AI VIETNAM
All-in-One Course

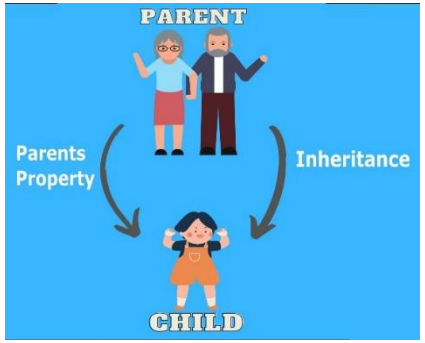
Object-Oriented Programming

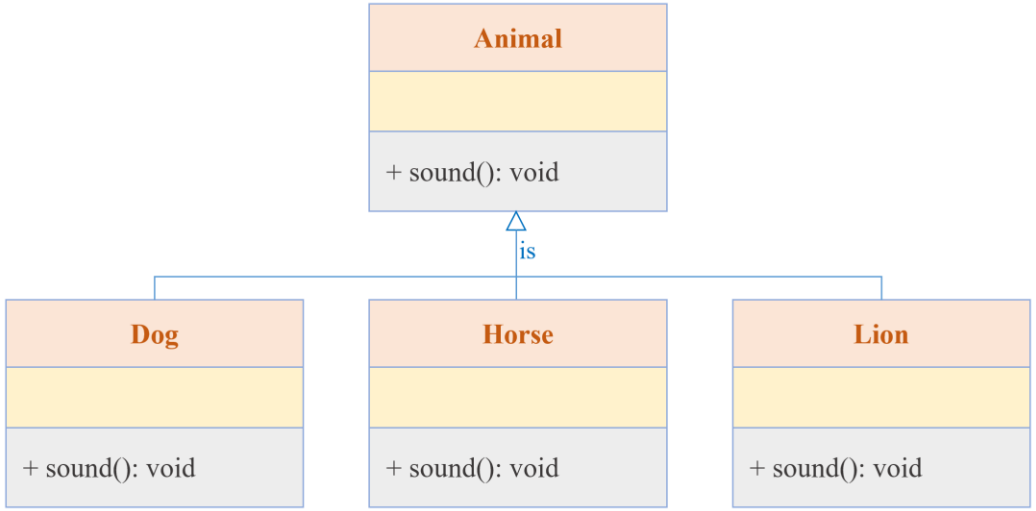
(Inheritance and Applications)

Year 2024

Objectives

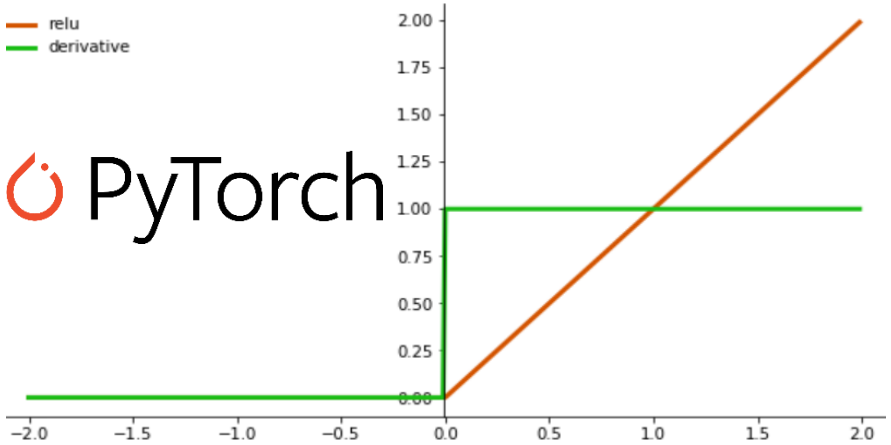
Inheritance





Applications

PyTorch



AI VIET NAM
All-in-One 2024

Review: Access Modifiers

❖ For a class

Private (__name):	Use only within the class
Public (name):	Can use everywhere
Protected (_name):	Should use within the class*

*Accessible from anywhere but intended for internal use. This is a convention rather than enforced protection.

```python
# public
class Cat:
    def __init__(self):
        self.name = 'Calico'

    def describe(self):
        print(self.name)

# test
a_cat = Cat()
a_cat.describe()
print(a_cat.name)
```

```
Calico
Calico
```

```python
# protected
class Cat:
    def __init__(self):
        self._name = 'Calico'

    def describe(self):
        print(self._name)

# test
a_cat = Cat()
a_cat.describe()
print(a_cat._name)
```

```
Calico
Calico
```

```python
# private
class Cat:
    def __init__(self):
        self.__name = 'Calico'

    def describe(self):
        print(self.__name)

# test
a_cat = Cat()
a_cat.describe()
print(a_cat.__name)
```

```
Calico

---------------------------------------------------------
AttributeError                            Traceback (m
Cell In[5], line 12
     10 a_cat = Cat()
     11 a_cat.describe()
---> 12 print(a_cat.__name)

AttributeError: 'Cat' object has no attribute '__name'
```

Outline

SECTION 1

Inheritance: To Reuse

SECTION 2

Inheritance: Overriding

SECTION 3

Inheritance: As a Template

SECTION 4

Multiple/Multilevel Inheritance

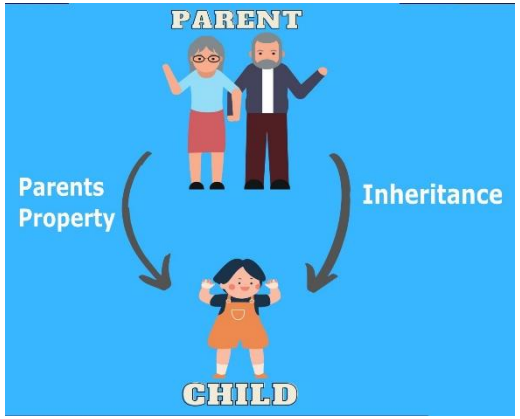
SECTION 5

Custom Class in PyTorch

Inheritance

❖ Introduction

Mechanism by which one class is allowed to inherit the features (attributes and methods) of another class.



Super Class: The class whose features are inherited is known as superclass (a base class or a parent class).

Subclass: The class that inherits the other class is known as subclass (a derived class, extended class, or child class).

The subclass can add its own attributes and methods in addition to the superclass attributes and methods.

Inheritance

❖ Introduction

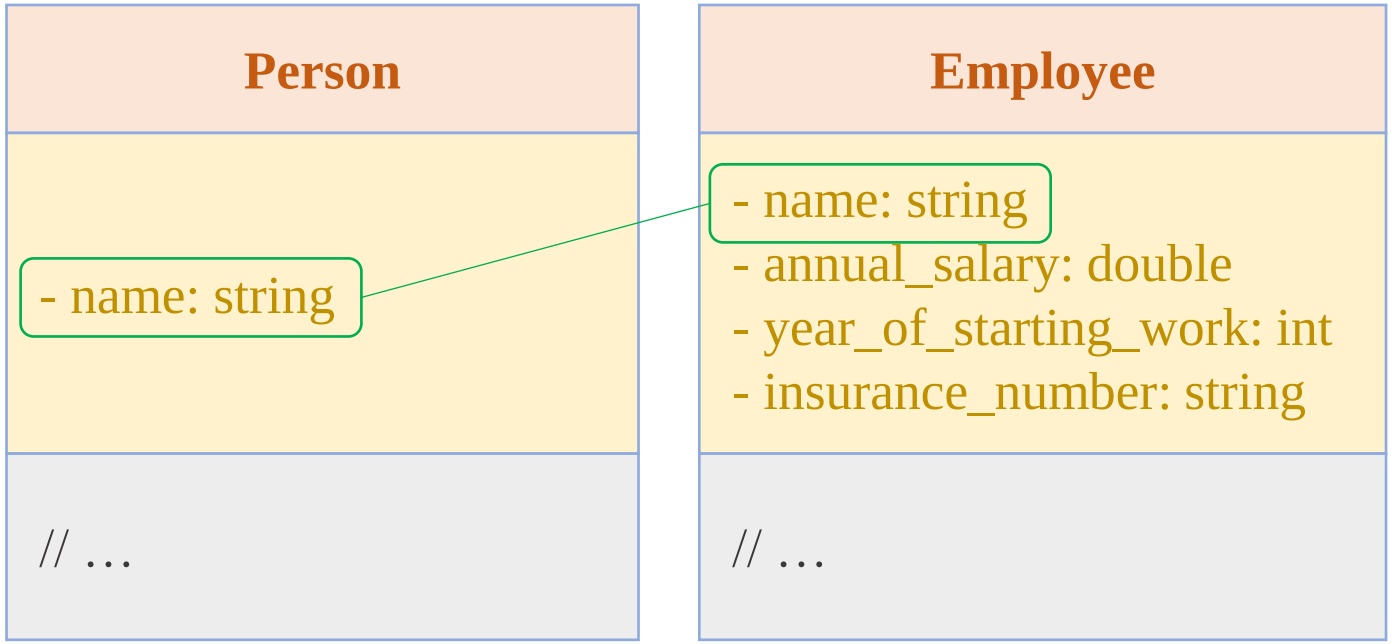
Create a class called Employee whose objects are records for an employee. This class will be a derived class of the class Person.

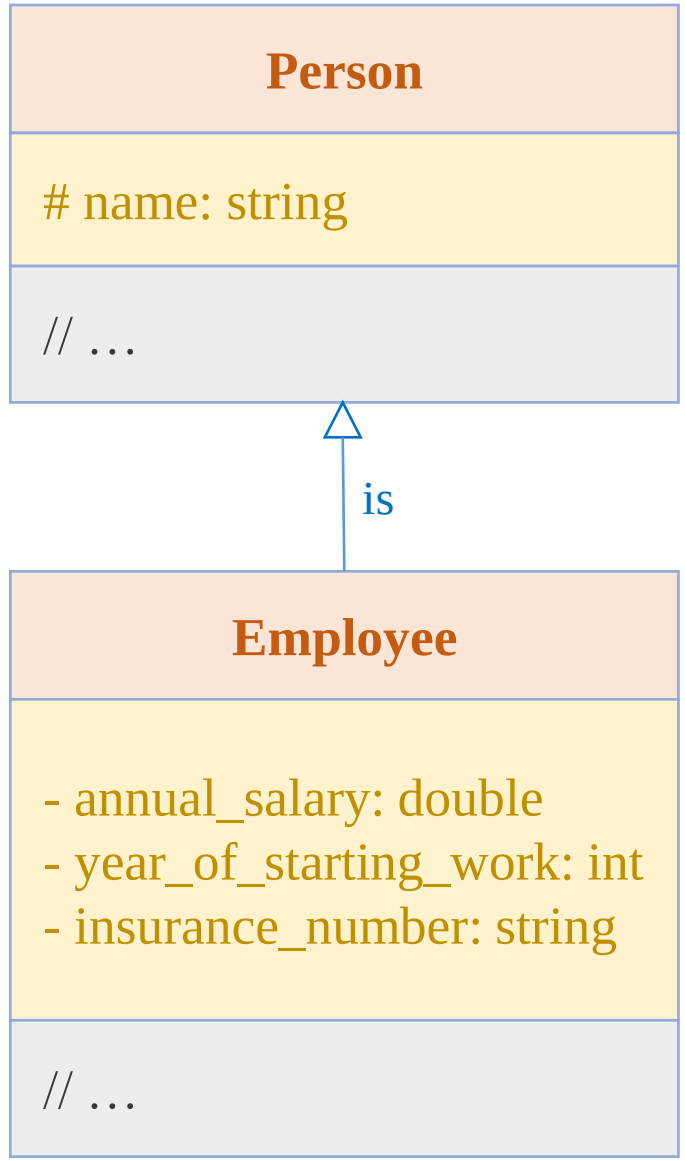
An employee record has an employee's name (inherited from the class Person), an annual salary represented as a single value of type double, a year the employee started work as a single value of type int and a national insurance number, which is a value of type String.

