

Python pour développeurs C++

Public cible : CIR2

Présentation du cours

- Ce cours s'adresse à des étudiants ayant les bases en algorithmique et en C/C++ (programmation impérative, procédurale et objet).
- L'objectif de ce cours est de donner un aperçu du langage Python.
- Les concepts seront souvent présentés en faisant une comparaison avec C/C++. Mais Python a ses concepts propres, dits *pythoniques*, qu'il faut connaître pour tirer le meilleur parti du langage (et qui sont très utilisés par la communauté).
- La documentation Python officielle présente de manière rigoureuse et exhaustive les différentes API du langage. Elle contient aussi de bons tutoriels et une FAQ très riche.

Qu'est-ce que Python ?

- Python est un langage :
 - Interprété et multi-plateforme
 - Dynamiquement typé
 - Possédant une couche objet et des exceptions
 - Possédant un ramasse-miettes
 - Permettant d'appeler du code en C/C++
- Python est devenu un langage de référence en en data science et en intelligence artificielle (ex : [pandas](#), [scikit-learn](#), [tensorflow](#))
- Python est également utilisé dans les systèmes embarqués (IoT, robotique) dans sa version « allégée » : MicroPython.
- Version actuelle : 3.10



Les bases

Règles d'écriture

- Convention de nommage recommandée ([PEP 8](#)) : snake case
- Pas de ; en fin de ligne
- Les commentaires commencent par #

```
# conversion de temperature
temperature_fahrenheit = (temperature_degrees * 9/5 + 32)
```

- Contrairement à C/C++, les blocs de code ne sont pas délimités par {}, mais définis par : et un niveau d'indentation

C/C++	Python
<pre>if (condition1) { // code du bloc }</pre>	<pre>if condition1: # code du bloc</pre>

Les blocs de code

- Chaque fois que l'on imbrique un bloc de code dans un autre, on augmente le niveau d'indentation
- Le mot-clé `pass` permet d'écrire un bloc vide (sans ce mot-clé, le bloc sera syntaxiquement incorrect)

```
for i in range(n):  
    for j in range(m):  
        if i + j % 2 == 0:  
            pass
```

Attention à l'indentation !

- L'indentation joue donc un rôle majeur en Python : elle identifie le bloc appartient une ligne de code
- Une erreur d'indentation peut radicalement changer le comportement d'un programme !

```
for i in range(1, n):  
    r = i % n  
    print(r)
```

≠

```
for i in range(1, n):  
    r = i % n  
print(r)
```



Pour faire une indentation, ne pas utiliser une tabulation mais 4 espaces (automatiquement fait par la plupart des IDE, et paramétrable).

Types de base

Type	Description	Type C++ similaire	Exemple
bool	Booléen (deux valeurs possibles)	bool	True False
int	Nombre entier (complément à 2, dynamique illimitée)	int sans taille fixe	42
float	Nombre flottant (double précision)	double	1.3
str	Chaîne de caractères	const string	"Hello"
list	Liste (= tableau dynamique)	std::vector	[1, 2, 3, 4, 5]
tuple	Liste non modifiable	const std::vector	(255, 0, 127)
dict	Dictionnaire (= tableau associatif)	std::map	{ "pierre" : 12, "marie" : 16 }

Typage dynamique

- Le type des variables n'est pas déclaré, il change dynamiquement selon le contenu affecté
- `type(v)` renvoie le type de la variable `v`
- `<type destination>(v)` permet de convertir la variable `v` dans le type `destination`

```
v = True
t = type(v)  # bool

v = "42"
t = type(v)  # str

v = int(v)   # conversion vers le type int
t = type(v)  # int
```

Opérateurs

- Mêmes opérateurs mathématiques et logiques qu'en C/C++, auxquels s'ajoute l'opérateur d'exponentiation ** :

```
x += 1          # incrémente x
r = x % 2       # calcule le reste de la division par 2
x3 = x**3       # calcule le cube de x
b = (x == 42)   # test d'égalité => b vaut False
```

- Contrairement au C/C++, l'opérateur / fait une division flottante
→ Un opérateur spécifique fait une division entière : //

```
x = 3/2    # x vaut 1.5
x = 3//2   # x vaut 1
```



Ceci est vrai pour Python 3, mais pas pour Python 2

Entrées/sorties

- La fonction `print()` permet d'afficher un ensemble de valeurs, séparées par des virgules, sur la sortie standard. Par défaut, des espaces entre chaque valeur et un saut de ligne sont insérés à l'affichage.
 - La fonction `input([message])` permet de lire une valeur sur l'entrée standard, en affichant un message optionnel.
- ⚠ Le retour de `input()` est de type `str`. Il faut le convertir en nombre avant d'y appliquer opérations mathématiques.

```
prix_ht = int(input("prix HT ?"))  
prix_ttc = prix_ht * 1.2 # erreur si int() oublié  
print("prix TTC :", prix_ttc)
```


Déclaration de fonctions

- Une fonction est un bloc introduit par le mot-clé `def` :

```
def faire_quelque_chose(param1 [=val1], param2 [=val2], ...):  
    # code de la fonction (indenté)  
    [return valeur_de_retour]
```

- Comme en C++, `vali` est la valeur par défaut pour le paramètre `parami`.
- Si l'instruction `return` est absente, la valeur de retour de la fonction est `None` (modélisant « absence de valeur »).

Appel de fonctions

- En Python, on peut appeler une fonction avec des arguments positionnels, nommés, ou un mélange des deux.
 - ➔ positionnels = appel « classique » : on renseigne la valeur des paramètres dans l'ordre de leur déclaration
 - ➔ nommés : on renseigne le nom et la valeur des paramètres avec la syntaxe `nom=valeur`
- Les arguments nommés de choisir l'ordre/le nombre d'arguments au moment de l'appel d'une fonction.
-  • Si on mélange des arguments positionnels et nommés, les arguments nommés sont nécessairement après les éventuels arguments positionnels.

