**第四部分 面向对象**

##### 类和实例

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

class Student(object):

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

self.name = name

s = Student('Bob')

s.score = 90

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

class Student(object):

name = 'Student'

当定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

class Student(object):

... name = 'Student'

...

s = Student() # 创建实例s

print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性

print(Student.name) # 打印类的name属性

s.name = 'Michael' # 给实例绑定name属性

print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性

print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问

del s.name # 如果删除实例的name属性

print(s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当删除实例属性后，再使用相同的名称，访问到的将是类属性。

##### 访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、score属性：

bart = Student('Bart Simpson', 59)

bart.score

bart.score = 99

bart.score

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，所以，把Student类改一改：

class Student(object):

def \_\_init\_\_(self, name, score):

self.\_\_name = name

self.\_\_score = score

def print\_score(self):

print('%s: %s' % (self.\_\_name, self.\_\_score))

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.\_\_name和实例变量.\_\_score了：

bart = Student('Bart Simpson', 59)

bart.\_\_name

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加get\_name和get\_score这样的方法：

class Student(object):

...

def get\_name(self):

return self.\_\_name

def get\_score(self):

return self.\_\_score

如果又要允许外部代码修改score怎么办？可以再给Student类增加set\_score方法：

class Student(object):

...

def set\_score(self, score):

self.\_\_score = score

也许会问，原先那种直接通过bart.score = 99也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

class Student(object):

...

def set\_score(self, score):

if 0 <= score <= 100:

self.\_\_score = score

else:

raise ValueError('bad score')

需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

有些时候，会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

bart.\_Student\_\_name

但是强烈建议不要这么做，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

最后注意下面的这种错误写法：

bart = Student('Bart Simpson', 59)

bart.get\_name()

bart.\_\_name = 'New Name' # 设置\_\_name变量！

bart.\_\_name

表面上看，外部代码“成功”地设置了\_\_name变量，但实际上这个\_\_name变量和class内部的\_\_name变量不是一个变量！内部的\_\_name变量已经被Python解释器自动改成了\_Student\_\_name，而外部代码给bart新增了一个\_\_name变量。不信试试：

bart.get\_name() # get\_name()内部返回self.\_\_name

##### 继承和多态

在OOP程序设计中，当定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

class Animal(object):

def run(self):

print('Animal is running...')

当需要编写Dog和Cat类时，就可以直接从Animal类继承：

class Dog(Animal):

pass

class Cat(Animal):

pass

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和Dog类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于Animial实现了run()方法，因此，Dog和Cat作为它的子类，什么事也没干，就自动拥有了run()方法：

dog = Dog()

dog.run()

cat = Cat()

cat.run()

当然，也可以对子类增加一些方法，比如Dog类：

class Dog(Animal):

def run(self):

print('Dog is running...')

def eat(self):

print('Eating meat...')

继承的第二个好处需要对代码做一点改进。看到了，无论是Dog还是Cat，它们run()的时候，显示的都是Animal is running...，符合逻辑的做法是分别显示Dog is running...和Cat is running...，因此，对Dog和Cat类改进如下：

class Dog(Animal):

def run(self):

print('Dog is running...')

class Cat(Animal):

def run(self):

print('Cat is running...')

当子类和父类都存在相同的run()方法时，说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。这样，就获得了继承的另一个好处：多态。

要理解什么是多态，首先要对数据类型再作一点说明。当定义一个class的时候，实际上就定义了一种数据类型。定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

a = list() # a是list类型

b = Animal() # b是Animal类型

c = Dog() # c是Dog类型

判断一个变量是否是某个类型可以用isinstance()判断：

isinstance(a, list)

isinstance(b, Animal)

isinstance(c, Dog)

看来a、b、c确实对应着list、Animal、Dog这3种类型。

但是测试以下写法：

isinstance(c, Animal)

看来c不仅仅是Dog，c还是Animal！

不过仔细想想，这是有道理的，因为Dog是从Animal继承下来的，当创建了一个Dog的实例c时，认为c的数据类型是Dog没错，但c同时也是Animal也没错，Dog本来就是Animal的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

b = Animal()

isinstance(b, Dog)

Dog可以看成Animal，但Animal不可以看成Dog。

要理解多态的好处，还需要再编写一个函数，这个函数接受一个Animal类型的变量：

def run\_twice(animal):

animal.run()

animal.run()

当传入Animal的实例时，run\_twice()就打印出：

run\_twice(Animal())

当传入Dog的实例时，run\_twice()就打印出：

run\_twice(Dog())

当传入Cat的实例时，run\_twice()就打印出：

run\_twice(Cat())

看上去没啥意思，但是仔细想想，现在，如果再定义一个Tortoise类型，也从Animal派生：

class Tortoise(Animal):

def run(self):

print('Tortoise is running slowly...')

