

2018 年春季《大学计算机基础》（理科）实验指导书

实验 4 问题的描述—自定义数据结构

1. 实验目的

- (1) 了解如何用较为复杂数据结构描述问题。
- (2) 掌握用 Python 语言实现数据结构并解决问题的方法。

2. 实验任务

实验任务 4-1 括号匹配

题目描述：

某些编程语言需要用括号来分割代码块,如果左右括号不匹配,运行代码就会出现错误。据说,Python 的创立者就是因为讨厌花括号才创立了这么一种不需要括号分割的编程语言。

现如今许多 IDE 都可以对括号是否匹配进行检测,如果括号不匹配便会给出提示。请你利用栈的相关性质及操作,实现一个能检测括号是否匹配的程序。

输入格式：

输入为一行,是一段仅包含英文左右括号(括号包括“()”和“{ }”以及“[]”)的字符串(长度不超过 100)。

输出格式：

输出为判断结果,括号匹配则输出 True,不匹配则输出 False。

实验指导：

1. 通过 `from Stack import Stack` 导入提供的类。`from Stack import Stack` 语句中第一个 `Stack` 对应的是导入的文件名,第二个 `Stack` 才是调用 `Stack` 类(如果文件名为 `abc`,那么导入的语句就为 `from abc import Stack`)。

导入后通过 `s = Stack()` 实例化栈。实例化后可通过 `s.xxx` 调用方法,例如 `s.push(item)` 就会将 `item` 入栈。在 OJ 中提交时,需将以下代码拷贝至题目代码开头。提

供的栈的相关操作参见下方注释：

```
class Stack:
    def __init__(self): # 初始化
        self.items = []

    def push(self, item): # 元素入栈
        self.items.append(item)

    def pop(self): # 元素出栈，如果栈为空，抛出 stackIsEmpty 异常
        if self.is_empty():
            raise Exception('stackIsEmpty')
        else:
            return self.items.pop()

    def peek(self): # 返回栈顶元素，元素不出栈
        if self.is_empty():
            raise Exception('stackIsEmpty')
        else:
            return self.items[-1]

    def is_empty(self): # 栈为空返回 True，否则返回 False
        return self.items == []

    def size(self): # 返回栈内元素个数
        return len(self.items)
```

2. 设计思路

对于输入的由左右括号组成的字符串，遍历字符串中的每个字符，判断其为左右括号后执行不同的操作。如果为左括号入栈，如果为右括号则判断栈中是否有括号与其进行匹配以及该匹配是否是相同括号的匹配。最后结合匹配结果和栈是否为空，返回 **True** 或 **False**。

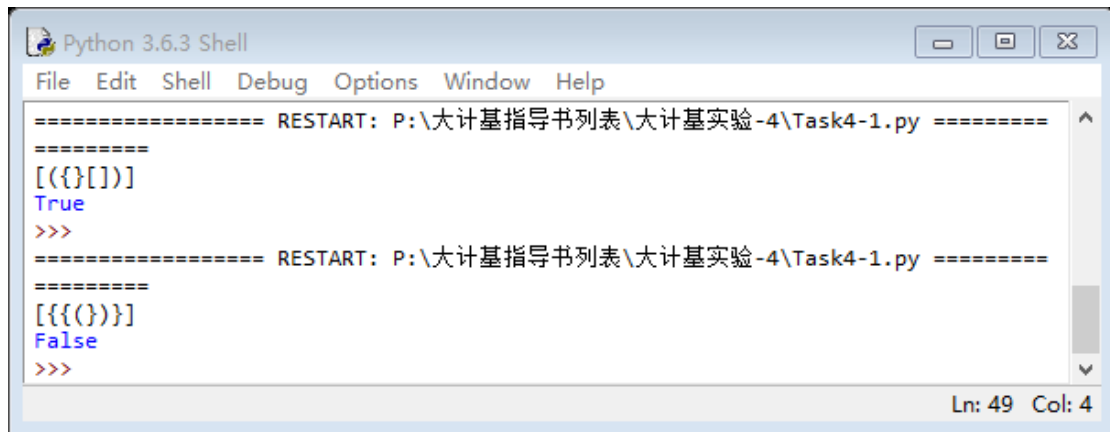
3. 本题只涉及三种括号，因此判断括号是左是右的一个简便方法就是分别对左右括号建立一个括号列表。通过 `if A in B` 判断语句可以很简单地判断 **A** 是否在 **B** 之中，**B** 可以为字符串、列表、元组、字典，该语句与普通的 `if` 条件语句用法相似，例如

```
if A in B: # 如果 A 在 B 中则会输出 xxx

    print('xxx')
```

