

函数式编程和高阶函数

函数是 Python 内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

高阶函数

变量可以指向函数

【示例】以 Python 内置的求绝对值的函数 `abs()` 为例

```
print(abs(-10))
```

执行结果：

A screenshot of a Python interpreter window. The input line shows `print(abs(-10))` and the output line shows the result `10`.

但是如果只写 `abs` 会输出如下结果：

A screenshot of a Python interpreter window. The input line shows `abs` and the output line shows the result `<built-in function abs>`.

可见，`abs(-10)` 是函数调用，而 `abs` 是函数本身。要获得函数调用结果，我们可以把结果赋值给变量：

```
x=abs(-10)
print(x)
```

【示例】把函数本身赋值给变量

```
f = abs
print(f)
```

执行结果如下：

A screenshot of a Python interpreter window. The input line shows `f = abs` and the output line shows the result `<built-in function abs>`.

函数本身也可以赋值给变量，即：变量可以指向函数。如果一个变量指向了一个函数，那么，可以通过该变量来调用这个函数。

【示例】通过变量来调用函数

```
f = abs
print(f(-10))
```

函数名也是变量

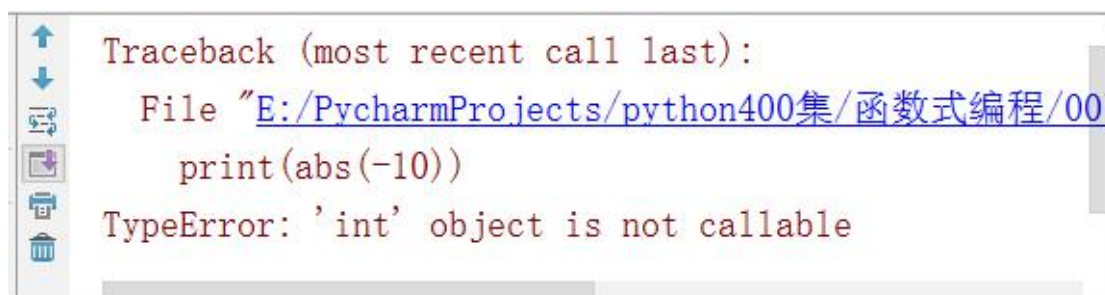
函数名其实就是指向函数的变量！对于 `abs()` 这个函数，完全可以把函数名 `abs` 看成变量，它指向一个可以计算绝对值的函数。

如果把 `abs` 指向其他对象，示例如下：

【示例】abs 指向其他对象

```
abs = 10
print(abs(-10))
```

执行结果：



```
Traceback (most recent call last):
  File "E:/PycharmProjects/python400集/函数式编程/00", line 1, in <module>
    print(abs(-10))
TypeError: 'int' object is not callable
```

把 `abs` 指向 10 后，就无法通过 `abs(-10)` 调用该函数了！因为 `abs` 这个变量已经不指向求绝对值函数了！

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

【示例】高阶函数的示例

```
def add(x, y, f):
    return f(x) + f(y)
print(add(-5, 6, abs))
```

执行结果：



```
11
```

当我们调用 `add(-5, 6, abs)` 时，参数 `x`、`y` 和 `f` 分别接收 -5、6 和 `abs`，根据函数定义，我们可以推导计算过程为：

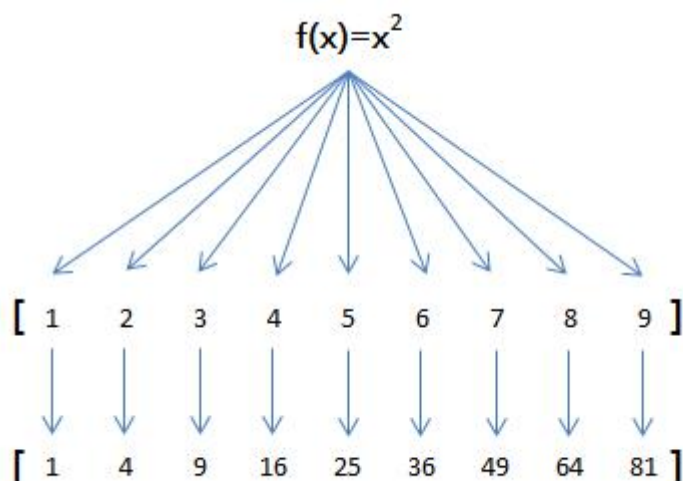
```
x ==> -5
y ==> 6
f ==> abs
f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
```

