NSD Python2 DAY02

1. 案例1：简单的加减法数学游戏
2. 案例2：简单GUI程序
3. 案例3：快速排序
4. 案例4：测试程序运行效率

1 案例1：简单的加减法数学游戏

1.1 问题

编写math_game.py脚本，实现以下目标：

1. 随机生成两个100以内的数字
2. 随机选择加法或是减法
3. 总是使用大的数字减去小的数字
4. 如果用户答错三次，程序给出正确答案

1.2 方案

创建4个函数，分别实现返回两数之和、返回两数之差、判断表达式正确性、是否继续计算四种方法：

1.首先调用main()函数（是否继续计算功能），main函数利用循环无限次调用exam()函数进行计算，计算结束，用户选择是否继续（此过程利用try语句捕获索引错误、ctrl+c（中断）错误、ctrl+d错误），如果选择n即结束循环，不再调用exam()函数，否则循环继续

2.调用exam()函数：

a)输出运算公式：利用列表切片将随机生成的两个数打印（这两个数利用random模块及列表生成式随机生成，并利用sort()方法进行降序排序，确保相减时一直是大的数字减小的数字），利用random模块随机生成“+” “-”号，输出在两数之间

b)用户输入值，利用for循环进行三次判断，如果运算公式结果与用户输入值相同，循环结束，系统输出“你赢了”，exam()函数执行结束，否则系统输出“你答错了”，循环继续，3次都回答错误，利用循环的else分支输出运算公式及结果

c)上诉运算公式结果：利用random模块随机生成“+” “-”值对关系调用（其中“+” “-”号作为字典键，返回和函数add()及返回差函数sub()作为值，调用时将随机生成的两个数字作为参数传递给add()函数及sub()函数）

1.3 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写脚本

```
[root@localhost day06]# vim math_game.py
#! /usr/bin/env python3

import random

```

Top

```
06.  def add(x, y):
07.      return x + y
08.
09.  def sub(x, y):
10.      return x - y
11.
12.  def exam():
13.      cmds = {'+': add, '-': sub}  #将函数存入字典
14.      nums = [random.randint(1, 100) for i in range(2)] #生成两个数
15.      nums.sort(reverse=True)  #降序排列
16.      op = random.choice('+-')
17.      result = cmds[op](*nums)  #调用存入字典的函数，把nums列表拆开，作为参数传入
18.      prompt = "%s %s %s = " %(nums[0], op, nums[1])
19.
20.      for i in range(3):
21.          try:
22.              answer = int(input(prompt))
23.          except:
24.              continue
25.
26.          if answer == result:
27.              print('你真棒，答对了！')
28.              break #答对了就不要再回答了，结束循环
29.          else:
30.              print('答错了')
31.      else:
32.          print("%s%s" %(prompt, result))  #只有循环不被break才执行
33.
34.
35.  def main():
36.      while True:
37.          exam()
38.          try:
39.              go_on = input('Continue(y/n)?').strip()[0]
40.          except IndexError:
41.              continue
42.          except (KeyboardInterrupt, EOFError):
43.              go_on = 'n'
44.
45.          if go_on in 'nN':
46.              print('\nBye-bye.')
```

Top

```
47.             break
48.
49.
50.     if __name__ == '__main__':
51.         main()
```

实现此案例还可利用while循环：

```
01.     [root@localhost day06]# vim mygui.py
02.
03.     #! /usr/bin/env python3
04.
05.     import random
06.
07.     def add(x, y):
08.         return x + y
09.
10.     def sub(x, y):
11.         return x - y
12.
13.     def exam():
14.         cmds = {'+': add, '-': sub}  #将函数存入字典
15.         nums = [random.randint(1, 100) for i in range(2)] #生成两个数
16.         nums.sort(reverse=True)  #降序排列
17.         op = random.choice('+-')
18.         result = cmds[op](*nums)  #调用存入字典的函数，把nums列表拆开，作为参数传入
19.         prompt = "%s %s %s = " %(nums[0], op, nums[1])
20.         tries = 0
21.
22.         while tries < 3:
23.             try:
24.                 answer = int(input(prompt))
25.             except:
26.                 continue
27.
28.             if answer == result:
29.                 print('你真棒，答对了！')
30.                 break #答对了就不要再回答了，结束循环
31.             else:
32.                 print('答错了')
```

Top

```
            tries += 1
        else:
            print("%s%s" %(prompt, result))    #只有循环不被break才执行


def main():
    while True:
        exam()
        try:
            go_on = input('Continue(y/n)?').strip()[0]
        except IndexError:
            continue
        except (KeyboardInterrupt, EOFError):
            go_on = 'n'

        if go_on in 'nN':
            print('\nBye-bye.')
            break


if __name__ == '__main__':
    main()
```

