

# Object-Oriented Programming

OOP

# Recap: Tables

- Just like other Spreadsheet applications

Name	Stu. No.	English	Math	Science	Social Studies
John	A1000000A	90	80	100	70
Peter	A1000009D	60	100	60	90
Paul	A1000003C	80	80	70	90
Mary	A1000001B	100	70	80	80

```
>>> records = [['John', 'A1000000A', 90, 80, 100, 70],  
                ['Peter', 'A1000009D', 60, 100, 60, 90],  
                ['Paul', 'A1000003C', 80, 80, 70, 90],  
                ['Mary', 'A1000001B', 100, 70, 80, 80]]
```

- How about more “attributes”

Name	Stu. No.	Gender	Year	English	Math	Science	Social Studies
John	A1000000A	M	2018	90	80	100	70
Peter	A1000009D	M	2018	60	100	60	90
Paul	A1000003C	M	2017	80	80	70	90
Mary	A1000001B	F	2019	100	70	80	80

# NUS Application

A. Personal Particulars		
Name (Please underline Family Name):	NRIC/Passport No.:	Gender: Male / Female #
Faculty / School and Department:	Matriculation No.:	Nationality:
Residential Address:	Home Tel:	NUS e-mail address:
	Mobile:	Other e-mail address:
Mailing Address (if different from above):	Date of Birth:	Please list the countries which you have visited for:
	Ethnicity:	Holidays:
		Others:
Names of Parent(s)/ Guardian(s)/ Next of Kin#:	Relationship:	Total Annual Household Income (of family members who are financially supporting the applicant in S\$):
Number of family members:	Number of Siblings:	Number of family members supported by this income:
Residential Address of Parent/ Guardian/ Next of Kin#:	Office Tel:	Occupation:
	Home Tel:	Designation:
Mailing Address (if different from above):	Mobile:	Employer:

# Stored as a 2D Array

- Like in Spreadsheet applications

Name	Stu. No.	English	Math	Science	Social Studies
John	A1000000A	90	80	100	70
Peter	A1000009D	60	100	60	90
Paul	A1000003C	80	80	70	90
Mary	A1000001B	100	70	80	80

- Or, you can setup a more “comprehensible” column index dictionary

```
>>> colIndex = {'name':0, 'SN':1, 'eng':2, 'math':3, 'sci':4,  
               'sos':5}
```

```
>>> records[3][colIndex['eng']]=0
```

```
>>> pprint(records)
```

```
[['John', 'A1000000A', 90, 80, 100, 70],  
 ['Peter', 'A1000009D', 60, 100, 60, 90],  
 ['Paul', 'A1000003C', 80, 80, 70, 100],  
 ['Mary', 'A1000001B', 0, 70, 80, 80]]
```

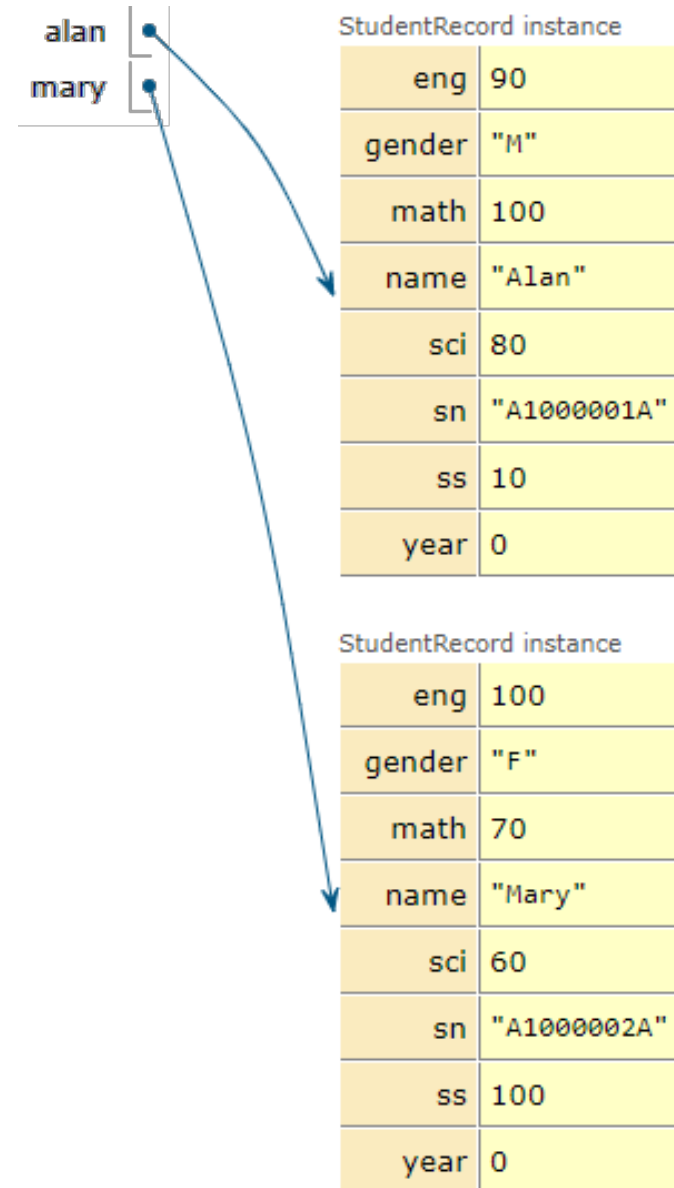
**MR. CLUMSY**

*By Roger Hargreaves*



# Isn't it nicer?

```
>>> alan.name
'Alan'
>>> alan.gender
'M'
>>> alan.sci
80
>>> alan.math
100
>>> mary.name
'Mary'
>>> mary.sn
'A1000002A'
>>> allStudents = [alan,mary]
>>> allStudents[0].name
'Alan'
>>> allStudents[1].sn
'A1000002A'
```



