第4章 面向对象

李宇

东北大学-计算机学院-智慧系统实验室

liyu@cse.neu.edu.cn

对象

- Python 支持多种不同数据类型

```
1234        3.14159        "Hello"        [1, 5, 7, 11, 13]

{"CA": "California", "MA": "Massachusetts"}
```

- 以上所有数据都是一个**对象**, 每个对象**都有**:
  - **身份（identity）**，即是存储地址，可以通过id()这个方法来查询
  - **类型（type）**，即对象所属的类型，可以用type()方法来查询
  - **值**，都会有各自的属性、方法
- 一个对象是一个类型的**实例**(instance)
  - `1234` 是`int`类型的一个实例
  - `"hello"` 是`string`类型的一个实例

例:[1,2,3,4] 的类型-列表

- 列表在内部是如何表示的？链表的结构

- 如何**操作**列表?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(),`
    `L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- 内部的表达和实现对外不可见
- 如果可见，即使正确的操作，也可能会对内部的实现产生干扰和修改。

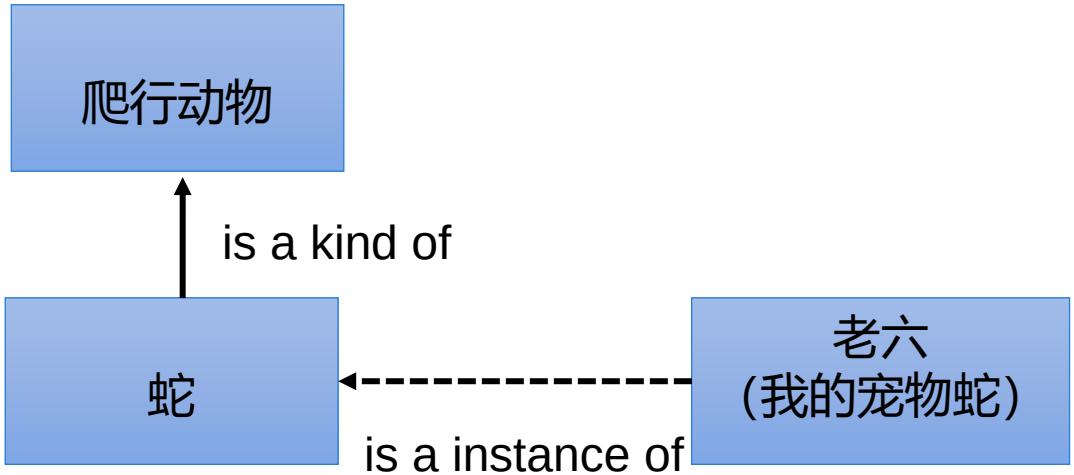
面向对象编程(OOP)

- **Python中，一切皆对象(并有一个类型)**
- 可以**创建**某一类型的对象
- 可以对 对象进行**操作**

- 可以**销毁对象**
  - 显式的使用`del` 或者 不去管它
  - python 系统会 回收被销毁或者不可读取的对象-被称为“垃圾回收机制”

如何理解一切皆对象

- 在面向对象体系里面，存在两种关系：
  - 父子关系，即继承关系
    - python里要查看一个类型的父类，使用它的__bases__属性可以查看
  - 类型实例关系
    - 使用它的__class__属性可以查看，或者使用type()函数查看



如何理解一切皆对象

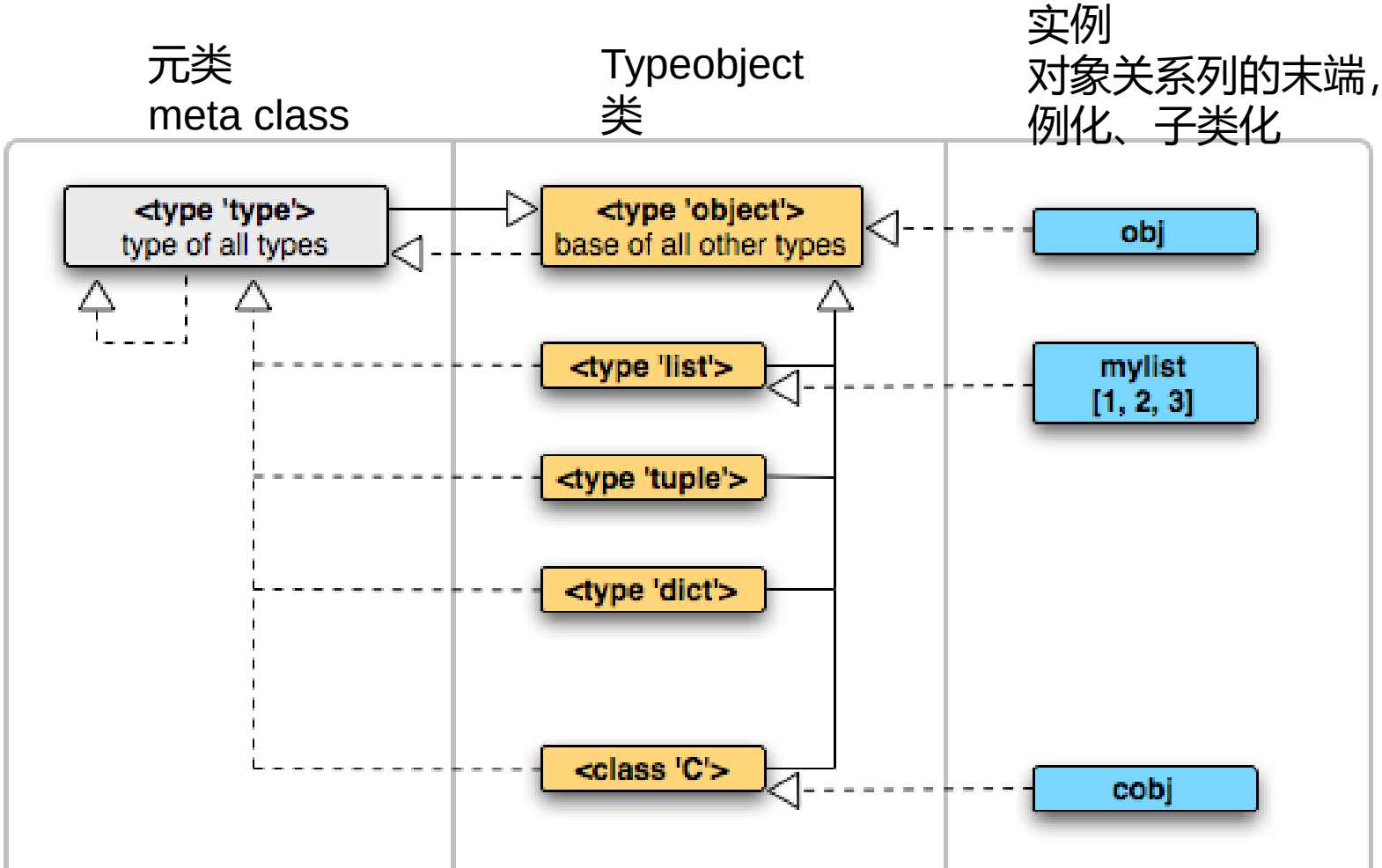
- **object**是继承关系的顶端，object是所有类型的父类（直接或间接）
- **type**是类型实例关系的顶端，所有对象都是它的实例的
- 二者关系：
  - object是type的一个实例
  - Type是object的子类

```
>>>object.__class__
>>>type(object)
#type
>>>object.__bases__
#()
#object是继承关系顶端，没有父类
```

```
>>> type.__bases__
#(object,)
#type是object的子类
>>> type.__class__
#type
#type的类型是type
```

