

# 面向对象

---

什么是面向对象呢？

一种认识世界、分析世界的方法论。将万事万物抽象为各种对象。

## 类class

类是抽象的概念，是万事万物的抽象，是一类事物的共同特征的集合。

用计算机语言来描述类，是**属性**和**方法**的集合。

## 对象instance、object

对象是类的具象，是一个实体。

对于我们每个人这个个体，都是抽象概念**人类**的不同的**实体**。

举例：

你吃鱼

你，就是对象；鱼，也是对象；吃就是动作

你是具体的人，是具体的对象。你属于人类，人类是个抽象的概念，是无数具体的人的个体的抽象。

鱼，也是具体的对象，就是你吃的这一条具体的鱼。这条鱼属于鱼类，鱼类是无数的鱼抽象出来的概念。

吃，是动作，也是操作，也是方法，这个吃是你的动作，也就是人类具有的方法。如果反过来，鱼吃人。吃就是鱼类的动作了。

吃，这个动作，很多动物都具有的动作，人类和鱼类都属于动物类，而动物类是抽象的概念，是动物都有吃的动作，但是吃法不同而已。

你驾驶车，这个车也是车类的具体的对象（实例），驾驶这个动作是鱼类不具有的，是人类具有的方法。

**属性**：对对象状态的抽象，用数据结构来描述。

**操作**：对对象行为的抽象，用操作名和实现该操作的方法来描述。

每个人都是人类的一个单独的实例，都有自己的名字、身高、体重等信息，这些信息是个人的属性，但是，这些信息不能保存在人类中，因为人类是抽象的概念，不能保留每个具体的个体的值。

而人类的实例，是具体的人，他可以存储这些具体的属性，而且可以不同人有不同的属性。

哲学

- 一切皆对象
- 对象是数据和操作的封装
- 对象是独立的，但是对象之间可以相互作用
- 目前OOP是最接近人类认知的编程范式

# 面向对象3要素

---

## 1. 封装

- 组装：将数据和操作组装到一起。
- 隐藏数据：对外只暴露一些接口，通过接口访问对象。比如驾驶员使用汽车，不需要了解汽车的构造细节，只需要知道使用什么部件怎么驾驶就行，踩了油门就能跑，可以不了解其中的机原理。

## 2. 继承

- 多复用，继承来的就不用自己写了
- 多继承少修改，OCP（Open-closed Principle），使用继承来改变，来体现个性

## 3. 多态

- 面向对象编程最灵活的地方，动态绑定

人类就是封装；  
人类继承自动物类，孩子继承父母特征。分为单一继承、多继承；  
多态，继承自动物类的人类、猫类的操作“吃”不同。

## 封装

封装就是定义类，将属性和操作组织在类中

## Python类定义

```
1 class ClassName:
2     语句块
```

1. 必须使用class关键字
2. 类名强烈建议使用**大驼峰**命名方式，即每个单词首字母大写。其本质上就是一个标识符
3. 类定义完成后，就产生了一个**类对象**，绑定到了标识符ClassName上

举例

```
1 class Person:
2     """A Example Class"""
3     x = 'abc' # 类属性
4
5     def showme(self): # 方法，也是类属性
6         return __class__.__name__ # 返回当前类的名称
7
8 print(Person)
9 print(Person.__name__) # 类名字
10 print(Person.__doc__) # 类文档
11 print(Person.showme) # 类属性
```

## 类及类属性

- **类对象**：类也是对象，类的定义执行后会生成一个类对象
- **类属性**：类定义中的变量和类中定义的**方法**都是类的属性。上例中类Person的x和showme
- **类变量**：属性也是标识符，也是变量。上例中类Person的x和showme

Person中，x、showme都是类的属性，`__name__`、`__doc__`是类的特殊属性

showme方法是类的属性，如同**吃是人类的方法**，但是**每一个具体的人才能吃东西**，也就是说**吃是人的实例**能调用的方法。

showme是**方法method**，本质上就是普通的函数对象function，它一般要求至少有一个参数。第一个形式参数可以叫做self（self只是个惯用标识符，可以换名字），这个参数位置就留给了self。

**self 指代当前实例本身。**

问题：上例中，类是谁？实例是谁？

## 实例化

```
1 a = Person() # 实例化
2 print(a)
3
4 <__main__.Person object at 0x000001F1F4B5C370>
5 __main__是模块
6 Person是类
7 Person object说明它是Person类的实例
8 0x000001F1F4B5C370是实例的内存地址的十六进制表达
```

