

Cégep Limoilou- Département d'informatique

# Notes de cours

420-SF3-RE Développement d'applications dans un  
environnement graphique



Julien Brunet  
Automne 2025

## Table des matières

|         |   |    |
|---------|---|----|
| 1.      | Généralités sur le python .....                               | 5  |
| 1.1.    | Python, un bref historique .....                              | 5  |
| 1.2.    | Python un langage interprété .....                            | 5  |
| 1.3.    | Python un langage orienté vers les Sciences .....             | 6  |
| 2.      | Base de programmation en python.....                          | 7  |
| 2.1.    | Un exemple à la ligne de commande .....                       | 7  |
| 2.2.    | Pycharm, notre environnement de développement .....           | 7  |
| 2.3.    | Les fichiers .py.....   | 8  |
| 2.4.    | Les types et les variables en python .....                    | 9  |
| 2.5.    | Les fonctions en Python .....                                 | 11 |
| 2.6.    | Les structures conditionnelles en Python .....                | 12 |
| 2.7.    | Les boucles en Python .....                                   | 13 |
| 2.8.    | Les listes en Python et les compréhensions de listes.....     | 16 |
| 2.8.1.  | Compréhensions de liste .....                                 | 16 |
| 2.8.2.  | Slicing de listes .....                                       | 16 |
| 2.9.    | Les dictionnaires et les compréhensions .....                 | 18 |
| 2.10.   | Les t-uples et les Set.....                                   | 19 |
| 2.11.   | Modules et importation en Python .....                        | 20 |
| 2.11.1. | Créer et appeler un module .....                              | 20 |
| 2.11.2. | Appeler un module existant .....                              | 20 |
| 2.11.3. | Installer et importer un module avec pip .....                | 22 |
| 2.12.   | NumPy – le calcul numérique en Python .....                   | 24 |
| 2.13.   | Matplotlib – La bibliothèque de visualisation en Python ..... | 27 |
| 2.14.   | Sympy - Calcul symbolique en python .....                     | 29 |
| 3.      | Programmation orientée objet en python.....                   | 34 |
| 3.1.    | Introduction générale : Python vs Java en POO .....           | 34 |
| 3.2.    | Création de classes et instances .....                        | 35 |
| 3.3.    | Attributs et accès.....                                       | 36 |
| 3.4.    | Méthodes et surcharges .....                                  | 37 |

|        |  |    |
|--------|--|----|
| 3.5.   | Attributs de classe (équivalent statique en Java) .....        | 38 |
| 3.6.   | Méthodes de classe et méthodes statiques .....                 | 38 |
| 3.7.   | Héritage et polymorphisme .....                                | 39 |
| 3.8.   | Interfaces et abstraction, module ABC .....                    | 40 |
| 3.9.   | Typage et déclaration des attributs en Python .....            | 41 |
| 3.10.  | Autres points utiles .....                                     | 42 |
| 4.     | Fichiers et sérialisation.....                                 | 43 |
| 4.1.   | Lecture et écriture de fichiers en Python .....                | 43 |
| 4.2.   | Sérialisation en Python.....                                   | 43 |
| 4.2.1. | Avec Pickle (sérialisation binaire) .....                      | 43 |
| 4.2.2. | Avec Json (sérialisation textuelle).....                       | 44 |
| 4.3.   | Gestion des exceptions en Python .....                         | 47 |
| 5.     | Interfaces graphiques en python avec QT .....                  | 49 |
| 5.1.   | Présentation de Qt.....  | 49 |
| 5.2.   | Installation et configuration.....                             | 49 |
| 5.3.   | Premier programme avec PyQt6 .....                             | 50 |
| 5.3.1. | Création d'une fenêtre simple.....                             | 50 |
| 5.3.2. | 3 composants de base de QT: .....                              | 50 |
| 5.3.3. | Les gestionnaires de mise en page (Layouts) .....              | 52 |
| 5.3.4. | Réagir aux actions de l'utilisateur (signaux et slots) .....   | 53 |
| 5.4.   | Callbacks et fonctions lambda .....                            | 55 |
| 5.4.1. | La notion de callback .....                                    | 55 |
| 5.4.2. | Fonctions lambda .....   | 56 |
| 5.5.   | Utilisation de QTDesigner .....                                | 56 |
| 5.6.   | Principaux composants graphiques (Widgets).....                | 59 |
| 5.6.1. | Les Fenêtres principales .....                                 | 60 |
| 5.6.2. | Les Widgets de base .....                                      | 62 |
| 5.6.3. | Les Conteneurs et l'organisation.....                          | 64 |
| 5.6.4. | Affichage de listes et de données.....                         | 66 |
| 5.7.   | Gérer l'état actif (grisé ou non) des widgets dans PyQt .....  | 68 |
| 5.8.   | Gestion des layouts pour le redimensionnement de fenêtre ..... | 69 |
| 5.9.   | Les boîtes de dialogue standards dans PyQt .....               | 72 |

|        |  |     |
|--------|--|-----|
| 5.10.  | Insérer un graphisme matplotlib dans une fenêtre QT .....              | 73  |
| 5.11.  | Menus et actions .....   | 75  |
| 5.12.  | Déclarer émettre et intercepter un signal personnalisé.....            | 76  |
| 5.13.  | Gestion des dockedWidgets.....   | 77  |
| 5.14.  | Validation de champs .....   | 79  |
| 5.15.  | Gestion des événements (souris, clavier...) dans PyQt .....            | 80  |
| 5.16.  | Séparation Vue et modèle : Utilisation de QComboBox et QListView ..... | 82  |
| 5.17.  | Appliquer un style qss .....   | 87  |
| 5.18.  | QSettings, données persistantes .....                                  | 89  |
| 6.     | Model-View-Controller (MVC) et autre MV Patterns .....                 | 91  |
| 6.1.   | Généralités sur le MVC .....   | 91  |
|        | .....  | 92  |
| 6.1.   | Un exemple simple d'utilisation du MVC .....                           | 92  |
| 6.2.   | Ajout d'un lien vue vers controller .....                              | 97  |
| 7.     | Arbres et graphes en python .....                                      | 98  |
| 7.1.   | Aperçu théorique sur les graphes .....                                 | 98  |
| 7.1.1. | Définition.....  | 98  |
| 7.1.2. | Voici les différents types de graphe .....                             | 99  |
| 7.1.3. | Représentations classiques .....                                       | 99  |
| 7.1.4. | Exemples d'applications .....  | 100 |
| 7.2.   | NetworkX, une bibliothèque de graphe en python.....                    | 100 |
| 7.2.1. | Présentation générale .....  | 100 |
| 7.2.2. | Les classes principales .....  | 100 |
| 7.2.3. | Structure interne.....   | 101 |
| 7.2.4. | De nombreuses fonctions déjà implémentées! .....                       | 101 |
| 7.2.5. | L'affichage des graphes avec NetworkX.....                             | 102 |
| 8.     | Multithread en python .....  | 105 |
| 8.1.1. | Notion de thread .....   | 105 |
| 8.1.2. | Lancer un thread en python .....                                       | 105 |
| 8.1.3. | Les dangers du multithreading.....                                     | 106 |
| 8.1.4. | Utilisation d'un QThread (PyQt).....                                   | 106 |
| 8.1.1. | Utilisation d'un QProgressBar .....                                    | 108 |

|        |   |     |
|--------|---|-----|
| 9.     | Simulation en python avec PyMunk : .....          | 110 |
| 9.1.   | Introduction.....                                 | 110 |
| 9.2.   | Les bases de PyMunk.....                          | 110 |
| 9.2.1. | L'espace physique (Space) .....                   | 110 |
| 9.2.2. | Corps et formes .....                             | 110 |
| 9.2.3. | Étape de simulation .....                         | 111 |
| 9.3.   | Utilisation de QT pour le rendu .....             | 112 |
| 9.3.1. | Le système de coordonnées Qt.....                 | 112 |
| 9.3.2. | QPainter : outil de dessin.....                   | 112 |
| 9.3.3. | QTimer : la clé pour une animation réussies ..... | 113 |
| 9.4.   | Intégration de pyMunk.....                        | 115 |
| 9.4.1. | Ajout d'une impulsion initiale .....              | 116 |
| 9.4.2. | Détection de collisions.....                      | 117 |
| 9.4.3. | Résumé des principaux composants de PyMunk.....   | 117 |

# 1. Généralités sur le python

## 1.1. Python, un bref historique

Python a été créé à la fin des années 1980 par Guido van Rossum, un informaticien néerlandais, et sa première version publique (0.9.0) a été publiée en 1991. Le langage visait à être à la fois simple à lire et puissant, avec une syntaxe claire et cohérente. Python 1.0 est sorti en 1994, introduisant des concepts fondamentaux comme les fonctions, les exceptions, et les modules.

En 2000, Python 2.0 marque une étape importante en introduisant des fonctionnalités comme la collecte automatique de mémoire (garbage collection) basée sur le comptage de références. Python 2 a connu un grand succès, mais avec le temps, certaines limitations de conception sont devenues apparentes.

Pour répondre à ces limites, Python 3.0 a été lancé en 2008. Cette version n'était pas rétro compatible avec Python 2, ce qui a nécessité une longue période de transition dans la communauté. Python 3 a introduit de nombreuses améliorations, notamment une meilleure gestion des chaînes de caractères (Unicode par défaut), une syntaxe plus, et des bibliothèques modernisées. La fin officielle du support de Python 2 a eu lieu en janvier 2020.

Aujourd'hui, Python continue d'évoluer activement. Les versions les plus récentes (actuellement Python 3.13 et 3.12) apportent des gains notables en performance, ainsi que des nouveautés dans la gestion des types, l'optimisation du code et l'ergonomie du langage. Python reste l'un des langages les plus populaires au monde, apprécié pour sa simplicité, sa communauté active et son immense écosystème de bibliothèques en particulier dans le monde scientifique.

## 1.2. Python un langage interprété

Contrairement à Java, qui est un langage compilé en bytecode et exécuté par la machine virtuelle Java (JVM), Python est un langage interprété. Cela signifie que le code Python est lu et exécuté ligne par ligne par un interpréteur, sans nécessiter une étape explicite de compilation. Pour un programmeur, cela induit une plus grande flexibilité et une boucle de développement plus rapide : il est possible de modifier du code et de le tester immédiatement, sans devoir le compiler. En contrepartie, cela peut parfois entraîner des performances légèrement inférieures à celles des programmes Java, notamment dans les applications très gourmandes en ressources. Cependant, pour beaucoup de cas d'usage,

l’interprétation permet un gain de productivité appréciable, surtout pour des scripts, des outils ou des prototypes.

### 1.3. Python un langage orienté vers les Sciences

Python est largement utilisé dans les contextes scientifiques en raison de sa richesse en bibliothèques spécialisées et de sa simplicité d’utilisation. Contrairement à Java, qui demande souvent une structure rigide pour les interfaces graphiques et le traitement scientifique, Python permet de créer rapidement des interfaces utilisateurs avec des bibliothèques comme **PyQt** ou **PySide**, qui s’appuient sur le framework Qt et offrent des outils puissants pour développer des applications interactives.

Côté calcul scientifique, Python brille grâce à des bibliothèques comme **NumPy** (pour le calcul numérique), **SciPy** (pour les algorithmes scientifiques), et **SymPy**, qui permet le **calcul symbolique** (manipulation de formules mathématiques de manière exacte, comme dans un logiciel de calcul formel). Pour la **visualisation de données**, **Matplotlib** est une référence incontournable : elle permet de produire des graphiques de qualité publication avec une grande flexibilité. Grâce à cette combinaison d’outils, Python est particulièrement adapté à l’exploration de données, à la modélisation, à l’expérimentation et à la présentation de résultats, ce qui en fait un choix naturel dans un contexte scientifique.

## 2. Base de programmation en python

### 2.1. Un exemple à la ligne de commande

Une des forces de Python est que vous pouvez tester du code immédiatement dans un terminal ou une console Python. Pour lancer l'interpréteur, tapez simplement :

```
C:\Users\julien.brunet>python
Python 3.10.5 (tags/v3.10.5:f377153, Jun  6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

Vous verrez apparaître un prompt interactif, souvent signalé par >>>, dans lequel vous pouvez entrer des instructions Python. Par exemple :

```
>>> print("Bonjour, Python !")
Bonjour, Python !
```

Pas besoin de public static void main, ni de classe. Une simple ligne suffit pour afficher du texte. Vous pouvez aussi effectuer directement des calculs :

```
>>> 2+3*4
14
```

Ou définir une variable et la réutiliser :

```
>>> print("Bonjour " + nom)
Bonjour Julien
```

**Dans le cadre du cours on utilisera la version Python 3.13.3 qui est actuellement installée dans nos laboratoires**

### 2.2. Pycharm, notre environnement de développement

Comme en java, on a rapidement besoin d'un environnement de développement. Nous utiliserons Pycharm. Le gros avantage est que des utilisateurs avertis (comme vous) d'IntelliJ ne devraient pas être déstabilisés!

**Dans le cadre du cours on utilisera la version PyCharm 2025.1.2 qui est actuellement installée dans nos laboratoires**

## 2.3. Les fichiers .py

En Python, au lieu de tout écrire dans une classe comme en Java, on peut simplement écrire les instructions directement dans un fichier texte avec l'extension .py. On peut signaler que la restriction - une classe publique un fichier avec le même nom - connue en java, est absente ici.

Par convention, les noms de fichiers Python sont **en minuscules**, avec des **tirets bas (\_)** pour séparer les mots, par exemple : mon\_premier\_programme.py

### Exemple 1.

```
# mon_premier_programme.py

nom = "Alice"
age = 30

if age >= 18:
    print(nom, " est majeur(e) ")
else:
    print(nom, " est mineur(e) ")
```

### À noter pour les programmeurs Java :

- **Pas besoin de point-virgule (;)** à la fin des lignes. En Python, les retours à la ligne suffisent pour séparer les instructions.
- **L'indentation est obligatoire** et fait partie de la syntaxe. Contrairement à Java, où on utilise des accolades {} pour délimiter les blocs de code, Python utilise des **indentations cohérentes** (généralement 4 espaces) pour indiquer les blocs (par exemple, les instructions d'un if, d'une boucle, etc.). Attention c'est une source d'erreur chez les débutants!
- **Les commentaires** sont précédés par le #

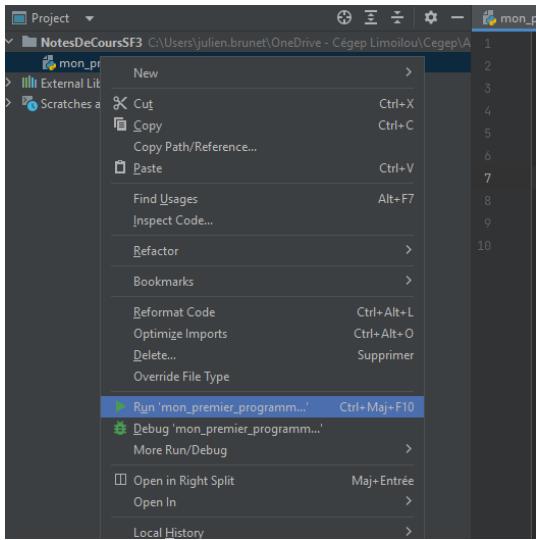
### Pour exécuter le programme :

Sauvegardez le fichier puis exécutez-le depuis un terminal :

```
C:\Users\julien.brunet\OneDrive - Cégep Limoilou\Cégep\A25\203\JBR\NotesDeCoursSF3>python mon_premier_programme.py
Alice est majeur(e)
```

Ou dans Pycharm :

Clic droit sur le fichier mon\_premier\_programme.py et Run (la flèche verte habituelle)



## 2.4. Les types et les variables en python

En Python, **il n'est pas nécessaire de déclarer le type d'une variable** : le langage utilise un système de **typage dynamique**, ce qui signifie que le type d'une variable est déterminé automatiquement au moment de l'exécution, en fonction de la valeur assignée. Cela diffère de Java, où chaque variable doit être déclarée avec un type explicite. En Python, une variable peut même changer de type en cours de programme, ce qui permet une grande flexibilité, mais demande aussi de rester vigilant.

### Exemple 2. Déclaration de variables

| En Java :   | En Python :                       |
|---|-----------------------------------|
| <pre>public static void main(String[] args) {     int age = 25;     String nom = "Alice";</pre> | <pre>age = 25 nom = "Alice"</pre> |

Aucun type à indiquer : Python comprend automatiquement que age est un entier (int) et que nom est une chaîne de caractères (str). Pour les programmeurs Java, cela peut paraître surprenant au, mais cela rend le code plus concis.

#### Remarque :

Il est également possible d'utiliser des **annotations de type** pour clarifier le code, bien que cela ne soit pas obligatoire :

**Exemple 3.**

```
age:int = 25
nom:str = "Alice"
```

Ces annotations sont ignorées à l'exécution mais peuvent être utilisées par des outils de vérification statique comme **mypy** ou des éditeurs de code pour l'auto-complétion et la détection d'erreurs. On en reparlera!

| Concept        | Java                         | Python   | Notes importantes   |
|----------------|------------------------------|--|---|
| Entier         | int, long                    | <b>int</b>                                     | En Python, int n'a pas de limite de taille  |
| Nombre décimal | float, double                | <b>float</b>                                   | Équivaut grosso modo à un double Java   |
| Booléen        | boolean                      | <b>bool</b>                                    | Valeurs : True / False (avec majuscule !)   |
| Caractère      | char                         | Pas de type char                               | Un caractère est une chaîne de longueur 1 (str)   |
| Chaîne         | String                       | <b>str</b>                                     | Très simple d'utilisation, concaténation avec +, délimitée par simple ou double guillemet |
| Liste/Array    | int[],<br>ArrayList<Integer> | <b>List</b> notées :<br>[ e1 , e2 , e3 ]       | Une liste Python est dynamique et hétérogène  |
| Dictionnaire   | HashMap<Key,<br>Value>       | Dict<br>Notés<br>{cle :valeur ,... :..., } , } | Clé/valeur, très utilisé en Python  |
| Ensemble       | Set<T>                       | Set<br>Notés :<br>{ el1, el2, el3}             | Même rôle, non ordonné et sans doublons   |

| Concept | Java | Python | Notes importantes              |
|---------|------|--------|--------------------------------|
| Null    | null | None   | Représente l'absence de valeur |

**Remarques :**

- En Python, **tout est objet**, même les types de base comme **int** ou **float**. Cela permet de les manipuler très librement.
- Les **chaînes de caractères** (str) sont très puissantes et supportent de nombreuses opérations intégrées (remplacement, recherche, formatage...).
- Les **listes** (list) peuvent contenir des éléments de types différents, ce qui n'est pas autorisé dans les tableaux Java.
- On peut ajouter les **t-uples** en python qui sont comme les listes mais immutables (donc non modifiables). On les note avec de parenthèses ( , , , ). Ils sont plus légers et plus performants que les listes, utiles lorsqu'on a un très grand nombre de données

## 2.5. Les fonctions en Python

En Python, les fonctions sont définies avec le mot-clé `def`, suivi du nom de la fonction et des paramètres entre parenthèses. Contrairement à Java, il n'est pas nécessaire de déclarer le type des paramètres ni le type de retour (même si on peut ajouter des annotations pour cela). La syntaxe est plus concise, et l'**indentation obligatoire** délimite le corps de la fonction, remplaçant les accolades {} utilisées en Java.

**Exemple 4.**

| En Java  | En python   |
|--|---|
| <pre>public class Exemple {     public static int addition(int a, int b) { 1 usage         return a + b;     }      public static void main(String[] args) {         int resultat = addition( a: 5, b: 3);         System.out.println("Résultat : " + resultat);     } }</pre> | <pre>def addition(a,b) :     return a+b; resultat = addition(5,3) print("Résultat :" +resultat)</pre> |

**Remarques :**

- En Python, pas besoin de déclarer les types des paramètres, ni le type de retour.

- L'indentation (4 espaces par convention) est obligatoire pour définir le corps de la fonction.
- Pas de point-virgule ni de structure de classe pour exécuter un petit programme.
- La fonction peut être appelée directement dans le corps du script, mais doit être déclarée avant l'appel
- Python offre la possibilité de retour multiples séparés par des virgules. En fait il s'agit de retourner un t-uple mais de manière implicite. **C'est une fonctionnalité à utiliser avec modération, car elle est difficilement documentable.**
- Python supporte de donner une valeur par défaut à un paramètre, **ce qui permet d'éviter de nombreuses surcharges de méthodes**
- On a un équivalent de la javadoc en python appelé docstring

**Exemple 5.**

```
def addition(a,b,c=0) :
    """
    Additionne deux nombres ou trois nombre.

    Args:
        a (float): Le premier nombre.
        b (float): Le deuxième nombre.
        c (float, optional 0 par défaut): Le troisième nombre.
    Returns:
        float: Le résultat de la somme a + b + c.
    """
    return a+b+c;
print(addition(2,3,4))
print(addition(2,3))
```

## 2.6. Les structures conditionnelles en Python

En Python, les instructions conditionnelles (`if`, `elif`, `else`) permettent de diriger le flux d'exécution en fonction de conditions logiques, comme en Java. Là aussi, la syntaxe est plus concise et repose sur l'indentation au lieu des accolades `{}`. De plus, les parenthèses autour des conditions sont facultatives (mais autorisées).

**Exemple 6.**

|         |           |
|---------|-----------|
| En Java | En python |
|---------|-----------|

|  |  |
|--|--|
| <pre> int age = 20;  if (age &gt;= 18) {     System.out.println("Majeur"); } else if (age &gt; 12) {     System.out.println("Adolescent"); } else {     System.out.println("Enfant"); } </pre> | <pre> age = 20 if age &gt;= 18:     print("Majeur") elif age &gt; 12:     print("Adolescent") else:     print("Enfant") </pre> |
|--|--|

#### Définitions clés :

- **elif** remplace else if .
- **Pas de parenthèses ni d'accolades** : l'indentation (souvent 4 espaces) indique les blocs.
- **Les conditions sont des expressions** : on utilise directement and, or, not au lieu de &&, ||, !.

#### Exemple 7.

|   |
|---|
| <pre> if age &gt;= 18 and age &lt; 65:     print("Adulte actif") </pre> |
|---|

## 2.7. Les boucles en Python

Python propose deux types principaux de boucles : `while` et `for`. Leur fonctionnement est similaire à celui de Java, mais leur syntaxe est plus concise. En particulier, la boucle `for` de Python est conçue pour **parcourir directement des séquences** (listes, chaînes, etc.), plutôt que d'utiliser des indices.

#### Exemple 8.

| En Java  | En python   |
|--|---|
| <pre> String[] noms = {"Alice", "Bob", "Charlie"}; for (int i = 0; i &lt; noms.length; i++) { </pre> | <pre> noms = ["Alice", "Bob", "Charlie"] for nom in noms:     print(nom) </pre> |

```
System.out.println(noms[i]);
}
```

En Python, on **parcourt directement les éléments** de la liste, pas besoin d'un compteur i.

### Exemple 9.

| En Java  | En python   |
|--|---|
| <pre>int compteur = 0; while (compteur &lt; 3) {     System.out.println("Compteur : " + compteur);     compteur++; }</pre> | <pre>compteur = 0  while compteur &lt; 3:     print("Compteur :", compteur)     compteur += 1</pre> |

Même logique qu'en Java, mais toujours sans parenthèses ni accolades, et avec indentation obligatoire.

### Bonus : la fonction range()

Python dispose de la fonction range() pour générer des suites de nombres, très pratique dans les boucles for.

### Exemple 10.

