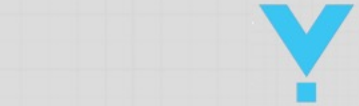
Day03 Presentation Slides

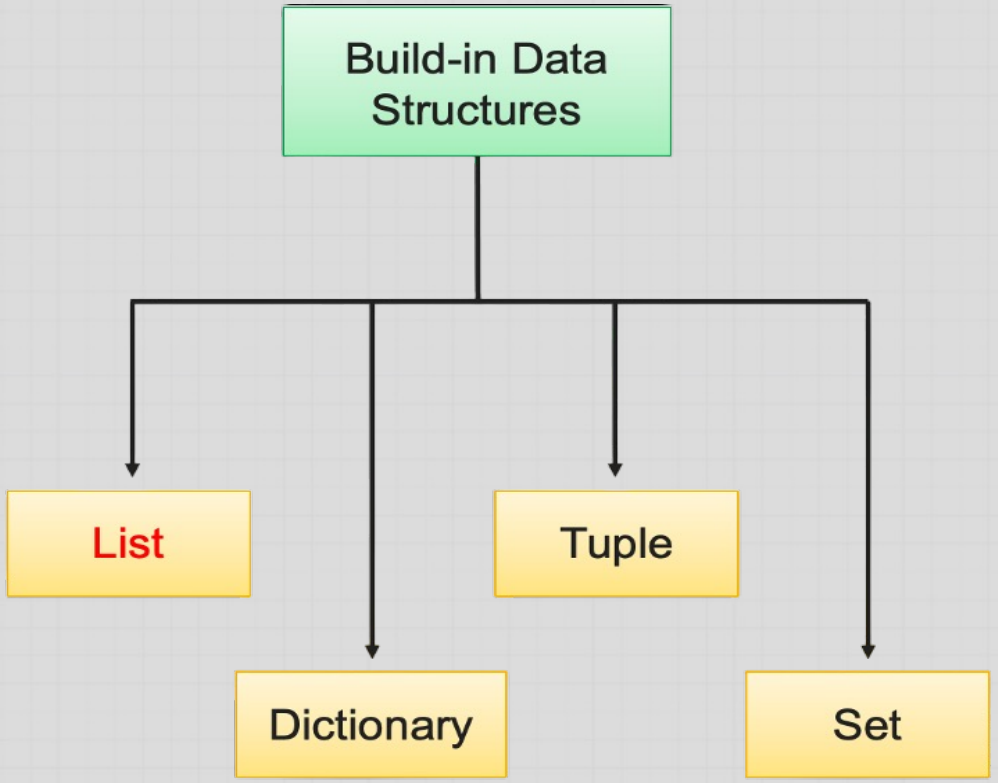
Contents

- List
- List Comprehensions
- Access Modifiers
- Encapsulation
- Inheritance
- Method Overriding
- Abstraction



List

- A special type of variable
- Used to store multiple values of any types
- Size is dynamic, and can be increased/decreased
- The values in the list are changeable
- Every element in a list has 2 index numbers:
  - Forward index
  - Reverse index



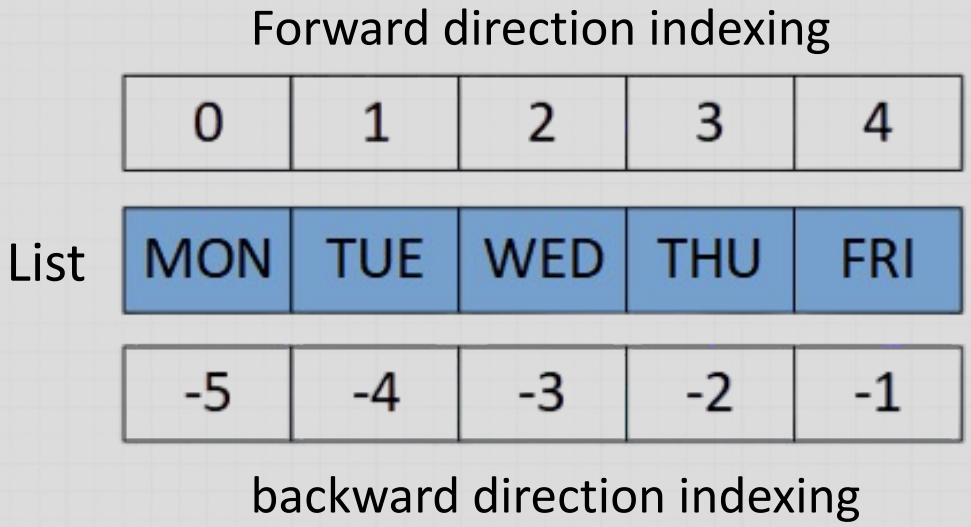
Creating List

- Created by placing all the elements inside square brackets [ ] separated by commas
- Elements are ordered, changeable, can be duplicated, and can be of any data type

```
days = ["MON", "TUE", "WED", "THU", "FRI", "SAT", 'SUN']

fruits = ['Lemon', "Apple", 'Cherry', "Banana", 'strawberry']

numbers = [100, 200, 300, 400, 500, 600, 700]

my_list = ['Lemon', "Apple", 100, 200, "MON", "TUE", True, False]
```

Accessing List Elements

- Elements of a tuple can be accessed by using the square brackets [ ]
- The index number (forward/backward index) of the element needs to be provided




```
days = ["MON", "TUE", "WED", "THU", "FRI"]

print(days[0])  # prints "MON"

print(days[-1]) # prints "FRI"
```

Updating List: Single Element

- Elements in the list are changeable
- We need to give the index number of a specific element within the square bracket and assign a new value to change the element. [index] = new value

```
numbers = [10, 20, 30, 40, 50, 60]

numbers[2] = 300
# Element at index 2 is updated
```

```
names = ["James", "Muhtar", "Bella"]

names[1] = "Python"
# Element at index 1 is updated
```

Updating List: Range of Elements

- We need to give the range of indexes of a specified range within the square bracket, and define a list with new elements, then assigned them to the specified range.

