14-装饰器与闭包（Decorators and Closures）

大纲：

- 装饰器的基础知识
- Python何时执行装饰器
- 注册装饰器
- 变量作用域规则
- 闭包
- nonlocal声明
- 实现一个简单的装饰器
- 标准库中的装饰器
- 参数化装饰器
- 习题

装饰器的基础知识

装饰器是一个可调用的对象，其参数是另一个函数（被装饰的函数）。装饰器可能会处理被装饰的函数，然后将其返回，或者将其替换为另一个函数或可调用对象。

In [6]:
```
# 定义一个装饰器
def deco(func):
    def inner():
        print('running inner()')

    # 使用内部函数替换被装饰的函数
    return inner
```

In [5]:
```
# 使用装饰器装饰另一个函数
@deco
def target():
    print('running target')
target()
```

```
running inner()
```

In [4]:
```
# 上面的代码等价于下面
def target():
    print('running target')

target = deco(target)
target()
```

```
running inner()
```

Python何时执行装饰器

装饰器的一个关键性质是，它们在被装饰的函数定义之后立即运行。这通常是在导入时（例如，当Python 加载模块时）

In [9]:
```
# tag::REGISTRATION[]

registry = []  # <1>

def register(func):  # <2>
    print(f'running register({func})')  # <3>
    registry.append(func)  # <4>
    return func  # <5>

@register  # <6>
def f1():
    print('running f1()')

@register
def f2():
    print('running f2()')

def f3():  # <7>
    print('running f3()')

def main():  # <8>
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main()  # <9>

# end::REGISTRATION[]
```

```
running register(<function f1 at 0x00000207AB132A20>)
running register(<function f2 at 0x00000207AB132DE0>)
running main()
registry -> [<function f1 at 0x00000207AB132A20>, <function f2 at 0x00000207AB132DE0>]
running f1()
running f2()
running f3()
```

1. registry列表用来存储被注册的函数
2. register装饰器用来注册函数，参数是被注册的函数
3. 打印被装饰的函数
4. 将func存入registry列表
5. 返回被装饰的函数
6. 使用@register装饰器注册函数f1和f2
7. 没有装饰f3
8. main函数打印registry列表，然后调用f1、f2和f3
9. 只有当前文件被执行时，才会调用main()函数

注册装饰器

考虑到装饰器在真实代码中的常用方式，示例有两处不寻常的地方。

- 示例中装饰器函数与被装饰的函数在同一个模块中定义。实际情况是，装饰器通常在一个模块中定义，然后再应用到其他模块中的函数上。
- register 装饰器返回的函数与通过参数传入的函数相同。实际上，大多数装饰器会在内部定义一个函数，然后将其返回。

变量作用域规则

In [ ]:
```
def f1(a):
    print(a)
    print(b)
```

In [ ]:
```
# 定义全局变量b
b = 6

def f2(a):
    # 打印局部变量a
    print(a)
    # 这里会发生什么？
    print(b)
    # 定义局部变量b
    b=9

f2(3)
```

In [2]:
```
# 定义全局变量b
b = 6

def f3(a):
    # 打印局部变量a
    print(a)

    global b
    print(b)
    # 给全局变量赋值
    b=9

f3(3)
# 打印全局变量b
print(b)
```

```
3
6
9
```

In [3]:
```
# dis模块可以反汇编python函数得到字节码
def f1(a):
    print(a)
    print(b)

from dis import dis
dis(f1)
```

```
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (a)
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_GLOBAL              0 (print)
             10 LOAD_GLOBAL              1 (b)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

In [4]:
```
# 定义全局变量b
b = 6
```

```
def f2(a):
    # 打印局部变量a
    print(a)
    # 这里会发生什么？
    print(b)
    # 定义局部变量b
    b=9

from dis import dis
dis(f2)
```

```
  6           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (a)
              4 CALL_FUNCTION            1
              6 POP_TOP

  8           8 LOAD_GLOBAL              0 (print)
             10 LOAD_FAST                1 (b)
             12 CALL_FUNCTION            1
             14 POP_TOP

 10          16 LOAD_CONST               1 (9)
             18 STORE_FAST               1 (b)
             20 LOAD_CONST               0 (None)
             22 RETURN_VALUE
```

闭包

In [5]:

```
# 一个计算累加值的类
class Averager:

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)

avg = Averager()

print(avg(10))
print(avg(11))
print(avg(12))
```

```
10.0
10.5
11.0
```

In [6]:

