**python 面向对象**

* 面向对象的三大特性：继承 封装 多态
  + 继承
    - 语法：class A:passclass B(A):pass
      * # B 继承 A， A是父类，B是子类# A 是 父类 基类 超类# B 是 子类 派生类
    - # 子类可以使用父类中方法和静态变量# 子类也可以重写从父类中继承的方法# 当子类和父类的方法重名时，子类只使用子类的方法；# 如果子类的方法中调用父类的方法：通过父类名.方法名(self)调用，# 注意：此self 传的是子类自己的self
      * class Animal: def \_\_init\_\_(self, name, food): self.name = name self.food = food self.blood = 100 self.waise = 100 def eat(self): print(f'{self.name} is eating {self.food}') def drink(self): print(f'{self.name} is drinking') def slep(self): print(f'{self.name} is sleeping')class Cat(Animal): def eat(self): self.blood += 100 Animal.eat(self) # 如果自己有还想用父类的： 直接在子类的方法中调用父类的方法: 父类名.方法名(self) def climb\_tree(self): print(f'{self.name}开始睡觉') self.drink()class Dog(Animal): def eat(self): self.waise += 100 Animal.eat(self) def house\_keep(self): print(f'{self.name}开始睡觉')
    - 继承中 父类和子类方法的选择
      * 子类的对象，如果去调用方法，永远优先调用子类自己的。# 如果自己有，用自己的# 如果自己没有，用父类的# 如果自己有还想调用父类的：直接在子类的方法中调用父类的方法：父类名.方法名(self)
    - 如何给子类定义个性话属性# 派生属性，也就是子类独有的
      * 在子类的\_\_init\_\_方法,首先通过 父类.\_\_init\_\_(self, 通用属性)，完成通用属性的初始化再给子类自己添加派生属性
        + class Animal: def \_\_init\_\_(self, name, food): self.name = name self.food = food self.blood = 100 self.waise = 100
        + class Cat(Animal): def \_\_init\_\_(self, name, food, eye\_color): Animal.\_\_init\_\_(self,name,food) # 调用了父类的初始化，去完成一些通用属性的初始化 self.eye\_colof = eye\_color # 派生属性
    - 单继承和多继承
      * 单继承# 一个类只有一个父类
        + class C:passclass B(C): def func(self): print('in B')class A(B):passA().func() # in B# A() 就是实例化A类，等价于 a = A(), a.func()
      * 多继承#一个类有多个父类# python 语言的特点，可以在面向对象中支持多继承# java 不支持多继承
        + 语法：class A: func(self):print('in A')class B:pass func(self):print('in B')class C:passclass D(A,B,C) # 多继承D().func() # in A # 多继承就是 离谁近，先找谁# D继承顺序为A，B，C， 所以先执行A类的func
      * 单继承和多继承，什么时候调用子类，什么时候调用父类？
        + 单继承：# 调子类：子类自己有的时候# 调父类：子类自己没有的时候# 调子类和父类：子类和父类都有，在子类中调用父类的， # 例如在子类的方法中先通过父类名.父类方法调用父类的属性或方法
        + 多继承：# 一个类有多个父类，在调用父类方法的时候，按照继承顺序，先继承的就先寻找
      * 继承中，在每一个self调用方法时，永远只看这个self属于谁
      * 子类继承父类，子类实例化过程
    - 新式类和经典类
      * 新式类
        + 只要继承object类，就是新式类python3中所有的类都继承object类，都是新式类python2中继承object类的就是新式类

# 在py3中# class A:pass # 新式类# class B(object):pass # 新式类

* + - * 经典类
        + python2中不主动继承object的类，就是经典类python3中没有经典类

# 在py2中# class A:pass # 经典类# class B(object):pass # 新式类

* + - * 经典类和新式类的区别
        + 单继承

1. 在单继承方面，无论是经典类和新式类，都是深度优先# 即： 从子类开始往父类走

class A: def func(self): passclass B(A): def func(self):passclass C(B): def func(self):passclass D(C): def func(self):pass# 寻找某一个方法的顺序：D - > C - > B -> A# 越往父类走，是深度

* + - * + 多继承

1. 在走到一个点，下一个点即可以从深度走，也可以从广度走的时候，总是先走广度，再走深度，广度优先2. 在经典类中，都是深度优先，总是在一条路走不通之后，再换一条路，走过的点不会再走

乌龟模型

深度优先

广度优先

C3算法

# C3算法(广度优先 乌龟模型)# A(O) = [AO]# B(A) = [BAO]# C(A) = [CAO]# D(B) = [DBAO]# E(C) = [ECAO]# F(D, E) = c3(D(B) + E(C))# = [F] + [DBAO] + [ECAO]# F = [DBAO] + [ECAO]# FD = [BAO] + [ECAO]# FDB = [AO] + [ECAO]# FDBE = [AO] + [CAO]# FDBEC = [AO] + [AO]# FDBECA = [O] + [O]# FDBECAO = [] + []