使用上面的语法，在类对象名称后面加上一个括号，就调用类的实例化方法，完成实例化。  
实例化就真正创建一个该类的对象（实例instance）。例如

```
1 tom = Person() # 不同的实例
2 jerry = Person() # 不同的实例
```

上面的tom、jerry都是Person类的实例，通过实例化生成了2个**不同的实例**。

通常，每次实例化后获得的实例，是**不同的实例**，即使是使用同样的参数实例化，也得到不一样的对象。

Python类**实例化**后，会自动调用 `__init__` 方法。这个方法第一个形式参数必须留给self，其它形式参数随意。

## 构造的2个阶段

确切地讲，`tom = Person()` 过程分为2个阶段：实例化和初始化。

如同，流水线上生成一辆汽车，首先得先造一个车的实例，即造一辆实实在在的一个真实的车。但是这个车不能直接交付给消费者。

而 `__init__` 方法称为初始化方法，要对生成出的每一辆车做出厂配置。这样才能得到一个能使用的汽车。

但是需要注意的是，很多人习惯上把这两个阶段不加区分含糊的叫做实例化、初始化，说的就是这两个阶段的总称。

### `__init__`方法

有些人把Python的 `__init__` 方法称为构造方法或构造器。

`Person()`实例化后，要初始化，要调用的是 `__init__(self)` 方法，可以不定义，如果没有定义会在实例化后**隐式**调用其父类的。

作用：对实例进行**初始化**

```
1 class Person:
2     def __init__(self):
3         print('init~~~~~')
4
5 print(Person) # 不会调用__init__
6 print(Person()) # 会调用__init__
7 tom = Person() # 会调用__init__
```

初始化函数可以多个参数，请注意第一个位置必须是self，例如 `__init__(self, name, age)`

```

1 class Person:
2     def __init__(self, name, age):
3         print('init~~~~~')
4         self.name = name
5         self.age = age
6
7     def showage(self):
8         print("{} is {}".format(self.name, self.age))
9
10 tom = Person('Tom', 20) # 实例化，会调用__init__方法并为实例进行属性的初始化
11 print(tom.name, tom.age)
12 tom.showage()
13
14 jerry = Person('Jerry', 18)
15 print(jerry.name, jerry.age)
16 jerry.age += 1
17 print(jerry.name, jerry.age)
18 jerry.showage()

```

注意：\_\_init\_\_() 方法**不能有返回值**，也就是只能是return None

## 实例对象instance

上例中，类Person实例化后获得一个该类的实例，就是**实例对象**。

上例中的tom、jerry就是Person类的实例。

**\_\_init\_\_** 方法的第一参数 self 就是**指代某一个实例自身**。

执行 Person('Tom', 20) 时，调用 \_\_init\_\_() 方法。self.name就是tom对象的name，name是保存在了tom对象上，而不是Person类上。所以，称为**实例变量**。

类实例化后，得到一个实例对象，调用方法时采用tom.showage()的方式，但是showage方法的形参需要一个形参self，我们并没有提供，并没有报错，为什么？

## 方法绑定

采用tom.showage()的方式调用，实例对象会**绑定**到方法上。这个self就是tom，**指向当前调用该方法的实例本身**。

tom.showage()调用时，会把方法的调用者tom实例作为第一参数self的实参传入\_\_init\_\_()方法。

## self

```

1 class Person:
2     def __init__(self):
3         print(1, 'self in init = {}'.format(id(self)))
4
5     def showme(self):
6         print(2, 'self in showme = {}'.format(id(self)))
7
8 tom = Person()
9 print(3, 'tom = {}'.format(id(tom)))
10 print('-' * 30)
11 tom.showme()
12
13 # 打印结果为

```

```

14 1 self in init = 2130444422560
15 3 tom = 2130444422560
16 -----
17 2 self in showme = 2130444422560

```

上例说明，self就是调用者，就是tom对应的实例对象。

self这个形参标识符的名字只是一个惯例，它可以修改，但是请不要修改，否则影响代码的可读性。

看打印的结果，思考一下执行的顺序，为什么？

## 实例变量和类变量

```

1  class Person:
2      age = 3
3      def __init__(self, name):
4          self.name = name
5
6      #tom = Person('Tom', 20) # 错误，只能传一个实参
7      tom = Person('Tom')
8      jerry = Person('Jerry')
9      print(tom.name, tom.age)
10     print(jerry.name, tom.age)
11     #print(Person.name) # 能访问吗？
12     print(Person.age)
13
14     Person.age = 30
15     print(Person.age, tom.age, jerry.age) # age分别是多少？
16
17
18     # 运行结果
19     Tom 3
20     Jerry 3
21     3
22     30 30 30

