AI VIET NAM
@aivietnam.edu.vn

Classes and Objects

TA. Đăng Nhã
TA. Dương Khanh

Outline

- Step into Object-Oriented Programming
- Class and Object
- Abstraction: Constructor, Attributes. Method
- Inheritance - Practice
- Q&A

Step into OOP

Programming

Programming

Abstraction

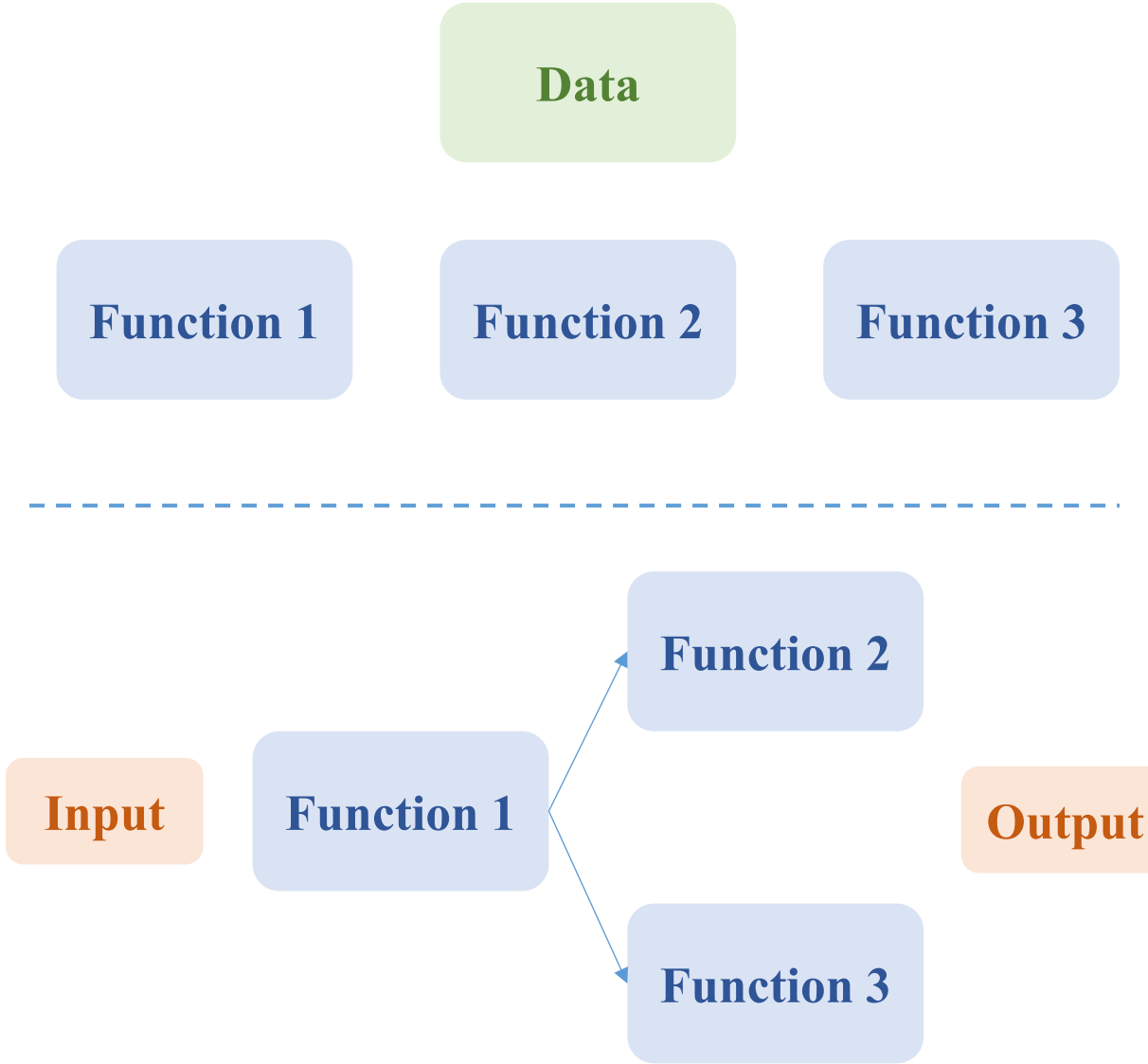
The real world we live 🌍

Software – Digitial World

Procedural Programming

❖ Library Management

```python
# Global list of books
books = []

def add_book(title, author):
    books.append({"title": title, "author": author})

def list_books():
    for book in books:
        print(f"{book['title']} by {book['author']}")

# Main program
add_book("1984", "George Orwell")
add_book("The Hobbit", "J.R.R. Tolkien")
list_books()
```

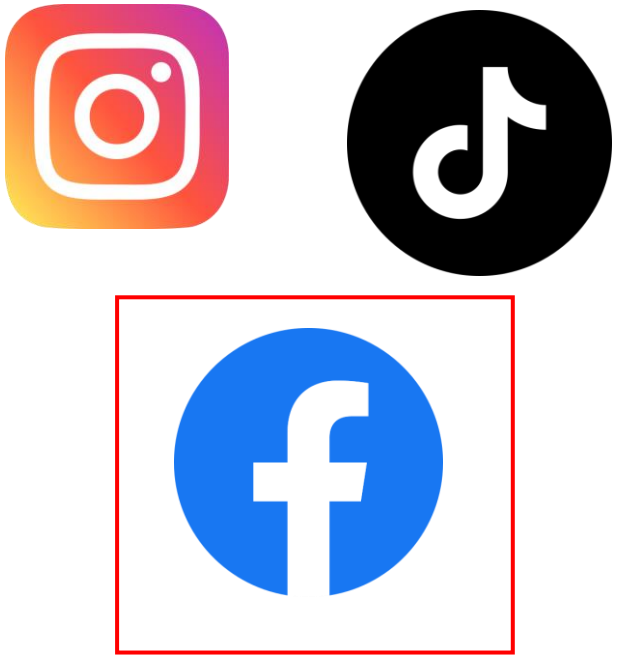


Abstraction

OOP

Social Media

Abstraction



Social Media

Object

User

Attributes

- Name
- Birthday
- Gender
- Phone
- Email

Methods

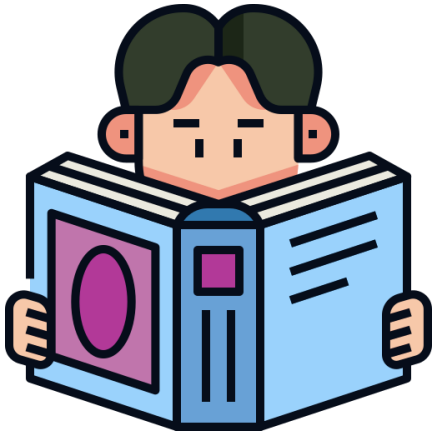
- Friends
- Post
- Message
- Like, comment

