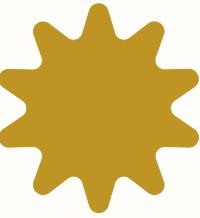


POLYTECH SORBONNE

INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET



Millan Mégane
(megane.millan@inria.fr)
2024 - 2025



PRÉSENTATION DU COURS

Objectif du cours

- Savoir utiliser git et les environnements virtuels
- Comprendre et maîtriser les concepts de base de python
- Découvrir la Programmation Orientée Objet (POO)

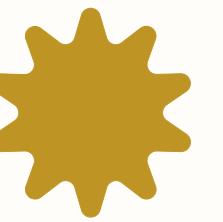
Déroulement

- 4 séances de cours (2h)
- 4 séances de TP (4h)
- 1 séance d'évaluation (1h)

Évaluation

- 60% TP
- 40% QCM



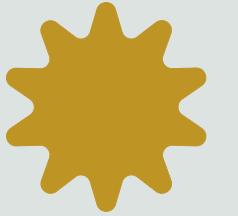


PROGRAMME & CONTENU

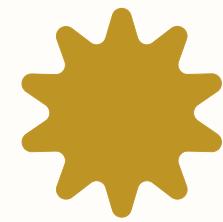
Utilisation de git et
des environnement
de programmation
pour python.

Introduction à Python

Introduction à la
POO



INTRODUCTION AUX OUTILS DE PROGRAMMATION - GIT ET ENVIRONNEMENT VIRTUEL



GIT

QU'EST-CE QUE GIT ?

- **Définition**

- Système de contrôle de version distribué qui permet de suivre les modifications dans le code source tout au long du développement d'un projet.

POURQUOI GIT ?

- **Gestion Efficace des Versions**

- Traçabilité - Chaque changement est sauvegardé.
- Sauvegarde - Facile de revenir à une version stable du code.

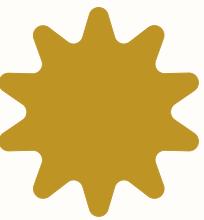
- **Facilitation de la Collaboration**

- Travail en équipe et fusion facile des différents développements.

- **Expérimentation et Innovation**

- Branches de Fonctionnalités - Possibilité de créer des branches pour expérimenter de nouvelles fonctionnalités sans risquer de casser le reste du code





ENVIRONNEMENT VIRTUEL

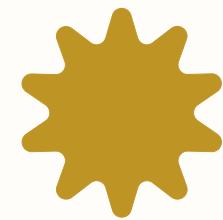
ENVIRONNEMENT VIRTUEL ?

- **Définition**
 - Un environnement virtuel est un espace isolé pour un projet Python où vous pouvez installer des paquets spécifiques sans interférer avec les autres projets ou la configuration globale du système

POURQUOI ?

- **Isolation des Dépendances**
 - Eviter les conflits entre différentes versions de paquets utilisés dans différents projets.
- **Reproductibilité**
 - Assurer que d'autres développeurs ou environnements peuvent reproduire exactement votre environnement.
- **Gestion Facile des Projets**
 - Faciliter la gestion de projets avec des dépendances spécifiques.





INSTALLATION D'UN ENVIRONNEMENT VIRTUEL - CONDA

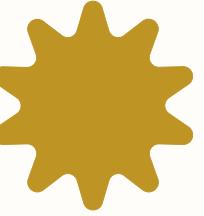
QU'EST-CE QUE CONDA ?

- Gestionnaire de paquets open-source et un système de gestion d'environnements, qui fonctionne sur **Windows, macOS et Linux**.
- Installation, mise à jour et gestion des logiciels et des bibliothèques, ainsi que création des environnements virtuels isolés.

POURQUOI CONDA ?

- **Gestion des Paquets facile**
 - Gestion des paquets Python, mais aussi d'autres langages comme R, Ruby, Lua, Scala, Java, JavaScript, C/C++, FORTRAN, etc.
- **Gestion des Dépendances**
 - Résout automatiquement les dépendances de paquets, évitant les conflits.
- **Large Écosystème**
 - Distribution populaire pour la science des données et le machine learning, avec plus de 7,500 paquets scientifiques et analytiques.



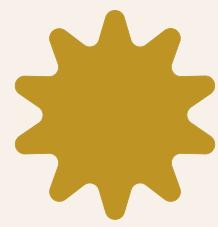


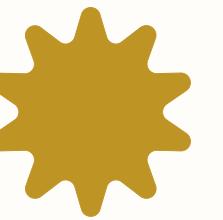
INSTALLATION DE CONDA

Lien vers [Installation Conda](#)



INTRODUCTION PYTHON





PYTHON

PYTHON ?

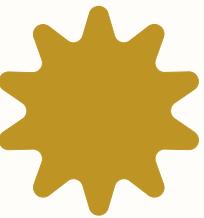
- langage polyvalent et haut niveau
- Syntaxe simple et lisible
- Langage interprété, facilitant le développement rapide

SYNTAXE ?

- chaque instruction occupe une ligne
- Indentation pour différencier les blocs (for, if, while...)
- Autorise la manipulation de type sans déclaration

```
a = 7.5 # déclaration et affectation d'un nombre
b = 'toto' # affectation d'une chaîne de caractères
if(a < 4):
    print('a is small')
```





EXAMPLE DE CODE PYTHON

Calcul des racines d'un polynôme du second degré

```
from cmath import sqrt

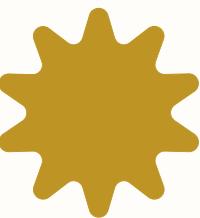
#Coefficient
a = 2
b = 4
c = -5

# Discriminant
delta = b**2 - 4*a*c

# Calcul des racines
x1 = (-b + sqrt(delta)) / (2*a)
x2 = (b + sqrt(delta)) / (2*a)

print("Les racines sont ", x1, " et ", x2)
```





TYPES DE DONNÉES

- **Type Numérique**

- Entier (int), flottant (float) et complexe (complex)

- **Type Chaine de Caractères**

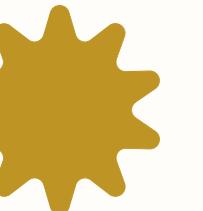
- Délimité par des guillemets simple ('') ou double (")

- **Type Booléen**

- Deux valeurs possibles True ou False

```
a = 2  
f = 5.4  
c = 4 + 6j  
s = "Hello World !"  
b = True
```





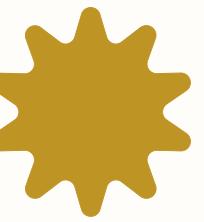
LES LISTES

- Séquence **ordonnée** de valeur
 - Éléments accessibles par leurs indices (0...n)
 - Taille accessible via la fonction len()

```
l = [1, 7, 3, -1, 9]
```

```
print(len(l)) #5
```





LES LISTES

OPÉRATIONS POSSIBLES

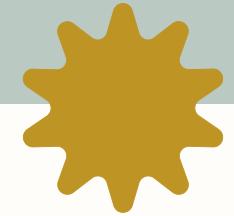
- Accès ([])
- Concaténation (+)
- Répétition (*)
- Appartenance (in)
- Comparaison
- Sous-liste ([:])
- Suppression (del)

```
l = [1, 7, 3, -1, 9]

print(l[0])
print(7 in l)
del(l[2])
print(l[0:3]) # équivalent à l[:3]

k = [5, 1]
print(k * 2)
print(l + k)
print(k == [5, 1])
```





Un peu de pratique

Vous êtes un assistant du maire du village médiéval de PyTown. Votre mission est d'aider à gérer les informations sur les habitants, leurs biens et leur participation à la vie du village à travers plusieurs événements. Le maire souhaite une liste des habitants du village et leurs âges pour gérer les festivités annuelles.

