées

Tests du code

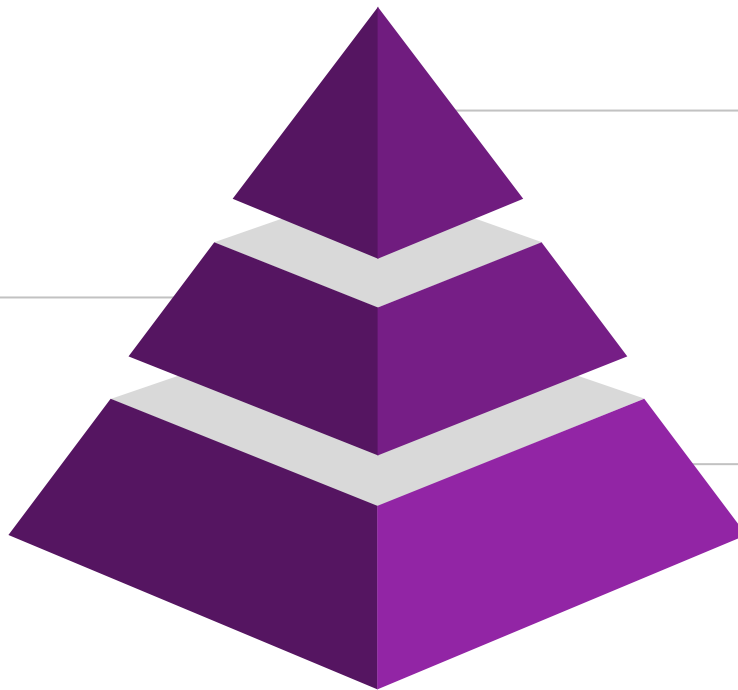
Tests du code - généralités



Tests d'intégration

Vérification que toutes les parties développées indépendamment fonctionnent correctement ensemble

2



Tests d'interface et tests fonctionnels



1

Vérification que l'application se comporte comme prévu dans les spécifications

Tests unitaires



3

Vérification que les structures élémentaires du code fonctionnent correctement indépendamment des autres parties du code

Tests du code - généralités

Les tests remplissent plusieurs fonctions :

- Vérifier le bon fonctionnement du code
- Maintenir stable le code au fil du projet
- Ne pas avoir peur de casser le code existant en ajoutant de nouvelles fonctionnalités
- Vérification que les spécifications sont respectées
- ...

Tests du code - doctest

Nous avons vu dans la partie sur les classes que la PEP8 préconise d'écrire des **docstrings** pour toutes les fonctions, classes, méthodes et modules. Python permet d'écrire des tests unitaires directement dans les **docstrings** : les **doctests**.

Pour que les doctests d'un fichier python s'exécutent au lancement du fichier, il faut ajouter les lignes de code suivantes en fin de fichier :

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

Tests du code - doctest

Pour tester du code dans les **doctests**, il faut ajouter `>>>` devant le code à tester et ajouter à la ligne suivante le résultat attendu du code.

```
def print_hello(name="World") :  
    """  
    Print hello to the name in parameter  
    >>> print_hello()  
    'Hello World !'  
    >>> print_hello("Pierre-Loic")  
    'Hello Pierre-Loic !'  
    """  
    return "Hello " + name + " !"
```

Tests du code - doctest

S'il n'y a pas d'erreur dans les doctests, rien ne s'affiche lorsque l'on exécute le code.

S'il y a des erreurs, elles s'affichent à l'exécution du code :

```
*****
File "d:/module.py", line 42, in __main__.print_hello
    Hello Pierre-Loic !
Got:
    'Hello Pierre-Loic !'
*****
1 items had failures:
  2 of  2 in __main__.print_hello
***Test Failed*** 2 failures.
```

Tests du code - unittest

Le module **unittest** est le module de tests unitaires de la bibliothèque standard de Python.

