闭包、装饰器

笔记本：	Python提高-2		
创建时间：	2018/5/9 15:10	**更新时间：**	2018/5/28 8:23
作者：	ly		

PyCharm快捷键
快捷键Ctrl + AIt + L让选中的代码规范化
按住AIt 点击光标可选中多个位置 同时在多个位置删除和增加代码

函数参数

函数在内存中是有空间，函数名记录着这个空间的地址

```
函数名()            # 代表去这边空间执行代码    调用函数
print(函数名)       # 打印这个函数的所在引用地址
```

```
def func1():
    print("111")

func1()      # 调用函数     输出：111
print(func1)   # 查看着函数名所在内存的地址   <function func1 at 0x00000000005C2E18>
```

函数也是能当参数使用的

```
def fun2(func):
    print(func)


func2(func1)    # 把上面的函数fun1当参数func传到fun2函数中，   输出：也是函数func1的内存地址
```

闭包：

- 1，函数嵌套
- 2，外层函数的返回值是内存函数的引用（地址/函数名）
- 3，外层函数需要有参数（内存函数使用）

```
def 外层函数名(形参):
    def 内层函数名():
        执行操作
    return 内存函数名
```

```
def func_out(tate):       # rate  ==> 0.7
    def func_in(money):      # func_in ==>  0x11
        print(reate * money)
    return func_in             # func_in  ==> 0x11
```

```
usa_rate = func_out(0.7)    # usa_rate  ==>  func_in    相当于内层函数
usa_rate(100)
```

```
29  def fun1(rate):
30      def fun2(money):
31          print(rate * money)
32
33      return fun2
34
35  ll = fun1(0.7)    # ll-->>  引用了fun2函数
36  ll(10000)
37  # 一个函数使用到另一函数的局部变量
```

1，执行29行==>定义行fun1,为fun1开辟内存空间
2,，执行35行==>调用函数fun1,把实参0.7传给形参rate
3，执行30行==>定义行数fun2,为fun2开辟空间
4，执行33行==>return返回函
5，ll 接收引用了fun2函数
6，调用ll 等于调用了func2函数

闭包的目的 让一个函数使用到了另一个函数的局部变量　　如:上面的rate

闭包中变量的使用：

```
def func_out(c):
    def func_in(a, b):
        print((a + b)/c)
    return func_in


f1 = func_out(2)
f1(10, 20)    # 与func_in() 括号内的参数对应
```

修改外部函数中的变量：

```
def func_out(func):
   a = 10
   # print(a)
   def func_in():
      nonlocal a    # 声明变量是非局部的， a为当前函数的最近外层(a = 10)的变量
      print(a)        # a = 10
      print(func)     # func = 100
      a = 1000
   return func_in


test = func_out(100)
test()                    输出：10      100
```

装饰器

作用：在不改变原有的代码的前提下 给函数增加新的功能

封闭开放原则：

- 封闭：已经实现功能的代码块 （封闭起来，不能随便修改）
- 开放：对扩展开发 （但是可以对他增加新的功能）

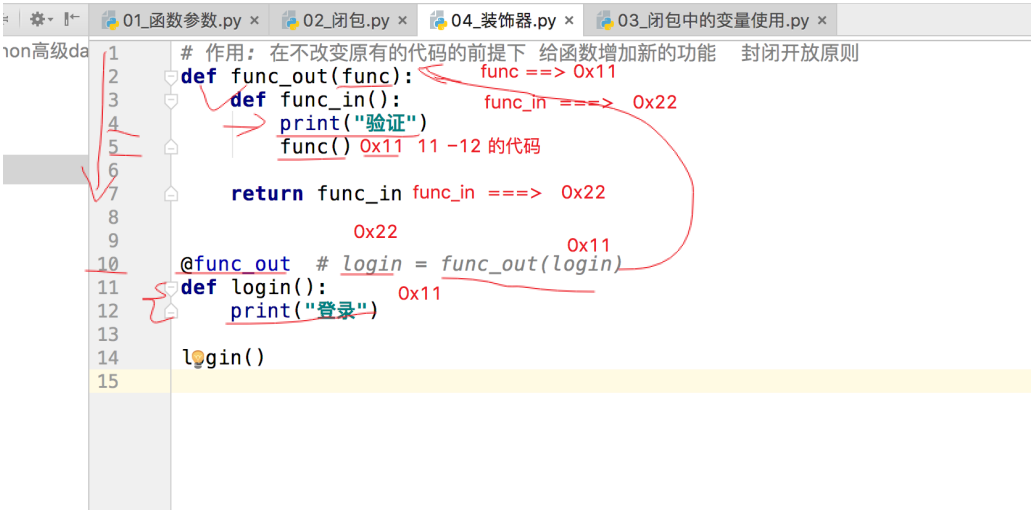
基本语法：

```
def w1(func):
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner


@w1               ==>  f1 = w1(f1) 把装饰器下面的函数当参数传入到w1的形参func中
def f1():
    print('f1')
```

