introduction à la

POO

en

python™

Décembre 2018

OUTOFPLUTO

Programmation Orientée Objet en Python

Concepts et modélisation / UML

Atelier n°9

Syntaxe Python

Constructeur-Destructeur

Méthodes magiques

Héritage, surcharge et polymorphisme

Visibilité

Attributs et méthodes de classe

Atelier n°10

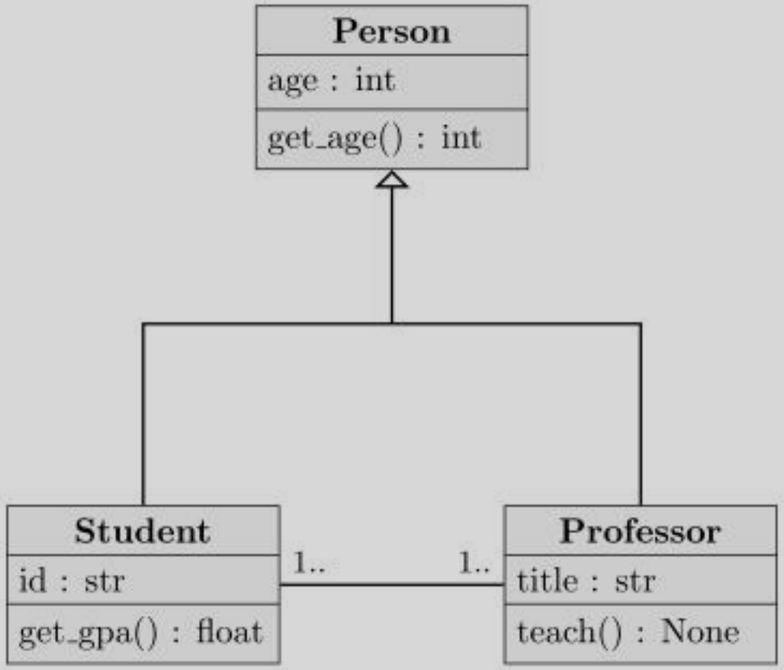
Abstraction et interface

MVC

Cadriciels OO

Concepts fondamentaux

La programmation orientée objet est indépendante du langage.



Définitions

Un **objet** est une "chose" pouvant être reconnue distinctement (une instance).
Une **classe** représente un ensemble d'objets similaires.

Exemple : Une Clio est un objet, une voiture est une classe.

Relations

Héritage (spécialisation / généralisation) : une classe hérite son interface et son implémentation d'une autre classe, la rédefinit partiellement et/ou l'augmente.
Agrégation ou composition : une classe agrège un nombre défini ou non d'instance d'une autre classe
Association : deux classes sont reliées en terme d'usage (via leurs interfaces).

Moyen mémo-technique

Héritage lorsque l'on peut dire un A **est** un B (un étudiant est une personne)
Agrégation ou composition lorsque l'on peut dire A **fait partie de** B (une roue fait partie d'une voiture) ou B **comprend** A (une voiture comprend 4 roues)
Association lorsqu'un autre verbe est nécessaire (un professeur **enseigne** à des étudiants)

