特殊属性

属性	含义
`__name__`	类、函数、方法等的名字
`__module__`	类定义所在的模块名
`__class__`	对象或类所属的类
`__bases__`	类的基类的元组，顺序为它们在基类列表中出现的顺序
`__doc__`	类、函数的文档字符串，如果没有定义则为None
`__mro__`	类的mro，class.mro()返回的结果的保存在 `__mro__` 中
`__dict__`	类或实例的属性，可写的字典

查看属性

方法	意义
`__dir__`	返回类或者对象的所有成员名称列表。dir()函数就是调用 `__dir__` ()。如果提供 `__dir__` ()，则返回属性的列表，否则会尽量从 `__dict__` 属性中收集信息

如果dir([obj])参数obj包含方法 `__dir__()` ，该方法将被调用。如果参数obj不包含 `__dir__()` ，该方法将最大限度地收集参数信息。
dir()对于不同类型的对象具有不同的行为：
如果对象是模块对象，返回的列表包含模块的属性名。
如果对象是类型或者类对象，返回的列表包含类的属性名，及它的基类的属性名。
否则，返回列表包含对象的属性名，它的类的属性名和类的基类的属性名。

```
# animal.py
class Animal:
    x = 123
    def __init__(self, name):
        self._name = name
        self.__age = 10
```

```
        self.weight = 20
print('animal Module\'s names  = {}'.format(dir())) # 模块的属性

# cat.py
import animal
from animal import Animal

class Cat(Animal):
    x = 'cat'
    y = 'abcd'

class Dog(Animal):
    def __dir__(self):
        return ['dog'] # 必须返回可迭代对象


print('---------')
print('Current Module\'s names = {}'.format(dir())) # 模块名词空间内的属性
print('animal Module\'s names  = {}'.format(dir(animal))) # 指定模块名词空间内的属性
print("object's __dict__      = {}".format(sorted(object.__dict__.keys()))) # object的字典
print("Animal's dir() = {}".format(dir(Animal))) # 类Animal的dir()
print("Cat's dir()    = {}".format(dir(Cat))) # 类Cat的dir()
print('~~~~~~~~~')
tom = Cat('tome')
print(sorted(dir(tom))) # 实例tom的属性、Cat类及所有祖先类的类属性
print(sorted(tom.__dir__())) # 同上
# dir()的等价 近似如下，__dict__字典中几乎包括了所有属性
print(sorted(set(tom.__dict__.keys()) | set(Cat.__dict__.keys()) | set(object.__dict__.keys())))

print("Dog's dir = {}".format(dir(Dog)))
dog = Dog('snoppy')
print(dir(dog))
print(dog.__dict__)

# 执行结果
animal Module's names  = ['Animal', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']
```

```
---------
Current Module's names = ['Animal', 'Cat', 'Dog', '__builtins__', '__cached__', '__
doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'animal']
animal Module's names  = ['Animal', '__builtins__', '__cached__', '__doc__', '__fil
e__', '__loader__', '__name__', '__package__', '__spec__']
object's __dict__       = ['__class__', '__delattr__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '
__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
Animal's dir() = ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__'
, '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex
__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__wea
kref__', 'x']
Cat's dir()    = ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__'
, '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex
__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__wea
kref__', 'x', 'y']
~~~~~~~~~
['_Animal__age', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__e
q__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weak
ref__', '_name', 'weight', 'x', 'y']
['_Animal__age', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__e
q__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weak
ref__', '_name', 'weight', 'x', 'y']
['_Animal__age', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__for
mat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '
__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_name', 'weight',
'x', 'y']

Dog's dir = ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__'
, '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__
le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref_
```

```
_', 'x']
['dog']
{'_Animal__age': 10, 'weight': 20, '_name': 'snoppy'}
```

魔术方法 ***

- 分类：
  - 创建、初始化与销毁
    - `__init__` 与 `__del__`
  - hash
  - bool
  - 可视化
  - 运算符重载
  - 容器和大小
  - 可调用对象
  - 上下文管理
  - 反射
  - 描述器
  - 其他杂项

hash

方法	意义
`__hash__`	内建函数 `hash()` 调用的返回值，返回一个整数。如果定义这个方法该类的实例就可hash。

