東南大學 SOUTHEAST UNIVERSITY 人工智能学院

Python编程

授课教师：王洪松

邮箱：hongsongwang@seu.edu.cn

Python面向对象程序设计

视频多目标跟踪代码

```
class Object:
    def __init__(self, id, bbox, frame_id):
        self.id = id
        self.bbox = bbox
        self.trajectory = [bbox]
        self.last_frame = frame_id

    def update(self, bbox, frame_id):
        self.bbox = bbox
        self.trajectory.append(bbox)
        self.last_frame = frame_id

class Detector:
    def detect(self, frame):
        # return list of bounding boxes

class Association:
    def match(self, detections, objects):
        # 基于最近邻或 IoU 匹配
```

```
class Video:
    def __init__(self, video_path, detector, association):
        self.frames = load_video(video_path)
        self.detector = detector
        self.association = association
        self.objects, self.next_object_id = {},  0

    def track(self):
        for frame_id, frame in enumerate(self.frames):
            detections = self.detector.detect(frame)
            matches, unmatched_dets =
                    self.association.match(detections, self.objects)
             # 更新已有 Object
            for obj_id, det_idx in matches:
                bbox = detections[det_idx]
                self.objects[obj_id].update(bbox, frame_id)
            # 新目标初始化
            for det_idx in unmatched_dets:
                bbox = detections[det_idx]
                new_id = self.next_object_id
                self.objects[new_id] = Object(new_id, bbox, frame_id
                self.next_object_id += 1
        return self.objects
```

视频多目标跟踪代码

多目标跟踪 面向对象编程 伪代码（Video / Object)

类结构：

Video：负责整体流程（读帧 → 检测 → 关联 → 更新对象）。

Object：单个目标实例（ID、状态、轨迹、更新方法等）。

Detector：负责从每帧中输出检测框。

Tracker / Association（简化）：负责判断检测属于哪个已有目标。

视频多目标跟踪代码2

```
class Object:
    def __init__(self, id, bbox, frame_id):
        self.id = id
        self.bbox = bbox
        self.trajectory = [bbox]
        self.last_frame = frame_id

    def update(self, bbox, frame_id):
        self.bbox = bbox
        self.trajectory.append(bbox)
        self.last_frame = frame_id

class Detector:
    def detect(self, frame):
        pass

class Association:
    def match(self, detections, objects):
        pass
```

```
class Video:
    def __init__(self, video_path, detector, association):
        self.frames = load_video(video_path)  # list of images
        self.detector = detector
        self.association = association
        self.objects = {}
        self.next_id = 0

    def track(self):
        first_frame = Frame(
            frame_id=0,
            image=self.frames[0],
            detector=self.detector,
            association=self.association,
            frames=self.frames,
            objects=self.objects,
            next_id=self.next_id
        )
        return first_frame.update_trajectory()
```

视频多目标跟踪代码2

```
class Frame:
    def __init__(self, frame_id, image, detector,
association, frames, objects, next_id):
        self.frame_id = frame_id
        self.image = image
        self.detector = detector
        self.association = association
        self.frames = frames
        self.objects = objects
        self.next_object_id = next_id

    def update_trajectory(self):
        if self.frame_id >= len(self.frames):
            return self.objects

        detections =self.detector.detect(self.image)
        matches, unmatched_dets =
self.association.match(detections, self.objects)
```

```
# 更新已有 Object
for obj_id, det_idx in matches:
    bbox = detections[det_idx]
    self.objects[obj_id].update(bbox, self.frame_id)
#   创建新 Object
for det_idx in unmatched_dets:
    bbox = detections[det_idx]
    new_id = self.next_object_id
    self.objects[new_id] = Object(new_id, bbox,
                                  self.frame_id)
    self.next_object_id += 1

# 递归处理下一帧
 if self.frame_id + 1 < len(self.frames):
    next_frame = Frame(
        frame_id=self.frame_id + 1,
        image=self.frames[self.frame_id + 1],
        detector=self.detector,
        association=self.association,
        frames=self.frames, objects=self.objects,
        next_id=self.next_object_id)
    return next_frame.update_trajectory()

return self.objects
```

视频多目标跟踪代码2

多目标跟踪 面向对象编程 + 递归设计：

每一帧是一个 Frame 类

Video 只负责初始化并调用第一帧，Video 类保持极简

Frame 的 update_trajectory() 方法递归处理下一帧

Object 仍然维护轨迹，帧负责调用更新/创建对象

Python程序设计

- Python编程
  - 面向过程编程?
  - 函数式编程?
  - 面向对象编程?