如何理解一切皆对象

一图总结



实线：继承
虚线：实例

面向对象的好处

- 将**数据**和**方法**进行**打包** 通过接口提供给使用者
- **分而治之(divide-and-conquer)** 的开发
  - 可以独立的开发和测试每个类
  - 使程序更加模块化，降低复杂度
- 类(classes) 让代码**复用**更容易
  - 很多Python的模块都有自定义的新类
  - 每个类都有自己独立的环境和命名(不与其他函数名冲突)
  - 使用继承使得子类重定义或者扩展父类的一些行为

创建和使用类的区别

- **创建**一个类有别于**使用类**来创建一个实例(instance)
- **创建** 类包括
  - 定义类名
  - 定义类的属性、方法
  - *例：写一段实现list的代码*
- **使用** 类包括
  - 创建类的**实例**
  - 对实例进行操作
  - *例,* `L=[1,2]` *和* `len(L)`

自定义类（类型）

- class 关键字： 定义一个新的类型

定义class 关键字

name/type(首字母大写)

class的父类

```
class Coordinate(object):
    #define attributes here
```

- 与def类似, 类中的内容需要缩进
- 例子中，object 表示 Coordinate 这个类 继承自object，有object类中所有的属性
  - Coordinate 类是 object 类的子类
  - object 是 Coordinate 的父类

什么是属性(attributes)

- 类中的数据和程序
- **数据属性**
  - 构成类的数据对象
  - *例： Coordinate 类 由 2 个数（坐标）构成*
- **方法** (程序属性)
  - 方法就是类中的函数，只作用于这个类或类的实例
  - 与对象进行交互
  - *例：在coordinate类中定义一个distance的方法，求两个坐标对象的距离，但distance无法作用在其他类型的对象上，比如list*

创建类的实例

- **构造方法(constructor)**：定义如何创建一个类的实例
- 使用特殊方法： **__init__** 初始化一些数据属性

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

用于初始化Coordinate对象的数据

创建实例时的初始化方法，注意__是双下划线

__init__方法的第一参数永远是self，表示创建的类实例本身

每个Coordinate类型的对象都有的2个数据属性,绑定到self上

创建类的实例

- 定义好了类之后，创建一个实例吧

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

创建一个新的对象c
类型为Coordinate
把3和4传给__init__
方法

用.来获取到实例c
的属性和方法

- 实例中的数据属性，比如c.x中的x，叫做**实例变量**
- 不用传递任何值给self，python会自动处理

什么是方法(method)

- 方法是类中的程序属性，一个**只作用于这个类的function(函数)**
- Python总是会把对象当作第一个参数
  - 惯例是把**self**当作所有方法的第一个参数
- “.” **操作符** 可以获取到任意属性
  - 对象的实例属性
  - 对象的方法(method)

写一个Coordinate类的方法


```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

用来引用传入的实例

方法的另一个参数

.操作符，用来获取数据

- 除了self和 . 操作符, 方法和函数的表达是一样的 (获取参数、做计算和操作、返回)

如何使用方法

```
def distance(self, other):
    # code here
```

方法的定义

- 一般使用方法

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

调用方法的对象　**方法**名　传参数(不用传入self，self这里就是c，会自动传入)

- 等价于用 类名.方法名

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

类名　**函数**名　传参数(此时需给self传入对象参数

方法和函数的区别：方法是实例对象的行为和动作！

如何使用方法(method)


```
class Class:
    def method(self):
        print('I have a self!')

def function():
    print("I don't...")

instance = Class()
instance.method()
instance.method = function
instance.method()

type(instance.method)
#I have a self!
#I don't...
#function
```

把function 赋值给instance.method

此时instance.method这个标签重新指向了function

如何使用方法(method)


```
class Bird:
    song = 'Squaawk!'
    def sing(self):
        print(self.song)

bird = Bird()
bird.sing()
birdsong = bird.sing
type(birdsong)
bird.sing()
bird.birdsong()

#Squaawk!
#method
Squaawk!
AttributeError: 'Bird' object has no attribute 'birdsong'
```

打印一个对象

```
>>> c = Coordinate(3,4)
>>> print(c)
#<__main__.Coordinate object at 0x7fa918510488>
```

- 直接打印对象时，无法打印出有效信息
- 给类定义一个**__str__**方法
- 使用print()打印对象时，Python 会调用__str__方法
- 可以自定义打印的内容和格式! 比如打印Coordinate对象时, 我们想要

```
>>> print(c)
<3,4>
```

自定义打印方法


```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

__str__的特殊方法名

规定打印格式，str()函数：强制转换成str类型

类和类型(Types)

- 可以查询对象实例的类型

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class '__main__.Coordinate'>
```

__str__的方法的返回值

对象c的类型是一个Coordinate类

- 直接打印一个类?

