魔术方法

- 分类：
  - 创建与销毁
    - `__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

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

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。

```
class Point:
    def __init__(self):
        print('init')
    def __enter__(self):
        print('enter')
    def __exit__(self, exc_type, exc_val, exc_tb):
        print('exit')


with Point() as f:
    print('do sth.')
```

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。

上下文管理的安全性

看看异常对上下文的影响。

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

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

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


with Point() as f:
    raise Exception('error')
    print('do sth.')
```

```
#init
#enter
#exit
```

可以看出在enter和exit照样执行，**上下文管理是安全的。**

极端的例子
调用sys.exit()，它会退出当前解释器。
打开Python解释器，在里面敲入sys.exit()，窗口直接关闭了。也就是说碰到这一句，Python运行环境直接退出了。

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

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

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

with Point() as f:
    sys.exit(-100)
    print('do sth.')

print('outer')
```

从执行结果来看，依然执行了 `__exit__` 函数，哪怕是退出Python运行环境。
说明**上下文管理很安全。**

with语句

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

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

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

p = Point()
with p as f:
    print(p == f) # 为什么不相等
    print('do sth.')
```

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

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

p = Point()
with p as f:
    print(p == f)
    print('do sth.')
```

`__enter__` 方法返回值就是上下文中使用的对象，with语法会把它的返回值赋给as子句的变量。

`__enter__` 方法和 `__exit__` 方法的参数

`__enter__` 方法 没有其他参数。

`__exit__` 方法有3个参数：
`__exit__(self, exc_type, exc_value, traceback)`
这三个参数都与异常有关。
如果该上下文退出时没有异常，这3个参数都为None。
如果有异常，参数意义如下
`exc_type` ，异常类型
`exc_value` ，异常的值

`traceback` ，异常的追踪信息
`__exit__` 方法返回一个等效True的值，则压制异常；否则，继续抛出异常

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

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

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

p = Point()
with p as f:
    raise Exception('New Error')
    print('do sth.')

print('outer')
```


练习

为加法函数计时
方法1、使用装饰器显示该函数的执行时长
方法2、使用上下文管理方法来显示该函数的执行时长

```
import time

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

装饰器实现

```
import time
```

```
import datetime
from functools import wraps

def timeit(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        print('{} took {}s '.format(fn.__name__,delta))
        return ret
    return wrapper

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

print(add(4, 5))
```

上下文实现

```
import time
import datetime
from functools import wraps

def timeit(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        print('{} took {}s '.format(fn.__name__,delta))
        return ret
    return wrapper

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

```python
class Timeit:
    def __init__(self, fn):
        self.fn = fn

    def __enter__(self):
        self.start = datetime.datetime.now()
        return self.fn

    def __exit__(self, exc_type, exc_val, exc_tb):
        delta = (datetime.datetime.now() - self.start).total_seconds()
        print("{} took {}s".format(self.fn.__name__, delta))


with Timeit(add) as fn:
    #print(fn(4, 6))
    print(add(4, 7))
```

另一种实现，使用可调用对象实现。

代码实现如下

```python
import time
import datetime
from functools import wraps

def timeit(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        print('{} took {}s '.format(fn.__name__,delta))
        return ret
    return wrapper

@timeit
def add(x, y):
    time.sleep(2)
```

```
    return x + y

class Timeit:
    def __init__(self, fn):
        self.fn = fn

    def __enter__(self):
        self.start = datetime.datetime.now()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        delta = (datetime.datetime.now() - self.start).total_seconds()
        print("{} took {}s".format(self.fn.__name__, delta))

    def __call__(self, x, y):
        print(x, y)
        return self.fn(x, y)


with Timeit(add) as timeitobj:
    print(timeitobj(5, 6))
```

根据上面的代码，能不能把类当做装饰器用？

```
import time
import datetime
from functools import wraps


class TimeIt:
    def __init__(self, fn):
        self.fn = fn

    def __enter__(self):
        self.start = datetime.datetime.now()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.delta = (datetime.datetime.now() - self.start).total_seconds()
        print('{} took {}s. context'.format(self.fn.__name__, self.delta))
```

```
        pass

    def __call__(self, *args, **kwargs):
        self.start = datetime.datetime.now()
        ret = self.fn(*args, **kwargs)
        self.delta = (datetime.datetime.now() - self.start).total_seconds()
        print('{} took {}s. call'.format(self.fn.__name__, self.delta))
        return ret


@TimeIt
def add(x, y):
    """This is add function."""
    time.sleep(2)
    return x + y

add(4, 5)
print(add.__doc__)
```

思考
如何解决文档字符串问题？

方法一
直接修改 `__doc__`

```
class TimeIt:
    def __init__(self, fn=None):
        self.fn = fn
        # 把函数对象的文档字符串赋给类
        self.__doc__ = fn.__doc__
```

方法二
使用functools.wraps函数

```
import time
import datetime
from functools import wraps, update_wrapper


class Timeit:
    """This is A Class"""
```

```
    def __init__(self, fn):
        self.fn = fn
        # 把函数对象的文档字符串赋给类
        # self.__doc__ = fn.__doc__
        # update_wrapper(self, fn)
        wraps(fn)(self)

    def __enter__(self):
        self.start = datetime.datetime.now()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        delta = (datetime.datetime.now() - self.start).total_seconds()
        print("{} took {}s. context".format(self.fn.__name__, delta))

    def __call__(self, *args, **kwargs):
        self.start = datetime.datetime.now()
        ret = self.fn(*args, **kwargs)
        delta = (datetime.datetime.now() - self.start).total_seconds()
        print("{} took {}s. call".format(self.fn.__name__, delta))
        return ret


@Timeit
def add(x, y):
    """This is add function."""
    time.sleep(2)
    return x + y


print(add(10, 5))
print(add.__doc__)

print(Timeit(add).__doc__)
```

上面的类即可以用在上下文管理，又可以用做装饰器

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

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，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

```
import contextlib
import datetime
import time

@contextlib.contextmanager
def add(x, y): # 为生成器函数增加了上下文管理
    start = datetime.datetime.now()
    try:
        yield x + y# yield 5，yield的值只能有一个，作为__enter__方法的返回值
    finally:
        delta = (datetime.datetime.now() - start).total_seconds()
        print(delta)


with add(4, 5) as f:
    #raise Exception()
    time.sleep(2)
    print(f)
```

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。