Appel de fonctions

Exemple d'appels de la fonction :

```
def trier_liste(liste, en_place=true, ordre_ascendant=true):
```

- Arguments positionnels uniquement :

```
trier_liste([2,1,3], true, true)
```

- Changement de l'ordre des paramètres :

```
trier_liste([2,1,3], ordre_ascendant=false, en_place=true)
```

- Omission d'un paramètre:

```
trier_liste([2,1,3], ordre_ascendant=false)
```



Structures de contrôle



Branchements conditionnels, ternaires

C/C++	Python
<pre>if (condition1) { // bloc 1 } else if (condition2) { // bloc 2 } else { // bloc 3 }</pre>	<pre>if condition1: # bloc 1 elif condition2: # bloc 2 else: # bloc 3</pre>
<pre>x = condition ? a : b;</pre>	<pre>x = a if condition else b</pre>

Boucles

C/C++	Python
<pre>while(condition) { // bloc }</pre>	<pre>while condition: # bloc</pre>
<pre>for(i = 0; i < n; i++){ // bloc }</pre>	<pre>for i in range(n): # bloc</pre>
<pre>for(i = a; i < b; i++){ // bloc }</pre>	<pre>for i in range(a, b): # bloc</pre>
<pre>for(i = a; i < b; i += step){ // bloc }</pre>	<pre>for i in range(a, b, step): # bloc</pre>

 **Attention** : la borne supérieure des range est **EXCLUE**



Les séquences



Les séquences

- Une séquence est une **suite ordonnée** d'éléments. Chaque élément est identifié par un **indice**.
- Les séquences ont en commun les opérations suivantes :
 - ➔ Consultation de la taille de la séquence avec la fonction `len()`
 - ➔ Manipulation d'un élément ou d'une portion (appelée tranche) de la séquence avec l'opérateur `[]`
 - ➔ Test d'appartenance ou balayage d'éléments de la séquence avec l'opérateur `in`.
 - ➔ Redéfinition des opérateurs `+` (concaténation) et `*` (répétition)
- Exemples de séquences :
 - ➔ Les listes et les tuples
 - ➔ Les chaînes de caractères

Les listes

- Type Python : `list`
- Une liste est un tableau dynamique, pouvant stocker des éléments n'importe quel type (le type des éléments n'a pas à être homogène)

```
l = [42, 1337, 404]
b = 42 in l  # test d'appartenance => b vaut True
x = l[1]     # n vaut 1337
n = len(l)   # l vaut 3
```

Parcours avec indice

```
for i in range(len(l)):
    x = l[i]
    print(x)
```

Parcours sans indice (for each)

```
for x in l:
    print(x)
```



Equivalent C++ : `for(int x : l)`

Les listes

- Le comportement de + et * est redéfini pour les listes :

```
notes = [12, 15]
notes2 = notes + [3, 20]    # concaténation => notes2 = [12, 15, 3, 20]
notes3 = notes * 3         # répétition => notes3 = [12, 15, 12, 15, 12, 15]
```

- Le type list contient de nombreuses méthodes de manipulation, par exemple :

```
l = [42, 1337]
l.append(404)    # ajoute 404 en fin de l => l = [42, 1337, 404]
n = l.pop()     # supprime et renvoie dernière la case
                # => l = [42, 1337] et n = 404
```

- Consultez la [documentation](#) pour une liste exhaustive des méthodes

Indices négatifs

- En Python, il est possible d'utiliser des indices négatifs, pour parcourir les valeurs de droite à gauche :
 - ➔ `l[-1]` permet d'accéder à la dernière case de la liste `l`
 - ➔ `l[-2]` permet d'accéder à l'avant-dernière case de la liste `l`
 - ➔ Et ainsi de suite...
- Si la liste contient `n` éléments, les indices négatifs évoluent de `-1` (dernière case) à `-n` (première case).

```
notes = [12, 15, 8, 20, 3]
derniere_note = notes[-1]    # derniere_note vaut 3
premiere_note = notes[-5]    # premiere_note vaut 12
```

Les tranches

- La syntaxe [debut:fin:pas] permet d'extraire des **tranches** de liste (*slices* en anglais). Attention : l'indice de fin est **exclu** !
- On peut omettre certaines valeurs ou utiliser des valeurs négatives.
- Quelques exemples :

```
l = [10, 20, 30, 40, 50]

l[:-1]      # [10, 20, 30, 40]
l[1:-1]     # [20, 30, 40]
l[::2]      # [10, 30, 50]
l[::-1]     # [50, 40, 30, 20, 10]
l[::-2]     # [50, 30, 10]
l[:]        # [10, 20, 30, 40, 50]
l[1:3]      # [20, 30]
l[-3:-1]    # [30, 40]
l[:3]       # [10, 20, 30]
l[3:]       # [40, 50]
```

Les tuples

- Type Python : `tuple`
- Un tuple est une liste non modifiable : son contenu est figé à l'initialisation
- Il se manipule de la même manière qu'une liste, sauf qu'on ne peut le manipuler qu'en lecture.
- On crée un tuple en indiquant les éléments entre (), séparés par des , :

```
pixel = (255, 128, 0)
```

- On peut égaliser deux tuples, ce qui permet notamment de décomposer le contenu d'un tuple en plusieurs variables :

```
(r, g, b) = pixel    # r vaut 255, g vaut 128 et b vaut 0
```


Sources d'erreurs avec les tuples

- Les parenthèses d'un tuple sont optionnelles. On peut par exemple écrire :

```
t = 1, 2, 3
```

- Si un tuple ne contient qu'un élément, il faut le terminer par une ,

```
t = (1,)
```

- Attention** : ces points peuvent être source d'erreur !

```
a = 1,5    # a est le tuple (1,5) et non le nombre 1.5 !  
b = (1)    # b est l'entier 1 et non le tuple (1,) !
```

Utilisations pratiques des tuples

Les tuples sont fréquemment utilisés pour :

- Echanger de deux valeurs :

```
a, b = b, a # intervertit les valeurs de a et b
```

- Retourner de plusieurs valeurs dans une fonction :

```
def calculer_statistiques(...):  
    ...  
    return mini, maxi, moyenne
```

Les chaînes de caractères

- Type Python : `str`
- On définit une chaîne de caractères en écrivant une suite de caractères entre guillemets simples `'` ou doubles `"` (aucune différence)
- On peut comparer deux chaînes de caractères avec `==`

```
texte1 = "une chaîne de caractères"  
texte2 = 'une autre'  
textes_egaux = (texte1 == texte2) # vaut False
```

- En tant que séquences, tout ce qui a été présenté pour les listes (opérateurs `in`, `+` et `*`, indices négatifs, tranches, ...) est utilisable sur les chaînes.
- Le type `str` inclut de nombreuses méthodes de manipulation (recherche, découpage, remplacement...). Consultez la [documentation](#) pour plus de détails.

