■对象的内置方法\_\_init\_\_

# 当使用class创建对象时，会自动执行以下操作：  
# 为对象在内存中分配空间，然后创建对象。  
# 为对象的属性设置初始值，初始方法\_\_init\_\_  
  
# \_\_init\_\_方法是专门用来定义一个类具有哪些属性的方法  
  
  
class Ta:  
  
 def \_\_init\_\_(self):

#self.属性名 = 属性的初始值  
 self.name = 'Tom'  
 self.age = 22  
  
  
a = Ta()  
print(a.name)  
print(a.age)  
# 输出：  
# Tom  
# 22

# 通过传参来设置初始值  
  
class Ta:  
  
 def \_\_init\_\_(self, name1, age1):  
 self.name = name1  
 self.age = age1  
  
  
a = Ta('Jery', 25)  
print(a.name)  
print(a.age)  
# 输出：  
# Jery  
# 25

■对象的内置方法\_\_del\_\_

\_\_init\_\_和\_\_del\_\_是相对的。\_\_del\_\_是在对象被从内存中销毁之前，系统自动调用的方法。

# 对象的生命周期和内置属性\_\_del\_\_  
  
  
class Ta:  
  
 def \_\_init\_\_(self):  
 print('你好我来了')  
  
 def \_\_del\_\_(self):  
 print('再见我走了')  
  
  
a = Ta()  
print('-' \* 20)  
# 输出如下：  
# 你好我来了  
# --------------------  
# 再见我走了  
  
# 说明：  
# 当对象被创建，系统自动调用\_\_init\_\_  
# 当代码执行完，对象a要从内存中被销毁了，系统会自动执行\_\_del\_\_  
# 因此在分割线执行完，说明程序执行完了，对象a要被销毁了，所以在分割线下又打印了‘再见我走了’

# 对象的生命周期和内置属性\_\_del\_\_  
  
  
class Ta:  
  
 def \_\_init\_\_(self):  
 print('你好我来了')  
  
 def \_\_del\_\_(self):  
 print('再见我走了')  
  
  
a = Ta()  
del a  
print('-' \* 20)  
# 输出如下：  
# 你好我来了  
# 再见我走了  
# --------------------  
  
# 说明：  
# 当对象被创建，系统自动调用\_\_init\_\_  
# 当对象a要从内存中被销毁了，系统会自动执行\_\_del\_\_  
# 这个例子中不是程序结束而注销a的，而是通过del方法在程序过程中销毁a，随意a销毁前系统执行\_\_del\_\_  
# 所以‘再见我走了’在分割线之前打印

■\_\_str\_\_

# 当用print输出对象变量，默认情况下会输出：  
# 这个变量引用的对象是哪一个类创建的对象，以及在内存中的地址  
  
class Aa:  
  
 def \_\_init\_\_(self):  
 self.name = 'Tom'  
  
  
a = Aa()  
print(a)  
# 输出：<\_\_main\_\_.Aa object at 0x0000020444011390>

# \_\_str\_\_内置方法可以自定义print对象时的输出结果  
# 注意\_\_str\_\_必须返回一个字符串  
  
  
class Aa:  
  
 def \_\_init\_\_(self):  
 self.name = 'Tom'  
  
 def \_\_str\_\_(self):  
 return '哈哈我是%s' % self.name  
  
  
a = Aa()  
print(a)  
# 输出：哈哈我是Tom

■私有变量

# 如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，  
# 在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），  
# 只有内部可以访问，外部不能访问  
  
  
class Test(object):  
  
 def \_\_init\_\_(self):  
 self.\_\_name = 'Tom'  
 self.age = 25  
  
 # 可以用这个函数将私有变量暴露出去  
 def get\_name(self):  
 return self.\_\_name  
  
 # 也可以用这个函数来让外部可以修改私有变量  
 def set\_name(self, name):  
 self.\_\_name = name  
  
 def pt(self):  
 print(self.\_\_name)  
  
  
a = Test()  
print(a.age)  
# 输出：25  
  
# 在外部无法访问私有变量\_\_name  
print(a.\_\_name)  
# 输出: AttributeError: 'Test' object has no attribute '\_\_name'  
  
# 而内部的pt函数就能访问内部的私有变量\_\_name  
a.pt()  
# 输出： Tom  
  
# 通过访问get\_name获取到私有变量  
print(a.get\_name())  
# 输出： Tom

■继承和多态

class Fat(object):  
  
 def \_\_init\_\_(self):  
 self.name = 'fuqin'  
 self.age = 35  
  
  
class Son(Fat):  
 pass  
  
  
a = Son()  
print(a.name)  
# 输出：fuqin  
  
# 可以在子类里用新的值来覆盖父类里的age属性  
a.age = 38  
  
# 也可以在子类里创建新属性  
a.jineng = 'codding'  
  
print(a.age)  
# 输出：38  
print(a.jineng)  
# 输出：codding  
  
# a的类型是Son同时也是其父类Fat  
print(isinstance(a, Son))  
# 输出： True  
  
print(isinstance(a, Fat))  
# 输出： True

■获取对象信息

三种方法

type(), isinstance(), dir()

class Shi(object):  
  
 def \_\_init\_\_(self):  
 self.name = 'Joe'  
 self.age = 26  
  
 def say\_hello(self):  
 print('Hello', self.name)  
  
  
a = Shi()  
  
