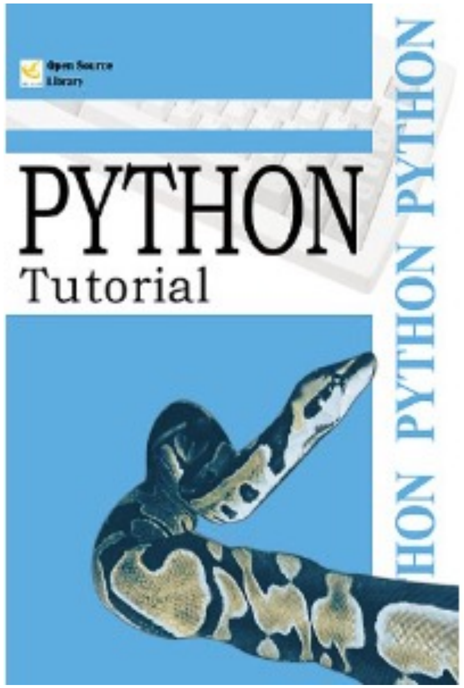
Inheritance



Dra. Mª Dolores Rodríguez Moreno

Universidad de Alcalá

ISG

Objectives

Specific Objectives

- Understanding what is inheritance
- Benefits of inheritance and polymorphism
- Inheritance syntax in Python

Source

- https://docs.python.org/3/reference/
- https://ellibrodepython.com/
- Python Tutorial - Tapa blanda. GuidoVan Rossum (2012)

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- Overriding Methods
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- Inheritance and decorators

Introduction

- Inheritance is a fundamental concept in OOP that allows a new class to be based on an existing class
- The new class (child/derived class) inherits attributes and methods from the existing class (parent/base class)
- Benefits of Inheritance:
  - Code Reusability: reuse common functionality in multiple classes
  - Organized Code: create a clear class hierarchy
  - Polymorphism: achieve dynamic method binding
  - Extensibility: easily extend functionality of existing code

Outline

- Introduction
- **Definition**
- Access Control in Derived Classes
- Overriding Methods
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- Inheritance and decorators

Definition

- Inheritance Syntax: use parentheses () to specify the base class
- Single Inheritance: a derived class inherits from one base class
- Attributes and Methods: derived class inherits all attributes and methods from the base class.

Derived class

```
class DerivedClass(BaseClass):
    # body of the derived class
    pass
```

Example: public attribute

```
class Animal:

    def __init__(self, name):

        self.name = name

    def speak(self):

        pass
```

Example: Dog class

```
class Dog(Animal):

    def speak(self):

        return f"{self.name} says Guau!"
```

Example: Cat class

```
class Cat(Animal):

    def speak(self):

        return f"{self.name} says Miau!"
```

base & subclass

```
print(Dog.__bases__)

# (<class '__main__.Animal'>,)

print(Animal.__subclasses__())

# [<class '__main__.Dog'>, <class
'__main__.Cat'>]
```

Example: Accessing CB's attribute/methods

```
Minino = Cat("Gato")

print(Minino.speak())

print(Minino.name)

Minino.name = "Minino"

print(Minino.name)

#########  Output

Gato says Miau!

Gato

Minino
```

Exercise

- Consider the attributes as it should be, private
- Extend the class(es) with this new change
- Use the corresponding decorators so we can use attributes names

Outline

- Introduction
- Definition
- **Access Control in Derived Classes**
- Overriding Methods
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- Inheritance and decorators

Access Control in Derived Classes

- Public: Accessible anywhere
- Protected: Accessible within the class and its subclasses
- Private: Accessible only within the class itself
- Remember:
  - Public Attributes/Methods: No underscore.
  - Protected Attributes/Methods: Single underscore (_).
  - Private Attributes/Methods: Double underscore (__).

Example: protected attribute

```
class Animal:
    def __init__(self, name):
        self._name = name     # protected attribute
        self._type = 'Animal'  # protected attribute


class Dog(Animal):
    def speak(self):
        return f"{self._type} {self._name} says Guau!"
```

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- **Overriding Methods**
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- Inheritance and decorators

Overriding Methods = Sobrescribir métodos

- Redefining a base class method in a derived class
- Purpose: provide specific implementation in the derived class
- Key Points:
    - Same Method Name: use the same method name and parameters
    - Dynamic Polymorphism: base class reference can refer to a derived class object and invoke overridden methods

Example: overriding method

```
class Animal:

    def speak(self):

        return "Some generic sound"

class Dog(Animal):

    def speak(self):

        return ”Guau!"



my_dog = Dog()

print(my_dog.speak())  # Output: Guau!
```

