HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW08

OOP part II

Information Technology

April 9, 2025

FH Zentralschweiz

Content

- Inheritance
  - What is inheritance?
  - The Class “object”
  - Syntax
  - Override
  - Exercise: Animal exercise with inheritance
- Multiple inheritance
  - Concept
  - MRO Introduction
  - MRO Rules
- Methods Overview
  - Object, Class and Static Methods
- Access Modifiers
  - Public, Protected ‘_’, Private ‘__’

HSLU Lucerne University of Applied Sciences and Arts

Inheritance

Information Technology

April 9, 2025

Inheritance: What is Inheritance?

Inheritance creates a hierarchy of classes.

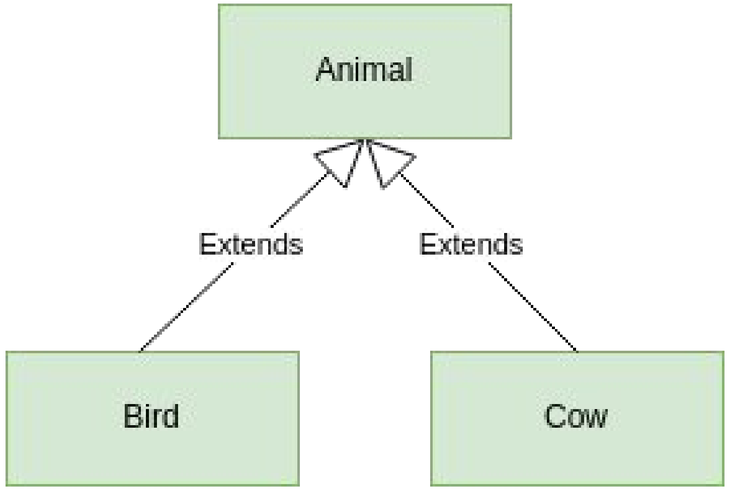
- From a base class “Animal” with basic commonalities,
- go to more specific groups like “Bird” or “Cow”.

Objectives of inheritance:

- avoid code duplication (reuse code).
- increase maintainability.

There are two built-in functions to identify the class of an object:

- `isinstance(object, classinfo)`
  True if object is instance of sub- or class of classinfo.
- `issubclass(class, classinfo)`
  True if class is subclass of classinfo.



`Parent, Super or Base Class`

`Child, Sub or Derived Class`

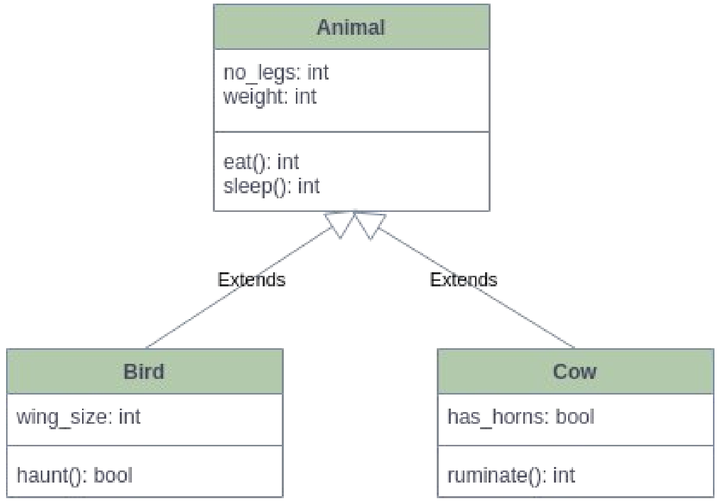
Inheritance: What is Inheritance?

Child classes get all properties (attributes) from parent class. This means, class or object **variables** or **methods** defined in the parent class “**Animal**” also hold in the child classes “**Bird**” and “**Cow**”.

In this way, common (or similar) attributes only need to be defined and maintained in the parent class.

- Size of full application gets reduced.
- Easy to add new subgroups.
- Simple debugging and maintenance.
- Simple to extend functionality.

