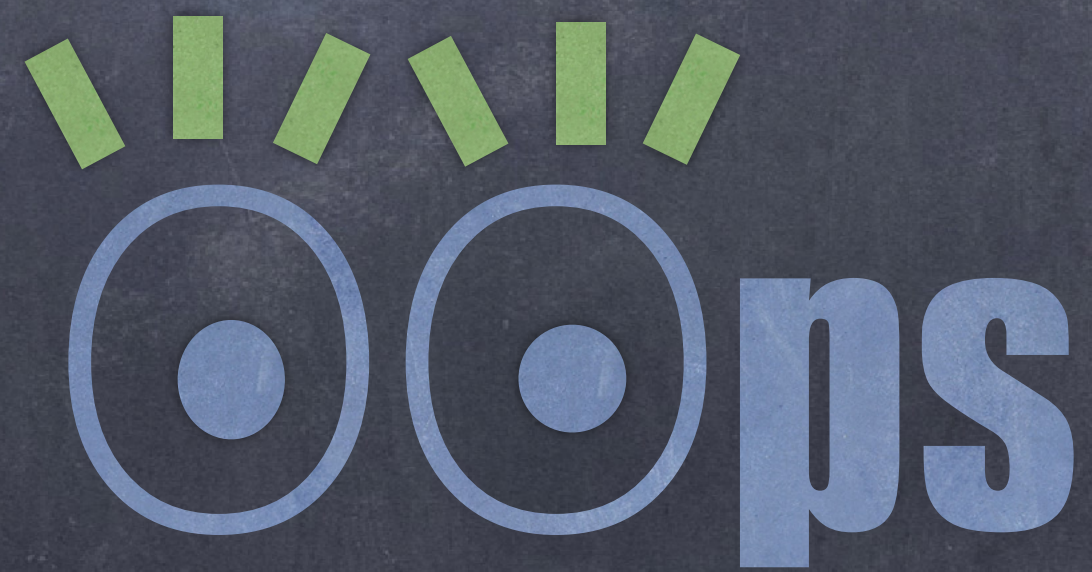


面向对象



面向对象编程 (Object Oriented Programming)

- 抽象可以帮助处理系统的复杂性
- 但是要让程序更加模块化和可维护，还需要进一步加强组织方式 - 将适配的函数和数据组织起来
- 面向对象编程 (OOP) 提供了一种有效的数据抽象方式，其统一抽象出了信息和相应的行为
 - 三大特征：封装 (encapsulation)、继承 (Inheritance)、多态 (polymorphism)

对象隐喻 (Object metaphor)

- 一种将计算视为 (组织为) 多个对象个体交互的过程：
 - ◆ 每个对象有自己的局部状态 (Local state) 和相应的方法 (method) 来管理自己的局部状态
 - ◆ 对象之间可以进行交互，而有效的计算可以视为对象之间交互的结果
 - ◆ 同样类型的对象共享一些行为
 - ◆ 某个类型的对象可以“继承”其相关类型对象的一些信息

例子：一个OOP的巧克力商店

name: Hershey

Price: \$9.9

Nutrition: 170cal, 19g sugar

inventory: 2 bars



name: Dove

Price: \$7.9

Nutrition: 200cal, 24g sugar

inventory: 3 bars



Order #1
Visa



Name: coco Lover
Address: 123 pinning st
Nibbsville, OH

Order #2
UnionPay



Name: Nomandy Norms
Address: 34 Slurpalot PL
Buttertwn, IN

Order #3
Mastercard



Name: Nomandy Norms
Address: 34 Slurpalot PL
Buttertwn, IN

Inventory tracking

Product(name, price, nutrition)

Product.get_label()

Product.get_nutrition_info()

Product.increase_inventory(amount)

Product.reduce_inventory(amount)

Product.get_inventory_report()

Customer tracking

Customer(name, address)

Customer.get_greeting()

Customer.get_formatted_address()

Customer.buy(product, quantity, cc_info)

Purchase tracking

Order(customer, product, quantity, cc_info)

Order.ship()

Order.refund(reason)

Shop management

ChocolateShop(name)

ChocolateShop.signup_customer(name, address)

ChocolateShop.add_product(name, price, nutrition)

OOP的一些术语

- 类 (class) : 定义一个新的类型的模版
- 对象 (object) : 一个类的实例 (instance)
- 实例变量 (instance variables) : 每个对象拥有的数据属性 (attributes), 以表达该对象的状态 (也叫数据成员, members)
- 方法 (method) : 每个对象拥有的函数属性



