高阶函数

一等公民

- 函数在Python是一等公民（First-Class Object）
- 函数也是对象，是可调用对象
- 函数可以作为普通变量，也可以作为函数的参数、返回值

高阶函数

高阶函数（High-order Function）

- 数学概念 y = f(g(x))
- 在数学和计算机科学中，高阶函数应当是至少满足下面一个条件的函数
  - 接受一个或多个函数作为参数
  - 输出一个函数

观察下面的函数定义，回答问题

```
def counter(base):
    def inc(step=1):
        base += step
        return base
    return inc
```

- 请问counter是不是高阶函数?
- 上面代码有没有问题？如果有，如何改?
- 如何调用以完成计数功能?
- f1 = counter(5)和f2=counter(5)，请问f1和f2相等吗?

```
def counter(base):
    def inc(step=1): # 有没有闭包？
        nonlocal base # 形参base也是外部函数counter的local变量
        base += step
        return base
    return inc

c1 = counter(5)
print(c1())
print(c1())
print(c1())

f1 = counter(5)
f2 = counter(5)
print(f1 == f2) # 相等吗？
```

柯里化

- 指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数
- z = f(x, y) 转换成 z = f(x)(y)的形式

例如

```
def add(x, y):
    return x + y
```

原来函数调用为`add(4, 5)`，柯里化目标是`add(4)(5)`。如何实现?

每一次括号说明是函数调用，说明`add(4)(5)`是2次函数调用。

```
add(4)(5)
等价于
t = add(4)
t(5)
```

也就是说add(4)应该返回函数。

```
def add(x):
    def _add(y):
        return x + y
    return _add

add(100, 200)
```

通过嵌套函数就可以把函数转成柯里化函数。

时间模块

datetime模块

datetime类是时间高级类

- 类方法，即使用类调用的方法，由类方法获得一个时间对象
  - now(tz=None) 返回当前时间的datetime对象，时间到微秒，如果tz为None，返回当前时区的不带时区信息的时间
  - utcnow() 不带时区的0时区时间
  - fromtimestamp(timestamp, tz=None) 从一个时间戳返回一个datetime对象
- 时间对象方法
  - timestamp() 返回一个到微秒的时间戳
    - 时间戳：格林威治时间1970年1月1日0点到现在的秒数
  - 构造方法 datetime.datetime(2016, 12, 6, 16, 29, 43, 79043)
  - year、month、day、hour、minute、second、microsecond，取datetime对象的年月日时分秒及微秒
  - weekday() 返回星期的天，周一0，周日6
  - isoweekday() 返回星期的天，周一1，周日7
  - date() 返回日期date对象
  - time() 返回时间time对象

```
import datetime

# 类方法获得时间对象
print(datetime.datetime.now(datetime.timezone(datetime.timedelta(hours=8)))) # 时区时间
print(datetime.datetime.now()) # 无时区时间
print(datetime.datetime.utcnow()) # UTC时间，可以认为是GMT或0时区时间

# 时间戳操作
stamp = datetime.datetime.now().timestamp() # 获得时间戳
print(stamp)
dt = datetime.datetime.fromtimestamp(stamp) # 从时间戳获得时间对象
print(dt)

print(type(dt.date()), dt.date())
print(type(dt.time()), dt.time())
```

Python中时间分为两种：

- naive，没有时区信息的时间，没法明确定位。这个时间表示那个地区的时间，全看程序理解
- aware，包含时区的时间
- 这两种时间不能混合计算

日期与格式化

- 类方法 strptime(date_string, format)，返回datetime对象（时间字符串+格式化字符串 => 时间对象）
- 对象方法 strftime(format)，返回字符串（时间对象通过格式字符串 => 时间字符串）
- 字符串format函数格式化（时间对象通过格式字符串 => 时间字符串）

```
import datetime

datestr = '2018-01-10 17:16:08'
dt = datetime.datetime.strptime(datestr, '%Y-%m-%d %H:%M:%S') #由字符串到时间对象
print(type(dt), dt)
print(dt.strftime('%Y/%m/%d-%H:%M:%S'))  # 输出为字符串
print("{:%Y/%m/%d %H:%M:%S}".format(dt)) # 输出为字符串
```

timedelta类

- datetime2 = datetime1 + timedelta
- datetime2 = datetime1 - timedelta
- timedelta = datetime1 - datetime2
- 构造方法
  - datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)
  - year = datetime.timedelta(days=365)
- timedelta对象有方法total_seconds()， 返回时间差的总秒数

time模块

- time.sleep(secs) 将调用线程挂起指定的秒数

装饰器

由来

需求：为一个加法函数增加记录实参的功能

```
def add(x, y):
    print('add called. x={}, y={}'.format(x, y)) # 增加的记录功能
    return x + y

add(4, 5)
```