# C3算法的内容： # 如果是单继承，那么总是按照从子类 - > 父类的顺序来计算查找顺序 # 如果是多继承，需要按照自己本类，父类1的继承顺序，父类2的继承顺序... # merge的规则： # 如果一个类出现在从左到右所有顺序的最左侧，并且没有在其他位置出现，那么先提出来作为继承顺序中的一个 # 或 一个类出现在从左到右顺序的最左侧，并没有在其他顺序中出现，那么先提出来作为继承顺序中的一个 # 如果一个从左到右第一个顺序中的第一个类处在后面且不是第一个，那么不能提取，顺序向后继续找其他顺序中符合上述条件的类

类名.mro()

在新式类中，通过 类名.mro() 可以查看类的继承关系

class A: def func(self):pass # print('A')class B(A):pass # def func(self): # print('B')class C(A):pass # def func(self): # print('C')class D(B, C): pass # def func(self): # print('D')d = D()d.func() # [<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

* 多态
  + 一个类型表现出来的多种状态# 例如支付接口 表现出来的 微信支付和支付宝支付着两种状态# 在java中，参数必须强制指定类型，如果想让两个类型的# 对象都可以传，那么必须让这两个类型继承同一个父类，# 形参在指定类型时，使用父类来指定# 例如：# class 车类# 非机动车类 继承 车类# 机动车类 继承 车类# 在创建对象时，参数的指定类型 指定为 车类，# 这样无论是机动车和非机动车，都可以接收进来
  + 鸭子类型
    - # 鸭子类型 # 子类继承父类,我们说子类是父类类型的(猫类继承动物,我们说猫也是动物) # 在python中,一个类是不是属于某一个类型 # 不仅仅可以通过继承来完成 # 还可以是不继承,但是如果这个类满足了某些类型的特征条件 # 我们就说它长得像这个类型,那么他就是这个类型的鸭子类型# 简单说就是，一个类没有继承另一个类，但是实现了另一个类的某个特定的方法，# 那这个类就是另一个类的鸭子类型
* 封装
  + # 语法# 给类中的静态变量，方法，方法中的变量加上 \_\_双下划线# 封装就是把属性或方法装起来，限制外界的访问，保护数据
  + 广义上的封装
    - # 把属性或方法装起来，外界不能直接调用，要通过类的名字来调用# 我们平时写的类 都属于广义上的封装
  + 狭义上的封装
    - 把属性或方法藏起来，外面不能调用，只能在内部偷偷调用
    - 私有的三种情况
      * # 不想让你看，也不想让你改
        + class User: def \_\_init\_\_(self, name, pwd): self.name = name self.\_\_pwd = pwdxb = A('alex', '123456')print(xb.\_\_pwd) # AttributeError# AttributeError: 'User' object has no attribute '\_\_pwd'
      * # 可以让你看，但不让你改
        + class User: def \_\_init\_\_(self, name, pwd): self.name = name self.\_\_pwd = pwd # 私有的对象属性 def get\_pwd(self): # 表示用户不能改，只能看 私有+某个get方法实现 return self.\_\_pwdxb = User('alex', '123456')print(xb.get\_pwd())
      * 可以看，也可以改，但是要按照我的要求改
        + import reclass User: def \_\_init\_\_(self, name, pwd): self.name = name self.\_\_pwd = pwd def get\_pwd(self): return self.\_\_pwd def change\_pwd(self, newpwd): # 表示用户必须调用我们自定义的修改方式修改 私有+change方法实现 if re.match('[a-zA-Z]', newpwd): self.\_\_pwd = newpwdxb = User('alex', '123456')print(xb.get\_pwd())xb.change\_pwd('abadddaa')
    - 封装的语法
      * 私有的静态变量
        + class User: \_\_Country = 'China' # 私有的静态变量 def func(self): print(self.\_\_Country) # 私有的静态变量在了类的内部可以使用，自动把这句话所在类的名字添加到私有变量名前面 # print(self.\_User\_\_Country)xb = User()# print(xb.\_\_Country) # AttributeError: 'User' object has no attribute '\_\_Country'print(User().func()) # China
      * 私有的实例变量
        + # print(xb.\_\_dict\_\_) # {'name': 'alex', '\_User\_\_pwd': '123456'}# 私有的实例变量 \_\_pwd ，实际是在名字前加了\_类名
      * 私有的绑定方法
        + import hashlibclass User: def \_\_init\_\_(self, name, pwd): self.name = name self.\_\_pwd = pwd # 私有的实例变量 def \_\_get\_md5(self): # 私有的绑定方法 m = hashlib.md5(self.name.encode('utf-8')) m.update(self.\_\_pwd.encode('utf-8')) return m.hexdigest() def get\_pwd(self): # 通过 私有实例变量 + 私有的绑定方法,对外界间接隐藏私有变量 return self.\_\_get\_md5()xb = User('alex', '123456') # 94e4ccf5e2749b0bfe0428603738c0f9print(xb.get\_pwd())print(xb.\_\_dict\_\_) # {'name': 'alex', '\_User\_\_pwd': '123456'}
    - 私有的特点
      * 能不能在类的内部使用？ 可以
        + class Foo(object): def \_\_init\_\_(self): self.\_\_func() # 此处实际是 self.\_Foo\_\_func(), # Son()实例化并调用init时，实际执行的是 self.\_Foo\_\_func(), 所以最终打印的 in Foo def \_\_func(self): print('in Foo')class Son(Foo): def \_\_func(self): # 此处实际是 self.\_Son\_\_func() print('in Son')Son() # in Foo
      * 能不能在类的外部使用？ 不可以
        + class Foo: \_\_Country = 'China' # 私有静态变量x = Foo()print(x.\_\_Country) # AttributeError: 'Foo' object has no attribute '\_\_Country'
