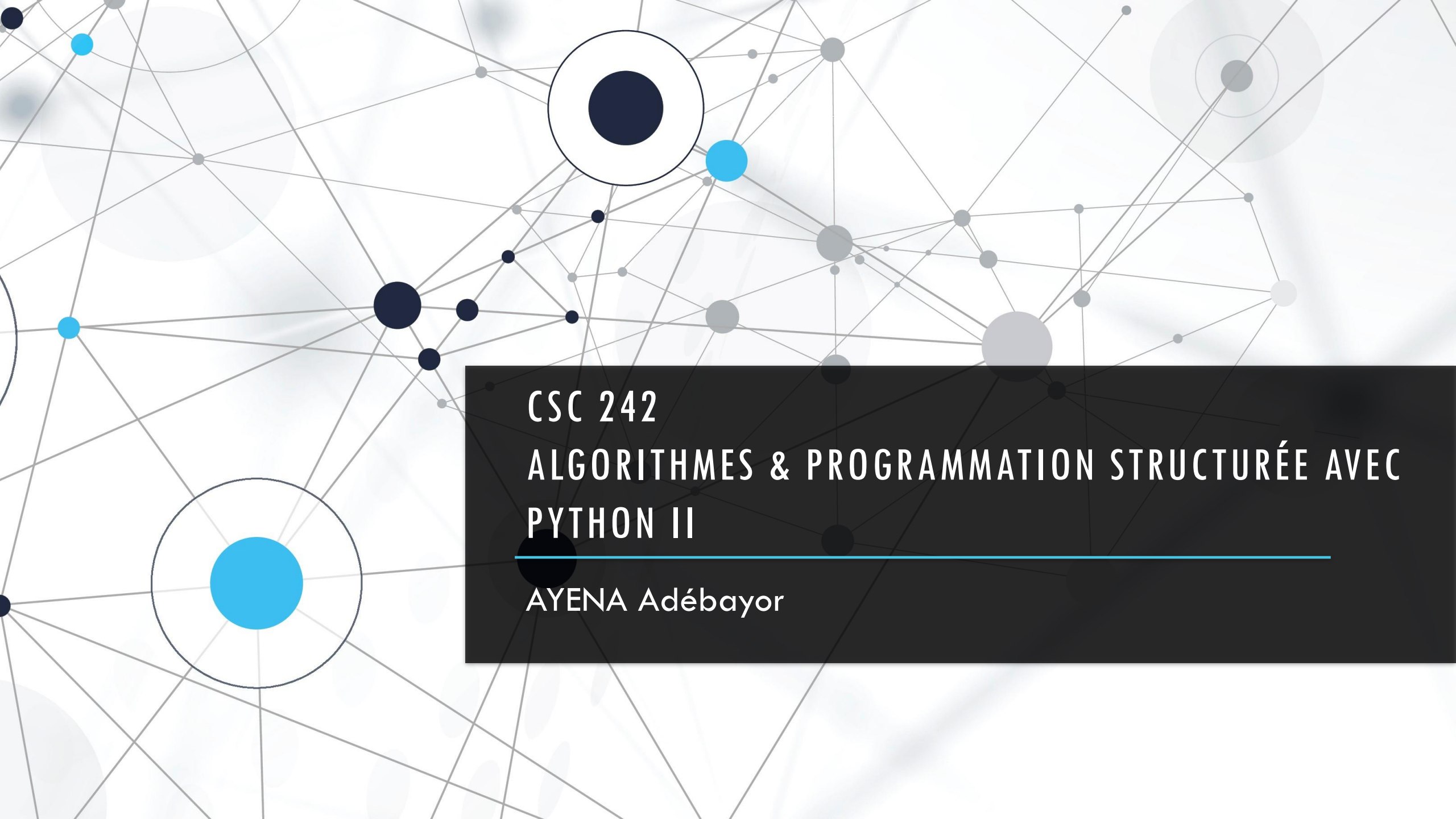


CSC 242
ALGORITHMES & PROGRAMMATION STRUCTURÉE AVEC PYTHON II

AYENA Adébayor

QUI SUIS – JE ?

AYENA Adébayor (adebayor.ayena@ipnetinstitute.com / bayorayena@gmail.com / 00228 92 07 79 45)

- ✓ Première année de Licence à la Faculté des Sciences (FDS) de l'Université de Lomé, parcours *Mathématiques* (2013 – 2014) ;
- ✓ Licence Professionnelle au Centre Informatique et de Calcul (CIC) de l'Université de Lomé, Option *Génie Logiciel* (2014 - 2017) ;

QUI SUIS – JE ?

- ✓ Oracle Certified Associate, Java SE 8 Programmer (Juin 2019) ;
- ✓ Oracle Certified Professional, Java SE 8 Programmer (Septembre 2019) ;
- ✓ Oracle Database SQL Certified Associate (Avril 2021).

i. DESCRIPTION ET OBJECTIFS DU COURS

- Un cours intermédiaire en résolution de problèmes, algorithmes et programmation.
- Les compétences en programmation sont renforcées par des tâches de programmation plus complexes et plus vastes.
- Les affectations serviront également à présenter différents domaines d'informatique (par exemple, une application client/serveur pour le domaine des systèmes distribués).

i. DESCRIPTION ET OBJECTIFS DU COURS

- Les classes et la programmation orientée objet sont motivées et introduites.

ii. CONCEPTS TECHNOLOGIQUES COUVERTS PAR LE COURS

- Examen des espaces de noms,
- Scope ;
- Classes
- Programmation orientée objet
- Interfaces graphiques
- Récursivité

ii. CONCEPTS TECHNOLOGIQUES COUVERTS PAR LE COURS

- Recherche
- Tri
- Développement d'applications Internet
- Development de base de données
- Python contre C ++ / Java

iii. PRÉ-REQUIS

- CONDITIONS PRÉALABLES : CSC 241

iv. RÉVISIONS

- Ce chapitre sera consacré aux notions de base couvertes au cours de CSC 241.
- On s'attardera entre autres sur :
  - La définition de l'algorithme, d'un langage de programmation, d'un programme informatique, etc.
  - Les variables ;

iv. RÉVISIONS

- Les structures conditionnelles ;
- Les structures répétitives (boucles) ;
- Les fonctions ;
- Les collections de données ;
- Etc.

CHAP. 1 : PORTÉE DES VARIABLES ET RÉFÉRENCES

1.1. LA PORTÉE DES VARIABLES

- En Python, comme dans la plupart des langages, on trouve des règles qui dénissent la portée des variables.
- La portée utilisée dans ce sens c'est quand et comment les variables sont-elles accessibles ?
- Quand vous définissez une fonction, quelles variables sont utilisables dans son corps ?

1.1. LA PORTÉE DES VARIABLES

- Uniquement les paramètres de la fonction ?
- Est-ce qu'on peut créer dans notre corps de fonction des variables utilisables en dehors ?
- Si vous ne vous êtes jamais posé ces questions, c'est normal.
- Nous allons nous attarder sur ce points dans cette section.

1.1.1. ACCÈS AUX VARIABLES DANS LES FONCTIONS

- Analysons le code de l'exemple suivant :

```
>>> a = 5
>>> def print_a():
...     """Fonction chargée d'afficher la variable a.
...     Cette variable a n'est pas passée en paramètre de la fonction.
...     On suppose qu'elle a été créée en dehors de la fonction, on veut voir
...     si elle est accessible depuis le corps de la fonction
    """
...
...     print("La variable a = {0}.".format(a))
...
>>> print_a()
La variable a = 5.
>>> a = 8
>>> print_a()
La variable a = 8.
>>>
```

1.1.1. ACCÈS AUX VARIABLES DANS LES FONCTIONS

- La variable « ***a*** » n'est pas passée en paramètre de la fonction « ***print_a*** ».
- Et pourtant, Python la trouve, tant qu'elle a été définie avant l'***appel*** de la fonction.
- C'est là qu'interviennent les différents espaces.

1.1.2. L'ESPACE LOCAL

- Dans notre fonction, quand on fait référence à une variable « ***a*** », Python vérifie dans l'espace local de la fonction.
- Cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps.
- Python apprend ainsi que la variable « ***a*** » n'existe pas dans l'espace local de la fonction.

1.1.2. L'ESPACE LOCAL

- Dans ce cas, il va regarder dans l'espace local dans lequel la fonction a été appelée.
- Et là, il trouve bien la variable « ***a*** » et peut donc l'afficher.

1.1.2. L'ESPACE LOCAL

- D'une façon générale, il est conseillé d'éviter d'appeler des variables qui ne sont pas dans l'espace local, sauf si c'est nécessaire.
- Ce n'est pas très clair à la lecture ; dans l'absolu, préférez travailler sur des variables globales, cela reste plus propre (nous verrons cela plus bas).
- Pour l'instant, on ne s'intéresse qu'aux mécanismes, on cherche juste à savoir quelles variables sont accessibles depuis le corps d'une fonction et de quelle façon.

1.1.3. LA PORTÉE DE NOS VARIABLES

- Nous allons voir et expliquer quelques cas concrets :

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

- Dans cet exemple, on essaie :
  - d'accéder à une variable qui n'est pas encore définie dans une fonction ;
  - Ensuite d'accéder à une variable définie dans une fonction en dehors de celle-ci.

```
1  def set_var(nouvelle_valeur):
2      """Fonction nous permettant de tester la portée des
           variables
3      définies dans notre corps de fonction"""
4
5      # On essaye d'afficher la variable var, si elle existe
6      try:
7          print("Avant l'affectation, notre variable var vaut {0}
               .".format(var))
8      except NameError:
9          print("La variable var n'existe pas encore.")
10     var = nouvelle_valeur
11     print("Après l'affectation, notre variable var vaut {0}.".
           format(var))
```

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

- Et maintenant, utilisons notre fonction :

```
>>> set_var(5)
La variable var n'existe pas encore.
Après l'affectation, notre variable var vaut 5.
>>> var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
>>>
```

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

- Explications :

1. Lors de notre appel à « ***set_var*** », notre variable var n'a pu être trouvée par Python : c'est normal, nous ne l'avons pas encore définie, ni dans le corps de la fonction, ni dans le corps de notre programme. Python affecte ensuite la valeur 5 à la variable « ***var*** », l'affiche et s'arrête.

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

2. Au sortir de la fonction, on essaye d'afficher la variable « ***var*** »... mais Python ne la trouve pas ! En effet : elle a été définie dans le corps de la fonction (donc dans son espace local) et, à la fin de l'exécution de la fonction, l'espace est détruit... donc la variable « ***var*** », définie dans le corps de la fonction, n'existe que dans ce corps et est détruite ensuite.

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

- Python a une règle d'accès spécifique aux variables extérieures à l'espace local : ***on peut les lire, mais pas les modifier***.
- C'est pourquoi, dans notre fonction « ***print_a*** », on arrivait à afficher une variable qui n'était pas comprise dans l'espace local de la fonction.
- En revanche, on ne peut modifier la valeur d'une variable extérieure à l'espace local, par affectation du moins.

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

- Si dans votre corps de fonction vous faites « ***var = nouvelle_valeur*** », vous n'allez en aucun cas modifier une variable extérieure au corps.
- En fait, quand Python trouve une instruction d'affectation, comme par exemple « ***var = nouvelle_valeur*** », il va changer la valeur de la variable dans l'espace local de la fonction.
- Et rappelez-vous que cet espace local est détruit après l'appel à la fonction.

1.1.3.1. VARIABLES DÉFINIES DANS LE CORPS D'UNE FONCTION

- Pour résumer, et c'est ce qu'il faut retenir, ***une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.***

1.1.3.2. UNE FONCTION MODIFIANT DES OBJETS

- Quand vous passez des paramètres à votre fonction, ce sont des objets qui sont transmis.
- Et pas les valeurs des objets, mais bien les objets eux-mêmes, ceci est très important.
- Bon. On ne peut affecter une nouvelle valeur à un paramètre dans le corps de la fonction.

1.1.3.2. UNE FONCTION MODIFIANT DES OBJETS

- En revanche, on pourrait essayer d'appeler une méthode de l'objet qui le modifie. . .
- Voyons cela :

```
>>> def ajouter(liste, valeur_a_ajouter):
...     """Cette fonction insère à la fin de la liste la valeur
     que l'on veut ajouter"""
...     liste.append(valeur_a_ajouter)
...
>>> ma_liste=['a', 'e', 'i']
>>> ajouter(ma_liste, 'o')
>>> ma_liste
['a', 'e', 'i', 'o']
>>>
```

1.1.3.2. UNE FONCTION MODIFIANT DES OBJETS

- Cela marche !
- On passe en paramètres notre objet de type ***list*** avec la valeur à ajouter.
- Et la fonction appelle la méthode ***append*** de l'objet.
- Cette fois, au sortir de la fonction, notre objet a bel et bien été modifié.

1.1.3.2. UNE FONCTION MODIFIANT DES OBJETS

- **Mais une fonction ne pouvait pas affecter de nouvelles valeurs aux paramètres ?**
- Absolument.
- Mais c'est cela la petite subtilité dans l'histoire : ***on ne change pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet.***
- Et cela change tout.

1.1.3.2. UNE FONCTION MODIFIANT DES OBJETS

- Si vous vous embrouillez, retenez que, dans le corps de fonction, si vous faites ***parametre = nouvelle_valeur***, le paramètre ne sera modifié que dans le corps de la fonction.

- Alors que si vous faites ***parametre.methode_pour_modifier(...)***, l'objet derrière le paramètre sera bel et bien modifié.

1.1.3.2. UNE FONCTION MODIFIANT DES OBJETS

- On peut aussi modifier les attributs d'un objet, par exemple changer une case de la liste ou d'un dictionnaire : ces changements aussi seront effectifs au-delà de l'appel de la fonction.

1.2. LES RÉFÉRENCES

- Nous allons parler des ***références des objets*** dans cette section.
- Les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.
- Concrètement, j'ai présenté les variables comme ceci : un nom identifiant pointant vers une valeur.
- Par exemple, notre variable nommée « ***a*** » possède une valeur (disons 0).

1.2. LES RÉFÉRENCES

- En fait, une variable est un nom identifiant, pointant vers une référence d'un objet.
- La référence, c'est un peu sa position en mémoire.
- Cela signifie que deux variables peuvent pointer sur le même objet.
- Nous parlons d'objets ici, pas de valeurs. Il y a une différence.
- Voyons un exemple, vous allez comprendre :

```
1 >>> ma_liste1 = [1, 2, 3]
2 >>> ma_liste2 = ma_liste1
3 >>> ma_liste2.append(4)
4 >>> print(ma_liste2)
5 [1, 2, 3, 4]
6 >>> print(ma_liste1)
7 [1, 2, 3, 4]
8 >>>
```

1.2. LES RÉFÉRENCES

- Nous créons une liste dans la variable ***ma_liste1***.
- À la ligne 2, nous affectons ***ma_liste1*** à la variable ***ma_liste2***.
- On pourrait croire que ***ma_liste2*** est une copie de ***ma_liste1***.
- Toutefois, quand on ajoute 4 à ***ma_liste2***, ***ma_liste1*** est aussi modifiée.

1.2. LES RÉFÉRENCES

- On dit que ***ma_liste1*** et ***ma_liste2*** contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.
- Vous ne pouvez pas dans ce cas modifier une liste sans toucher à l'autre.
- Eh bien c'est impossible, vu la manière dont nous avons défini nos listes.

1.2. LES RÉFÉRENCES

- Les deux variables pointent sur le même objet par jeu de références et donc, inévitablement, si vous modifiez l'objet, vous allez voir le changement depuis les deux variables.

- Toutefois, il existe un moyen pour créer un nouvel objet depuis un autre :

```
1 >>> ma_liste1 = [1, 2, 3]
2 >>> ma_liste2 = list(ma_liste1) # Cela revient à copier le
      contenu de ma_liste1
3 >>> ma_liste2.append(4)
4 >>> print(ma_liste2)
5 [1, 2, 3, 4]
6 >>> print(ma_liste1)
7 [1, 2, 3]
8 >>>
```

1.2. LES RÉFÉRENCES

- À la ligne 2, nous avons demandé à Python de créer un nouvel objet basé sur ***ma_liste1***.
- Du coup, les deux variables ne contiennent plus la même référence : elles modifient des objets différents.
- Vous pouvez utiliser la plupart des constructeurs (c'est le nom qu'on donne à ***list*** pour créer une liste par exemple) dans ce but.

1.2. LES RÉFÉRENCES

- Pour des dictionnaires, utilisez le constructeur ***dict*** en lui passant en paramètre un dictionnaire déjà construit et vous aurez en retour un dictionnaire, semblable à celui passé en paramètre, mais seulement semblable par le contenu.
- En fait, il s'agit d'une copie de l'objet, ni plus ni moins.

