



# Programmation en langage Python

2023/2024

Ettaheri Nizar
nizar.ettaheri@ofppt.ma

# A propos du langage Python

**Python** est un langage de programmation de haut niveau interprété pour la programmation à usage général. Créé par Guido van Rossum, et publié pour la première fois en 1991.

Python repose sur une philosophie de conception qui met l'accent sur la lisibilité du code, notamment en utilisant des espaces significatifs. Il fournit des constructions permettant une programmation claire à petite et grande échelle.

python™

# Quelles sont les principales raisons qui poussent à apprendre Python ?

1. Utilisé par des sites web
2. Richesse en outils
3. Python est orienté objet
4. Simplicité et lisibilité du code
5. Python est open source donc gratuit
6. Python est multiplateforme
7. Python est très puissant en terme de production

https://www.youtube.com/watch?v=YFfWOZkJ8i0&list=PLZpzLuUp9qXwerbFOLoUpLQFBs2sT3SYW

# Les variables

Une **variable** est une zone de la mémoire de l'ordinateur dans laquelle une valeur est stockée.

Aux yeux du programmeur, cette variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire.

Une variable est créée au moment où vous lui affectez une valeur.

Exemple. de variables en python

```
x = 7

y = ”Albert ”

print (x)

print (y)
```

# Les variables

Une variable python possède toujours un **type**, même s'il est non déclarée.

le type se définie au moment de l'introduction de la variable et peut être changé par la suite, ce qui justifie le dynamisme et la puissance du langage Python

Les **4 principaux** types sont les entiers (**integer** ou **int**), les nombres décimaux que nous appellerons **floats**, les chaînes de caractères (**string** ou **str**) et les booléens (**True**, **False**).

```
x = 3

# x est de type int

x = " Hello "

# x est maintenant transformé en type string
```

# Affichage d’une Variable

L’instruction **print** Python (on verra qu’il s’agit d’une fonction) est souvent utilisée pour générer la sortie des variables.

```
x = 5

print (x) # affiche 5
```

# Lecture d'une Variable

L'instruction **input** est souvent utilisée pour lire les variables.

```
x = input()
```

```
x = input("Entrez la valeur de X")
```

Si un chiffre/nombre on doit faire un **cast**

```
x = int(input("Entrez la valeur de X")
```

pour forcer le x d'être un chiffre au lieu chaîne de caractère.

```
x = input() # donne "5"
```

```
x = int(input()) # donne 5
```

# Les opérateurs en Python

Les opérateurs sont utilisés en Python pour effectuer des opérations sur les variables et

les valeurs associées. Python classifie les opérateurs selon les groupes suivants :

1. **Opérateurs arithmétiques**
2. **Opérateurs d’assignation**
3. **Opérateurs de comparaison**
4. **Opérateurs logiques**

# Les opérateurs arithmétiques

| Opérateur | Description |
|---|---|
| '+ ' | addition |
| '-' | soustraction |
| '*' | muliplication |
| '/' | division |
| '%' | modulo ( reste de la division euclidienne) |
| '**' | Exponentiation |
| '//' | quotient de la division euclidienne |

# Opérateurs de comparaison

| Opérateur | Description |
|---|---|
| = = | opérateur d'égalité |
| ! = | opérateur différent |
| > | opérateur supérieur |
| < | opérateur inférieur |
| > = | opérateur supérieur ou égale |
| < = | opérateur inférieur ou égale |

# Opérateurs logiques

| Opérateur | Description |
|---|---|
| and | **et** logique |
| **or** | **ou** logique |
| **not** | **Négation** logique |

# Les opérateurs d'assignation

| Opérateur | Exemple | Explication |
|---|---|---|
| = | x = 7 | x prends la valeur 7 |
| + = | x + = 5 | x = x + 5 |
| – = | x – = 5 | x = x -5 |
| * = | x * = 5 | x = x *5 |
| / = | x / = 5 | x = x / 5 |
| % = | x % = 5 | x = x %5 (reste de la division euclidienne de x par 5) |
| // = | x // = 5 | x = x //5 (quotient de la division euclidienne de x par 5) |
| ** = | x ** = 3 | x = x **3 ( x^3 ie x*x*x ) |
| & = | x & = 5 | x = x &5 (& désigne l'opérateur binaire) |

# Les conditions

La structure sélective **if** ...**else**, permet d'exécuter un ensemble d'instructions lorsqu'une condition est réalisée.

**Syntaxe :**

```
if ( condition ) :
        instructions . . .
else :
        autres instructions . . .
```

# structure if ... else...

**Exemple :**

```
age = 19
if (age >= 18) :
        print ( ”Vous êtes majeur !” )
else :
        print ( ”Vous êtes mineur !” )
        # affiche vous êtes majeur
```

# L’instruction elif

L’instruction **elif** est employée généralement lorsque l’exception comporte 2 ou plusieurs cas à distinguer. Dans notre exemple ci-dessus l’exception est age < 18 qui correspond au cas mineur.

Or le cas mineur comporte les deux cas :

1. Enfance **age** < 14

2. Adolescence 14 < **age** < 18

# L’instruction elif

L’instruction else sélectionne la condition contraire qui est age < 18 et donc ne peut distinguer entre les deux cas enfance et adolescence. Ainsi pour palier à ce problème, on utilise l’instruction elif :

```
age = int (input( ’tapez votre age : ’ ) )

 if (age >= 18) :

        print ( ”Vous êtes majeur !” )

elif (age<15) :

        print ( ”Vous êtes trop petit !” )

 else :

        print ( ”Vous êtes adolescent !” )
```

# La structure répétitive For

La boucle **for**, permet d'exécuter des instructions répétés. Sa syntaxe est :

**for** compteur **in range**(début_compteur , fin_compteur )

    instructions . . .

**Remarque**

1. Noter que dans la boucle for i in range(1,n) le dernier qui est **n** n'est pas inclus !

Cela veut dire que la boucle s'arrête à l'ordre **n-1**.

# La structure répétitive While

La structure **while** permet d'exécuter un ensemble d'instructions tant qu'une condition est réalisée et que l'exécution s'arrête lorsque la condition n'est plus satisfaite. Sa syntaxe est :

```
while ( condition ) :

        instructions . . .
```

# Les chaînes de caractères en Python

Comme tous les autres langage, **les chaines de caractères** en python sont entourés de guillemets simples ou de guillemets doubles. "CMC Tamesna" est identique à 'CMC Tamesna'.

# Les chaînes de caractères en Python

Les chaînes peuvent être affichées à l'écran en utilisant la fonction d'impression **print**().

Comme beaucoup d'autres langages de programmation populaires, les chaînes en Python sont des tableaux d'octets représentant des caractères Unicode.

Cependant, Python **ne possède pas** de type de données caractère (**char**) comme char type en C, un seul caractère est simplement une chaîne de longueur 1. Les crochets peuvent être utilisés pour accéder aux

éléments de la chaîne.

```
s = ”CMC Tamesna”

print ( ”Le deuxième caractère de s est : ” , s [1])

# affiche : ”Le deuxième caractère de s est M
```

# Les fonctions en Python

Le langage Python possède déjà des fonctions prédéfinies comme **print**() pour afficher du texte ou une variable, input() pour lire une saisie clavier... Mais il offre à l'utilisateur la possibilité de créer ses propres fonctions :

**Déclaration :**

```
def maFonction(x) :
        return 2*x
```

**Utilisation :**

```
print ( "Le double de 5 est : " , maFonction(5)
```

# Les collections

Une **liste** est une collection qui est commandée et modifiable.

• En Python, les liste sont écrites entre crochets.

```
liste = [“A”,”B”,”C”]
```

Modification de la valeur d’un élément du tableau :

```
liste [1] = “E”
```

Affichage d’une liste :

```
for i in liste :
    print(i)
```

# Les collections

Une **liste** est une collection qui est commandée et modifiable.

- En Python, les liste sont écrites entre **crochets**.

```
liste = [“A”,”B”,”C”]
```

# IDE : PyCharm

**Video :**

https://www.youtube.com/watch?v=fpJjlnuCSI8&list=PLZpzLuUp9qXwerbFOLoUpLQFBs2sT3SYW&index=2

**Quizz :**

https://quizizz.com/embed/quiz/670f7be0306387886df4e233

# Programmation Orientée objet

2023/2024

Ettaheri Nizar
nizar.ettaheri@ofppt.ma

# POO

La **programmation orienté objet** est un type de programmation basée sur la création
des classes et des objets via une méthode appelée instanciation.

Une **classe** est un prototype (**modèle ou plan**) codé en un langage de programmation dont le but de créer des objets dotés d'un ensemble de méthodes et attributs qui caractérisent n'importe quel objet de la classe.

# POO

Les **attributs** sont des types de données (**variables de classe** et **variables d'instance**) et des **méthodes**, accessibles via la concaténation par points.

En programmation orientée objet, la **déclaration d'une classe** regroupe des méthodes et propriétés (attributs) communs à un ensemble d'objets.
Ainsi on pourrait dire qu'une **classe** représente une catégorie d'objets.

Elle apparaît aussi comme **une usine** permettant de créer des objets ayant un ensemble d'**attributs** et **méthodes** communes.

Depuis sa création, **Python** est un langage de programmation orienté objet. Pour cette raison, la création et l'utilisation de classes et d'objets en Python est une opération assez simple.

# Objet


Un **objet** est une entité qui possède des **caractéristiques** et qu'on va pouvoir **manipuler**.

On peut faire un parallèle immédiat avec les objets du monde réel.

Prenons, par exemple, l'objet **téléphone**, qui possède une certaine : marque, un modèle, un numéro de série, une couleur, un niveau de charge de la batterie, un niveau de luminosité de l'écran, etc.

Ces différentes caractéristiques sont appelées **attributs** de l'objet.

Pour un objet donné, chaque **attribut** possède une **valeur**, dont l'ensemble définit l'état de l'objet.

La figure 1 montre deux objets de type **téléphone**, avec leurs cinq **attributs** et **valeurs** associées.

# Objet

L'**objet** est au cœur de la programmation orientée objet, par opposition à la programmation procédurale où un programme se construit sur base de **fonctions**.

Ici, on commence par **créer un objet,** pour ensuite l'utiliser par le biais de ses méthodes.

# Objet et classe

Pour pouvoir créer des **objets** dans un programme, il faut avant tout avoir défini une **classe**. Il s'agit en quelque sorte d'un **plan**, qui décrit les caractéristiques de l'objet.

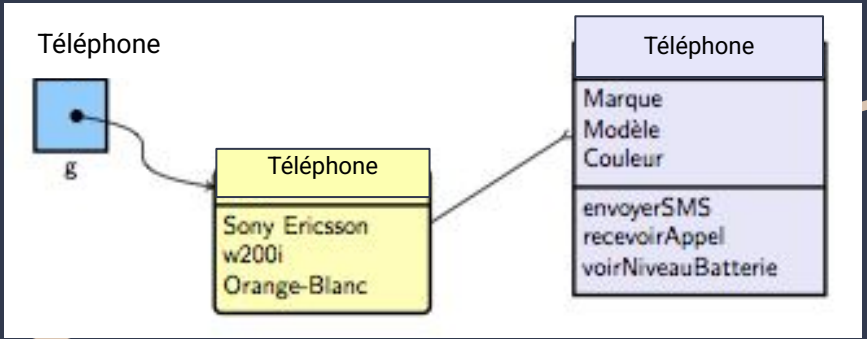
En particulier, une **classe** définit les deux éléments constitutifs des objets que sont ses **attributs** et **méthodes**.

**Un objet est donc construit à partir d'une classe** et on dit qu'il en est une **instance**.

**La classe est le modèle à partir duquel il est possible de créer autant d'objets que l'on souhaite.**

La figure  illustre ce concept en montrant quatre instances qu'il est possible d'obtenir à partir d'une classe Téléphone (représentée au centre).

# Attribut et fonctionnalité




Pour rappel, un **objet** possède des **attributs** et offre des **fonctionnalités**.

Un **attribut** est une donnée stockée dans l'objet et qui permet de le caractériser.

L'ensemble des valeurs de tous les attributs d'un objet donné définit son **état**.

Une **fonctionnalité** permet d'effectuer une opération grâce à l'objet.

On peut, par exemple, l'interroger pour obtenir une information le concernant, ou alors effectuer une action sur cet objet.

# Classe

On définit une nouvelle classe à l'aide du mot réservé **class** suivi d'un nom, d'un **deux-points** (:) et on termine avec le corps de la classe indenté d'un niveau.

L'exemple suivant définit une nouvelle classe appelée `Telephone` qui permet de représenter un élément.

```
class Telephone:
```

On peut dorénavant créer des **instances** à partir de cette classe.

L'exemple suivant crée deux objets à partir de la classe :

```
a = Telephone()

b = Telephone()
```

# Définition de constructeur

le **constructeur** est appelé lors de la création d'un nouvel objet.

Il s'agit en fait d'une **méthode** particulière dont le code est exécuté lorsqu'une classe est **instanciée**.

Le **constructeur** se définit dans une classe comme une fonction avec **deux** particularités :

- le nom de la fonction doit être **__init__** ;
- la fonction doit accepter **au moins un paramètre**, dont le nom doit être **self**, et qui doit être le premier paramètre.

# Définition de constructeur

Le paramètre **self** représente en fait l'objet cible, c'est-à-dire que c'est une variable qui contient une référence vers l'objet qui est en cours de création.

Grâce à ce dernier, on va pouvoir accéder aux attributs et fonctionnalités de l'objet cible.

```
class Telephone:
    def __init__(self, marque, couleur, prix):
        print(marque)
        print(couleur)
        print(prix)


tel = Telephone(‘Samsung’, 'Noir', 6000)
```

# Définition de constructeur

Le constructeur de la **classe Telephone** reçoit donc trois paramètres, en plus du paramètre spécial **self**, et se contente de les afficher.

**tel** = **Telephone**('Samsung', 'Noir', 6000)

Qu'en est-il de la variable **tel** ? Cette dernière contient tout simplement une référence vers l'objet qui a été créé, c'est-à-dire une indication sur son emplacement en mémoire.

```
print(tel)
<__main__.Telephone object at
0x10f805c88>
```

# Définition de destructeur

• Le **destructeur** est une méthode qui permet la destruction d'un objet non référencé.

• Un **destructeur** permet de:

- Gérer les erreurs
- Libérer les ressources utilisées de manière certaine
- Assurer la fermeture de certaines parties du code.

**Remarque** :
Python utilise des **ramasse-miettes** qui n'offrent pas le mécanisme des destructeurs puisque le programmeur ne gère pas la mémoire lui-même.

• Un ramasse-miettes est programme de gestion de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.

# Définition de destructeur

La méthode **__del __ ()** est une méthode appelée destructeur en Python.

• Le destructeur est appelé lorsque toutes les références à l'objet ont été supprimées, c'est-à-dire lorsqu'un objet est nettoyé.

```
class Personne :
# Les paramètres d'âge et de sexe ont une valeur par défaut.
def __init__ (self, nom, age = 1, sexe = "Male" ):
self. nom = nom
self.age = age
self.sexe= sexe
def __del__(self):
print("je suis le destructeur")

if __name__=='__main__':
aimee =Personne("Aimee",21,"Female")
del aimee # affiche je suis le destructeur
```

# Membre de classe vs Membre d’instance

Un **membre** est un attribut ou une méthode.
Il existe 2 types de membres (de classe/d’instance)

# Membre de classe

• **Membre de classe**

• Un **Attribut** de classe est associé à sa classe et **non** à une instance de cette classe.