- **Création d'une liste d'habitants :**

- Créez une liste habitants contenant les noms de 5 habitants du village.
- Ajoutez 2 nouveaux habitants à la liste.
- Supprimez l'un des habitants qui a quitté le village.
- Affichez la longueur de la liste.

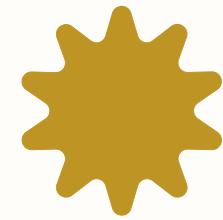
- **Ajout des âges des habitants :**

- Créez une liste ages contenant les âges des habitants (assurez-vous qu'elle correspond à la liste habitants).
- Le maire souhaite organiser une fête pour les habitants âgés de plus de 18 ans. Filtrez la liste des habitants et affichez uniquement ceux qui ont plus de 18 ans.

- **Modification de la liste :**

- Un nouveau-né vient de naître dans le village, ajoutez son prénom et son âge (0) à la liste.
- Affichez la liste mise à jour des habitants.





LES TUPLES

- **Séquence ordonnée et non modifiable d'éléments**

- Les éléments d'un tuple ne peuvent pas être modifiés
- Défini par ()
 - Parenthèse non obligatoire lorsqu'au moins un élément

```
#Tuple vide
```

```
a = ()
```

```
# Tuples contenant un seul élément
```

```
b = 1,
```

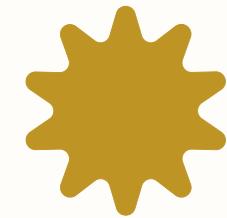
```
c = (1,)
```

```
# Tuples contenant trois éléments
```

```
d = 1, 2, 3
```

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





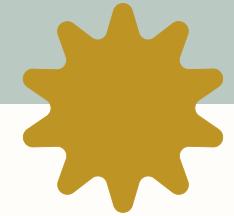
LES TUPLES

- Accès aux éléments d'un tuple avec les crochets
 - En lecture seulement, l'accès en écriture est interdit
- Taille d'un tuple obtenue avec la fonction `len()`
- Parcours avec `for` or `while`
- Définition avec parenthèses parfois **obligatoire**
 - dans des appels de fonctions
 - Crédit d'un tuple vide

```
def sum(values):  
    sum = 0  
    for v in values:  
        sum += v  
    return sum
```

```
sum(1, 2, 3)      # Erreur  
sum((1, 2, 3))  # Correct
```





Un peu de pratique

Chaque habitant du village possède une maison. Le maire souhaite garder une trace des maisons et de leurs localisations dans le village.

1. Création d'un tuple pour chaque maison :

- Créez un tuple pour chaque habitant, contenant son nom, l'adresse de sa maison (un numéro de rue), et la taille de la maison (en mètres carrés). Exemple : ("Alice", "Rue des Oliviers, 5", 100)

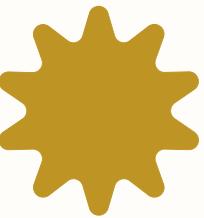
2. Affichage des informations des maisons :

- Affichez les informations de chaque maison sous forme de texte : "Alice vit à Rue des Oliviers, 5 dans une maison de 100m²."

3. Modification des tuples :

- L'un des habitants a agrandi sa maison. Étant donné que les tuples sont immuables, reconvertissez les informations en liste, modifiez la taille de la maison, puis recréez un tuple pour cet habitant.





LES ENSEMBLES

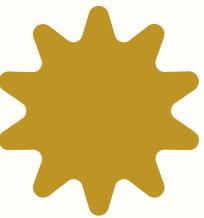
- **Collection non-ordonnée d'éléments distincts**
 - Pas de doublons et pas d'ordre entre les éléments
 - Défini par {}

```
ens = {54, 6, 8}
```

```
print(len(ens))  
print(6 in ens)
```

```
for el in ens:  
    print(el)
```





LES ENSEMBLES

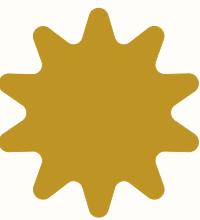
- **Ensemble vide avec set()**
- **Création à partir d'une séquence**
 - Suppression automatique des doublons
- **Modification d'un ensemble**
 - fonctions add et remove

Attention - Element unique = non modifiable

```
G = set('robotique')
F = set()
S = {n for n in range(100) if n%4 == 0}

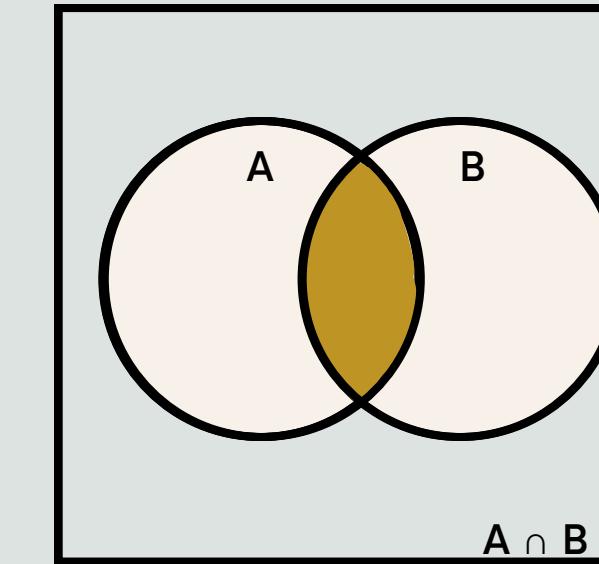
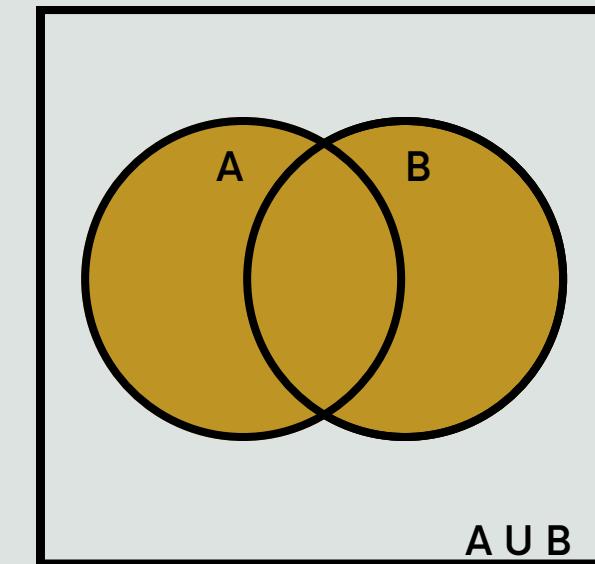
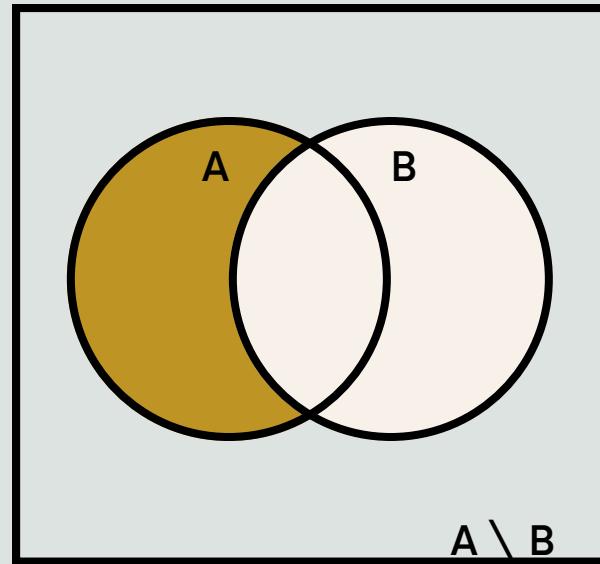
F.add(5)
F.add(67)
S.remove(4)
```





LES ENSEMBLES

- Opérations possibles sur les ensembles
 - Différence, Union et intersection



A = {0, 2, 4, 5}

B = {3, 4, 5}

```
print(A - B)      # {0, 2}
print(A & B)      # {4, 5}
print(A | B)      # {0, 2, 3, 4, 5}
```



Un peu de pratique

Le maire organise une grande fête au village. Il y a plusieurs événements, et il souhaite savoir qui y participe pour éviter les doublons dans les invitations.