Les chaînes de caractères sont immuables

- Les chaînes de caractères font partie des types dits **immuables** (c'est-à-dire **non modifiables**) en Python

```
nom = "DUPOND"  
nom[-1] = "T" # erreur
```

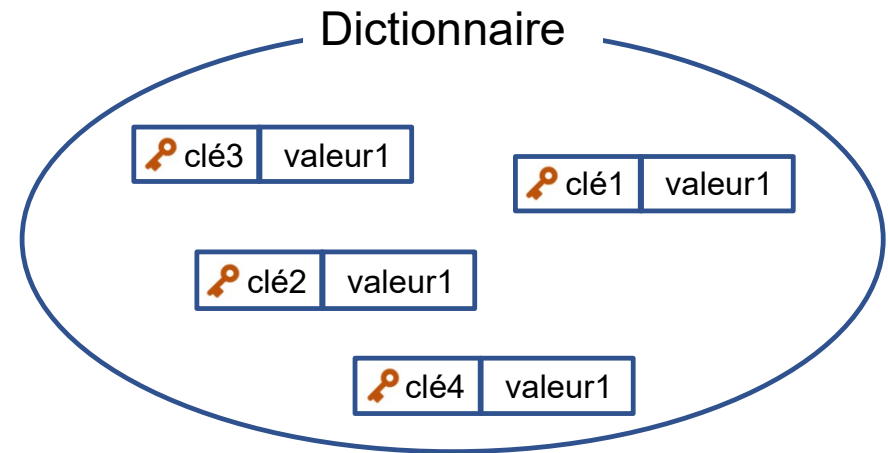
- Conséquences :
 - ➔ Les fonctions qui réalisent des traitements sur des chaînes ne modifient pas la chaîne originale mais créent une nouvelle chaîne
 - ➔ Si on veut modifier un ou plusieurs caractères dans une chaîne, il faut passer par l'intermédiaire d'une liste.



Les dictionnaires

Qu'est ce qu'un dictionnaire ?

- Type Python : dict
- Un dictionnaire permet d'associer une **valeur** à un identifiant unique appelé **clé**.
- Les clés et les valeurs peuvent être de n'importe quel type.
- Dans d'autres langages, les dictionnaires sont appelées « tableaux associatifs » ou « tables de hachage ». En C++, le type équivalent serait `std::map`.
- Les dictionnaires de base ne sont pas ordonnés. L'ordre des clés n'est donc pas garanti. (Mais le type [OrderedDict](#) existe au besoin).



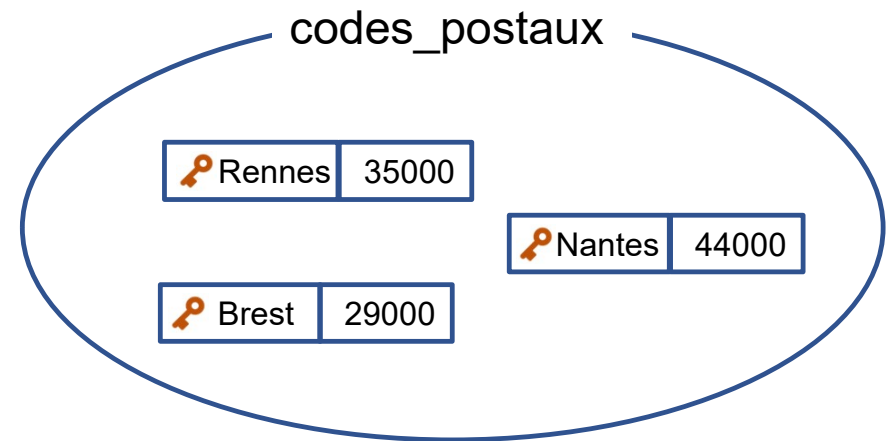
Créer un dictionnaire

- Pour créer un dictionnaire, il faut écrire des couples clé : valeur à l'intérieur de {}, séparés de ,
- Exemple : Association ville → code postal

```
codes_postaux = {  
    "Nantes" : 44000,  
    "Brest" : 29000,  
    "Rennes" : 35000  
}
```



Ici les clés sont des chaînes de caractères et les valeurs sont des entiers.



- {} représente un dictionnaire vide.

Lire et écrire des données dans un dictionnaire

- Pour lire ou écrire des données dans un dictionnaire, on utilise [] comme pour une liste (mais avec une clé au lieu d'un indice).
- Lecture d'une valeur :

```
code_postal_nantes = codes_postaux["Nantes"]  
print(code_postal_nantes)
```



Affichage

44000

- Modification d'une valeur :

```
codes_postaux["Brest"] = 29200 # c'est mieux !  
print(codes_postaux)
```



Affichage

```
{'Nantes': 44000,  
'Brest': 29200,  
'Rennes': 35000}
```


Vérifier l'existence d'une clé

- Si vous tentez de lire une valeur associée à une clé inexistante, vous obtiendrez une erreur :

```
print(codes_postaux["Caen"]) # KeyError: 'Caen'
```

- Pour éviter ceci, on peut tester l'existence d'une clé avec l'opérateur in :

```
print("Brest" in codes_postaux) # affiche True  
print("Caen" in codes_postaux) # affiche False
```



Pour avoir un code robuste aux erreurs, il est recommandé de tester l'existence des clés avant de lire une donnée dans un dictionnaire (surtout quand les données proviennent d'une source externe)

Ajout d'un élément dans un dictionnaire

- La syntaxe est la même pour modifier ou ajouter un élément dans un dictionnaire
- Quand on écrit une valeur dans un dictionnaire, la clé est automatiquement créée si elle n'existait pas :

```
print("Avant", codes_postaux)  
  
codes_postaux["Caen"] = 14000  
  
print("Après", codes_postaux)
```



Affichage

```
Avant {'Nantes': 44000, 'Brest': 29200,  
      'Rennes': 35000}  
Après {'Nantes': 44000, 'Brest': 29200,  
      'Rennes': 35000, 'Caen': 14000}
```

Itérer sur les clés

- `d.keys()` permet d'itérer sur les clés du dictionnaire `d`
- Itérer sur `d.keys()` ou sur `d` revient au même.
- Les deux codes suivants sont équivalents :

