闭包

作用域

- **内置名称（built-in names）**， Python 语言内置的名称，比如函数名 abs、char 和异常名称 BaseException、Exception 等等。
- **全局名称（global names）**，模块中定义的名称，记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。
- **局部名称（local names）**，函数中定义的名称，记录了函数的变量，包括函数的参数和局部定义的变量。（类中定义的也是）

作用域是程序运行时变量可被访问的范围，定义在函数内的变量是局部变量，局部变量的作用范围只能是函数内部范围内，它不能在函数外引用。

```
def foo():
    num = 10  # 局部变量


print(num)  # NameError: name 'num' is not defined
```

定义在模块最外层的变量是全局变量，它是全局范围内可见的，当然在函数里面也可以读取到全局变量的。例如：

```
num = 10  # 全局变量


def foo():
    print(num)  # 10
```

嵌套函数

函数不仅可以定义在模块的最外层，还可以定义在另外一个函数的内部，像这种定义在函数里面的函数称之为**嵌套函数**（nested function）例如：

```
def print_msg():
    # print_msg 是外围函数
    msg = 10

    def printer():
        # printer是嵌套函数
        print(msg)

    printer()


# 输出
print_msg()
```

对于嵌套函数，它可以访问到其外层作用域中声明的非局部（non-local）变量，比如代码示例中的变量 `msg` 可以被嵌套函数 `printer` 正常访问。

那么有没有一种可能即使脱离了函数本身的作用范围，局部变量还可以被访问得到呢？答案是闭包

global 和 nonlocal关键字

当内部作用域想修改外部作用域的变量时，就要用到 global 和 nonlocal 关键字了。

global 用于声明全局变量， nonlocal 用于闭包向上查找变量

什么是闭包

闭包就是能够读取其他函数内部变量的函数，
由于在 Python 语言中，只有函数内部的子函数才能读取局部变量，
因此可以把闭包简单理解成 “定义在一个函数内部的函数”。
所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包的用途：

- 可以在函数外部读取函数内部成员
- 让函数内成员始终存活在内存中

函数身为第一类对象，它可以作为函数的返回值返回，现在我们来考虑如下的例子：

```
def foo():
    """闭包有自己的环境"""
    num = random.randint(1, 10)

    def printer():
        return num

    return printer


# 把函数当做变量使用
pri = foo()
print('pri', pri())
print('pri', pri())
```

不同的地方在于内部函数 `printer` 直接作为返回值返回了。

一般情况下，函数中的局部变量仅在函数的执行期间可用，一旦 `print_msg()` 执行过后，我们会认为 `msg` 变量将不再可用。然而，在这里我们发现 print_msg 执行完之后，在调用 foo 的时候 num 变量的值就返回了，这就是闭包的作用，闭包使得局部变量在函数外被访问成为可能。

看完这个例子，我们再来定义闭包，维基百科上的解释是:

> 在计算机科学中，闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。

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。

装饰器（Decorator）

今天来说说 Python 里的装饰器 (decorator)。它不难，但却几乎是 “精通” Python 的路上的第一道关卡。让我们来看看它到底是什么东西，为什么我们需要它。

手写装饰器

现在我们要写一个函数：

```
from time import time, sleep

def add(x, y=10):

    return x + y
```

然后我们想看看运行的结果，于是写了几个 print 语句：

```
print("add(10)",       add(10))
print("add(20, 30)",   add(20, 30))
print("add('a', 'b')", add('a', 'b'))

# Results:
"""
add(10) 20
add(20, 30) 50
add('a', 'b') ab
"""
```

现在我们想看看测试这个函数的性能，于是我们加上这个代码：

```
from time import time

before = time()
print("add(10)",       add(10))
after = time()
print("time taken: {}".format(after - before))

before = time()
print("add(20, 30)",   add(20, 30))
after = time()
print("time taken: {}".format(after - before))

before = time()
print("add('a', 'b')", add('a', 'b'))
after = time()
print("time taken: {}".format(after - before))

# Results
"""
add(10) 20
time taken: 1.0007071495056152
add(20, 30) 50
time taken: 1.0004420280456543
add('a', 'b') ab
time taken: 1.0001146793365479
"""
```

代码马上变得很复杂。但最重要的是，我们得写一堆代码（复制粘贴），程序员是懒惰的，所以我们就想到一些更简单的方法，与其写这么多次，我们可以只写一次代码：

```python
from time import time, sleep
def add(x, y=10):
    before = time()
    result = x + y
    sleep(1)
    after = time()
    print('elapsed: ', after - before)
    return result

print("add(10)",       add(10))
print("add(20, 30)",   add(20, 30))
print("add('a', 'b')", add('a', 'b'))

# Results
"""
elapsed:  1.000178575515747
add(10) 20
elapsed:  1.000016450881958
add(20, 30) 50
elapsed:  1.0001020431518555
add('a', 'b') ab
"""
```