1. Création d'ensembles pour les événements :

- Créez deux ensembles `event_musique` et `event_danse`, contenant les noms des habitants qui participent respectivement aux événements de musique et de danse.
- Exemple : `event_musique = {"Alice", "Bob", "Claire"}` et `event_danse = {"Claire", "David", "Eve"}`

2. Manipulation des ensembles :

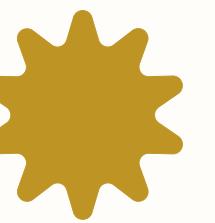
- Affichez les habitants qui participent aux deux événements. &
- Affichez les habitants qui participent à au moins un des événements. |
- Affichez les habitants qui participent uniquement à l'événement de musique.

3. Ajout et suppression d'éléments :

- Un habitant décide de rejoindre l'événement de danse. Ajoutez son nom à `event_danse`.
- Un autre habitant décide de ne plus participer à l'événement de musique. Retirez son nom de `event_musique`.

va σημειωθεί πως η εάν
έχουμε ένα ensemble και
αφαιρούμε πράγματα,
εαν δεν υπάρχει κάτι
στο ensemble 1 και
υπάρχει μόνο στο
ensemble 2, τότε δεν θα
εμαφνίσει κάτι με μείον
μπροστά !!! Βλέπε
παραδειγμα διόρθωσης
εάν δεν το
καταλαβαίνεις





LES DICTIONNAIRES

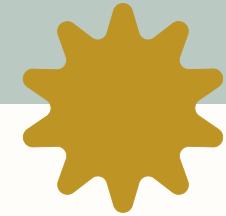
- Ensemble de **paires** clé-valeur
 - Les clés sont uniques et non-modifiables
- Définition avec **dict()** (vide) ou **{}**

```
dic = {'Luc' : 45, 'Lea' : 34, 'Henri': 12}

print(dic)
print(len(dic))
print('Luc' in dic)

map = {i : i for i in range(54, 70)}
```





Un peu de pratique

Le maire souhaite maintenant gérer les biens des habitants. Chaque habitant possède un certain nombre de vaches, de moutons, et de champs.

1. Crédation d'un dictionnaire des biens :

- Créez un dictionnaire `biens_habitants` où chaque clé est le nom d'un habitant et la valeur associée est un sous-dictionnaire contenant :
 - Le nombre de vaches.
 - Le nombre de moutons.
 - Le nombre de champs.

2. Accès et modification des données :

- Le maire souhaite savoir combien de vaches possède chaque habitant. Parcourez le dictionnaire et affichez cette information.
- Un habitant vient de vendre deux moutons. Mettez à jour le dictionnaire en conséquence.

3. Ajout de nouveaux habitants :

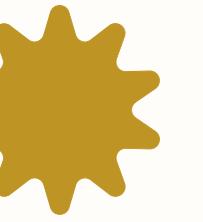
- Un nouvel habitant arrive au village avec des biens. Ajoutez-le au dictionnaire.

4. Somme totale des biens :

- Calculez le nombre total de vaches, de moutons, et de champs dans le village en parcourant le dictionnaire.

-

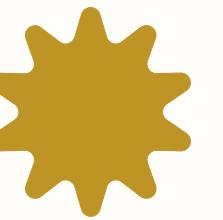




QUALITÉ DU CODE

- Différence entre code **fonctionnel** et code de **qualité**
- **Pourquoi vouloir un code de qualité ?**
 - Facilité de lecture
 - Mise à jour plus aisée
 - Moins de bug
- Ensemble de règles de bonne pratique





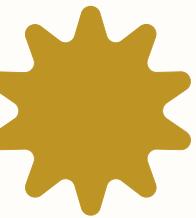
QUALITÉ DU CODE

- **Vérification automatisée de nombreuses règles**
 - Convention de codage
 - Détection d'erreurs (interface, import...)
 - Aide au refactoring (code dupliqué....)
- **Suggestion d'amélioration du code**
- **Interface graphique simple en Tkinter**



```
conda install conda-forge::pylint
```

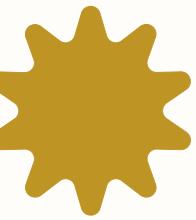




TEST UNITAIRES

- **Objectif**
 - Faire un programme qui marche !
- Cycle de développement habituel
 - Coder une fonction
 - Ecrire un main
 - Tester rapidement sans forcément voir les cas limites
 - ...

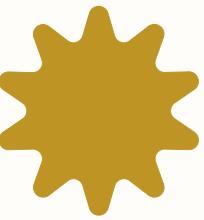




TEST UNITAIRES

- **Problèmes**
 - Très artisanal
 - Pas du tout exhaustif
 - Test mélangés avec le main
- Les tests sont fondamentaux dans un projet
 - Temps de débug > temps d'écriture
 - Isolement des bugs plus tôt
 - Des bons tests peuvent servir d'exemple





TEST UNITAIRES

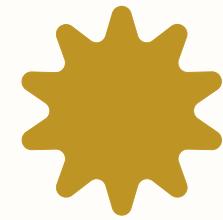
Objectifs - Déetecter les bugs le plus tôt possible dans le cycle de développement

- Tester une nouvelle méthode dès qu'elle est écrite
- Répéter l'ensemble des tests après chaque modification

Question à se poser

- Quel sont les cas limites ? Les cas usuels ?
- Après modification du code, a-t-on toujours le même résultat ?



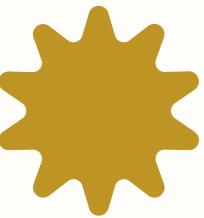


TEST UNITAIRES

Caractéristiques - Au niveau de l'unité logicielle (ici, une méthode ou une classe)

- Un jeu de test par classe
- Une ou plusieurs fonction de test par méthode
- **Automatique** - On doit pouvoir exécuter l'ensemble des tests automatiquement quand on veut
- **Disponible** - Les tests doivent être fourni avec le code source





TEST UNITAIRES

En Python - plusieurs paquets disponibles (pyunit, unittest...)

- Modules intégrés dans la bibliothèque standard de Python
- Facilite l'écriture et exécution de tests unitaires

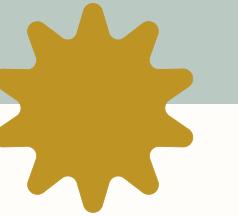
```
import unittest

def addition (a , b ) :
    return a + b

class TestAddition ( unittest . TestCase ) :
    def test_addition_entiers ( self ) :
        resultat = addition (2 , 3)
        self . assertEquals ( resultat , 5)

    def test_addition_floats ( self ) :
        resultat = addition (2.5 , 3.5)
        self . assertEquals ( resultat , 6.0)
```



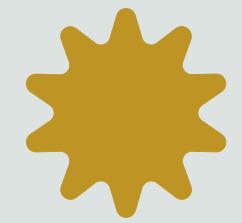


Un peu de pratique

Pour les fonctions précédentes, vous allez écrire des tests unitaires.