  到底选哪个?
  三种程序设计思想有各自适用场景!

为什么需要面向对象程序设计？

- 面向对象编程（OOP, Object-Oriented Programming）在软件开发中被广泛采用，不仅仅是语法或习惯，而是源于软件复杂性管理和可维护性的需求：
  - 现代软件系统庞大且复杂，涉及多种实体和交互。OOP 提供 “对象” 的概念，每个对象对应现实世界或系统中的一个实体。
  - OOP 的 继承（Inheritance） 和 多态（Polymorphism） 可以复用代码，减少重复代码，降低维护成本。
  - 提高代码可维护性和扩展性。封装隐藏对象内部实现，只暴露接口，降低了模块之间的耦合。新功能可以通过继承或添加新类实现，而不破坏现有系统。
  - 面向对象编程的思想源自现实世界实体建模：对象=实体，属性=状态，方法=行为
  - 支持软件工程原则 OOP 支持高内聚、低耦合 配合 SOLID 原则、设计模式，能写出可扩展、可维护的大型软件。

```
class Object:
    def update(self, bbox):
        # 这是接口的一部分
        pass
```

接口：update(bbox) 这个方法及其参数。

内部实现：函数体内部的代码。

面向对象的接口绑定在对象上，可以利用封装和多态面向过程的接口是独立函数，复用和扩展能力有限。

面向对象程序设计

- **面向过程编程/函数式编程**
  - 优点：流程化、简单化
  - 缺点：可扩展性差
  - 经典案例：Linux内核、Windows内核…
  - 适用场景：一旦完成基本很少改变，永久性或一次性
- **面向对象编程**
  - 优点：可扩展性好
  - 缺点：复杂度高、可控性差
  - 经典案例：游戏、浏览器…
  - 适用场景：需求经常变化

面向对象程序设计

- 面向对象编程的三大基本特征
  - **封装**：使用类将对象的属性和行为封装起来，类通常对用户隐藏其实现细节
  - **继承**：子类通过继承复用了父类的属性和行为的同时，又添加子类特有的属性和行为
  - **多态**：“一个接口，多种实现”，指一个父类中派生出了不同的子类，且每个子类在继承了同样的方法名的同时又对父类的方法做了不同的实现，表现出的多种形态

面向对象程序设计：类

- 对象：将数据以及对数据的操作封装在一起，组成的一个相互依存的整体
- 类：对于相同类型的对象进行分类、抽象后，将共同的特征封装成类

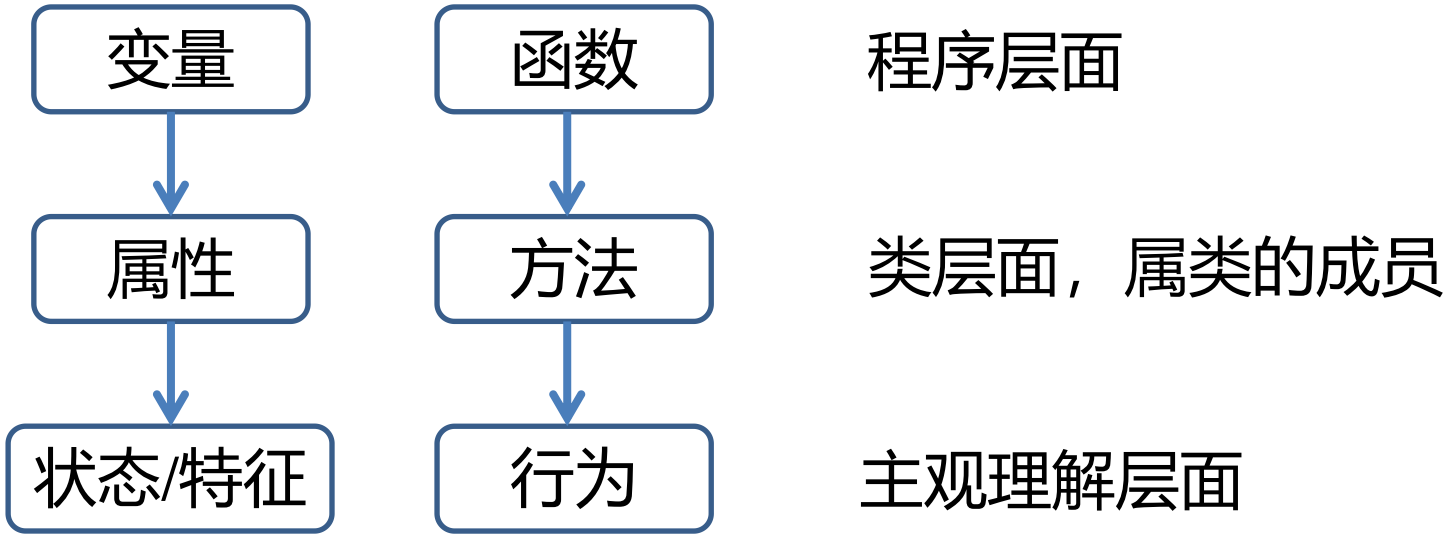
- 面向程序设计的难点就是合理地定义和组织程序中**类**以及**类之间的关系**