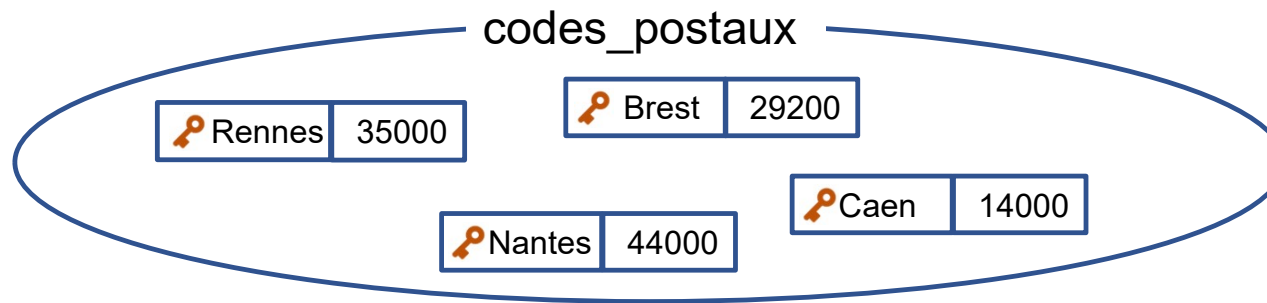
```
for ville in codes_postaux.keys():  
    print(ville)
```

```
for ville in codes_postaux:  
    print(ville)
```



Affichage

```
Nantes  
Brest  
Rennes  
Caen
```



Itérer sur les valeurs

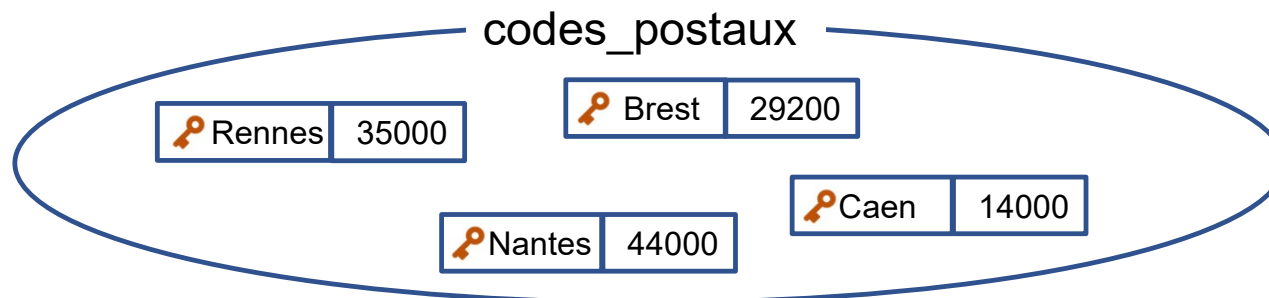
- `d.values()` permet d'itérer sur les valeurs du dictionnaire `d`
- Exemple :

```
for code_postal in codes_postaux.values():  
    print(code_postal)
```



Affichage

```
44000  
29200  
35000  
14000
```



Itérer sur les éléments

- `d.items()` permet d'itérer sur les éléments du dictionnaire `d`.
- Chaque élément est un tuple (clé, valeur)
- Exemple :

```
for ville_code in codes_postaux.items():
    print(ville_code)
```



Affichage

```
('Nantes', 44000)
('Brest', 29200)
('Rennes', 35000)
('Caen', 14000)
```





Les fichiers

Etapes de manipulation un fichier

1. Ouvrir le fichier
2. Lire/écrire des données
3. Fermer le fichier



Ne pas oublier cette étape ! Si un fichier n'est pas libéré, il ne peut pas être ouvert par d'autres programmes.

Exemple

data.txt

```
ligne 1  
ligne 2  
ligne 3
```

Programme

```
fichier = open("data.txt", "r") # étape 1  
contenu = fichier.read()       # étape 2  
print(contenu)  
fichier.close()                # étape 3
```



Affichage

```
ligne 1  
ligne 2  
ligne 3
```


Modes d'ouverture

Mode	r	r+	w	w+	a	a+
Lecture du fichier	✓	✓		✓		✓
Ecriture dans le fichier		✓	✓	✓	✓	✓
Création du fichier			✓	✓	✓	✓
Remise à zéro du fichier			✓	✓		
Position initiale en début de fichier	✓	✓	✓	✓		
Position initiale en fin de fichier					✓	✓

Méthodes de lecture d'un fichier

Les méthodes suivantes permettent de lire des données dans un fichier `f`

Appel	Action
<code>f.read()</code>	Lit l'intégralité du fichier <code>f</code> et renvoie le contenu sous la forme d'une chaîne de caractères
<code>f.read(n)</code>	Lit <code>n</code> caractères dans le fichier <code>f</code> et les renvoie la sous forme d'une chaîne
<code>f.readline()</code>	Lit une ligne dans le fichier <code>f</code> et la renvoie sous la forme d'une chaîne de caractères. Le caractère de fin de ligne <code>"\n"</code> est conservé. Renvoie une chaîne vide si la fin du fichier est atteinte.
<code>f.readlines()</code>	Lit toutes les lignes du fichier <code>f</code> et les renvoie sous la forme d'une liste de chaînes de caractères (une chaîne par ligne)

Méthodes d'écriture dans un fichier

Appel	Action
<code>f.write(s)</code>	Ecrit le contenu de la chaîne <code>s</code> dans le fichier <code>f</code>
<code>f.writelines(l)</code>	Ecrit le contenu des chaînes contenues dans la liste <code>l</code> dans le fichier <code>f</code>

→ Exemple :

Programme

```
fichier = open("hello.txt", "w")  
fichier.write("Hello, World!")  
fichier.close()
```



hello.txt

Hello, World!



Dans cet exemple, le mode `w` est utilisé, ce qui implique une création automatique du fichier `hello.txt` et/ou sa remise à zéro.

Libération automatique de ressources avec `with`

- Si un code utilise une ressource, on peut le placer dans un bloc `with`
- La ressource acquise avec `with` sera automatiquement libérée à la fin du bloc

```
with {acquisition ressource} as nom_ressource :  
    # utilisation de la ressource  
  
# la ressource est libérée ici
```

- Dans le cas d'un fichier :
 - ➔ Acquisition de la ressource = ouverture du fichier
 - ➔ Libération de la ressource = fermeture du fichier

Fermeture automatique d'un fichier

Sans with

```
fichier = open("data.txt", "r")  
contenu = fichier.read()  
print(contenu)  
fichier.close()
```



Avec with

```
with open("data.txt", "r") as fichier:  
    contenu = fichier.read()  
    print(contenu)
```



En utilisant `with`, il n'est plus nécessaire d'appeler `close()` : ceci est fait automatiquement



Les modules

Qu'est ce qu'un module ?