```
for i in range(3):
    print("i =", i)
```

Cela remplace le classique for (int i = 0; i < 3; i++) de Java.

### Boucles imbriquées et instructions break / continue

Comme en Java, on peut imbriquer des boucles en Python, par exemple pour parcourir une matrice (liste de listes) :

### Exemple 11.

| En Java   | En python   |
|---|---|
| <pre>int[][] matrice = {     {1, 2},     {3, 4} };  for (int i = 0; i &lt; matrice.length; i++) {     for (int j = 0; j &lt; matrice[i].length; j++) {         System.out.println(matrice[i][j]);     } }</pre> | <pre>matrice = [     [1, 2],     [3, 4] ]  for ligne in matrice:     for valeur in ligne:         print(valeur)</pre> |

### Instruction break

Comme en java elle permet de sortir **immédiatement** d'une boucle.

### Exemple 12.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

**Résultat:** seuls les nombres de 0 à 4 s'afficheront

### Instruction continue

Comme en java elle permet d'**ignorer le reste de l'itération** courante et de passer à la suivante.

### Exemple 13.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

**Résultat:** seul le nombre de 2 ne s'affichera pas

**Remarque importante :** En Python, on peut aussi ajouter un else **après une boucle** (for ou while). Il ne s'exécute que si la boucle **n'a pas été interrompue par un break**. C'est une fonctionnalité unique que Java ne propose pas... Pertinent ou pas, je vous laisse juge...

## 2.8. Les listes en Python et les compréhensions de listes

Les **listes** en Python sont des collections **dynamiques** : elles peuvent contenir n'importe quel type d'élément (mélangé si besoin), et leur taille peut changer à tout moment.

### Exemple 14.

```
noms = ["Alice", "Bob", 123]
noms.append("Diana")
print(noms[1])
print(noms[2])
```

### 2.8.1. Compréhensions de liste

Une compréhension de liste (list comprehension) est une manière compacte de générer une nouvelle liste à partir d'une séquence existante, en une seule ligne lisible.

#### Exemple 15. Doubler les nombres d'une liste

```
resultats = [i * 2 for i in range(10)]
```

#### Exemple 16. Filtrer des éléments et ne garder que les nombres pairs de 0 à 9 :

```
pairs = [i for i in range(10) if i % 2 == 0]
```

En Java, cela nécessiterait une boucle for + un if + un add().

Syntaxe générale : [expression for élément in séquence if condition]

C'est un outil très puissant, mais à utiliser avec modération : si la logique devient trop complexe, mieux vaut revenir à une boucle classique pour garder la lisibilité.

### 2.8.2. Slicing de listes

On peut également **extraire des sous listes** d'une liste données en utilisant le **slicing**

Le slicing de liste en Python, avec la syntaxe ma\_liste[début:fin:pas], permet d'extraire une sous-liste d'une liste existante. Vous spécifiez un indice de départ, un indice de fin (exclus) et un pas optionnel pour définir la plage d'éléments à inclure dans la nouvelle liste. Cela offre une manière concise de manipuler les listes, de créer des copies (y compris des copies superficielles) ou d'inverser des listes sans recourir à des boucles complexes.

Syntaxe générale: ma\_liste [début:fin:pas]

- début:

L'indice de l'élément à partir duquel le slice commence. S'il est omis, le slice commence au début de la liste (indice 0).

- **fin:**

L'indice de l'élément jusqu'où le slice va, mais cet élément n'est pas inclus dans le résultat. S'il est omis, le slice va jusqu'à la fin de la liste.

- **pas:**

L'incrément entre les éléments du slice. S'il est omis, le pas est de 1.

**Exemple 17.** Prenons une liste lettres = ['A', 'B', 'C', 'D', 'E', 'F'] :

- **Extraire une partie de la liste**

- lettres[1:4] donnera ['B', 'C', 'D']. Elle commence à l'indice 1 et s'arrête avant l'indice 4.

- **Obtenir les premiers éléments**

- lettres[:3] donnera ['A', 'B', 'C']. C'est équivalent à lettres[0:3].

- **Obtenir les derniers éléments**

- lettres[3:] donnera ['D', 'E', 'F']. C'est équivalent à lettres[3:len(lettres)].

- **Utiliser un pas**

- lettres[0:6:2] donnera ['A', 'C', 'E']. Elle prend chaque deuxième élément, en commençant par l'indice 0.

- **Inverser une liste**

- lettres[::-1] donnera ['F', 'E', 'D', 'C', 'B', 'A']. Un pas de -1 inverse la liste.

- **Créer une copie de la liste**

- lettres[:] créera une copie superficielle complète de la liste lettres, un nouvel objet liste avec les mêmes éléments. Attention! Ce n'est pas vrai avec numpy dont on perlera plus tard!

Avantages du slicing

- **Concision** : Rends le code plus court et plus lisible qu'avec des boucles.
- **Flexibilité** : Permet de manipuler des séquences (listes, chaînes de caractères, tuples) de manière très flexible.
- **Copies** : Permet de créer des copies de listes, ce qui est crucial pour éviter que les modifications d'une liste affectent l'autre

## 2.9. Les dictionnaires et les compréhensions

Un **dictionnaire** en Python est une collection **clé-valeur**, équivalent à une Map en Java. Les clés sont uniques, et les valeurs peuvent être de n'importe quel type.

**Exemple 18.**

```
notes = {"Alice": 18, "Bob": 15}
print(notes["Alice"]) # 18
```

Tout comme les listes, Python permet de **générer des dictionnaires en une seule ligne** avec une syntaxe claire, appelée **compréhension de dictionnaire**

**Exemple 19. Associer un carré à chaque nombre**

```
carres = {x: x*x for x in range(5)}
print(carres) # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

**Exemple 20. Dictionnaire contenant les nombres associés aux les carres des nombres s'ils sont pairs**

```
pairs = {x: x*x for x in range(10) if x % 2 == 0}
print(pairs)
```

Ces compréhensions sont très utiles pour transformer ou filtrer des données de manière expressive, là où Java nécessiterait plusieurs lignes de code et une boucle explicite.

En Python, l'opérateur étoile \* devant un dictionnaire "déballe" ses clés pour les transformer en un itérable (tuple, liste, set), tandis que l'opérateur double étoile \*\* sert à déballer les paires clé-valeur d'un ou plusieurs dictionnaires afin de les fusionner dans un nouveau dictionnaire.

**Exemple 21. Utiliser \* et \*\* devant un dictionnaire**

```
mon_dict = {"a": 1, "b": 2, "c": 3}
liste_de_cles = [*mon_dict]
print(liste_de_cles)
# Sortie : ['a', 'b', 'c']

tuple_de_cles = (*mon_dict,) # Notez la virgule pour forcer le tuple
print(tuple_de_cles)
# Sortie : ('a', 'b', 'c')

dict1 = {"nom": "Albert", "age": 30}
dict2 = {"ville": "Berlin", "métier": "Physicien"}
```

```
# Fusion de dictionnaires
dict_fusionne = {**dict1, **dict2}
print(dict_fusionne)
# Sortie : {'nom': 'Albert', 'age': 30, 'ville': 'Berlin', 'métier': 'Physicien'}

# Déballage d'arguments pour une fonction
def afficher_infos(nom, age, ville):
    print(f"Nom: {nom}, Âge: {age}, Ville: {ville}")

infos_personne = {"nom": "Bob", "age": 25, "ville": "Quebec"}
afficher_infos(**infos_personne)
# Sortie : Nom: Bob, Âge: 25, Ville: Quebec
```

## 2.10. Les t-uples et les Set

Un **t-uple** en Python est une collection **notée entre parenthèse** qui se comporte comme une liste, mais qui ne peut pas être modifiée. On peut le comparer au mot-clé final en java.  
Pour déclarer un tuple on utilise la notation

```
a=(1, 2, 3)
```

En constate qu'on ne peut pas assigner un élément de la liste, l'instruction suivante lève une exception :`TypeError: 'tuple' object does not support item assignment`

```
a=(1, 2, 3)
a[0] = 2
```

Un set en Python, est exactement l'implémentation de la notion d'ensemble en mathématique : une collection non ordonnée et sans répétition.

### Exemple 22. Utiliser un Set

```
a = {5, 1, 2, 1, 3, 4, 1}
print(a)
```

Conduit au résultat d'exécution: {1, 2, 3, 4, 5}

## 2.11. Modules et importation en Python

### 2.11.1. Créer et appeler un module

En Python, un **module** est tout simplement un fichier .py contenant du code (fonctions, classes, variables, etc.). Cela permet d'organiser le code en plusieurs fichiers pour le rendre plus lisible et réutilisable — exactement comme les **packages** ou **classes** en Java.

#### Exemple 23. Créer et appeler un module

Supposons qu'on a un fichier `outils.py` avec cette fonction :

```
# outils.py
def saluer(nom):
    print(f"Bonjour, {nom}!")
```

On peut ensuite l'utiliser dans un autre fichier :

```
import outils
outils.saluer("Alice")
```

Ici, `import outils` charge le module, et on accède à son contenu avec le préfixe `outils`.

On peut aussi importer uniquement une partie du module.

```
from outils import saluer
saluer("Bob") # Pas besoin de prefixe
```

On n'a alors pas besoin du préfixe `outils`.

Contrairement à Java, **le nom du fichier est le nom du module**, et il n'y a **pas besoin de déclarer un package**

### 2.11.2. Appeler un module existant

Python fournit une **bibliothèque standard riche** que tu peux importer directement, par exemple `math`.

#### Exemple 24. Appeler un module existant de la bibliothèque standard

```
import math  
print(math.sqrt(16)) # 4.0
```

Voici un aperçu des modules les plus utiles, regroupés par thème :

### **Maths, nombres et statistiques**

- math : fonctions mathématiques de base (sqrt, cos, log, etc.).
- statistics : moyenne, médiane, écart type, etc.
- decimal et fractions : précision arbitraire et calculs exacts.

### **Collections et structures de données**

- collections : types spécialisés comme Counter, deque, defaultdict.
- array : tableaux typés.
- heapq : files de priorité (heap).
- bisect : insertion rapide dans une liste triée.

### **Dates et temps**

- datetime : manipulation de dates et d'heures.
- time : accès bas-niveau à l'horloge système.
- calendar : calendrier, années bissextiles, etc.

### **Fichiers et système**

- os : interaction avec le système d'exploitation (chemins, variables d'environnement).
- shutil : gestion des fichiers (copie, suppression, etc.).
- pathlib : manipulation moderne de chemins de fichiers.
- glob : recherche de fichiers avec des motifs (\*.txt).

### **Lecture/écriture de fichiers texte et données**

- csv : lecture/écriture de fichiers CSV.
- json : manipulation de données JSON.
- configparser : lecture de fichiers INI (configuration).

### **Réseau et internet**

- http.client, urllib : requêtes HTTP.

- socket : sockets bas-niveau.
- email : création et parsing d'e-mails.
- ftplib, smtplib : FTP, envoi de mails.

### Sécurité et chiffrement

- hashlib : fonctions de hachage (md5, sha256, etc.).
- hmac, secrets : authentification et sécurité.

### Tests et débogage

- unittest : framework de test unitaire (comme JUnit).
- doctest : tests dans la doc.
- logging : journalisation.
- traceback, pdb : gestion des erreurs et débogage.

### Concurrence et parallélisme

- threading, multiprocessing : exécution concurrente ou parallèle.
- asyncio : programmation asynchrone (comme async/await en JS/Java).

On fera la découverte de certaines d'entre elle petit à petit à travers le cours

### 2.11.3. Installer et importer un module avec pip

pip est le gestionnaire de paquets officiel de Python. Il permet d'installer facilement des **modules tiers** (librairies) disponibles sur le dépôt [PyPI](#).

On utilise pour cela le terminal et la commande `pip install nom_du_module`

On verra dans le paragraphe suivant comment installer `numpy` à titre d'exemple.



## 2.12. NumPy – le calcul numérique en Python

**NumPy** (*Numerical Python*) est la **bibliothèque fondatrice** de tout l'écosystème scientifique Python. Elle apporte une structure de données puissante : le **ndarray** (n-dimensional array), un tableau homogène et performant, similaire aux **tableaux multidimensionnels en Java**, mais avec des opérations **vectorisées** rapides et concises.

**Ses principaux atouts sont:**

- Manipulation efficace de **vecteurs et matrices**.
- Syntaxe claire pour faire du **calcul matriciel, statistique, algèbre linéaire, génération aléatoire**, etc.
- Supporte des **opérations élémentaires vectorisées**, bien plus rapides que des boucles for.

**Exemple 25. Création de tableaux :**

```
import numpy as np

a = np.array([1, 2, 3])           # Tableau 1D
b = np.array([[1, 2], [3, 4]])    # Matrice 2D

zeros = np.zeros((2, 3))          # Matrice 2x3 remplie de zéros
ones = np.ones(5)                 # Vecteur de 5 éléments à 1
range_ = np.arange(0, 10, 2)      # [0 2 4 6 8]
```

**Exemple 26. Opérations vectorisées :**

```
x = np.array([1, 2, 3])
y = np.array([10, 20, 30])

print(x + y)        # [11 22 33]
print(x * 2)        # [2 4 6]
print(x ** 2)       # [1 4 9]
```

Il n'y a pas besoin de boucle, tout est **appliqué en parallèle**.

**Exemple 27. Statistiques:**

Voici quelques-unes des multiples fonctionnalités du module statistique

```
valeurs = np.array([4, 7, 1, 8])

print(np.mean(valeurs))    # Moyenne : 5.0
print(np.median(valeurs))  # Médiane : 5.5
print(np.std(valeurs))     # Écart-type : ~2.59
```

**Exemple 28. Algèbre linéaire :**

Pareil pour l'algèbre linéaire en utilisant le sous-module `linalg` pour toutes les fonctions de **mathématiques matricielles avancées** : calcul de déterminant, inversion, résolution de systèmes linéaires, , etc.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

produit = np.dot(A, B)
inverse = np.linalg.inv(A)
det = np.linalg.det(A)
```

---

**Exemple 29. Génération aléatoire :**

```
np.random.seed(0)
tirages = np.random.randint(1, 7, size=10) # 10 dés lancés
```

Remplace avantageusement `java.util.Random`.

**Comparaison avec Java**

| Fonction               | Java   | NumPy (Python)                |
|------------------------|--|-------------------------------|
| Tableau dynamique      | ArrayList, ou int[] manuel   | <code>np.array(...)</code>    |
| Produit scalaire       | Boucle + accumulation  | <code>np.dot(x, y)</code>     |
| Moyenne, médiane, etc. | Apache Commons Math, ou manuel <code>np.mean(arr), np.median(arr)</code> |                               |
| Inversion de matrice   | Lib externe + objets Matrix  | <code>np.linalg.inv(A)</code> |

**Gestion des copies :**

- **Pour les listes Python standards**

- `[:] et .copy() → copie superficielle (mêmes références mais nouveau tableau).`

- `copy.deepcopy()` → vraie copie profonde.
- **En NumPy ndarray**
  - `[:] → fabrique une vue (partage mémoire).`
  - `.copy() → vraie copie indépendante des données.`

## 2.13. Matplotlib – La bibliothèque de visualisation en Python

**Matplotlib** est la bibliothèque la plus utilisée en Python pour créer des **graphiques 2D** (courbes, barres, histogrammes, nuages de points, etc.). Elle fonctionne très bien avec **NumPy**. On va en particulier s'intéresser au sous-module pyplot

Il suffit de l'installer avec `pip install matplotlib`

### Exemple 30. Exemple simple – tracer une fonction

Ce code trace la fonction  $\sin(x)$  avec une grille, un titre, et une légende — en quelques lignes seulement.

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

# construit un tableau numpy de 100 points espacés régulièrement entre
# 0 et 10
x = np.linspace(0, 10, 100)
# applique le sinus à tous les points de x et les stocke dans le
# tableau numpy sy
y = np.sin(x)

# dessine le nuage de point x,y reliés par des segments de droite
plt.plot(x, y)

# Affiche le résultat dans le backend graphique
plt.show()
```

La fonction `np.linspace(début, fin, nombre_point)` est vraiment importante, il faut la mémoriser!

On peut personnaliser le graphique avec les différentes instructions suivantes :

```
plt.plot(x, y, label="sin(x)", color="red", linestyle="--")
plt.title("Courbe de sin(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()
```

On peut représenter des histogrammes

```
def plot_histo():
    # Histogramme
    data = np.random.normal(0, 1, 1000) # 1000 valeurs aléatoires
```

```
suivant N(0,1)

plt.hist(data, bins=30, color="green", edgecolor="black")
plt.title("Histogramme d'une loi normale")
plt.xlabel("Valeurs")
plt.ylabel("Fréquence")
plt.show()
```

ou des diagrammes en barres

```
def plot_barres():
    # Diagramme en barres
    categories = ["A", "B", "C", "D"]
    valeurs = [3, 7, 4, 6]

    plt.bar(categories, valeurs, color="orange")
    plt.title("Exemple de diagramme en barres")
    plt.show()
```

et bien d'autres possibilités encore comme des diagrammes en camembert (pie) , nuages de points (scatter), etc.

Si on souhaite présenter plusieurs courbes sur la même image... il suffit d'en tracer plusieurs!

```
x = np.linspace(0, 2*np.pi, 100)
plt.plot(x, np.sin(x), label="sin(x)")
plt.plot(x, np.cos(x), label="cos(x)")
plt.title("Trigonométrie")
plt.legend()
plt.show()
```

ou plusieurs graphiques dans une même page :

```
x = np.linspace(0, 10, 100)
y1, y2 = np.sin(x), np.cos(x)

plt.subplot(2, 1, 1)    # 2 lignes, 1 colonne, 1er graphe
plt.plot(x, y1)
plt.title("sin(x)")

plt.subplot(2, 1, 2)    # 2 lignes, 1 colonne, 2e graphe
plt.plot(x, y2)
plt.title("cos(x)")

plt.tight_layout()      # Ajuste les marges
plt.show()
```

Enfin, on peut sauvegarder l'image dans un fichier!

```
plt.plot(x, y1, label="sin(x)")  
plt.legend()  
plt.savefig("graphique.png")
```

Voici un petit résumé des principales commandes, clairement non exhaustif!

| Fonction                   | Rôle   |
|----------------------------|--|
| plt.plot(x, y)             | Courbe 2D (points reliés par des segments de droite) |
| plt.scatter(x, y)          | NUAGE DE POINTS (POINTS NON RELIÉS)                  |
| plt.bar(x, h)              | DIAGRAMME EN BARRES                                  |
| plt.hist(data)             | HISTOGRAMME  |
| plt.pie(parts)             | CAMEMBERT  |
| plt.title()                | TITRE DU GRAPHIQUE                                   |
| plt.xlabel(), plt.ylabel() | NOM DES AXES   |
| plt.legend()               | LÉGENDE  |
| plt.subplot()              | Sous-graphes   |
| plt.savefig()              | SAUVEGARDER LE GRAPHE                                |
| plt.show()                 | AFFICHER LE GRAPHE                                   |

## 2.14. Sympy - Calcul symbolique en python

La bibliothèque **Sympy** est un module Python spécialisé dans le calcul symbolique, permettant de manipuler des expressions mathématiques sous forme exacte, contrairement au calcul numérique qui produit des approximations. Elle est particulièrement utile dans les domaines de l'enseignement, de la recherche, ou pour le prototypage rapide de formules mathématiques.

Avant toute opération, il faut définir des symboles (variables symboliques) à l'aide de la fonction `symbols`. Par exemple, `x = symbols('x')` permet de travailler avec la variable `x` comme une inconnue algébrique.

### Exemple 31. Résoudre une équation

Pour la résolution d'équations, la fonction `solve()` est très puissante :  
`solve(x**2 - 4, x)` permet de résoudre l'équation  $x^2 - 4 = 0$ , et renvoie la liste  $[-2, 2]$ , correspondant aux racines exactes de l'équation.

```
import sympy as sp
x = sp.symbols('x')
solutions = sp.solve(x**2 - 4, x)

print(solutions)
```

SymPy peut également résoudre des équations **polynomiales, rationnelles, logarithmiques ou trigonométriques**, ou même des **systèmes d'équations**.

Pour le **calcul différentiel**, la fonction `diff()` est utilisée.

### Exemple 32. Calculer une dérivée

Pour la résolution d'équations, on utilise la fonction `diff()` : `diff(sin(x**2), x)` qui calcule la dérivée de  $\sin(x^2)$

```
import sympy as sp
x = sp.symbols('x')
derivee = sp.diff(sp.sin(x**2), x)

print(derivee)
```

SymPy prend également en charge les dérivées d'ordre supérieur (`diff(expr, x, 2)` pour la dérivée seconde).

En ce qui concerne le **calcul intégral**, `integrate()` permet de calculer des **intégrales définies ou indéfinies**.

Par exemple, `integrate(exp(-x**2), x)` donne une primitive de  $e^{-x^2}$ , Pour une `integrate(sin(x), (x, 0, pi))` et SymPy retourne 2.

SymPy supporte aussi la **simplification d'expressions** (`simplify, expand, factor`), l'**algèbre linéaire symbolique**, les **équations différentielles**, les **séries de Taylor**, les **limites**, et même l'**affichage en LaTeX**.