类抽象

python 中的类

syntax:

```
class <name>:  
    <suite>
```

semantics:

Class statement 创建一个新的类，并且将该类绑定到当前环境帧的<name>上。

<suite> 中的赋值语句和def语句创建了该类的属性

( 这些属性不是帧中的名字，而绑定在类上)

python 中的类

```
# Define a new type of data  
class Product:
```

```
    # Set the initial values
```

```
    def __init__(self, name, price, nutrition_info):  
        self.name = name  
        self.price = price  
        self.nutrition_info = nutrition_info  
        self.inventory = 0
```

```
    # Define methods
```

```
    def increase_inventory(self, amount):  
        self.inventory += amount
```

```
    def reduce_inventory(self, amount):  
        self.inventory -= amount
```

```
    def get_label(self):  
        return "Foxolate Shop: " + self.name
```

```
    def get_inventory_report(self):  
        if self.inventory == 0:  
            return "There are no bars!"  
        return f"There are {self.inventory} bars."
```

```
pina_bar = Product("Piña Chocolotta", 7.99,  
                  ["200 calories", "24 g sugar"])
```

```
pina_bar.increase_inventory(2)
```

Name convention:

Class names are conventionally
written using the CapWords

类实例化 (Class instantiation)

● 类的实例化就是对象的构造 (Object construction)

● 当 `<class name> (args)` 被调用时,

● 1. 类 `<class name>` 的实例被创建

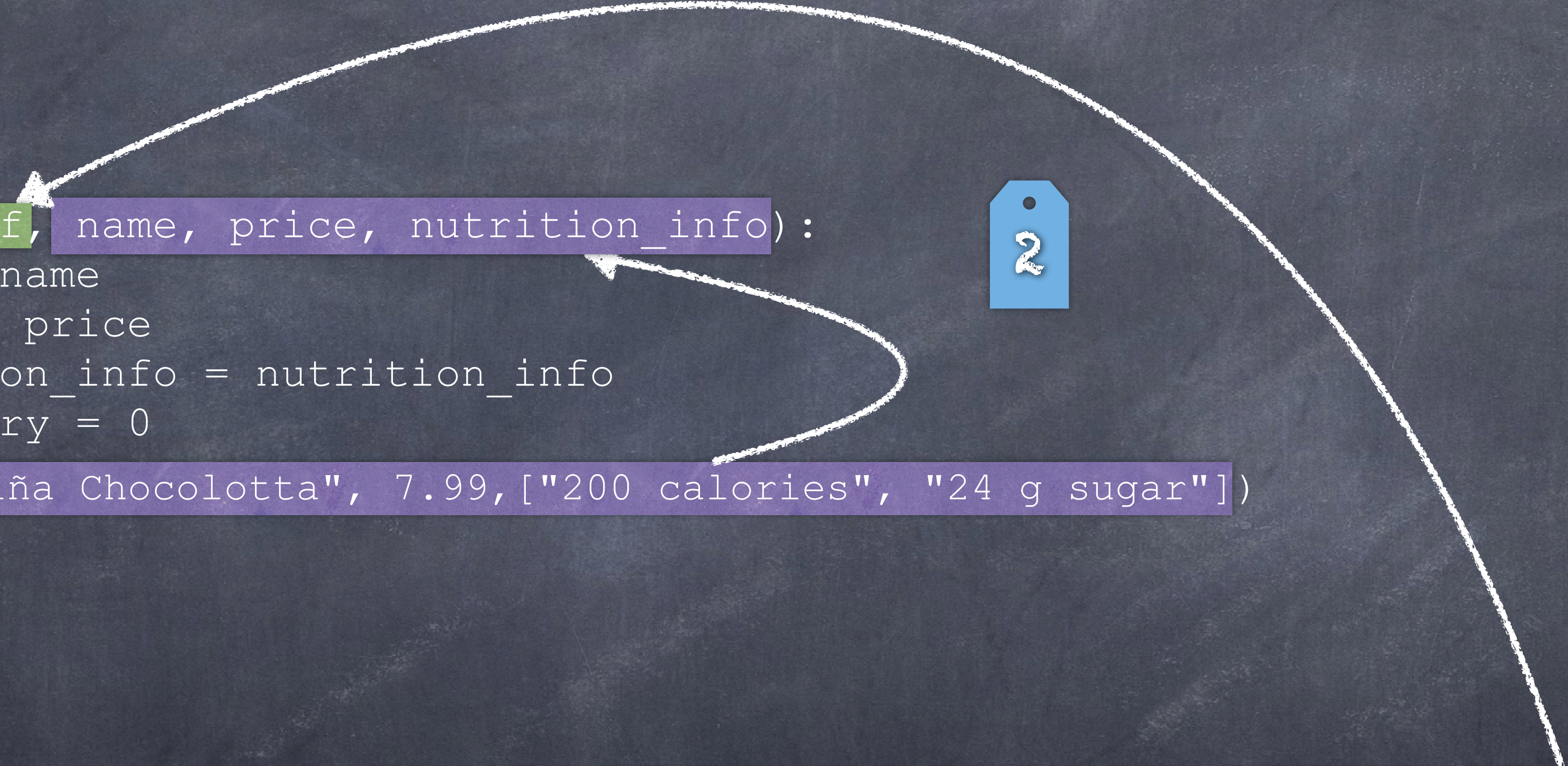
● 2. 类中的 `__init__` 方法被调用, 新创建的实例被作为实参传入
(名字为 `self`), 同时, 额外的参数 (`args`) 也被传入

