CISC6000 Deep Learning
Python Classes and Objects

Dr. Yijun Zhao

Fordham University

Motivation

Object-Oriented Programming (OOP)

- A programming paradigm that relies on the concept of **classes** and **objects**.
- A **class** is a blueprint for objects with similar properties and behaviors.
  - E.g., Car class, Employee class, Employer class, BankAccount class, .etc.
- An **object** is an instance of a **class**
  - E.g., the car you own, employee John, employer Fordham, Spock's bank account, etc.
- A **class** can have many **objects** (instances).
- OOP languages: C++, JAVA, Python, Scala, Ruby, JADE, Emerald

Motivation

What's in a class's blueprint

- Constructor
  - the magic __init__ function
- Data members
- Methods
- The magic "self" parameter
  - Never pass in "self".
    It is automatic.

```
class Car:
    def __init__(self, year, make, speed):
        self.year_model = year
        self.make = make
        self.speed = speed


    #methods
    def accelerate(self):
        self.speed +=5

    def brake(self):
        self.speed -=5
```

Motivation

What's in a class's blueprint?

- Constructor
  - the magic **__init__** function
- Data members
- Methods
- The magic "self" parameter
  - Never pass in "self".
    It is automatic.

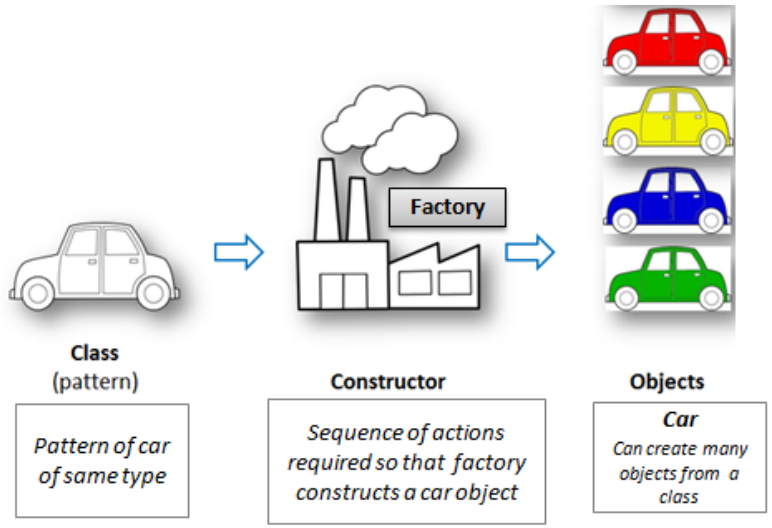
```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    #methods
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Motivation

How to create/use an object?

```
mycar = Car("2020", "Toyota", 0)
mycar.accelerate()
print ("speed = "+ str(mycar.speed))
```