# Class and Instance

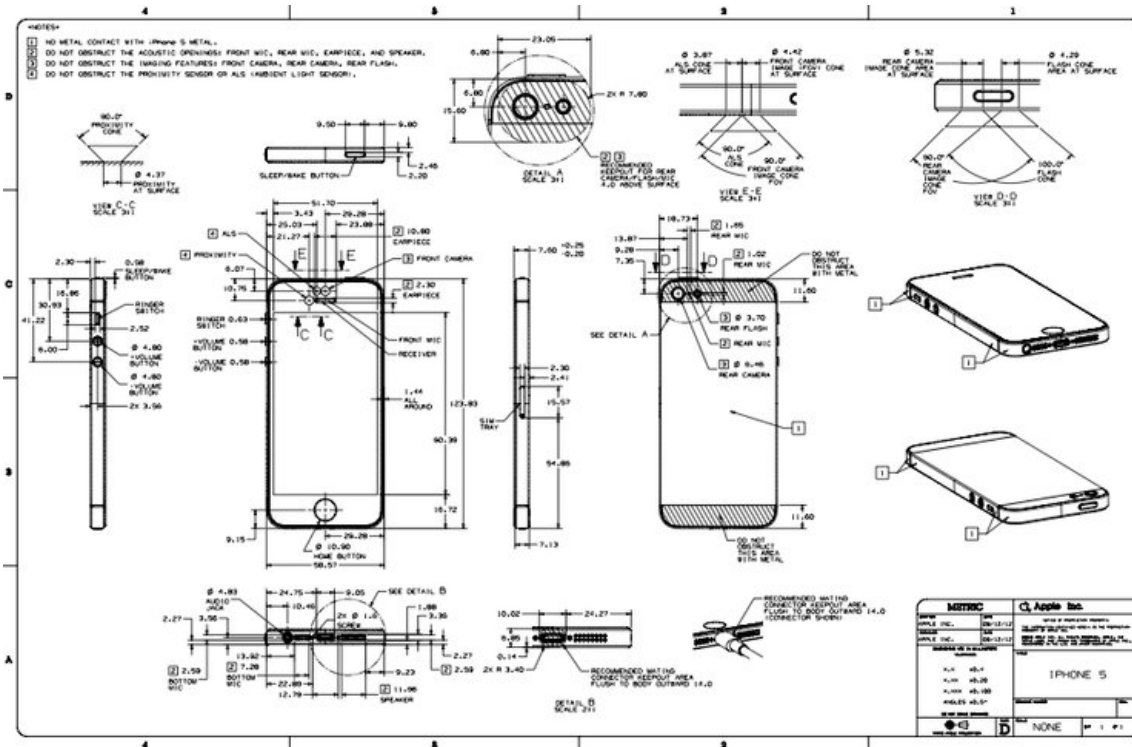
# Definitions

- **Class:**
  - specifies the common behavior of entities.
  - a *blueprint* that defines properties and behavior of an object.
- **Instance:**
  - A particular object or entity of a given class.
  - A concrete, usable object created from the blueprint.

# Classes vs Instances

- **Class**

- Blueprints, designs



- **Instance**

- Actual copies you use



One **blueprint** can produce a lot of copies of **iPhone**

One **class** can produce a lot of copies of **instances**



# Example

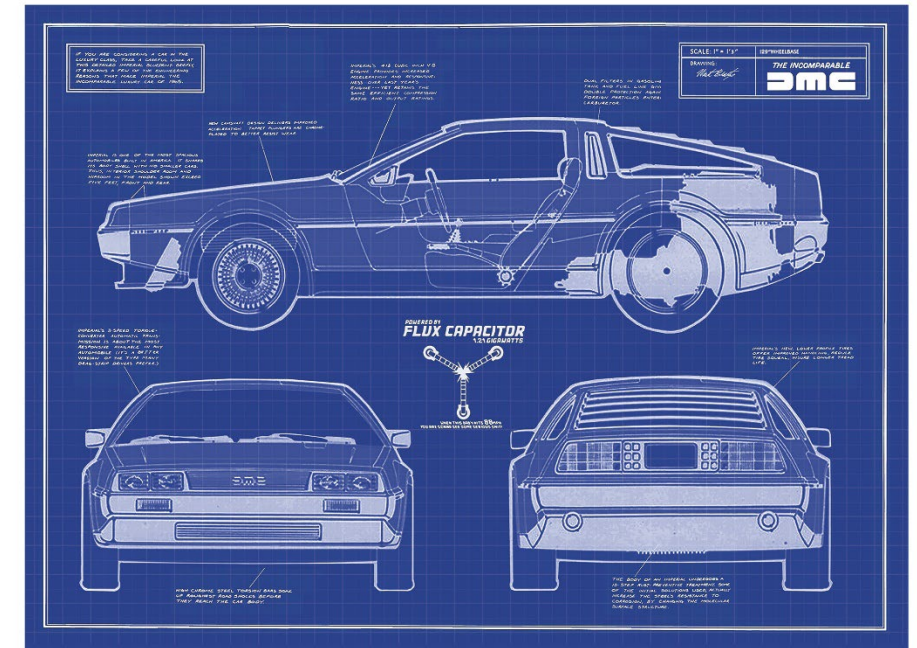
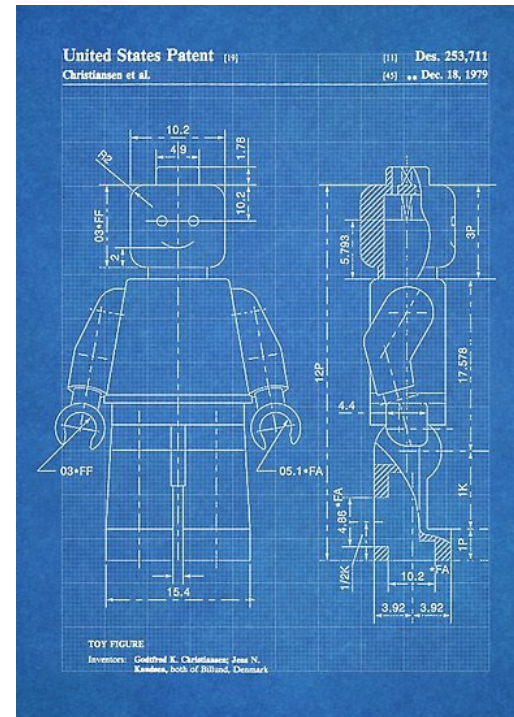
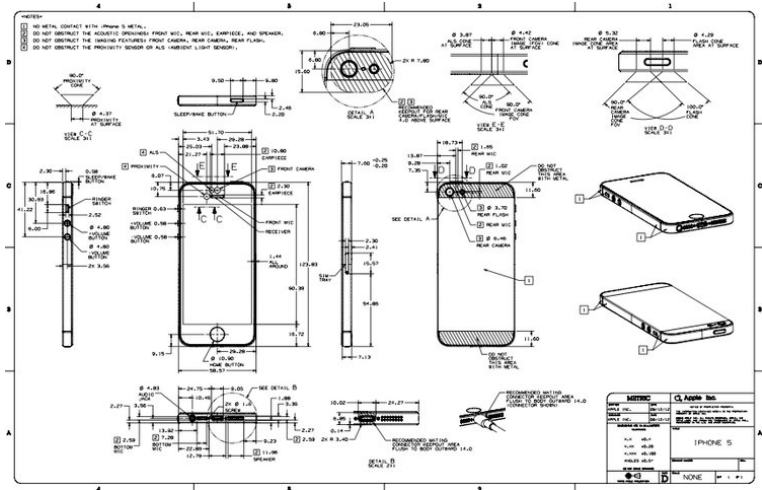
- String is a **class** in Python

```
>>> s = 'abc'  
>>> type(s)  
<class 'str'>
```

- The variable **s** above, is an **instance** of the class **str** (String)
  - And you can create other instances, e.g. **s1**, **str1**, **s2**, etc. of ONE class **str**
- And *each* instance will store different values

# Designing our own class

- Python OOP means we can design our own class and methods!



- Let's try to create a class called "StudentRecord"

# Class StudentRecord

- Design
  - In a student record, we want to store
    - Name, student number, gender, year, and marks for English, math, science and social studies
    - Or maybe more?

