AI VIET NAM
@aivietnam.edu.vn

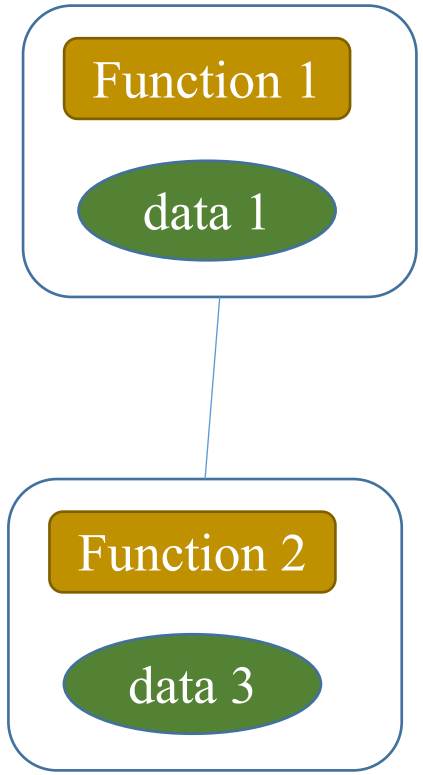
Object-Oriented Programming

(Objects and Classes)

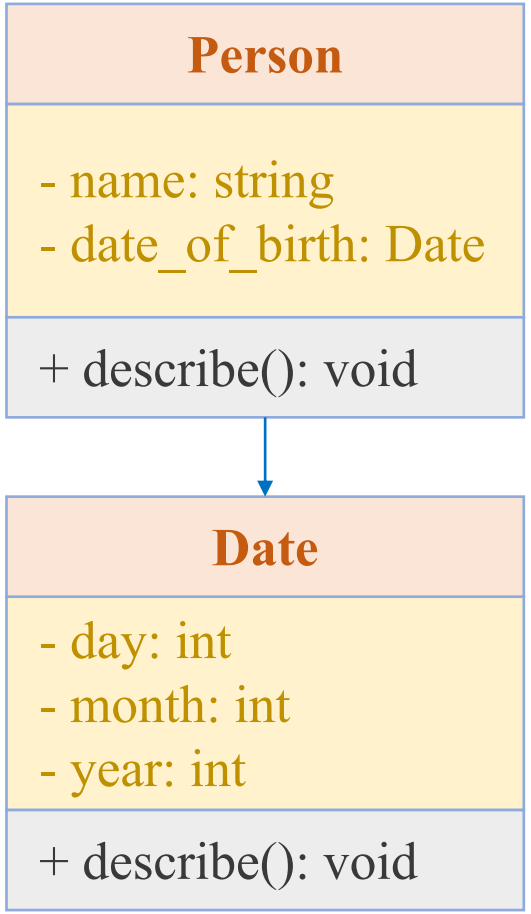
Quang-Vinh Dinh
PhD in Computer Science

OOP Objectives

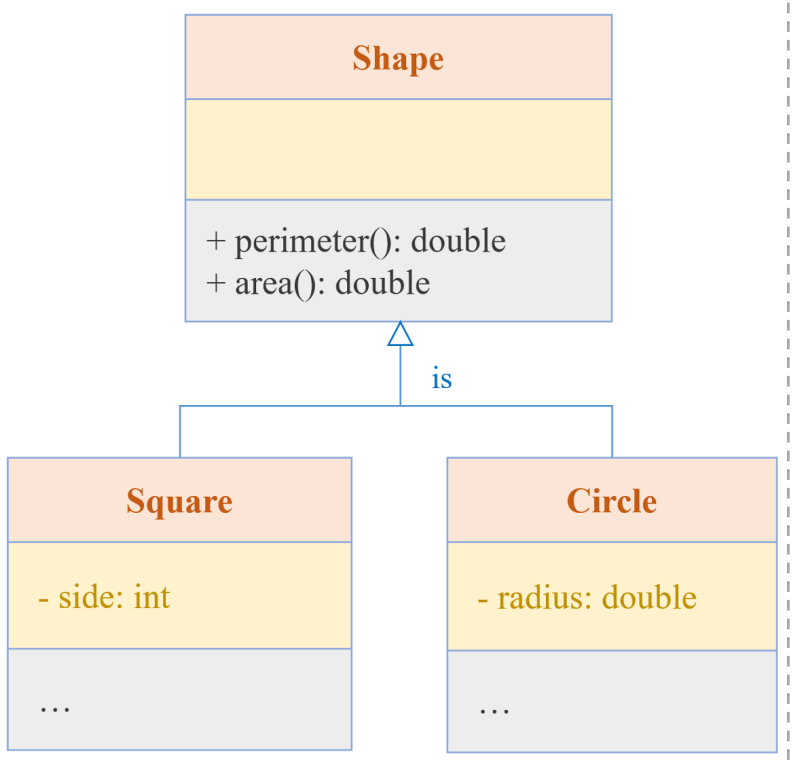
Class (Encapsulation)



Delegation



Inheritance



Outline

SECTION 1

Introduction to OOP

SECTION 2

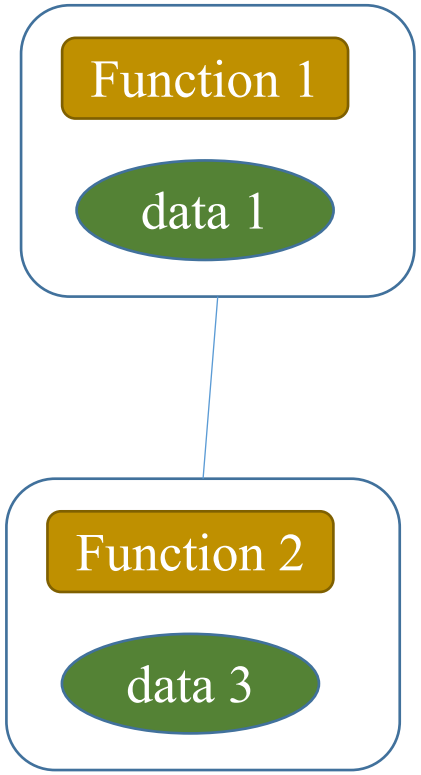
Objects and Classes

SECTION 3

Delegation

SECTION 4

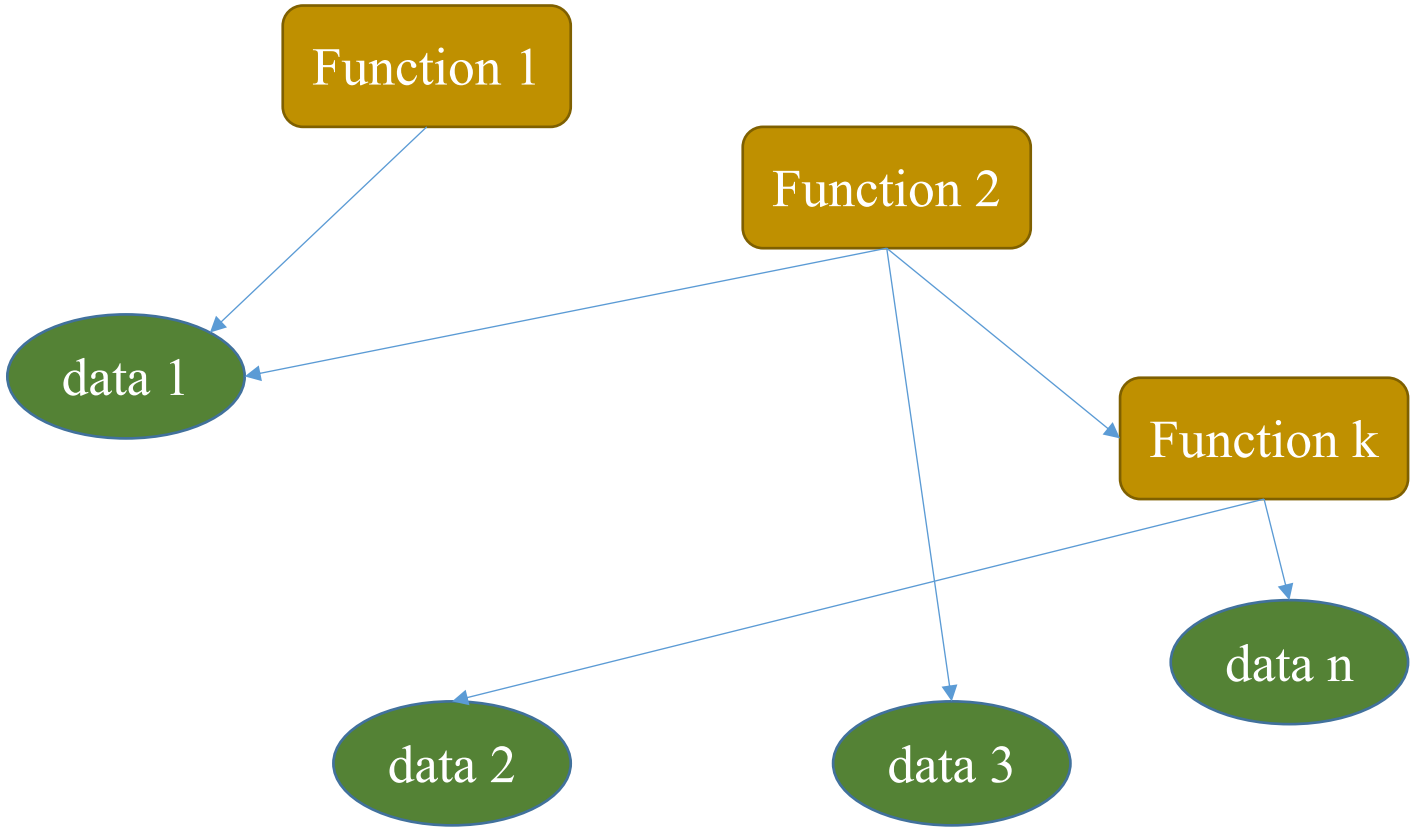
Inheritance

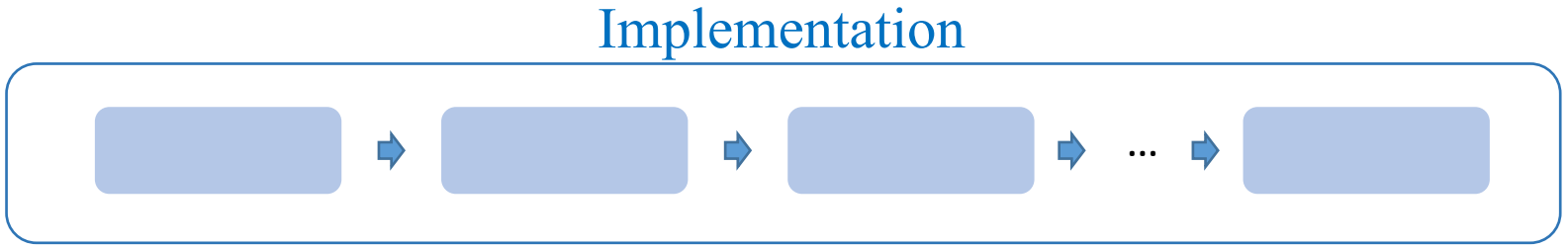


OOP Introduction

❖ Procedural programming

- Top-down approach
- Security
- Maintenance
- Reusability
- Implementation





OOP Introduction

❖ OOP programming

- Bottom-up approach
- Security
- Maintenance
- Reusability
- Implementation

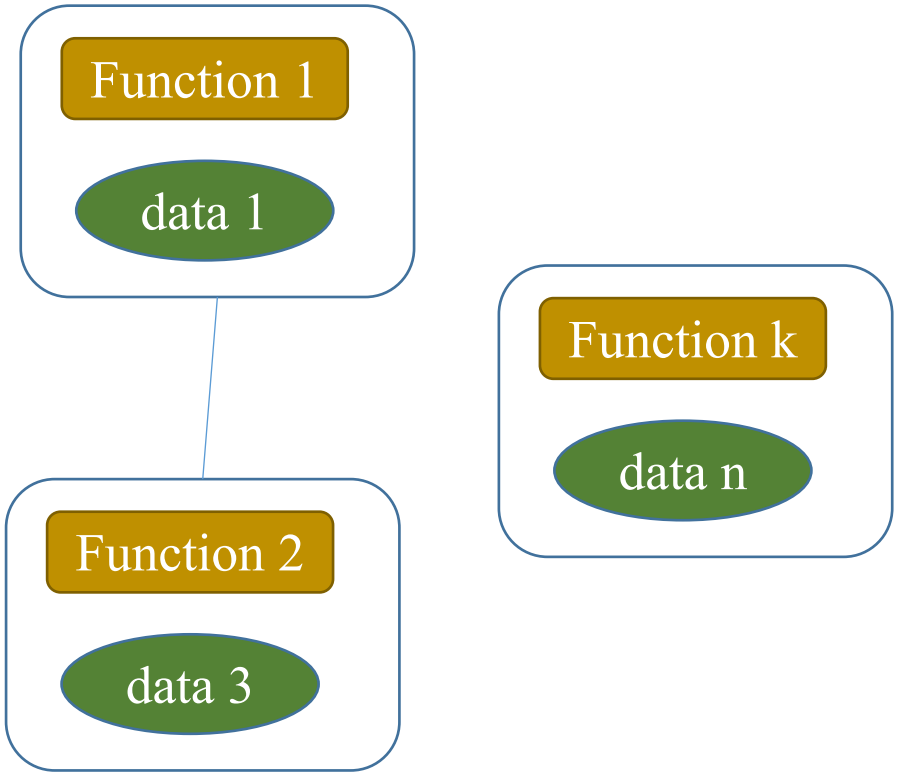
Access modifiers

Inheritance

Objects/Classes

Encapsulation

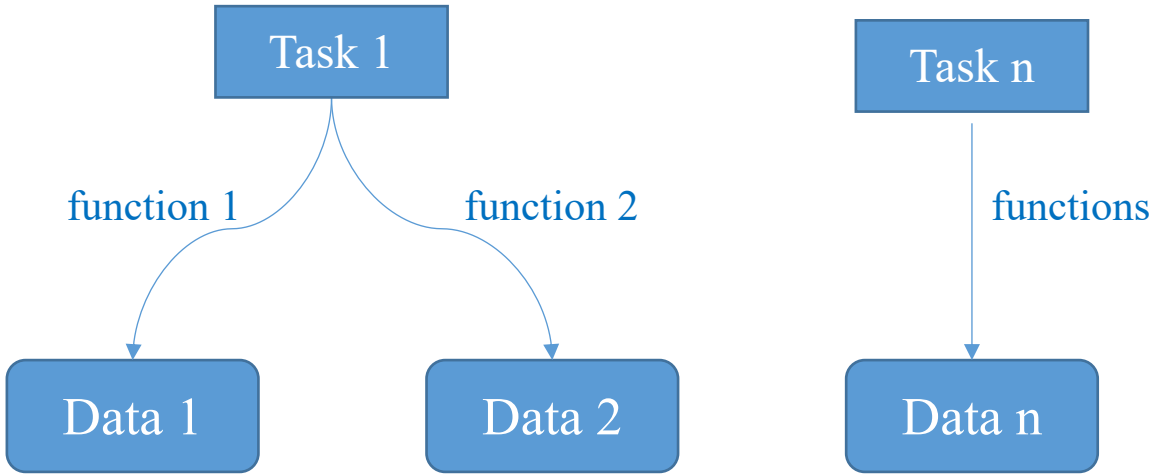
Polymorphism



Introduction

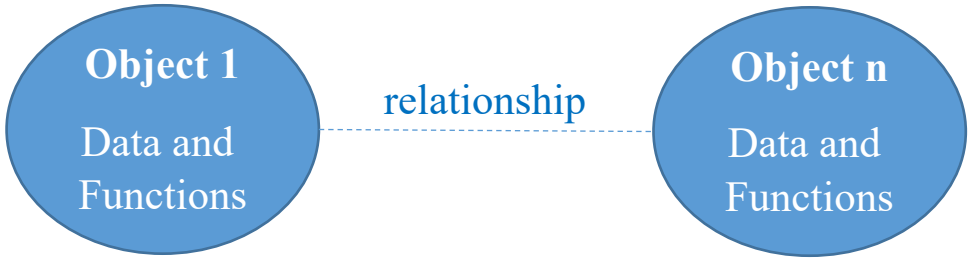
❖ What is OOP?

