函数式编程

高阶函数介绍

变量指向函数

- 变量可以指向：常量，列表，字典，还可以指向函数

```
# 1. 调用函数abs
abs(-10)
# 2. abs(-10)是对函数的调用，而只写abs是函数本身
abs
# # 3. 获取函数调用的结果
x = abs(-10)
print(x)
y = abs
print(y)
z = y(-1000)
print(z)
```

```
10
<built-in function abs>
1000
```

- 结论：函数本身也可以赋值给变量，即：变量可以指向函数

函数作为参数

- 变量可以指向函数，而函数可以接收变量，所以：**一个函数可以接收另一个函数作为参数**，这种函数称为**高阶函数**

```
# 定义一个高阶函数
def add(x, y, f):
    return f(x) + f(y)
# 把abs当参数传进去
add(-5, -6, abs)
```

```
11
```

过程解释：

1. x = -5
2. y = -6
3. f = abs
4. f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
5. return 11

结论：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。

内建高阶函数

map

- map()函数：接收两个参数，第一个是函数，第二个是可迭代对象
- 作用：将可迭代对象依次代入这个函数，然后将结果组成一个列表返回
- 使用场景：对一个列表统一进行某个操作

```
# 将一个列表的所有数据转换成字符串
a = [1, 2, 3, 4, 5]
b = []
for i in a:
    t = str(i)
    b.append(t)
print(b)

# 使用map函数
c = map(str, a)
# print(c)
print(list(c))
```

```
['1', '2', '3', '4', '5']
['1', '2', '3', '4', '5']
```

- 练习1：把不规范的英文名字，改成大写开头的规范名字，如：输入`['tOm', 'jerrY', 'JULY']`，输出`['Tom', 'Jerry', 'July']`

```
# 定义1个函数
def modify_name(name):
    return name[0].upper() + name[1:].lower()
# a = "XIAOBAI"
# modify_name(a)

a = ['tOm', 'jerrY', 'JULY']
# # 注意，只能传名称，不能带括号
b = map(modify_name, a)
print(list(b))
```

```
['Tom', 'Jerry', 'July']
```

filter

- filter()函数：接收一个函数，和一个可迭代对象
- 作用：将可迭代对象依次传入该函数，通过返回值是True还是False决定去留
- 使用场景：过滤和筛选元素

```
# 筛选大于5的数
a = [1,2,3,4,5,6,7]
def get_5(x):
    if x > 5:
        return True
# filter过滤，当不满足条件当时候，函数return None，filter判断为False，则不取该元素
b = filter(get_5, a)
print(list(b))
```

```
[6, 7]
```

- 练习：找出列表中的http链接：`['http://www.baidu.com', 'apple', 'http://weibo.cn', '中国人']`

```
a = ['http://www.baidu.com', 'apple', 'http://weibo.cn', '中国人']

def get_http(x):
    if 'http' in x:
        return True

b = filter(get_http, a)
print(list(b))
```

```
['http://www.baidu.com', 'http://weibo.cn']
```

```
# 取回数
a = [12321, 34543, 1231, 3344]
def is_palindrome(n):
    if str(n) == str(n)[::-1]:
        return True
b = filter(is_palindrome, a)
print(list(b))
```

```
[12321, 34543]
```

sorted

- sorted()函数：排序
- sorted()也是一个高阶函数。用sorted()排序的关键在于实现一个映射函数。
- 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。

```
# 按大小排序
# sorted([36, 5, -12, 9, -21])
# 按绝对值排序
# sorted([36, 5, -12, 9, -21], key=abs)
# # 反向排序
sorted([36, 5, -12, 9, -21], reverse=True)
```

```
[36, 9, 5, -12, -21]
```