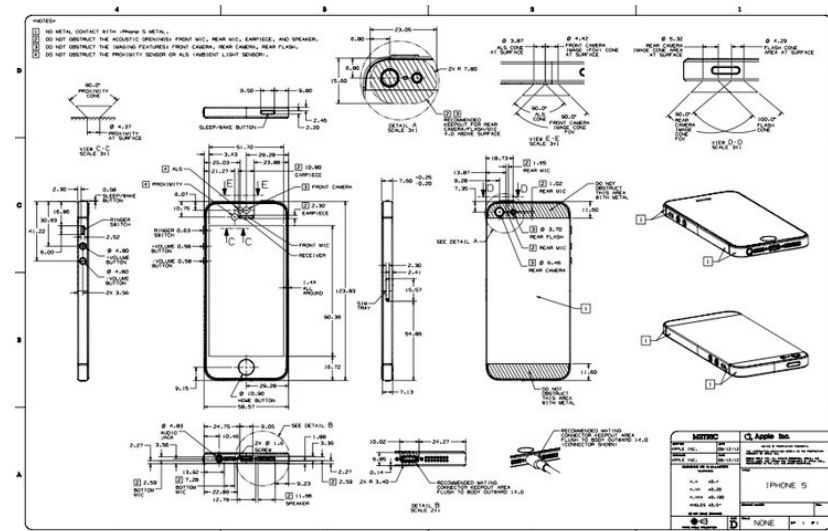
```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

In OOP, these are  
called attributes

# Class StudentRecord

- Design
  - In a student record, we want to store
    - Name, student number, gender, year, and marks for English, math, science and social studies
    - Or maybe more?
- But this is ONLY the **class**
  - Namely the blueprint
  - Can we use this phone blueprint to call someone?
    - You have to **MAKE** a phone by it

```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```



# Create an Instance

- When you create a new **instance**/variable:  
    >>> alan = StudentRecord()
- It's like you create a new variable x for integer  
    >>> x = 1
- A new instance/variable is born/created
- Important:

```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

**When you create an instance, the  
constructor function is called**

# A “self” variable?

- Every class definition has access to a `self` variable
- `self` is a reference to the entire instance

# What is `__init__()` ?

- `def __init__(self):`
  - called when the object is first initialized
  - `self` argument is a reference to the object calling the method.
  - It allows the method to reference properties and other methods of the class.
- Are there other special methods?
  - Yes! Special methods have `__` in front and behind the name

# Create an Instance

- When you create a new instance/variable:  
    >>> alan = StudentRecord()
- When you create an instance, the constructor function is called
  - What? What constructor?!
- In every class, you have to define a function called “\_\_init\_\_()”
  - “self” means your own record
    - To distinguish from a local variable in a function (as we had learn so far) that will be destroyed after the function ended

```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

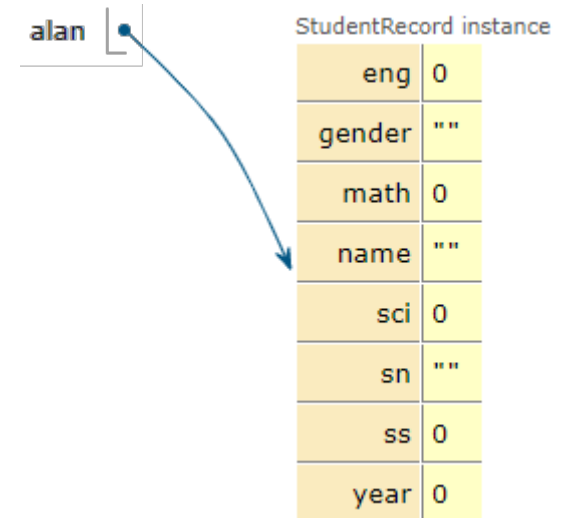
With two underscores “\_\_”



# Create an Instance

- When you create a new instance/variable:  
    >>> alan = StudentRecord()
- When you create an instance, the constructor function is called
- So after the line above, the instance **alan** will contain
  - So, the values in the constructor can be considered the default values for initialization

```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

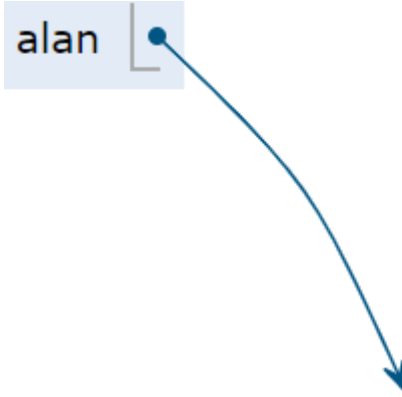


# You Store Values into an Instance

```
14 alan = StudentRecord()  
15 alan.name = 'Alan'  
16 alan.sn = 'A1000001A'  
17 alan.gender = 'M'  
18 alan.eng = 90  
→ 19 alan.math = 100  
→ 20 alan.sci = 80  
21 alan.ss = 10
```

alan

StudentRecord instance



eng	90
gender	"M"
math	100
name	"Alan"
sci	0
sn	"A1000001A"
ss	0
year	0

Before the red arrow

StudentRecord instance

eng	90
gender	"M"
math	100
name	"Alan"
sci	80
sn	"A1000001A"
ss	10
year	0

Finally. (Oops, I forgot to assign "year")

# Create as Many Instances as You Want

```
alan = StudentRecord()  
alan.name = 'Alan'  
alan.sn = 'A1000001A'  
alan.gender = 'M'  
alan.eng = 90  
alan.math = 100  
alan.sci = 80  
alan.ss = 10
```

```
mary = StudentRecord()  
mary.name = 'Mary'  
mary.sn = 'A1000002A'  
mary.gender = 'F'  
mary.eng = 100  
mary.math = 70  
mary.sci = 60  
mary.ss = 100
```

alan  
mary

StudentRecord instance

eng	90
gender	"M"
math	100
name	"Alan"
sci	80
sn	"A1000001A"
ss	10
year	0

StudentRecord instance

eng	100
gender	"F"
math	70
name	"Mary"
sci	60
sn	"A1000002A"
ss	100
year	0

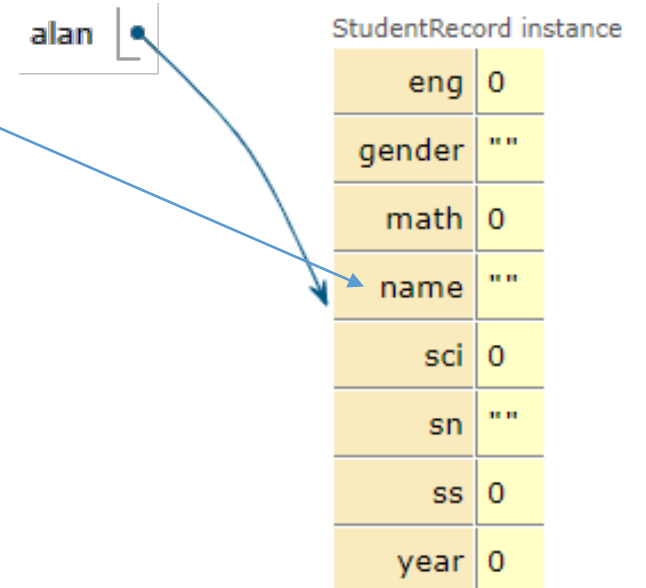
# A “self” variable?

- Every class definition has access to a `self` variable
- `self` is a reference to the entire instance
- A self variable will NOT `disappear` when the function exits





# With or without “self.”

- The variable “name” will disappear after the completion of the function `__init__()`
- The variable “self.name” will remain after the completion of the function `__init__()`