1.2. LES RÉFÉRENCES

- Pour approcher de plus près les références, vous avez la fonction ***id*** qui prend en paramètre un objet.
- Elle renvoie la position de l'objet dans la mémoire Python sous la forme d'un entier (plutôt grand).
- Je vous invite à faire quelques tests en passant divers objets en paramètre à cette fonction.

1.2. LES RÉFÉRENCES

- Sachez au passage que ***is*** compare les ID des objets de part et d'autre et == compare les contenus des listes :

```
>>> ma_liste1 = [1, 2]
>>> ma_liste2 = [1, 2]
>>> ma_liste1 == ma_liste2 # On compare le contenu des listes
True
>>> ma_liste1 is ma_liste2 # On compare leur référence
False
>>>
```

1.2. LES RÉFÉRENCES

- **Si vous essayez de faire la même chose avec des variables contenant des entiers, cela ne marche pas.**
- Les ***entiers (int)***, les ***flottants (float)***, les ***chaînes de caractères (str)***, n'ont aucune méthode travaillant sur l'objet lui-même.
- Les chaînes de caractères, comme nous l'avons vu, ne modifient pas l'objet appelant mais renvoient un nouvel objet modifié.

1.2. LES RÉFÉRENCES

- Et comme nous venons de le voir, le processus d’affectation n'est pas du tout identique à un appel de méthode.

1.3. LES VARIABLES GLOBALES

- Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci.
- On utilise pour cela des variables globales.
- Cette distinction entre variables locales et variables globales se retrouve dans d'autres langages et on recommande souvent d'éviter de trop les utiliser.
- Elles peuvent avoir leur utilité, toutefois, puisque le mécanisme existe.

1.3.1. LE PRINCIPE DES VARIABLES GLOBALES

- On ne peut faire plus simple.
- On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal.
- Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

1.3.1. LE PRINCIPE DES VARIABLES GLOBALES

- Python va regarder dans les différents espaces : celui de la fonction, celui dans lequel la fonction a été appelée. . . ainsi de suite jusqu'à mettre la main sur notre variable.
- S'il la trouve, il va nous donner le plein accès à cette variable dans le corps de la fonction.

1.3.1. LE PRINCIPE DES VARIABLES GLOBALES

- Cela signifie que nous pouvons y accéder en lecture (comme c'est le cas sans avoir besoin de la définir comme variable globale) mais aussi en écriture.
- Une fonction peut donc ainsi changer la valeur d'une variable directement.
- Mais assez de théorie, voyons comment déclarer et utiliser une variable globale.

1.3.2. UTILISATION DES VARIABLES GLOBALES

- Pour déclarer à Python, dans le corps d'une fonction, que la variable qui sera utilisée doit être considérée comme globale, on utilise le mot-clé ***global***.
- On le place généralement après la définition de la fonction, juste en-dessous de la ***docstring***, cela permet de retrouver rapidement les variables globales sans parcourir tout le code (c'est une simple convention).
- On précise derrière ce mot-clé le nom de la variable à considérer comme globale :

```
>>> i = 4 # Une variable, nommée i, contenant un entier
>>> def inc_i():
...     """Fonction chargée d'incrémenter i de 1"""
...     global i # Python recherche i en dehors de l'espace
    local de la fonction
...     i += 1
...
>>> i
4
>>> inc_i()
>>> i
5
>>>
```

1.3.2. UTILISATION DES VARIABLES GLOBALES

- Si vous ne précisez pas à Python que « *i* » doit être considérée comme globale, vous ne pourrez pas modifier réellement sa valeur, comme nous l'avons vu plus haut.

- En précisant global « *i* » , Python permet l'accès en lecture et en écriture à cette variable, ce qui signifie que vous pouvez changer sa valeur par affectation.

1.4. EN RÉSUMÉ

- Les variables locales définies avant l'appel d'une fonction seront accessibles, depuis le corps de la fonction, en lecture seule.
- Une variable locale définie dans une fonction sera supprimée après l'exécution de cette fonction.
- On peut cependant appeler les attributs et méthodes d'un objet pour le modifier durablement.

1.4. EN RÉSUMÉ

- Les variables globales se définissent à l'aide du mot-clé ***global*** suivi du nom de la variable préalablement créée.
- Les variables globales peuvent être modifiées depuis le corps d'une fonction (à utiliser avec prudence).

PARTIE 2 : LA PROGRAMMATION ORIENTÉE OBJET

CHAP. 2 : PREMIÈRE APPROCHE DE LA PROGRAMMATION ORIENTÉE OBJET

- Dans ce chapitre, nous allons essayer de comprendre les mécanismes de la Programmation Orientée Objet (POO).
- Nous allons aussi sans plus attendre, créer nos premières classes, nos premiers attributs et nos premières méthodes.

2.1. DÉFINITION

- La programmation orientée objet est une technique de programmation célèbre qui existe depuis des années maintenant.
- La programmation orientée objet, que nous abrègerons POO va vous aider à mieux organiser votre code, à le préparer à de futures évolutions et à rendre certaines portions réutilisables pour gagner en temps et en clarté.
- C'est pour cela que les développeurs professionnels l'utilisent dans la plupart de leurs projets.

2.2. LES CLASSES ET LES OBJETS

2.2.1. DÉFINITION DES CLASSES ET DES OBJETS

- Il y a beaucoup de définitions très différentes mais nous allons voir des définitions très simples d'une classe et d'un objet :
  - Une ***classe*** est une forme de type de donnée, elle permet de définir des fonctions et variables propres au type.
  - Un ***objet*** est une structure de données, une variable, qui peut contenir elle-même d'autres variables et fonctions.

2.2.1. DÉFINITION DES CLASSES ET DES OBJETS

- Les ***objets***, présentés comme des variables, pouvant contenir d'autres variables ou fonctions (que l'on appelle méthodes). On appelle une méthode d'un objet à partir de l'instruction ***objet.methode()***.

- Les ***classes***, présentées comme des types de données. Une classe est un modèle qui servira à construire un objet ; c'est dans la classe qu'on va déclarer les variables et définir les méthodes propres à l'objet.

2.2.2. CHOIX DU MODÈLE D'UNE CLASSE

- Nous venons de définir, une classe comme un modèle suivant lequel on va créer des objets.
- C'est dans la classe que nous allons définir nos ***méthodes*** et ***attributs***, les attributs étant des variables contenues dans notre objet.
- Mais qu'allons-nous modéliser ?

2.2.2. CHOIX DU MODÈLE D'UNE CLASSE

- L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter des données un peu plus complexes qu'un simple nombre, ou qu'une chaîne de caractères.
- Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie.
- Mais on serait bien limité si on ne pouvait pas créer ses propres classes.

2.2.2. CHOIX DU MODÈLE D'UNE CLASSE

- Ainsi pour chaque classe à créer, il faut définir son modèle c'est-à-dire la liste des attributs et des méthodes de cette classe.
- Pour l'instant, nous allons modéliser. . . une ***personne***.

2.2.2. CHOIX DU MODÈLE D'UNE CLASSE

- La définition d'une classe est composée de deux parties :
  - ***Les attributs*** **:** ce sont les propriétés ou caractéristiques de l'objet, c'est-à-dire les différentes variables qui définissent la classe, son état.
  - ***Les méthodes*** **:** elles définissent les actions possibles qu'on peut effectuer sur la classe, ses comportements.
- Ces deux (2) parties sont appelées les ***membres*** de la classe.

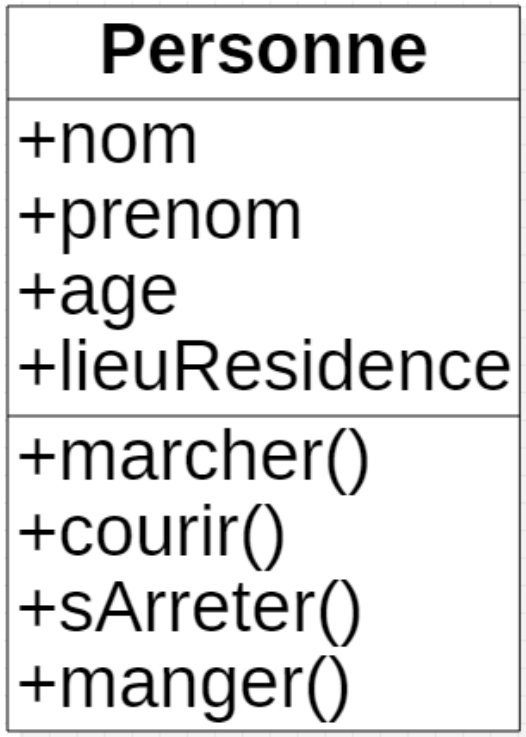
2.2.2. CHOIX DU MODÈLE D'UNE CLASSE

- Une personne peut être caractérisée par : son ***nom***, son ***prénom***, son ***âge***, son ***lieu de résidence***, etc. Ces différentes caractéristiques constituent les attributs de la classe ***Personne***.

- Les actions qu'on peut effectuer à partir d'une personne sont : ***marcher***, ***courir***, ***s'arrêter***, ***manger***, etc. Ces différentes actions constituent donc les méthodes qu'on peut implémenter dans la classe ***Personne***.

2.2.2. CHOIX DU MODÈLE D'UNE CLASSE

- Le modèle de la classe Personne peut être représentée à la figure ci-dessous :



2.3. DÉFINITION D'UNE CLASSE DE BASE EN LANGAGE ALGORITHMIQUE

2.3.1. DÉCLARATION D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Pour déclarer une classe en langage algorithmique, on utilise le mot clé ***« Classe »***.
- La définition des classes en langage algorithmique se fait en dehors des algorithmes principaux.
- On précède donc la déclaration des classes par le mot clé ***« Type »***.
- Illustration à la figure suivante :

```
Type
    Classe NomDeLaClasse1

        /* Définition du corps de la Classe1 (attributs + méthodes) */

    FinClasse

    Classe NomDeLaClasse2

        /* Définition du corps de la Classe2 (attributs + méthodes) */

    FinClasse

    ...
```

2.3.1. DÉCLARATION D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- En prenant l'exemple de la classe Personne, la syntaxe de déclaration en langage algorithmique est :

```
Type
        Classe Personne

                /* Définition du corps de la Personne (attributs + méthodes) */

        FinClasse
```

2.3.1. DÉCLARATION D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Nous allons maintenant passer à la déclarer des membres de la classe (attributs et méthodes).

2.3.2. DÉCLARATION DES ATTRIBUTS ET MÉTHODES D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- La déclaration des attributs et méthodes d'une classe se fait dans le corps de la classe.

- En langage algorithmique, pour déclarer :
  - Les attributs d'une classe, on utilise le mot clé « ***attributs*** ».
  - Les méthodes, on utilise le mot clé « ***méthodes*** ».

2.3.2. DÉCLARATION DES ATTRIBUTS ET MÉTHODES D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Il existe deux (2) types d'attributs dans une classe : les ***attributs d'instances*** et les ***attributs de classes.***
- Les ***attributs d'instances*** sont propres à chaque objet de la classe et les ***attributs de classes*** sont communs à tous les objets de la classe.
- Syntaxes pour déclarer les attributs :

```
Type
    Classe NomDeLaClasse1
        attributs
            attribut1 : type
            attribut2 : type
            ...
        méthodes
            Procédure nomDeLaProcedure(paramètres)
            Fonction nomDeLaFonction(paramètres) : typeDeRetour
            ...
    FinClasse
```

2.3.2. DÉCLARATION DES ATTRIBUTS ET MÉTHODES D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Appliquons cela à la classe ***Personne*** :

```
Type
    Classe Personne
        attributs
            nom : chaîne de caractères
            prenom : chaîne de caractères
            age : entier
            lieuDeResidence : chaîne de caractères
        méthodes
            Procédure marcher()
            Procédure courir()
            Procédure sArreter()
            Procédure manger()
    FinClasse
```

2.3.2. DÉCLARATION DES ATTRIBUTS ET MÉTHODES D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Pour le moment, nous allons déclarer toutes les méthodes de la classe Personne comme des « ***procédures*** » et nous n'allons pas encore implémenter leur corps.

2.3.2. DÉCLARATION DES ATTRIBUTS ET MÉTHODES D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Pour déclarer un attribut statique en langage algorithmique, il faut ajouter le mot clé **« statique »** devant la déclaration de la variable.

- Syntaxe :

Type

Classe Personne

attributs

nom : chaîne de caractères

prenom : chaîne de caractères

age : entier

lieuDeResidence : chaîne de caractères

nombrePersonnesCrees : entier statique

méthodes

Procédure marcher()

Procédure courir()

Procédure sArreter()

Procédure manger()

FinClasse

2.3.3. CRÉATION DES CONSTRUCTEURS D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Quand vous instanciez une classe, c'est-à-dire quand vous créez un objet du type de la classe données, il faut bien souvent appeler diverses méthodes pour initialiser les attributs avec les bonnes valeurs.

- Il existe un moyen plus efficace et implicite d'initialiser les bonnes valeurs des attributs (er d'effectuer toute autre opération nécessaire) dès l'instanciation ou la déclaration de l'objet.

2.3.3. CRÉATION DES CONSTRUCTEURS D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Le constructeur est une méthode particulière qui est appelée automatiquement dès vous créez un objet sans que vous ayez à le préciser.

- Dans cette méthode, libre à vous de faire ce que vous voulez, mais dans un grand nombre de cas, son rôle sera de donner aux attributs leurs bonnes valeurs initiales, ou des valeurs par défaut.

2.3.3. CRÉATION DES CONSTRUCTEURS D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Un constructeur doit respecter les deux propriétés suivantes :
  - Il porte le même nom que la classe ;
  - Il ne retourne pas de valeur.
- Par le mécanise de surcharges de méthodes, vous pouvez créer autant de constructeurs que vous le souhaitez, un par cas selon les paramètres passés.

2.3.3. CRÉATION DES CONSTRUCTEURS D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- En langage algorithmique, on rajoute le mot-clé « ***Constructeur*** » devant le nom de la méthode en remplacement des mots-clés « ***Procédure*** » ou « ***Fonction*** ».

- Syntaxe :

```
Type
    Classe NomDeLaClasse
        attributs
            attribut1 : type
            attribut2 : type
            ...
        méthodes
            Constructeur NomDeLaClasse()
            Constructeur NomDeLaClasse(paramètres)
            ...
            Procédure nomDeLaProcedure(paramètres)
            Fonction nomDeLaFonction(paramètres) : typeDeRetour
            ...
    FinClasse
```

2.3.3. CRÉATION DES CONSTRUCTEURS D'UNE CLASSE EN LANGAGE ALGORITHMIQUE

- Lors de la création d'un objet de la classe, le constructeur approprié (en fonction des paramètres) sera exécuté de manière implicite.

2.4. DÉFINITION D'UNE CLASSE DE BASE EN PYTHON

2.4.1. CRÉATION D'UNE CLASSE ET CONVENTION DE NOMMAGE

- Il est préférable d'utiliser pour des noms de classes la convention dite ***Camel Case***.
- Cette convention n'utilise pas le signe souligné « _ » pour séparer les mots.
- Le principe consiste à mettre en majuscule chaque lettre débutant un mot, par exemple : « ***MaClasse*** » au lieu de « ***ma_classe*** ».

2.2.3. CRÉATION D'UNE CLASSE ET CONVENTION DE NOMMAGE

- C'est donc cette convention que nous allons utiliser pour les noms de classes dans ce cours.
- Libre à vous d'en changer, encore une fois rien n'est imposé.
- Pour définir une nouvelle classe en Python, on utilise le mot-clé ***class***.
- Sa syntaxe est assez intuitive : **class NomDeLaClasse :**

2.2.3. CRÉATION D'UNE CLASSE ET CONVENTION DE NOMMAGE

```
1   class NomDeLaClasse :
2
3       # Définition du contenu de la classe
4       # Attributs et méthodes
5
6
```

2.2.3. CRÉATION D'UNE CLASSE ET CONVENTION DE NOMMAGE

- N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.
- Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne ?
- Beaucoup de choses, vous en conviendrez mais on ne va en retenir que quelques unes : le ***nom***, le ***prénom***, l'***âge***, le ***lieu de résidence***.

2.2.3. CRÉATION D'UNE CLASSE ET CONVENTION DE NOMMAGE

- Cela nous fait donc quatre attributs.
- Ce sont les ***variables internes*** à notre objet, qui vont le caractériser.
- Une personne telle que nous la modélisons sera caractérisée par son ***nom***, son ***prénom***, son ***âge*** et son ***lieu de résidence***.
- Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe.