- Python使用class关键字来定义类，类名首字母建议大写

```
>>> class Car:
        price = 10000
        def info(self):
            print('This is a car!')
```

面向对象程序设计：类

- 梳理类的相关定义



面向对象程序设计：类

- 定义类以后，可以实例化对象，并通过“**对象名.成员**”的方法来访问其中的属性和方法

```
>>> car = Car()
>>> car.price
10000
>>> car.info()
This is a car!
```

- 内置方法isinstance()可以用测试一个对象是否为某个类的实例

```
>>> isinstance(car, Car)
True
>>> isinstance(car, list)
False
```

面向对象程序设计：类

- Python提供“pass”关键字，类似于空语句，可以用在类和函数的定义中或选择结构中
- 当暂时没有确定如何实现功能时，或为以后升级预留空间时，可以使用“pass”来“占位”

```
>>> class A:
        pass

>>> def demo():
        pass

>>> if 1 < 2:
        pass
```

面向对象程序设计：类

- self参数
  - 类的所有方法至少有一个名为self的参数，并且必须是方法的第一个形参（编程规范）
  - self代表将来要创建的对象本身
  - 在外部通过对象调用成员方法时不需要传递self参数
    - 若在外部通过类调用对象方法则需要显式为self参数传值

```
>>> class Person:
        def __init__(self,name):
            self.name=name
        def sayhello(self):
            print('My name is:',self.name)
```

面向对象程序设计：类

- self参数使用示例

```
>>> class Person:
        def __init__(self,name):
            self.name=name
        def sayhello(self):
            print('My name is:',self.name)

>>> p=Person('Kong')
>>> p.sayhello
<bound method Person.sayhello of <__main__.Person
object at 0x03D39BB0>>
>>> p.sayhello()
My name is: Kong
>>> p.sayhello('Xianglong')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    p.sayhello('Xianglong')
TypeError: sayhello() takes 1 positional argument
but 2 were given
```

面向对象程序设计：类

- 类与实例
  - 实例成员
    - 一般在构造函数__init__()中定义，须以self为前缀
  - 类成员
    - 在类中所有方法之外定义
  - 使用规则
    - 在类外部，实例成员只能通过对象名访问
    - 类成员可以通过类名或对象名访问

```
>>> class Car:
        price=100000                    #类成员
        def __init__(self, c):
            self.color=c                #实例成员

>>> car1=Car("Red")

>>> car1=Car("Red")
>>> car1.price
100000
>>> Car.price
100000
>>> car1.color
'Red'
>>> Car.color
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    Car.color
AttributeError: type object 'Car' has no attribut
e 'color'
```

面向对象程序设计：类

- Python中可以动态地为自定义类和对象增加或删除成员，这和常用的面向对象程序设计语言不同

```
>>> class Car:
        price=100000                    #类成员
        def __init__(self, c):
            self.color=c                #实例成员

>>> import types
>>> def setSpeed(self,s):
        self.speed = s

>>> car1=Car("Red")
>>> car1.setSpeed = types.MethodType(setSpeed, car1)
>>> car1.setSpeed(50)
>>> print(car1.price,car1.color,car1.speed)
100000 Red 50
```

面向对象程序设计：删除类成员/类方法

- 删除实例属性（对象的成员）

```python
class MyClass:
    def __init__(self):
        self.x = 10
        self.y = 20

obj = MyClass()
print(obj.x)  # 10

del obj.x
# print(obj.x)  # AttributeError: 'MyClass' object has no attribute 'x'
```