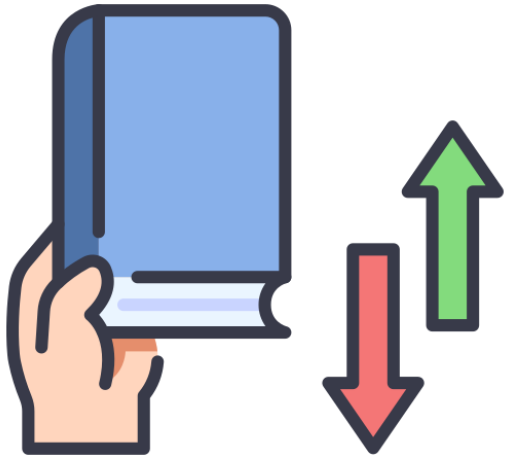
Library Management

❖ Problem Define

Book

Reader





Process

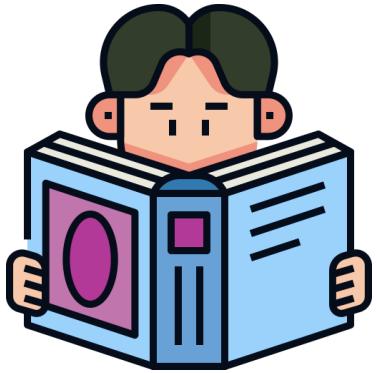
Librarian

Library Management

❖ Classes

```
class Book

class Reader

class Librarian
```

OOP Syntax



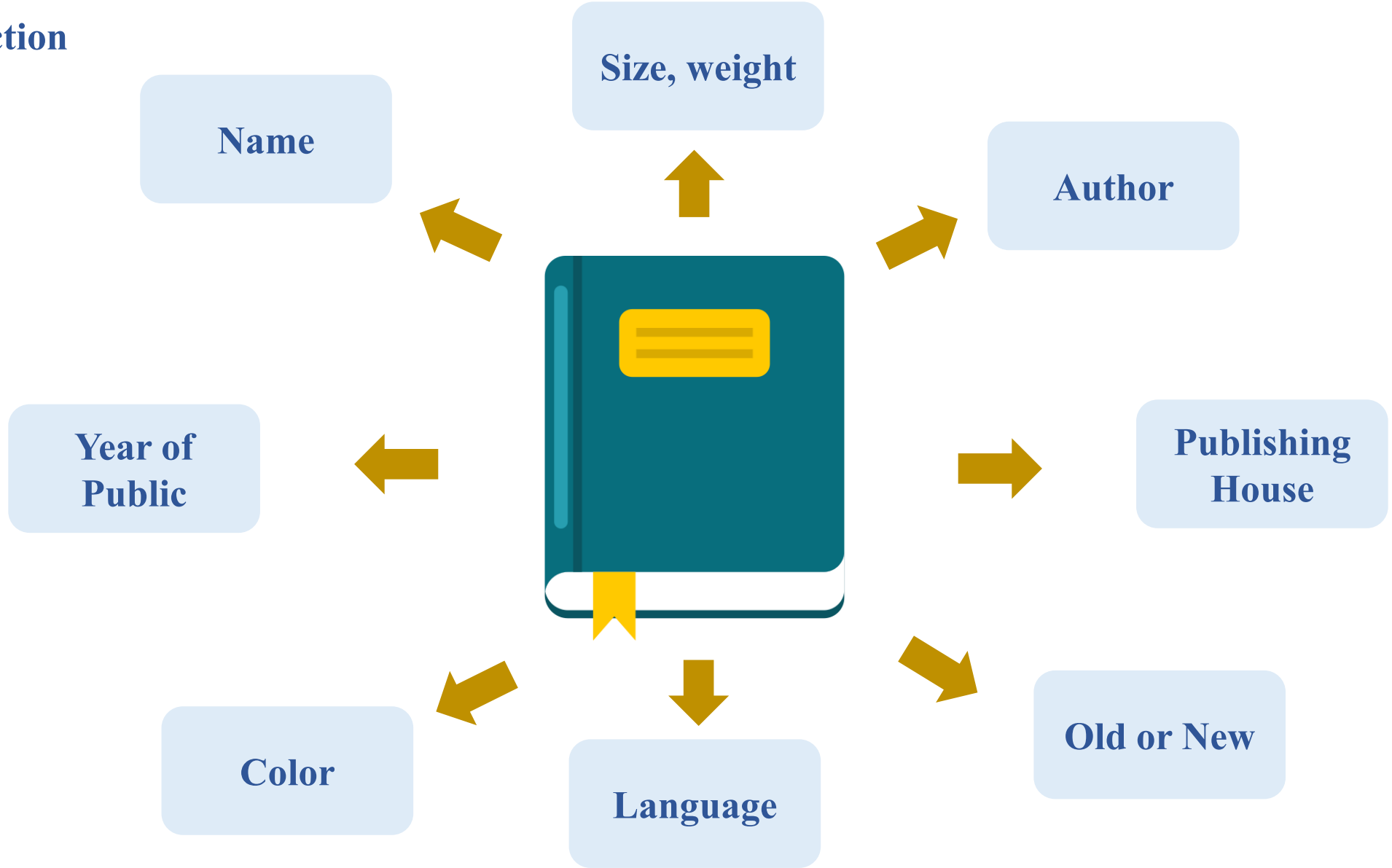
Reader

Librarian

Book

Library Management

❖ **Abstraction**



Library Management

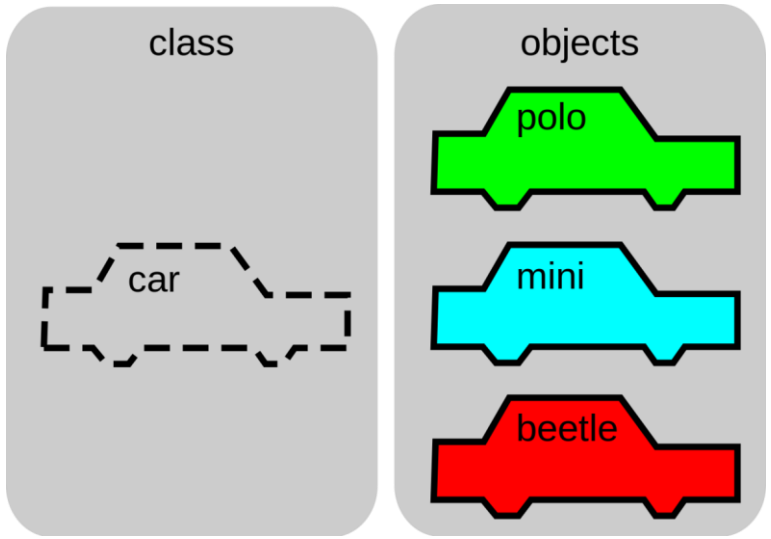
❖ **What we really need?**

Book Management

title	author	rm_time	is_borrow
Mastery	A	5 days	True
Five form of in Telligent	B	0 days	False
Python programming	C	1 days	True
Machine Learning	D	0 days	False

Class & Object

❖ Attributes



In **object-oriented programming**, a **class** is a *blueprint* for creating **objects** (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

```
book1 = Book("Mastery", "Robert Greene")
book2 = Book("1984", "Geogre Orwell")
```

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.rm_time = 0
        self.is_borrow = False
```

Constructor

❖ Method

```python
class Book:
    ...
    def borrow(self):
        if not self.is_borrowed:
            self.is_borrowed = True
            print(f"Bạn đã mượn: {self.title}")
        else:
            print(f"Sách {self.title} đã có người mượn.")