Person
- name: string
// …

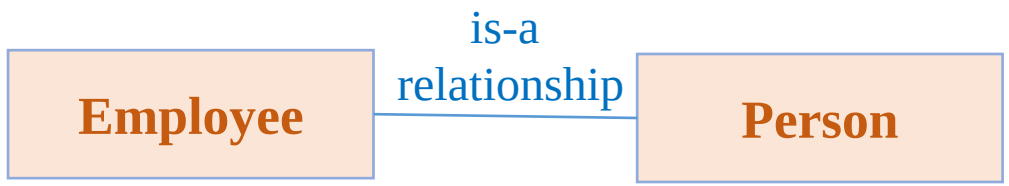
Employee
- name: string - annualSalary: double - yearOfStartingWork: int - insuaranceNumber: string
// …

Inheritance





Access modifiers
- private
+ public
# protected

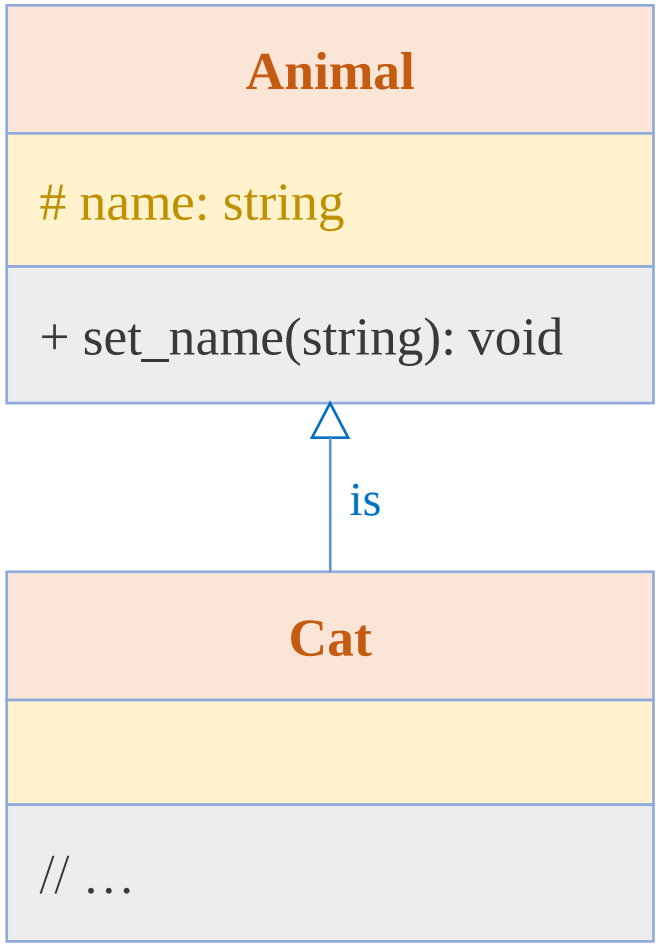


An employee is a person.

Inheritance

Inherit attributes and methods from one class to another

Benefit: Code reusability



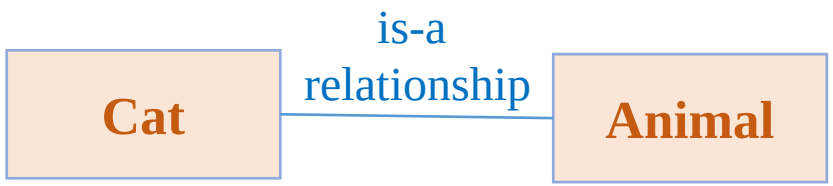

```python
class Animal:
    def __init__(self, name):
        self._name = name

    def set_name(self, name):
        self._name = name

    def describe(self):
        print(self._name)

class Cat(Animal):
    def __init__(self, name):
        super().__init__(name)

# Test creating a Cat object
test_cat = Cat("Calico")
test_cat.describe()
```

```
Calico
```

Derived class (child) - the class that inherits from another class

Base class (parent) - the class being inherited from

DerivedClass(BaseClass)



A cat is an animal.

Example

❖ Implement the two classes below

Math1
+ is_even(int): bool + factorial(int): int

Math2
+ is_even(int): bool + factorial(int): int + estimate_euler(int): double

Example

❖ Implement the two classes below

Math1
+ is_even(int): bool + factorial(int): int

```python
class Math1:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

```python
# test Math1
math1 = Math1()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math1.is_even(5))
print(math1.is_even(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math1.factorial(4))
print(math1.factorial(5))
```

✓ 0.0s

```
False
True
24
120
```

Example

❖ Implement the two classes below

$$e = 2.71828$$

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$$

Math2
+ is_even(int): bool + factorial(int): int + estimate_euler(int): double

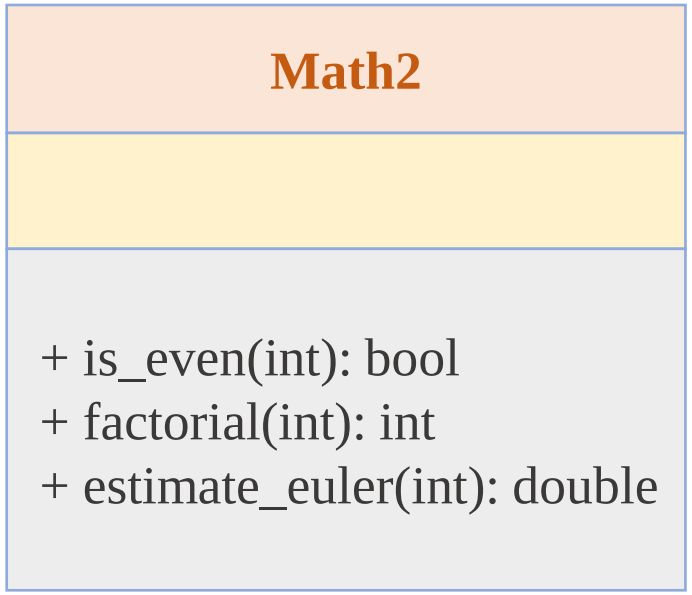
```python
class Math2:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result

    def estimate_euler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/self.factorial(i)

        return result
```

Example

❖ Implement the two classes below

$$e = 2.71828$$

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$$




```python
# test Math2
math2 = Math2()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math2.is_even(5))
print(math2.is_even(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math2.factorial(4))
print(math2.factorial(5))

# estimateEuler() sample: number=2 -> 2.5
# estimateEuler() sample: number=8 -> 2.71
print(math2.estimate_euler(2))
print(math2.estimate_euler(8))
```

✓ 0.0s

```
False
True
24
120
2.5
2.71827876984127
```

Example

❖ How to reuse an existing class?

Math1
+ is_even(int): bool + factorial(int): int

```python
class Math1:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

Math2
+ is_even(int): bool + factorial(int): int + estimate_euler(int): double

```python
class Math2:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result

    def estimate_euler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/self.factorial(i)

        return result
```

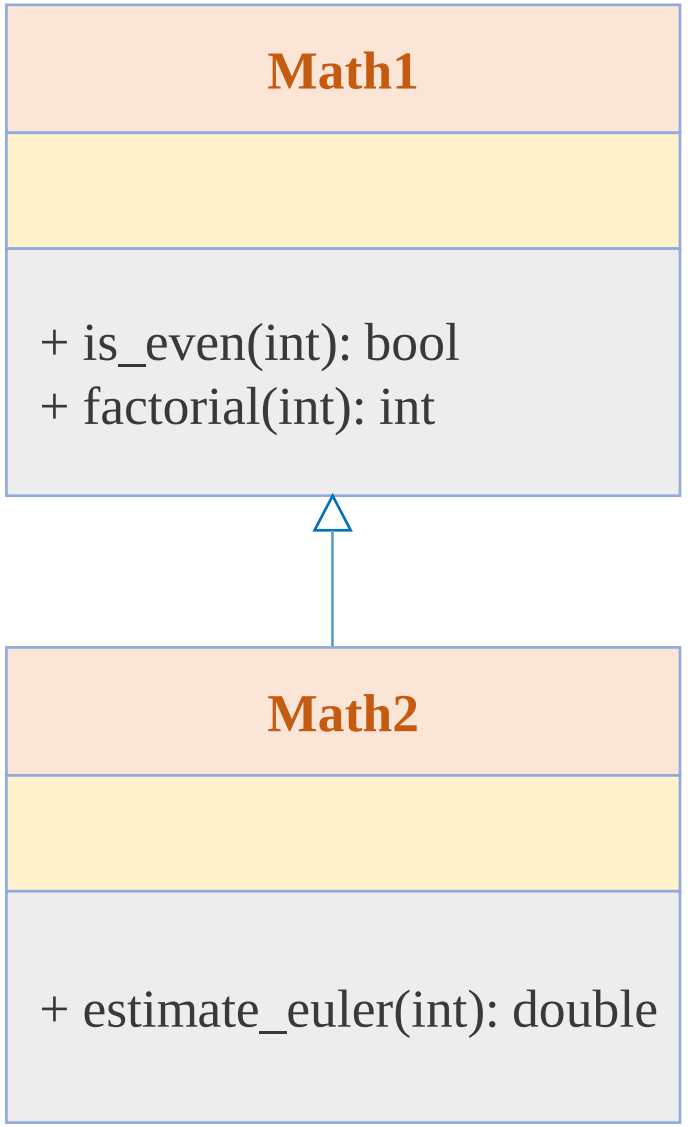
Example

❖ Inheritance

Math1: super class or parent class

Math2: child class or derived class

Child classes can use the public and protected attributes and methods of the super classes.



```python
class Math1:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

```python
class Math2(Math1):
    def estimate_euler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/self.factorial(i)

        return result
```

```python
# test Math2
math2 = Math2()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math2.is_even(5))
print(math2.is_even(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math2.factorial(4))
print(math2.factorial(5))

# estimateEuler() sample: number=2 -> 2.5
# estimateEuler() sample: number=8 -> 2.71
print(math2.estimate_euler(2))
print(math2.estimate_euler(8))
```

✓ 0.0s

```
False
True
24
120
2.5
2.71827876984127
```

```python
class Math1:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

✓ 0.0s

```python
class Math2(Math1):
    def estimate_euler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/super().factorial(i)

        return result
```

✓ 0.0s

```python
# test Math2
math2 = Math2()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math2.is_even(5))
print(math2.is_even(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math2.factorial(4))
print(math2.factorial(5))

# estimateEuler() sample: number=2 -> 2.5
# estimateEuler() sample: number=8 -> 2.71
print(math2.estimate_euler(2))
print(math2.estimate_euler(8))
```