Diagramme de classes

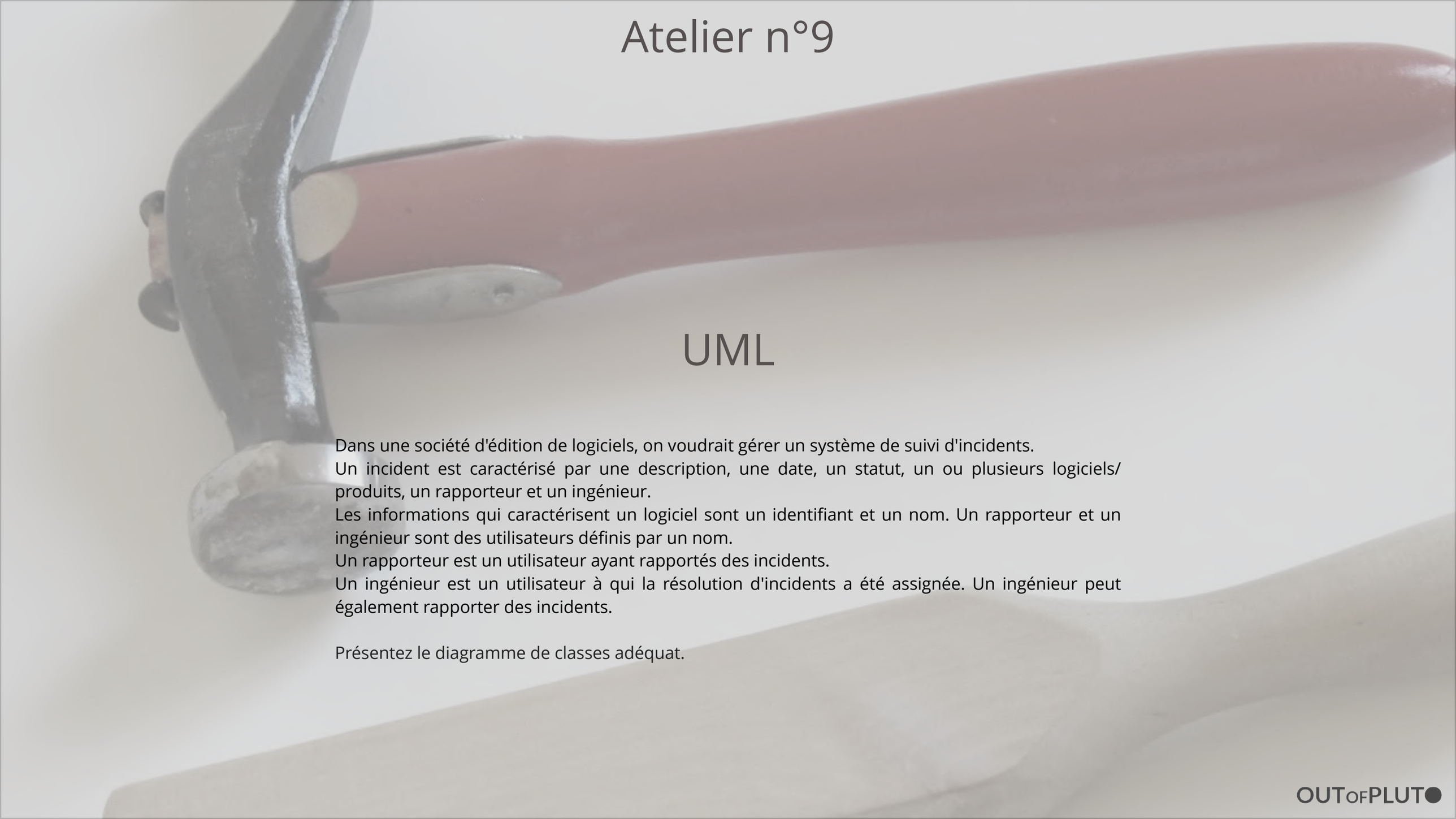
L'UML (Unified Modeling Language) met à disposition un formalisme pour plusieurs types de représentations.

Il existe aussi les diagrammes de classe permettant de représenter la structure et les relations d'une modélisation objet.

Les bases de la répresentation graphique :

https://en.wikipedia.org/wiki/Class_diagram

Atelier n°9



UML

Dans une société d'édition de logiciels, on voudrait gérer un système de suivi d'incidents.
Un incident est caractérisé par une description, une date, un statut, un ou plusieurs logiciels/ produits, un rapporteur et un ingénieur.
Les informations qui caractérisent un logiciel sont un identifiant et un nom. Un rapporteur et un ingénieur sont des utilisateurs définis par un nom.
Un rapporteur est un utilisateur ayant rapportés des incidents.
Un ingénieur est un utilisateur à qui la résolution d'incidents a été assignée. Un ingénieur peut également rapporter des incidents.