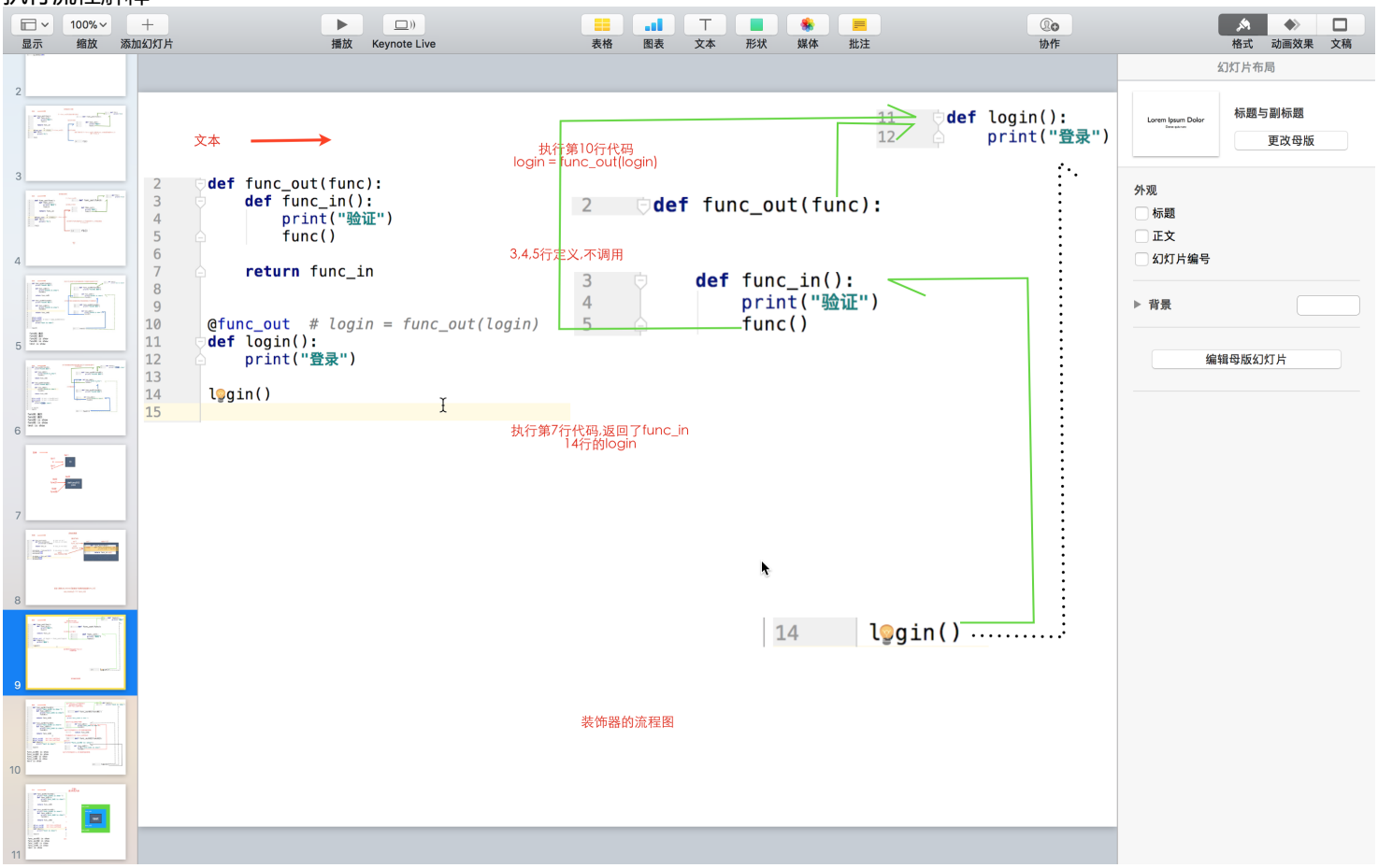
实例：给已经实现登陆功能的函数增加一个验证功能

```
def func_out(func):
    def func_in():
        print("验证")
        func()
    return func_in


@func_out          # login = func_out(login)
def login():
    print("登陆")


login()
# 输出：    验证
          登陆
```