- 删除类方法

```python
class MyClass:
    def greet(self):
        print("Hello")

obj = MyClass()
obj.greet()  # Hello

del MyClass.greet
# obj.greet()  # AttributeError: 'MyClass' object has no attribute 'greet'
```

- 删除类成员/类属性

```python
class MyClass:
    z = 30  # 类属性

print(MyClass.z)  # 30

del MyClass.z
# print(MyClass.z)  # AttributeError: type object 'MyClass' has no attribute 'z'
```

面向对象程序设计：类

- 类成员与实例成员

```
>>> class Car:
        price=100000                    #类成员
        def __init__(self, c):
            self.color=c                #实例成员

>>> car1=Car("Red")

>>> def deal(self):
        print('Sold!')

>>> car1.deal()
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    car1.deal()
AttributeError: 'Car' object has no attribute 'deal'
>>> car1.deal = types.MethodType(deal, car1)
>>> car1.deal()
Sold!
```

面向对象程序设计：类

- 类成员与实例成员

```
>>> class TestClass(object):
        val1 = 100
        def __init__(self):
            self.val2 = 200
        def func(self,val = 400):
            val3 = 300
            self.val4 = val
            self.val5 = 500
```

提问：val1~val5
都是什么成员变量?

```
>>> inst = TestClass()
>>> TestClass.val1
100
>>> inst.val1
100
>>> inst.val2
200
>>> inst.val3
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    inst.val3
AttributeError: 'TestClass' object has no attribute '
val3'
```

```
>>> inst.val4
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    inst.val4
AttributeError: 'TestClass' object has no attribute '
val4'
```

面向对象程序设计：类

- 类成员与实例成员

```
>>> class TestClass(object):
        val1 = 100                          #类成员
        def __init__(self):
            self.val2 = 200                 #实例成员
        def func(self,val = 400):
            val3 = 300                      #局部变量
            self.val4 = val                 #self局部变量
            self.val5 = 500                 #self局部变量

>>> inst = TestClass()
>>> inst.func()
>>> print(inst.val4,inst.val5)
400 500
>>> inst.val3
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    inst.val3
AttributeError: 'TestClass' object has no attribute
'val3'
```

实例成员

面向对象程序设计：类

- 私有成员和公有成员
  - Python没有对私有成员提供严格的访问保护机制
  - 在定义类的成员时，如果成员名以两个下划线__或更多下划线开头而不以同样下划线结尾，则表示是私有成员
  - 私有成员在类的外部不能直接访问，需要通过调用对象的公开方法来访问，也可以通过Python中的特殊方式来访问
  - 公有成员在类的外部和内部均可以被直接访问

面向对象程序设计：类

- 私有成员和公有成员

```
>>> class TestClass:
        val1 = 100
        def __init__(self):
            self.val2 = 200
            self._val2 = 200
            self.__val2 = 200

>>> test = TestClass()
>>> test.val2                  #公有成员
200
>>> test._val2                 #受保护成员
200
>>> test.__val2                #私有成员
Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    test.__val2               #私有成员
AttributeError: 'TestClass' object has no attribute '
__val2'
>>> test._TestClass__val2          #特殊访问方式
200
```

面向对象程序设计：类

- Python 用下划线作为特殊变量的前缀和后缀
  - _xxx：不能用'from module import *'导入，这样的对象叫做保护变量，只有类对象和子类对象能访问这些变量
  - __xxx__：系统定义名字
  - __xxx：类中的私有成员，只有类对象自己能访问，连子类对象也不能访问到这个成员。但在对象外部可以通过对象名._类名__xxx来访问（Python改名机制）
  - Python中没有纯粹的C++意义上的私有成员

面向对象程序设计：类

- 关于下划线 _
  - 命令行模式下， _ 表示解释器最后一次显示的内容或最后一次语句正确执行的输出结果
  - 在程序中，可以用 _ 表示不关心该变量的值

```
>>> 8 * 9
72
>>> _ / 24
3.0
>>> a, b = 1, 2
>>> a + b
3
>>> _ + 4
7
```

```
>>> for _ in range(3):
        print('Hello!')

Hello!
Hello!
Hello!
>>> a, _ = divmod(70, 15)
>>> a
4
>>> a = 70 // 15
>>> a
4
```