Writing functions that perform operations on the data



Procedural programming

Creating objects that contain both data and functions



Object-oriented programming

Introduction


```python
# code segment 1
alpha = 0.01
def function1(value):
    return max(value, value*alpha)

# test
print(function1(5))
print(function1(-3))

# code segment 2
def function2(name):
    return 'Hi ' + name

# test
data3 = 'John'
print(function2(data3))
```

```
5
-0.03
Hi John
```

```python
# code segment 1
alpha = 0.01
def function1(value):
    return max(value, value*alpha)

# test
print(function1(5))
print(function1(-3))

# code segment 2
def function2(name):
    return 'Hi ' + name + str(alpha)

# test
data3 = 'John'
print(function2(data3))
```

```
5
-0.03
Hi John0.01
```

❖ Example

```python
class LeakyReLU:
    def __init__(self, alpha):
        self.alpha = alpha

    def __call__(self, value):
        return max(value,
                   value*self.alpha)

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

    def greet(self):
        return 'Hi ' + self.name
```

Introduction

❖ Classes and Objects

A class is a template for objects, and an object is an instance of a class.

Fruit

Strawberry
Apple
Banana

Introduction

❖ Classes and Objects

A class is a template for objects, and an object is an instance of a class.

Animal

Cat
Deer
Tiger

Introduction

❖ Classes and Objects

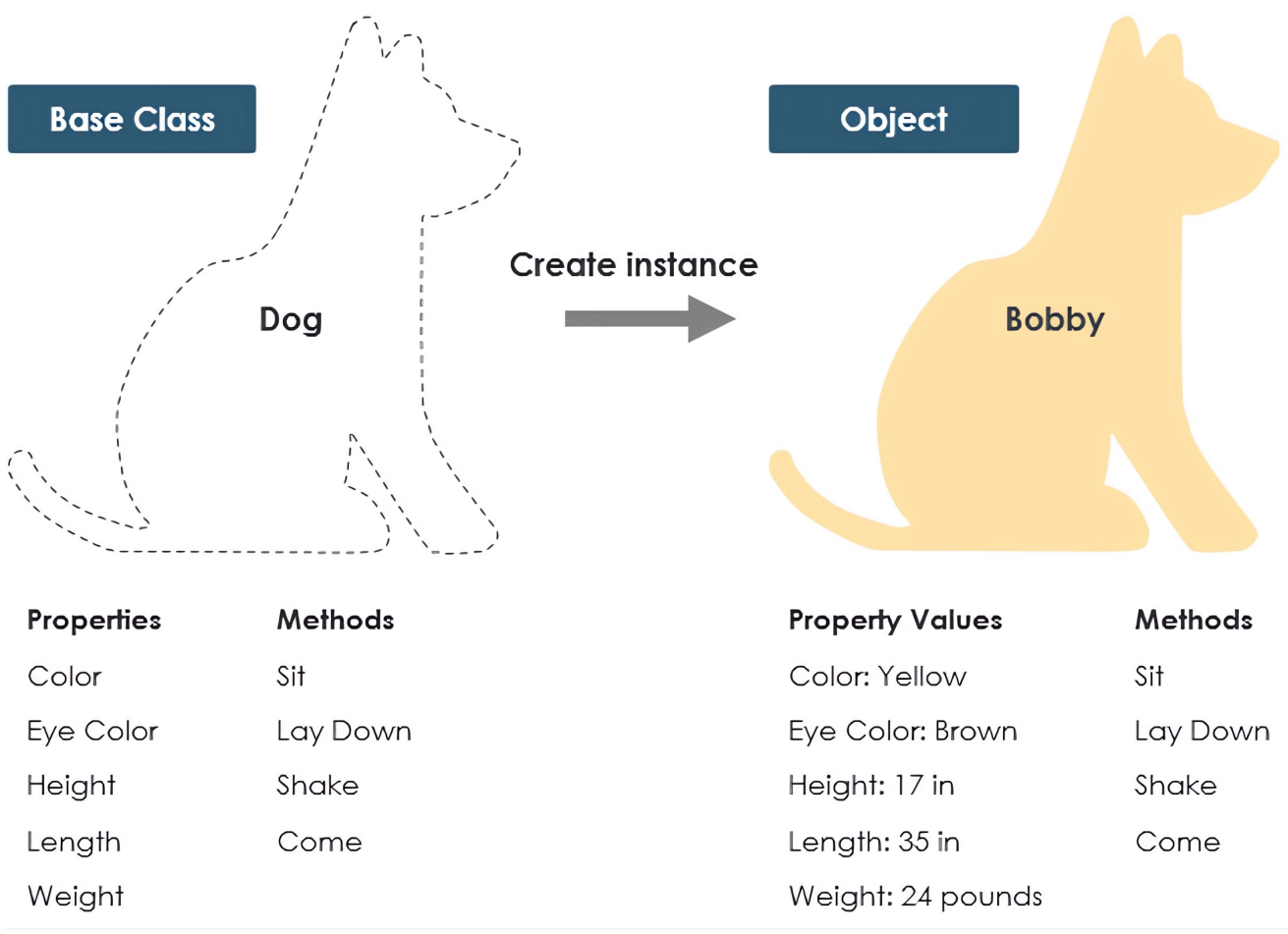
A class is a template for objects, and an object is an instance of a class.

Cat

Japanese Bobtail
Scottish Fold
Calico

Introduction

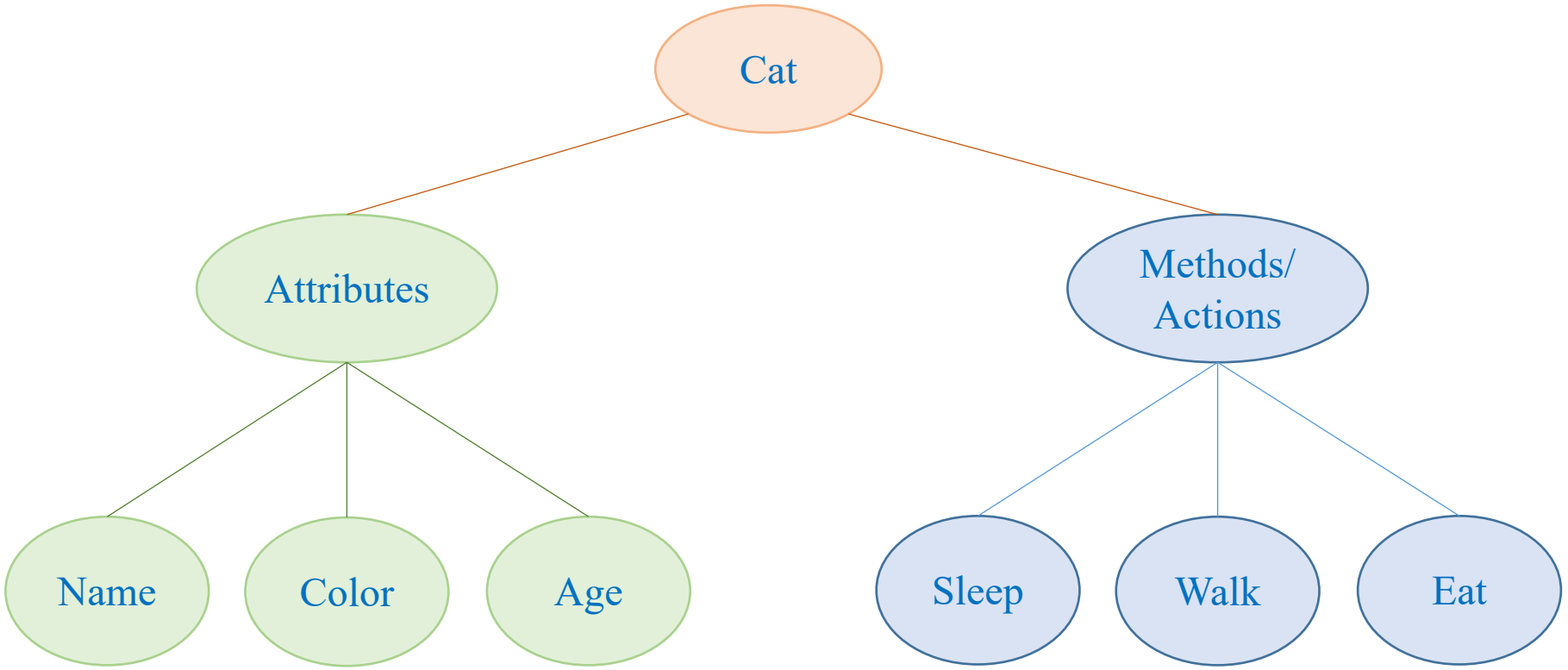
❖ Classes and Objects



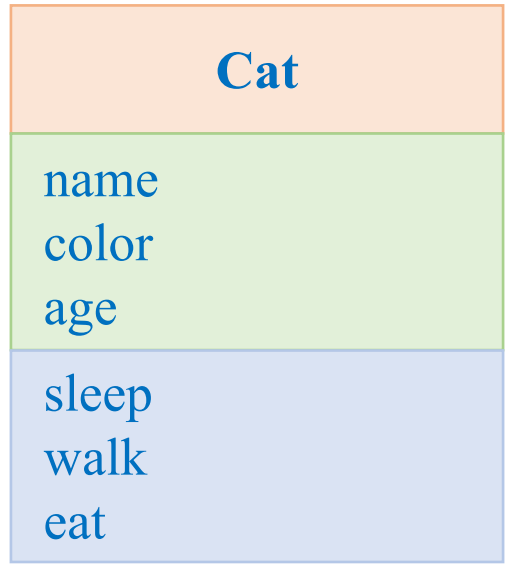
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

Classes and Objects

❖ Abstract view



Class Diagram



Class Diagram

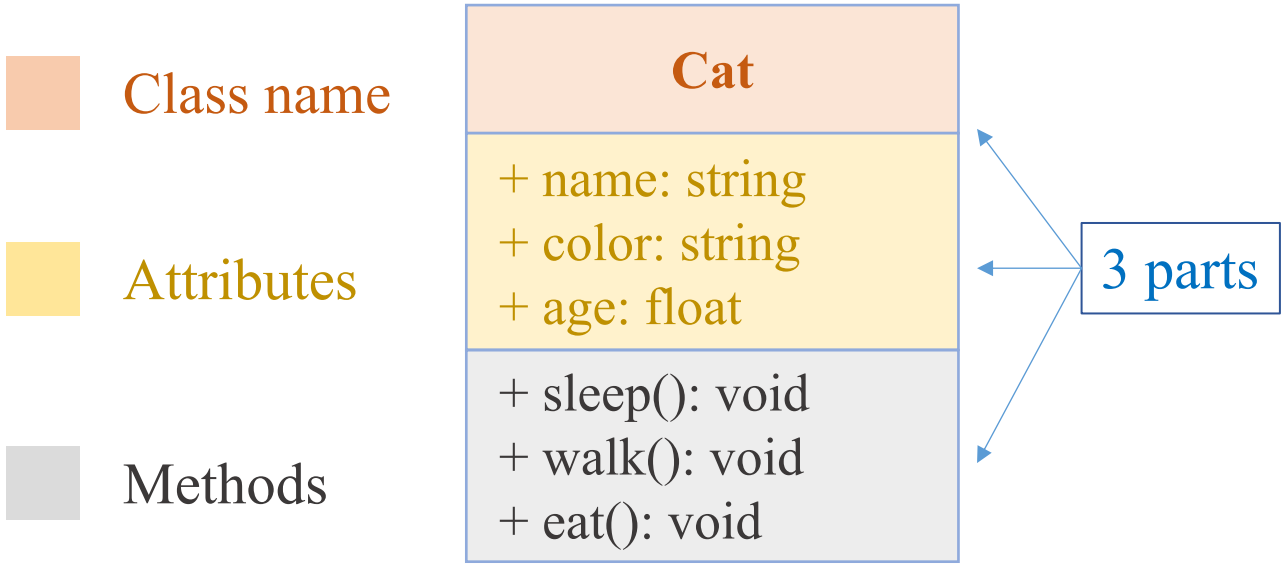
❖ Describe the structure of a system

Classes
their attributes
operations (methods)
relationships among objects

Access modifiers
- private
+ public

A cat includes a name, a color, and an age. The daily activities of the cat consists of sleeping, walking, and eating.

Draw a class diagram for the above description. All the attributes and methods are publicly accessed.



Outline

SECTION 1

Introduction to OOP

SECTION 2

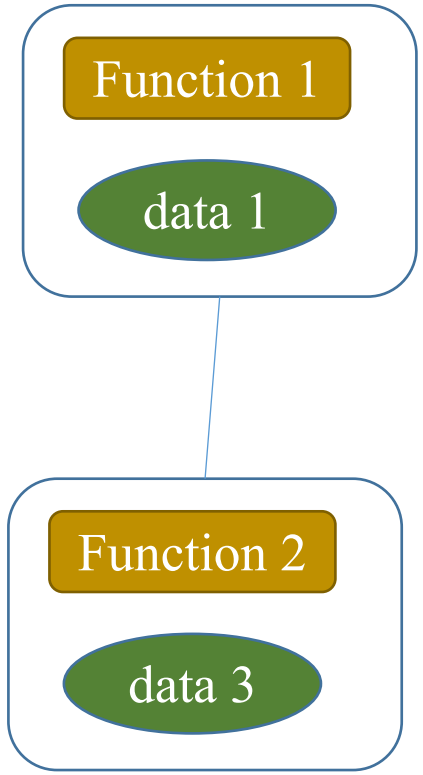
Objects and Classes

SECTION 3

Delegation

SECTION 4

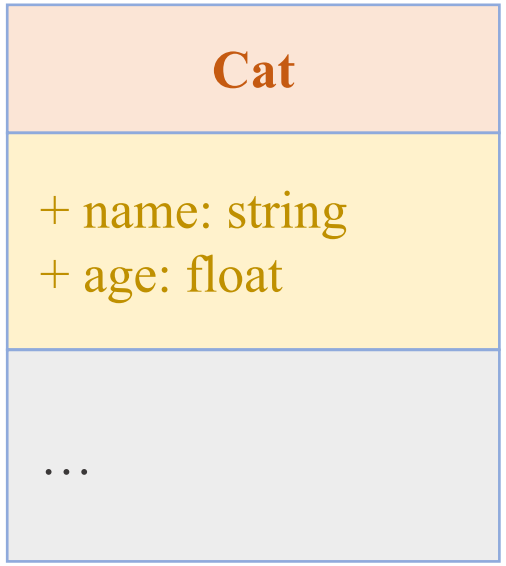
Inheritance



Classes and Objects

❖ Implementation using Python

A cat has a name and an age.
By default, a cat is named 'Calico' and is 0.8 years old.



class name

keyword

attribute

```
# create a class
class Cat:
    name = 'Calico'
    age = 0.8

# test: create an object
cat = Cat()
print(cat.name)
print(cat.age)
```

```
Calico
0.8
```

```
type(cat)
```

```
__main__.Cat
```

We have a new data type

```
cat = Cat()
```

variable — create an object

To access an attribute

```
cat.name
```

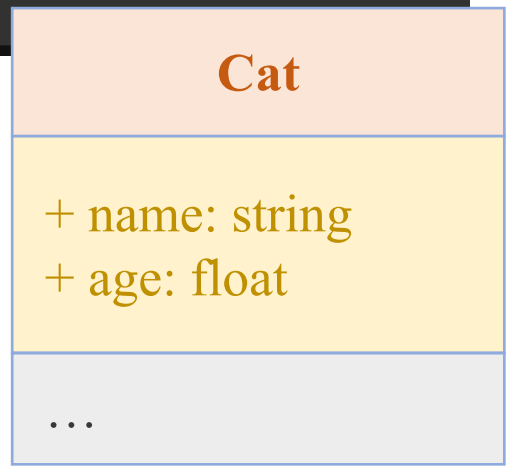
variable — attribute name

Classes and Objects

❖ Implementation using Python

```
# create a class
class Cat:
    name = 'unknown'
    age = -1

# test: create an object
cat = Cat()
print(cat.name)
print(cat.age)
```

```
unknown
-1
```



A cat has a name and an age.
By default, a cat is named 'Calico' and is 0.8 years old.