- **Listes**
 - Test unitaire pour la fonction de filtrage
- **Tuples**
 - Test unitaire pour la fonction modifier_maison
- **Ensemble**
 - Test unitaire pour toutes les fonctions
- **Dictionnaire**
 - Test unitaire pour compter les animaux





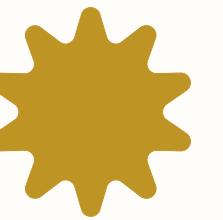
PROGRAMMATION ORIENTÉE OBJET



EXEMPLE

- Imaginons que l'on programme un logiciel de dessin
- **Plusieurs type de formes**
 - Rectangles
 - Cercles
 - Flèches
 - ...
- Chaque forme a ses caractéristiques
 - **Cercle** - Centre, rayon, couleur
 - **Rectangle** - Centre, longueur, largeur
 - **Flèche** - Départ, arrivée
 - ...





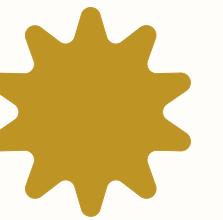
EXEMPLE

```
# Définition d'un point comme un dictionnaire
def create_point(x, y):
    return {'x': x, 'y': y}

# Définition d'un cercle comme un dictionnaire
def create_circle(center, radius):
    return {'center': center, 'radius': radius}

# Définition d'un rectangle comme un dictionnaire
def create_rect(center, width, height):
    return {'center': center, 'width': width, 'height': height}
```





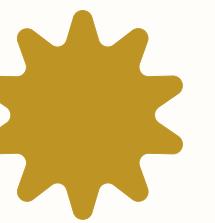
EXEMPLE

```
# Déplace un cercle en modifiant les coordonnées du centre
def move_circle(c, x, y):
    c['center']['x'] += x
    c['center']['y'] += y

# Déplace un rectangle en modifiant les coordonnées du centre
def move_rect(r, x, y):
    r['center']['x'] += x
    r['center']['y'] += y
```

Ne diffère que par le type de forme





EXEMPLE

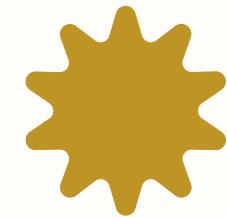
Code difficile à écrire et surtout à maintenir

- Programmation Objet - Modéliser des concepts abstraits sous formes de familles de classes d'objets

Exemple

- **Forme** - Classe "Abstraite"
- **Cercle** - Classe "Concrète", **dérivée** de Forme
- **Rectangle** - Classe "Concrète", **dérivée** de Forme

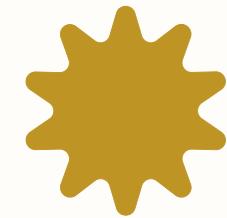




POURQUOI DES OBJETS ?

- Les objets permettent de représenter "naturellement" la réalité
 - Le monde est fait d'objets (ordinateurs, pommes...)
 - Plus facile de penser par objet
- Les classes d'objet permettent de "ranger" la réalité
 - Les "chaises" appartiennent à la classe "meuble"
 - Classification en biologie (animaux, mammifères, insectes...)
- Beaucoup de programmes modélisent la réalité
 - jeu de courses
 - banque
 - ...
- Plus la structure de notre programme ressemble à notre réalité, mieux on arrive à réfléchir



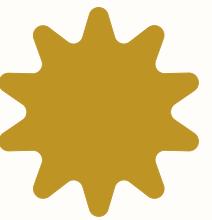


POURQUOI DES OBJETS ?

Les 20 dernières années ont montré que cette idée permettaient de créer des logiciels :

- complexes
- extensibles
- divisés de manière assez naturelle
- élégants



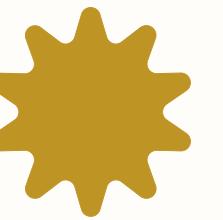


LIEN ENTRE OBJET ET PROGRAMMATION

Comment traduire ces objets dans les langages de programmation ?

- Les Classes - le "type" des objets
- **Exemples**
 - Tous les objets "Chaises" appartiennent à la même classe
 - Toutes les "voitures" appartiennent également à la même classe





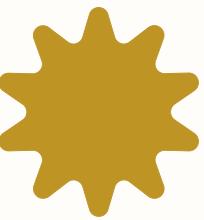
LIEN ENTRE OBJET ET PROGRAMMATION

Comment traduire ces objets dans les langages de programmation ?

- **Classe** - Plan de construction
- **Objet** - Une **instance** d'une classe
 - Créer un objet = **Instancier** une classe

Exemple - dans un programme, on aura une classe "**Cercle**" mais plusieurs objets instanciés avec cette classe.

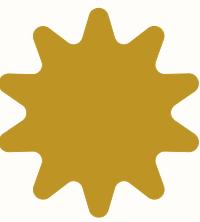




VOCABULAIRE

- Données de ma classe - **Attributs**
 - Exemple
 - La classe Point va avoir 2 attributs x et y
- Chaque classe peut également avoir des "fonctions" pour manipuler ses attributs - **Méthodes**
 - Exemple
 - La classe Point va avoir une méthode move(int x, int y)
- Méthode particulière et obligatoire - **Constructeur**
 - C'est cette méthode qui va initialiser les attributs de chaque instance de la classe avec les valeurs voulues.





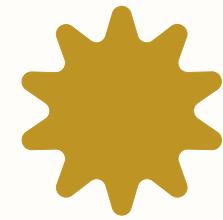
EXEMPLE EN PYTHON

```
# Exemple avec les points et les rectangles
class Point ( object ) :
    # Initialisation de la classe
    # Cette fonction est un constructeur
    def __init__ ( self , x , y ) :
        # On initialise les attributs de la classe
        self . x = x
        self . y = y

    # Methode de la classe Point
    def move ( self , x , y ) :
        self . x += x
        self . y += y

class Rectangle ( object ) :
    def __init__ ( self , point , h , w ) :
        self . center = point
        self . h = h
        self . w = w
```





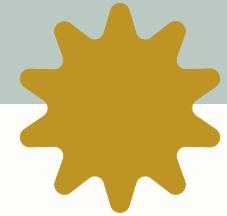
EXEMPLE EN PYTHON

```
def main () :  
    x = 1.2  
    y = 2.4  
    p = Point (x , y ) # Nouvelle instance de point  
    h = 3  
    w = 4  
    r = Rectangle (p , h , w ) # Nouvelle instance rectangle  
    p . move (2 , 3) # Appel de la fonction
```

Est-ce que le centre du rectangle est modifié par
l'appel à la fonction move ?

Normallement il n'est pas modifié





Un peu de pratique

Reprendons l'exemple du village.