构造子 (constructor)

类实例化 (Class instantiation)

```
class Product:
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0

pina_bar = Product("Piña Chicolotta", 7.99, ["200 calories", "24 g sugar"])
```



1 object

3
Name: "Piña Chicolotta"
Price: 7.99
nutrition_info: ["200 calories", "24 g sugar"]

实例变量

● 实例变量就是对象所包含的数据属性

```
class Product:
```

```
    def __init__(self, name, price, nutrition_info):
```

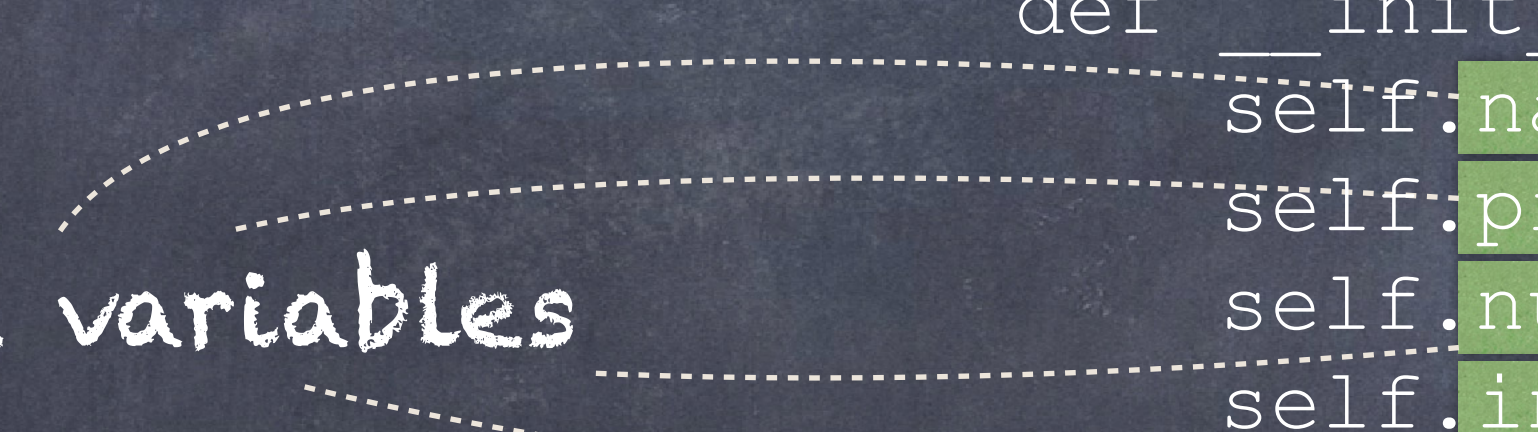
```
        self.name = name
```

```
        self.price = price
```

```
        self.nutrition_info = nutrition_info
```

```
        self.inventory = 0
```

Instance variables

A dashed white line originates from the text 'Instance variables' and branches out to point at the four lines of code in the __init__ method: 'self.name = name', 'self.price = price', 'self.nutrition_info = nutrition_info', and 'self.inventory = 0'. This indicates that these four lines represent instance variables.