      * 能不能在子类中使用？ 不可以
        + class Foo: def \_\_func(self): # 2. 父类中中的私有方法是 \_Foo\_\_func()，所有没找到 \_Son\_func,报错 print('in Foo')class Son(Foo): def \_\_init\_\_(self): self.\_\_func() # 1. 初始执行 self.\_Son\_\_func()，本类中没有去父类找Son() # AttributeError: 'Son' object has no attribute '\_Son\_\_func'
    - 封装的原理
      * 如何变形？# \_类名\_\_私有属性或方法
      * 在哪里用的时候变形？# 1. 外界调用或修改私有属性或方法时# 2. 子类中没有，在父类中寻找私有属性或方法时# 3. 子类init初始化时，用到父类的私有属性或方法时
    - 类中的变量的级别？哪些是python支持的？
      * public 共有的# py支持
        + 类内类外都能用，父类子类都能用
      * protect 受保护的# py不支持
        + 类内能用，父类子类都能用，类外不能用
      * private 私有的# py支持
        + 本类的类内部都能用，其他地方都不能用
* 类的补充
  + object类所有在python3中的类，都继承自object类
    - dir(类名)：查询一个名字列表类名.\_\_dict\_\_：查出一个字典，key 属性，value 属性值
  + 特殊的类属性
    - 类名.\_\_name\_\_# 类的名字(字符串)类名.\_\_doc\_\_# 类的文档字符串类名.\_\_base\_\_# 类的第一个父类(在讲继承时会讲)类名.\_\_bases\_\_# 类所有父类构成的元组(在讲继承时会讲)类名.\_\_dict\_\_# 类的字典属性类名.\_\_module\_\_# 类定义所在的模块类名.\_\_class\_\_# 实例对应的类(仅新式类中)
  + from types import FunctionType, MethodTypeFunctionType. 函数MethonType 方法instance. type. 判断类型以及判断类与类的关系
    - class A: def func(self): print('in func')print(A.func) # 函数 # <function A.func at 0x106b61550>a = A()print(a.func) # 方法 # <bound method A.func of <\_\_main\_\_.A object at 0x106c2df70>>print(isinstance(a.func, FunctionType)) # Falseprint(isinstance(a.func, MethodType)) # Trueprint(isinstance(A.func, FunctionType)) # Trueprint(isinstance(A.func, MethodType)) # False结论： 对象名.函数名 是方法 MethodType类名.函数名 是函数 FunctionType
* self 关键字
  + # self作为形参接收的是内存中的一块空间，# 例如：# class A:#. def \_\_init\_\_(self):pass# a = A()# 在实例化A() 的时候，内存中开辟了一块内存空间，# 并指向了实例化对象 a, # 在调用 \_\_init\_\_方法的时候, 指向a对象的内存空间# 作为参数传递给了init方法的形参 self, # 而self接收的参数，也就保存在它所在的空间中了
* super 关键字
  + 在多继承中，super训训mro算法# super 是按照mro顺序来寻找当前类的下一个类# 在py3中不要传参数，自动就帮我们寻找当前类的mro顺序的下一个类中的同名方法# 在pyt2中的新式类中，需要我们主动传递参数super(子类的名字,子类的对象).函数名()# 这样才能够帮我们调用到这个子类的mro顺序的下一个类中的方法# 在py2的经典类中，并不支持用super来找下一个类# 在D类中找super的func,那么可以这样写 super().func()# 也可以这样写 super(D, self).func(), (并且在py2的新式类中必须这样写)
    - # class A(object):# def func(self):# print('A')# class B(A):# def func(self):# super().func()# print('B')# class C(A):# def func(self):# super().func()# print('C')# class D(B, C):# def func(self):# super().func()# print('D')# D().func() # A, C, B, D# mro() 顺序 D,B,C,A# mro() 顺序 D,B,C,A
  + 在单继承中，super就是找父类
    - # 在单继承的程序中，super就是找父类class User: def \_\_init\_\_(self,name): self.name = nameclass VIPUser(User): def \_\_init\_\_(self, name, level, start\_date, end\_date): # User.\_\_init\_\_(self,name) super().\_\_init\_\_(name) # 单继承中推荐使用 super() 访问父类的方法， 本质上还是使用 mro() 方法 # super(VIPUser, self).\_\_init\_\_(name) self.level = level self.start\_date = start\_date self.end\_date = end\_datetb = VIPUser('tb', 6, '2022-03-22', '2022-04-03')print(tb.\_\_dict\_\_)
* 抽象类
  + # 抽象类 是一个开发规范，# 约束了它的所有子类必须实现一些和他同名的方法
  + 归一化设计
    - # 归一化设计，即对应相似的功能，# 对外提供统一的调用方式# 例如 第三方支付接口，# 微信支付/支付宝支付/银联等，对外提供统一的pay方法
  + raise 关键字
    - raise NotImplementedError('提示信息')
    - 当不满足某些条件时，可以用来主动抛出异常信息
  + 实现抽象类的两种方式
    - # 方式一：# 通过定义一个抽象类型# 在抽象类里指定 需要实现的函数或初始化参数，# 并通过 raise NotImplementedError(msg) 主动抛出异常# 子类主动继承这个抽象类，并且必须实现抽象类中定义的方法，# 如果 子类未实现，会抛出异常信息提示抽象类中的方法未实现