Pour créer un test unitaire avec **unittest**, il faut créer une classe qui hérite de la classe *TestCase* :

```
import unittest

class MonTest(unittest.TestCase):
    pass

if __name__ == '__main__':
    unittest.main()
```

```
-----
Ran 0 tests in 0.000s

OK
```

Tests du code - unittest

Voici un exemple du test d'une fonction qui renvoie *True* si le nombre passé en paramètre est pair et renvoie *False* sinon.

```
import unittest

def pair(nombre):
    """ Retourne True si le nombre est pair False sinon """
    return True if nombre%2==0 else False

class MonTest(unittest.TestCase):
    def test_pair(self):
        self.assertTrue(pair(2))
        self.assertFalse(pair(1))

if __name__ == '__main__':
    unittest.main()
```

Ran 1 test in 0.001s

OK

Tests du code - unittest

Dans la pratique, le fichier de code à tester est dans un fichier différent du code de test :

module.py

```
def pair(nombre):  
    """ Retourne True si le nombre est pair  
    False sinon """  
    return True if nombre%2==0 else False
```

test_module.py

```
import unittest  
import module  
  
class MonTest(unittest.TestCase):  
    def test_pair(self):  
        self.assertTrue(module.pair(2))  
        self.assertFalse(module.pair(1))  
  
if __name__ == '__main__':  
    unittest.main()
```

Tests du code - unittest

Voici quelques assertions qu'on peut utiliser avec *unittest* pour vérifier le bon fonctionnement du code :

- *assertEqual(a, b)*
- *assertNotEqual(a, b)*
- *assertTrue(x)*
- *assertFalse(x)*
- *assertIs(a, b)*
- *assertIsNot(a, b)*
- *assertIsNone(x)*
- *assertIsNotNone(x)*
- *assertIn(a, b)*
- *assertNotIn(a, b)*
- *assertIsInstance(a, b)*
- *assertNotIsInstance(a, b)*

Tests du code - unittest

Dans la classe de tests d'**unittest**, on peut aussi créer une méthode d'instance qui s'exécute avant chacun des tests de la classe (méthode **setUp(self)**) ou après chacun des tests (méthode **tearDown(self)**).

```
class SomeTest(unittest.TestCase):  
    def setUp(self):  
        self.mock_data = [1,2,3,4,5]  
  
    def test(self):  
        self.assertEqual(len(self.mock_data), 5)  
  
    def tearDown(self):  
        self.mock_data = []
```

Tests du code - unittest

On peut aussi créer une méthode de classe qui s'exécute une seule fois avant tous les tests de la classe (méthode **setUpClass(cls)**) ou après tous les tests (méthode **tearDownClass(cls)**).

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

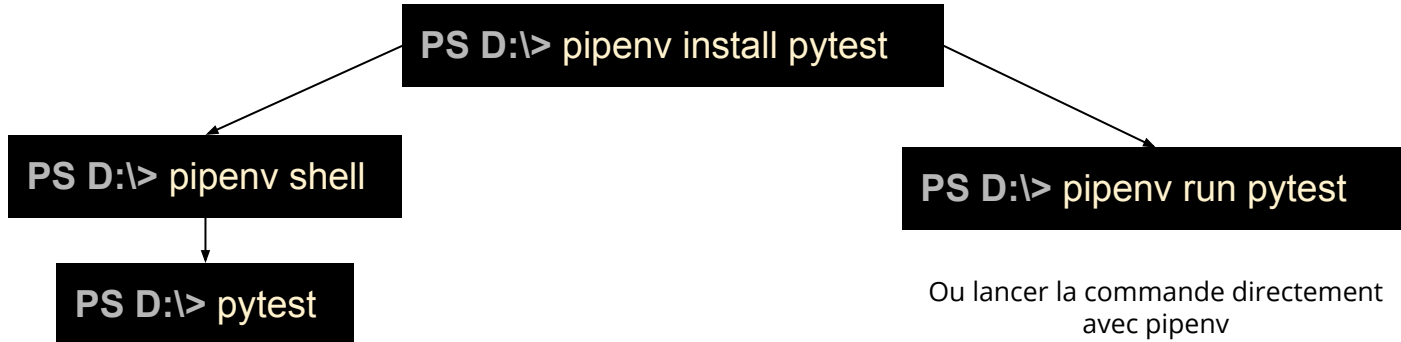
    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

Tests du code - pytest

Une des principales bibliothèques tierces de tests en Python s'appelle **pytest**. Elle détecte les fichiers Python de tests commençant par *test_* ou finissant par *_test*.