    def return_book(self):
        self.is_borrowed = False
        print(f"Đã trả sách: {self.title}")
```

Class and Object

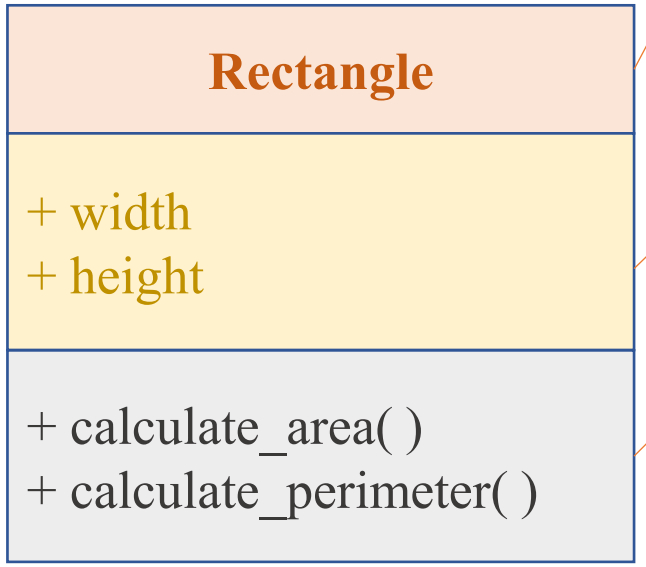
Classes and Objects

❖ Syntax for creating a class

Class name

Attributes

Methods




```python
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        self.area = self.width * self.height
        return self.area

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```python
my_rec = Rectangle(4, 7)
print(my_rec.calculate_area())
print(my_rec.calculate_perimeter())
```

```
28
22
```

Create Object

Classes and Objects

❖ Syntax for creating a class

- A class is a template for creating object.
- It is possible to create *multiple objects* from *one class*.

- An object is an instance of a class.
- Another term for object is instance.

Class

```
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        self.area = self.width * self.height
        return self.area

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```
my_rec = Rectangle(4, 7)
print(my_rec.calculate_area())
print(my_rec.calculate_perimeter())
```

Objects

Classes and Objects

❖ Class - Constructor

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

The __init__() method is used to initialize the attributes of the object with specific values.

```
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        self.area = self.width * self.height
        return self.area

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```
my_rec = Rectangle(4, 7)
print(my_rec.calculate_area())
print(my_rec.calculate_perimeter())
```

```
28
22
```

Classes and Objects

❖ Class - Constructor

Note: Not all attributes have to be initialized in the __init__() method. Attributes can be created in other methods.

```
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        self.area = self.width * self.height
        return self.area

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```
my_rec = Rectangle(4, 7)
```

```
print(my_rec.calculate_area())
print(my_rec.calculate_perimeter())
```

```
28
22
```

```
# Check the attributes
print(vars(my_rec))
```

```
{'width': 4, 'height': 7, 'area': 28}
```

```
{'width': 4, 'height': 7}
```

Classes and Objects

❖ Another approach to declaring a class

```
class Rectangle:
    width = 6
    height = 8
```

```
my_rec = Rectangle()
print(my_rec.width)
print(my_rec.height)
```

```
6
8
```

```
your_rec = Rectangle()
print(your_rec.width)
print(your_rec.height)
```

```
6
8
```

How to customize the values of the constants in the class?

```
your_rec.width = 16
your_rec.height = 18
print(your_rec.width)
print(your_rec.height)
```

```
16
18
```

```
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        return self.width * self.height

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```
my_rec = Rectangle(4, 7)
print(my_rec.calculate_area())
print(my_rec.calculate_perimeter())
```

```
28
22
```

Classes and Objects

❖ Self keyword

What will happen if the __init__ function is used but the self keyword is not?

Can all the variables that appear in the class be considered attributes?

```
class Rectangle:
    def __init__(my_width, my_height):
        width = my_width
        height = my_height

    def calculate_area():
        return width * height

    def calculate_perimeter():
        return (width + height) * 2
```

```
my_rec = Rectangle(4, 7)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-3510747769> in <cell line: 0>()
----> 1 my_rec = Rectangle(4, 7)

TypeError: Rectangle.__init__() takes 2 positional arguments but 3 were given
```

Classes and Objects

❖ Self keyword

The self keyword is used to represent the instance of the class.

Variables prefixed with self are the *attributes* of the class, while others are merely *local variables* of the class.

```
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        return self.width * self.height

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```
my_rec = Rectangle(4, 7)
print(my_rec.calculate_area())
print(my_rec.calculate_perimeter())
```

```
28
22
```

Classes and Objects

❖ Some rules when using self keyword

The self keyword must always be the first parameter in each method.

When call a method, it is not necessary to pass the self variable.

```
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

calc = Calculator()
result_add = calc.add(10, 5)
result_subtract = calc.subtract(10, 5)

print("Addition result:", result_add)
print("Subtraction result:", result_subtract)
```

```
Addition result: 15
Subtraction result: 5
```

Classes and Objects

❖ Replacement for self keyword

Fun fact: We can certainly replace self variable with *another word*. Python automatically interprets the *first parameter* of a method as the instance variable.

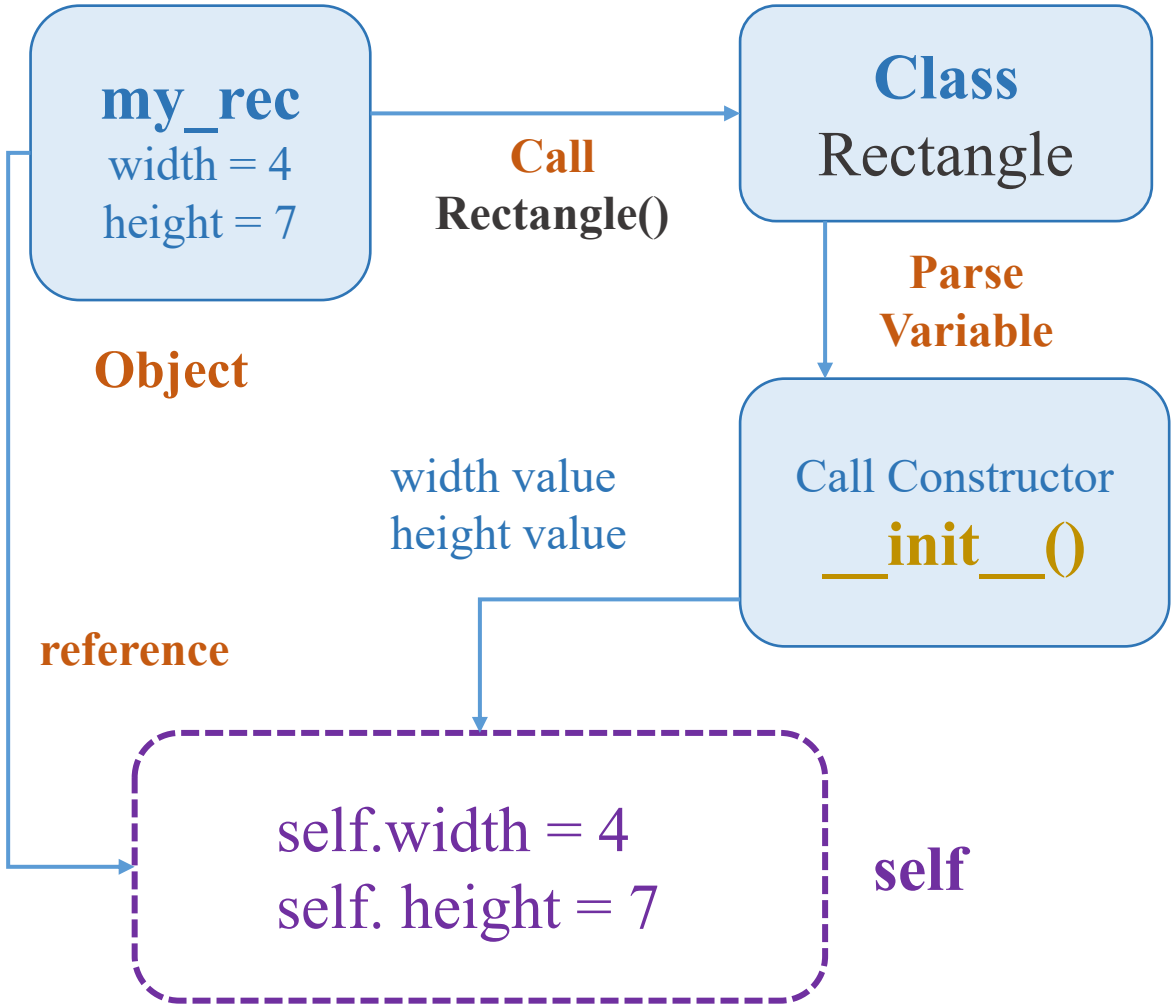
```
class Point:
    def __init__(this, x, y):
        this.x = x
        this.y = y