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。Python 内建的高阶函数有 `map`、`reduce`、`filter`、`sorted`。

map

`map()` 函数接收两个参数，一个是函数，一个是序列，`map` 将传入的函数依次作用到序列的每个元素，并把结果作为新的 `list` 返回。

比如我们有一个函数 $f(x)=x^2$ ，要把这个函数作用在一个 `list [1, 2, 3, 4, 5, 6, 7, 8, 9]` 上，就可以用 `map()` 实现如下：



可能会想，不需要 `map()` 函数，也可以计算出结果，写一个循环，实现代码如下：

【示例】原始 Python 代码实现

```
def f(x):  
    return x * x  
  
L = []  
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:  
    L.append(f(n))  
  
print(L)
```

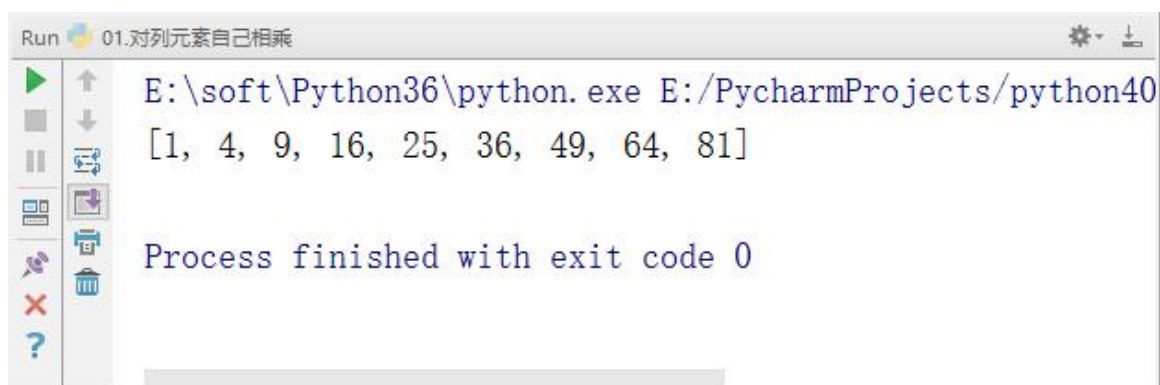
执行结果如下所示：

The screenshot shows a Python IDE window titled "Run 01.对列元素自己相乘". The command prompt shows the execution of the Python script: `E:\soft\Python36\python.exe E:/PycharmProjects/python40`. The output is `[1, 4, 9, 16, 25, 36, 49, 64, 81]`. The status bar indicates "Process finished with exit code 0".

【示例】map 实现

```
def f(x):  
    return x * x  
L=map(f,[1, 2, 3, 4, 5, 6, 7, 8, 9])  
print(list(L))
```

执行结果如下所示：



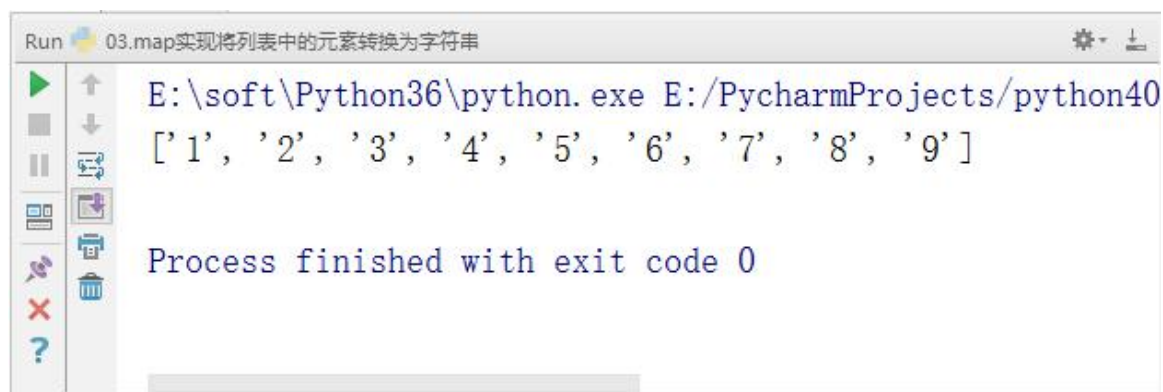
```
Run 01.对列元素自己相乘  
E:\soft\Python36\python.exe E:/PycharmProjects/python40  
[1, 4, 9, 16, 25, 36, 49, 64, 81]  
Process finished with exit code 0
```