```
class A:
    def __init__(self, name, age=18):
        self.name = name

    def __hash__(self):
        return 1

    def __repr__(self):
        return self.name
```

```
print(hash(A('tom')))
print((A('tom'), A('tom')))
print([A('tom'), A('tom')])
print('~~~~~~~~~~~~~~~~~~~~~~')
s = {A('tom'), A('tom')} # set
print(s) # 去重了吗
print({tuple('t'), tuple('t')})
print({('tom',), ('tom',)})
print({'tom', 'tom'})
```

上例中set为什么不能剔除相同的key？

```
class A:
    def __init__(self, name, age=18):
        self.name = name

    def __hash__(self):
        return 1

    def __eq__(self, other): # 这个函数作用?
        return self.name == other.name

    def __repr__(self):
        return self.name


print(hash(A('tom')))
print((A('tom'), A('tom')))
print([A('tom'), A('tom')])
print('~~~~~~~~~~~~~~~~~~~~~~')
s = {A('tom'), A('tom')} # set
print(s)
print({tuple('t'), tuple('t')})
print({('tom',), ('tom',)})
print({'tom', 'tom'})
```

方法	意义
`__eq__`	对应==操作符，判断2个对象是否相等，返回bool值

`__hash__` 方法只是返回一个hash值作为set的key，但是 `去重` ，还需要 `__eq__` 来判断2个对象

是否相等。
hash值相等，只是hash冲突，不能说明两个对象是相等的。
因此，一般来说提供 `__hash__` 方法是为了作为set或者dict的key，所以 `去重` 要同时提供 `__eq__` 方法。

不可hash对象isinstance(p1, collections.Hashable)一定为False。
`去重` 需要提供 `__eq__` 方法。

思考：
list类实例为什么不可hash？

练习
设计二维坐标类Point，使其成为可hash类型，并比较2个坐标的实例是否相等？

list类实例为什么不可hash
源码中有一句 `__hash__` = None，也就是如果调用 `__hash__` ()相当于None()，一定报错。
所有类都继承object，而这个类是具有 `__hash__` ()方法的，如果一个类不能被hash，就把 `__hash__` 设置为None。

练习参考

```
from collections import Hashable

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

p1 = Point(4, 5)
p2 = Point(4, 5)
print(hash(p1))
print(hash(p2))

print(p1 is p2)
print(p1 == p2) # True 使用__eq__
print(hex(id(p1)), hex(id(p2)))
```

```
print(set((p1, p2)))

print(isinstance(p1, Hashable))
```

bool

方法	意义
`__bool__`	内建函数bool()，或者对象放在逻辑表达式的位置，调用这个函数返回布尔值。没有定义 `__bool__` ()，就找 `__len__` ()返回长度，非0位真。如果 `__len__` ()也没有定义，那么所有实例都返回真

```
class A: pass

print(bool(A()))
if A():
    print('Real A')

class B:
    def __bool__(self):
        return False

print(bool(B))
print(bool(B()))
if B():
    print('Real B')

class C:
    def __len__(self):
        return 0

print(bool(C()))
if C():
    print('Real C')
```


可视化

方法	意义
`__repr__`	内建函数repr()对一个对象获取字符串表达。 调用 `__repr__` 方法返回字符串表达，如果 `__repr__` 也没有定义，就直接返回object的定义就是显示内存地址信息
`__str__`	str()函数、内建函数format()、print()函数调用，需要返回对象的字符串表达。如果没有定义，就去调用 `__repr__` 方法返回字符串表达，如果 `__repr__` 没有定义，就直接返回对象的内存地址信息
`__bytes__`	bytes()函数调用，返回一个对象的bytes表达，即返回bytes对象