- Un module regroupe des éléments (fonctions, variables) d'une même thématique
 - ➔ Par exemple, le module math contient un ensemble de fonctions et de constantes mathématiques
- Les modules permettent :
 - ➔ De structurer le code d'un projet en sous-parties
 - ➔ De créer des « bibliothèques » réutilisables
- Python intègre quelques modules correspondant à des traitements de base.
- Mais la popularité de Python tient beaucoup à son large choix de modules externes, en particulier en data science et en intelligence artificielle.

Importer un module

Pour utiliser les éléments d'un module, vous pouvez :

1. Importer le module entier :

```
import nom_du_module
```

Dans ce cas, il faudra préfixer tous les éléments de ce module par le nom du module

Exemple :

```
import math  
print( math.cos(42) )  
print( math.exp(3) )
```


Importer un module

Pour utiliser les éléments d'un module, vous pouvez :

1. Importer le module entier
2. Importer le(s) élément(s) utilisé(s) dans le module :

```
from nom_du_module import element1 (,element2, element3, ...)
```

On utilise alors directement ces éléments, sans préfixe.

Exemple :

```
from math import cos, exp  
print( cos(42) )  
print( exp(3) )
```

Ne pas utiliser `import *`

Dans certains codes existants, il pourra vous arriver de voir des personnes importer tous les éléments d'un module en utilisant `*`. Par exemple :

```
from math import *
```

Ceci n'est **PAS** une bonne pratique, car :

- On ne voit pas explicitement ce qui est importé (utilise-t-on une seule fonction de `math` ou 100 ? Lesquelles ?)
- Ceci augmente considérablement le risque de **conflit de noms** avec des variables d'autres modules ou de votre propre code. Exemple :

```
e = 42  
from math import *  
print(e) # n'affiche PAS 42
```

Les alias

- Les alias permettent d'importer un module ou un de ses éléments sous un autre nom
- Ceci permet par exemple de raccourcir un nom ou de résoudre un conflit de nom
- Certains modules sont traditionnellement importés avec un alias, notamment NumPy et matplotlib :

```
import numpy as np  
import matplotlib.pyplot as plt
```

Ces conventions sont largement utilisées dans la communauté Python.

Les paquets et pip

- Les modules externes font partie de **paquets** (*packages*) référencés dans un dépôt public appelé **PyPI** (pour *Python Package Index*)
- Un outil appelé **gestionnaire de paquets** permet de télécharger et d'installer un paquet par son nom
- Le gestionnaire de paquets de Python s'appelle **pip** (pour *pip installs packages*)

Installer un paquet

Pour installer un paquet nommé « nom-du-paquet » :

1. Ouvrir une invite de commande (cmd sous Windows, Terminal sur Mac)
2. Taper la commande :

```
pip install nom-du-paquet
```

pip va alors installer le paquet demandé, mais également tous les paquets utilisés par ce paquet (les **dépendances**)



Les IDE comme PyCharm permettent d'installer des paquets en mode « clic-clic », mais ceci cache l'exécution de telles commandes en coulisses.

Créer ses propres modules

- Un module Python est simplement un fichier du même nom
- Donc créer un module m consiste simplement à créer un fichier m.py !
- Exemple (dans le même répertoire) :

arithmetique.py

```
def pgcd(a, b):  
    ...  
  
def nombres_premiers(n):  
    ...  
  
def est_parfait(n) :  
    ...
```

main.py

```
import arithmetique as arithm  
  
a = 15  
b = 21  
pgcd_ab = arithm.pgcd(a, b)  
print("Le pgcd de", a, "et", b, "est", pgcd_ab)
```

Attention au nom de vos fichiers !



Ne jamais appeler vos fichiers comme des modules existants : ces fichiers vont masquer ces modules !

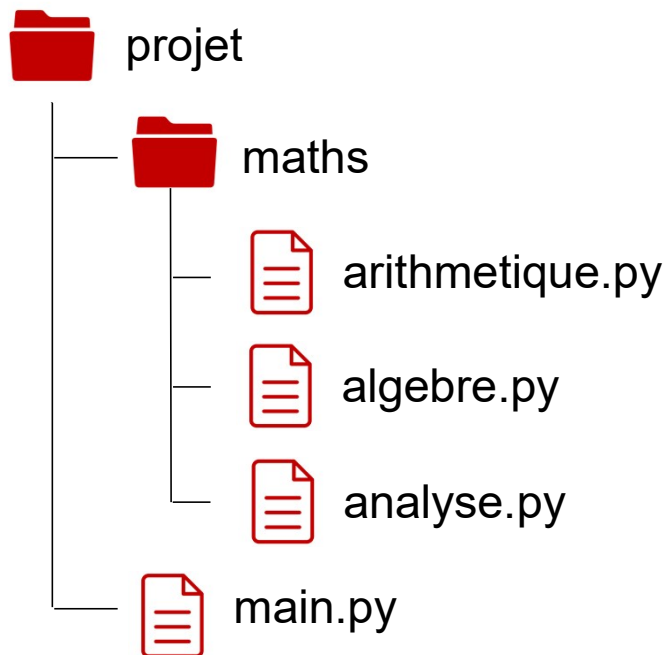
Exemple : imaginons que vous voulez importer le module Python « math » dans un fichier appelé math.py :

math.py

```
import math # importe le fichier math.py (donc lui-même) -> ERREUR
```

Les paquets (*packages*)

- Les paquets sont des regroupements de modules
- Sur le disque, un (sous-)répertoire est associé à chaque (sous-)paquet
- Pour importer le module `a/b/c/m.py`, on écrit : `import a.b.c.m`
- Exemple :



main.py

```
import maths.arithmetique as arithm

a = 15
b = 21
pgcd_ab = arithm.pgcd(a, b)
print("Le pgcd de", a, "et", b, "est", pgcd_ab)
```


Conditionner l'exécution du code dans un module

- Quand on importe un module, l'ensemble de son code est exécuté
- Si vous avez écrit du code de test à la fin d'un module, et que vous importez ce module dans un programme, alors vous verrez l'exécution de ces tests !
- Pour éviter ceci, il est possible d'indiquer à Python de n'exécuter du code que si le module est explicitement lancé (et non simplement importé)
- La syntaxe est la suivante :

```
if __name__ == "__main__": # vrai si le module est lancé  
    # code
```



POO en Python



Définition d'une classe en Python

- Une classe se définit par avec le mot-clé `class`
- Contrairement au C++, une classe ne peut avoir qu'un seul constructeur. Il s'appelle `__init__()`
- L'instance courante (`this` en C++) est le paramètre de toutes les méthodes, traditionnellement appelée `self`
- Les membres sont rendus privés en les préfixant par `__`



Concrètement, un membre `__m` d'une classe `C` est automatiquement renommé en `_C__m` à l'exécution, ce qui décourage son accès direct (mais reste possible si on connaît ce mécanisme). Les membres apparaîtront de cette façon dans un débogueur.