内存地址解释



执行流程解释



从上往下执行：遇到函数先定义 开辟空间

def func_out(func) 定义函数func_out 开辟空间

@func_out 等价于===> login = func_out(login)

执行时先执行等于号右边的代码：

func_out(login) 调用函数把login当参数传入函数fun_out中
def func_in() 定义函数fun_in 为func_in 开辟内存空间
return func_in 返回函数

login　　login引用返回函数func_in的地址

结论

```
login()    直接调用func_in()
print("验证")     打印输出
func()     执行func()实际就是执行login()里面的代码
```

装饰有返回值的函数（return）

```
def func_out(func):
    def func_in():        # func_in 接收了login函数的返回值
        return func()
    return func_in


@func_out         login = func_out(login)
def login():     # 有返回值的函数login
    return 100   # return 哪个函数里面有return 则它会返回后面的值给该函数


ret = login()   # longin引用了func_in
print(ret)      # 输出： 100
```

装饰有参数的函数(login(a, b))

```
def func_out(func):
    def func_in():
        pass
    return func_in


@func_out
def login(a, b):
    print(a + b)
```

装饰有参数的函数_通用版（*args， **args 不定长参数来接收）

```
def func_out(func):
    def func_in(*args, **kwargs):
        return func(*args, **kwargs)
    return func_in


@func_out
def login(*args, **kwargs):
    print（args)
    print(kwargs)
    return 100


ret = login(10, 20, name = "ly", age = 18)
print(ret)
```

