# Inheritance

**Instructor:** Paruj Ratanaworabhan

**Sources:**

- MIT course 6.0001
- w3schools.com

# Quick recap on OOP

# WHY USE OOP AND CLASSES OF OBJECTS?

- mimic real life
- group different objects part of the same type

Jelly
1 year old
brown

5 years old
brown

Tiger
2 years old
brown

Bean
0 years old
black

2 years old
white

1 year old
b/w

Image Credits, clockwise from top: Image Courtesy Harald Wehner, in the public Domain. Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License CC-BY

# WHY USE OOP AND CLASSES OF OBJECTS?

- mimic real life
- group different objects part of the same type

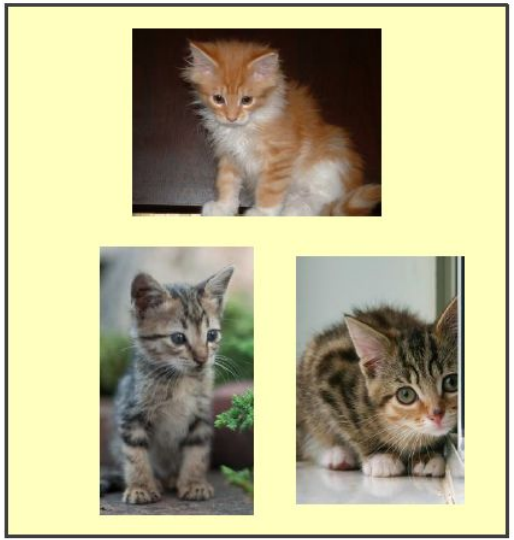


Image Credits, clockwise from top: Image Courtesy Harald Wehner, in the public Domain. Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License CC-BY

# A class in an OO program

E.g., a Circle class

What are some state variables associated with that class?

State variables: (also referred to as attributes)

Location: (x, y)

Radius: r

Color: (r, g, b)

What are some functions to manipulate the state variables?

Functions: (also referred to as operations)

def draw(location, radius, color)

def move(location)

# Quick recap on information hiding

# INFORMATION HIDING

- author of class definition may **change data attribute** variable names

replaced age data attribute by years

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- if you are **accessing data attributes** outside the class and class **definition changes**, may get errors

# PYTHON NOT GREAT AT INFORMATION HIDING

- allows you to **access data** from outside class definition

```
print(a.age)
```

- allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- it's **not good style** to do any of these!

# Information hiding in Python

```
class Animal(object):
    def __init__(self, age):
        self.__age = age
        self.__name = None
    def get_age(self):
        return self.__age
    def get_name(self):
        return self.__name
    def set_age(self, newage):
        self.__age = newage
    def set_name(self, newname=""):
        self.__name = newname
    def __str__(self):
        return "animal:"+str(self.__name)+":"+str(self.__age)

print("\n---- animal tests ----")
a = Animal(4) a.__age X
print(a)
print(a.get_age())
a.set_name("fluffy")
print(a)
a.set_name()
print(a)
```

A non-Pythonic way

```
class Animal(object):
    def __init__(self, age):
        self.__age = age
        self.__name = None
    @property
    def age(self):
        return self.__age
    @property
    def name(self):
        return self.__name
    @age.setter
    def age(self, newage):
        self.__age = newage
    @name.setter
    def name(self, newname):
        self.__name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)

print("\n---- animal tests ----")
a = Animal(4)
print(a)
print(a.age)
a.name = "fluffy"
print(a)
a.name = ""
print(a)
```