步骤二：测试脚本执行

```
[root@localhost day06]# python3 math_game.py
54+19=
54+19=
54+19=73
你真棒，答对了！
Continue(y/n)?y
60+39=99
你真棒，答对了！
Continue(y/n)?y
18+15=33
你真棒，答对了！
Continue(y/n)?y
35+20=55
你真棒，答对了！
Continue(y/n)?y
```

Top

```
16.  37 + 35 = 72
17.  你真棒，答对了！
18.  Continue( y /n) ? y
19.  77 -  57 = 20
20.  你真棒，答对了！
21.  Continue( y /n) ? y
22.  35 + 23 = 5
23.  答错了
24.  35 + 23 = 6
25.  答错了
26.  35 + 23 = 7
27.  答错了
28.  35 + 23 = 58
29.  Continue( y /n) ? y
30.  75 + 47 = 122
31.  你真棒，答对了！
32.  Continue( y /n) ? ^C
33.  Bye- bye.
```

2 案例2：简单GUI程序

2.1 问题

创建mygui.py脚本，要求如下：

1. 窗口程序提供三个按钮
2. 其中两个按钮的前景色均为白色，背景色为蓝色
3. 第三个按钮前景色为红色，背景色为红色
4. 按下第三个按钮后，程序退出

2.2 方案

1.导入tkinter模块、创建顶层窗口，顶层窗口只应该创建一次

2.添加窗口部件：用Label控件创建标签、用Button控件来创建按钮

3.引入偏函数partial把tkinter.Button的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数重复创建按钮会更简单。对于有很多可调用对象，并且许多调用都反复使用相同参数的情况，使用偏函数比较合适。

4.创建第三个按钮需command绑定退出命令

5.最后将按钮及标签填充到界面

6.运行这个GUI应用

2.3 步骤

Top

实现此案例需要按照如下步骤进行。

步骤一：编写脚本

```
[root@localhost day06]# vim mygui.py

#! /usr/bin/env python3

import tkinter
from functools import partial

root = tkinter.Tk()    #创建顶层窗口
lb1 = tkinter.Label(root, text="hello world! ", font = "Aria 16 bold") #创建标签
b1 = tkinter.Button(root, bg='blue', fg='white', text="Button 1") #创建按钮
mybutton = partial(tkinter.Button, root, bg='blue', fg='white')
#调用新的函数时，给出改变的参数即可
b2 = mybutton(text='Button 2')
b3 = tkinter.Button(root, bg='red', fg='red', text='QUIT', command=root.quit)    #创建按
lb1.pack()    #填充到界面
b1.pack()
b2.pack()
b3.pack()
root.mainloop()    #运行这个GUI应用
```

实现此案例还可用以下方法，需要注意的是：

Command绑定闭包函数返回函数块，上传多个参数调用闭包函数时，内层函数利用config方法替换标签内容

```
[root@localhost day06]# vim mygui.py

#! /usr/bin/env python3

import tkinter
from functools import partial

def say_hi(word):
    def welcome():
        lb1.config(text='hello %s! ' %word)
    return welcome

root = tkinter.Tk()
lb1 = tkinter.Label(root, text="hello world! ", font = "Aria 16 bold")
```

Top

```
15.  mybutton = partial(tkinter.Button, root, bg='blue', fg='white')
16.  b1 = mybutton(text='Button 1', command=say_hi('tedu'))
17.  b2 = mybutton(text='Button 2', command=say_hi('达内'))
18.  b3 = tkinter.Button(root, bg='red', fg='red', text='QUIT', command=root.quit)
19.  lb1.pack()
20.  b1.pack()
21.  b2.pack()
22.  b3.pack()
23.  root.mainloop()
```

步骤二：测试脚本执行，结果如图-1、图-2、图-3所示：

```
01.  [root@localhost day05]# python3 mygui.py
```

图-1

```
01.  [root@localhost day06]# python3 mygui2.py
```

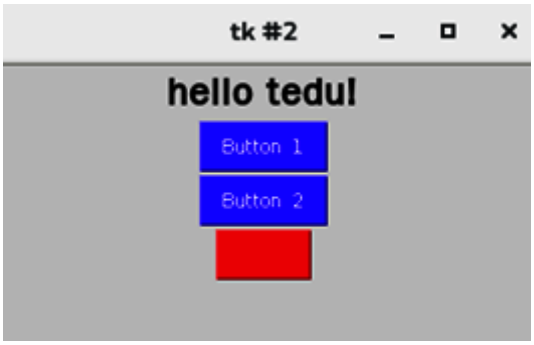

图-2

Top

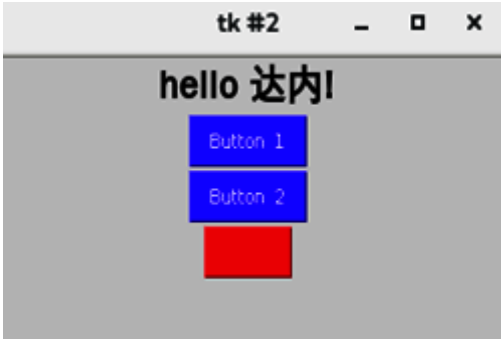

图-3

3 案例3：快速排序

3.1 问题

创建qsort.py文件，实现以下目标：

1. 随机生成10个数字
2. 利用递归，实现快速排序

3.2 方案

将要排序的数据分割成独立的三部分，任意选取一个数据作为关键数据，然后将所有比它小的数都放到它前面，所有比它大的数都放到它后面，这个过程称为一趟快速排序，整个排序过程通过递归进行，以此达到整个数据变成有序序列。