```
# 计算累加值的函数式实现

def make_averager():
    series = []

    def averager(new_value):
        # series = [1, 2, 3]
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager

avg2 = make_averager()
print(avg2(10))
print(avg2(11))
```

```
print(avg2(11))
print(avg2(12))
```

```
10.0
10.5
11.0
```

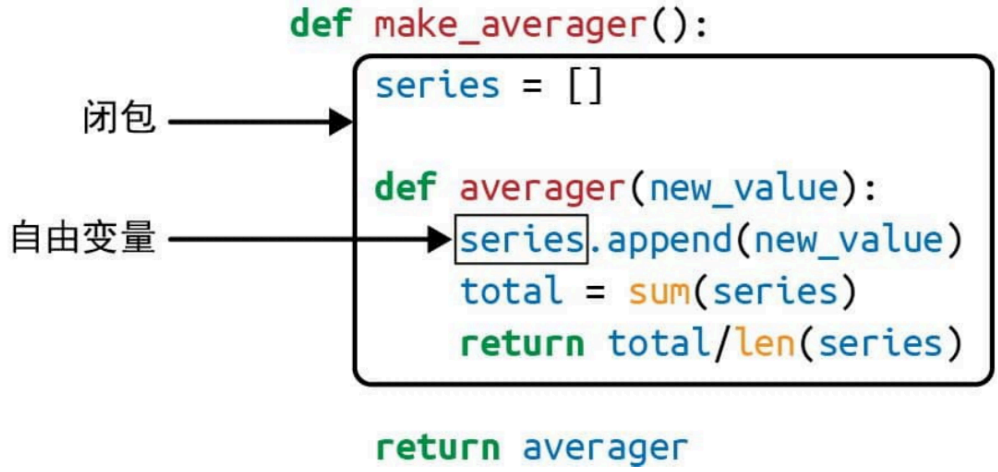

图 9-1：`averager` 函数的闭包延伸到自身的作用域之外，包含自由变量 `series` 的绑定

In [8]:
```
# 查看局部变量
avg2.__code__.co_varnames
```

Out[8]: ('new_value', 'total')

In [10]:
```
# 查看自由变量
avg2.__code__.co_freevars
```

Out[10]: ('series',)

In [13]:
```
# 查看闭包对象
print(avg2.__closure__)

# 查看闭包对象保存的数据
print(avg2.__closure__[0].cell_contents)
```

```
(<cell at 0x000002A70D6EDA50: list object at 0x000002A70DA34EC0>,)
[10, 11, 12]
```

nonlocal声明

前面实现 make_averager 函数的方法效率不高。我们把所有值存储在历史数列中，然后在每次调用 averager 时使用 sum 求和。更好的实现方式是，只存储目前的总值和项数，根据这两个数计算平均值。

In [ ]:
```
def make_averager2():
    count = 0
    total = 0
    def averager(new_value):
        # 因为对counter赋值，counter被当作局部变量
        count += 1
        total += new_value
        return total / count
    return averager
```

```
avg3 = make_averager2()
avg3(10)
```

In [18]:

```
def make_averager3():
    count = 0
    total = 0
    def averager(new_value):
        # nonlocal把变量标记为自由变量
        nonlocal count, total
        count += 1
        total += new_value
        return total / count
    return averager

avg4 = make_averager3()
avg4(10)
```

Out[18]: 10.0

变量查找逻辑：

- 如果是 global x 声明，则 x 来自模块全局作用域，并赋予那个作用域中 x 的值。
- 如果是 nonlocal x 声明，则 x 来自最近一个定义它的外层函数，并赋予那个函数中局部变量 x 的值。
- 如果 x 是参数，或者在函数主体中赋了值，那么 x 就是局部变量。
- 如果引用了 x，但是没有赋值也不是参数，则遵循以下规则。
  - 在外层函数主体的局部作用域（非局部作用域）内查找 x。
  - 如果在外层作用域内未找到，则从模块全局作用域内读取。
  - 如果在模块全局作用域内未找到，则从 **builtins.dict** 中读取。

实现一个简单的装饰器

一个会显示函数运行时间的简单的装饰器

In [1]:

```
import time

def clock(func):
    def clocked(*args):  # <1>
        t0 = time.perf_counter()
        result = func(*args)  # <2>
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked  # <3>
```

1. 定义内部函数 clocked，它接受任意个位置参数。
2. 这行代码行之有效，因为 clocked 的闭包中包含自由变量 func。
3. 返回内部函数，取代被装饰的函数。

In [3]:

```
@clock
def snooze(seconds):
    time.sleep(seconds)

print('*' * 20, 'Calling snooze(.123)')
```

```
snooze(.123)
```

```
******************** Calling snooze(.123)
[0.12324730s] snooze(0.123) -> None
```

In [4]:

```
@clock
def factorial(n):
    return 1 if n<2 else n*factorial(n-1)

print('*' * 20, 'Calling factorial(6)')
factorial(6)
```

```
******************** Calling factorial(6)
[0.00000030s] factorial(1) -> 1
[0.00001160s] factorial(2) -> 2
[0.00001570s] factorial(3) -> 6
[0.00001910s] factorial(4) -> 24
[0.00002240s] factorial(5) -> 120
[0.00002610s] factorial(6) -> 720
```

Out[4]: 720

In [5]:

```
# 我们实际调用的是clock函数的内部函数clocked
factorial.__name__
```

Out[5]: 'clocked'

clocked函数具体实现：

1. 记录初始执行时间t0
2. 调用被装饰的函数factorial,保存结果
3. 计算运行时间
4. 格式化打印运行时间以及函数名
5. 返回第2步保存的结果

上面实现的clock装饰器有几个缺点：

- 不支持关键字参数
- 遮盖了被装饰函数的 __name__ 和 __doc__ 属性

使用 functools.wraps 装饰器把相关的属性从func身上复制到了clocked,还能正确处理关键字参数。

In [6]:

```
import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):  # <1>
        t0 = time.perf_counter()
        result = func(*args, **kwargs)  # <2>
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = [repr(arg) for arg in args]
        arg_lst.extend(f'{k}={v!r}' for k,v in kwargs.items())
        arg_str = ', '.join(arg_lst)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked  # <3>
```

In [7]:
```
@clock
def factorial(n):
    return 1 if n<2 else n*factorial(n-1)

print('*' * 20, 'Calling factorial(6)')
factorial(6)
```

```
******************** Calling factorial(6)
[0.00000060s] factorial(1) -> 1
[0.00002450s] factorial(2) -> 2
[0.00003230s] factorial(3) -> 6
[0.00003960s] factorial(4) -> 24
[0.00004570s] factorial(5) -> 120
[0.00005260s] factorial(6) -> 720
```

Out[7]: 720

In [8]:
```
factorial.__name__
```

Out[8]: 'factorial'

标准库中的装饰器

In [9]:
```
@clock
def fibonacci(n):
    if n<2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

print(fibonacci(6))
```

```
[0.00000070s] fibonacci(0) -> 0
[0.00000040s] fibonacci(1) -> 1
[0.00026790s] fibonacci(2) -> 1
[0.00000030s] fibonacci(1) -> 1
[0.00000040s] fibonacci(0) -> 0
[0.00000030s] fibonacci(1) -> 1
[0.00001520s] fibonacci(2) -> 1
[0.00002920s] fibonacci(3) -> 2
[0.00031260s] fibonacci(4) -> 3
[0.00000020s] fibonacci(1) -> 1
[0.00000020s] fibonacci(0) -> 0
[0.00000020s] fibonacci(1) -> 1
[0.00000620s] fibonacci(2) -> 1
[0.00001260s] fibonacci(3) -> 2
[0.00000020s] fibonacci(0) -> 0
[0.00000020s] fibonacci(1) -> 1
[0.00000630s] fibonacci(2) -> 1
[0.00000030s] fibonacci(1) -> 1
[0.00000040s] fibonacci(0) -> 0
[0.00000020s] fibonacci(1) -> 1
[0.00001000s] fibonacci(2) -> 1
[0.00001970s] fibonacci(3) -> 2
[0.00003660s] fibonacci(4) -> 3
[0.00005630s] fibonacci(5) -> 5
[0.00037970s] fibonacci(6) -> 8
8
```

In [10]:
```
# 利用缓存速度更快

import functools

@functools.cache
```

```
@clock
def fibonacci(n):
    if n<2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

print(fibonacci(6))
```

```
[0.00000050s] fibonacci(0) -> 0
[0.00000060s] fibonacci(1) -> 1
[0.00029590s] fibonacci(2) -> 1
[0.00000060s] fibonacci(3) -> 2
[0.00030950s] fibonacci(4) -> 3
[0.00000050s] fibonacci(5) -> 5
[0.00032290s] fibonacci(6) -> 8
8
```

- 如果缓存的数据多，functools.cache有可能耗尽所有可用内存，@cache更适合短期运行的脚本使用。
- 长期运行的进程，推荐使用functools_lru_cache,并合理设置maxsize参数

In [5]:
```
import functools
# lru_cache默认缓存大小128
@functools.lru_cache
def fibonacci(n):
    if n<2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

fibonacci(6)
```

Out[5]: 8

In [ ]:
```
# 为了得到最佳性能，应该将maxsize设为2的整数次幂
@functools.lru_cache(maxsize=2**20, typed=True)
def fibonacci(n):
    if n<2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)
```

typed参数默认为false，决定是否把不同参数类型得到的结果分开保存。

- 例如，在默认设置下，被认为是值相等的浮点数参数和整数参数只存储一次，即 f(1) 调用和 f(1.0) 调用只对应一个缓存条目。
- 如果设为 typed=True，f(1) 调用和 f(1.0) 调用将分别存储不同的缓存条目。

参数化装饰器

In [13]:
```
registry = set()  # <1>

def register(active=True):  # <2>
    def decorate(func):  # <3>
        print('running register'
              f'(active={active})->decorate({func})')
        if active:   # <4>
            registry.add(func)
        else:
            registry.discard(func)  # <5>
```