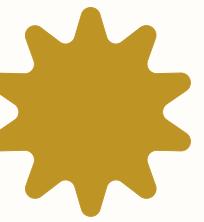
- Créer une classe habitant qui aura les attributs suivant : Nom, age, adresse, ainsi qu'un dictionnaire des animaux possédés par l'habitant.
 - Cette classe devra également avoir deux méthodes : **affichage_adresse()** et **compte_animal(animal: str)** (qui devra compter le nombre d'animaux du type animal que l'habitant possède)

```
class Habitant(object):
    def __init__(self, nom : str, age : int, adresse : str , dict_animaux : dict)
        self.age = age
        self.adresse = adresse
        self.nom = nom
        self.dict_animaux = dict_animaux

    def affichage_adresse(self)
        print(self.adresse)

    def compte_animal(self, animal:str)
        if animal in self.dict_animaux
            retrun self.dict_animaux[animal]
```





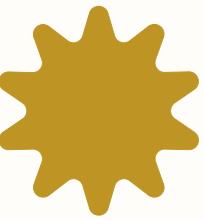
MÉMOIRE EN PYTHON

- En interne, les variables sont souvent considérées comme contenant des **références** vers les objets et non les objets eux-mêmes
 - **Exemple**

```
p1 = Point(2, 5)  
p2 = p1
```

Ici, p1 crée une référence vers un nouvel objet point. Et p2 = p1 **copie** la référence - p1 et p2 se réfèrent au même objet point.





AGRÉGATION ET COMPOSITION

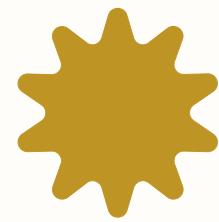
Relation de **Composition** - un objet est composé de plusieurs autres objets

Exemple

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
class Circle:  
    def __init__(self, centerX, centerY, radius):  
        self.center = Point(centerX, centerY)  
        self.radius = radius  
  
class Rectangle:  
    def __init__(self, centerX, centerY, width, height):  
        self.center = Point(centerX, centerY) # Composition : le centre n'existe pas en dehors du  
rectangle  
        self.width = width  
        self.height = height  
  
# Exemple d'utilisation  
# Le point est utilisé dans le cercle et le rectangle, mais ils n'en sont pas propriétaires  
circle = Circle(10, 20, 5.5)  
rectangle = Rectangle(10, 20, 10, 20)
```

Si le point est détruit, le cercle et le rectangle ne peuvent plus exister





AGRÉGATION ET COMPOSITION

Relation d'**Agrégation** - un objet est composé de plusieurs autres objets

Exemple

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

class Circle:
    def __init__(self, center, radius):
        self.center = center # Agrégation : le centre existe en dehors du cercle
        self.radius = radius

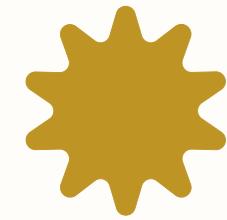
class Rectangle:
    def __init__(self, center, width, height):
        self.center = center # Agrégation : le centre existe en dehors du rectangle
        self.width = width
        self.height = height

# Exemple d'utilisation
center_point = Point(10, 20) # Création d'un point indépendant

# Le point est utilisé dans le cercle et le rectangle, mais ils n'en sont pas propriétaires
circle = Circle(center_point, 5.5)
rectangle = Rectangle(center_point, 10, 20)
```

Si le point est détruit, le cercle et le rectangle ne peuvent plus exister



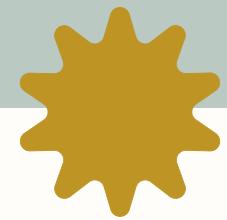


AGRÉGATION ET COMPOSITION

Différence clés entre Composition et Agrégation

- **Composition** - Une forte dépendance entre les objets. Si l'objet "tout" est détruit, l'objet "partie" l'est aussi.
- **Agrégation** - Une dépendance plus faible. L'objet "partie" peut exister indépendamment de l'objet "tout".





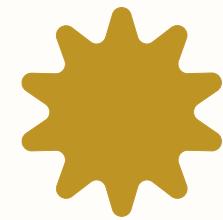
Un peu de pratique

Créer une classe village qui aura deux attributs : un nom et une liste des habitants.

- Ajouter deux fonctions à votre classe, qui permette d'ajouter des habitants à la liste: une par composition et une par agrégation.



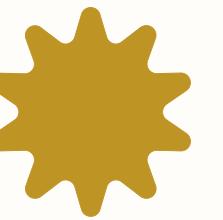
sos



Contrôle d'accès

- **Concept** - Cacher certains attributs à l'utilisateur de la classe
 - Maintenir l'intégrité du code
 - Cacher les détails d'implémentation
- C'est pourquoi les **attributs** et les **méthodes** peuvent être:
 - **Privés (private)** - accessible uniquement à l'intérieur de la classe
 - **Publics (public)** - accessible depuis l'extérieur
- **Interface d'une classe** - Ensemble des méthodes publiques
 - Elle n'intéragit avec les autres classes que par elle





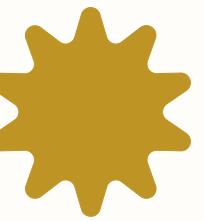
Contrôle d'accès

- **En Python** - Tous les attributs et méthodes sont publics par défaut.

○

```
class MaClasse :  
    def __init__ ( self , att ):  
        self . attribut_public = " Ceci est un attribut public "  
        obj = MaClasse ()  
  
    print ( obj . attribut_public ) # Accessible directement depuis l'exterieur
```



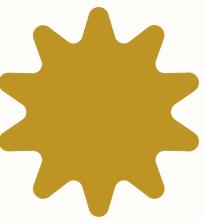


Contrôle d'accès

- **En Python** - Tous les attributs et méthodes sont publics par défaut.
 - Les attributs privés sont conventionnellement marqués en préfixant leur nom avec deux traits de soulignement (`__`).
 - Cela ne rend pas l'attribut réellement privé, mais cela le rend moins accessible depuis l'extérieur de la classe.

```
class MaClasse :  
    def __init__ ( self ) :  
        self . __attribut_prive = " Ceci est un attribut prive "  
  
    obj = MaClasse ()  
    # print ( obj . __attribut_prive ) # Cela provoquerait une AttributeError  
    # Acces avec " name mangling "  
    print ( obj . _MaClasse_attribut_prive )
```

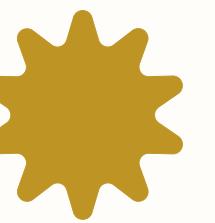




Contrôle d'accès

- **Règle générale** - Tous les attributs sont **privés**
 - Pour y accéder ou les modifier, on utilise des **accesseurs**
- **Intérêt** - Isoler l'accès aux attributs de leur représentation et avoir un meilleur contrôle des droits d'accès
- **Convention** - `get()` et `set()`
 - On ne définit pas forcément ces fonctions pour tous les attributs
 - Seulement ceux qui doivent être accessibles depuis l'**extérieur** de la classe

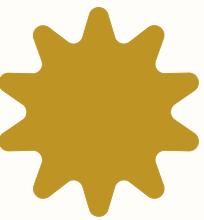




CONTRÔLE D'ACCÈS

```
class Circle :  
    # Attributs  
    def __init__ ( self ) :  
        self . radius = 0  
        self . area = 0.0  
  
    # Méthodes  
    def set_radius ( self , r ):  
        self . radius = r  
        self . compute_area ()  
  
    def get_radius ( self ) :  
        return self . radius  
  
    def compute_area ( self ) :  
        self . area = 3.14 * ( self . radius ** 2)
```



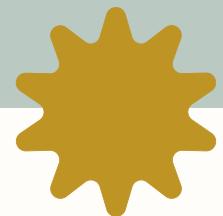


Contrôle d'accès

En Python, on peut simplifier les accesseurs:

- **@property** transforme une méthode en propriété (ie, en attribut accessible uniquement en lecture)
- **@attribut.setter** permet de mettre à jour la valeur de l'attribut.