      * # 抽象类class Payment: def pay(self, money): raise NotImplementedError(self.\_\_class\_\_, '未实现pay方法')class Wechat(Payment): def \_\_init\_\_(self, name): self.name = name def pay1(self, money): # Wechat 类为实现抽象类执行的方法名 pay print(f"{self.name}通过微信成功支付了{money}元")class Alipay(Payment): def \_\_init\_\_(self, name): self.name = name def pay(self, money): print(f"{self.name}通过支付宝成功支付了{money}元")# a = Wechat('alex')# a.pay(1)# 归一化设计# 统一外部访问 支付接口的调用方式def pay(name, money, kind): if kind.lower() == 'wechat': obj = Wechat(name) elif kind.lower() == 'alipay': obj = Alipay(name) obj.pay(money)def main(): pay('tbjx', 200.0, 'alipay') # pay('tbjx', 100, 'Wechat')'''Traceback (most recent call last): File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/04抽象类实现方式一.py", line 52, in <module> main() File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/04抽象类实现方式一.py", line 49, in main pay('tbjx', 100, 'Wechat') File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/04抽象类实现方式一.py", line 44, in pay obj.pay(money) File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/04抽象类实现方式一.py", line 16, in pay raise NotImplementedError(self.\_\_class\_\_, '未实现pay方法')NotImplementedError: (<class '\_\_main\_\_.Wechat'>, '未实现pay方法')'''if \_\_name\_\_ == '\_\_main\_\_': main()
    - # 方式二# 通过系统内置的模块来实现# 1. 导包 from abc import ABCMeta, abstractmethod# 2. 抽象类 继承 metaclass = ABCMeta# 3. 对指定方法添加 装饰器 @abstractmethod
      * ''' 抽象类# 抽象类 # 是一个开发的规范，约束它的所有子类必须实现一些和它同名的方法# 支付程序 # 微信支付 url连接，告诉你参数是什么格式 # {'username:'用户名', 'money':200} # 支付宝支付 url连接，告诉你参数是什么格式 # {'uname:'用户名', 'price':200} # 苹果支付'''# 抽象类 实现方式二from abc import ABCMeta, abstractmethodclass Payment(metaclass=ABCMeta): @abstractmethod def pay(self, money):passclass Wechat(Payment): def \_\_init\_\_(self, name): self.name = name def pay1(self, money): # Wechat 类为实现抽象类执行的方法名 pay print(f"{self.name}通过微信成功支付了{money}元")class Alipay(Payment): def \_\_init\_\_(self, name): self.name = name def pay(self, money): print(f"{self.name}通过支付宝成功支付了{money}元")# a = Wechat('alex')# a.pay(1)'''a = Wechat('alex')TypeError: Can't instantiate abstract class Wechat with abstract method pay'''# 归一化设计# 统一外部访问 支付接口的调用方式def pay(name, money, kind): if kind.lower() == 'wechat': obj = Wechat(name) elif kind.lower() == 'alipay': obj = Alipay(name) obj.pay(money)def main(): pay('tbjx', 200.0, 'alipay') # pay('tbjx', 100, 'Wechat')'''Traceback (most recent call last): File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/05抽象类实现方式二.py", line 58, in <module> main() File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/05抽象类实现方式二.py", line 55, in main pay('tbjx', 100, 'Wechat') File "/Users/erwei.zheng/PycharmProjects/xiaoBaiJuJia/pyBasic/day25\_经典类新式类多继承抽象类多态/homework/05抽象类实现方式二.py", line 46, in pay obj = Wechat(name)TypeError: Can't instantiate abstract class Wechat with abstract method pay'''if \_\_name\_\_ == '\_\_main\_\_': main()
    - # 抽象类 # 为什么要用抽象类 : 为了规范子类必须实现和父类的同名方法 # 抽象类用到的格式(至少会写一种,两种见到了都要认识) # 不需要模块的 # class 父类: # def 子类必须实现的方法名(self,参数们): # raise NotImplementedError('提示信息') # class 子类(父类): # def 父类要求实现的方法(self,参数们): # print('''code''') # 需要模块的 # from abc import ABCMeta,abstractmethod # class 父类(metaclass = ABCMeta): # @abstractmethod # def 子类必须实现的方法名(self,参数们):pass # class 子类(父类): # def 父类要求实现的方法(self,参数们): # print('''code''')
* 类中的三个装饰器propertyclassmethodstaticmethod
  + @property
    - # 用处：# 把类中的方法添加@property，# 可以把方法伪装成属性，# 在调用这个方法的时候不用加(), 就可以直接调用# 被property装饰的方法，不能有参数
      * from math import piclass Circle: def \_\_init\_\_(self, r): self.r = r @property # 将绑定方法伪装成属性，直接访问 def area(self): return pi \* self.r \*\* 2c = Circle(5)print(c.area) # 78.53981633974483print(c.\_\_dict\_\_) # {'r': 5}
    - property的三个应用场景
      * # 场景一# 对外界间接提供处理过的属性# 例如 年龄，一般通过生日去推算，不会直接存age属性