✓ 0.0s

```
False
True
24
120
2.5
2.71827876984127
```

Outline

SECTION 1

Inheritance: To Reuse

SECTION 2

Inheritance: Overriding

SECTION 3

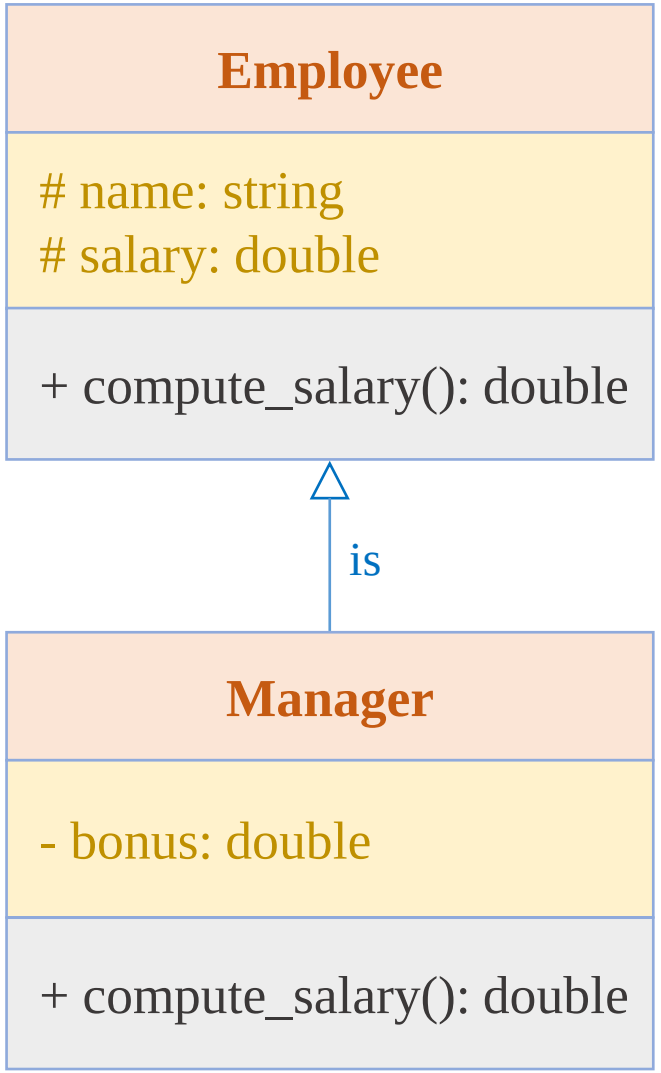
Inheritance: As a Template

SECTION 4

Multiple/Multilevel Inheritance

SECTION 5

Custom Class in PyTorch



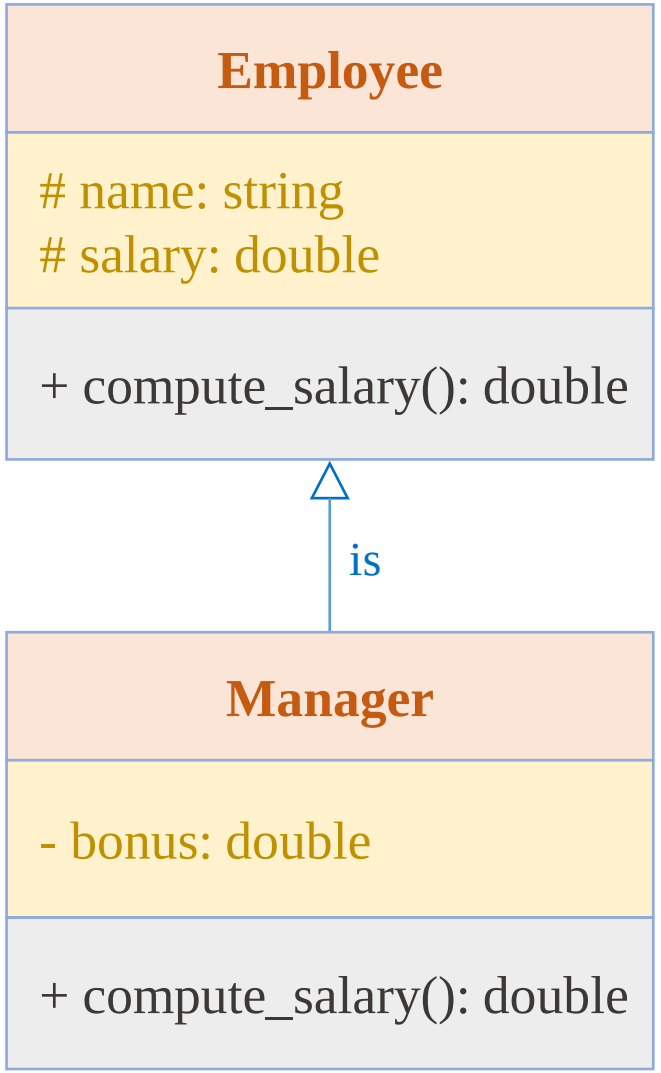
Inheritance

❖ Introduction

To extend an existing class

UML Annotation
'-' stands for 'private'
'#' stands for 'protected'
'+' stands for 'public'

Super Class



Subclass

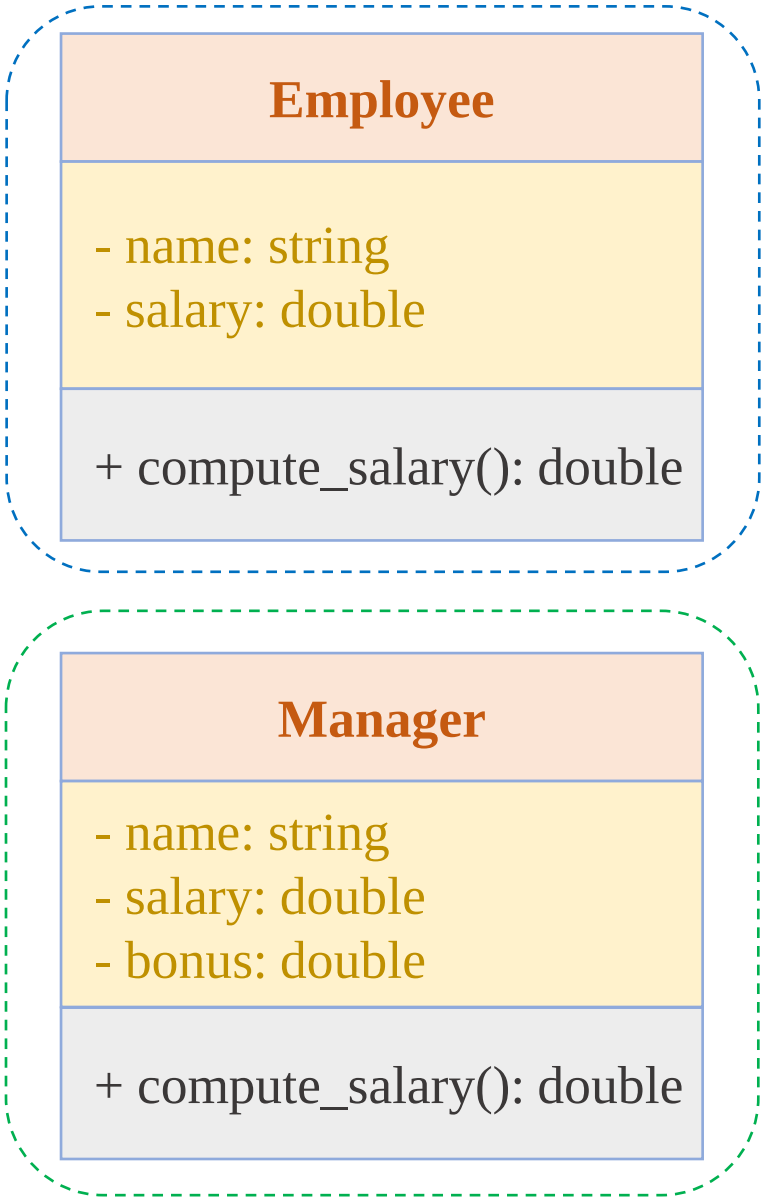
What features does a manager inherit?

Another Example

❖ Employee-Manager Example: Simple requirement

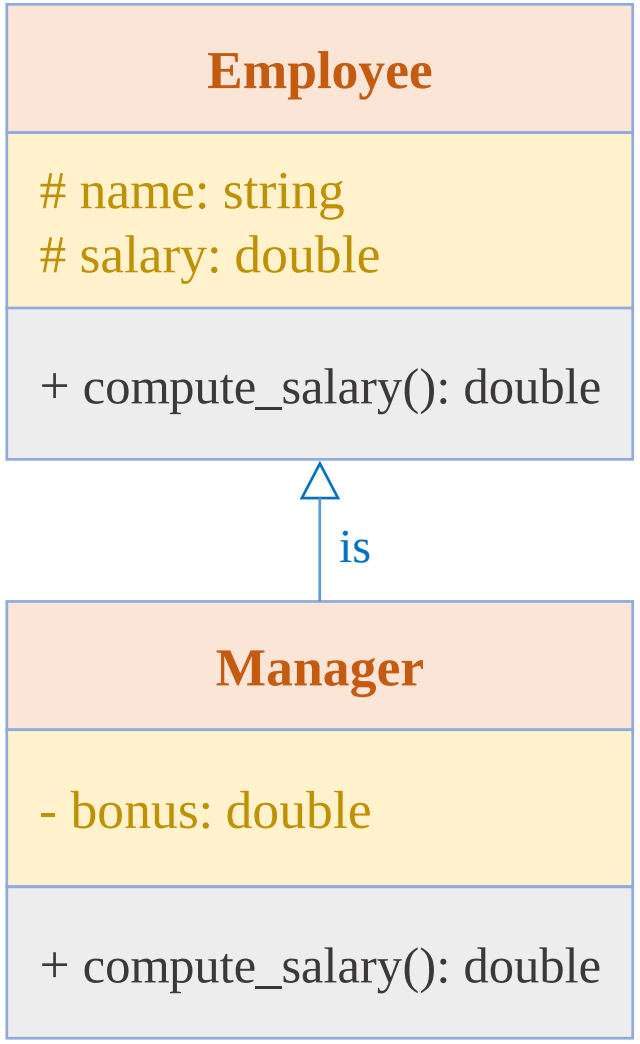
A standard employee of company X includes his/her name and base salary. For example, Peter is working for X, and his base salary is 60000$ a year. Implement the Employee class and the computeSalary() method to compute the final salary for an employee. The salary for an employee is his/her base salary*3.0.

A manager includes his/her name, base salary, and bonus. The final salary for the manager comprises the base salary and a bonus. For example, Mary is a manager in the company. Her base salary and bonus are 60000$ and 50000$ a year, respectively. Yearly, she gets paid 230000$ a year. Implement the Manager class and the computeSalary() method to compute the final salary.



Another Example

Employee-Manager




```python
class Employee:
    def __init__(self, name, salary):
        self._name = name
        self._salary = salary

    def compute_salary(self):
        return self._salary

class Manager(Employee):
    def __init__(self, name, salary, bonus):
        self._name = name
        self._salary = salary
        self.__bonus = bonus

    def compute_salary(self):
        return super().compute_salary() + self.__bonus
```

✓ 0.0s

```python
peter = Manager('Peter', 100, 20)
salary = peter.compute_salary()
print(f'Peter Salary: {salary}')
```