    def func(this, factor):
        return (this.x + this.y) * factor
```

```
my_point = Point(4, 5)
print(my_point.func(2))
```

```
18
```

Classes and Objects

❖ How we create an object




```python
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

    def calculate_area(self):
        self.area = self.width * self.height
        return self.area

    def calculate_perimeter(self):
        return (self.width + self.height) * 2
```

```python
my_rec = Rectangle(4, 7)
```

Classes and Objects

❖ The special function: __call__( ) method

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

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

    def __call__(self):
        print(f"Hello, {self.name}!")

greet = Greeting("Alice")
greet()
```

```
Hello, Alice!
```

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

    def __call__(self, greeting):
        return f"{greeting}, {self.name}!"

greet = Greeting("Alice")

print(greet("Hello"))
print(greet("Good morning"))
```

```
Hello, Alice!
Good morning, Alice!
```

Naming Conventions

❖ Survey the naming styles of a few popular repos

```python
class Detect(nn.Module):
    # YOLOv5 Detect head for detection models
    stride = None  # strides computed during build
    dynamic = False  # force grid reconstruction
    export = False  # export mode

    def __init__(self, nc=80, anchors=(), ch=(), inplace=True):
        """Initializes YOLOv5 detection layer with specified classes, anchors, channels, and inplace operations."""
        super().__init__()
        self.nc = nc  # number of classes
        self.no = nc + 5  # number of outputs per anchor
        self.nl = len(anchors)  # number of detection layers
        self.na = len(anchors[0]) // 2  # number of anchors
        self.grid = [torch.empty(0) for _ in range(self.nl)]  # init grid
        self.anchor_grid = [torch.empty(0) for _ in range(self.nl)]  # init anchor grid
        self.register_buffer("anchors", torch.tensor(anchors).float().view(self.nl, -1, 2))  # shape(nl,na,2)
        self.m = nn.ModuleList(nn.Conv2d(x, self.no * self.na, 1) for x in ch)  # output conv
        self.inplace = inplace  # use inplace ops (e.g. slice assignment)
```

https://github.com/ultralytics/yolov5/blob/master/models/yolo.py

Naming Conventions

❖ Survey the naming styles of a few popular repos

```python
class GaussianDiffusion(Module):
    def __init__(
        self,
        model,
        *,
        image_size,
        timesteps = 1000,
        sampling_timesteps = None,
        objective = 'pred_v',
        beta_schedule = 'sigmoid',
        schedule_fn_kwargs = dict(),
        ddim_sampling_eta = 0.,
        auto_normalize = True,
        offset_noise_strength = 0.,  # https://www.crosslabs.org/blog/diffusion-with-offset-noise
        min_snr_loss_weight = False, # https://arxiv.org/abs/2303.09556
        min_snr_gamma = 5
    ):
        super().__init__()
        assert not (type(self) == GaussianDiffusion and model.channels != model.out_dim)
        assert not hasattr(model, 'random_or_learned_sinusoidal_cond') or not model.random_or_learned_sinusoidal_cond

        self.model = model

        self.channels = self.model.channels
        self.self_condition = self.model.self_condition
```

https://github.com/lucidrains/denoising-diffusion-pytorch/blob/main/denoising_diffusion_pytorch/denoising_diffusion_pytorch.py

Naming Conventions

❖ Survey the naming styles of a few popular repos

```python
class GaussianDiffusion(Module):
    def predict_start_from_noise(self, x_t, t, noise):
        return (
            extract(self.sqrt_recip_alphas_cumprod, t, x_t.shape) * x_t -
            extract(self.sqrt_recipm1_alphas_cumprod, t, x_t.shape) * noise
        )

    def predict_noise_from_start(self, x_t, t, x0):
        return (
            (extract(self.sqrt_recip_alphas_cumprod, t, x_t.shape) * x_t - x0) / \
            extract(self.sqrt_recipm1_alphas_cumprod, t, x_t.shape)
        )

    def predict_v(self, x_start, t, noise):
        return (
            extract(self.sqrt_alphas_cumprod, t, x_start.shape) * noise -
            extract(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape) * x_start
        )

    def predict_start_from_v(self, x_t, t, v):
        return (
            extract(self.sqrt_alphas_cumprod, t, x_t.shape) * x_t -
            extract(self.sqrt_one_minus_alphas_cumprod, t, x_t.shape) * v
        )
```

https://github.com/lucidrains/denoising-diffusion-pytorch/blob/main/denoising_diffusion_pytorch/denoising_diffusion_pytorch.py

Naming Conventions

SuperCat
+ cat_name + cat_color + cat_age
+ get_name( ) + set_name( )

For class names
Including words concatenated
Each word starts with upper case

For attribute names
Use nouns or noun phrases
Words separated by underscores

For method names
Prioritize using verbs or phrasal verbs
Words separated by underscores

```python
class SuperCat:
    def __init__(self, cat_name, cat_color, cat_age):
        self.cat_name = cat_name
        self.cat_color = cat_color
        self.cat_age = cat_age

    def get_name(self):
        return self.cat_name

    def set_name(self, new_name):
        self.cat_name = new_name
```

```python
my_cat = SuperCat("Joey", "White", "2")
print(my_cat.get_name())
```

```
Joey
```

```python
my_cat.set_name("Rachel")
print(my_cat.get_name())
```

```
Rachel
```

Inheritance

Inheritance

❖ Motivation

W Inheritance

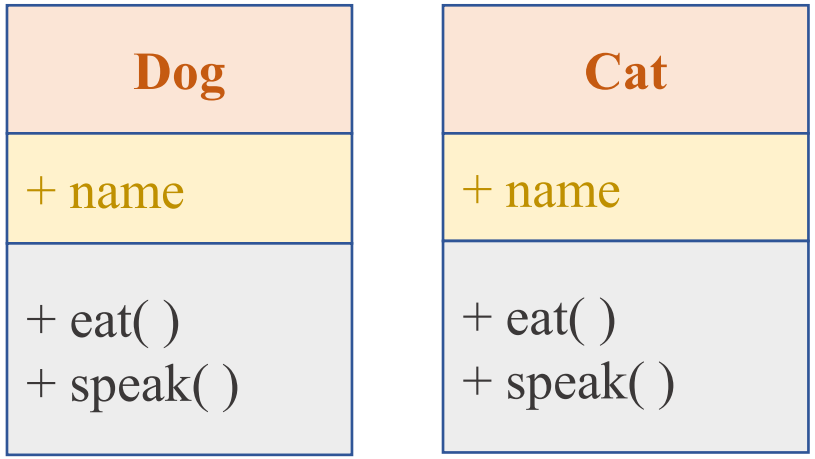
```python
class Animal:
    def __init__(self, name):
        self.name = name

    def eat():
        print("Eating")