2.2.3. CRÉATION D'UNE CLASSE ET CONVENTION DE NOMMAGE

- Voyons cela de plus près.

2.3. LES ATTRIBUTS D'UNE CLASSE

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- Nous avons défini les attributs qui allaient caractériser notre objet de classe Personne.
- Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un ***constructeur***, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.
- Concrètement, un ***constructeur*** est une méthode de notre objet se chargeant de créer nos attributs.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.
- Voyons le code, ce sera plus parlant :

```
1  class Personne: # Définition de notre classe Personne
2      """Classe définissant une personne caractérisée par :
3      - son nom
4      - son prénom
5      - son âge
6      - son lieu de résidence"""
7
8
9      def __init__(self): # Notre méthode constructeur
10         """Pour l'instant, on ne va définir qu'un seul attribut
               """
11         self.nom = "Dupont"
```

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- Voyons en détail :
  - D'abord, la définition de la ***classe***. Elle est constituée du mot-clé ***class***, du nom de la classe et des deux points rituels « **:** ».
  - Une ***docstring*** commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous vous lancerez dans de grands projets, notamment à plusieurs.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- La définition de notre ***constructeur***. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom ***__init__***, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés (***__nommethode__***) sont des méthodes spéciales. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé ***self***.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- Une nouvelle ***docstring***. Je ne complique pas inutilement, je précise donc qu'on va simplement définir un seul attribut pour l'instant dans notre constructeur.

- Dans notre constructeur, nous trouvons l'instanciation de notre attribut ***nom***. On crée une variable ***self.nom*** et on lui donne comme valeur ***Dupont***. Nous voir en détails un peu plus bas ce qui se passe ici.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

```
1 >>> bernard = Personne()
2 >>> bernard
3 <__main__.Personne object at 0x00B42570>
4 >>> bernard.nom
5 'Dupont'
6 >>>
```

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- Quand on demande à l'interpréteur d'afficher directement notre objet ***bernard***, il nous sort quelque chose d'un peu « ***bizarre*** »...
- Bon, l'essentiel est la mention précisant la classe dont l'objet est issu.
- On peut donc vérifier que c'est bien de notre classe ***Personne*** dont est issu notre objet.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- On essaye ensuite d'afficher l'attribut ***nom*** de notre objet ***bernard*** et on obtient ***'Dupont'*** (la valeur définie dans notre constructeur).

- Notez qu'on utilise le point « . », encore et toujours utilisé pour une relation d'appartenance (nom est un attribut de l'objet ***bernard***) c'est-à-dire utilisé pour pouvoir accéder à un attribut ou une méthode d'un objet.

- Encore un peu d'explications :

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- Quand on tape ***Personne()***, on appelle le ***constructeur*** de notre classe ***Personne***, d'une façon quelque peu indirecte que je ne détaillerai pas ici.
- Celui-ci prend en paramètre une variable un peu mystérieuse : ***self***.
- En fait, il s'agit tout bêtement de notre objet qui est entrain de se créer.
- On écrit dans cet objet l'attribut ***nom*** le plus simplement du monde : ***self.nom = "Dupont"***.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- À la fin de l'appel au ***constructeur***, Python renvoie notre objet ***self*** modifié, avec notre attribut.
- On va réceptionner le tout dans notre variable ***bernard***.

2.3.1. CRÉATION DE NOTRE PREMIER ATTRIBUT A PARTIR DU CONSTRUCTEUR

- À la fin de l'appel au ***constructeur***, Python renvoie notre objet ***self*** modifié, avec notre attribut.
- On va réceptionner le tout dans notre variable ***bernard***.

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- On avait dit quatre attributs pour la classe ***Personne***, on n'en a fait qu'un.
- Et puis notre constructeur pourrait éviter de donner les mêmes valeurs par défaut à chaque fois qu'on créé un objet à partir de la classe.
- Dans un premier temps, on va se contenter de définir les autres attributs, le ***prénom***, l'***âge***, le ***lieu de résidence***.

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- Voici le code pour définir les autres attributs et leurs assigner des valeurs par défaut :

```python
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""


    def __init__(self): # Notre méthode constructeur
        """Constructeur de notre classe. Chaque attribut va être instancié
        avec une valeur par défaut... original"""


        self.nom = "Dupont"
        self.prenom = "Jean" # Quelle originalité
        self.age = 33 # Cela n'engage à rien
        self.lieu_residence = "Paris"
```

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- Cela vous paraît évident ?
- Encore un petit code d'exemple :

```
>>> jean = Personne()
>>> jean.nom
'Dupont'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
'Paris'
>>> # Jean déménage..
... jean.lieu_residence = "Berlin"
>>> jean.lieu_residence
'Berlin'
>>>
```

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- Cet exemple paraît assez clair, sur le principe de définition des attributs, accès aux attributs d'un objet créé, modification des attributs d'un objet.
- Une toute petite explication en ce qui concerne la ligne 11 : en général, on déconseille de modifier un attribut d'instance (un attribut d'un objet) comme on vient de le faire, en faisant simplement ***objet.attribut = valeur***.
- Nous verrons dans le prochain chapitre, pourquoi et comment on p procède.

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- Il s'agit des ***accesseurs*** et ***mutateurs***.
- Pour l'instant, il vous suffit de savoir que, quand vous voulez modifier un attribut d'un objet, vous écrivez ***objet.attribut = nouvelle_valeur***.
- Nous verrons les cas particuliers plus loin.

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- Il nous reste encore à faire un constructeur un peu plus intelligent.
- Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence.
- On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons. . . le nom et le prénom, pour commencer.

```python
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""


    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self.lieu_residence = "Paris"
```

```
>>> bernard = Personne("Micado", "Bernard")
>>> bernard.nom
'Micado'
>>> bernard.prenom
'Bernard'
>>> bernard.age
33
>>>
```

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- N'oubliez pas que le premier paramètre du constructeur de la classe doit être ***self***.
- En dehors de cela, un constructeur est une fonction plutôt classique : vous pouvez définir des paramètres, par défaut ou non, nommés ou non.
- Quand vous voudrez créer votre objet, vous appellerez le nom de la classe en passant entre parenthèses les paramètres à utiliser.

2.3.2. CRÉATION DE TOUS LES ATTRIBUTS D'UNE CLASSE

- Faites quelques tests, avec plus ou moins de paramètres, vous saisirez très rapidement le principe.

2.3.3. ATTRIBUTS DE CLASSE

- Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet.
- Ils sont propres à l'objet : si vous créez plusieurs objets, les attributs nom, prenom,. . . de chacun ne seront pas forcément identiques d'un objet à l'autre.
- Mais on peut aussi définir des attributs dans notre classe.
- Voyons un exemple :

```
1  class Compteur:
2      """Cette classe possède un attribut de classe qui s'incré
           mente à chaque
3      fois que l'on crée un objet de ce type"""
4
5
6      objets_crees = 0 # Le compteur vaut 0 au départ
7      def __init__(self):
8          """À chaque fois qu'on crée un objet, on incrémente le
               compteur"""
9          Compteur.objets_crees += 1
```

2.3.3. ATTRIBUTS DE CLASSE

- On définit un attribut de classe directement dans le corps de la classe, sous la définition et la ***docstring***, avant la définition du constructeur.
- Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe.
- Et on y accède de cette façon également, en dehors de la classe.
- Voyez plutôt :

```
>>> Compteur.objets_crees
0
>>> a = Compteur() # On crée un premier objet
>>> Compteur.objets_crees
1
>>> b = Compteur()
>>> Compteur.objets_crees
2
>>>
```

2.3.3. ATTRIBUTS DE CLASSE

- À chaque fois qu'on crée un objet de type ***Compteur***, l'attribut de classe ***objets_crees*** s'incrémente de 1.
- Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques.
- Nous aurons l'occasion d'en reparler par la suite.

2.4. LES MÉTHODES D'UNE CLASSE

2.4.1. LES MÉTHODES D'INSTANCE

- Les attributs sont des variables propres à notre objet, qui servent à le caractériser.
- Les méthodes sont plutôt des actions, comme nous l'avons vu dans la partie précédente, agissant sur l'objet.
- Par exemple, la méthode ***append*** de la classe ***list*** permet d'ajouter un élément dans l'objet ***list*** manipulé.

2.4.1. LES MÉTHODES D'INSTANCE

- Pour créer nos premières méthodes, nous allons modéliser. . . un ***tableau***.
- Notre tableau va posséder une surface (un attribut) sur laquelle on pourra écrire, que l'on pourra lire et effacer.
- Pour créer notre classe ***TableauNoir*** et notre attribut ***surface***, vous ne devriez pas avoir de problème :

```
1 class TableauNoir:
2     """Classe définissant une surface sur laquelle on peut é
          crire,
3     que l'on peut lire et effacer, par jeu de méthodes. L'
          attribut modifié
4     est 'surface'"""
5
6
7     def __init__(self):
8         """Par défaut, notre surface est vide"""
9         self.surface = ""
```

2.4.1. LES MÉTHODES D'INSTANCE

- Nous avons déjà créé une méthode, aussi vous ne devriez pas être trop surpris par la syntaxe que nous allons voir.
- Notre constructeur est en effet une méthode, elle en garde la syntaxe. Nous allons donc écrire notre méthode ***ecrire*** pour commencer.

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut é
        crire,
    que l'on peut lire et effacer, par jeu de méthodes. L'
        attribut modifié
    est 'surface'"""


    def __init__(self):
        """Par défaut, notre surface est vide"""
        self.surface = ""
    def ecrire(self, message_a_ecrire):
        """Méthode permettant d'écrire sur la surface du
            tableau.
        Si la surface n'est pas vide, on saute une ligne avant
            de rajouter
        le message à écrire"""


        if self.surface != "":
            self.surface += "\n"
        self.surface += message_a_ecrire
```

2.4.1. LES MÉTHODES D'INSTANCE

- Passons aux tests :

```
>>> tab = TableauNoir()
>>> tab.surface
''
>>> tab.ecrire("Coooooool ! Ce sont les vacances !")
>>> tab.surface
"Coooooool ! Ce sont les vacances !"
>>> tab.ecrire("Joyeux Noël !")
>>> tab.surface
"Coooooool ! Ce sont les vacances !\nJoyeux Noël !"
>>> print(tab.surface)
Coooooool ! Ce sont les vacances !
Joyeux Noël !
>>>
```

2.4.1. LES MÉTHODES D'INSTANCE

- Notre méthode ***ecrire*** se charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.
- On retrouve ici notre paramètre ***self***.
- Il est temps de voir un peu plus en détail à quoi il sert.

2.4.1. LES MÉTHODES D'INSTANCE

- Notre méthode ***ecrire*** se charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.
- On retrouve ici notre paramètre ***self***.
- Il est temps de voir un peu plus en détail à quoi il sert.

2.4.2. LE PARAMÈTRE SELF

- Dans nos méthodes d'instance, qu'on appelle également des méthodes d'objet, on trouve dans la définition ce paramètre ***self***.
- L'heure est venue de comprendre ce qu'il signifie.
- Une chose qui a son importance : quand vous créez un nouvel objet, ici un tableau noir, les attributs de l'objet sont propres à l'objet créé.

2.4.2. LE PARAMÈTRE SELF

- C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface.
- Donc les attributs sont contenus dans l'objet.
- En revanche, les méthodes sont contenues dans la classe qui définit notre objet.
- C’est très important.

2.4.2. LE PARAMÈTRE SELF

- Quand vous tapez ***tab.ecrire(...)***, Python va chercher la méthode ***ecrire*** non pas dans l'objet ***tab***, mais dans la classe ***TableauNoir***.

```
>>> tab.ecrire
<bound method TableauNoir.ecrire of <__main__.TableauNoir
    object at 0x00B3F3F0>>
>>> TableauNoir.ecrire
<function ecrire at 0x00BA5810>
>>> help(TableauNoir.ecrire)
Help on function ecrire in module __main__:
ecrire(self, message_a_ecrire)
    Méthode permettant d'écrire sur la surface du tableau.
    Si la surface n'est pas vide, on saute une ligne avant de
        rajouter
    le message à écrire.
>>> TableauNoir.ecrire(tab, "essai")
>>> tab.surface
'essai'
>>>
```

2.4.2. LE PARAMÈTRE SELF

- Comme vous le voyez, quand vous tapez ***tab.ecrire(...)***, cela revient au même que si vous écrivez ***TableauNoir.ecrire(tab, ...)***.
- Votre paramètre ***self***, c'est l'objet qui appelle la méthode.
- C'est pour cette raison que nous modifions la surface de l'objet en appelant ***self.surface***.

2.4.2. LE PARAMÈTRE SELF

- Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer par ***self***.
- Le nom ***self*** est une très forte convention de nommage. Je vous déconseille de changer ce nom.
- Certains programmeurs, qui trouvent qu'écrire ***self*** à chaque fois est excessivement long, l'abrègent en une unique lettre **s**.

2.4.2. LE PARAMÈTRE SELF

- Évitez ce raccourci.
- De manière générale, évitez de changer le nom.
- Une méthode d'instance travaille avec le paramètre ***self***.
- Cela peut sembler long de devoir toujours travailler avec ***self*** à chaque fois qu'on souhaite faire appel à l'objet.

2.4.2. LE PARAMÈTRE SELF

- C'est d'ailleurs l'un des reproches qu'on fait au langage Python.
- Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement.
- Mais c'est moins clair et cela peut susciter la confusion.
- En Python, dès qu'on voit ***self***, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

2.4.2. LE PARAMÈTRE SELF

- Voyons maintenant nos autres méthodes.
- Nous devons encore coder la méthode ***lire*** qui va se charger d'afficher notre surface et la méthode ***effacer*** qui va effacer le contenu de notre surface.
- Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples.
- Sinon, n'hésitez pas à relire, jusqu'à ce que vous compreniez.

2.4.2. LE PARAMÈTRE SELF

- Voyons maintenant nos autres méthodes.
- Nous devons encore coder la méthode ***lire*** qui va se charger d'afficher notre surface et la méthode ***effacer*** qui va effacer le contenu de notre surface.
- Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples.
- Sinon, n'hésitez pas à relire, jusqu'à ce que vous compreniez.

2.4.2. LE PARAMÈTRE SELF

- Voyons maintenant nos autres méthodes.
- Nous devons encore coder la méthode ***lire*** qui va se charger d'afficher notre surface et la méthode ***effacer*** qui va effacer le contenu de notre surface.
- Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples.
- Sinon, n'hésitez pas à relire, jusqu'à ce que vous compreniez.