A Pythonic way

# Inheritance

# HIERARCHIES

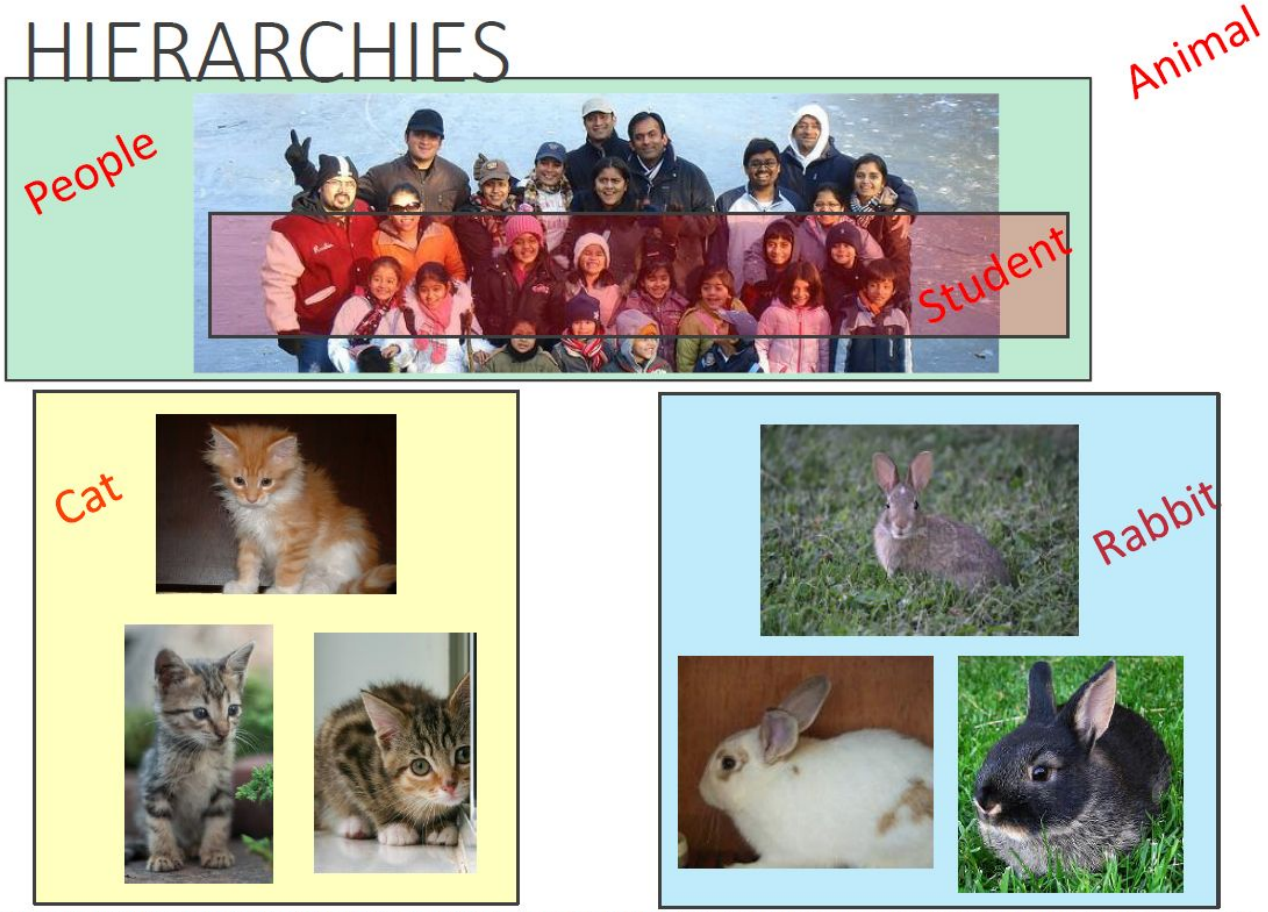
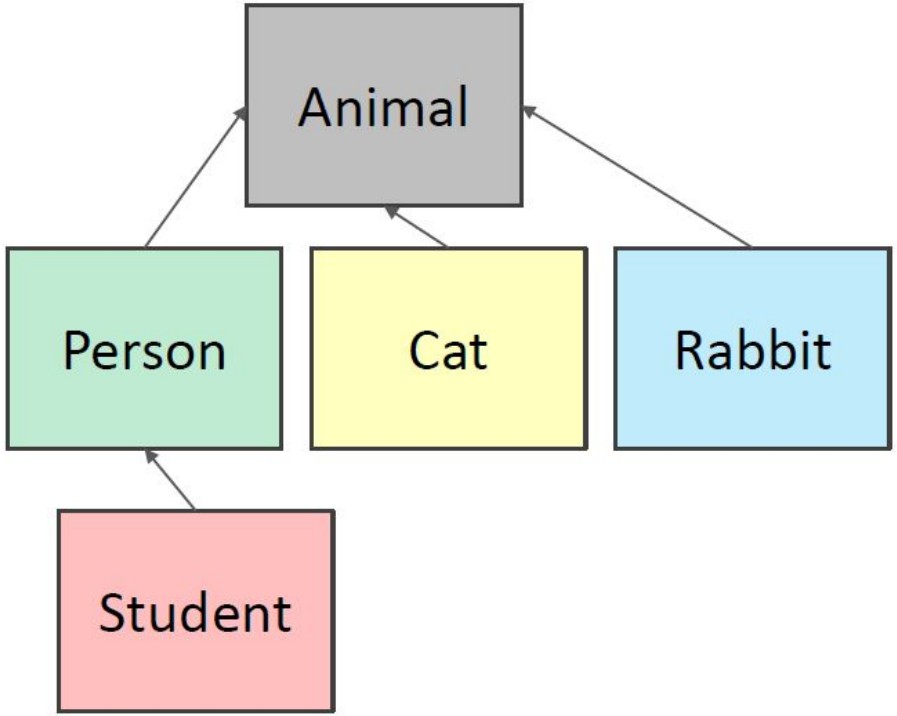



Image Credits, clockwise from top: Image Courtesy Deeeep, CC-BY-NC. Image Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License CC-BY. Courtesy Harald Wehner, in the public Domain.

# HIERARCHIES

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior



# Inheritance: parent class

```
class Animal(object):
    def __init__(self, age):
        self.__age = age
        self.__name = None
    @property
    def age(self):
        return self.__age
    @property
    def name(self):
        return self.__name
    @age.setter
    def age(self, newage):
        self.__age = newage
    @name.setter
    def name(self, newname):
        self.__name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

# Inheritance: subclass

```
class Animal(object):
    def __init__(self, age):
        self.__age = age
        self.__name = None
    @property
    def age(self):
        return self.__age
    @property
    def name(self):
        return self.__name
    @age.setter
    def age(self, newage):
        self.__age = newage
    @name.setter
    def name(self, newname):
        self.__name = newname
    def __str__(self):
        return
"animal:"+str(self.name)+":"+str(self.age)
```

```
class Cat(Animal):
```

Inherits all attributes of Animal

```
def speak(self):
    print("meow")
```

Add a new functionality

```
def __str__(self):
    return "cat:"+str(self.name)+":"+str(self.age)
```

Overrides __str__

- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

# WHICH METHOD TO USE?

- subclass can have **methods with same name** as superclass
- for an instance of a class, look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.name = name
        self.__friends = []
```

Parent class is Animal

Call Animal __init__

Set name attribute explicitly

Add a new attribute

```
    @property
    def friends(self):
        return self.__friends

    def speak(self):
        print("hello")

    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)

    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
```

Add new functionalities via new methods

```
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)+":"+str(self.friends)
```

Overrides __str__ in Animal

# Another inheritance example

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

# In summary

Recap why OOP

Recap class anatomy

Recap information hiding in Python

Learn inheritance

# Code demo