Présentez le diagramme de classes adéquat.

```python
class User:
    def get_name(self):
        return self.name

class Product:
    def __init__(self, name):
        self.name = name

class Engineer(User):
    def __init__(self, name, product):
        self.name = name
        self.product = Product(product)


ab = Engineer("Anthony Baillard", "argamato")
print(ab.get_name())
```

Syntaxe en Python

Concepts fondamentaux

Héritage et polymorphisme

Une méthode peut avoir plusieurs formes. Lors de l'héritage, elle peut être "surchargée" ou "redéfinie".
On distingue le polymorphisme statique (surcharge, signature différente) et dynamique (redéfinition, signature identique).
Certains langages permettent l'héritage multiple.

La programmation orientée objet permet l'abstraction de types de données et de traitements algorithmiques.

Une classe peut par exemple représenter une liste et contenir des algorithmes de recherche, de tri, d'insertion...
C'est une méthode efficace de factorisation du code.

Constructeurs, destructeurs

Il existe des fonctions spécifiques aux objets, le constructeur et le destructeur.

Constructeurs et destructeurs

En Python, les objets possèdent deux fonctions spécifiques appelées automatiquement : le constructeur et le destructeur.

Comme leurs noms l'indiquent :

- Le constructeur est appelé lors de l'instanciation d'un objet.
- Le destructeur est appelé lors de la libération d'un objet.

Les deux méthodes sont optionnelles, elles permettent d'ajouter du code spécifique à ces deux événements, comme l'ouverture et la fermeture d'une connexion à une base de données par exemple.

```
def __init__(self, param1, param2, ...):
    ...


def __del__(self):
    ...
```

A noter que le destructeur n'accepte aucun paramètre.

Les méthodes magiques/spéciales

Les méthodes magiques ou méthodes spéciales sont des méthodes inhérentes au Python et qui, si elles sont déclarées, sont automatiquement appelées dans un certain contexte d'exécution.

Toutes les méthodes spéciales ont un nommage identique `__<methode>__`

Par exemple, les constructeurs et les destructeurs.

Autres exemples :

`__iter__`	appelée lors de l'utilisation d'un itérable dans une boucle
`__next__`	appelée lors de l'utilisation de `next()` sur un itérateur
`__contains__`	appelée lors d'une condition `in`
`__str__`	appelée lors de l'appel de `print()` sur un objet

⚠ Il existe également des symboles magiques comme `__file__`

Héritage

Le principe de l'héritage est de **spécialiser** une classe en sous-classe(s).
Les sous-classes (classes filles / classes enfants) ont les attributs et les méthodes de la classe parent.

En Python :

```
class Vehicle:
    def __init__(self, speed):
        self.speed = speed

    def get_speed(self):
        return "{}".format(self.speed)

class Car(Vehicle):
    pass

clio = Car(50)
print(clio.get_speed())
```

Appels parents et autres fonctions

Appel de fonctions du parent grâce à `super` :

```
class Car(Vehicle):
    def get_speed(self):
        return "{} km/h".format(super(Car, self).get_speed())

clio = Car(50)
print(clio.get_speed())
```

Quelques fonctions utiles :

`isinstance()`	https://docs.python.org/3/library/functions.html#isinstance
`issubclass()`	https://docs.python.org/3/library/functions.html#issubclass

Héritage multiple

L'héritage multiple est autorisé en Python.

```
class Wheels:
    def get_number_of_wheels(self):
        return 4

class Transporter:
    def get_number_of_passengers(self):
        return 5


class Car(Vehicle, Wheels, Transporter):
    pass
```

⚠ Il faut être prudent dans l'usage de cette possibilité.

La recherche de méthode ou d'attribut s'effectue en profondeur d'abord et dans l'ordre des déclarations.

Surcharge

La surcharge de méthode est de deux types :
Overriding ou surcharge dynamique : Surcharge d'une méthode de la classe parent (Même signature ou non).
Overloading ou surcharge statique : Surcharge d'une méthode avec plusieurs signatures dans une **même** classe.