        + import timeclass Person: def \_\_init\_\_(self, name, birth): self.name = name self.birth = birth @property # 被property装饰的方法，不能有参数 def age(self): return time.localtime().tm\_year - self.birthtb = Person('tbjx', 1990)print(tb.age) # 32
      * # 场景二# 和私有属性合作# 对外提供对私有属性的查询或加工过的数据
        + import hashlibclass User: def \_\_init\_\_(self, name, pwd): self.name = name self.\_\_pwd = pwd @property # 对外提供经过加密的私有属性数据 def pwd(self): return hashlib.md5(f'{self.\_\_pwd}{self.\_\_pwd}'.encode('utf-8')).hexdigest()xb = User('tbjx', '123456')print(xb.pwd) # ea48576f30be1669971699c09ad05c94print(xb.\_\_dict\_\_) # {'name': 'tbjx', '\_User\_\_pwd': '123456'}
      * # 场景三# 和类的静态变量合作# 通过静态变量，对外返回经过加工的数据
        + class Goods: Discount = 0.80 # 商品折扣率 def \_\_init\_\_(self, name, orignal\_price): self.name = name self.\_\_price = orignal\_price @property def price(self): return round(self.\_\_price \* self.Discount, 2)apple = Goods('apple', 14.5)print(apple.price) # 11.6
    - property的进阶
      * # @被property注解的函数名.setter# 被setter装饰的方法可以接收参数，并修改私有属性# 注意：方法名必须与property装饰的方法相同
        + class Goods: Discount = 0.80 def \_\_init\_\_(self,name, orignal\_price): self.name = name self.\_\_price = orignal\_price @property def price(self): return round(self.Discount \* self.\_\_price, 2) @price.setter # 方法名需与property装饰的方法相同 def price(self, newprice): if isinstance(newprice, int) or isinstance(newprice, float): self.\_\_price = newpriceapple = Goods('apple', 15.5)print(apple.price) # 12.4apple.price = 18print(apple.price) # 14.4print(apple.\_\_dict\_\_) # {'name': 'apple', '\_Goods\_\_price': 18}
      * # @被property注解的函数名.deleter# 被 deleter装饰过后# 外部可以使用 del 方法名，删除属性# 此处并不是真的删除，只是调用了被 deleter装饰的方法而已
        + class Goods: Discount = 0.80 def \_\_init\_\_(self, name, orignal\_price): self.name = name self.\_\_price = orignal\_price @property def price(self): return round(self.Discount \* self.\_\_price, 2) @price.setter def price(self, newprice): if isinstance(newprice, int) or isinstance(newprice, float): self.\_\_price = round(newprice, 2) @price.deleter # 执行被deleter装饰的方法，外界创建的对象会无法访问 被property装饰的属性 def price(self): del self.\_\_priceapple = Goods('apple', 15.5)print(apple.price) # 12.4del apple.price # 删除 apple.price, 并不能真的删除什么，只是调用对应的被 @price.deleter装饰的方法而已# print(apple.price) # AttributeError: 'Goods' object has no attribute '\_Goods\_\_price'print(apple.\_\_dict\_\_) # {'name': 'apple'}, apple对象的 price 属性被删除
* 反射
  + # 定义：# 用字符串数据类型的名字来操作这个名字对应的函数/实例变量/绑定方法等
  + 反射的使用场景
    - # 1. 反射模块中的内容# 被导入的模块# 当前模块的py文件 -- 脚本# 模块中的静态变量/实例变量/绑定方法
      * # a.py 文件class User: Birth = 1980 def \_\_init\_\_(self, name, age): self.name = name self.age = age def func(self): print(f'我叫{self.name}, 今年{self.age}岁，{self.Birth}年出生.')alex = User('alex', 42)lis = [123, 456]
      * import a # 导入模块aimport sysprint(getattr(a, 'User'), a.User) # 导入的a 模块中的 User类print(getattr(a, 'lis')) # 导入模块中的 静态变量print(getattr(a, 'alex')) # 导入模块中的实例对象print(getattr(a, 'alex').name) # 导入模块的实例对象的属性getattr(a, 'alex').func() # 导入模块的实例对象的绑定方法
    - # 2.反射本文件中的内容# 本py文件中的变量/类或对象的属性和方法# sys.modules - > 查看系统内置模块，列表的形式# 本文件在内置模块的 key 是 '\_\_main\_\_'
      * class A:passa = 'abc'b = [1, 2, 3]def func(): print(a, b)print(sys.modules['\_\_main\_\_'])print(getattr(sys.modules['\_\_main\_\_'], 'a')) # abc 本模块中的变量 aprint(getattr(sys.modules['\_\_main\_\_'], 'b')) # [1, 2, 3] 本模块中的变量 bprint(getattr(sys.modules['\_\_main\_\_'], 'func')) # <function func at 0x102d26550> 本模块 func的内存地址getattr(sys.modules['\_\_main\_\_'], 'func')() # abc [1, 2, 3] 反射本模块中的方法 func的内存地址,并调用print(getattr(sys.modules['\_\_main\_\_'], 'A')) # <class '\_\_main\_\_.A'> 反射的本模块中的A类
