# 第一部分 用特殊方法实现python风格的类

为了实现更好的可扩展性，Python语句提供了大量的特殊方法。

* 特性访问（Attribute Access）：这类特殊方法实现了对象的特性访问，使用方式为object.attribute,既可以用来赋值，也可以del语句中执行删除操作。
* 可调用对象（Callables）：这个方法的适用对象为参数，l例如\_\_len\_\_（）。
* 集合（Collections）：这类方法提供了很多集合操作的功能。
* 数字（Numbers）：
* 上下文（Context）：这类函数通常使用with语句来实现上下文管理。
* 迭代器（Iterator）：

## \_\_init\_\_方法

### 1.1 隐式的基类

每个python类的定义都会隐式的继承自object类。

Python2.x

旧式类：

>>> class X:pass

>>> dir(X)

['\_\_doc\_\_', '\_\_module\_\_']

新式类：

>>> class X(object):pass

>>> dir(X)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_']

>>> X.\_\_class\_\_

<type 'type'>

>>> X.\_\_class\_\_.\_\_base\_\_

<type 'object'>

>>>

Python3.x

>>> class X:pass

>>> dir(X)

['\_\_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\_\_']

>>> X.\_\_class\_\_

<class 'type'>

>>> X.\_\_class\_\_.\_\_base\_\_

<class 'object'>

>>>

### 1.2 基类中的\_\_init\_\_()方法

Object作为所有类的基类，已经为\_\_init\_\_()方法提供了默认实现。

延迟赋值的实现方式在python中是合法的，但却给调用者带来了潜在的困惑，对于每个\_\_init\_\_()方法，都应显示的指定要初始化的变量。

### 1.3 在基类中实现init()方法

#coding=utf-8

class **Card**(object):

def **\_\_init\_\_**(*self*, rank, suit):

*self*.suit = suit

*self*.rank = rank

*self*.hard, *self*.soft = *self*.\_points()

class **NumberCard**(Card):

def **\_points**(*self*):

return int(*self*.rank), int(*self*.rank)

class **AceCard**(Card):

def **\_points**(*self*):

return 1, 11

class **FaceCard**(Card):

def **\_points**(*self*):

return 10, 10

cards = [AceCard(*'A'*, *'♠'*), NumberCard(*'2'*, *'♠'*), NumberCard(*'3'*, *'♠'*)]

\_\_init\_\_()把公共的初始化方法引入到基类中，每个子类能够共享基类的初始化逻辑。常见的多态设计，每个子类为\_points()方法提供实现。

### 1.4 使用\_\_init\_\_()方法创建常量清单

class **Suit**(object):

def **\_\_init\_\_**(*self*, name, symbol):

*self*.name = name

*self*.symbol = symbol

...

Club, Diamond, Heart, Spade = Suit(*'Club'*, *'♣'*), Suit(*'Diamond'*, *'♦'*), Suit(*'Heart'*, *'♥'*), Suit(*'Spade'*, *'♠'*)

cards = [AceCard(*'A'*, Spade), NumberCard(*'2'*, Spade), NumberCard(*'3'*, Spade)]

### 1.5 通过工厂函数调用\_\_init\_\_()

* 定义一个函数，返回不同类的对象。
* 定义一个类，包含了创建对象的方法。

什么时候使用catch-all语句？仅当条件非常明确时才使用，如果条件不明确，使用else语句将抛出异常。因此避免使用模糊的else语句。

**工厂函数：**

def **cardFactory1**(rank, suit):

if rank == 1:

return AceCard(*'A'*, suit)

elif 2<= rank <11:

return NumberCard(str(rank), suit)

elif rank == 11:

return FaceCard(*'J'*, suit)

elif rank == 12:

return FaceCard(*'Q'*, suit)

elif rank == 13:

return FaceCard(*'K'*, suit)

else:

raise Exception(*"Rank out of range"*)