[start_index : end_index] = [list of new elements]

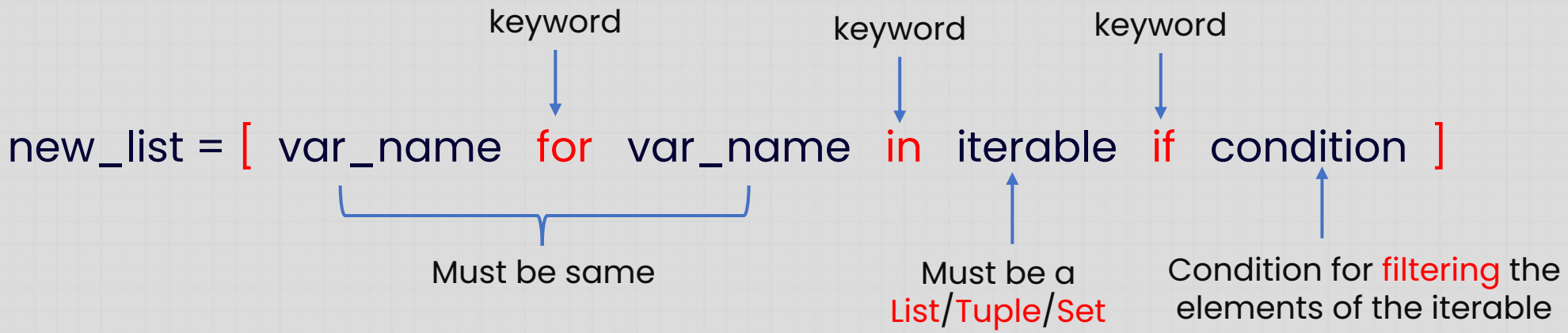
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers[2:5] = [30, 40, 50]
# elements between index 2 to index 5 are updated

print(numbers)
# [1, 2, 30, 40, 50, 6, 7, 8, 9, 10]
```

List Methods

Method Name	Method Name	Method Name
append()	clear()	extend()
sort()	reverse()	copy()
remove()	pop()	insert()
index()	count()	

List Comprehensions

- Used to create a new list based the values of an exiting iterable (list/set/tuple)




```
nums_list = [1, 2, 3, 4, 5, 6, 7, 8]

new_list = list()

for x in nums_list:
    if x > 5:
        new_list.append(x)

print(new_list) #[6, 7, 8]
```

```
nums_list = [1, 2, 3, 4, 5, 6, 7, 8]

new_list = [ x for x in nums_list if x > 5 ]

print(new_list) #[6, 7, 8]
```

Access Modifiers

- There are 3 access modifiers available in Python:
  - public
  - Protected (_)
  - Private (__)

Access Modifier	Same class	Same package	Sub class	Other Packages
public	Yes	Yes	Yes	Yes
Protected (_)	Yes	Yes	Yes	No
Private (__)	Yes	No	No	No

What is Encapsulation

- An object hides its internal data from code that's outside the class
- Hide an attribute by giving private access modifier, and making the methods that access those fields public
- These public methods are called getters & setters (accessor and mutator)

Access modifier	Description
private	When the private access modifier is applied to a class member, the member can not be accessed by code outside the class.
public	When the public access modifier is applied to a class member, the member can be accessed by code inside the class or outside.

Making the attributes private

- The private access modifier can be given by giving two underscores (__) to the attributes to make them inaccessible from outside the class

```
class Employee:

    def __init__(self):
        self.__name = None
        self.__age = None
        self.__salary = None
        self.__job_title = None
```

Getters & Setters

- The getter is used for reading the private data (instance variable) only
- The setter is used for writing (modifying) the private data (instance variable) only
- Allows the class to have complete control over what is stored in its fields

Getter Method

- A public instance method that returns the private instance variable's value
- Does not pass any parameter

```
class Employee:

    def __init__(self):
        self.__name = None

    def get_name(self):
        return self.__name
```

Setter Method

- A public instance method that does not return any value
- Passes a parameter, and assigns the given argument to the private attributes

```
class Employee:

    def __init__(self):
        self.__name = None

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

Encapsulation Example

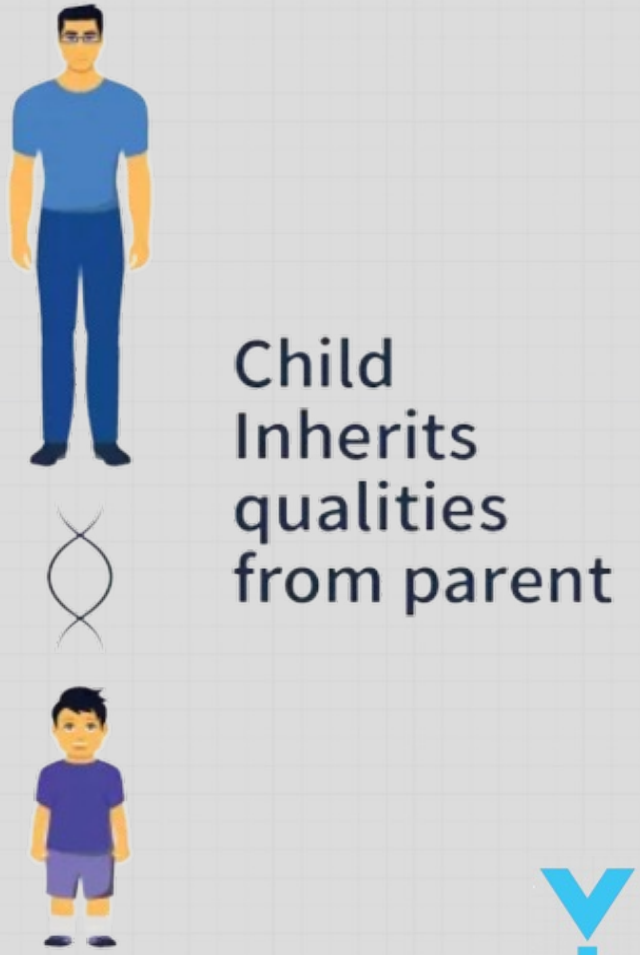
- Attributes of Person class objects can only be accessed or modified by getters and setters

```
class Employee:

    def __init__(self):
        self.__name = None
        self.__salary = None

    def get_name(self):
        return self.__name

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

    def get_salary(self):
        return self.__salary

    def set_salary(self, salary):
        self.__salary = salary
```