Typically, child classes **extend** parent classes.



Inheritance: Override

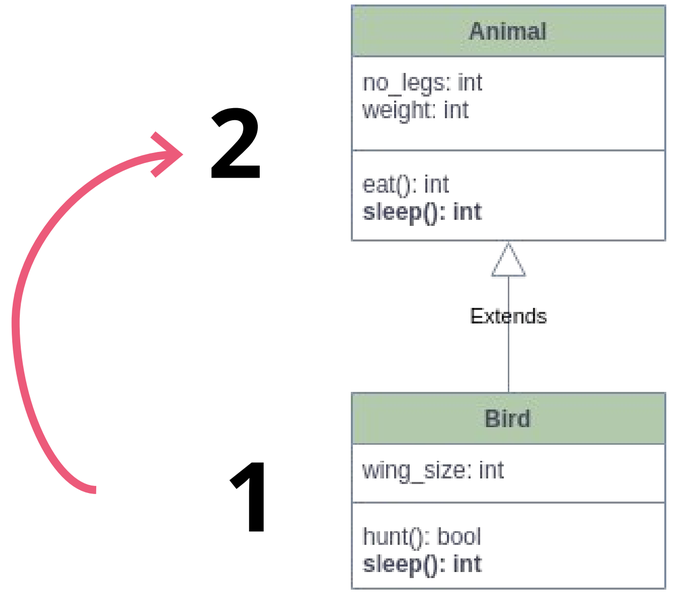
In some cases, child classes do not totally match parent class attributes and require different method behavior. In this case, a child class can **override variables** or **methods** from parent class.

As the name implies, the variable or method defined in base class can be overridden with exactly the same name.

This allows to:

- extend methods with additional functionality or
- completely substitute the behavior of the parent method.

When a method of an object is called, Python always seeks the method in the **class of object** first, before seeking in **parent class**.
-> this also holds for methods such as `__str__()`.

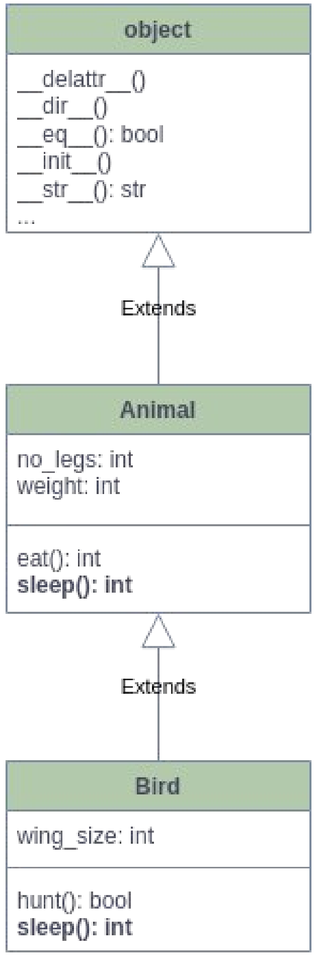


Inheritance: The Class “object”

In Python, class “object” provides basic functionalities.
And **each** class (except object) inherits from the class “object”.

```
class object
 |  The base class of the class hierarchy.
 |
 |  When called, it accepts no arguments and returns a new featureless
 |  instance that has no instance attributes and cannot be given any.
```

Docu: https://docs.python.org/3/library/functions.html#object



Inheritance: Syntax

Similar to a function (or method), a class is able to takeover a parameter – the name of the parent class.
-> when declaring a **child** class, the reference to the **parent** class has to be passed to the class header.

In order to inherit from the parent class “Animal”:

- the parent class “Animal” must be declared before referring to it.
- the parent class name “Animal” has to be passed to the child class header.
- the constructor of the child class must call the constructor of the parents class too.

```
class Animal:
     def __init__(self, weight):
          self.weight = weight

class Bird(Animal):
     def __init__(self, weight):
          super().__init__(weight)
          # or Animal.__init__(self, weight)
```

Inheritance: Syntax

When extending functionality of parent class by **overriding** a method (or in general to **use** parent's methods), they can be referred to using `super()` or parent's class name.