从上面的实例可以看到，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的 $f(x)=x^2$ ，还可以计算任意复杂的函数，比如，把这个 list 所有数字转为字符串，实例代码如下：

【示例】map 实现将 list 所有数字转为字符串

```
L=map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
print(list(L))
```

执行结果如下所示：



```
Run 03.map实现将列表中的元素转换为字符串  
E:\soft\Python36\python.exe E:/PycharmProjects/python40  
['1', '2', '3', '4', '5', '6', '7', '8', '9']  
Process finished with exit code 0
```

【示例】map 函数传入两个列表

```
def f2(x,y):  
    return x+y  
L=map(f2,[1,2,3,4],[10,20,30])  
print(list(L))
```

执行结果如下所示：



reduce

reduce 把一个函数作用在一个序列[x1, x2, x3...]上，这个函数必须接收两个参数，reduce 把结果继续和序列的下一个元素做累积计算，其效果就是：

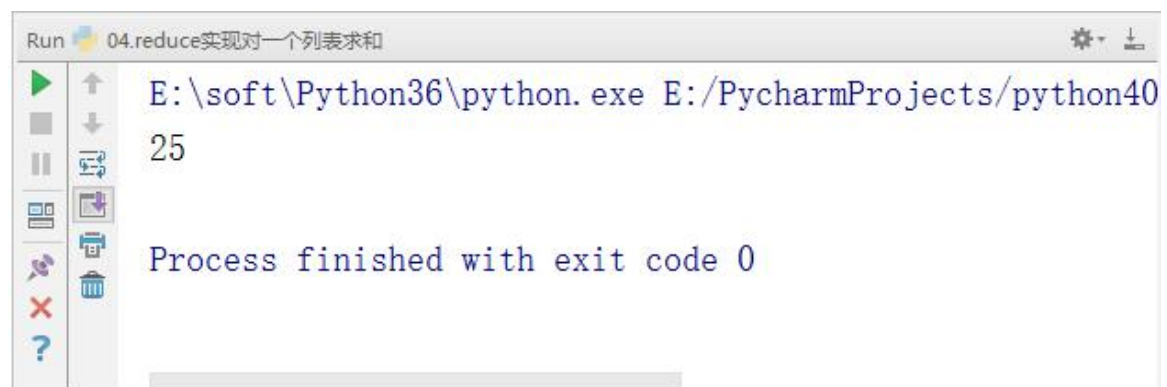
```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比如说对一个序列求和，就可以用 reduce 实现，实例如下：

【示例】reduce 实现对一个序列求和