当调用run\_twice()时，传入Tortoise的实例：

run\_twice(Tortoise())

会发现，新增一个Animal的子类，不必对run\_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

多态的好处就是，当需要传入Dog、Cat、Tortoise……时，只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Animal子类；

对修改封闭：不需要修改依赖Animal类型的run\_twice()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。

##### 获取对象信息

当拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

###### 使用type()

首先，来判断对象类型，使用type()函数：

基本类型都可以用type()判断：

type(123)

type('str')

type(None)

如果一个变量指向函数或者类，也可以用type()判断：

type(abs)

type(a)

但是type()函数返回的是什么类型呢？它返回对应的Class类型。如果要在if语句中判断，就需要比较两个变量的type类型是否相同：

type(123)==type(456)

type(123)==int

type('abc')==type('123')

type('abc')==str

type('abc')==type(123)

判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

import types

def fn():

... pass

...

type(fn)==types.FunctionType

type(abs)==types.BuiltinFunctionType

type(lambda x: x)==types.LambdaType

type((x for x in range(10)))==types.GeneratorType

###### 使用isinstance()

对于class的继承关系来说，使用type()就很不方便。要判断class的类型，可以使用isinstance()函数。

回顾上次的例子，如果继承关系是：

object -> Animal -> Dog -> Husky

那么，isinstance()就可以告诉，一个对象是否是某种类型。先创建3种类型的对象：

a = Animal()

d = Dog()

h = Husky()

然后，判断：

isinstance(h, Husky)

没有问题，因为h变量指向的就是Husky对象。

再判断：

isinstance(h, Dog)

h虽然自身是Husky类型，但由于Husky是从Dog继承下来的，所以，h也还是Dog类型。换句话说，isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，可以确信，h还是Animal类型：

isinstance(h, Animal)

同理，实际类型是Dog的d也是Animal类型：

isinstance(d, Dog) and isinstance(d, Animal)

但是，d不是Husky类型：

isinstance(d, Husky)

能用type()判断的基本类型也可以用isinstance()判断：

isinstance('a', str)

isinstance(123, int)

isinstance(b'a', bytes)

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

isinstance([1, 2, 3], (list, tuple))

isinstance((1, 2, 3), (list, tuple))

总是优先使用isinstance()判断类型，可以将指定类型及其子类“一网打尽”。

###### 使用dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

dir('ABC')

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：

len('ABC')

'ABC'.\_\_len\_\_()

自己写的类，如果也想用len(myObj)的话，就自己写一个\_\_len\_\_()方法：

class MyDog(object):

... def \_\_len\_\_(self):

... return 100

...

dog = MyDog()

len(dog)

剩下的都是普通属性或方法，比如lower()返回小写的字符串：

'ABC'.lower()

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，可以直接操作一个对象的状态：

class MyObject(object):

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

... self.x = 9

... def power(self):

... return self.x \* self.x

...

obj = MyObject()

紧接着，可以测试该对象的属性：

hasattr(obj, 'x') # 有属性'x'吗？

obj.x

hasattr(obj, 'y') # 有属性'y'吗？

setattr(obj, 'y', 19) # 设置一个属性'y'

hasattr(obj, 'y') # 有属性'y'吗？

getattr(obj, 'y') # 获取属性'y'

obj.y # 获取属性'y'

如果试图获取不存在的属性，会抛出AttributeError的错误：

getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'MyObject' object has no attribute 'z'

可以传入一个default参数，如果属性不存在，就返回默认值：

getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404

也可以获得对象的方法：

hasattr(obj, 'power') # 有属性'power'吗？

getattr(obj, 'power') # 获取属性'power'

fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn

fn # fn指向obj.power

fn() # 调用fn()与调用obj.power()是一样的

##### 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

class Student(object):

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

self.name = name

s = Student('Bob')

s.score = 90

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

class Student(object):

name = 'Student'

当定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

class Student(object):

... name = 'Student'

...

s = Student() # 创建实例s

print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性

print(Student.name) # 打印类的name属性

s.name = 'Michael' # 给实例绑定name属性

print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性

print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问

del s.name # 如果删除实例的name属性

print(s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当删除实例属性后，再使用相同的名称，访问到的将是类属性。

##### \_\_slots\_\_

###### 属性设置

正常情况下，当定义了一个class，创建了一个class的实例后，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：

class Student(object):

pass

然后，尝试给实例绑定一个属性：

s = Student()

s.name = 'Michael' # 动态给实例绑定一个属性

print(s.name)

还可以尝试给实例绑定一个方法：

def set\_age(self, age): # 定义一个函数作为实例方法

... self.age = age

...

from types import MethodType

s.set\_age = MethodType(set\_age, s) # 给实例绑定一个方法

s.set\_age(25) # 调用实例方法

s.age # 测试结果

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

s2 = Student() # 创建新的实例

s2.set\_age(25) # 尝试调用方法

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute 'set\_age'

为了给所有实例都绑定方法，可以给class绑定方法：

def set\_score(self, score):

... self.score = score

...

