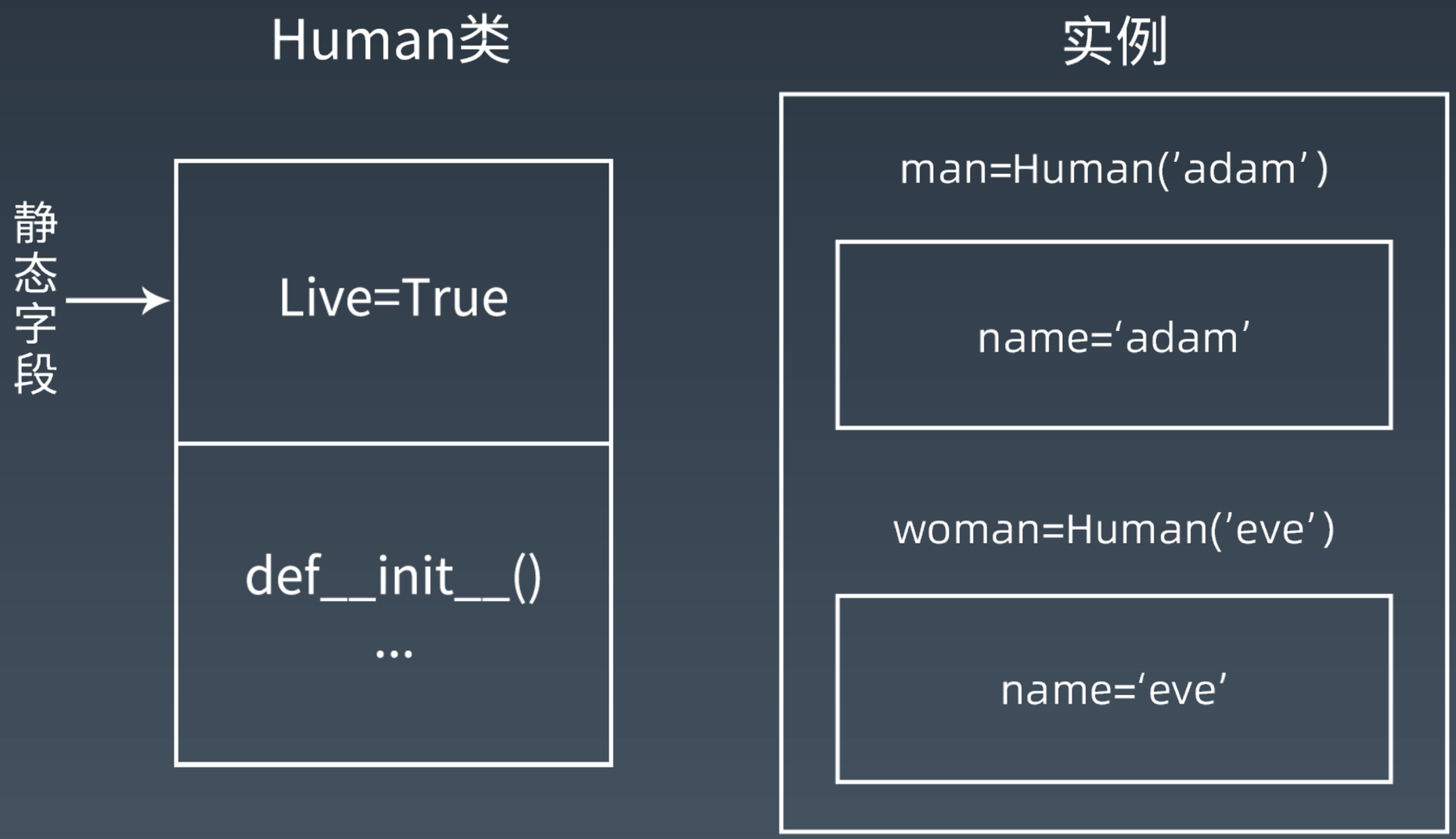
面向对象编程

- 现实世界的对象和 Python 世界的对象
- 对象是一个数据以及相关行为的集合
- Python 的经典类与新式类
- 类的两大成员： 属性和方法

属性

- 类属性与对象属性
- 类属性字段在内存中只保存一份
- 对象属性在每个对象都保存一份


Human类
静态字段
Live=True
def__init__()
...
实例
man=Human('adam')
name='adam'
woman=Human('eve')
name='eve'