```
from functools import reduce
def add(x, y):
    return x + y
sum=reduce(add, [1, 3, 5, 7, 9])
print(sum)
```

执行结果如下所示：

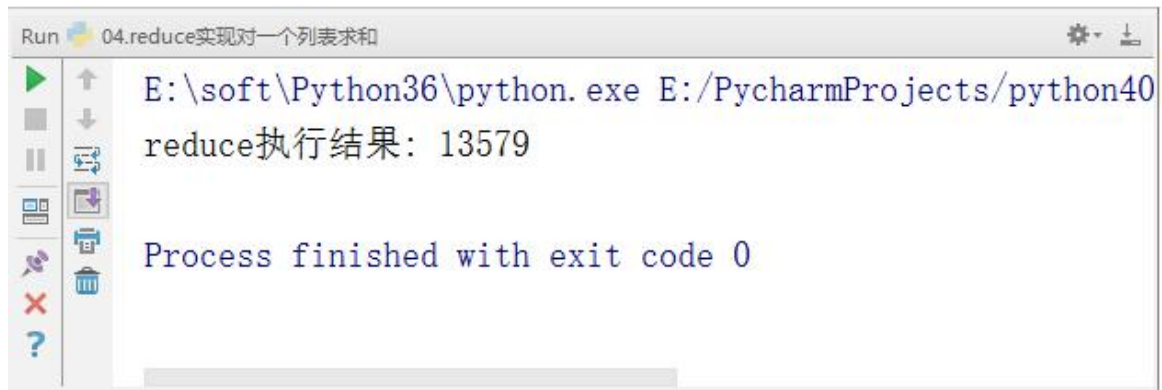


如果想把序列[1, 3, 5, 7, 9]变换成整数 13579。将列表中的每个元素乘以 10 加上后一个元素。

【示例】reduce 实现对一个序列求和

```
def fn(x, y):
    return x * 10 + y
a = reduce(fn, [1, 3, 5, 7, 9])
print("reduce 执行结果:",a)
```

执行结果如下所示：



```
Run 04.reduce实现对一个列表求和
E:\soft\Python36\python.exe E:/PycharmProjects/python40
reduce执行结果: 13579
Process finished with exit code 0
```

filter

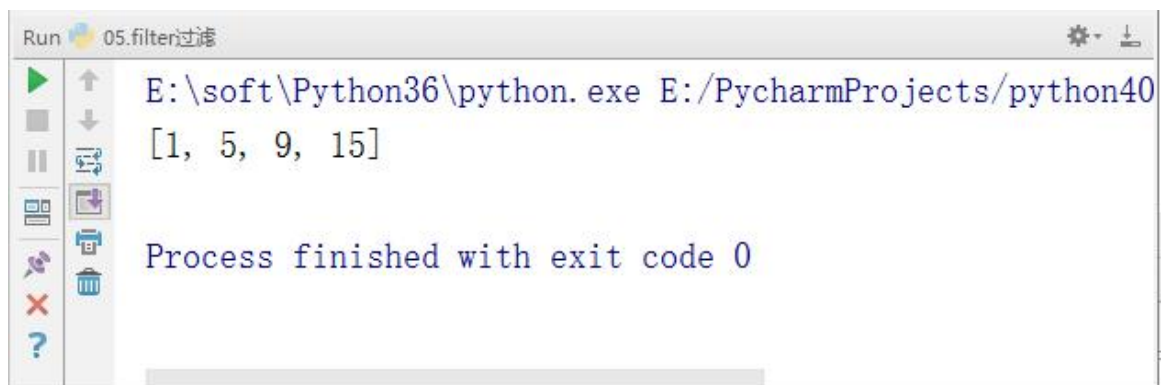
Python 内建的 filter() 函数用于过滤序列。和 map() 类似，filter() 也接收一个函数和一个序列。和 map() 不同的时，filter() 把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。

在一个 list 中，删掉偶数，只保留奇数。

【示例】filter 过滤列表，删掉偶数，只保留奇数

```
# 在一个 list 中，删掉偶数，只保留奇数
def is_odd(n):
    return n % 2 == 1
L=filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])
print(list(L))
```

执行结果如下所示：



