魔术方法 ***

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?

应用

设计一个购物车，能够方便增加商品，能够方便的遍历

```
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):
        # yield from self.items
        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()
# 长度、bool
print(cart, bool(cart), len(cart))

cart.additem(1)
cart.additem('abc')
cart.additem(3)

# 长度、bool
print(cart, bool(cart), len(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()，就是调用此函数对象的`__call__()`方法

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())

# 累加
class Adder:
    def __call__(self, *args):
        self.result = sum(args)
        return self.result

adder = Adder()
print(adder(*range(4, 7)))
print(adder.result)
```

应用

定义一个斐波那契数列的类，方便调用，计算第n项。
增加迭代数列的方法、返回数列长度、支持索引查找数列项的方法。

解法：

为了方便调用，类初始化后，直接在实例上使用参数调用，如下

```
class Fib:
    pass

fib = Fib()
print(fib(10))
```

由此得到代码如下

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

    def __call__(self, index):
        if index < 0: # 不支持负索引
            raise IndexError('Wrong Index')

        if index < len(self.items): # 用Fib()(3)思考边界
```

```
            return self.items[index]

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

fib = Fib()
print(fib(101))
```

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: # 不支持负索引
            raise IndexError('Wrong Index')

        # if index < len(self): # 用Fib()(3)思考边界
        #     return self.items[index]

        # index >= len(self)
        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 enumerate(fib):
    print(x)

print(fib[5], fib[6]) # 索引访问，已经算过，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。

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

上下文应用

如何用支持上下文的类来对add函数计时

```
import time
import datetime

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

class Timeit:
    def __enter__(self):
        self.start = datetime.datetime.now()
        print('开始计时')
        return self

```

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

with Timeit() as t:
    add(4, 5)
```

如果想使用函数名，怎么办?

```
import time
import datetime

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()
        print('开始计时')
        return self

    def __call__(self, *args, **kwargs):
        return self.__fn(*args, **kwargs)

    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 t:
    print(add(4, 5))
    print(t(5, 6))
```