```
class StudentRecord():  
    def __init__(self):  
        name = 'whatever'  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```



# Other than Student Records

	Remius Lv 1	Holy Knight HP 558/ 558 MP 98/ 98
	Naia Lv 1	Monk HP 682/ 682 MP 46/ 46
	Elicia Lv 1	Knight HP 645/ 645 MP 95/ 95
	Erik Lv 1	Warrior HP 562/ 562 MP 41/ 41

	13-inch MacBook Air	13-inch MacBook Pro	15-inch MacBook Pro
Base Processor	8th Gen Intel dual-core 1.6GHz i5, turbo boost up to 3.6Ghz	8th Gen Intel quad-core 1.4GHz i5, turbo boost up to 3.9GHz	9th Gen Intel 6-core 2.6GHz i7, turbo boost up to 4.5GHz
Storage	128GB, 256GB, 512GB, or 1TB SSD	128GB, 256GB, 512GB, 1TB, or 2TB SSD	256GB, 512GB, 1TB, 2TB, or 4TB SSD
RAM	8GB or 16GB	8GB or 16GB	16GB or 32GB
Graphics	Integrated Intel UHD 617	Integrated Intel Plus Iris 645	Base w/ Radeon Pro 555X 4GB of GDDR5 memory + Intel UHD Graphics 630
Wi-Fi	802.11ac	802.11ac	802.11ac
Touch ID	✓	✓	✓
Touch Bar	✗	✓	✓
T2 Security Chip	✓	✓	✓

# Or Bank Accounts

- What attributes should we store in a bank account record?
  - Name
  - Balance
- Isn't it a bit clumsy to set the values?
  - We know that some attributes must be initialized before used

```
class BankAccount():  
    def __init__(self):  
        self.name = ''  
        self.balance = 0
```

```
myAcc = BankAccount()  
myAcc.name = 'Alan'  
myAcc.balance = 1000
```

```
johnAcc = BankAccount()  
johnAcc.name = 'John Wick'  
johnAcc.balance = 1000000000
```

# Initialization through Constructors

- We know that some attributes must be initialized before used
- When we create an instance, we initialize through the constructor

```
class BankAccount():  
    def __init__(self, name, balance):  
        self.name = name  
        self.balance = balance  
  
myAcc = BankAccount('Alan', 1000)  
johnAcc = BankAccount('John Wick', 1000000000)
```

- It is a good way to “force” the instance to be initialized

```
>>> myAcc.balance  
1000  
>>> myAcc.name  
'Alan'
```



# Modifying Attributes

- Of course, we can change the attributes of any instance

```
>>> myAcc.balance += 999  
>>> myAcc.balance  
1999
```

- However, is it always good to be changed like that?

```
>>> myAcc.balance -= 10000  
>>> myAcc.balance  
-8001
```

- There are always some rules to control how to modify the attributes
  - In real life, how do you withdraw money from your account?

# “Rules”

- Can you walk in the bank and get any **amount** even **more** than your balance? Or any other bank transactions?
- Must through some “mechanism”, e.g.
  - Bank tellers, ATM, phone/internet banking, etc.
- And these mechanisms have some **rules**,
  - E.g. you cannot withdraw more than your balance

# Bank Accounts with “Methods”

```
class BankAccount():  
    def __init__(self, name, balance):  
        self.name = name  
        self.balance = balance  
    def withdraw(self, amount):  
        if self.balance < amount:  
            print(f"Money not enough! You do not have ${amount}")  
            return 0  
        else:  
            self.balance -= amount  
            return amount  
    def showBalance(self):  
        print(f'Your balance is ${self.balance}')
```

Attributes

Methods

# Bank Accounts with “Methods”

```
>>> myAcc = BankAccount('Alan', 1000)
```

```
>>> myAcc.showBalance()
```

```
Your balance is $1000
```

```
>>> myAcc.withdraw(123)
```

```
123
```

```
>>> myAcc.showBalance()
```

```
Your balance is $877
```

```
>>> myAcc.withdraw(99999)
```

```
Money not enough! You do not have $99999
```

```
0
```

# Is it a *\*really\** a new thing?

- Recall your previous lectures...

```
lst = [1, 2, 3]
```

```
lst.append(4)
```

```
lst → [1, 2, 3, 4]
```

- Conceptually, append is a method defined in the `List` class.
- Just like withdraw is a method defined in the `BankAccount` class

# Inheritance



guess who's  
inheriting the money

# Let's Define a class Sportcar



```
class Sportcar:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0,0)
    def setVelocity(self, vx, vy):
        self.velocity = (vx,vy)
    def move(self):
        self.pos = (self.pos[0]+self.velocity[0],self.pos[1]+self.velocity[1])
        print(f"Move to {self.pos}")
    def turnOnTurbo(self):
        print("VROOOOOOOOM.....")
        self.velocity = (self.velocity[0]*2,self.velocity[1]*2)
        print(f"Velocity increased to {self.velocity}")
```



# Test Run

```
>>> myCar = Sportscar((0,0))
```

```
>>> myCar.setVelocity(0,40)
```

```
>>> myCar.move()
```

```
Move to (0, 40)
```

```
>>> myCar.turnOnTurbo()
```

```
VR0000000M.....
```

```
Velocity increased to (0, 80)
```

```
>>> myCar.move()
```

```
Move to (0, 120)
```

How about a class Lorry?



```
class Lorry:
    def __init__(self,pos):
        self.pos = pos
        self.velocity = (0,0)
        self.cargo = []
    def setVelocity(self,vx,vy):
        self.velocity = (vx,vy)
    def move(self):
        self.pos = (self.pos[0]+self.velocity[0],self.pos[1]+self.velocity[1])
        print(f"Move to {self.pos}")
    def load(self,cargo):
        self.cargo.append(cargo)
    def unload(self,cargo):
        if cargo in self.cargo:
            self.cargo.remove(cargo)
            print(f"Cargo {cargo} unloaded.")
        else:
            print(f"Cargo {cargo} not found.")
    def inventory(self):
        print("Inventory:"+str(self.cargo))
```

# Test Run

```
>>> myTruck = Lorry((10,10))
>>> myTruck.setVelocity(10,0)
>>> myTruck.move()
Move to (20, 10)
>>> myTruck.load("Food")
>>> myTruck.load("Supplies")
>>> myTruck.inventory()
Inventory: ['Food', 'Supplies']
```

```
>>> myTruck.unload("Food")
Cargo Food unloaded.
>>> myTruck.inventory()
Inventory: ['Supplies']
>>> myTruck.unload("Gold")
Cargo Gold not found.
```

# Compare the Two Classes

## **Sportscar**

- Attributes
  - pos
  - velocity
- Methods
  - `__init__()`
  - `setVelocity()`
  - `move()`
  - `turnOnTurbo()`

## **Lorry**

- Attributes
  - pos
  - velocity
  - cargo
- Methods
  - `__init__()`
  - `setVelocity()`
  - `move()`
  - `load()`
  - `unload()`
  - `inventory()`

What are the **common** attributes/methods?