```
speed = 5
```



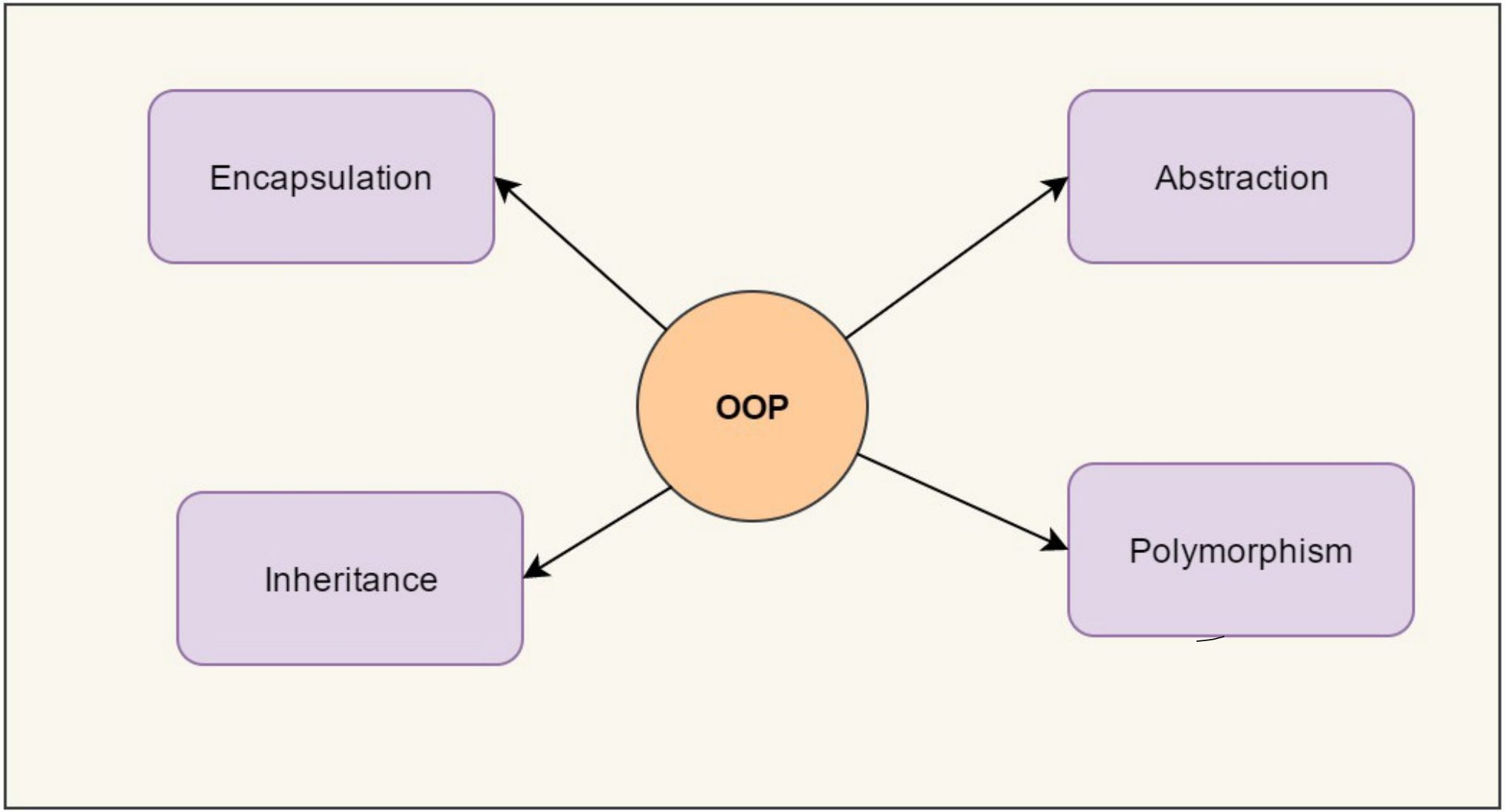

```
class Car:
    def __init__(self, year, make, speed):
        self.year_model = year
        self.make = make
        self.speed = 0

    #methods
    def accelerate(self):
        self.speed +=5

    def brake(self):
        self.speed -=5
```

Motivation

How to create/use an object?