面向对象程序设计：方法和属性

- 类成员：方法和属性
  - 受保护的：_xxx
  - 私有的：__xx
  - 公有的：默认格式

- 此外，方法还可通过修饰器形成：
  - 静态方法
  - 类方法
- 静态方法和类方法都通过类名和对象名调用，但不能访问属于对象的成员，只能访问属于类的成员

面向对象程序设计：方法和属性

- 方法和属性示例

```
>>> class Root:
        __total = 0                     #私有属性
        def __init__(self,v):          #构造函数
            self.__value = v
            Root.__total += 1
        def show(self):                 #普通实例方法
            print('self.__value:',self.__value)
            print('Root.__total:',Root.__total)
        @classmethod                    #修饰器，声明类方法
        def classShowTotal(cls):#类方法
            print(cls.__total)
        @staticmethod                   #修饰器，声明静态方法
        def staticShowTotal():
            print(Root.__total)
```

面向对象程序设计：方法和属性

```
>>> r = Root(3)
>>> r.classShowTotal()              #通过对象来调用类方法
1
>>> r.staticShowTotal()             #通过对象来调用静态方法
1
>>> r.show()
self.__value: 3
Root.__total: 1
>>> r2 = Root(3)
>>> Root.classShowTotal()           #通过类名来调用类方法
2
>>> Root.staticShowTotal()          #通过类名来调用静态方法
2
>>> Root.show()
Traceback (most recent call last):
  File "<pyshell#131>", line 1, in <module>
    Root.show()
TypeError: show() missing 1 required positional argument
: 'self'
>>> Root.show(r)
self.__value: 3
Root.__total: 2
>>> Root.show(r2)          #通过类名调用实例方法时，self显式
传递对象名
self.__value: 3
Root.__total: 2
```

面向对象程序设计：常用特殊方法

- **构造函数**：Python中类的构造函数是__init__()，用来为属性设置初值，在建立对象时自动执行。如果用户没设计构造函数，Python将提供一个默认的构造函数。
- **析构函数**：Python中类的析构函数是__del__()，用来释放对象占用的资源，在Python收回对象空间之前自动执行。如果用户没设计析构函数，Python将提供一个默认的析构函数。
- 在Python中，运算符重载是通过重写特殊函数来实现的

面向对象程序设计：常用特殊方法

方法名	功能说明
__init__()	构造函数，生成对象时调用
__del__()	析构函数，释放对象时调用
__new__()	类的静态方法，用于确定是否要创建对象
__add__()	+
__sub__()	-
__mul__()	*
__truediv__()	/
__floordiv__()	//
__mod__()	%
__pow__()	**
__eq__()、__ne__()、 __lt__()、__le__()、 __gt__()、__ge__()	==、!=、<、<=、>、>=
__lshift__()、__rshift__()	<<、>>
__iadd__()、__isub__()	+=、-=

面向对象程序设计：常用特殊方法

方法名	功能说明
__and__()、__or__()、 __invert__()、__xor__()	&、|、~、^
__pos__()、__neg__()	一元运算符+，一元运算符-
__contains__()	与成员测试运算符in对应
__radd__()、__rsub__()	反射加法、反射减法
__abs__()、__bool__() __bytes__()、__complex__() __dir__()、__divmod__() __float__()、__int__()	对应内置函数abs()、bool() bytes()、complex() dir()、divmod() float()、int()
__hash__()、__len__() __next__()、__reduce__() __reversed__()、__str__() __round__()	对应内置函数hash()、len() next()、reduce() reversed()、str() round()
__repr__()	打印、转换
__getitem__()、__setitem__()	按照索引获取值、按照索引赋值
__getattr__()、__delattr__()	读取对象指定属性、删除对象指定属性

面向对象程序设计：常用特殊方法

方法名	功能说明
__setattr__()	写（设置）对象指定属性的值
__getattribute__()	读取对象指定属性，如果同时定义了__getattr__()，则__getattr__()一般不会被调用
__base__()	返回该类的基类
__class__()	返回对象所属的类
__dict__()	返回对象所包含的属性与值的字典
__subclasses__()	返回该类的子类
__call__()	包含该特殊方法的类的实例可以像函数一样被调用
__get__()	定义这三个特殊方法中任何一个的类称为描述符，描述符对象一般作为其他类的属性使用，这三个方法分别在读取属性、修改属性或删除属性时被调用
__set__()	
__delete__()	
….	