# Compare the Two Classes

## Sportscar

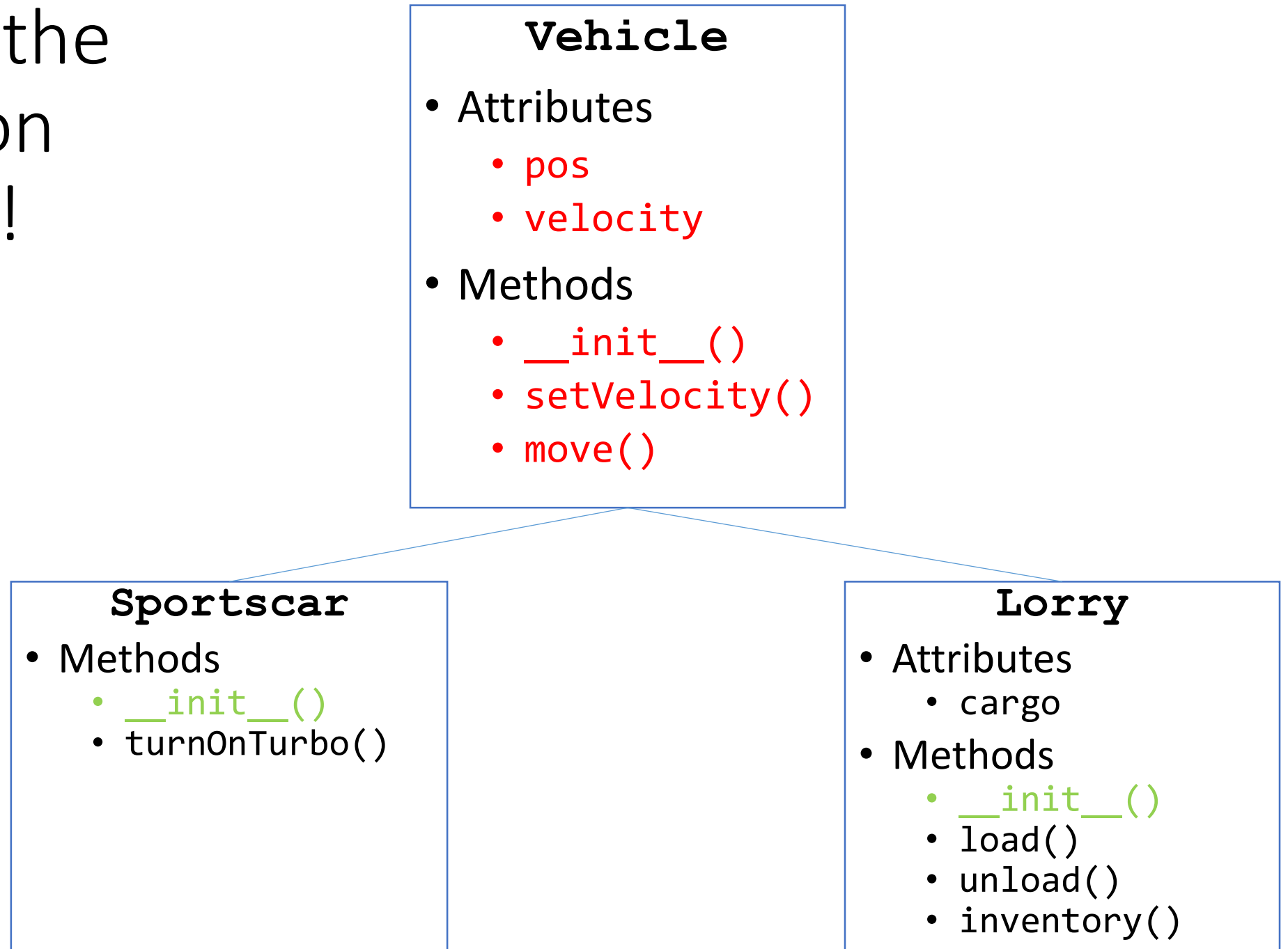
- Attributes
  - pos
  - velocity
- Methods
  - `__init__()`
  - `setVelocity()`
  - `move()`
  - `turnOnTurbo()`

## Lorry

- Attributes
  - pos
  - velocity
  - cargo
- Methods
  - `__init__()`
  - `setVelocity()`
  - `move()`
  - `load()`
  - `unload()`
  - `inventory()`

What are the common attributes/methods?

# Extract the Common Pattern!



# The Classes Vehicle and Sportscar

```
class Vehicle:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0, 0)
    def setVelocity(self, vx, vy):
        self.velocity = (vx, vy)
    def move(self):
        self.pos = (self.pos[0]+self.velocity[0], self.pos[1]+self.velocity[1])
        print(f"Move to {self.pos}")

class Sportscar(Vehicle):
    def turnOnTurbo(self):
        print("VROOOOOOOM.....")
        self.velocity = (self.velocity[0]*2, self.velocity[1]*2)
        print(f"Velocity increased to {self.velocity}")
```

← Sportscar inherits EVERYTHING from Vehicle



# How about Lorry?

- In the OLD Lorry code

```
class Lorry:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0,0)
        self.cargo = []
    def setVelocity(self, vx, vy):
        self.velocity = (vx, vy)
    def move(self):
        self.pos = (self.pos[0]+self.velocity[0], self.pos[1]+self.velocity[1])
        print(f"Move to {self.pos}")
    def load(self, cargo):
        self.cargo.append(cargo)
```

Extra to the `__init__()`  
in Vehicle

# If We Inherit Lorry from Vehicle

- Two ways to implement the constructor
- Method 1: Overriding
  - Simple redefining the method will override the one in Vehicle

or

- Method 2: Calling super class
  - Redefine a constructor, but call the constructor in `super()` (Vehicle class) instead

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0,0)  
        self.cargo = []
```


```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        super().__init__(pos)  
        self.cargo = []
```

# Super()

- A way to access a method in your parent/higher classes

```
class Vehicle:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0, 0)

class Lorry(Vehicle):
    def __init__(self, pos):
        super().__init__(pos)
        self.cargo = []
```



# Which one is better?

- Usually we prefer Method 2 because
  - No duplication of code
- Method 2: Calling super class
  - Redefine a constructor, but call the constructor in super() (Vehicle class) instead

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0,0)  
        self.cargo = []
```

