Python 全栈文档

第二章 面向对象进阶

前面我们了解到数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。

在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

python是动态语言，动态编程语言 是 高级程序设计语言 的一个类别，在计算机科学领域已被广泛应用。它是一类在 运行时可以改变其结构 的语言 ： 例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。动态语言目前非常具有活力

例如，我们现在创建一个人的类，在这个类里面，定义了两个初始属性name和age

```
calss Person:
    def __init__(self, name = None, age = None):
        self.name = name
        self.age = age
```

现在我们实例化一个人，P1对象对应的就是诸葛亮这个人，我们传入p1对象的两个属性，姓名和年龄，这个p1对象就好像是自己，我们把自己的姓名和年龄的属性，挂在自己身上，但可见可不见

```
>>> p1 = Person('诸葛亮', '21')
```

接着，此时出现问题，若我不知道不认识这个人，例如我在人这个一个系统里面，茫茫人海无意间看到了有这个p1对象，有点好奇，或者需要查到他一些资料，给到他，但我不熟悉有这个人，我想看看这个人是男是女，

```
>>> p1.name
诸葛亮
>>> # 如果我这里写上p1.sex就会存在问题吧，因为没有这个人的性别信息，那我查找到，该填进来吧
>>> p1.sex = '男'
>>> p1.sex
男
```

这时候就发现问题了，我们定义的类里面没有sex这个属性啊！怎么回事呢？这就是动态语言的魅力和坑！ 这里 实际上就是 动态给实例绑定属性！

在运行的过程中给类绑定属性,看下面的例子

```
>>> P2 = Person("小丽", "25")
>>> P2.sex
Traceback (most recent call last):4
    .......
AttributeError: Person instance has no attribute 'sex'
>>>
```

我们尝试打印P2.sex，发现报错，P2没有sex这个属性！---- 给P1这个实例绑定属性对P2这个实例不起作用！ 那我们要给所有的Person的实例加上 sex属性怎么办呢？ 答案就是直接给Person绑定属性！

```
>>>> Person.sex = None #给类Person添加一个属性
>>> P2 = Person("小丽", "25")
>>> print(P1.sex) #如果P1这个实例对象中没有sex属性的话，那么就会访问它的类属性
None #可以看到没有出现异常
>>>
```

我们直接给Person绑定sex这个属性，重新实例化P2后，P2就有sex这个属性了！ 那么function呢？怎么绑定？

```
class Person(object):
    def __init__(self, name = None, age = None):
        self.name = name
        self.age = age
    def eat(self):
        print("eat food")


# 单独定义一个run方法
def run(self, speed):
    print("%s在移动，速度是 %d km/h"%(self.name, speed))
```

```
>>> P = Person("曹操", 24)
>>> p.eat()
eat food
>>> p.run()
Traceback (most recent call last):
......
AttributeError: Person instance has no attribute 'run'

>>> import types
>>> P.run = types.MethodType(run, P)
>>> P.run(180)
曹操在移动,速度是 180 km/h
```

既然给类添加方法，是使用 类名.方法名 = xxxx ，

那么给对象添加一个方法也是类似的 对象.方法名 = xxxx

看完整代码，对类方法，方法，增加绑定

```
# -*- coding: UTF-8 -*-
# 文件名: class_a.py


import types


class Person:
    num = 0
```

```
    def __init__(self, name=None, age=None):
        self.name = name
        self.age = age

    def eat(self):
        print("eat food")


# 单独定义的一个run方法
def run(self, v):
    print(f"{self.name}在移动，速度是 {v} km/h")


# 定义一个类方法
@classmethod
def classClass(cls):
    cls.num = 100


# 定义一个静态方法
@staticmethod
def staticClass():
    print("-------static method-------")


# 创建实例对象
p1 = Person('刘备', 21)

# 调用class中的方法，类中eat的方法
p1.eat()

# 给这个对象添加实例方法,types.MethodType函数将run方法绑定到p1这个对象上
p1.run = types.MethodType(run, p1)
# 调用对象的方法
p1.run(120)

# 给Person类绑定类方法,这里的是方法名
Person.classClass = classClass
# 调用类方法
print(Person.num)
Person.classClass()
print(Person.num)

# 给Person绑定静态方法
Person.staticClass = staticClass
# 调用静态方法
Person.staticClass()
```

那既然有增加，就有删除

删除对象与属性的方法

1. del 对象.属性名

2. delattr(对象, "属性名")