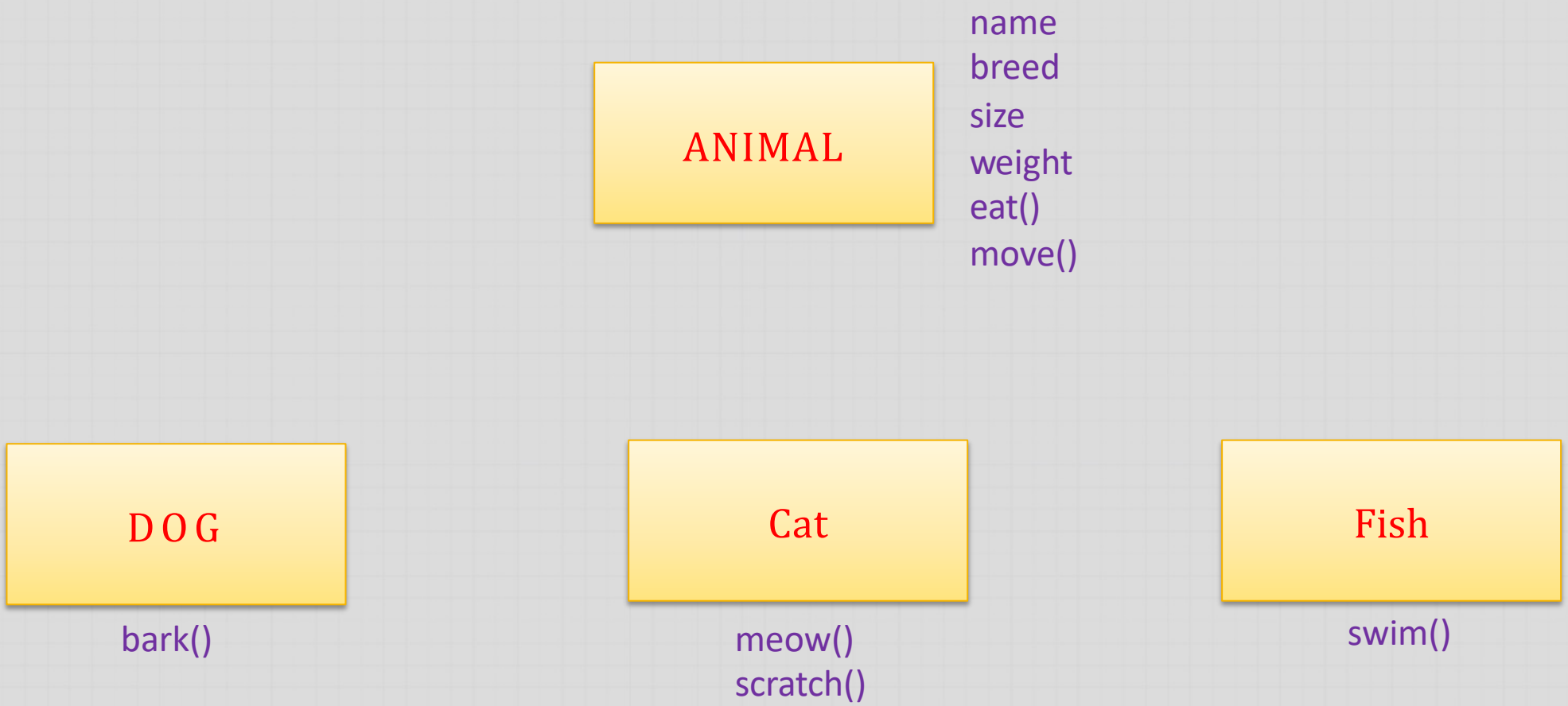
```
employee1 = Employee()

employee1.set_name('James')
employee1.set_salary(100_000)

print(employee1.get_name())
print(employee1.get_salary())
```

Inheritance

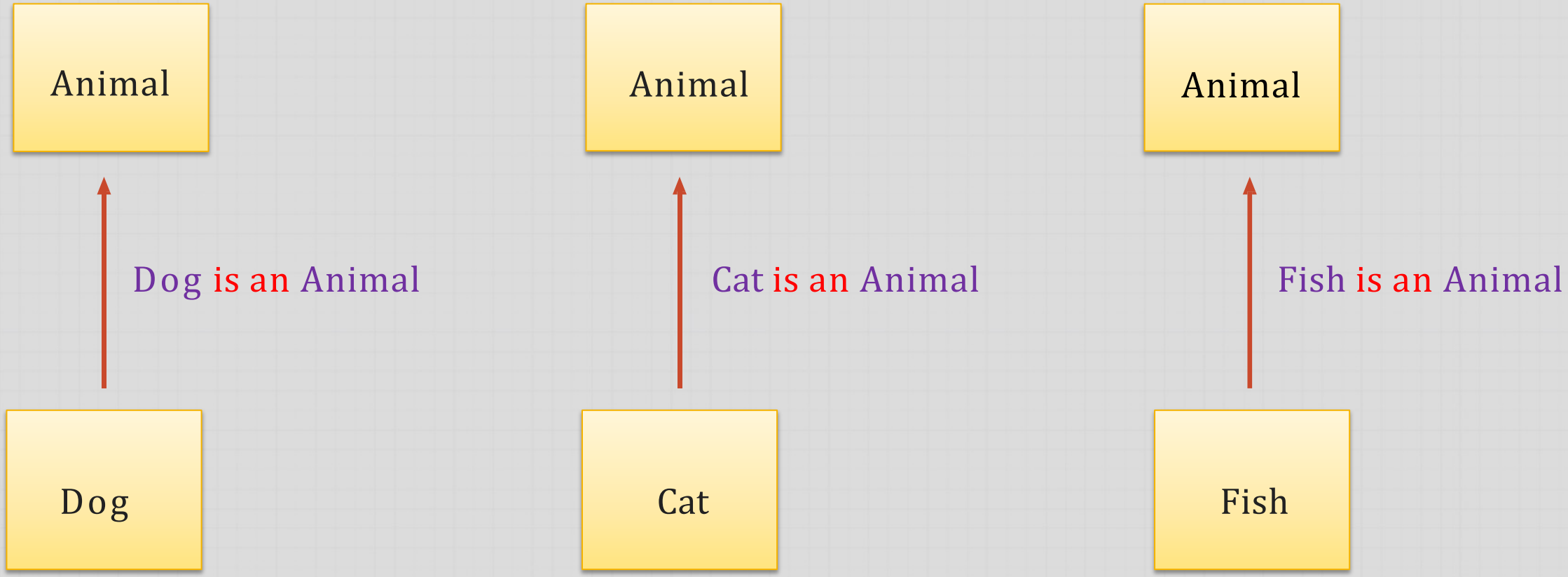
- Used for creating Is A relationship among the classes
- Allows one class to inherit the variables and methods from other class(es)