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        super().__init__(pos)  
        self.cargo = []
```

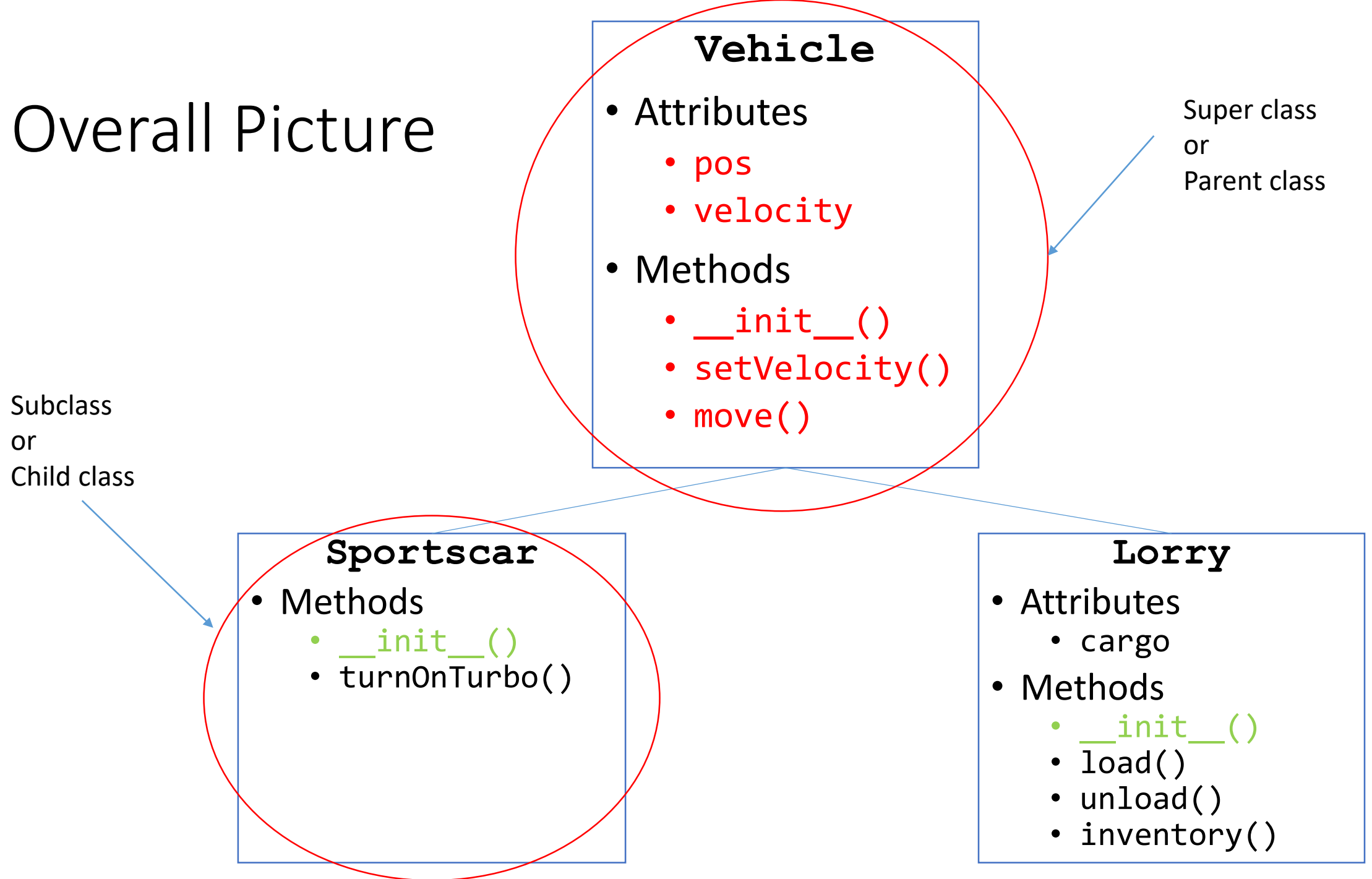


# Class Lorry

- Inherit all what class Vehicle has
- In addition, add more functionalities like `load()` and `unload()`

```
class Lorry(Vehicle):
    def __init__(self, pos):
        super().__init__(pos)
        self.cargo = []
    def load(self, cargo):
        self.cargo.append(cargo)
    def unload(self, cargo):
        if cargo in self.cargo:
            self.cargo.remove(cargo)
            print(f"Cargo {cargo} unloaded.")
        else:
            print(f"Cargo {cargo} not found.")
    def inventory(self):
        print("Inventory:" + str(self.cargo))
```

# Overall Picture

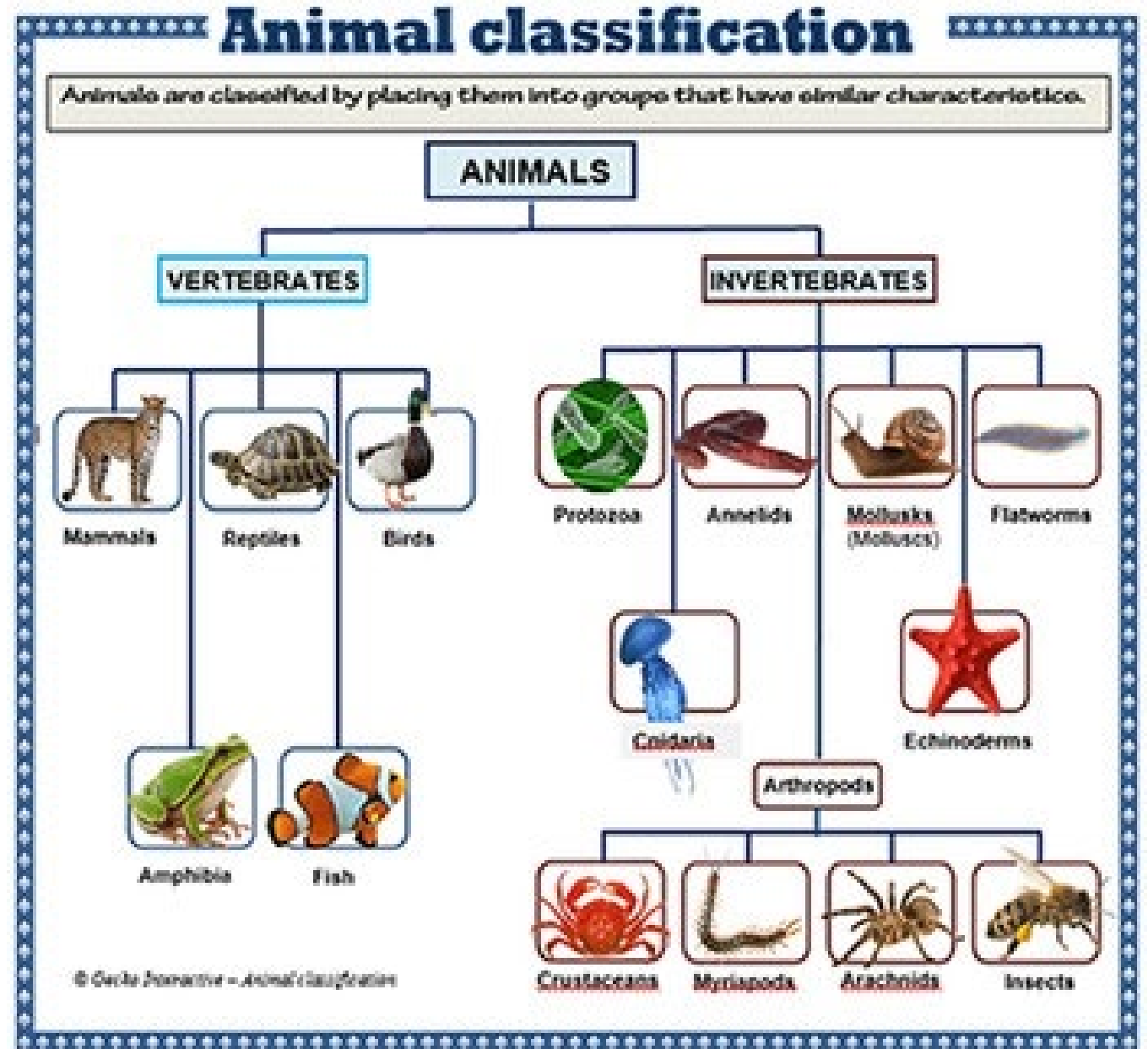


# Inheritance

- Objects that exhibit similar functionality should “inherit” from the same base object, called the **superclass**.
- An object that inherits from another is called the **subclass**.

# Subclass and Superclass

- Usually a subclass is a more *specific* type of its parent class
  - Like our classification of animals, any “children” in the tree is a more “specific” type of the “parents”
  - (Unless we talk about multiple inheritance later)





# Let's define a new class Bisarca

- Car carrier trailer
  - It is a type of truck that carries other cars
- The truck and also load and unload, but only for cars
  - not any type of cargos



Where should  
we put class  
Bisarca?

### Vehicle

- Attributes: `pos, velocity`
- Methods: `__init__(), setVelocity(), move()`

### Sportscar

- Methods: `__init__(), turnOnTurbo()`

### Lorry

- Attributes: `cargo`
- Methods: `__init__(), load(), unload(), inventory()`

### Bisarca

- Methods: `load()`

# Class Biscar