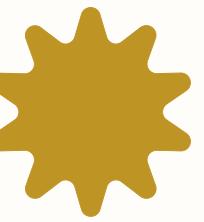
Un peu de pratique

Reprenez votre classe habitant et faites en sorte que ses attributs soient privés.

- Ajouter les accesseurs nécessaires pour pouvoir avoir accès aux attributs en dehors de la classe.

#Προσοχή, εδώ κάνουμε ίσον διότι η σύνταξη είναι τέτοια που δεν μας επιτρέπει να περάσουμε variables στις παρενθέσεις διότι έχουμε πάνω κάτω την ίδια function σε όνομα με διαφορετικές λειτουργίες η κάθε μία
h.name = "Jane Doe"
h.age = 25
h.adresse = "456 Elm St"
h.dict_animals = {"chien": 2, "tortue": 1}





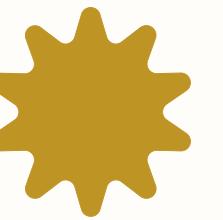
SURCHARGE

Définir plusieurs méthodes/fonctions avec le même nom mais avec des types d'arguments différents.

Attention

La surcharge est **de base** impossible en Python. Donner le même nom à 2 deux fonctions effacera la première. De même que donner le même nom à une variable et une fonction. Le plus ancien des 2 se fera écraser par l'autre.





SURCHARGE

En python pour faire de la surcharge,
il faut utiliser le paquet
multipledispatch.

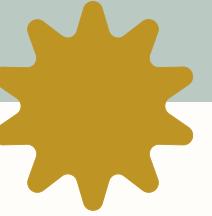
```
from multipledispatch import dispatch

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

    @dispatch(int, int)
    def move(self, x, y):
        ...

    @dispatch(Point)
    def move(self, p):
        ...
```

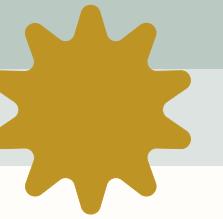




Un peu de pratique

Reprenez votre classe habitant surchargez le constructeur pour pouvoir faire une copie d'un habitant (**def __init__(self, other_habitant)**)

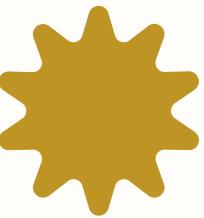




RÉSUMÉ

Une bonne classe

- Pas ou peu d'attributs publics
- Avoir un sens – représenter un concept
- Une interface simple
 - peu de méthodes publiques
- Et doit être documentée



HERITAGE

Il existe des classes plus ou moins abstraites :

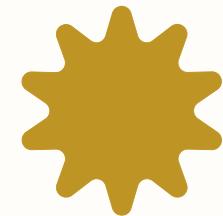
- Voiture
- Voiture de course
- Voiture de course rallye
- Voiture de course sur circuit
- ...

⇒ On peut les ordonner par ordre d'abstraction

- Toutes les voitures de course sont des voitures :
 - Mêmes fonctions (+ éventuellement d'autres)
 - Même structure (+ éventuellement des accessoires)

⇒ La classe "voiture de course" peut être **dérivée** de la classe "voiture". La classe "voiture de course" **hérite** de "voiture"



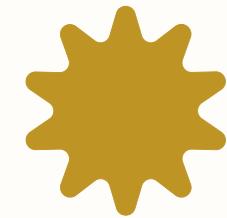


HERITAGE

L'héritage permet :

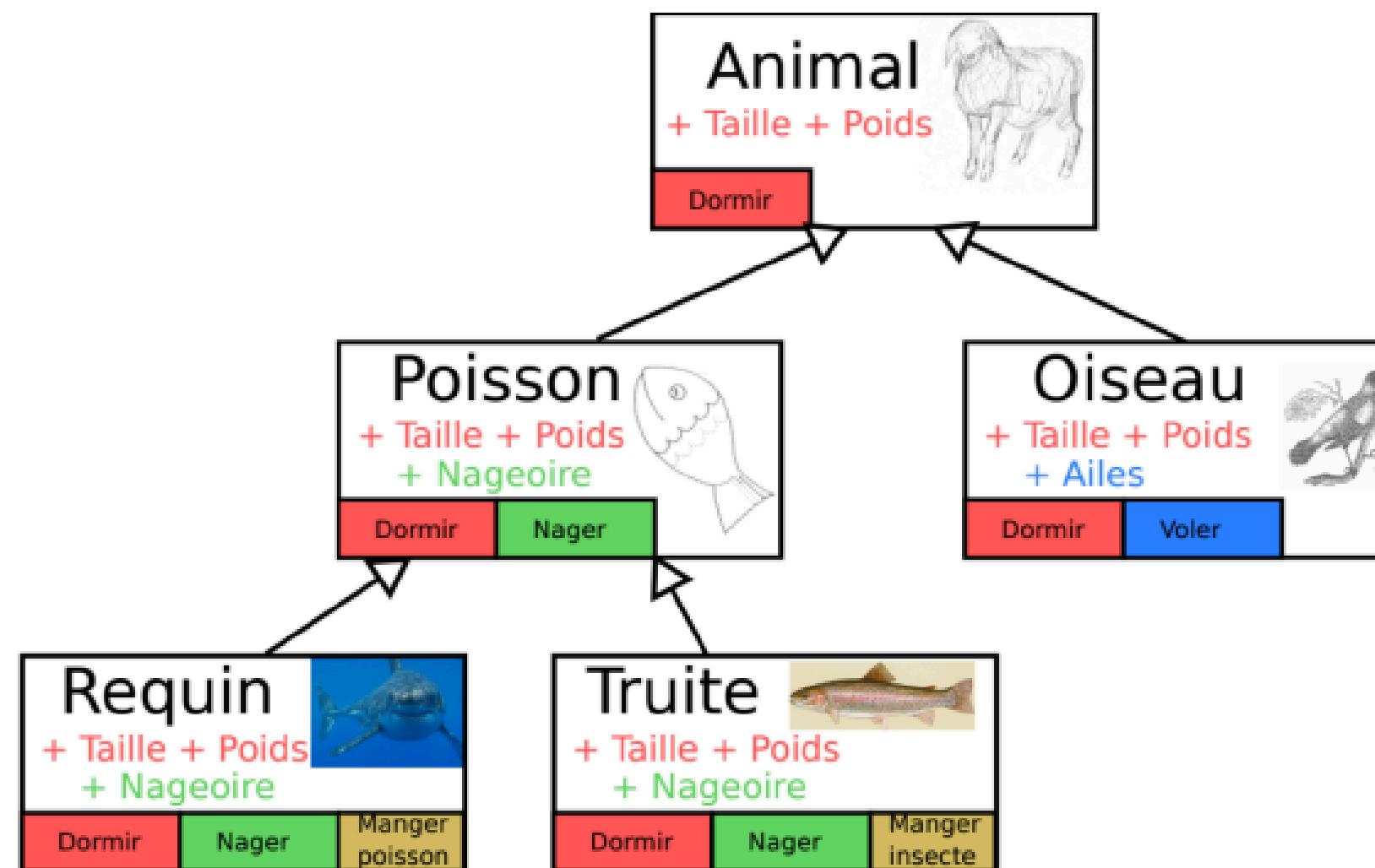
- Manipuler des concepts abstraits
 - On ne se préoccupe pas des détails quand ils ne sont pas nécessaires
 - Indispensable dans les logiciels complexes
 - Permet des traitements génériques
- Étendre les logiciels existants sans toucher au code
 - Indispensable pour la maintenance/amélioration du code