**使用映射和类**：

def **cardFactory2**(rank, suit):

class\_, rank\_str = {1:(AceCard, *'A'*), 11:(FaceCard, *'J'*), 12:(FaceCard, *'Q'*), 13:(FaceCard, *'K'*)}.get(rank, (NumberCard, str(rank)))

return class\_(rank\_str, suit)

class **CardFactory**(object):

def **rank**(*self*, rank):

*self*.class\_, *self*.rank\_str = {1:(AceCard, *'A'*), 11:(FaceCard, *'J'*), 12:(FaceCard, *'Q'*), 13:(FaceCard, *'K'*)}.get(rank, (NumberCard, str(rank)))

return *self*

def **suit**(*self*, suit):

return *self*.class\_(*self*.rank\_str, suit)

deck = [card.rank(r+1).suit(s) for r in range(13) for s in (Club, Diamond, Heart, Spade)]

### 1.6 在每个子类中实现\_\_init\_\_方法

### 1.7 简单的组合对象

一个对象元素也可以成为容器。

**random.shuffle()** 方法将序列的所有元素随机排序。

* 封装：基于现有的集合类来定义一个新类，属于外观模式的一个使用场景。
* 扩展：对现有集合类进行扩展，通常使用定义子类的方式实现。
* 创建：即重新设计

#### 1.7.1 封装集合类

### 1.7.2 扩展集合类

class **Deck1**(list):

*"""*

*扩展集合类*

*"""*

def **\_\_init\_\_**(*self*):

super().\_\_init\_\_(CardFactory().rank(rank).suit(s) for rank in range(1,14) for s in [Club, Diamond, Heart, Spade])

random.shuffle(*self*)

#### 1.7.3 可适应更多需求的一种设计

#### 1.12.2 初始化、封装和私有

Python中关于可见度的命名规则如下：

* 大部分名称是共有的
* 以\_\_开始的名字通常不完全公有。使用它们来命名那些经常变化的函数，这些函数通常是实现细节。
* 以\_\_作为前缀和后缀的函数通常是Python内部的。程序中不该使用；命名要参考编程语言的定义。

## 与python无缝集成

### 2.1 \_\_repr\_\_()和\_\_str\_\_()方法

尽管str(),repr()和``运算在特性和功能方面都非常相似，事实上repr()和``做的是完全一样的事情，它们返回的是一个对象的“官方”字符串表示，也就是说绝大多数情况下可以通过求值运算（使用内建函数eval()）重新得到该对象。

但str()则有所不同，str()致力于生成一个对象的可读性好的字符串表示，它的返回结果通常无法用于eval()求值，但很适合用于print语句输出。需要再次提醒的是，并不是所有repr()返回的字符串都能够用 eval()内建函数得到原来的对象。 也就是说 repr() 输出对 Python比较友好，而str()的输出对用户比较友好。

有以下两种情况，我们可以考虑重写\_\_str\_\_()和\_\_repr\_\_()

* 非集合对象：一个不包括任何其他集合对象的“简单”对象，这类对象的格式化通常不会特别复杂。
* 集合对象：一个包含集合的对象，这类集合的对象的格式化会更复杂。
* 打印操作会首先尝试\_\_str\_\_和str内置函数。
* \_\_repr\_\_用于所有其他环境：用于交互模式下提示回应以及repr函数。

\_\_repr\_\_用于任何地方，除了当定义一个\_\_str\_\_的时候。

### 2.2\_\_format\_\_()方法

string.format()和内置的format()函数都使用了\_\_format\_\_()方法。它们都是为了获得给定对象的一个符合要求的字符串表示。

给\_\_format\_\_()传参的两种方式：

* someobject.\_\_format\_\_(“”)：当应用程序中出现format（someobject）或者“{0}”.format(someobject)时，会默认以这种方式调用\_\_format\_\_().在这些情况下，会传递一个空字符串，\_\_format\_\_()的返回值会以默认格式表示。
* someobject.\_\_format\_\_(specification):当应用程序中出现format(someobject,specification)或者”{0:specification}”.format(someobject)时，会默认以这种方式调用\_\_format\_\_().