Inheritance



Inheritance

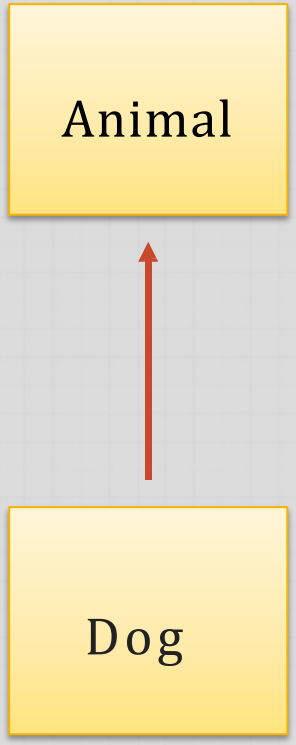


Inheritance

- One or more classes can be inherited by a class by specifying the parent class(es) in parentheses after the class name

```
class Animal:
    pass


class Dog(Animal):
    pass


class Cat(Animal):
    pass
```

Inheritance



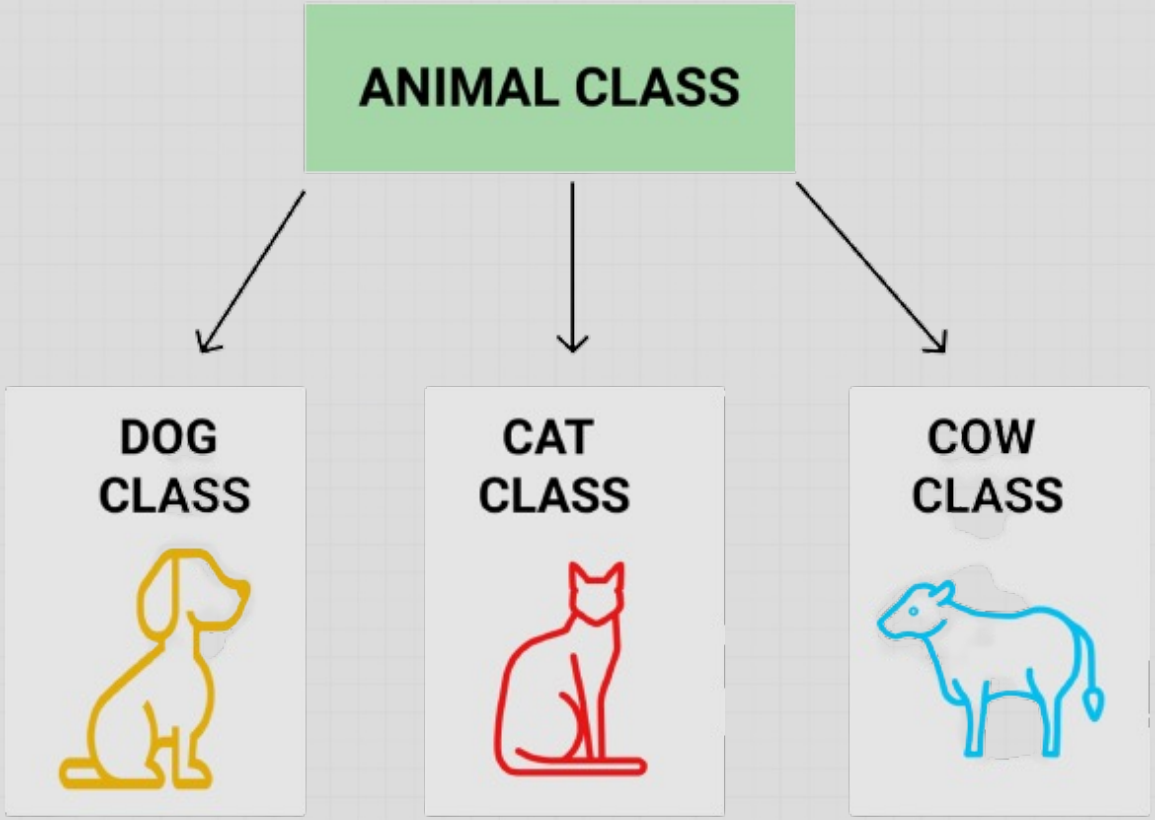
The animal is called SUPER class and the Dog is called SUB class

OR

The animal is called PARENT class and the Dog is called CHILD class

What is Inherited to Sub Classes

- All the accessible variables & methods
- Private variables and methods are not inherited
- Constructors are not inherited



Super keyword

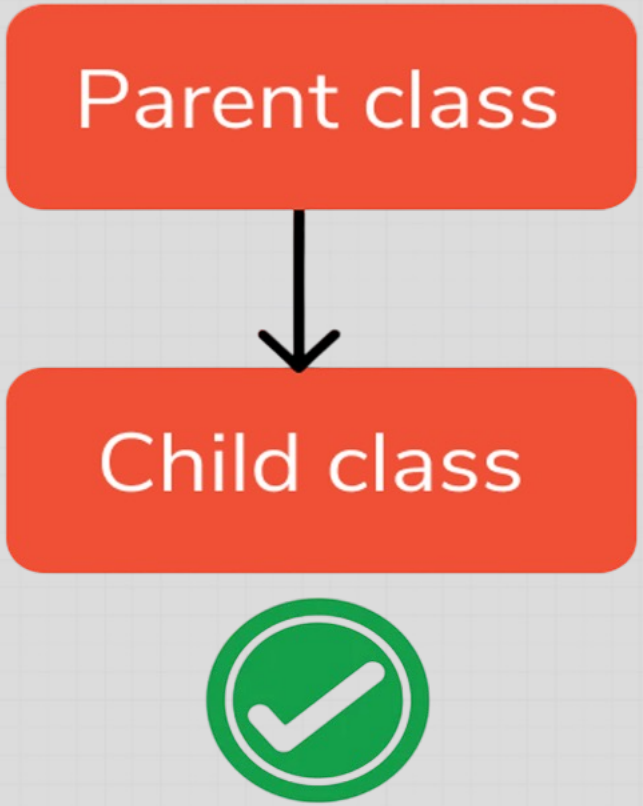
- The super function refers to an object's superclass (Parent). We can use super() to call a super class's constructor and methods

```
class Person:


    def __init__(self, name, gender, age):
        self.name = name
        self.gender = gender
        self.age = age



class Employee(Person):

    def __init__(self, name, gender, age, salary, job_title):
        super().__init__(name, gender, age)
        self.salary = salary
        self.job_title = job_title
```