- 练习1：按学生成绩排序
- 练习2：列表按字符串的长度重新排序

```
# 按成绩排序
a = [('Tom', 75), ('Jerry', 92), ('Apple', 66), ('Ben', 88)]
def ll(x):
    return x[1]

sorted(a, key=lambda x: x[1])
```

```
[('Apple', 66), ('Tom', 75), ('Ben', 88), ('Jerry', 92)]
```

```
a = ["hello", "Tom", "ok"]
sorted(a, key=lambda x: len(x))
```

```
['ok', 'Tom', 'hello']
```

返回函数

- 高阶函数不仅可以接收函数作为参数，还可以把把函数当结果返回

作用域

- 作用域指程序运行期间**变量**可被访问的范围
- 定义在函数内的变量叫局部变量，局部变量规定只能在函数内部使用，不能在函数外部使用

```
def foo():
     # 局部变量
    num = 11
    print(num)
# NameError: name 'num' is not defined
print(num)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-22-3d100ad9edba> in <module>
      4     print(num)
      5 # NameError: name 'num' is not defined
----> 6 print(num)
```

```
NameError: name 'num' is not defined
```

- 定义在模块最外层的变量是全局变量，它是全局可用的，函数内部也可以使用

```
num = 10 # 全局变量
def foo():
    print(num)  # 10
foo()
```

```
10
```

嵌套函数

- 函数内部再定义一个函数，称为：**嵌套函数**

```
 # print_msg 是外部函数
def print_msg():
    msg = "duoceshi"
    # printer是嵌套函数，也叫内部函数
    def printer():
        print(msg)
    # 调用函数，在外部函数中执行这个内部函数
    printer()
# 调用外部函数，输出 duoceshi
print_msg()
#
# print(msg)
```

```
duoceshi
```

解释：

1. printer嵌套函数(内部函数)可以正常访问msg，因为对于printer而言 `msg` 像是一个全局变量
2. 这个msg对于print_msg外部函数，又像一个局部变量
3. 这个特殊的msg变量，专业术语叫：非局部变量(non-local)
4. 如何在脱离函数本身的作用范围，局部变量还可以被访问到呢？答案是通过**闭包**

闭包函数

- 把函数作为返回值返回

```
# 外部函数
def print_msg():
    msg = "duoceshi"
    # 内部函数
    def printer():
        print(msg)
    # 返回内部函数，注意：没有括号
    return printer

# 调用外部函数，获得一个函数对象
another = print_msg()
print(another)
# 在外部，再次调用这个函数对象，即可输出duoceshi
another()
```

```
<function print_msg.<locals>.printer at 0x1050b1a60>
duoceshi
```

- 解释：
    1. 局部变量仅在函数执行期间可用，参考作用域第一个例子
    2. 通过闭包，我们可以访问到函数内部的变量

3. 这个 `another` 就是一个闭包，闭包本质上是一个函数，它又两部分组成：`printer` 函数，`msg` 变量
4. 闭包使得这些变量的值始终保存在内存里

- 总结：闭包，顾名思义，就是一个封闭的包裹，里面包裹着自由变量，就像在类里面定义的属性值一样，自由变量的可见范围随同包裹，哪里可以访问到这个包裹，哪里就可以访问到这个自由变量。

闭包使用

1. 避免使用全局变量，因为全局变量不安全
2. 闭包允许函数与环境关联起来，类似面向对象编程(对象允许我们将某些数据与1个或多个方法相关联)

```
def adder(x):
    def wrapper(y):
        return x + y
    return wrapper

# 这里的5，相当于申明了一个全局变量，后续函数的调用都可以使用
adder5 = adder(100)
print(adder5)
# # 输出 15
# adder5(10)
# # 输出 11
adder5(6)
# # 查看属性，发现有个cell对象，证明这是个闭包函数
adder5.__closure__
# # 可以查看闭包的自由变量
# adder5.__closure__[0].cell_contents
```

```
<function adder.<locals>.wrapper at 0x1052389d8>
```

```
(<cell at 0x105a421f8: int object at 0x101fc9c80>,)
```