面向对象程序设计：常用特殊方法

- 特殊方法的实际含义

方法	实际调用	功能说明
new(cls [,...])	instance = MyClass(arg1, arg2)	**new** 在创建实例的时候被调用
init(self [,...])	instance = MyClass(arg1, arg2)	**init** 在创建实例的时候被调用
pos(self)	+self	一元加运算符
neg(self)	-self	一元减运算符
invert(self)	~self	取反运算符
index(self)	x[self]	对象被作为索引使用
getattr(self,name)	self.name	读取一个对象的属性值
setattr(self, name, val)	self.name = val	对一个对象的属性值赋值
……		

面向对象程序设计：继承

- 继承是用于实现代码复用和设计复用的机制，是面向对象程序设计的重要特性之一。设计一个新类时，如能继承一个已有良好设计的类然后进行二次开发，无疑会大幅度减少工作量
- 在继承关系中，已有的、设计好的类称为***父类或基类***，新设计的类称为***子类或派生类***。派生类可以继承父类的公有成员，但是不能继承其私有成员

```
>>> class Person(object): #基类继承于object
        def __init__(self,name='',age=20):
                self.setName(name)
                self.setAge(age)
        def setName(self, name):
                if type(name) != str:
                        print('name must be string.')
                        return
                self.__name = name
        def setAge(self, age):
                if type(age) != int:
                        print('age must be integer.')
                        return
                self.__age = age
        def show(self):
                print('Name:',self.__name,'; Age:',self.__age)
```

面向对象程序设计：继承

- 子类示例

```
>>> class Teacher(Person):
        def __init__(self,name='',age=30,dept='AI'):
            super(Teacher, self).__init__(name, age,)
            #也可使用如下形式来初始化基类数据成员
            #Person.__init__(self, name, age)
            self.setDept(dept)
        def setDept(self, dept):
            if type(dept) != str:
                print('department must be string.')
                return
            self.__dept = dept
        def show(self):
            super(Teacher, self).show()
            print('Department: ',self.__dept)
```

面向对象程序设计：继承

- 继承示例

```
>>> zhang3 = Person('Zhang San', 19)
>>> zhang3.show()
Name: Zhang San ; Age: 19

>>> li4 = Teacher('Li Si',32,'CS')
>>> li4.show()
Name: Li Si ; Age: 32
Department:  CS

>>> li4.setAge(40)
>>> li4.show()
Name: Li Si ; Age: 40
Department:  CS
```

面向对象程序设计：继承

- 构造函数、私有方法和公有方法的继承原理

```
>>> class A(object):
        def __init__(self):          #构造方法可能被子类继承
            self.__private()
            self.public()
        def __private(self):         #私有方法在子类中不能直接访问
            print('__private() method of A')
        def public(self):            #公有方法在子类中可以直接访问、覆盖
            print('public() method of A')

>>> class B(A):                      #类B没有构造方法，会继承类A的构造方法
        def __private(self):         #这不会覆盖父类的私有方法
            print('__private() method of B')
        def public(self):            #覆盖了从类A继承来的公有方法
            print('public() method of B')

>>> b = B()
__private() method of A
public() method of B
```

面向对象程序设计：继承

```
>>> class C(A):
        def __init__(self):          #显式定义构造函数
            self.__private()
            self.public()
        def __private(self):
            print('__private() method of C')
        def public(self):
            print('public() method of C')


>>> c = C()
__private() method of C
public() method of C
```

面向对象程序设计：多重继承

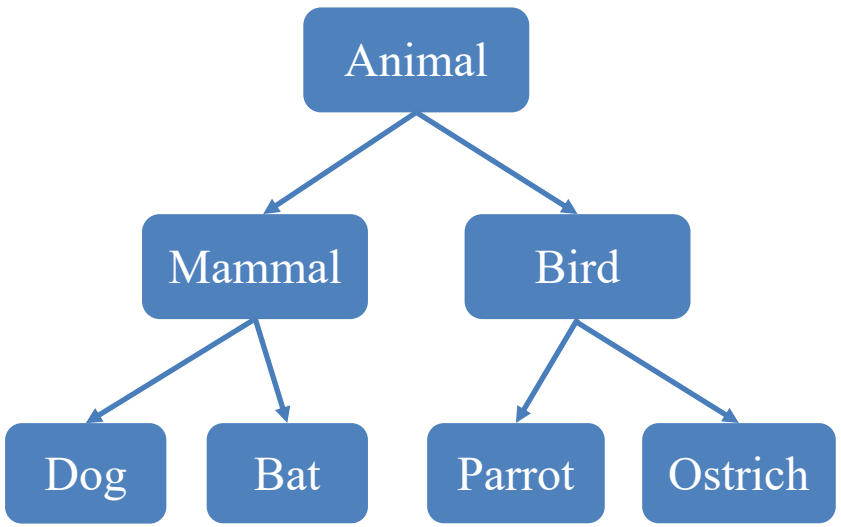
- Python的混入机制Mix-in
  - 某种功能单元的类被其他子类继承，将功能组合到子类中
  - 动态为自定义类及其实例对象增加新的属性和行为