Exemple d'*Overriding* du constructeur :

```
class Vehicle:
    def __init__(self, speed):
        self.speed = speed

class Engine:
    pass

class Car(Vehicle):
    def __init__(self, speed):
        self.speed = speed
        self.engine = Engine()
```

Surcharge

L'*Overloading* n'est pas autorisé en tant que tel en Python.

```
class Car(Vehicle):
    def __init__(self, speed):
        self.speed = speed
        self.engine = Engine()

    def __init__(self, speed, engine):
        self.speed = speed
        self.engine = engine
```

Mais il y a une alternative : les paramètres optionnels

Surcharge

Overloading avec paramètre(s) optionnel(s).
Les paramètres étant optionnels, on peut considérer que la méthode a plusieurs signatures.
Ce qui n'est pas exact, car il s'agit toujours de la **même** méthode.
Dans le corps de la fonction, les paramètres optionnels doivent être gérés dans tous les cas.

```
class Car(Vehicle):
    def __init__(self, speed, engine=None):
        self.speed = speed
        self.engine = engine if engine is not None else Engine()


clio = Car(50)
twingo = Car(50, Engine())
```

Surcharge et héritage multiple

```
class Wheels:
    def __init__(self, nb):
        self.nb_wheels = nb

class Transporter:
    def __init__(self, nb):
        self.nb_passengers = nb


class Car(Vehicle, Wheels, Transporter):
    pass

clio = Car(50)
print(clio.get_speed(), clio.get_number_of_wheels(), clio.get_number_of_passengers())
```

⚠ Que va-t-il se passer ? Corriger si nécessaire.

Visibilité

Par défaut, les attributs et les méthodes sont considérés `public`. Ils sont visibles dans trois contextes :

- Depuis la classe de leur déclaration
- Depuis les classes qui implémentent la classe de leur déclaration (classes enfants)
- Depuis l'extérieur de la classe de leur déclaration (i.e, de n'importe où).

Dans le cas de `protected`, ils sont visibles dans deux contextes :

- Depuis la classe de leur déclaration
- Depuis les classes qui implémentent la classe de leur déclaration

Dans le cas de `private`, ils sont visibles dans un seul contexte :

- Depuis la classe de sa déclaration

En Python 3, il n'y a pas de concept de visibilité `protected`.

Tous les attributs et méthodes sont publiques par défault.
Tous les attributs ou méthodes commençant par `__` sont privés.

Variables et méthodes de classe

Les attributs et les méthodes de classe (ou statiques) sont définis pour la classe et non par instance.
Les attributs et les méthodes statiques sont accessibles même si la classe n'est pas instanciée.