不论是代码的修改量还是代码的美观程度，都比之前的版本要好！

但是，现在我们写了另一个函数：

```python
def sub(x, y=10):
    return x - y
```

我们必须再为 `sub` 函数加上和 `add` 相同的性能测试代码：

```python
def sub(x, y=10):
    before = time()
    result = x - y
    after = time()
    print('elapsed: ', after - before)
    return result
```

作为一个懒惰的程序员，我们立马就发现了，有一个 “模式” 反复出现，即执行一个函数，并计算这个函数的执行时间。于是我们就可以把这个模式抽象出来，用函数：

```python
from time import time

def timer(func, x, y = 10):
    before = time()
    result = func(x, y)
    after = time()
    print("elapsed: ", after - before)
    return result

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

def sub(x, y = 10):
    return x - y
```

```
15
16 print("add(10)", timer(add, 10))
17 print("add(20, 30)", timer(add, 20, 30))
```

但这样还是很麻烦，因为我们得改到所有的测试用例，把 `add(20, 30)` 改成 `timer(add, 20, 30)`。于是我们进一步改进，让 timer 返回函数：

```
1  def timer(func):
2      def wraper(x, y=10):
3          before = time()
4          result = func(x, y)
5          after = time()
6          print("elapsed: ", after - before)
7          return result
8      return wraper
9
10 def add(x, y = 10):
11     return x + y
12
13 add = timer(add)
14
15 def sub(x, y = 10):
16     return x - y
17
18 sub = timer(sub)
19
20 print("add(10)",        add(10))
21 print("add(20, 30)",    add(20, 30))
```

这里的最后一个问题是，我们的 timer 包装的函数可能有不同的参数，于是我们可以进一步用 `*args, **kwargs` 来传递参数：

```
1  def timer(func):
2      def wraper(*args, **kwargs):
3          before = time()
4          result = func(*args, **kwargs)
5          after = time()
6          print("elapsed: ", after - before)
7          return result
8      return wraper
```

这里的 `timer` 函数就是一个“装饰器”，它接受一个函数，并返回一个新的函数。在装饰器的内部，对原函数进行了“包装”。

注：上面的例子取自 What Does it Take to Be an Expert At Python。

@ 语法糖

上一节是一个懒惰的程序员用原生的 Python 写的装饰器，但在装饰器的使用上，用的是这个代码：

```
def add(x, y = 10):
    return x + y
add = timer(add)        # <- notice this

def sub(x, y = 10):
    return x - y
sub = timer(sub)
```

上面这个语句里，我们把 `add` 的名字重复了 3 次，如果函数改了名字，我们就得改 3 处。懒惰的程序员就想了一个更“好”的方法，提供了一个语法来替换上面的内容：

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

这就是我们最常见的装饰器的形式了，这两种写法完全等价，只是 `@` 写法更简洁一些。

带参数的装饰器

我们知道下面两种代码是等价的：

```
@dec
def func(...):
    ...

func = dec(func)
```

我们可以把它当成是纯文本的替换，于是可以是这样的：

```
@dec(arg)
def func(...):
    ...

func = dec(arg)(func)
```

这也就是我们看到的“带参数”的装饰器。可见，只要 `dec(arg)` 的返回值满足 “装饰器” 的定义即可。（接受一个函数，并返回一个新的函数）

这里举一个例子（来源）：

```
def use_logging(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if level == "warn":
                logging.warn("%s is running" % func.__name__)
            elif level == "info":
                logging.info("%s is running" % func.__name__)
            return func(*args)
        return wrapper

    return decorator

@use_logging(level="warn")
def foo(name='foo'):
```

```
    print("i am %s" % name)
```

先不管 `use_logging` 长什么样，先关心它的返回值 `decorator`，看到 `decorator` 本身是一个函数，并且参数是函数，返回值是函数，于是确认 `decorator` 是一个 “装饰器”。于是上面这种“带参数的装饰器”的作用也就很直接了。

类作为装饰器

如果说 Python 里一切都是对象的话，那函数怎么表示成对象呢？其实只需要一个类实现 `__call__` 方法即可。

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

    def __call__(self, *args, **kwargs):
        before = time()

        result = self._func(*args, **kwargs)

        after = time()
        print("elapsed: ", after - before)

        return result

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

也就是说把类的构造函数当成了一个装饰器，它接受一个函数作为参数，并返回了一个对象，而由于对象实现了 `__call__` 方法，因此返回的对象相当于返回了一个函数。因此该类的构造函数就是一个装饰器。

小结

装饰器中还有一些其它的话题，例如装饰器中元信息的丢失，如何在类及类的方法上使用装饰器等。但本文里我们主要目的是简单介绍装饰器的原因及一般的使用方法，能用上的地方就大胆地用上吧！