4. 左右括号在 **ASCII** 编码表中挨得很近，你也可以利用这点来匹配对应的括号。

参考运行结果：



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help

===== RESTART: P:\大计基指导书列表\大计基实验-4\Task4-1.py =====
=====
[( {} [ ] )]
True
>>>
===== RESTART: P:\大计基指导书列表\大计基实验-4\Task4-1.py =====
=====
[( { ( ) } )]
False
>>>

Ln: 49 Col: 4
```

实验任务 4-2 由 9 和 0 组成的数字

题目描述:

给定一个数字 N ，求一个最小的正整数，该数字仅由 9 和 0 组成，而且是 N 的倍数。

定义队列的文件 `queue_class.py` 已给出，可直接导入并使用该文件。

输入格式:

输入数据包含一行，为一个正整数 N ，含义如题目描述所示。

输出格式:

输入数据包含一行，为所求的正整数，最高位应为 9。保证所求结果小于 10^{17} 。

实验指导:

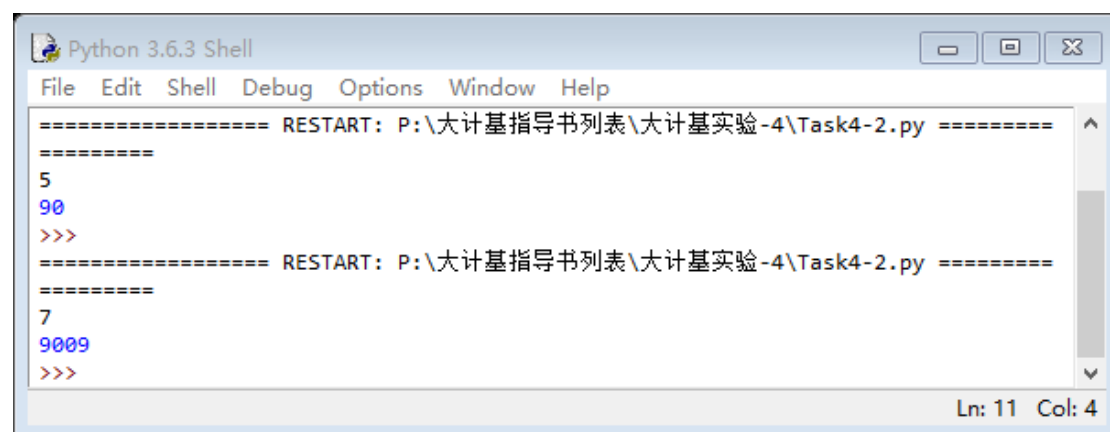
1. 与其他很多编程语言不同，Python 自带处理大整数的能力。你可以在 shell 中输入两个百余位的整数相乘，可以得到他们的精确乘积。在本题中，符合数据范围的整数仅有六万余个，因此你无需顾虑整数的范围问题。

2. 本题同样建议使用 `from queue_class import Queue` 导入提供的类，提交时将提供的 `Queue` 类粘贴至代码开头以替换 `import` 语句。

3. 设计思路

在本题中，我们只需从小到大依次判断由 9 和 0 组成的数字是否是 N 的倍数即可。由小到大产生此种数字的过程可以使用队列完成。试想有一个依次包含 900, 909, 990, 999 的队列，每次取队首元素进行判断，若判断不成功，则使用队首数字生成较大的数字，加入队尾。大家可以自行模拟，用什么生成方法能够得到我们需要的数字序列。

参考运行结果:



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
===== RESTART: P:\大计基指导书列表\大计基实验-4\Task4-2.py =====
=====
5
90
>>>
===== RESTART: P:\大计基指导书列表\大计基实验-4\Task4-2.py =====
=====
7
9009
>>>
Ln: 11 Col: 4
```

