

魔术方法 ***

容器相关方法

方法	意义
<code>__len__</code>	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 <code>__bool__()</code> 方法, 则会看 <code>__len__()</code> 方法是否存在, 存在返回非0为真
<code>__iter__</code>	迭代容器时, 调用, 返回一个 新的迭代器对象
<code>__contains__</code>	in 成员运算符, 没有实现, 就调用 <code>__iter__</code> 方法遍历
<code>__getitem__</code>	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
<code>__setitem__</code>	和 <code>__getitem__</code> 的访问类似, 是设置值的方法
<code>__missing__</code>	字典或其子类使用 <code>__getitem__()</code> 调用时, key不存在执行该方法

```
1 class A(dict):
2     def __missing__(self, key):
3         print('Missing key : ', key)
4         return 0
5
6 a = A()
7 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

```
1 class Cart:
2     def __init__(self):
3         self.items = []
4
5     def __len__(self):
6         return len(self.items)
7
8     def additem(self, item):
9         self.items.append(item)
10
11    def __iter__(self):
12        # yield from self.items
13        return iter(self.items)
14
15    def __getitem__(self, index): # 索引访问
16        return self.items[index]
```

```

17
18     def __setitem__(self, key, value): # 索引赋值
19         self.items[key] = value
20
21     def __str__(self):
22         return str(self.items)
23
24     def __add__(self, other): # +
25         self.items.append(other)
26         return self
27
28 cart = Cart()
29 # 长度、bool
30 print(cart, bool(cart), len(cart))
31
32 cart.additem(1)
33 cart.additem('abc')
34 cart.additem(3)
35
36 # 长度、bool
37 print(cart, bool(cart), len(cart))
38
39 # 迭代
40 for x in cart:
41     print(x)
42
43 # in
44 print(3 in cart)
45 print(2 in cart)
46
47 # 索引操作
48 print(cart[1])
49 cart[1] = 'xyz'
50 print(cart)
51
52 # 链式编程实现加法
53 print(cart + 4 + 5 + 6)
54 print(cart.__add__(17).__add__(18))

```

可调对象

Python中一切皆对象，函数也不例外。

```

1 def foo():
2     print(foo.__module__, foo.__name__)
3
4 foo()
5 # 等价于
6 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
<code>__call__</code>	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。

```

1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __call__(self, *args, **kwargs):
7          return "<Point {}:{}>".format(self.x, self.y)
8
9  p = Point(4, 5)
10 print(p)
11 print(p())
12
13 # 累加
14 class Adder:
15     def __call__(self, *args):
16         self.result = sum(args)
17         return self.result
18
19 adder = Adder()
20 print(adder(*range(4, 7)))
21 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

```

1  class Fib:
2      pass
3
4  fib = Fib()
5  print(fib(10))

```

由此得到代码如下

```

1  class Fib:
2      def __init__(self):
3          self.items = [0, 1, 1]
4
5      def __call__(self, index):
6          if index < 0: # 不支持负索引
7              raise IndexError('Wrong Index')
8
9          if index < len(self.items): # 用Fib()(3)思考边界

```

```

10         return self.items[index]
11
12     # index >= len(self.items)
13     for i in range(len(self.items), index+1):
14         self.items.append(self.items[i-1] + self.items[i-2])
15     return self.items[index]
16
17 fib = Fib()
18 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

```

1 class Fib:
2     def __init__(self):
3         self.items = [0, 1, 1]
4
5     def __call__(self, index):
6         return self[index]
7
8     def __iter__(self):
9         return iter(self.items)
10
11     def __len__(self):
12         return len(self.items)
13
14     def __getitem__(self, index):
15         if index < 0: # 不支持负索引
16             raise IndexError('Wrong Index')
17
18         # if index < len(self): # 用Fib()(3)思考边界
19         #     return self.items[index]
20
21         # index >= len(self)
22         for i in range(len(self), index+1):
23             self.items.append(self.items[i-1] + self.items[i-2])
24         return self.items[index]
25
26     def __str__(self):
27         return str(self.items)
28
29     __repr__ = __str__
30
31 fib = Fib()
32 print(fib(5), len(fib)) # 全部计算
33 print(fib(10), len(fib)) # 部分计算
34 for x in enumerate(fib):
35     print(x)
36
37 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。

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

仿照上例写一个自己的类，实现上下文管理