上面的代码满足了需求，但有缺点：
记录信息的功能，可以是一个单独的功能。显然和add函数耦合太紧密。加法函数属于业务功能，输出信息属于非功能代码，不该放在add函数中

1、提供一个函数logger完成记录功能

```
def add(x, y):
    return x + y

def logger(fn):
    print('调用前增强')
    ret = fn(4, 5)
    print('调用后增强')
    return ret

print(logger(add))
```


2、改进传参

```
def add(x, y):
    return x + y

def logger(fn, *args, **kwargs):
    print('调用前增强')
    ret = fn(*args, **kwargs) # 参数解构
    print('调用后增强')
    return ret

print(logger(add, 4, 5))
```

3、柯里化

```
def add(x, y):
    return x + y

def logger(fn):
    def wrapper(*args, **kwargs):
        print('调用前增强')
        ret = fn(*args, **kwargs) # 参数解构
        print('调用后增强')
        return ret
    return wrapper
```

调用

```
print(logger(add)(4, 5))
```

或者

```
inner = logger(add)
x = inner(4, 5)
print(x)
```

再进一步

```
def add(x, y):
    return x + y

def logger(fn):
    def wrapper(*args, **kwargs):
        print('调用前增强')
        ret = fn(*args, **kwargs) # 参数解构
        print('调用后增强')
        return ret
    return wrapper

add = logger(add)
print(add(100, 200))
```

4、装饰器语法

```
def logger(fn):
    def wrapper(*args, **kwargs):
        print('调用前增强')
        ret = fn(*args, **kwargs) # 参数解构
        print('调用后增强')
        return ret
    return wrapper

@logger # 等价于 add = wrapper <=> add = logger(add)
def add(x, y):
    return x + y

print(add(100, 200))
```

@logger就是装饰器语法

等价式非常重要，如果你不能理解装饰器，开始的时候一定要把等价式写在后面

无参装饰器

- 上例的装饰器语法，称为无参装饰器
- @符号后是一个函数
- 虽然是**无参装饰器**，但是@后的函数本质上是**单参函数**
- 上例的logger函数是一个高阶函数

日志记录装饰器实现

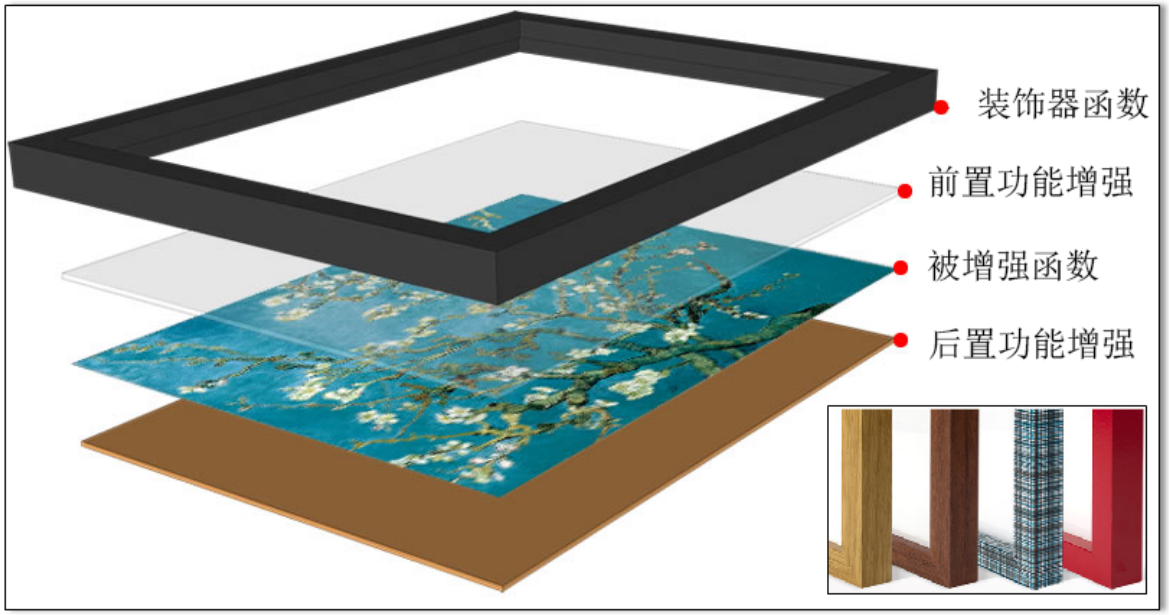
```
import time
import datetime

def logger(fn):
    def wrapper(*args, **kwargs):
        print('调用前增强')
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs) # 参数解构
        print('调用后增强')
        delta = (datetime.datetime.now() - start).total_seconds()
        print('Function {} took {}s.'.format(fn.__name__, delta))
        return ret
    return wrapper

@logger # 等价于 add = wrapper <=> add = logger(add)
def add(x, y):
    time.sleep(2)
    return x + y

print(add(100, 200))
```