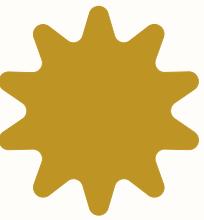




HERITAGE

Une classe dérivée **hérite** de la structure et du comportement de sa classe parent





HERITAGE

Une classe dérivée hérite de la structure et du comportement de sa classe parent

- "hérite" des attributs
- "hérite" des méthodes

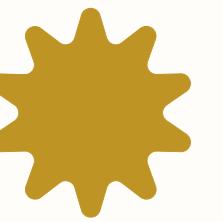
Une classe dérivée peut

- Redéfinir des méthodes
- Ajouter des méthodes et des attributs

Mais

- Elle ne peut rien enlever
- Ne peut dériver que d'une seule classe (en Python)





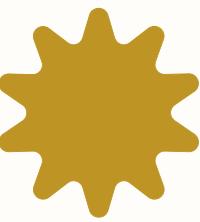
HERITAGE

Attention

Héritage != Composition

- A hérite de B → un A est un B
- A compose B → un A contient un B

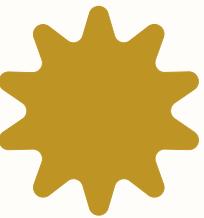




HERITAGE

```
class Voiture :  
    def __init__(self) :  
        self.roues = None  
        self.moteur = None  
  
    def avancer(self) :  
        pass # Placeholder for avancer method  
        implementation  
  
class VoitureDeCourse(Voiture) :  
    def __init__(self) :  
        super().__init__()  
        self.aileron = None
```





HERITAGE

Intérêts

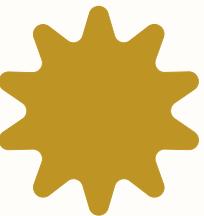
- **Organiser les classes**

→ Meilleure compréhension et modélisation de la réalité

- **Factoriser le code**

→ Le code commun à toutes les classes (filles et mère) n'est écrit qu'une seule fois





HERITAGE

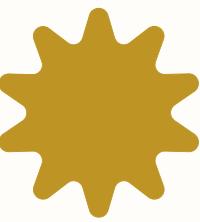
Redéfinition

On peut aussi redéfinir des méthodes. Dans ce cas, la méthode appelée est celle qui est le plus bas dans l'arbre de dérivation

- **Intérêt** - Modifier le comportement d'une classe sans tout réécrire
 - Redéfinition = **même signature**
→ Même nom et mêmes arguments

SOS



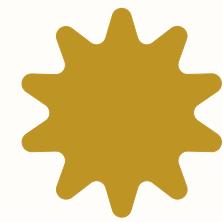


HERITAGE

Redéfinition

```
class A :  
    def test1 ( self ) :  
        print ( " test1 de A" )  
        self . test2 ()  
  
    def test2 ( self ) :  
        print ( " test2 de A" )  
  
class B ( A ) :  
    def test2 ( self ) :  
        print ( " test2 de B" )  
  
if __name__ == " __main__ " :  
    a = A ()  
    a . test1 () # Prints " test1 de A" and then " test2 de A"  
    b = B ()  
    b . test1 () # Calls the inherited method , prints "test1 de A" and then " test2 de B"
```





HERITAGE

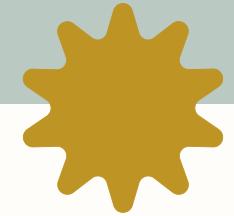
Classe Abstraite

- Certaines classes sont uniquement des concepts
- **Exemple** - On ne peut pas créer un objet Shape uniquement, il faut être plus précis (quel type de forme ?)
- On dit que ces classes sont **abstraites**
- Pour garantir que l'utilisateur ne crée pas d'objet de type Shape uniquement, on utilise le module abc

```
from abc import ABC

class Shape ( ABC ) :
    pass
```



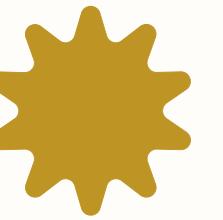


Un peu de pratique

Transformez votre classe habitant pour qu'elle devienne abstraite et créer ensuite deux classes dérivées : Adulte et Enfant. **Attention** - vérifiez bien que les attributs sont cohérents.

- Ajoutez une fonction `calcul_nombre_annee_avant_retraite()` à votre classe habitant. Cette fonction devra être abstraite dans la classe habitant (`from abc import abstractmethod`)
- Implémentez cette fonction dans vos classes dérivées
 - Pour la classe Adulte, elle devra marcher
 - Pour la classe enfant, elle renverra un message d'erreur





POLYMORPHISME

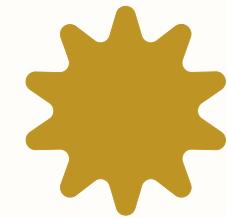
Pour l'instant, on a utilisé l'héritage pour

- Organiser nos classes
- Factoriser le code

Mais il est aussi utile pour

- Les traitements génériques
- L'extension d'un programme





POLYMORPHISME

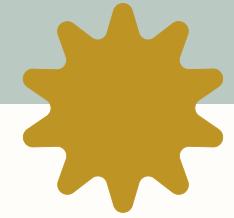
Si une classe B dérive d'une classe A

- le type B est un type A
- Toute méthode de A peut être invoquée sur une instance de la classe A

Subsomption - Dans toute expression "qui attend" un A, je peux "placer" un B

```
class Voiture:  
    ...  
  
class VoitureDeCourse:  
    ...  
  
def affiche(v : voiture):  
    print("Ceci est une voiture")  
  
if __name__ == "__main__":  
    Voiture v;  
    VoitureDeCourse vc;  
    affiche(v)  
    affiche(vc)
```





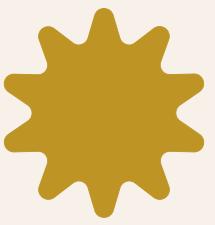
Un peu de pratique

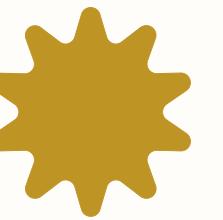
Modifiez la fonction `affichage_adresse(self)` en utilisant la fonction spécial `__str__(self)` qui renvoie une chaîne de caractère.

Ajoutez ensuite une fonction `affichage(h: Habitant)` qui prend en argument un habitant et qui l'affiche.