```
1 class Point:
2     pass
3
4 with Point() as p: # AttributeError: __exit__
5     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
<code>__enter__</code>	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
<code>__exit__</code>	退出与此对象相关的上下文。

```
1 import time
2
3 class Point:
4     def __init__(self):
5         print('init ~~~~~~')
6         time.sleep(1)
7         print('init over')
8
9     def __enter__(self):
10        print('enter ~~~~~~')
11
12    def __exit__(self, exc_type, exc_val, exc_tb):
13        print('exit =====')
14
15 with Point() as p:
16     print('in with-----')
17
18     time.sleep(2)
19     print('with over')
20
21 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。

```
1  import time
2
3  class Point:
4      def __init__(self):
5          print('init ~~~~~~')
6          time.sleep(1)
7          print('init over')
8
9      def __enter__(self):
10         print('enter ~~~~~~')
11
12     def __exit__(self, exc_type, exc_val, exc_tb):
13         print('exit =====')
14
15 with Point() as p:
16     print('in with-----')
17     raise Exception('error')
18     time.sleep(2)
19     print('with over')
20
21 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

```
1  # t3.py文件中写入下面代码
2  class Point:
3      def __init__(self):
4          print('init')
5
6      def __enter__(self):
7          print('enter')
8
9      def __exit__(self, exc_type, exc_val, exc_tb):
10         print('exit')
11
12 f = open('t3.py')
13 with f as p:
14     print(f)
15     print(p)
16     print(f is p) # 打印什么
17     print(f == p) # 打印什么
18
19 p = f = None
20
21 p = Point()
22 with p as f:
23     print('in with-----')
24     print(p == f)
```

```

25     print('with over')
26
27     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

```

1  class Point:
2      def __init__(self):
3          print('init ~~~~~~')
4
5      def __enter__(self):
6          print('enter ~~~~~~')
7          return self # 增加返回值
8
9      def __exit__(self, exc_type, exc_val, exc_tb):
10         print('exit =====')
11
12     p = Point()
13     with p as f:
14         print('in with-----')
15         print(p == f)
16         print('with over')
17
18     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，可以等价于 `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

```

1  import time
2  import datetime
3
4  def add(x, y):
5      time.sleep(2)
6      return x + y
7
8  class Timeit:
9      def __enter__(self):
10         self.start = datetime.datetime.now()
11         print('开始计时')
12         return self
13

```

```

14     def __exit__(self, exc_type, exc_val, exc_tb):
15         delta = (datetime.datetime.now() - self.start).total_seconds()
16         print('took {}s.'.format(delta))
17
18     with Timeit() as t:
19         add(4, 5)

```

如果想使用函数名，怎么办？

```

1  import time
2  import datetime
3
4  def add(x, y):
5      time.sleep(2)
6      return x + y
7
8  class Timeit:
9      def __init__(self, fn):
10         self.__fn = fn
11
12     def __enter__(self):
13         self.start = datetime.datetime.now()
14         print('开始计时')
15         return self
16
17     def __call__(self, *args, **kwargs):
18         return self.__fn(*args, **kwargs)
19
20     def __exit__(self, exc_type, exc_val, exc_tb):
21         delta = (datetime.datetime.now() - self.start).total_seconds()
22         print('{} took {}s.'.format(self.__fn.__name__, delta))
23
24     with Timeit(add) as t:
25         print(add(4, 5))
26         print(t(5, 6))

```