```
class Biscar(Lorry):  
    def load(self, cargo):  
        if isinstance(cargo, Vehicle):  
            super().load(cargo)  
        else:  
            print(f'Your cargo ({cargo}) is not a vehicle!')
```

- The function `isinstance(obj, cal)` check if an instance `obj` is a class or subclass of a certain class `cal`.

# Sample Run

```
>>> myDadTruck = Biscarica((0,0))
```

```
>>> myDadTruck.load("Food")
```

```
Your cargo (Food) is not a vehicle!
```

```
>>> myDadTruck.load(myCar)
```

```
>>> myDadTruck.load(myTruck)
```

```
>>> myDadTruck.inventory()
```

```
Inventory:[<__main__.Sportcar object at 0x10d3ecd50>,  
<__main__.Lorry object at 0x10d39dc10>]
```

# Method Overriding

- When you redefine a same method that was in your parent class
- You own class will call your new redefined method
  - Instead of your parent's one
- This is called **overriding**

## **Lorry**

- Attributes: cargo
- Methods: `__init__()`, **`load()`**, `unload()`, `inventory()`

## **Bisarca**

- Methods: **`load()`**

# Multiple Inheritance



# Let's Create a Class Cannon



# Class Cannon

- Sample run:

```
>>> myCannon = Cannon()
>>> myCannon.fire()
No more ammo
>>> myCannon.reload()
Cannon reloaded
>>> myCannon.reload()
Unable to reload
>>> myCannon.fire()
Fire!!!!!!
>>> myCannon.fire()
No more ammo
```

```
class Cannon:
    def __init__(self):
        self.numAmmo = 0
    def fire(self):
        if self.numAmmo:
            print("Fire!!!!!!")
            self.numAmmo -= 1
        else:
            print("No more ammo")
    def reload(self):
        if self.numAmmo:
            print("Unable to reload")
        else:
            print("Cannon reloaded")
            self.numAmmo += 1
```



# What Do You Have When You...

- Merge a cannon and a vehicle?



# We Want to Have BOTH!

```
>>> myTank = Tank((0,0))
>>> myTank.setVelocity(40,10)
>>> myTank.move()
Move to (40, 10)
>>> myTank.move()
Move to (80, 20)
>>> myTank.reload()
Cannon reloaded
>>> myTank.fire()
Fire!!!!!!!
```

# Where should we put the class Tank?

## Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

## Cannon

- Attributes: numAmmo
- Methods: fire()

## Sportscar

- Methods: \_\_init\_\_(), turnOnTurbo()

## Lorry

- Attributes: cargo
- Methods: \_\_init\_\_(), load(), unload(), inventory()

## Tank

## Bisarca

- Methods: load()

# A Bit Trouble

- Which constructor `__init__()` should the Tank call?

```
class Cannon:
    def __init__(self):
        self.numAmmo = 0
```

```
class Vehicle:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0,0)
```

- Seems like we need BOTH

```
class Tank(Vehicle, Cannon):
    def __init__(self, pos):
        Vehicle.__init__(self, pos)
        Cannon.__init__(self)
```

**Call BOTH!!!**

# Resolving Methods

# So far we have

## Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

## Cannon

- Attributes: numAmmo
- Methods: fire()

## Sportscar

- Methods: \_\_init\_\_(), turnOnTurbo()

## Lorry

- Attributes: cargo
- Methods: \_\_init\_\_(), load(), unload(), inventory()

## Tank

## Bisarca

- Methods: load()

# What Do You Have When You...

- Merge a Biscar and a Cannon?



+



=



# Let's Construct Class BattleBisarca

```
class BattleBisarca(Bisarca, Cannon):  
    def __init__(self, pos):  
        Bisarca.__init__(self, pos)  
        Cannon.__init__(self)
```

```
OptimasPrime = BattleBisarca((0,0))  
OptimasPrime.load("Food")
```

- Wait... Which load() is called?





## Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

## Cannon

- Attributes: numAmmo
- Methods: fire()

## Lorry

- Attributes: cargo
- Methods: \_\_init\_\_(), load(), unload(), inventory()

## Bisarca

- Methods: load()

## BattleBisarca

Which  
load() is  
called by  
BattleBisarca  
?

## Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

## Lorry

- Attributes: cargo
- Methods: \_\_init\_\_(), load(), unload(), inventory()

## Bisarca

- Methods: load()

## Cannon

- Attributes: numAmmo
- Methods: fire()

The nearest one will be called

## BattleBisarca

# Let's Construct Class BattleBisarca

```
class BattleBisarca(Bisarca, Cannon):  
    def __init__(self, pos):  
        Bisarca.__init__(self, pos)  
        Cannon.__init__(self)
```

```
OptimasPrime = BattleBisarca((0,0))  
OptimasPrime.load("Food")
```

- Wait... Which load() is called?