### Exemple 33. Exemples variés d'utilisation de Sympy

On utilise dans l'exemple suivant le « pretty print » de `sympy` qui permet de simuler un affichage mathématique correct à la console. Celui-ci a ses limites mais permet tout de même souvent une lecture plus aisée.

```

from sympy import symbols, Eq, solve, diff, integrate, sin, cos,
exp, simplify, pprint, init_printing, pi, erf

# Active l'affichage "joli" dans la console ou dans un notebook
init_printing()

# Définition des variables symboliques
x, y = symbols('x y')

# 1. Résolution d'équation algébrique :  $x^2 - 4 = 0$ 
eq1 = Eq(x**2 - 4, 0) # Définit l'équation  $x^2 - 4 = 0$ 
sol_eq1 = solve(eq1, x)
print("Solutions de l'équation  $x^2 - 4 = 0$  :")
pprint(sol_eq1) # Affichage plus lisible

# 2. Calcul différentiel : dérivée de  $\sin(x^2)$ 
f = sin(x**2)
derivative_f = diff(f, x)
print("\nDérivée de  $\sin(x^2)$  par rapport à  $x$  :")
pprint(derivative_f)

# 3. Dérivée seconde de  $\exp(-x^2)$ 
g = exp(-x**2)
second_derivative = diff(g, x, 2)
print("\nDérivée seconde de  $\exp(-x^2)$  :")
pprint(second_derivative)

# 4. Intégrale indéfinie :  $\int e^{-x^2} dx$ 
integral1 = integrate(exp(-x**2), x)
print("\nIntégrale indéfinie de  $\exp(-x^2)$  :")
pprint(integral1)

# 5. Intégrale définie :  $\int_0^\pi \sin(x) dx$ 
integral2 = integrate(sin(x), (x, 0, pi))
print("\nIntégrale définie de  $\sin(x)$  entre 0 et  $\pi$  :")
pprint(integral2)

# 6. Simplification d'une expression
expr = (x**2 - 1)/(x - 1)
simplified_expr = simplify(expr)
print("\nSimplification de  $(x^2 - 1)/(x - 1)$  :")
pprint(simplified_expr)

# 7. Résolution d'un système d'équations
eq_system = [Eq(x + y, 3), Eq(x - y, 1)]
sol_system = solve(eq_system, (x, y))
print("\nSolution du système :  $x + y = 3$  et  $x - y = 1$ ")
pprint(sol_system)

```

Dont voici le résultat d'exécution :



```
Solutions de l'équation  $x^2 - 4 = 0$  :  
[-2, 2]
```

```
Dérivée de  $\sin(x^2)$  par rapport à x :  

$$\frac{d}{dx} \sin(x^2) = 2x \cos(x^2)$$

```

```
Dérivée seconde de  $\exp(-x^2)$  :  

$$\frac{d^2}{dx^2} \exp(-x^2) = 2 \cdot (2x^2 - 1) \cdot e^{-x^2}$$

```

```
Intégrale indéfinie de  $\exp(-x^2)$  :  

$$\int \exp(-x^2) dx = \frac{\sqrt{\pi}}{2} \operatorname{erf}(x)$$

```

```
Intégrale définie de  $\sin(x)$  entre 0 et  $\pi$  :  

$$\int_0^\pi \sin(x) dx = 2$$

```

```
Simplification de  $(x^2 - 1)/(x - 1)$  :  

$$\frac{x^2 - 1}{x - 1} = x + 1$$

```

```
Solution du système :  $x + y = 3$  et  $x - y = 1$   
{x: 2, y: 1}
```

**Remarque :** comme on le sait  $e^{-x^2}$  n'a pas de primitive explicite, d'où la présence de `erf()` qui représente [la fonction d'erreur...](#)

#### Exemple 34. Évaluer une fonction avec SymPy

Pour évaluer une fonction SymPy en Python, utilisez la méthode `evalf()` pour obtenir une approximation numérique de l'expression exacte, ou la fonction `subs()` pour remplacer des variables par des valeurs numériques afin de calculer la valeur de la fonction à ces points spécifiques.

```
x = sp.symbols('x')
f = sp.sin(x**2)
exacte = f.subs(x, 2)
approx= exacte.evalf()
```

```
print("sin(2**2) = ", exacte, "=", approx)
```

Enfin terminons par mentionner la possibilité de tracer un graphe avec `sympy` à l'aide de la méthode `plot` :

**Exemple 35. Dessiner un graphe avec Sympy**

```
x = sp.symbols('x')
f = sp.sin(x**2)

sp.plot(f, (x, 0, 7))
```

Notons que cette méthode offre beaucoup moins d'options que `pyplot`. Pour revenir à l'utilisation de `matplotlib`, il faut connaître l'usage de la méthode `lambdify()` qui permet de récupérer une fonction numérique utilisable par `numpy`!

**Exemple 36. Revenir à numpy et matplotlib**

```
x = sp.symbols('x')
f = sp.sin(x**2)
f_num = sp.lambdify(x, f)

x=np.linspace(-np.pi,np.pi,100)

plt.plot(x,f_num(x))
plt.show()
```

Ainsi, SymPy s'impose comme un outil éducatif et scientifique très complet pour toute personne souhaitant manipuler rigoureusement des expressions mathématiques en Python.

## 3. Programmation orientée objet en python

Ce chapitre a pour but de présenter l'objet à des personnes ayant une bonne maîtrise de la POO en Java. Il ne réexplique pas les concepts objets, mais se concentre sur les **différences de syntaxe, de philosophie et de mécanismes** entre Java et Python.

Python propose une approche beaucoup plus souple de la POO que Java. Cette liberté permet d'écrire du code plus concis, mais demande plus de discipline du développeur. Il n'y a pas de compilation stricte ni de contrôle fort des types, mais les conventions (et les outils comme mypy ou les annotations) permettent de garder un code propre et maintenable.

Pour des programmeurs Java, la principale difficulté est de "lâcher prise" sur les contraintes habituelles du typage et de la structure formelle, tout en gardant les bonnes pratiques de lisibilité, de testabilité et de documentation.

### 3.1. Introduction générale : Python vs Java en POO

Commençons par un tableau récapitulatif des différences entre les deux langages.

| Aspect                   | Java                               | Python                                 |
|--------------------------|------------------------------------|--|
| Typage                   | Statique, explicite                | Dynamique, implicite                   |
| Portée des membres       | public, private, protected         | Convention : __, __                    |
| Constructeurs multiples  | Par surcharge                      | Paramètres par défaut, *args, **kwargs |
| Interfaces / abstraction | Interfaces, classes abstraites     | abc.ABC, duck typing                   |
| Polymorphisme            | Basé sur types et héritage         | Duck typing : comportement             |
| Surcharge (méthodes)     | Autorisée (signatures différentes) | Non native, simulation avec *args      |
| Surcharge (opérateurs)   | Non                                | Possible via __add__, __eq__, etc.     |

### 3.2. Création de classes et instances

Contrairement à Java, Python ne s'encombre pas de mots-clés comme `public`, `private`, `new`, `static` ou de la nécessité de prédefinir les types des attributs. Le langage repose sur un modèle de création d'objets plus simple et flexible. Cela peut paraître étrange pour un programmeur Java habitué au typage statique et à la vérification stricte à la compilation, mais cette souplesse est au cœur de la philosophie Python.

#### Exemple 37. Déclaration de variables

| En Java :  | En Python :   |
|--|---|
| <pre>public class Voiture {     String marque;      public Voiture(String marque) {         this.marque = marque;     }      public static void main(String[] args) {         Voiture v = new Voiture("Toyota");     } }</pre> | <pre>class Voiture:     def __init__(self, marque):         self.marque = marque  v = Voiture("Toyota")</pre> |

#### Différences clés :

- Pas de mot-clé `public`, `private`, `protected`, `this`, `new`, `static` en Python
- `self` est **obligatoire et explicite** pour accéder aux attributs
- Pas besoin de déclarer les types ni les attributs d'avance (on voit que **l'attribut marque est déclaré lors de son utilisation** dans le constructeur).
- Le **constructeur en Python** est la méthode spéciale `__init__()`, qui joue le même rôle que le constructeur en Java. Cependant, contrairement à Java, il n'est pas possible de surcharger plusieurs versions de `__init__` directement. Pour gérer différents cas d'instanciation, on utilise des paramètres optionnels.

### Exemple 38. Arguments optionnels pour un constructeur

```
class Utilisateur:
    def __init__(self, nom, age=None):
        self.nom = nom
        if age is not None:
            self.age = age
        else:
            self.age = "Non spécifié"

u1 = Utilisateur("Alice")
u2 = Utilisateur("Bob", 30)
```

Cela simule deux constructeurs, l'un avec seulement le nom, l'autre avec nom et âge, comme avec n'importe quelle fonction en python.

## 3.3. Attributs et accès

L'une des grandes différences entre Java et Python réside dans la gestion de la visibilité des attributs. Là où Java impose des modificateurs d'accès stricts pour encapsuler les données, Python repose sur une convention implicite : un attribut commençant par un underscore (ex. `_nom`) est considéré comme protégé (usage interne ou pour les sous-classes), tandis qu'un double underscore (ex. `__secret`) indique un attribut privé\*.

Cette approche plus souple permet d'accéder plus directement aux attributs, mais demande plus de rigueur de la part du développeur.

\*En fait il s'agit d'un pseudo-attribut privé qui sera renommé automatiquement par le mécanisme de name mangling (`_NomDeClasse__secret`). Le **name mangling** est un processus interne qui modifie le nom réel de l'attribut dans l'objet afin de limiter les risques de conflits ou d'accès accidentel. Par exemple, un attribut `__data` dans une classe `Compteur` deviendra automatiquement accessible sous le nom `_Compteur__data`. Cela n'empêche pas l'accès, mais dissuade les manipulations directes en rendant leur nom moins prévisible à l'extérieur de la classe.

### Exemple 39. Gestion des attributs

| En Java :   | En Python :  |
|---|--|
| <pre>public class Personne {     private String nom;     public String getNom() {         return nom;     } }</pre> | <pre>class Personne:     def __init__(self, nom):         self._nom = nom  @property</pre> |

```
public void setNom(String nom) {
    this.nom = nom;
}
```

```
def nom(self):
    return self._nom

@nom.setter
def nom(self, valeur):
    self._nom = valeur
```

En python :

- On accède directement à l'attribut sauf si on définit des @property, dans ce cas de l'appel au getter et au setter est implicite, et se fait avec la même syntaxe que l'appel à l'attribut lui-même, soit p.nom pour un objet p.
- \_nom est une convention pour "protected"
- \_\_nom (double underscore) représente un attribut privé (et enclenche le name mangling)

### 3.4. Méthodes et surcharges

En Python, la surcharge de méthode, comme on la connaît en Java, n'est pas directement supportée. Cependant, il existe des mécanismes pour obtenir un comportement similaire, principalement en utilisant les paramètres par défaut.

#### Exemple 40. Pas de surcharge liée au type des paramètres en python

| En Java :   | En Python :   |
|---|---|
| <pre>public class Calculette {     public int addition(int a, int b) {         return a + b;     }     public double addition(double a, double b)     {         return a + b;     } }</pre> | <pre>class Calculette:     def addition(self, a, b):         return a + b</pre> |

En Python, pas de surcharge native :

- Les types ne sont pas imposés
- Une seule définition de fonction par nom
- On peut simuler la surcharge avec des paramètres optionnels comme dans l'exemple du constructeur ci-dessus (exemple 31)

### 3.5. Attributs de classe (équivalent statique en Java)

En Python, on peut définir des attributs **au niveau de la classe**. Ils sont **partagés entre toutes les instances**.

#### Exemple 41. Attributs de classe

```
class Employe:
    compteur = 0 # attribut de classe (statique)

    def __init__(self, id_employe):
        self.id_employe = id_employe
        Employe.compteur += 1

c1 = Employe("E001")
c2 = Employe("E002")
print(Employe.compteur) # 2
```

#### Remarque

Compteur est partagé. Mais si une instance le modifie directement (`c1.compteur = 5`), cela crée en fait un nouvel attribut d'instance qui masque celui de la classe. C'est dangereux!!!

### 3.6. Méthodes de classe et méthodes statiques

En plus des méthodes d'instance, Python propose :

- **Méthodes de classe** : décorateur `@classmethod`, reçoivent la classe (`cls`) comme premier argument.
- **Méthodes statiques** : décorateur `@staticmethod`, ne reçoivent ni `self` ni `cls`. Ce sont simplement des fonctions rangées dans la classe pour des raisons de lisibilité.

#### Exemple 42. Méthodes de classe et statiques

```
class Employe:
    compteur = 0 # attribut de classe (statique)

    def __init__(self, id_employe):
        self.id_employe = id_employe
        Employe.compteur += 1

    @classmethod
    def combien_crees(cls):
        return cls.compteur

    @staticmethod
    def description_emploi():
        return "Le Cégep Limoilou est situé ..."
```

```
c1 = Employe("E001")
c2 = Employe("E002")

print(Employe.combien_crees())      # 2
print(Employe.description_emploi)   #Le Cégep Limoilou est situé
...
```

### 3.7. Héritage et polymorphisme

Comme en Java, l'héritage en Python permet à une classe d'en hériter une autre pour réutiliser ou redéfinir ses comportements. La syntaxe est simple et lisible : `class Enfant(Parent)`. Une classe enfant hérite automatiquement des attributs et méthodes publiques ou protégées de la classe parente, et peut les redéfinir selon ses besoins.

#### Exemple 43. Héritage en python

| En Java :  | En Python :   |
|--|---|
| <pre>class Animal {     void parler() {         System.out.println("...");     } }  class Chien extends Animal {     void parler() {         System.out.println("Wouf");     } }</pre> | <pre>class Animal:     def parler(self):         print("...")  class Chien(Animal):     def parler(self):         print("Wouf")</pre> |

#### Remarques :

- Pas besoin de mot-clé `extends` ou `implements`
- Le polymorphisme repose sur **le comportement**, pas le type, on appelle ça le **Duck Typing** (si l'objet se comporte comme un canard, c'en est un...). Si une méthode existe, elle sera appelée, même si l'objet n'est pas d'une classe prévue
- Comme en Java, une sous-classe peut redéfinir le constructeur (`__init__`) ou certaines méthodes de sa classe parente. Lorsqu'on veut réutiliser le code du parent avant ou après avoir ajouté des traitements spécifiques, on utilise la fonction `super()`.

- Python supporte également l'héritage multiple, ce qui signifie qu'une classe peut hériter de plusieurs classes parentes en même temps : `class C(A, B)`. Cette fonctionnalité peut être puissante mais doit être utilisée avec prudence pour éviter des conflits notamment grâce à la résolution d'ordre de méthode (MRO). Ce mécanisme détermine dans quel ordre les classes sont consultées lorsqu'une méthode est appelée. On peut consulter cet ordre avec `MaClasse.__mro__` ou `MaClasse.mro()`.

### 3.8. Interfaces et abstraction, module ABC

En Java, les interfaces et les classes abstraites sont un pilier central pour définir des comportements que plusieurs classes peuvent implémenter sans forcément partager d'héritage commun. En Python, ce rôle est souvent rempli de manière plus souple grâce au Duck Typing : si un objet fournit les méthodes attendues, il est accepté, sans avoir à hériter d'une interface formelle.

Toutefois, pour plus de rigueur, Python propose le module `abc` qui permet de créer des interfaces via des classes abstraites. Il faut pour cela faire hériter la classe abstraite de ABC

#### Exemple 44. Simuler l'abstraction et les interfaces en python

```
from abc import ABC, abstractmethod

class Vehicule(ABC):
    @abstractmethod
    def demarrer(self):
        pass

class Voiture(Vehicule):
    def demarrer(self):
        print("La voiture démarre")

class Avion(Vehicule):
    def toto(self):
        print("toto")

v = Voiture()
v.demarrer() # A
a = Avion()
a.demarrer()
```

Ici, linstanciation de la voiture est correcte, mais celle de lAvion va lever une erreur : “*TypeError: Can't instantiate abstract class Avion with abstract method demarrer*”.

### 3.9. Typage et déclaration des attributs en Python

Contrairement à Java où le typage des attributs est obligatoire et fait partie intégrante de la déclaration, Python utilise un système d'**annotations de type** optionnelles, via le module `typing`. Ces annotations ne sont pas strictement vérifiées à l'exécution mais servent d'aide à la lecture, à la documentation, et aux outils de vérification statique comme `mypy`.

#### Exemple 45. Utiliser les annotations de type

```
def carre(x: int) -> int:
    return x * x

class Personne:
    nom: str
    age: int
```

Je ne peux que vous encourager à les utiliser, au moins pour les attributs de classe dans un modèle objet, pour assurer la lisibilité et la rigueur dans votre code.

#### Remarque importante :

ici on déclare les attributs (nom et age) au niveau de la classe. On pourrait penser qu'il s'agit d'attributs de classe. Mais lors du premier appel (à `self.nom` par exemple) celui-ci sera automatiquement considéré comme une variable d'instance. Il est donc judicieux d'Accompagner les variables de classe d'un commentaire!!!

Pour déclarer simplement une classe avec des attributs typés et bénéficier automatiquement d'un constructeur, on utilise le module `dataclasses` :

#### Exemple 46. Utiliser le `dataclasses` et les attributs optionnels de `typing`

```
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class Personne:
    nom: str
    age: Optional[int] = None

p1 = Personne("Alice")
p2 = Personne("Bob", 21)

print(p1)
print(p2)
```

On constate à l'exécution, la gestion du `__str__()`

```
Python Console
Personne(nom='Alice', age=None)
Personne(nom='Bob', age=21)
```

### 3.10. Autres points utiles

| Fonctionnalité                 | Java   | Python  |
|--------------------------------|--|---|
| Affichage d'objet              | <code>toString()</code>                          | <code>__str__()</code> ou <code>__repr__()</code>                 |
| Égalité et comparaison d'objet | <code>equals()</code> et <code>hashCode()</code> | <code>__eq__()</code> et <code>__hash__()</code>                  |
| Gestion des exceptions         | Obligatoire pour <code>throws</code>             | Typée mais optionnelle  |
| Importation                    | <code>import package.Class;</code>               | <code>import module</code> ou <code>from module import ...</code> |

## 4. Fichiers et sérialisation

### 4.1. Lecture et écriture de fichiers en Python

En Python, la lecture et l'écriture de fichiers se font avec la fonction intégrée `open()`. On utilise généralement la clause `with` pour garantir la bonne fermeture du fichier, même en cas d'erreur. Sans entrer dans les détails cette clause permet de libérer la ressource utilisée pour ouvrir le fichier même si une exception survient.

**Exemple 47. Écriture et lecture d'un fichier texte**

```
# Écriture dans un fichier
with open("donnees.txt", "w") as f:
    f.write("Bonjour monde\n")

# Lecture du fichier
with open("donnees.txt", "r") as f:
    contenu = f.read()
    print(contenu)
```

Les modes d'ouverture les plus courants sont :

- '`r`' : lecture (`read`)
- '`w`' : écriture (`write`, écrase le fichier s'il existe)
- '`a`' : ajout à la fin (`append`)
- '`b`' : mode binaire (souvent utilisé avec images, objets sérialisés, etc.)

### 4.2. Sérialisation en Python

#### 4.2.1. Avec Pickle (sérialisation binaire)

La sérialisation est le processus de conversion d'un objet Python en une forme stockable (dans un fichier ou à envoyer sur le réseau), puis sa restauration ultérieure.

Python propose plusieurs outils pour cela, dont le plus courant est le module `pickle`.

### Exemple 48. Sérialisation binaire avec pickle d'un objet

```
import pickle
```

```
class Personne :  
    def __init__(self, nom, age):  
        self.nom=nom  
        self.age=age
```

```
# dictionnaire à sérialiser  
mon_objet = Personne(nom="Alice", age=30)  
  
# Sérialisation (sauvegarde dans un fichier binaire)  
with open("data.pkl", "wb") as f:  
    pickle.dump(mon_objet, f)  
  
# Désérialisation (lecture depuis le fichier)  
with open("data.pkl", "rb") as f:  
    dictionnaire_charge = pickle.load(f)  
  
print(dictionnaire_charge)
```

#### Remarques :

- pickle fonctionne bien pour la plupart des objets Python.
- La sérialisation est complètement transparente, on a besoin d'écrire aucun code.
- Le format n'est **pas lisible par d'autres langages** et peut **présenter des risques de sécurité** (ne jamais charger un fichier pickle non fiable).
- Pour une sérialisation plus portable (et plus sûre), on peut utiliser le module json (mais il est **limité aux types de base** comme dictionnaires, listes, chaînes, nombres).

#### 4.2.2. Avec Json (sérialisation textuelle)

C'est le format le plus courant. Il est basé sur la syntaxe de liste et de dictionnaire. Comme json ne sait gérer que les **types de base** (dict, list, str, int, float, bool, None), il faut convertir les objets de classe personnalisée en dictionnaires.

On fait ça en ajoutant :

- une méthode **to\_dict()** → transforme l'objet en dictionnaire sérialisable en JSON,

- une méthode **from\_dict()** (souvent @classmethod) → recrée l'objet à partir du dictionnaire.

### Exemple 49. Sérialisation au format json

```

import json

class Etudiant:
    def __init__(self, nom, age, notes=[]):
        self.nom = nom
        self.age = age
        self.notes = notes

    def __str__(self):
        return f"Etudiant(nom={self.nom}, age={self.age}, notes={self.notes})"

    # ◊ Conversion en dict (sérialisable en JSON)
    def to_dict(self):
        return {
            "nom": self.nom,
            "age": self.age,
            "notes": self.notes
        }

    # ◊ Reconstruction depuis un dict
    @classmethod
    def from_dict(cls, data):
        # appel du constructeur à 2 paramètres
        etu = cls(data["nom"], data["age"])
        etu.notes = data["notes"]
        return etu

# -----
# Exemple d'utilisation
# -----

# Création d'un objet
etu = Etudiant("Alice", 20, [15, 18, 17])
print("Objet original :", etu)

# Sérialisation en JSON
with open("etudiants.json", "w", encoding="utf-8") as f:
    json_str = json.dump(etu.to_dict(), f)

# Déserialisation depuis JSON
with open("etudiants.json", "r", encoding="utf-8") as f:

```

```
data = json.load(f)
etudiant = Etudiant.from_dict(data)
print("Objet reconstruit :", etudiant)
```

**Remarques :**

- Le format JSON est plus adapté à l'échange entre langages (Python, Java, JavaScript, etc.), et est plus sécuritaire!
- On peut gérer manuellement les attributs que l'on souhaite sérialiser ou non. Avec pickle on peut aussi le faire mais il faut utiliser des mécanismes comme field() du module dataclasses()
- Il existe un outil de dataclass qui permet de serialiser automatiquement les objets. Il faut ajouter les décorateurs `@dataclass_json` et `@dataclass`. Dans ce cas, il faut absolument laisser gérer les constructeurs au module dataclasses. A noter que l'on peut ajouter quand même du code spécifique à la construction de l'objet en redéfinissant la méthode `__post_init__()`

**Exemple 50. Utilisation de dataclasses:**

```
from dataclasses import dataclass
from dataclasses_json import dataclass_json
import json

@dataclass_json
@dataclass
class Note:
    matiere: str=None
    note:int=0

    def __post_init__(self):
        if not 0<=self.note<=100:
            raise ValueError("Note invalide!!")

@dataclass_json
@dataclass
class Etudiant:
    nom: str
    age: int
    notes: list[Note]
```

```

        def __str__(self):
            return f"Etudiant(nom={self.nom}, age={self.age},
notes={self.notes})"

# -----
# Exemple d'utilisation
# -----

# Création d'un objet
etu = Etudiant("Alice", 20, [Note("Prog", 80), Note("Calcul
Diff", 58), Note("Educ", 95)])
print("Objet original :", etu)

# Sérialisation en JSON
with open("etudiants.json", "w", encoding="utf-8") as f:
    print(etu.to_json())

    json_str = json.dump(etu.to_json(), f)

# Déserialisation depuis JSON
with open("etudiants.json", "r", encoding="utf-8") as f:
    json_str = json.load(f)
    print(json_str)
    etudiant = Etudiant.from_json(json_str)
    print("Objet reconstruit :", etudiant)

```

### 4.3. Gestion des exceptions en Python

En Python, les exceptions permettent de gérer proprement les erreurs qui peuvent survenir à l'exécution (comme une division par zéro, l'ouverture d'un fichier inexistant, etc.). Cela permet d'éviter que le programme ne plante brutalement.