INTRODUCTION AU DESIGN PATTERN

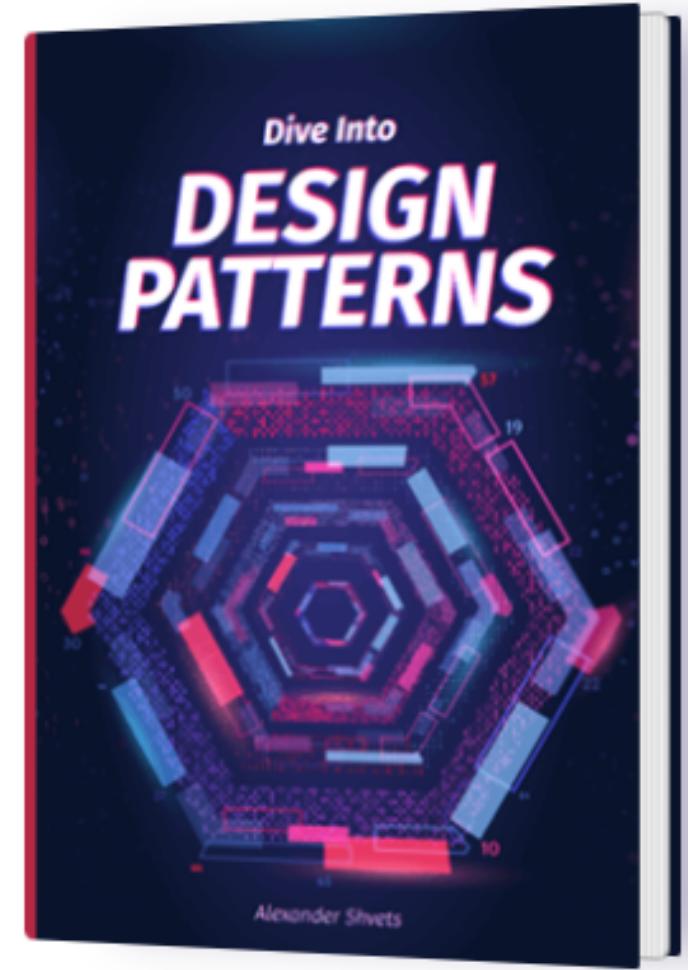




DESIGN PATTERNS

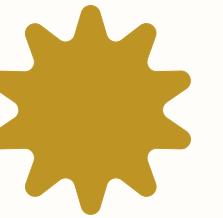
Modèles de Conception

- Solutions efficaces pour des problèmes récurrents
 - Utile à connaître
 - Permet d'avoir un vocabulaire commun



sourcemaking.com/design_patterns





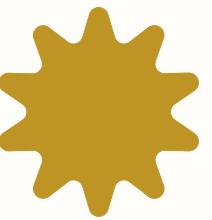
DESIGN PATTERNS

Type Créateur

- Permet d'instancier des classes de manière efficace

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton





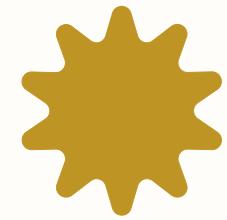
DESIGN PATTERNS

Type Structure

- Utiliser l'héritage pour composer des interfaces.
- Permet d'obtenir de nouvelles fonctionnalités.

- Adapter
- Bridge
- Composite
- **Decorator**
- Facade
- Flyweight
- Private Class Data
- Proxy





DECORATOR

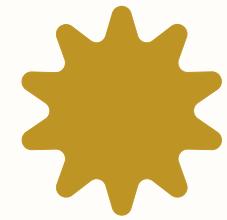
But

Ajouter dynamiquement des responsabilités supplémentaires à un objet.

Problème

Ajouter un comportement ou un état à des objets individuels lors de l'exécution. L'héritage n'est pas envisageable car il est statique et s'applique à toute une classe.





DECORATOR

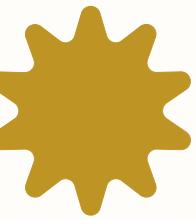
But

Ajouter dynamiquement des responsabilités supplémentaires à un objet.

Problème

Ajouter un comportement ou un état à des objets individuels lors de l'exécution. L'héritage n'est pas envisageable car il est statique et s'applique à toute une classe.





DECORATOR

```
# Classe de base du Robot
class Robot:
    def __init__(self, name):
        self.name = name

    def operation(self):
        return f"{self.name} est un robot de base."

# Décorateur pour ajouter la capacité de voler
def flying_robot_decorator(func):
    def wrapper(self):
        return f"{func()} Maintenant, il peut voler."
    return wrapper

# Décorateur pour ajouter des lasers
def laser_robot_decorator(func):
    def wrapper(self):
        return f"{func()} Maintenant, il peut tirer des lasers."
    return wrapper

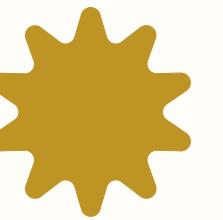
# Création de l'instance du robot
robot1 = Robot("R2-D2")
robot2 = Robot("C-3PO")

# Appliquer dynamiquement le décorateur "voler" uniquement sur robot1
robot1.operation = flying_robot_decorator(robot1.operation)

# Appliquer dynamiquement le décorateur "laser" uniquement sur robot2
robot2.operation = laser_robot_decorator(robot2.operation)

# Résultats
print(robot1.operation()) # Robot 1 peut voler
print(robot2.operation()) # Robot 2 peut tirer des lasers
```





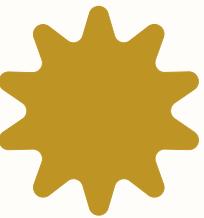
DESIGN PATTERNS

Type Comportement

- Communication entre les objets
- Définir comment les objets coopèrent pour accomplir des tâches et déléguer des responsabilités, facilitant ainsi des relations dynamiques entre eux.

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- **Visitor**





VISITOR

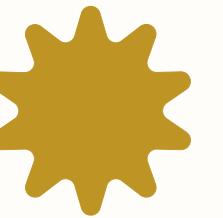
Intention

Le Visiteur vous permet de définir une nouvelle opération sans modifier les classes sur lesquels elle opère.

Problème

Vous souhaitez éviter de « polluer » les classes de nœuds avec ces opérations. De plus, vous ne voulez pas avoir à interroger le type de chaque nœud et à convertir au type correct avant d'exécuter l'opération souhaitée.





VISITOR

```
import abc

class Robot(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def accept(self, visitor):
        pass

class CleaningRobot(Robot):
    def accept(self, visitor):
        visitor.visit_cleaning_robot(self)

class DeliveryRobot(Robot):
    def accept(self, visitor):
        visitor.visit_delivery_robot(self)

class RobotVisitor(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def visit_cleaning_robot(self, cleaning_robot):
        pass

    @abc.abstractmethod
    def visit_delivery_robot(self, delivery_robot):
        pass

class MaintenanceVisitor(RobotVisitor):
    def visit_cleaning_robot(self, cleaning_robot):
        print("Performing maintenance on the Cleaning Robot.")

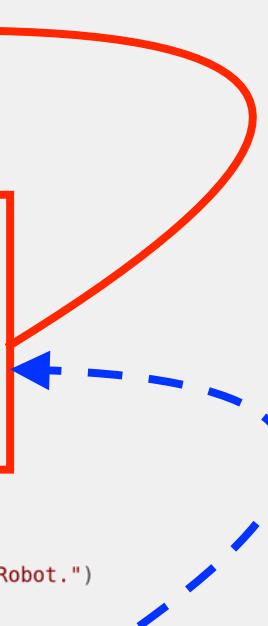
    def visit_delivery_robot(self, delivery_robot):
        print("Performing maintenance on the Delivery Robot.")

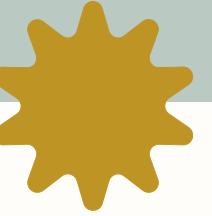
def main():
    maintenance_visitor = MaintenanceVisitor()

    cleaning_robot = CleaningRobot()
    delivery_robot = DeliveryRobot()

    cleaning_robot.accept(maintenance_visitor)
    delivery_robot.accept(maintenance_visitor)

if __name__ == "__main__":
    main()
```





Un peu de pratique

Ajoutez une classe VisiteurHabitant, qui aura deux fonctions visit_adulte() qui enlevera un chiffre entre 1 et 10 à l'âge et visit_enfant() qui ajoutera un chiffre entre un et 10 à l'âge.

Pensez également à ajouter les fonction nécessaires dans les classes Adulte et Enfant.