Accesses the parent class members

Single Inheritance

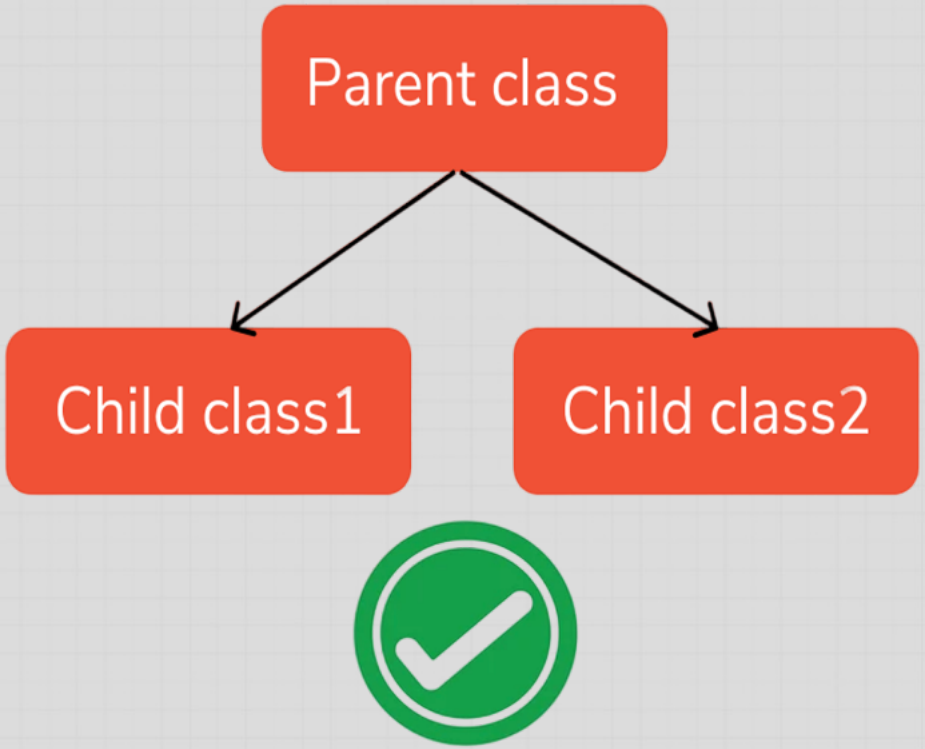
- Single Inheritance: Subclasses inherit the features of one super class




```
class Person:
    pass


class Employee(Person):
    pass
```

Hierarchical Inheritance

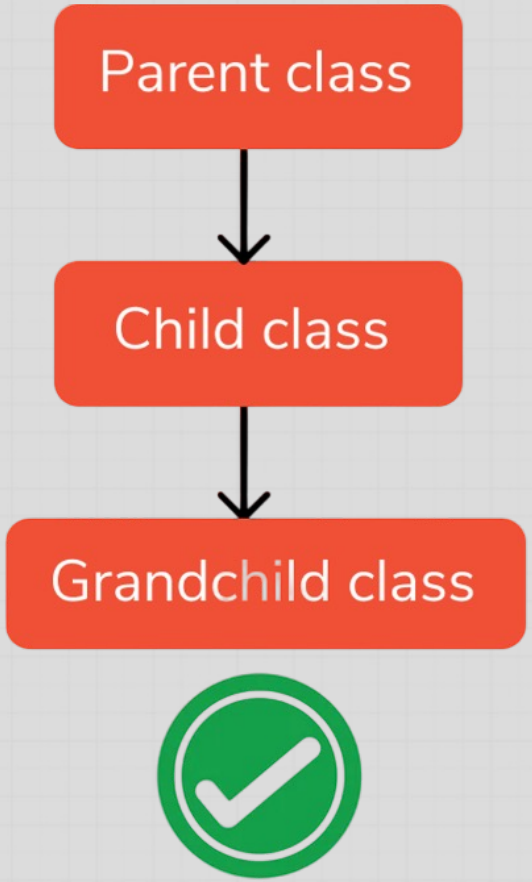
- Hierarchical Inheritance: One class serves as a superclass for more than one subclass




```
class Person:
    pass


class Employee(Person):
    pass


class Student(Person):
    pass
```

Multilevel Inheritance

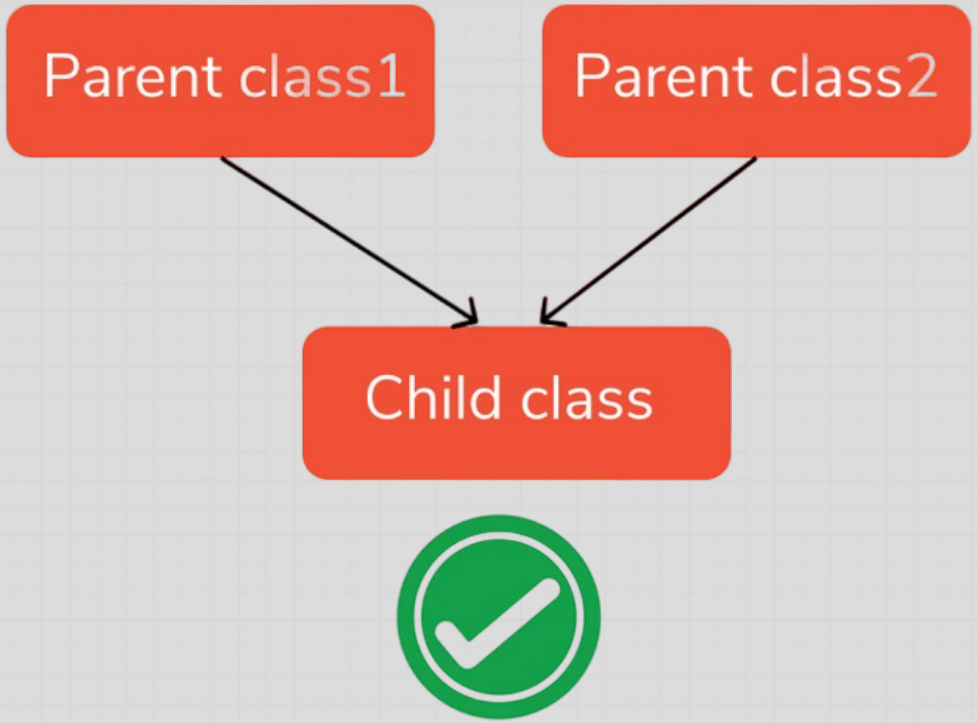
- Multilevel Inheritance: Sub class inherits from a superclass and the sub-class also acts as the Super Class to the other class




```
class Person:
    pass


class Employee(Person):
    pass


class Teacher(Employee):
    pass
```