A class can be considered as a new space.
We can define functions in a class.

Functions insides classes are call **methods**

```
# create a class
class Cat:
    name = 'unknown'
    age = -1

    def set_init_values(input_name, input_age):
        name = input_name
        age = input_age

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
print(cat.name)
print(cat.age)
```

```
---------------------------------------------------------
TypeError                                 Traceback
```

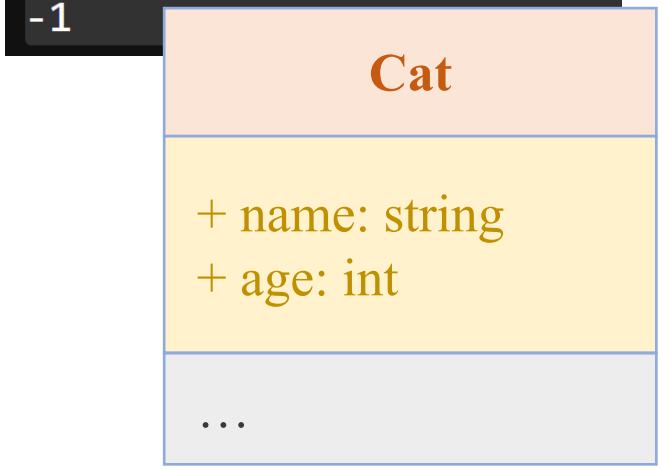
Not that easy!
Let's find out!

Classes and Objects

❖ A step inside classes in Python

```python
# create a class
class Cat:
    name = 'unknown'
    age = -1

# test: create an object
cat = Cat()
print(cat.name)
print(cat.age)
```

```
unknown
-1
```

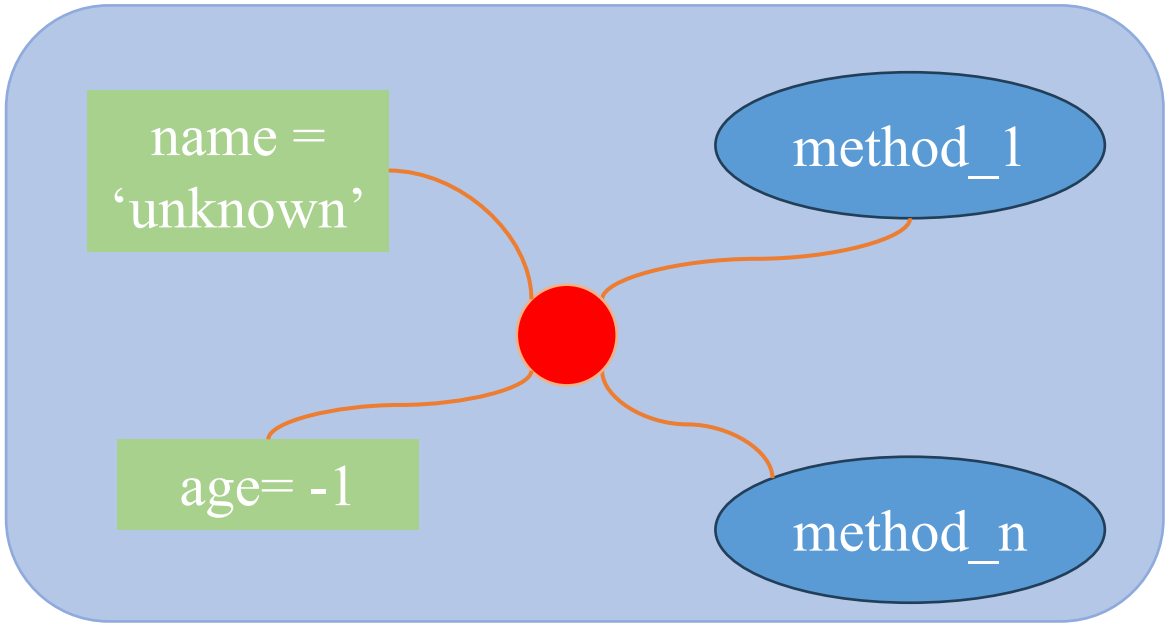


class

attributes

methods

self



To access attributes and methods inside a class

Using self.name
self.age
self.method_1(…)
self.method_n(…)

self is always the first parameter of a method

Classes and Objects

A cat has a name and an age.
By default, a cat is named 'Calico' and is 0.8 years old.

❖ Back to our problem

Cat
+ name: string + age: float
…

Using self.name
self.age
self.method_1(…)
self.method_n(…)

self is always the first parameter of a method

How to improve more?

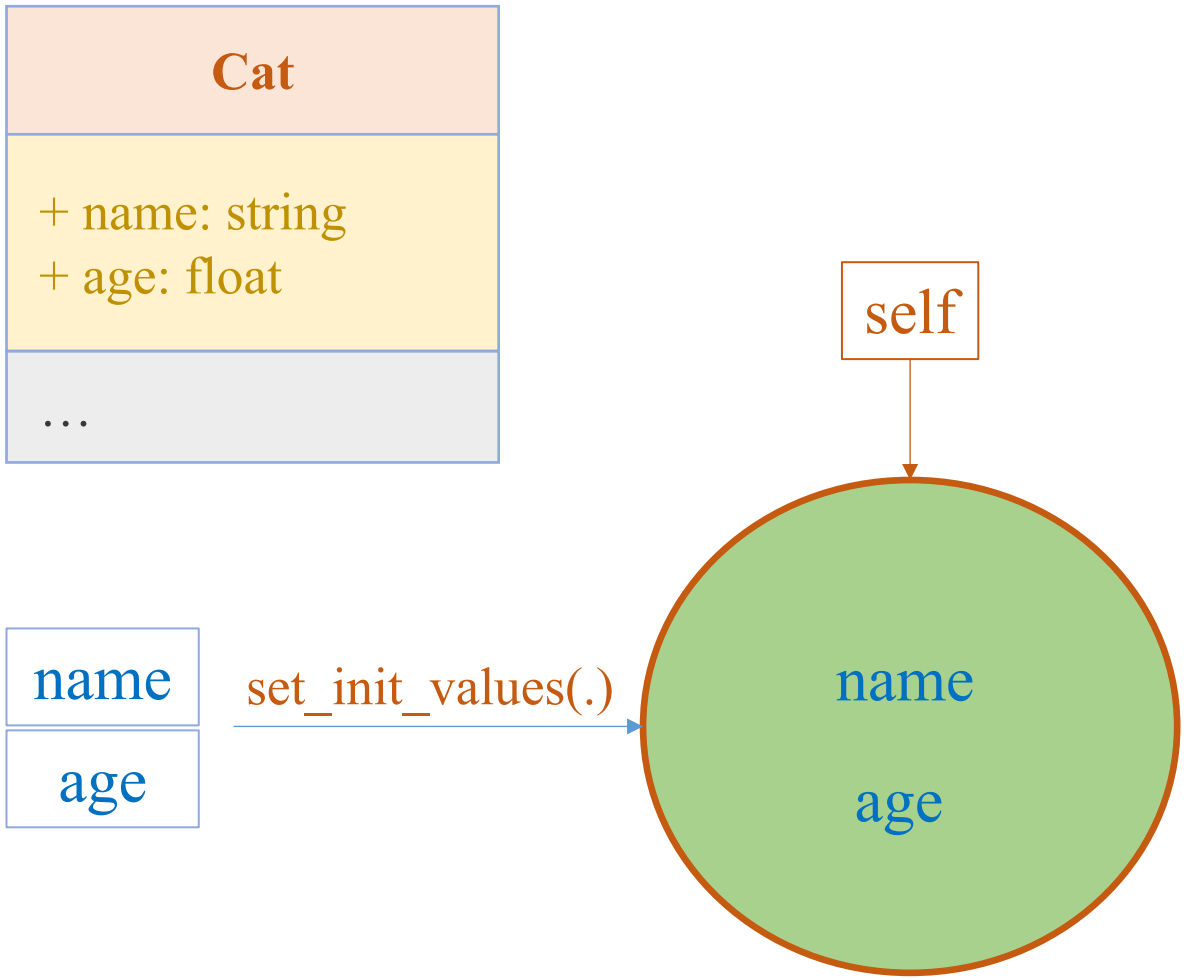
```
# create a class
class Cat:
    name = 'unknown'
    age = -1

    def set_init_values(self, input_name, input_age):
        self.name = input_name
        self.age = input_age

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
print(cat.name)
print(cat.age)
```

```
Calico
0.8
```

Classes and Objects

A cat has a name and an age.
By default, a cat is named 'Calico' and is 0.8 years old.

❖ Naming efficiently



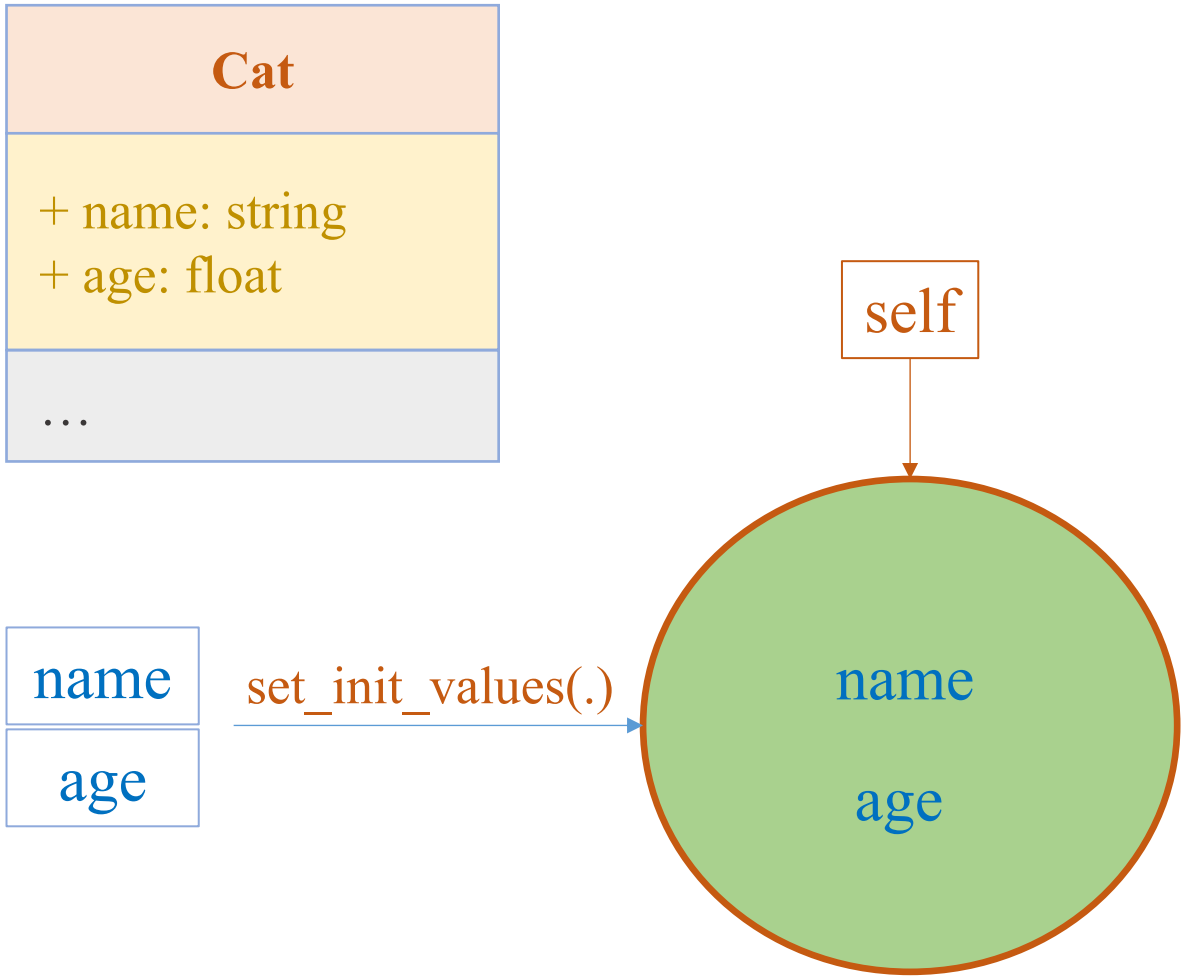

```python
# create a class
class Cat:
    name = 'unknown'
    age = -1

    def set_init_values(self, name, age):
        self.name = name
        self.age = age

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
print(cat.name)
print(cat.age)
```

```
Calico
0.8
```

Classes and Objects

A cat has a name and an age.
By default, a cat is named 'Calico' and is 0.8 years old.

❖ **We can create attributes with self**




```python
# create a class
class Cat:
    def set_init_values(self, name, age):
        self.name = name
        self.age = age

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
print(cat.name)
print(cat.age)
```

```
Calico
0.8
```

Classes and Objects

❖ Some more methods

Cat
+ name: string + age: float
+ set_init_values(name, age) + increase_age(age) + describe()

name = ‘Calico’

age= 0.8

name = ‘Calico’

age= 1.0

Observation?

```python
# create a class
class Cat:
    def set_init_values(self, name, age):
        self.name = name
        self.age = age

    def increase_age(self, age):
        self.age = self.age + age

    def describe(self):
        print(f'Name: {self.name} - Age: {self.age}')

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
cat.describe()

cat.increase_age(0.2)
cat.describe()
```

```
Name: Calico - Age: 0.8
Name: Calico - Age: 1.0
```

Make it simpler


```python
# create a class
class Cat:
    def set_init_values(self, name, age):
        self.name = name
        self.age = age

    def increase_age(self, age):
        self.age = self.age + age

    def describe(self):
        print(f'Name: {self.name} - Age: {self.age}')

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
cat.describe()

cat.increase_age(0.2)
cat.describe()
```

```
Name: Calico - Age: 0.8
Name: Calico - Age: 1.0
```

```python
# create a class
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def increase_age(self, age):
        self.age = self.age + age

    def describe(self):
        print(f'Name: {self.name} - Age: {self.age}')

# test: create an object
cat = Cat('Calico', 0.8)
cat.describe()

cat.increase_age(0.2)
cat.describe()
```

```
Name: Calico - Age: 0.8
Name: Calico - Age: 1.0
```

Make it simpler


```python
# create a class
class Cat:
    def set_init_values(self, name, age):
        self.name = name
        self.age = age

    def increase_age(self, age):
        self.age = self.age + age

    def describe(self):
        print(f'Name: {self.name} - Age: {self.age}')

# test: create an object
cat = Cat()
cat.set_init_values('Calico', 0.8)
cat.describe()

cat.increase_age(0.2)
cat.describe()
```

```
Name: Calico - Age: 0.8
Name: Calico - Age: 1.0
```

```python
# create a class
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def increase_age(self, age):
        self.age = self.age + age

    def __call__(self):
        print(f'Name: {self.name} - Age: {self.age}')

# test: create an object
cat = Cat('Calico', 0.8)
cat()

cat.increase_age(0.2)
cat()
```

```
Name: Calico - Age: 0.8
Name: Calico - Age: 1.0
```

To sum up

❖ Up to this point

The __init__() function is called automatically every time the class is being used to create a new object.

The self parameter is a reference to the current instance of the class.