```
1  class TableauNoir:
2      """Classe définissant une surface sur laquelle on peut é
           crire,
3      que l'on peut lire et effacer, par jeu de méthodes. L'
           attribut modifié
4      est 'surface'"""
5
6
7      def __init__(self):
8          """Par défaut, notre surface est vide"""
9          self.surface = ""
```

```
10      def ecrire(self, message_a_ecrire):
11          """Méthode permettant d'écrire sur la surface du
                tableau.
12          Si la surface n'est pas vide, on saute une ligne avant
                de rajouter
13          le message à écrire"""
14
15
16          if self.surface != "":
17              self.surface += "\n"
18          self.surface += message_a_ecrire
19      def lire(self):
20          """Cette méthode se charge d'afficher, grâce à print,
21          la surface du tableau"""
22
23
24          print(self.surface)
25      def effacer(self):
26          """Cette méthode permet d'effacer la surface du tableau
                """
27          self.surface = ""
```

2.4.2. LE PARAMÈTRE SELF

- Et encore une fois, le code de test :

```
1  >>> tab = TableauNoir()
2  >>> tab.lire()
3  >>> tab.ecrire("Salut tout le monde.")
4  >>> tab.ecrire("La forme ?")
5  >>> tab.lire()
6  Salut tout le monde.
7  La forme ?
8  >>> tab.effacer()
9  >>> tab.lire()
10 >>>
```

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Comme on trouve des attributs propres à la classe, on trouve aussi des méthodes de classe, qui ne travaillent pas sur l'instance self mais sur la classe même.
- C'est un peu plus rare mais cela peut être utile parfois.
- Notre méthode de classe se définit exactement comme une méthode d'instance, à la différence qu'elle ne prend pas en premier paramètre ***self*** (l'instance de l'objet) mais ***cls*** (la classe de l'objet).

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- En outre, on utilise ensuite une fonction ***built-in*** de Python pour lui faire comprendre qu'il s'agit d'une méthode de classe, pas d'une méthode d'instance.

- Voyons un exemple :

```
1  class Compteur:
2      """Cette classe possède un attribut de classe qui s'incré
           mente à chaque
3      fois que l'on crée un objet de ce type"""
4
5
6      objets_crees = 0 # Le compteur vaut 0 au départ
7      def __init__(self):
8          """À chaque fois qu'on crée un objet, on incrémente le
               compteur"""
9          Compteur.objets_crees += 1
10     def combien(cls):
11         """Méthode de classe affichant combien d'objets ont été
                crées"""
12         print("Jusqu'à présent, {} objets ont été créés.".
               format(
13                 cls.objets_crees))
14     combien = classmethod(combien)
```

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Le résultat :

```
>>> Compteur.combien()
Jusqu'à présent, 0 objets ont été créés.
>>> a = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 1 objets ont été créés.
>>> b = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 2 objets ont été créés.
>>>
```

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Une méthode de classe prend en premier paramètre non pas ***self*** mais ***cls***.
- Ce paramètre contient la classe (ici ***Compteur***).
- Notez que vous pouvez appeler la méthode de classe depuis un objet instancié sur la classe.
- Vous auriez par exemple pu écrire ***a.combien()***.

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Enfin, pour que Python reconnaisse une méthode de classe, il faut appeler la fonction ***classmethod*** qui prend en paramètre la méthode que l'on veut convertir et renvoie la méthode convertie.

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Si vous êtes un peu perdus, retenez la syntaxe de l'exemple.
- La plupart du temps, vous définirez des méthodes d'instance comme nous l'avons vu plutôt que des méthodes de classe.
- On peut également définir des ***méthodes statiques***.
- Elles sont assez proches des méthodes de classe sauf qu'elles ne prennent aucun premier paramètre, ni ***self*** ni ***cls***.

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Elles travaillent donc indépendamment de toute donnée, aussi bien contenue dans l'instance de l'objet que dans la classe.
- Voici la syntaxe permettant de créer une méthode statique.
- Je vous laisse faire vos propres tests :

2.4.3. MÉTHODES DE CLASSE ET MÉTHODES STATIQUES

- Elles travaillent donc indépendamment de toute donnée, aussi bien contenue dans l'instance de l'objet que dans la classe.
- Voici la syntaxe permettant de créer une méthode statique.
- Je vous laisse faire vos propres tests :

```
class Test:
    """Une classe de test tout simplement"""
    def afficher():
        """Fonction chargée d'afficher quelque chose"""
        print("On affiche la même chose.")
        print("peu importe les données de l'objet ou de la
            classe.")
    afficher = staticmethod(afficher)
```

2.5. L'ENCAPSULATION

2.5.1. DÉFINITION DU PRINCIPE D'ENCAPSULATION

- L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet.
- Dans la plupart des langages orientés objet, tels que le **C++**, le ***Java*** ou le ***PHP***, on va considérer que nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe.
- Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe, ***mon_objet.mon_attribut***.

2.5.1. DÉFINITION DU PRINCIPE D'ENCAPSULATION

- On va définir des méthodes un peu particulières, appelées des ***accesseurs*** et ***mutateurs***.
- Les ***accesseurs*** donnent accès en lecture à l'attribut alors que les ***mutateurs*** permettent de le modifier.
- Concrètement, au lieu d'écrire ***mon_objet.mon_attribut*** pour récupérer l'attribut ***mon_attribut*** de l'objet ***mon_objet***, nous allons écrire ***mon_objet.get_mon_attribut()***.

2.5.1. DÉFINITION DU PRINCIPE D'ENCAPSULATION

- De la même manière, pour modifier l'attribut écrivez ***mon_objet.set_mon_attribut(valeur)*** et non pas ***mon_objet.mon_attribut = valeur***.

- « ***get*** » signifie « ***récupérer*** », c'est le préfixe généralement utilisé pour un accesseur.

2.5.1. DÉFINITION DU PRINCIPE D'ENCAPSULATION

- « ***set*** » signifie, dans ce contexte, « ***modifier*** » ; c'est le préfixe usuel pour un mutateur.
- On peut accéder aux attributs d'un objet directement, comme on l'a fait au chapitre précédent.
- Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages ; en Python, c'est un peu plus subtil.

2.5.1. DÉFINITION DU PRINCIPE D'ENCAPSULATION

- Il peut être très pratique de sécuriser certaines données de notre objet, par exemple faire en sorte qu'un attribut de notre objet ne soit pas modifiable, ou alors mettre à jour un attribut dès qu'un autre attribut est modifié.

- Les cas sont multiples et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

2.5.1. DÉFINITION DU PRINCIPE D'ENCAPSULATION

- Python a une philosophie un peu différente : pour tous les objets dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait au chapitre précédent.
- On peut y accéder et les modifier en écrivant simplement ***mon_objet.mon_attribut***.
- Et pour certains, on va créer des ***propriétés*** (***property***).

2.5.2. LES PROPRIÉTÉS EN PYTHON (PROPERTY)

- Pour commencer, une petite précision : en C++ ou en Java par exemple, dans la définition de classe, on met en place des principes d'accès qui indiquent si l'attribut (ou le groupe d'attributs) est ***privé*** ou ***public***.
- Pour schématiser, si l'attribut est ***public***, on peut y accéder depuis l'extérieur de la classe et le modifier ; s'il est privé, on ne peut pas.
- On doit passer par des ***accesseurs*** ou ***mutateurs***.

2.5.2. LES PROPRIÉTÉS EN PYTHON (PROPERTY)

- En Python, il n'y a pas d'attribut privé ; tout est public.
- Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez.
- Pour faire respecter l'encapsulation propre au langage, on la fonde sur des conventions que nous allons découvrir un peu plus bas mais surtout sur le bon sens de l'utilisateur de notre classe (à savoir, si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, je ne vais pas chercher à y accéder depuis l'extérieur de la classe).

2.5.2. LES PROPRIÉTÉS EN PYTHON (PROPERTY)

- Les ***propriétés*** sont un moyen transparent de manipuler des attributs d'objet.
- Elles permettent de dire à Python : « ***Quand un utilisateur souhaite modifier cet attribut, fais ceci ou cela*** ».
- De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable.

2.5.2. LES PROPRIÉTÉS EN PYTHON (PROPERTY)

- Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.
- Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut.
- C'est dans la définition de la classe que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.

2.5.2. LES PROPRIÉTÉS EN PYTHON (PROPERTY)

- Les propriétés sont des objets un peu particuliers de Python.
- Elles prennent la place d'un attribut et agissent différemment en fonction du contexte dans lequel elles sont appelées.
- Si on les appelle pour modifier l'attribut, par exemple, elles vont rediriger vers une méthode que nous avons créée, qui gère le cas où « ***on souhaite modifier l'attribut*** ».

2.5.2. LES PROPRIÉTÉS EN PYTHON (PROPERTY)

- Mais assez de théorie, passons à la pratique.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Une propriété ne se crée pas dans le constructeur mais dans le corps de la classe.
- Il s'agit d'une classe, son nom est ***property***.
- Elle attend quatre paramètres, tous optionnels :
  - la méthode donnant accès à l'attribut ;
  - la méthode modifiant l'attribut ;

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- la méthode appelée quand on souhaite supprimer l'attribut ;
- la méthode appelée quand on demande de l'aide sur l'attribut.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit nos accesseur et mutateur d'objet.
- Mais j'imagine que ce n'est pas très clair dans votre esprit.
- Considérez le code suivant, on verra les détails plus bas comme d'habitude :

```python
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom ;
    - son prénom ;
    - son âge ;
    - son lieu de résidence"""


    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._lieu_residence = "Paris" # Notez le souligné _
            devant le nom
```

```
15      def _get_lieu_residence(self):
16      """Méthode qui sera appelée quand on souhaitera accéder en
            lecture
17          à l'attribut 'lieu_residence'"""
18
19
20          print("On accède à l'attribut lieu_residence !")
21          return self._lieu_residence
22      def _set_lieu_residence(self, nouvelle_residence):
23          """Méthode appelée quand on souhaite modifier le lieu
                de résidence"""
24          print("Attention, il semble que {} déménage à {}.".
                format( \
25                      self.prenom, nouvelle_residence))
26          self._lieu_residence = nouvelle_residence
27      # On va dire à Python que notre attribut lieu_residence
            pointe vers une
28      # propriété
29      lieu_residence = property(_get_lieu_residence,
            _set_lieu_residence)
```

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Vous devriez reconnaître la syntaxe générale de la classe.
- En revanche, au niveau du lieu de résidence, les choses changent un peu :

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Tout d'abord, dans le constructeur, on ne crée pas un attribut ***self.lieu_residence*** mais ***self._lieu_residence***. Il n'y a qu'un petit caractère de différence, le signe souligné « _ » placé en tête du nom de l'attribut. Et pourtant, ce signe change beaucoup de choses. La convention veut qu'on n'accède pas, depuis l'extérieur de la classe, à un attribut commençant par un souligné « _ ». C'est une convention, rien ne vous l'interdit... sauf, encore une fois, le bon sens.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- On définit une première méthode, commençant elle aussi par un souligné _, nommée ***_get_lieu_residence***. C'est la même règle que pour les attributs : on n'accède pas, depuis l'extérieur de la classe, à une méthode commençant par un souligné _. Si vous avez compris ma petite explication sur les accesseurs et mutateurs, vous devriez comprendre rapidement à quoi sert cette méthode : elle se contente de renvoyer le lieu de résidence.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Là encore, l'attribut manipulé n'est pas ***lieu_residence*** mais ***_lieu_residence***. Comme on est dans la classe, on a le droit de le manipuler.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- La seconde méthode a la forme d'un mutateur. Elle se nomme ***_set_lieu_residence*** et doit donc aussi être inaccessible depuis l'extérieur de la classe. À la différence de l'accesseur, elle prend un paramètre : le nouveau lieu de résidence. En effet, c'est une méthode qui doit être appelée quand on cherche à modifier le lieu de résidence, il lui faut donc le nouveau lieu de résidence qu'on souhaite voir affecté à l'objet.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Enfin, la dernière ligne de la classe est très intéressante. Il s'agit de la définition d'une propriété. On lui dit que l'attribut ***lieu_residence*** (cette fois, sans signe souligné _) doit être une propriété. On définit dans notre propriété, dans l'ordre, la méthode d'accès (***l'accesseur***) et celle de modification (le ***mutateur***).

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Quand on veut accéder à ***objet.lieu_residence***, Python tombe sur une propriété redirigeant vers la méthode ***_get_lieu_residence***.
- Quand on souhaite modifier la valeur de l'attribut, en écrivant ***objet.lieu_residence = valeur***, Python appelle la méthode ***_set_lieu_residence*** en lui passant en paramètre la nouvelle valeur.
- Voyons cet exemple :

```
>>> jean = Personne("Micado", "Jean")
>>> jean.nom
'Micado'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Paris'
>>> jean.lieu_residence = "Berlin"
Attention, il semble que Jean déménage à Berlin.
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Berlin'
>>>
```

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Notre accesseur et notre mutateur se contentent d'afficher un message, pour bien qu'on se rende compte que ce sont eux qui sont appelés quand on souhaite manipuler l'attribut ***lieu_residence***.

- Vous pouvez aussi ne définir qu'un accesseur, dans ce cas l'attribut ne pourra pas être modifié.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Il est aussi possible de définir, en troisième position du constructeur ***property***, une méthode qui sera appelée quand on fera ***del objet.lieu_residence*** et, en quatrième position, une méthode qui sera appelée quand on fera ***help(objet.lieu_residence)***.

- Ces deux dernières fonctionnalités sont un peu moins utilisées mais elles existent.

2.5.3. IMPLÉMENTATION DU PRINCIPE DE L'ENCAPSULATION EN PYTHON

- Voilà, vous connaissez à présent la syntaxe pour créer des propriétés.
- Entraînez-vous, ce n'est pas toujours évident au début.
- C'est un concept très puissant, il serait dommage de passer à côté.

2.5.4. RÉSUMÉ DU PRINCIPE D'ENCAPSULATION EN PYTHON

- Dans cette section, nous allons condenser un peu tout le chapitre.
- Nous avons vu qu'en Python, quand on souhaite accéder à un attribut d'un objet, on écrit tout simplement ***objet.attribut***.
- Par contre, on doit éviter d'accéder ainsi à des attributs ou des méthodes commençant par un signe souligné « _ », question de convention.

2.5.4. RÉSUMÉ DU PRINCIPE D'ENCAPSULATION EN PYTHON

- Si par hasard une action particulière doit être menée quand on accède à un attribut, pour le lire tout simplement, pour le modifier, le supprimer. . ., on fait appel à des ***propriétés***.
- Pour l'utilisateur de la classe, cela revient au même : il écrit toujours ***objet.attribut***.

2.5.4. RÉSUMÉ DU PRINCIPE D'ENCAPSULATION EN PYTHON

- Mais dans la définition de notre classe, nous faisons en sorte que l'attribut visé soit une ***propriété*** avec certaines méthodes, ***accesseur***, ***mutateur*** ou autres, qui définissent ce que Python doit faire quand on souhaite ***lire***, ***modifier***, ***supprimer*** l'attribut.

2.5.4. RÉSUMÉ DU PRINCIPE D'ENCAPSULATION EN PYTHON

- Certaines classes ont besoin qu'un traitement récurrent soit effectué sur leurs attributs.
- Par exemple, quand je souhaite modifier un attribut de l'objet (n'importe quel attribut), l'objet doit être enregistré dans un fichier.
- Dans ce cas, on n'utilisera pas les propriétés, qui sont plus utiles pour des cas particuliers, mais plutôt des méthodes spéciales, que nous découvrirons au prochain chapitre.

2.5.5. EN RÉSUMÉ

- On définit une classe en suivant la syntaxe « ***class NomClasse :*** ».
- Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe.
- Les méthodes d'instance prennent en premier paramètre self, l'instance de l'objet manipulé.

2.5.5. EN RÉSUMÉ

- On construit une instance de classe en appelant son constructeur, une méthode d'instance appelée ***__init__***.
- On définit les attributs d'une instance dans le constructeur de sa classe, en suivant cette syntaxe : ***self.nom_attribut = valeur***.
- Les ***propriétés*** permettent de contrôler l'accès à certains attributs d'une instance.

2.5.5. EN RÉSUMÉ

- Elles se définissent dans le corps de la classe en suivant cette syntaxe : ***nom_propriete = propriete(methode_accesseur, methode_mutateur, methode_suppression, methode_aide***).
- On y fait appel ensuite en écrivant ***objet.nom_propriete*** comme pour n'importe quel attribut.

2.5.5. EN RÉSUMÉ

- Si l'on souhaite juste lire l'attribut, c'est la méthode définie comme ***accesseur*** qui est appelée.
- Si l'on souhaite modifier l'attribut, c'est la méthode ***mutateur***, si elle est définie, qui est appelée.
- Chacun des paramètres à passer à ***property*** est optionnel.

CHAP. 3 : LES MÉTHODES SPÉCIALES

- Les méthodes spéciales sont des méthodes d'instance que Python reconnaît et sait utiliser dans certains contextes.
- Elles peuvent servir à indiquer à Python ce qu'il doit faire quand il se retrouve devant certaines expressions.
- Et, encore plus fort, elles contrôlent la façon dont un objet se crée, ainsi que l'accès à ses attributs.

CHAP. 3 : LES MÉTHODES SPÉCIALES

- Bref, encore une fonctionnalité puissante et utile du langage, que je vous invite à découvrir.
- Prenez note du fait que je ne peux pas expliquer dans ce chapitre la totalité des méthodes spéciales.
- Faites un tour sur le site officiel de Python pour plus d'informations.

3.1. CRÉATION DE L'OBJET ET ACCÈS AUX ATTRIBUTS

- Vous avez déjà vu, dès le début de cette partie, un exemple de ***méthode spéciale***.
- Pour ceux qui ont oublié, il s'agit de notre ***constructeur***.
- Une méthode spéciale, en Python, voit son nom entouré de part et d'autre par deux signes « ***souligné*** » __.
- Le nom d'une méthode spéciale prend donc la forme : ***__methodespeciale__***.

3.1. CRÉATION DE L'OBJET ET ACCÈS AUX ATTRIBUTS

- Pour commencer, nous allons voir les méthodes qui travaillent directement sur l'objet.
- Nous verrons ensuite, plus spécifiquement, les méthodes qui permettent d'accéder aux attributs.

3.1.1. CRÉATION DE L'OBJET

- Les méthodes que nous allons voir permettent de travailler sur l'objet.
- Elles interviennent au moment de le créer et au moment de le supprimer.
- La première, vous devriez la reconnaître : c'est notre ***constructeur***.
- Elle s'appelle ***__init__***, prend un nombre variable d'arguments et permet de contrôler la création de nos attributs.

3.1.1. CRÉATION DE L'OBJET