Pour commencer à utiliser pytest, il est préférable de l'installer dans un environnement virtuel avec la commande :

Puis activer
l'environnement
virtuel et lancer
la commande



Ou lancer la commande directement
avec pipenv

Tests du code - pytest

A la différence de unittest, pytest s'appuie sur le mot clé **assert** de Python pour réaliser les tests. De plus, on peut créer des fonctions ou des classes de tests avec Pytest.

```
def hello(name):  
    return 'Hello ' + name  
  
def test_hello():  
    assert hello('plb') == 'Hello plb'
```

Tests du code - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/library/unittest.html>
- Documentation **officielle** de Pytest : <https://docs.pytest.org/en/latest/>

Ressource video :

- Socratica - Unit Tests in Python || Python Tutorial || Learn Python Programming (ENG - 08:48) : <https://www.youtube.com/watch?v=1Lfv5tUGsn8&t=308s>
- Corey Schafer - Python Tutorial: Unit Testing Your Code with the unittest Module (ENG - 39:12) : <https://www.youtube.com/watch?v=6tNS--WetLI&t=276s>

Chapitre Fonctionnalités avancées

Débogueur

Débogueur - pdb

Le débogueur permet d'accéder à **beaucoup plus de fonctionnalités** qu'un simple **print()** pour trouver des **bugs** dans un code.

On peut avoir accès aux **états des variables** à un endroit du code et les modifier.

Il existe un débogueur dans la **bibliothèque standard** de Python : il s'appelle **pdb**.

Débogueur - pdb

Avant la version 3.7 de Python, il faut ajouter la **syntaxe** `import pdb;`
`pdb.set_trace()` à l'endroit du code que l'on souhaite examiner.

A partir de la version 3.7 de Python, il suffit d'ajouter la **syntaxe** `breakpoint()`
à l'endroit du code que l'on souhaite examiner.

```
class MaClasse:
    def __init__(self):
        breakpoint()
        self.__attribut_privé = 5
instance = MaClasse()
```

```
-> self.__attribut_privé = 5
(Pdb) █
```

Débogueur - pdb

Dans le débogueur, certaines **commandes** permettent d'**avancer** :

- **n**(ext) : avancement minimal, exécution de la ligne en cours
- **s**(tep) : va jusqu'à la ligne suivante de la fonction en cours
- **c**(ontinue) : va jusqu'au point d'arrêt suivant
- **unt**(il) : va jusqu'au numéro de ligne indiqué

```
-> self.__attribut_prive = 5  
(Pdb) unt 25
```

Débogueur - pdb

Le débogueur fonctionne comme un **interpréteur** Python. On peut donc utiliser des fonctions natives comme **dir()** ou **print()**.

```
class MaClasse:  
    def __init__(self):  
        self.value = 5  
        breakpoint()  
instance = MaClasse()
```

```
(Pdb) print(self.value)  
5
```

Débogueur - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/library/pdb.html>

Ressource video :

- London App Développeur - Introduction to the Python Debugger (ENG - 15:18) : https://www.youtube.com/watch?v=7Vmik1M_ry0
- PyCon Australia - "Goodbye Print Statements, Hello Debugger!" - Nina Zakharenko (PyCon AU 2019) (ENG - 29:24) : <https://www.youtube.com/watch?v=HHrVBKZLolg>