```
>>> print(Coordinate)
<class '__main__.Coordinate'>
>>> print(type(Coordinate))
<type 'type'>
```

Coordinate是一个类

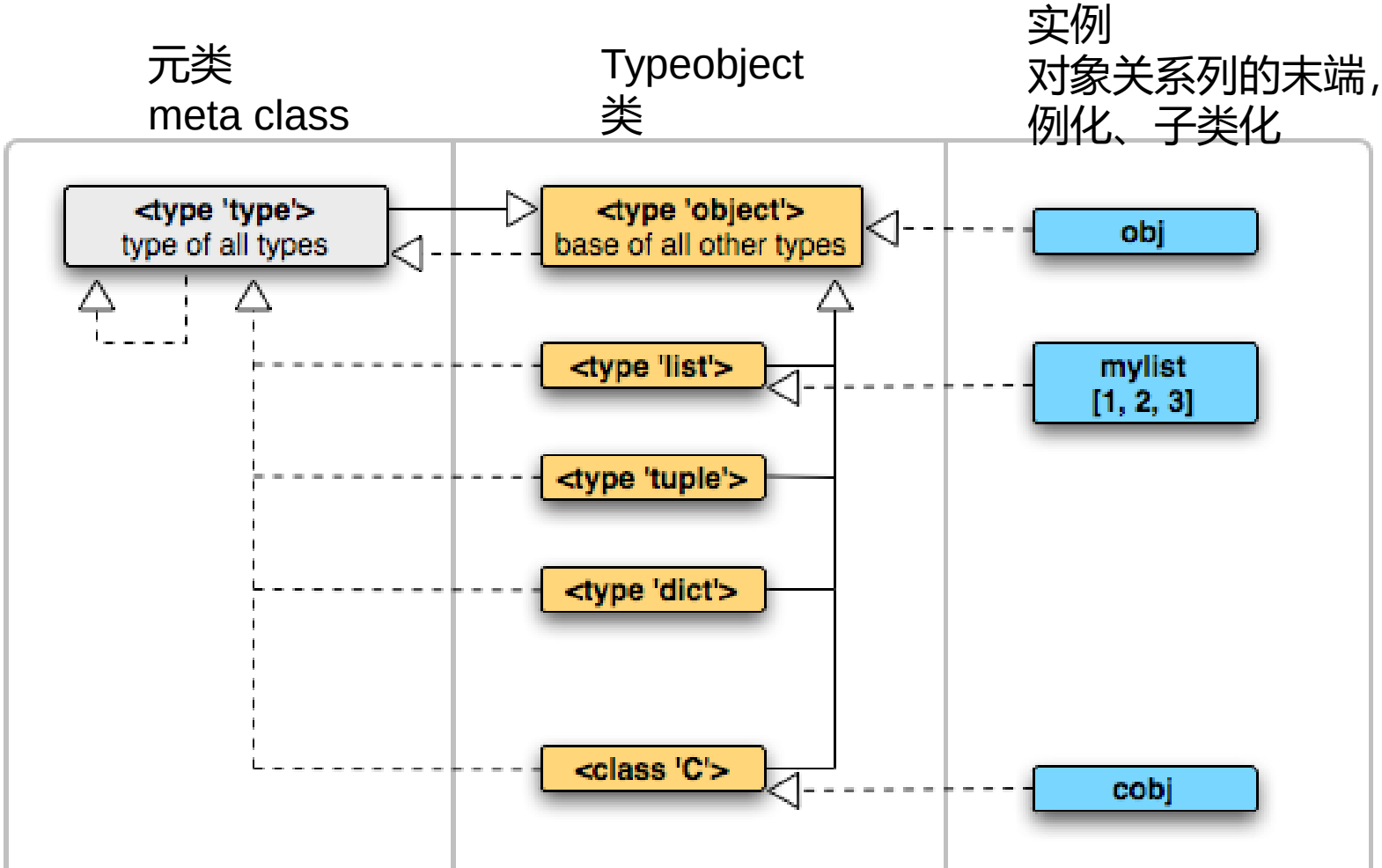
Coordinate类的类型是一个Object

- 用 isinstance() 来检查c是一个Coordinate类的对象

```
>>> print(isinstance(c, Coordinate))
True
```

如何理解一切皆对象

一图总结



实线：继承
虚线：实例

实现类 VS 使用类

用类来**实现**一个新的对象类型：

- **定义** 类
- 定义 **数据属性**
  (WHAT IS the object)
- 定义 **方法**
  (HOW TO use the object)

在程序中**使用**新的对象类型

- 创建一个类的**实例**
- 对类的的实例进行**操作**

类定义 VS 类的实例

- 类的名字就是**类型(type)**
  - `class Coordinate(object)`
- 类的定义是通用于所有实例的
  - 定义类时，用 `self` 来引用实例
  - `(self.x - self.y)**2`
    - `self` 是定义类时传入方法的参数
- 类定义了所有实例的数据属性和方法

- 实例是**一个具体**的对象