__call__() function: instances behave like functions and can be called like a functions.

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def sum(self):
        return self.x + self.y

    def __call__(self):
        return self.x*self.y
```

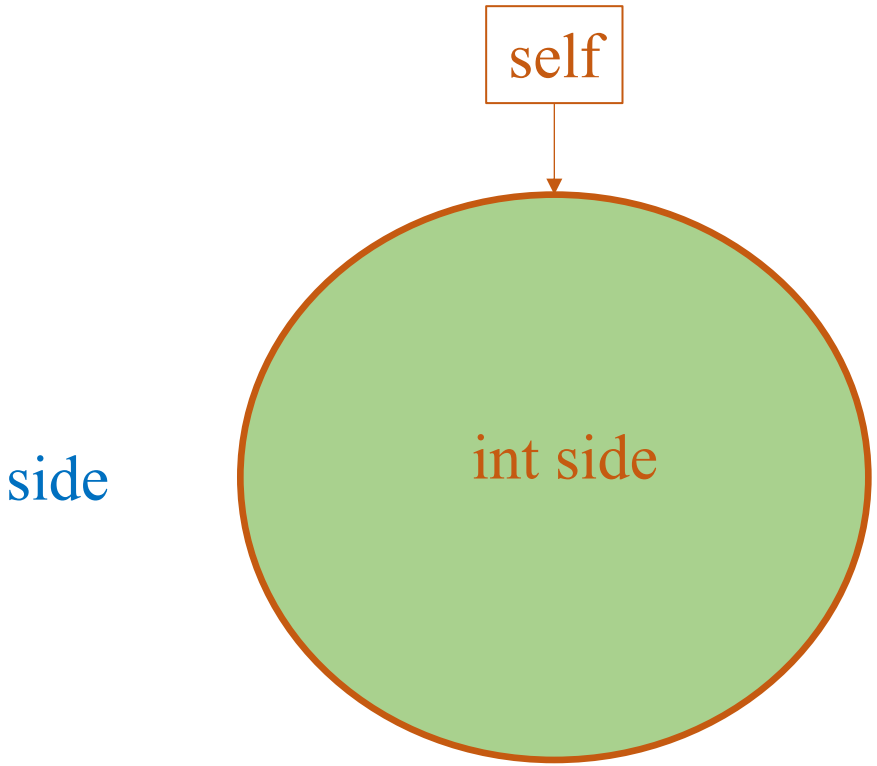
```python
point = Point(4, 5)
print(point.sum())
print(point())
```

```
9
20
```

self Keyword

Must be the first argument of methods

Used to create and access data members




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

    def compute_area(self):
        return self.side*self.side

# test sample: side=5 -> 25
square = Square(5)
area = square.compute_area()
print(f'Square area is {area}')
```

```
Square area is 25
```

Classes and Objects

Cat
+ name: string + color: string + age: float
...

```python
class Cat:
    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.age = age

# test
cat = Cat('Calico', 'Black, white, and brown', 2)
print(cat.name)
print(cat.color)
print(cat.age)
```

```
Calico
Black, white, and brown
2
```

We have a new data type

cat = Cat('Calico', 'BW', 2)

variable — create an object

Naming conventions

For class names

Including words concatenated

Each word starts with upper case

For attribute names

Using nouns

Words are connected using underscores

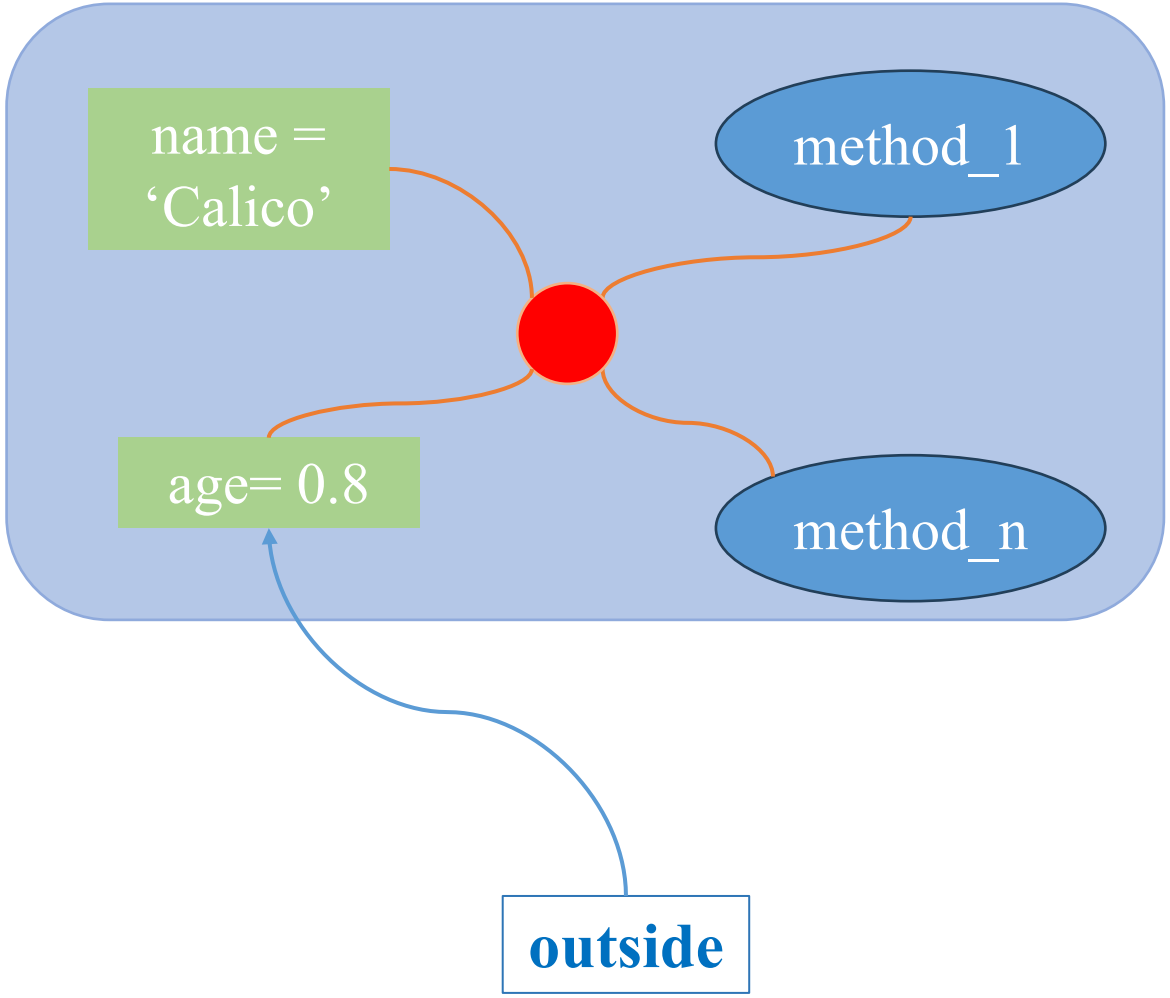
QAs

To continue

Any problem?

```python
# create a class
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __call__(self):
        print(f'Name: {self.name} - Age: {self.age}')

# test: create an object
cat = Cat('Calico', 0.8)
cat()

cat.age = cat.age + 0.3
cat()
```

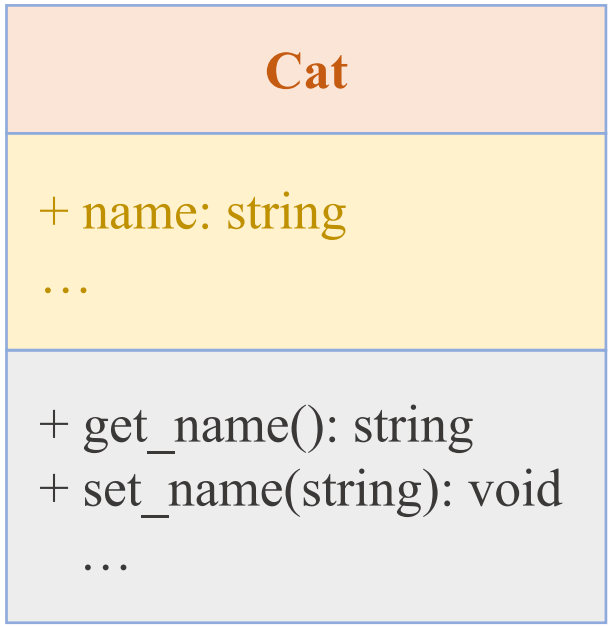
```
Name: Calico - Age: 0.8
Name: Calico - Age: 1.1
```



Classes and Objects

Problem and Solution:

Step 1 – implement getter and setter methods



Access modifiers
- private
+ public

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

    def get_name(self):
        return self.name

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

# test
cat = Cat('Calico')
print(cat.get_name())

cat.set_name('Japanese Bobtail')
print(cat.get_name())
```

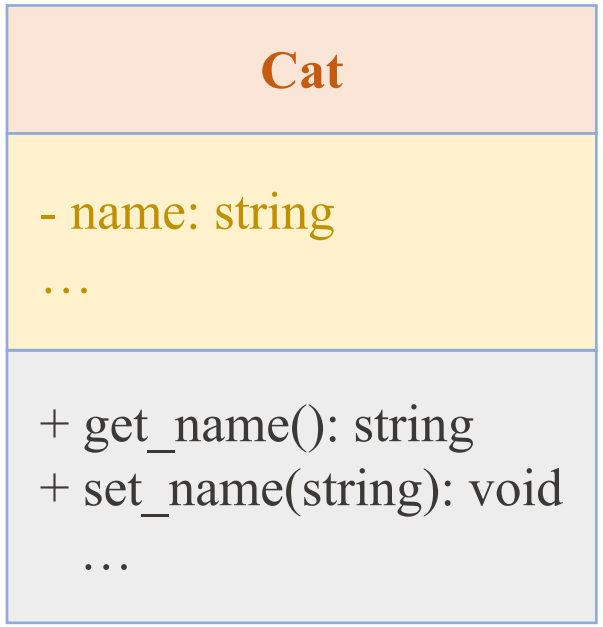
```
Calico
Japanese Bobtail
```

Classes and Objects

Solution: Step 2

Using private for attributes

Access modifiers
- private
+ public




```
print(cat.__name)
```

```
---------------------------------------------------------------
AttributeError                                Traceback (m
Cell In[4], line 1
----> 1 print(cat.__name)

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

```
class Cat:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

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

# test
cat = Cat('Calico')
print(cat.get_name())

cat.set_name('Japanese Bobtail')
print(cat.get_name())
```

```
Calico
Japanese Bobtail
```

Classes and Objects

Takeaways

Use the private access modifiers for typical attributes

Create getter and setter methods to access the attributes

Use the public access modifiers for the getter and setter functions

Access modifiers
- private
+ public

A cat includes a name, a color, and an age.

Cat
- name: string - color: string - age: float
+ get_name(): string + set_name(string): void + get_color(): string + set_color(string): void + get_age(): int + set_age(int): void

Outline

SECTION 1

Introduction to OOP

SECTION 2

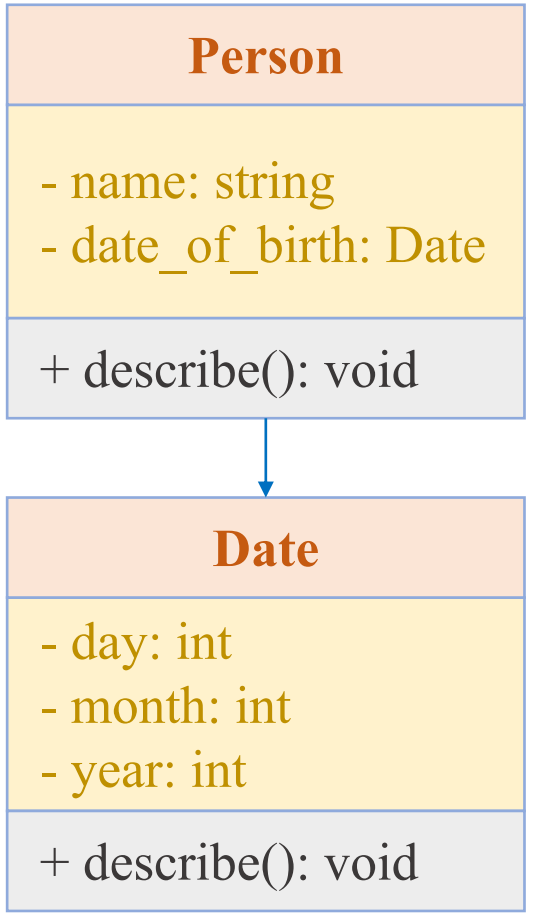
Objects and Classes

SECTION 3

Delegation

SECTION 4

Inheritance



Class Data Type

❖ Using a class as a data type

A person comprises a name in string and a date of birth. A date consists of day, month, and year.

Write a function to check if two people have the same name.

Write a function to check if two people have the same date of birth.

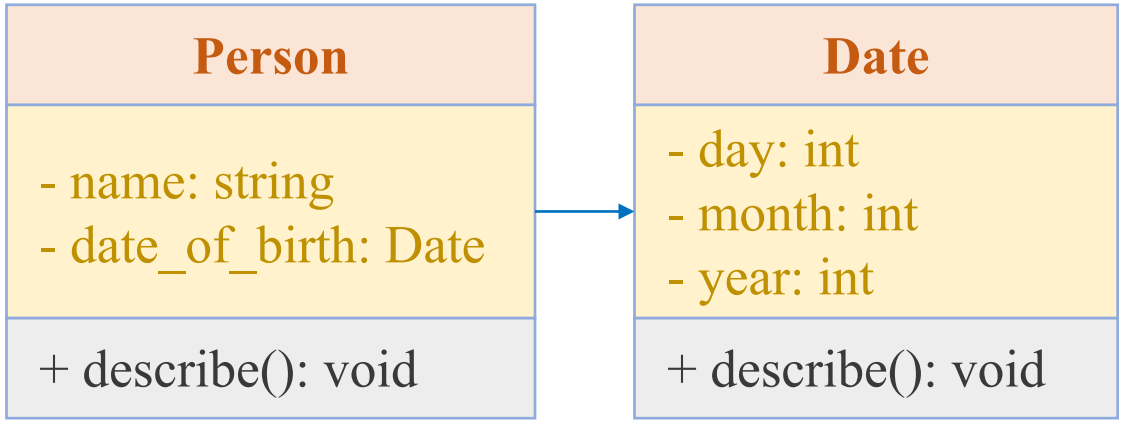
Draw a class diagram and implement in Python

Class Data Type

❖ Using a class as a data type

A person comprises a name in string and a date of birth. A date consists of day, month, and year.

Draw a class diagram and implement in Python

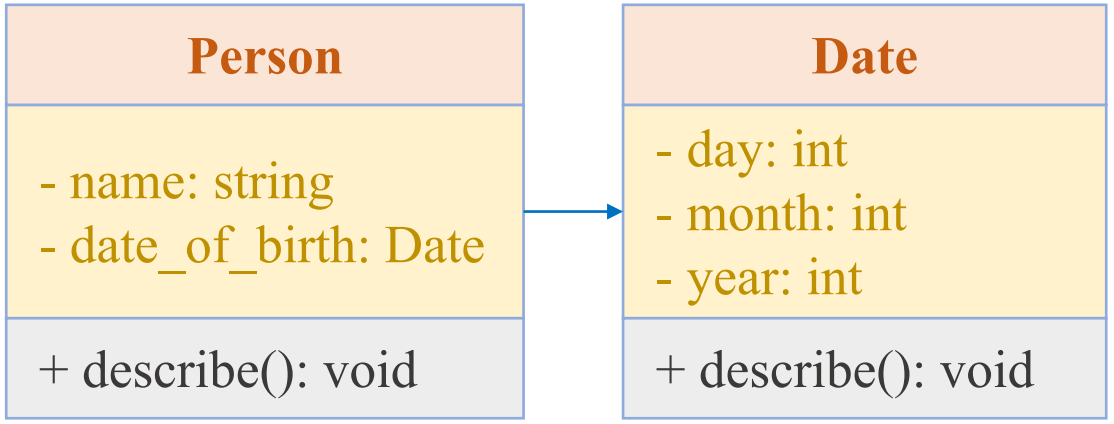



```python
class Date:
    def __init__(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year

    def get_day(self):
        return self.__day

    def get_month(self):
        return self.__month

    def get_year(self):
        return self.__year
```

Class Data Type

❖ Using a class as a data type

A person comprises a name in string and a date of birth. A date consists of day, month, and year.