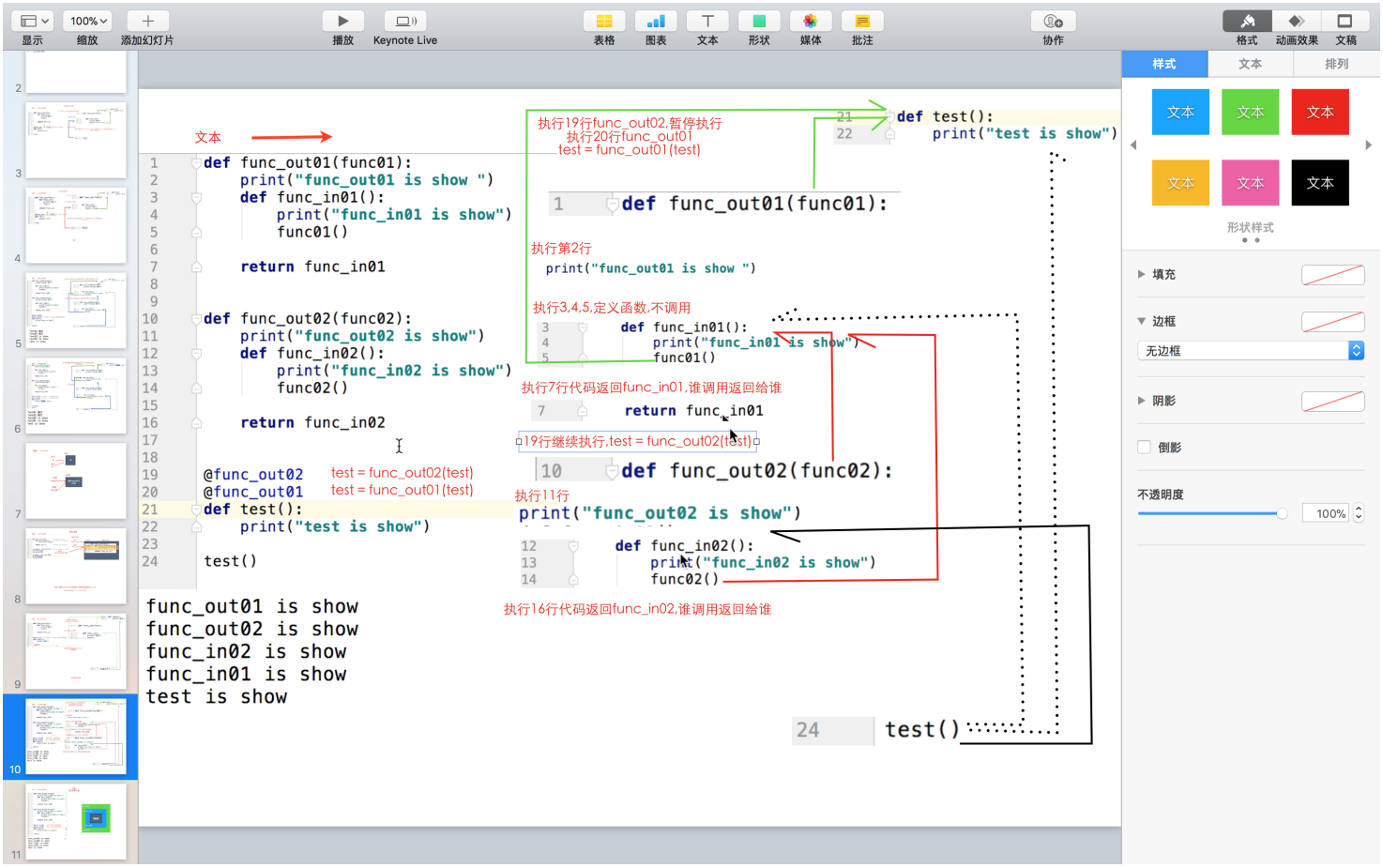
类装饰器（了解）

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

    def __call__(self):
        print("验证")
        self.func()


@Foo    # login = Foo(login)
def login():
    print("登陆")

login()    # 输出：     验证     登陆
```

创建类来当修饰器 并没有节约资源，一般情况不会用

两个装饰器装饰同一个函数

```
def func_out01(func01):
    print("func_out01 is show")
    def func_in01():
        print("func_in01 is show")

    return func_in01


def func_out2(func02):
    print("func_out02 is show")
    def func_in02():
        print("func_in02 is show")

    return func_in02


@func_out02    # 只有当它的下面是个函数的时候才会执行
@func_out01
def test():    # test进行了两次翻新
    print("test is show")


test()

输出：
func_out01 is show
func_out02 is show
func_in02 is show
func_in01 is show
test is show
```


显示
100%
缩放
添加幻灯片
播放
Keynote Live
表格
图表
文本
形状
媒体
批注
协作
格式
动画效果
文稿
样式
文本
排列
文本
文本
文本
文本
文本
文本
形状样式
填充
边框
无边框
阴影
倒影
不透明度
100%
文本
def func_out01(func01):
print("func_out01 is show ")
def func_in01():
print("func_in01 is show")
func01()
return func_in01
def func_out02(func02):
print("func_out02 is show")
def func_in02():
print("func_in02 is show")
func02()
return func_in02
@func_out02 test = func_out02(test)
@func_out01 test = func_out01(test)
def test():
print("test is show")
test()
func_out01 is show
func_out02 is show
func_in02 is show
func_in01 is show
test is show
执行19行func_out02,暂停执行
执行20行func_out01
test = func_out01(test)
执行第2行
print("func_out01 is show ")
执行3,4,5,定义函数,不调用
执行7行代码返回func_in01,谁调用返回给谁
return func_in01
19行继续执行,test = func_out02(test)
执行11行
print("func_out02 is show")
执行16行代码返回func_in02,谁调用返回给谁
24 test()