补充：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

匿名函数

- 方便，可读性强

```
# 直接定义
def f(x):
    return x * x
list(map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

# 匿名函数，x表示函数参数，x * x 表示返回内容，只能有一个表达式
list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- 匿名函数特点：
    1. 只能有一个表达式
    2. 匿名函数也是一个函数对象，可以赋值给一个变量
    3. 经常配合高阶函数一起使用

```
# 赋值给变量
f = lambda x: x * x
f(2)
```

```
4
```

- 匿名函数的优点：
    1. 代码简洁，逻辑清晰
    2. 没有函数名，不用担心函数名冲突
- 练习：求出1-20的奇数

```
L = list(filter(lambda x : x % 2==1, range(1, 20)))
print(L)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

- 小结：Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

装饰器

logging模块

- loggin用于日志输出

```
# 初级使用
def add(a=0, b=0):
    print("[正在执行的函数是]:", add.__name__)
    print("[参数a为]:", a)
    print("[参数b为]:", b)
    c = a + b
    return c
add(1,2)
```

```
[正在执行的函数是]: add
[参数a为]: 1
[参数b为]: 2
```

```
3
```

```
# 进阶使用
import logging
# 定义日志级别，和日志打印形式
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')
def add(a=0, b=0):
    logging.info("[正在执行的函数是]:{}".format(add.__name__))
    logging.info("[参数a为]:{}".format(a))
    logging.info("[参数b为]:{}".format(b))
    c = a + b
    return c
add(3, 5)
```

```
2019-08-30 19:30:41,432 INFO     [正在执行的函数是]:add
2019-08-30 19:30:41,433 INFO     [参数a为]:3
2019-08-30 19:30:41,434 INFO     [参数b为]:5
```

```
8
```

回调函数

- 考虑到如果写了很多函数，每个函数都要写一遍这些重复代码，很繁琐，于是咱们进行以下优化

```
# 通过回调函数优化
import logging
# 定义日志级别，和日志打印形式
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 单独写1个回调函数，专门用来打印日志，这里接收三个参数，函数名，可变参数，关键字参数
def use_logging(func, *args, **kwargs):
    # 打印函数名
    logging.info("[当前调用的函数是]: {}".format(func.__name__))
    # 打印位置参数
    for i in range(len(args)):
        logging.info("[第{}个参数是]: {}".format(i, args[i]))
    # 打印关键字参数
    for k, v in kwargs:
        logging.info("key is: {}  value is: {}".format(k, v))
    # 实际调用函数，然后将结果返回
    return func(*args, **kwargs)

def add(a, b):
    c = a + b
    return c
# 也能实现效果，但是破坏了原本的逻辑结构，需要重写代码
use_logging(add, 4, 2)
```

```
2019-08-30 20:11:19,547 INFO     [当前调用的函数是]: add
2019-08-30 20:11:19,548 INFO     [第0个参数是]: 4
2019-08-30 20:11:19,549 INFO     [第1个参数是]: 2
```

```
6
```

- 解释：
    1. 通过回调函数，将函数名，和参数一起传入回调函数，然后进行日志增补后，再将函数执行再返回结果
    2. 回调函数会真实的进行函数的调用，破坏了原本的逻辑结构

基础型装饰器

- 将回调函数改写成装饰器

```
# 通过回调函数优化
import logging
# 定义日志级别，和日志打印形式
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 回调函数，增加一个内层函数，改写成装饰器
def use_logging(func):
    # 内层函数
    def wrapper(*args, **kwargs):
        logging.info("[当前调用的函数是]: {}".format(func.__name__))
        for i in range(len(args)):
            logging.info("[第{}个参数是]: {}".format(i, args[i]))
        for k, v in kwargs:
            logging.info("key is: {}  value is: {}".format(k, v))
        # 执行函数，并把结果返回
        return func(*args, **kwargs)
    # 返回wrapper内层函数
    return wrapper

def add(a, b):
    c = a + b
    return c

# 装饰器第一种用法
# 使用装饰器包裹一下我们的函数，并赋值给变量
f = use_logging(add)
# 使用函数变量去调用
f(2,3)
```

```
2019-08-30 20:23:57,073 INFO     [当前调用的函数是]: add
2019-08-30 20:23:57,074 INFO     [第0个参数是]: 2
2019-08-30 20:23:57,074 INFO     [第1个参数是]: 3
```

```
5
```