```
>>> spock_account = Account('Spock')
>>> spock_account.deposit(100)
100
>>> spock_account.withdraw(90)
10
>>> spock_account.withdraw(90)
'Insufficient funds'
>>> spock_account.holder
'Spock'
```

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    #methods
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```



Four Pillars/Principles of Objected Oriented Programming

Encapsulation

All important information is contained inside an object and only select information is exposed.

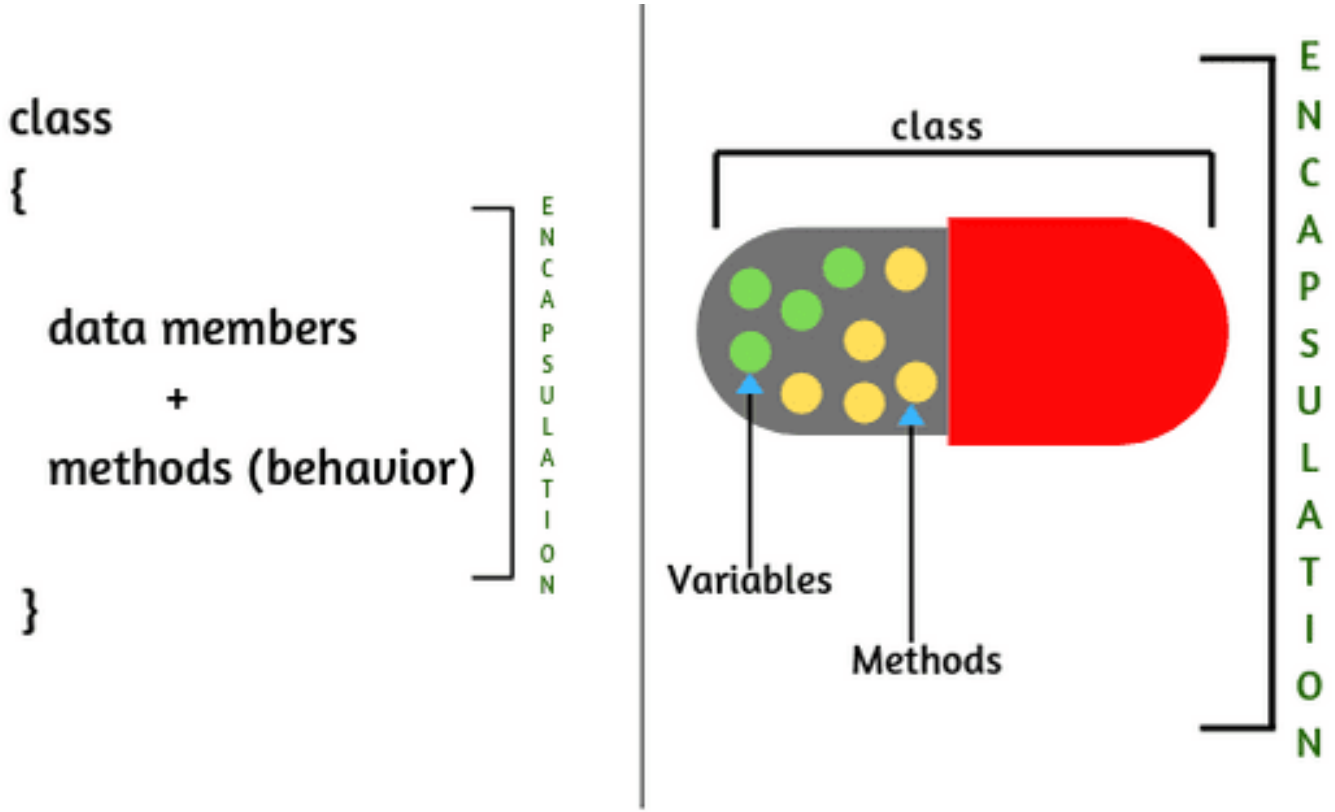


Fig: Encapsulation

Encapsulation

A better Car class with private data members

```
class Car:
    def __init__(self, year, make, speed):
        self.__year_model = year
        self.__make = make
        self.__speed = speed

    def set_year_model(self, year):
        self.__year_model = year

    def set_make(self, make):
        self.__make = make

    def set_speed(self, speed):
        self.__speed = speed

    def get_year_model(self):
        return self.__year_model

    def get_make(self):
        return self.__make

    def get_speed(self):
        return self.__speed
```

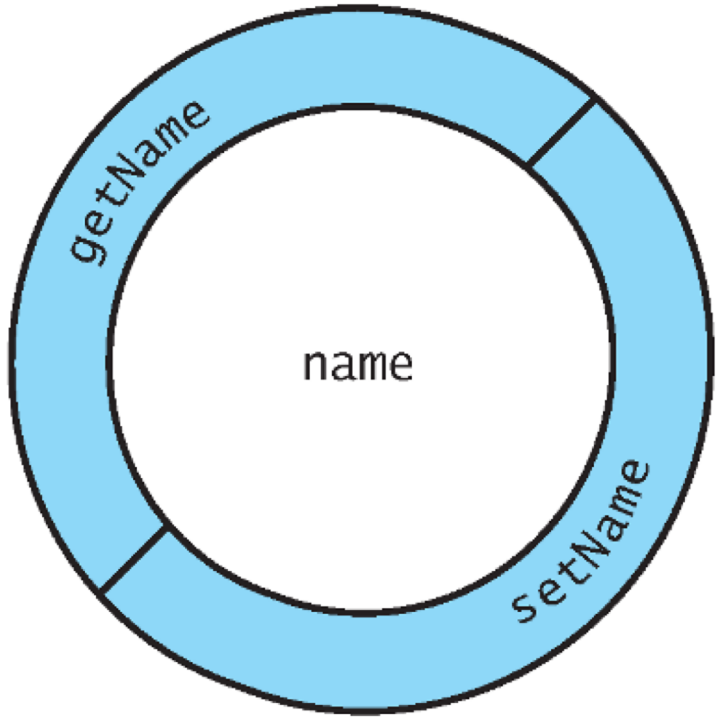
```
    def accelerate(self):
        self.speed +=5