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- Overriding Methods
- **Calling Base Class Methods**
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- Inheritance and decorators

Calling Base Class Methods

- Definition: *super()* is used to call a method from the base class
- The purpose is to extend/modify the behavior of base class methods
- Advantages:
    - Avoid Direct Base Class Name: *super()* is more maintainable and flexible
    - Works with Multiple Inheritance: resolves the correct method to call in a hierarchy

Example: init

```
class Animal:

    def __init__(self, name):

        self._name = name

        print("Calling Base class method")
```

Example: Option 1

```
class Dog(Animal):

    def __init__(self, name, breed):

        self._name = name

        self._breed = breed

    def speak(self):

        return f"{self._name}, the {self._breed}, says Guau!"
```

Example: init

```
class Animal:

    def __init__(self, name):

        self._name = name

        print("Calling Base class method")
```

Example: super() (Option 2)

```
class Dog(Animal):

    def __init__(self, name, breed):

        super().__init__(name)

        self._breed = breed

    def speak(self):

        return f"{self._name}, the {self._breed}, says Guau!"
```

Inits in Derived Classes

- Base Class *init* is called before the derived class *init*
- The derived class can call the base class *init* using *super()*
- Advantages:
  - Base Class Initialization ensures base class attributes are properly initialized
  - Extended Initialization adds new attributes or initialization steps in the derived class

Example:

```
class Animal:

    def __init__(self, name):

        self.name = name

        print("Animal init called")

class Dog(Animal):

    def __init__(self, name, breed):

        super().__init__(name)

        self.breed = breed

        print("Dog init called")

my_dog = Dog("Buddy", "Golden Retriever")

# Output: # Animal init called\n # Dog init called
```

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- Overriding Methods
- Calling Base Class Methods
- **Duck typing**
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- Inheritance and decorators

Duck Typing

- A concept in Python and other dynamically typed languages where the type or class of an object is determined by its behavior (methods and properties) rather than its explicit class inheritance or type"
- "If it walks like a duck and quacks like a duck, then it probably is a duck."
- Key Idea: focuses on behavior rather than type

How does it works?

- In Python, as long as an object has the necessary methods/attributes, it can be used in place of any other object
- Dynamic typing means we don't check the type explicitly

Pros/Cons

- Benefits
  - Flexible Code: reduces dependency on specific types
  - Promotes Polymorphism: allows for cleaner and more modular code
  - Enhances Testing: easily test objects that mimic necessary behavior
- Disadvantages
  - Error-Prone: without type checking, unexpected objects may cause runtime errors
  - Readability: code can be harder to understand without explicit types
  - Documentation: good documentation is essential to know expected behaviors

Example:

```python
class Duck:

    def quack(self):
        return "Quack!"

    def fly(self):
        return "I'm flying!"

class Person:

    def quack(self):
        return "I'm imitating a duck!"

    def fly(self):
        return "I'm pretending to fly!"

def let_it_fly_and_quack(entity):
    print(entity.quack())
    print(entity.fly())
```

Example:

```
# Works with both Duck and Person

let_it_fly_and_quack(Duck())

let_it_fly_and_quack(Person())

#Output

Quack!

I'm flying!

I'm imitating a duck!

I'm pretending to fly!
```

Examples of Duck Typing

- Any example with Duck Typing so far previously seen?

Exercise

- Crea un sistema que maneje distintos tipos de empleados en una empresa
- Define una clase base *Employee* y crea clases derivadas *Manager* y *Developer*
  - Atributos: name (público), _id (protegido), __salary (privado).
  - Método público display_details() para mostrar el nombre y el ID del empleado.
  - Los atributos deben estar controlados con decoradores para acceso seguro.
  - Define un método work() con una implementación general.
- Clases Derivadas:
  - Manager: Sobrescribe el método work() para imprimir "Overseeing the team"
  - Developer: Sobrescribe el método work() para imprimir "Writing code"
  - display_details(self): además del rol en cada clase, debe imprimir: name, Id
- Duck Typing: crea una función *perform_work()* que tome un objeto de tipo *Employee* (o cualquiera que tenga un método *work()*) y llame a su método *work()*.