```
>>> OptimasPrime = BattleBisarca((0,0))
```

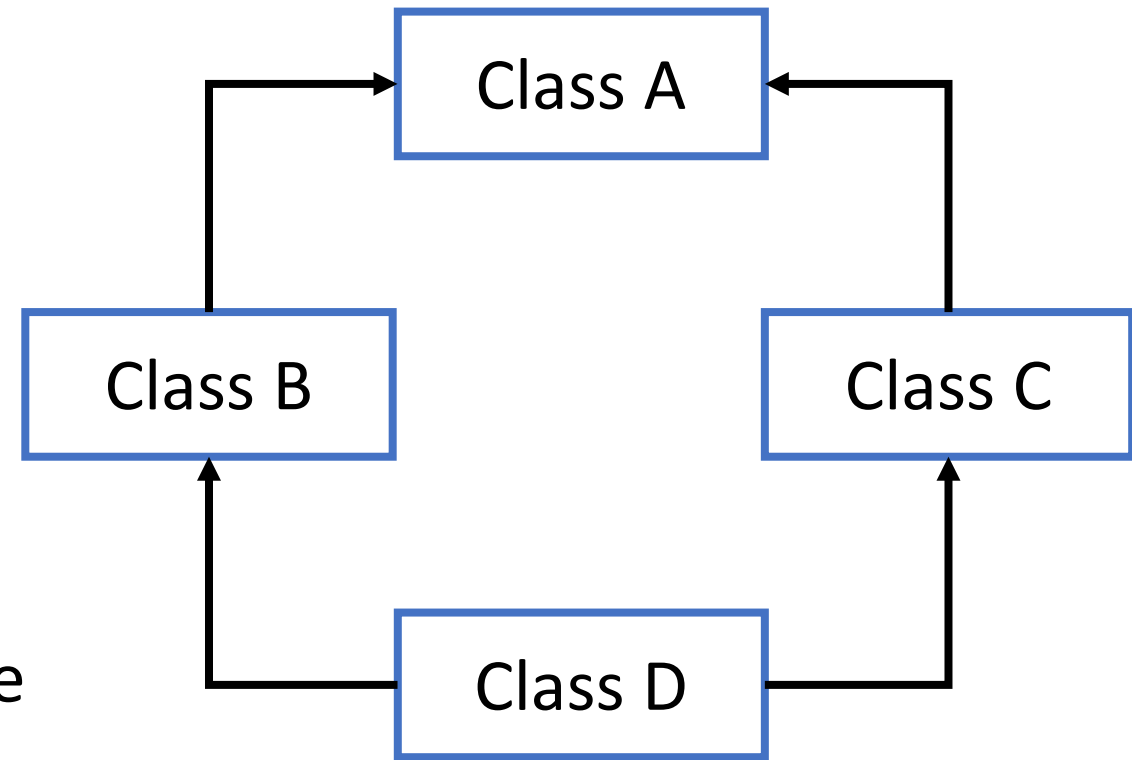
```
>>> OptimasPrime.load("Food")
```

Your cargo (Food) is not a vehicle!



# Multiple Inheritance

- Complication arises when the same method is available in two distinct superclasses
- And how about **Diamond Inheritance**
- But if you are really interested in these
  - Check out:
    - <http://python-history.blogspot.com/2010/06/method-resolution-order.html>



# Multiple Inheritance

- Not many OOP Language support MI
  - E.g. no MI in C++, Java
- MI causes more trouble sometime because you may call the unexpected method in a complicated inheritance structure
- Recommendation is, only use MI if the parents are very different
  - E.g. Vehicle and Cannon
  - Or Tablet (computer) + calling device = smart phone

# Private vs Public

# Private vs Public

- So far, all our methods in a class are all *public*
- Meaning they can be called with the instance
- E.g. for the class BankAccount
  - Even we set up the method withdraw() to “prevent” illegal access

```
def withdraw(self, amount):  
    if self.balance < amount:  
        print(f"Money not enough! You do not have ${amount}")  
        return 0  
    else:  
        self.balance -= amount  
        return amount
```

- But we can still do

```
>>> myAcc.showBalance()
```

```
Your balance is $1000
```

```
>>> myAcc.balance -= 9999
```

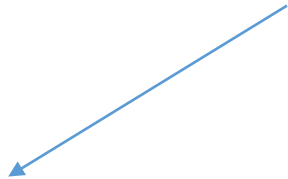
```
>>> myAcc.showBalance()
```

```
Your balance is -$8999
```

# Another Example: Remember the Biscarca?

```
>>> myDadTruck.load(myCar)
>>> myDadTruck.load(myTruck)
>>> myDadTruck.inventory()
```

So ugly



```
Inventory: [<__main__.Sportcar object at
0x10d3ecd50>, <__main__.Lorry object at
0x10d39dc10>]
```

- What I really want is

```
>>> myDadTruck.inventory()
Inventory: ['Sportscar', 'Lorry']
```



# So I change my Biscar class into

```
class Biscar(Lorry):  
    def convertCargo(self):  
        output = []  
        for c in self.cargo:  
            output.append(str(type(c)).split('.')[1].split('\\')[0])  
        return output  
    def inventory(self):  
        print("Inventory:" + str(self.convertCargo()))
```

- Wait, but I actually do not want anyone to use the method `convertCargo()`, it's not for anyone
  - I want to make it **private**

# Private Methods

- If you add two **underscore** before the method name

```
class Biscarca(Lorry):  
    def __convertCargo(self):  
        output = []  
        for c in self.cargo:  
            output.append(str(type(c)).split('.')[1].split('\')[0])  
        return output  
    def inventory(self):  
        print("Inventory:" + str(self.__convertCargo()))
```

- That function can be used inside your class but cannot be called outside!

```
>>> myDadTruck.__convertCargo()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#1>", line 1, in <module>
```

```
    myDadTruck.__convertCargo()
```

```
AttributeError: 'Biscarca' object has no attribute '__convertCargo'
```

# Private but not Private

- However, it's not very true...
- You can add '\_' and the class name to access it

```
>>> myDadTruck._Bisarca__convertCargo()  
['Sportscar', 'Lorry']
```

- But why do we have this?!

# “Private” Methods

- Originally, in a lot of other OOP languages (e.g. C++, Java), a private method/variable will NOT be accessible by anyone other than the class itself.
- The purpose is to prevent any programmers to access the method/variable in a wrong way
  - E.g. directly change the balance of a bank account like  
`myAcc.balance = 1000000000`
- However, Python does not have that “full protection”

Don't forget Archipelagos

# Conclusion

# Benefits of OOP

- Pros
  - Simplification of complex, possibly hierarchical structures
  - Easy reuse of code
  - Easy code modifiability
  - Intuitive methods
  - Hiding of details through message passing and polymorphism
- Cons
  - Overhead associated with the creation of classes, methods and instances

# Major Programming Paradigms

- Imperative Programming
  - C, Pascal, Algol, Basic, Fortran
- Functional Programming
  - Scheme, ML, Haskell,
- Logic Programming
  - Prolog, CLP
- Object-oriented programming
  - Java, C++, Smalltalk

Python??