`super()` returns a proxy object that delegates method calls to a parent or sibling class of type.

Docu: https://docs.python.org/3/library/functions.html#super

```
class Animal:
    def __init__(self, weight):
        self.weight = weight

    def get_weight(self):
        return self.weight

class Bird(Animal):
    def __init__(self, weight):
        super().__init__(weight)
        # or Animal.__init__(self, weight)

    def get_weight(self):
        return super().get_weight()/10
        # or return Animal.get_weight(self)/10
```

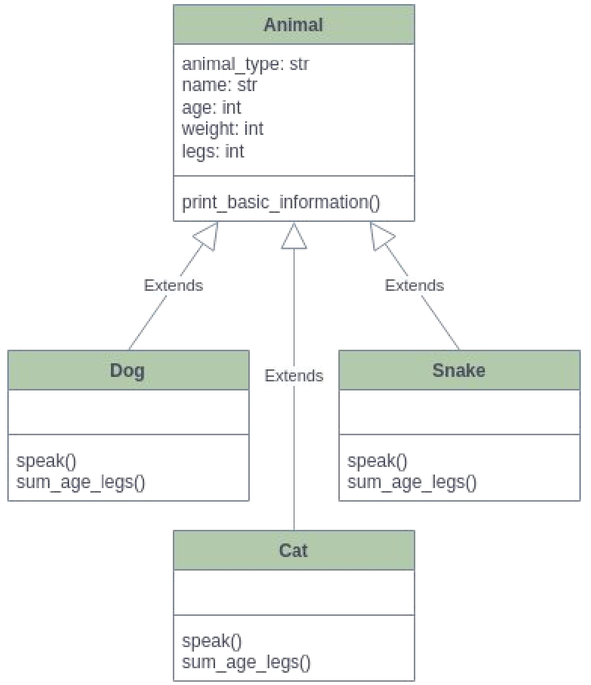
Inheritance: Exercise

Use the file animals.csv to read information about animals.

a) Produce a class structure where Dog, Cat and Snake are implemented as child classes of a base class Animal. Create the object variables in the base class, along with a method 'print_basic_information', which prints 'type' and 'name' to the console.
   - In each child class, use the base class constructor with 'super' and add an additional method 'speak', specific to that child class.
   - In each child class, add an additional method where you return the sum of the object variables 'age' and 'legs'. For each element in the animals.csv, create an object of the corresponding subclass, let it print basic information (method from 'Animal'), let it speak (method from child class) and get the sum of 'age' and 'legs' as a return value.

b) Often, there are many different solutions how one can solve a particular task. Ideas like code readability and avoiding code duplication become more important in more complex applications. Do you find a way to reduce code duplication in the structure of part a)?

c) Override the 'print_basic_information' method in the child class Dog by using the class name as a parameter instead of self.animal_type. You can access the name of the class by: self.__class__.__name__



HSLU Lucerne University of Applied Sciences and Arts

Multiple Inheritance

Information Technology

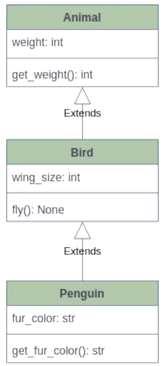
April 9, 2025

Multiple Inheritance: Concept

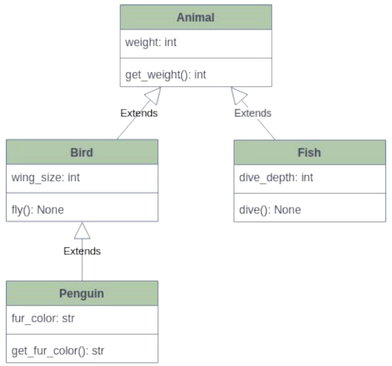
A child class that inherits from a parent class, inherits all except of `__private` attributes from all parent class**es**.