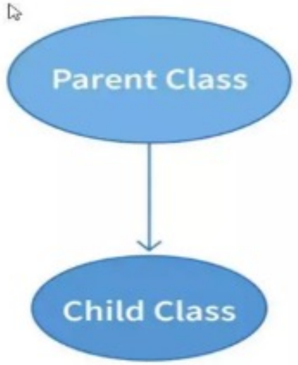
Outline

Types of inheritance

- Single Inheritance: a child class inherits from one parent class
- Multiple Inheritance: a child class inherits from multiple parent classes
- Multilevel Inheritance: a child class inherits from a parent class, which in turn inherits from another class
- Hierarchical Inheritance: multiple child classes inherit from a single parent class
- Hybrid Inheritance: a combination of 2 or more of the previous ones

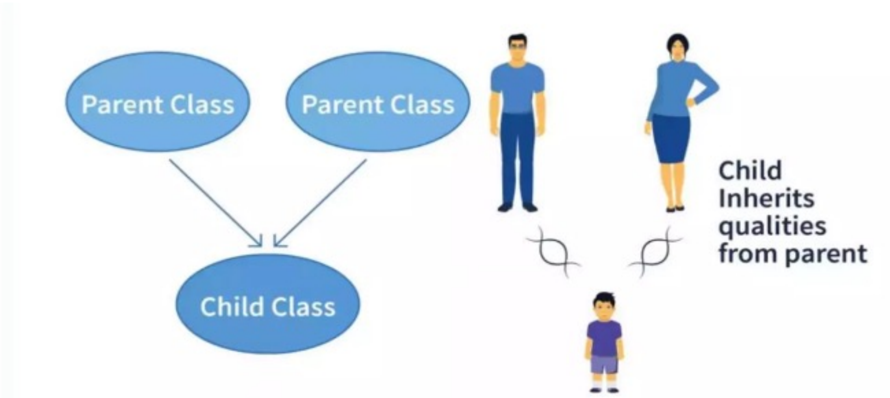
Single inheritance

- It is the simplest form of inheritance
- Also called simple inheritance



Multiple inheritance

- A single child class is inherited from 2 or more parent classes
- A child class has access to all parents' attributes and methos



Example:

```
class A:
    def method_a(self):
        return "Method A"
class B:
    def method_b(self):
        return "Method B"
class C(A, B):
    pass
c = C()
print(c.method_a())  # Output: Method A
print(c.method_b())  # Output: Method B
```

Method Resolution Order (MRO)

- MRO determines the order in which Python searches for methods in inheritance
- It uses the C3 Linearization algorithm
- You can check the MRO using the __mro__ attribute or the mro() method

MRO

```
print(C.__mro__)

# Output: (<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 
'object'>)
```

Example:

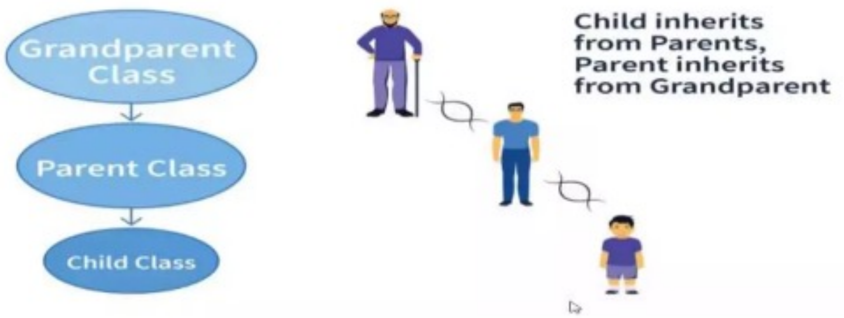
```
class A:
    def method(self):
        print("A method")
class B(A):
    def method(self):
        print("B method")
class C(A):
    def method(self):
        print("C method")
class D(B, C):
    pass
```

Example:

```
# Example usage
d = D()
d.method()  # Output: B method
print(D.mro())
#Output: [D, B, C, A, object]
```

Multilevel inheritance

- It introduces more levels than just parent and child (2 levels seen until now)
- A child can be a parent of another child class



Multilevel:

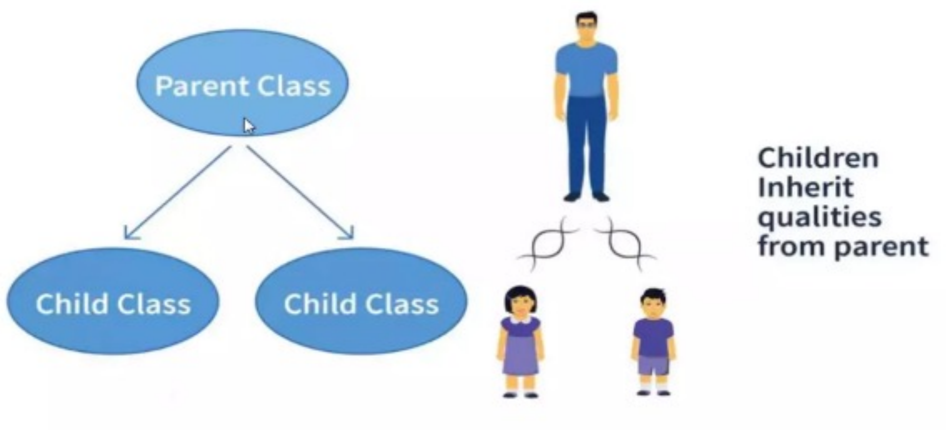
```
class grandparent:

    def func1(self):

        print("Hello Grandparent")  # first level


class parent(grandparent):

    def func2(self):

        print("Hello Parent")  # second level


class child(parent):

    def func3(self):

        print("Hello Child")  # third level
```

Multilevel:

```
test = child()   # object created
test.func1()     # 3rd level calls 1st level
test.func2()     # 3rd level calls 2nd level
test.func3()     # 3rd level calls 3rd level


# Output:
# Hello Grandparent
# Hello Parent
# Hello Child
```

Hierarchical inheritance

- There are multiple derived classes from a single parent class



Hierarchical

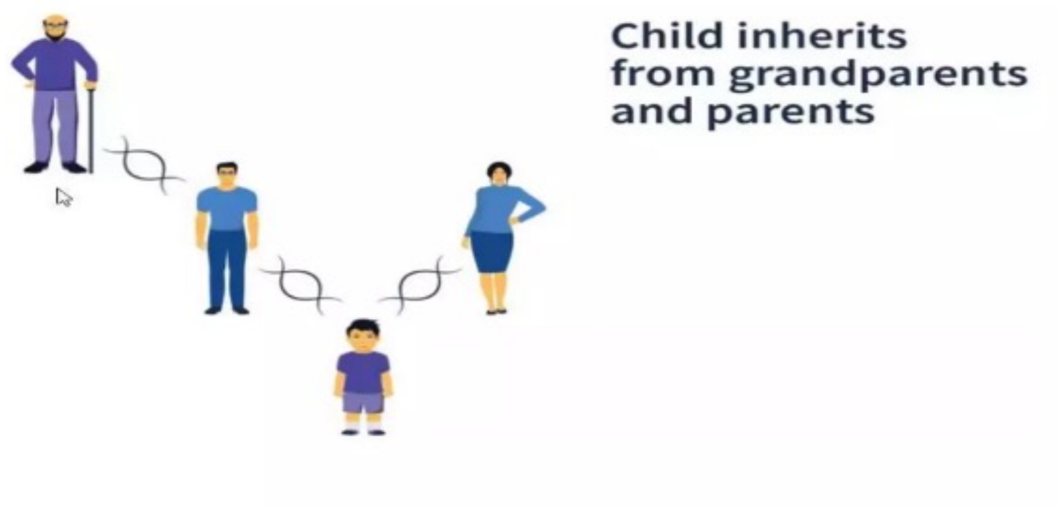
```
class parent1:

    def func1(self):

        print("Hello Parent")

class child1(parent1):

    def func2(self):

        print("Hello Child1")

class child2(parent1):

    def func3(self):

        print("Hello Child2")
```

Hierarchical

```
test1 = child1()

test2 = child2()



test1.func1()   # child1 calling parent1 method

test1.func2()   # child1 calling its own method



test2.func1()   # child2 calling parent1 method

test2.func3()   # child2 calling child2 method
```

Hybrid inheritance

- It is the mix of 2 or more types of inheritance already explained before



Hybrid

```
class parent1:
    def func1(self):
        print("Hello Parent")
class parent2:
    def func2(self):
        print("Hello Parent2")
class child1(parent1):
    def func3(self):
        print("Hello Child1")
class child2(child1, parent2):
    def func4(self):
        print("Hello Child2")
```

Hybrid

```
# Driver Code

test1 = child1()

test2 = child2()

test1.func1()   # child1 calling parent1 method

test1.func3()   # child1 calling its own method



test2.func1()   # child2 calling parent1 method

test2.func2()   # child2 calling parent2 method

test2.func3()   # child2 calling child1 method

test2.func4()   # child2 calling its own method
```

Outline

Abstract Base Classes

- Abstract Base Classes (ABCs) are classes that are meant to be inherited from, but not instantiated
- They're defined in the *abc module*
- They can have abstract methods that must be implemented by child classes