✓ 0.0s

```
Peter Salary: 120
```

Outline

SECTION 1

Inheritance: To Reuse

SECTION 2

Inheritance: Overriding

SECTION 3

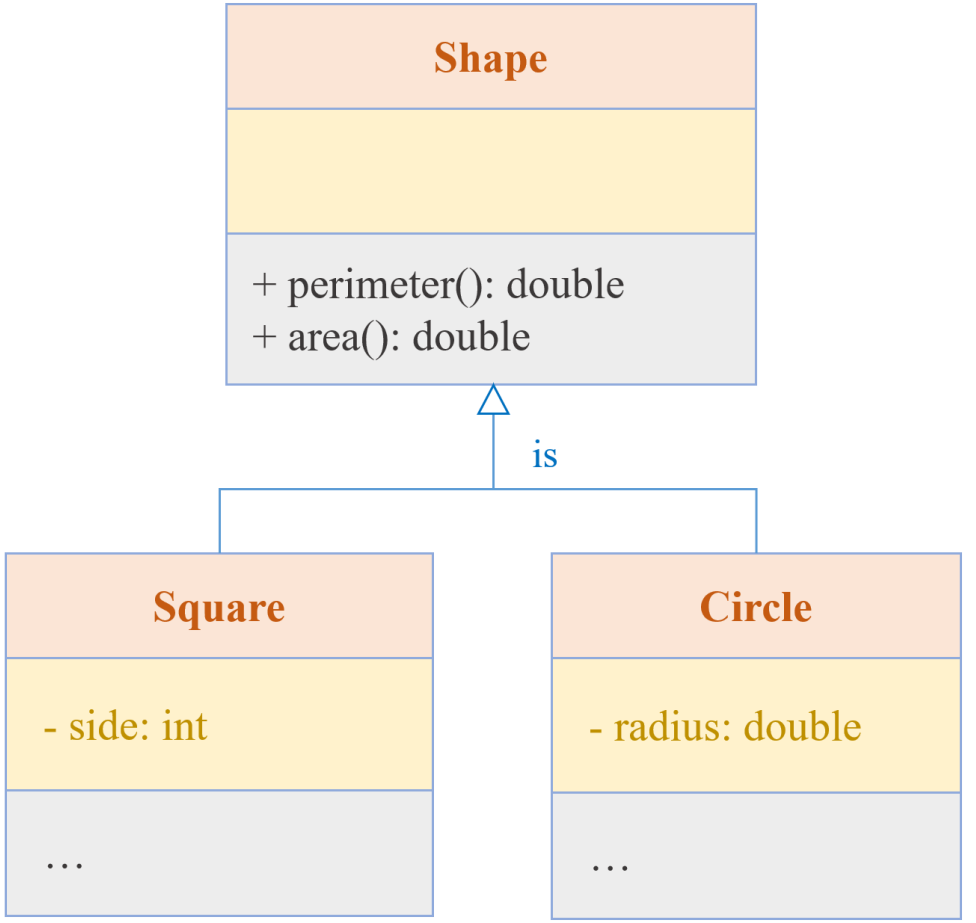
Inheritance: As a Template

SECTION 4

Multiple/Multilevel Inheritance

SECTION 5

Custom Class in PyTorch



Inheritance

❖ As a template

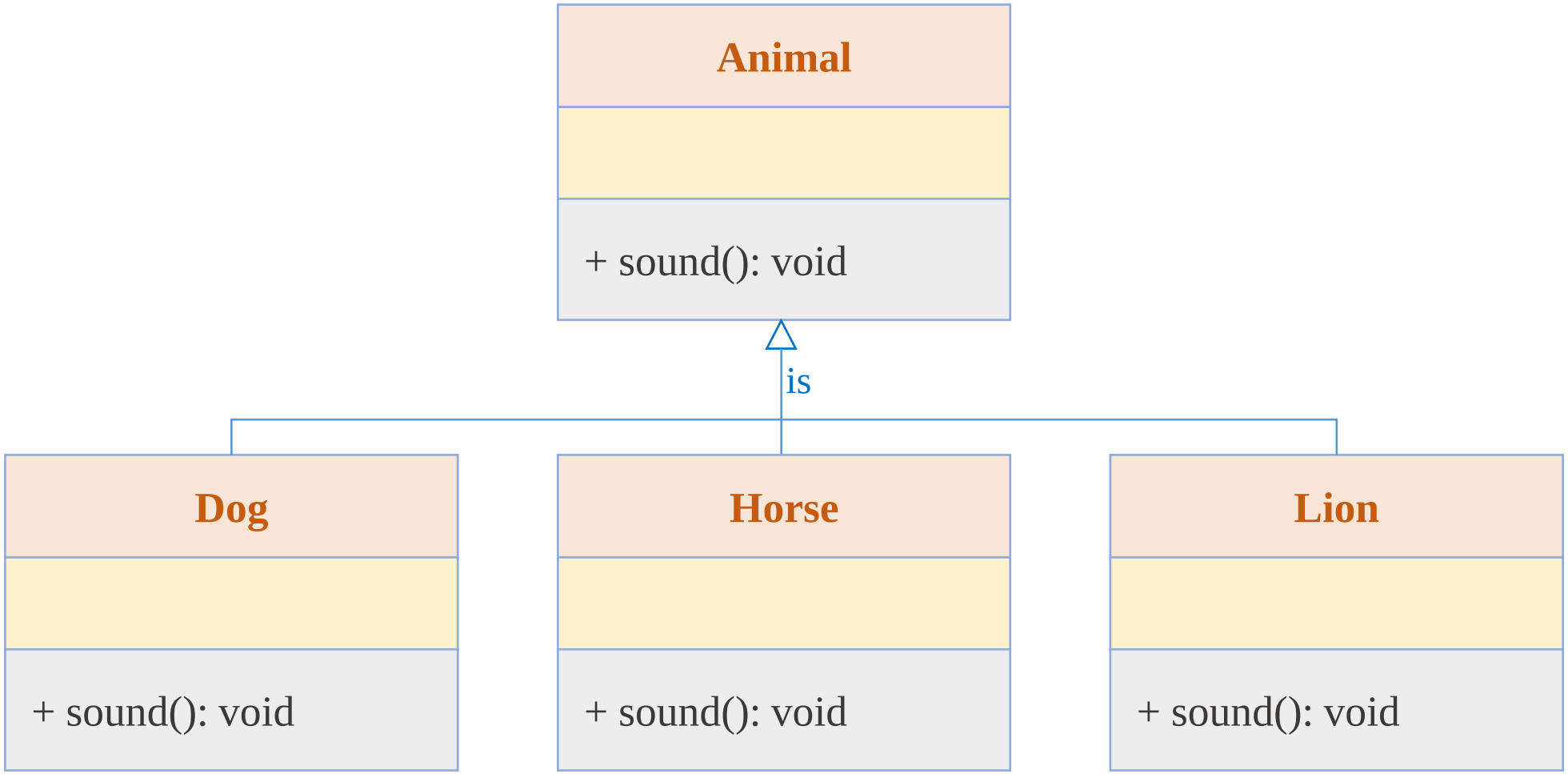
A class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat, etc.

Since the animal sound differs from one animal to another, there is no point to implement this method in parent class.

This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

Inheritance

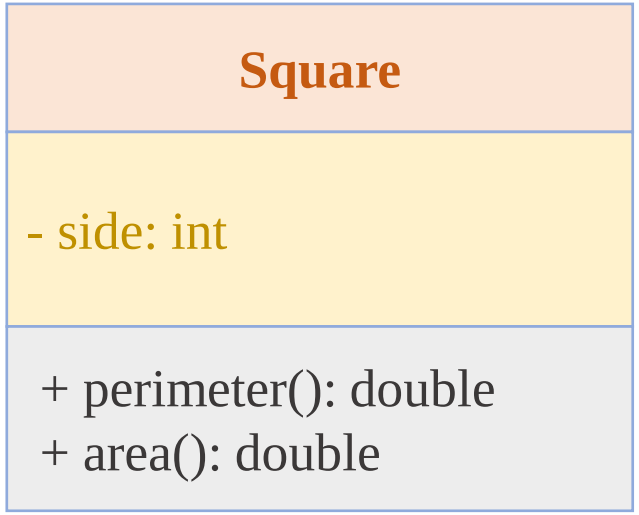
❖ As a template

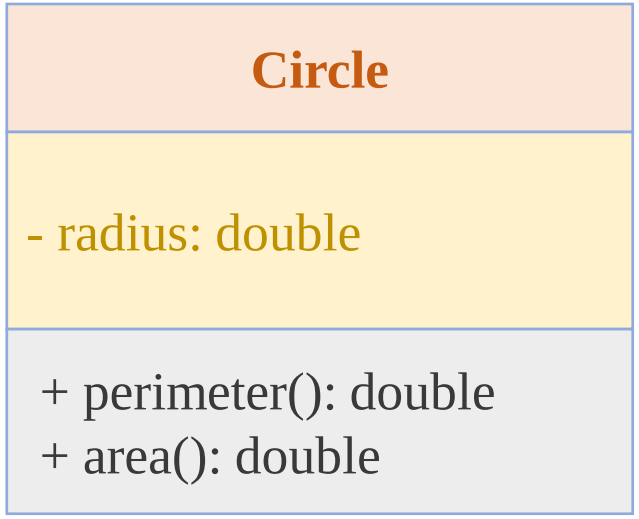


Example

❖ Inheritance recognition

Squares and circles are both examples of shapes. There are certain questions one can reasonably ask of both a circle and a square (such as, ‘what is the area?’ or ‘what is the perimeter?’) but some questions can be asked only of one or the other but not both (such as, ‘what is the length of a side?’ or ‘what is the radius?’)