我们知道，正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：然后尝试给实例绑定一个属性，还可以绑定一个方法，但是一个实例方法对另一个实例不起作用，那就得给类整个类绑定一个方法或属性，这样所有的实例都可以调用

需要注意的是我们的动态语言在运行后还能修改的，但是静态语言是不可以的，这就会造成不严谨

1、'__slots__'

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。

动态语言：可以在运行的过程中，修改代码

静态语言：编译时已经确定好代码，运行过程中不能修改

但是，如果我们想要限制class的属性怎么办？比如，上例只允许对人类的实例添加`name`和`age`属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的`__slots__`变量，来限制该class能添加的属性：

先定义一个学生的类class：

```
# python交互式环境

class Student(object):
    # 用tuple定义允许绑定的属性名称
    __slots__ = ('name', 'age')
```

测试

```
>>> s1 = Student()
>>> s1.name = "刘备"
>>> s1.age = 20
>>> s1.score = 100
Traceback (most recent call last):
......
AttributeError: Person instance has no attribute 'score'
>>>
```

首先，我们创建了一个学生对象s1，给s1一个name和age属性，给定成功，

但是，当我们给实例s1一个score的属性时，提示"Person instance has no attribute 'score'"报错

由此可以看出来，`__slots__`限制了对象的属性随意添加。

需要注意一点的是：使用`__slots__`要注意，`__slots__`定义的属性仅对当前类实例起作用，对继承的子类是不起作用的

```
# python交互式环境

class subStudent(Student):
    pass
```

测试

```
>>> sub1 = subStudent()
>>> sub1.score = 100
>>> sub1.score
100
```

2、@property

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，导致可以把成绩随便改

```
class Student:
    pass

s = Student()
s.score = 9999
```

我们可以看到类中的属性被随意修改，很明显不能这样操作，为了限制score的范围，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 Student:

    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

s = Student()
s.set_score(60) # ok!
s.get_score()
60
s.set_score(9999)

# Traceback (most recent call last):
# ...
# ValueError: score must between 0 ~ 100!
```

我们发现，当我们设置限定score的范围的时候，超过对应的范围set_score设置值时raise方法会自动给我们抛出异常，但是上面的方法太过于复杂，所以引入我们的一个装饰器，装饰器是可以给函数动态加上功能。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):

    @property
    def score(self):
        return self._score

    @score.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
```

`@property`的实现比较复杂，先观察使用。把一个getter方法变成属性，只需要加上`@property`就可以了，此时，`@property`本身又创建了另一个装饰器`@score.setter`，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作

3、多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能:

举例说明一个:

例如`Animal`类即动物类，层次的设计，假设我们要实现以下4种动物:

```
Dog  狗
Cat  猫
Parrot 鹦鹉
Cuckoo 杜鹃
```

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次

```
                                    --------- > Dog  狗
                  -> (Mammal 哺乳类) ----
                -                   ----------> Bat - 蝙蝠
              -
(Animal 动物类)
              -
                -                   ----------> Parrot 鹦鹉
                  -> (Bird 鸟类)    -----
                                    ----------> Ostrich 鸵鸟
```

但是如果按照运动方式来分，例如能跑，能飞

```
                                        --------- > Dog  狗
                    -> (Runnable 跑) ----
                  -                     ----------> Ostrich 鸵鸟
                -
(Animal 动物类)
                -
                  -                     ----------> Parrot 鹦鹉
                    -> (Flyable 飞)  ----
                                        ----------> Bat - 蝙蝠
```

如果要把上面的两种分类都包含进来，我们就得设计更多的层次

```
哺乳类：能跑的哺乳类，能飞的哺乳类；

鸟类：能跑的鸟类，能飞的鸟类。
```

那这个结构层次就复杂起来了，如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的；

所以在类中使用多重继承，按主要的类层次区分，先按照哺乳类和鸟类设计

```
>>> class Animal(object):
        pass

# 大类:
>>> class Mammal(Animal):
        pass

>>> class Bird(Animal):
        pass
```

接着就是各种动物类

```
# 各种动物:
>>> class Dog(Mammal):
        pass

>>> class Bat(Mammal):
        pass

>>> class Parrot(Bird):
        pass

>>> class Ostrich(Bird):
        pass
```