属性作用域

作用域：

- _name　　人为约定不可修改
- __name　　私有属性
- __name__　　魔术方法

魔术方法：

- 双下划线开头和结尾的方法，实现了类的特殊成员，这类称作魔术方法
- 不是所有的双下划线开头和结尾的方法都是魔术方法
- 魔术方法类似其他语言的接口

私有属性是可以访问到的，Python 通过改名机制隐藏了变量名称

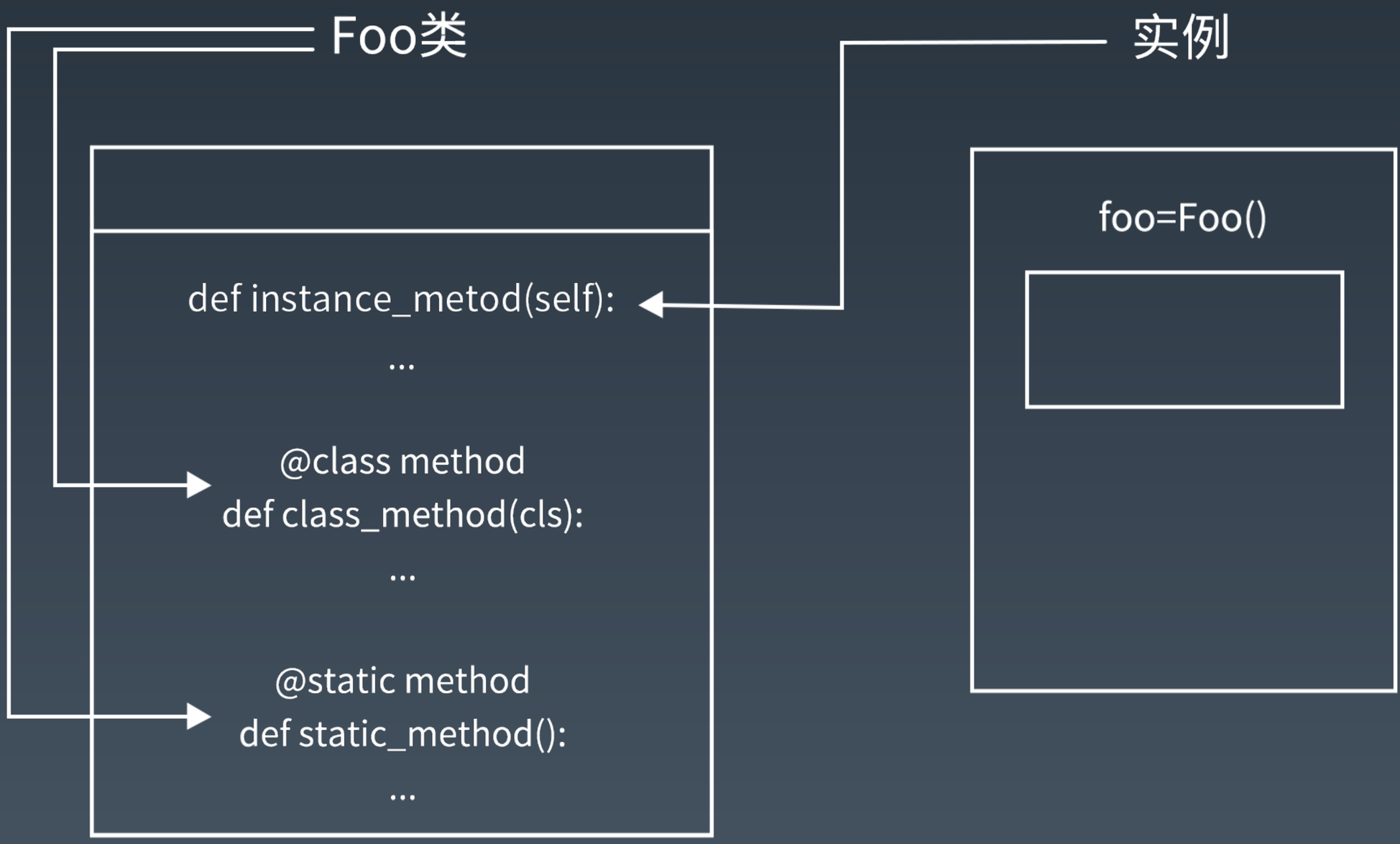
- class.__dict__

方法

三种方法：

- 普通方法　至少一个 self 参数，表示该方法的对象
- 类方法　　至少一个 cls 参数，表示该方法的类
- 静态方法　由类调用，无参数

三种方法在内存中都归属于类


Foo类
实例
def instance_metod(self):
...
@class method
def class_method(cls):
...
@static method
def static_method():
...
foo=Foo()

特殊属性与方法

- __init__()
  - __init__() 方法所做的工作是在类的对象创建好之后进行变量的初始化。
  - __init__() 方法不需要显式返回，默认为 None，否则会在运行时抛出 TypeError。

- self
  - self 表示实例对象本身
  - self 不是 Python 的关键字（cls也不是），可以将 self 替换成任何你喜欢的名称，如 this、obj 等，实际效果和 self 是一样的（不推荐）。
  - 在方法声明时，需要定义 self 作为第一个参数，调用方法的时候不用传入 self。

属性的处理

在类中，需要对获取属性这一行为进行操作，可以使用：

- __getattribute__()
- __getattr__()

属性描述符 property

描述符：实现特定协议的类

property 类需要实现 __get__、__set__、 __delete__ 方法

```
class Teacher:
    def __init__(self, name):
        self.name = name

    def __get__(self):
        return self.name

    def __set__(self, value):
        self.name = value

pythonteacher =  Teacher('yin')

pythonteacher.name = 'wilson'
print(pythonteacher.name)
```