Chapitre Fonctionnalités avancées

Interfaces graphiques

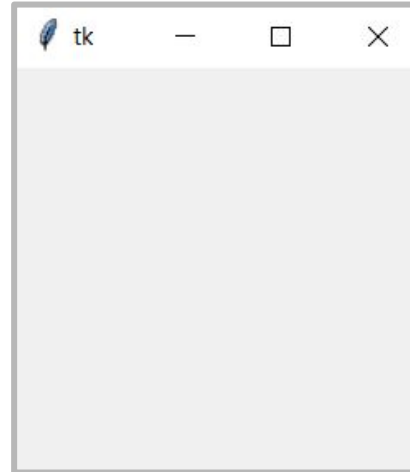
Interfaces graphiques - Tkinter

Tkinter est le paquet de la bibliothèque standard de Python qui permet de créer des interfaces graphiques. Le code suivant permet de créer une structure d'**interface graphique de base** pour insérer des **widgets**.

```
import tkinter as tk

class Application(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)

app = Application()
app.mainloop()
```



Interfaces graphiques - Tkinter

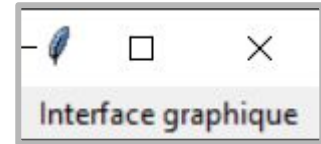
On peut maintenant ajouter un **widget** à la fenêtre comme par exemple un **label**.

```
import tkinter as tk

class Application(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self.add_widgets()

    def add_widgets(self):
        self.label = tk.Label(self.master, text="Interface graphique")
        self.label.pack()

app = Application()
app.mainloop()
```



Interfaces graphiques - Tkinter

On peut aussi ajouter un **bouton** lié à une **méthode** ou à une **fonction**.

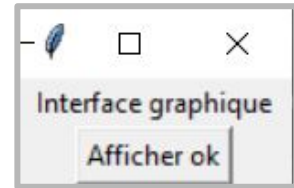
```
import tkinter as tk

class Application(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self.add_widgets()

    def add_widgets(self):
        self.label = tk.Label(self.master, text="Interface graphique")
        self.label.pack()
        self.bouton = tk.Button(self, text="Afficher ok", command=self.print_ok)
        self.bouton.pack()

    def print_ok(self):
        print("ok")

app = Application()
app.mainloop()
```



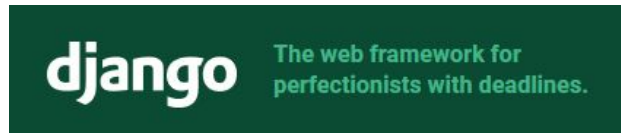
Interfaces graphiques - Tkinter

Voici une liste de quelques widgets supplémentaires utilisables avec Tkinter :

- *Checkbutton* : affiche des cases à cocher.
- *Entry* : demande à l'utilisateur de saisir une valeur / une phrase.
- *Listbox* : affiche une liste d'options à choisir
- *Radiobutton* : implémente des « boutons radio ».
- *Menubutton* et *Menu* : affiche des menus déroulants.
- *Message* : affiche un message sur plusieurs lignes (extensions du *widget* Label).
- *Scale* : affiche une règle graduée pour que l'utilisateur choisisse parmi une échelle de valeurs.
- *Scrollbar* : affiche des ascenseurs (horizontaux et verticaux).
- *Text* : crée une zone de texte dans lequel l'utilisateur peut saisir un texte sur plusieurs lignes
- *Spinbox* : sélectionne une valeur parmi une liste de valeurs.

Interfaces graphiques - web

Une autre façon de créer une interface graphique est de créer une application web. Il existe plusieurs bibliothèques tierces pour créer des **applications web** avec Python. Les deux bibliothèques les plus utilisées sont **Django** et **Flask**.



Interfaces graphiques - Flask

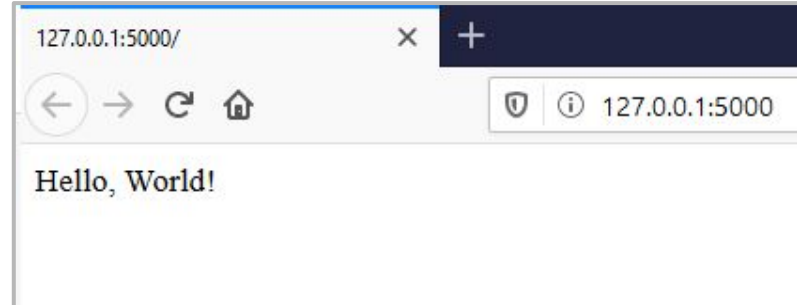
Grâce à *Flask*, en 6 lignes de code, on peut mettre en place une application web minimaliste.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

