魔术方法 ***

实例化

方法	意义
`__new__`	实例化一个对象 该方法需要返回一个值，如果该值不是cls的实例，则不会调用`__init__` 该方法永远都是静态方法

```
class A:
    def __new__(cls, *args, **kwargs):
        print(cls)
        print(args)
        print(kwargs)
        #return super().__new__(cls)
        #return 1
        return None

    def __init__(self, name):
        self.name = name

a = A()
print(a)
```

`__new__`方法很少使用，即使创建了该方法，也会使用`return super().__new__(cls)`基类object的`__new__`方法来创建实例并返回。

可视化

方法	意义
`__str__`	str()函数、format()函数、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
`__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
`__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('{}'.format(A('tom')))
print([A('tom')]) # []使用__str__，但其内部使用__repr__
print([str(A('tom'))]) # []使用__str__，其中的元素使用str()函数也调用__str__
print(bytes(A('tom')))
```

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为真。 如果`__len__`()也没有定义，那么所有实例都返回真

```
class A: pass
a = A()

print(bool(A))
print(bool(a))

class B:
    def __bool__(self):
        return False

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

class C:
    def __len__(self):
        return 0

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

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__`	

实现自定义类的实例的大小比较（非常重要，排序时使用）

```
class A:
    pass

print(A() == A()) # 可以吗？
print(A() > A()) # 可以吗？
```

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

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

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

    def __ge__(self, other):
        return self.age >= other.age
tom = A('tom')
jerry = A('jerry', 16)
print(tom == jerry, tom != jerry)
print(tom > jerry, tom < jerry)
print(tom >= jerry, tom <= jerry)
```

`__eq__`等于可以推断不等于
`__gt__`大于可以推断小于
`__ge__`大于等于可以推断小于等于
也就是用3个方法，就可以把所有比较解决了

实现两个学生的成绩差

```python
class A:
    def __init__(self, name, score):
        self.name = name
        self.score = score

tom = A('tom', 80)
jerry = A('jerry', 85)
print(tom.score - jerry.score)
```

```python
class A:
    def __init__(self, name, score):
        self.name = name
        self.score = score

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

tom = A('tom', 80)
jerry = A('jerry', 85)
print(tom.score - jerry.score)
print(tom - jerry)
print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')

jerry -= tom # 调用什么
print(tom)
print(jerry) # 显示什么
```

```python
class A:
    def __init__(self, name, score):
        self.name = name
        self.score = score

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

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

    def __repr__(self):
        return "<A name={}, score={}>".format(self.name, self.score)

tom = A('tom', 80)
jerry = A('jerry', 85)
print(tom.score - jerry.score)
print(tom - jerry)
print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')

jerry -= tom # 调用什么
print(tom)
print(jerry)
```

思考：list的+和+=的区别。tuple呢?

上下文管理

文件IO操作可以对文件对象使用上下文管理，使用with...as语法。

```
with open('test') as f:
    pass
```

仿照上例写一个自己的类，实现上下文管理

```
class Point:
    pass

with Point() as p: # AttributeError: __exit__
    pass
```

提示属性错误，没有`__exit__`，看了需要这个属性
某些版本会显示没有`__enter__`

上下文管理对象

当一个对象同时实现了`__enter__`()和`__exit__`()方法，它就属于上下文管理的对象

方法	意义
`__enter__`	进入与此对象相关的上下文。如果存在该方法，with语法会把该方法的返回值作为绑定到as子句中指定的变量上
`__exit__`	退出与此对象相关的上下文。

```
import time

class Point:
    def __init__(self):
        print('init ~~~~~~~~')
        time.sleep(1)
        print('init over')

    def __enter__(self):
        print('enter ~~~~~~~~')

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('exit ============')

with Point() as p:
    print('in with-------------')

    time.sleep(2)
    print('with over')

print('=======end==========')
```

实例化对象的时候，并不会调用enter，进入with语句块调用`__enter__`方法，然后执行语句体，最后离开with语句块的时候，调用`__exit__`方法。

with可以开启一个上下文运行环境，在执行前做一些准备工作，执行后做一些收尾工作。
注意，with并不开启一个新的作用域。

上下文管理的安全性

看看异常对上下文的影响。

```
import time

class Point:
    def __init__(self):
        print('init ~~~~~~~~')
        time.sleep(1)
        print('init over')

    def __enter__(self):
        print('enter ~~~~~~~~')