此时根据功能来进行增加类

```
>>> class Runnable(object):
        def run(self):
```

```
            print('Running...')

>>> class Flyable(object):
        def fly(self):
            print('Flying...')

# 例如这里给Dog增加能跑的类
>>> class Dog(Mammal, Runnable):
        pass
#  这里给Bat增加能飞的类
>>> class Bat(Mammal, Flyable):
        pass
```

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

MixIn

我们在设计类的继承关系时，通常主线都是单一继承下来的，例如，`Ostrich`继承自`Bird`。但是，如果需要加入其它的功能，通过多重继承就可以实现，比如，让`Ostrich`除了继承自`Bird`外，再同时继承`Runnable`。这种叫做MixIn

MixIn的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类

4、定制类

看到类似`__slots__`这种形如`__xxx__`的变量或者函数名就要注意，这些在Python中是有特殊用途的。`__slots__`我们已经知道怎么用了，`__len__()`方法我们也知道是为了能让class作用于`len()`函数。这些在Python有另外的一些名称叫魔术方法

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

__str__

我们先定义一个`Student`类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print(Student('Michael'))
<__main__.Student object at 0x109afb190>
```

打印出一堆`<__main__.Student object at 0x109afb190>`，不好看。

怎么才能打印得好看呢？只需要定义好`__str__()`方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用`print`，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是`__str__()`，而是`__repr__()`，两者的区别是`__str__()`返回用户看到的字符串，而`__repr__()`返回程序开发者看到的字符串，也就是说，`__repr__()`是为调试服务的。

解决办法是再定义一个`__repr__()`。但是通常`__str__()`和`__repr__()`代码都是一样的，所以，有个偷懒的写法：

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

__iter__

如果一个类想被用于`for ... in`循环，类似list或tuple那样，就必须实现一个`__iter__()`方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的`__next__()`方法拿到循环的下一个值，直到遇到`StopIteration`错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a，b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值
```

现在，试试把Fib实例作用于for循环：

```
>>> for n in Fib():
...     print(n)
...
1
1
2
3
5
...
46368
75025
```

__getitem__

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

```
>>> Fib()[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Fib' object does not support indexing
```

要表现得像list那样按照下标取出元素，需要实现`__getitem__()`方法：

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
>>> f[0]
1
>>> f[1]
1
>>> f[2]
2
>>> f[3]
3
>>> f[10]
89
>>> f[100]
573147844013817084101
```

但是list有个神奇的切片方法：

```
>>> list(range(100))[5:10]
[5, 6, 7, 8, 9]
```

对于Fib却报错。原因是`__getitem__()`传入的参数可能是一个int，也可能是一个切片对象`slice`，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int): # n是索引
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice): # n是切片
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

现在试试Fib的切片：

```
>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

但是没有对step参数作处理：

```
>>> f[:10:2]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

也没有对负数作处理，所以，要正确实现一个`__getitem__()`还是有很多工作要做的。

此外，如果把对象看成`dict`，`__getitem__()`的参数也可能是一个可以作key的object，例如`str`。

与之对应的是`__setitem__()`方法，把对象视作list或dict来对集合赋值。最后，还有一个`__delitem__()`方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

__getattr__

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义`Student`类：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'
```

调用`name`属性，没问题，但是，调用不存在的`score`属性，就有问题了：

```
>>> s = Student()
>>> print(s.name)
Michael
>>> print(s.score)
Traceback (most recent call last):
  ...
AttributeError: 'Student' object has no attribute 'score'
```

错误信息很清楚地告诉我们，没有找到`score`这个attribute。

要避免这个错误，除了可以加上一个`score`属性外，Python还有另一个机制，那就是写一个`__getattr__()`方法，动态返回一个属性。修改如下：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如`score`，Python解释器会试图调用`__getattr__(self, 'score')`来尝试获得属性，这样，我们就有机会返回`score`的值：

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
```

返回函数也是完全可以的：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
```

只是调用方式要变为：

```
>>> s.age()
25
```

注意，只有在没有找到属性的情况下，才调用`__getattr__`，已有的属性，比如`name`，不会在`__getattr__`中查找。

此外，注意到任意调用如`s.abc`都会返回`None`，这是因为我们定义的`__getattr__`默认返回就是`None`。要让class只响应特定的几个属性，我们就要按照约定，抛出`AttributeError`的错误：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\'Student\' object has no attribute \'%s\'' % attr)
```

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

__call__

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用`instance.method()`来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个`__call__()`方法，就可以直接对实例进行调用。请看示例：

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

    def __call__(self):
        print('My name is %s.' % self.name)
```