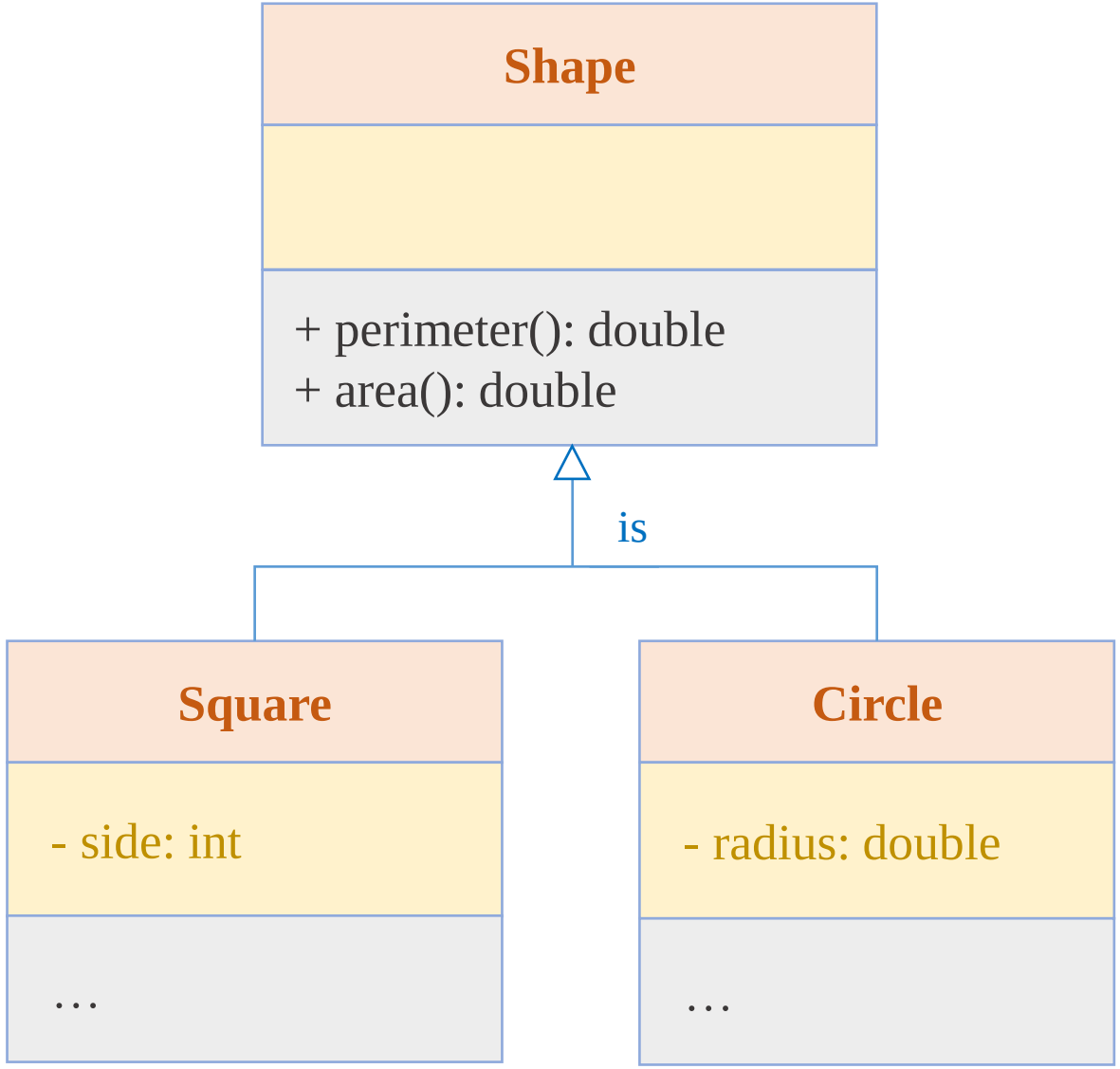
Example

❖ Inheritance recognition

Shape does not know how to compute its perimeter and area

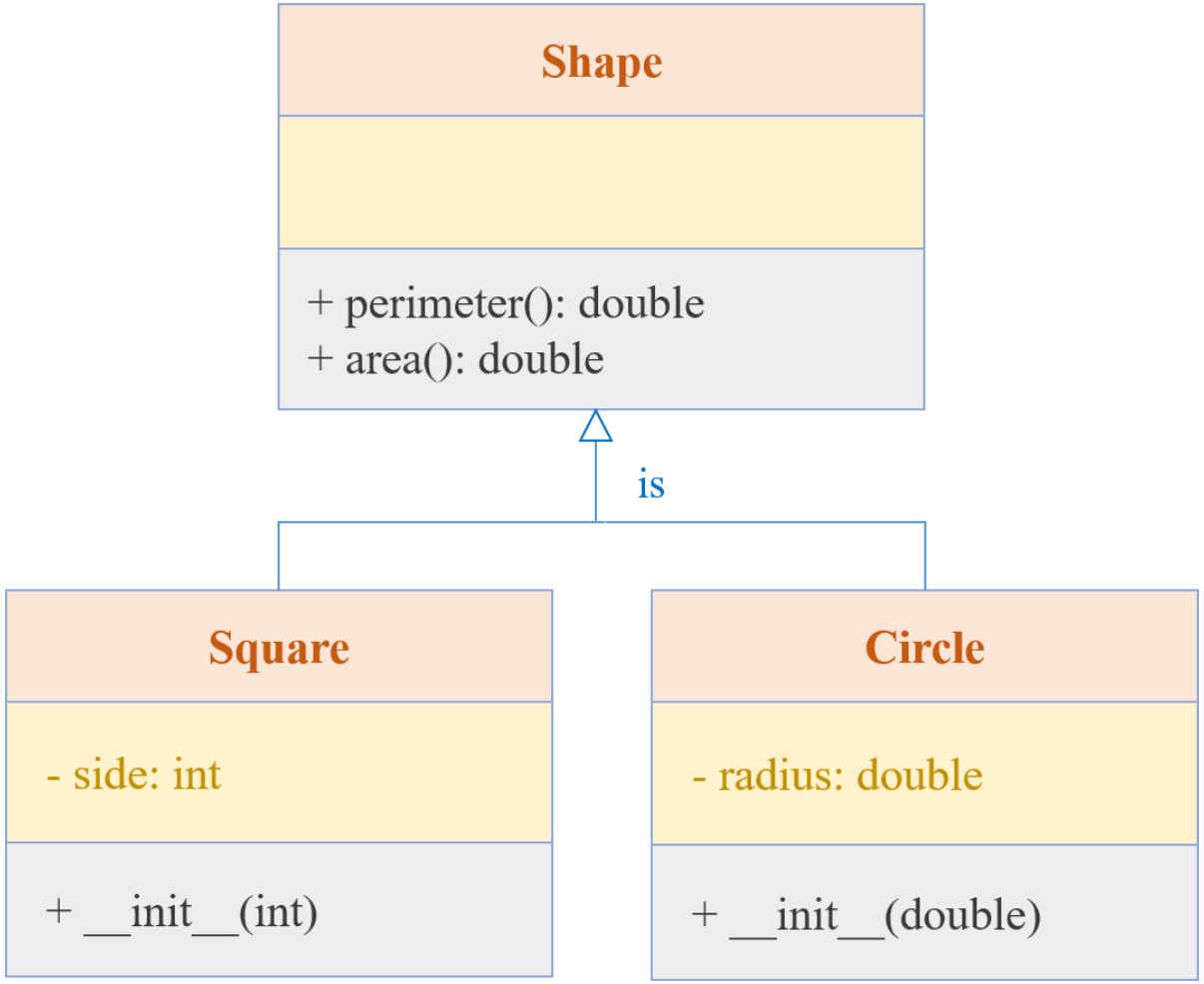
Use @abstractmethod to ask its child to implement them

Using pass in the abstract method



Example

❖ Inheritance recognition




```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def compute_area(self):
        pass
```

✓ 0.0s

```python
class Square(Shape):
    def __init__(self, side):
        self.__side = side

    def compute_area(self):
        return self.__side*self.__side
```

✓ 0.0s

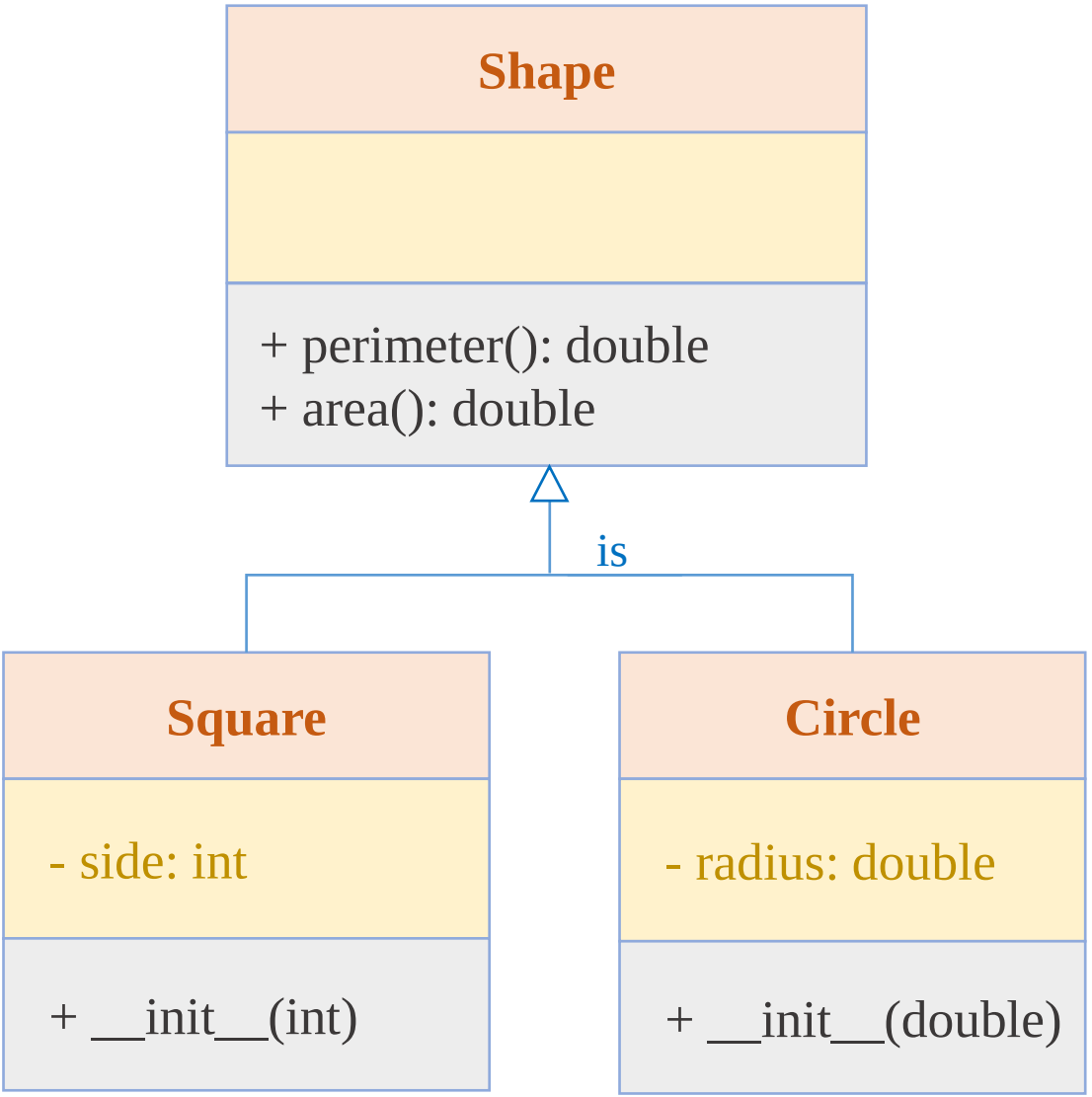
```python
square = Square(5)
print(square.compute_area())
```

✓ 0.0s

```
25
```

Example

❖ Inheritance recognition




```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def compute_area(self):
        pass
```

✓ 0.0s

```python
class Cicle(Shape):
    def __init__(self, radius):
        self.__radius = radius

# test
circle = Cicle(5)
```

⊗ 0.0s

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 6
      3         self.__radius = radius
      5 # test
----> 6 circle = Cicle(5)

TypeError: Can't instantiate abstract class Cicle with abstract method compute_area
```

Example

❖ Inheritance recognition

The Circle class **must** implement the compute_area() method.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def compute_area(self):
        pass
```

✓ 0.0s

```python
import math

class Cicle(Shape):
    def __init__(self, radius):
        self.__radius = radius

    def compute_area(self):
        return self.__radius*self.__radius*math.pi

# test
circle = Cicle(5)
print(circle.compute_area())
```

✓ 0.0s

```
78.53981633974483
```

```python
class Square(Shape):
    def __init__(self, side):
        self.__side = side

    def compute_area(self):
        return self.__side*self.__side
```

✓ 0.0s

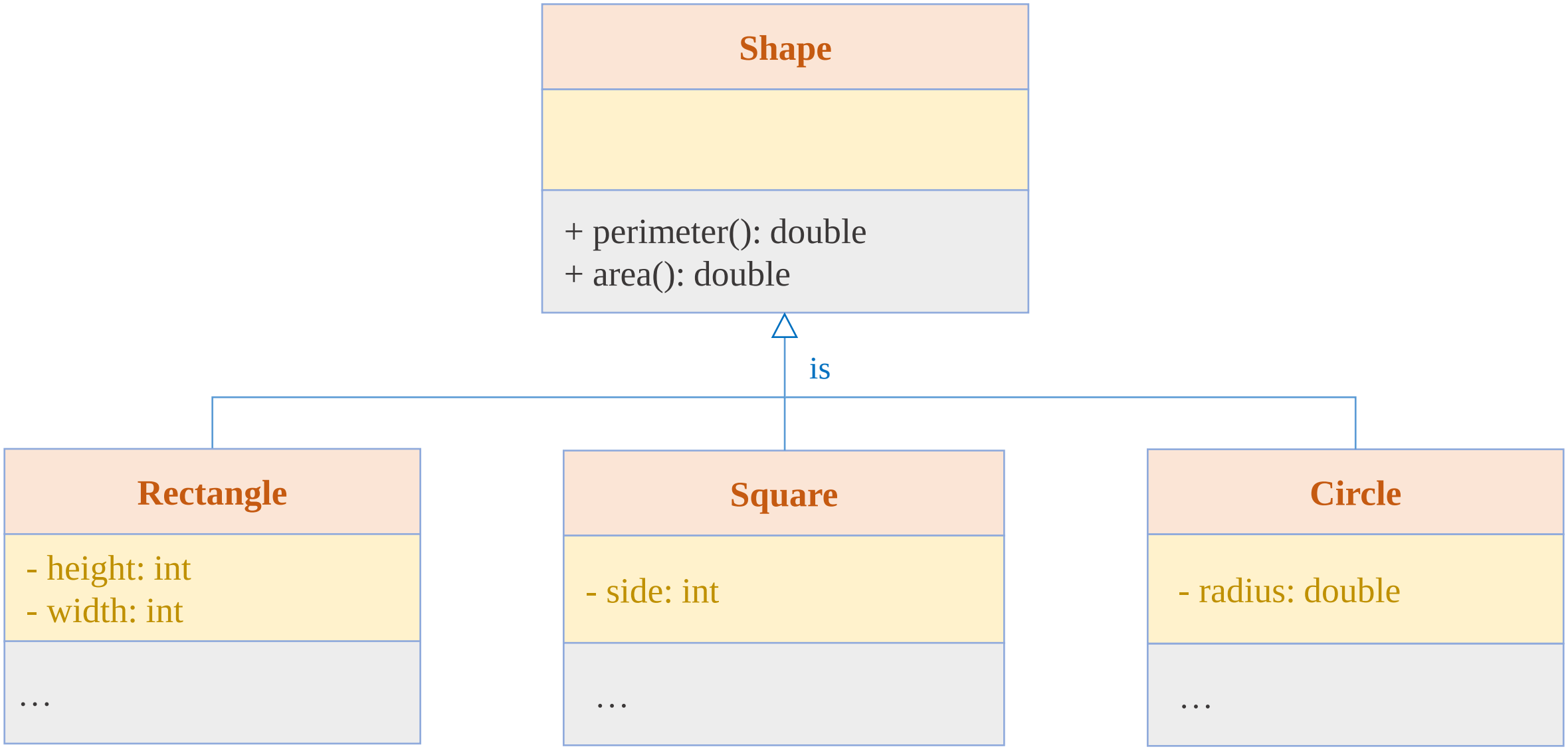
```python
square = Square(5)
print(square.compute_area())
```