装饰器本质



如何类比上图和装饰器呢?

文档字符串

- Python文档字符串Documentation Strings
- 在函数（类、模块）语句块的第一行，且习惯是多行的文本，所以多使用三引号
- 文档字符串也算是合法的一条语句
- 惯例是首字母大写，第一行写概述，空一行，第三行写详细描述
- 可以使用特殊属性__doc__访问这个文档

```
def add(x, y):
    """这是加法函数的文档"""
    return x + y

print("{}'s doc = {}".format(add.__name__ , add.__doc__))
```

```
import time
import datetime

def logger(fn):
    def wrapper(*args, **kwargs):
        "wrapper's doc"
        print('调用前增强')
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs) # 参数解构
        print('调用后增强')
        delta = (datetime.datetime.now() - start).total_seconds()
        print('Function {} took {}s.'.format(fn.__name__, delta))
        return ret
    return wrapper

@logger # 等价于 add = wrapper <=> add = logger(add)
def add(x, y):
    """add's doc"""
    time.sleep(0.1)
    return x + y

print("name={}, doc={}".format(add.__name__ , add.__doc__))
```

被装饰后，函数名和文档都不对了。如何解决?

functools模块提供了一个wraps装饰器函数，本质上调用的是update_wrapper，它就是一个属性复制函数。

wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)

- wrapped就是被包装函数
- wrapper就是包装函数
- 用被包装函数的属性覆盖包装函数的同名属性
- 元组WRAPPER_ASSIGNMENTS中是要被覆盖的属性
    - `__module__`, `__name__`, `__qualname__`, `__doc__`, `__annotations__`

- 模块名、名称、限定名、文档、参数注解

```
import time
import datetime
from functools import wraps


def logger(fn):
    @wraps(fn) # 用被包装函数fn的属性覆盖包装函数wrapper的同名属性
    def wrapper(*args, **kwargs): # wrapper = wraps(fn)(wrapper)
        "wrapper's doc"
        print('调用前增强')
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs) # 参数解构
        print('调用后增强')
        delta = (datetime.datetime.now() - start).total_seconds()
        print('Function {} took {}s.'.format(fn.__name__, delta))
        return ret
    return wrapper

@logger # 等价于 add = wrapper <=> add = logger(add)
def add(x, y):
    """add's doc"""
    time.sleep(0.1)
    return x + y

print("name={}, doc={}".format(add.__name__ , add.__doc__))
```

带参装饰器

- @之后不是一个单独的标识符，是一个函数调用
- 函数调用的返回值又是一个函数，此函数是一个无参装饰器
- 带参装饰器，可以有任意个参数
    - @func()
    - @func(1)
    - @func(1, 2)