对象同一性

每次调用 `<class name> (args)` 构造的对象都是一个不同的个体

可以使用 `is` 来判断

```
>>> a = Product("Piña Chicolotta", 7.99,
["200 calories", "24 g sugar"])
>>> b = Product("Piña Chicolotta", 7.99,
["200 calories", "24 g sugar"])
>>> c = a
>>> a is c
True
>>> a is b
False
```

```
>>> a == b
```



```
__eq__(self, other)
```


方法 (method)

- 方法就是定义在类里的函数

- def语句会创建一个函数对象，

不过与普通函数不同的是，这些函数的名字被绑定在类的属性上（而不是某个实例的名字）

```
class Product:
    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0

    # Define methods
    def increase_inventory(self, amount):
        self.inventory += amount
    def reduce_inventory(self, amount):
        self.inventory -= amount

    def get_label(self):
        return "Foxolate Shop: " + self.name
    def get_inventory_report(self):
        if self.inventory == 0:
            return "There are no bars!"
        return f"There are {self.inventory} bars."
```


方法调用 (invoking method)

- 所有被调用的方法都通过 `self` 参数来访问对象，他们可以访问和修改对象的状态
- “点”的标记会自动提供 `self` 的实参

```
class Product:
```

```
# Define methods
```

```
def increase_inventory(self, amount):  
    self.inventory += amount
```

```
def reduce_inventory(self, amount):  
    self.inventory -= amount
```

Defined with two parameters

invoked with one argument

`pina_bar.increase_inventory(2)`

Bound to self

等价于 `Product.increase_inventory(pina_bar, 2)`

方法调用 (invoking method)

- 可以获取对象实例变量的值的方法就是该对象的选择子 (selector) 或者叫访问子 (accessor)
- 可以改变对象实例变量的值的方法就是该对象的变异子 (mutator)

方法调用 (invoking method)

```
class Product:
```

```
    # Define methods
```

```
    def increase_inventory(self, amount):  
        self.inventory += amount
```

```
    def reduce_inventory(self, amount):  
        self.inventory -= amount
```

```
    def get_label(self):  
        return "Foxolate Shop: " + self.name
```

```
    def get_inventory_report(self):  
        if self.inventory == 0:  
            return "There are no bars!"  
        return f"There are {self.inventory} bars."
```

mutator

accessor

内建的访问子和变异子

`__getattribute__(self, name)` 可以获得对象里名字为“name”的属性

`__setattr__(self, name, value)` 可以将对象里名字为“name”的属性的值
改变为新的“value”

```
class Product:
    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0
    def get_label(self):
        return "Foxolate Shop: " + self.name

pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])
pina_bar.__getattribute__("name")
pina_bar.__setattr__("price", 100)
pina_bar.__getattribute__("get_label")()
```

也可以通过 `getattr(obj, name)`, `setattr(obj, name, value)` 做到

点表达式 (dot expressions)

点表达式

`<expression> . <name>`

`<expression>` 可以是任何python的合法表达式，只要其求值为一个对象的引用

`<name>` 必须是一个简单的名字

点表达式会求值为`<expression>`所求值的对象中属性的名字为`<name>`的值

```
pina_bar.name
```

```
pina_bar.increase_inventory(2)
```

getattr and dot expressions look up a name in the same way

点表达式的属性查找

`<expression> . <name>`

- 首先查找该对象中的相应属性，如果找到，直接返回其值
- 否则，就去找类中名字为`<name>`的属性，即类属性，返回其值
- 如果类中也没有相应的属性，会产生一个`AttributeError`的异常

类变量 (Class variables)

- 类变量是在类中 (但不在方法中) 包含的数据属性
- 类变量被该类的所有实例 (对象) 所共享, 因为其不属于哪一个对象, 而是隶属于类自身的属性

```
class Product:
```

```
    sales_tax = 0.07
```

```
    def get_total_price(self, quantity):
```

```
        return (self.price * (1 + self.sales_tax)) * quantity
```

```
pina_bar = Product("Piña Chicolotta", 7.99, ["200 calories", "24 g sugar"])
```

```
truffle_bar = Product("Truffalapagus", 9.99, ["170 calories", "19 g sugar"])
```

```
pina_bar.sales_tax
```

```
truffle_bar.sales_tax
```

```
pina_bar.get_total_price(4)
```

```
truffle_bar.get_total_price(4)
```

An assignment inside the class that isn't inside a method body.