一趟快速排序的算法是：

1.创建两个空列表分别用于存放比关键数小的数据和比关键数大的数据smaller和larger

2.For循环遍历将要排序的数据，将数据与关键数对比，比关键数小的放入smaller列表中，比关键数大的放入larger列表中

3.函数返回值为，以smaller列表为参数调用自身函数、关键数、以larger列表为参数调用自身函数：此时，函数每一次调用都会基于上一次的调用进行，会持续调用自身函数，参数列表数据会越来越少，我们规定，参数列表长度为0或1，递归结束，输出最终数据

4.注意：在调用qsort函数时，根据上传数据类型不同，一定要注意数据类型转化

3.3 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写脚本

```
[root@localhost day05]# vim qsort.py
#! /usr/bin/env python3

from random import randint

def qsort(data):
    data = list(data)
    if len(data) == 0 or len(data) == 1:  # 长度为0或1，直接返回
```

Top

```
09.             return data
10.
11.         middle = data.pop()   # 假设最后一项是中间值
12.         smaller = []
13.         larger = []
14.
15.         for item in data:
16.             if item > middle:   # 比middle大的放到larger，否则放到smaller
17.                 larger.append(item)
18.             else:
19.                 smaller.append(item)
20.
21.         return qsort(smaller) + [middle] + qsort(larger)
22.
23.
24.     if __name__ == '__main__':
25.         nums = [randint(1, 100) for i in range(10)]
26.         print(nums)
27.         print(qsort(nums))
28.         astr = 'qwertyuio'
29.         print(''.join(qsort(astr)))
30.         atuple = (10, 2, 34, 234, 1, 66, 88, 77)
31.         print(tuple(qsort(atuple)))
```

步骤二：测试脚本执行

```
01.     [root@localhost day06]# python3 qsort.py
02.     [31, 87, 88, 22, 26, 91, 99, 6, 7, 44]
03.     [6, 7, 22, 26, 31, 44, 87, 88, 91, 99]
04.     eioqrtuwy
05.     (1, 2, 10, 34, 66, 77, 88, 234)
```

4 案例4：测试程序运行效率

4.1 问题

创建deco.py脚本，要求如下：

1. 有个程序包含多个函数
2. 程序运行耗时较长
3. 为了确定哪个函数是瓶颈，需要计算出每个函数运行时间
4. 要求使用装饰器实现

Top

4.2 方案

如果一个程序有多个函数，查看每个函数运行耗时时长，这时如果修改每个函数，为它加上计时的功能，我们需要耗时去了解每个函数的功能，思考如何修改去增加计时功能，这样会相当繁琐，为避免这种现象出现，我们利用装饰器函数在不变动其他函数基础上，新增计时功能，在每个函数前添加语法糖（调用装饰器函数）即可，实施方法如下：

1.首先，定义一个函数loop()，该函数可以拥有任何功能，这里将for循环输出的数字添加到空列表中，并让进程挂起0.3s的时间

2.定义装饰器函数计算loop()函数运行耗时时长，装饰器函数deco，该函数的传入参数是loop（即被装饰函数），返回参数是内层函数。这里的内层函数-timeit，其实就相当于闭包函数，它起到装饰给定函数的作用

3.@ deco这个语法相当于执行loop = deco(loop)，为loop函数装饰并返回

4.装饰器函数在调用loop函数前后各返回一个当前时间，返回两个时间差即计算出运行耗时。

5.装饰器函数参数是你要装饰的函数名，装饰器函数返回是装饰完的函数名

需要要注意的是：为了不破坏原函数的逻辑，我们要保证内层函数timeit和被装饰函数loop的传入参数和返回值类型必须保持一致。

4.3 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写脚本

```
[root@localhost day05]# vim deco.py
#! /usr/bin/env python3

import time

def deco(func):
    def timeit():
        start = time.time()
        result = func()
        end = time.time()
        return end - start, result   # 将会返回元组
    return timeit

@deco
def loop():
    result = []
    for i in range(1, 6):
        result.append(i)
        time.sleep(0.3)
    return result

```

Top

```
22.  if __name__ == '__main__':
23.      # loop = deco( loop)
24.      print( loop( ) )
```

步骤二：测试脚本执行

```
01.  [ root@localhost day06] # python3 deco.py
02.  ( 1.50368070602417, [ 1, 2, 3, 4, 5] )
```

Top