```
coord = Coordinate(1,2)
```

- 实体之间的数据属性可能不同

```
c1 = Coordinate(1,2)
c2 = Coordinate(3,4)
```

  - `c1`和`c2` 有不同的数据属性值 `c1.x` 和 `c2.x`，因为他们是不同的对象
- 实例拥有整个类的结构

为什么用面向对象和类-实体概念

- 模拟真实世界
- 把同一类型的对象聚到一类

Jelly
1岁
棕色

5岁
棕色

Tiger
2岁
棕色

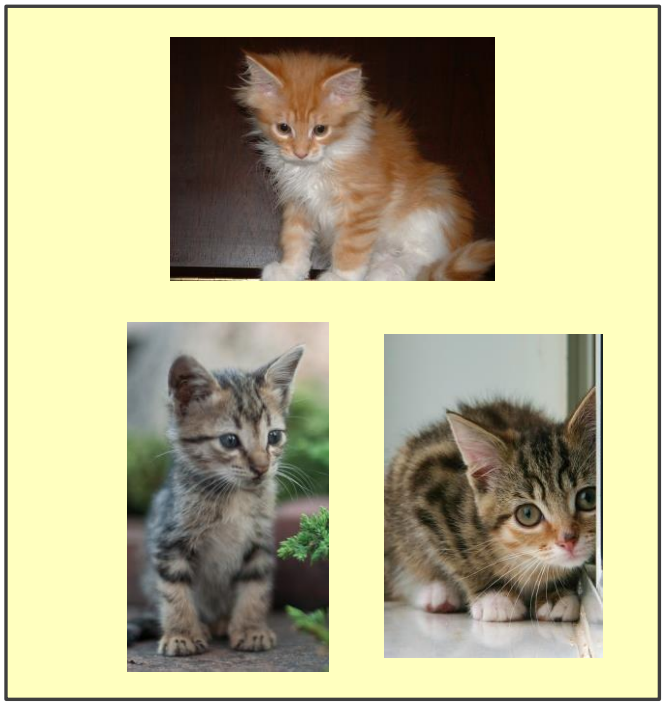
Bean
0岁
黑色

2岁
白色

1岁
黑/白

为什么用面向对象和类-实体概念

- 模拟真实世界
- 把同一类型的对象聚到一类



GETTER AND SETTER 方法

東北大學 Northeastern University

```
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)
```

getter

setter

- **在class的外部获取数据属性时，应使用getters 和 setters**

不能从外部访问的属性、方法

- Python**没有**为私有属性提供**直接的**支持
- 但可以通过在名称前以**两个下划线开头**即可(__attr)

```
class Secretive:
    def __inaccessible(self):
        print("Bet you can't see me ...")
    def accessible(self):
        print("The secret message is:")
        self.__inaccessible()
s = Secretive()
s.__inaccessible() #输出什么?
s.accessible() #输出什么?
```

- 如果非要从类外访问私有方法：对象.单下划线+类名

s._Secretive__inaccessible()

一个实例 和 .(dot)符号

- **一个对象的实例**的创建叫做实例化：

```
a = Animal(3)
```

- **.符号用来获取属性(数据和方法)**，但是最好使用getters和setters来获取数据属性

```
a.age
```

直接通过‘.’操作符获取数据属性，可行，但不推荐

```
a.get_age()
```

getter，用来获取数据

为何不用.符号：信息隐藏

- 创建类的程序员可能会修改数据属性名，但你不知道

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

用名称year来代替age属性

- 如果从类的外部获取数据属性，并且类的内部定义有修改，可能会产生Error
- 在类的外部，最好使用getters和setters，比如 `a.get_age()`，而不是 `a.age`
  - 代码美观，有风格
  - 易于后期维护
  - 防止bugs

Python的信息隐藏并不到位

- 允许从类定义的外部**获取数据**

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

- 允许从类定义的外部**写入数据**

```
a.age = 'infinite'  #age本应是一个整数
```

- 允许从类定义的外部**创建新的数据属性**

```
a.size = "tiny" #（size并没有在类中定义）
```

- 以上的行为均**不提倡**！

方法的默认参数

- 没传入参数，则使用默认参数值

```
def set_name(self, newname=""):
    self.name = newname
```

- 使用默认参数值

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

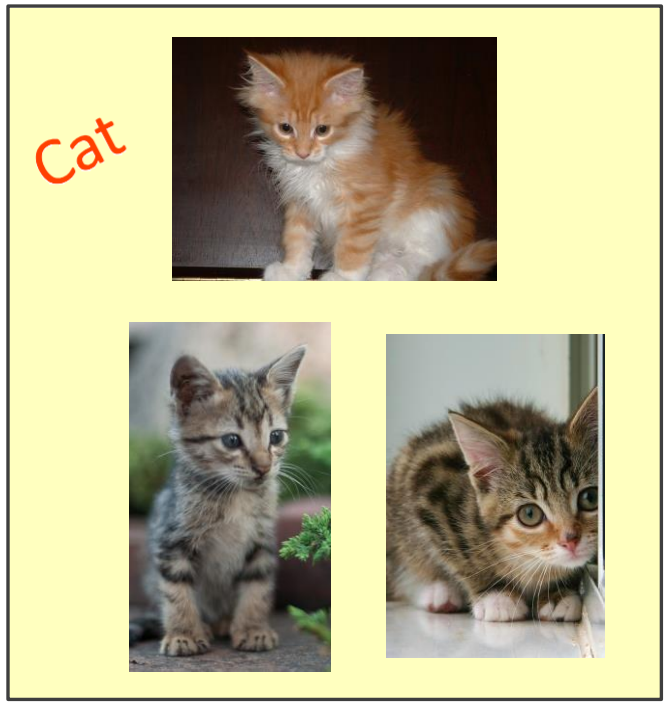
打印 “”

- 传入参数

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

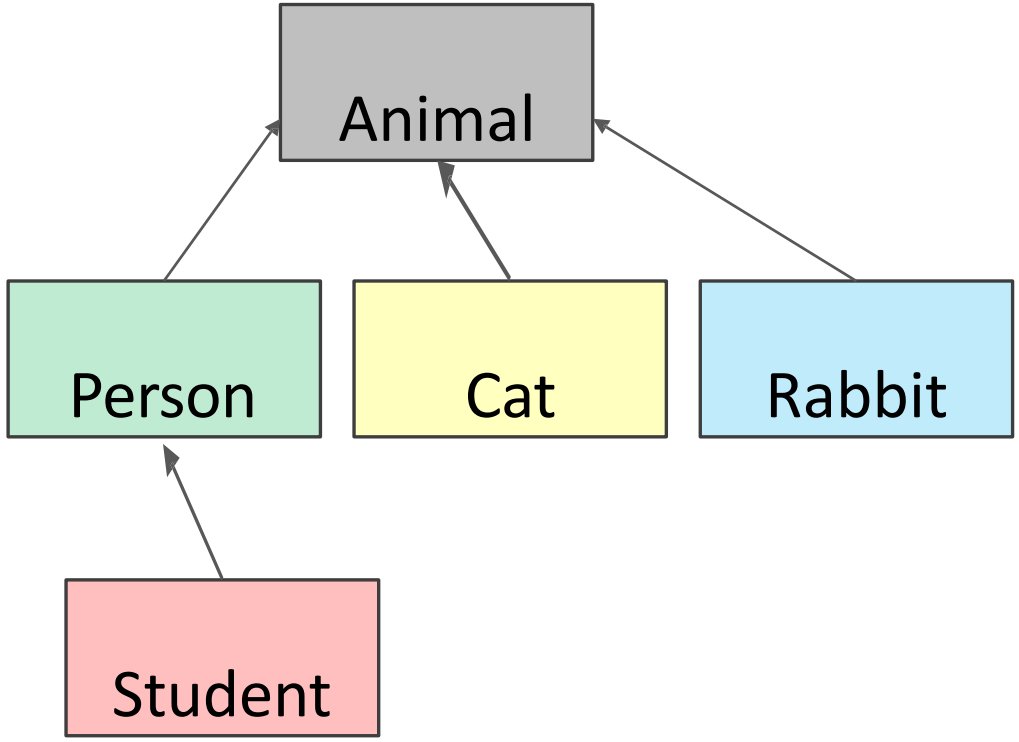
打印 “fluffy”

类的层级



类的层级：继承

- **父类(parent class)** (superclass)
- **子类(child class)** (subclass)
  - **继承(inherits)**父类里所有的数据和行为
  - **添加**数据属性
  - **添加**方法
  - **重写/覆写(override)** 方法



父类(Parent class)