On utilise un bloc `try` pour encapsuler le code à risque. Si une erreur se produit, le bloc `except` prend le relais :

```

try:
    x = int(input("Entrez un entier : "))
    y = 10 / x
except ZeroDivisionError:
    print("Erreur : division par zéro.")
except ValueError:
    print("Erreur : entrée invalide.")
finally:
    print("Fin de la tentative.")

```

Le bloc `finally`, optionnel, est toujours exécuté, qu'il y ait eu une erreur ou non — utile pour libérer des ressources ou fermer un fichier, par exemple.

### Lever une exception avec raise

Le mot-clé `raise` permet de **générer manuellement une exception**, soit pour signaler un problème, soit pour relancer une exception capturée. Par exemple :

```
def racine(x):
    if x < 0:
        raise ValueError("Impossible de calculer la racine d'un nombre
négatif.")
    return x ** 0.5

try:
    print(racine(-4))
except ValueError as e:
    print(f"Erreur : {e}")
```

On peut aussi relancer une exception interceptée pour qu'elle remonte plus haut :

```
try:
    raise RuntimeError("Quelque chose s'est mal passé")
except RuntimeError as e:
    print("Traitement local...")
    raise # relance l'exception
```

Il est aussi possible de définir ses propres exceptions personnalisées en dérivant de la classe `Exception`.

## 5. Interfaces graphiques en python avec QT

Il existe plusieurs framework pour les interfaces graphiques (dont TkInter). Dans le cadre de ce cours on utiliser plutôt QT, plus flexible et aux fonctionnalités plus nombreuses et plus actuelles.

### 5.1. Présentation de Qt

Qt est un framework libre et multiplateforme écrit en C++, conçu pour le développement d'interfaces graphiques (GUI) et d'applications multiplateformes. Il est utilisé dans de nombreux domaines, du desktop au mobile, en passant par l'embarqué.

Créé initialement en 1991 par l'entreprise norvégienne Trolltech, Qt est désormais maintenu par The Qt Company. Il est actuellement à sa version 6.x, avec un support toujours actif de la version 5.x. Plusieurs bindings Python existent pour Qt, notamment :

- **PyQt** (développé par Riverbank Computing)
- **PySide** (maintenu par The Qt Company)

Qt est utilisé pour construire des applications comme :

- Interfaces utilisateur riches
- Outils de configuration
- Logiciels scientifiques
- Applications mobiles et embarquées

**Dans le cadre du cours on utilisera PyQt6**

### 5.2. Installation et configuration

Assurez-vous d'avoir Python 3.7 ou plus récent installé. Vous pouvez utiliser l'installateur officiel depuis <https://www.python.org>.

Via pip : 3.7 ou plus récent installé, on installe PyQt6

```
pip install PyQt6
```

Évitez d'utiliser l'outil intégré de Pycharm, comme on le fait d'habitude, il gère mal l'installation.

## 5.3. Premier programme avec PyQt6

### 5.3.1. Crédit d'une fenêtre simple.

#### Exemple 51. Fenêtre simple et vide QT

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
fenetre = QWidget()
fenetre.setWindowTitle("Ma première fenêtre Qt")
fenetre.resize(300, 200)
fenetre.show()
sys.exit(app.exec())
```

#### Remarques :

- L'appel à `app.exec()` démarre la boucle d'événements de Qt : c'est ce qui permet à l'interface de répondre aux interactions de l'utilisateur (clics, frappes, redimensionnements, etc.).
- `sys.exit(...)` est une fonction du module `sys` de Python qui termine le programme proprement. On peut lui passer un entier (code de sortie) ou un objet (souvent une chaîne de caractères). En principe si tout va bien le code devrait être 0.

### 5.3.2.3 composants de base de QT:

- `QPushButton` : un simple bouton cliquable
- `QLabel` : une étiquette de texte ou d'image, non éditable par l'utilisateur
- `QLineEdit` : un champ de saisie de texte.

#### Exemple 52. Disposer trois composant dans une fenêtre

```
from PyQt6.QtWidgets import QApplication, QLineEdit, QVBoxLayout,
QWidget, QPushButton, QLabel
import sys

app = QApplication(sys.argv)
fenetre = QWidget()
fenetre.setWindowTitle("Fenêtre à 3 composants")

champ = QLineEdit()
champ.setPlaceholderText("Entrez votre nom")
```

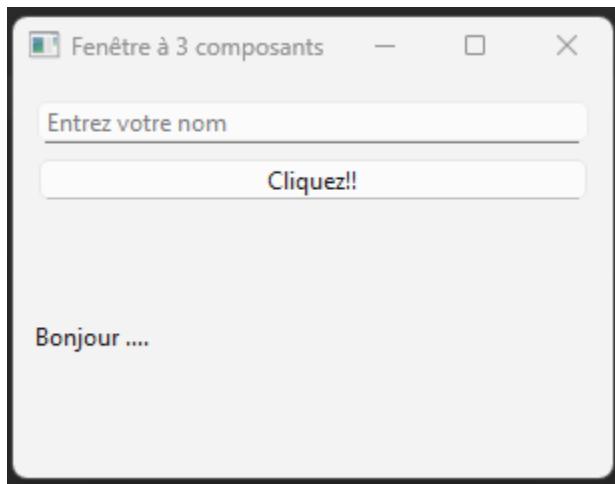
```
bouton = QPushButton()
bouton.setText("Cliquez!!")

label = QLabel()
label.setText("Bonjour ....")

layout = QVBoxLayout()
layout.addWidget(champ)
layout.addWidget(bouton)
layout.addWidget(champ)
fenetre.setLayout(layout)

fenetre.resize(300, 200)
fenetre.show()
sys.exit(app.exec())
```

Et voici le résultat d'exécution :



On observe dans cet exemple :

- La création des 3 composants et leur configuration (élémentaire ici)
- L'ajout d'une couche indispensable appelée Layout à laquelle on ajoute les composants et qui doit être passé à la fenêtre dont on va parler un peu après
- L'interface ne fait rien il va falloir configurer des réactions aux entrées de l'utilisateur, on y vient aussi.

### 5.3.3. Les gestionnaires de mise en page (Layouts)

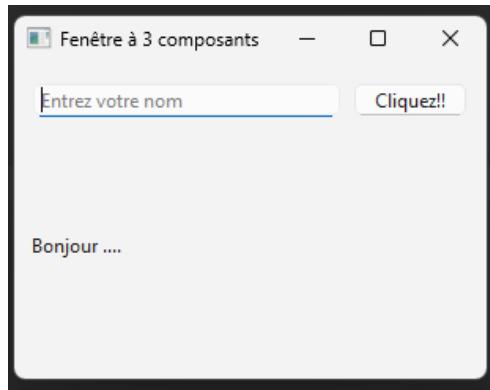
Qt fournit plusieurs gestionnaires de mise en page (QVBoxLayout, QHBoxLayout, QGridLayout) pour organiser les widgets dans les fenêtres.

**Exemple 53. Utilisation de QVBoxLayout et QHBoxLayout:**

```
from PyQt6.QtWidgets import QApplication, QLineEdit,  
QVBoxLayout, QHBoxLayout, QWidget, QPushButton, QLabel  
import sys  
  
app = QApplication(sys.argv)  
fenetre = QWidget()  
fenetre.setWindowTitle("Fenêtre à 3 composants")  
  
champ = QLineEdit()  
champ.setPlaceholderText("Entrez votre nom")  
  
bouton = QPushButton()  
bouton.setText("Cliquez !!")  
  
label = QLabel()  
label.setText("Bonjour ....")  
  
hLayout = QHBoxLayout()  
hLayout.addWidget(champ)  
hLayout.addWidget(bouton)  
  
vLayout = QVBoxLayout()  
  
vLayout.addLayout(hLayout)  
vLayout.addWidget(label)  
fenetre.setLayout(vLayout)  
  
fenetre.resize(300, 200)  
fenetre.show()  
sys.exit(app.exec())
```

Remarquons la méthode addLayout() du layout qui permet l'imbrication de layout

Et voici le résultat d'exécution :



La gestion des layouts pour un redimensionnement de fenêtre fonctionnel fera l'objet d'un paragraphe détaillé

#### 5.3.4. Réagir aux actions de l'utilisateur (signaux et slots)

Un **signal** est émis lorsqu'un événement se produit (ex. clic sur un bouton). Un **slot** est une méthode appelée en réponse.

Pour connecter l'un à l'autre on procède ainsi:

```
bouton.clicked.connect (ma_fonction)
```

où `ma_fonction` est une méthode dans laquelle on effectue le travail souhaité. On peut connecter un signal à n'importe quelle fonction ou méthode compatible.

Dans cet exemple le **signal** est le `clicked` défini dans `QPushButton` et le **slot** est la méthode `ma_fonction`

**Exemple 54. Réaction au clic du boutons**

```
from PyQt6.QtWidgets import QApplication, QLineEdit,  
QVBoxLayout, QHBoxLayout, QWidget, QPushButton, QLabel  
import sys  
  
def afficher_salutations():  
    label.setText("Bonjour " + champ.text() + " bienvenue dans ta  
premiere application graphique!")  
  
app = QApplication(sys.argv)  
fenetre = QWidget()  
fenetre.setWindowTitle("Fenêtre à 3 composants")  
  
champ = QLineEdit()  
champ.setPlaceholderText("Entrez votre nom")  
  
bouton = QPushButton()  
bouton.setText("Cliquez!!")  
bouton.clicked.connect(afficher_salutations)  
  
label = QLabel()  
label.setText("Bonjour")  
  
hLayout = QHBoxLayout()  
hLayout.addWidget(champ)  
hLayout.addWidget(bouton)  
  
vLayout = QVBoxLayout()  
  
vLayout.addLayout(hLayout)  
vLayout.addWidget(label)  
fenetre.setLayout(vLayout)  
  
fenetre.resize(300, 200)  
fenetre.show()  
sys.exit(app.exec())
```

Remarquons qu'un composant Qt possède un certain nombre de signaux prédéfinis qui peuvent transporter avec eux des paramètres. On accède à ces paramètres directement dans la méthode appelée.

**Exemple 55. Un signal avec un paramètre**

```
from PyQt6.QtWidgets import QApplication, QLineEdit,
QVBoxLayout, QHBoxLayout, QWidget, QPushButton, QLabel
import sys

def on_text_changed(value):
    label.setText("Bonjour " + value + " bienvenue dans ta premiere
application graphique!")

app = QApplication(sys.argv)
fenetre = QWidget()
fenetre.setWindowTitle("Fenêtre à 2 composants")

champ = QLineEdit()
champ.setPlaceholderText("Entrez votre nom")
champ.textChanged.connect(on_text_changed)

label = QLabel()
label.setText("")

vLayout = QVBoxLayout()
vLayout.addWidget(champ)
vLayout.addWidget(label)

fenetre.setLayout(vLayout)

fenetre.resize(300, 200)
fenetre.show()
sys.exit(app.exec())
```

La [documentation officielle de riverBankComputing](#) est une bonne référence pour connaître la liste des signaux d'un composant.

On verra plus tard comment on peut envoyer soi-même un signal.

## 5.4. Callbacks et fonctions lambda

### 5.4.1. La notion de callback

Sans y prendre vraiment garde, on a utilisé quelque chose de très nouveau, **la possibilité de passer une méthode en paramètre d'une méthode**, ce que l'on appelle un callback. En programmation, un callback est une fonction que l'on passe en argument à une autre fonction, pour qu'elle soit appelée plus tard, généralement lorsqu'un événement se produit. Ce mécanisme est central dans la programmation événementielle, comme avec PyQt, où

l'on connecte des événements (signaux) (clics, saisie, curseurs, etc.) à des fonctions de traitement (slots).

Dans l'exemple précédent, le paramètre `afficher_salutations` est un callback sur la fonction `afficher_salutations()`

```
bouton.clicked.connect(afficher_salutations)
```

#### 5.4.2. Fonctions lambda

En Python, il est fréquent d'utiliser des fonctions anonymes (sans nom), définies à la volée avec le mot-clé `lambda`, pour créer rapidement des callbacks simples. La syntaxe est :

```
lambda param1, param2, ...: expression
```

`param1`, `param2`, etc jouent le rôle de paramètre de la fonction alors que `expression` est renvoyée directement, sans mot-clé `return`.

```
# Fonction équivalente à : def f(x): return x * 2
f = lambda x: x * 2
print(f(4)) # Affiche 8
```

En contexte PyQt, l'usage de `lambda` est très courant pour connecter un signal à une logique courte :

```
slider.valueChanged.connect(lambda val: print(f"Valeur: {val}"))
combo.currentTextChanged.connect(lambda txt: label.setText(f"Choix: {txt}"))
```

Comme en Java, les `lambda` en Python sont utiles quand on ne veut pas définir une fonction entière juste pour une action simple. La différence est qu'en Python, `lambda` est limité à une seule expression, et on ne peut pas faire d'affectation ce qui en limite l'usages.

*Indication : A ce stade, réaliser l'exercice 1 du formatif de la semaine*

Comme on l'a vu dans cet exercice, la pratique est de déclarer une classe, qui hérite d'un composant de base de QT (QWidget, QMainWindow ou QDialog), ce qui nous permet de travailler dans un environnement de type orienté objet propre. On verra plus tard la différence entre ces composants de base.

#### 5.5. Utilisation de QTDesigner

Qt Designer est un outil graphique fourni avec Qt qui permet de créer des interfaces utilisateur visuellement, sans écrire de code. Il produit des fichiers `.ui`, qui décrivent l'interface en XML. Ces fichiers peuvent ensuite être utilisés directement dans une application PyQt.

On ne va pas détailler l'utilisation de l'interface de QTDesigner ici, ce serait fastidieux et peu utile, mais bien qu'elle soit assez intuitive, il est important de se familiariser avec l'interface. On mentionnera simplement que la sauvegarde génère un fichier .ui. On ajoute alors ce fichier au projet Pycharm dans un répertoire ui en général.

Pour charger un fichier .ui dans une application Python, on utilise la fonction loadUi disponible dans le module PyQt6.uic.

#### **Exemple 56. Utilisation d'un fichier .ui**

Voici le contenu du fichier Exemple45.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
<width>372</width>
<height>135</height>
</rect>
</property>
<property name="windowTitle">
<string>MainWindow</string>
</property>
<widget class="QWidget" name="centralwidget">
<widget class="QPushButton" name="button">
<property name="geometry">
<rect>
<x>50</x>
<y>30</y>
<width>75</width>
<height>23</height>
</rect>
</property>
<property name="text">
<string>Clique</string>
</property>
</widget>
<widget class=" QLabel" name="label">
<property name="geometry">
<rect>
<x>150</x>
<y>30</y>
<width>91</width>
<height>21</height>
</rect>
</property>
<property name="text">
<string>Vous avez cliqué</string>
```

```

</property>
</widget>
<widget class="QLabel" name="compteCLiqueLabel">
<property name="geometry">
<rect>
<x>240</x>
<y>30</y>
<width>31</width>
<height>21</height>
</rect>
</property>
<property name="text">
<string>0</string>
</property>
</widget>
<widget class="QLabel" name="label_2">
<property name="geometry">
<rect>
<x>260</x>
<y>30</y>
<width>91</width>
<height>21</height>
</rect>
</property>
<property name="text">
<string>fois</string>
</property>
</widget>
</widget>
<widget class="QMenuBar" name="menubar">
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
<width>372</width>
<height>21</height>
</rect>
</property>
</widget>
<widget class="QStatusBar" name="statusbar"/>
</widget>
<resources/>
<connections/>
</ui>

```

Et le fichier .py qui permet de le charger:

Ici, on commence par créer une sous-classe Fenetre de QMainWindow (ou de QDialog), et on appelle la méthode loadUi (nom\_fichier, fenetre) dans le constructeur.

```

import sys
from PyQt6.QtWidgets import QApplication, QWidget, QMainWindow

```

```
from PyQt6.uic import loadUi

class Fenetre(QMainWindow):
    compteClic:int = 0
    def __init__(self):
        super().__init__()
        loadUi("ui/exemple45.ui", self)
        self.button.clicked.connect(self.incrementer_compteur)

    def incrementer_compteur(self):
        self.compteClic+=1
        self.compteCLiqueLabel.setText(str(self.compteClic))

app = QApplication(sys.argv)
fenetre = Fenetre()
fenetre.show()
sys.exit(app.exec())
```

Il est important de noter que les variable `button` et `compteCLiqueLabel` ont été générées par l'appel de la méthode `loadUI()`. On constate également que la complétion automatique de l'IDE ne fonctionne pas même si tout est correct à l'exécution.

**La solution est de déclarer avec indication de type les variables correspondant aux composants.** C'est plus lisible et l'IDE s'y retrouve! Je vous recommande de toujours conserver cette pratique!

```
class Fenetre(QMainWindow):
    compteClic:int = 0
    button : QPushButton
    compteCLiqueLabel : QLabel
```

### Deux remarques importantes :

- Si on change le nom d'un widget dans Qt Designer (par exemple `button` → `monBouton`), alors **il faut aussi utiliser `self.monBouton` dans le code.**
- L'utilisation de QTDesigner permet de séparer la **logique de l'interface** (dans Python) de sa **construction graphique** (via Qt Designer). C'est une méthode très pratique pour développer des interfaces complexes de manière modulaire et rapide.
- Il existe un mécanisme (`pyuic`) permettant de générer le code correspondant à ce qui est fait dans `loadui()`, mais on ne l'examinera pas dans le cadre de ce cours.

## 5.6. Principaux composants graphiques (Widgets)

**La classe de base QWidget dont héritent tous les composants graphiques**

- Le **composant de base** de toutes les interfaces Qt.
- Peut servir de fenêtre, de conteneur ou de base pour d'autres composants

### 5.6.1. Les Fenêtres principales

| Widget      | Description   | Exemple d'utilisation |
|-------------|---|-----------------------|
| QMainWindow | Fenêtre principale d'application avec barre de menu, toolbar, statusbar | QMainWindow()         |
| QDialog     | Fenêtre de dialogue modale ou non-modale                                | QDialog()             |
| QWidget     | Fenêtre/widget de base, conteneur générique                             | QWidget()             |

#### QWidget

- C'est la classe de base **de tous les composants graphiques** (labels, boutons, etc.).
- On peut utiliser un QWidget comme **fenêtre principale minimale**.
- Il n'a **aucune "structure" prédéfinie** : pas de barre de menus, pas de status bar, pas de dock...

Pour une fenêtre simple, juste avec un layout et des widgets, on peut utiliser un QWidget.

#### QMainWindow

- C'est une sous-classe de QWidget prévue pour une **fenêtre d'application classique**.
- Elle apporte un **cadre standard** :
  - menuBar() → barre de menus
  - statusBar() → barre d'état en bas
  - toolBar et dockWidget possibles
  - setCentralWidget(widget) → zone principale de la fenêtre

Pour une application "classique" avec menu, barre d'état, outils, on utilise une QMainWindow.

#### QDialog

- C'est une sous-classe de QWidget, conçue pour être **une fenêtre modale ou secondaire** (boîte de dialogue).
- Typiquement : fenêtre de préférences, boîte d'alerte, popup de saisie...
- Peu-être **modale** ( bloque le reste tant qu'elle est ouverte) ou **non modale**.
- A un mécanisme de retour (accept(), reject(), exec()) pour savoir ce que l'utilisateur a choisi.

Pour une **fenêtre secondaire** (comme un "paramètres" ou "sauvegarder sous..."), on utilise une QDialog.