```

- 实例变量是每一个实例自己的变量，是自己独有的
- 类变量是类的变量，是类的所有实例共享的属性或方法

## 特殊属性

特殊属性	含义
<code>__name__</code>	对象名
<code>__class__</code>	对象的类型
<code>__dict__</code>	对象的属性的字典
<code>__qualname__</code>	类的限定名

注意：

Python中每一种对象都拥有不同的属性。函数是对象，类是对象，类的实例也是对象。

## 属性本质

```
1 class Person:
2     age = 3
3     def __init__(self, name):
4         self.name = name
5
6 print('----类----')
7 print(Person.__class__, type(Person), Person.__class__ is type(Person)) # 类型
8 print(sorted(Person.__dict__.items()), end='\n\n') # 类字典
9
10
11 tom = Person('Tom')
12 print('----通过实例访问类----')
13 print(tom.__class__, type(tom), tom.__class__ is type(tom))
14 print(tom.__class__.__name__, type(tom).__name__)
15 print(sorted(tom.__class__.__dict__.items()))
16 print('----实例自己的属性----')
17 print(sorted(tom.__dict__.items())) # 实例的字典
```

上例中，可以看到类属性保存在类的 `__dict__` 中，实例属性保存在实例的 `__dict__` 中，如果从实例访问类的属性，**也可以借助 `__class__`** 找到所属的类，再通过类来访问类属性，例如 `tom.__class__.age`。

有了上面知识，再看下面的代码

```
1 class Person:
2     age = 3
3     height = 170
4
5     def __init__(self, name, age=18):
6         self.name = name
7         self.age = age
8
9 tom = Person('Tom') # 实例化、初始化
10 jerry = Person('Jerry', 20)
11
12 Person.age = 30
13 print(1, Person.age, tom.age, jerry.age) # 输出什么结果
14
15 print(2, Person.height, tom.height, jerry.height) # 输出什么结果
16 jerry.height = 175
17 print(3, Person.height, tom.height, jerry.height) # 输出什么结果
18
19 tom.height += 10
20 print(4, Person.height, tom.height, jerry.height) # 输出什么结果
21
22 Person.height += 15
23 print(5, Person.height, tom.height, jerry.height) # 输出什么结果
24
25 Person.weight = 70
26 print(6, Person.weight, tom.weight, jerry.weight) # 输出什么结果
27
```

```
28 print(7, tom.__dict__['height']) # 可以吗
29 print(8, tom.__dict__['weight']) # 可以吗
```

## 总结

是类的，也是这个类所有实例的，其实例都可以访问到；

是实例的，就是这个实例自己的，通过类访问不到。

类变量是属于类的变量，这个类的所有实例可以**共享**这个变量。

对象（实例或类）可以动态的给自己增加一个属性（赋值即定义一个新属性）。这也是动态语言的特性。

实例.\_\_dict\_\_[变量名] 和 实例.变量名 都可以访问到实例自己的属性（注意这两种访问是有本质区别的）。

对实例访问来说，实例的同名变量会**隐藏**掉类变量，或者说是覆盖了这个类变量。但是注意类变量还在那里，并没有真正被覆盖。

## 实例属性的查找顺序

指的是实例使用 .点号 来访问属性，会先找自己的 \_\_dict\_\_，如果没有，然后通过属性 \_\_class\_\_ 找到自己的类，再去类的 \_\_dict\_\_ 中找

注意：如果实例使用 \_\_dict\_\_[变量名] 访问变量，将不会按照上面的查找顺序找变量了，这是指明使用字典的key查找，不是属性查找。

一般来说，**类变量可使用全大写来命名**。

## 类方法和静态方法

前面的例子中定义的 \_\_init\_\_ 等方法，这些方法本身都是类的属性，第一个参数必须是self，而self必须指向一个对象，也就是类实例化之后，由实例来调用这个方法。

## 普通函数

```
1 class Person:
2     def general_function():
3         print('普通的函数')
4
5 print(Person.general_function) # 类中定义的标识符，依然是类属性
6 Person.general_function()      # 可以吗？
7 Person().general_function()    # 可以吗？
8 print(Person.__dict__)         # 是类属性吗？
```