    - # 3.反射对象的属性或方法
      * # 1. 反射对象的实例变量,绑定方法class User: def \_\_init\_\_(self, name, age): self.name = name self.age = age def func(self): print(f'我叫{self.name},今年{self.age}岁.')xb = User('alex', 18)# 访问对象的实例变量print(getattr(xb, 'name')) # alexprint(getattr(xb, 'age')) # 18# 访问对象的绑定方法print(xb.func, getattr(xb, 'func')) # <bound method User.func of <\_\_main\_\_.User object at 0x10b933fd0>># <bound method User.func of <\_\_main\_\_.User object at 0x10b933fd0>>getattr(xb, 'func')() # 我叫alex,今年18岁.
    - # 4.反射类的静态变量/绑定方法/其他方法..
      * class User: Country = 'China' # 静态变量 def \_\_init\_\_(self, name, age): self.name = name self.age = age def func(self): # 绑定方法 print(f"我叫{self.name}, 今年{self.age}岁, 国籍是:{self.Country}")print(getattr(User, 'Country')) # China 访问类中的静态变量print(getattr(User, 'func')) # <function User.func at 0x106d9f040> 访问类中的绑定方法的内存地址getattr(User('alex', 18), 'func')() # 我叫alex, 今年18岁, 国籍是:China 调用类中的绑定方法
  + hasattr
    - # hasattr(类模块对象名等等, 字符串形式的变量)# 判断某一个类或对象能不能反射某一个变量
      * # hasattr 判断某一个类或对象能不能反射某个变量# callable 判断某个反射的变量或内存地址，是不是可调用或可执行的# getattr 通过字符串类型的名字，来操作这个名字对应的变量/对象/绑定方法等class User: def \_\_init\_\_(self, name, age): self.name = name self.age = age def func(self): print(f'我叫{self.age}, 今年{self.age}岁.')alex = User('alex', 18)if hasattr(alex, 'func'): # 判断 func 是否可以反射 if callable(getattr(alex, 'func')): # 判断 func是否可调用 getattr(alex, 'func')() # 如果可调用，就执行
  + getattr
    - # getattr(类模块对象名等等, 字符串形式的变量)
  + callable
    - # callable# 2.如果能反射，还想要执行，还需要判断一下这个反射# 的变量或内存地址是不是可调用或可执行的，# 如果是，才能加括号调用
  + 反射实例
    - # 在归一化设计中，通过反射获取类的内存地址，# 获取到地址以后，再实例化对象，# 最后通过实例化对象调用统一的接口或方法# 简化对外接口的判断条件
      * # 通过反射 实现支付类的 归一化设计class Payment: def pay(self, money): ''' 支付接口统一实现pay方法''' raise NotImplementedError(self.\_\_class\_\_,'未实现Payment中的pay方法')class Wechat(Payment): def \_\_init\_\_(self, name): self.name = name def pay(self, money): if isinstance(money, int) or isinstance(money, float): print(f'{self.name}通过微信支付了{money}元')class Alipay(Payment): def \_\_init\_\_(self, name): self.name = name def pay(self, money): if isinstance(money, int) or isinstance(money, float): print(f'{self.name}通过支付宝支付了{money}元')# 归一化设计import sysdef pay(name, price, kind): ''' :param name: :param price: :param kind: 类名 :return: 判断kind是否可以反射， 如果可以获取kind对应内存空间并实例化类，在调用类中的pay方法，return结果 ''' if hasattr(sys.modules['\_\_main\_\_'], kind): return getattr(sys.modules['\_\_main\_\_'], kind)(name).pay(price)pay('alex', 10, 'Wechat') # alex通过微信支付了10元pay('alex', 10.10, 'Alipay') # alex通过支付宝支付了10.1元
* 类 对象/实例 实例化
  + # 面向对象 # 类 对象/实例 实例化 # 类是具有相同属性和相似功能的一类事物 # 一个模子，大概的范围，抽象 # \*\*\* # 你可以清楚的知道这一类事物有什么属性，有什么动作 # 但是你不知道这些属性有什么值 # 对象 == 实例 # 给类中所有的属性填上具体的值就是一个对象或者实例 # 只有一个类，但是可以有多个对象都是这个类的对象 # 实例化 # 实例 = 类名() # # 首先开辟空间， # # 再调用init方法， # # 调用init的时候，总是把新开的空间作为参数传递给self # init方法中一般完成：把属性的值存储在self空间里 - 对象的初始化 # self这个地址会作为返回值，返回给实例 # 方法：定义在类里的函数，并且还带有self参数 # 实例变量：self.名字
  + 类和对象之间的关系
    - # 类和对象之间的关系？ # 类 是一个大范围 是一个模子，它约束了事物有哪些属性，但是不能约束具体的值 # 对象 是一个具体的内容，是模子的产物，它遵循了类的约束，同时给属性赋上具体的值
  + 类实例化的过程
    - 1. 类名() 之后第一件事：开辟一块内存空间
    - 2.调用 \_\_init\_\_把空间的内存地址作为形参self的参数，传递给函数内部
    - 3.所有这个类的对象需要使用的属性都需要和self关联起来
    - 4.执行完init中的逻辑后，self变量会自动的被返回到调用处，也就是发生实例化的地方； 例如：class A:passA() # 实例化A类，此时它就拿到了A类中的属性和方法A() 等价于 obj = A(), 此时的A()可以调用类中的属性和方法
  + 类的命名空间和组合
    - 命名空间和组合总结
      * 命名空间
        + 类的命名空间