类的属性 vs 对象的属性

- 在python中，类也是first-class citizen，他们也是一种对象（可以被赋值、传参、返回等）
- 因此，类也有自己的属性
- 类的属性可以通过dot expression进行访问和修改

类的属性 vs 对象的属性

```
class Product:
    sales_tax = 0.07
    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0

    # Define methods
    def increase_inventory(self, amount):
        self.inventory += amount
    def reduce_inventory(self, amount):
        self.inventory -= amount

    def get_label(self):
        return "Foxolate Shop: " + self.name
    def get_inventory_report(self):
        if self.inventory == 0:
            return "There are no bars!"
        return f"There are {self.inventory} bars."
```

```
pina_bar = Product("Piña Chocolotta", 7.99,
                  ["200 calories", "24 g sugar"])
```

```
>>> dir(Product)
>>> dir(pina_bar)
>>> type(Product.get_label)
>>> type(pina_bar.get_label)
```

A method object is created by packing the object instance and function object together

类的属性 vs 对象的属性

```
class Product:
    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0
```

```
pina_bar = Product("Piña Chicolotta", 7.99,
    ["200 calories", "24 g sugar"])
```

```
>>> type(pina_bar)
<class '__main__.Product'>
>>> type(Product)
<class 'type'>
```

```
>>> type(list)
<class 'type'>
```

```
>>> type(int)
<class 'type'>
```

```
>>> type(str)
<class 'type'>
```

我们用类去定义对象:

`type(my_object) -> MyClass`

类本身是一个对象, 那什么来定义类? ->

`MetaClass` (元类别, 即类的类)

`type(MyClass) -> MetaClass`

`type`就是python中的`MetaClass`

```
def consPro(self, name, price, nutrition_info):
    self.name = name
    self.price = price
    self.nutrition_info = nutrition_info
    self.inventory = 0

Product = type('Product', (object,), dict(__init__=consPro))
pina_bar = Product("Piña Chicolotta", 7.99,
    ["200 calories", "24 g sugar"])
type(Product)
type(pina_bar)
```


类的属性 vs 对象的属性

● 点表达式对属性的赋值：

`<expression>.<name> = <value>`

- 如果 `<expression>` 是一个对象实例，那么该表达式对实例的属性进行赋值（即使 `<name>` 在类里面也有相应的属性）
- 如果 `<expression>` 是一个类，那么该表达式对类的属性进行赋值

类的属性 vs 对象的属性

```
class Product:
    sales_tax = 0.07
    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0
```

```
pina_bar = Product("Piña Chocolotta", 7.99,
["200 calories", "24 g sugar"])
truffle_bar = Product("Truffalapagus", 9.99,
["170 calories", "19 g sugar"])
```

```
>>> Product.sales_tax = 1
>>> pina_bar.sales_tax = 2
>>> Product.sales_tax
>>> truffle_bar.sales_tax
>>> pina_bar.sales_tax
```


封装性

公有和私有 (Public vs. Private)

- 属性都是公有的

- 只要你能得到一个对象的引用，你就能访问和修改其属性

```
pina_bar = Product("Piña Chicolotta", 7.99,  
    ["200 calories", "24 g sugar"])
```

```
current = pina_bar.inventory  
pina_bar.inventory = 5000000  
pina_bar.inventory = -5000
```

- 对于python这种动态语言来说，甚至可以添加对象属性

```
pina_bar.brand_new_attribute_haha = "instanception"
```

不在 `__init__` 里 (这称为动态属性)

"私有"属性

- 私有属性无法通过dot expression来访问该属性，只能通过类提供的访问子、变异子来访问和修改该属性
- 为了能够表达属性的访问层次（python没有提供真正的private支持，只是一种开发人员的约定convention）：

__ (double underscore) before very private attribute names
_ (single underscore) before semi-private attribute names
no underscore before public attribute names

```
class Product:
    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self._price = price
        self.__nutrition_info = nutrition_info
```

```
pina_bar = Product("Piña Chocolotta", 7.99,
                  ["200 calories", "24 g sugar"])
pina_bar.name
pina_bar._price
pina_bar.__nutrition_info
```


封装 (encapsulation)

- 对象的实例变量可以表达对象的状态
- oop提倡通过方法的调用来管理对象的状态，不必关心其内部实现（即利用dot expression去访问改变）

```
>>> pina_bar = Product("Piña Chicolotta", 7.99,  
                        ["200 calories", "24 g sugar"])
```

```
>>> pina_bar.get_inventory_report()  
"There are NO bars!"
```

```
>>> pina_bar.increase_inventory(3)  
>>> pina_bar.get_inventory_report()  
"There are 3 bars total (worth $23.97 total)."
```

这就是OOP的封装性

Any questions ?