```
1 class Exemple:
2     """Un petit exemple de classe"""
3     def __init__(self, nom):
4         """Exemple de constructeur"""
5         self.nom = nom
6         self.autre_attribut = "une valeur"
```

3.1.1. CRÉATION DE L'OBJET

- Pour créer notre objet, nous utilisons le nom de la classe et nous passons, entre parenthèses, les informations qu'attend notre constructeur :

```
1 | mon_objet = Exemple("un premier exemple")
```

3.1.1. CRÉATION DE L'OBJET

- À partir du moment où l'objet est créé, on peut accéder à ses attributs grâce à ***mon_objet.nom_attribut*** et exécuter ses méthodes grâce à ***mon_objet.nom_methode(...)***.
- Il existe également une autre méthode, ***__del__***, qui va être appelée au moment de la destruction de l'objet.

3.1.2. DESTRUCTION DE L'OBJET

- Il y a plusieurs cas pour la destruction d'un objet :
  - D'abord, quand vous voulez le supprimer explicitement, grâce au mot-clé ***del*** (***del mon_objet***).
  - Ensuite, si l'espace de noms contenant l'objet est détruit, l'objet l'est également.

    Par exemple, si vous instanciez l'objet dans le corps d'une fonction : à la fin de l'appel à la fonction, la méthode ***__del__*** de l'objet sera appelée.

3.1.2. DESTRUCTION DE L'OBJET

- Enfin, si votre objet résiste envers et contre tout pendant l'exécution du programme, il sera supprimé à la fin de l'exécution.

- Exemple d'implémentation de la méthode ***__del__*** :

```
1 def __del__(self):
2         """Méthode appelée quand l'objet est supprimé"""
3         print("C'est la fin ! On me supprime !")
```

3.1.2. DESTRUCTION DE L'OBJET

- Python s’en sort tout seul et n’a pas besoin d’aide pour gérer la destruction d’un objet.
- Mais parfois, on peut vouloir récupérer des informations d'état sur l'objet au moment de sa suppression ou avoir besoin d’effectuer une action bien précise lors de la suppression des objets d’une classe.

3.1.2. DESTRUCTION DE L'OBJET

- Sachez que si vous ne définissez pas de méthode spéciale pour telle ou telle action, Python aura un comportement par défaut dans le contexte où cette méthode est appelée.
- Écrire une méthode spéciale permet de modifier ce comportement par défaut.
- Dans l'absolu, vous n'êtes même pas obligés d'écrire un constructeur.

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- Nous allons voir deux méthodes spéciales qui permettent de contrôler comment l'objet est représenté et affiché.

- Vous avez sûrement déjà pu constater que, quand on instancie des objets issus de nos propres classes, si on essaye de les afficher directement dans l'interpréteur ou grâce à ***print***, on obtient quelque chose d'assez laid :

```
1 <__main__.XXX object at 0x00B46A70>
```

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- On a certes les informations utiles, mais pas forcément celles qu'on veut, et l'ensemble n'est pas magnifique, il faut bien le reconnaître.
- La première méthode permettant de remédier à cet état de fait est ***__repr__***.
- Elle affecte la façon dont est affiché l'objet quand on tape directement son nom.

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- On la redéfinit quand on souhaite faciliter le ***debug*** sur certains objets :

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

```
1  class Personne:
2      """Classe représentant une personne"""
3      def __init__(self, nom, prenom):
4          """Constructeur de notre classe"""
5          self.nom = nom
6          self.prenom = prenom
7          self.age = 33
8      def __repr__(self):
9          """Quand on entre notre objet dans l'interpréteur"""
10         return "Personne: nom({}), prénom({}), âge({})".format(
11                 self.nom, self.prenom, self.age)
```

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- Et le résultat :

```
1 >>> p1 = Personne("Micado", "Jean")
2 >>> p1
3 Personne: nom(Micado), prénom(Jean), âge(33)
4 >>>
```

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- Comme vous le voyez, la méthode ***__repr__*** ne prend aucun paramètre (sauf, bien entendu, ***self***) et renvoie une chaîne de caractères : la chaîne à afficher quand on entre l'objet directement dans l'interpréteur.

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- On peut également obtenir cette chaîne grâce à la fonction ***repr***, qui se contente d'appeler la méthode spéciale ***__repr__*** de l'objet passé en paramètre :

```
>>> p1 = Personne("Micado", "Jean")
>>> repr(p1)
'Personne: nom(Micado), prénom(Jean), âge(33)'
>>>
```

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- Il existe une seconde méthode spéciale, ***__str__***, spécialement utilisée pour afficher l'objet avec ***print***.
- Par défaut, si aucune méthode ***__str__*** n'est définie, Python appelle la méthode ***__repr__*** de l'objet.
- La méthode ***__str__*** est également appelée si vous désirez convertir votre objet en chaîne avec le constructeur ***str***.

```
1  class Personne:
2      """Classe représentant une personne"""
3      def __init__(self, nom, prenom):
4          """Constructeur de notre classe"""
5          self.nom = nom
6          self.prenom = prenom
7          self.age = 33
8      def __str__(self):
9          """Méthode permettant d'afficher plus joliment notre
               objet"""
10         return "{} {}, âgé de {} ans".format(
11                 self.prenom, self.nom, self.age)
```

3.1.3. REPRÉSENTATION OU AFFICHAGE DE L'OBJET

- Et en pratique :

```
1  >>> p1 = Personne("Micado", "Jean")
2  >>> print(p1)
3  Jean Micado, âgé de 33 ans
4  >>> chaine = str(p1)
5  >>> chaine
6  'Jean Micado, âgé de 33 ans'
7  >>>
```

3.2. LES MÉTHODES DE CONTENEUR

- Nous allons commencer à travailler sur ce que l'on appelle la surcharge d'opérateurs.
- Il s'agit assez simplement d'expliquer à Python quoi faire quand on utilise tel ou tel opérateur.
- Nous allons ici voir quatre méthodes spéciales qui interviennent quand on travaille sur des objets conteneurs.

3.2.1. ACCÈS AUX ÉLÉMENTS D'UN CONTENEUR

- Les objets conteneurs, j'espère que vous vous en souvenez, ce sont les chaînes de caractères, les listes et les dictionnaires, entre autres.
- Tous ont un point commun : ils contiennent d'autres objets, auxquels on peut accéder grâce à l'opérateur [].
- Les trois premières méthodes que nous allons voir sont ***__getitem__***, ***__setitem__*** et ***__delitem__***.

3.2.1. ACCÈS AUX ÉLÉMENTS D'UN CONTENEUR

- Elles servent respectivement à définir quoi faire quand on écrit :
  - ***objet[index] ;***
  - ***objet[index] = valeur ;***
  - ***del objet[index] ;***

3.2.1. ACCÈS AUX ÉLÉMENTS D'UN CONTENEUR

- Pour cet exemple, nous allons voir une classe contenant un dictionnaire.
- Nous allons créer une classe que nous allons appeler ***ZDict***.
- Elle va posséder un attribut auquel on ne devra pas accéder de l'extérieur de la classe, un dictionnaire que nous appellerons ***_dictionnaire***.
- Quand on créera un objet de type ***ZDict*** et qu'on voudra faire ***objet[index]***, à l'intérieur de la classe on fera ***self._dictionnaire[index]***.

```python
class ZDict:
    """Classe enveloppe d'un dictionnaire"""
    def __init__(self):
        """Notre classe n'accepte aucun paramètre"""
        self._dictionnaire = {}
    def __getitem__(self, index):
        """Cette méthode spéciale est appelée quand on fait
            objet[index]
        Elle redirige vers self._dictionnaire[index]"""

        return self._dictionnaire[index]
    def __setitem__(self, index, valeur):
        """Cette méthode est appelée quand on écrit objet[index
            ] = valeur
        On redirige vers self._dictionnaire[index] = valeur"""

        self._dictionnaire[index] = valeur
```

3.2.1. ACCÈS AUX ÉLÉMENTS D'UN CONTENEUR

- Vous avez un exemple d'utilisation des deux méthodes ***__getitem__*** et ***__setitem__*** qui, je pense, est assez clair.
- Pour ***__delitem__***, c'est assez évident, elle ne prend qu'un seul paramètre qui est l'index que l'on souhaite supprimer.
- Vous pouvez étendre cet exemple avec d'autres méthodes que nous avons vues plus haut, notamment ***__repr__*** et ***__str__***.

3.2.1. ACCÈS AUX ÉLÉMENTS D'UN CONTENEUR

- N'hésitez pas, entraînez-vous, tout cela peut vous servir.

3.2.2. LA MÉTHODE SPÉCIALE DERRIÈRE LE MOT-CLÉ in

- Il existe une quatrième méthode, appelée ***__contains__***, qui est utilisée quand on souhaite savoir si un objet se trouve dans un conteneur.

- Exemple classique :

```
1 ma_liste = [1, 2, 3, 4, 5]
2 8 in ma_liste # Revient au même que ...
3 ma_liste.__contains__(8)
```

3.2.2. LA MÉTHODE SPÉCIALE DERRIÈRE LE MOT-CLÉ in

- Ainsi, si vous voulez que votre classe puisse utiliser le mot-clé ***in*** comme une liste ou un dictionnaire, vous devez redéfinir cette méthode ***__contains__*** qui prend en paramètre, outre ***self***, ***l'objet*** qui nous intéresse.
- Si l'objet est dans le conteneur, on doit renvoyer ***True*** ; sinon ***False***.
- Je vous laisse redéfinir cette méthode, vous avez toutes les indications nécessaires.

3.2.3. CONNAÎTRE LA TAILLE D'UN CONTENEUR

- Il existe enfin une méthode spéciale ***__len__***, appelée quand on souhaite connaître la taille d'un objet conteneur, grâce à la fonction ***len***.
- ***len(objet)*** équivaut à ***objet.__len__()***.
- Cette méthode spéciale ne prend aucun paramètre et renvoie une taille sous la forme d'un entier.
- Là encore, je vous laisse faire l'essai.

3.3. LES MÉTHODES MATHÉMATIQUES

- Pour cette section, nous allons continuer à voir les méthodes spéciales permettant la surcharge d'opérateurs mathématiques, comme **+**, **-**, *, etc.

3.3.1. CE QU'IL FAUT SAVOIR

- Pour cette section, nous allons utiliser un nouvel exemple, une classe capable de contenir des durées.
- Ces durées seront contenues sous la forme d'un nombre de minutes et un nombre de secondes.
- Voici le corps de la classe :

```
1  class Duree:
2      """Classe contenant des durées sous la forme d'un nombre de
           minutes
3      et de secondes"""
4
5      def __init__(self, min=0, sec=0):
6          """Constructeur de la classe"""
7          self.min = min # Nombre de minutes
8          self.sec = sec # Nombre de secondes
9      def __str__(self):
10         """Affichage un peu plus joli de nos objets"""
11         return "{0:02}:{1:02}".format(self.min, self.sec)
```

3.3.1. CE QU'IL FAUT SAVOIR

- On définit simplement deux attributs contenant notre nombre de minutes et notre nombre de secondes, ainsi qu'une méthode pour afficher tout cela un peu mieux.

- Si vous vous interrogez sur l'utilisation de la méthode format dans la méthode ***__str__***, sachez simplement que le but est de voir la durée sous la forme ***MM:SS*** ; pour plus d'informations sur le formatage des chaînes, vous pouvez consulter la documentation de Python.

3.3.1. CE QU'IL FAUT SAVOIR

- Créons un premier objet ***Duree*** que nous appelons ***d1*** :

```
1  >>> d1 = Duree(3, 5)
2  >>> print(d1)
3  03:05
4  >>>
```

3.3.1. CE QU'IL FAUT SAVOIR

- Si vous essayez de faire ***d1 + 4***, par exemple, vous allez obtenir une erreur.
- Python ne sait pas comment additionner un type ***Duree*** et un ***int***.
- Il ne sait même pas comment ajouter deux durées c'est-à-dire deux objets ***Duree*** comme ***d1 + d2*** !
- Nous allons donc lui expliquer.

3.3.1. CE QU'IL FAUT SAVOIR

- La méthode spéciale à redéfinir est ***__add__***.
- Elle prend en paramètre l'objet que l'on souhaite ajouter.
- Voici deux lignes de code qui reviennent au même :

```
1 d1 + 4
2 d1.__add__(4)
```

3.3.1. CE QU'IL FAUT SAVOIR

- Comme vous le voyez, quand vous utilisez le symbole + ainsi, c'est en fait la méthode **__add__** de l'objet **Duree** qui est appelée.
- Elle prend en paramètre l'objet que l'on souhaite ajouter, peu importe le type de l'objet en question.
- Et elle doit renvoyer un objet exploitable, ici il serait plus logique que ce soit une nouvelle durée.

3.3.1. CE QU'IL FAUT SAVOIR

- Si vous devez faire différentes actions en fonction du type de l'objet à ajouter, testez le résultat de ***type(objet_a_ajouter)***.
- Implémentation de la méthode ***__add__*** dans la classe ***Duree*** :

```
def __add__(self, objet_a_ajouter):
        """L'objet à ajouter est un entier, le nombre de
           secondes"""
        nouvelle_duree = Duree()
        # On va copier self dans l'objet créé pour avoir la mê
           me durée
        nouvelle_duree.min = self.min
        nouvelle_duree.sec = self.sec
        # On ajoute la durée
        nouvelle_duree.sec += objet_a_ajouter
        # Si le nombre de secondes >= 60
        if nouvelle_duree.sec >= 60:
            nouvelle_duree.min += nouvelle_duree.sec // 60
            nouvelle_duree.sec = nouvelle_duree.sec % 60
        # On renvoie la nouvelle durée
        return nouvelle_duree
```

3.3.1. CE QU'IL FAUT SAVOIR

- Prenez le temps de comprendre le mécanisme et le petit calcul pour vous assurer d'avoir une durée cohérente.
- D'abord, on crée une nouvelle durée qui est l'équivalent de la durée contenue dans ***self***.
- On l'augmente du nombre de secondes à ajouter et on s'assure que le temps est cohérent (le nombre de secondes n'atteint pas 60).

3.3.1. CE QU'IL FAUT SAVOIR

- Si le temps n’est pas cohérent, on le corrige.
- On renvoie enfin notre nouvel objet modifié.
- Voici un petit code qui montre comment utiliser notre méthode :

3.3.1. CE QU'IL FAUT SAVOIR

```
>>> d1 = Duree(12, 8)
>>> print(d1)
12:08
>>> d2 = d1 + 54 # d1 + 54 secondes
>>> print(d2)
13:02
>>>
```

3.3.1. CE QU'IL FAUT SAVOIR

- Pour mieux comprendre, remplacez ***d2 = d1 + 54*** par ***d2 = d1.__add__(54)*** : cela revient au même.
- Ce remplacement ne sert qu'à bien comprendre le mécanisme.
- Il va de soi que ces méthodes spéciales ne sont pas à appeler directement depuis l'extérieur de la classe, les opérateurs n'ont pas été inventés pour rien.

3.3.1. CE QU'IL FAUT SAVOIR

- Pour mieux comprendre, remplacez d2 = d1 + 54 par d2 = d1.__add__(54) : cela revient au même.
- Ce remplacement ne sert qu'à bien comprendre le mécanisme.
- Sachez que sur le même modèle, il existe les méthodes :
  - ***__sub__*** **:** surcharge de l'opérateur **-** ;
  - ***__mul__*** **:** surcharge de l'opérateur * ;

3.3.1. CE QU'IL FAUT SAVOIR

- ***__truediv__*** : surcharge de l'opérateur / ;
- ***__floordiv__*** : surcharge de l'opérateur // (division entière) ;
- ***__mod__*** : surcharge de l'opérateur % (modulo) ;
- ***__pow__*** : surcharge de l'opérateur ** (puissance) ;
- Etc.

3.3.1. CE QU'IL FAUT SAVOIR

- Il y en a d'autres que vous pouvez trouver en consultant la documentation de Python.

3.3.2. IMPORTANCE DE L'ORDRE DES OPÉRANDES (OBJETS)

- Vous l'avez peut-être remarqué, écrire ***objet1 + objet2*** ne revient pas au même qu'écrire ***objet2 + objet1*** siles deux objets ont des types différents.
- En effet, suivant le cas, c'est la méthode ***__add__*** de l'un ou l'autre des objets qui est appelée.
- Cela signifie que, lorsqu'on utilise la classe ***Duree***, si on écrit ***d1 + 4*** cela fonctionne, alors que ***4 + d1*** ne marche pas.

3.3.2. IMPORTANCE DE L'ORDRE DES OPÉRANDES (OBJETS)

- En effet, la class ***int*** ne sait pas quoi faire de votre objet ***Duree***.
- Il existe cependant une panoplie de méthodes spéciales pour faire le travail de ***__add__*** si vous écrivez l'opération dans l'autre sens.
- Il suffit de préfixer le nom des méthodes spéciales par un ***r*** :