Exemple : classe Fraction

C++ (méthodes *inline*)

```
class Fraction
{
private:
    short num;
    short den;

public:
    Fraction(short num, short den){
        this->num = num;
        this->den = den;
    }

    short getNumerator() const {
        return this->num;
    }

    short setNumerator(short num){
        this->num = num;
    }

    ...
};
```

Python

```
class Fraction:

    def __init__(self, num, den):
        self.__num = num
        self.__den = den

    def getNumerator(self):
        return self.__num

    def setNumerator(self, num):
        self.__num = num

    ...
```

Importation et instantiation

- Il est recommandé de déclarer chaque classe dans un fichier séparé du même nom. Par exemple Fraction.py pour la classe Fraction
- Avec cette convention, on importe une classe C avec la syntaxe suivante :

```
from C import C
```



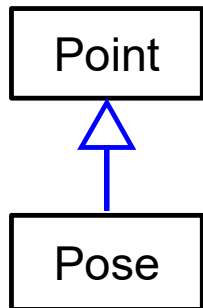
- On instancie une classe appelant le constructeur, sans mot-clé particulier (**pas de new comme en C++**). Exemple avec la classe Fraction :

```
from Fraction import Fraction  
f = Fraction(1, 2)
```

L'héritage en Python

- On modélise que B hérite de A par la syntaxe suivante : `class B(A)`
- **Rappel** : Le constructeur de B ne doit initialiser que les membres spécifiques à B. L'initialisation des membres hérités doit se faire par **délégation** au constructeur de A.
- Pour appeler une méthode `f(p1, ... pn)` de A depuis B (en particulier le constructeur), on peut utiliser d'une des syntaxes suivantes :
 - ➔ `A.f(self, p1, ..., pn)`
 - ➔ `super().f(p1, ..., pn)`
- On peut faire de l'héritage multiple en Python : il suffit d'indiquer plusieurs classes entre parenthèses : `class B(A, C, D)`

Exemple d'héritage



```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
    ...
```

Implémentation 1

```
class Pose(Point):
    def __init__(self, x, y, theta):
        super().__init__(x, y)
        self.__theta = theta
    ...
```

Implémentation 2

```
class Pose(Point):
    def __init__(self, x, y, theta):
        Point.__init__(self, x, y)
        self.__theta = theta
    ...
```



C'est bien la classe Point qui initialise x et y, par délégation.

Les exceptions

- Le mot-clé **raise** permet de lancer une exception
- La gestion des exceptions se fait avec un bloc **try/except**, similaire au bloc try/catch en C++. Comme en C++, les gestionnaires doivent être écrits de l'exception la plus spécifique à la plus générale.

```
def diviser(a, b):  
    if(b == 0):  
        raise RuntimeError("b est nul")  
    return a / b
```

```
try:  
    d = diviser(1, 0)  
except RuntimeError as re:  
    print(re)  
except Exception as e:  
    print("Gestionnaire générique")
```

- Comme C++, Python possède toute une hiérarchie d'exceptions standard décrites [ici](#).



Qt en Python

- Il existe plusieurs portages de Qt en Python, dont [PyQt](#) et [PySide](#). Ils exploitent la capacité de Python d'appeler du code C++.
- Nous allons utiliser PyQt 6 en TP.
- PyQt est structuré en modules. Nous allons principalement utiliser :
 - QtWidgets pour les widgets
 - QtGui pour tout ce qui est pas widget mais utile dans une interface graphique, par exemple : les couleurs, images, icônes, polices... (mais aussi les événements)
 - Éventuellement QtCore pour les fonctions/classes utilitaires

Rappels : widgets et layouts

- Une interface graphique est un ensemble de **widgets**.
- Les widgets peuvent être inclus les uns dans les autres : ceci permet de structurer l'interface en groupes (par exemple un formulaire, une barre d'outils...). On parle alors de widgets parent/enfant.
- Les widgets ne doivent pas être placés « à la main » (avec une position et une taille « en dur »), mais à l'aide de **layouts**.
- Les layouts réorganisent automatiquement les widgets enfants en fonction du widget parent (suite à un redimensionnement du widget parent par exemple)

Etapes de construction d'une interface graphique

1. Création d'un widget principal (typiquement, un panneau occupant toute la fenêtre)
2. Création d'un layout correspondant au premier niveau de structuration de l'interface (ligne, colonne, grille, ...)
3. Création des widgets enfants
4. Ajout des widgets enfants au layout
5. Affectation du layout au widget principal



Un widget enfant peut lui-même contenir des widgets, dans ce cas on réapplique la procédure

```
from PyQt6.QtWidgets import QLabel, QWidget, QPushButton, \
QApplication, QVBoxLayout
app = QApplication([])
```

```
mainWidget = QWidget()
```

Etape 1

```
layout = QVBoxLayout()
```

Etape 2

```
label = QLabel("Ceci est un QLabel")
button = QPushButton("Ceci est un QPushButton")
```

Etape 3

```
layout.addWidget(label)
layout.addWidget(button)
```

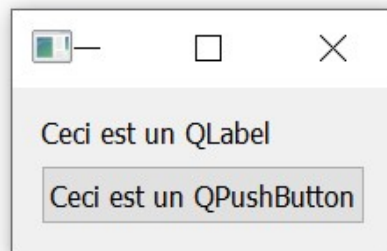
Etape 4

```
mainWidget.setLayout(layout)
```