• **Méthode** de classe est associée à une classe et **non** à un objet.

**Remarque:**

• Une méthode de classe ne peut exploiter que des membres de classe

• Une méthode d'instance peut utiliser n'importe quel type de membre (classe ou instance)

# Attribut de classe

Un **attribut** de **classe** est une variable qui est associée à la classe elle-même, plutôt qu'à une instance spécifique de cette classe.

```
class MaClasse:
    # Attribut de classe
    attribut_de_classe = "Ceci est un attribut de classe"

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

# Créez deux instances de la classe MaClasse
objet1 = MaClasse("Objet 1")
objet2 = MaClasse("Objet 2")

print(objet1.attribut_de_classe)
# Affiche "Ceci est un attribut de classe"
print(objet2.attribut_de_classe)
#ou
print(MaClasse.attribut_de_classe)
```

# Méthode de classe

Une **méthode de classe** est une méthode associée à la classe elle-même plutôt qu'à une instance particulière de cette classe.

# Membre d'instance

- **Membre d'instance**

Un **attribut** d'instance est associée à une instance de la classe.

Chaque **objet** possède donc sa propre copie de la propriété

Une **méthode** d'instance est associée à une instance d'une classe.

Chaque **objet** possède donc sa propre copie de la méthode.

# Attribut d'instance

La classe **Telephone**, telle qu'elle est pour le moment, n'est pas encore des plus utiles. En effet, il serait souhaitable de pouvoir **stocker les informations** du telephone dans l'objet qui a été créé.

```
class Telephone:
    def __init__(self, marque, couleur, prix):
        self.marque = marque
        self.couleur = couleur
        self.prix = prix
```

Il est important de faire la différence entre les deux types de variables qui se trouvent dans le code de ce constructeur :

- la variable **self.marque** représente l'Attribut d'instance, c'est-à-dire celle associée à l'objet, qui existe à partir de la création de l'objet jusque sa destruction ;
- la variable **marque** représente le paramètre reçu par le constructeur et n'existe que dans le corps de ce dernier.

# Attribut d'instance

Le paramètre **self** permet donc d'accéder aux **Attributs d'instance**, c'est-à-dire aux attributs de l'objet cible, depuis le constructeur.

Revenons maintenant à la création d'un objet de type **Telephone**.

Puisqu'on a initialisé des variables d'instance dans le constructeur, on va pouvoir y accéder pour n'importe quel objet créé.

Si on reprend **tel**, on va pouvoir afficher la valeur des trois variables d'instance à l'aide de l'opérateur d'accès (.) :

```
tel = Telephone('Samsung', 'Noir', 6000)

print(tel.marque)

print(tel.couleur)

print(tel.prix)
```

# Attribut d'instance

**Les Attributs d'instance** sont spécifiques à chaque objet et ce sont d'ailleurs elles qui définissent l'état de l'objet.

```
tel = Telephone('iPhone', 'Rouge', 9000)

tel_1 = Telephone('Samsung', 'Vert', 3000)
```

# Méthode d’instance

Une **méthode** se définit dans une classe comme une fonction, avec comme particularité qu'elle doit accepter **au moins un paramètre**, dont le nom doit être **self**, et qui doit être le premier paramètre.

Ce **paramètre** représente l'objet cible sur lequel la méthode est appelée. Il permet notamment d'avoir accès aux variables d'instance de l'objet.

```
class Telephone:

    def __init__(self, marque, couleur, prix=0):

        self.marque = marque

        self.couleur = couleur

        self.prix = prix

    def setPrix(self, prix):

        self.prix = prix
```

La méthode **setPrix** reçoit simplement un **prix** en paramètre et écrase la valeur de la variable d'instance **prix** avec cette nouvelle valeur reçue.

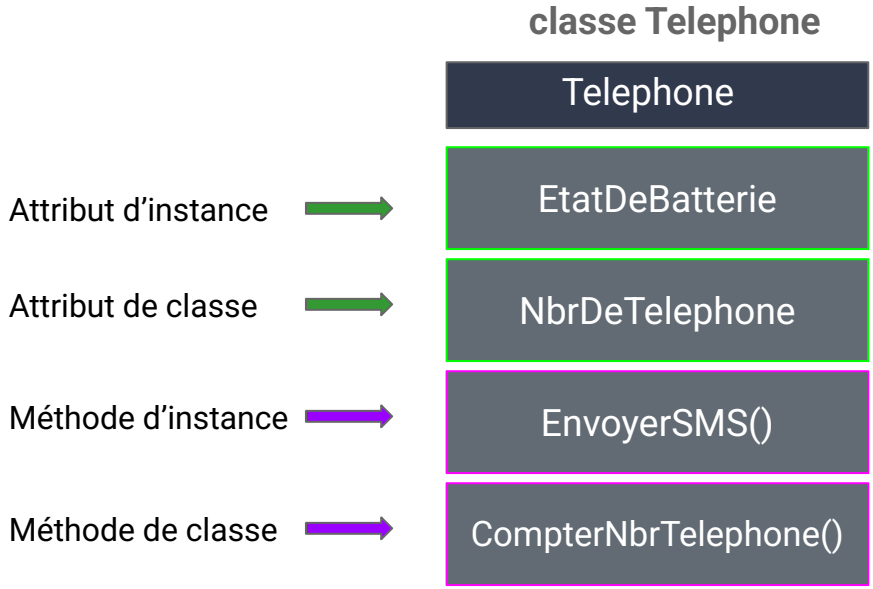
# Appel de méthode

Pour appeler une **méthode**, il faut indiquer l'objet cible suivi de la méthode à appeler en utilisant l'opérateur d'appel (.).

L'exemple suivant crée un objet de la classe Telephone, sans préciser le prix, puis modifie ce dernier en appelant la méthode **setPrix** :

```
tel = Telephone('Sony', 'Vert')

print(tel.prix)

tel.setPrix(8293)

print(tel.prix)

#Affichage

0

8293
```

# Membre de classe vs membre d'instance



# une classe Python sans constructeur

Le **chaînage** des constructeurs, souvent appelé "constructeur par défaut", peut être réalisé en appelant un autre constructeur à l'intérieur du constructeur actuel.

Si vous souhaitez créer une classe Python **sans constructeur**, c'est tout à fait possible.

Une classe sans constructeur utiliserait le constructeur **par défaut fourni par Python**, qui ne prendrait aucun argument.

Voici un exemple simple d'une telle classe :

```
class MaClasseSansConstructeur:

    def affiche(self):

        print("Ceci est une classe sans constructeur.")

# Créer une instance de MaClasseSansConstructeur

objet_sans_constructeur = MaClasseSansConstructeur()

# Appeler la méthode affiche()

objet_sans_constructeur.affiche()
```

# TP

**Exercice :**

1- Implémenter la classe **Telephone** avec les attributs :

- marque, couleur, prix.

et les méthodes :

- setMarque, setCouleur et setPrix.

2- Implémenter la classe **Personne** avec les attributs :

- first_name, last_name, age.

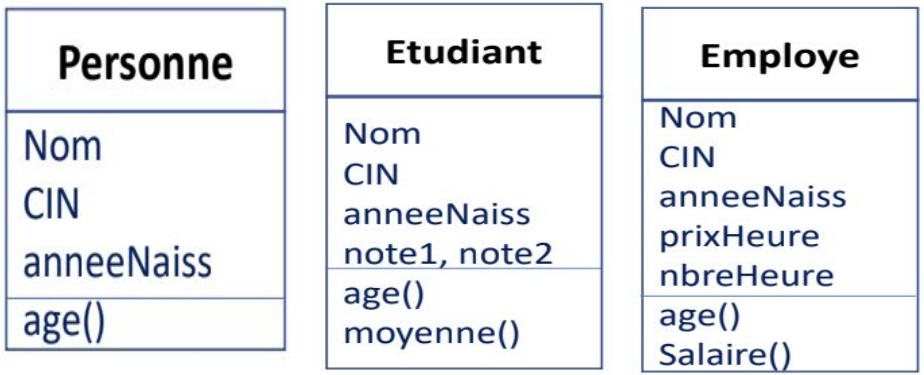
et les méthodes :

- setFirstName, setLastname et setAge.

# Principe et intérêt de l’héritage

# Principe et intérêt de l'héritage

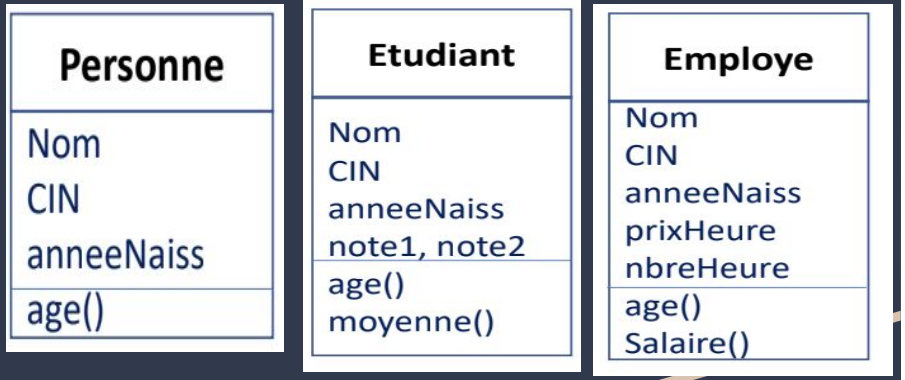
Considérons la définition des 3 classes **Personne**, **Etudiant** et **Employé** suivantes :



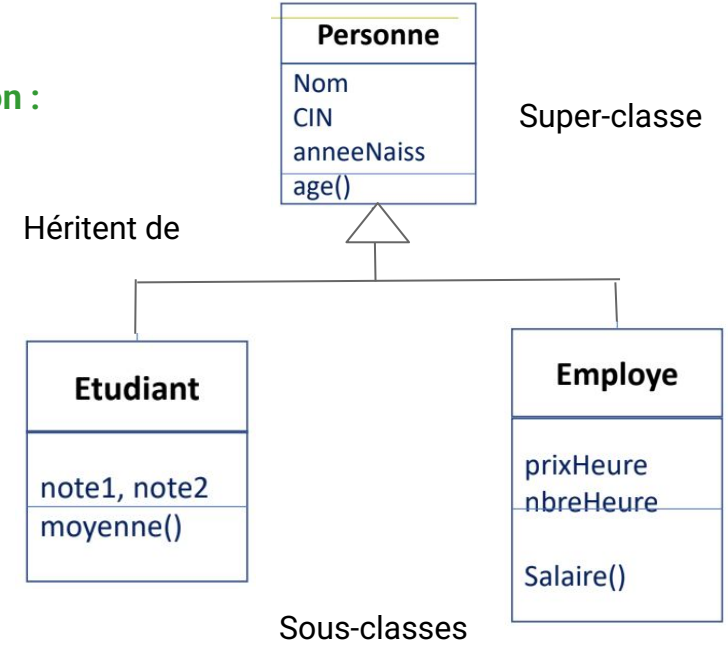

**Problème :**

- Duplication du code
- Une modification faite sur un **attribut** ou **méthode** doit être refaite sur les autres classes

# Principe et intérêt de l'héritage




**Solution :**

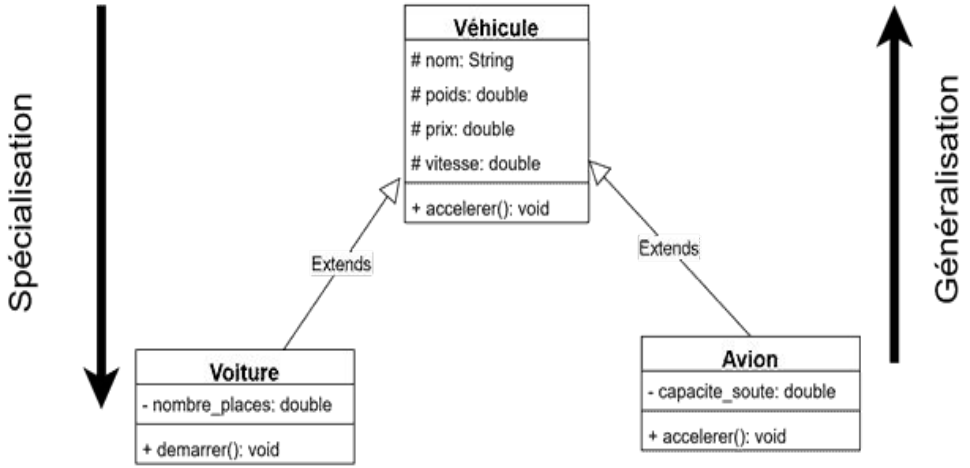



- Placer dans la classe mère (super-classe) **toutes les informations communes** à toutes les classes.
- Les classes filles ne comportent que les attributs ou méthodes plus spécifiques.

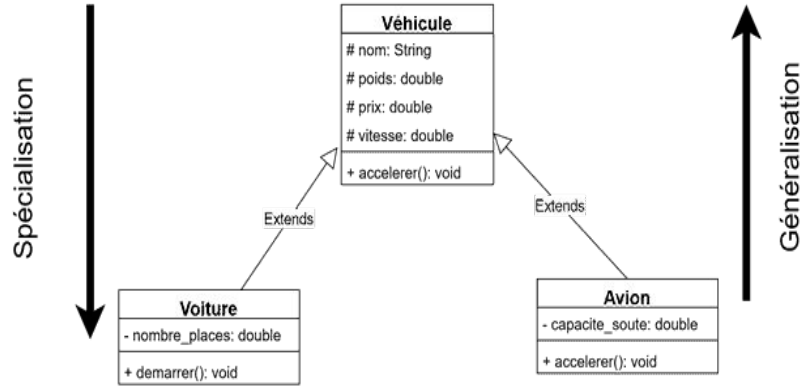
# Principe et intérêt de l’héritage

**L'héritage :**

Le concept de l'héritage spécifie une relation de **spécialisation/généralisation** entre les classes



# Principe et intérêt de l'héritage

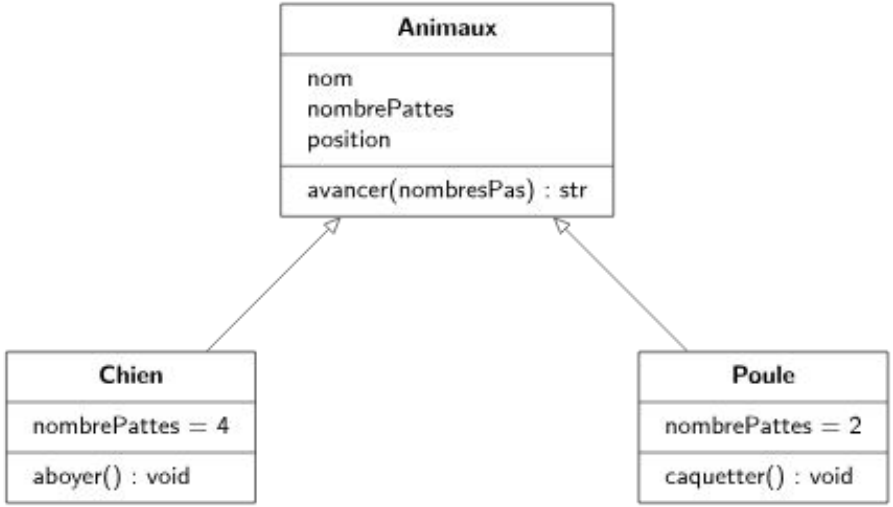



- Lorsqu'une classe **Voiture** hérite d'une classe **Véhicule** :
- **Voiture** possède toutes les caractéristiques de **Véhicule** et aussi, d'autres caractéristiques qui sont spécifiques à **Voiture**
- **Voiture** est une spécialisation de **Véhicule** (un cas particulier)
- **Véhicule** est une généralisation de **Voiture** (cas général)
- **Voiture** est appelée classe dérivée (fille)
- **Véhicule** est appelée classe de base (mère ou super-classe)