调用方式如下：

```
>>> s = Student('Michael')
>>> s() # self参数不要传入
My name is Michael.
```

`__call__()`还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个`Callable`对象，比如函数和我们上面定义的带有`__call__()`的类实例：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('str')
False
```

通过`callable()`函数，我们就可以判断一个对象是否是“可调用”对象。

5、枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

```
JAN = 1
FEB = 2
... ...
NOV = 11
DEC = 12
```

但是这样的定义的类型是`int`，并且仍然是变量，并且在运算中，无法时时创建对应的值，当然这种指代是以更好的方式去使用变量数值。

这里存在更好的方法是为这样的枚举类型定义一个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。

优化如下

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec'))
```

这样我们就获得了`Month`类型的枚举类，可以直接使用`Month.Jan`来引用一个常量，或者枚举它的所有成员

通过for循环进行取值，

Month.**members**是Month的内置变量，可以打印成员标签，

如果在这个方法调用上.items()函数所有成员都打印出来，数据类型为元组

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)

Jan => Month.Jan , 1
Feb => Month.Feb , 2
... ...
Nov => Month.Nov , 11
Dec => Month.Dec , 12
```

6、元类

Python属于动态类型的语言，而动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时创建的，而是运行时动态创建的，比方说我们要定义一个`Hello`的class，就写一个`hello.py`模块：

```
class Hello(object):
    def hello(self, name='world'):
        print(f'Hello, {name}')
```

当Python解释器载入`hello`模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个`Hello`的class对象，测试如下，（注意，是引入一个hello.py脚本）

```
# 这里根据模块路径引入
from demo_biji.text1.hello import Hello

h = Hello()
# 调用类方法
h.hello()
Hello, world

# 打印类型
print(type(Hello))
# 打印结果 <class 'type'>

# 文本类型
print(type(h))
# 打印结果 <class 'demo_biji.text1.hello.Hello'>
```

这里是用来`type()`函数，可以查看一个类型或变量，的类型，`Hello`是一个class，它的类型就是`type`，而`h`是一个实例，它的类型就是class `Hello`。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用`type()`函数

`type()`函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过`type()`函数创建出`Hello`类，而无需通过`class Hello(object)...`的定义，此时type的第二种用法，我们只要type传入object是可以返回该对象类型的，但是当我们的type存在三位参数存在时，

```
type(name, bases, dict)

name -- 类的名称。
bases -- 基类的元组。
dict -- 字典，类内定义的命名空间变量。

# 返回新的类型对象。
```

```
# 定义一个函数
>>> def fn(self, name='world'):
    print(f'Hello,{name}')


>>> Hello = type('Hello', (object,), dict(hello=fn))
>>> h = Hello()
>>> h.hello()
Hello, world
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

我们通过`type()`函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用`type()`函数创建出class

在正常情况下，我们都用`class Xxx...`来定义类，但是，`type()`函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂

除了使用`type()`动态创建类以外，要控制类的创建行为，还可以使用metaclass，也就是元类

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类

元类就是用来创建这些类（对象）的，元类就是类的类

我们先看一个简单的例子，这个metaclass可以给自定义的类增加一个`add`方法

定义`ListMetaclass`，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclas

```
# -*- coding: UTF-8 -*-
# 文件名  : mxdxjj_3_2_2.py

# 定义了一个列表元类
class ListMetaclass(type):

    #  __new__ 是在一个对象实例化的时候所调用的第一个方法
    def __new__(cls, name, bases, attrs):
        # 通过匿名函数，将值键入，通过方法使用append方法建立attrs字典中，add对应的列表值
        attrs['add'] = lambda self, value: self.append(value)

        # 并返回数据类型
        return type.__new__(cls, name, bases, attrs)


class MyList(list, metaclass=ListMetaclass):
    pass


L = MyList()
```

```
L.add(1)
print(L)

# 打印结果：[1]
```

我们在定义MyList类时，使用关键字参数将指示使用ListMetaclass来定制类

当我们传入关键字参数`metaclass`时，魔术就生效了，它指示Python解释器在创建`MyList`时，要通过`ListMetaclass.__new__()`来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义

> `__new__()`方法接收到的参数依次是：
>
> 1. 当前准备创建的类的对象；
> 2. 类的名字；
> 3. 类继承的父类集合；
> 4. 类的方法集合。