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says Woof!")

class Cat(Animal):
    def speak(self):
        print(f"{self.name} says Meow!")
```

Wo Inheritance

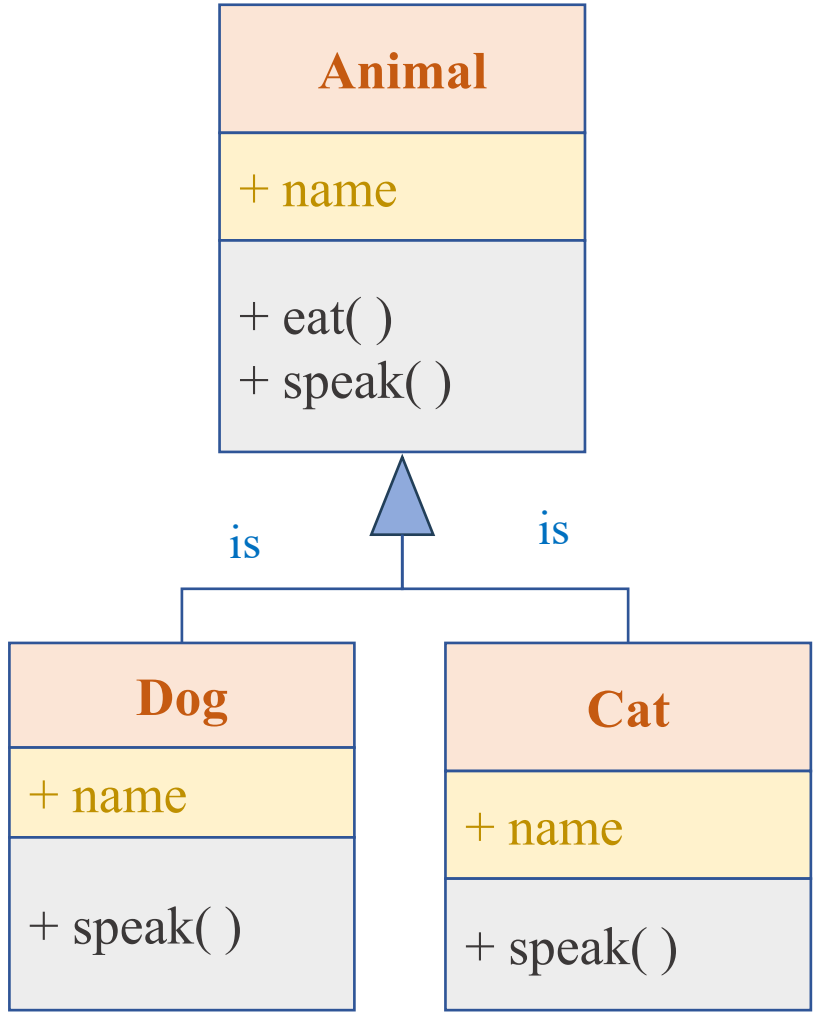
```python
class Dog:
    def __init__(self, name):
        self.name = name

    def eat():
        print("Eating")

    def speak(self):
        print(f"{self.name} says Woof!")

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

    def eat():
        print("Eating")

    def speak(self):
        print(f"{self.name} says Meow!")
```

Inheritance

❖ Motivation



W Inheritance



Wo Inheritance

Inheritance

❖ Some benefits of Inheritance

Some benefits that Inheritance provides, similar to using variables in coding.

➢ Code Reusability

Inheritance allows you to reuse previously written code segments.

➢ Scalability

You can easily extend the functionality of classes by modifying the SuperClass.

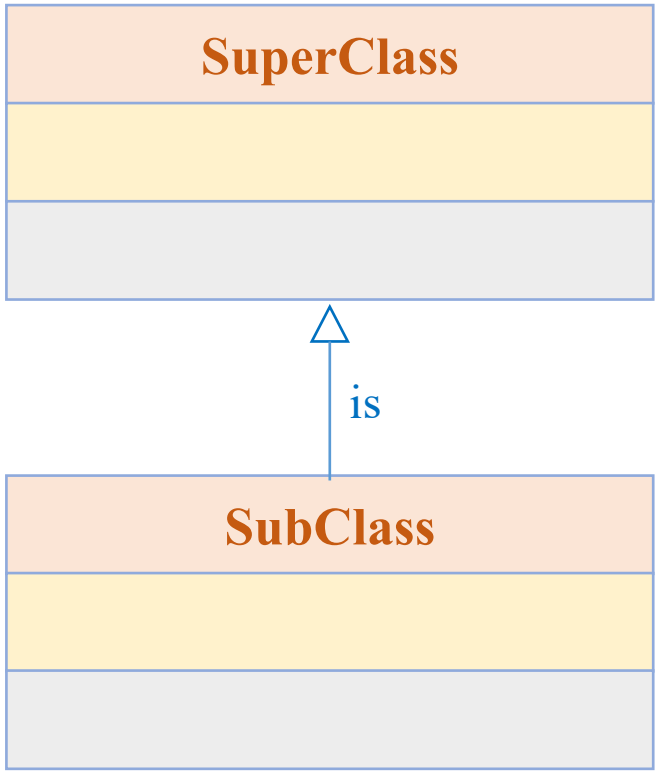
```python
class Animal:
    def __init__(self, name):
        self.name = name

    def eat():
        print("Eating")

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says Woof!")

class Cat(Animal):
    def speak(self):
        print(f"{self.name} says Meow!")
```

Inheritance

❖ Definition and simple syntax

Inheritance is a mechanism in object-oriented programming (OOP) that allows a *new class* to inherit the attributes and methods of an *existing class*.

