python中可作为高阶函数的条件

> - [x] 函数可以作为输入参数
> - [x] 函数可作为参数返回

两个条件满足其一即可作为高阶函数

filter函数

```
filter(function, iterable) # 将可迭代对象按照function指定的操作进行过滤

# 将 'a' 过滤出去，剩下['b', 'c', 'd']
s = ['a', 'b', 'c', 'd']
def func(s):
    if i != 'a':
        return i

ret = filter(func, s)
print(list(ret))
```

> 输出： ['b', 'c', 'd']

map函数

```
map(function, iterable)  # 对可迭代对象的每个元素做function指定的操作

# 把每一个元素后面加上'xxx'
s = ['per1', 'per2', 'per3']

def func(s):
    return s + 'xxx'

print(list(map(func, s)))
```

> 输出: ['per1xxx', 'per2xxx', 'per3xxx']

reduce函数

```
reduce(funtion, iterable)
# 将传入的两个参数做function操作以后的结果作为下一次调用function操作的第一个参数，
# 并将iterable中的第三个元素作为第二个参数传入，以此类推

from functools import reduce
```

```
def func(a, b):
    return a + b

print(reduce(func, range(1, 10)))
```

> 输出： 45

lambda表达式

```
lambda <parameters> : <operation> # lambda表达式，匿名函数,缩短代码

#lambda与reduce配合使用求阶乘
from functools import reduce
print(reduce(lambda a,b:a*b, range(1, 10)))
```

> 输出: 362880

闭包

如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就认为是闭包(closure)

```
def outer():
    x = 5
    def inner(): #这是一个闭包(closure)
        print(x)

    return inner

f = outer()
f()
```

> 输出: 5

装饰器

> 引入场景：为某功能函数加上统计执行时间的功能

```
def foo():
    print('foo...')
    time.sleep(3)

def bar():
    print('bar...')
    time.sleep(2)
```

```
def show_time(f):
    def inner():
        begin = time.time()
        f()
        end = time.time()
        print('time is : %s ' % (end-begin))
    return inner
    # 还原调用方式
foo = show_time(foo)
bar = show_time(bar)
foo()
bar()
```

> 输出:

```
foo...
time is : 3.0008883476257324
bar...
time is : 2.0000112056732178
```

- **python提供的优雅方式使用装饰器 - 完整demo**

```
import time

# 定义装饰器函数
def show_time(f):
    '''
    为某一个功能函数加上统计其执行时间的功能
    :param f:将被装饰的功能函数
    :return: 经过装饰后的函数
    '''
    def inner():
        begin = time.time()
        f()
        end = time.time()
        print("%s function is run at %s s" % (f.__name__, (end - begin)))

    return inner

@show_time
# 等价于 fun1 = show_time(fun1)
def fun1():
    print("fun1...")
    time.sleep(3)
@show_time
# 等价于 fun2 = show_time(fun2)
def fun2():
    print("fun2...")
    time.sleep(3)

fun1()
fun2()
```

> 输出:

```
fun1...
fun1 function is run at 3.000779390335083 s
fun2...
fun2 function is run at 3.0008695125579834 s
```

- **扩展装饰器 —— *当功能函数需要更多参数时***

```
import time
def show_time(f):
    '''
    装饰器函数，统计功能函数执行的时间
    :param f: 被装饰的功能函数
    :return: 装饰后的函数句柄
    '''
    def inner(*args):
        begin = time.time()
        f(*args)
        end = time.time()
        print("%s funtion is run at: %s s" % (f.__name__, (end - begin)))

    return inner
@show_time
def add(*args):
    '''
    打印多个数求和的结果
    :param args:一组数
    :return:None
    '''
    sum = 0
    for i in args:
        sum += i
    print('sum is :',sum)
    time.sleep(3)

add(1, 2, 3, 4, 5)
```

> 输出:

```
sum is : 15
add funtion is run at: 3.000849962234497 s
```

- **扩展装饰器 —— *当装饰器函数需要更多参数时***

```
# 除了统计执行时间之外，还要进行日志记录
import time
def logging(flag):
    '''
    增加打印日志的功能
    :param flag:
        是否打印日志 'log'-> 打印
    :return: 装饰器函数句柄
    '''
```

```
    def show_time(f):
        '''
        装饰器函数，统计功能函数执行的时间
        :param f: 被装饰的功能函数
        :return: 装饰后的函数句柄
        '''
        def inner(*args):
            begin = time.time()
            f(*args)
            end = time.time()
            print("%s funtion is run at: %s s" % (f.__name__, (end - begin)))
            if flag == 'log':
                print("Logging...")
        return inner

    return show_time

@logging('log')
def add(*args):
    '''
    打印多个数求和的结果
    :param args:一组数
    :return:None
    '''
    sum = 0
    for i in args:
        sum += i
    print('sum is :',sum)
    time.sleep(3)

add(1, 2, 3, 4, 5)
```

> 输出：

```
sum is : 15
add funtion is run at: 3.000110149383545 s
Logging...
```