# Principe et intérêt de l'héritage

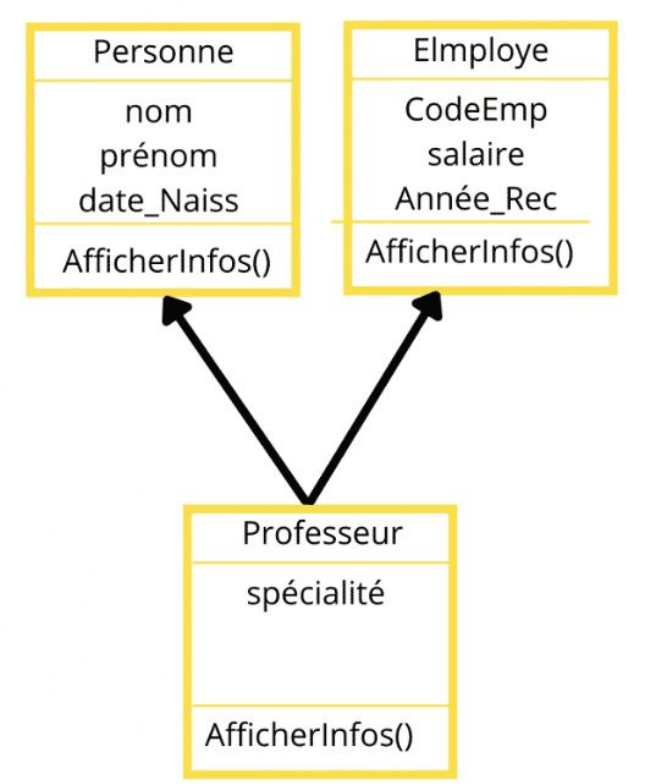
**L'héritage :**

Un autre exemple de l'héritage :




Tout objet instancié de **Poule ou Chien** est considéré, aussi, comme un objet de type Animaux

# Types de l'héritage




**Héritage multiple :**

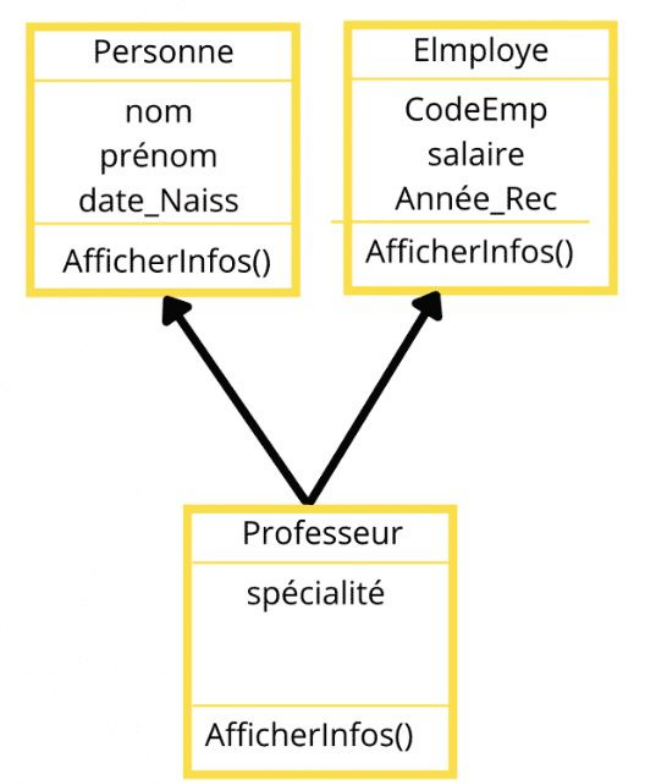
• Une classe peut hériter de **plusieurs** classes

Exemple :

Un **Professeur** est à la fois une **Personne** et un **Employe**

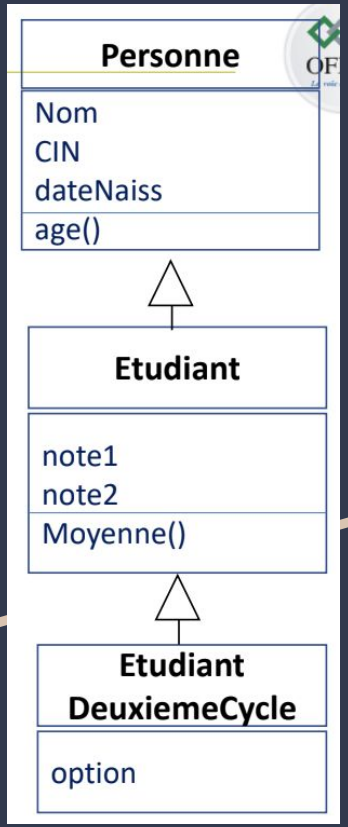
• La classe **Professeur** hérite les **attributs** et les **méthodes** des **deux** classes → Il deviennent ses propres attributs et méthodes.

# Types de l’héritage




```
class ClasseA:
    def __init__(self)
        print("Constructeur de ClasseA")
    def methode_a(self):
        print("Méthode de la ClasseA")
class ClasseB:
    def __init__(self):
        print("Constructeur de ClasseB")
    def methode_b(self):
        print("Méthode de la ClasseB")
class ClasseC(ClasseA, ClasseB):
    def __init__(self):
        super().__init__()  # Appelle le constructeur de ClasseA
        super(ClasseA, self).__init__()  # Une autre façon d'appeler le constructeur de ClasseA
        super().methode_b()  # Appelle la méthode_b de ClasseB
    def methode_c(self):
        print("Méthode de la ClasseC")
```

# Types de l'héritage




**Héritage en cascade :**

• Une classe sous-classe peut être elle-même une super-classe.

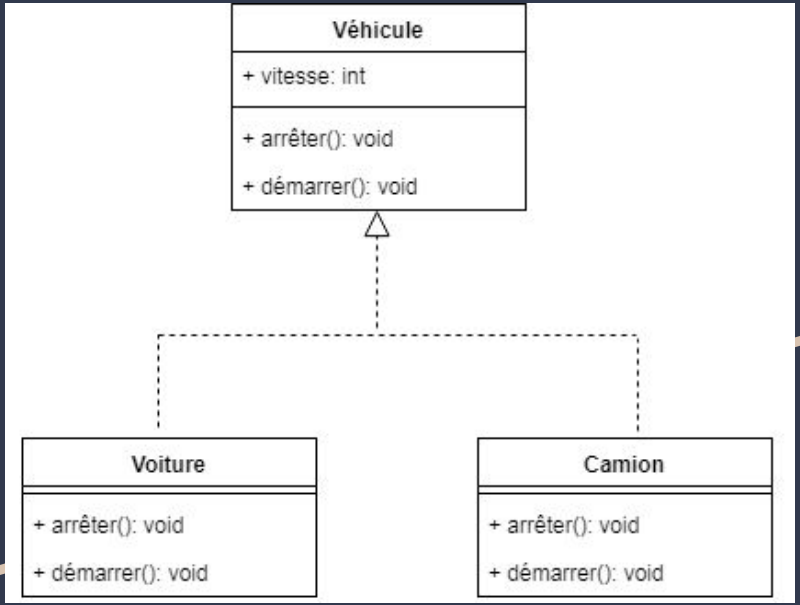
Exemple:

**Etudiant** hérite de **Personne**

**EtudiantDeuxièmeCycle** hérite de **Etudiant**

→ **EtudiantDeuxièmeCycle** hérite de **Personne.**

# Redéfinition des méthodes



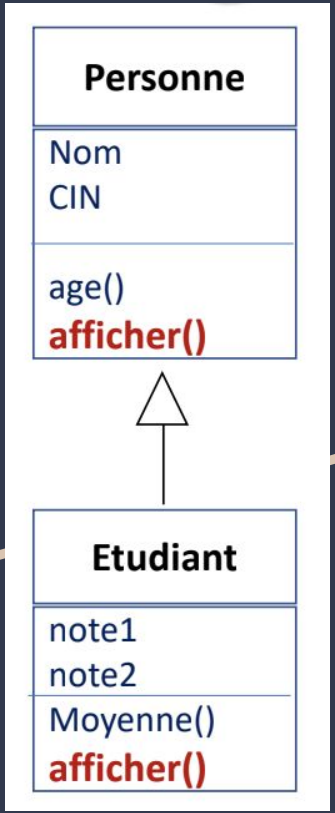

• Une méthode **héritée** peut être redéfinie si sa version initiale n'est pas satisfaisante pour la classe dérivée

• La **redéfinition** consiste à **conserver** l'entête de la méthode et à proposer un code différent

• **Si une méthode héritée est redéfinie**, c'est uniquement la nouvelle version qui fait parti de la description de la classe dérivée

• Si la méthode définie au niveau de la classe dérivée est de **type différent**, ou de **paramètres différents**, alors il s'agit d'une nouvelle méthode qui s'ajoute à celle héritée de la classe de base

# Redéfinition des méthodes




**Exemple:**

• Soit la classe **Etudiant** qui hérite de la classe **Personne**

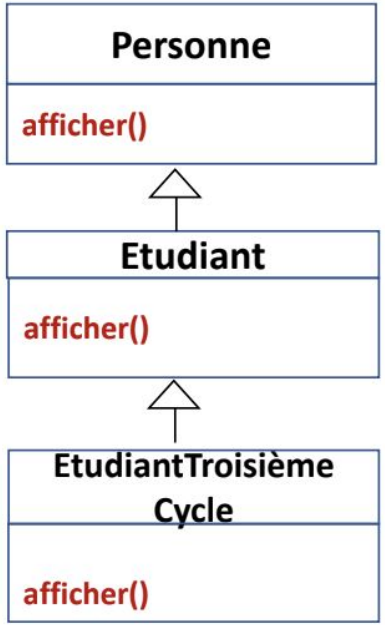
• La méthode **afficher()** de la classe **Personne** affiche les attribut Nom, CIN d'une personne

• La classe **Etudiant** hérite la méthode afficher() de la classe Personne et **la redéfinit** , elle propose un nouveau code (la classe Etudiant ajoute l'affichage des attributs **notes1** et **note2** de l'étudiant)

# Mécanisme de la liaison retardée

**Exemple:** Soit **etc** un objet de type **EtudiantTroisièmeCycle**

**1er cas**

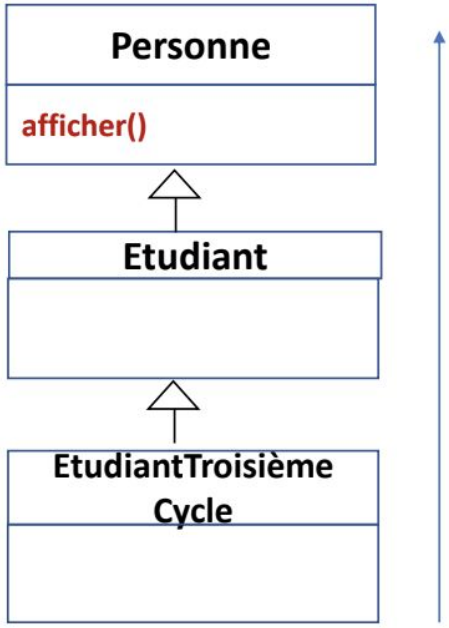



Afficher() de la classe EtudiantTroisièmeCycle est exécutée

**etc.afficher()** →

**2ème cas**

**etc.afficher()** →




Afficher() de la classe Personne est exécutée

# TP 3

# Principe du polymorphisme

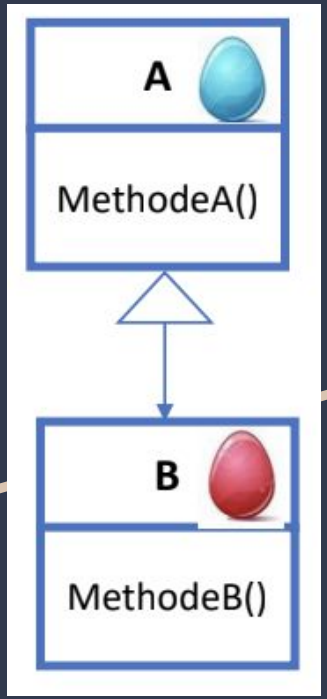
• Le **polymorphisme** permet de modifier le comportement d’une classe fille par rapport à sa classe mère.

il utilise l’héritage comme un mécanisme d’extension en adaptant le comportement des objets.

**Polymorphisme** nous permet de définir des méthodes dans la classe enfant portant le même nom que les méthodes de la classe mère.

Le **Polymorphisme** est un concept de la programmation orientée objet (POO) qui se réfère à la capacité d'un objet à prendre plusieurs formes.

# Principe du polymorphisme




le type de la variable est utilisé par le compilateur pour déterminer si on accède à un membre (attribut ou méthode) **valide**

Variable :

- **Type A** => **a1**
- **Type B** => **b1**
- **Type A** => **a2**

**b1**.MethodeA() → OK car b1 est de type déclaré B qui hérite de A

**b1**.MethodeB() → OK car b1 est de type déclaré B

**a2**.MethodeA() → OK car a2 est de type déclaré A

**a2**.MethodeB()→ ERREUR car a2 est de type A (même si le type l'objet référencé est B)

# Principe du polymorphisme

En Python, plusieurs classes peuvent avoir des méthodes avec **le même nom sans déclencher d'erreur**. Lorsque vous appelez une méthode, Python utilise la méthode de la classe de l'objet en cours d'utilisation.

```
class Chat(Animal):
    def faire_bruit(self):
        return "Miaou"
class Chien(Animal):
    def faire_bruit(self):
        return "Haw"
animaux = [Chat(), Chien()]
for animal in animaux:
    print(animal.faire_bruit())
# Utilise le polymorphisme pour appeler la méthode appropriée
```

# Méthode qui retourne rien

Le mot-clé **pass** en Python est une instruction null. Lorsqu'un interpréteur Python atterrit sur cette instruction, il l'analyse, mais rien ne se passe.

Généralement, les développeurs l'utilisent comme espace réservé pour le code qu'ils prévoient d'écrire dans un avenir proche.

# L’héritage en code

```
# Classe mère
class Personne():
    # Constructeur
    def __init__(self, nom, cin):
        self.nom = nom
        self.cin = cin
    def afficher(self):
        print("Nom : ",self.nom)
        print("CIN : ",self.cin)

class Employe( Personne ):
    def __init__(self, nom, cin, salaire):
        self.salaire = salaire
        # appeler __init__ de la classe mère (Personne)
        super().__init__( nom, cin)
```

# Principe de l’encapsulation

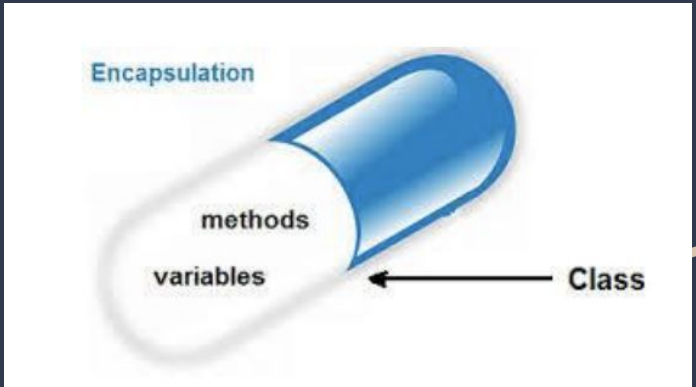
# Principe de l’encapsulation

• **L’encapsulation** est l’un des **quatre** concepts fondamentaux de la programmation orientée objet, qui sont l’abstraction, l’encapsulation, l’héritage et le polymorphisme.