```
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)
```

一切皆对象(object)

object类实现了Python的基本操作，比如变量赋值等等

子类

继承所有Animal类的属性:
__init__()
age,name
get_age(),get_name()
set_age(),set_name()
__str__()

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

添加新的方法
speak()

重写/复写(overrides)__str__

- 添加一个新的功能/行为 `speak()`
  - `Cat` 类型的实例可以调用新的方法
  - `Animal`类型的实例如果调用新方法`speak()` 会抛出异常(throws error)
- `__init__` 构造方法并没有消失，从`Animal`父类中继承

实例使用的是哪个方法？

- 子类中的方法可以与父类中的**方法同名**
- 对于一个类的实例，会**在自己的类定义**中寻找方法名
- 如果没找到，去**上级父类**中寻找方法名
(先在父类中找，然后去“爷爷”类中找，以此类推)
- 调用最先找到的那个方法

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

父类是 Animal

调用Animal的构造方法

调用Animal的set_name方法

添加新的数据属性

新的方法

重写Animal的__str__方法

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

导入random模块

继承Person类和Animal类的属性

新的数据属性

查询random模块的文档，random()方法返回(0,1)的float数

类变量(class variable)

- 所有的实例**共享类变量(class variables)**

父类是Animal

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

类变量 class variable

实例变量 instance variable

获取类变量

修改类变量tag会改变所有实例中的tag的值

- tag 用来给每一个新的rabbit实例一个**唯一的id**

类变量 vs 成员变量

```
class TestClass(object):
    val1 = 100

    def __init__(self):
        self.val2 = 200

    def fcn(self,val = 400):
        val3 = 300
        self.val4 = val
        self.val5 = 500
```

类变量

成员变量/属性

局部变量

类内全局变量

Rabbit GETTER 方法

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal._init_(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

zfill方法会给string按位补全0，比如str='1'，str.zfill(3)='001'

Rabbit类实现的getter方法

除此之外，还有从父类Animal类中继承的get_name和get_age方法

魔法方法重载

```
def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

再次调用 Rabbit类的构造方法__init__(self, age, parent1=None, parent2=None)

- __add__ 方法定义 **+ 操作符** 作用于2个Rabbit实例之间，比如定义r4 = r1 + r2 时做什么
  r1 and r2 都是 Rabbit 的实例
  - r4 是一个新的Rabbit的实例，age是0
  - r4 用 self 代表parent1，others 代表parent2
  - 在__init__ 里**parent1 和 parent2 都是Rabbit类型**

魔法方法重载

- 内置函数dir()可以查看对象的所有方法和属性，其中**双下划线开头和结尾的就是该对象具有的魔法方法/属性。**以int对象为例：
- >>>dir(int)
- ['__abs__', '__**add**__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', …]
- 整数对象下具有__add__方法，这也是为什么我们可以直接在python中运算1+2,当python识别到+时，就去调用该对象的__add__方法来完成计算

比较两个Rabbits的特殊方法

- 写**r1==r2**时，比较两个rabbits实例是否相等（两个parent相同），自动调用**__eq__方法**

booleans

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
    and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
    and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

- 比较parent的id(rid)，因为id是唯一的（类变量）
- 注意：不可以在__eq__方法里直接比较传入的实例
  - 例 `self.parent1 == other.parent1`
  - 这个操作会一直调用`__eq__` 方法 直到 调用`None`，当尝试调用`None.parent1`时返回`AttributeError`

运算符重载的魔法方法

常用的运算符重载如下表：

函数名	运算	
__sub__	-	
__mul__	*	
__truediv__	/	
__floordiv__	//	
__mod__	%	
__pow__	**	
__and__	&	
__xor__	^	
__or__	\	

常见魔法方法

魔法方法	含义
	基本的魔法方法
__new__(cls[, ...])	1. __new__ 是在一个对象实例化的时候所调用的第一个方法 2. 它的第一个参数是这个类，其他的参数是用来直接传递给 __init__ 方法 3. __new__ 决定是否要使用该 __init__ 方法，因为 __new__ 可以调用其他类的构造方法或者直接返回别的实例对象来作为本类的实例，如果 __new__ 没有返回实例对象，则 __init__ 不会被调用 4. __new__ 主要是用于继承一个不可变的类型比如一个 tuple 或者 string
__init__(self[, ...])	构造器，当一个实例被创建的时候调用的初始化方法
__del__(self)	析构器，当一个实例被销毁的时候调用的方法
__call__(self[, args...])	允许一个类的实例像函数一样被调用：x(a, b) 调用 x.__call__(a, b)
__len__(self)	定义当被 len() 调用时的行为
__repr__(self)	定义当被 repr() 调用时的行为
__str__(self)	定义当被 str() 调用时的行为
__bytes__(self)	定义当被 bytes() 调用时的行为
__hash__(self)	定义当被 hash() 调用时的行为
__bool__(self)	定义当被 bool() 调用时的行为，应该返回 True 或 False
__format__(self, format_spec)	定义当被 format() 调用时的行为