    def brake(self):
        self.speed -=5
```

"__" makes a data member private

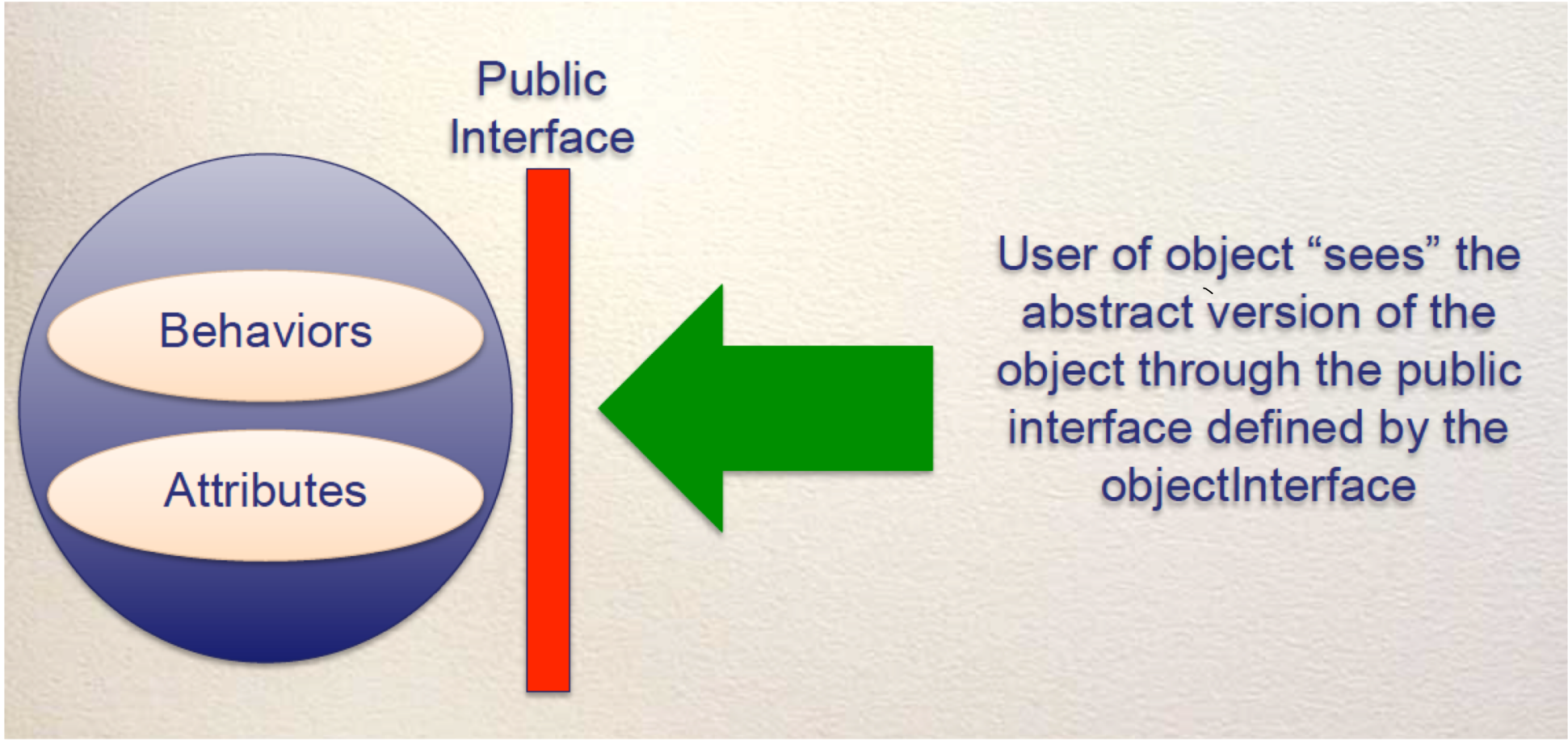
Encapsulation

Benefits of Setter and Getter functions

- Centralized control of how a certain attribute is initialized and provided to the client
  - E.g., display balance in different currencies.
- Data validation
  - Try to set balance to a negative number
- Protect sensitive data
- Make debugging easier



Abstraction



Encapsulation makes abstraction possible

Abstraction

Abstraction in your life

?

You know the public interface. Do you know implementation details? Do you care?

As long as the public interface stays the same, you don't care about implementation changes

Abstraction

Private methods in Python