**L’encapsulation** est le regroupement de données et de méthodes dans une classe afin que vous puissiez masquer les informations et restreindre l’accès de l’extérieur.

Ce concept permet à une classe de **protéger** son état interne de l’extérieur.

# Principe de l’encapsulation

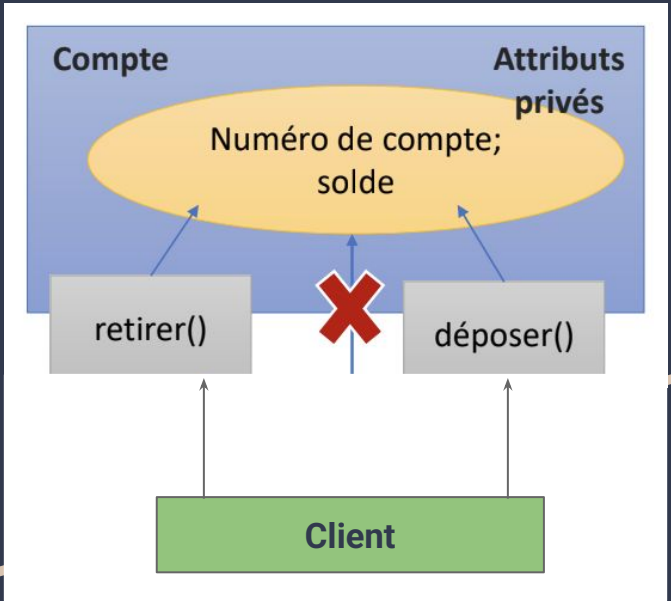



• **L’encapsulation** est le fait de réunir à l'intérieur d'une même entité (**objet**) le code (**méthodes**) + données (**attributs**).

• **L’encapsulation** consiste à protéger l’information contenue dans un objet.

• Il est donc possible de masquer les informations d'un objet aux autres objets.

# Intérêt de l'encapsulation




Les objets restreignent leur accès qu'aux méthodes de leur classe.

→ Ils protègent leurs attributs

• Cela permet d'avoir un contrôle sur tous les accès.

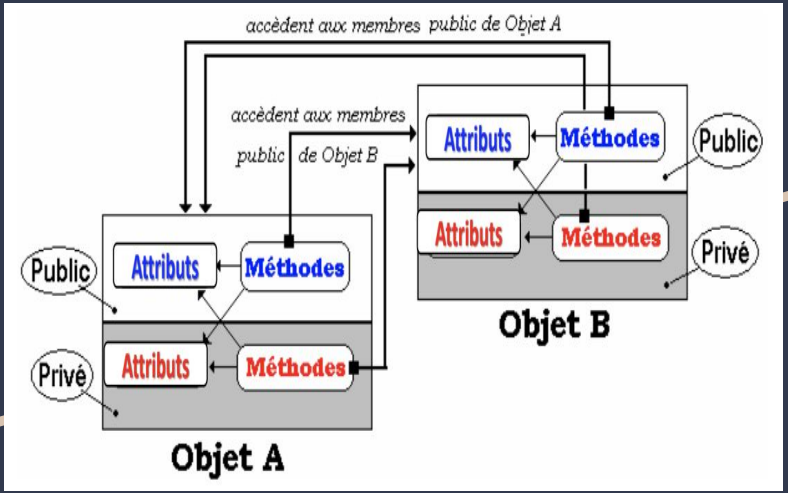
**Exemple:**

Les attributs de la classe compte sont **privés**

Ainsi le solde d'un compte n'est pas accessible par un client qu'à travers les méthodes **retirer()** et **déposer()** qui sont publiques

→ Un client ne peut modifier son solde qu'en effectuant une opération de **dépôt** ou de **retrait**

# Niveaux de visibilité des membres




Il existe **3** niveaux de visibilité des membres d'une classe :

• **Publique** : les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité publique.
→ Il s'agit du plus bas niveau de protection des données

• **Privé** : l'accès aux données est limité aux méthodes de la classe elle-même
→ Il s'agit du niveau de protection des données le plus élevé

• **Protégé (ou Protected)** : Les membres déclarés comme protégés sont généralement accessibles uniquement à l'intérieur de la **classe** et de **ses sous-classes**.

En Python, la convention est de préfixer le nom du membre avec un seul trait de soulignement (_).

# Les niveaux intermédiaires d'encapsulation

• **Classes Amies**

une classe pourrait décider de se rendre entièrement accessible à quelques autres classes, privilégiées, qu'elle déclarera comme faisant partie de ses « **amies** »

• **Une classe dans une autre**

Une classe rend accessible ses attributs et ses méthodes déclarés **privés** à une autre classe lorsque cette seconde classe est créée à l'intérieur de la première.
La classe **englobée** aura un accès privilégié à tout ce qui constitue la classe **englobante**.

# Les niveaux intermédiaires d’encapsulation

- **Mécanisme d’héritage**

Consiste à ne permettre qu’aux seuls enfants de la classe (ses héritiers) un accès aux attributs et méthodes **protected** du parent.

- **Utilisation des paquetages**

Consiste à libérer l’accès, uniquement aux classes faisant partie d’un même paquetage (un même répertoire)

# Variables Privés

Pour implémenter une variable privée en Python, nous utiliserons deux trait de soulignement (__) avant le nom de la variable.

```
class Person:
    def __init__(self, name, age, ID):
        self.name = name
        self.age = age
        self.__ID = ID
```

Pour accéder à **ID**, vous devez créer une méthode :

```
    def getID(self):
        return self.__ID
```

Pour Insérer/Modifier **ID**, vous devez créer une méthode :

```
    def setID(self, ID):
        self.__ID = ID
```

# Les getters/Setters

les **getters** et les **setters** en POO permettent de contrôler l'accès aux attributs d'une classe, de garantir l'encapsulation des données et d'appliquer des règles métier si nécessaire lors de la lecture et de la modification des attributs.

# Les getters/Setters

**Getter** (Accesseur) :

Un **getter** est une méthode qui permet de récupérer la valeur d'un attribut privé d'une classe. Il est généralement nommé avec un préfixe "**get**" suivi du nom de l'attribut (par exemple, **get_nom()** pour récupérer la valeur de l'attribut "nom").
Les getters sont utilisés pour accéder aux valeurs des attributs sans accéder directement à ces derniers.

```
def getID(self):
    return self.__ID
```

# Les getters/Setters

**Setter** (Mutateur) :

Un **setter** est une méthode qui permet de modifier la valeur d'un attribut privé d'une classe.
Il est généralement nommé avec un préfixe "**set**" suivi du nom de l'attribut (par exemple, **set_nom()** pour modifier la valeur de l'attribut "nom").
Les setters sont utilisés pour contrôler les modifications d'attributs et appliquer des règles métier si nécessaire.

```
def setID(self, ID):
    self.__ID = ID
```

# Méthodes privées

En Python, il n'y a pas de véritable concept de "**méthode privée**" comme on peut en trouver dans certains autres langages de programmation.
Cepend d'une **convention** de **nommage** pour indiquer que certaines méthodes ou attributs ne devraient pas être utilisés en dehors de la classe où ils sont définis.
Cette convention consiste à préfixer le nom de la méthode ou de l'attribut par un double souligné "__"

# Méthodes privées

```
class MaClasse:
    def __init__(self):
        self.__attribut_prive = 42

    def __methode_privee(self):
        print("Ceci est une méthode privée.")

    def methode_publique(self):
        print("Ceci est une méthode publique.")
        self.__methode_privee()  # Vous pouvez appeler une méthode privée à l'intérieur de la classe.


# Exemple d'utilisation :
objet = MaClasse()
objet.methode_publique()  # Appel de la méthode publique
#objet.__methode_privee()  # Cela générerait une erreur si vous essayez d'appeler la méthode privée en dehors de la classe.
```

# Variables inutilisées

Un seul trait de soulignement est un nom de variable temporaire qui ne sera pas réutilisée.

```
for _ in range(3):
    print('Python')
```

```
Python
Python
Python
```

# Surcharge des méthodes

En Python, la **surcharge** (method overloading) et la **redéfinition** (method overriding) sont deux concepts différents relatifs à la façon dont les méthodes d'une classe sont gérées.

Python ne prend pas en charge la surcharge de méthode basée uniquement sur les signatures de paramètres (c'est-à-dire les types ou le nombre de paramètres).
**Seule** la dernière définition de méthode dans la classe est conservée.

# Surcharge des méthodes

```
class Exemple:
    def methode_surchargee(self, param1):
        print(f"Méthode avec un paramètre : {param1}")

    def methode_surchargee(self, param1, param2):
        print(f"Méthode avec deux paramètres : {param1}, {param2}")

obj = Exemple()
# Appel de la méthode avec deux paramètres (la deuxième définition prévaut)
obj.methode_surchargee(1, 2)
```

# Surcharge des opérateurs

La **surcharge** d'opérateurs en programmation orientée objet consiste à donner une signification spécifique à un opérateur lorsqu'il est utilisé avec des objets de votre classe.

En Python, **la surcharge des opérateurs** est réalisée en implémentant certaines méthodes spéciales dans votre classe. Ces méthodes spéciales ont un double soulignement avant et après leur nom, par exemple __add__ pour la surcharge de l'opérateur d'addition (+).

# Surcharge des opérateurs

Voici quelques-unes des méthodes spéciales les plus couramment utilisées pour surcharger les opérateurs :

**__init__(self, ...):** Le constructeur, appelé lors de la création d'un nouvel objet.

**__str__(self):** Méthode appelée par la fonction str() pour convertir l'objet en une chaîne de caractères.

**__repr__(self)**: Méthode appelée par la fonction repr() pour obtenir une représentation officielle de l'objet.

**__add__(self, other)**: Surcharge de l'opérateur d'addition (+).

**__sub__(self, other):** Surcharge de l'opérateur de soustraction (-).

**__mul__(self, other):** Surcharge de l'opérateur de multiplication (*).

**__eq__(self, other)**: Surcharge de l'opérateur d'égalité (==).

**__lt__(self, other)**: Surcharge de l'opérateur de comparaison inférieure (<).

**__gt__(self, other):** Surcharge de l'opérateur de comparaison supérieure (>).

# Surcharge des opérateurs

Voici un exemple simple pour illustrer la surcharge d'opérateurs en Python :

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

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        else:
            raise ValueError("Addition not supported for non-Point objects")

    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        else:
            return False

# Exemple d'utilisation
point1 = Point(1, 2)
point2 = Point(3, 4)

# Surcharge de l'opérateur d'addition
resultat = point1 + point2
print(resultat)  # Affichera Point(4, 6)

# Surcharge de l'opérateur d'égalité
print(point1 == point2)  # Affichera False
```

# Méthodes statiques

- **Une méthode statique** est une méthode dont l'exécution ne dépend pas d'une instance de la classe. Elle est utilisée pour définir des fonctions utiles à toute la classe
- Les méthodes statiques peuvent être appelées sans création d'instance au préalable.
- Une méthode statique peut être appelée sans instance. De ce fait, le paramètre **Self** est inutile.
- La déclaration d'une méthode statique se fait avec le décorateur **@staticmethod**

```
class Essai_class:
@staticmethod
def methode(): #déclaration d'une méthode statique
print("méthode statique")

 if __name__=='__main__':
     Essai_class.methode()
```

# TP 4

# Caractériser l’abstraction

# Classes et méthodes abstraites

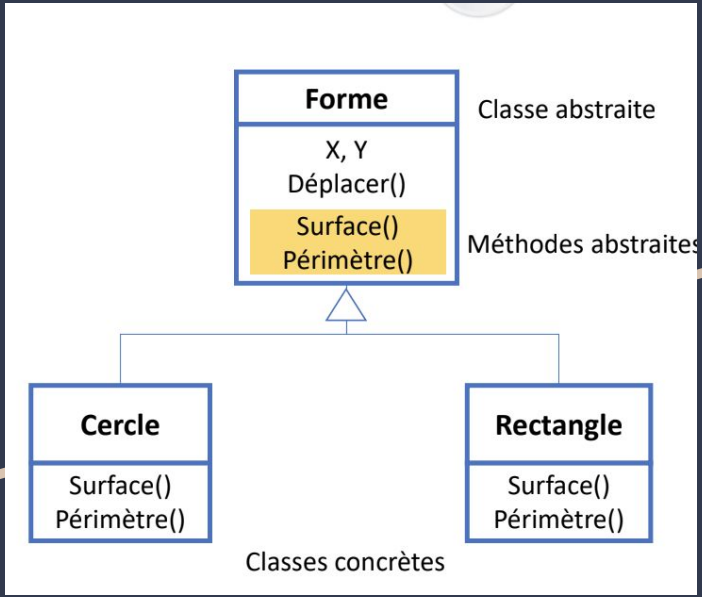
• Une **classe abstraite** est une classe qui contient une ou plusieurs méthodes **abstraites**.
Elle peut malgré tout contenir d'autres méthodes habituelles (appelées parfois méthodes **concrètes**).

• Une méthode **abstraite** est une méthode qui ne contient **pas de corps**.
Elle possède simplement une signature de définition (pas de bloc d'instructions)

# Classes et méthodes abstraites

• Une classe possédant une ou plusieurs méthodes abstraites devient obligatoirement une classe abstraite.

# Classes et méthodes abstraites




• La classe Forme ne peut pas implémenter ces deux méthodes **Surface()** et **Périmètre()** car on ne peut pas calculer le périmètre et la surface sans connaître le détail de la forme.
• On pourrait naturellement implémenter ces méthodes dans chacune des sous-classes de Forme mais **il n'y a aucune garantie que toutes les sous-classes les possèdent.**

• **Solution :** Déclaration dans la classe **Forme** deux méthodes **abstraites** périmètre() et surface()ce qui rend la classe Forme elle-même abstraite et impose aux créateurs de sous-classes de les implémenter.

# Classes et méthodes abstraites

**Les règles suivantes s'appliquent aux classes abstraites:**

• Une classe **abstraite** ne peut pas être **instanciée**
• Une sous-classe d'une classe abstraite ne peut être instanciée que si elle redéfinit chaque méthode abstraite de sa classe parente et qu'elle fournit une implémentation(un corps) pour chacune des méthodes abstraites.

• Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, **cette sous-classe est elle-même abstraite** (et ne peut donc pas être instanciée).

# Utilité des classes abstraites

• les classes abstraites permettent de **définir des fonctionnalités**(des comportements) que les sous-classes devront impérativement implémenter
• Les utilisateurs des sous-classes d'une classe **abstraite** sont donc assurés de trouver toutes les méthodes définies dans la classe abstraite dans chacune des sous-classes concrètes

• Les classes abstraites constituent donc **une sorte de contrat** (spécification contraignante) qui garantit que certaines méthodes seront disponibles dans les sous-classes et qui oblige les programmeurs à les
implémenter dans toutes les sous-classes concrètes.

# Principe d’abstraction