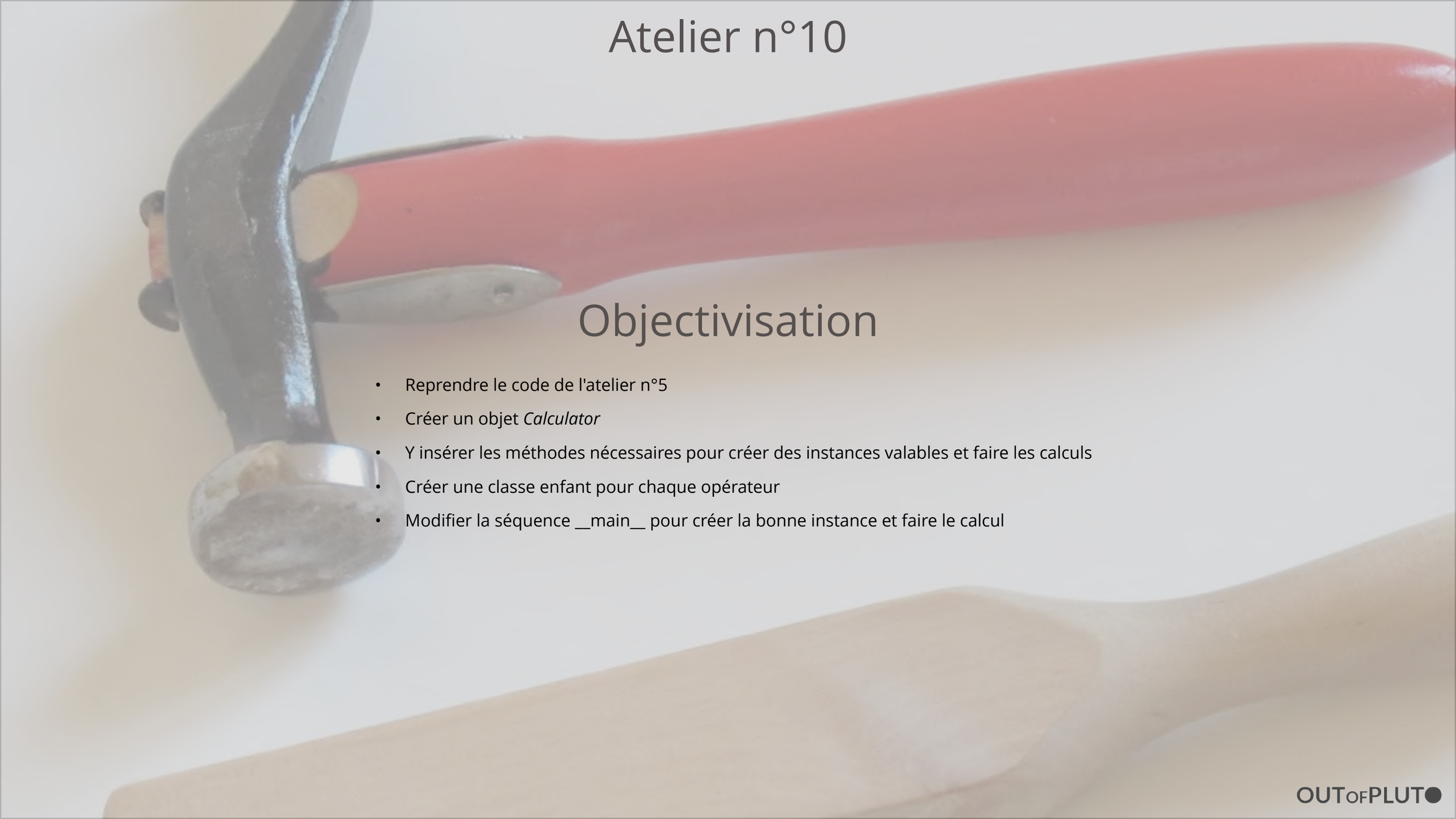
Cela permet de "partager" des informations entre toutes les instances d'une classe.

```
class Vehicle:
    class_name = "Véhicule indéterminé"

    @staticmethod
    def get_class_name():
        return Vehicle.class_name . " 0.1\n"

print(Vehicle.get_class_name())
clio = Vehicle()
clio.class_name = "Véhicule";
twingo = Vehicle()
print(clio.get_class_name())
print(twingo.get_class_name())
```

Atelier n°10



Objectivisation

- Reprendre le code de l'atelier n°5
- Créer un objet *Calculator*
- Y insérer les méthodes nécessaires pour créer des instances valables et faire les calculs
- Créer une classe enfant pour chaque opérateur
- Modifier la séquence __main__ pour créer la bonne instance et faire le calcul

Abstraction

Une classe abstraite est une classe qui peut être héritée mais **jamais instanciée**.

Cela permet de définir des méthodes qui devront être implémentées dans les classes spécialisant la classe abstraite *("voilà ce que je suis"*).

Seule une classe abstraite peut contenir des méthodes abstraites.
Mais une classe abstraite peut également contenir des méthodes non abstraites dont hériteront les classes spécialisées.

Python ne propose pas de classe abstraite. Il existe deux méthodes "équivalentes".