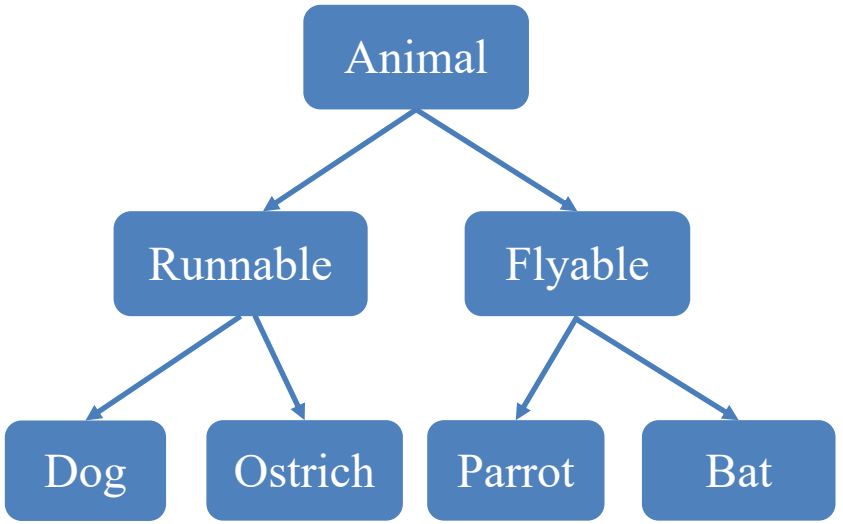
```
class Person:
        def __init__(self, name, gender, age):
                self.name = name
                self.gender = gender
                self.age = age
p = Person("小陈", "男", 18)
print(p.name) # "小陈"
```

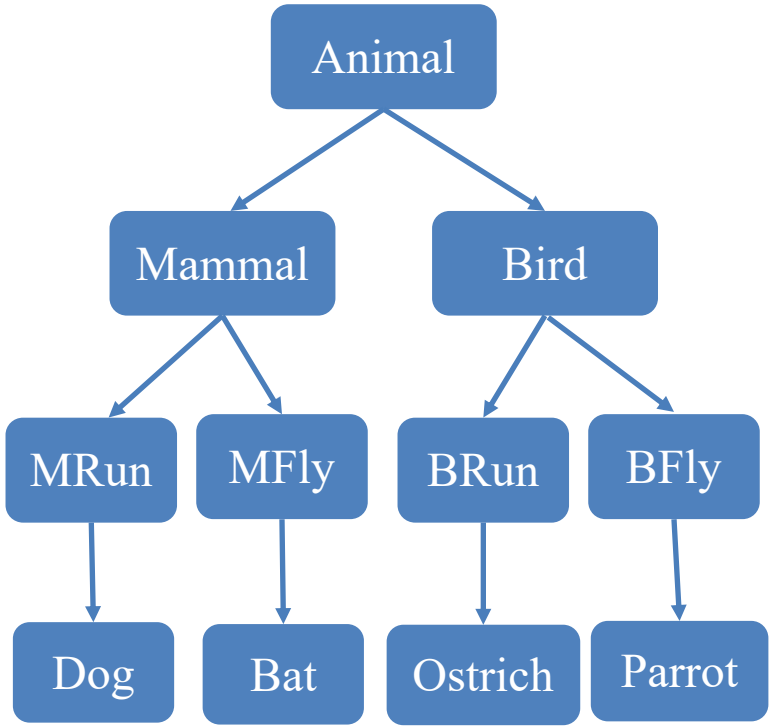
```
class MappingMixin:
        def __getitem__(self, key):
                return self.__dict__.get(key)
        def __setitem__(self, key, value):
                return self.__dict__.set(key, value)
class Person(MappingMixin):
        def __init__(self, name, gender, age):
                self.name = name
                self.gender = gender
                self.age = age
p = Person("小陈", "男", 18)
print(p['name']) # "小陈"
print(p['age']) # 18
```