```
from abc import ABC, abstractmethod

# Création de classe abstraite
class Vehicule(ABC):

    @abstractmethod
    def demarrer(self):
        pass

    @abstractmethod
    def arreter(self):
        pass

    def afficher_details(self):
        print(“je suis une voiture”)
```

# TP 5

# Interface

# Principe d’interface

• Une **interface** est une forme particulière de classe où **toutes** les méthodes sont **abstraites**

• Lorsqu'une classe implémente une **interface**, elle indique ainsi qu'elle s'engage à fournir une implémentation (c'est-à-dire un **corps**) pour chacune des méthodes abstraites de cette interface.

• Si une classe implémente **plus** d'une interface, elle doit implémenter toutes les méthodes abstraites de chacune des interfaces.

# Principe d’interface

```
from abc import ABC, abstractmethod

# Création de l'interface (classe abstraite)
class Vehicule(ABC):

    @abstractmethod
    def demarrer(self):
        pass

    @abstractmethod
    def arreter(self):
        pass
```

# Principe d’interface

```
# Implémentation de la classe Moto

class Moto(Vehicule):

    def demarrer(self):
        print("Démarrage de la moto")

    def arreter(self):
        print("Arrêt de la moto")


moto = Moto()

moto.demarrer()
moto.arreter()
```

# TP 5

# Manipuler les données

# Les Collections

Les **collections** sont souvent utilisées pour stocker et manipuler des groupes d'objets. Python propose plusieurs types de collections intégrées qui peuvent être utilisées dans le contexte de la POO.
Voici quelques-unes des collections les plus couramment utilisées :

1- Listes (list)
2- Tuples (tuple)
3- Ensembles (set)
4- Dictionnaires (dict)

# Les Listes

**Listes (list) :**

Les listes sont des collections ordonnées et modifiables d'objets.
Elles sont déclarées à l'aide de crochets [].
**ma_liste** = [1, 2, 3, "quatre", 5.0]

Quelques méthodes pour gérer les listes :

- **append**(element) : Ajoute un élément à la fin de la liste.
- **extend**(iterable) : Étend la liste en ajoutant les éléments d'un iterable (par exemple, une autre liste).
- **insert**(index, element) : Insère un élément à un index spécifié.
- **remove**(element) : Supprime la première occurrence d'un élément spécifié.
- **pop**([index]) : Supprime et renvoie l'élément à l'index spécifié, ou le dernier élément si aucun index n'est spécifié.

# Les fichiers

Pour **manipuler des fichiers** en Python, vous pouvez utiliser les opérations de base fournies par la bibliothèque standard Python. Voici comment vous pouvez créer, lire, écrire et manipuler des fichiers en Python :

- Ouverture de fichiers
- Lecture de fichiers
- Écriture dans des fichiers
- Ajout de contenu à un fichier
- Fermeture de fichiers
- Gestion des erreurs
- Utilisation de la boucle **for** pour parcourir un fichier

# Ouverture de fichiers

Pour ouvrir un fichier, vous pouvez utiliser la fonction **open()**. Vous devez spécifier le chemin du fichier et le mode d'ouverture (lecture, écriture, ajout, etc.). Par exemple :

```
# Ouverture d'un fichier en mode lecture
with open('mon_fichier.txt', 'r') as fichier:
    contenu = fichier.read()
```

# Lecture de fichiers

Pour lire le contenu d'un fichier, vous pouvez utiliser les méthodes **read**(), **readline**(), ou **readlines**().

```
with open('mon_fichier.txt', 'r') as fichier:
    contenu = fichier.read()  # Lecture du contenu complet du fichier
    ligne = fichier.readline()  # Lecture d'une ligne à la fois
    lignes = fichier.readlines()  # Lecture de toutes les lignes dans une liste
```

# Écriture dans des fichiers

Pour écrire dans un fichier, vous pouvez utiliser le mode "**w**" lors de l'ouverture du fichier.
Vous pouvez ensuite utiliser les méthodes **write**() pour écrire du texte dans le fichier.

```
with open('mon_fichier.txt', 'w') as fichier:
    fichier.write("Ceci est une ligne dans le fichier.\n")
```

# Ajout de contenu à un fichier

Si vous souhaitez ajouter du contenu à un fichier existant sans écraser son contenu, vous pouvez utiliser le mode "**a**" (append).

```
with open('mon_fichier.txt', 'a') as fichier:
    fichier.write("Ceci est une nouvelle ligne ajoutée à la fin.\n")
```

# Fermeture de fichiers

Il est important de fermer les fichiers après les avoir ouverts en utilisant la clause **with**.
Cela garantit que toutes les ressources associées au fichier sont correctement libérées.

# Gestion des erreurs

Lorsque vous travaillez avec des fichiers, il est important de gérer les erreurs, notamment lors de l'ouverture du fichier. Vous pouvez utiliser des blocs **try**...**except** pour gérer les exceptions liées aux fichiers.

```
try:
    with open('mon_fichier_inexistant.txt', 'r') as
fichier:
        contenu = fichier.read()
except FileNotFoundError:
    print("Le fichier n'existe pas.")
```

# Utilisation de la boucle **for** pour parcourir un fichier

Vous pouvez également utiliser une boucle **for** pour parcourir un fichier ligne par ligne.

```
with open('mon_fichier.txt', 'r') as fichier:
    for ligne in fichier:
        print(ligne, end='')
```

# La lecture du fichier JSON

Le JavaScript Object Notation (JSON) est le format d'échange de données, populaire auprès des développeurs, car il se présente sous la forme de texte léger et lisible en plus de nécessiter moins de codage et de rendre les processus plus rapides.

Pour lire un fichier **JSON** en Python, vous pouvez utiliser le module json intégré. Voici un exemple simple :
Supposons que vous ayez un fichier JSON appelé "**exemple.json**" avec le contenu suivant :

```
{
    "nom": "John Doe",
    "age": 30,
    "ville": "New York"
}
```

Vous pouvez lire ce fichier en utilisant le code suivant :

```
import json


with open('exemple.json', 'r') as fichier_json:

    donnees = json.load(fichier_json)

     print(donnees)

#https://jsonplaceholder.typicode.com/todos/1
```

# La lecture du fichier CSV

Un fichier CSV (valeurs séparées par des virgules) est un type de fichier spécial qu'il est possible de créer ou de modifier dans Excel. Plutôt que de stocker les informations en colonnes, les fichiers CSV les stockent en les séparant par des points-virgules.

Pour lire un fichier **CSV** en Python, vous pouvez utiliser le module json intégré. Voici un exemple simple : Supposons que vous ayez un fichier JSON appelé "**exemple.csv**" avec le contenu suivant :

```
Nom,Âge,Ville
John Doe,30,New York
Jane Doe,25,Los Angeles
Bob Smith,35,Chicago
```

Vous pouvez lire ce fichier en utilisant le code suivant :

```
import csv


with open('exemple.csv, 'r',newline='') as fichier_csv:

    lecteur_csv = csv.reader(fichier_csv)

    for ligne in lecteur_csv:
        print(ligne)
```

# TP 6

# les
# expressions régulières

# les expressions régulières

**Les expressions régulières** également appelées **regex** ou **regexp**, sont un outil puissant pour la recherche, la validation et la manipulation de motifs de texte dans Python. Python fournit le module re pour travailler avec des expressions régulières.

1- Importer le module **re** :

```
import re
text = "Les numéros de téléphone sont 123-456-7890 et 987-654-3210."
numbers = re.findall(r'\d{3}-\d{3}-\d{4}', text)
print("Numéros de téléphone trouvés :", numbers)
```

**findall** : retourne une liste.

# les expressions régulières

^ Marque le début de la chaine, la ligne.
$ Marque la fin d'une chaine, la ligne.
. N'importe quel caractère.
* 0, 1 ou plusieurs occurrences.
+ 1 ou plusieurs occurrences.
? 0 ou 1 occurrence.
| Alternative - ou reconnaît l'un ou l'autre.
[ ] Tous les caractères énumérés dans la classe.
[^ ] Tous les caractères sauf ceux énumérés.
( ) Utilisée pour limiter la portée d'un masque ou de l'alternative.
\w Les lettres (w pour word).
\d Les chiffres (d pour digit).
\s Les espaces (s pour spaces).
[A-Z] Les majuscules.
[abd;_] Les lettres a, b, et d, le point-virgule (;), et l’underscore (_).

# les expressions régulières

**Validation de formats de texte** (par exemple, un numéro de téléphone) :
Vous pouvez utiliser des expressions régulières pour valider des formats de texte, comme un numéro de téléphone.

```
import re

motif = r"\d{3}-\d{3}-\d{4}"
numero = "123-456-7890"

if re.match(motif, numero):
    print("Numéro de téléphone valide.")
else:
    print("Numéro de téléphone invalide.")
```

**match :** est utilisée pour rechercher un motif spécifique au début d'une chaîne de caractères.
**search** : cherche des correspondances dans toute la chaîne.

# les expressions régulières

**La validation d'une adresse e-mail**.

```
import re

# Motif pour la validation d'une adresse e-mail simple
motif_email =
r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"

# Liste d'exemples d'adresses e-mail à valider
adresses_email = [
    "exemple@email.com",
    "nom_utilisateur@domaine.co.uk",
    "trop@de@symboles.com",
]

# Valider chaque adresse e-mail
for adresse in adresses_email:
    if re.match(motif_email, adresse):
        print(adresse," est une adresse e-mail valide.")
    else:
        print(adresse, "n'est pas une adresse e-mail
valide.")
```

# les expressions régulières

**Utilisation des expressions régulières dans un projet :**

BiblioRequest : récupérer les emails d’une page web.

# TP

**Exercice :**

https://python.doctor/page-expressions-regulieres-regular-python

# Les fonctions imbriquées

# Les fonctions imbriquées

Les **fonctions imbriquées** sont des fonctions définies à l'intérieur d'une autre fonction.
En Python, vous pouvez déclarer une fonction à l'intérieur d'une autre fonction, et la fonction interne a accès aux variables de la fonction externe.

Voici un exemple pour illustrer cela :

```
def fonction_externe(x):
    def fonction_interne(y):
        return x + y

    resultat = fonction_interne(5)
    return resultat

# Appel de la fonction externe
resultat_final = fonction_externe(10)

print(resultat_final)  # Affichera 15
```

# Fonction Lambda

# Fonction Lambda

**Lambda :**

**Lambda** est utilisé pour créer des fonctions anonymes, également appelées fonctions lambda.
Les fonctions **lambda** sont des fonctions inline simples qui peuvent avoir n'importe quel nombre de paramètres d'entrée, mais elles ne peuvent comporter qu'une seule expression. Elles sont souvent utilisées pour des opérations courtes et simples.

Voici la syntaxe de base d'une fonction lambda :

```
lambda arguments: expression
```

Exemple :

```
carre = lambda x: x**2
print(carre(5))  # Sortie : 25

#sans lambda

def carre(x):
    return x**2

# Utilisation de la fonction
resultat = carre(5)
print(resultat)  # Sortie : 25
```

# None

# None

En Python, **None** est un objet qui représente **l'absence de valeur** ou une **valeur nulle.**
C'est souvent utilisé pour indiquer qu'une variable ou une fonction ne renvoie aucune valeur.

**Affectation initiale :**
variable = None #null

**Suppression de valeur :**
ma_variable = 42
ma_variable = None  # Supprime la valeur précédente et attribue None

**Paramètre par défaut dans une méthode :**
```
def méthode_avec_parametre(parametre=None):
    if parametre is None:
        print("Pas de valeur fournie pour le paramètre.")

méthode_avec_parametre()  # Affichera: Pas de valeur fournie pour le paramètre.
```

En résumé, **None** est utilisé pour représenter l'absence de valeur explicite ou la non-existence d'une valeur.

# Administrer les exceptions

# Administrer les exceptions

**Types d'erreurs :**

En Python, il existe principalement deux types d'erreurs : **les erreurs de syntaxe** et **les erreurs d'exécution**.

**Erreurs de syntaxe** : Ces erreurs surviennent lorsqu'il y a une violation des règles de syntaxe de Python. Elles empêchent votre code de s'exécuter correctement et sont généralement détectées par l'interpréteur Python lors de la phase d'analyse. Par exemple, les erreurs d'indentation, les parenthèses non fermées ou les guillemets manquants sont des erreurs de syntaxe courantes.

**Erreurs d'exécution** : Ces erreurs se produisent lorsqu'un programme est en cours d'exécution, mais rencontre un problème inattendu. Elles sont généralement causées par des conditions inattendues, telles que la division par zéro, l'indice hors limites d'une liste, ou l'utilisation d'une variable non définie.

# Administrer les exceptions

**Types d'exceptions :**

Les exceptions sont des objets Python qui représentent des erreurs d'exécution. Elles sont utilisées pour signaler et gérer des erreurs lors de l'exécution d'un programme. Python offre un grand nombre de types d'exceptions intégrés, comme :

- **ZeroDivisionError**: Se produit lorsque vous tentez de diviser par zéro.
- **IndexError**: Se produit lorsque vous accédez à un indice invalide dans une séquence (par exemple, une liste ou une chaîne de caractères).
- **NameError**: Se produit lorsque vous tentez d'accéder à une variable qui n'a pas été définie.
- **TypeError**: Se produit lorsque vous effectuez des opérations incompatibles sur des types de données.
- **FileNotFoundError**: Se produit lorsque vous tentez d'ouvrir un fichier qui n'existe pas.

# Administrer les exceptions

**Gestion des exceptions :**

Pour gérer les exceptions en Python, vous pouvez utiliser des blocs **try** et **except**. Voici comment cela fonctionne :

**Utilisation de blocs try, except, else et finally :**

```
try:
    # Code susceptible de générer une exception
    resultat = 10 / 0
except ZeroDivisionError:
    # Bloc exécuté si une exception de type ZeroDivisionError est levée
    print("Division par zéro détectée!")
else:
    # Bloc exécuté si aucune exception n'est levée
    print("Aucune exception détectée.")
finally:
    # Bloc exécuté quoi qu'il arrive
    print("Fin de la gestion des exceptions.")
```

# Administrer les exceptions

**Capturer plusieurs types d'exceptions :**

```
try:
    # Code susceptible de générer une exception
    resultat = int("abc")
except (ValueError, TypeError):
    # Bloc exécuté si une exception de type
ValueError ou TypeError est levée
    print("Erreur de conversion.")
```

# Administrer les exceptions

**Capturer n'importe quelle exception :**

```
try:
    # Code susceptible de générer une exception
    resultat = int("abc")
except Exception as e:
    # Bloc exécuté si n'importe quelle exception est levée
    print(f"Une exception s'est produite: {e}")
```

# Administrer les exceptions

**Lever des exceptions personnalisées :**

```
class MonException(Exception):
    pass

try:
    # Code susceptible de générer une exception

except MonException as e:
    # Bloc exécuté si une exception de type
MonException est levée
    print(f"Exception personnalisée: {e}")
```

# TP

# Manipuler les modules et les bibliothèques

# Création des modules

En Python, vous pouvez créer des modules pour **organiser** et **réutiliser** du code.

Un **module** est simplement un fichier contenant des définitions des méthodes de classes et des attributs.
Vous pouvez ensuite **importer** ces modules dans d'autres scripts Python pour réutiliser le code.

# Création des modules

Voici comment créer et utiliser des modules en Python :

**1- Créez un fichier Python**
**2- Définissez vos méthodes, classes et attributs**
**3- Utilisez le module dans un autre script**
**4- Exécuter le script**

# Création des modules

**1- Créez un fichier Python :**

Tout d'abord, créez un nouveau fichier Python avec l'extension "**.py**" (par exemple, "mon_module.py"). C'est dans ce fichier que vous allez définir vos fonctions, classes et variables.

# Création des modules

**2- Définissez vos fonctions, classes et variables :**
À l'intérieur de votre fichier Python, définissez les fonctions, les classes et les variables que vous souhaitez inclure dans votre module.
Par exemple :