```
Run 05.filter过滤
E:\soft\Python36\python.exe E:/PycharmProjects/python40
[1, 5, 9, 15]
Process finished with exit code 0
```

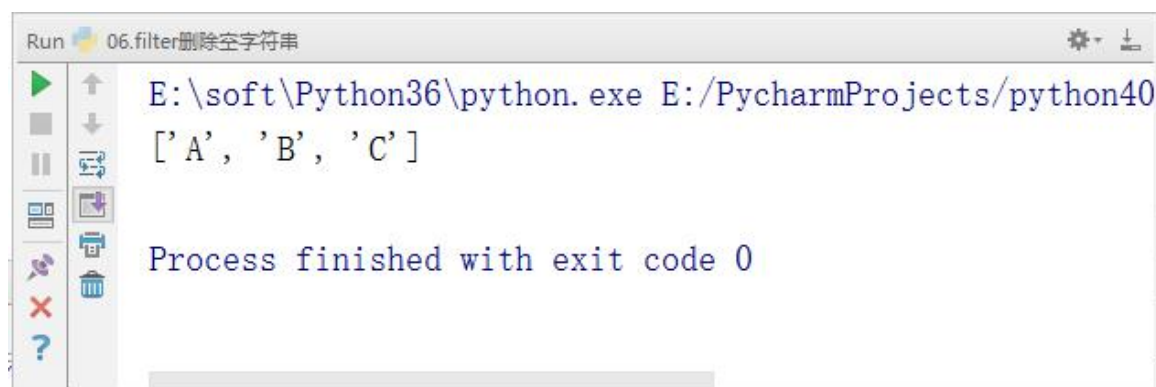
把一个序列中的空字符串删掉，可以这么写：

【示例】filter 序列中的空字符串删掉

```
def not_empty(s):
    return s and s.strip()

L=filter(not_empty, ['A', '', 'B', None, 'C', ' '])
print(list(L))
```

执行结果如下所示：



```
Run 06.filter删除空字符串
E:\soft\Python36\python.exe E:/PycharmProjects/python40
['A', 'B', 'C']
Process finished with exit code 0
```

sorted

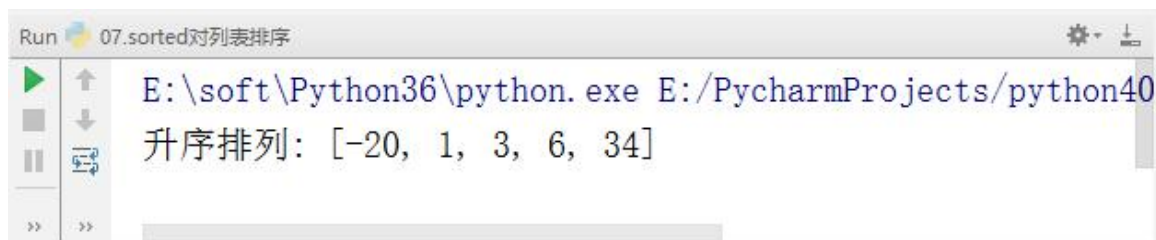
排序算法，排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个 dict 呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素 x 和 y ，如果认为 $x < y$ ，则返回 -1，如果认为 $x == y$ ，则返回 0，如果认为 $x > y$ ，则返回 1，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

Python 内置的 `sorted()` 函数就可以对 list 进行排序：

【示例】sorted 对 list 进行排序

```
sorter1 = sorted([1,3,6,-20,34])
print("升序排列:",sorter1)
```

执行结果如下所示：



```
Run 07.sorted对列表排序
E:\soft\Python36\python.exe E:/PycharmProjects/python40
升序排列: [-20, 1, 3, 6, 34]
```

`sorted()` 函数也是一个高阶函数，它还可以接收一个 `key` 函数来实现自定义的排序

【示例】sorted 的使用

```
sorter1 = sorted([1,3,6,-20,34])
print("升序排列:",sorter1)

# sorted() 函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序
sorter2 = sorted([1,3,6,-20,-70],key=abs)
print("自定义排序:",sorter2)
```



```

sorter2 = sorted([1,3,6,-20,-70],key=abs,reverse=True)
print("自定义反向排序:",sorter2)
# 4.2 字符串排序依照 ASCII
sorter3 = sorted(["ABC","abc","D","d"])
print("字符串排序:",sorter3)
# 4.3 忽略大小写排序
sorter4 = sorted(["ABC","abc","D","d"],key=str.lower)
print("忽略字符串大小写排序:",sorter4)
# 4.4 要进行反向排序，不必改动 key 函数，可以传入第三个参数 reverse=True:
sorter5 = sorted(["ABC","abc","D","d"],key=str.lower,reverse=True)
print("忽略字符串大小写反向排序:",sorter5)

```

执行结果如下所示：