app.run()
```

```
* Serving Flask app "module2" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [29/May/2020 09:44:09] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [29/May/2020 09:44:09] "GET /favicon.ico HTTP/1.1" 404 -
```



Interfaces graphiques - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/fr/3/library/tkinter.html>
- Documentation **officielle** de Flask : <https://flask.palletsprojects.com/en/1.1.x/>

Ressource video :

- Formation Video - Python #20 - introduction tkinter (FR - 21:41) :
https://www.youtube.com/watch?v=H0BFsl2_St4
- Miguel Grinberg - The Flask Mega-Tutorial :
<https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

Chapitre Fonctionnalités avancées

Bases de données

Base de données - SQLite

SQLite est un moteur de base de données relationnelles stocké dans la fichier (pas de fonctionnement client-serveur comme avec PostgreSQL, MySQL...) La librairie standard de Python possède un module pour travailler directement avec SQLite : il s'agit de *sqlite3*.

Le code de connexion à la base de données est le suivant :

```
import sqlite3
conn = sqlite3.connect('sqlitebdd.db')
```

Base de données - SQLite

Ensuite, on crée un curseur (comme pour les autres systèmes de bases de données) pour exécuter du code SQL :

```
import sqlite3
conn = sqlite3.connect('sqlitebdd.db')
curseur = conn.cursor()
curseur.execute("CREATE TABLE matable (nom text, numero integer)")
conn.commit()
conn.close()
```


Base de données - SQLite

Si vous n'avez pas besoin de persister vos données, vous pouvez les stocker dans la RAM de votre ordinateur avec la syntaxe suivante :

```
import sqlite3
conn = sqlite3.connect(':memory:')
curseur = conn.cursor()
curseur.execute("CREATE TABLE matable (nom text, numero integer)")
conn.commit()
conn.close()
```

Base de données - Base de données SQL

Pour communiquer en Python avec les différentes bases SQL, il faut installer une librairie externe :

Système de gestion de base de données relationnelle	Librairie client de base de données en Python
PostgreSQL	psycopg2
MySQL	python3-mysql.connector
Oracle	cx_Oracle
Maria DB	python3-mysql.connector
Microsoft SQL server	pyodbc

Respect de la
PEP-249

Base de données - Base de données SQL

Pour une utilisation plus haut niveau des bases de données relationnelles, il est aussi possible d'utiliser un **ORM** (Object Relational Mapper) : le plus utilisé en Python est **SQLAlchemy**



Base de données - ressources complémentaires

Ressource texte :

- Documentation **officielle** de Python : <https://docs.python.org/3.8/library/sqlite3.html>

Ressource video :

- Corey Schafer - Python SQLite Tutorial: Complete Overview - Creating a Database, Table, and Running Queries (ENG - 29:48) : <https://www.youtube.com/watch?v=pd-0G0MigUA>



Ressources complémentaires



Ressources complémentaires - base & intermédiaire

MOOC

- Socratica - Python Programming Tutorials :
<https://www.youtube.com/playlist?list=PLi01XoE8jYohWFPpC17Z-wWhPOSuh8Er->
- OpenClassrooms - Apprenez à programmer en Python :
<https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python>

LIVRES

- Fabrizio Romano - Packt Publishing - **Learn Python Programming - Second Edition** - 2018

Ressources complémentaires - avancé

Conférences

- PyCon 2019 : <https://www.youtube.com/channel/UCxs2IIVXaEHHA4BtTiWZ2mQ>
- EuroPython : https://www.youtube.com/channel/UC98CzaYuFNAA_gOINFB0e4Q
- Association Francophone Python :
https://www.youtube.com/channel/UCOT0Jouy4KgGWvRr5Q_Htxw

LIVRES

- Rick van Hattem - Packt Publishing - **Mastering Python** - 2016