```
# mon_module.py

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

ma_variable = 42

class MaClasse:
    def __init__(self):
        print("Ceci est ma classe")
```

# Importation du module

**3- Utilisez le module dans un autre script :**
Vous pouvez maintenant importer votre module dans un autre script Python et utiliser ses éléments.
Pour ce faire, utilisez l'instruction **import** :

```
# script.py

import mon_module

mon_module.ma_fonction()  # Appeler la fonction du module
print(mon_module.ma_variable)  # Accéder à la variable du module

obj = mon_module.MaClasse()  # Créer une instance de la classe du module
```

# Importation du module

**4- Exécuter le script :**

Exécutez votre script Python comme d'habitude. Il importera le module et utilisera ses éléments.

# Création d’un package

# Création d'un package

```
mon_package/
  ├──── __init__.py
  ├──── module1.py
  ├──── module2.py
```

Un **package** est simplement un répertoire (**dossier**) qui contient un fichier **spécial** appelé __init__.py
Voici comment vous pouvez créer un package en Python :

**1- Créez la structure du package :**
Assurez-vous d'avoir une structure de dossier qui ressemble à ceci :

- **mon_package** est le nom de votre package.
- **__init__.py** est un fichier spécial qui peut être vide, mais il doit être présent pour que Python reconnaisse le dossier comme un package.
- **module1.py, module2.py, etc.**, sont des modules (fichiers Python) inclus dans le package.

# Création d'un package

```
mon_package/
 ├──── __init__.py
 ├──── module1.py
 ├──── module2.py
```

**2- Contenu des fichiers :**

- mon_package/__init__.py (peut être vide) :
- mon_package/module1.py :

```
def fonction_module1():
    print("Je suis dans le module 1")
```

- mon_package/module2.py :

```
def fonction_module2():
    print("Je suis dans le module 2")
```

# Création d'un package

```
mon_package/
 ├── __init__.py
 ├── module1.py
 ├── module2.py
```

### 3- Utilisation du package

Vous pouvez ensuite utiliser votre package dans d'autres scripts Python en important les modules nécessaires. Par exemple :

from **mon_package** import **module1**

module1.**fonction_module1()**

# TP 9

# Manipuler les bibliothèques à distance

# Installation des bibliothèques externes (pip)

L'installation de bibliothèques externes en Python se fait généralement à l'aide de l'outil **pip** (Package Installer for Python).

Utilisez la commande **pip install** suivie du nom de la bibliothèque que vous souhaitez installer.

Par exemple, pour installer la bibliothèque **requests**, vous pouvez exécuter la commande suivante dans votre terminal ou votre invite de commande :

```
pip install requests
```

# Installation des bibliothèques externes (pip)

Exemple d'utilisation de la bibliothèque **requests**

```
import requests


url = 'https://www.google.com'

# GET request
response = requests.get(url)

if response.status_code == 200:
    print(response.text)
else:
    print("Error: ",response.status_code)
```

# Installation des bibliothèques externes (pip)

TP 10 Exercice 1

# Bibliothèques standards

Python est livré avec une riche bibliothèque standard qui offre de nombreux modules et packages prêts à l'emploi pour effectuer une variété de tâches.

Voici quelques-unes des bibliothèques standard les plus couramment utilisées :

**os** : Fournit une interface pour interagir avec le système d'exploitation, comme la gestion des fichiers et des répertoires.
**math** : Offre des fonctions mathématiques pour les opérations courantes.
**datetime** : Permet de manipuler des objets de date et d'heure.
**random** : Fournit des outils pour générer des nombres aléatoires.
**re** : Permet d'utiliser des expressions régulières pour la recherche et la manipulation de chaînes de caractères.

Exemple d’utilisation :

```
import re
```

# Bibliothèques graphiques

Il existe plusieurs bibliothèques graphiques en Python qui vous permettent de créer des interfaces graphiques (**GUI**) pour vos applications. Voici quelques-unes des bibliothèques populaires :

**Tkinter** est la bibliothèque GUI standard de Python et est souvent inclus avec l'installation par défaut de Python.
**PyQt** et **PySide** sont des ensembles de liaisons Python pour la bibliothèque Qt.
**Kivy** est une bibliothèque open-source conçue pour le développement d'applications multitouch.
**wxPython** est une liaison Python pour le toolkit wxWidgets.
**Pygame** est une bibliothèque spécialisée dans le développement de jeux vidéo en 2D

# Création des bibliothèques
## à distance

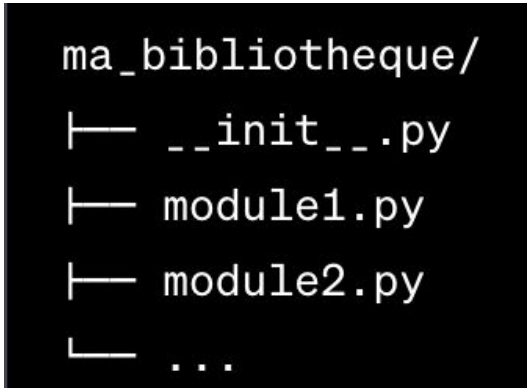
# Création des bibliothèques

La création de bibliothèques en Python implique la définition et l'organisation de modules réutilisables que d'autres développeurs peuvent intégrer dans leurs propres projets. Voici les étapes générales pour créer une bibliothèque en Python :

**1- Organisez votre code dans un répertoire**
**2- Créez des modules**
**3- Documentez votre code**
**4- Ajoutez des tests**
**5- Créez un fichier setup.py**
**6- Publiez votre bibliothèque**

# Création des bibliothèques

**1- Organisez votre code dans un répertoire**
Créez un répertoire dédié à votre bibliothèque. Ce répertoire peut contenir plusieurs fichiers, chacun représentant un module ou une partie spécifique de votre bibliothèque.




Le fichier **__init__.py** indique à Python que le répertoire doit être traité comme un package.

# Création des bibliothèques

**2- Créez des modules**

Divisez votre code en modules, où chaque module est un fichier Python contenant des fonctions, des classes ou d'autres éléments liés.

Par exemple, **module1.py** pourrait contenir les fonctions liées à une partie spécifique de votre bibliothèque.

```
# module1.py

def fonction1():
    # ...

def fonction2():
    # ...
```

# Création des bibliothèques

**3- Documentez votre code**

Utilisez des docstrings pour documenter chaque fonction et module. Une documentation claire facilite la compréhension et l'utilisation de votre bibliothèque par d'autres développeurs.

```
# module1.py

def fonction1():
    """
    Documentation de la fonction1.
    """
    # ...

def fonction2():
    """
    Documentation de la fonction2.
    """
    # ...
```

# Création des bibliothèques

**4- Ajoutez des tests**

Créez des **tests unitaires** pour assurer la fiabilité de votre code. Vous pouvez utiliser des outils tels que unittest ou pytest

# Création des bibliothèques

**5- Créez un fichier setup.py**

(si vous envisagez de distribuer votre bibliothèque) :
Le fichier **setup.py** contient des informations sur votre bibliothèque, comme son nom, sa version, et les dépendances nécessaires.

```
# setup.py
from setuptools import setup, find_packages

setup(
    name='ma_bibliotheque',
    version='0.1',
    packages=find_packages(),
    install_requires=[
        # Listez vos dépendances ici
    ],
)
```

# Création des bibliothèques

**6- Publiez votre bibliothèque**

Si vous souhaitez partager votre bibliothèque avec d'autres développeurs, vous pouvez la publier sur le Python Package Index (**PyPI**).
Assurez-vous d'avoir installé **twine** via pip **install twine**

# Importation des bibliothèques

L'importation des bibliothèques en Python se fait à l'aide du mot-clé **import**.
Voici comment vous pouvez importer des bibliothèques dans votre code Python :

**1- Importation simple :**
Vous pouvez importer toute la bibliothèque ou un module spécifique en utilisant le mot-clé **import**.

**2- Importation sélective :**
Si vous n'avez besoin que de quelques éléments spécifiques d'une bibliothèque, vous pouvez les importer directement.

```
from ma_bibliotheque.module1 import fonction1, fonction2
```

**3- Importation de tous les éléments :**
Bien que cela ne soit généralement pas recommandé pour éviter les conflits de noms, vous pouvez importer tous les éléments d'une bibliothèque dans l'espace de noms actuel.

```
from ma_bibliotheque import *
```

# Installation de bibliothèques

L'installation de bibliothèques Python se fait généralement à l'aide de l'outil **pip** (Package Installer for Python).

**1- Installer une bibliothèque spécifique :**
**pip install** nom_de_la_bibliotheque

**2- Installer une version spécifique d'une bibliothèque :**
pip install nom_de_la_bibliotheque==**version**

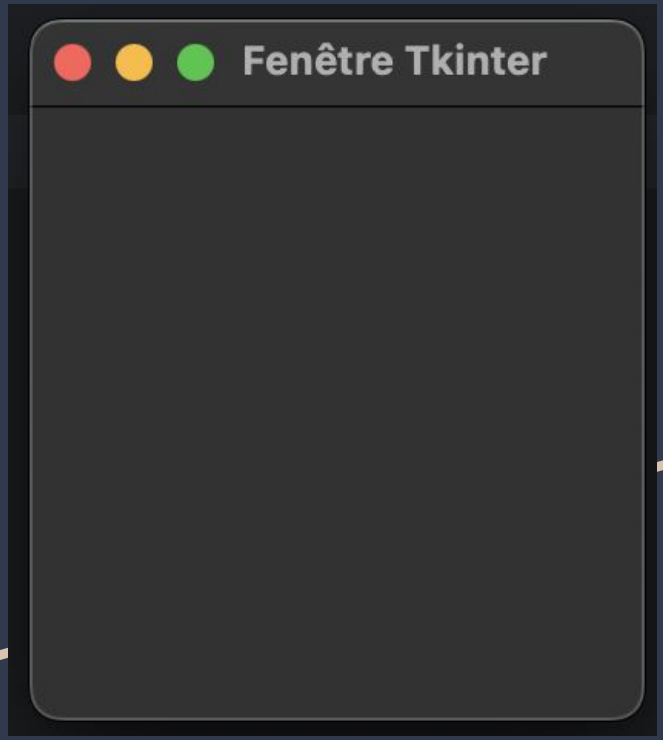
**3- Installer depuis un fichier local :**
pip install /chemin/vers/votre/bibliotheque

# Les interfaces graphiques

# Tkinter

**Tkinter** est la bibliothèque graphique standard pour créer des interfaces graphiques utilisateur (GUI) en Python. Elle est basée sur le toolkit graphique Tk, et elle est généralement incluse avec l'installation par défaut de Python.

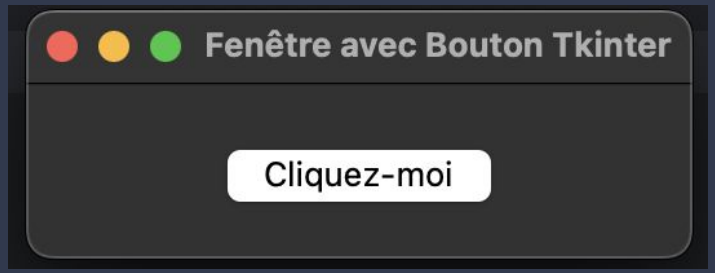
# Fenêtre




La création d'une fenêtre avec Tkinter est une étape fondamentale pour développer des applications GUI en Python. Voici un exemple de base pour créer une fenêtre Tkinter :

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Fenêtre Tkinter")

# Votre contenu ici

fenetre.mainloop()
```

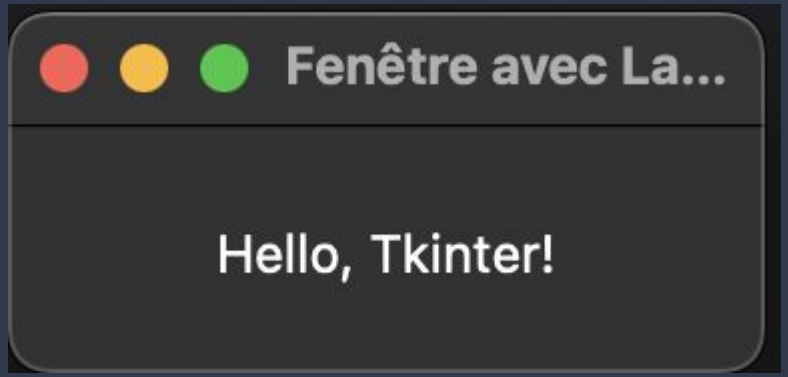
# Bouton




Un exemple simple de création d'une fenêtre Tkinter avec un bouton.

```
import tkinter as tk

def clic_sur_bouton():
    label.config(text="Bonjour, Tkinter!")

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Bouton Tkinter")

# Création d'un bouton
bouton = tk.Button(fenetre, text="Cliquez-moi",
command = clic_sur_bouton)
bouton.pack(pady=20, padx=20)
# Ajoute un peu d'espace autour du bouton

fenetre.mainloop()
```

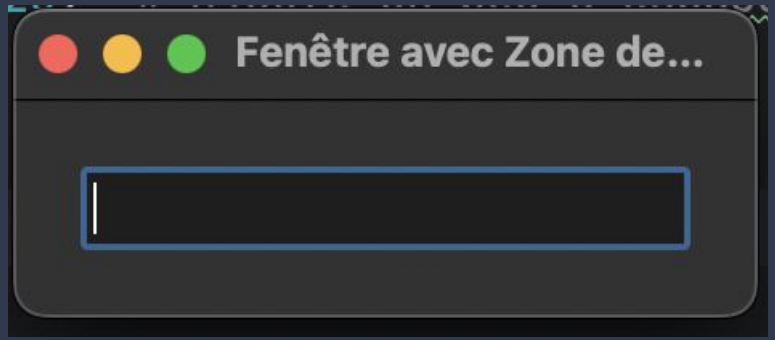
# Label




Si vous souhaitez ajouter un label à votre fenêtre Tkinter avec le bouton, voici comment vous pouvez le faire :

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Label Tkinter")

label = tk.Label(fenetre, text="Hello, Tkinter!")
label.pack(pady=20, padx=20)
# Ajoute un peu d'espace autour du label

fenetre.mainloop()
```

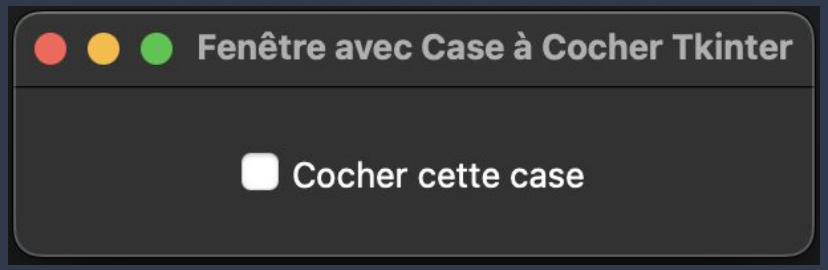
# Zone de saisie




Si vous souhaitez créer une fenêtre Tkinter avec uniquement une zone de saisie (Entry):

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Zone de Saisie Tkinter")

# Création de la zone de saisie
zone_saisie = tk.Entry(fenetre)
zone_saisie.pack(pady=20, padx=20)
# Ajoute un peu d'espace autour de la zone de saisie

fenetre.mainloop()
```

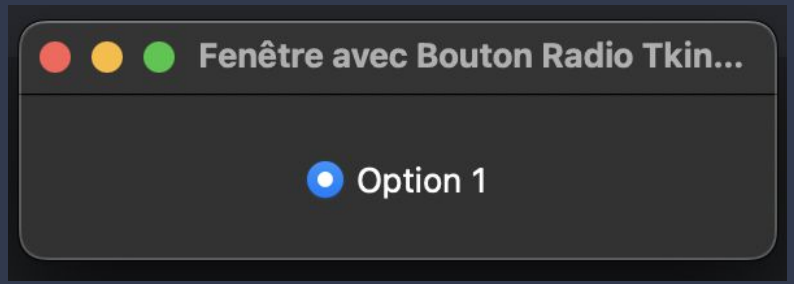
# Case à cocher




Si vous souhaitez créer une fenêtre Tkinter avec uniquement une case à cocher (Checkbutton).