```
class A:
    def __init__(self, name, age=18):
        self.name = name
        self.age = age

    def __repr__(self):
        return 'repr: {},{}'.format(self.name, self.age)

    def __str__(self):
        return 'str:  {},{}'.format(self.name, self.age)

    def __bytes__(self):
        #return "{} is {}".format(self.name, self.age).encode()
        import json
        return json.dumps(self.__dict__).encode()


print(A('tom')) # print函数使用__str__
print([A('tom')]) # []使用__str__，但其内部使用__repr__
print(([str(A('tom'))])) # []使用__str__，str()函数也使用__str__

print('str:a,1') # 字符串直接输出没有引号
s = '1'
print(s)
print(['a'],(s,)) # 字符串在基本数据类型内部输出有引号
print({s, 'a'})

print(bytes(A('tom')))
```

运算符重载

operator模块提供以下的特殊方法，可以将类的实例使用下面的操作符来操作

运算符	特殊方法	含义
<, <=, ==, >, >=, !=	`__lt__`, `__le__`, `__eq__`, `__gt__`, `__ge__`, `__ne__`	比较运算符
+, -, *, /, %, //, **, divmod	`__add__`, `__sub__`, `__mul__`, `__truediv__`, `__mod__`, `__floordiv__`, `__pow__`, `__divmod__`	算数运算符，移位、位运算也有对应的方法
+=, -=, *=, /=, %=, //=, **=	`__iadd__`, `__isub__`, `__imul__`, `__itruediv__`, `__imod__`, `__ifloordiv__`, `__ipow__`	

```
class A:
    def __init__(self, name, age=18):
        self.name = name
        self.age = age

    def __sub__(self, other):
        return self.age - other.age

    def __isub__(self, other):
        return A(self.name, self - other)

tom = A('tom')
jerry = A('jerry', 16)

print(tom - jerry)
print(jerry - tom, jerry.__sub__(tom))

print(id(tom))
tom -= jerry
print(tom.age, id(tom))
```


练习：

完成Point类设计，实现判断点相等的方法，并完成向量的加法

在直角坐标系里面,定义原点为向量的起点.两个向量和与差的坐标分别等于这两个向量相应坐标的和与差若向量的表示为(x,y)形式,

`A(X1,Y1) B(X2,Y2)，则A+B=（X1+X2，Y1+Y2），A-B=（X1-X2，Y1-Y2）`

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def add(self, other):
        return (self.x + other.x, self.y + other.y)

    def __str__(self):
        return '<Point: {},{}>'.format(self.x, self.y)

p1 = Point(1,1)
p2 = Point(1,1)
points = (p1, p2)
print(points[0].add(points[1]))
# 运算符重载
print(points[0] + points[1])

print(p1 == p2)
```


运算符重载应用场景

往往是用面向对象实现的类，需要做大量的运算，而运算符是这种运算在数学上最常见的表达方式。例如，上例中的对+进行了运算符重载，实现了Point类的二元操作，重新定义为Point + Point。
提供运算符重载，比直接提供加法方法要更加适合该领域内使用者的习惯。
int类，几乎实现了所有操作符，可以作为参考。

@functools.total_ordering 装饰器

`__lt__`，`__le__`，`__eq__`，`__gt__`，`__ge__`是比较大小必须实现的方法，但是全部写完太麻烦，使用@functools.total_ordering 装饰器就可以大大简化代码。

但是要求`__eq__`必须实现，其它方法`__lt__`，`__le__`，`__gt__`，`__ge__`实现其一

```
from functools import total_ordering

@total_ordering
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return self.age == other.age

    def __gt__(self, other):
        return self.age > other.age



tom = Person('tom', 20)
jerry = Person('jerry', 16)

print(tom > jerry)
print(tom < jerry)
print(tom >= jerry) #
print(tom <= jerry)
```

上例中大大简化代码，但是一般来说比较实现等于或者小于方法也就够了，其它可以不实现，所以这个装饰器只是看着很美好，且可能会带来性能问题，建议需要什么方法就自己创建，少用这个装饰器。

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return self.age == other.age
```