In particular, we have three different scenarios how inheritance may appear:

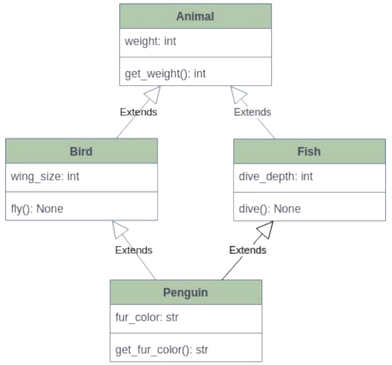
Single inheritance: **one child** class inherits from **one parent** class.



Multiple single inheritance: **multiple child** classes inherit from **one parent** (the same) class.



Multiple inheritance: **one child** class inherits from **multiple parent** classes.



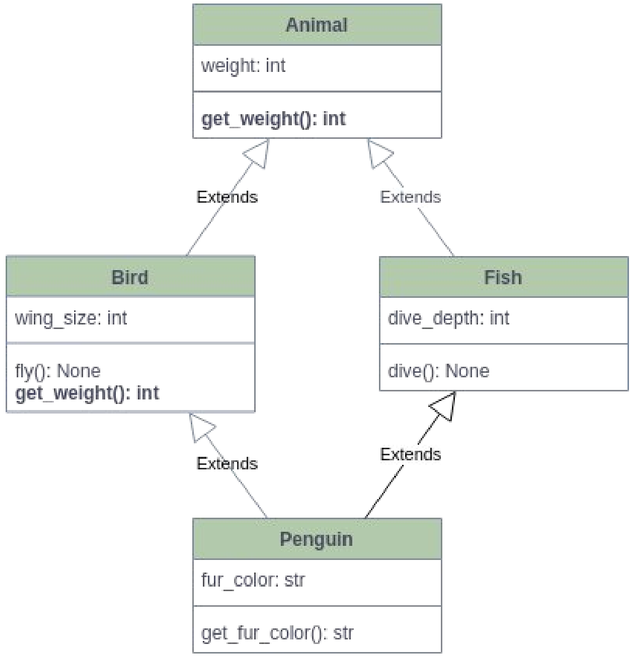
Multiple Inheritance: MRO Introduction

Dealing with multiple inheritance causes conflicts in resolving referenced attributes. This originates from **multiple references** to the same or overridden attributes inherited from **multiple parent classes**.

In the example, the class `Penguin` inherits the attribute **get_weight** from both parent classes: `Bird` and `Fish(Animal)`

- Which `get_weight()` method will be called when an object of class Penguin calls it from parent classes?

```
my_penguin = Penguin()
print(my_penguin.get_weight())
```



To solve these conflicts, by the version 2.3 Python introduced the Method Resolution Order (**MRO**).

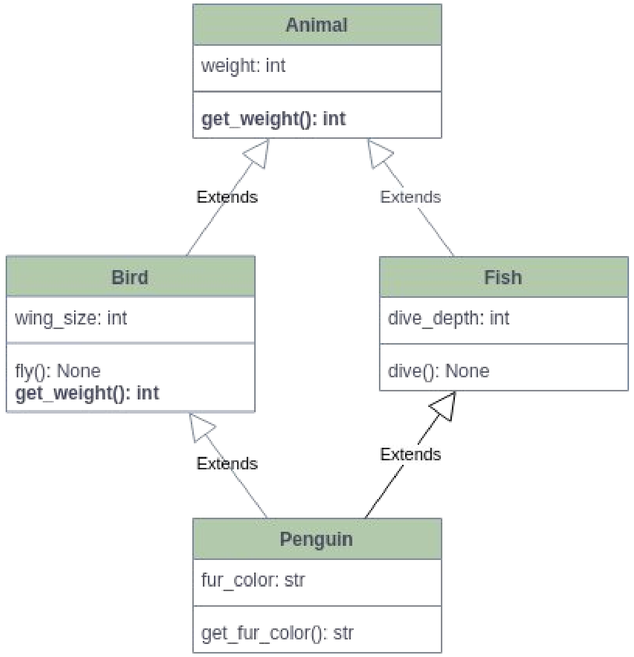
Multiple Inheritance: MRO Rules

The Method Resolution Order (MRO) applies some rules and attempts to find a linearization of all classes.