```
1 def __radd__(self, objet_a_ajouter):
2         """Cette méthode est appelée si on écrit 4 + objet et
              que
3         le premier objet (4 dans cet exemple) ne sait pas
              comment ajouter
4         le second. On se contente de rediriger sur __add__
              puisque,
5         ici, cela revient au même : l'opération doit avoir le m
              ême résultat,
6         posée dans un sens ou dans l'autre"""
7
8         return self + objet_a_ajouter
```

3.3.2. IMPORTANCE DE L'ORDRE DES OPÉRANDES (OBJETS)

- À présent, on peut écrire ***4 + d1***, cela revient au même que ***d1 + 4***.
- N'hésitez pas à relire ces exemples s'ils vous paraissent peu clairs.

3.3.3. D'AUTRES OPÉRATEURS MATHÉMATIQUES

- Il est également possible de surcharger les opérateurs **+=**, **-=**, etc.
- On préfixe cette fois-ci les noms de méthode que nous avons vus par un ***i***.
- Exemple de méthode ***__iadd__*** pour notre classe ***Duree*** :

```
1  def __iadd__(self, objet_a_ajouter):
2          """L'objet à ajouter est un entier, le nombre de
               secondes"""
3          # On travaille directement sur self cette fois
4          # On ajoute la durée
5          self.sec += objet_a_ajouter
6          # Si le nombre de secondes >= 60
7          if self.sec >= 60:
8              self.min += self.sec // 60
9              self.sec = self.sec % 60
10         # On renvoie self
11         return self
```

3.3.3. D'AUTRES OPÉRATEURS MATHÉMATIQUES

- Résultats du test :

```
1  >>> d1 = Duree(8, 5)
2  >>> d1 += 128
3  >>> print(d1)
4  10:13
5  >>>
```

3.4. LES MÉTHODES DE COMPARAISON

- Pour finir, nous allons voir la surcharge des opérateurs de comparaison que vous connaissez depuis quelque temps maintenant : **==, !=, <, >, <=, >=.**
- Ces méthodes sont donc appelées si vous tentez de comparer deux objets entre eux.
- Comment Python sait-il que 3 est inférieur à 18 ?
- Une méthode spéciale de la classe **int** le permet, en simplifiant.

3.4. LES MÉTHODES DE COMPARAISON

- Donc si vous voulez comparer des durées, par exemple, vous allez devoir redéfinir certaines méthodes que nous allons voir plus bas.
- Elles devront prendre en paramètre l'objet à comparer à ***self***, et doivent renvoyer un ***booléen*** (***True*** ou ***False***).
- Je vais me contenter de vous faire un petit tableau récapitulatif des méthodes à redéfinir pour comparer deux objets entre eux :

Opérateur	Méthode spéciale	Résumé
==	`def __eq__(self, objet_a_comparer):`	Opérateur d'égalité (*equal*). Renvoie `True` si `self` et `objet_a_comparer` sont égaux, `False` sinon.
!=	`def __ne__(self, objet_a_comparer):`	Différent de (*non equal*). Renvoie `True` si `self` et `objet_a_comparer` sont différents, `False` sinon.
>	`def __gt__(self, objet_a_comparer):`	Teste si `self` est strictement supérieur (*greather than*) à `objet_a_comparer`.
>=	`def __ge__(self, objet_a_comparer):`	Teste si `self` est supérieur ou égal (*greater or equal*) à `objet_a_comparer`.
<	`def __lt__(self, objet_a_comparer):`	Teste si `self` est strictement inférieur (*lower than*) à `objet_a_comparer`.
<=	`def __le__(self, objet_a_comparer):`	Teste si `self` est inférieur ou égal (*lower or equal*) à `objet_a_comparer`.

3.4. LES MÉTHODES DE COMPARAISON

- Sachez que ce sont ces méthodes spéciales qui sont appelées si, par exemple, vous voulez trier une liste contenant vos objets.
- Sachez également que, si Python n'arrive pas à faire ***objet1 < objet2***, il essayera l'opération inverse, soit ***objet2 >= objet1***.
- Cela vaut aussi pour les autres opérateurs de comparaison que nous venons de voir.

3.4. LES MÉTHODES DE COMPARAISON

- Nous allons voir deux exemples malgré tout, il ne tient qu'à vous de redéfinir les autres méthodes présentées plus haut :

```
1  def __eq__(self, autre_duree):
2          """Test si self et autre_duree sont égales"""
3          return self.sec == autre_duree.sec and self.min ==
               autre_duree.min
4      def __gt__(self, autre_duree):
5          """Test si self > autre_duree"""
6          # On calcule le nombre de secondes de self et
               autre_duree
7          nb_sec1 = self.sec + self.min * 60
8          nb_sec2 = autre_duree.sec + autre_duree.min * 60
9          return nb_sec1 > nb_sec2
```

3.5. EN RÉSUMÉ

- Les méthodes spéciales permettent d'influencer la manière dont Python accède aux attributs d'une instance et réagit à certains opérateurs ou conversions.
- Les méthodes spéciales sont toutes entourées de deux signes « ***souligné*** » (_).
- Les méthodes ***__getitem__***, ***__setitem__*** et ***__delitem__*** surchargent l'indexation (*[]*).

3.5. EN RÉSUMÉ

- Les méthodes ***__add__***, ***__sub__***, ***__mul__***, etc. surchargent les opérateurs mathématiques.
- Les méthodes ***__eq__***, ***__ne__***, ***__gt__***, etc. surchargent les opérateurs de comparaison.

CHAP. 4 : L'HÉRITAGE

- On dit souvent qu'un langage de Programmation Orientée Objet n'incluant pas l'héritage serait incomplet, sinon inutile.
- Nous allons voir en détails dans ce chapitre, l'utilité de ce concept et voir aussi plusieurs exemples d'applications.
- Nous allons donc devoir nous pencher sur de la théorie et travailler sur quelques exemples de modélisation.

4.1. DÉFINITION

- Ne faisons plus durer le suspense longtemps : « ***l'héritage*** » est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la ***classe parente***, ou la ***classe mère***.
- La classe nouvelle classe qu'on définit est appelée ***classe fille***.
- Concrètement, si une classe ***B*** hérite de la classe ***A***, les objets créés sur le modèle de la classe ***B*** auront accès aux méthodes et attributs de la classe ***A***.

4.1. DÉFINITION

- Dans l'exemple précédent, la classe ***A*** est la ***classe mère*** et ***B*** est la ***classe fille***.
- La première chose, c'est que la classe ***B*** dans notre exemple ne se contente pas de reprendre les méthodes et attributs de la classe ***A*** : ***elle va pouvoir en définir d'autres***.
- La classe B va pouvoir donc définir d'autres méthodes et d'autres attributs qui lui seront propres, en plus des méthodes et attributs de la classe ***A***.

4.1. DÉFINITION

- Et elle va pouvoir également redéfinir les méthodes de la classe mère.
- Prenons un exemple simple : on a une classe ***Animal*** permettant de définir des animaux.
- Les animaux tels que nous les modélisons ont certains attributs (le régime : carnivore ou herbivore) et certaines méthodes (manger, boire, crier. . .).

4.1. DÉFINITION

- On peut maintenant définir une classe ***Chien*** qui hérite de ***Animal***, c'est-à-dire qu'elle reprend ses méthodes.
- Nous allons voir plus bas ce que cela implique exactement.

4.1. DÉFINITION

- Si vous ne voyez pas très bien dans quel cas on fait hériter une classe d'une autre, faisons le test :
  - On fait hériter la classe ***Chien*** de ***Animal*** parce qu'un « ***chien EST UN animal*** » ;
  - On ne fait pas hériter ***Animal*** de ***Chien*** parce qu'un « ***Animal n'est pas un Chien*** ».

4.1. DÉFINITION

- Sur ce modèle, vous pouvez vous rendre compte qu'une « ***voiture est un véhicule*** ».
- La classe ***Voiture*** pourrait donc hériter de la classe ***Vehicule***.
- Intéressons-nous à présent au code.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- On oppose « ***l'héritage simple*** », dont nous venons de voir les aspects théoriques dans la section précédente, à « ***l'héritage multiple*** » que nous verrons dans la prochaine section.
- Il est temps d'aborder la syntaxe de l'héritage.
- Nous allons définir une première classe ***A*** et une seconde classe ***B*** qui hérite de la classe ***A***.

```
1  class A:
2      """Classe A, pour illustrer notre exemple d'héritage"""
3      pass # On laisse la définition vide, ce n'est qu'un exemple
4
```

```
5  class B(A):
6      """Classe B, qui hérite de A.
7      Elle reprend les mêmes méthodes et attributs (dans cet
           exemple, la classe
8      A ne possède de toute façon ni méthode ni attribut)"""
9
10     pass
```

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Vous pourrez expérimenter par la suite sur des exemples plus constructifs.
- Pour l'instant, l'important est de bien noter la syntaxe qui, comme vous le voyez, est des plus simples :

class MaClasse(MaClasseMere) :

- Dans la définition de la classe, entre le nom et les deux points, vous précisez entre parenthèses la classe dont elle doit hériter.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Comme dit, dans un premier temps, toutes les méthodes de la classe ***A*** se retrouveront dans la classe ***B***.
- Quand une classe ***B*** hérite d'une classe ***A***, les objets de type ***B*** reprennent bel et bien les méthodes de la classe ***A*** en même temps que celles de la classe ***B***.
- Mais, assez logiquement, ce sont celles de la classe ***B*** qui sont appelées d'abord.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Si vous faites ***objet_de_type_b.ma_methode()***, Python va d'abord chercher la méthode ***ma_methode*** dans la classe ***B*** dont l'objet est directement issu.
- S'il ne trouve pas, il va chercher récursivement dans les classes dont hérite ***B***, c'est-à-dire ***A*** dans notre exemple.
- Ce mécanisme est très important : il induit que si aucune méthode n'a été redéfinie dans la classe, on cherche dans la classe mère.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- On peut ainsi redéfinir une certaine méthode dans une classe et laisser d'autres directement hériter de la classe mère.
- Petit code d'exemple :

```
1  class Personne:
2      """Classe représentant une personne"""
3      def __init__(self, nom):
4          """Constructeur de notre classe"""
5          self.nom = nom
6          self.prenom = "Martin"
7      def __str__(self):
8          """Méthode appelée lors d'une conversion de l'objet en
               chaîne"""
9          return "{0} {1}".format(self.prenom, self.nom)
10
```

```
11 class AgentSpecial(Personne):
12     """Classe définissant un agent spécial.
13     Elle hérite de la classe Personne"""
14
15     def __init__(self, nom, matricule):
16         """Un agent se définit par son nom et son matricule"""
17         self.nom = nom
18         self.matricule = matricule
19     def __str__(self):
20         """Méthode appelée lors d'une conversion de l'objet en
               chaîne"""
21         return "Agent {0}, matricule {1}".format(self.nom, self
               .matricule)
```

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Vous voyez ici un exemple d'héritage simple.
- Seulement, si on essaie de créer et manipuler des agents spéciaux, on risque d'avoir de drôles de surprises :

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> agent.nom
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prenom
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AgentSpecial' object has no attribute 'prenom'
>>>
```

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Mais une classe fille reprenait les méthodes et attributs de sa classe mère non ?
- Si. Mais en suivant bien l'exécution, vous allez comprendre : tout commence à la création de l'objet.
- Quel constructeur appeler ? S'il n'y avait pas de constructeur défini dans notre classe ***AgentSpecial***, Python appellerait celui de ***Personne***.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Mais il en existe bel et bien un constructeur dans la classe ***AgentSpecial*** et c'est donc celui-ci qui est appelé.
- Dans ce constructeur, on définit deux attributs, ***nom*** et ***matricule***.
- Mais c'est tout : le constructeur de la classe ***Personne*** n'est pas appelé, sauf si vous l'appelez explicitement dans le constructeur de la classe ***AgentSpecial***.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Dans le premier chapitre, on avait vu que ***mon_objet.ma_methode()*** revenait au même que ***MaClasse.ma_methode(mon_objet)***.
- Dans notre méthode ***ma_methode***, le premier paramètre ***self*** sera ***mon_objet***.
- Nous allons nous servir de cette équivalence.
- La plupart du temps, écrire ***mon_objet.ma_methode()*** suffit.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Mais dans une relation d'héritage, il peut y avoir, comme nous l'avons vu, plusieurs méthodes du même nom définies dans différentes classes.
- Laquelle appeler ?
- Python choisit, s'il la trouve, celle définie directement dans la classe dont est issu l'objet, et sinon parcourt la hiérarchie de l'héritage jusqu'à tomber sur la méthode.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Mais on peut aussi se servir de la notation ***MaClasse.ma_methode(mon_objet)*** pour appeler une méthode précise d'une classe précise.
- Et cela est utile dans notre cas :

```
1  class Personne:
2      """Classe représentant une personne"""
3      def __init__(self, nom):
4          """Constructeur de notre classe"""
5          self.nom = nom
6          self.prenom = "Martin"
7      def __str__(self):
8          """Méthode appelée lors d'une conversion de l'objet en
               chaîne"""
9          return "{0} {1}".format(self.prenom, self.nom)
10
```

```
11 class AgentSpecial(Personne):
12     """Classe définissant un agent spécial.
13     Elle hérite de la classe Personne"""
14
15     def __init__(self, nom, matricule):
16         """Un agent se définit par son nom et son matricule"""
17         # On appelle explicitement le constructeur de Personne
               :
18         Personne.__init__(self, nom)
19         self.matricule = matricule
20     def __str__(self):
21         """Méthode appelée lors d'une conversion de l'objet en
               chaîne"""
22         return "Agent {0}, matricule {1}".format(self.nom, self
               .matricule)
```

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Si cela vous paraît encore un peu vague, expérimentez : c'est toujours le meilleur moyen.
- Entraînez-vous, contrôlez l'écriture des attributs, ou revenez au premier chapitre de cette partie pour vous rafraîchir la mémoire au sujet du paramètre ***self***, bien qu'à force de manipulations vous avez dû comprendre l'idée.
- Reprenons notre code de tout à l'heure qui, cette fois, passe sans problème :

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

```
1  >>> agent = AgentSpecial("Fisher", "18327-121")
2  >>> agent.nom
3  'Fisher'
4  >>> print(agent)
5  Agent Fisher, matricule 18327-121
6  >>> agent.prenom
7  'Martin'
8  >>>
```

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Cette fois, notre attribut ***prenom*** se trouve bien dans notre agent spécial car le constructeur de la classe ***AgentSpecial*** appelle explicitement celui de ***Personne***.
- Vous pouvez noter également que, dans le constructeur de la classe ***AgentSpecial***, on n'instancie pas l'attribut ***nom***.
- Celui-ci est en effet écrit par le constructeur de la classe ***Personne*** que nous appelons en lui passant en paramètre le nom de notre agent.

4.2. IMPLÉMENTATION DE L'HÉRITAGE SIMPLE

- Notez que l'on pourrait très bien faire hériter une nouvelle classe de notre classe ***Personne***, la classe mère est souvent un modèle pour plusieurs classes filles.

4.3. QUELQUES FONCTIONS PRATIQUES

- Python définit deux fonctions qui peuvent se révéler utiles dans bien des cas : ***issubclass*** et ***isinstance***.

4.3. QUELQUES FONCTIONS PRATIQUES

1. issubclass

- Comme son nom l'indique, elle vérifie si une classe est une sous-classe d'une autre classe.
- Elle renvoie ***True*** si c'est le cas, ***False*** sinon :

4.3. QUELQUES FONCTIONS PRATIQUES

```
1 >>> issubclass(AgentSpecial, Personne) # AgentSpecial hérite de
      Personne
2 True
3 >>> issubclass(AgentSpecial, object)
4 True
5 >>> issubclass(Personne, object)
6 True
7 >>> issubclass(Personne, AgentSpecial) # Personne n'hérite pas
     d'AgentSpecial
8 False
9 >>>
```

4.3. QUELQUES FONCTIONS PRATIQUES

2. isinstance

- ***isinstance*** permet de savoir si un objet est issu d'une classe ou de ses classes filles :

4.3. QUELQUES FONCTIONS PRATIQUES

```
1 >>> agent = AgentSpecial("Fisher", "18327-121")
2 >>> isinstance(agent, AgentSpecial) # Agent est une instance d'
      AgentSpecial
3 True
4 >>> isinstance(agent, Personne) # Agent est une instance hérité
      e de Personne
5 True
6 >>>
```

4.4. L'HÉRITAGE MULTIPLE

- Python inclut un mécanisme permettant l'héritage multiple.
- L'idée est en substance très simple : au lieu d'hériter d'une seule classe, on peut hériter de plusieurs.