Abstraction et exceptions

La première grâce aux exceptions.

```
class Vehicle:
    def get_power_source(self):
        raise Exception("Not implemented")

class Bike(Vehicle):
    def get_power_source(self):
        return self.biker

class Car(Vehicle):
    def get_power_source(self):
        return self.engine
```

Abstraction et abc

La deuxième grâce au module abc.
https://docs.python.org/3/library/abc.html

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def get_power_source(self):
        pass

class Bike(Vehicle):
    def get_power_source(self):
        return self.biker

class Car(Vehicle):
    def get_power_source(self):
        return self.engine
```

Interfaces

Les interfaces permettent de déclarer une liste de méthodes pour toute classe qui souhaite l'implémenter
L'intérêt d'une interface est d'harmoniser des implémentateurs.
Les interfaces sont souvent nommé en `*able` pour indiquer que la classe qui implémente l'interface en possède les capacités (*"voilà ce que je peux faire"*).

```
class Drivable:
    def speed_up(self, accel):
        raise Exception("Not implemented")

    def slow_down(self, accel):
        raise Exception("Not implemented")

}

class Vehicle(Drivable):
    def speed_up(self, accel) {
            self.speed += accel
    }
    ... // all other methods
}
```

Abstraction ou interface

La classe qui implémente l'interface en possède les capacités (*"voilà ce que je peux faire"*).
La classe qui spécialise une classe abstraite en possède la forme *("voilà ce que je suis"*).

Une interface ne contient que des signatures de fonctions.
Une classe abstraite peut contenir des méthodes implémentées et des attributs, éventuellement de classe.

En Python, la différence n'existe pas formellement.

Modèle de conception MVC 1/2

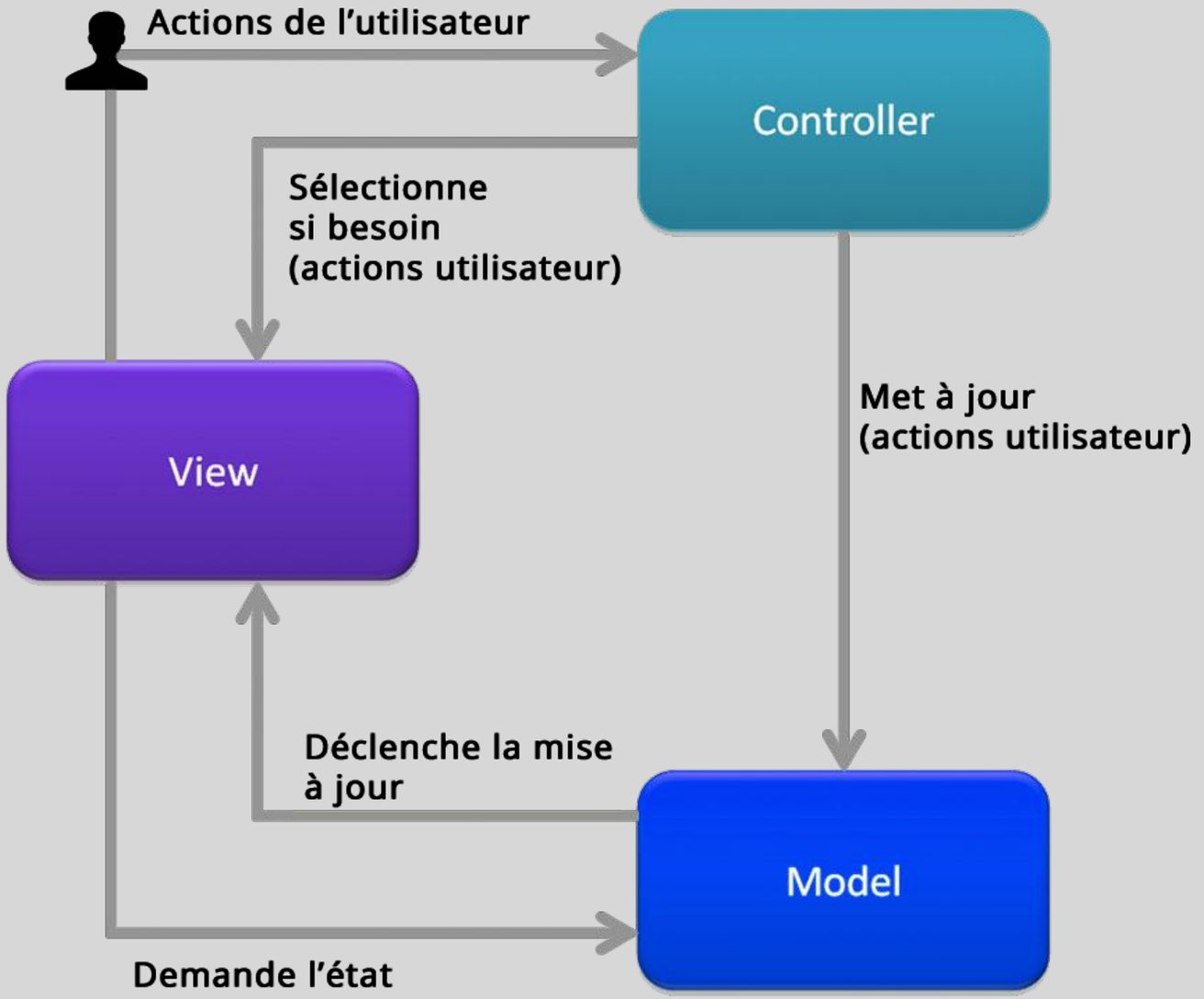
Model-View-Controller ou MVC.

Système d'architecture de programmation basé sur 3 classes principales.

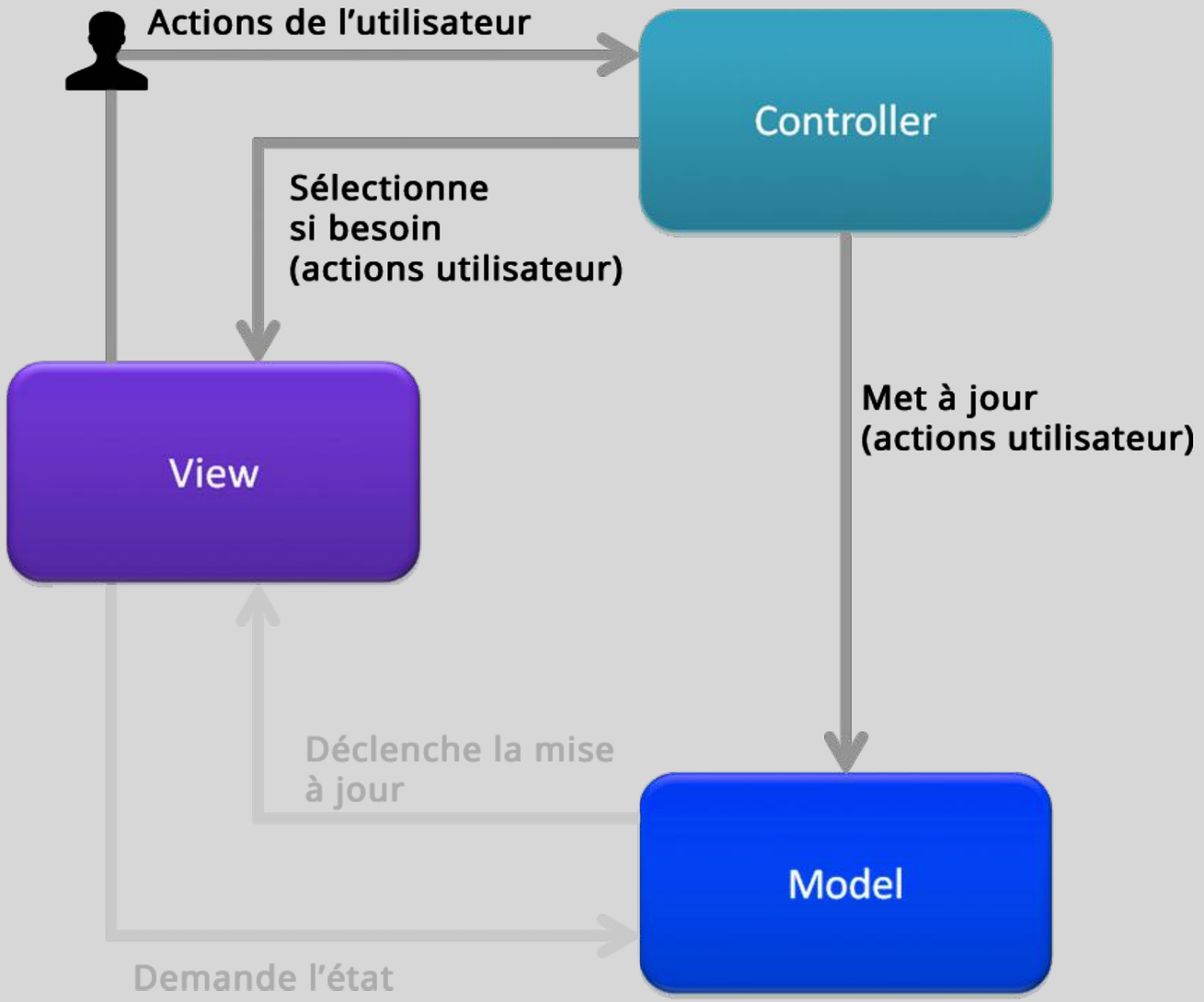
Le but est de séparer les composantes :

- Model : la gestion des données
- View : l'interface et la présentation
- Controller : la logique applicative

L'utilisateur se "place" entre *View* et *Controller.*



Modèle de conception MVC 2/2



Dans un modèle permettant une réutilisation générique de *View,* les liens entre *View* et *Model* ne sont pas directs et passent par le *Controller*.

Dans ce cas, seul *Controller* demande à être modifié d'un projet sur l'autre.

Model

 ***Model* ne dépend ni de *Controller* ni de *View.***

Le seul rôle de *Model* est de représenter les données.
Il peut être interfacé à une base de données mais cela n'est pas obligatoire.

En Python :