#### **Exemple 57. Utilisation des trois types principaux de fenêtres**

```
import sys
from PyQt6.QtWidgets import (
    QApplication, QWidget, QMainWindow, QDialog,
    QVBoxLayout, QPushButton, QLabel
)

class FenetreWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("QWidget simple")
        layout = QVBoxLayout()
        layout.addWidget(QLabel("Ceci est un QWidget utilisé comme fenêtre"))
        self.setLayout(layout)

class FenetreMain(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("QMainWindow")

        # Zone centrale-Obligatoire pour une main window
        # Automatique quand on utilise QTDesigner!
        central = QWidget()
        layout = QVBoxLayout()
        layout.addWidget(QLabel("Zone centrale du QMainWindow"))
        central.setLayout(layout)
        self.setCentralWidget(central)

        # Barre d'état
        self.statusBar().showMessage("Prêt")

        # Barre de menu
        menu = self.menuBar().addMenu("Fichier")
        menu.addAction("Quitter", self.close)
```

```

class FenetreDialog(QDialog):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("QDialog")
        layout = QVBoxLayout()
        layout.addWidget(QLabel("Ceci est une boîte de dialogue"))
        bouton_ok = QPushButton("OK")
        bouton_ok.clicked.connect(self.accept)
        layout.addWidget(bouton_ok)
        self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)

    w = FenetreWidget()
    w.show()

    m = FenetreMain()
    m.show()

    d = FenetreDialog()
    d.show()  # Non modale ; si on voulait bloquer : d.exec()

    sys.exit(app.exec())

```

## 5.6.2. Les Widgets de base

| Widget       | Description                             | Exemple d'utilisation    |
|--------------|---|--------------------------|
| QLabel       | Affiche du texte ou une image.          | QLabel("Bonjour")        |
| QPushButton  | Bouton cliquable.                       | QPushButton("OK")        |
| QLineEdit    | Champ de saisie d'une ligne.            | QLineEdit()              |
| QTextEdit    | Zone de texte multiligne.               | QTextEdit()              |
| QCheckBox    | Case à cocher.                          | QCheckBox("Accepter")    |
| QRadioButton | Bouton radio (exclusif dans un groupe). | QRadioButton("Option A") |
| QComboBox    | Menu déroulant (drop-down).             | QComboBox()              |

| Widget         | Description                     | Exemple d'utilisation              |
|----------------|---------------------------------|------------------------------------|
| QSpinBox       | Champ numérique entier.         | QSpinBox()                         |
| QDoubleSpinBox | Champ numérique à virgule.      | QDoubleSpinBox()                   |
| QSlider        | Curseur horizontal ou vertical. | QSlider(Qt.Orientation.Horizontal) |
| QProgressBar   | Barre de progression.           | QProgressBar()                     |

**Exemple 58. Utilisation des principaux widgets :**

```

import sys
from PyQt6.QtWidgets import (
    QApplication, QWidget, QVBoxLayout,
    QLabel, QPushButton, QLineEdit,
    QCheckBox, QRadioButton, QComboBox,
    QSpinBox, QSlider, QProgressBar
)
from PyQt6.QtCore import Qt

app = QApplication(sys.argv)
fenetre = QWidget()
fenetre.setWindowTitle("Exemples de Widgets Qt")
layout = QVBoxLayout()

# QLabel
label = QLabel("Bonjour Qt")
layout.addWidget(label)

# QPushButton
button = QPushButton("Cliquez-moi")
def on_click():
    label.setText("Bouton cliqué")
button.clicked.connect(on_click)
layout.addWidget(button)

# QLineEdit
champ = QLineEdit()
champ.setPlaceholderText("Entrez votre nom")
champ.textChanged.connect(lambda txt: label.setText(f"Bonjour {txt}"))
layout.addWidget(champ)

# QCheckBox
checkbox = QCheckBox("Activer quelque chose")
checkbox.stateChanged.connect(lambda state: label.setText("Activé" if
state else "Désactivé"))
layout.addWidget(checkbox)

# QRadioButton

```

```

radio = QRadioButton("Choisir cette option")
radio.toggled.connect(lambda checked: label.setText("Option
sélectionnée" if checked else ""))
layout.addWidget(radio)

# QComboBox
combo = QComboBox()
combo.addItems(["Option 1", "Option 2", "Option 3"])
combo.currentTextChanged.connect(lambda text: label.setText(f"Choix :
{text}"))
layout.addWidget(combo)

# QSpinBox
spin = QSpinBox()
spin.setRange(0, 100)
spin.valueChanged.connect(lambda val: label.setText(f"Valeur : {val}"))
layout.addWidget(spin)

# QSlider
slider = QSlider(Qt.Orientation.Horizontal)
slider.setRange(0, 100)
slider.valueChanged.connect(lambda val: label.setText(f"Slider :
{val}"))
layout.addWidget(slider)

# QProgressBar
progress = QProgressBar()
progress.setRange(0, 100)
slider.valueChanged.connect(progress.setValue)
layout.addWidget(progress)

fenetre.setLayout(layout)
fenetre.resize(300, 500)
fenetre.show()
sys.exit(app.exec())

```

### 5.6.3. Les Conteneurs et l'organisation

| Widget         | Description  | Exemple                   |
|----------------|--|---------------------------|
| QGroupBox      | Encadre un groupe de widgets avec un titre.                    | QGroupBox("Informations") |
| QFrame         | Zone de délimitation personnalisable.                          | QFrame()                  |
| QTabWidget     | Interface à onglets.   | QTabWidget()              |
| QStackedWidget | Affiche une page à la fois (utile pour des écrans successifs). | QStackedWidget()          |

Ce sont tous des **conteneurs** (QWidget spécialisés) mais chacun a un rôle bien différent.

### QGroupBox

- Sert à **regrouper visuellement des widgets liés**.
- Affiche un cadre + un **titre** (optionnel).
- Ne gère pas la navigation entre plusieurs contenus → juste un regroupement.
- Exemple classique : un formulaire où tu mets "Coordonnées personnelles" d'un côté, "Adresse" de l'autre.

Sert à **structurer** l'interface pour l'utilisateur.

#### Exemple 59. Utilisation de QGroupBox

```
box = QGroupBox("Informations")
layout = QVBoxLayout()
layout.addWidget(QLabel("Nom:"))
layout.addWidget(QLabel("Prénom:"))
box.setLayout(layout)
```

### QFrame

- C'est un QWidget de base qui peut avoir un **cadre personnalisable** (bordures, styles...).
- Très utilisé pour :
  - séparer visuellement des zones (ligne horizontale, verticale)
  - créer un fond encadré autour d'un widget
- Pas de titre, contrairement à QGroupBox.

Sert à **décorer** ou **séparer** visuellement.

#### Exemple 60. Utilisation de QFrame

```
frame = QFrame()
layout = QVBoxLayout()
layout.addWidget(QLabel("Nom:"))
layout.addWidget(QLabel("Prénom:"))
frame.setLayout(layout)
```

```
frame.setFrameShape(QFrame.Shape.Box) # Box, HLine, VLine...
frame.setFrameShadow(QFrame.Shadow.Sunken)
```

### QTabWidget

- Un conteneur avec **plusieurs onglets**.
- Chaque onglet contient un widget différent (souvent un layout complet).
- Exemple : dans les préférences d'une appli → "Général", "Affichage", "Avancé".

Sert à **organiser des écrans parallèles**, faciles à basculer.

#### Exemple 61. Utilisation de QTabWidgets

```
tabs = QTabWidget()
tabs.addTab(QWidget(), "Onglet 1")
tabs.addTab(QWidget(), "Onglet 2")
layout.addWidget(tabs)
```

### QStackedWidget

- Conteneur qui contient plusieurs widgets **superposés**, mais **un seul visible à la fois**.
- Contrairement à QTabWidget, **pas d'onglets visibles** → c'est toi qui décides quand changer de page (par code ou via un bouton/menu).
- Exemple : assistant de configuration (écran 1 → "Bienvenue", écran 2 → "Options", écran 3 → "Résumé").

Sert à **basculer entre des écrans** (wizard, navigation multi-écrans).

#### Exemple 62. Utilisation de QStackedWidgets

```
stack = QStackedWidget()
stack.addWidget.QLabel("Page 1"))
stack.addWidget.QLabel("Page 2"))
stack.setCurrentIndex(0) # afficher la page 1
```

### 5.6.4. Affichage de listes et de données

| Widget      | Description                               | Exemple       |
|-------------|---|---------------|
| QListWidget | Liste simple d'éléments (avec sélection). | QListWidget() |
| QTreeWidget | Arborescence (dossiers/fichiers, etc.).   | QTreeWidget() |

| Widget       | Description                   | Exemple            |
|--------------|-------------------------------|--------------------|
| QTableWidget | Tableau à colonnes et lignes. | QTableWidget(3, 4) |

```

import sys
from PyQt6.QtWidgets import ( QApplication, QWidget, QHBoxLayout,
                             QListWidget, QTableWidget,
                             QTableWidgetItem, QTreeWidget, QTreeWidgetItem)

app = QApplication(sys.argv)

fenetre = QWidget()
fenetre.setWindowTitle("Liste, Tableau et Arbre")
layout = QHBoxLayout()
fenetre.setLayout(layout)

# --- QListWidget ---
list_widget = QListWidget()
list_widget.addItems(["Élément 1", "Élément 2", "Élément 3"])
layout.addWidget(list_widget)

# --- QTableWidget ---
table_widget = QTableWidget(3, 2) # 3 lignes, 2 colonnes
table_widget.setHorizontalHeaderLabels(["Colonne 1", "Colonne 2"])
table_widget.setItem(0, 0, QTableWidgetItem("A1"))
table_widget.setItem(0, 1, QTableWidgetItem("B1"))
table_widget.setItem(1, 0, QTableWidgetItem("A2"))
table_widget.setItem(1, 1, QTableWidgetItem("B2"))
table_widget.setItem(2, 0, QTableWidgetItem("A3"))
table_widget.setItem(2, 1, QTableWidgetItem("B3"))
layout.addWidget(table_widget)

# --- QTreeWidget ---
tree_widget = QTreeWidget()
tree_widget.setColumnCount(1)
tree_widget.setHeaderLabels(["Arbre"])
parent = QTreeWidgetItem(tree_widget, ["Parent 1"])
QTreeWidgetItem(parent, ["Enfant 1"])
QTreeWidgetItem(parent, ["Enfant 2"])
parent2 = QTreeWidgetItem(tree_widget, ["Parent 2"])
QTreeWidgetItem(parent2, ["Enfant 3"])
layout.addWidget(tree_widget)

fenetre.show()
sys.exit(app.exec())

```

Pour des besoins plus avancés, on utilise plutôt les **modèles et vues** (QListView, QTableView et QTreeView), on les étudiera plus tard.

Je vous rappelle que la [documentation officielle de riverBankComputing](#) est source d'information indispensable pour travailler avec les composants QT.

## 5.7. Gérer l'état actif (grisé ou non) des widgets dans PyQt

Dans une interface graphique, il est courant de vouloir **activer ou désactiver certains widgets** (les rendre « grisés », donc non interactifs) selon le contexte ou l'état de l'application. En PyQt, cela se fait facilement via la méthode `setEnabled(bool)`.

Par exemple, pour désactiver un bouton :

```
mon_bouton.setEnabled(False) # Le bouton est désactivé (grisé)  
et pour le réactiver :
```

```
mon_bouton.setEnabled(True)
```

Il existe aussi une méthode équivalente plus explicite en négatif : `setDisabled(True)`.

### Exemple 63. Activer un bouton uniquement seulement si une case est cochée

```
checkbox = QCheckBox("Activer le bouton")  
bouton = QPushButton("Action")  
bouton.setEnabled(False)  
  
checkbox.stateChanged.connect(lambda state: bouton.setEnabled(state ==  
Qt.CheckState.Checked))
```

Ce code vérifie si la case est cochée `Qt.CheckState.Checked` et active ou désactive le bouton en conséquence.

### Remarques importantes sur les bonnes pratiques :

- **Initialiser clairement l'état** des widgets dès la création de la fenêtre (ex. : certains boutons peuvent être désactivés par défaut).
- **Centraliser la logique d'activation/désactivation** dans des méthodes si plusieurs widgets sont liés.
- **Éviter le hard-coding de True/False** sans justification : utiliser des conditions claires pour refléter la logique métier.

## 5.8. Gestion des layouts pour le redimensionnement de fenêtre

Les **layouts** sont des **gestionnaires de placement automatique** des widgets dans une fenêtre ou un conteneur. Plutôt que de positionner les widgets manuellement avec `setGeometry()`, les layouts s'occupent du **redimensionnement**, de l'alignement et de l'espacement, ce qui rend l'interface **adaptative** à la taille de la fenêtre.

### 1. QBoxLayout

- Dispose les widgets **horizontalement**, de gauche à droite.
- Idéal pour : boutons alignés sur une ligne, barre d'outils horizontale.

### 2. QVBoxLayout

- Dispose les widgets **verticalement**, de haut en bas.
- Idéal pour : formulaires simples, listes de boutons ou labels empilés.

### 3. QGridLayout

- Dispose les widgets dans une **grille** (lignes × colonnes).
- Permet de **positionner précisément chaque widget**, éventuellement sur plusieurs colonnes ou lignes (avec `rowSpan` / `colSpan`).
- Idéal pour : formulaires complexes, tableaux de saisie, alignement rigoureux.

### 4. QFormLayout

- Dispose les widgets en **paires label-champ** : label à gauche, widget à droite.
- Automatiquement aligné et uniforme.
- Idéal pour : formulaires classiques (nom, prénom, email...).

### 5. QStackedLayout (*moins utilisé directement, souvent via QStackedWidget*)

- Superpose plusieurs widgets **au même endroit**, un seul visible à la fois.
- Idéal pour : wizard, multi-écrans dans une même zone.

### Conseils d'utilisation

- Pour des interfaces simples → QBoxLayout et QVBoxLayout.
- Pour des formulaires bien alignés → QFormLayout ou QGridLayout.
- Pour des interfaces multi-écrans → QStackedLayout/QStackedWidget

- On peut **imbriquer les layouts** : par exemple, un QVBoxLayout principal contenant un QHBoxLayout pour une ligne de boutons.

**Exemple 64. Utilisation d'un QGridLayout**

```
• import sys
  from PyQt6.QtWidgets import QApplication, QWidget, QGridLayout,
  QLabel, QLineEdit, QPushButton

  app = QApplication(sys.argv)

  fenetre = QWidget()
  fenetre.setWindowTitle("Exemple QGridLayout")

  # Création du layout en grille
  layout = QGridLayout()

  # Ajout de labels et champs (formulaire simple)
  layout.addWidget(QLabel("Nom:"), 0, 0)          # ligne 0,
  colonne 0
  layout.addWidget(QLineEdit(), 0, 1)           # ligne 0, colonne
  1
  layout.addWidget(QLabel("Prénom:"), 1, 0)        # ligne 1, colonne
  0
  layout.addWidget(QLineEdit(), 1, 1)           # ligne 1, colonne
  1
  layout.addWidget(QLabel("Email:"), 2, 0)         # ligne 2, colonne
  0
  layout.addWidget(QLineEdit(), 2, 1)           # ligne 2, colonne
  1

  # Bouton qui occupe deux colonnes
  layout.addWidget(QPushButton("Envoyer"), 3, 0, 1, 2)  #
  rowspan=1, colSpan=2

  # Appliquer le layout à la fenêtre
  fenetre.setLayout(layout)

  fenetre.show()
  sys.exit(app.exec())
```

Voici comment interpréter l'exemple ci-dessus :

- Les labels et champs sont **alignés en deux colonnes**.
- Le bouton "Envoyer" **occupe toute la largeur** (2 colonnes) grâce aux paramètres rowspan et colSpan.
- La grille gère automatiquement les **espacements et le redimensionnement** des widgets si on change la taille de la fenêtre.

### Redimensionnement des fenêtres et rôle des layouts :

Dans Qt, **redimensionner une fenêtre** ne se limite pas à changer sa taille : il faut également que les **widgets à l'intérieur s'adaptent automatiquement**. C'est là qu'interviennent les **layouts**.

Quand une fenêtre change de taille :

- Les **widgets contenus dans un layout** sont **redimensionnés et repositionnés automatiquement** selon les règles du layout choisi (QHBoxLayout, QVBoxLayout, QGridLayout, etc.).
- Si aucun layout n'est défini, les widgets restent **fixes à leur position initiale**, ce qui peut donner une interface moche ou des composants coupés.

Qt utilise aussi des concepts complémentaires :

- **Stretch factors** : permettent de déterminer quel widget prend plus ou moins d'espace lorsqu'on agrandit la fenêtre. Ici le 1 signifie que 100% du resize va être donné au 3eme composant du layout.

| Layout               |                      |
|----------------------|----------------------|
| layoutName           | vertical.layout      |
| layoutLeftMargin     | 11                   |
| layoutTopMargin      | 11                   |
| layoutRightMargin    | 11                   |
| layoutBottomMargin   | 11                   |
| layoutSpacing        | 7                    |
| layoutStretch        | 0,0,1,0              |
| layoutSizeConstraint | SetDefaultConstraint |

- **Size policies** : définissent si un widget peut s'agrandir horizontalement ou verticalement, ou s'il doit rester fixe.
- **Margins et spacing** : gèrent l'espacement entre les widgets et par rapport aux bords de la fenêtre.

**Règle pratique** : toujours mettre vos widgets dans des **layouts imbriqués**, même pour des fenêtres simples. Cela rend l'interface **adaptive à toutes les tailles de fenêtre et résolutions d'écran**.

### Redimensionnement et layouts dans Qt Designer

#### 1. Ajouter des widgets

- Dans Qt Designer, on place les widgets sur la fenêtre principale ou un conteneur (ex. QWidget, QGroupBox, QFrame).

#### 2. Appliquer un layout

- **Sélectionner** la fenêtre ou le conteneur dans l'éditeur.
- Dans la barre d'outils, on trouve les icônes de layout :

- Lay Out Horizontally → QHBoxLayout
- Lay Out Vertically → QVBoxLayout
- Lay Out in a Grid → QGridLayout
- Cliquer sur l'icône appropriée pour envelopper les widgets sélectionnés dans le layout.

### 3. Configurer les propriétés de redimensionnement

- Chaque widget a une **sizePolicy** : Horizontal et Vertical (Fixed, Minimum, Preferred, Expanding).
- Le layout utilise ces politiques pour décider comment les widgets grandissent ou se contractent quand la fenêtre est redimensionnée.
- On peut aussi régler les **marges** et **espacements** dans le panneau des propriétés du layout.

### 4. Aperçu et ajustement

- Qt Designer propose l'option **Preview** (Ctrl+R) pour voir comment la fenêtre réagit au redimensionnement.
- Si un widget ne se redimensionne pas correctement, vérifier **sizePolicy** et que tous les widgets sont bien dans un layout (les widgets non mis dans un layout restent fixes).

## 5.9. Les boîtes de dialogue standards dans PyQt

PyQt propose plusieurs **boîtes de dialogue prêtes à l'emploi**, appelées *boîtes de dialogue standards*, qui permettent d'interagir rapidement avec l'utilisateur pour afficher des messages, choisir des fichiers, des couleurs, des polices, etc. Elles sont accessibles via des classes comme `QMessageBox`, `QFileDialog`, `QColorDialog` ou encore `QFontDialog`.

#### Afficher un message

```
from PyQt6.QtWidgets import QMessageBox

QMessageBox.information(self, "Titre", "Ceci est une boîte
d'information")
QMessageBox.warning(self, "Attention", "Ceci est un avertissement")
QMessageBox.critical(self, "Erreur", "Une erreur s'est produite")
```

Ces fonctions bloquent l'exécution jusqu'à ce que l'utilisateur ferme la boîte. Ce sont des fenêtres modales.

### Sélection de fichiers

```
from PyQt6.QtWidgets import QFileDialog

fichier, _ = QFileDialog.getOpenFileName(self, "Ouvrir un fichier", "",  
"Texte (*.txt);;Tous les fichiers (*)")
```

Cela affiche une boîte de sélection de fichiers. Le chemin sélectionné est retourné dans la variable fichier.

### Choix de couleur

```
from PyQt6.QtWidgets import QColorDialog

couleur = QColorDialog.getColor()
if couleur.isValid():
    print(couleur.name()) # Affiche la couleur en format hexadécimal
```

### Sélecteur de police

```
from PyQt6.QtWidgets import QFontDialog

police, ok = QFontDialog.getFont()
if ok:
    widget.setFont(police)
```

Ces boîtes sont très pratiques car elles offrent une expérience utilisateur cohérente avec le système d'exploitation, sans nécessiter de développement d'interface personnalisé.

## 5.10. Insérer un graphisme matplotlib dans une fenêtre QT

Il faut bien sur avoir installé préalablement matplotlib et PyQt6

Il faut ensuite importer le backend Qt de Matplotlib. Pour PyQt5 et PyQt6 on utilise le même import:

```
from matplotlib.backends.backend_qtagg import
FigureCanvasQTAgg
```

Il faut ensuite créer une figure (et des axes) Matplotlib comme on l'a fait dans le TP1.

```
# Crédation de la Figure matplotlib et des Axes
fig, ax = plt.subplots()
```

Il faut ensuite l'intégrer dans un widget Qt via FigureCanvasQTAgg ou plutôt une classe héritée de celle-ci. On peut ensuite dessiner dans notre canvas avec les axes...!

**Exemple 65. Un exemple minimal d'intégration de graphisme matplotlib**

```

import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QVBoxLayout,
QWidget
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
import matplotlib.pyplot as plt
import numpy as np

class MPLCanvas(FigureCanvasQTAgg):
    def __init__(self):
        # Création de la figure matplotlib
        self.__fig, self.__ax = plt.subplots()
        #appel du constructeur de FigureCanvas avec la fig créée en
parametre
        super().__init__(self.__fig)
        self.plot()

    def plot(self):
        #Add specific code to the canvas!
        x = np.linspace(0, 10, 100)
        y = np.sin(x)
        self.__ax.plot(x, y)
        self.__ax.set_title("Exemple Sin(x)")

class FenetreGraph(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Graphique Matplotlib dans Qt")

        # Widget central
        central = QWidget()
        self.setCentralWidget(central)
        layout = QVBoxLayout()
        central.setLayout(layout)

        # Canvas Qt pour matplotlib
        canvas = MPLCanvas()
        layout.addWidget(canvas)

    if __name__ == "__main__":
        app = QApplication(sys.argv)
        fen = FenetreGraph()
        fen.show()
        sys.exit(app.exec())

```

**Explications**

- `MPLCanvas` est une classe que l'on fait hériter de `FigureCanvas`, qui est un widget Qt qui peut contenir une figure Matplotlib.
- `layout.addWidget(canvas)` : intègre le graphique dans le layout de la fenêtre.
- On peut bien sûr combiner ça avec des layout pour mettre plusieurs graphiques ou d'autres widgets autour.

Si on veut des **interactions (zoom, pan, save)**, on peut ajouter un **NavigationToolbar** :

```
from matplotlib.backends.backend_qtagg import NavigationToolbar2QT
toolbar = NavigationToolbar2QT (canvas, self)
layout.addWidget(toolbar)
```

Cela permet de manipuler le graphique directement depuis l'interface Qt.

## 5.11. Menus et actions

Dans PyQt6, les **menus** et les **actions** servent à organiser les fonctionnalités d'une application dans la barre de menus ou dans des menus contextuels. Les menus (`QMenu`) regroupent des commandes accessibles à l'utilisateur, généralement via la barre de menus d'une `QMainWindow` ou un clic droit. Les actions (`QAction`) représentent des commandes individuelles (comme Ouvrir, Enregistrer, Quitter). Une action peut être placée dans plusieurs menus, barres d'outils ou associée à un raccourci clavier : c'est toujours la même instance qui est réutilisée. On peut ainsi centraliser son état (activée/désactivée, cochée/décochée) et sa réaction à l'événement déclenché. L'intégration se fait en créant des `QAction`, en leur attribuant un texte, une icône, éventuellement un raccourci, puis en les ajoutant à un `QMenu` ou à une `QToolBar`. Quand l'utilisateur clique sur l'action, le signal `triggered` est émis, et on le connecte à une fonction ou une méthode pour exécuter le code voulu.