4.4.1. UTILITÉ DE L'HÉRITAGE MULTIPLE

- On peut s'asseoir dans un fauteuil. On peut dormir dans un lit. Mais on peut s'asseoir et dormir dans certains canapés.
- Notre classe ***Fauteuil*** pourra hériter de la classe ***ObjetPourSAsseoir*** et notre classe ***Lit***, de notre classe ***ObjetPourDormir***.
- Mais notre classe ***Canape*** alors ? Elle devra logiquement hériter de nos deux classes ***ObjetPourSAsseoir*** et ***ObjetPourDormir***.

4.4.1. UTILITÉ DE L'HÉRITAGE MULTIPLE

- C'est un cas où l'héritage multiple pourrait se révéler utile.
- Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités définies dans une classe mère.
- Par exemple, une classe peut produire des objets destinés à être enregistrés dans des fichiers.

4.4.1. UTILITÉ DE L'HÉRITAGE MULTIPLE

- C'est un cas où l'héritage multiple pourrait se révéler utile.
- Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités définies dans une classe mère.
- Par exemple, une classe peut produire des objets destinés à être enregistrés dans des fichiers.

4.4.1. UTILITÉ DE L'HÉRITAGE MULTIPLE

- Mais ces mêmes classes pourront hériter d'autres classes incluant, pourquoi pas, d'autres fonctionnalités.
- C'est une des utilisations de l'héritage multiple et il en existe d'autres.

4.4.1. UTILITÉ DE L'HÉRITAGE MULTIPLE

- Mais ces mêmes classes pourront hériter d'autres classes incluant, pourquoi pas, d'autres fonctionnalités.
- C'est une des utilisations de l'héritage multiple et il en existe d'autres.

4.4.2. IMPLÉMENTATION DE L'HÉRITAGE MULTIPLE

- La syntaxe permettant d'implémenter l'héritage multiple en Python est la suivante :

```
1 class MaClasseHeritee(MaClasseMere1, MaClasseMere2):
```

4.4.2. IMPLÉMENTATION DE L'HÉRITAGE MULTIPLE

- Vous pouvez faire hériter votre classe de plus de deux autres classes.
- Au lieu de préciser, comme dans les cas d'héritage simple, une seule classe mère entre parenthèses, vous en indiquez plusieurs, séparées par des virgules.

4.5. EN RÉSUMÉ

- L'héritage permet à une classe d'hériter du comportement d'une autre en reprenant ses méthodes. La syntaxe de l'héritage est :

class NouvelleClasse(ClasseMere) :

- On peut accéder aux méthodes de la classe mère directement à partir de la syntaxe :

ClasseMere.methode(self)

4.5. EN RÉSUMÉ

- L'héritage multiple permet à une classe d'hériter de plusieurs classes mères.

La syntaxe de l'héritage multiple s'écrit donc de la manière suivante :

class NouvelleClasse(ClasseMere1, ClasseMere2, ClasseMereN):

PARTIE 3 : LA PROGRAMMATION DES INTERFACES GRAPHIQUES

PARTIE 3 : LA PROGRAMMATION DES INTERFACES GRAPHIQUES

- Nous allons maintenant voir comment créer des interfaces graphiques à l'aide d'un module présent par défaut dans Python : ***Tkinter***.
- Ce module permet de créer des interfaces graphiques en offrant une passerelle entre Python et la bibliothèque ***Tk***.
- Nous allons pouvoir apprendre dans ce chapitre à :
  - Créer des fenêtres ;

PARTIE 3 : LA PROGRAMMATION DES INTERFACES GRAPHIQUES

- Créer des widgets (boutons, champs de saisie, etc.) ;
- Faire réagir nos objets graphiques à certains évènements (faire une action lors d'un clic sur un bouton, etc.) ;
- Etc.

3.1. PRÉSENTATION DE TKINTER

- ***Tkinter*** (***Tk interface***) est un module intégré à la bibliothèque standard de Python.
- Il offre un moyen de créer des interfaces graphiques via Python.
- ***Tkinter*** est disponible sur Windows et la plupart des systèmes Unix.
- Les interfaces que vous pourrez développer auront donc toutes les chances d'être portables d'un système à l'autre.

3.1. PRÉSENTATION DE TKINTER

- Notez qu'il existe d'autres bibliothèques pour créer des interfaces graphiques en Python.
- ***Tkinter*** a l'avantage d'être disponible par défaut, sans nécessiter une installation supplémentaire.
- Pour savoir si vous pouvez utiliser le module ***Tkinter*** via la version de Python installée sur votre système, tapez dans l'interpréteur en ligne de commande Python :

```
1 from tkinter import *
```

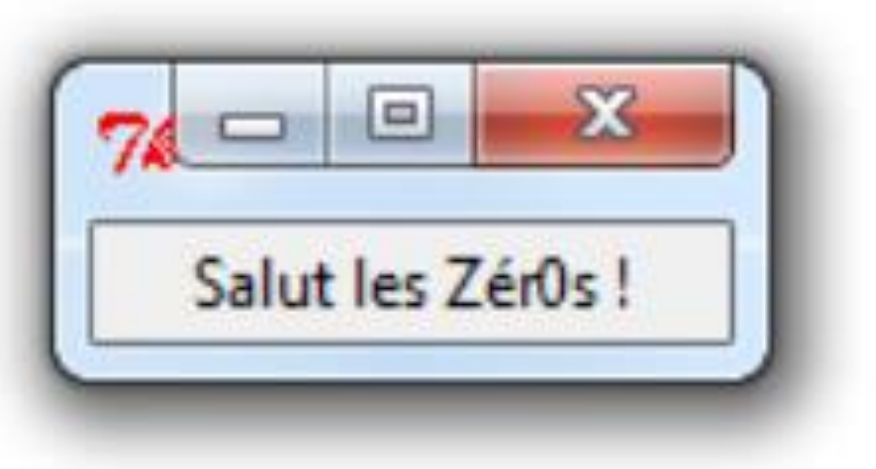
3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Nous allons commencer par voir le code minimal pour créer une fenêtre avec ***Tkinter***.
- Petit à petit, nous allons apprendre à rajouter des choses, mais commençons par voir la base de code que l'on retrouve d'une interface ***Tkinter*** à l'autre.
- Étant en Python, ce code minimal est plutôt court :

```python
"""Premier exemple avec Tkinter.

On crée une fenêtre simple qui souhaite la bienvenue à l'
    utilisateur.

"""

# On importe Tkinter
from tkinter import *

# On crée une fenêtre, racine de notre interface
fenetre = Tk()

# On crée un label (ligne de texte) souhaitant la bienvenue
# Note : le premier paramètre passé au constructeur de Label
    est notre
# interface racine
champ_label = Label(fenetre, text="Salut les Zér0s !")

# On affiche le label dans la fenêtre
champ_label.pack()

# On démarre la boucle Tkinter qui s'interompt quand on ferme
    la fenêtre
fenetre.mainloop()
```

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Vous pouvez voir le résultat à la figure suivante :



3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Vous pouvez ensuite exécuter votre programme, ce qui affiche une fenêtre (simple, certes, mais c'une fenêtre).
- Comme vous pouvez le voir, la fenêtre est tout juste assez grande pour que le message s'affiche.

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Regardons le code d'un peu plus près :

1. On commence par importer ***Tkinter***, sans grande surprise.

2. On crée ensuite un objet de la classe ***Tk***. La plupart du temps, cet objet sera la fenêtre principale de notre interface.

3. On crée un ***Label***, c'est-à-dire un objet graphique affichant du texte.

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

4. On appelle la méthode ***pack*** de notre ***Label***. Cette méthode permet de positionner l'objet dans notre fenêtre (et, par conséquent, de l'afficher).

5. Enfin, on appelle la méthode ***mainloop*** de notre fenêtre racine. Cette méthode ne retourne que lorsqu'on ferme la fenêtre.

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Quelques petites précisions :
- Nos objets graphiques (boutons, champs de texte, cases à cocher, barres de progression, ...) sont appelés des ***widgets***.
- On peut préciser plusieurs options lors de la construction de nos widgets.
- Ici, on définit l'option ***text*** de notre ***Label*** à ***"Salut les Zér0s !"***.

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Il existe d'autres options communes à la plupart des widgets (la couleur de fond ***bg***, la couleur du widget ***fg***, etc.) et d'autres plus spécifiques à un certain type de widget.
- Le ***Label*** par exemple possède l'option ***text*** représentant le texte affiché par le ***Label***.
- Comme nous l'avons vu, vous pouvez modifier des options lors de la création du widget.

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Mais vous pouvez aussi en modifier après :

```
>>> champ_label["text"]
'Salut les ZérOs !'
>>> champ_label["text"] = "Maintenant, au revoir !"
>>> champ_label["text"]
'Maintenant, au revoir !'
>>>
```

3.2. NOTRE PREMIÈRE INTERFACE GRAPHIQUE

- Comme vous le voyez, vous passez entre crochets (comme pour accéder à une valeur d'un ***dictionnaire***) le nom de l'option.
- C'est le même principe pour accéder à la valeur actuelle de l'option ou pour la modifier.
- Nous allons voir quelques autres widgets de ***Tkinter*** à présent.

3.3. LES WIDGETS

- Comme vous le voyez, vous passez entre crochets (comme pour accéder à une valeur d'un ***dictionnaire***) le nom de l'option.
- C'est le même principe pour accéder à la valeur actuelle de l'option ou pour la modifier.
- Nous allons voir quelques autres widgets de ***Tkinter*** à présent.

3.3. LES WIDGETS

- ***Tkinter*** définit un grand nombre de widgets pouvant être utilisés dans notre fenêtre.
- Nous allons voir quelques-uns dans cette section.

3.3.1. LES WIDGETS LES PLUS COMMUNS

3.3.1.1. LES LABELS

- C'est le premier widget que nous avons vu, hormis notre fenêtre principale qui en est un également.
- On s'en sert pour afficher du texte dans notre fenêtre, du texte qui ne sera pas modifié par l'utilisateur.

```
1 champ_label = Label(fenetre, text="contenu de notre champ label
      ")
2 champ_label.pack()
```

3.3.1.1. LES LABELS

- N'oubliez pas que, pour qu'un widget apparaisse, il faut :
  - qu'il prenne, en premier paramètre du constructeur, la fenêtre principale ;
  - qu'il fasse appel à la méthode ***pack***.
- La méthode ***pack*** permet de positionner un objet dans une fenêtre ou dans un cadre, nous verrons plus loin quelques-uns de ses paramètres optionnels.

3.3.1.2. LES BOUTONS

- Les boutons sont des widgets sur lesquels on peut cliquer et qui peuvent déclencher des actions ou commandes comme nous le verrons ultérieurement plus en détail.

```
1 bouton_quitter = Button(fenetre, text="Quitter", command=
      fenetre.quit)
2 bouton_quitter.pack()
```

3.3.1.2. LES BOUTONS

- Sûrement que vous vous posez des questions sur le dernier paramètre passé à notre constructeur de Button.
- Il s'agit de l'action liée à un clic sur le bouton.
- Ici, c'est la méthode ***quit*** de notre fenêtre racine qui est appelée.
- Ainsi, quand vous cliquez sur le bouton ***Quitter***, la fenêtre se ferme.
- Nous verrons plus tard comment créer nos propres commandes.

3.3.1.3. UNE ZONE DE SAISIE

- Le widget que nous allons voir à présent est une zone de texte dans lequel l'utilisateur peut écrire.
- On va créer une variable ***Tkinter*** associée au champ de texte.
- Regardez le code qui suit :

```
1 var_texte = StringVar()
2 ligne_texte = Entry(fenetre, textvariable=var_texte, width=30)
3 ligne_texte.pack()
```

3.3.1.3. UNE ZONE DE SAISIE

- À la ligne 1, nous créons une variable ***Tkinter***.
- En résumé, c'est une variable qui va ici contenir le texte de notre ***Entry*** (zone de saisie).
- Il est possible de lier cette variable à une méthode de telle sorte que la méthode soit appelée quand la variable est modifiée (l'utilisateur écrit dans le champ ***Entry***).

3.3.1.3. UNE ZONE DE SAISIE

- Le widget ***Entry*** n'est qu'une zone de saisie.
- Pour que l'utilisateur sache ce qu'il doit y écrire, il pourrait être utile de lui mettre une indication auprès du champ.
- Le widget ***Label*** est le plus approprié dans ce cas.
- Notez qu'il existe également le widget ***Text*** qui représente un champ de texte à plusieurs lignes.

3.3.1.4. LES CASES À COCHER

- Les cases à cocher sont définies dans la classe ***Checkbutton***.
- Là encore, on utilise une variable pour surveiller la sélection de la case.
- Pour surveiller l'état d'une case à cocher (qui peut être soit active soit inactive), on préférera créer une variable de type ***IntVar*** plutôt que ***StringVar***, bien que ce ne soit pas une obligation.

3.3.1.4. LES CASES À COCHER

```
1 var_case = IntVar()
2 case = Checkbutton(fenetre, text="Ne plus poser cette question"
      , variable=var_case)
3 case.pack()
```

3.3.1.4. LES CASES À COCHER

- Vous pouvez ensuite contrôler l'état de la case à cocher en interrogeant la variable :

```
1 var_case.get()
```

- Si la case est cochée, la valeur renvoyée par la variable sera 1 ; si elle n'est pas cochée, ce sera 0.

3.3.1.4. LES CASES À COCHER

- Notez qu'à l'instar d'un bouton, vous pouvez lier la case à cocher à une commande qui sera appelée quand son état change.

3.3.1.5. LES BOUTONS RADIO

- Les boutons radio sont des boutons généralement présentés en groupes.
- C'est, à proprement parler, un ensemble de cases à cocher mutuellement exclusives : quand vous cliquez sur l'un des boutons, celui-ci se sélectionne et tous les autres boutons du même groupe se désélectionnent.
- Ce type de bouton est donc surtout utile dans le cadre d'un regroupement.

3.3.1.5. LES BOUTONS RADIO

- Pour créer un groupe de boutons, il faut simplement qu'ils soient tous associés à la même variable (là encore, une variable ***Tkinter***).
- La variable peut posséder le type que vous voulez.
- Quand l'utilisateur change le bouton sélectionné, la valeur de la variable change également en fonction de l'option ***value*** associée au bouton.
- Voyons un exemple :

3.3.1.5. LES BOUTONS RADIO

- Pour créer un groupe de boutons, il faut simplement qu'ils soient tous associés à la même variable (là encore, une variable ***Tkinter***).
- La variable peut posséder le type que vous voulez.
- Quand l'utilisateur change le bouton sélectionné, la valeur de la variable change également en fonction de l'option ***value*** associée au bouton.
- Voyons un exemple :

```
var_choix = StringVar()

choix_rouge = Radiobutton(fenetre, text="Rouge", variable=
    var_choix, value="rouge")
choix_vert = Radiobutton(fenetre, text="Vert", variable=
    var_choix, value="vert")
choix_bleu = Radiobutton(fenetre, text="Bleu", variable=
    var_choix, value="bleu")

choix_rouge.pack()
choix_vert.pack()
choix_bleu.pack()
```

3.3.1.5. LES BOUTONS RADIO

- Pour récupérer la valeur associée au bouton actuellement sélectionné, interrogez la variable :

```
1 | var_choix.get()
```

3.3.1.6. LES LISTES DÉROULANTES

- Ce widget permet de construire une liste dans laquelle on peut sélectionner un ou plusieurs éléments.
- Le fonctionnement n'est pas tout à fait identique aux boutons radio.
- Ici, la liste comprend plusieurs lignes et non un groupe de boutons.

3.3.1.6. LES LISTES DÉROULANTES

- Créer une liste se fait assez simplement, vous devez commencer à vous habituer à la syntaxe :

```
1 liste = Listbox(fenetre)
2 liste.pack()
```

3.3.1.6. LES LISTES DÉROULANTES

- On insère ensuite des éléments en utilisant la méthode ***insert***.
- La méthode insert prend deux paramètres :

1. la position à laquelle insérer l'élément ;
2. l'élément même, sous la forme d'une chaîne de caractères.