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Case à Cocher Tkinter")

# Variable pour stocker l'état de la case à cocher
etat_case = tk.BooleanVar()

# Création de la case à cocher
case_a_cocher = tk.Checkbutton(fenetre,
text="Cocher cette case", variable=etat_case)
case_a_cocher.pack(pady=20, padx=20)

# Ajoute un peu d'espace autour de la case à
cocher

fenetre.mainloop()
```

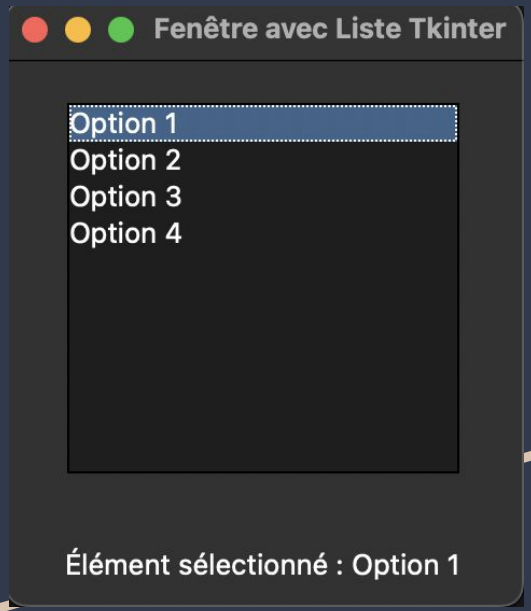
# Bouton radio




Si vous souhaitez créer une fenêtre Tkinter avec uniquement un bouton radio (Radiobutton).

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Bouton Radio Tkinter")

# Variable pour stocker la valeur sélectionnée du bouton radio
valeur_selectionnee = tk.StringVar()

# Création du bouton radio
bouton_radio = tk.Radiobutton(fenetre, text="Option 1", variable=valeur_selectionnee, value="Option 1")
bouton_radio.pack(pady=20, padx=20)
# Ajoute un peu d'espace autour du bouton radio

fenetre.mainloop()
```

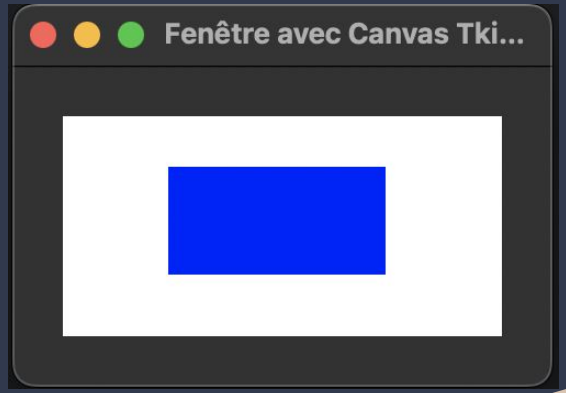
# Listes




une fenêtre Tkinter avec une liste :

```
import tkinter as tk

def selectionner_element(event):
    index = liste_curseur.curselection()
    element_selectionne = liste_curseur.get(index)
    label_resultat.config(text=f"Élément sélectionné :
{element_selectionne}")

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Liste Tkinter")

elements = ["Option 1", "Option 2", "Option 3", "Option 4"]

liste_curseur = tk.Listbox(fenetre, selectmode=tk.SINGLE)
liste_curseur.bind("<<ListboxSelect>>", selectionner_element)
liste_curseur.pack(padx=20, pady=20)

label_resultat = tk.Label(fenetre, text="Élément sélectionné : ")
label_resultat.pack(pady=10)

for element in elements:
    liste_curseur.insert(tk.END, element)

fenetre.mainloop()
```

# Canvas




un **Canvas** (zone de dessin) de dimensions 200x100 pixels et une couleur de fond blanche. Un rectangle bleu est dessiné à l'intérieur du Canvas.

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Canvas Tkinter")

canvas = tk.Canvas(fenetre, width=200,
height=100, bg="white")
canvas.pack(padx=20, pady=20)

# Dessiner un rectangle sur le Canvas
rectangle = canvas.create_rectangle(50, 25, 150,
75, fill="blue")

fenetre.mainloop()
```

# Scale


Le widget **Scale** en Tkinter est un composant graphique qui permet à l'utilisateur de sélectionner une valeur numérique dans une plage prédéfinie en faisant glisser un curseur.

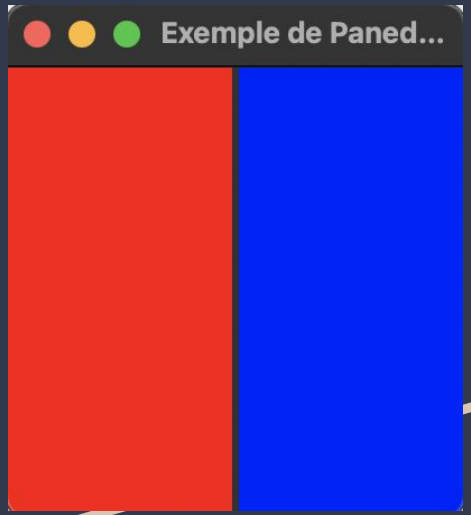
```
import tkinter as tk

def mise_a_jour_valeur(valeur):
    label_resultat.config(text=f"Valeur : {valeur}")

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Scale Tkinter")

# Création du widget Scale
echelle = tk.Scale(fenetre, from_=0, to=100,
orient=tk.HORIZONTAL,
command=mise_a_jour_valeur)
echelle.pack(padx=20, pady=20)

label_resultat = tk.Label(fenetre, text="Valeur : 0")
label_resultat.pack(pady=10)

fenetre.mainloop()
```

# Frame


Un **Frame** en Tkinter est un conteneur rectangulaire utilisé pour regrouper d'autres **widgets** (éléments graphiques) dans une fenêtre graphique.

```
import tkinter as tk

def clic_sur_bouton():
    label_resultat.config(text="Bouton cliqué!")

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Frame Tkinter")

# Création du cadre (Frame)
cadre = tk.Frame(fenetre, borderwidth=2, relief=tk.GROOVE)
cadre.pack(padx=20, pady=20)

# Création d'un bouton à l'intérieur du cadre
bouton = tk.Button(cadre, text="Cliquez-moi",
command=clic_sur_bouton)
bouton.pack(pady=10, padx=10)

# Création d'une étiquette à l'intérieur du cadre
label_resultat = tk.Label(cadre, text="Résultat ici")
label_resultat.pack(pady=10)

fenetre.mainloop()
```

# PanedWindow




**PanedWindow** est un widget Tkinter qui fournit une fenêtre divisée (splitter window) qui peut être utilisée pour créer des interfaces utilisateur avec des zones redimensionnables.

```
import tkinter as tk

fenetre = tk.Tk()
fenetre.title("Exemple de PanedWindow")

# Création du PanedWindow
paned_window = tk.PanedWindow(fenetre,
orient=tk.HORIZONTAL)
paned_window.pack(expand=True, fill=tk.BOTH)

# Ajout de deux cadres au PanedWindow
cadre1 = tk.Frame(paned_window, background="red",
width=100, height=200)
cadre2 = tk.Frame(paned_window, background="blue",
width=100, height=200)

paned_window.add(cadre1)
paned_window.add(cadre2)

fenetre.mainloop()
```

# Spinbox


La **Spinbox** est un widget Tkinter qui fournit une boîte de saisie de nombres avec des flèches d'incrémentation et de décrémentation à côté.

```
import tkinter as tk


# Fonction appelée lors de la modification de la Spinbox
def mise_a_jour_valeur():
    valeur = spinbox.get()
    label_resultat.config(text=f"Valeur : {valeur}")

fenetre = tk.Tk()
fenetre.title("Fenêtre avec Spinbox Tkinter")

# Création de la Spinbox
spinbox = tk.Spinbox(fenetre, from_=0, to=10,
command=mise_a_jour_valeur)
spinbox.pack(pady=20)

# Étiquette pour afficher la valeur de la Spinbox
label_resultat = tk.Label(fenetre, text="Valeur : ")
label_resultat.pack(pady=10)

fenetre.mainloop()
```

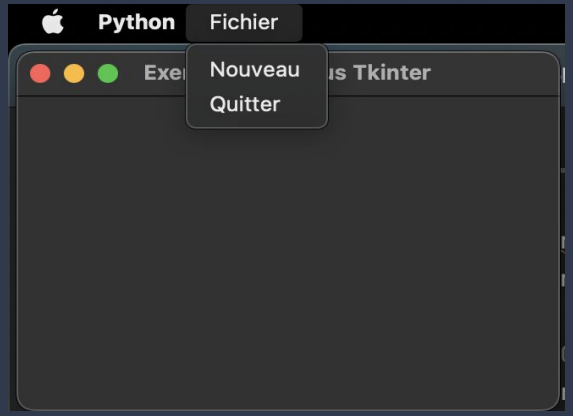
# Alertes (MessageBox)

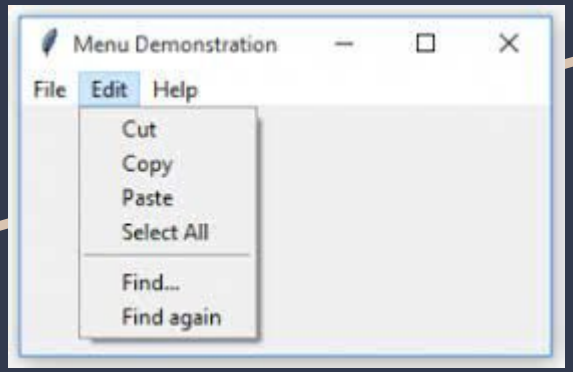

les boîtes de dialogue **MessageBox** sont utilisées pour afficher des alertes, des avertissements, et demander des confirmations à l'utilisateur.

Tkinter fournit trois types :
**1- showinfo:** Affiche une boîte de dialogue d'information.
**2- showwarning:** Affiche une boîte de dialogue d'avertissement.
**3- showerror:** Affiche une boîte de dialogue d'erreur.

```
import tkinter as tk
from tkinter import messagebox

def afficher_alerte():
    messagebox.showinfo("Information", "Ceci est un message d'information.")

fenetre = tk.Tk()
fenetre.title("Exemple MessageBox Tkinter")

bouton = tk.Button(fenetre, text="Afficher Alerte", command=afficher_alerte)
bouton.pack(pady=20)

fenetre.mainloop()
```

# Menus




les **menus** sont utilisés pour créer des barres de menus déroulants, des menus contextuels, et d'autres interfaces utilisateur basées sur des choix. Les menus peuvent contenir des éléments tels que des **options**, des **sous-menus,** des **cases à cocher,** des **boutons radio**, etc.

```
import tkinter as tk
from tkinter import messagebox

def afficher_alerte():
    messagebox.showinfo("Information", "Ceci est un message
d'information.")

def quitter_application():
    fenetre.quit()

fenetre = tk.Tk()
fenetre.title("Exemple de Menus Tkinter")

# Création de la barre de menus
barre_menus = tk.Menu(fenetre)

# Création d'un menu "Fichier" avec deux commandes
menu_fichier = tk.Menu(barre_menus, tearoff=0)
menu_fichier.add_command(label="Nouveau", command=afficher_alerte)
menu_fichier.add_command(label="Quitter", command=quitter_application)

# Ajout du menu "Fichier" à la barre de menus
barre_menus.add_cascade(label="Fichier", menu=menu_fichier)
# Configuration de la fenêtre pour utiliser la barre de menus
fenetre.config(menu=barre_menus)

fenetre.mainloop()
```

# Manipulation des méthodes/événements et options des widgets

La manipulation des méthodes, événements et options des widgets en Tkinter est essentielle pour créer des interfaces utilisateur interactives.
Voici un aperçu de ces concepts :

**Méthodes des Widgets :**
Les widgets en Tkinter ont des méthodes que vous pouvez appeler pour effectuer certaines actions ou obtenir des informations.
Par exemple, pour changer le texte d'une étiquette, vous utilisez la méthode **config** ou **configure** :

```
etiquette = tk.Label(fenetre, text="Bonjour")
etiquette.config(text="Nouveau texte")
```

# Manipulation des méthodes/événements et options des widgets

**Événements :**
Les événements sont des actions déclenchées par l'utilisateur, tels que des clics de souris, des pressions de touches, etc. Vous pouvez lier des fonctions à ces événements pour effectuer des actions spécifiques.

```
def clic_sur_bouton():
    print("Bouton cliqué!")

bouton = tk.Button(fenetre, text="Cliquez-moi",
command=clic_sur_bouton)
```

# Manipulation des méthodes/événements et options des widgets

**Options des Widgets :**
Les widgets ont des options (paramètres) que vous pouvez spécifier lors de leur création ou modifier ultérieurement.
Par exemple, pour définir la couleur d'arrière-plan d'un cadre :

```
cadre = tk.Frame(fenetre, bg="blue")
```

# Manipulation des méthodes/événements et options des widgets


Dans cet exemple, nous avons utilisé les méthodes (**config**, **pack**), les événements (**command**), et les options (**bg**) pour créer une fenêtre Tkinter.