### Exemple 66. Réalisation d'un menu quitter

```
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow
from PyQt6.QtGui import QAction

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Exemple menus et actions")
        self.resize(400, 300)
```

```

# Création de l'action Quitter
action_quit = QAction("Quitter", self)

action_quit.setShortcut("Ctrl+Q") # raccourci clavier
action_quit.triggered.connect(self.close) # quand déclenchée →
fermer la fenêtre

# Création du menu Fichier
menu_file = self.menuBar().addMenu("Fichier")
menu_file.addAction(action_quit) # on y ajoute l'action

if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = MainWindow()
    win.show()
    sys.exit(app.exec())

```

Les menus peuvent bien sûr être ajoutés dans QTDesigner, il suffit alors de les connecter à la QAction

## 5.12. Déclarer émettre et intercepter un signal personnalisé

En Qt6, un signal se déclare en utilisant directement les classes de `QtCore`. Pour déclarer un signal, il faut instancier la classe `Signal` dans la vue. Pour émettre un signal, on utilise la méthode `emit()`. Attention une classe qui émet un signal doit absolument hériter de `QObject`

### Exemple 67. Déclaration et d'émission d'un signal en Qt6:

```

from PyQt6.QtCore import QObject, pyqtSignal, pyqtSlot

class Envoyer(QObject):

    mon_signal = pyqtSignal(str) # Déclaration du signal avec le type
    de donnée (ici, une chaîne)

    def __init__(self):
        super().__init__()

    def methode_qui_emet(self, message):
        self.mon_signal.emit(message) # Émission du signal avec le
        message

class Recepteur:

    def fonction_de_traitement(self, message):
        print(f"Signal reçu : {str(message)}")

# Exemple d'utilisation

```

```

if __name__ == '__main__':
    envoiteur = Envoyer()
    receveur=Recepteur()

    envoiteur.mon_signal.connect(recepteur.fonction_de_traitement) # Connexion du signal à une fonction

    envoiteur.methode_qui_emet("Bonjour le monde!") # Appel de la méthode qui émet le signal

```

Explication:

1. `from PyQt6.QtCore import QObject, pyqtSignal, pyqtSlot`: On importe les classes nécessaires de `PyQt6.QtCore`. `pyqtSignal` est la classe pour déclarer un signal et `pyqtSlot` est utilisé pour les fonctions qui peuvent être connectées à des signaux, mais n'est pas requis dans cet exemple basique.
2. `class Envoyer(QObject)`: La classe `Envoyer` hérite de `QObject`.
3. `mon_signal = pyqtSignal(str)`: On déclare un signal nommé `mon_signal`. Le type de données du signal est spécifié entre parenthèses (ici, `str` pour une chaîne de caractères). Il est important de noter que les signaux sont définis comme des attributs de la classe.
4. `def methode_qui_emet(self, message)`: Cette méthode illustre comment émettre le signal.
5. `self.mon_signal.emit(message)`: On émet le signal en utilisant la méthode `emit()` et on lui passe le message comme argument.
6. `envoyer.mon_signal.connect(recepteur.fonction_de_traitement)`: On connecte le signal `mon_signal` à la fonction `fonction_de_traitement`. Lorsque le signal est émis, la fonction connectée sera appelée.
7. `envoyer.methode_qui_emet("Bonjour le monde!")`: On appelle la méthode qui émet le signal, simulant ainsi l'événement qui déclenche l'envoi du signal.

En résumé, pour déclarer un signal en Qt6 sans PySide, on utilise `pyqtSignal` dans une classe héritant de `QObject` et on émet le signal avec `emit()`. Les signaux sont ensuite connectés à des fonctions (slots) pour gérer les événements.

### 5.13. Gestion des dockedWidgets

Un **QDockWidget** est un composant de Qt conçu pour être attaché ("docké") à une fenêtre principale, généralement une  **QMainWindow**. Il sert à afficher des panneaux secondaires (outils, explorateurs de fichiers, propriétés, console, etc.)

que l'utilisateur peut réorganiser librement autour de la zone centrale. Un dock peut être placé sur les bords (haut, bas, gauche, droite) de la fenêtre principale, détaché pour flotter comme une fenêtre indépendante, ou parfois masqué. Les QDockWidget sont donc très utilisés dans les interfaces complexes (comme PyCharm par exemple) car ils offrent une grande flexibilité d'agencement, en laissant l'utilisateur personnaliser son espace de travail.

#### **Exemple 68. Utilisation d'un QDockWidget**

```
from PyQt6.QtWidgets import (
    QApplication, QMainWindow, QTextEdit, QDockWidget, QListWidget
)
import sys

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Exemple Dockable")

        # Widget central (obligatoire dans un QMainWindow)
        central = QTextEdit()
        central.setText("Zone principale")
        self.setCentralWidget(central)

        # Création d'un dock
        dock = QDockWidget("Outils", self)
        dock.setAllowedAreas(
            Qt.DockWidgetArea.LeftDockWidgetArea |
            Qt.DockWidgetArea.RightDockWidgetArea
        )

        # Contenu du dock (ici une liste)
        list_widget = QListWidget()
        list_widget.addItems(["Option 1", "Option 2", "Option 3"])
        dock.setWidget(list_widget)

        # Ajout du dock à la fenêtre
        self.addDockWidget(Qt.DockWidgetArea.RightDockWidgetArea, dock)

    if __name__ == "__main__":
        app = QApplication(sys.argv)
        window = MainWindow()
        window.resize(600, 400)
        window.show()
        sys.exit(app.exec())
```

## 5.14. Validation de champs

En Qt, un `QValidator` permet de contrôler la validité du texte entré dans un champ de saisie (`QLineEdit`) avant qu'il ne soit accepté. Cela évite à l'utilisateur de taper des valeurs incorrectes et simplifie le traitement des données dans le programme.

Pour les nombres à virgule, on peut utiliser la classe `QDoubleValidator`, qui vérifie qu'une entrée est bien un nombre flottant et peut même imposer des bornes minimales, maximales et une précision en nombre de décimales. Lorsqu'on associe un validateur à un champ texte avec `setValidator()`, Qt empêche automatiquement la saisie invalide (par exemple des lettres ou plusieurs points décimaux). On peut également connecter une méthode de validation pour mener une action spécifique (ici colorer en rouge si invalide)

```
from PyQt6.QtWidgets import QApplication, QLineEdit, QWidget,
QVBoxLayout
from PyQt6.QtGui import QDoubleValidator
from PyQt6.QtCore import Qt
import sys

class Fenetre(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Exemple QValidator avec coloration")

        layout = QVBoxLayout()

        # Champ avec validateur float
        self.champ = QLineEdit()
        self.validateur = QDoubleValidator(-100.0, 100.0, 2, self)
        self.champ.setValidator(self.validateur)
        self.champ.setPlaceholderText("Entrez un nombre flottant (-100.00 à 100.00)")

        # Détection du changement de texte
        self.champ.textChanged.connect(self.verifier_saisie)

        layout.addWidget(self.champ)
        self.setLayout(layout)

    def verifier_saisie(self, texte):
        # Vérifie si le texte est acceptable par le validateur
        etat, _, _ = self.validateur.validate(texte, 0)
        if etat == QDoubleValidator.State.Acceptable:
```

```

        self.champ.setStyleSheet("background-color: white;")
    else:
        self.champ.setStyleSheet("background-color:
lightcoral;") # rouge clair

if __name__ == "__main__":
    app = QApplication(sys.argv)
    fenetre = Fenetre()
    fenetre.show()
    sys.exit(app.exec())

```

### Types de validateurs disponibles dans Qt

- **QIntValidator:** Valide uniquement des entiers dans une plage donnée.

Par exemple, `QIntValidator(0, 100, parent)`, n'autorise que les entiers entre 0 et 100.

- **QDoubleValidator:** Valide les nombres flottants dans une plage donnée, avec un nombre de décimales défini.

Par exemple, `QDoubleValidator(-10.0, 10.0, 3, parent)`, n'autorise que les float entre -10.0 et 10.0 avec **3 décimales max.**

- **QRegularExpressionValidator:** Valide le texte en fonction d'une **expression régulière** (`QRegularExpression`).

Très flexible pour les cas personnalisés : emails, numéros de téléphone, formats spécifiques... Par exemple:

```

from PyQt6.QtGui import QRegularExpressionValidator
from PyQt6.QtCore import QRegularExpression

# Regex pour une date YYYY-MM-DD
regex1 = QRegularExpression(r"\d{4}-\d{2}-\d{2}") # format YYYY-MM-DD
validator1 = QRegularExpressionValidator(regex1)
# Regex simple pour un email basique : quelquechose@domaine.extension
regex2 = QRegularExpression(r"[\w\.-]+@[^\w\.-]+\.\w{2,3}")
validator2 = QRegularExpressionValidator(regex2)

```

On ne fera pas ici l'étude détaillée des regex en python, mais vous avez [ici](#) une excellente référence pour réaliser celle dont vous pourriez avoir besoin.

### 5.15. Gestion des événements (souris, clavier...) dans PyQt

Dans PyQt, la gestion des événements bas niveau (clics de souris, frappes clavier, etc.) se fait en **surchargeant les méthodes d'événement** d'un widget, comme

mousePressEvent(), keyPressEvent(), enterEvent(), etc. Cela permet de capturer précisément les interactions de l'utilisateur et d'y répondre de manière personnalisée.

### Événements de souris

Pour détecter un clic de souris sur un widget :

```
def mousePressEvent(self, event):
    print(f"Clic détecté à la position : {event.position().toPoint()}")
```

D'autres méthodes disponibles :

- mouseMoveEvent(event) – mouvement de souris

Il faut activer le suivi de mouvement avec self.setMouseTracking(True) si on veut détecter les mouvements **sans clic**.

### Événements clavier

Pour capter les frappes clavier :

```
def keyPressEvent(self, event):
    if event.key() == Qt.Key.Key_Escape:
        self.close() # Ferme la fenêtre si Esc est pressé
    else:
        print(f"Touche pressée : {event.text()}")
```

Autres événements disponibles :

- keyReleaseEvent(event) – relâchement d'une touche
- focusInEvent, focusOutEvent – gestion du focus clavier

### Remarques importantes sur les bonnes pratiques :

- Toujours appeler la version de la classe parente si on surcharge super().mousePressEvent(event) si on veut préserver le comportement standard.
- Utiliser QEvent et event.type() si on souhaite intercepter plusieurs types d'événements dans une méthode générique eventFilter().

### Exemple 69. Filtrage d'événements avec event() et QEvent.Type

```
import sys
from PyQt6.QtWidgets import QApplication, QLabel, QWidget, QVBoxLayout
from PyQt6.QtCore import QEvent, Qt

class MonWidget(QWidget):
```

```

def __init__(self):
    super().__init__()
    self.setWindowTitle("Filtrage d'événements avec QEvent")
    self.resize(300, 150)

    self.label = QLabel("Faites une action (clic, touche...)")
    self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

    layout = QVBoxLayout()
    layout.addWidget(self.label)
    self.setLayout(layout)

    # Active le suivi de la souris même sans clic
    self.setMouseTracking(True)

def event(self, e):
    if e.type() == QEvent.Type.MouseButtonPress:
        self.label.setText("Clic souris détecté")
    elif e.type() == QEvent.Type.KeyPress:
        self.label.setText(f"Touche pressée : {e.text()}")
    elif e.type() == QEvent.Type.MouseMove:
        pos = e.position().toPoint()
        self.label.setText(f"Souris : x={pos.x()}, y={pos.y()}")
    else:
        return super().event(e) # traitement par défaut
    return True # événement traité

app = QApplication(sys.argv)
fenetre = MonWidget()
fenetre.show()
sys.exit(app.exec())

```

#### Explications:

- On redéfinit `event(self, e)` au lieu de surcharger plusieurs méthodes séparées.
- On teste le type d'événement via `e.type()` et compare avec des constantes `QEvent.Type`.
- Cela permet de centraliser la logique pour plusieurs interactions (clavier, souris, etc.).

## 5.16. Séparation Vue et modèle : Utilisation de QComboBox et QListView

On a vu au sparagraphe 5.6.4 comment utiliser un `QComboBox`, un `QListView`, un `QTableWidget` ou un `QTreeWidget` afin d'afficher des chaines de caractères

Lorsqu'on souhaite afficher des objets on va privilégier l'usage de `QComboBox` (le même), `QListView`, `QTableView` ou un `QTreeView`

Ces classes travaillent de paire avec un modèle comme on l'a déjà vu précédemment dans les formatifs.

Supposons qu'on a un fichier bibliothèque qui contient les classes suivantes :

```
class Bibliotheque:
    @classmethod
    def bibliotheque(cls):
        auteur1=Auteur("Einstein", "Albert")
        auteur2=Auteur("Gauss", "Karl-Friedrich")
        auteur3=Auteur("Hawking", "Stephen")
        auteur4=Auteur("Berlu", "Hurlu")

        livre1=Livre("La physique pour les pros", [auteur1,auteur4])
        livre2=Livre("La physique pour les
nuls", [auteur1,auteur3,auteur4])
        livre3=Livre("Math etc..", [auteur2])
        bibli = [livre1,livre2,livre3]
        return bibli

@dataclass
class Auteur:
    __nom:str
    __prenom:str

    @property
    def nom(self):
        return self.__nom

    @nom.setter
    def nom(self, value):
        self.__nom=value

    @property
    def prenom(self):
        return self.__prenom

    @prenom.setter
    def prenom(self, value):
        self.__prenom=value

    def __str__(self):
        return f"{self.__prenom} {self.__nom}"

@dataclass
class Livre:
    __titre:str
    __auteurs:list[Auteur]

    @property
    def titre(self):
        return self.__titre

    @titre.setter
```

```

def titre(self, value):
    self.__titre=value

@property
def auteurs(self):
    return self.__auteurs

@auteurs.setter
def auteurs(self, value):
    self.__auteurs=value

def __str__(self):
    return f"{self.titre} Auteurs:{[auteur.__str__() for auteur in
self.auteurs]} "

```

Pour un `QComboBox` ou un `QListWidget`, on va créer une classe de modèle qui dérive de `QAbstractListModel` et qui va contenir la liste à afficher. On doit alors implementer certaines méthodes abstraites pour faire fonctionner le modèle:

- `rowCount(self, parent: QModelIndex = QModelIndex()) -> int`
  - But : Qt appelle cette méthode pour savoir combien d'éléments contient le modèle, retourne la longueur de la liste
- `data(self, index: QModelIndex, role: int = Qt.ItemDataRole.DisplayRole) ->`
  - Qt appelle cette méthode pour obtenir les données à afficher pour un item donné. En fonction du rôle demandé, (DisplayRole pour l'affichage, UserRole pour L'objet correspondant,...) le retour va différer.
  - Paramètres :
    - `index.row()` → l'index de l'élément.
    - `role` → type de données demandées (affichage, contenu, décoration, tooltip...)

Les autres méthodes (`flags`, `add_item`, `remove_item`, `headerData`) sont optionnelles mais très utiles dans une vraie application.

#### **Exemple 70. Utilisation de QListview et QComboBox**

```

class LivresListModel(QAbstractListModel):
    def __init__(self,data):
        super().__init__()

```

```

        self.__livres=data

    def data(self, index, role):
        if not index.isValid():
            return None
        livre = self.__livres[index.row()]
        if role == Qt.ItemDataRole.DisplayRole: # texte affiché
            return livre.__str__()
        elif role == Qt.ItemDataRole.UserRole: # objet complet
            return livre
        elif role == Qt.ItemDataRole.ToolTipRole:
            return livre.titre
        return None

    def rowCount(self, parent=QModelIndex()):
        return len(self.__livres)

class MainWindow(QWidget):
    listView:QListView
    comboBox:QComboBox
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Demonstration fonctionnement View Widgets")
        loadUi("ui/ViewWidgets.ui",self)
        # Modèle et données initiales
        self.list_model = LivresListModel(Bibliotheque.bibliotheque())

        # ComboBox utilisant le modèle
        self.comboBox.setModel(self.list_model)

        # ListView utilisant le même modèle
        self.listView.setModel(self.list_model)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.resize(400, 300)
    window.show()
    sys.exit(app.exec())

```

et le fichier xml associé :

```

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>Form</class>
  <widget class="QWidget" name="Form">

```

```
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
<width>973</width>
<height>747</height>
</rect>
</property>
<property name="windowTitle">
<string>Form</string>
</property>
<layout class="QVBoxLayout" name="verticalLayout">
<item>
<item>
<widget class="QListView" name="listView"/>
</item>
</item>
<item>
<widget class="QComboBox" name="comboBox"/>
</item>
</layout>
</widget>
<resources/>
<connections/>
</ui>
```

### Remarque :

Pour ajouter un élément au `QAbstractListModel` on ajoute la méthode `add_item` dont voici une implémentation typique :

```
def add_item(self, item):
    self.beginInsertRows(QModelIndex(), self.rowCount(),
self.rowCount())
    self._items.append(item)
    self.endInsertRows()
```

Attention, tout code qui modifie `_items` doit être entre `beginInsertRows` et `endInsertRows`.

Pour notifier le changement de vue on peut utiliser le signal [dataChanged](#)

Le fonctionnement de `QTableView` avec un `QAbstractTableModel` est sensiblement le même, bien qu'un peu plus complexe. Le fonctionnement de `QTreeView` se fait en dérivant de `QAbstractItemModel` directement. Ces deux composants peuvent être utilisés, mais ne font pas l'objet de l'évaluation dans le cadre de ce cours.

## 5.17. Appliquer un style qss

En Qt, on peut **personnaliser l'apparence des widgets avec des stylesheets CSS**, que l'on appelle ici QSS, un peu comme dans le développement web. Chaque widget possède la méthode `setStyleSheet`, qui accepte une chaîne de caractères contenant du CSS adapté à Qt. Cela permet de modifier la couleur, les bordures, les polices, les marges, ou même de gérer l'état actif, survolé ou désactivé d'un widget.

**Exemple 71.** avec un QPushButton :

```
from PyQt6.QtWidgets import QApplication, QPushButton

class Fenetre(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Exemple QValidator avec coloration")

        layout = QVBoxLayout()

        button = QPushButton("Clique-moi")
        button.setStyleSheet("""
            QPushButton {
                background-color: #3498db;
                color: white;
                border-radius: 10px;
                padding: 5px 15px;
                font-weight: bold;
            }
            QPushButton:hover {
                background-color: #2980b9;
            }
            QPushButton:pressed {
                background-color: #1c5980;
            }
        """)
        layout.addWidget(button)

        self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    fenetre = Fenetre()
    fenetre.show()
    sys.exit(app.exec())
```

On peut aussi, et c'est recommandé utiliser un fichier css qui va couvrir l'ensemble des Widget de l'Application. Il faut alors charger le fichier dans l'Application avec la méthode `setStyleSheet()`

**Exemple 72. Avec un fichier .qss**

```
/* style.qss */

/* QPushButton */
QPushButton {
    background-color: #3498db;
    color: white;
    border-radius: 8px;
    padding: 6px 20px;
    font-weight: bold;
    border: 2px solid #2980b9;
}
QPushButton:hover {
    background-color: #2980b9;
}
QPushButton:pressed {
    background-color: #1c5980;
}

/* QLabel */
QLabel {
    color: #2c3e50;
    font-size: 16px;
    font-weight: bold;
}

/* QLineEdit */
QLineEdit {
    border: 2px solid #bdc3c7;
    border-radius: 6px;
    padding: 4px 8px;
    background-color: #ecf0f1;
    color: #2c3e50;
}
QLineEdit:focus {
    border: 2px solid #3498db;
    background-color: #ffffff;
}
```

```
from PyQt6.QtWidgets import QApplication, QPushButton

class Fenetre(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Exemple QValidator avec coloration")
```

```

layout = QVBoxLayout()
# Création des widgets
label = QLabel("Étiquette stylée")
line_edit = QLineEdit()
line_edit.setPlaceholderText("Entrez du texte ici...")
button = QPushButton("Bouton")

# Ajouter les widgets
layout.addWidget(label)
layout.addWidget(line_edit)
layout.addWidget(button)

self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    # Charger le fichier CSS
    with open("style_example.qss", "r") as f:
        app.setStyleSheet(f.read())
    fenetre = Fenetre()
    fenetre.show()
    sys.exit(app.exec())

```

#### Remarques :

- La réalisation d'un fichier .qss n'est pas évaluée dans ce cours, vous pouvez chercher des modèles existants ou bon vous semble!
- **Voici quelques ressources :**
  - [https://koor.fr/Python/Tutoriel\\_PySide/pyside\\_themes\\_styles\\_css.wp](https://koor.fr/Python/Tutoriel_PySide/pyside_themes_styles_css.wp)
  - <https://doc.qt.io/qt-6/stylesheets-examples.html>

## 5.18. QSettings, données persistantes

QSettings est une classe Qt pour sauvegarder et restaurer les préférences de l'application.

Elle permet de stocker des **valeurs clé → valeur** dans un fichier INI, le registre Windows, ou d'autres backends selon la plateforme.

#### Exemple 73. Usage classique, taille et la position de la fenêtre:

```

from PyQt6.QtCore import QSettings

settings = QSettings("MonEntreprise", "MonApp")
settings.setValue("fenetre/width", 800)
settings.setValue("fenetre/height", 600)

```

```
width = settings.value("fenetre/width", 640)    # 640 = valeur par
défaut
height = settings.value("fenetre/height", 480)
```

Quand l'application redémarre, on peut **restaurer ces valeurs** et retrouver l'interface telle que l'utilisateur l'avait laissée.

## 6. Model-View-Controller (MVC) et autre MV Patterns

### 6.1. Généralités sur le MVC

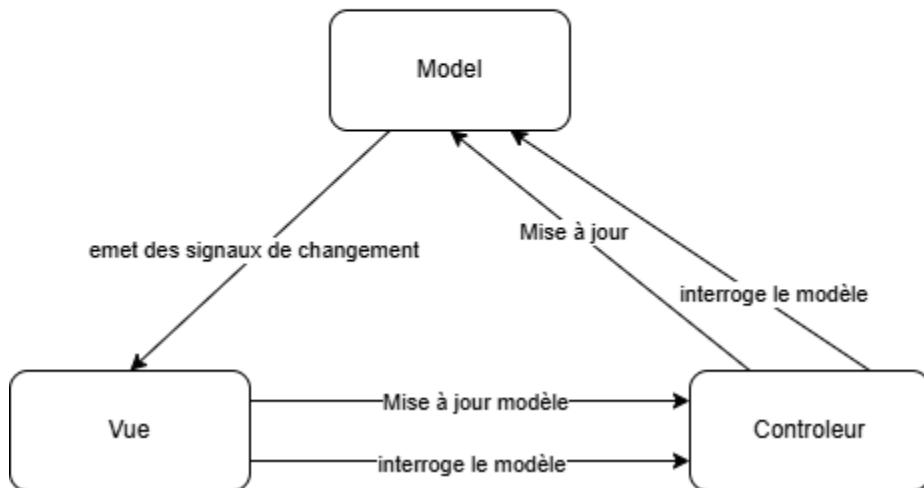
On a déjà vu de manière informelle comment séparer la vue du modèle. En développement d'application graphique on utilise fréquemment un patron de conception nommé MVC pour Model-View-Controller. Il s'agit d'une extension de ce que l'on a déjà pratiqué, qui permet de résoudre quelques petits problèmes rencontrés lors du premier TP et surtout permet de structurer le code de manière claire et maintenable.