```
mycar = Car("2020", "Toyota", 0)
mycar.accelerate()
print ("speed = "+ str(mycar.get_speed()))

mycar.addAccident()
mycar.addAccident()
print ("accidents = "+ str(mycar.reportAccidents()))
```

```
speed = 5

---------------------------------------------------------
--------------------
AttributeError                            Traceback (mo
st recent call last)
<ipython-input-19-072fc01c3b0a> in <module>
     44 mycar.addAccident()
     45 mycar.addAccident()
---> 46 print ("accidents = "+ str(mycar.reportAccident
s()))
     47
     48

AttributeError: 'Car' object has no attribute 'reportAc
cidents'
```

```
class Car:
    def __init__(self, year, make, speed, accidents=0):
        self.__year_model = year
        self.__make = make
        self.__speed = speed
        self.__accidents = accidents

    ... #setters and getters

    #methods
    def accelerate(self):
        self.__speed +=5

    def brake(self):
        self.__speed -=5

    def addAccident(self):
        self.__accidents +=1

    def __reportAccidents(self):
        return self.__accidents;
```

"__" makes a method private

Abstraction

Private methods in Python

```
mycar = Car("2020", "Toyota", 0)
mycar.accelerate()
print ("speed = "+ str(mycar.get_speed()))

mycar.addAccident()
mycar.addAccident()
print ("accidents = "+
       str(mycar._Car__reportAccidents()))
```

```
speed = 5
accidents = 2
```

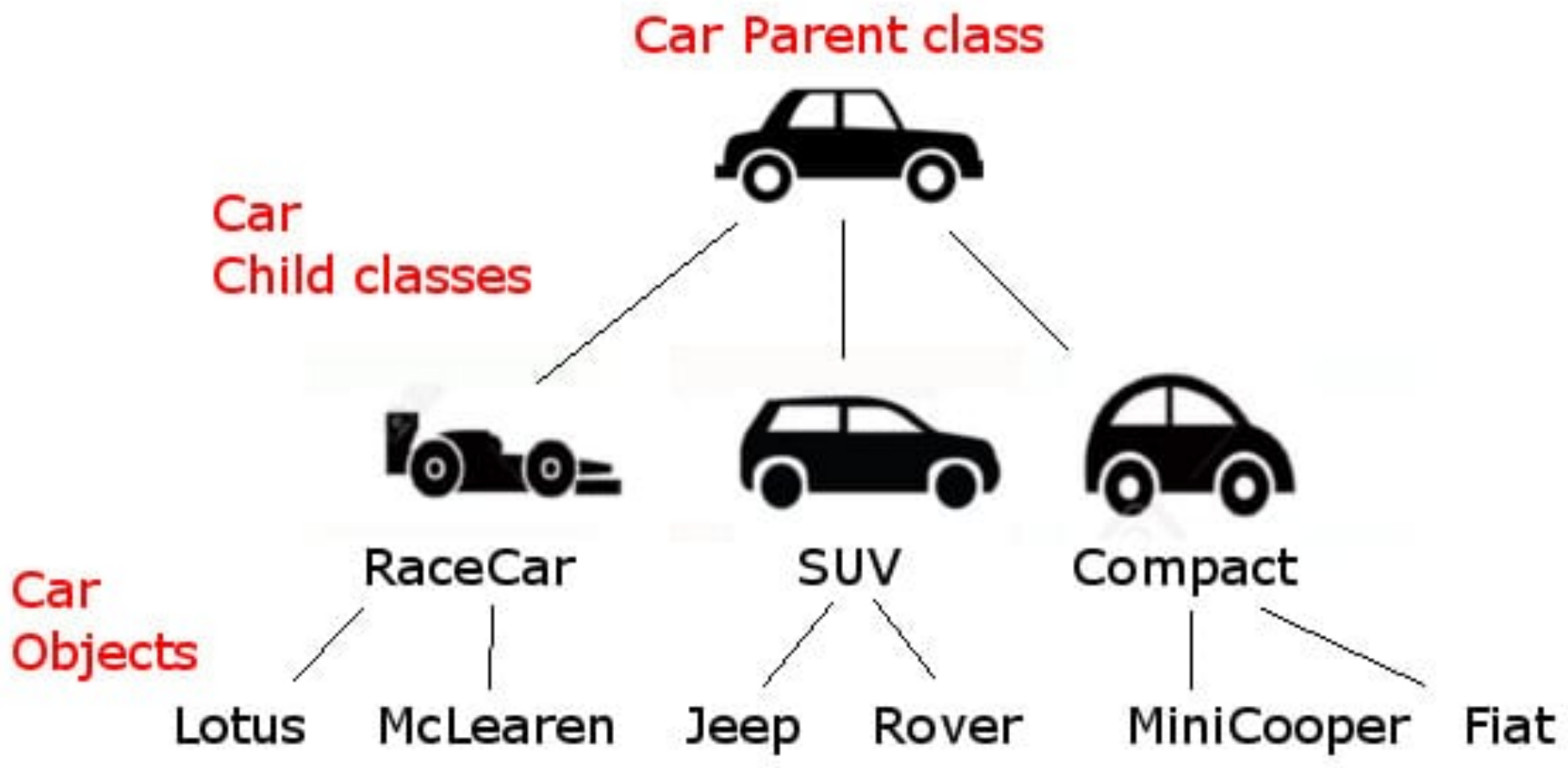
It is possible to peek at the private data in Python, but good programmers know not to do it.

