Laboratoire 2 : Programmation orientée objet

Adrien Voisin, Didier Valentin, Bastien Bodart
Programmation 3

Henallux 2025-2026

1 Objets et classes

1.1 Classe

Un objet possède toujours un type, défini par une classe.

```
>>> type("hello")
<class 'str'>
```

Ceci nous montre que la variable *s* contient un objet dont le type est `str`.

Exercice :

— Créez différentes variables avec des valeurs numériques et des structures de données et affichez leurs classes avec print() et type().

Pour définir de nouveaux objets, on peut déclarer ses propres classes, en fonction des besoins. Voici une déclaration de classe, qui définit un objet `MyObject`, sans aucune caractéristique particulière :

```
>>> class MyObject:
...     pass
```

Pour créer un nouvel objet, on utilise le nom de la classe suivi de parenthèses qui contiennent les paramètres nécessaires à cet objet. Le notre n'en a aucun.

```
>>> m = MyObject()
>>> print(type(m))
<class '__main__.MyObject'>
```

Dans ce cas, notre objet `m` est de type `MyObject`, classe définie cette fois dans l'environnement `__main__`, l'environnement principal d'exécution de notre script et non dans un module.

Exercices :

— Trouvez la différence entre les instructions `m = MyObject()` et `m = MyObject`. Que contient `m` dans ces deux cas ? Laquelle crée une nouvelle instance ?

— Créez deux instances de `MyObject` et comparez les avec l'opérateur ==. Que pouvez-vous conclure ? Et avec l'opérateur *is* ?

1.2 Méthodes

Les méthodes sont des fonctionnalités des objets, qui sont définies comme des fonctions dans la déclaration des classes. On y fait appel avec l'opérateur . (point). Le paramètre `self`, une référence à l'objet lui-même est inclus par défaut dans la définition de la méthode.

```
class MyObject:

    def say_hello(self):
        print("Hello!")

m = MyObject()
m.say_hello() # Affiche Hello!
```

Exercices :

— Ajoutez une méthode `add` à la classe `MyObject` qui affiche le résultat de l'addition de deux nombres passés en paramètres.

— Essayez de faire un appel à la méthode `say_hello` dans elle-même au moyen du paramètre `self`. Quelle erreur obtenez-vous ? Que pouvez vous en conclure sur la récursivité terminale dans Python ?

1.3 Propriétés et méthode `__init__`

Les propriétés (ou variables d'instance) sont une caractéristiques des objets, des variables qui leur sont associées et qui varient pour chaque instance. Celles-ci sont définies dans la méthode réservée `__init__` dont les paramètres sont fournis au moment de la création de l'instance. Cette méthode est exécutée au moment de la création de l'objet.

```
class MyObject:

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

    def say_your_message(self):
        print(self.message)

m = MyObject("Hello!")
m.say_your_message() # Affiche Hello!
```

Exercices :
— Créez une classe `Animal` avec deux propriétés, `weight` et `species`.
— Ajoutez une méthode `print_info` qui affiche les propriétés de l'objet.
— Créez deux instances de `Animal` avec des valeurs au choix et testez la méthode `print_info`
— Vérifiez ce que contient la propriété réservée `__class__` d'une instance

1.4 Variables de classe

Une variable de classe est une variable interne à une classe mais dont la valeur est commune à toutes les instances de cette classe au moment de leur création. Elle se définit comme une variable dans la déclaration de la classe. On peut également accéder à cette variable via la classe et pas seulement une instance.

```
class MyObject:

    class_var = 1

print(MyObject.class_var) # Affiche 1
```

Exercices :
— Créez deux instances de `MyObject` et essayez de modifier la variable `class_var` d'une d'entre elles. Comparez ensuite sa valeur avec celle de la classe et de l'autre instance. Que remarquez-vous ?
— Même chose mais cette fois redéfinissez la valeur initiale de `class_var` en une liste vide et ajoutez un élément via une des instances. Que remarquez-vous ? Quel type d'erreur ce comportement peut-il provoquer ?