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('exit =============')

with Point() as p:
    print('in with-------------')
    raise Exception('error')
    time.sleep(2)
    print('with over')

print('=======end==========')
```

可以看出在抛出异常的情况下，with的__exit__照样执行，**上下文管理是安全的。**

with语句

```
# t3.py文件中写入下面代码
class Point:
    def __init__(self):
        print('init')

    def __enter__(self):
        print('enter')

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('exit')

f = open('t3.py')
with f as p:
    print(f)
    print(p)
    print(f is p) # 打印什么
    print(f == p) # 打印什么

p = f = None

p = Point()
with p as f:
    print('in with-------------')
    print(p == f)
```

```
    print('with over')

print('=======end==========')
```

问题在于`__enter__`方法上，它将自己的返回值赋给f。修改上例

```
class Point:
    def __init__(self):
        print('init ~~~~~~~~')

    def __enter__(self):
        print('enter ~~~~~~~~')
        return self # 增加返回值

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('exit =============')

p = Point()
with p as f:
    print('in with-------------')
    print(p == f)
    print('with over')

print('=======end==========')
```

with语法，会调用with后的对象的__enter__方法，如果有as，则将该方法的返回值赋给as子句的变量。

上例，可以等价为f = p.__enter__()

上下文应用场景

1. 增强功能
   在代码执行的前后增加代码，以增强其功能。类似装饰器的功能。
2. 资源管理
   打开了资源需要关闭，例如文件对象、网络连接、数据库连接等
3. 权限验证
   在执行代码之前，做权限的验证，在`__enter__`中处理

思考：如何用支持上下文的类来对add函数计时

生成器函数

```
# 生成器函数 # 调用后返回什么呢？生成器对象（生成器表达式、生成器函数），惰性求值
# 在一个函数定义中，出现了yield语句，此函数就是生成器函数

def foo():
    while True:
        yield 1
# 无限可迭代对象
f = foo()
```

contextlib.contextmanager

contextlib.contextmanager它是一个装饰器实现上下文管理，装饰一个**函数**，而不用像类一样实现`__enter__`和`__exit__`方法。
对下面的函数有要求：必须有yield，也就是这个函数必须返回一个生成器，且只有yield一个值。

也就是这个装饰器接收一个生成器对象作为参数。

```
import contextlib

@contextlib.contextmanager
def foo(): #
    print('enter')  # 相当于__enter__()
    yield # yield 5，yield的值只能有一个，相当于作为__enter__方法的返回值
    print('exit') # 相当于__exit__()


with foo() as f:
    #raise Exception()
    print(f)
```

f接收yield语句的返回值。

上面的程序看似不错但是，增加一个异常试一试，发现不能保证exit的执行，怎么办？
增加try finally。

```
import contextlib

@contextlib.contextmanager
def foo():
    print('enter')
    try:
        yield # yield 5，yield的值只能有一个，作为__enter__方法的返回值
    finally:
        print('exit')

with foo() as f:
    raise Exception()
    print(f)
```

上例这么做有什么意义呢？
当yield发生处为生成器函数增加了上下文管理。这是为函数增加上下文机制的方式。

- 把yield之前的当做__enter__方法执行
- 把yield之后的当做__exit__方法执行
- 把yield的值作为__enter__的返回值

练习：为add函数计时

```
import contextlib
import datetime
import time

@contextlib.contextmanager
def timeit():
    print('enter')
```

```
    start = datetime.datetime.now()
    try:
        yield
    finally:
        print('exit')
        delta = (datetime.datetime.now() - start).total_seconds()
        print('delta = {}'.format(delta))

def add(x, y):
    time.sleep(2)
    return x + y

with timeit():
    add(4, 5)
```

总结

如果业务逻辑简单可以使用函数加contextlib.contextmanager装饰器方式，如果业务复杂，用类的方式加`__enter__`和`__exit__`方法方便。

反射

概述

运行时，runtime，区别于编译时，指的是程序被加载到内存中执行的时候。
反射，reflection，指的是运行时获取类型定义信息。
一个对象能够在运行时，像照镜子一样，反射出其类型信息。
简单说，在Python中，能够通过一个对象，找出其type、class、attribute或method的能力，称为反射或者自省。
具有反射能力的函数有 type()、isinstance()、callable()、dir()、getattr()等

内建函数	意义
getattr(object, name[, default])	通过name返回object的属性值。当属性不存在，将使用default返回，如果没有default，则抛出AttributeError。name必须为**字符串**
setattr(object, name, value)	object的属性存在，则覆盖，不存在，新增
hasattr(object, name)	判断对象是否有这个名字的属性，name必须为**字符串**

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

p1 = Point(4, 5)
print(p1)
```