✓ 0.0s

```
25
```

Example

❖ Open for extension



Outline

SECTION 1

Inheritance: To Reuse

SECTION 2

Inheritance: Overriding

SECTION 3

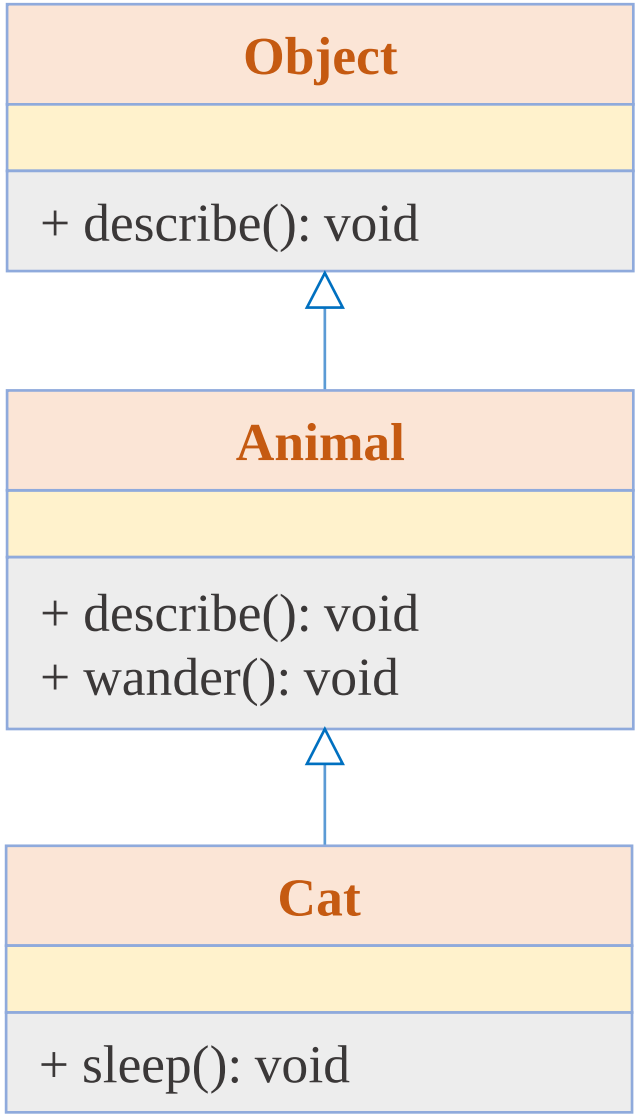
Inheritance: As a Template

SECTION 4

Multiple/Multilevel Inheritance

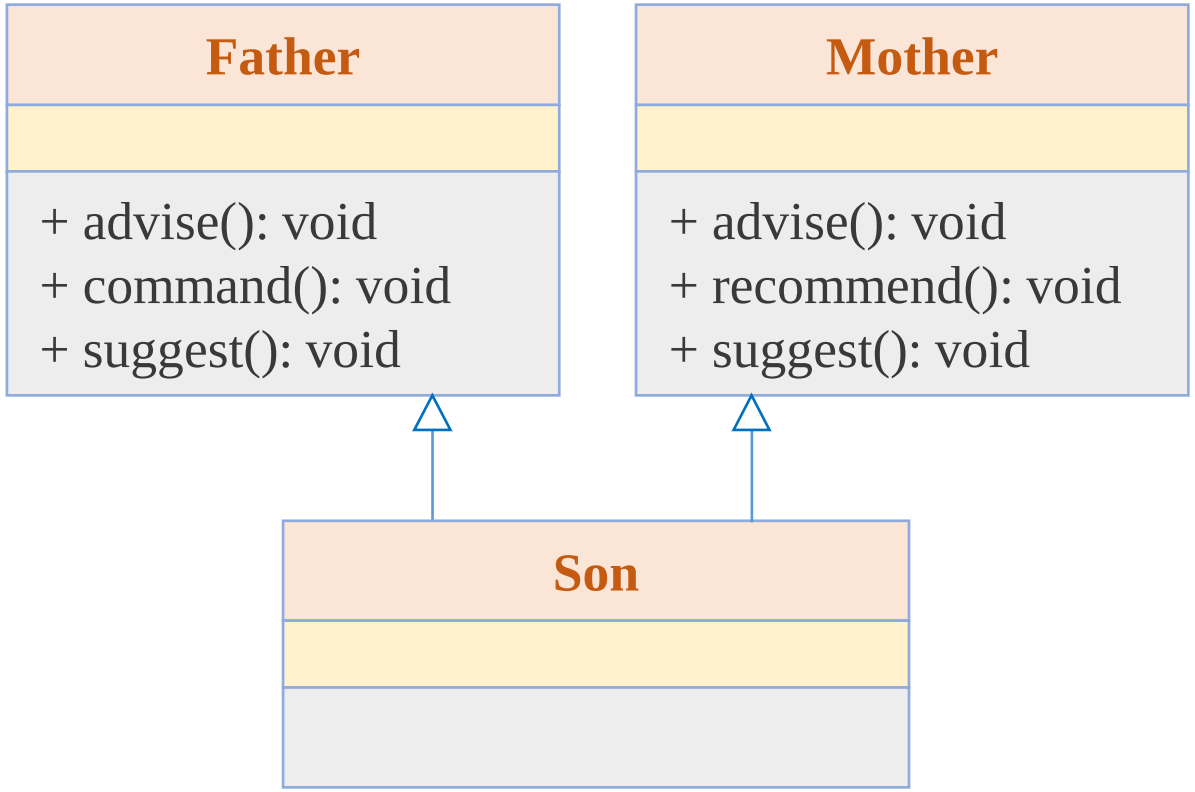
SECTION 5

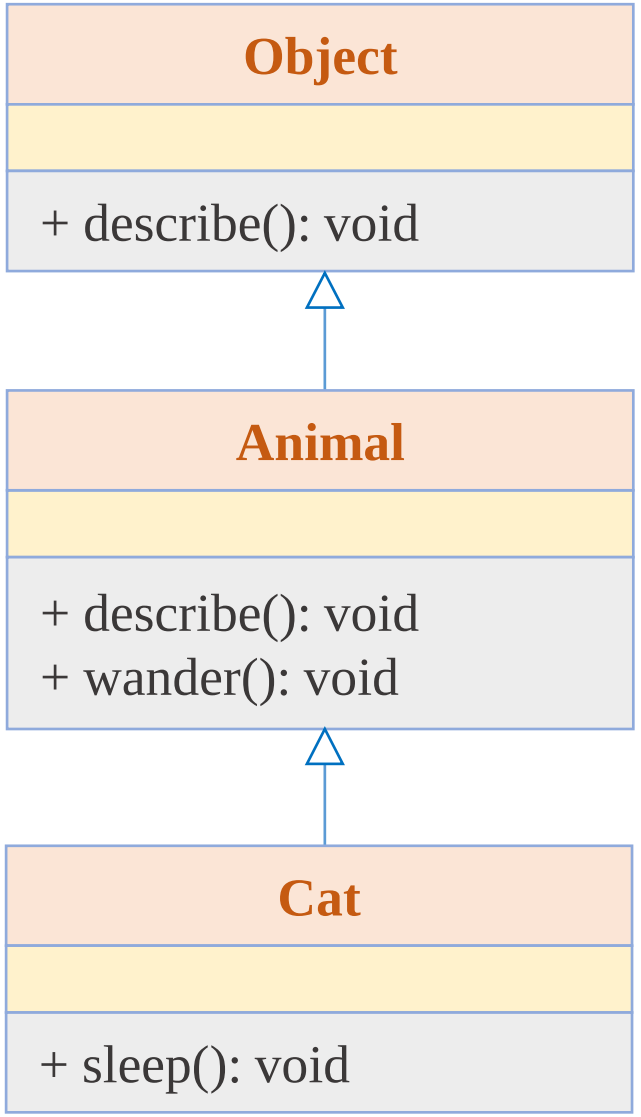
Custom Class in PyTorch



Inheritance

❖ Multiple and multi-level Inheritances

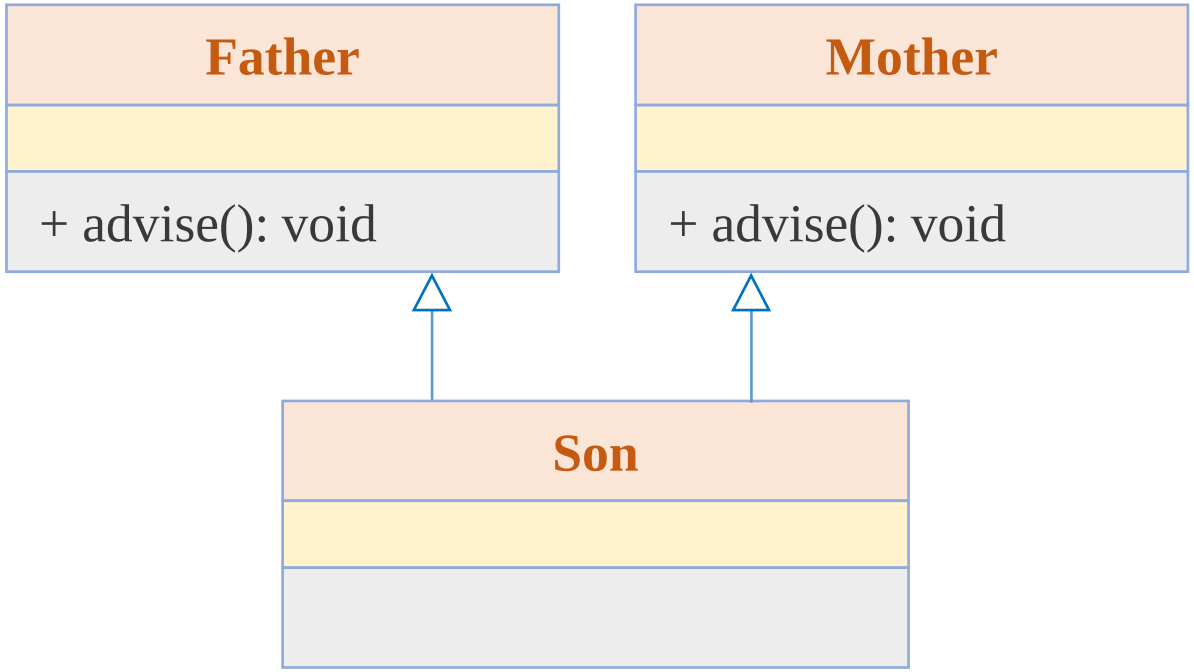




Inheritance

Multiple Inheritance

A class can be derived from more than one base class, using a comma-separated list.