- **Modèle (Model)** : contient les données et la logique métier. Il ne connaît ni l'interface graphique ni l'utilisateur. Dans Qt, il peut hériter de QObject pour émettre des **signaux** lorsqu'une donnée change, afin de notifier la vue. Dans l'ensemble il ne devrait importer que des éléments dans QCORE
- **Vue (View)** : s'occupe de l'affichage et de l'interaction avec l'utilisateur. Dans Qt, ce sont typiquement des widgets (QWidget, QMainWindow, QDialog, etc.) et des composants comme QListview ou QTableView. La vue se connecte aux signaux du modèle pour se mettre à jour automatiquement lorsque les données changent.
- **Contrôleur (Controller)** : agit comme intermédiaire entre la vue et le modèle. Il reçoit les événements utilisateur depuis la vue, effectue la logique nécessaire et met à jour le modèle.

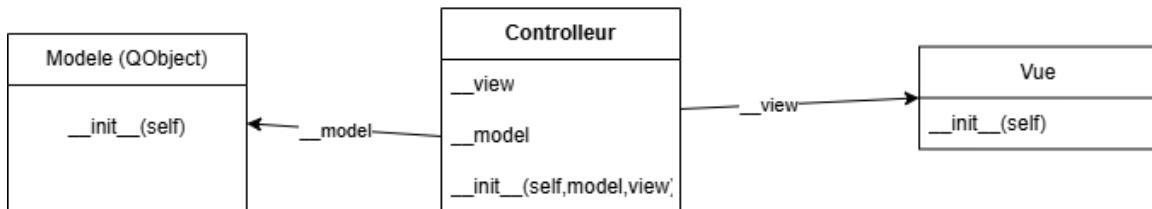
L'avantage du MVC est double :

1. **Séparation claire des responsabilités**, ce qui facilite la maintenance et les tests unitaires.
2. **Réactivité automatique** grâce aux signaux et slots de Qt : lorsqu'une donnée du modèle change, toutes les vues connectées peuvent se mettre à jour sans code redondant.

## MVC-Modèle Vue Controleur



Architecture MVC en QT



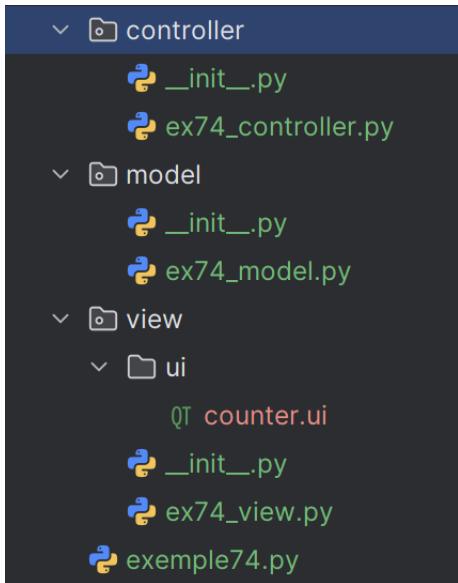
### 6.1. Un exemple simple d'utilisation du MVC

Une fenêtre avec un **compteur** et deux boutons + et -.

Le modèle stocke la valeur, la vue l'affiche, le contrôleur fait le lien.

#### Exemple 74. Exemple très simple : un compteur MVC

Voici tout d'abord la structure du projet, que vous utiliserez dans le futur pour vos applications MVC



Puis le code :

```

#fichier exemple74.py
import sys
import traceback

from PyQt6.QtWidgets import QApplication
from controller.ex74_controller import CounterController
from view.ex74_view import CounterView
from model.ex74_model import CounterModel

def qt_exception_hook(exctype, value, tb):
    traceback.print_exception(exctype, value, tb)

if __name__ == "__main__":
    sys.excepthook = qt_exception_hook
    app = QApplication(sys.argv)

    model = CounterModel()
    view = CounterView()
    controller = CounterController(model, view)

    view.show()
    sys.exit(app.exec())

```

**Remarque :** On remarque la construction dans le main des trois objets model, vue, controller avec le controller qui connaît vue et modele...

```
#fichier ex74_models.py
from PyQt6.QtCore import QObject, pyqtSignal

class CounterModel(QObject):
    valueChanged = pyqtSignal(int)

    def __init__(self):
        super().__init__()
        self._value = 0

    def increment(self):
        self._value += 1
        self.valueChanged.emit(self._value)

    def decrement(self):
        self._value -= 1
        self.valueChanged.emit(self._value)

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value = value
        self.valueChanged.emit(self._value)
```

**Remarque :** Le modèle n'importe rien d'autre de QT que QCORE et ne connaît ni controller ni vue.

```
#fichier ex74_view.py
from PyQt6.QtWidgets import QWidget, QLabel, QPushButton
from PyQt6.uic import loadUi


class CounterView(QWidget):

    valueLabel: QLabel
    minusPushButton: QPushButton
    plusPushButton: QPushButton

    def __init__(self):
        super().__init__()
        loadUi("view/ui/counter.ui", self)
```

**Remarque :** Dans cette approche, la vue reste extrêmement simple et ne connaît même pas son contrôleur. Ce n'est pas une règle absolue : on pourrait choisir de lui passer une référence vers le contrôleur, afin qu'elle puisse lui déléguer certaines actions directement (plutôt que d'émettre des signaux). On y perdrait en **découplage** entre les composants, mais on y gagnerait en **simplicité** de communication (moins de signaux et de connexions à gérer).

```
#fichier ex74_controller.py
from ..model.ex74_model import CounterModel
from ..view.ex74_view import CounterView
class CounterController:
    model:CounterModel
    view:CounterView

    def __init__(self, model, view):
        self.model = model
        self.view = view

        # Connexions
        self.view.plusPushButton.clicked.connect(self.increment)
        self.view_MINUSPushButton.clicked.connect(self.decrement)
        self.model.valueChanged.connect(self.update_label)

        # Initialisation
        self.update_label(self.model.value())

    def increment(self):
        self.model.increment()

    def decrement(self):
        self.model.decrement()

    def update_label(self, value):
        self.view.valueLabel.setText(str(value))
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>Form</class>
  <widget class="QWidget" name="Form">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>356</width>
        <height>234</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>Form</string>
```

```
</property>
<widget class="QWidget" name="">
<property name="geometry">
<rect>
<x>80</x>
<y>90</y>
<width>195</width>
<height>53</height>
</rect>
</property>
<layout class="QGridLayout" name="gridLayout">
<item row="0" column="0">
<widget class="QPushButton" name="minusPushButton">
<property name="text">
<string>-</string>
</property>
</widget>
</item>
<item row="0" column="1">
<widget class="QPushButton" name="plusPushButton">
<property name="text">
<string>+</string>
</property>
</widget>
</item>
<item row="1" column="0" colspan="2">
<widget class=" QLabel" name="valueLabel">
<property name="text">
<string/>
</property>
</widget>
</item>
</layout>
</widget>
</widget>
<resources/>
<connections/>
</ui>
```

### Bonnes pratiques:

- Les connections signaux-slots qui sont **responsables de mettre à jour les données du modèle** se déclarent dans le constructeur du controller
- Les connections signaux slots qui sont responsable **de la gestion de l'interface graphique** (état des boutons, validateurs, etc) se déclarent dans la vue.
- Au besoin on peut ajouter (souvent utile) un attribut controller dans la vue pour pouvoir interroger le controller directement dans ce cas il ne faut pas importer le controller pour éviter un import circulaire...ss

Le gros avantage de cette approche est que si on souhaite maintenant ajouter une deuxième vue par exemple sous la forme d'un slider qui représente le compteur, on ne doit pas modifier ni la vue existante. On va qu'ajouter des éléments au controller, ajouter une deuxième vue et ajuster un peu le main...

**Exercice : réaliser cette modification!**

## 6.2. Ajout d'un lien vue vers controller

Comme on l'a évoqué, il est souvent pertinent d'ajouter un lien entre la vue et son contrôleur. On ajoute pour cela un attribut « contrôleur » dans la vue et une méthode permettant de setter le contrôleur à la vue après les appels aux constructeurs. On aurait ainsi un main modifié comme suit:

```
model = MyModel()
view = MyView()
controller = MyController(model, view)
view.set_controller(controller)
```

Si on souhaite bénéficier de la complétion automatique dans le contrôleur, on tombe malheureusement sur un os... Un import circulaire :

```
ImportError: cannot import name 'TasksController' from
partially initialized module ..
```

En gros le contrôleur importe la vue qui importe le contrôleur, qui importe la vue etc.

Le mécanisme qui permet de gérer cela est nommé inversion de dépendance, et c'est un gros morceau. Python nous propose un mécanisme assez simple pour régler ce problème, le TYPE CHECKING.

```
if TYPE_CHECKING :
    from controller.controller import TasksController

class TasksView(QMainWindow):
    tasksListView:QListView
    if TYPE_CHECKING:
        __controller: TasksController | None

    def __init__(self):
        super().__init__()
        loadUi("view/ui/tasksView.ui", self)
        self.__controller = None
```

```
def set_controller(self, controller):
    self.__controller = controller
    self.controller = controller
```

A l'exécution **TYPE\_CHECKING est False** et l'import n'est pas effectué, la variable n'est pas déclarée est donc pas d'import circulaire.

Dans l'IDE, **TYPE\_CHECKING est True**, donc l'import est effectué, la variable est typée et tout fonctionne!

**Remarque** : On a ajouté dans la déclaration de la variable la possibilité de la typer à None pour ne pas avoir de warning à l'initialisation, c'est optionnel.

## 7. Arbres et graphes en python

### 7.1. Aperçu théorique sur les graphes

#### 7.1.1. Définition

Un **graphe** est une structure mathématique servant à modéliser des relations entre objets. Vous les étudierez en mathématique discrète. En voici la définition :

Formellement, un graphe G est un couple  $G = (V, E)$  tel que :

- $V$  : un ensemble de **sommets** (ou **nœuds**)
- $E$  : un ensemble d'**arêtes** (ou **liens**) reliant les sommets

Chaque arête de  $E$ ,  $e = (u, v)$  relie deux sommets ( $u$ ) et ( $v$ ).

Un graphe peut être :

- **non orienté** → la relation entre ( $u$ ) et ( $v$ ) est symétrique
- **orienté** → chaque arête a un sens (on parle d'**arc**)

### 7.1.2. Voici les différents types de graphe

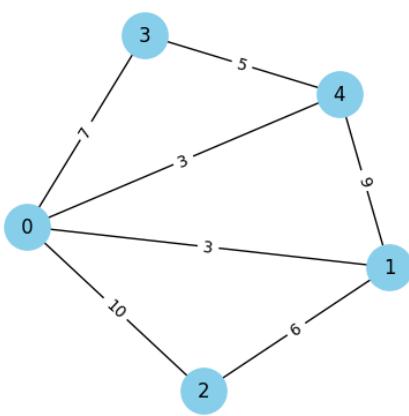
| Type de graphe            | Description                                     | Exemple                      |
|---------------------------|---|------------------------------|
| <b>Simple</b>             | Pas de boucles, ni d'arêtes multiples           | Graphe de relations amicales |
| <b>Orienté (digraphe)</b> | Chaque lien a une direction                     | Graphe de dépendances        |
| <b>Pondéré</b>            | Chaque arête a un poids (distance, coût, etc.)  | Graphe routier               |
| <b>Multigraphe</b>        | Plusieurs arêtes peuvent relier les mêmes nœuds | Réseaux de transport         |

### 7.1.3. Représentations classiques

Trois représentations principales sont utilisées

1. **Graphique** : les sommets sont des points et les arêtes sont des segments entre les points
2. **Matrice d'adjacence** : une matrice ( $n \times n$ ) où chaque cellule indique la présence (et parfois le poids) d'une arête.
3. **Listes d'adjacence** : chaque sommet a une liste de ses voisins (plus économique en mémoire pour les graphes creux).

**Exemple 75. Représentations graphique et matricielle d'un graphe pondéré :**



$G=(V,E)$  avec

Les nœuds :  $V=\{0,1,2,3,4\}$

Les arêtes munies de leur poids  $E=\{ (0,1,3), (0,2,10), (0,3,7), (0,4,3), (1,2,6), (1,4,9), (3,4,5) \}$

Matrice d'adjacence :

$$\begin{matrix}
 0 & 3 & 10 & 7 & 3 \\
 3 & 0 & 6 & 0 & 9 \\
 10 & 6 & 0 & 0 & 0 \\
 7 & 0 & 0 & 0 & 5 \\
 3 & 9 & 0 & 5 & 0
 \end{matrix}$$

### 7.1.4. Exemples d'applications

- Réseaux sociaux (amis, abonnements)
- Routage et GPS, recherche du plus court chemin
- Réseaux informatiques
- Chaînes de dépendances (compilation, tâches)
- Analyse de données (graphe de similarité)

## 7.2. NetworkX, une bibliothèque de graphe en python

### 7.2.1. Présentation générale

**NetworkX** est une bibliothèque Python conçue pour **créer, manipuler, afficher et analyser** des graphes.

Elle offre un modèle orienté objet très flexible et compatible avec les structures Python standards (dictionnaires, listes, tuples).

Pour l'installer :      pip install networkx

### 7.2.2. Les classes principales

**NetworkX** propose plusieurs types d'objets selon la nature du graphe :

| Classe         | Type de graphe                           |
|----------------|--|
| Graph()        | Graphe simple non orienté                |
| DiGraph()      | Graphe orienté                           |
| MultiGraph()   | Graphe non orienté avec arêtes multiples |
| MultiDiGraph() | Graphe orienté avec arêtes multiples     |

**Exemple 76. Construire un graphe nx et l'afficher avec matplotlib**

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_node("A")
G.add_nodes_from(["B", "C", "D"])
G.add_edge("A", "B")
G.add_edges_from([(("A", "C"), ("B", "D")), ("B", "C")])
```

```
nx.draw(G, with_labels=True)
plt.show()
```

### 7.2.3. Structure interne

Les graphes dans nx sont basés sur des **dictionnaires imbriqués** :

- G.\_adj : dictionnaire d'adjacence ({noeud: {voisin: attributs}})
- G.\_node : dictionnaire contenant les attributs des nœuds
- G.\_edge : dictionnaire contenant les attributs des arêtes (accès indirect)

Dans l'exemple précédent on aurait :

```
print("Noeuds:", G.nodes)
print("Aretes:", G.edges)
print("Adjacences:", G.adj)
```

qui afficherait :

|  |
|--|
| Noeuds: ['A', 'B', 'C', 'D']   |
| Aretes: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'C')]   |
| Adjacences: {'A': {'B': {}, 'C': {}}, 'B': {'A': {}, 'D': {}, 'C': {}}, 'C': {'A': {}, 'B': {}}, 'D': {'B': {}}} |

De plus, chaque nœud et arête peut porter des **attributs personnalisés (en particulier le weight)**:

```
G.add_node("E", label="Départ", color="green")
G.add_edge("E", "B", weight=5, capacity=10)
```

On y accède selon la syntaxe des dictionnaires :

```
print(G.nodes["E"])          # {'label': 'Départ', 'color': 'green'}
print(G["E"]["B"])           # {'weight': 5, 'capacity': 10}
print(G["E"]["B"]['weight'])  # 5
```

### 7.2.4. De nombreuses fonctions déjà implémentées!

NetworkX contient un grand nombre d'algorithmes standards :

```
a=nx.shortest_path(G, "A", "D")          # Chemin le plus court
b=nx.degree(G)                          # Degré de chaque
sommet
c=nx.connected_components(G)           # Composantes connexes
d=nx.coloring.greedy_color(G,
```

```
strategy="largest_first")           # Coloration minimale
print(a,b,c,d)
```

Ces outils permettent une **analyse algorithmique avancée** sans devoir tout réimplémenter.

### 7.2.5. L'affichage des graphes avec NetworkX

Comme on l'A vu en introduction, **Network** inclut un module de visualisation simple basé sur **Matplotlib** :

```
import matplotlib.pyplot as plt

#Construction de G

nx.draw(G, with_labels=True)
plt.show()
```

Le placement des nœuds influence la lisibilité. **NetworkX** fournit plusieurs algorithmes de positionnement. Pour résoudre ce problème, ce module calcule automatiquement une disposition des nœuds (layout) et affiche les arêtes.

#### Voici les différents layouts disponibles

| Fonction                               | Description  |
|--|--|
| <code>nx.spring_layout(G)</code>       | Disposition « à ressorts » (par défaut)            |
| <code>nx.circular_layout(G)</code>     | Disposition circulaire                             |
| <code>nx.shell_layout(G)</code>        | Disposition en cercles concentriques               |
| <code>nx.kamada_kawai_layout(G)</code> | Disposition optimisée pour la distance géométrique |

Notons que ce layout est simplement un dictionnaire {Nœud : tableau numpy de position} comme par exemple :

```
{'A': array([ 0.28286314, -0.67881607]),
 'B': array([ 0.0057961 , -0.10580847]),
 'C': array([-0.37532591, -0.7527182 ]),
 'D': array([0.03297379, 0.53734274]),
 'E': array([0.05369288, 1.      ])}
```

## Visualisation avancée

Vous pouvez personnaliser chaque aspect :

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_node("A", label="Départ", color ="blue")
G.add_nodes_from(["B", "C", "D"])
G.add_node("E", label="Arrivée", color ="blue")

G.add_edges_from([
    ("A", "B", {"weight": 3}),
    ("A", "C", {"weight": 1}),
    ("B", "D", {"weight": 5}),
    ("B", "C", {"weight": 2}),
    ("D", "E", {"weight": 8})
])
nx.draw(
    G,
    pos=nx.spring_layout(G),
    with_labels=True,
    node_size=1000,
    node_color="lightgreen",
    font_weight="bold",
    width=2,
    edge_color="gray"
)
plt.show()
```

Pour un meilleur contrôle de l'affichage on peut dessiner séparément tous les éléments du graphe. Ici on travaille avec un graphe pondéré (avec poids) et on affiche certaines arêtes en couleur :

```
G = nx.Graph()

# Ajouter des arêtes avec un poids
G.add_edge("A", "B", weight=3)
G.add_edge("A", "C", weight=1.5)
G.add_edge("B", "C", weight=2)
G.add_edge("C", "D", weight=4)

# Calcul automatique de la disposition des nœuds
pos = nx.spring_layout(G)

# Dessiner les nœuds et leurs labels
nx.draw_networkx_nodes(G, pos, node_color="lightblue", node_size=700)
nx.draw_networkx_labels(G, pos, font_size=12, font_weight="bold")

# Arêtes avec poids impair
edges_rouges = [(u, v) for u, v, d in G.edges(data=True) if d['weight'] % 2 == 1]
nx.draw_networkx_edges(G, pos, edgelist=edges_rouges, edge_color="red", width=2)
```

```
% 2 == 1]

# Dessiner les arêtes normales en noir
edges_noires = [(u, v) for u, v in G.edges() if (u, v) not in
edges_rouges and (v, u) not in edges_rouges]

nx.draw_networkx_edges(G, pos, edgelist=edges_noires, width=2,
edge_color="black")
nx.draw_networkx_edges(G, pos, edgelist=edges_rouges, width=2,
edge_color="red")

# Extraire les poids pour les labels
edge_labels = {(u, v): d['weight'] for u, v, d in G.edges(data=True)}

# Afficher les poids sur le graphe
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_color="red")

plt.axis("off")
plt.show()
```

On en aura besoin dans le TP à venir pour gérer la sélection et la mise en évidence d'un chemin dans le graphe.

### Points clés à retenir

- Un **graph** est un ensemble de nœuds reliés par des arêtes.
- **NetworkX** fournit des classes prêtes à l'emploi (Graph, DiGraph, etc.).
- Les **attributs** enrichissent la modélisation, en particulier weight sur les aretes
- Le **module de dessin** permet une analyse visuelle, il est basé sur un dictionnaire Nœuds :position appelé le layout du graphes
- Les **algorithmes intégrés** permettent une exploration rapide (chemins, composantes, centralité...).

## 8. Multithread en python

### 8.1.1. Notion de thread

Un **thread** (ou fil d'exécution) est une **unité d'exécution indépendante** à l'intérieur d'un même programme. Un processus peut contenir **plusieurs threads** s'exécutant en parallèle et partageant la même mémoire.

Cela permet, par exemple :

- D'exécuter une tâche longue sans bloquer le reste du programme ;
- D'améliorer la réactivité d'une interface graphique.

**Exemple classique :**

Pendant qu'un thread télécharge un fichier, un autre peut continuer à gérer l'affichage ou répondre aux clics de l'utilisateur.

### 8.1.2. Lancer un thread en python

Python offre le module `threading` pour créer et gérer des threads. C'est la version élémentaire, on se penchera plutôt ici sur la solution offerte par QT

**Exemple 77. Un thread tout simple**

```
import threading
import time

def tache():
    for i in range(5):
        print(f"Tâche en cours... {i}")
        time.sleep(1)

# Création du thread
t = threading.Thread(target=tache, daemon=False)

# Démarrage du thread
t.start()

print()
print("Le thread est lancé !")
print("Le programme principal sait qu'il doit attendre avant de terminer que le thread lance soit fini car daemon = False")
```

### Explications :

- target=tache : indique la fonction à exécuter dans le thread.
- t.start() : lance le thread (la fonction tache() s'exécute en parallèle).
- time.sleep(1) simule une tâche longue.
- daemon=False : oblige le programme à attendre le thread avant de terminer.

### 8.1.3. Les dangers du multithreading

Le **multithreading** permet d'exécuter plusieurs tâches en parallèle, mais il introduit des **risques de concurrence**. Lorsque plusieurs threads accèdent ou modifient **en même temps** une même ressource (variable, fichier, base de données...), on peut avoir des **incohérences de données** ou des comportements imprévisibles. On parle alors de **problèmes d'accès concurrent**.

Pour éviter cela, on utilise des **mécanismes de synchronisation** comme les verrous (Lock, RLock, Semaphore) qui garantissent qu'un seul thread à la fois peut accéder à la ressource protégée. Toutefois, une mauvaise utilisation de ces verrous peut entraîner un **deadlock** (ou interblocage) : deux threads se bloquent mutuellement, chacun attendant indéfiniment que l'autre libère une ressource.

**En résumé :** le multithreading améliore la réactivité et les performances, mais il doit être utilisé avec précaution pour éviter les **erreurs de synchronisation** et les **blocages** du programme.