使用类直接调用该方法，没有任何实例的绑定，直接调用即可。

实例调用类属性定义的方法时，会绑定实例到该方法上，general\_function定义时，没有指定一个形参，导致绑定对象注入时没有对应的形参。

上例只是为了学习原理，在Python类定义的时候，**禁用**这种定义方式。

## 普通方法

下面定义的方式，是最常见的定义方法的方式，应该重点掌握。

```

1 class Person:
2     def showself(self):
3         print('普通的方法，和我自己有关。self={}'.format(self))
4
5 Person.showself()          # 可以吗？
6 Person().showself()       # 可以吗？
7 print(Person.__dict__)    # 是类属性吗？
8 print(Person().showself)  # 看看实例绑定

```

showself是类属性，也是方法。

Person.showself()即使用类访问该方法，没有实例绑定，那么第一参数self缺失而报错。

Person().showself()即使用实例访问该方法，有实例绑定，实参Person()是自动注入的。

注意普通方法定义中self，往往提醒这个方法和实例有关。

## 类方法

类方法，相对普通方法使用较少。

```

1 class Person:
2     @classmethod
3     def class_method(cls):
4         print('类方法')
5         print("{}'s name = {}'.format(cls.__name__, cls))
6         cls.HEIGHT = 170
7
8 # 调用
9 Person.class_method()    # 可以吗？
10 Person().class_method() # 可以吗？
11 print(Person.__dict__)  # 是类属性吗？

```

## 类方法

1. 在类定义中，使用@classmethod装饰器修饰的方法
2. 必须至少有一个参数，且第一个参数留给了cls，cls指代调用者即类对象自身
3. cls这个标识符可以是任意合法名称，但是为了易读，请不要修改
4. 通过cls可以直接操作类的属性

通过类、实例都可以非常方便地调用类方法。classmethod装饰器内部将类（或提取实例的类）注入到类方法的第一个参数中。

注意：无法通过cls操作类的实例。为什么？

## 静态方法

静态方法，用的极少，偶尔在源码中见到。



```

1 class Person:
2     HEIGHT = 180
3
4     @staticmethod
5     def static_method():
6         print('静态方法')
7         print(Person.HEIGHT)
8
9 # 调用
10 Person.static_method()      # 可以吗?
11 Person().static_method()   # 可以吗?
12 print(Person.__dict__)     # 是类属性吗?

```

## 静态方法

1. 在类定义中，使用@staticmethod装饰器修饰的方法
2. 调用时，不会隐式的传入参数

通过类、实例都可以调用静态方法，**不会**像普通方法、类方法那样注入参数。

静态方法，只是表明这个方法属于这个名词空间。函数归在一起，方便组织管理。

## 方法的调用

类可以定义这么多种方法，究竟如何调用它们？

类几乎可以调用所有内部定义的方法，但是调用 **普通的方法** 时会报错，原因是第一参数应该是类的实例。

实例也几乎可以调用所有的方法，**普通的函数** 的调用一般不可能出现，因为原则上不允许这么定义。

总结：

- 类除了普通方法都可以调用
- 普通方法需要对象的实例作为第一参数
- 实例可以调用所有类中定义的方法（包括类方法、静态方法），普通方法传入实例自身，静态方法和类方法内部都要使用实例的类

回答下面问题

```

1 class Person:
2     def method(self):
3         print(self)          # 实例自身
4         print(__class__)     # 当前类
5         print(__class__.__name__) # 当前类名字
6         print(__name__)      # 当前模块名
7
8 tom = Person()
9 tom.method()                # 可以吗
10 Person.method(1)           # 可以吗
11 Person.method(tom)         # 可以吗
12 tom.__class__.method(tom)  # 可以吗

```

tom.method()调用的时候，会绑定实例，调用method方法时，实例tom会注入到method中，这样第一参数就满足了。

Person.method(), 使用类调用, 不会有实例绑定, 调用method方法时, 就缺少了第一参数, 可以手动的填入。

思考:

- `print(sorted(['a', 'Ab', '2', 'Abc']))`, 元素按照小写进行排序, 如何做?