- 装饰器提供了一种特殊用法：`@`语法糖，在定义函数的时候直接使用，避免赋值操作，简化代码

```
# 最终版本
```

```
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')

def use_logging(func):
    def wrapper(*args, **kwargs):
        logging.info("[当前调用的函数是]: {}".format(func.__name__))
        for i in range(len(args)):
            logging.info("[第{}个参数是]: {}".format(i, args[i]))
        for k, v in kwargs.items():
            logging.info("key is: {}  value is: {}".format(k, v))
        return func(*args, **kwargs)
    return wrapper

@use_logging
def add(a, b):
    c = a + b
    return c

@use_logging
def test(x, y, z=2):
    return x + y -z

add(a=1, b=44)
test(1,2,z=3)
```

```
2019-08-30 21:47:37,377 INFO     [当前调用的函数是]: add
2019-08-30 21:47:37,377 INFO     key is: a  value is: 1
2019-08-30 21:47:37,378 INFO     key is: b  value is: 44
2019-08-30 21:47:37,379 INFO     [当前调用的函数是]: test
2019-08-30 21:47:37,380 INFO     [第0个参数是]: 1
2019-08-30 21:47:37,380 INFO     [第1个参数是]: 2
2019-08-30 21:47:37,381 INFO     key is: z  value is: 3
```

```
0
```

参数型装饰器

- 带参数的装饰器，实际上是对原有装饰器的再一次封装，可以理解为一个带参数的闭包
- 当我们调用`@use_logging("info")`的时候，python能够发现这一层的封装，并把参数传递到装饰器的环境中

```python
# 最终版本
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')
# 再封装一层，用来带参数
def use_logging(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if level == "info":
                logging.info("[当前调用的函数是]: {}".format(func.__name__))
                for i in range(len(args)):
                    logging.info("[第{}个参数是]: {}".format(i, args[i]))
                for k, v in kwargs.items():
                    logging.info("key is: {}  value is: {}".format(k, v))
            return func(*args, **kwargs)
        return wrapper
    return decorator

@use_logging("info")
def add(a, b):
    c = a + b
    return c

add(a=1, b=44)
```

```
2019-08-30 21:53:03,794 INFO     [当前调用的函数是]: add
2019-08-30 21:53:03,795 INFO     key is: a  value is: 1
2019-08-30 21:53:03,796 INFO     key is: b  value is: 44
```

```
45
```

类装饰器

- 相比函数装饰器，类装饰器具有：灵活度大，高内聚，封装性等优点

1. 内外层都不带参数的装饰器

```python
# 内外层都不带参数
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')
```

```
class Logit(object):
    def __init__(self, func):
        self._func = func

    def __call__(self, *args, **kwargs):
        logging.info("[当前调用的函数是]: {}".format(self._func.__name__))
        # 因为被修饰函数没有返回，所以这里也可以不反回，但建议都进行return
        return self._func()

@Logit
def add():
    print("I'm ok")

add()
```

```
2019-08-30 23:01:07,543 INFO     [当前调用的函数是]: add
```

```
I'm ok
```

2. 类装饰器带参数，函数不带参数

```
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

class Logit(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __call__(self, func, *args, **kwargs):
        logging.info("[当前调用的函数是]: {}".format(func.__name__))
        total = self.x + self.y
        logging.info("[类变量相加为]: {}".format(total))
        return func

@Logit(1, 2)
def add():
    print("hellokitty")

add()
```

```
2019-08-30 23:01:15,390 INFO     [当前调用的函数是]: add
2019-08-30 23:01:15,391 INFO     [类变量相加为]: 3
```

```
hellokitty
```