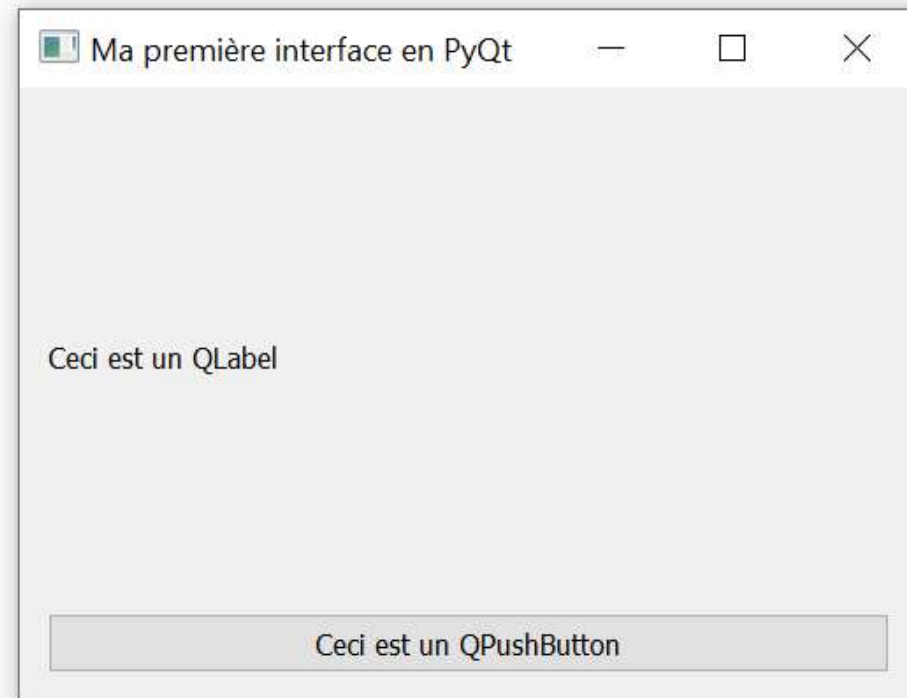
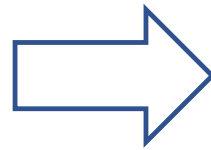
Etape 5

```
mainWidget.setWindowTitle("Ma première interface en PyQt")
mainWidget.show()
app.exec() ← Lance la boucle d'attente d'événements (bloquant)
```

Etapes de construction d'une interface graphique

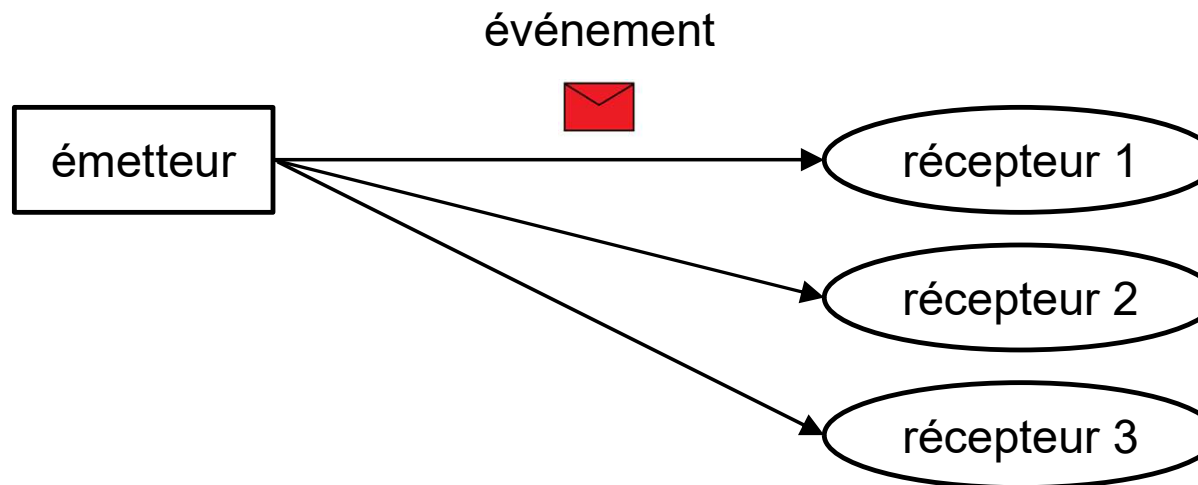


Redimensionnement



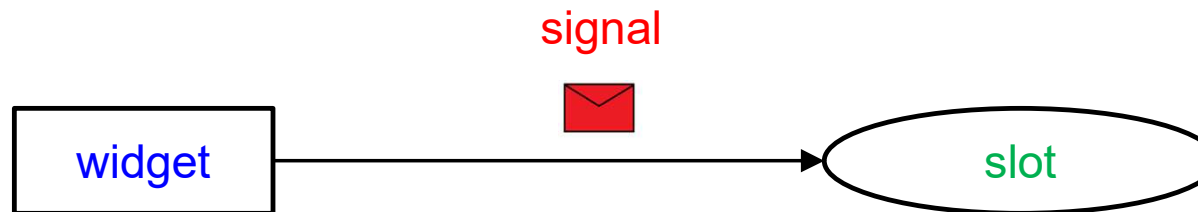
Rappels : programmation événementielle

- Dans le cadre général de la programmation événementielle, un événement est un message émis par une entité du programme pour informer d'un changement de son état, qui peut être écouté par d'autres entités.
- Plusieurs entités peuvent écouter le même événement.



Rappels : signaux et slots

- Dans le cadre particulier des interfaces graphiques en Qt :
 - ➔ Les émetteurs sont la plupart du temps des **widgets**
 - ➔ Les événements sont appelés **signaux**, et les récepteurs des **slots**
 - ➔ Les slots sont des fonctions



- Ecouter un événement consiste à **connecter** un signal à un slot

Connexion signal-slot en PyQt

Il y a deux syntaxes possibles en PyQt pour connecter un signal à un slot :

- La syntaxe « traditionnelle » (Qt < 5) :

```
QObject.connect(widget, SIGNAL("signal()"), slot)
```

→ Dans cette syntaxe, le signal est une **chaîne de caractères** contenant des ()

- La nouvelle syntaxe (Qt ≥ 5) – *recommandée*

```
widget.signal.connect(slot)
```

→ Dans cette syntaxe, le signal est une **méthode** de widget.