Django中 的 property

site-packages/django/db/models/base.py

```
class Model(metaclass=ModelBase):
    def  _get_pk_val(self, meta=None):
        meta = meta or self._meta
        return getattr(self, meta.pk.attname)
    def  _set_pk_val(self, value):
        return setattr(self, self._meta.pk.attname, value)

    pk = property(_get_pk_val, _set_pk_val)
```

面向对象编程的特性

封装
- 将内容封装到某处
- 从某处调用被封装的内容

继承
- 基本继承
- 多重继承

重载
- Python 无法在语法层面实现数据类型重载，需要在代码逻辑上实现
- Python 可以实现参数个数重载

多态
- Pyhon 不支持 Java 和 C# 这一类强类型语言中多态的写法，
- Python 使用“鸭子类型”

新式类

新式类和经典类的区别

- 当前类或者父类继承了 object 类，那么该类便是新式类，否则便是经典类

object 和 type 的关系

- object 和 type 都属于 type 类 (class 'type')
- type 类由 type 元类自身创建的。object 类是由元类 type 创建
- object 的父类为空，没有继承任何类
- type 的父类为 object 类 (class 'object')

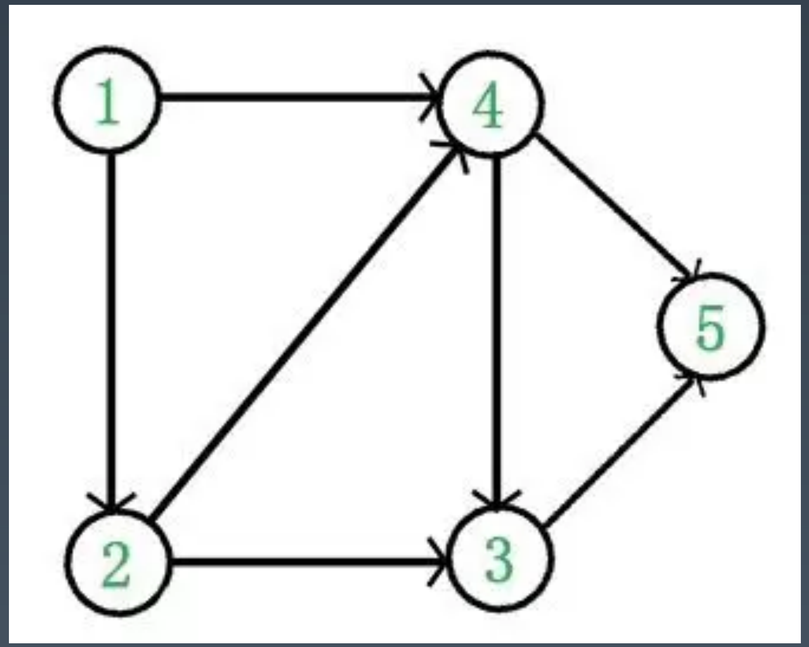
类的继承

- 单一继承
- 多重继承
- 菱形继承（钻石继承）
- 继承机制 MRO
- MRO 的 C3 算法

多重继承的顺序问题

有向无环图：DAG(Directed Acyclic Graph)

- DAG 原本是一种数据结构，因为 DAG 的拓扑结构带来的优异特性，经常被用于处理动态规划、寻求最短路径的场景。



SOLID 设计原则

- 单一责任原则 The Single Responsibility Principle
- 开放封闭原则 The Open Closed Principle
- 里氏替换原则 The Liskov Substitution Principle
- 依赖倒置原则 The Dependency Inversion Principle
- 接口分离原则 The Interface Segregation Principle

设计模式

- 设计模式用于解决普遍性问题
- 设计模式保证结构的完整性

单例模式

1. 对象只存在一个实例
2. __init__ 和 __new__ 的区别：
   - __new__ 是实例创建之前被调用，返回该实例对象，是静态方法
   - __init__ 是实例对象创建完成后被调用，是实例方法
   - __new__ 先被调用，__init__ 后被调用
   - __new__ 的返回值（实例）将传递给 __init__ 方法的第一个参数，__init__ 给这个实例设置相关参数

元类

- 元类是关于类的类，是类的模板。
- 元类是用来控制如何创建类的，正如类是创建对象的模板一样。
- 元类的实例为类，正如类的实例为对象
- 创建元类的两种方法
  1. class
  2. type
     - type（类名，父类的元组（根据继承的需要，可以为空，包含属性的字典（名字和值））

抽象基类

- 抽象基类（abstract base class，ABC）用来确保派生类实现了基类中的特定方法。
- 使用抽象基类的好处：
  - 避免继承错误，使类层次易于理解和维护。
  - 无法实例化基类。
  - 如果忘记在其中一个子类中实现接口方法，要尽早报错。

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Mixin 模式

在程序运行过程中，重定义类的继承，即动态继承。好处：

- 可以在不修改任何源代码的情况下，对已有类进行扩展
- 进行组件的划分

THANKS!