```

Run 07.sorted对列表排序
E:\soft\Python36\python.exe E:/PycharmProjects/python40
升序排列: [-20, 1, 3, 6, 34]
自定义排序: [1, 3, 6, -20, -70]
自定义反向排序: [-70, -20, 6, 3, 1]
字符串排序: ['ABC', 'D', 'abc', 'd']
忽略字符串大小写排序: ['ABC', 'abc', 'D', 'd']
忽略字符串大小写反向排序: ['D', 'd', 'ABC', 'abc']

```

匿名函数

在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在 Python 中，对匿名函数提供了支持。计算 $f(x)=x^2$ 时，除了定义一个 $f(x)$ 的函数外，还可以直接传入匿名函数。使用 `lambda` 可以声明一个匿名函数。

`lambda` 表达式就是一个简单的函数。使用 `lambda` 声明的函数可以返回一个值，在调用函数时，直接使用 `lambda` 表达式的返回值。使用 `lambda` 声明函数的语法格式如下。

```
lambda arg1,arg2,arg3... : <表达式>
```

其中 `arg1/arg2/arg3` 为函数的参数。`<表达式>` 相当于函数体。运算结果是：表达式的运算结果。

匿名函数有个限制，就是只能有一个表达式，不用写 `return`，返回值就是该表达式的结果。

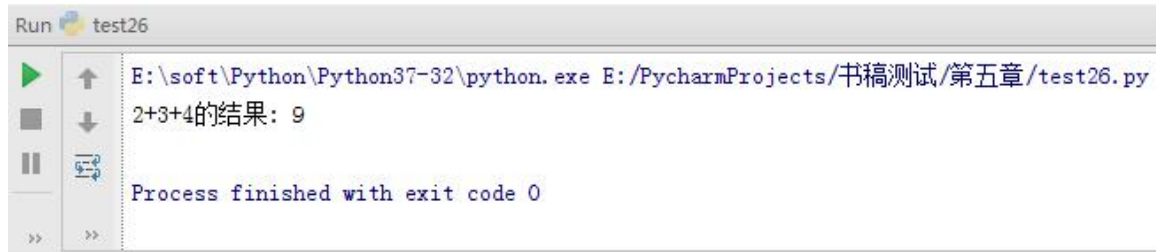
用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是

一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数。

【示例】lambda 表达式使用

```
f = lambda a,b,c:a+b+c
print('2+3+4 的结果:',f(2,3,4))
```

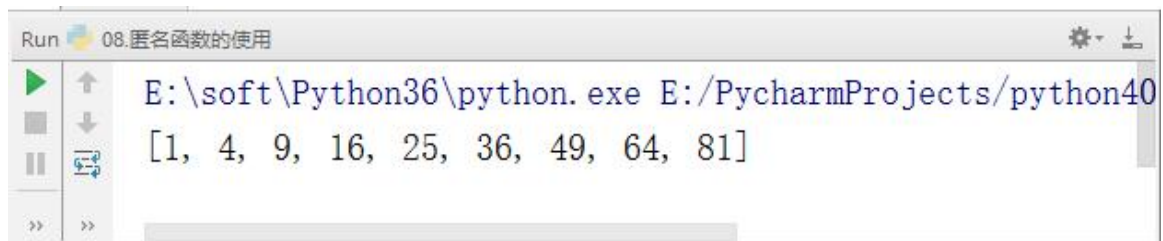
执行结果如图所示：



【示例】匿名函数实现 $f(x)=x*x$

```
L=map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(list(L))
```

执行结果如下所示：



【示例】sorted 对自定义对象进行排序

```
class Student:
    def __init__(self,age,name):
        self.name=name
        self.age=age
stu1=Student(11,'aaa')
stu2=Student(21,'ccc')
stu3=Student(31,'bbb')
student_list=sorted([stu1,stu2,stu3],key=lambda x:x.age)
# student_list=sorted([stu1,stu2,stu3],key=lambda x:x.name)
for stu in student_list:
    print('name:',stu.name,'age:',stu.age)
```

执行结果如下所示：