## 访问控制

### 私有 (Private) 成员

在Python中, 在类变量或实例变量前使用两个下划线的变量, 称为私有成员, 包括私有属性、私有方法。

```
1 class Person:
2     def __init__(self, name, age=18):
3         self.__name = name
4         self.__age = age
5
6     def __showage(self):
7         print(self.__age)
8
9 print(Person.__name)      # 可以吗?
10 print(Person.__showage)  # 可以吗?
11 tom = Person('Tom')
12 print(tom.__name)        # 可以吗?
13 print(tom.__showage)     # 可以吗?
```

在类的定义范围内, 使用前置双下划线的标识符, 在类外部将不能直接访问。

### 私有成员本质

```
1 class Person:
2     def __init__(self, name, age=18):
3         self.__name = name
4         self.__age = age
5
6     def __showage(self):
7         print(self.__age)
8
9 print(Person.__dict__)
10 tom = Person('Tom')
11 print(tom.__dict__)
```

打开类字典和实例字典, 一目了然, 都被悄悄的改了名称, 所以使用定义的名字就访问不了了。名称都被前置了 `__类名` 前缀。

如果知道了改后的名称, 照样可以访问, 就绕过了Python做的限制。

Python就没有真正的私有成员! 但是请遵守这个约定, 不要在类外面访问类私有或者实例的私有成员。因为类的作用就是封装, 私有成员就是要被隐藏的数据或方法。

## 保护成员

在类变量或实例变量前使用一个下划线的变量，称为保护成员。

```
1 class Person:
2     def __init__(self, name, age=18):
3         self._name = name
4         self._age = age
5
6     def _showage(self):
7         print(self._age)
8
9 print(Person.__dict__)
10 tom = Person('Tom')
11 print(tom.__dict__)
12 tom._showage() # 可以吗
13 print(tom._name, tom._age) # 可以吗
```

保护成员不是Python中定义的，是Python编程者自我约定俗成的。请遵守这个约定。

## 总结

在Python中使用 `_` 单下划线 或者 `__` 双下划线来标识一个成员被保护或者被私有化隐藏起来。

但是，不管使用什么样的访问控制，都不能真正的阻止用户修改类的成员。Python中没有绝对的安全的保护成员或者私有成员。

因此，前导的下划线只是一种警告或者提醒，请遵守这个约定。除非真有必要，不要修改或者使用保护成员或者私有成员，更不要修改它们。

在Pycharm中，已经对访问私有、保护成员访问的时候不会直接提示，就是一种善意的提醒。

## 属性装饰器

一般好的设计是：把实例的某些属性保护起来，不让外部直接访问，外部使用getter读取属性和setter方法设置属性。

```
1 class Person:
2     def __init__(self, name):
3         self._name = name
4
5     def name(self):
6         return self._name
7
8     def set_name(self, value):
9         self._name = value
10
11 tom = Person('Tom')
12 print(tom.name())
13 tom.set_name('Jerry')
14 print(tom.name())
```

Python提供了property装饰器，简化调用。

```
1 class Person:
```

```

2     def __init__(self, name):
3         self._name = name
4
5     @property
6     def name(self):
7         return self._name
8
9     @name.setter
10    def name(self, value):
11        self._name = value
12
13    @name.deleter
14    def name(self):
15        #del self._name
16        print('del name')
17
18    tom = Person('Tom')
19    print(tom.name)
20    tom.name = 'Jerry'
21    print(tom.name)
22    del tom.name

```

特别注意：使用property装饰器的时候这三个方法同名

property装饰器

- 后面跟的函数名就是以后的属性名。它就是getter。这个必须有，有了它至少是只读属性
- setter装饰器
  - 与属性名同名，且接收2个参数，第一个是self，第二个是将要赋值的值。有了它，属性可写
- deleter装饰器
  - 可以控制是否删除属性。很少用
- property装饰器必须在前，setter、deleter装饰器在后
- property装饰器能通过简单的方式，把对方法的操作变成对属性的访问，并起到了一定隐藏效果

其它写法

```

1  class Person:
2      def __init__(self, name):
3          self._name = name
4
5      def get_name(self):
6          return self._name
7
8      def set_name(self, value):
9          self._name = value
10
11     def del_name(self):
12         #del self._name
13         print('del name')
14
15     name = property(get_name, set_name, del_name)
16

```

```
17 tom = Person('Tom')
18 print(tom.name)
19 tomname = 'Jerry'
20 print(tom.name)
21 del tom.name
```

这种定义方式，适合get\_name、set\_name、del\_name还可以单独使用，即可以当方法使用。

## 封装总结

面向对象的三要素之一，封装Encapsulation

封装

- 将数据和操作组织到类中，即属性和方法
- 将数据隐藏起来，给使用者提供操作（方法）。使用者通过操作就可以获取或者修改数据。getter和setter
- 通过访问控制，暴露适当的数据和操作给用户，该隐藏的隐藏起来，例如保护成员或私有成员