Docu: https://docs.python.org/3/howto/mro.html

MRO starts from at the root and tries to find linearizations (`L(class)`) for all more specialized classes:

```
L(Animal) = Animal
L(Bird) = Bird
L(Fish) = Fish
L(Penguin) = Penguin + merge(Bird Animal, Fish Animal)
...
L(Penguin) = Penguin Bird Fish Animal
```



The linearization process depends on the order of inheritance.

`class Penguin(Bird, Fish)` or `class Penguin(Fish, Bird)`

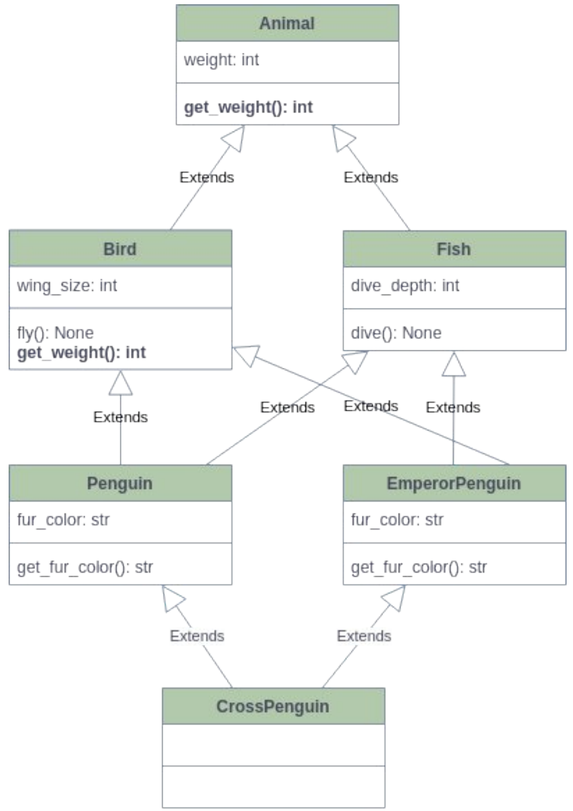
Multiple Inheritance: MRO Rules

Trying to find a linearization for a given **order** of parent classes passed to the child class header, the MRO not always finds a feasible solution.

The given case provokes an MRO inconsistency error:

```
class A(): pass
class B(A): pass
class C(A): pass
class D(B, C): pass
class E(C, B): pass

class F(D, E): pass
```



To solve this, classes D and E must have the **same** inheritance **order**.

```
class D(B, C): pass
class E(B, C): pass
```

Overview Methods

Overview Methods: Object, Class and Static Methods

Wrapping up, Python provides three different types of Methods:

Object Methods

- Belong to a particular **object**.
  -> modifying object's state

```
def get_weight(self):
    return self.weight
```

Class Methods

- Belong to a particular **class**.
  -> modifying class' state

```
class Animal:
    instances = 0

    def __init__(self, weight):
        self.weight = weight
        Animal.instances += 1

    @classmethod
    def reset_instances(cls):
        cls.instances = 0
```

Static Methods

- Don't belong to object nor class.
- Provide operations independent of any object or class state.
- Fully functional: return relies only on input (parameters).

```
class Animal:

    @staticmethod
    def upper_case(name):
        return name.upper()
```

HSLU Lucerne University of Applied Sciences and Arts

Access Modifiers

Information Technology

April 9, 2025

Access Modifiers

For encapsulation purposes, declared variables and methods in a class can have three different states of visibility. The visibility defines what class attributes can be accessed:

1. from an object level (**public**),
2. in declaration of a subclass (**protected**), or
3. in declaration of the own class only (**private**).