On verra qu'en QT le problème a été résolu en remettant la responsabilité sur le programmeur de ne pas générer d'accès concurrents sur les composants Qt.

### 8.1.4. Utilisation d'un QThread (PyQt)

Dans une application Qt, il ne faut **jamais bloquer le thread principal**, car il gère l'interface graphique et les événements. Qt propose donc QThread pour exécuter des tâches en arrière-plan **sans geler l'interface**.

**Exemple 78. Lancement d'un thread pour réaliser une opération longue**

```

from PyQt6.QtCore import QThread, pyqtSignal
import time
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QLabel,
QPushButton

class Worker(QThread):
    progress = pyqtSignal(int) # signal émis vers la GUI

    def run(self):
        # Code exécuté dans le thread secondaire
        for i in range(5):
            #Simule un délai
            time.sleep(1)
            # informer la vue de la progression
            self.progress.emit(i + 1)
        #encore un délai avant de finir
        time.sleep(3)

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.label = QLabel("En attente...")
        self.button = QPushButton("Démarrer")
        layout = QVBoxLayout(self)
        layout.addWidget(self.label)
        layout.addWidget(self.button)
        self.button.clicked.connect(self.lancer_thread)

    def lancer_thread(self):
        #construction du thread
        self.worker = Worker()
        #écooute des signaux de progression (personnalisés)
        self.worker.progress.connect(self.mettre_a_jour)
        #écooute des signaux de fin (standard)
        self.worker.finished.connect(self.on_finished)
        #lancement thread
        self.worker.start()

    def mettre_a_jour(self, valeur):
        self.label.setText(f"Progression : {valeur}/5")

    def on_finished(self):
        self.label.setText(f"Thread terminé!")

app = QApplication([])
fen = MainWindow()
fen.show()
app.exec()

```

### Principes à retenir :

- La classe Worker hérite de QThread et redéfinit la méthode run().
- Elle **n'interagit jamais directement avec les widgets**. En effet QT exige pour éviter des erreurs d'accès concurrents que seul le Thread QT d'affichage manipule les widgets. **Le non-respect de cette règle ne génère pas forcément un crash, mais cela peut arriver à tout moment. C'est donc à éviter à tout prix!**
- La communication avec l'interface se fait via **les signaux**, pour éviter les problèmes ci-dessus!

### Limitations et bonnes pratiques avec QThread

| Correct  | A éviter  |
|--|---|
| Utiliser des <b>signaux/slots</b> pour communiquer entre threads | Modifier directement un widget depuis un thread secondaire  |
| Mettre les traitements lourds dans run()                         | Appeler sleep() dans le thread principal QT.  |
| Gérer la fin du thread (ex. finished.connect(...))               | Redéfinir le constructeur sans appeler super().__init__()   |
| Garder une référence au thread (ex. self.worker = ...)           | Laisser le thread être détruit avant la fin de l'exécution par le garbage collector (si on ne garde pas de référence) |

#### 8.1.1. Utilisation d'un QProgressBar

Une **ProgressBar** permet de visualiser l'avancement d'une tâche. Elle peut fonctionner en **mode déterminé** ou **mode indéterminé** :

1. **Mode déterminé** : la ProgressBar reflète un pourcentage précis de la tâche. On met à jour sa valeur au fur et à mesure de l'avancement (souvent de 0 à 100%). En PyQt/PySide, on utilise la méthode setValue(valeur) pour définir l'avancement courant. Exemple : lors d'un téléchargement, progressBar.setValue(50) indique que 50 % du fichier est transféré.
2. **Mode indéterminé** : on ne connaît pas l'avancement exact ou la durée de la tâche. La ProgressBar affiche un **mouvement continu** indiquant que le travail est en cours. En Qt, on active ce mode en utilisant setRange(0,0) ; la barre se met à défiler automatiquement jusqu'à ce que la tâche soit terminée. Cela est utile pour des opérations dont la durée est inconnue, comme l'attente d'une réponse réseau.

En résumé : **déterminé = pourcentage précis, indéterminé = activité en cours sans valeur exacte.**

Le formatif 8 nous permettra d'expérimenter la chose...

## 9. Simulation en python avec PyMunk :

### 9.1. Introduction

**PyMunk** est un moteur physique 2D en Python basé sur Chipmunk. Il permet de simuler des corps rigides, des collisions, des joints et des contraintes physiques avec peu de code.

**Pourquoi combiner PyMunk et Qt ?** PyMunk s'occupe de la simulation physique. Qt s'occupe du rendu graphique fluide et des interactions avec l'utilisateur. Ensemble, ils permettent de créer des simulations interactives, des jeux ou des expériences scientifiques.

**Remarque :** On voit souvent PyGame utilisé pour le rendu lors des simulations Pymunk. Pour des simulations intégrées à Qt, il est conseillé de **ne pas utiliser PyGame**. QPainter suffit pour dessiner les objets et les animations, et évite les conflits ou erreurs liés au contexte vidéo de PyGame.

### 9.2. Les bases de PyMunk

#### 9.2.1. L'espace physique (Space)

- L'espace est le conteneur principal de tous les corps et formes.
- On y définit les paramètres globaux, comme la gravité.
- L'origine (0,0) est **en bas à gauche**. L'axe X est horizontal et **l'axe Y est vertical orienté vers le bas**.

```
import pymunk
space = pymunk.Space()
space.gravity = (0, -900) # gravité vers le bas (axe Y)
```

La gravité est un vecteur (x, y). Dans cet exemple, 900 pixels/sec<sup>2</sup> vers le bas.

#### 9.2.2. Corps et formes

- Un corps (Body) représente un objet physique avec une masse, une position et une vitesse.
- Une forme (Shape) définit la forme géométrique attachée au corps (ex : cercle, segment, polygone) et ses propriétés de collision.

```

mass = 1
radius = 20
body = pymunk.Body(mass, pymunk.moment_for_circle(mass, 0,
    radius))
body.position = (300, 50) # position initiale
shape = pymunk.Circle(body, radius)
shape.elasticity = 0.9 # rebond
space.add(body, shape)

```

- `moment_for_circle` calcule l'inertie pour un cercle.
- `elasticity` définit le rebond, `friction` la friction.
- Pour lier `body` et `shape` dans l'espace `pymunk`, on fait `space.add(body, shape)`

### 9.2.3. Étape de simulation

- Pour faire avancer la physique, on appelle `space.step(dt)` à chaque frame, dans une boucle.
- `dt` est le pas de temps en secondes (souvent 1/60 pour 60 FPS).

#### Exemple : chute d'un corps avec rebonds

On va simuler une balle qui rebondit :

```

import sys
import pymunk
from PyQt6.QtWidgets import QApplication, QWidget
from PyQt6.QtGui import QPainter, QBrush
from PyQt6.QtCore import QTimer, Qt

class PyMunkBasic:
    def __init__(self):
        super().__init__()

        self.init_simulation()

    def init_simulation(self):
        # Espace
        self.space = pymunk.Space()
        self.space.gravity = (0, -900)

        # --- SOL -----
        # Segment statique de (0, 50) à (600, 50)
        ground_y = 50
        ground = pymunk.Segment(self.space.static_body,
                               (0, ground_y),
                               (600, ground_y),
                               2)
        ground.elasticity = 0.8
        ground.friction = 1.0

```

```

        self.space.add(ground)

        # --- BOULE -----
        mass = 5
        self.radius = 20

        self.body = pymunk.Body(mass, pymunk.moment_for_circle(mass, 0,
self.radius))
        self.body.position = (200, 300)
        shape = pymunk.Circle(self.body, self.radius)
        shape.elasticity = 0.8

        self.space.add(self.body, shape)

    def update_simulation(self, step):
        dt = 1 / 60
        self.space.step(dt)
        print(f"Step {step}: Ball position: {self.body.position} ")

if __name__ == "__main__":
    simul = PyMunkBasic()
    for step in range(200):
        simul.update_simulation(step)

```

Bien sûr tout cela est textuel donc uniquement à la console. On veut maintenant pouvoir faire une animation graphique!

### 9.3. Utilisation de QT pour le rendu

Dans une application QT standard on va commencer par apprendre à dessiner dans un QWidget. Cela se fait avec QPainter et la méthode paintEvent(event)

#### 9.3.1. Le système de coordonnées Qt

Attention, contrairement à Pymunk, **l'origine (0,0) en QT est en haut à gauche**. L'axe X est horizontal et **l'axe Y est vertical orienté vers le bas**. La collaboration QT-Pymunk demandera donc une conversion sur l'axe y...On y reviendra.

#### 9.3.2. QPainter : outil de dessin

QPainter sert à dessiner formes, lignes, cercles, images, texte dans un widget. Il fonctionne dans la méthode paintEvent(event) que l'on peut redéfinir. Elle est appelée automatiquement par Qt lorsqu'il faut redessiner le widget.

```

from PyQt6.QtGui import QPainter, QColor

def paintEvent(self, event):
    painter = QPainter(self)
    painter.setBrush(QColor("blue"))

```

```
painter.drawEllipse(50, 50, 100, 100) # cercle bleu
```

Pour dessiner une image de fond :

```
from PyQt6.QtGui import QPixmap
self.background = QPixmap("background.png")
painter.drawPixmap(self.rect(), self.background)
```

Attention, le fond doit être dessiné en premier pour ne pas cacher les autres dessins...

### 9.3.3. QTimer : la clé pour une animation réussies

Pour gérer la boucle de l'animation, il existe de nombreuses solutions : pygame qui gère tout pour vous, fonctionne dans une fenêtre spécifique, et est particulièrement adaptée à un jeu. On pourrait aussi imaginer lancer un thread spécifique pour l'animation, mais la multiplication des signaux que cela engendrerait risque de pas mal compliquer et alourdir le processus.

La bonne solution en QT est d'utiliser un QTimer, qui va appeler une méthode sur une base régulière et ceci dans la boucle d'événement QT. Ici on a un timer qui appelle la méthode update\_simulation toute les 16 ms ce qui assure un Frame Rate de 60 FS (Frame par secondes)

```
# Timer pour l'animation
self.timer = QTimer()
self.timer.timeout.connect(self.update_simulation)
self.timer.start(16) # ~60 FPS
```

#### Exemple 79. Une première animation sans le moteur physique pymunk

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget
from PyQt6.QtGui import QPainter, QPixmap, QBrush, QPen, QColor
from PyQt6.QtCore import QTimer, Qt

class TestQtWidget(QWidget):
    def __init__(self):
        super().__init__()

        # Taille du widget
        self.W, self.H = 600, 400
        self.setFixedSize(self.W, self.H)

        # Image de fond
        self.background = QPixmap("resources/32557.jpg") # chemin vers ton image
```

```

# Timer pour l'animation
self.timer = QTimer()
self.timer.timeout.connect(self.update_simulation)
self.timer.start(16) # ~60 FPS

#les parametres du soleil...qui pourraient etre dans un modele!
self.sun_x = 0.0
self.sun_y = self.H/4
self.sun_r = 30

def update_simulation(self):
    self.update_sun_pos()
    self.update() # déclenche paintEvent

def paintEvent(self, event):
    painter = QPainter(self)

    # Dessiner le fond
    painter.drawPixmap(self.rect(), self.background)

    # Dessiner un 2eme soleil
    # Récupérer la position du soleil en vue de l'animation
    x= int(self.sun_x)
    y = int(self.sun_y)

    painter.setBrush(QBrush(Qt.GlobalColor.yellow))
    painter.drawEllipse(x - self.sun_r, y - self.sun_r,
2*self.sun_r ,2*self.sun_r)

def update_sun_pos(self):
    if self.sun_x > self.W + 50:
        self.sun_x = - 50
    self.sun_x += 1
    if self.sun_x < self.W/2 :
        self.sun_y -= 1/10
    else :
        self.sun_y += 1/10

# --- App Qt ---
app = QApplication(sys.argv)
window = TestQtWidget()
window.show()
sys.exit(app.exec())

```

**Remarque :** La méthode `self.update()` permet de forcer un appel à `paint` sans quoi l'animation se déroule, mais l'affichage ne suit pas!

## 9.4. Intégration de pyMunk

Reprendons l'exemple de la chute libre et intégrons la représentation graphique dans un composant QT

```

import sys
import pymunk
from PyQt6.QtWidgets import QApplication, QWidget
from PyQt6.QtGui import QPainter, QBrush
from PyQt6.QtCore import QTimer, Qt

class PyMunkBasicWidget(QWidget):
    def __init__(self):
        super().__init__()

        self.W, self.H = 600, 400
        self.setFixedSize(self.W, self.H)

        self.timer = QTimer()
        self.timer.timeout.connect(self.update_simulation)
        self.timer.start(16)

        self.init_simulation()

    def init_simulation(self):
        # Espace
        self.space = pymunk.Space()
        self.space.gravity = (0, -900)

        # --- SOL -----
        # Segment statique de (0, 50) à (600, 50)
        ground_y = 50
        ground = pymunk.Segment(self.space.static_body,
                               (0, ground_y),
                               (self.W, ground_y),
                               2)
        ground.elasticity = 0.8
        ground.friction = 1.0
        self.space.add(ground)

        # --- BOULE -----
        mass = 5
        self.radius = 20

        self.body = pymunk.Body(mass, pymunk.moment_for_circle(mass, 0,
                                                               self.radius))
        self.body.position = (200, 300)
        shape = pymunk.Circle(self.body, self.radius)
        shape.elasticity = 0.8

        self.space.add(self.body, shape)

```

```

def update_simulation(self):
    dt = 1 / 60
    self.space.step(dt)
    self.update()

def paintEvent(self, event):
    p = QPainter(self)

    # --- Dessine le sol ---
    p.setBrush(Qt.GlobalColor.gray)
    p.drawRect(0,
               self.H - 50,           # Qt = inversé
               self.W,
               50)

    # --- Dessine la balle ---
    x = int(self.body.position.x - self.radius)
    y = int(self.H - self.body.position.y - self.radius)  #
conversion PyMunk -> Qt

    p.setBrush(Qt.GlobalColor.red)
    p.drawEllipse(x, y, 2 * self.radius, 2 * self.radius)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = PyMunkBasicWidget()
    window.show()
    sys.exit(app.exec())

```

#### 9.4.1. Ajout d'une impulsion initiale

On peut ajouter une impulsion initiale avec l'instruction suivante

```
self.body.apply_impulse_at_local_point((300,0),(0,0))
```

Remarque Il existe deux méthode `apply_impulse` et `apply_force`, dont l'effet est assez différent :

- `apply_impulse_at_local_point(impulse, pos)` : applique une impulsion instantanée, simule un coup qui modifie immédiatement la vitesse du corps.
- `apply_force_at_local_point(force, pos)` : applique une force continue pendant un pas de temps `dt`. La vitesse change progressivement, selon la physique  $F = m * a$ . C'est idéal pour simuler un vent, moteur ou propulsion.

### 9.4.2. Détection de collisions

PyMunk permet d'identifier les collisions avec des “collision\_type”.

On va commencer par typer les collisions dans init\_simulation() :

```
shape.collision_type = 1 # balle  
ground.collision_type = 2 # sol
```

Puis on va ajouter un callback sur une méthode appelée lors de la collision des types concernés :

```
self.space.on_collision(1, 2, begin=self.on_ball_hit_ground)
```

Enfin on définit la méthode

```
def on_ball_hit_ground(self, arbiter, space, data):  
    print("Collision avec le sol !")
```

Notons que plusieurs callbacks autre que begin peuvent être définis :

Deux shapes se touchent

- begin : appelé une seule fois au début
- pre\_solve : appelé chaque frame tant qu'ils se touchent
- post\_solve : après que la physique a appliqué les forces
- separate : quand ils se séparent

### 9.4.3. Résumé des principaux composants de PyMunk

#### 1. Space

- C'est l'**espace de simulation**, le conteneur de tous les objets physiques.
- Contient :
  - bodies : tous les corps (Body)
  - shapes : toutes les formes (Circle, Segment, Poly)
  - constraints : toutes les contraintes (DampedSpring, PinJoint, etc.)
- Paramètres :

- gravity : gravité globale (x, y)
- damping : amortissement global de vitesse
- Méthodes importantes :
  - step(dt) : avance la simulation
  - on\_collision(type\_a, type\_b, ...) : gérer les collisions

## 2. Body

- Représente un **objet physique dynamique, statique ou cinématique**.
- Types :
  - Body.DYNAMIC : affecté par forces et gravité
  - Body.STATIC : immobile, utilisé pour sol/murs
  - Body.KINEMATIC : déplaçable manuellement, ignore la gravité
- Paramètres :
  - mass : masse du corps
  - moment : moment d'inertie
  - position, velocity, angular\_velocity, angle
- Méthodes :
  - apply\_force\_at\_local\_point(force, point)
  - apply\_impulse\_at\_local\_point(impulse, point)

## 3. Shapes

- Définissent la **forme physique attachée à un Body**.
- Types principaux :
  - Circle(body, radius, offset=(0,0))
  - Segment(body, a, b, radius)
  - Poly(body, vertices, transform=None)
- Propriétés :

- elasticity : rebond
- friction : friction au contact
- collision\_type : pour gérer collisions
- Chaque shape doit être ajoutée à un Space.

#### **4. Constraints (Contraintes / Joints)**

- Connectent deux corps ensemble. Types principaux :
  - PinJoint : distance fixe entre deux points
  - SlideJoint : distance min/max entre deux points
  - PivotJoint : articulation fixe en un point
  - DampedSpring : ressort avec raideur et amortissement
  - DampedRotarySpring : ressort qui agit sur la rotation
  - GearJoint : corps liés avec un ratio de rotation
  - RotaryLimitJoint : limite l'angle relatif entre deux corps

#### **5. CollisionHandler / Callbacks**

- Déclenchés quand deux shapes se touchent :
  - begin : au début du contact
  - pre\_solve : avant la résolution physique
  - post\_solve : après la résolution physique
  - separate : quand ils se séparent

#### **6. Vecteurs**

- Vec2d : représente un vecteur 2D (x, y)
- Utilisé pour positions, forces, vitesses
- Supporte les opérations mathématiques : +, -, \*, /, normalisation, rotation

#### **7. Utilities**

- `moment_for_circle(mass, inner_radius, outer_radius)` : calcule le moment d'inertie pour un cercle
- `moment_for_poly(mass, verts)` : pour polygones
- `Vec2d` méthodes comme `get_distance()`, `rotated()`

## 10. Annexes

### 10.1.1. Gestion de branches avec Git

Ce court guide présente les notions essentielles pour créer, gérer et fusionner des branches Git directement depuis PyCharm. Il s'adresse à un utilisateur ayant une connaissance de base de Git en ligne de commande mais souhaitant travailler plus efficacement via l'interface.

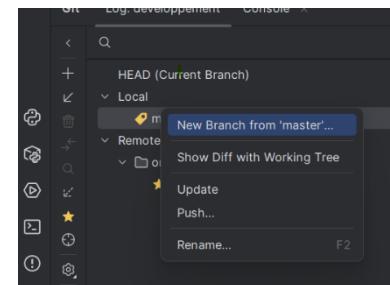
#### 1. Pourquoi utiliser des branches ?

Les branches Git permettent de développer des fonctionnalités, corriger des bogues ou tester des idées **sans impacter la branche principale** (généralement main ou master). Dans PyCharm, l'usage est simplifié grâce à une interface graphique centralisée.

#### 2. Créer une nouvelle branche

Créer une branche depuis PyCharm :

1. Cliquer sur le **nom de la branche** en bas à droite.
2. Sélectionner **New Branch....**
3. Donner un nom explicite (ex. *feature/login-ui*).
4. Cocher **Checkout branch** pour basculer directement dessus.



PyCharm crée ensuite la branche localement à partir de l'état courant.

On peut créer une branche à partir d'une autre branche. Il suffit de se placer sur la branche souhaitée avant de créer la nouvelle branche.

#### 3. Changer de branche (Checkout)

Pour changer de branche :

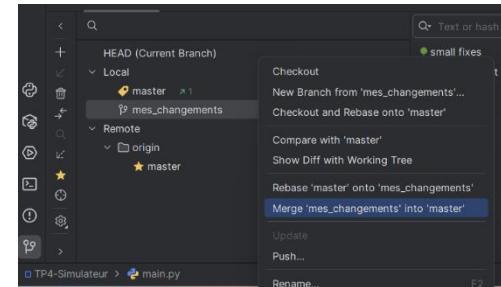
1. Cliquer sur le nom de la branche en cours.
2. Sélectionner la branche souhaitée dans la liste.

PyCharm gère automatiquement les changements de fichiers et avertit en cas de modifications non commit.

#### 4. Fusionner une branche (Merge)

Fusionner une branche A dans une branche B revient à :

1. Se placer (checkout) sur **la branche qui va recevoir les changements** (ex. main).
2. Cliquer droit sur la branche à fusionner.
3. Choisir **Merge into Current....**
4. Sélectionner la branche source (ex. feature/login-ui).



PyCharm applique le merge et affiche une alerte si un conflit est détecté.

#### Résolution de conflits

En cas de conflits :

- PyCharm ouvre un **merge tool** visuel avec 3 panneaux :
  - *Left* (source), *Right* (destination), *Base* (version commune).
- On choisit pour chaque bloc la version à conserver.
- Une fois terminé, cliquer **Apply** puis faire un **commit**.

#### 5. Pousser / synchroniser les branches (Push, Pull, Fetch)

PyCharm simplifie les opérations réseau :

- **Push** : envoie les commits de la branche locale vers le serveur.
- **Pull** : récupère et fusionne les changements du serveur.
- **Fetch** : récupère les nouvelles branches / commits sans fusionner automatiquement.

Dans PyCharm, ces boutons se trouvent dans la barre supérieure ou via Ctrl+Shift+K (push) / Ctrl+T (update project).

## 6. Publier une nouvelle branche sur le serveur

Après création d'une branche locale :

1. Cliquer **Push**.
2. PyCharm vous proposera de **Publish Branch**.

## 7. Supprimer une branche

Une fois une fonctionnalité fusionnée :

1. Cliquer sur le nom de la branche active.
2. Choisir **Delete...** dans la liste.
3. Confirmer la suppression locale.

PyCharm propose ensuite de supprimer la branche sur le serveur (remote) si elle existe.

## 8. Bonnes pratiques de gestion des branches

Dans un projet de relative grande envergure on suggère de travailler sur le modèle nommé gitflow, avec :

- Une branche master sur laquelle on va faire les release du logiciel pour le public
- Une branche de développement sur laquelle on va pousser les nouvelles fonctionnalités quand elles sont dans un état stable
- Des branches personnelles sur lequel le développement se fait

C'est un sujet trop vaste pour être abordé ici, mais voici un [article](#) qui résume bien la question.