```
class SuperClass:
    # Attributes and methods of Super Class

class SubClass(SuperClass):
    # Attributes and methods of Sub Class
```

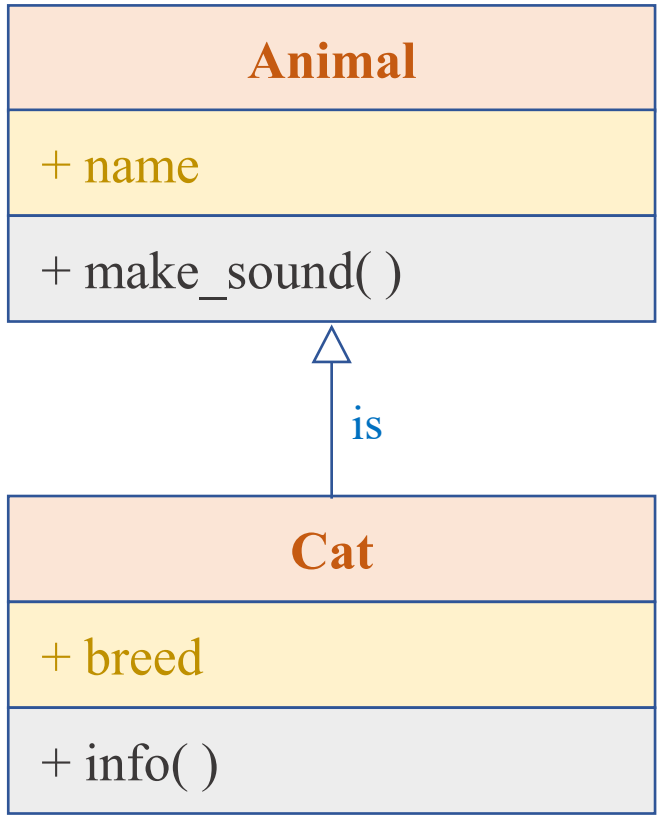


Inheritance

❖ Example

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

    def make_sound(self):
        return "Some generic animal sound"

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

    def info(self):
        return f"{self.name} is a Cat of breed {self.breed}"

my_cat = Cat(name="Joey", breed="Siamese")
print(my_cat.info())
print(my_cat.make_sound())
```

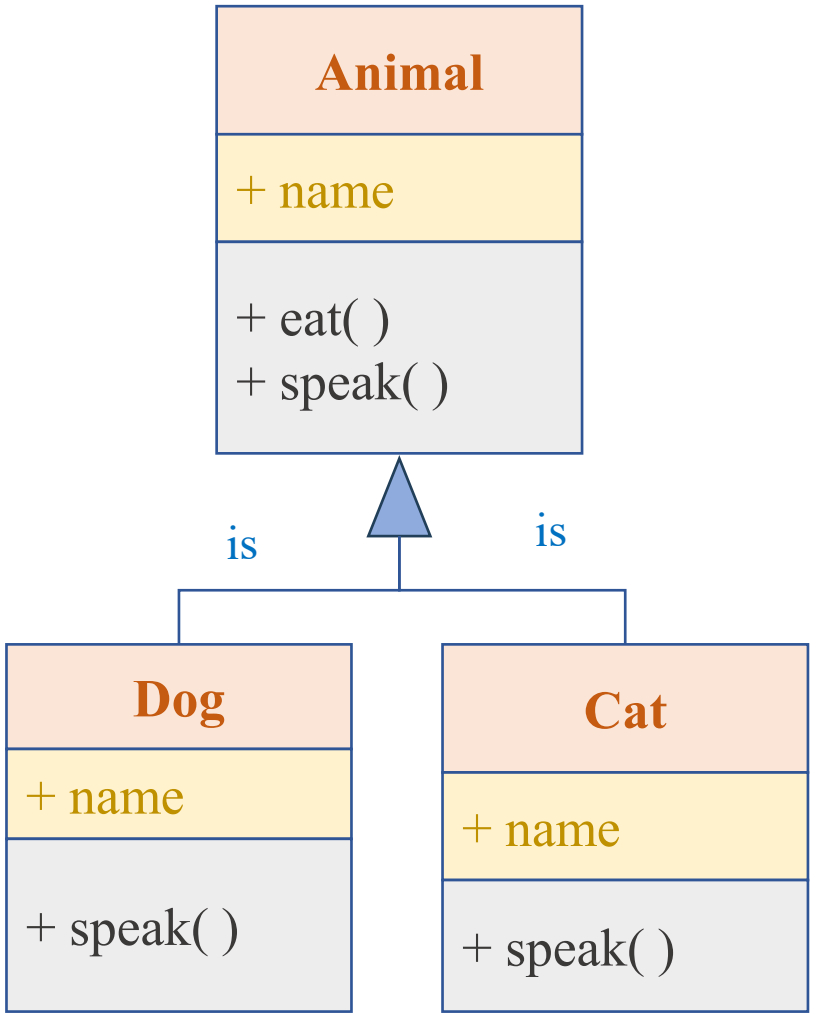
```
Joey is a Cat of breed Siamese
Some generic animal sound
```



Inheritance

❖ Overriding

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

    def eat():
        print("Eating")

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says Woof!")

class Cat(Animal):
    def speak(self):
        print(f"{self.name} says Meow!")
```

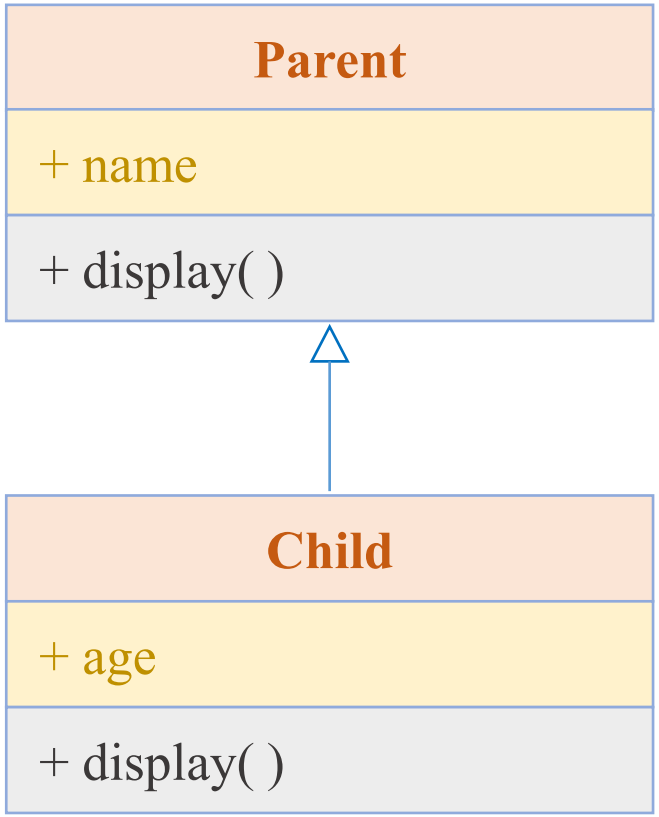


Types of Inheritance

❖ Single Inheritance

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

    def display(self):
        print(f"Parent Name: {self.name}")

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def display(self):
        super().display()
        print(f"Child Age: {self.age}")

# Create object from Child
child = Child("Alice", 20)
child.display()
```

```
Parent Name: Alice
Child Age: 20
```




QUIZ
TIME

Example

❖ **Implement the two classes below**

Math1
+ is_even( ) + factorial( )

Math2
+ is_even( ) + factorial( ) + estimate_euler( )

Example

❖ **Implement the two classes below**

Math1
+ is_even( ) + factorial( )

```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()

# is_even() sample: number=5 -> False
# is_even() 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))
```

```
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( ) + factorial( ) + estimate_euler( )

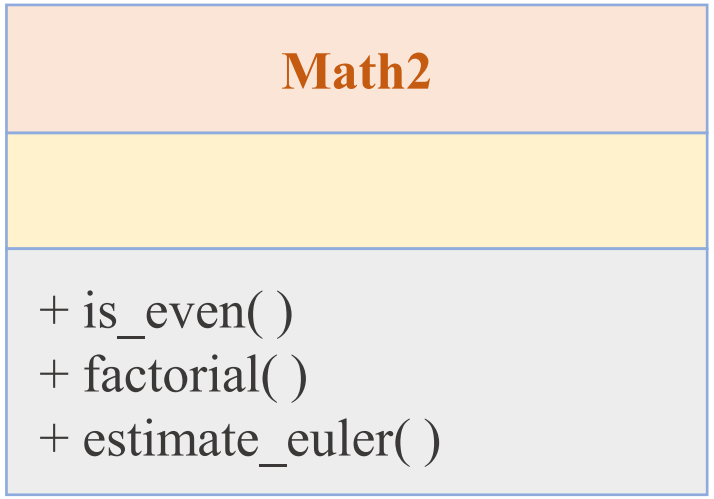
```python
class Math2:
    def is_even(self, number):
        if number%2:
            return False
        else:
            return True

    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()

# is_even() sample: number=5 -> False
# is_even() 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))

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

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

Example

❖ How to reuse an existing class?