Using Date as a data type

how to improve?

```python
class Person:
    def __init__(self, name, date_of_birth):
        self.__name = name
        self.__date_of_birth = date_of_birth

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

        # print date
        day = self.__date_of_birth.get_day()
        month = self.__date_of_birth.get_month()
        year = self.__date_of_birth.get_year()
        print(f'{day}/{month}/{year}')
```

```python
date = Date(10, 1, 2000)
peter = Person('Peter', date)
peter.describe()
```

```
Peter
10/1/2000
```

Class Data Type


```python
class Date:
    def __init__(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year

    def get_day(self):
        return self.__day

    def get_month(self):
        return self.__month

    def get_year(self):
        return self.__year

    def describe(self):
        print(f'{self.__day}/{self.__month}/{self.__year}')
```

delegation

```python
class Person:
    def __init__(self, name, date_of_birth):
        self.__name = name
        self.__date_of_birth = date_of_birth

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

        # print date
        self.__date_of_birth.describe()
```

```python
date = Date(10, 1, 2000)
peter = Person('Peter', date)
peter.describe()
```

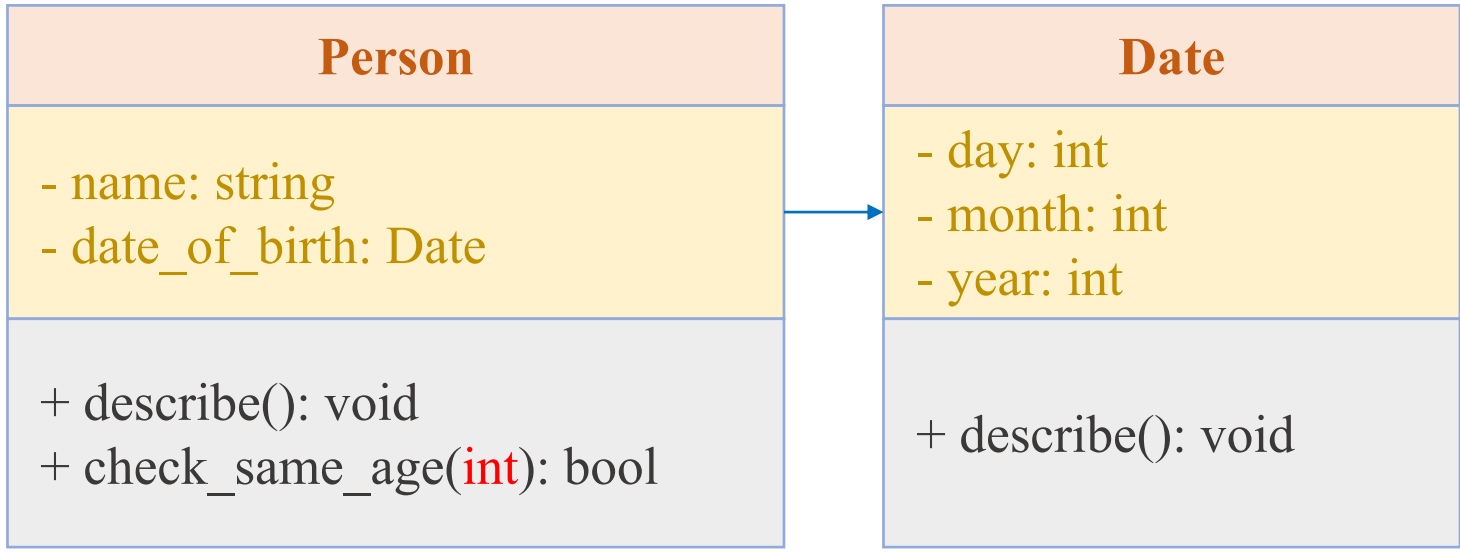
```
Peter
10/1/2000
```

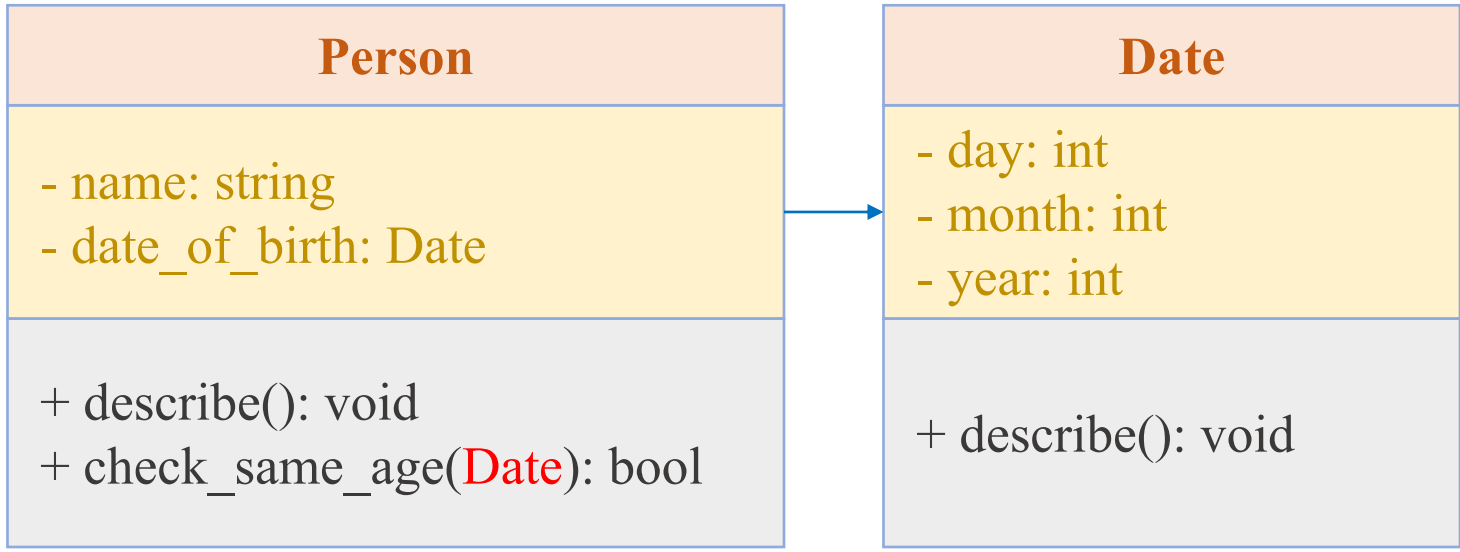
Class Data Type

❖ Using a class as a data type

A person comprises a name in string and a date of birth. A date consists of day, month, and year.

Write a function to check if two people have the same date of birth.





Inheritance

SECTION 4.1

Inheritance: To Reuse

SECTION 4.2

Inheritance: Overriding

SECTION 4.3

Inheritance: As a Template

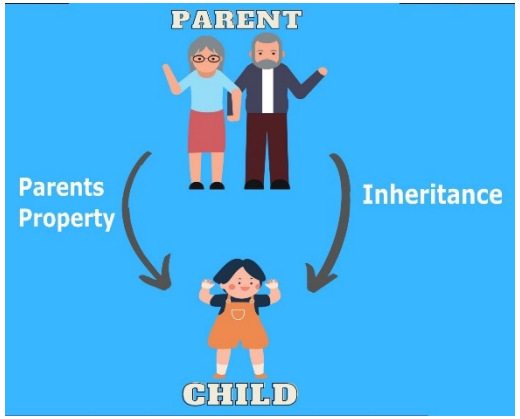
SECTION 4.4

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

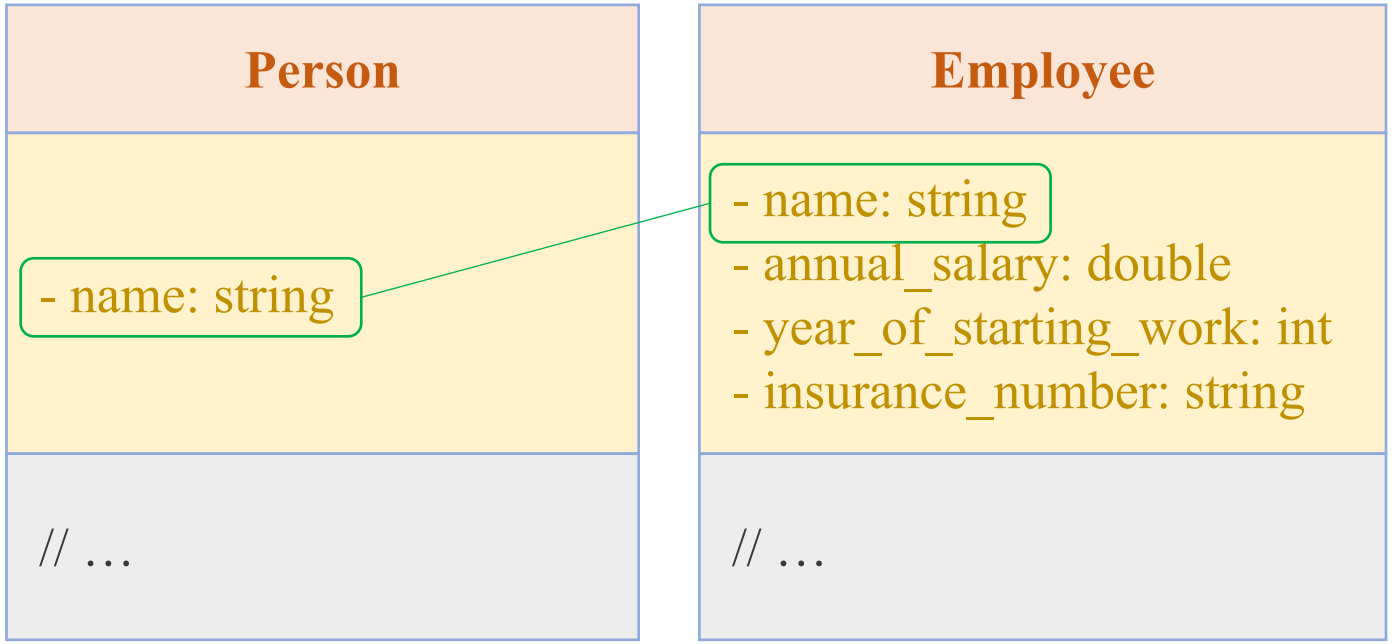
Create a class called Employee whose objects are records for an employee. This class will be a derived class of the class Person.

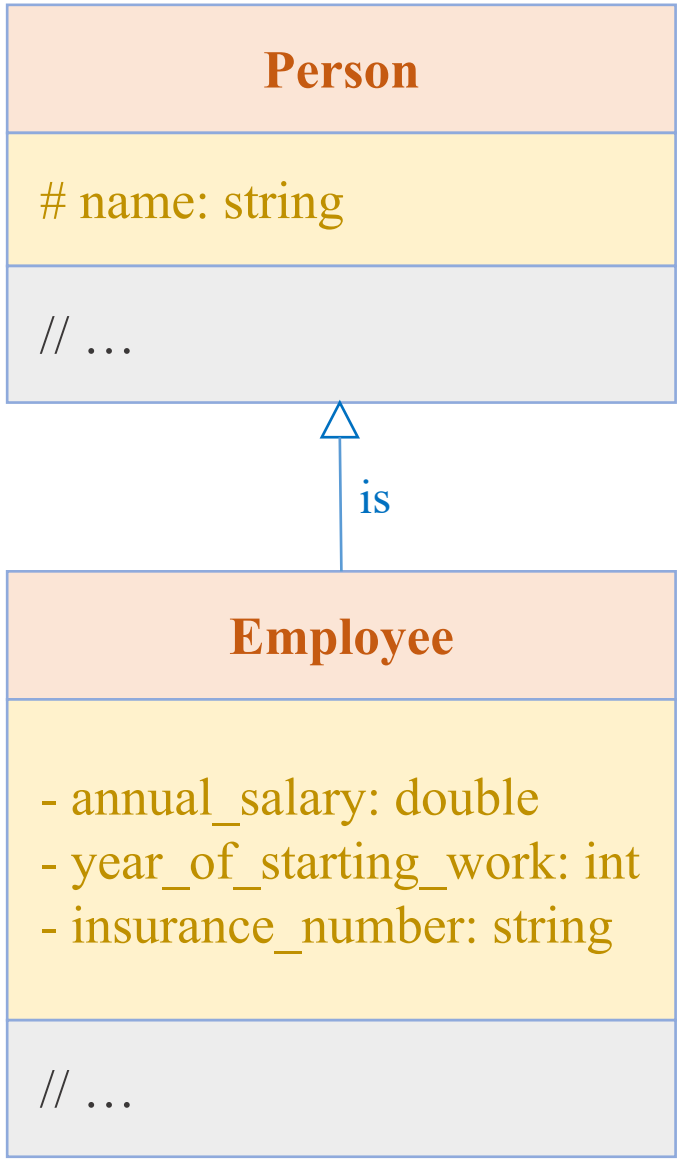
An employee record has an employee's name (inherited from the class Person), an annual salary represented as a single value of type double, a year the employee started work as a single value of type int and a national insurance number, which is a value of type String.

Person
- name: string
// …

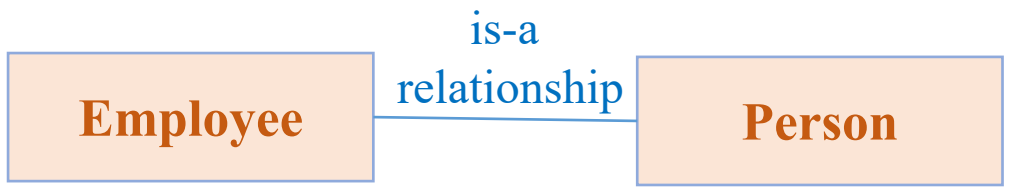
Employee
- name: string - annual_salary: double - year_of_starting_work: int - insuarance_Number: string
// …

Inheritance





Access modifiers
- private
+ public
# protected



An employee is a person.

Inheritance

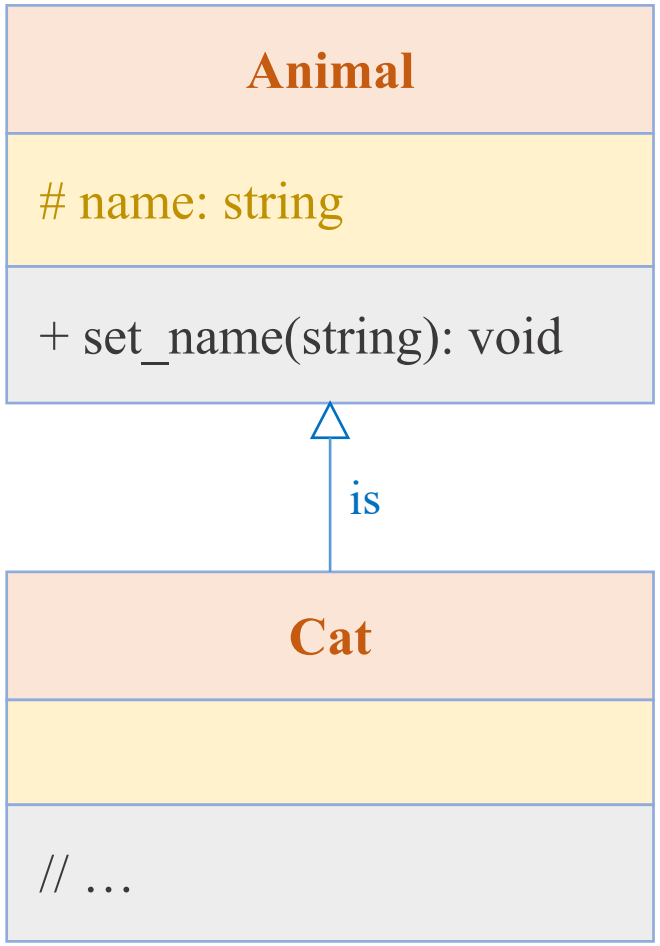
Inherit attributes and methods from one class to another

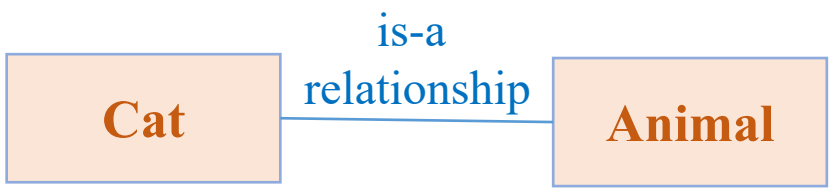
Benefit: Code reusability

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.

```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
```