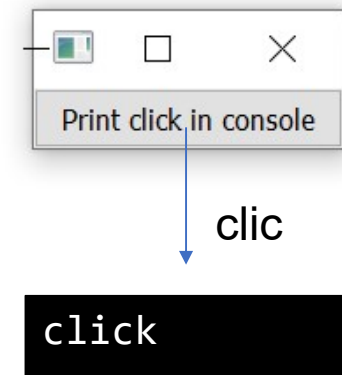
Exemple (nouvelle syntaxe)

Le code suivant permet d'afficher « click » dans la console à chaque fois que l'utilisateur clique sur le bouton de l'interface.

```
from PyQt6.QtWidgets import QApplication, QPushButton

def buttonClicked():
    print("click")

app = QApplication([])
button = QPushButton("Print click in console")
button.clicked.connect(buttonClicked)
button.show()
app.exec()
```



Ecrire un code orienté objet

- Dans le cadre de la Programmation Orientée Objet, il est recommandé de :
 - Implémenter une classe héritant d'un widget Qt existant
 - Définir les slots en tant que méthodes de cette classe
 - Connecter les signaux aux slots dans le constructeur de cette classe
- Avantages :
 - La connexion des slots est faite au sein du widget, ce qui est plus simple pour l'utilisateur, qui a juste à l'instancier
 - Comme le slot est une méthode du widget, on a facilement accès à ses attributs hérités

Exemple : bouton personnalisé

```
from PyQt6.QtWidgets import QApplication, QPushButton

# définition du bouton personnalisé
class MyButton(QPushButton):

    def __init__(self, text):
        super().__init__(text)
        self.clicked.connect(self.buttonClicked)

    def buttonClicked(self):
        print("click")

# utilisation
app = QApplication([])
button = MyButton("Print click in console") # c'est tout !
button.show()
app.exec()
```



Annexes

Compréhensions de liste

- Les compréhensions de liste (list comprehension) permettent de générer une liste à partir d'une autre, avec un code très concis.
- Syntaxe générale :

```
new_list = [function(item) for item in list if condition(item)]
```

- Exemple : sélection des nombres pairs dans une liste :

Sans	Avec
<pre>l_pair = [] for x in l: if l % 2 == 0: l_pair.append(x)</pre>	<pre>l_pair = [x for x in l if x % 2 == 0]</pre>



A utiliser avec parcimonie !

Chaînes « brutes » (*raw strings*)

- Dans les chaînes classiques, le caractère `\` peut introduire une séquence d'échappement (tabulation, saut de ligne, etc.)

```
print("Enregistré dans c:\toto")
```



```
Enregistré dans c:  oto
```

- Si on souhaite désactiver ce comportement, il faut déclarer la chaîne comme étant « brute » en la préfixant avec la lettre `r` (comme *raw*)

```
print(r"Enregistré dans c:\toto")
```



```
Enregistré dans c:\toto
```

→ `\t` n'est pas interprété comme une tabulation

Chaînes formatées

- On injecte des valeurs dans une chaîne en la préfixant de la lettre **f**
- Le nom des variables à injecter est alors indiqué entre {}

```
ville = "Nantes"  
nb_habitants = 325000  
chaîne_formatee = f"La ville de {ville} a {nb_habitants} habitants"  
print(chaîne_formatee)
```

Affichage 

La ville de Nantes a 325000 habitants



Le concept et la syntaxe sont similaires aux [gabarits de chaînes](#) en JavaScript

Concepts avancées de POO

- Les classes abstraites n'existent pas de base en Python, mais on peut les définir en utilisant le module [ABC](#) (Abstract Base Class).

```
class Forme(ABC):  
    ...  
    @abstractmethod  
    def calculer_aire(self):  
        ...
```

Concepts avancées de POO

- Les classes abstraites n'existent pas de base en Python, mais on peut les définir en utilisant le module [ABC](#) (Abstract Base Class).
- Les [propriétés](#) permettent d'appeler automatiquement les accesseurs avec la syntaxe d'un accès direct.

```
class Fraction:
    ...
    @property
    def numerator(self):
        return self.__num

    @numerator.setter
    def numerator(self, num):
        self.__num = num
```

```
f = Fraction(1,2)

# appelle automatiquement self.numerator(3)
f.numerator = 3
```

Chaînes sur plusieurs lignes

- Il est possible d'écrire des chaînes sur plusieurs lignes en utilisant 3 guillemets
- Exemple :

```
html = """
<html>
    <body>
        <title> Hello ! </title>
    </body>
</html>
"""
```

- Utilisations :
 - ➔ Écrire des données structurées comme XML ou JSON
 - ➔ Documenter des éléments de code avec des *docstrings*

Docstrings

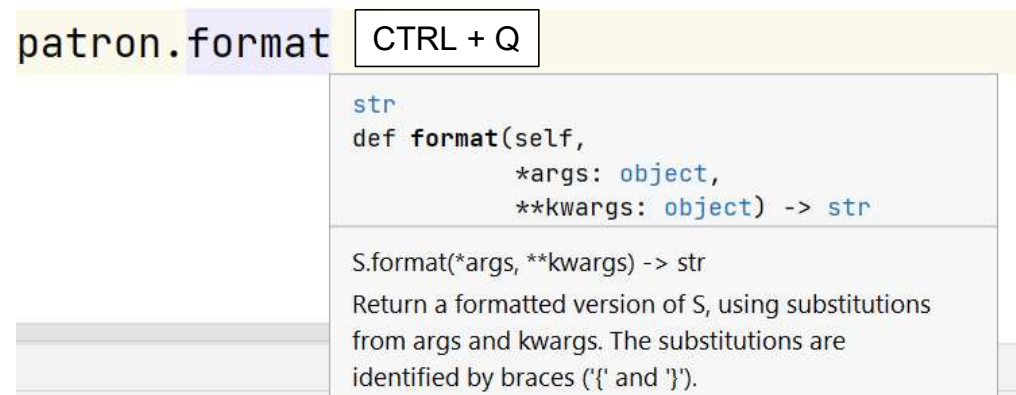
- Les *docstrings* sont utilisées pour documenter les fonctions Python de base ou provenant de modules externes
- Exemple : `format()`

```
def format(self, *args, **kwargs): """
    S.format(*args, **kwargs) -> str

    Return a formatted version of S, using substitutions from args
    and kwargs.
    The substitutions are identified by braces ('{' and '}').
    """
    ...
```

Docstrings

- Les *docstrings* sont lues par les IDE et peuvent être affichées dans un encadré au cours de la saisie
 - Exemple : dans PyCharm, appuyer sur CTRL + Q



- On peut également générer automatiquement une documentation HTML à partir des docstrings. Plus d'informations [ici](#).