```
class Car:
    def __init__(self, year, make, speed, accidents=0):
        self.__year_model = year
        self.__make = make
        self.__speed = speed
        self.__accidents = accidents

    ... #setters and getters

    #methods
    def accelerate(self):
        self.__speed +=5

    def brake(self):
        self.__speed -=5

    def addAccident(self):
        self.__accidents +=1

    def __reportAccidents(self):
        return self.__accidents;
```

"__" makes a method private

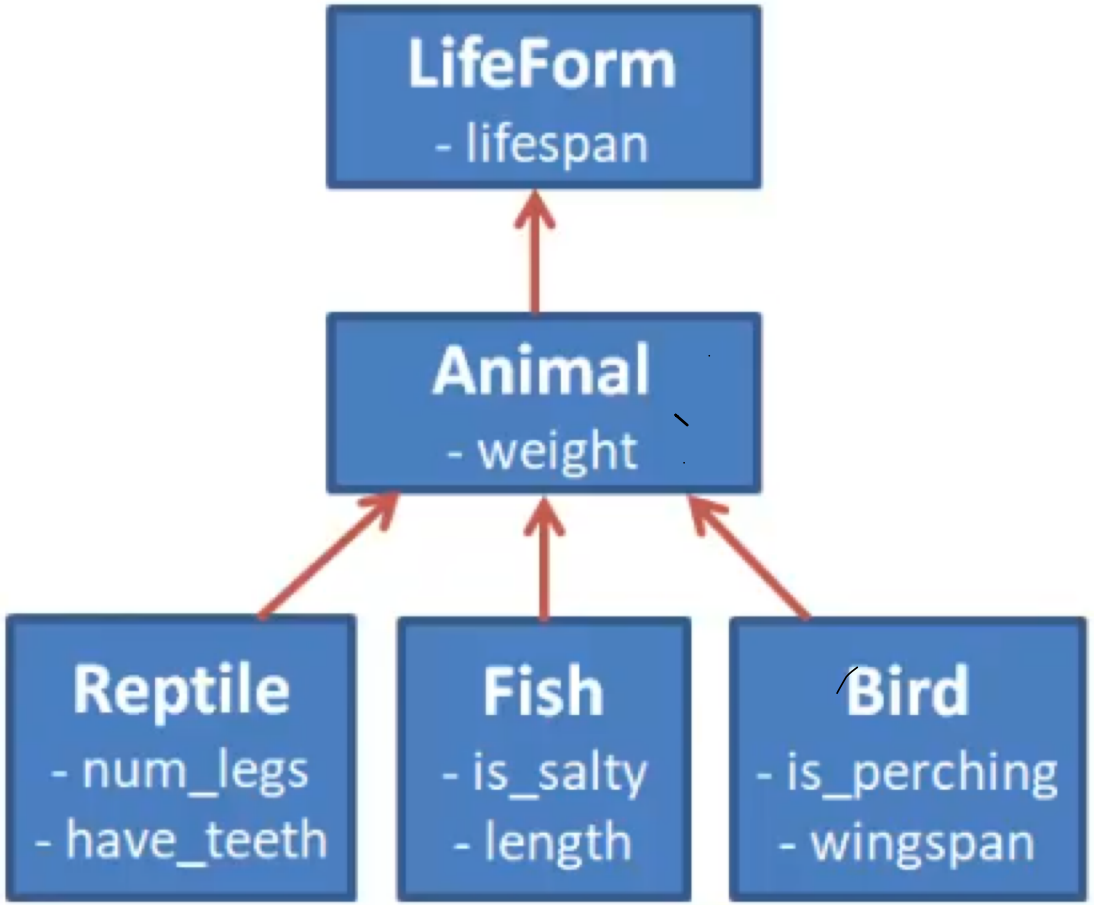
Inheritance

Classes can reuse code from other classes.



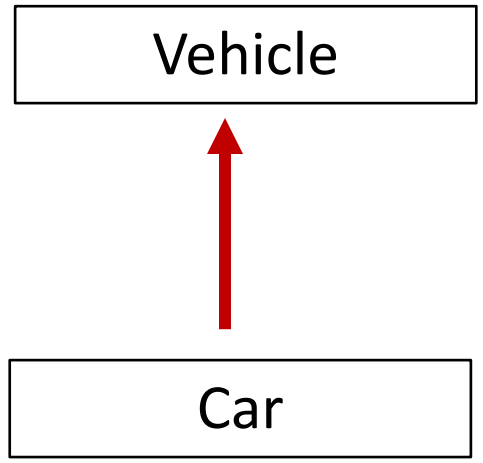
Inheritance

Classes can reuse code from other classes.



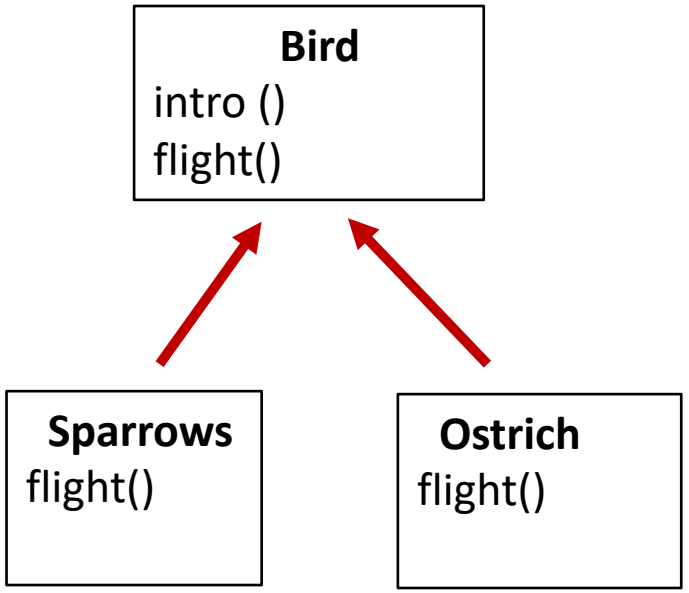
Inheritance