Example:

```python
from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod

    def speak(self):

        pass

class Dog(Animal):

    def speak(self):

        return "Woof!"

# dog = Animal()  # This would raise an error

dog = Dog()

print(dog.speak())  # Output: Woof!
```

Outline

Attributes with the Same Name

- Redefining an attribute of the parent class in the child class hides the one from the parent class
- To access the parent class attribute, you can use the parent class name

Example:

```
class Parent:
    x = 1
class Child(Parent):
    x = 2
    def print_x(self):
        print(f"x in Child: {self.x}")
        print(f"x in Parent: {Parent.x}")
c = Child()
c.print_x()
# Output: # x in Child: 2\n # x in Parent: 1
```

Example:

```
class Parent:
    x = 1
class Child(Parent):
    x = 2
    def print_x(self):
        print(f"x in Child: {self.x}")
        print(f"x in Parent: {super().x}")  #????????
c = Child()
c.print_x()
# Output: # x in Child: 2\n # x in Parent: 1
```

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- Overriding Methods
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- **Class and Static Methods**
- Mixins
- Inheritance and decorators

Class and Static Methods

- Class and static methods are inherited just like regular methods
- A single copy is maintained for the class that defines them and for its derivatives

Example:

```
class Parent:

    @classmethod

    def class_method(cls):

        print(f"Class method called from {cls.__name__}")

    @staticmethod

    def static_method():

        print("Static method")

class Child(Parent):
   pass

Child.class_method()    # Output: Class method called from Child
Parent.class_method()   # Output: Class method called from Parent
Child.static_method()   # Output: Static method
```

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- Overriding Methods
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- **Mixins**
- Inheritance and decorators

Mixins

- Mixins are classes that provide additional methods to other classes
- They're often used with multiple inheritance to add functionality
- Benefit: isolate the functionality implemented in a different module, thus being able to reuse it in multiple different contexts.

Example Mixin

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass
class TerrestrialMixin:  # Mixin para animales terrestres
    def walk(self):
        return f"{self.name} camina por la tierra."
class AquaticMixin:  # Mixin para animales acuáticos
    def swim(self):
        return f"{self.name} nada en el agua."
```

Example Mixin

```
class Dog(Animal, TerrestrialMixin):

    def speak(self):

        return f"{self.name} dice Guau!"

class Fish(Animal, AquaticMixin):

    def speak(self):

        return f"{self.name} hace burbujas.”

dog = Dog("Buddy")

print(dog.speak())  # Output: Buddy dice Guau!

print(dog.walk())   # Output: Buddy camina por la tierra.

fish = Fish("Nemo")

print(fish.speak())  # Output: Nemo hace burbujas.

print(fish.swim())   # Output: Nemo nada en el agua.
```

Outline

- Introduction
- Definition
- Access Control in Derived Classes
- Overriding Methods
- Calling Base Class Methods
- Duck typing
- Types of inheritance
- Abstract Base Classes
- Attributes with the same name
- Class and Static Methods
- Mixins
- **Inheritance and decorators**

Inheritance and Decorators

- Decorators can be inherited and overridden
- This allows modifying method behavior in subclasses

Example: Decorator

```
def log_call(func):
    def wrapper():
        print(f"Calling {func.__name__}")
        return func()
    return wrapper
class Parent:
    @log_call
    def greet(self):
        print("Hello from Parent")
```

```
class Child(Parent):
    @log_call
    def greet(self):
        super().greet()
        print("Hello from Child")
Child().greet() #ERROR -  1parameter
```

Example: Decorator

```python
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
class Parent:
    @log_call
    def greet(self):
        print("Hello from Parent")
```

Example: Decorator

```python
class Child(Parent):
    @log_call
    def greet(self):
        super().greet()
        print("Hello from Child")
Child().greet()
# Output:
# Calling greet
# Calling greet
# Hello from Parent
# Hello from Child
```

Advantage of inheritance

- Modular Codebase: Increases modularity, i.e., breaking down codebase into modules, making it easier to understand. Each class we define becomes a separate module that can be inherited separately by one or many classes

- Code Reusability: the child class copies all the attributes and methods of the parent class into its class and use. It saves time and coding effort by not rewriting them, thus following modularity paradigms

- Less Development and Maintenance Costs: changes need to be made in the base class; all derived classes will automatically follow

Disadvantages of Inheritance

- Decreases the Execution Speed: loading multiple classes because they are interdependent
- Tightly Coupled Classes: this means that even though parent classes can be executed independently, child classes cannot be executed without defining their parent classes