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

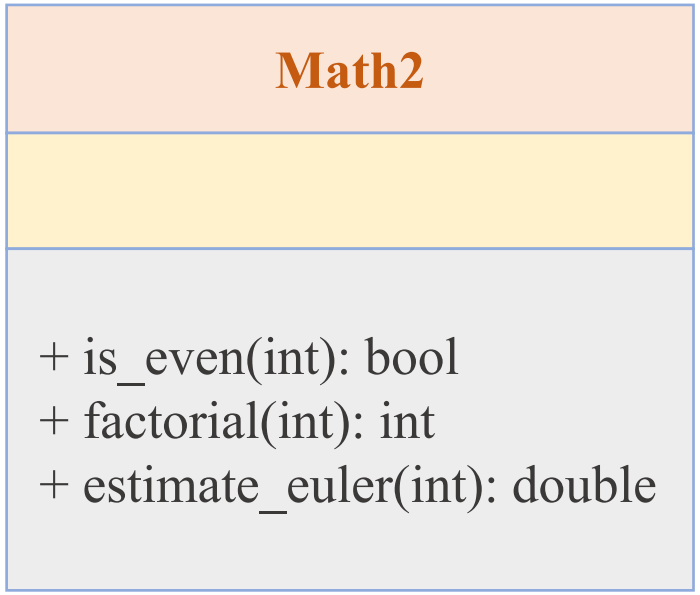
```
False
True
24
120
```

Example

❖ Implement the two classes below

$$e = 2.71828$$

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




```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

$$e = 2.71828$$

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

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

```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
```

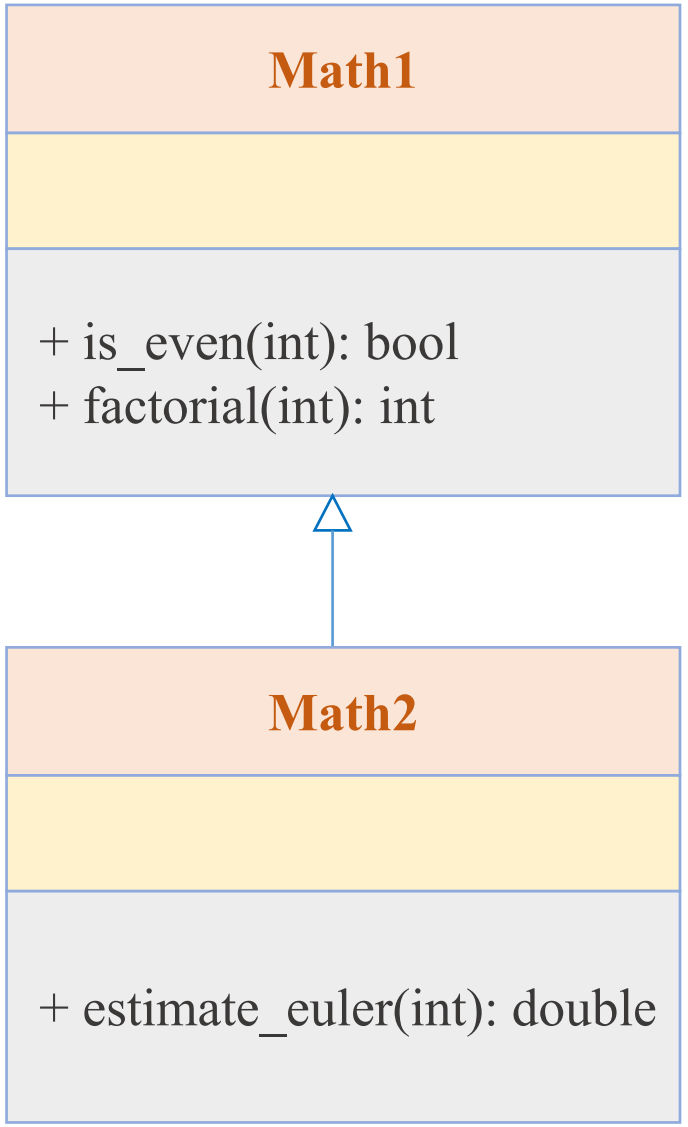
Example

❖ 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

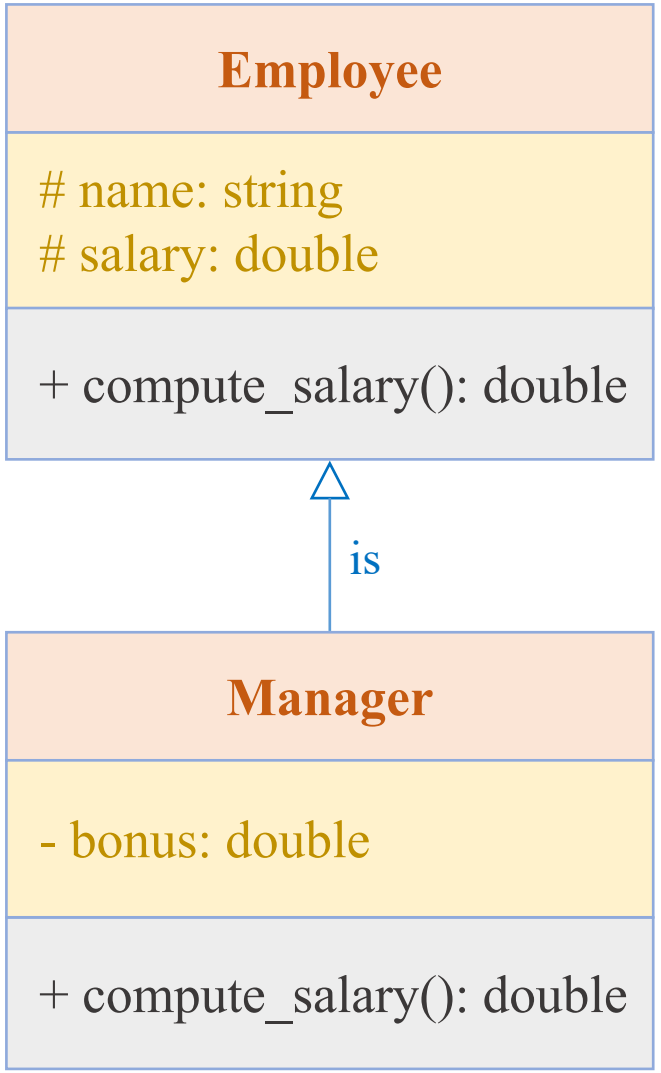
Inheritance: Overriding

SECTION 3

Inheritance: As a Template

SECTION 4

Custom Class in PyTorch

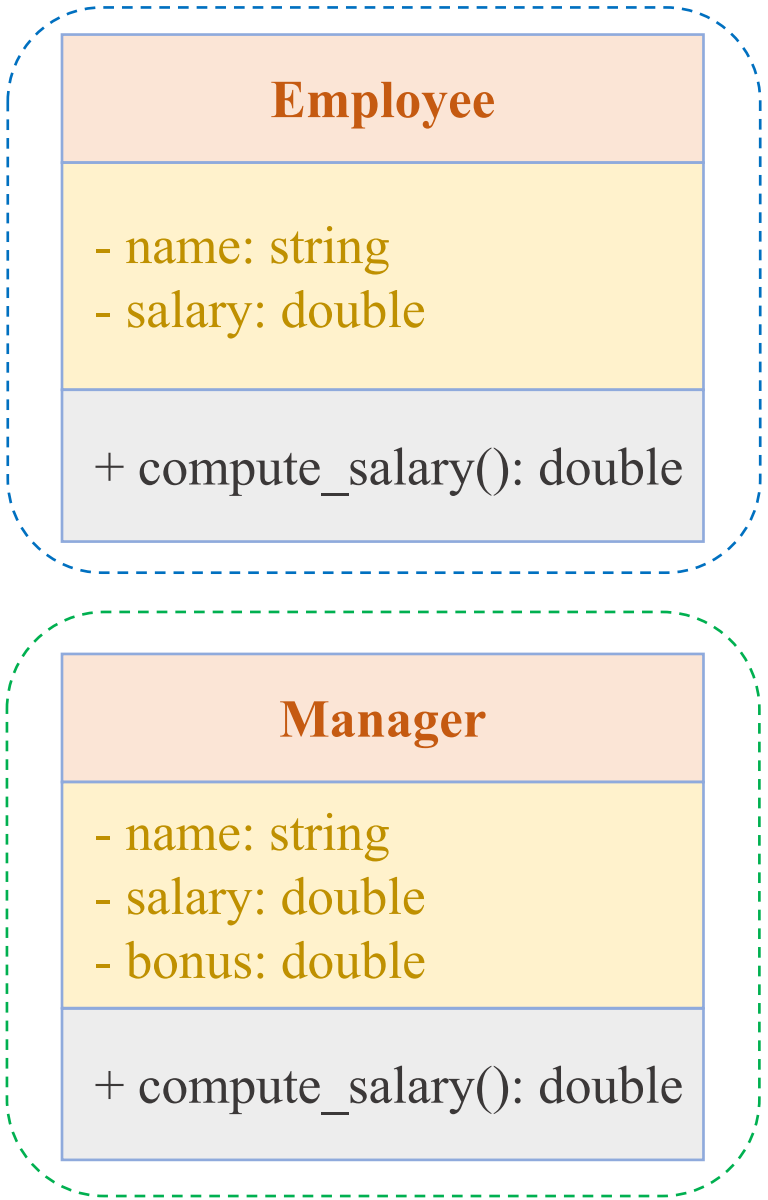


Example

❖ 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 compute_salary() method to compute the final salary for an employee.

A manager includes his/her name, base salary, and bonus. The final salary for the manager comprises the Employee 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.



Inheritance

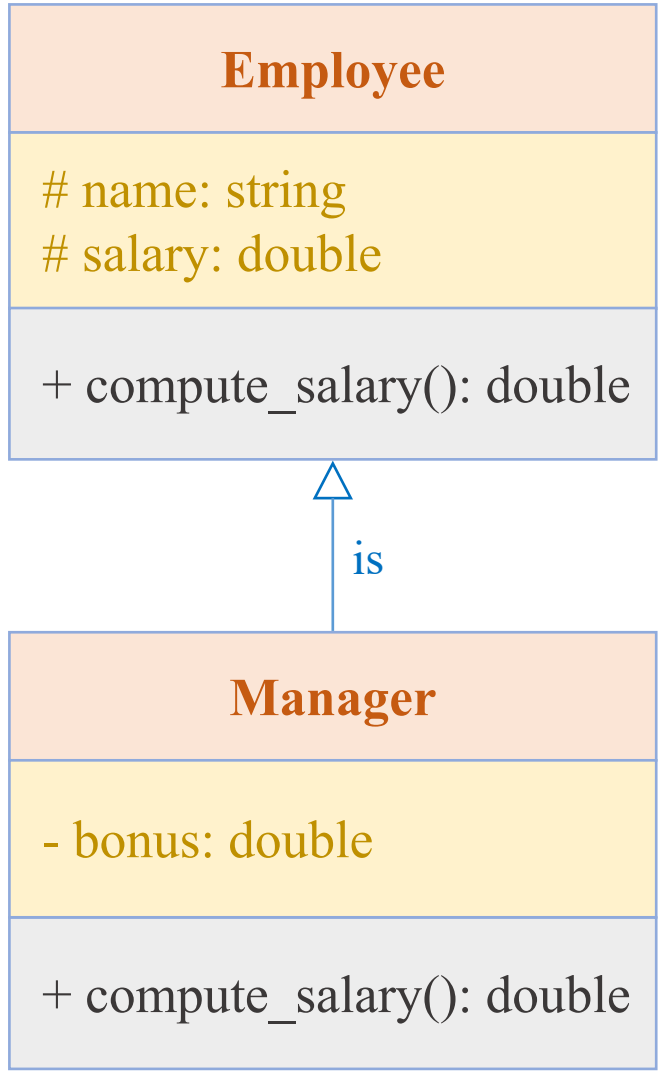
❖ Introduction

To extend an existing class

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

Super Class

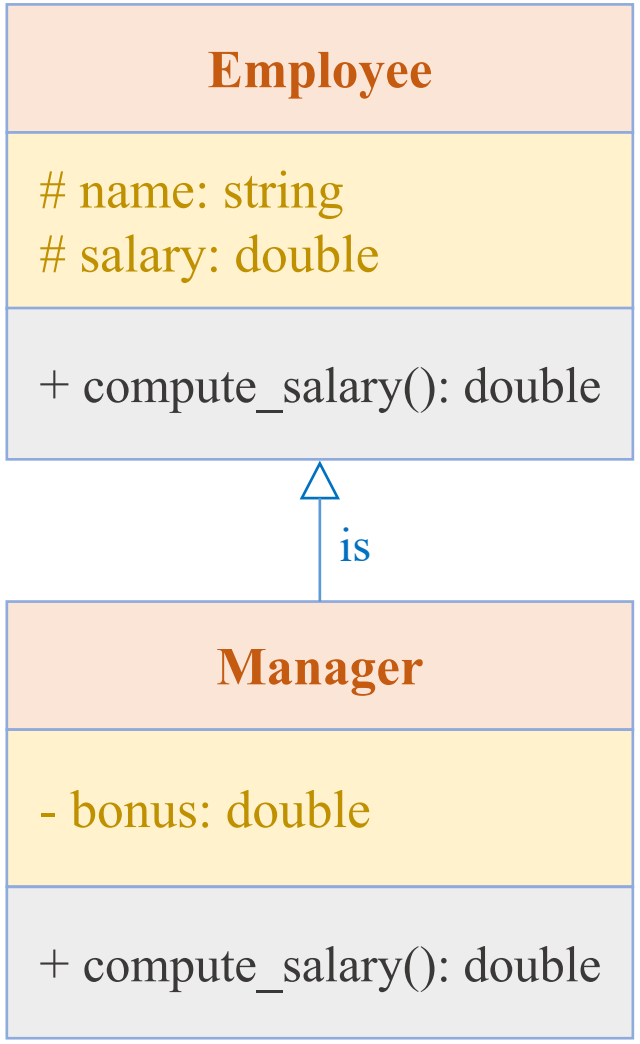
Subclass



What features does a manager inherit?

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

Employee ~ super()