实验任务 4-3 矩阵运算

题目描述:

我们在高等代数课程中学过方阵的转置、加法、乘法等运算。在程序设计中我们可以定义矩阵的类，如此一来仅需输入矩阵并调用相应的函数，就可以轻松实现矩阵的运算。

在本次实验任务中，我们已经编写好了 **4x4** 矩阵类的部分方法，请你补全类定义中其他方法（详见下文），并通过我们写好的主程序展示你编写的结果。输入和输出由我们的主程序确定，你只需了解我们要求的属性和方法即可。

属性列表:

本题中的 **Matrix** 类含一个属性 **content**，为一个由 **16** 个元素组成的列表，每 **4** 个元素表示矩阵中一行，由上至下排列。原则上除 **__init__** 方法、**get_pos** 方法，**set_pos** 方法外不访问该属性。

方法列表:

本题中的 **Matrix** 类含八个方法:

__init__ 方法为构造方法，没有参数，目的是建立一个 **4x4** 的零矩阵；

get_pos 方法有两个参数 **x, y**，获取矩阵第 **x** 行第 **y** 列的元素，作为返回值；

set_pos 方法有三个参数 **x, y, value**，目的是将矩阵第 **x** 行第 **y** 列的元素值改为 **value**；

initialize 有一个参数 **matlist**，为一个列表，保证该列表包含四个子列表，每个子列表包含四个整型数值；

output 方法没有参数，目的是将矩阵以某种格式输出至标准输出流（屏幕）；

trans 方法没有参数，计算矩阵的转置矩阵，作为返回值；

plus 方法有一个参数 **m2**，计算本矩阵与 **m2** 矩阵的和，作为返回值；

multiply 方法有一个参数 **m2**，计算本矩阵与 **m2** 矩阵的乘积，作为返回值。

在本题中，你需要补全的为 **__init__**, **get_pos**, **initialize** 和 **multiply** 四个方法。

为了让解释器在格式处理中不产生问题，需补全的方法函数体都使用了 **pass** 语句占位。**pass** 语句不进行任何运算，没有任何含义。

实验指导:

可以依照之前学习过程中接触到的类的内容和这次程序中提供的其他方法模仿完成实验任务。

参考运行结果:

```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
===== RESTART: P:\大计基指导书列表\大计基实验-4\Task4-3.py =====
=====
1 0 0 0
0 1 0 1
0 1 0 0
0 0 1 0
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
x before initialization:
| 0 0 0 0 |
| 0 0 0 0 |
| 0 0 0 0 |
| 0 0 0 0 |
x after initialization:
| 1 0 0 0 |
| 0 1 0 1 |
| 0 1 0 0 |
| 0 0 1 0 |
y after initialization:
| 1 2 3 4 |
| 5 6 7 8 |
| 9 10 11 12 |
| 13 14 15 16 |

Transpose x is:
| 1 0 0 0 |
| 0 1 1 0 |
| 0 0 0 1 |
| 0 1 0 0 |
Transpose y is:
| 1 5 9 13 |
| 2 6 10 14 |
| 3 7 11 15 |
| 4 8 12 16 |
Transpose x+y is:
| 2 5 9 13 |
| 2 7 11 14 |
| 3 7 11 16 |
| 4 9 12 16 |
x*y is:
| 1 2 3 4 |
| 18 20 22 24 |
| 5 6 7 8 |
| 9 10 11 12 |
>>>
```

Ln: 49 Col: 4