函数调用

打印

super()

- super()是python内置函数
- 单继承中，super()主要是用来调用父类的方法

```
class A:
    def method(self):
        print("I am in parent")
class B(A):
    def method(self):
        print("I am in child.")
        super().method()
b = B()
b.method()
#I am in child.
#I am in parent
```

通过super() 调用父类A中的method方法

super()

```
class A:
    def __init__(self):
        self.n = 2

    def add(self, m):
        print('self is {0} @A.add'.format(self))
        self.n += m

class B(A):
    def __init__(self):
        self.n = 3

    def add(self, m):
        print('self is {0} @B.add'.format(self))
        super().add(m)
        self.n += 3
b = B()
b.add(2)
print(b.n)
```

```
#self is <__main__.B object at 0x0000020087B8A1D0> @B.add
#self is <__main__.B object at 0x0000020087B8A1D0> @A.add
#8
```

super()的高阶用法

- super(type, obj)
  - obj需要是type的实例，或者type子类的实例
- super(type, type1)
  - type1需要是type的子类

```
#练习题
def myrepr(cls):
    cls.__repr__ = lambda self:super(cls,self).__repr__()[10:15]
    return cls

@________
class classwithlonglonglongname():
      pass
c = classwithlonglonglongname()
print(c)
#class
```

多重继承

```
class Calculator:
        def calculate(self, expression):
                self.value = eval(expression)
class Talker:
        def talk(self):
                print('Hi, my value is', self.value)
class TalkingCalculator(Calculator, Talker):
        pass
```

- 子类TalkingCalculator什么都没做
  - 从Calculator类那里继承calculate方法
  - 从Talker那里继承talk方法

魔法方法重载

代表当前的类本身

```
def __new__(cls):
    #当代码中实例化一个类的时候，第一个调用执行的是__new__()方法
    #而非__init__方法
    return super().__new__(cls)
```

__new__方法重载时写法相对固定，调用父类的__new__，(object类)

- 当实例化一个类的时候，第一个调用执行的是__new__()方法
- 当定义的类中没有重载__new__()方法时候，Python会默认调用该父类的__new__()方法来构造该实例
- __new__的作用：它是类的一个方法，先创建一个空间，然后创建实例化对象，用开辟的空间存放这个实例化对象
- __init__其实不是实例化一个类的时候第一个被调用的方法，它用于初始化实例对象

魔法方法重载


```
class Mycls:
        def __new__(cls):
            print('new')
            return super().__new__(cls)
        def __init__(self):
            print('init')
#用Mycls类创建一个实例化对象my
my=Mycls()
```

#new
#init

__new__方法的应用场景

- 单例模式：确保一个类只有一个实例存在

```
class Mycls:
    instance = None
    def __new__(cls):
        if cls.instance == None: #判断Mycls类的instance是否为空；若空，应该调用父类的new方法，为第一个对象分配空间
            cls.instance = object.__new__(cls) # 把类属性中保存的对象引用返回给python的解释器
            return cls.instance
        else:    # 如果cls.instance不为None,直接返回已经实例化了的实例对象
            return cls.instance
    def __init__(self):
            print('init')
my1=Mycls()
print(my1)
my2=Mycls()
print(my2)
```

```
#init
#<__main__.Mycls object at 0x0000020087B8ADA0>
#init
#<__main__.Mycls object at 0x0000020087B8ADA0>
#可以看到虽然两次创建对象，my1 和my2，但是他们都是同一个对象，这就是单例模式的应用
```

魔法方法重载

- __call__方法
  - 该方法的功能类似于在类中重载 () 运算符，使得类实例对象可以像调用普通函数那样，以“对象名()”的形式使用

```
class Mycls:
# 定义__call__方法
    def __call__(self,name,age):
        print("调用__call__()方法",name,age)
person = Mycls()
person("你儿子","8岁")

#调用__call__()方法 你儿子 8岁
```

- 可以看到，通过在 Mycls 类中实现 __call__() 方法，使的 person 实例对象变为了可调用对象。
- person(“你儿子”,“8岁”) 等同于 person.__call__ ("你儿子","8岁")

抽象类

- 抽象类：
  - 职责：包含子类应该实现的一组抽象方法
  - 特征：不能实例化

```
from abc import ABC, abstractmethod
class Talker(ABC):
        @abstractmethod
        def talk(self):
                pass
Talker()
```

抽象类需继承ABC类,装饰器abstractmethod,声明talk是抽象方法，可以不实现，等待子类实现

#报错 TypeError: Can't instantiate abstract class Talker with abstract methods talk

```
class Knigget(Talker):
        pass
class Herring(Talker):
        def talk(self):
                print(”Hi!")
```

没有实现Talker里的抽象方法，所以Knigget也是抽象类

抽象类：register的用法

```
class Talker(ABC):
    @abstractmethod #装饰器
    def talk(self):
        pass

class Knigget(Talker):
    def talk(self):
        print("Hi!")

class Herring:
    def talk(self):
        print("Blub.")

h = Herring()
isinstance(h,Talker)              #False  (没有继承Talker)
Talker.register(Herring) #将Herring注册为Talker的子类
isinstance(h,Talker)              #True
issubclass(Herring, Talker)       #True
```

register的注意事项

```
from abc import ABC, abstractmethod

class Talker(ABC):
    @abstractmethod #装饰器
    def talk(self):
        pass

class Clam:
    pass

Talker.register(Clam)
issubclass(Clam, Talker)
#True
c = Clam()
isinstance(c, Talker)
#True
c.talk()
#AttributeError: 'Clam' object has no attribute 'talk'
```