```
class Vehicle:
    def __init__(self, make, color, model):
        self.make = make
        self.color = color
        self.model = model
    def printDetails(self):
        print ("Manufacturer: ", self.make)
        print ("Color: ", self.color)
        print ("Model: ", self.model)


class Car(Vehicle):
    def __init__(self, make, color, model, doors):
        #Vehicle.__init__(self, make, color, model)
        super().__init__(make, color, model)
        self.doors = doors
    def printCarDetails(self):
        self.printDetails()
        print ("Doors: ", self.doors)
```



Polymorphism

Objects are designed to share behaviors and they can take on more than one form.




```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class Sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class Ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")
```

Polymorphism

Objects are designed to share behaviors and they can take on more than one form.

```
obj_bird = Bird()
obj_spr = Sparrow()
obj_ost = Ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class Sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class Ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")
```

Polymorphism

Objects are designed to share behaviors and they can take on more than one form.

```
def fly(obj):
    obj.flight()


fly(Bird())
fly(Sparrow())
fly(Ostrich())
```

```
Most of the birds can fly but some cannot.
Sparrows can fly.
Ostriches cannot fly.
```

```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class Sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class Ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")
```

Benefits of OOP

- **Modularity.** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
- **Reusability.** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
- **Productivity.** Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
- **Easily upgradable and scalable.** Programmers can implement system functionalities independently.

Benefits of OOP

- **Interface descriptions.** Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.
- **Security.** Using encapsulation and abstraction, complex code is hidden, software maintenance is easier.
- **Flexibility.** Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

NN.ipynb (Posted in BB) **:** template file for your HW1, Q2