1.5 Propriétés protégées et privées (encapsulation)

Certaines conventions d'écriture permettent de restreindre l'accès aux propriétés d'une classe. Une propriété que l'on veut protéger aura un nom commençant par _ (underscore) tandis qu'une que l'on veut garder privée commencera par __ (double underscore).

```
class MyObject:

    def __init__(self):
        self._protected_var = 1
        self.__private_var = 2
```

Néanmoins, en Python, ces "restrictions" sont plus des conventions que de réelles restrictions.
Exercices :
— Essayez d'accéder aux propriétés de la classe `MyObject`. Que constatez-vous ?
— Trouvez un/des moyen(s) permettant de quand même accéder à une propriété privée.

2 Héritage

L'héritage permet de définir une classe en partant d'une définition de classe existante. Toutes les propriétés et méthodes de la classe parent seront alors disponibles pour la classe enfant. On dit alors que la classe enfant étend la classe parent. Pour cela on va renseigner le parent dans l'en-tête de déclaration de classe :

```
class A:

    def say_hello(self):
        print("Hello!")

class B(A):
        pass

b = B()
b.say_hello() # Affiche Hello!
```

Exercices :

- Reprenez votre classe `Animal` et créez deux autres classes, `Cat` et `Pigeon` qui étendent cette classe
- Pour chacune, définissez la méthode `__init__` avec une propriété `fur_color` pour `Cat` et `wingspan` pour `Pigeon`. Utilisez la fonction `super()` pour faire appel à `__init__` du parent
- Créez deux instances de `Cat` et `Pigeon` avec des valeurs au choix et testez la méthode `print_info`

2.1 Classes et sous-classes

Cet héritage forme ainsi une hiérarchie de classes et sous-classes, dont le parent commun à toutes les classes est la classe `object`. Il est possible d'obtenir des informations sur un objet ou une classe à l'aide des fonctions `issubclass` et `isinstance`.

```
class MyObject:
    pass

m = MyObject()
print(issubclass(MyObject, object))      # True
print(isinstance(m, object))             # True
```

Exercices :

- Utilisez les fonctions `issubclass` et `isinstance` sur vos instances de `Pigeon` et `Cat`. Qu'observez-vous ?
- Créez une classe `Bird` étendant `Animal` et modifiez votre classe `Pigeon` pour qu'elle étende `Bird`
- Revérifiez à l'aide des fonctions `issubclass` et `isinstance`. Qu'est-ce qui a changé ?

2.2 Overriding

L'overriding, ou redéfinition, permet de redéfinir des méthodes de la classe parent dans la classe enfant pour leur donner un comportement différent.

```
class A:

    def say_hello(self):
        print("Hello!")

class B(A):

    def say_hello(self):
        print("Hello World!")

a = A()
a.say_hello() # Affiche Hello!
b = B()
b.say_hello() # Affiche Hello World!
```

Exercices :

- Testez la méthode `print_info` sur une instance de `Pigeon`.

— Redéfinissez la méthode `print_info` dans la classe `Cat` afin d'afficher plus d'informations et testez la.
— Dans la classe `Cat`, essayez d'appeler la méthode `print_info` de `Animal` en vous servant de `super`

2.3 Polymorphisme

Le polymorphisme est un principe permettant d'écrire des méthodes ou fonctions qui pourront s'appliquer sur plusieurs types d'objets. En Python, ceci est accompli grâce à ce qu'on appelle le "duck-typing" [1] : un objet est avant tout défini par ses propriétés et ses méthodes et non par son type. Tout appel à une méthode ou propriété existante est donc valide.

```
class Car:

    def start(self):
        print("Car engine has started!")

class Race:

    def start(self):
        print("Race has begun!")

def start_something(startable):
    startable.start()

start_something(Car())          # "Car engine has started!"
start_something(Race())         # "Race has begun!"
```

Exercices :
— Créez une classe `Plane` avec une méthode `fly`, ajoutez aussi cette méthode à `Bird`
— Créez une fonction permettant d'appeler la méthode `fly` sur un paramètre et testez la avec des instances de `Plane` et `Bird`

2.4 Héritage multiple

Il est parfois utile de pouvoir étendre plusieurs classes pour bénéficier des caractéristiques de plusieurs types.

```
class A:

    def say_hello(self):
        print("Hello!")

class B:

    def say_thanks(self):
        print("Thanks!")

class C(A, B):
    pass

c = C()
c.say_hello() # Affiche Hello!
c.say_thanks() # Affiche Thanks!
```

Exercices :
— Créez une classe `Pet` avec une propriété `name`
— Modifiez la classe `Cat` pour qu'elle étende `Pet`