静态方法

```
class A(object):
    @staticmethod
    def f(x, y):
        return x+y
```

静态方法中参数没有self

```
>>> A.f(1, 9) #通过类名调用f静态方法
#10
>>> a = A()
>>> a.f(1,9) #通过实例调用f静态方法
#10
```

使用一个类:

- 一般使用方法

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

调用方法的对象　方法名　传参数(不用传入self，self这里就是c，会自动传入)

- 等价于用 类名.方法名

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

类名　方法名　传参数(包括了给self传入的对象

类方法


```
class A(object):
    @classmethod
    def f(cls, x):
        print('class name: %s.' %cls.__name__)
        print('the argument you passed in: %s.'%x)

A.f('hello') #通过类名调用类方法，不需要手动传入cls
#class name: A.
#the argument you passed in: hello.

a = A()
a.f('world') #通过实例调用类方法
#class name: A.
#the argument you passed in: world.
```

类方法第一个参数为cls，cls参数就是class的类型：A

__str__()方法和__repr__()方法

- __str__()和__repr__()都是用来显示/打印的
- print()方法打印一个实例对象时，首先会尝试__str__()，

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
    def __str__(self):
        return '(Person: %s, %s)' % (self.name, self.gender)

p = Person('Bob', 'male')
print(p)
# (Person: Bob, male)
p
#<__main__.Person at 0x1146f9b20>
```

打印实例对象，会调用__str__()

直接敲p，__str__()不会被调用,需要在类里加入__repr__()

__str__()方法和__repr__()方法

- Python 定义了__str__()和__repr__()两种方法，__str__()用于显示给用户，而__repr__()用于显示给开发人员。

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

    def __repr__(self):
        return '(Person: %s, %s)' % (self.name, self.gender)

p = Person('Bob', 'male')
p
#(Person: Bob, male)

print(p)
#(Person: Bob, male)
```

类中只定义了__repr__()，print(p)和p都好用

给参数注释

- s和ss是参数，冒号后面（str）是参数的注释，写什么都不影响程序运行

```
 def fun(s:str, ss:str="hello"):
     print(s, ss)
 fun(12)
 #12 hello

def fun(s:str, ss:str="hello") -> 'int':
    print(s, ss)
    print(fun.__annotations__)
fun(12)
#12 hello
#{'s': <class 'str'>, 'ss': <class 'str'>, 'return': 'int'}
```

-> 是函数返回值的注释，也不影响函数功能

- 这些注释信息都是函数的元信息，保存在fun.__annotations__字典中

__slots__

- __slots__ 属性: 可以避免用户频繁的给实例对象动态地添加属性或方法

```
class Student(object):
        pass

s = Student()
s.name = 'Michael' # 动态给实例绑定一个属性
print(s.name)
#Michael

#可以给实例动态绑定一个方法
def set_age(self, age): # 定义一个函数作为实例方法（注意，带上了self）
    self.age = age

from types import MethodType
s.set_age = MethodType(set_age, s) # 给实例s绑定set_age()方法
s.set_age(25) # 调用实例方法
s.age # 测试结果
```

__slots__

- 给一个实例绑定的方法，对另一个实例是不起作用的

```
s2 = Student() # 创建新的实例s2
s2.set_age(25) # s2调用方法set_age,报错!
```

- 若要给所有实例绑定方法，可以给class绑定方法

```
def set_score(self, score):
    self.score = score

Student.set_score = set_score #给class绑定方法
#Student.set_age = MethodType(set_age, Student) 也可以

s2.set_score(99) #此时s2可以调用新绑定的方法和属性
s2.score
#99
```

__slots__

- 如果要限制Student类中的实例属性，比如只允许实例添加name和age属性时，在定义class时，定义__slots__

```
class Student(object):
        __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称

s = Student() # 创建新的实例
s.name = ‘Michael’ # 绑定属性‘name’
s.age = 25 # 绑定属性‘age’

s.score = 99
# 绑定属性‘score’, 报错! score不在__slots__里
```

__slots__注意事项

- __slots__定义的属性仅对当前类实例起作用，对继承的**子类不起作用**

```
#GraduateStudent 继承Student
class GraduateStudent(Student):
        pass

g = GraduateStudent()
g.score = 9999
 #Student的子类实例仍然能够动态绑定score
```

- 除非在子类中也定义__slots__，这样，子类实例允许定义的属性就是自身的__slots__加上父类的__slots__

__slots__注意事项

- __slots__ 只能限制为实例对象动态添加属性和方法，而**无法限制动态地为类添加属性和方法。**

```
class Student(object):
        __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称

s = Student() # 创建新的实例

s.score = 99 # 绑定属性‘score’, 报错! score不在__slots__里

Student.score = 111
#__slot__限制不住为Student类动态添加一个属性或方法
```

__slots__的好处

- 节省memory

```
#两个类唯一区别就是BarSlotted类多了一个__slots__属性
class Bar(object):
  def __init__(self, a):
    self.a = a

class BarSlotted(object):
  __slots__ = "a",
  def __init__(self, a):
    self.a = a

# create class instance
bar = Bar(1)
bar_slotted = BarSlotted(1)
dir(bar)
# [很多属性] 含有'__dir__'
dir(BarSlotted)
#[很多属性]  不含有'__dir__'，省空间
```

```
#查看两个类的空间大小
from pympler import asizeof
asizeof.asizeof(bar)
# >> 256 bytes
asizeof.asizeof(bar_slotted)
# >> 80 bytes
```

装饰器

- 在代码运行期间动态的给函数或类增加功能的方式，称之为“装饰器”（Decorator）

```
#定义一个函数now
def now():
    print('2015-3-25')
#功能太简单，想事后给它增加一些功能，比如，打印日志的功能
#定义一个日志函数log，接受一个函数对象func，打印”call func.__name__”，并返回一个函数调用
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)#打印日志
        return func(*args, **kw)#调用传入的func函数并返回
    return wrapper
```

装饰器

- 在代码运行期间动态的给函数或类增加功能的方式，称之为“装饰器”（Decorator）

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)#增加功能，打印日志
        return func(*args, **kw)#调用传入的func函数并返回
    return wrapper