```
    def __gt__(self, other):
        return self.age > other.age

    def __ge__(self, other):
        return self.age >= other.age

tom = Person('tom', 20)
jerry = Person('jerry', 16)

print(tom > jerry)
print(tom < jerry)
print(tom >= jerry) #
print(tom <= jerry)

print(tom == jerry)
print(tom != jerry)
```

容器相关方法

方法	意义
`__len__`	内建函数len()，返回对象的长度（>=0的整数），如果把对象当做容器类型看，就如同list或者dict。bool()函数调用的时候，如果没有 `__bool__()` 方法，则会看 `__len__()` 方法是否存在，存在返回非0为真
`__iter__`	迭代容器时，调用，返回一个新的迭代器对象
`__contains__`	in 成员运算符，没有实现，就调用 `__iter__` 方法遍历
`__getitem__`	实现self[key]访问。序列对象，key接受整数为索引，或者切片。对于set和dict，key为hashable。key不存在引发KeyError异常
`__setitem__`	和 `__getitem__` 的访问类似，是设置值的方法
`__missing__`	字典或其子类使用 `__getitem__()` 调用时，key不存在执行该方法

```
class A(dict):
    def __missing__(self, key):
        print('Missing key : ', key)
```

```
        return 0

a = A()
print(a['k'])
```

思考
为什么空字典、空字符串、空元组、空集合、空列表等可以等效为False？

练习
将购物车类改造成方便操作的容器类

```
class Cart:
    def __init__(self):
        self.items = []

    def __len__(self):
        return len(self.items)

    def additem(self, item):
        self.items.append(item)

    def __iter__(self):
        return iter(self.items)

    def __getitem__(self, index): # 索引访问
        return self.items[index]

    def __setitem__(self, key, value): # 索引赋值
        self.items[key] = value

    def __str__(self):
        return str(self.items)

    def __add__(self, other): # +
        self.items.append(other)
        return self

cart = Cart()
cart.additem(1)
cart.additem('abc')
cart.additem(3)
```

```
# 长度、bool
print(len(cart))
print(bool(cart))

# 迭代
for x in cart:
    print(x)

# in
print(3 in cart)
print(2 in cart)

# 索引操作
print(cart[1])
cart[1] = 'xyz'
print(cart)

# 链式编程实现加法
print(cart + 4 + 5 + 6)
print(cart.__add__(17).__add__(18))
```


可调用对象

Python中一切皆对象，函数也不例外。

```
def foo():
    print(foo.__module__, foo.__name__)

foo()
# 等价于
foo.__call__()
```

函数即对象，对象foo加上()，就是调用对象的 `__call__()` 方法

可调用对象

方法	意义
`__call__`	类中定义一个该方法，实例就可以像函数一样调用

可调用对象：定义一个类，并实例化得到其实例，将实例像函数一样调用。

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __call__(self, *args, **kwargs):
        return "<Point {}:{}>".format(self.x, self.y)

p = Point(4, 5)
print(p)
print(p())

class Adder:
    def __call__(self, *args):
        ret = 0
        for x in args:
            ret += x
        self.ret = ret
        return ret

adder = Adder()
print(adder(4, 5, 6))
print(adder.ret)
```

练习：
定义一个斐波那契数列的类，方便调用，计算第n项

```
class Fib:
    def __init__(self):
        self.items = [0, 1, 1]

    def __call__(self, index):
        if index < 0:
            raise IndexError('Wrong Index')
        if index < len(self.items):
            return self.items[index]

        for i in range(3, index+1):
```

```
            self.items.append(self.items[i-1] + self.items[i-2])
        return self.items[index]

print(Fib()(100))
```

上例中，增加迭代的方法、返回容器长度、支持索引的方法

```
class Fib:
    def __init__(self):
        self.items = [0, 1, 1]

    def __call__(self, index):
        return self[index]

    def __iter__(self):
        return iter(self.items)

    def __len__(self):
        return len(self.items)

    def __getitem__(self, index):
        if index < 0:
            raise IndexError('Wrong Index')
        if index < len(self.items):
            return self.items[index]

        for i in range(len(self), index+1):
            self.items.append(self.items[i-1] + self.items[i-2])
        return self.items[index]

    def __str__(self):
        return str(self.items)

    __repr__ = __str__

fib = Fib()
print(fib(5), len(fib)) # 全部计算
print(fib(10), len(fib)) # 部分计算
for x in fib:
    print(x)
```

```
print(fib[5], fib[6]) # 索引访问，不计算
```

可以看出使用类来实现斐波那契数列也是非常好的实现，还可以缓存数据，便于检索。