```
class PageModel:
    def __init__(self, data):
        self.verbose = "Page de blog"
        self.__data = data

    def get_data(self):
        return self.__data
```

View (cas 1)

View affiche les données de *Model* et envoie les actions de l'utilisateur à *Controller*.
Une *View* peut être dépendante d'un *Model* et/ou d'un *Controller* (cas 1).

En Python (cas 1) :

```
class PageView:
    def __init__(self):
        self.__model = PageModel()
        self.__controller = PageController()

    def output():
        return "<article>" + self.__model.get_data() + "</article>"
```

View (cas 2)

Une *View* peut également être indépendante de *Model* et de *Controller* (cas 2).

En Python (cas 2) :

```
class PageView:
    def render(data):
        return "<article>" + data + "</article>"
```

Dans ce cas, c'est *Controller* qui enverra les `data` à *View* dans le format attendu par `output`.

Controller

Controller interprète les actions et les répercute vers *View* et/ou *Model*.
Le Controller dépend de *View* et *Model*.

En Python :

```
class PageController:
    def __init__(self, model, view):
        self.__model = model
        self.__view = view
```

MVC en Python

Les trois classes sont en relation via le *Controller*.

```
model = PageModel()
view = PageView()
controller = PageController(model, view)

controller.load_page("home")
```

C'est dans le *Controller* que se trouve le code appelant *Model* et *View*.

```
class PageController:
    def load_page(name):
        template = self.__model.get_page(name)
        self.__view.render_page(template)
```

Cadriciels Orientés Objet

Un cadriciel orienté objet fourni des classes qui vont pouvoir être (ou non) spécialisées.

De cette façon, il est possible d'utiliser tout la mécanique du cadriciel.

Les classes spécialisées ont par défaut le comportement des classes parents.

On peut alors surcharger certaines méthodes soit :

- parce qu'on souhaite modifier leur comportement
- parce qu'elles doivent être implémentées (abstraction et interface)

Exemples

ORM (Object-relational Mapper), accès à une base de données

https://ponyorm.com/
https://www.sqlalchemy.org/

Web et API

https://docs.apistar.com/
https://www.djangoproject.com/ et https://www.django-rest-framework.org/

Tableurs (excel jusqu'à 2010)

https://openpyxl.readthedocs.io/en/stable/