

# 第七周问答作业

朱士杭 231300027

2024 年 4 月 11 日

## 1 如何使用函数对功能进行抽象，有什么优势及可能存在的问题？

通过一个函数实现某一个功能，当在程序中需要使用该功能的时候，直接调用该函数即可，没有必要再对该功能的代码再写一遍

### 1.1 优势

节省代码量、修改的时候比较方便、提高开发效率、层级之间比较分明对于整体程序框架结构更明了

### 1.2 可能存在的问题

可能会产生副作用、函数不灵活对功能抽象的时候需要很多函数的复合相互调用导致程序更加复杂

## 2 对比 C/C++ 以及 python 对函数嵌套的规定，并简述允许函数嵌套对程序设计的影响

### 2.1 对比

1. C++ 不允许在函数中定义函数，但是 python 允许，并且优先调用 local-frame 中定义的函数
2. C++ 和 python 都允许在一个函数中调用另一个函数，并且在栈中创建一个新的相对独立的空间给这个新函数，使得函数之间会有层次感
3. C++ 中

可以通过函数指针进行传参，python 是将函数名对象作为参数进行传递，相似之处在于可以使用传入进来的函数形参，使得函数使用更加灵活

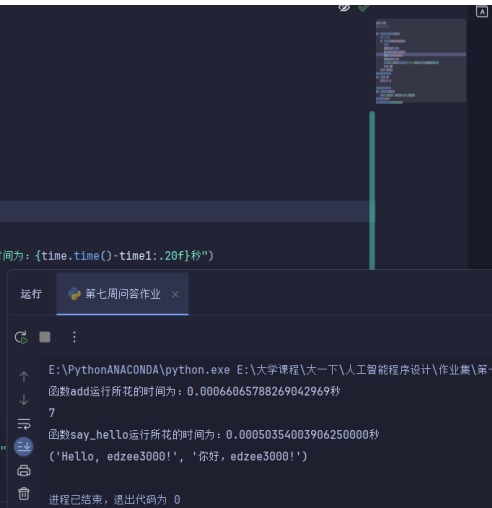
## 2.2 影响

1. 允许函数嵌套使得程序设计更具有结构化
2. 函数嵌套使得程序员开发的时候将大量的时间可以放在逻辑设计上而不是写大量冗杂的代码
3. 使得程序设计更加灵活，不用总是 ctrl-c、ctrl-v 复制粘贴大量相似的代码，避免污染眼睛
3. 在 Python 中，Decorator 是什么？请解释装饰器如何工作，以及如何使用它们来修改或增强已有函数的行为。

## 3 请编写一个用于性能检测的装饰器，可以输出被他装饰的函数的运行时间

python 装饰器是闭包的一种应用，用于拓展原来函数功能的一种函数，这个函数的特殊之处在于它的返回值也是一个函数，使用 python 装饰器的好处就是在不用更改原函数的代码前提下给函数增加新的功能，使用时在需要装饰的函数前一行加上 @ 装饰函数即可编写一个用于性能检测的装饰器代码如下：

```
10 import time
11 #输出被装饰函数的运行时间
12 2 用法
13 def Decoration_func(func):
14     '''装饰器函数'''
15     def wrapper(*args,**kwargs):
16         """
17         """
18         time1=time.time()
19         res=func(*args,**kwargs)
20         time.sleep(0.0000001)
21         time2=time.time()
22         print(f"函数{func.__name__}运行所花的时间为: {time.time()-time1:.20f}秒")
23         return res
24     return wrapper
25 @Decoration_func
26 def add(a, b):
27     return a + b
28 1个用法
29 @Decoration_func
30 def say_hello(name):
31     return f"Hello, {name}!", f"你好, {name}!"
32 print(add(*args: 3,4))
33 print(say_hello("edzee3000"))
34 Decoration_func() -> wrapper()
```



运行 第七周问答作业

E:\Python\ANACONDA\python.exe E:\大学课程\大一下\人工智能程序设计\作业集\第七周问答作业

函数add运行所花的时间为: 0.00066065788269042969秒

7

函数say\_hello运行所花的时间为: 0.00050354003906250000秒

('Hello, edzee3000!', '你好, edzee3000!')