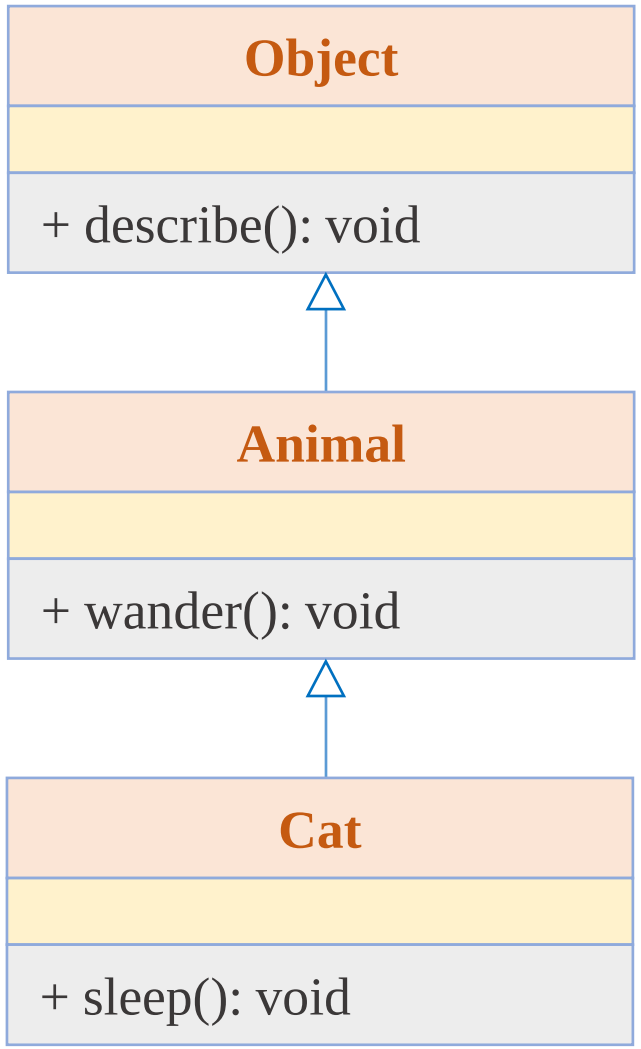



```python
class Father:
    def advise(self):
        print('Work hard!')

class Mother:
    def advise(self):
        print('Take a rest!')

class Child(Father, Mother):
    def __init__(self, name):
        self.__name = name

    def describe(self, condition):
        print(self.__name)
        if condition == True:
            Father.advise(self)
        else:
            Mother.advise(self)

# test
john = Child('John')
john.describe(False)
```

✓ 0.0s

```
John
Take a rest!
```

Multi-level Inheritance



Multilevel Inheritance

A class can be derived from one class, which is already derived from another class.

```python
class Species:
    def describe(self):
        print('I am living!')

class Animal(Species):
    def eat(self):
        print('I am eating!')

class Cat(Animal):
    def wander(self):
        print('I am wandering!')

# test
a_cat = Cat()
a_cat.describe()
a_cat.eat()
a_cat.wander()
```

✓ 0.0s

```
I am living!
I am eating!
I am wandering!
```

Multilevel Inheritance

A Design Pattern

Develop a system to manage birds for a shop.

Now, the shop want to manage only parrots. A parrot has color, and its activities include eating and flying.

Design a system that is extensible

Multilevel Inheritance

A Design Pattern

Multilevel Inheritance

A Design Pattern

```
# test
a_parrot = Parrot()
a_parrot.eat()
a_parrot.fly()

a_penguin = Penguin()
a_penguin.eat()
```

✓ 0.0s

```
I like eating ...
I can fly
I like eating ...
```

```python
from abc import ABC, abstractmethod

# Bird
class Bird(ABC):
    @abstractmethod
    def eat(self):
        pass

# FlyingBird
class FlyingBird(Bird):
    def fly(self):
        pass

# Parrot
class Parrot(FlyingBird):
    def eat(self):
        print('I like eating ...')
    def fly(self):
        print('I can fly')

# Penguin
class Penguin(Bird):
    def eat(self):
        print('I like eating ...')
```


QUIZ TIME

Outline

SECTION 1

Inheritance: To Reuse

SECTION 2

Inheritance: Overriding

SECTION 3

Inheritance: As a Template

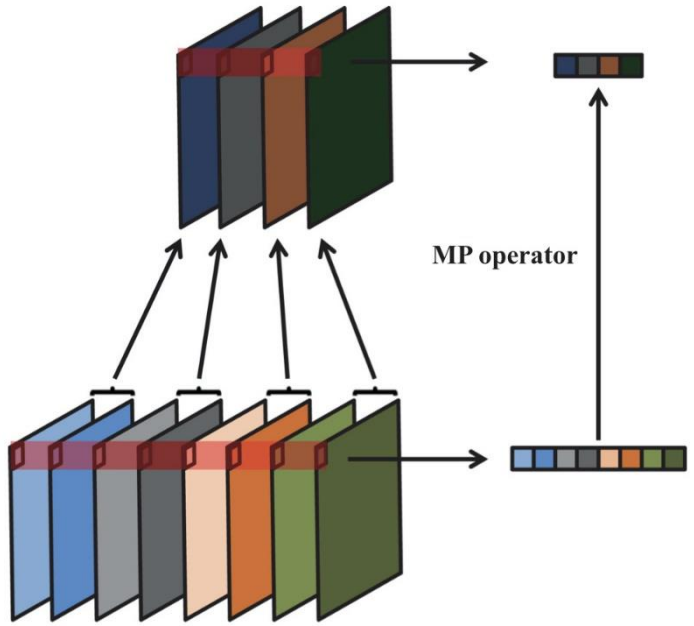
SECTION 4

Multiple/Multilevel Inheritance

SECTION 5

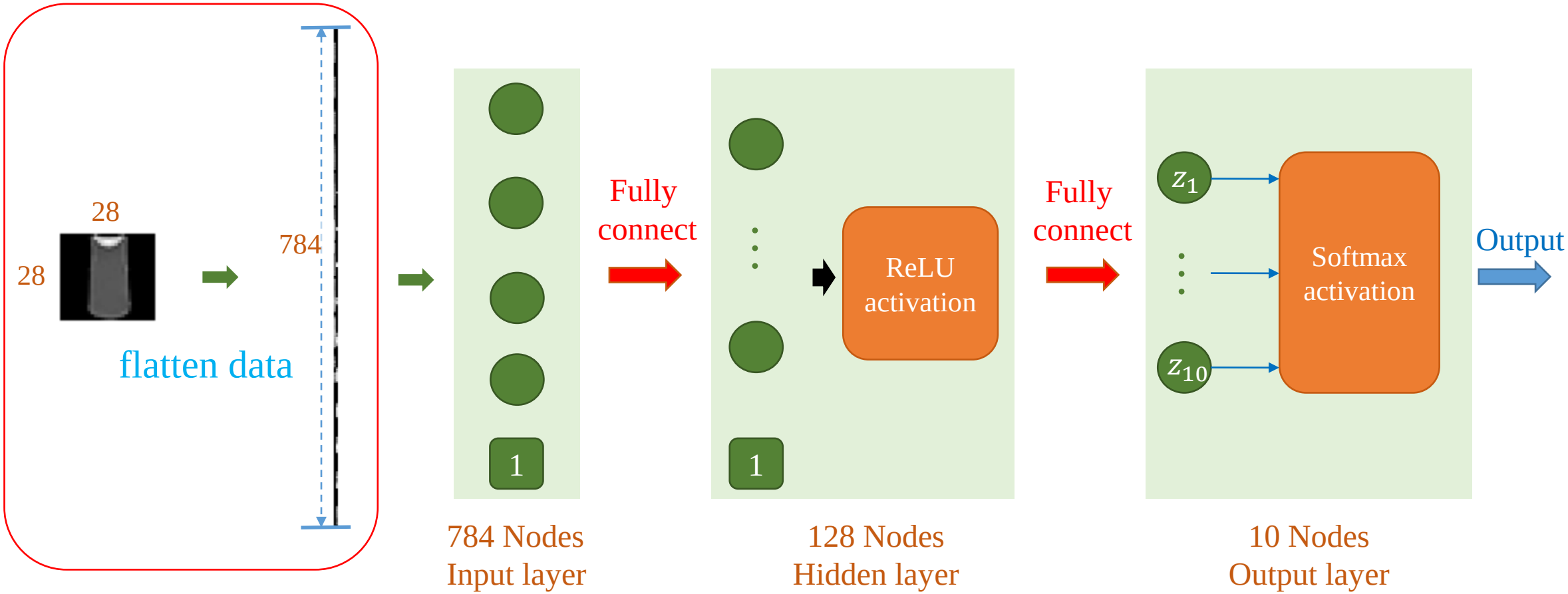
Custom Class in PyTorch

PyTorch



Custom Layer in PyTorch

❖ ReLU Layer



Custom Layer in PyTorch

❖ Custom ReLU

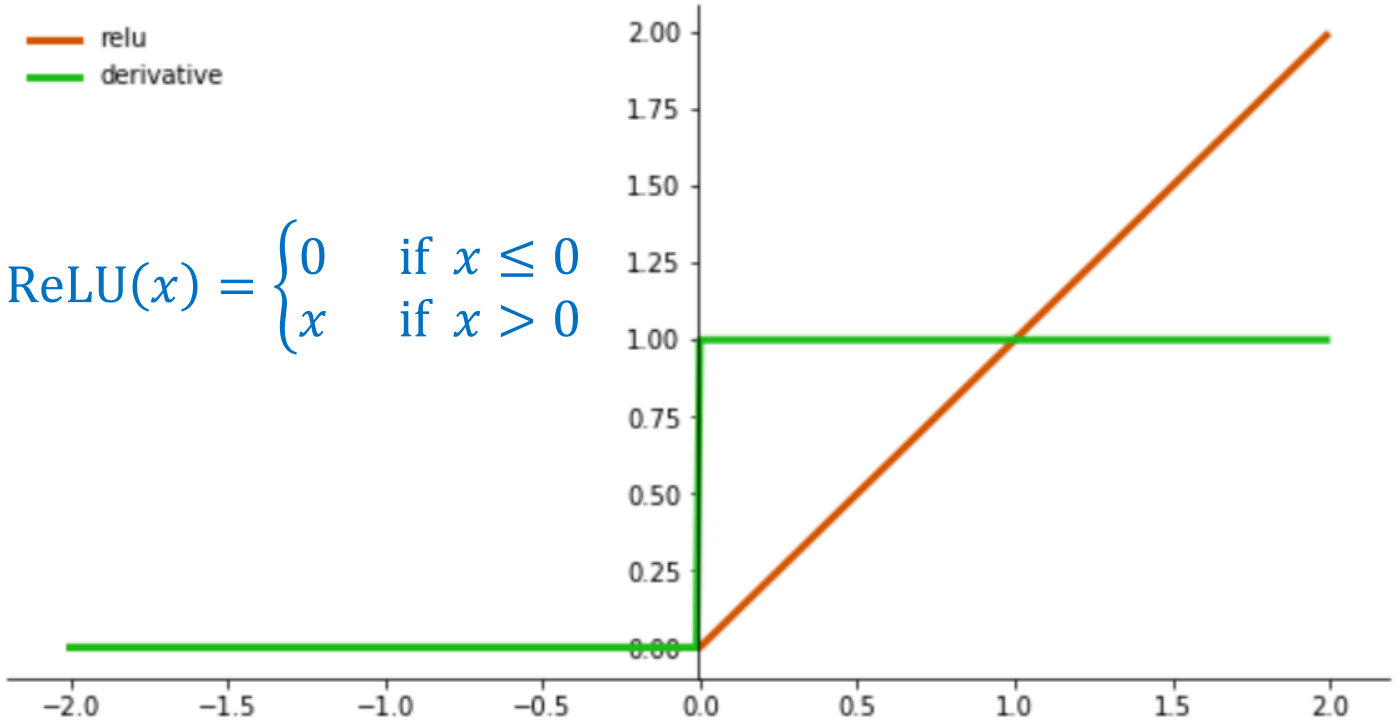
init method

Initialize values/variables necessary for a class

call method

Forward computation
max(0, x)

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$



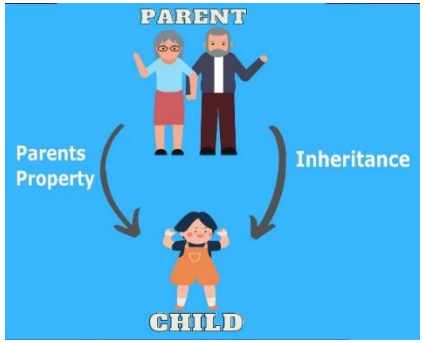

```
class MyReluActivation(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, x):
        zeros = torch.zeros_like(x)
        return torch.maximum(x, zeros)
```