静态变量

绑定方法

* + - * + 对象的命名空间

类指针

对象的属性(实例化的变量)

* + - * + 调用习惯

类名.静态变量

对象.静态变量# 对象调用静态变量的时候，不能对变量进行赋值操作（对象.静态变量 = 123）# 静态变量只能由类自己来修改

* + - * 组合
        + 一个类的对象，是另一个类的对象的属性
        + 两个类之间有什么有关系的地方： # 例如 班级有学生，学生有班级，班级有课程 学生有成绩# 其实就是类似于数据库里的外键
    - 类的成员和空间
      * # 类中的变量是静态变量# 对象中的变量只属于对象本身，每个对象有属于自己的空间来存储对象的变量# 当使用对象名去调用某一个属性的时候，会优先在自己的空间寻找，找不到再到的对应的类中寻找# 如果自己没有就引用类的，如果类也没有，就抛异常
      * 子主题
    - 类中静态变量的用处
      * # 如果一个变量是所有对象共享的值，那么这个变量应该定义成静态变量# 所有和静态变量相关的增删改查都应该使用类名来处理，# 而不应该用 对象名来直接修改静态变量
        + class A: Country = '中国' # 静态变量/静态属性 存储在类的命名空间里的 def \_\_init\_\_(self): # 绑定方法 存储在类的命名空间里的 pass def func1(self): print(self) def func2(self):pass Country = '印度'
        + # 练习题：实现一个类，能够自动统计这个类中实例化了多少个对象class A: count = 0 # 类的静态变量 def \_\_init\_\_(self): A.count += 1 # 每次创建对象时，都是初始化\_\_init\_\_, 通过类来修改类中的静态变量
    - 组合
      * 当一个类的对象是另一个类对象的属性时，就是组合。例如：学生类 班级类，
        + class Clas: def \_\_init\_\_(self,cname,begint,teacher): self.cname = cname self.begint = begint self.teacher = teacherclass Course: def \_\_init\_\_(self,name,period,price): self.name = name self.period = period self.price = pricepy22 = Clas('python全栈22期','2019-4-26','小白')linux57 = Clas('linux运维57期','2019-3-27','李导')linux58 = Clas('linux运维58期','2019-6-27','李导')python = Course('python','6 months',21800)linux = Course('linux','5 months',19800)py22.course = pythonlinux57.course = linuxlinux58.course = linuxprint(py22.course.period)print(linux57.course.price)
      * 小技巧：对象名.属性 = 另一个类的属性 就是在本类的对象中添加属性对象名.属性 仅仅是访问/调用本对象自己的属性
    - 对象命名空间和类命名空间的访问顺序
* pickle补充（重要）# 通过pickle实现将对象写入文件，再通过文件读取对象
  + 实例化 Course类
    - class Course: def \_\_init\_\_(self, name, period, price): self.name = name self.period = period self.price = pricepython = Course('python', '6 month', 21800)
  + 1. 将实例化对象写入文件(wb模式)
    - # 利用 pickle.dump 将对象写入到文件import picklewith open('pickle\_file', 'wb) as f: pickle.dump(python. f)
  + 2.将保存在文件中的对象从文件中读取出来（前提是有这个类）
    - with open('pickle\_file', rb) as f: while True: try: obj = pickle.load(f) print(obj.name, obj.period, obj.price) except EOFError: break
* python内置函数（随用随写）
  + 序列化和反序列化
    - 序列化
      * 将内存中的数据，转换成字节串，用以保存在文件或通过网络传输，成为序列化
    - 反序列化
      * 从文件中或网络中获取的数据，转换成内存中原来的数据类型，成为反序列化过程
    - json
      * JavaScript Object Notation# java脚本兑现标记语言，已成称为一种简单的数据交换语言# json 文件通常一次性写，一次性读；# 如果需要多次写，多次读，需借助文件的 write() 方法，把多次序列化的对象 写入到文件中，# 再通过for 循环文件句柄，把分次序列化的json字符，反序列化回来# json 不支持python中的 集合类型 {} set(), 也不支持将对象类型写入文件（暂时不确定）
      * 序列化
        + json.dumps(obj)
        + json.dump(obj, 文件句柄)
      * 反序列化