# type方法查看一个对象的类型  
print(type(a))  
# 输出： <class '\_\_main\_\_.Shi'>  
  
# isinstance方法判断一个对象是否从属于一个类  
print(isinstance(a, Shi))  
# 输出： True  
  
# dir方法列出对象的所有方法和属性  
print(dir(a))  
# 输出：  
# ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_',  
# '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_',  
# '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_',  
# '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_',  
# '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'age', 'name', 'say\_hello']

■\_\_solts\_\_

# 使用 \_\_slots\_\_  
  
  
# 使用\_\_solts\_\_，用tuple定义允许绑定的属性名称  
class Af(object):  
 \_\_slots\_\_ = ('name', 'age')  
  
  
a = Af()  
a.name = 'Tom'  
a.age = 22  
# 这样写会报错，因为\_\_solts\_\_中没有hobby这个属性名  
a.hobby = 'wan'  
  
print(a.name)  
# 输出: Tom  
print(a.age)  
# 输出：22  
print(a.hobby)  
# 输出：AttributeError: 'Af' object has no attribute 'hobby'

■@property

直接读取属性

class Sp(object):  
  
 def \_\_init\_\_(self):  
 self.name = 'Tom'  
 self.age = 22  
  
  
a = Sp()  
a.age = 33  
print(a.age)  
# 输出：33  
# 在这个例子中，在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，  
# 但是，没办法检查参数，导致可以把age随便改

使用对象的方法来读取和修改属性

# 可以通过一个set\_score()方法来设置成绩，再通过一个get\_score()来获取成绩，  
# 这样，在set\_score()方法里，就可以检查参数：  
  
  
class Sp(object):  
  
 def \_\_init\_\_(self):  
 self.name = 'Tom'  
 self.\_score = 60  
  
 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  
  
  
a = Sp()  
  
# 获取属性值  
print(a.get\_score())  
# 输出：60  
  
# 设置属性值  
a.set\_score(99)  
print(a.get\_score())  
# 输出：99

使用@property来把一个方法变成属性来调用

# 上面的调用方法又略显复杂，没有直接用属性这么直接简单。  
# 有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？  
# 还记得装饰器（decorator）可以给函数动态加上功能吗？  
# 对于类的方法，装饰器一样起作用。  
# Python内置的@property装饰器就是负责把一个方法变成属性调用的：  
  
class Student(object):  
  
 def \_\_init\_\_(self):  
 self.name = 'Tom'  
 self.\_score = 66  
  
 @property  
 def score3(self):  
 return self.\_score  
  
 @score3.setter  
 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  
  
  
a = Student()  
print(a.\_score)  
# 输出：66  
  
a.\_score = 99  
print(a.\_score)  
# 输出：99  
  
  
  
# 还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：  
class Student(object):  
  
 @property  
 def birth(self):  
 return self.\_birth  
  
 @birth.setter  
 def birth(self, value):  
 self.\_birth = value  
  
 @property  
 def age(self):  
 return 2015 - self.\_birth  
# 上面的birth是可读写属性，而age就是一个只读属性，因为age可以根据birth和当前时间计算出来。

■枚举类

# 为枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。  
# Python提供了Enum类来实现这个功能：  
from enum import Enum  
  
Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))  
  
for name, member in Month.\_\_members\_\_.items():  
 print(name, '=>', member, ',', member.value)  
  
# value属性则是自动赋给成员的int常量，默认从1开始计数。  
# 输出：  
# Jan => Month.Jan , 1  
# Feb => Month.Feb , 2  
# Mar => Month.Mar , 3  
# Apr => Month.Apr , 4  
# May => Month.May , 5  
# Jun => Month.Jun , 6  
# Jul => Month.Jul , 7  
# Aug => Month.Aug , 8  
# Sep => Month.Sep , 9  
# Oct => Month.Oct , 10  
# Nov => Month.Nov , 11  
# Dec => Month.Dec , 12

■元类

type()

metaclass()

<https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014319106919344c4ef8b1e04c48778bb45796e0335839000>

■定制类

<https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014319098638265527beb24f7840aa97de564ccc7f20f6000>