通过测试我们发现我们定义的MyList是有这个方法的，而普通的`list`没有`add()`方法

动态修改有什么意义？直接在`MyList`定义中写上`add()`方法不是更简单吗？正常情况下，确实应该直接写。

我们再拿上方的type来讲，

函数type实际上是一个元类。type就是Python在背后用来创建所有类的元类。现在你想知道那为什么type会全部采用小写形式而不是Type呢？有可能也是为了和str保持一致性，str是用来创建字符串对象的类，而int是用来创建整数对象的类。type就是创建类对象的类。你可以通过检查**class**属性来看到这一点。Python中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。它们全部都是对象，而且它们都是从一个类创建而来，这个类就是type

```
>>> score = 35
>>> score.__class__
<class 'int'>

>>> name = 'bob'
>>> name.__class__
<class 'str'>

>>> def foo():
        pass
>>>foo.__class__
<class 'function'>

>>> class Bar(object):
        pass
>>> b = Bar()
>>> b.__class__
<class '__main__.Bar'>
```

现在，对于任何一个__class__的__class__属性又是什么呢？继续输入

```
>>> a.__class__.__class__
<type 'type'>
>>> age.__class__.__class__
<type 'type'>
>>> foo.__class__.__class__
<type 'type'>
>>> b.__class__.__class__
<type 'type'>
```

发现，**元类就是创建类这种对象的东西**。type就是Python的内建元类，当然了，你也可以创建自己的元类

元类的主要目的就是为了当创建类时能够自动地改变类。通常，你会为API做这样的事情，你希望可以创建符合当前上下文的类。

假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定**metaclass**。采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。

幸运的是，**metaclass**实际上可以被任意调用，它并不需要是一个正式的类。

```
# -*- coding: UTF-8 -*-
# 文件名  : text1.py


def per_attr(future_class_name, future_class_parents, future_class_attr):

    # 遍历属性字典，把不是__开头的属性名字变为大写
    new_attr = {}
    for name, value in future_class_attr.items():
        if not name.startswith("__"):
            new_attr[name.upper()] = value

    # 调用type来创建一个类
    return type(future_class_name, future_class_parents, new_attr)


class Foo(object, metaclass=per_attr):
    bar = 'bip'


print(hasattr(Foo, 'bar'))
print(hasattr(Foo, 'BAR'))

f = Foo()
print(f.BAR)
```

现在让我们再做一次，这一次用一个真正的class来当做元类。

```
# -*- coding: UTF-8 -*-


class UpperAttrMetaClass(type):
```

```
    # __new__ 是在__init__之前被调用的特殊方法
    # __new__是用来创建对象并返回之的方法
    # 而__init__只是用来将传入的参数初始化给对象
    # 你很少用到__new__，除非你希望能够控制对象的创建
    # 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写__new__
    # 如果你希望的话，你也可以在__init__中做些事情
    # 还有一些高级的用法会涉及到改写__call__特殊方法，但是我们这里不用
    def __new__(cls, future_class_name, future_class_parents, future_class_attr):
        # 遍历属性字典，把不是__开头的属性名字变为大写
        newAttr = {}
        for name, value in future_class_attr.items():
            if not name.startswith("__"):
                newAttr[name.upper()] = value

        # 方法1：通过'type'来做类对象的创建
        # return type(future_class_name, future_class_parents, newAttr)

        # 方法2：复用type.__new__方法
        # 这就是基本的OOP编程，没什么魔法
        # return type.__new__(cls, future_class_name, future_class_parents, newAttr)

        # 方法3：使用super方法
        return super(UpperAttrMetaClass, cls).__new__(cls, future_class_name,
future_class_parents, newAttr)


class Foo(object, metaclass=UpperAttrMetaClass):
    bar = 'bip'


print(hasattr(Foo, 'bar'))
# 输出: False
print(hasattr(Foo, 'BAR'))
# 输出:True

f = Foo()
print(f.BAR)
# 输出:'bip'
```

就是这样，除此之外，关于元类真的没有别的可说的了。但就元类本身而言，它们其实是很简单的：

> 拦截类的创建
>
> 修改类
>
> 返回修改之后的类

现在回到我们的大主题上来，究竟是为什么你会去使用这样一种容易出错且晦涩的特性？好吧，一般来说，你根本就用不上它：

“元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。” —— Python界的领袖 Tim Peters