3.3.1.6. LES LISTES DÉROULANTES

- Si vous voulez insérer des éléments à la fin de la liste, utilisez la constante END définie par Tkinter :

```
1 liste.insert(END, "Pierre")
2 liste.insert(END, "Feuille")
3 liste.insert(END, "Ciseau")
```

3.3.1.6. LES LISTES DÉROULANTES

- Pour accéder à la sélection, utilisez la méthode ***curselection*** de la liste.
- Elle renvoie un tuple de chaînes de caractères, chacune étant la position de l'élément sélectionné.
- Par exemple, si ***liste.curselection()*** renvoie le tuple ***('2',)***, c'est le troisième élément de la liste qui est sélectionné (***Ciseau*** en l'occurrence).

3.3.2. ORGANISATION DES WIDGETS DANS LA FENÊTRE

- Il existe plusieurs widgets qui peuvent contenir d'autres widgets.
- L'un d'entre eux se nomme ***Frame***.
- C'est un cadre rectangulaire dans lequel vous pouvez placer vos widgets... ainsi que d'autres objets ***Frame*** si besoin est.

3.3.2. ORGANISATION DES WIDGETS DANS LA FENÊTRE

- Si vous voulez qu'un widget apparaisse dans un cadre, utilisez le ***Frame*** comme parent à la création du widget :

```
1 cadre = Frame(fenetre, width=768, height=576, borderwidth=1)
2 cadre.pack(fill=BOTH)
3
4 message = Label(cadre, text="Notre fenêtre")
5 message.pack(side="top", fill=X)
```

3.3.2. ORGANISATION DES WIDGETS DANS LA FENÊTRE

- Comme vous le voyez, nous avons passé plusieurs arguments nommés à notre méthode ***pack***.
- Cette méthode, comme on l'avait vu, sert à placer nos widgets dans la fenêtre (ici, dans le cadre).
- En précisant ***side="top"***, on demande à ce que le widget soit placé en haut de son parent (ici, notre cadre).

3.3.2. ORGANISATION DES WIDGETS DANS LA FENÊTRE

- Il existe aussi l'argument nommé ***fill*** qui permet au widget de remplir le widget parent, soit en largeur si la valeur est ***X***, soit en hauteur si la valeur est ***Y***, soit en largeur et hauteur si la valeur est ***BOTH***.
- D'autres arguments nommés existent, bien entendu.
- Si vous voulez une liste exhaustive, rendez-vous sur le chapitre consacré à ***Tkinter*** dans la documentation officielle de Python.

3.3.2. ORGANISATION DES WIDGETS DANS LA FENÊTRE

- Notez qu'il existe aussi le widget ***Labelframe***, un cadre avec un titre, ce qui nous évite d'avoir à placer un label en haut du cadre.

- Il se construit comme un ***Frame*** mais peut prendre en argument, à la construction, le texte représentant le titre :

cadre = Labelframe(..., text="Titre du cadre")

3.3.3. D'AUTRES WIDGETS

- Vous devez vous en douter, ceci n'est qu'une approche très sommaire de quelques widgets de Tkinter.
- Il en existe de nombreux autres et ceux que nous avons vus ont bien d'autres options.
- Il est notamment possible de créer :
  - une barre de menus avec ses menus imbriqués ;

3.3.3. D'AUTRES WIDGETS

- d'afficher des images ;
- des canvas dans lequel vous pouvez dessiner pour personnaliser votre fenêtre ;
- Etc.

- Bref, il vous reste bien des choses à voir, même si ce chapitre ne peut pas couvrir tous ces widgets et options.

PARTIE 4 : PYTHON ET BASE DE DONNÉES (MYSQL)

PARTIE 4 : PYTHON ET BASE DE DONNÉES (MYSQL)

- Dans cette partie, nous apprendrons à faire communiquer un programme Python avec une base de données.
- Nous apprendrons à :
  - installer un SGBD (MySQL) ;
  - créer des bases de données ;
  - créer des tables ;

PARTIE 4 : PYTHON ET BASE DE DONNÉES (MYSQL)

- insérer des données dans les tables ;
- modifier des données dans les tables ;
- supprimer des données des tables ;
- Sélectionner des données dans les tables ;
- et bien plus encore à partir d'un Programme Python.

PARTIE 4 : PYTHON ET BASE DE DONNÉES (MYSQL)

- Ce cours n'est pas destiné à vous enseigner la syntaxe ou le langage ***SQL*** complet.
- Ce cours vous apprendra comment vous pouvez travailler avec ***MySQL*** en Python.
- Cf. l'UE ***« IT 240 : Introduction aux Bases de Données UML-SQL »*** pour le cours sur le langage SQL.

6.1. INSTALLATION DE PYTHON

- MySQL est l'un des Systèmes de Gestion de Bases de Données (SGBD) les plus populaires et les plus utilisés.
- Téléchargez et installez MySQL depuis le site officiel de MySQL.
- Vous devez installer le serveur MySQL pour suivre ce tutoriel.
- Ensuite, vous devez installer la bibliothèque ***mysql.connector*** pour Python afin de pouvoir se connecter à MySQL à partir de Python.

6.1. INSTALLATION DE PYTHON

- Téléchargez la bibliothèque ***mysql.connector*** à partir de ce https://dev.mysql.com/downloads/connector/python/ et installez-le sur votre ordinateur.

- Maintenant, vérifiez si vous avez correctement installé ***mysql.connector*** en utilisant le code suivant :

```
import mysql.connector
```

6.1. INSTALLATION DE PYTHON

- Si le code ci-dessus s'exécute sans aucune erreur, alors vous avez installé avec succès ***mysql.connector*** et il est prêt à être utilisé.

6.2. CONNEXION AU SGBD MYSQL

- Maintenant, nous allons nous connecter au SGBD MySQL en utilisant le nom d'utilisateur et le mot de passe de MySQL.
- Si vous ne vous souvenez plus de votre nom d'utilisateur ou de votre mot de passe, créez un nouvel utilisateur avec un mot de passe.
- Pour créer un nouvel utilisateur, consulter le cours ***IT 240*** ou la documentation officielle de MySQL.

6.2. CONNEXION AU SGBD MYSQL

- La méthode ***connect()*** va nous permettre de nous connecter au SGBD MySQL et prend trois (3) paramètres obligatoires :
  - L'adresse du serveur sur lequel le SGBD est installé ;
  - Le nom d'utilisateur ;
  - Le mot de passe de l'utilisateur.
- L'utilisation de la méthode ***connect()*** est illustrée à la figure suivante :

```
## Connecting to the database


## importing 'mysql.connector' as mysql for convenient
import mysql.connector as mysql


## connecting to the database using 'connect()' method
## it takes 3 required parameters 'host', 'user', 'passwd'
db = mysql.connect(
    host = "localhost",
    user = "root",
    passwd = "dbms"
)


print(db) # it will print a connection object if everything is fine
```

```
<mysql.connector.connection_cext.CMySQLConnection object at 0x0000020C26A84C50>
```

6.2. CONNEXION AU SGBD MYSQL

- Nous sommes maintenant connecté au SGBD MySQL.
- Nous pouvons maintenant envoyer des requêtes au SGBD MySQL à partir d'un script Python.

6.3. CRÉATION D'UNE BASE DE DONNÉES

- Maintenant, nous allons créer une base de données avec le nom ***datacamp*** sur le SGBD.

- Pour créer une base de données dans MySQL, nous utilisons l'instruction suivante :

CREATE DATABASE database_name

```python
import mysql.connector as mysql


db = mysql.connect(
    host = "localhost",
    user = "root",
    passwd = "dbms"
)


## creating an instance of 'cursor' class which is used to execute the 'SQL' statements in 'Python'
cursor = db.cursor()


## creating a databse called 'datacamp'
## 'execute()' method is used to compile a 'SQL' statement
## below statement is used to create tha 'datacamp' database
cursor.execute("CREATE DATABASE datacamp")
```

6.3. CRÉATION D'UNE BASE DE DONNÉES

- Si la base de données existe déjà, vous obtiendrez une erreur.
- Assurez-vous que la base de données n'existe pas.

6.4. CRÉATION DES TABLES

- Nous allons passer maintenant à la création de tables dans la base de données pour stocker les informations.
- Avant de créer des tables, nous devons d'abord sélectionner une base de données.
- Exécutez le code suivant pour sélectionner la base de données ***datacamp*** que nous avons créée une minute auparavant :

```
import mysql.connector as mysql


db = mysql.connect(
    host = "localhost",
    user = "root",
    passwd = "dbms",
    database = "datacamp"
)
```

6.4. CRÉATION DES TABLES

- Le code ci-dessus s'exécutera sans erreur si la base de données existe.
- Maintenant, vous êtes connecté à la base de données appelée ***datacamp***.
- Utilisez ***CREATE TABLE nom_table*** pour créer une table dans la base de données sélectionnée.
- Illustration avec le code suivant :

```
import mysql.connector as mysql


db = mysql.connect(
    host = "localhost",
    user = "root",
    passwd = "dbms",
    database = "datacamp"
)


cursor = db.cursor()


## creating a table called 'users' in the 'datacamp' database
cursor.execute("CREATE TABLE users (name VARCHAR(255), user_name VARCHAR(255))")
```

6.4. CRÉATION DES TABLES

- Vous avez créé avec succès la table ***users*** dans la base de données ***datacamp***.

- Voir toutes les tables présentes dans la base de données à l'aide de l'instruction ***SHOW TABLES*** :

```
cursor = db.cursor()


## getting all the tables which are present in 'datacamp' database
cursor.execute("SHOW TABLES")


tables = cursor.fetchall() ## it returns list of tables present in the database


## showing all the tables one by one
for table in tables:
    print(table)
```

6.4.1. AFFICHAGE DE LA STRUCTURE D'UNE TABLE

- Pour afficher la structure d'une table, il faut utiliser la syntaxe ***DESC table_name*** comme illustrée à la figure suivante :

6.4.1. AFFICHAGE DE LA STRUCTURE D'UNE TABLE

```
cursor = db.cursor()


## 'DESC table_name' is used to get all columns information
cursor.execute("DESC users")


## it will print all the columns as 'tuples' in a list
print(cursor.fetchall())
```

6.4.2. SUPPRESSION D'UNE COLONNE D'UNE TABLE

- Nous utilisons l'instruction ***ALTER TABLE nom_table DROP nom_colonne*** pour supprimer une colonne dans une table.

```
cursor = db.cursor()


## dropping the 'id' column
cursor.execute("ALTER TABLE users DROP id")


cursor.execute("DESC users")


print(cursor.fetchall())
```

6.4.2. AJOUT D'UNE COLONNE DANS UNE TABLE

- Illustration de l'ajout d'une colonne avec la figure suivante :

6.4.2. AJOUT D'UNE COLONNE DANS UNE TABLE

```
cursor = db.cursor()


## adding 'id' column to the 'users' table
## 'FIRST' keyword in the statement will add a column in the starting of the table
cursor.execute("ALTER TABLE users ADD COLUMN id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST")


cursor.execute("DESC users")


print(cursor.fetchall())
```

6.4.2. AJOUT D'UNE COLONNE DANS UNE TABLE

- Nous avons ajouté la colonne ***id*** à la table ***users***.

6.5. INSERTION DE DONNÉES DANS UNE TABLE

- Place maintenant à l'insertion des données dans une table pour les stocker.
- Utilisez l'instruction ***INSERT INTO nom_table (noms_colonne) VALUES (données)*** pour insérer des données dans la table.

6.5.1. INSERTION D'UNE LIGNE DE DONNÉES DANS UNE TABLE

- Ci-dessous, un exemple d'insertion d'une ligne de données dans une table :

```
cursor = db.cursor()


## defining the Query
query = "INSERT INTO users (name, user_name) VALUES (%s, %s)"
## storing values in a variable
values = ("Hafeez", "hafeez")


## executing the query with values
cursor.execute(query, values)


## to make final output we have to run the 'commit()' method of the database object
db.commit()


print(cursor.rowcount, "record inserted")
```

6.5.1. INSERTION D'UNE LIGNE DE DONNÉES DANS UNE TABLE

- Le code ci-dessus insère une ligne dans la table ***users***.

6.5.2. INSERTION DE PLUSIEURS LIGNES DE DONNÉES DANS UNE TABLE

- Voyons comment insérer plusieurs lignes dans le tableau.
- Pour insérer plusieurs lignes dans la table, nous utilisons la méthode ***executemany()***.
- Il prend une liste de tuples contenant les données comme deuxième paramètre et une requête comme premier argument.

```
cursor = db.cursor()

## defining the Query
query = "INSERT INTO users (name, user_name) VALUES (%s, %s)"
## storing values in a variable
values = [
    ("Peter", "peter"),
    ("Amy", "amy"),
    ("Michael", "michael"),
    ("Hennah", "hennah")
]

## executing the query with values
cursor.executemany(query, values)

## to make final output we have to run the 'commit()' method of the database object
db.commit()
```

6.5.2. INSERTION DE PLUSIEURS LIGNES DE DONNÉES DANS UNE TABLE

- Le code ci-dessus a inséré quatre enregistrements dans la table ***users***.

6.6. SÉLECTION DES DONNÉES DANS UNE TABLE

- Pour récupérer les données d'une table nous utiliserons l'instruction suivante :

SELECT column_names FROM table_name.

6.6.1. SÉLECTION DE TOUS LES ENREGISTREMENTS DANS UNE TABLE

- Pour obtenir tous les enregistrements d'une table, nous utilisons * à la place des noms de colonnes et sans les clauses de filtrage.

- Récupérons toutes les données de la table ***users*** que nous avons insérée auparavant avec l'instruction suivante :

```
cursor = db.cursor()


## defining the Query

query = "SELECT * FROM users"


## getting records from the table

cursor.execute(query)


## fetching all records from the 'cursor' object

records = cursor.fetchall()


## Showing the data

for record in records:

    print(record)
```

6.6.2. SÉLECTION DE CERTAINES COLONNES DANS UNE TABLE

- Pour sélectionner certaines colonnes de la table, mentionnez le nom de la colonne après le SELECT dans l'instruction.
- S'il y a plusieurs colonnes à récupérer, mentionnez leurs noms séparer par des virgules.
- Récupérons la colonne ***user_name*** de la table ***users***.

```
cursor = db.cursor()


## defining the Query

query = "SELECT user_name FROM users"


## getting 'user_name' column from the table

cursor.execute(query)


## fetching all usernames from the 'cursor' object

usernames = cursor.fetchall()


## Showing the data

for username in usernames:

    print(username)
```

6.6.2. SÉLECTION DE CERTAINES COLONNES DANS UNE TABLE

- Vous pouvez complétez la requête SELECT avec les clauses ci-dessous :
  - WHERE ;
  - ORDER BY ;
  - Etc.

6.7. SUPPRESSIONS DES ENREGISTREMENTS DANS UNE TABLE

- Le mot clé ***DELETE*** est utilisé pour supprimer les enregistrements dans une table.
- L'instruction ***DELETE FROM table_name WHERE condition*** est utilisée pour supprimer des enregistrements d'une table.
- Si vous ne spécifiez pas la condition, tous les enregistrements seront supprimés de la table.

6.7. SUPPRESSIONS DES ENREGISTREMENTS DANS UNE TABLE

- Supprimons un enregistrement de la table ***users*** avec l'***id 5***.

```
cursor = db.cursor()


## defining the Query

query = "DELETE FROM users WHERE id = 5"


## executing the query

cursor.execute(query)


## final step to tell the database that we have changed the table data

db.commit()
```

6.8. MISES À JOUR DES ENREGISTREMENTS DANS UNE TABLE

- Le mot clé ***UPDATE*** est utilisé pour mettre à jour les données d'un ou de plusieurs enregistrements.
- L'instruction ***UPDATE nom_table SET nom_colonne = nouvelle_valeur WHERE condition*** est utilisée pour mettre à jour la valeur d'une ligne spécifique.
- Mettons à jour le nom du 1er enregistrement de Hafeez à Kareem.

6.8. MISES À JOUR DES ENREGISTREMENTS DANS UNE TABLE

- Le mot clé ***UPDATE*** est utilisé pour mettre à jour les données d'un ou de plusieurs enregistrements.
- L'instruction ***UPDATE nom_table SET nom_colonne = nouvelle_valeur WHERE condition*** est utilisée pour mettre à jour la valeur d'une ligne spécifique.
- Mettons à jour le nom du 1er enregistrement de Hafeez à Kareem.

```
cursor = db.cursor()


## defining the Query

query = "UPDATE users SET name = 'Kareem' WHERE id = 1"


## executing the query

cursor.execute(query)


## final step to tell the database that we have changed the table data

db.commit()
```