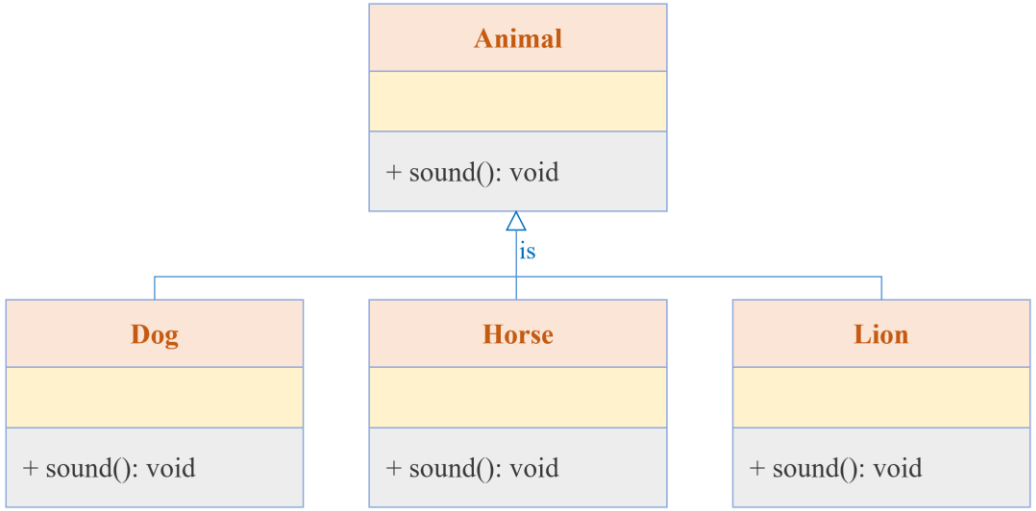
```
data = torch.Tensor([1, -2, 3])
my_relu = MyReluActivation()
output = my_relu(data)
print(output)
```

```
tensor([1., 0., 3.])
```

Summary

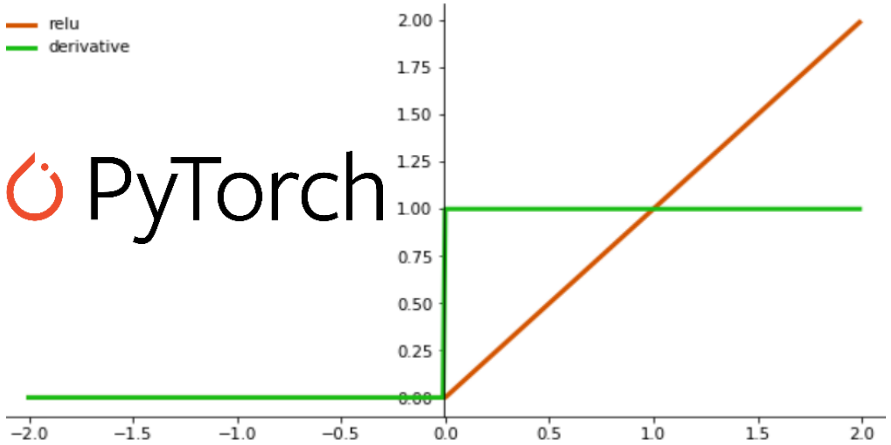
Inheritance





Applications

PyTorch



Library

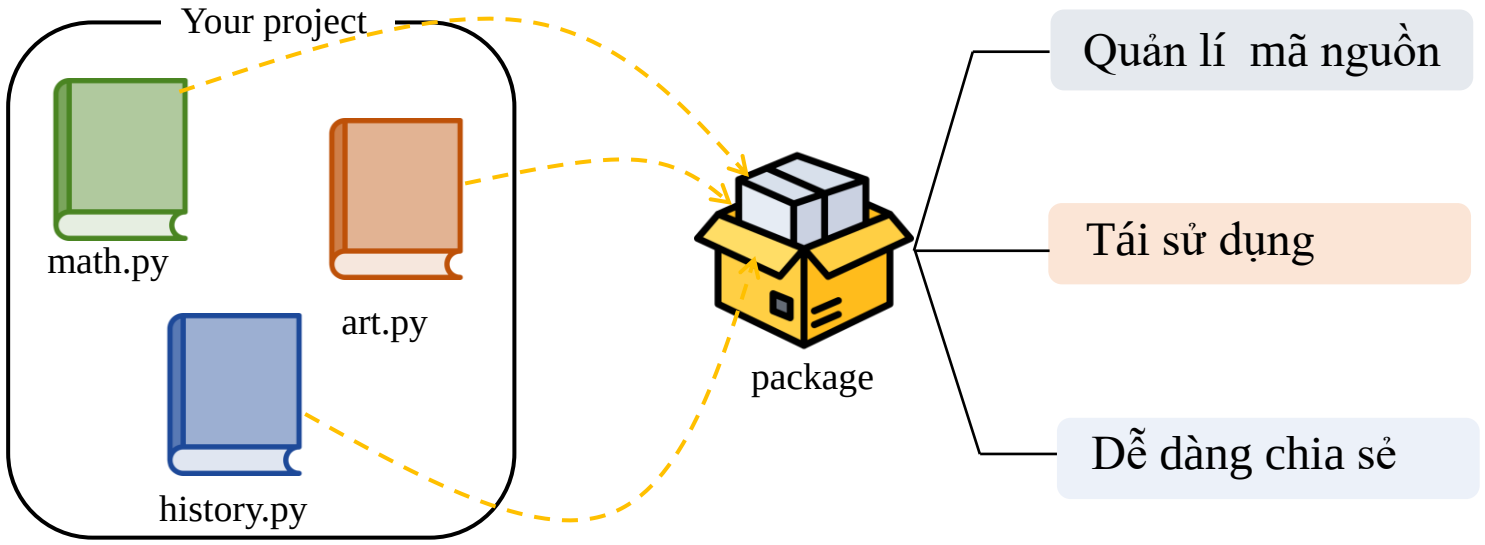
NumPy

python

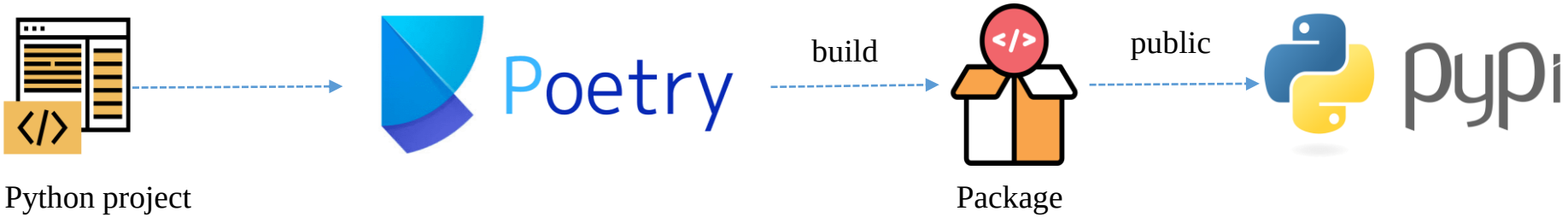
pandas

```
pip install numpy, pandas
```

Problem



Solution



1. Cài đặt Poetry
2. Thiết lập dự án
3. Xây dựng package
4. Public/share package

Install Poetry

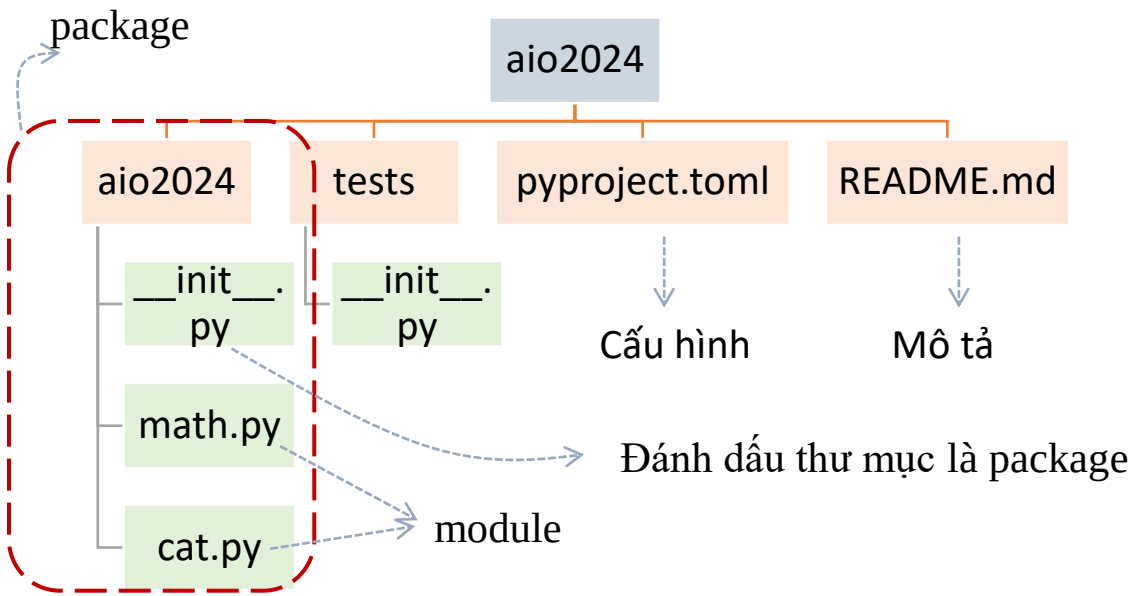
Step 1 `py-m pip install--user pipx`

Step 2 `pipx install poetry`

Step 3 `poetry --version`

Create a new poetry project

`poetry new aio2024`



pyproject.toml

```
[tool.poetry]
name = “aio2024"
version = "0.1.0 "
description = " "
authors = [“author name"]
readme = "README.md"
packages = [{include = “aiomath"}]
```

Tên dự án

Phiên bản

```
[tool.poetry.dependencies]
python = "^3.10 "
```

Thôn tin về các package, thư viện cần cho dự án sẽ tự động thêm ở đây

```
[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

math.py

```python
class MyMath:
    def __init__(self, value:int) -> None:
        self.value = value
    def factorial(self) -> int:
        if self.value == 0:
            return 1
        else:
            return self.value * MyMath(self.value - 1).factorial()
```

```python
n = 4
fact_n = MyMath(n).factorial()
print(f"Factorial of {n} is {fact_n}")
```

```
Output: Factorial of 4 is 24
```

Buid package

`poetry shell` — Kích hoạt môi trường

`poetry add name_package` — Cài đặt các gói phụ thuộc

`poetry install` — Cài đặt nhiều gói phụ thuộc thiết lập trong pyproject.toml

Thực hiện đóng gói

`poetry build`

Kết quả:

```
Building aio2024 (0.1.0)
  - Building sdist
  - Built aio2024-0.1.0.tar.gz
  - Building wheel
  - Built aio2024-0.1.0-py3-none-any.whl
```

publish package

1. Tạo tài khoản pypi

2. Tạo API tokens pypi

3. Thêm API vào Poetry

```
poetry config http-basic.foo  <username> <password>
```

```
poetry config pypi-token.pypi <my-token>
```

4. Public package

```
poetry publish
```

https://drive.google.com/drive/folders/19VvhWNQCtOlHJzYQ9kuHcwEGWvuiGfoW

using package

```
pip install aio2024-0.1.0-py3-none-any.whl
```

Hoặc cài từ pypi

```
pip install aio2024
```

```python
from aio2024 import math, cat

n = 5
fact_n = math.MyMath(n).factorial()
print(f"Factorial of {n} is {fact_n}")
new_cat = cat.Cat("Tom").discribe()
```

```
Output: Factorial 5 is 120
        Tom
```