        + json.loads(字符串)
        + json.load(文件句柄)
      * 代码实例
        + '''# 自定义 json，借助 json 模块来完成简化的 dump 和 load # json dump # 打开文件 # 把数据dump到文件里 # json load # 打开文件 # 读数据# 举例：# 对象 = Myjson('文件路径')# 对象.load() 能拿到这个文件中所有的对象# 对象.dump(要写入文件的对象)'''import osimport jsonclass MyJson: ''' 封装json 模块，实现文件中实例化对象的读和写 ''' def \_\_init\_\_(self, path): self.path = path self.obj = [] # 将对象分行写入文件 def dump(self, obj): self.obj.append(obj) with open(self.path, mode='a', encoding='utf-8') as write\_f: write\_f.seek(0,2) write\_f.write(json.dumps(obj)+'\n') write\_f.flush() def load(self): with open(self.path, mode='r', encoding='utf-8') as read\_f: for i in read\_f: yield json.loads(i.strip())class Student(): def \_\_init\_\_(self, name, age, id): self.name = name self.age = age self.id = iddef main(): path = os.path.join(os.path.dirname(\_\_file\_\_), 'myjson.json') s1 = [1, 2, 3, 4] s2 = {'name':'zew', 'age':30} # MyJson(path).dump(s1) # MyJson(path).dump(s2) for i in MyJson(path).load(): print(i)if \_\_name\_\_ == '\_\_main\_\_': main()
    - pickle
      * 序列化
        + pickle.dumps(obj)
        + pickle.dump(obj, 文件句柄)
      * 反序列化
        + pickle.loads(obj)
        + pickle.load(文件句柄)
      * 代码实例
        + '''# 自定义pickle，借助pickle模块来完成简化的 dump 和 load # pickle dump # 打开文件 # 把数据dump到文件里 # pickle load # 打开文件 # 读数据# 举例：# 对象 = Mypickle('文件路径')# 对象.load() 能拿到这个文件中所有的对象# 对象.dump(要写入文件的对象)'''import os, sysimport pickleclass MyPickle: '''封装pickle模块，实现对象的读写''' def \_\_init\_\_(self, path): self.path = path def dump(self, obj): with open(self.path, mode='ab') as write\_f: pickle.dump(obj, write\_f) def load(self): with open(self.path, mode='rb') as read\_f: while True: try: yield pickle.load(read\_f) # 返回生成器 except EOFError: breakpath = os.path.join(os.path.dirname(\_\_file\_\_), 'mypickle.txt')class Student(): def \_\_init\_\_(self, name, age, id): self.name = name self.age = age self.id = iddef main(): s1 = Student('li', 15, '001') s2 = Student('zhou', 14, '002') MyPickle(path).dump(s1) MyPickle(path).dump(s2) for i in MyPickle(path).load(): print(i, i.\_\_dict\_\_)if \_\_name\_\_ == '\_\_main\_\_': main()
    - pickle 和 json 的区别
      * pickle :将python中所有的数据类型，转换成字节串，序列化过程。将字节串转换成python中数据类型，反序列化.用法同 json总结：1.pickle 常用场景：和json一样,一次性写入，一次性读取2. json 和 pickle的区别：json:1> 不是所有的数据类型都可以序列化，例如 集合，结果是字符串。2> 不能多次对同一个文件序列化3> json数据可以跨语言pickle:1> 所有的python数据类型都可以序列化，结果是字节串。2> 可以多次对同一个文件序列化3> pickle数据不可跨语言
      * 注意：# pickle 和 json 最大的区别是：pickle 支持将对象写入文件，但不能跨语言；# json支持跨语言，但不支持对象序列化