```

↑
↓
name: aaa age: 11
name: ccc age: 21
name: bbb age: 31
>>

```

闭包

根据字面意思，可以形象地把闭包理解为一个封闭的包裹，这个包裹就是一个函数。当然，还有函数内部对应的逻辑，包裹里面的东西就是自由变量，自由变量可以随着包裹到处游荡。还需要有个前提，这个包裹是创建出来的。在 Python 语言中，闭包意味着要有嵌套定义，内部函数使用外部函数中定义的变量，如果调用一个函数 A，这个函数 A 返回一个函数 B，这个返回的函数 B 就叫作闭包。

【示例】生成一个闭包

```

def func_out(num1):
    def func_in(num2):
        return num1+num2
    return func_in
f=func_out(10)
result=f(20)
print('结果: ',result)

```

执行结果：

```

↑
↓
结果:  30
>>

```

【示例】求两点之间的距离(传统方式实现)

```

import math
def getDis(x1,y1,x2,y2):
    return math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))
#求(1,1)和 (2,2) 分别到原点(0,0)的距离
print(getDis(1,1,0,0))
print(getDis(2,2,0,0))

```

【示例】求两点之间的距离(闭包方式实现)

```

#使用闭包求两点之间的距离
def getDisOut(x1,y1):
    def getDisIn(x2,y2):
        return math.sqrt((x1-x2)**2+(y1-y2)**2)

```

```

    return getDisIn

#求点(1,1)距离原点(0,0)的距离
#调用外部函数
getDisIn=getDisOut(0,0)
result=getDisIn(1,1)
print('点(1,1)距离原点(0,0)的距离',result)

#求点(2,2)距离原点(0,0)的距离
result=getDisIn(2,2)
print('点(2,2)距离原点(0,0)的距离',result)

```

闭包的特殊用途，可以在不修改源代码的前提下，添加新的功能。

【示例】添加日志功能

```

import time
def writeLog(func):
    try:
        f=open('log.txt','a')
        f.write(func.__name__)
        f.write('\t')
        f.write(time.asctime())
        f.write('\n')
    except Exception as e:
        print(e.args)
    finally:
        f.close()
def fun1():
    # writeLog(fun1)
    print('功能 1')

def fun2():
    # writeLog(fun2)
    print('功能 2')

# fun1()
# fun2()

```

#不修改源代码的基础上，添加日志功能

```
def func_out(func):
    def func_in():
        #调用添加日志功能的方法
        writeLog(func)
        func()
    return func_in
fun_in1=func_out(fun1)
fun_in1()

fun_in2=func_out(fun2)
fun_in2()
```

装饰器

在 python 程序中，装饰器就是一种闭包，它可以是闭包的访问方式更简单。

例如有定义 fun1 和 fun2 函数，代码如下：

```
def fun1():
    print('功能 1')
def fun2():
    print('功能 2')
```

现在，假设我们要增强 fun1()函数和 fun2()函数的功能，比如，在函数调用前自动打印日志，但又不希望修改函数的代码，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator 就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的 decorator，可以定义如下：

```
import time
def writeLog(func):
    try:
        f=open('log.txt','a')
        f.write(func.__name__)
        f.write('\t')
        f.write(time.asctime())
        f.write('\n')
    except Exception as e:
        print(e.args)
```

```

    finally:
        f.close()
#不修改源代码的基础上，添加日志功能
def func_out(func):
    def func_in():
        #调用添加日志功能的方法
        writeLog(func)

        func()

    return func_in

```

在 Python 中使用装饰器，需要使用一个特殊的符号 “@” 来实现。在定义装饰器函数或类时，使用 “@装饰器名称” 的形式将符号 “@” 放在函数或类的定义行之前。

```

@func_out
def fun1():
    print('功能 1')

@func_out
def fun2():
    print('功能 2')

```

在使用了装饰器后，在调用函数 fun1 和 fun2 和普通函数调用没有区别，而装饰器定义的功能会自动插入函数 fun1 和 fun2 中。