```
        return func  # <6>
    return decorate  # <7>

@register(active=False)  # <8>
def f1():
    print('running f1()')

@register()  # <9>
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

```
running register(active=False)->decorate(<function f1 at 0x000001DD4F8F9E10>)
running register(active=True)->decorate(<function f2 at 0x000001DD4F5F3880>)
```

In [14]:
```
registry
```

Out[14]:
```
{<function __main__.f2()>}
```

1. registry现在是一个set对象，这样添加和删除元素更快。
2. register装饰器现在接受一个可选的active参数，控制被装饰的函数是否被注册。默认值为True。
3. 内部函数decorate的参数是一个函数。
4. 只有active参数的值为True时，才会将函数添加到registry中。
5. 如果active参数的值为False，而且func在registry中，那么将它删除。
6. decorate返回一个函数。
7. register返回内部函数decorate。
8. register后面有圆括号，是函数调用的形式，并传入了参数active=False.
9. 没有参数时，register后面也有圆括号。

In [15]:
```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT):  # <1>
    def decorate(func):      # <2>
        def clocked(*_args): # <3>
            t0 = time.perf_counter()
            _result = func(*_args)  # <4>
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)  # <5>
            result = repr(_result)  # <6>
            print(fmt.format(**locals()))  # <7>
            return _result  # <8>
        return clocked  # <9>
    return decorate  # <10>


@clock()  # <11>
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

```
[0.13447570s] snooze(0.123) -> None
[0.13551430s] snooze(0.123) -> None
[0.13655760s] snooze(0.123) -> None
```

In [16]:
```
@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

```
snooze(0.123) dt=0.136s
snooze(0.123) dt=0.136s
snooze(0.123) dt=0.137s
```

缩短数值的过滤器(Number Shortening Filter)

难度：6kyu

在这个kata中，我们将创建一个函数，它返回另一个缩短长数字的函数。给定一个初始值数组替换给定基数的 X 次方。如果返回函数的输入不是数字字符串，则应将输入本身作为字符串返回。

例子：

```
# shorten_number接受的输入是一个后缀列表，和一个基数，返回一个函数
filter1 = shorten_number(['','k','m'],1000)

# filter是一个函数，它接受一个数字字符串并返回一个数字字符串
filter1('234324') == '234k'
filter1('98234324') == '98m'
filter1([1,2,3]) == '[1,2,3]'

filter2 = shorten_number(['B','KB','MB','GB'],1024)
filter2('32') == '32B'
filter2('2100') == '2KB'
filter2('pippi') == 'pippi'
```

代码提交地址： https://www.codewars.com/kata/56b4af8ac6167012ec00006f

按照下面的模式来编写自己的高阶函数：

- 定义一个外部的函数
- 在外部函数内部定义一个内部的函数
- 外部函数最后返回内部定义的函数

In [1]:
```
# 定义的外部函数
def shorten_number(suffixes, base):

    # 定义一个内部函数,真正实现数据格式转换的逻辑
    def my_filter(number):
        print(suffixes)
        print(base)
        # 在函数内部可以使用外部的变量suffixes，base
        return number

    # 返回值是一个函数
    return my_filter

my_fun = shorten_number(['','k','m'],1000)
my_fun('234234')
```

```
['', 'k', 'm']
```

1000

Out[1]: '234234'

In [2]:
```
def shorten_number(suffixes, base):

    # 定义一个函数
    def my_filter(data):
        try:
            # 将函数输入转换为整数
            number = int(data)

        # 如果输入的数据不能转换为整数，直接转换为str返回
        except (TypeError, ValueError):
            return str(data)

        # 输入的number可以转换为整数
        else:
            # i用来跟踪suffixes列表的索引
            i = 0

            # 每次循环将输入的数字除以base，索引i+1
            # 如果除以base等于0或者索引等于len(suffixes)-1，结束循环
            while number//base > 0 and i < len(suffixes)-1:
                number //= base
                i += 1
            return str(number) + suffixes[i]

    # 返回值是一个函数
    return my_filter

filter1 = shorten_number(['','k','m'],1000)
print(filter1('234324'))  # == '234k'
print(filter1('98234324')) # == '98m'
print(filter1([1,2,3])) # == '[1,2,3]'

filter2 = shorten_number(['B','KB','MB','GB', 'TB'],1024)
print(filter2('32')) # == '32B'
print(filter2('2100'))  # == '2KB';
print(filter2('2100000000000000000000'))  # == '2KB';
print(filter2('pippi')) # == 'pippi'
```

234k
98m
[1, 2, 3]