Multiple Inheritance

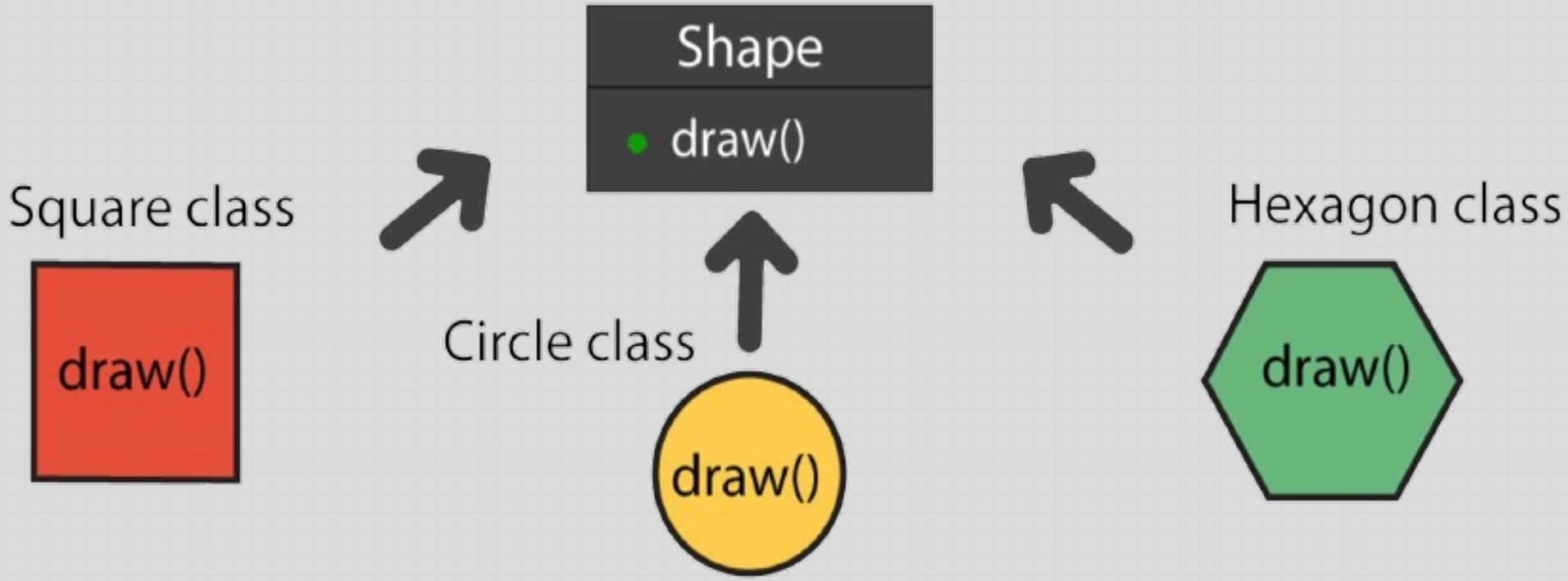
- Multiple Inheritance: One class can have more than one superclass and inherit features from all parent classes



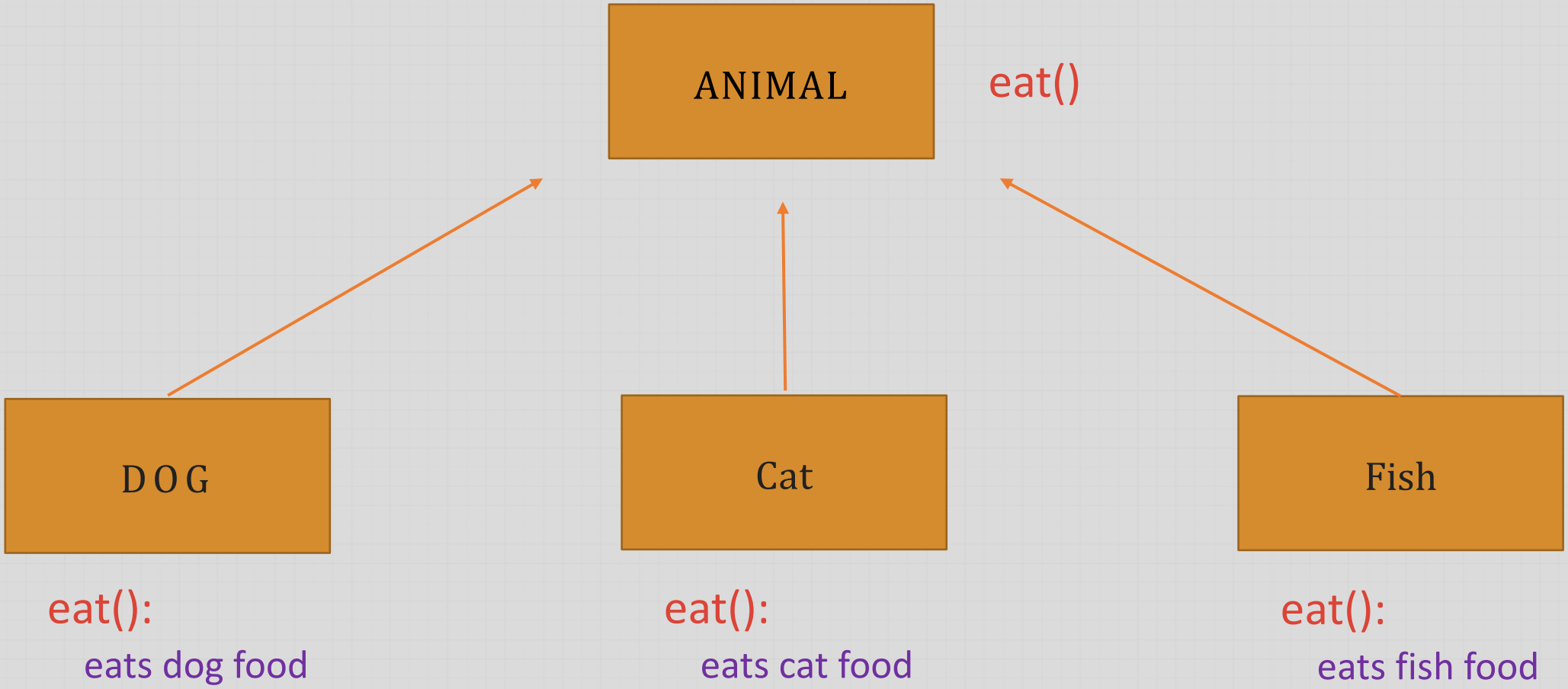

```
class Person:
    pass


class Employee(Person):
    pass


class Developer(Employee, Programmer):
    pass
```