```

fun1()
fun2()

```

【示例】使用装饰器给 foo() 函数增加功能

```

#给 foo 函数运行之前加输出 print('I am foo')
def funOut(func):
    def funIn():
        print('I am foo')
        func()

    return funIn

@funOut
def foo():
    print('foo 函数正在运行')

foo()

```

执行结果：



```
I am foo
foo 函数正在运行
```

【示例】多个装饰器

```
def funOut(func):
    print('我是装饰器 1')
    def funIn():
        print('I am foo')
        func()
    return funIn

def funOut2(func):
    print('我是装饰器 2')
    def funIn():
        print('hello')
        func()
    return funIn

@funOut2
@funOut
def foo():
    print('foo 函数正在运行')

foo()
```

执行结果：



```
我是装饰器1
我是装饰器2
hello
I am foo
foo 函数正在运行
```

由上面的示例可以看到，如果给功能函数添加多个装饰器时候，离功能函数最近的先进行装饰。

【示例】指定参数的装饰器

```
def fun_out(func):
    def fun_in(x,y):
        func(x,y)
    return fun_in

@fun_out
def test(a,b):
    print('参数 a b 的值: ',a,b)

test(10,20)

#三个参数
def fun_out1(func):
    def fun_in(x,y,z):
        func(x,y,z)
    return fun_in

def test1(a,b,c):
    print('参数 a b c 的值: ',a,b,c)

test1(10,20,30)
```

执行结果:



```
↑ 参数a b 的值: 10 20
↓ 参数a b c的值: 10 20 30
←
→
>>
```

【示例】通用的装饰器

```
def fun_out(func):
    def fun_in(*args,**kwargs):
        return func(*args,**kwargs)
    return fun_in

@fun_out
def test1(a):
    print('一个参数 a: ',a)

@fun_out
def test2(a,b):
```

```

print('两个参数 a b: ',a,b)\

@fun_out
def test3(a,b,c):
    print('两个参数 a b c: ',a,b,c)

test1(10)
test2(10,20)
test3(10,20,30)

```

执行结果：



```

一个参数a: 10
两个参数a b: 10 20
两个参数a b c: 10 20 30

```

偏函数

Python 的 `functools` 模块提供了很多有用的功能, 其中一个就是偏函数(Partial function)。要注意, 这里的偏函数和数学意义上的偏函数不一样。

偏函数是用于对函数固定属性的函数, 作用就是把一个函数某些参数固定住(也就是设置默认值), 返回一个新的函数, 调用这个新的函数会更简单。

在介绍函数参数的时候, 我们讲到, 通过设定参数的默认值, 可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下:

`int()`函数可以把字符串转换为整数, 当仅传入字符串时, `int()`函数默认按十进制转换, 代码如下:

```
print(int('12345'))
```

但 `int()`函数还提供额外的 `base` 参数, 默认值为 10。如果传入 `base` 参数, 就可以做 N 进制的转换:

```

#base 参数
print('转换为八进制',int('12345', base=8))
print('转换为十六进制',int('12345', 16))

```

假设要转换大量的二进制字符串, 每次都传入 `int(x, base=2)`非常麻烦, 于是, 我们想到, 可以定义一个 `int2()`的函数, 默认把 `base=2` 传进去, 现在定义一个 `int2` 函数, 代码如下:

```

def int2(x, base=2):
    return int(x, base)

```

这样，我们转换二进制就非常方便了：

```
print(int2('1000000')) #64
print(int2('1010101')) #85
```

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()`，可以直接使用下面的代码创建一个新的函数 `int2`：

```
import functools
int2 = functools.partial(int, base=2)
print(int2('1000000')) #64
print(int2('1010101')) #85
```

所以，简单总结 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 `int2` 函数，仅仅是把 `base` 参数重新设定默认值为 2，但也可以在函数调用时传入其他值：

```
print(int2('1000000', base=10)) # 1000000
```

当函数的参数个数太多，需要简化时，使用 `functools.partial` 可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。