进阶

```
import datetime
from functools import wraps


def logger(fn):
    @wraps(fn) # 用被包装函数fn的属性覆盖包装函数wrapper的同名属性
    def wrapper(*args, **kwargs): # wrapper = wraps(fn)(wrapper)
        "wrapper's doc"
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs) # 参数解构
        delta = (datetime.datetime.now() - start).total_seconds()
        print('Function {} took {}s.'.format(fn.__name__, delta))
        return ret
    return wrapper
```

```
@logger # 等价于 add = wrapper <=> add = logger(add)
def add(x, y):
    """add function"""

@logger
def sub(x, y):
    """sub function"""

print(add.__name__, sub.__name__)
```

- logger什么时候执行?
- logger执行过几次?
- wraps装饰器执行过几次?
- wrapper的`__name__`等属性被覆盖过几次?
- `add.__name__`打印什么名称?
- `sub.__name__`打印什么名称?

匿名函数

Python中，匿名函数也叫lambda表达式。

匿名：隐藏名字，即没有名称

匿名函数：没有名字的函数。

函数没有名字该如何定义？函数没有名字如何调用?

Lambda表达式

Python中，使用Lambda表达式构建匿名函数。

```
def foo(x):
    return x ** 2

lambda x: x ** 2 # 定义
(lambda x: x ** 2)(4) # 调用

foo = lambda x,y: (x+y) ** 2 # 定义函数
foo(1, 2)
# 等价于
def foo(x,y):
    return (x+y) ** 2
```

- 使用lambda关键字定义匿名函数，格式为`lambda [参数列表]: 表达式`
- 参数列表不需要小括号。无参就不写参数
- 冒号用来分割参数列表和表达式部分
- 不需要使用return。表达式的值，就是匿名函数的返回值。表达式中不能出现等号
- Python的lambda表达式（匿名函数）**只能写在一行上**，也称为单行函数

匿名函数往往用在为高阶函数传参时，使用lambda表达式，往往能简化代码

```
# 返回常量的函数
print((lambda :0)())
print((lambda x:0)())
```

```
# 加法匿名函数，带缺省值
print((lambda x, y=3: x + y)(5))
print((lambda x, y=3: x + y)(5, 6))
# keyword-only参数
print((lambda x, *, y=30: x + y)(5))
print((lambda x, *, y=30: x + y)(5, y=10))

# 可变参数
print((lambda *args: (x for x in args))(*range(5)))          # 生成器
print((lambda *args: [x+1 for x in args])(*range(5)))        # 列表
print((lambda *args: {x%2 for x in args})(*range(5)))        # 集合
print((lambda *args: {str(x):x for x in args})(*range(5)))   # 字典

print(dict(map(lambda x: (chr(65+x), 10-x), range(5))))  #  高阶函数，构建字典

d = dict(map(lambda x: (chr(65+x), 10-x), range(5)))     # 高阶函数
sorted(d.items(), key=lambda x:x[1])
```

内建函数

排序sorted

定义`sorted(iterable, *, key=None, reverse=False) ->list`

```
sorted(lst, key=lambda x:6-x) # 返回新列表
list.sort(key=lambda x: 6-x) # 就地修改

sorted([1, '2', 3], key=lambda x: str(x))
sorted([1, '2', 3], key=str)
```

过滤filter

- 定义`filter(function, iterable)`
- 对可迭代对象进行遍历，返回一个迭代器
- function参数是一个参数的函数，且返回值应当是bool类型，或其返回值等效布尔值。
- function参数如果是None，可迭代对象的每一个元素自身等效布尔值

```
list(filter(lambda x: x%3==0, [1,9,55,150,-3,78,28,123]))
list(filter(None, range(5)))
list(filter(None, range(-5, 5)))
```

映射map

- 定义`map(function, *iterables) -> map object`
- 对多个可迭代对象的元素，按照指定的函数进行映射
- 返回一个迭代器

```
list(map(lambda x: 2*x+1, range(10)))
dict(map(lambda x: (x%5, x), range(500)))
dict(map(lambda x,y: (x,y), 'abcde', range(10)))
```

拉链函数zip

- zip(*iterables)
- 像拉链一样，把多个可迭代对象合并在一起，返回一个迭代器
- 将每次从不同对象中取到的元素合并成一个元组

```
list(zip(range(10),range(10)))
list(zip(range(10),range(10),range(5),range(10)))

dict(zip(range(10),range(10)))
{str(x):y for x,y in zip(range(10),range(10))}
```