```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

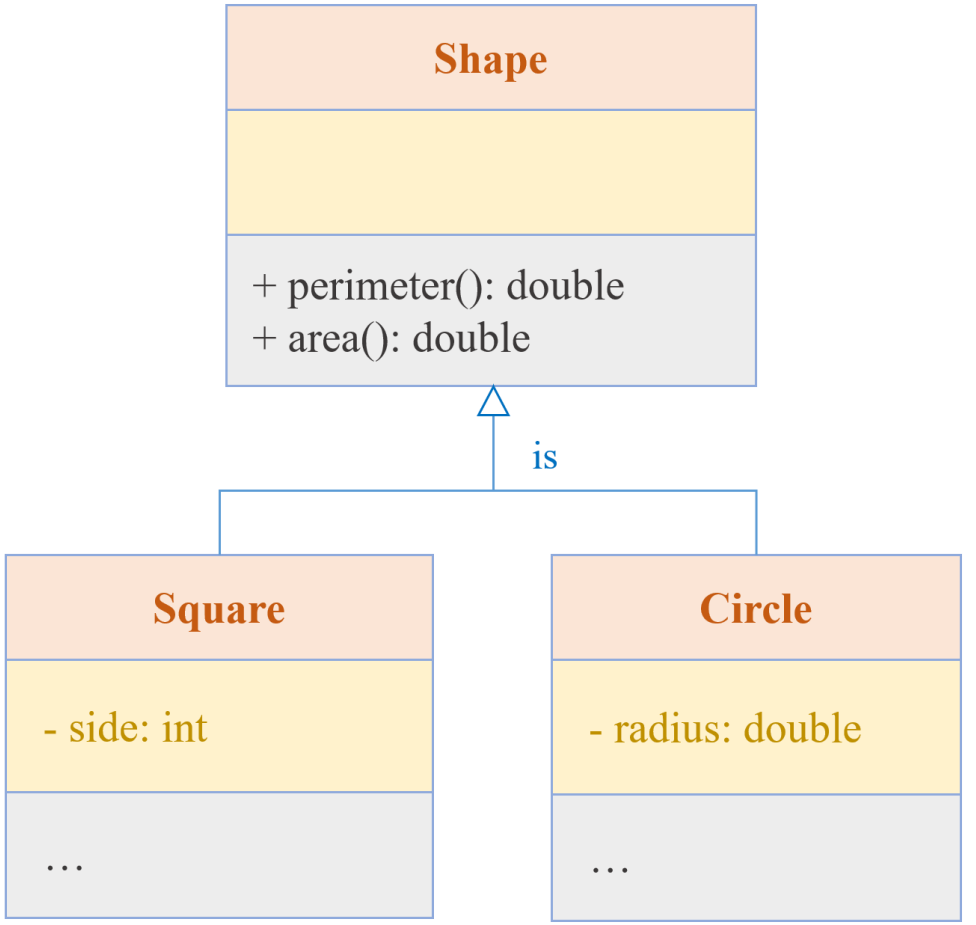
Inheritance: Overriding

SECTION 3

Inheritance: As a Template

SECTION 4

Custom Class in PyTorch



Inheritance

❖ As a template

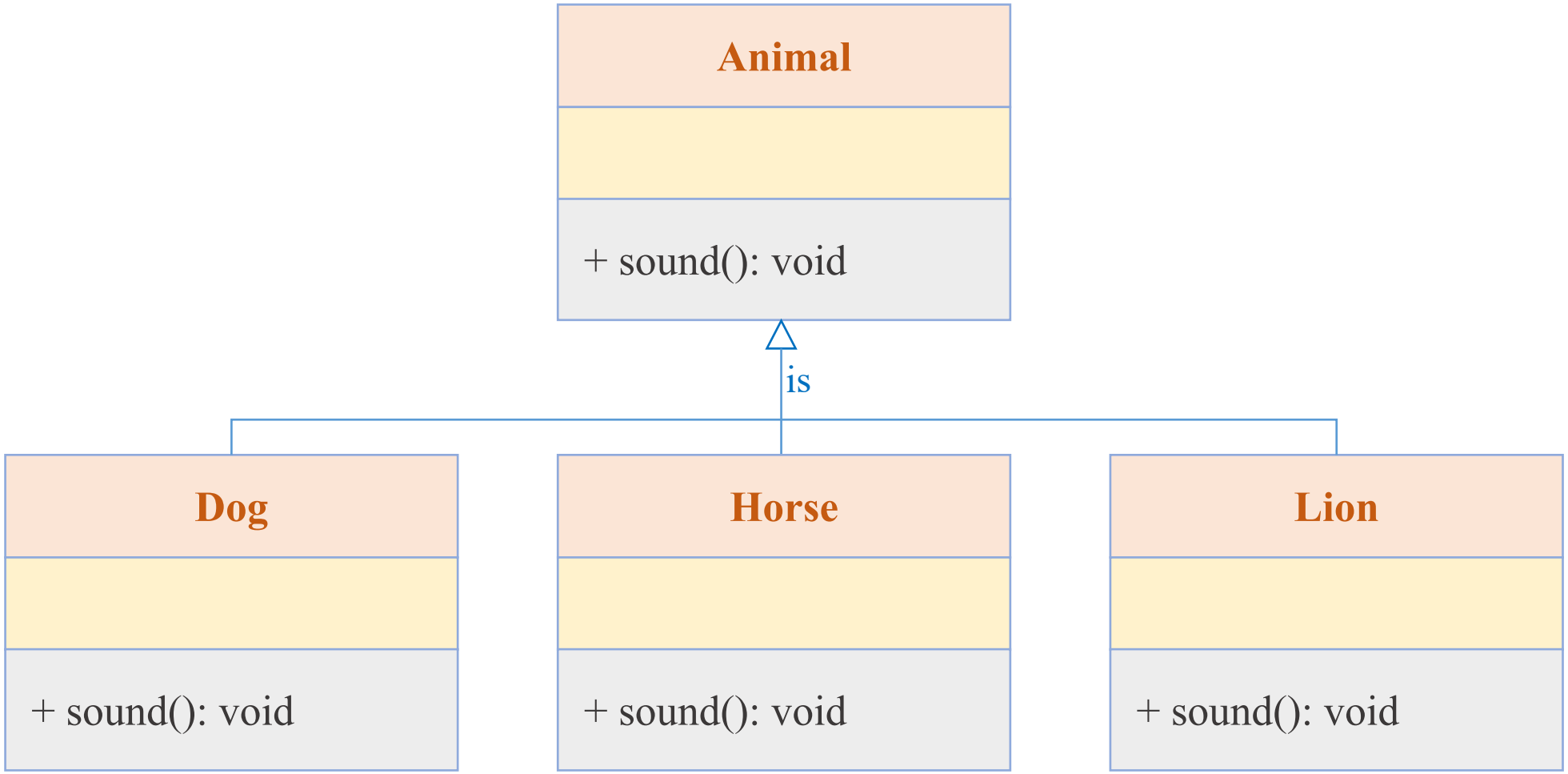
A class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat, etc.

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

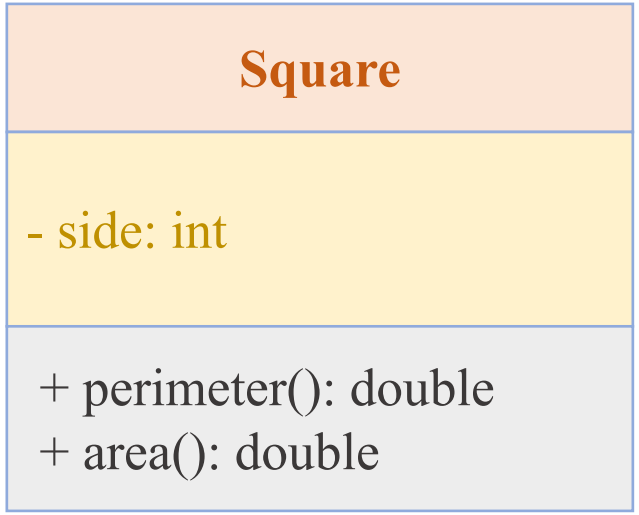
❖ As a template

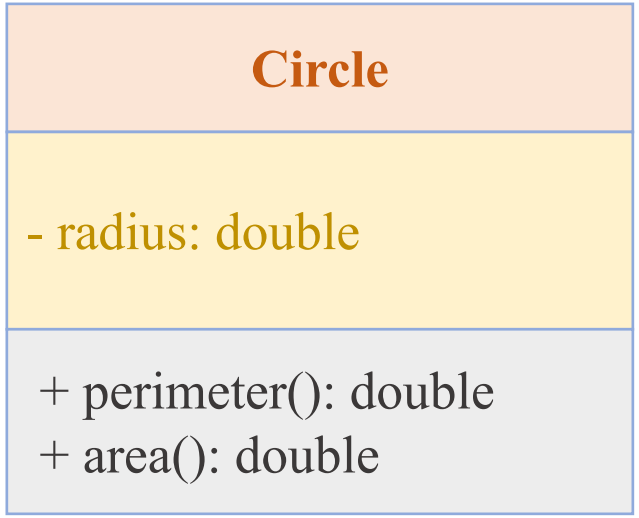


Example

❖ 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?’)





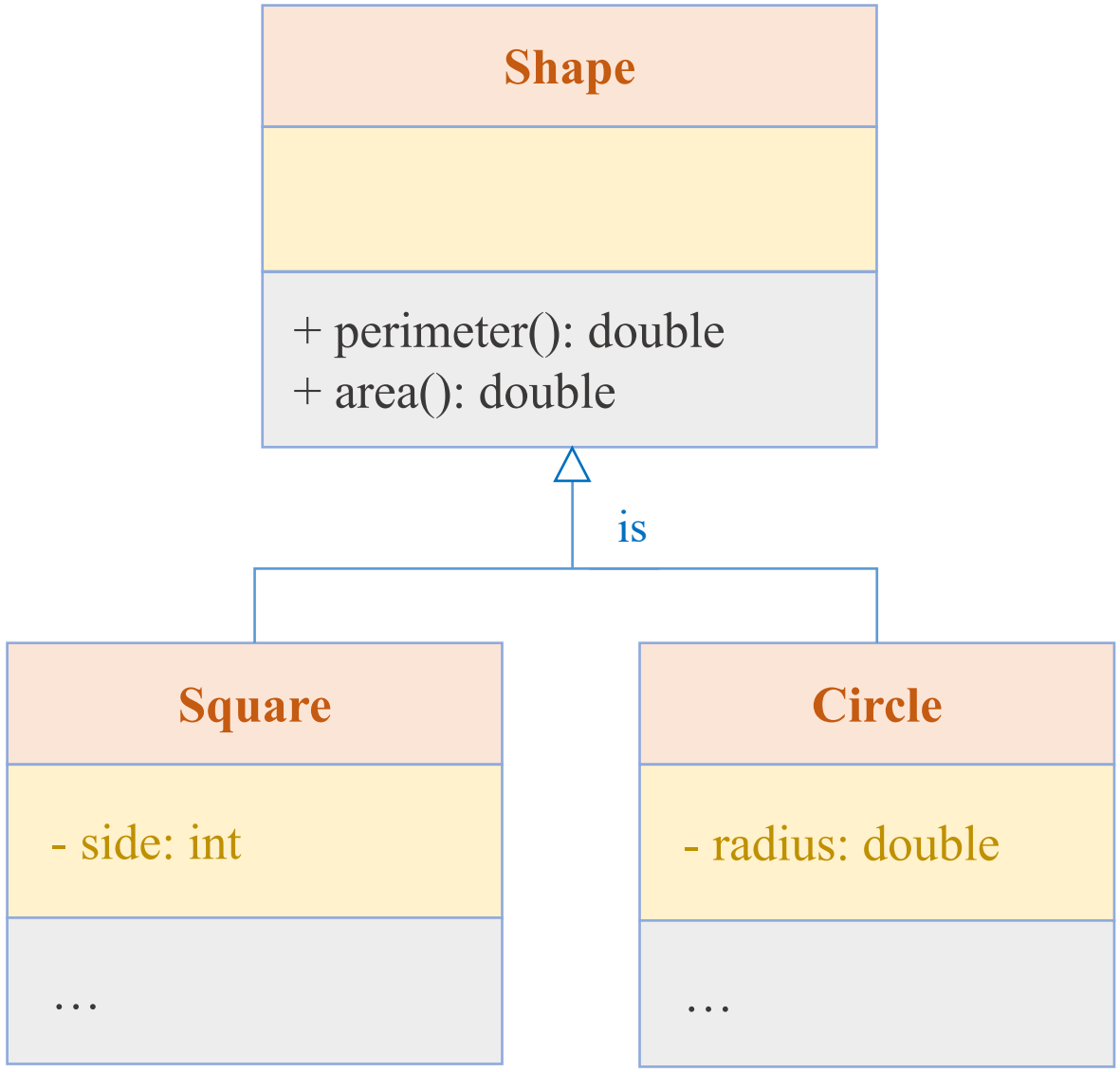
Example

❖ Inheritance recognition

Shape does not know how to compute its perimeter and area

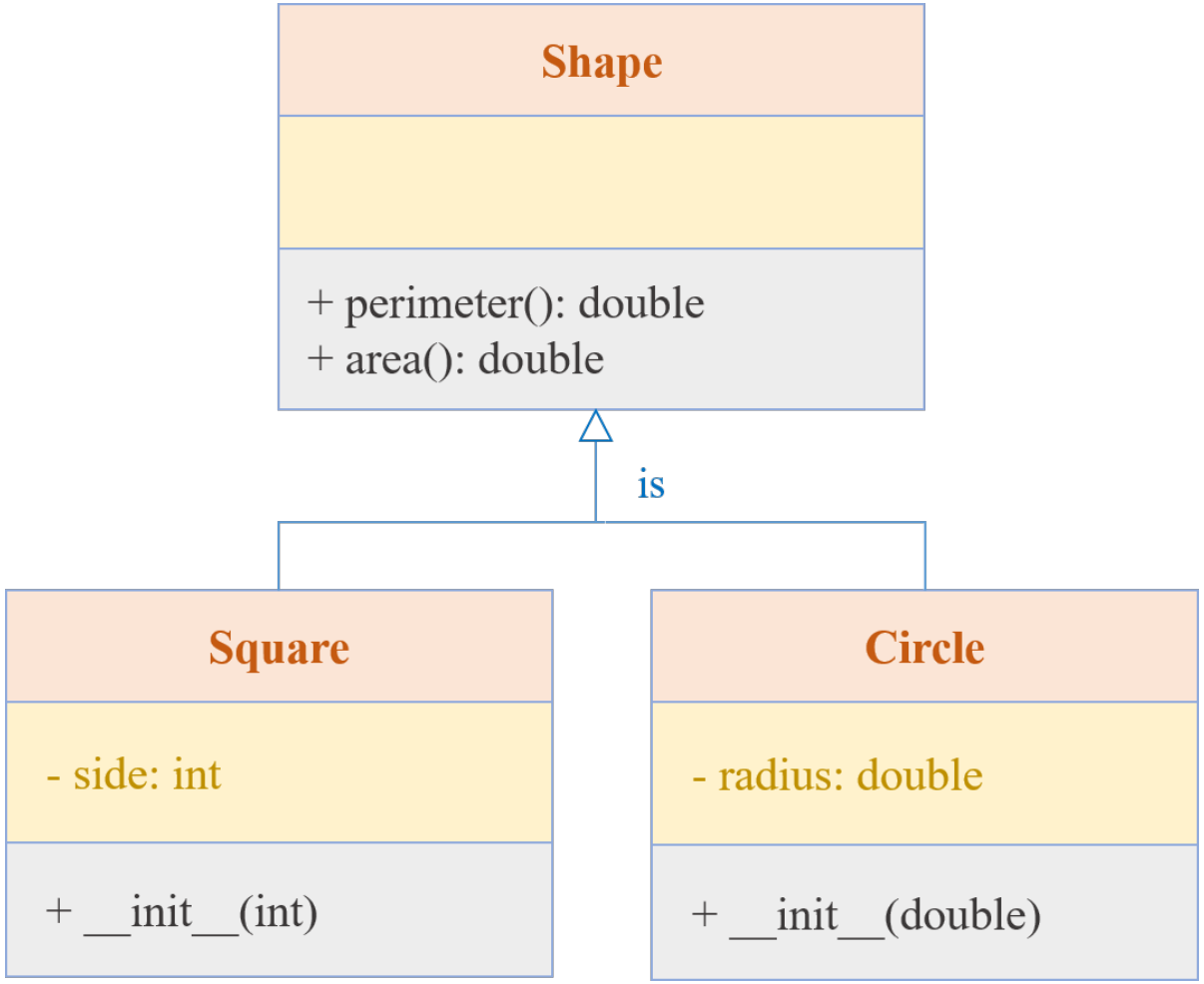
Use @abstractmethod to ask its child to implement them