为上面Point类增加打印的方法

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

p1 = Point(4, 5)
print(p1)
print(p1.x, p1.y)
print(getattr(p1, 'x'), getattr(p1, 'y'))
setattr(p1, 'x', 10)
setattr(Point, '__str__', lambda self: "<Point {},{}>".format(self.x, self.y))
print(p1)
```

反射相关的魔术方法

`__getattr__()`、`__setattr__()`、`__delattr__()`这三个魔术方法，分别测试这三个方法

`__getattr__()`

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

    def __getattr__(self, item):
        print('getattr~~~')
        print(item)
        return 100

p1 = Point(4, 5)
print(p1.x)
print(p1.y)
print(p1.z)
```


实例属性查找顺序为：
`instance.__dict__ --> instance.__class__.__dict__ --> 继承的祖先类（直到object）的__dict__ ---找不到--> 调用__getattr__()`

`__setattr__()`

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

    def __getattr__(self, item):
        print('getattr~~~')
        print(item)
        return 100

    def __setattr__(self, key, value):
```

```
        print('setattr~~~, {}={}'.format(key, value))

p1 = Point(4, 5)
print(p1.x)
print(p1.y)
print(p1.__dict__)
```

p1的实例字典里面什么都没有，而且访问x和y属性的时候竟然访问到了`__getattr__()`，为什么?

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

    def __getattr__(self, item):
        print('getattr~~~')
        print(item)
        return 100

    def __setattr__(self, key, value):
        print('setattr~~~, {}={}'.format(key, value))
        self.__dict__[key] = value
        #setattr(self, key, value) # 对吗

p1 = Point(4, 5)
print(p1.x)
print(p1.y)
print(p1.__dict__)
```

`__setattr__()`方法，可以拦截对实例属性的增加、修改操作，如果要设置生效，需要自己操作实例的`__dict__`。

`__delattr__()`

```
class Point:
    Z = 100
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __delattr__(self, item):
        print('delattr, {}'.format(item))

p1 = Point(4, 5)
del p1.x
del p1.y
del p1.Z
print(p1.__dict__)
print(Point.__dict__)
del Point.Z
print(Point.__dict__)
```

通过实例删除属性，就会尝试调用该魔术方法。

__getattribute__

```
class Point:
    Z = 100
    def __init__(self, x, y):
        self.x = x
        self.y = y

p1 = Point(4, 5)
print(p1.x, p1.y)
print(Point.Z, p1.Z)
print('-' * 30)

# 为Point类增加__getattribute__，观察变化
class Point:
    Z = 100
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getattribute__(self, item):
        print(item)

p1 = Point(4, 5)
print(p1.x, p1.y)
print(Point.Z, p1.Z)
print(p1.__dict__)
```

实例的所有的属性访问，第一个都会调用__getattribute__方法，它阻止了属性的查找，该方法应该返回（计算后的）值或者抛出一个AttributeError异常。

- 它的return值将作为属性查找的结果。
- 如果抛出AttributeError异常，则会直接调用__getattr__方法，因为表示属性没有找到。

```
class Point:
    Z = 100
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getattr__(self, item):
        return 'missing {}'.format(item)

    def __getattribute__(self, item):
        print(item)
        #raise AttributeError('Not Found')
        #return self.__dict__[item]
        #pass
        #return object.__getattribute__(self, item)
        return super().__getattribute__(item)

p1 = Point(4, 5)
print(p1.x, p1.y)
print(Point.Z, p1.Z)
print(p1.__dict__)
```

`__getattribute__`方法中为了避免在该方法中无限的递归，它的实现应该永远调用基类的同名方法以访问需要的任何属性，例如`object.__getattribute__(self, name)`。
注意，除非你明确地知道`__getattribute__`方法用来做什么，否则不要使用它。

总结

魔术方法	意义
`__getattr__()`	当通过搜索实例、实例的类及祖先类**查不到**属性，就会调用此方法
`__setattr__()`	通过`.`访问实例属性，进行增加、修改都要调用它
`__delattr__()`	当通过实例来删除属性时调用此方法
`__getattribute__`	实例所有的属性调用都从这个方法开始

实例属性查找顺序:
`实例调用__getattribute__() --> instance.__dict__ --> instance.__class__.__dict__ --> 继承的祖先类（直到object）的__dict__ --> 调用__getattr__()`