Student.set\_score = set\_score

给class绑定方法后，所有实例均可调用：

s.set\_score(100)

s.score

s2.set\_score(99)

s2.score

通常情况下，上面的set\_score方法可以直接定义在class中，但动态绑定允许在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

###### 使用\_\_slots\_\_

但是，如果想要限制实例的属性怎么办？比如，只允许对Student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实例能添加的属性：

class Student(object):

\_\_slots\_\_ = ('name', 'age') # 用tuple定义允许绑定的属性名称

然后，试试：

s = Student() # 创建新的实例

s.name = 'Michael' # 绑定属性'name'

s.age = 25 # 绑定属性'age'

s.score = 99 # 绑定属性'score'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute 'score'

由于'score'没有被放到\_\_slots\_\_中，所以不能绑定score属性，试图绑定score将得到AttributeError的错误。

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：

class GraduateStudent(Student):

... pass

...

g = GraduateStudent()

g.score = 9999

除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

##### @property

在绑定属性时，如果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：

s = Student()

s.score = 9999

这显然不合逻辑。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(object):

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

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

s = Student()

s.set\_score(60) # ok!

s.get\_score()

s.set\_score(9999)

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

但是，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？对于追求完美的Python程序员来说，这是必须要做到的！

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):

@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，先考察如何使用。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，就拥有一个可控的属性操作：

s = Student()

s.score = 60 # OK，实际转化为s.set\_score(60)

s.score # OK，实际转化为s.get\_score()

s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

注意到这个神奇的@property，在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

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。

##### 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。

###### 多重继承

class Animal(object):

pass

# 大类:

class Mammal(Animal):

pass

class Bird(Animal):

pass

# 各种动物:

class Dog(Mammal):

pass

class Bat(Mammal):

pass

class Parrot(Bird):

pass

class Ostrich(Bird):

pass

现在，要给动物再加上Runnable和Flyable的功能，只需要先定义好Runnable和Flyable的类：

class Runnable(object):

def run(self):

print('Running...')

class Flyable(object):

def fly(self):

print('Flying...')

对于需要Runnable功能的动物，就多继承一个Runnable，例如Dog：

class Dog(Mammal, Runnable):

pass

对于需要Flyable功能的动物，就多继承一个Flyable，例如Bat：

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

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“混入”额外的功能，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。这种设计通常称之为MixIn。

为了更好地看出继承关系，把Runnable和Flyable改为RunnableMixIn和FlyableMixIn。类似的，还可以定义出肉食动物CarnivorousMixIn和植食动物HerbivoresMixIn，让某个动物同时拥有好几个MixIn：

class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):

pass

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，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，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。

##### 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\_\_已经知道怎么用了，\_\_len\_\_()方法也知道是为了能让class作用于len()函数。

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

###### \_\_str\_\_

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'))

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。

###### \_\_repr\_\_

\_\_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。为使\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以这样写：

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。

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]

但是list支持切片：

list(range(100))[5:10]

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]

f[:10]

但是没有对step参数作处理：

f[:10:2]

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。

\_\_getitem\_\_()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。

###### \_\_getattr\_\_

正常情况下，当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)

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，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，就有机会返回score的值：

s = Student()

s.name

s.score

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()

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，这是因为定义的\_\_getattr\_\_默认返回就是None。要让class只响应特定的几个属性，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。

###### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当调用实例方法时，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

\_\_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，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，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，比如函数和上面定义的带有\_\_call\_\_()的类实例：

callable(Student())

callable(max)

callable([1, 2, 3])

callable(None)

callable('str')

通过callable()函数，就可以判断一个对象是否是“可调用”对象。

##### 使用枚举类

当需要定义常量时，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

MAR = 3

...

NOV = 11

DEC = 12

好处是简单，缺点是类型是int，并且仍然是变量。

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：

from enum import Enum

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

这样就获得了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：

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。

如果需要更精确地控制枚举类型，可以从Enum派生出自定义类：

from enum import Enum, unique

@unique

class Weekday(Enum):

Sun = 0 # Sun的value被设定为0

Mon = 1

Tue = 2

Wed = 3

Thu = 4

Fri = 5

Sat = 6

@unique装饰器可以帮助检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

day1 = Weekday.Mon

print(day1)

Weekday.Mon

print(Weekday.Tue)

print(Weekday['Tue'])

print(Weekday.Tue.value)

print(day1 == Weekday.Mon)

print(day1 == Weekday.Tue)

print(Weekday(1))

print(day1 == Weekday(1))

Weekday(7)

for name, member in Weekday.\_\_members\_\_.items():

... print(name, '=>', member)

可见，既可以用成员名称引用枚举常量，又可以直接根据value的值获得枚举常量。

##### 动态创建类

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说要定义一个Hello的class，就写一个hello.py模块：

class Hello(object):

def hello(self, name='world'):

print('Hello, %s.' % 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：

from hello import Hello

h = Hello()

h.hello()

print(type(Hello))

print(type(h))

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。

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，可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义：

**def fn(self, name='world'): # 先定义函数**

**... print('Hello, %s.' % name)**

...

Hello = type(**'Hello', (object,), dict(hello=fn)**) # 创建Hello class

h = Hello()

h.hello()

print(type(Hello))

print(type(h))

要创建一个class对象，type()函数依次传入3个参数：

* class的名称
* 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
* class的方法名称与函数绑定，这里把函数fn绑定到方法名hello上

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。

正常情况下，都用class Xxx...来定义类，但是，type()函数也允许动态创建出类来，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。