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

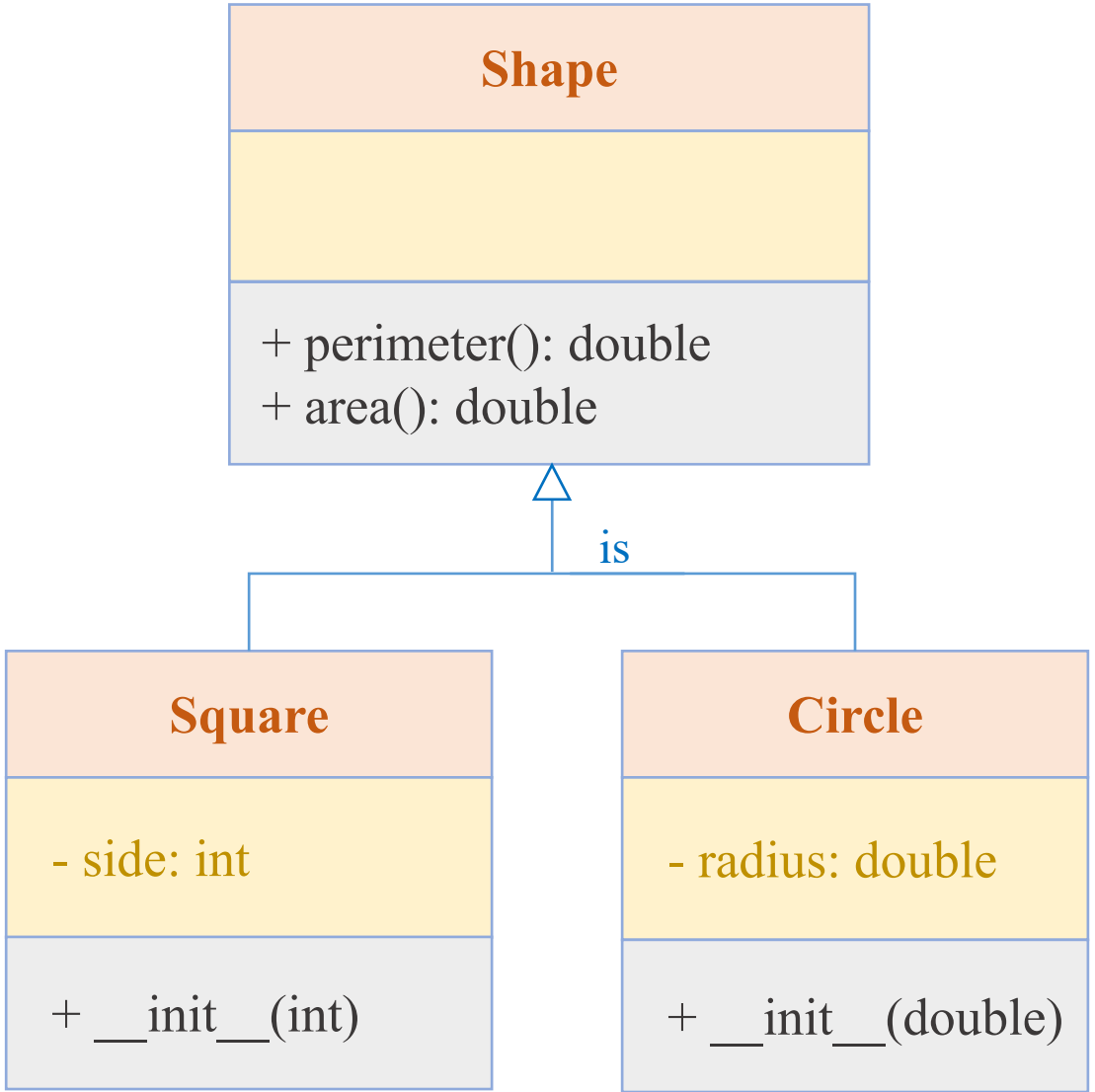
```python
square = Square(5)
print(square.compute_area())
```

✓ 0.0s

```
25
```

Example

❖ Inheritance recognition




```python
from abc import ABC, abstractmethod

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

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

# test
circle = Circle(5)
```

⊗ 0.0s

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

TypeError: Can't instantiate abstract class Circle without an implementation
                                                  for abstract method 'compute_area'
```

Example

❖ Inheritance recognition

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

```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
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

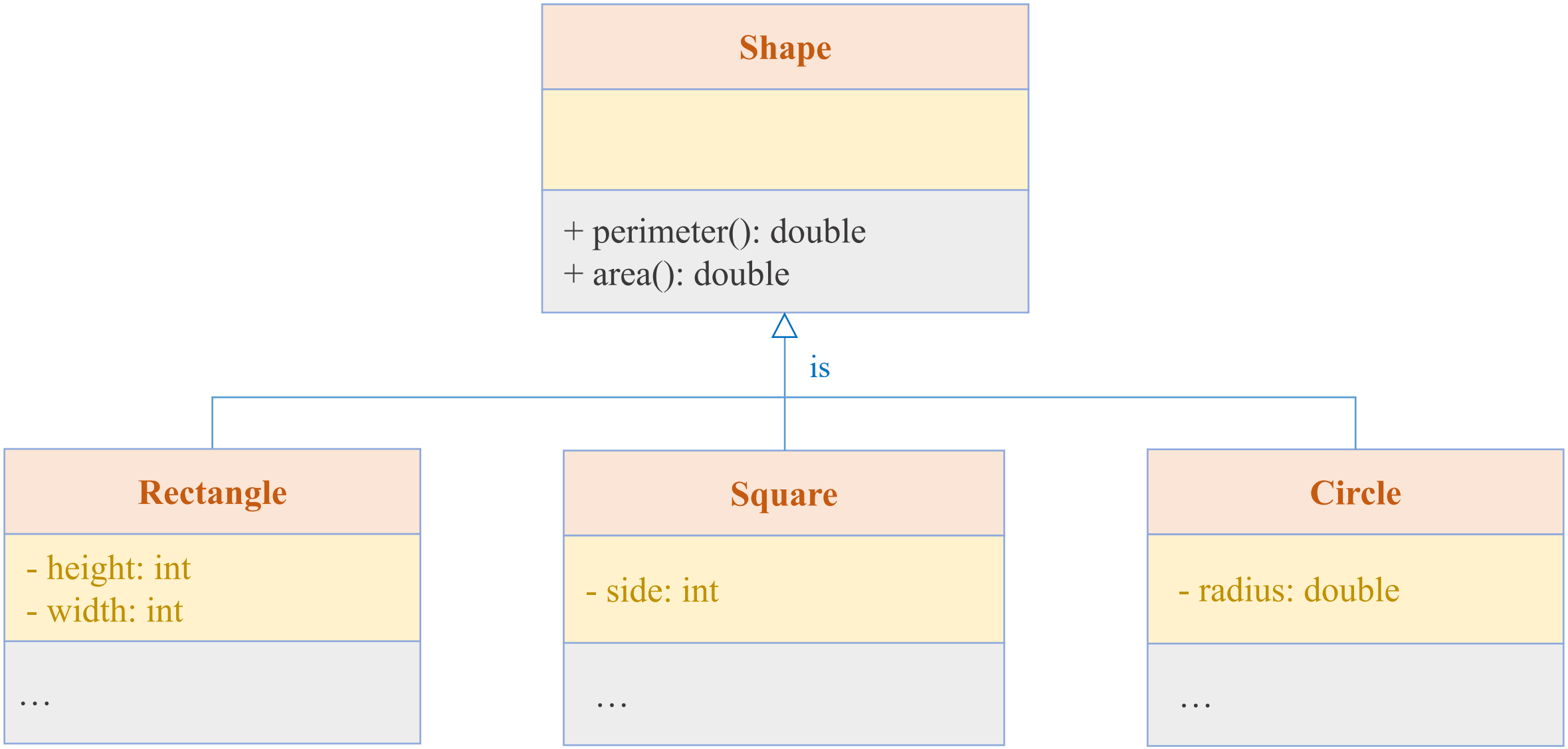
```python
square = Square(5)
print(square.compute_area())
```

✓ 0.0s

```
25
```

Example

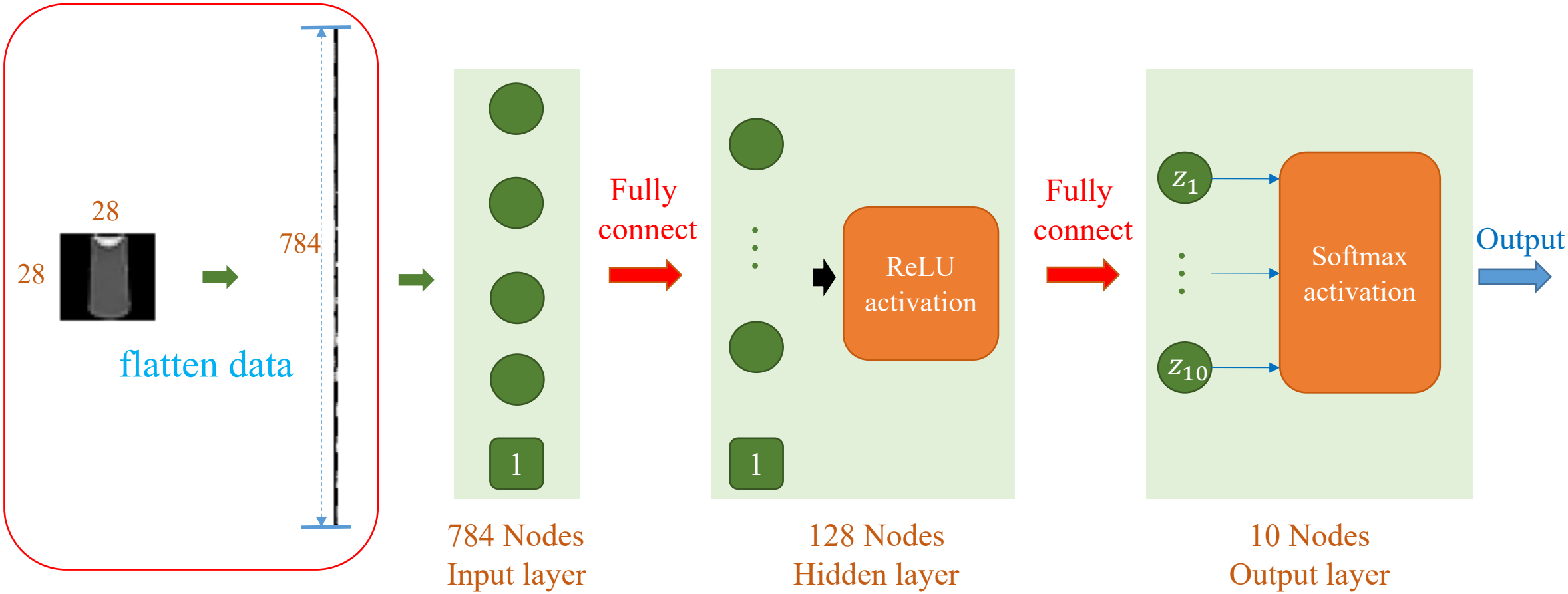
❖ Open for extension



Custom Class in PyTorch

Custom Layer in PyTorch

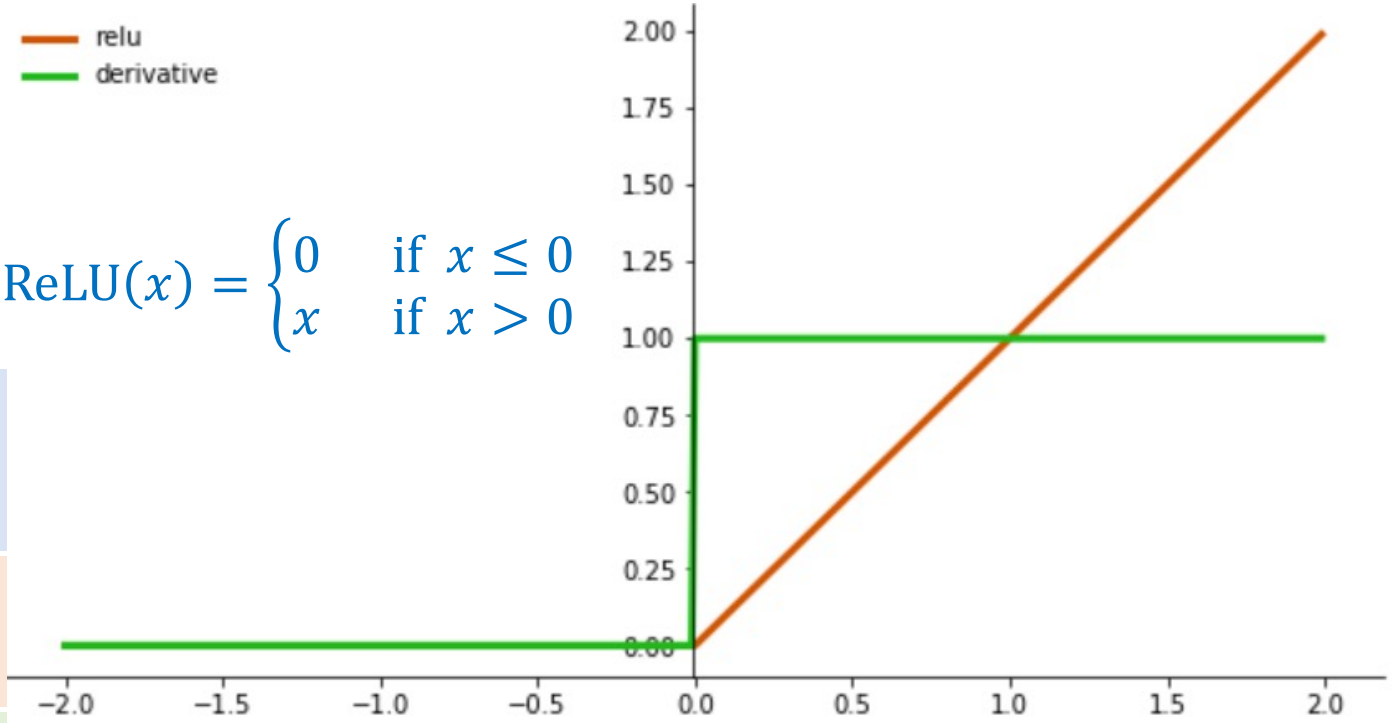
❖ ReLU Layer



Custom Layer in PyTorch

❖ Custom ReLU

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



Custom class must inherit the Module class

init() method: need call the init method of the Module class

forward() method

```python
import torch
import torch.nn as nn

class MyReluActivation(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return torch.clamp(x, min=0)
```

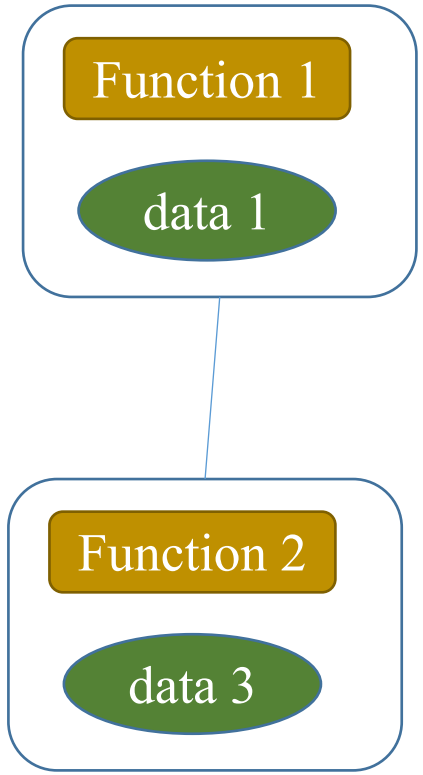
Given
torch.clamp(x, min=0) ~ ReLU

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

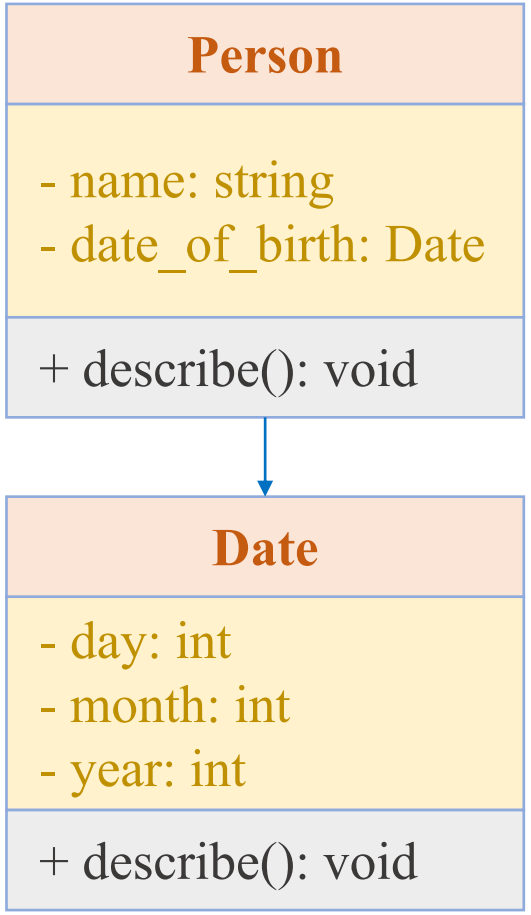
```
tensor([1., 0., 3.])
```

Summary

Class (Encapsulation)



Delegation



Stack & Queue