面向对象程序设计：多重继承

- 以Animal类为例








```
class Dog(Mammal, Runnable):
    pass
```

面向对象程序设计：多态

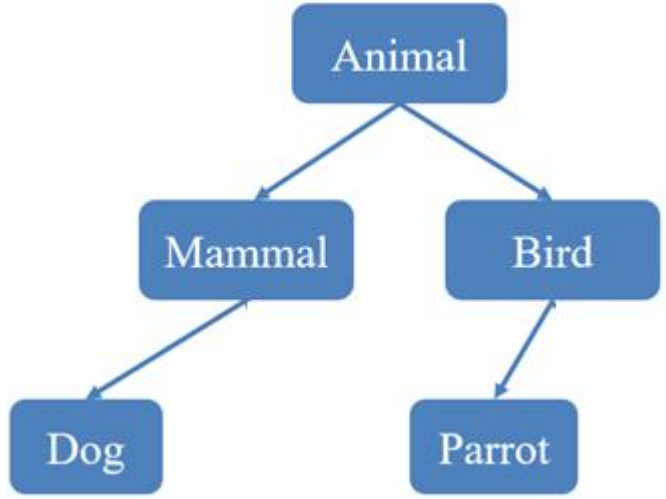
- **多态**，是指基类的一个方法在不同派生类对象中具有不同的表现和行为
- 派生类继承了基类的行为和属性之后，
  - 可能增加某些特定的行为和属性
  - 可能会对继承来的某些行为进行一定的改变
- Python运算符自带多态
  - 同一个运算符可用不同类型的操作数
  - 对不同类型的操作数可以有不同的功能
  - 此种多态是通过特殊方法与运算符重载实现的

面向对象程序设计：多态

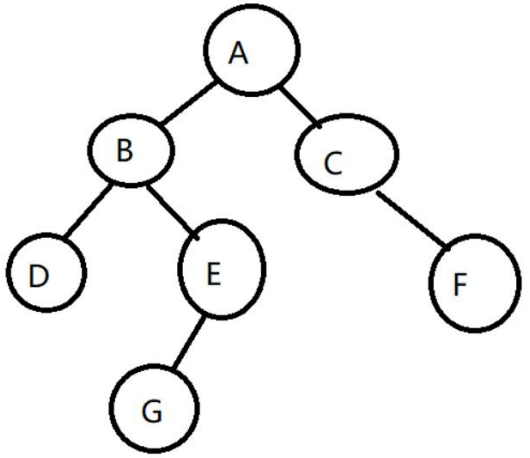
```
>>> class Animal(object):          #定义基类
        def show(self):
            print('It is an animal.')

>>> class Cat(Animal):             #定义派生类
        def show(self):
            print('It is a cat.') #覆盖基类的方法

>>> class Dog(Animal):             #定义派生类
        def show(self):
            print('It is a dog.') #覆盖基类的方法

>>> class Others(Animal):          #定义派生类
        pass                        #没有覆盖基类方法

>>> x = [item() for item in (Animal, Cat, Dog, Others)]
>>> for i in x:
        i.show()                    #遍历基类和派生类

It is an animal.
It is a cat.
It is a dog.
It is an animal.
```

作业题

- 类和继承
  - 定义一个抽象类Animal
  - 定义Mammal和Bird两个类继承Animal, 分别定义多个这两个类特有的方法和成员
  - 定义Dog和Parrot两个类分别继承Mammal和Bird，在Dog类中定义方法访问和修改Mammal类中的成员



- 实现二叉树的前序遍历和层次遍历
  - 用一个类定义二叉树
  - 输出前序遍历：A, B, D, E, G, C, F
  - 输出层次遍历：A, B, C, D, E, F, G



作业题

- 用面向对象编程实现班级排名和成绩排名
  - 定义一个学生类，该类的成员有某门课的考试分数
  - 定义一个班级类，成员为学生；定义一个学校类，成员为班级
  - 在学校类定义方法实现对所有学生的成绩排序，输出排序后的 [(学生名字，分数), ...]
  - 不采用面向对象编程，即不用类实现对学校所有学生的成绩排序