Toutefois cela peut poser problème lorsque les classes parent possèdent une méthode identique. [2] Python apporte une solution grâce à l'ordre dans lequel les parents sont déclarés.

```
class A:
```

1. https://en.wikipedia.org/wiki/Duck_typing
2. https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

```
    def who_am_i(self):
        print("A")

class B:

    def who_am_i(self):
        print("B")

class C(B, A):
    pass

c = C()
c.who_am_i() # Affiche B
```

Exercices :

- Modifiez l'exemple précédent en rajoutant une classe étendue par A et B, possédant aussi une méthode `who_am_i`
- Modifiez ces méthodes dans A et B pour qu'en plus elles appellent la méthode `who_am_i` sur `super()`. Exécutez le code. Que constatez-vous ?
- Renseignez-vous sur l'utilisation de `super` et sur le MRO (Method Resolution Order) en cas d'héritage multiple.

3 Abstraction

Le principe de l'abstraction est de fournir des informations sur un type d'objet tout en maintenant caché son implémentation. Cela permet de créer un "super-type" générique que des sous-classes pourront implémenter comme elles le souhaitent.

3.1 Classes abstraites

Une classe abstraite permettra de définir des propriétés et des méthodes, utilisables par des sous-classes. Elle ne pourra (normalement) pas être instanciée. En Python, la classe `ABC` (Abstract Base Class) est la classe de base d'une classe abstraite. En étendant celle-ci, on peut créer sa propre classe abstraite.

```
from abc import ABC

class MyAbstractClass(ABC):

    def say_hello(self):
        print("Hello!")

class MyConcreteClass(MyAbstractClass):
    pass

m = MyConcreteClass()
m.say_hello()           # Affiche Hello!
m = MyAbstractClass()   # On peut ici creer une instance car MyAbstractClass ne
                        # possede de pas de methode abstraite
```

3.2 Méthodes abstraites

Une méthode abstraite est une méthode pour laquelle aucune implémentation n'a été définie dans la classe parent. C'est la classe qui l'étend qui devra obligatoirement en fournir une. On utilise le décorateur `@abstractmethod` pour signaler que la méthode est abstraite.

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):

    @abstractmethod
    def say_hello(self):
```

```
        pass

class MyConcreteClass(MyAbstractClass):

    def say_hello(self):
        print("Hello!")

m = MyConcreteClass()
m.say_hello()            # Affiche Hello!
m = MyAbstractClass()    # ERREUR
```

Néanmoins, une méthode abstraite peut quand même contenir une implémentation dans la classe abstraite. Celle-ci pourra alors être appelée avec `super`.

Exercices :
— Modifiez la classe `Animal` en classe abstraite et modifiez `print_info` en méthode abstraite
— Modifiez les sous-classes de `Animal` pour refléter ces changements
— Testez vos classes en créant divers objets qui étendent `Animal` et appelez la méthode `print_info`

4 Exercice récapitulatif

Il vous est demandé d'écrire un programme qui implémente les fonctionnalités et respecte les spécifications suivantes.

4.1 Modèle

- Classe Car avec les propriétés $brand$ pour la marque, $fuel$ pour la jauge de réservoir, km pour le kilométrage
- Classe $Truck$ avec les propriétés $brand$ pour la marque, $fuel$ pour la jauge de réservoir, km pour le kilométrage, $weight$ avec le poids maximum autorisé
- Classe $Boat$ avec les propriétés $brand$ pour la marque, $fuel$ pour la jauge de réservoir, $tonnage$ pour le tonnage
- Classe $Plane$ avec les propriétés $brand$ pour la marque, $fuel$ pour la jauge de réservoir, $range$ pour la distance franchissable maximale
- Classe abstraite $RoadVehicle$ qui reprendra les propriétés communes et sera étendue par Car et $Truck$
- Classe abstraite $Vehicle$ qui reprendra les propriétés communes et sera étendue par $Boat$, $Plane$ et $RoadVehicle$

4.2 Fonctionnalités

Votre programme devra permettre à l'utilisateur de faire trois choses, selon son choix :

- Créer un nouveau véhicule du type de son choix et définir des valeurs pour ses propriétés. Il sera ensuite sauvegardé dans le programme.
- Afficher toutes les informations de chaque véhicule sauvegardé précédemment.
- Quitter le programme.

Renseignez vous sur la fonction $input()$ qui permet de récupérer l'information au clavier.