- 类装饰器不带参数，被修饰对象带参数

```
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

class Logit(object):
    def __init__(self, func):
        self._func = func

    def __call__(self, *args, **kwargs):
        logging.info("[当前调用的函数是]: {}".format(self._func.__name__))
        for i in range(len(args)):
            logging.info("[第{}个参数是]: {}".format(i, args[i]))
        for k, v in kwargs.items():
            logging.info("key is: {}  value is: {}".format(k, v))
        return self._func(*args, **kwargs)

@Logit
def add(a, b):
    c = a + b
    return c

add(2, 3)
```

```
2019-08-30 23:01:59,016 INFO     [当前调用的函数是]: add
2019-08-30 23:01:59,017 INFO     [第0个参数是]: 2
2019-08-30 23:01:59,018 INFO     [第1个参数是]: 3
```

```
5
```

- 类装饰器和被修饰对象，都带参数，完全体

```
# 最终版本
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')
```

```
class Logit(object):
    def __init__(self, level):
        self.level = level

    def __call__(self, func, *args, **kwargs):
        # 内部还需要再封装一层，定义内部函数
        def wrapper(*args, **kwargs):
            # 类变量可以直接使用
            if self.level == "info":
                logging.info("[当前调用的函数是]: {}".format(func.__name__))
                for i in range(len(args)):
                    logging.info("[第{}个参数是]: {}".format(i, args[i]))
                for k, v in kwargs.items():
                    logging.info("key is: {}  value is: {}".format(k, v))
                # 执行函数
                return func(*args, **kwargs)
        # 将内部函数对象返回
        return wrapper

@Logit(level="info")
def add(a, b):
    c = a + b
    return c

add(33, b =44)
```

```
2019-08-30 23:06:06,450 INFO     [当前调用的函数是]: add
2019-08-30 23:06:06,451 INFO     [第0个参数是]: 33
2019-08-30 23:06:06,451 INFO     key is: b  value is: 44
```

```
77
```

- 作业：写一个类装饰器，打印函数的运算时间

```
import time
class Timer:
    def __init__(self, func):
        self._func = func
    def __call__(self, *args, **kwargs):
        before = time.time()
        result = self._func(*args, **kwargs)
```

```
        after = time.time()
        print("函数执行时长: ", after - before)
        return result

@Timer
def add(x, y=10):
    return x + y

add(0, 0)
```

```
函数执行时长:  9.5367431640625e-07
```

```
0
```

元信息丢失

- 自定义装饰器会覆盖掉

```
def logged(func):
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging

@logged
def f(x):
    return x + x * x

# 等价于
def f(x):
    return x + x * x
f = logged(f)
# 发现元信息被装饰器替代了
print(f.__name__)
print(f.__doc__)
```

```
with_logging
None
```

- 利用wraps函数拷贝元信息

```python
from functools import wraps
def logged(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging

@logged
def f(x):
    return x + x * x

print(f.__name__)
print(f.__doc__)
```

```
f
None
```

内置装饰器

@staticmathod、@classmethod、@property

装饰器顺序

```python
@a
@b
@c
def f ():
    print("hello")

# 等价于:
t = a(b(c(f)))
```

偏函数

- 固定函数的某个参数，返回一个新的函数，方便调用

```python
import functools
# 把N进制转成十进制
int("123", base=16)

# 每次都要输入base=16很麻烦，所以
int16 = functools.partial(int, base=16)
# 直接调用
int16("12312")
```

```
74514
```

- 小结：
    1. 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。
    2. 大多数时候用的不好容易画蛇添足，还不如直接调用

总结

1. 常用高阶函数方便某些统一操作，比如：统一修改字符，统一过滤，统一按照某个规则排序
2. 闭包函数主要是为了将全局变量变为非局部变量
3. 匿名函数主要是为了快速定义函数，增加代码可读性，减少代码冗余
4. 装饰器：在不改变原有函数逻辑的情况下新增功能，用法非常多且实用，如：日志操作，单元测试，性能测试等等，需要重点掌握
5. 偏函数主要是两个功能：拷贝元信息；固定参数