#因为有wrapper，传进来的func无论参数是什么样的，都能处理
now = log(now)
#log(now)执行完后，返回的是wrapper,相当于now=wrapper
#再调用now()的话，就相当于调用wrapper()
#所以log是一个装饰器
```

装饰器

- 前面的写法很复杂，有了装饰器之后，用Python的@语法(语法糖)，把decorator置于函数的定义处。now函数就被log装饰了，功能也增强了，具备了log函数的功能。

```
@log
def now():
    print('2015-3-25')

now()
#call now():
#2015-3-25
```

- 把@log放到now()函数的定义处，相当于执行了语句：

```
now = log(now)
```

装饰器

- **装饰器的几种用法:**
  - **一：函数装饰函数**
  - **二：函数装饰类** :让被装饰的类拥有@函数的功能
  - **三：类装饰函数**
  - **四：类装饰类**

```
def myrepr(cls):
    cls.__repr__ = lambda self:super(cls,self).__repr__()[10:15]
    return cls
@myrepr
class classwithlonglonglongname():
    pass
c = classwithlonglonglongname()
print(c)
#class
```

装饰器 @property

- “实例对象.属性”的方式访问类中定义的属性，其实是欠妥的，因为它破坏了类的封装原则
  - 应在类中设置多个getter或setter方法
  - 通过 ”实例对象.方法” 的方式操作属性
  - 但这种反复调用getter和setter操作类属性的方式比较麻烦

```
s = Student()
s.score = 9999 #将属性暴露出去，不安全，无法检查值的合法性
```

- 同@classmethod, @abstractmethod一样，@**property**也是一个内置装饰器
  - 负责把一个类的getter方法伪装成属性调用
  - @property还会创建@xxx.setter和@xxx.deleter装饰器

装饰器 @property

- 常规操作，使用getter和setter操作属性

```
class Student(object):

    def get_score(self):
        return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
s = Student()
s.set_score(60) # ok!
s.get_score()
s.set_score(9999) #报错
```

装饰器 @property

- 稍稍修改一下Student类，使用@property装饰器装饰getter和setter方法

```
class Student(object):
    @property #此时，添加@property会把getter方法变成使用.获取属性时调用
    def score(self):
        return self._score
    @score.setter #装饰器@score.setter，负责把setter方法变成属性赋值
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
s = Student()
s.score = 9999
#报错，虽然通过点操作符对属性score进行赋值，但是因为我们在类
#中使用了装饰器@score.setter, 所以仍然会调用setter方法
```

装饰器 @property

```
class Student(object):
    @property #getter方法变成属性
    def score(self):
        return self._score

    @score.setter #装饰器@score.setter，负责把setter方法变成属性赋值
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value

    @score.deleter
    def score(self):
        print("删除数据！")
```

```
s = Student()
s.score = 99
del s.score
#删除数据！
```

property()内置函数用法

- 效果和使用@property装饰器一样

- 使用格式:
  - 属性名=**property**(fget=None, fset=None, fdel=None, doc=None)
  - fget :指定getter方法，fset :指定setter方法，fdel:指定删除该属性的方法，doc : 文档字符串，用于说明此函数的作用。

```
class Student:
    def __init__(self,n):
        self.__name = n
    def setname(self,n):
        self.__name = n
    def getname(self):
        return self.__name
    def delname(self):
        self.__name= "xxx"
    #为name属性配置 property() 函数
    name = property(getname, setname, delname, '指明出处')
#print(Stduent.name.__doc__) #查看文档字符串
help(Student.name) #查看文档字符串
s = Student("Li")
#调用 getname() 方法
print(s.name) #此时，s.name就符合封装原则了，调用的是getter方法，而非直接访问
#调用 setname() 方法
s.name="Wang"
print(s.name)
#调用 delname() 方法
del s.name
```

type()高阶：动态创建类

- type()用法:
  - type(obj) #查看类型
  - type(name, bases, dict) #动态创建类

```
#name -- name 表示类的名称
#bases -- bases 表示一个元组，其中存储的是该类的父类(们)
#dict -- 表示一个字典，用于表示类内定义的属性或者方法

def mymethod(self):
    return "hello in mymethod"
#接下来，使用type()创建一个新的类型,类名是Myclass,父类/基类是object，有一个方法叫method，关联#mymethod()
klass = type("Myclass", (object,), {'method': mymethod})
inst = klass()
inst.method()
# “hello in mymethod”
```

命名规则

- 由于 Python 3 支持 UTF-8 字符集，因此 Python 3 的**标识符可以使用 UTF-8 所能表示的多种语言的字符。**Python 语言是区分大小写的，因此 abc 和 Abc 是两个不同的标识符。
  1. 标识符可以由字母、数字、下画线（_）组成，其中**数字不能打头。**
  2. 标识符不能是 Python 关键字，但可以包含关键字。
  3. 标识符**不能包含空格。**

哪些不合法?

- abc_xyz
- HelloWorld
- abc
- xyz#abc
- abc1
- 1abc

命名规则

窗口程序集名	保 留	保 留	备 注
窗口程序集1			
变量名	类 型	数组	备 注
内存	内存操作类		
F1自动攻击	整数型		

子程序名	返回值类型	公开	备 注
_窗口1_创建完毕			

```
更换新皮肤 (14)
    ※皮肤索引： 14
打开网页 ("www.cq521.com.cn")
    ※网址： "www.cq521.com.cn"
时钟1.时钟周期 = 500
标签3.标题 = "当前进程" + 到文本 (取进程名 (进程ID))
    ※被赋值的变量或变量数组： 标签3.标题
    ※用作赋于的值或资源： "当前进程" + 到文本 (取进程名 (进程ID))
进程ID = 取进程ID ("DNF.exe")
鼠标显示 ()
内存.提升权限 ()
内存.打开进程 (进程ID)
进程ID = 取窗口进程ID (窗口1.取窗口句柄 ())
F1自动攻击 = 热键.注册 (窗口1.取窗口句柄 (), 0, #F1键, &F1自动攻击)
```

易语言的代码片段

- 感谢