Math1

+ is_even( )
+ factorial( )

```
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( )
+ factorial( )
+ estimate_euler( )

```
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
```

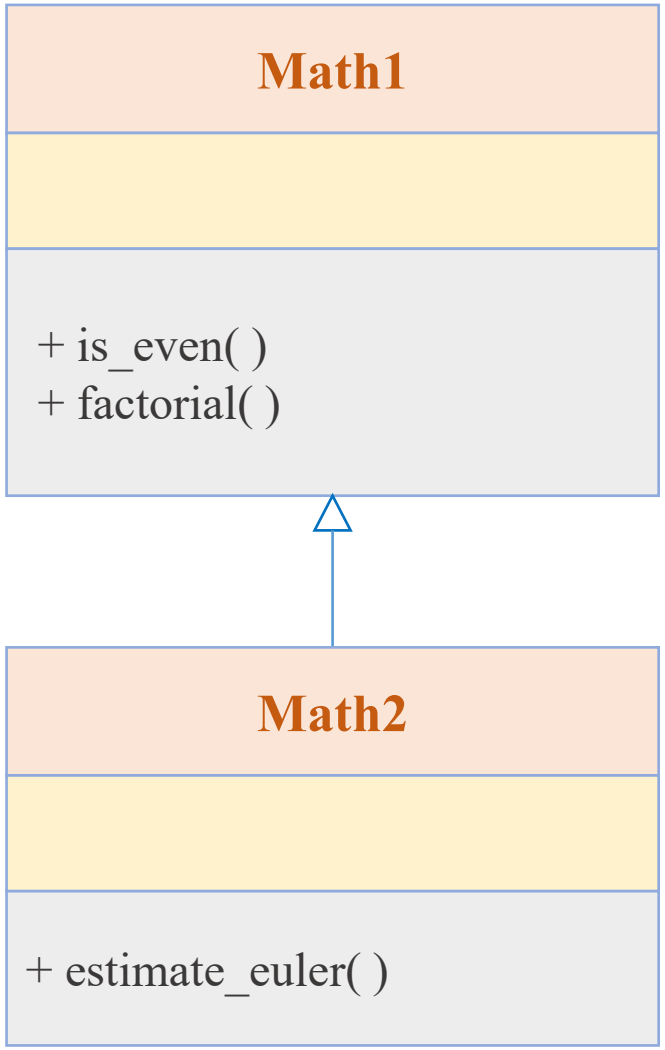
Example

❖ 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()

# is_even() sample: number=5 -> False
# is_even() 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))

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

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

Bonus

Classes and Objects

Fun fact:
In Python, everything is an object.

Therefore, an object of this class can be an *attribute* of another class.

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

    def __call__(self):
        return f"{self.day:02d}/{self.month:02d}/{self.year}"
```

```python
day = 4
month = 1
year = 1643
birth = Date(day, month, year)
print(birth())
```

```
04/01/1643
```

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

    def info(self):
        print(f"Name: {self.name} - Birth: {self.birth()}")
```

```python
name = "Isaac Newton"
birth = Date(4, 1, 1643)
physicist = Person(name, birth)
physicist.info()
```

```
Name: Isaac Newton - Birth: 04/01/1643
```

Lists and Classes

```
list_int = [1, 5, 4, 7, 3, 9]
list_int.sort()
print(list_int)
```

```
[1, 3, 4, 5, 7, 9]
```

```
s1 = Square(3)
s2 = Square(8)
s3 = Square(1)
s4 = Square(6)
s5 = Square(5)
```

```
list_squares = [s1, s2, s3, s4, s5]
for square in list_squares:
    square.describe()
```

```
Side is 3
Side is 8
Side is 1
Side is 6
Side is 5
```

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

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

    def describe(self):
        print(f"Side is {self.side}")

    def __lt__(self, other):
        return self.side < other.side
```

```
list_squares.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[50], line 1
----> 1 list_squares.sort()

TypeError: '<' not supported between instances of 'Square' and 'Square'
```

Is sorting like ***list*** possible?
If so, what ***criteria*** will it sort by?

Lists and Classes

```
list_int = [1, 5, 4, 7, 3, 9]
list_int.sort()
print(list_int)
```

```
[1, 3, 4, 5, 7, 9]
```

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

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

    def describe(self):
        print(f"Side is {self.side}")

    def __lt__(self, other):
        return self.side < other.side
```

```
s1 = Square(3)
s2 = Square(8)
s3 = Square(1)
s4 = Square(6)
s5 = Square(5)
```

```
list_squares.sort()
```

```
list_squares = [s1, s2, s3, s4, s5]
for square in list_squares:
    square.describe()
```

```
Side is 3
Side is 8
Side is 1
Side is 6
Side is 5
```

Approach 2:

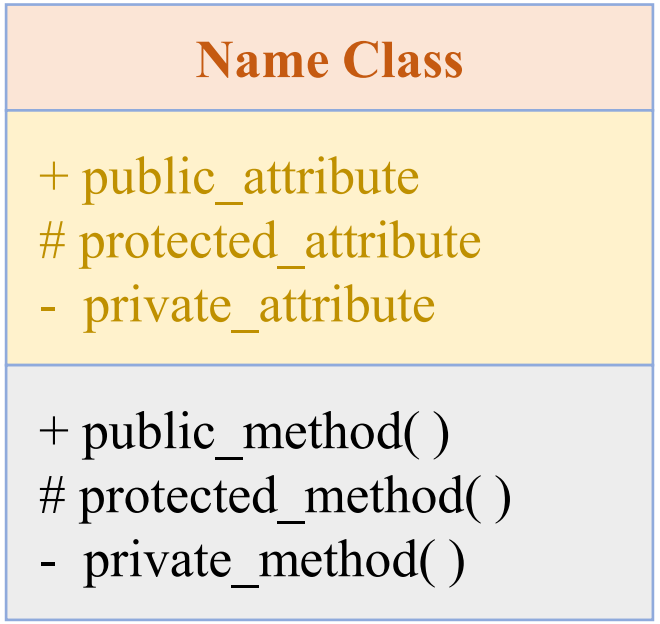
```
list_squares = [s1, s2, s3, s4, s5]
list_squares.sort(key=lambda x: x.side)
for square in list_squares:
    square.describe()
```