Method Overriding

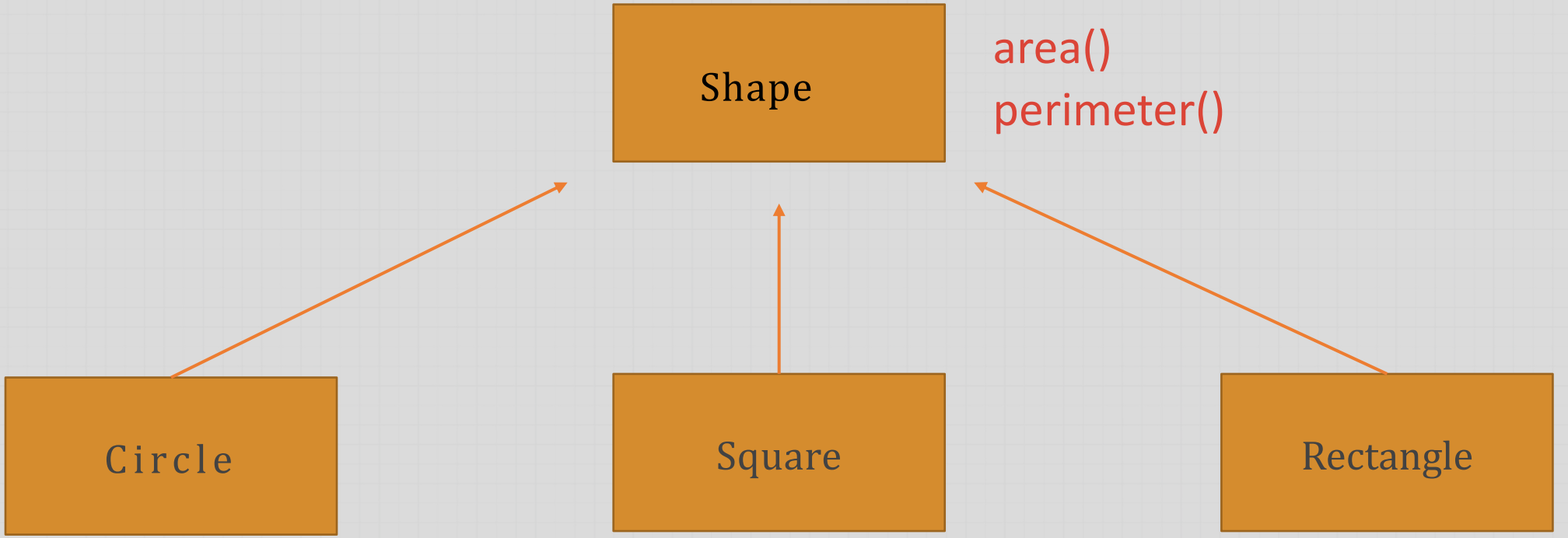
- Giving different implementations to the method
- Must take place in a sub class
- Less memory usage and improves the reusability



Method Overriding Example 1



Method Overriding Example 2



area():
radius * radius * pi

perimeter():
2 * radius * pi

area():
side * side

perimeter():
side * 4

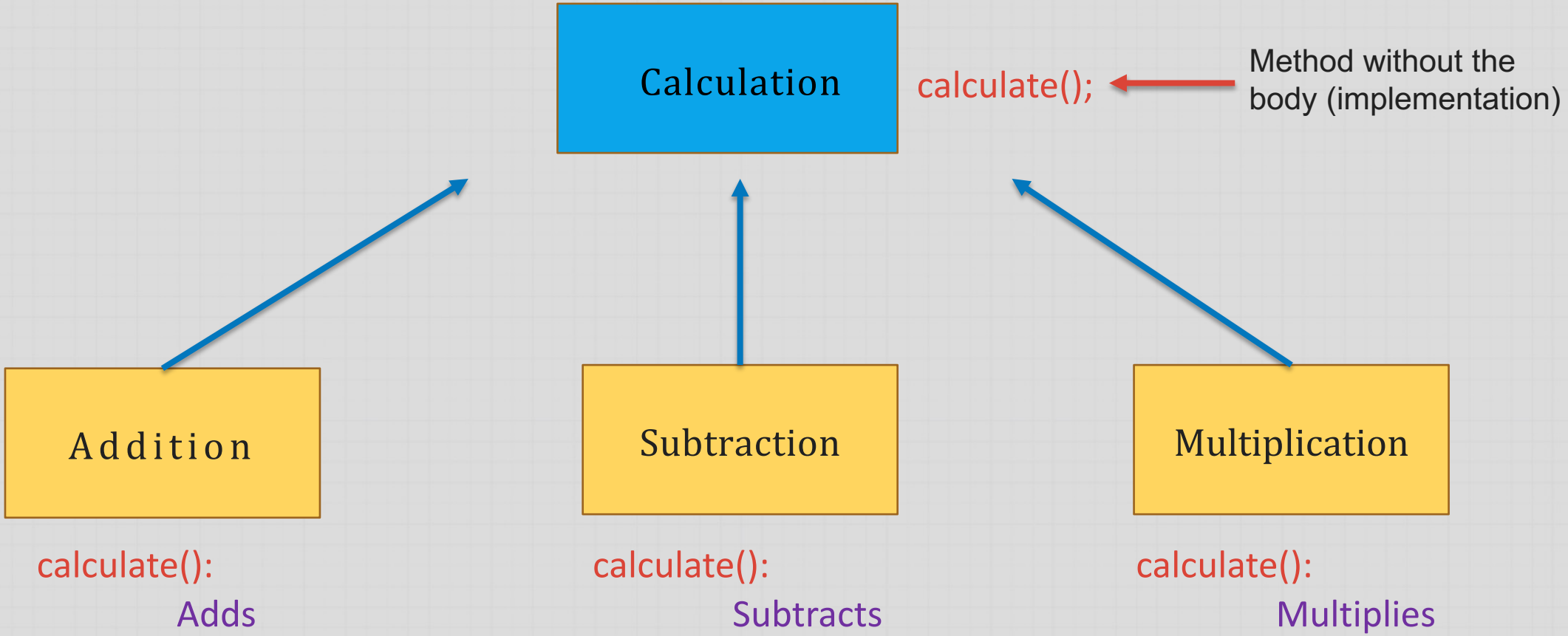
area():
width * length

perimeter():
2 * (width + length)