```
import tkinter as tk

def clic_sur_bouton():
    etiquette.config(text="Bouton cliqué!")

fenetre = tk.Tk()
fenetre.title("Exemple Tkinter")

cadre = tk.Frame(fenetre, bg="lightgray")
cadre.pack(padx=20, pady=20)

etiquette = tk.Label(cadre, text="Hello, Tkinter!")
etiquette.pack(pady=10)
bouton = tk.Button(cadre, text="Cliquez-moi",
command=clic_sur_bouton)
bouton.pack()
fenetre.mainloop()
```

# TP 11 & 12: UI

# Les Principaux Widgets Tkinter

| Widget | Description |
| --- | --- |
| Label | Affiche du texte ou une image |
| Button | Un bouton cliquable |
| Entry | Champ de saisie de texte |
| Text | Zone de texte multi-lignes |
| Listbox | Liste déroulante |
| Checkbutton | Case à cocher |
| Frame | Conteneur pour d'autres widgets |
| Canvas | Dessin graphique |
| Menu | Barre de menus |

# Exemple

Un **Label** et un **Entry** pour saisir du texte

```
import tkinter as tk

root = tk.Tk()

root.title("Exemple Label et Entry")

tk.Label(root, text="Nom :").pack()

entry_nom = tk.Entry(root)

entry_nom.pack()

root.mainloop()
```

# SQLite

## Définition

**SQLite** est une base de données légère, stockée dans un fichier local. Contrairement aux bases de données classiques (MySQL, PostgreSQL), SQLite ne nécessite pas de serveur.

**Pourquoi utiliser SQLite ?**

✔ Simple à utiliser

✔ Intégré à Python (sqlite3)

✔ Idéal pour les petites applications

✔ Pas besoin de configuration complexe

# SQL

**Les commandes :**

CREATE TABLE : Créer une table

INSERT INTO : Ajouter des données

SELECT : Lire les données

UPDATE : Modifier des données

DELETE : Supprimer des données

# Lier Tkinter et SQLite

Pour lier une interface Tkinter avec une base SQLite, on suit ces étapes :

**1 Créer une base de données SQLite**

**2 Créer une interface utilisateur avec Tkinter**

**3 Relier les widgets Tkinter aux actions SQL (insertion, suppression, affichage, etc.)**

**4 Mettre à jour l'interface après chaque action**

# Lier Tkinter et SQLite

## Créer une base de données SQLite

SQLite est une base de données légère qui ne nécessite pas de serveur. On utilise le module `sqlite3` de Python pour interagir avec elle.

**Étapes :**

- Importer le module `sqlite3`
- Se connecter à la base de données (le fichier `.db` est créé automatiquement s'il n'existe pas)
- Créer une table pour stocker les données

# Lier Tkinter et SQLite

```
import sqlite3

# Connexion à la base (créée si elle n'existe pas)

conn = sqlite3.connect("exemple.db")

cursor = conn.cursor()

# Création d'une table

cursor.execute(""" CREATE TABLE IF NOT EXISTS
utilisateurs (

    id INTEGER PRIMARY KEY AUTOINCREMENT,

    nom TEXT NOT NULL,

    age INTEGER NOT NULL ) """)

# Insertion d'un utilisateur

cursor.execute("INSERT INTO utilisateurs (nom, age)
VALUES (?, ?)", ("Alice", 25))

conn.commit()  # Sauvegarde les modifications

conn.close()   # Ferme la connexion
```

# Lier Tkinter et SQLite

## Créer une interface utilisateur avec Tkinter

Tkinter est la bibliothèque standard de Python pour créer des interfaces graphiques.

**Étapes :**

- Importer `tkinter`
- Créer une fenêtre principale
- Ajouter des widgets (`Label`, `Entry`, `Button`, `Listbox`, etc.)

# Lier Tkinter et SQLite

```
import tkinter as tk

# Création de la fenêtre principale
root = tk.Tk()
root.title("Gestion des Utilisateurs")

# Ajout de widgets
label_nom = tk.Label(root, text="Nom:")
label_nom.grid(row=0, column=0)

entry_nom = tk.Entry(root)
entry_nom.grid(row=0, column=1)

label_age = tk.Label(root, text="Âge:")
label_age.grid(row=1, column=0)

entry_age = tk.Entry(root)
entry_age.grid(row=1, column=1)

btn_ajouter = tk.Button(root, text="Ajouter")
btn_ajouter.grid(row=2, column=0, columnspan=2)

root.mainloop()
```

# Lier Tkinter et SQLite

### Relier les widgets Tkinter aux actions SQL

On va maintenant **ajouter des fonctions** pour insérer, afficher et supprimer des données.

**Étapes :**

- Ajouter une fonction `ajouter_utilisateur()` pour insérer des données
- Ajouter une fonction `afficher_utilisateurs()` pour récupérer et afficher les données
- Ajouter une fonction `supprimer_utilisateur()` pour supprimer une ligne

# Lier Tkinter et SQLite

```
def afficher_utilisateurs():
    conn = sqlite3.connect("utilisateurs.db")
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM
utilisateurs")
    rows = cursor.fetchall()
    conn.close()

    listbox.delete(0, tk.END)
    for row in rows:
        listbox.insert(tk.END, row)
```

```
def ajouter_utilisateur():
    nom = entry_nom.get()
    age = entry_age.get()

    conn = sqlite3.connect("utilisateurs.db")
    cursor = conn.cursor()
    cursor.execute("INSERT INTO utilisateurs (nom,
age) VALUES (?, ?)", (nom, age))
    conn.commit()
    conn.close()
    afficher_utilisateurs()
```

```
def supprimer_utilisateur():
    selection = listbox.curselection()
    if not selection:
        return
    id_utilisateur = listbox.get(selection)[0]
    conn = sqlite3.connect("utilisateurs.db")
    cursor = conn.cursor()
    cursor.execute("DELETE FROM utilisateurs
WHERE id = ?", (id_utilisateur,))
    conn.commit()
    conn.close()
  afficher_utilisateurs()
```

# Lier Tkinter et SQLite

## Mettre à jour l’interface après chaque action

Après chaque action, on met à jour l'affichage pour que les nouvelles données soient visibles immédiatement.

**Étapes :**

- Appeler `afficher_utilisateurs()` après chaque ajout ou suppression
- Associer les boutons aux fonctions

# TP 17

# Flask

# Flask


**Flask** est un micro-framework léger pour Python qui permet de créer des applications web facilement. Il est très populaire pour le développement d'applications web, notamment pour des projets de petite et moyenne envergure, mais peut aussi être étendu pour des applications complexes grâce à ses nombreuses extensions.

**Flask** est souvent choisi en raison de sa simplicité, sa flexibilité et son faible encombrement. Il ne prend pas de décisions à votre place, ce qui vous permet de choisir les composants que vous souhaitez utiliser.

# Flask

## Pourquoi Flask est-il populaire ?

**Simplicité** : Flask est minimaliste, vous n'avez pas besoin d'une grosse configuration pour démarrer.
**Extensible** : Vous pouvez ajouter des bibliothèques et des outils pour ajouter des fonctionnalités spécifiques (authentification, bases de données, etc.).
**Flexible** : Flask ne vous impose pas de structure stricte pour vos projets. Vous pouvez organiser votre code comme vous le souhaitez.
**Bonne documentation** : Flask est bien documenté et dispose d'une grande communauté qui propose des ressources et des extensions.

# Flask

## Installation de Flask

Si vous n'avez pas Flask sur votre machine, vous pouvez l'installer facilement en utilisant **pip** :

```
pip install flask
```

# Flask

## Créer une Application Flask Simple

Créez un fichier Python, par exemple app.py, et ajoutez le code suivant :

```
from flask import Flask

# Création de l'application Flask
app = Flask(__name__)

# Définir une route pour la page d'accueil
@app.route('/')
def hello_world():
    return 'Hello, World!'

# Lancer l'application
if __name__ == '__main__':
    app.run(debug=True)
```

# Flask

## Affichage d'un Template HTML

Flask permet d'utiliser des templates pour générer dynamiquement du contenu HTML.

Voici un exemple où nous affichons un fichier HTML en utilisant un template.

Créez un fichier HTML **templates/index.html** :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Flask Exemple</title>
</head>
<body>
    <h1>Bienvenue, {{ name }}!</h1>
    <h2>Hello, {{ name }}!</h2>
</body>
</html>
```

# Flask

## Affichage d'un Template HTML

Modifiez votre code Flask dans **app**.py :

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def hello_world():
    return render_template('index.html', name='Alex')

if __name__ == '__main__':
    app.run(debug=True)
```

Lorsque vous exécutez votre application Flask et allez à l'URL **http://127.0.0.1:5000/**, vous verrez une page HTML qui dit "Bienvenue, Étudiant!".

# Flask

Flask peut aussi gérer des formulaires HTML pour recevoir des données de l'utilisateur. Voici un exemple simple d'une page de connexion.

## Recevoir des Données d'un Formulaire HTML

```
from flask import Flask, request, render_template_string

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        # Exemple simple de vérification
        if username == "admin" and password == "password123":
            return f"Connexion réussie, {username}!"
        else:
            return "Nom d'utilisateur ou mot de passe incorrect"
    return '''
        <form method="POST">
            <input type="text" name="username" placeholder="Nom
d'utilisateur"><br>
    <input type="password" name="password" placeholder="Mot de
passe"><br>
            <input type="submit" value="Se connecter">
        </form> '''
if __name__ == '__main__':
    app.run(debug=True)
```

# TP 15

# Flask

**Flask** offre une intégration facile avec les bases de données relationnelles grâce à Flask-SQLAlchemy, qui est une extension de SQLAlchemy, un **ORM** (Object Relational Mapper).

**SQLAlchemy** permet d'interagir avec une base de données en Python au lieu d'écrire du SQL brut, ce qui rend le code plus structuré et portable.

# Flask

## Qu'est-ce que SQLAlchemy ?

SQLAlchemy est une bibliothèque Python qui permet de gérer une base de données **avec des objets Python** au lieu d'écrire directement des requêtes SQL.

- Il offre une **abstraction du SQL**, facilitant la migration entre différentes bases de données (SQLite, MySQL, PostgreSQL...).
- Il gère les **relations entre tables**, **transactions**, **migrations**, et **optimisation des requêtes**.

# Flask

**Qu'est-ce que SQLite ?**

**SQLite** est une base de données légère et intégrée, stockée dans un fichier .db.

Aucune installation de serveur n'est requise.

Idéale pour les petits projets ou le développement local.

Flask peut utiliser **SQLAlchemy** avec **SQLite** pour bénéficier des avantages des deux :

✔️ Simplicité (SQLite ne nécessite pas de configuration)

✔️ Facilité d'utilisation (SQLAlchemy rend les requêtes plus lisibles)

# Flask

**Configuration de Flask avec SQLAlchemy et SQLite**

Installation des dépendances :

pip install flask flask-sqlalchemy flask-migrate

flask-sqlalchemy : Intègre SQLAlchemy à Flask.

flask-migrate : Permet de gérer les migrations de base de données avec Alembic.

# Flask

**id** → Clé primaire auto-incrémentée.
**name** → Nom obligatoire, type VARCHAR(100).
**email** → Adresse e-mail unique.

**Définition d'un modèle SQLAlchemy**

Avec **SQLAlchemy**, une table est définie comme une **classe** Python (modèle).

**Exemple de modèle User :**

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
# Clé primaire
 name = db.Column(db.String(100), nullable=False)
# Champ obligatoire
    email = db.Column(db.String(100), unique=True, nullable=False)  # Unique
    def __repr__(self):
        return f"<User {self.name}>"
```

# Flask

**Création de la base de données et des tables**

Ajoutez ce code à la fin de app.py :

```
with app.app_context():
db.create_all()
```

Puis exécutez :

```
python app.py
```

Cela génère automatiquement le fichier **database.db** et la table **User**.

# Flask

**Ajout, récupération et suppression de données**

- **Ajouter un utilisateur :**

```
@app.route('/add_user', methods=['POST'])

def add_user():
    data = request.get_json()
# Récupérer les données envoyées

    new_user = User(name=data['name'], email=data['email'])
# Créer un objet User
    db.session.add(new_user)
 # Ajouter l'utilisateur à la session

    db.session.commit()
 # Enregistrer dans la base
    return jsonify({"message": "Utilisateur ajouté"}), 201
```

# Flask

- **Récupérer tous les utilisateurs**

```
@app.route('/users', methods=['GET'])
def get_users():
    users = User.query.all()
# Récupère tous les utilisateurs
    return jsonify(
[  {"id": user.id, "name": user.name, "email": user.email}
        for user in users
]
        )
```

# Flask

- **Supprimer un utilisateur**

```
@app.route('/delete_user/<int:id>', methods=['DELETE'])
def delete_user(id):
    user = User.query.get_or_404(id)
        # Trouver l'utilisateur par ID
    db.session.delete(user)
        # Supprimer de la session
    db.session.commit()
        # Appliquer la suppression
    return jsonify({"message": "Utilisateur supprimé"})
```

# Flask

## Gestion des Migrations (Flask-Migrate)

Si vous ajoutez de nouveaux champs, il faut migrer la base pour éviter de la recréer à chaque modification.

- **Installation** :

```
pip install flask-migrate
```

- **Initialisation** : Ajoutez ceci dans app.py :

```
from flask_migrate import Migrate
migrate = Migrate(app, db)
```

- **Exécuter les migrations :**

```
flask db init
flask db migrate -m "Initial migration"
flask db upgrade
```

# Flask


**Tester l'API avec Postman**

- **Ajouter un utilisateur :**

```
POST http://127.0.0.1:5000/add_user
Content-Type: application/json

{
  "name": "Nizar",
  "email": "nizar@example.com"
}
```

- **Obtenir tous les utilisateurs :**

```
GET http://127.0.0.1:5000/users
```

- **Supprimer un utilisateur :**

```
DELETE http://127.0.0.1:5000/delete_user/1
```

# TP 16

Quiz :

https://quizizz.com/admin/quiz/67b311542227c748d90e3745

# PyGame

# Installation de Pygame

Assurez-vous d'installer Pygame en utilisant la commande suivante :

**pip install pygame**

# Installation de Pygame

**Première fenêtre Pygame**

Créez un fichier **jeu.py** et ajoutez le code suivant pour créer une fenêtre Pygame simple :

```
import pygame
import sys
# Initialisation de Pygame
pygame.init()
# Dimensions de la fenêtre
largeur, hauteur = 800, 600
# Création de la fenêtre
fenetre = pygame.display.set_mode((largeur, hauteur))
pygame.display.set_caption("Mon Premier Jeu Pygame")

# Boucle principale du jeu
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
    # Effacement de l'écran
    fenetre.fill((255, 255, 255))  # Fond blanc
    # Mise à jour de l'écran
    pygame.display.flip()
```

# Installation de Pygame

Dans ce code, nous avons créé une **fenêtre** Pygame simple avec une boucle principale qui surveille les événements, notamment l'événement QUIT (fermeture de la fenêtre).
La fenêtre est effacée à chaque itération de la boucle et l'écran est mis à jour avec **pygame.display.flip()**.

# Installation de Pygame

**Dessiner un rectangle**

Ajoutez un rectangle à votre fenêtre :

```
import pygame
import sys


pygame.init()
largeur, hauteur = 800, 600
fenetre = pygame.display.set_mode((largeur, hauteur))
pygame.display.set_caption("Mon Premier Jeu Pygame")
# Position du rectangle
x, y = 50, 50
largeur_rectangle, hauteur_rectangle = 100, 50
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
    fenetre.fill((255, 255, 255))
    # Dessin du rectangle
    pygame.draw.rect(fenetre, (0, 0, 255), (x, y, largeur_rectangle,
hauteur_rectangle))
    pygame.display.flip()
```

# Installation de Pygame

Ici, nous avons ajouté un rectangle blanc à la fenêtre en utilisant **pygame.draw.rect()**. Vous pouvez ajuster les coordonnées (x, y) et les dimensions (largeur_rectangle, hauteur_rectangle) selon vos besoins.

# Installation de Pygame

**Mouvement du rectangle**

Ajoutons un mouvement simple du rectangle avec les touches fléchées :

```
import pygame
import sys
pygame.init()
largeur, hauteur = 800, 600
fenetre = pygame.display.set_mode((largeur, hauteur))
pygame.display.set_caption("Mon Premier Jeu Pygame")
x, y = 50, 50
largeur_rectangle, hauteur_rectangle = 100, 50
vitesse = 5
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
    keys = pygame.key.get_pressed()
    if keys[pygame.K_LEFT] and x > 0:
        x -= vitesse
    if keys[pygame.K_RIGHT] and x < largeur - largeur_rectangle:
        x += vitesse
    if keys[pygame.K_UP] and y > 0:
        y -= vitesse
    if keys[pygame.K_DOWN] and y < hauteur - hauteur_rectangle:
```

# Installation de Pygame

Dans ce code, nous avons ajouté un mouvement horizontal et vertical au rectangle en fonction des touches fléchées.
La vitesse du mouvement est contrôlée par la variable **vitesse**.

Ceci est un début simple pour comprendre comment utiliser Pygame.
Vous pouvez explorer davantage en ajoutant des fonctionnalités telles que la gestion des collisions, la création d'images, la gestion des événements de la souris, etc.
Pygame propose de nombreux tutoriels et ressources pour approfondir vos connaissances.

# TP 14

# Fin

**Merci !**