```
Side is 1
Side is 3
Side is 5
Side is 6
Side is 8
```

Encapsulation

❖ Access Modifiers

- **Public data**: Accessible anywhere from otside oclass.
- **Private data**: Accessible within the class
- **Protected data**: Accessible within the class and its sub-classes.

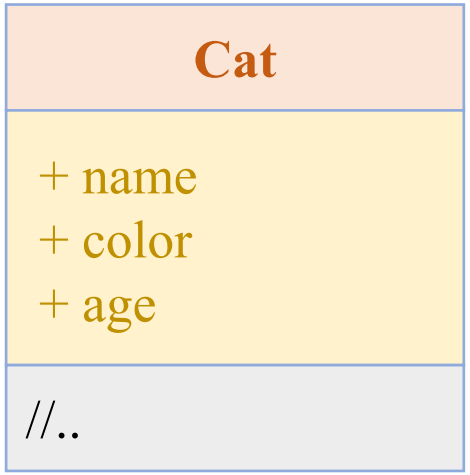


Encapsulation

❖ Access Modifiers: Public

```
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
```



Encapsulation

❖ Access Modifiers: Private

```
class Cat:
    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.__age = age # private

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

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

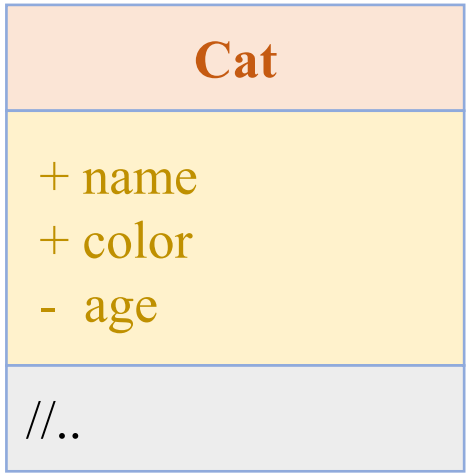
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[57], line 11
      9 print(cat.name)
     10 print(cat.color)
---> 11 print(cat.__age)

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

Encapsulation

❖ Access Modifiers: Private

```
class Cat:
    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.__age = age # private

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

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

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




```
cat.set_age(4)
print(cat.get_age())
```

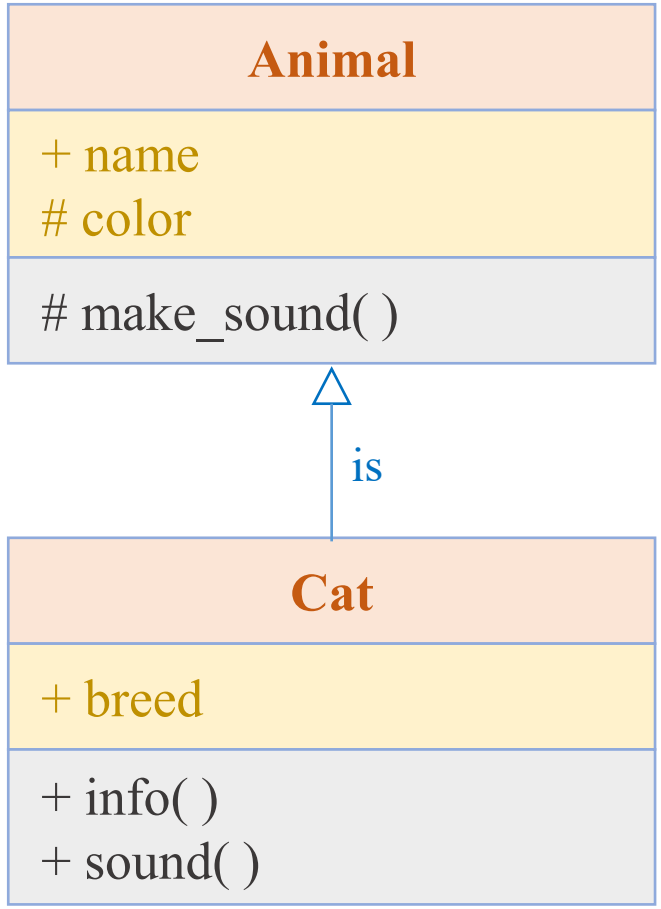
```
4
```

Encapsulation

❖ Access Modifiers: Protected

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

    def _make_sound(self):
        return "Some generic animal sound"

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

    def info(self):
        return f"{self.name} is a {self._color} Cat of breed {self.breed}"

    def sound(self):
        return self._make_sound()

my_cat = Cat(name="Joey", color="white", breed="Siamese")

print(my_cat.info())
print(my_cat.sound())
```

```
Joey is a white Cat of breed Siamese
Some generic animal sound
```

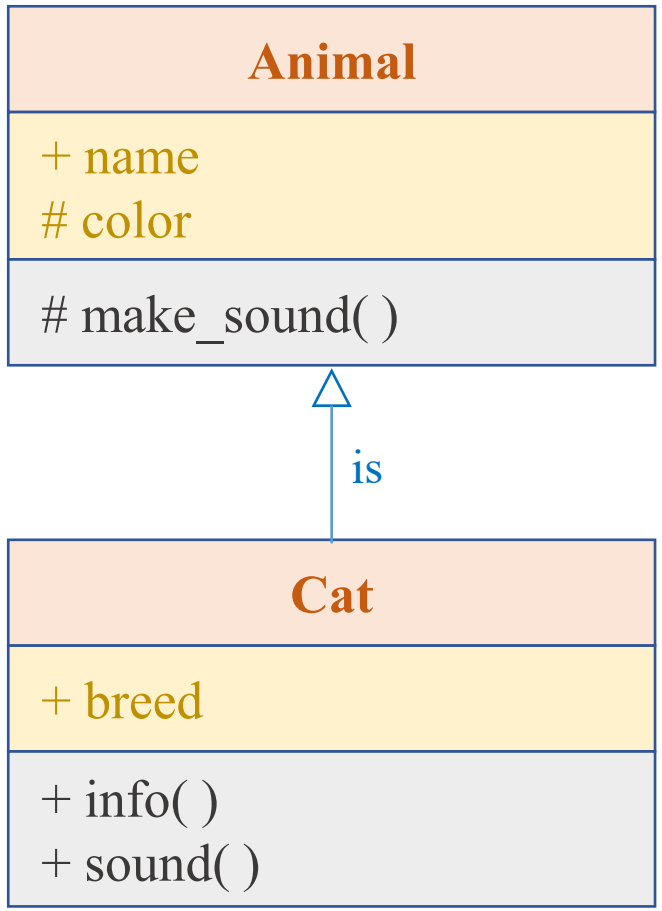


Encapsulation

❖ Access Modifiers: Protected

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

    def _make_sound(self):
        return "Some generic animal sound"

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

    def info(self):
        return f"{self.name} is a {self._color} Cat of breed {self.breed}"

    def sound(self):
        return self._make_sound()

my_cat = Cat(name="Joey", color="white", breed="Siamese")

print(my_cat._color)           # This is allowed but not encouraged.
print(my_cat._make_sound())    # This is allowed but not encouraged.
```

```
white
Some generic animal sound
```



Summary

Classes and Objects

- Class diagram
- Syntax for creating a class and objects
- Constructor **__init__**
- **self** keyword
- Special method **__call__**
- Naming convention
- Other ways to use class

Inheritance

- Definition and syntax
- Override
- Types of inheritance


Thank you!