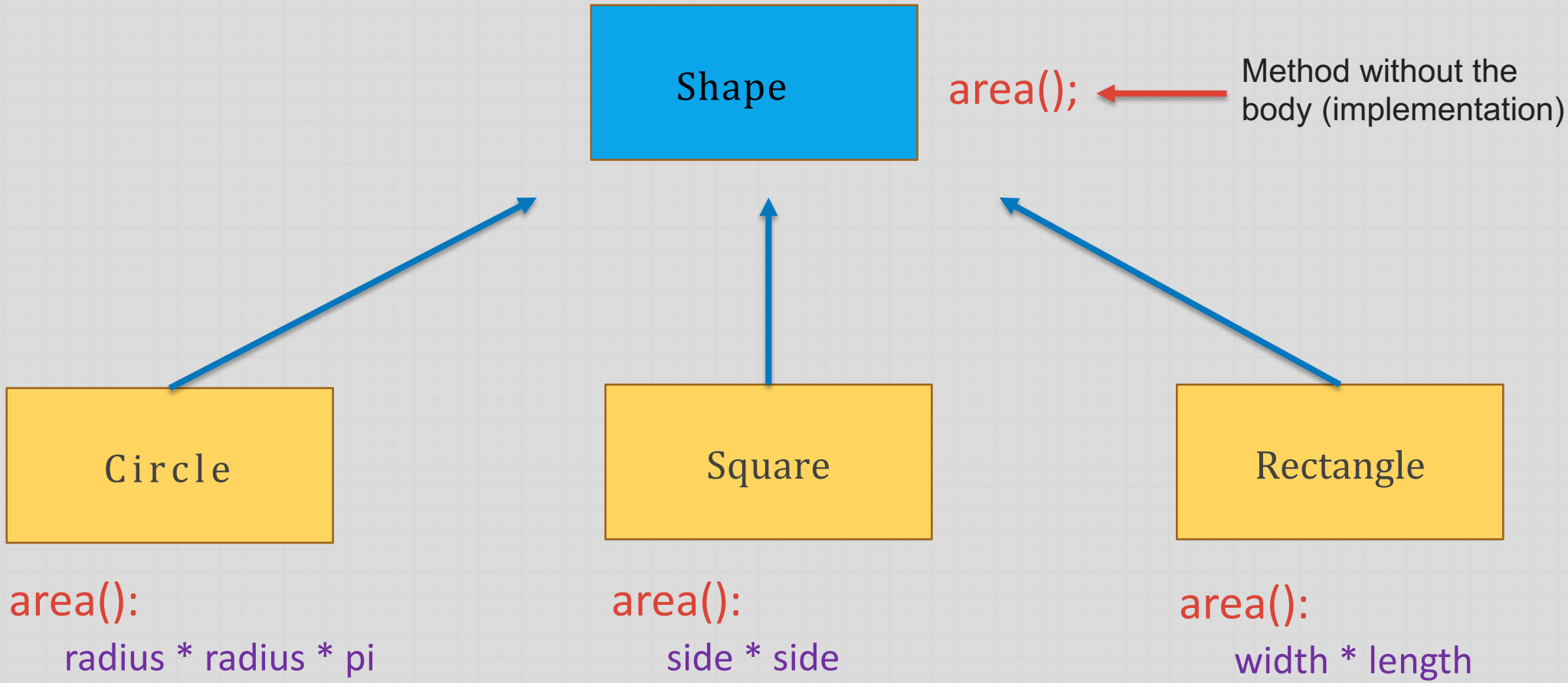
Abstraction

- Process of hiding the implementation details of functions from the user
- Only the functionality will be provided to the user
- The user will have the information on what the object does instead of how it does

Abstraction Example 2



Abstraction Example 3



How to Achieve Abstraction?

- There are two ways to achieve abstraction in python:
  - The Base class: includes common properties and methods of sub classes that will inherit, common methods will be declared without any implementations
  - The Abstract class: a class that inherits the ABC class, and it includes abstract methods that must be implemented by subclass(es). The class can not be instantiated

Abstraction

- We can create methods with no implementation by giving the pass statements in the parent class's methods that will be inherited to the child classes

```python
class Shape:

    def __init__(self):
        self.name = type(self).__name__

    def calc_area(self) -> float:
        pass

    def calc_perimeter(self) -> float:
        pass

    def __str__(self):
        return f'{self.name}{self.__dict__}'
```

No implementation is given to the method

No implementation is given to the method

```python
class Circle(Shape):

    def __init__(self, radius: float):
        super().__init__()
        self.radius = radius

    def calc_area(self) -> float:
        return self.radius ** 2 * math.pi

    def calc_perimeter(self) -> float:
        return self.radius * 2 * math.pi
```

```python
class Square(Shape):

    def __init__(self, side: float):
        super().__init__()
        self.side = side

    def calc_area(self) -> float:
        return self.side * self.side

    def calc_perimeter(self) -> float:
        return self.side * 4
```

Abstraction

- We can inherit the ABC class from abc module to create abstract class to define abstract methods (a method without no implementation and meant to be overridden)that must be implemented to the subclass(es)

```python
from abc import ABC, abstractmethod


class Shape(ABC):
    def __init__(self):
        self.name = type(self).__name__

    @abstractmethod
    def area(self) -> float:
        pass

    @abstractmethod
    def perimeter(self) -> float:
        pass

    def __str__(self):
        return f'{self.name}{self.__dict__}'
```

No implementation is given to the method

No implementation is given to the method

```python
class Circle(Shape):

    def __init__(self, radius: float):
        super().__init__()
        self.radius = radius

    def calc_area(self) -> float:
        return self.radius ** 2 * math.pi

    def calc_perimeter(self) -> float:
        return self.radius * 2 * math.pi
```

```python
class Square(Shape):

    def __init__(self, side: float):
        super().__init__()
        self.side = side

    def calc_area(self) -> float:
        return self.side * self.side

    def calc_perimeter(self) -> float:
        return self.side * 4
```