进程已结束,退出代码为 0

图 1: 自定义装饰器来测试函数性能/运行时间

## 4 折扣策略

### 4.1 实现每种折扣策略具体的折扣计算函数

```
2 # 输入订单的数据示例，为字典类型
3 Order_sample = {"is_VIP_customer": True, "cart" :{"Item1": 10, "Item2": 20, "Item3": 2}}
4 #注意商品单价均为1
5 1个用法
6 def fidelity_promo(order):
7     """为VIP顾客提供5%折扣"""
8     if order["is_VIP_customer"]:
9         return sum(order["cart"].values())*0.05
10    else:
11        return 0
12 1个用法
13 def bulkitem_promo(order):
14     """单个商品为20个或以上时为该商品提供10%折扣"""
15     if sum(order["cart"].values()) >= 20:
16         return sum(order["cart"].values())*0.1
17    else:
18        return 0
19 promos = [fidelity_promo, bulkitem_promo]
20 1个用法
21 def best_promo(order):
22     """选择可用的最佳折扣，并返回最大折扣"""
23     max_preferential=0
24     for func in promos:
25         if func(order=order)>max_preferential:
26             max_preferential=func(order=order)
27     return max_preferential
28 print(best_promo(Order_sample))
29
30
31
32
```

图 2: 折扣计算函数

### 4.2 额外的折扣策略

```
47 promos = [fidelity_promo, bulkitem_promo]
48 1个用法
49 def best_promo(order,promos):
50     """选择可用的最佳折扣，并返回最大折扣"""
51     max_preferential=0
52     for func in promos:
53         if func(order=order)>max_preferential:
54             max_preferential=func(order=order)
55     return max_preferential
56 1个用法
57 def large_order_promo(order):
58     """订单中的不同商品达到3个或以上时为多有商品提供7%折扣"""
59     if len(order["cart"].keys()) >= 3:
60         return sum(order["cart"].values()) * 0.07
61 promos.append(large_order_promo)
62 print(best_promo(Order_sample,promos=promos))
63
64
65
66
```

图 3: 额外折扣计算函数

## 4.3 装饰器函数

```
2 # 输入订单的数据示例，为字典类型
3 Order_sample = {"is_VIP_customer": True, "cart" :{"Item1": 10, "Item2": 20, "Item3": 2}}
3 用法
4 def Decoration_func(func):
5     '''装饰器函数'''
6     def wrapper(*args,**kwargs):
7         '''
8         func_name=func.__name__
9         res=func(*args,**kwargs)
10        print(f'在优惠策略{func_name}下，优惠金额为{res}，该顾客是否为VIP: {kwargs["order"]]["is_VIP_customer"]}')
11        return res
12    return wrapper
13 #注意商品单价均为1
14 1个用法
15 @Decoration_func
16 > def fidelity_promo(order):...
17 1个用法
21 @Decoration_func
22 > def bulkitem_promo(order):...
23 1个用法
28 > def best_promo(order,promos):...
29 1个用法
35 @Decoration_func
36 > def large_order_promo(order):...
42 promos = [fidelity_promo, bulkitem_promo]
43 promos.append(large_order_promo)
44 print(best_promo(Order_sample,promos=promos))
45
46
47
```

图 4: 装饰器函数

## 4.4 装饰器函数的作用与适用范围

### 4.4.1 作用

在 Python 中，装饰器的作用主要是为了增加现有代码（函数、方法或类）的行为，而不需要改变代码本身

### 4.4.2 适用范围

它允许程序员以声明性的方式扩展函数的功能，在日志记录、性能测试、事务处理、缓存、权限校验等方面都是适用的。