Since Python is a script language and does not compile code in advance to execution, it lacks of keywords defining attribute's accessibility. Yet, there is a convention in the **naming** of attributes using **'_'** for treating access restrictions.

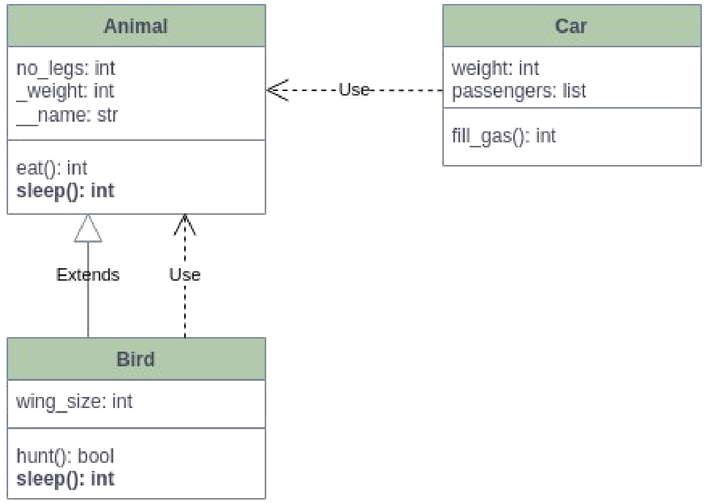
But, this is only a **convention**. Python has no hard restrictions and allows to access all attributes on object level.

Access Modifiers: public

Public attributes can be accessed by any class by means of the object reference. This attributes follow the regular Python naming style without any special characters.

Public attributes can always be accessed by:

- `self.public_attribute` or
- `object.public_attribute`



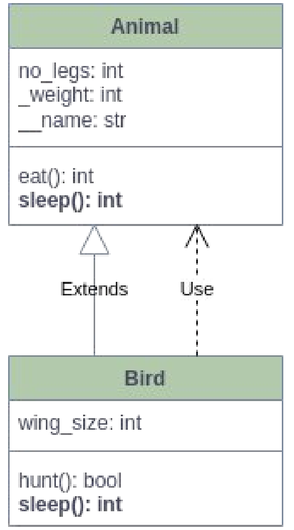
Access Modifiers: protected

Protected attributes can only be accessed from the own class or subclasses. Classes (except subclasses) that instance an object which has protected attributes **must not** access these attributes.

Protected attributes are indicated with a leading underscore '_'
in the attribute name.

Protected attributes can be accessed in **own** or **derived** classes by:

- `self._protected_attribute`



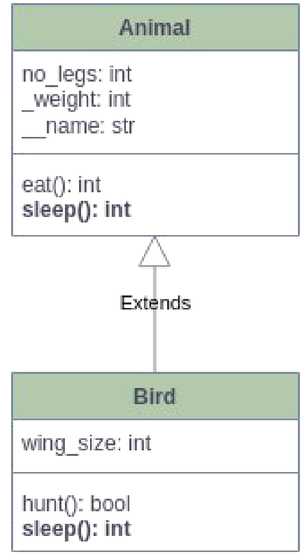
Access Modifiers: private

Private attributes can **only** be accessed from the own class. All classes that instance an object which has private attributes **must not** access these attributes.

Protected attributes are indicated with **two** leading underscore '_ _'
in the attribute name.

Protected attributes can always be accessed by:

- `self.__private_attribute`



HSLU Lucerne University of Applied Sciences and Arts

School of Computer Science and Information Technology
Research
Ramón Christen
Research Associate Doctoral Student

Phone direct +41 41 757 68 96
ramon.christen@hslu.ch

HSLU T&A, Competence Center Thermal Energy Storage
Research
Andreas Melillo
Lecturer

Phone direct +41 41 349 35 91
andreas.melillo@hslu.ch