“{0!r}”.format()和”{0!s}”.format()并不会调用\_\_format\_\_()方法。它们会直接调用\_\_repr\_\_()或者\_\_str\_\_().

格式规范：

[[fill]align][sign][#][0][width][,][.precision][type]

这个规范的正则表达式：

re.compile(

r”(?P<fill\_ailgn>.?[\<\>=\^])?”

“(?P<sign>[-+ ])?”

“?P<alt>#”

“(?P<padding>0)?”

“(?p<width>\d\*)”

“(?P<comma>,)?”

“(?P<precision>\.\d\*)?”

“(?P<type>[bcdeEfFgGnosxX%])?”)

### 2.3 \_\_hash\_\_()方法

内置的hash()函数默认调用了\_\_hash\_\_()方法。哈希是一种将相对复杂的值简化为小整数的计算方式。

python中有两个哈希库。其中，hashlib可以提供密码级别的哈希函数，zlib模块包含两个高效的哈希函数：adler32()和crc32().

hash函数主要用来创建set、frozenset和dict这些集合类型的键。这些集合利用不可变对象的哈希值来高效的查找集合中的对象。

>>> x = object()

>>> hash(x)

2001678

>>> id(x)

32026848L

>>> id(x)/16

2001678L

\_\_hash\_\_()方法**默认的行为**是要保证每一个对象都是可哈希的，并且哈希值是唯一的，即使这些对象包含同样的值。

#### 2.3.1 决定哈希的对象

等价性比较的3个层次：

* 哈希值相等：这意味两个对象可能相等。
* 比较结果相等：意味着两个对象的哈希值已经是相等的，这个比较的是运算符==运算符。如果结果相等，那么两个对象有可能是同一个。
* IDD相等：比较的两个对象是同一个对象。

基本哈希法（Fundamental Law of Hash，FLH）：比较相等的对象的哈希值一定相同。

需要使用\_\_eq\_\_()和\_\_hash\_\_()方法来定义相等性测试和哈希值。

* 不可变对象：不可以修改的无状态类型对象。
* 不用自定义的\_\_hash\_\_()和\_\_eq\_\_()。这意味着直接使用继承而来的行为。\_\_hash\_\_()返回一个简单的函数代表对象的ID值，然后\_\_eq\_\_()比较对象的ID值。
* 自定义\_\_hash\_\_()和\_\_eq\_\_()。这种自定义必须针对的是不可变对象。
* 可变对象：都是有状态的对象，它们允许从内部修改。
* 自定义\_\_eq\_\_()，但是设置\_\_hash\_\_为None。这些对象不可以作为dict的键和set中的项目。

#### 2.3.2 有关不可变对象和继承的默认行为

## 属性访问、特性和修饰符

### 3.1 属性的基本操作

默认情况下，创建任何类内部的属性都将支持以下4种操作：

* 创建新属性
* 为已有新属性赋值
* 获取属性的值
* 删除属性

>>> class Generic:

pass

>>> g = Generic()

>>> g.attribute = "value"

>>> g.attribute

'value'

>>> g.attribute = "new value"

>>> g.attribute

'new value'

>>> del g.attribute

>>> g.attribute

Traceback (most recent call last):

File "<pyshell#9>", line 1, in <module>

g.attribute

AttributeError: Generic instance has no attribute 'attribute'

>>>

### 3.2 创建特性

特性是一个函数，而且可以被调用，而不仅仅是用于存储的对象的引用。

可以使用@property修饰符或者使用property函数来创建特性。

特性的两个基本设计模式：

* 主动计算（Eager Calculation）：每当更新新特性时，其他相关特性值都会立即被重新计算。
* 延迟计算（Lazy calculation）：仅当访问特性时，才会触发计算过程。