《大学计算机基础》（常规班）实验指导书

实验 4　问题的描述——数据结构（2）

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　日期计算

输入某年某月某日，判断这一天是这一年的第几天？

实验目的：通过数据结构解决一些实际问题。

实验指导：

(1)定义一个函数来计算该年是否是闰年；

(2)用户输入年月日；

(3)列表里存放每个月的天数；

(4)计算天数；

(5)输出结果。

参考代码：

```
# 4-1.py
#函数 leapYear 用来判断输入的是否是闰年，如果是，返回 1。
def leapYear(n):
    ……

#提示用户输入年月日
year =……
month = ……
day =……

#列表 l 里存放每个月的天数（暂不考虑闰年），s 用于记录天数
l=[……]
s=0
month=int(month)
year=int(year)
day=int(day)

#根据输入的月份把前面几个月的天数全部加起来，并判断当年是否是闰年，是的话天数+1
for i in range(1,month):
    ……
s+=day
if fun(year)==1:
    ……
print(year,"年",month,"月",day,"日是这一年的第",s,"天")
```

实验任务 4-2 序列求和

计算 s=a+aa+aaa+aaaa+aa...a，其中 a 是一个数字。

例如 2+22+222+2222+22222(此时共有 5 个数相加)，几个数相加由用户输入 n 控制。

实验目的：学会通过 python 的控制结构和数据结构进行简单的计算。

实验指导：

(1) 接收用户输入的 a 和加数个数 n；

(2) 利用循环结构将 a、aa、aaa 等每个加数计算出来并存入列表；

(3) 将列表内各元素相加并输出。

参考代码：

```
# 4-2.py
#用户输入 a 和加数个数 n

#列表 l 用于存放 a、aa、aaa 等每一个加数
l=[a]
x=a
s=0
#循环中的 x 代表每一个加数，每次的 x 比上次的 x 多 a×10 的 i 次方
#向列表添加元素要用到列表的 append 方法
for i in range(1,n):
    ……
    l.append(x)
#将列表所有元素相加
for i in l:
    ……
print(s)
```

实验任务 4-3 大整数加法的实现

编写一个程序，实现两大整数的加法（两整数具有相同的位数）。注意不可直接使用加法表达式。

要求：多次运行程序，验证输入不同的数值，程序能否得到正确统计结果；并将程序运行结果截图后，粘贴到实验报告中。

实验目的：

学习使用合适的数据结构来解决相关问题。

实验指导：

（1） 问题分析

实现两大整数相加，可模拟做竖式的方法，从个位开始相加，满 10 则向前进位。因为在输入数据时我们通常由高位到低位输入，而在这里使用数据时需由低位到高位取出，很显然满足“后进先出”的规则，可采用栈来完成此题目。

（2） 具体步骤

①第一个加数数据按照输入顺序（从高位到低位）入栈 1，此时栈顶为最低位；

②第二个加数数据按照输入顺序（从高位到低位）入栈 2，此时栈顶为最

低位；

③将栈 1、栈 2 均弹出栈顶，并作加法，考虑进位，计算结果入栈 3，这时栈 3 正好是低位入栈；

④将栈 3 出栈，正好是由高位到低位的计算结果。

注：实现输入的数字按照由高位到低位入栈，可采取将输入数字的字符串每一位与字符"0"的 ASCII 码值作差的方法，如：

str = "123"

list=[]

for c in str

    list.append(ord(c)-ord("0"))

则列表 list = [1,2,3]

（3） 如果你对列表的属性和方法比较熟悉，可以将列表当作堆栈使用。此时会用到列表的 append()、pop()方法。这两个方法具体使用的例子如下：

```
>>> st = [3, 4, 5]
>>> st.append(6)
>>> st.append(7)
>>> st
[3, 4, 5, 6, 7]
>>> st.pop()
7
>>> st
[3, 4, 5, 6]
>>> st.pop()
6
>>> st.pop()
5
>>> st
[3, 4]
```

你也可以用课本所给出的 stack 类来完成此问题，类的定义如下：

```
#Stack.py
class Stack:
    #创建一个空栈
    def __init__(self):
        self.items=[]

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

    #元素出栈
    def pop(self):
        return self.items.pop()

    #返回栈顶的元素
    def peek(self):
        return self.items[len(self.items)-1]

    #判断栈是否为空
    def isEmpty(self):
        return self.items==[]

    #返回栈的大小
    def size(self):
        return len(self.items)
```

参考代码：

```
#4-3.py
#利用栈实现大整数加法
#栈的实现
……

#主函数入口
if __name__=="__main__":

    #输入两个加数
    ……
    #建立建立加数栈 1、加数栈 2 和结果栈
    ……

    #加数 1 和加数 2 按照由高位到低位分别入栈
    ……
    tmp=0
    while ……：  #栈 1 和栈 2 均非空时执行循环
        tmp += ……  #取栈 1 和栈 2 的栈顶相加，存入临时变量 tmp
        ……  #栈 1 弹出栈顶
        ……  #栈 2 弹出栈顶
        ……  #将 tmp 除以 10 的余数压入结果栈
        tmp = ……  #取 tmp 除以 10 的商，结果只能为 0 和 1,1 代表有进位
    if tmp:  #最高位相加后若有进位则将进位结果压入结果栈
        ……

    #结果输出
    ……
```

程序运行结果如下：

```
请输入第一个加数：98765
请输入第二个加数：12345
98765与12345相加的结果是：111110
```

实验任务 4-4 重复密钥加密法的实现（文科班选做）

凯撒加密法是一种简单的消息编码方式，它是按照字母表将消息中的每个字母移动常量的 k 位。但是这种方式极易破解，因为字母的移动只有 26 种可能。

因此，为了解决这一问题，我们提出了重复密钥加密法：这时不是将每个字母移动常数位，而是利用一个密钥值列表，将各个字母移动不同的位数。如果消息比密钥值长，可以从头再使用这个密钥值列表。

请编写一个程序，实现重复密钥加密法，并进行解密，以验证方法的正确性。为简化问题，约定移动的范围在整个 ASCII 码表内，密钥值可为正、负、零，加密时将原字符的 ASCII 码值加上密钥值即可，规定密钥值的绝对值不超过 15。

要求：多次运行程序，验证输入不同的原文，程序能否得到正确密文和解密结果；并将程序运行结果截图后，粘贴到实验报告中。

实验目的：

学习使用合适的数据结构来解决相关问题。

实验指导：

（1） 问题分析

加密（或解密）本身的方法非常简单，只需将原文（或密文）的 ASCII 码值加上（或减去）对应的密钥值即可，问题的重点在于密钥的重复性。按照题目要求，每使用完一个密钥值，则将其放在密钥值列表的最后，实现循环使用。由此可知其满足“先进先出”的规则，可使用队列来方便地完成此题目。

（2） 具体步骤

①给出密钥列表和待加密的原文；

②建立两个队列，分别储存一份密钥，模拟加密者使用一份密钥，解密者使用一份密钥；

③依次取一个密钥值，对原文加密，再将密钥值放回队列；

④输出加密后的密文，再按照同样的方法进行解密，输出解密后的原文，进行比较。

（3） 如果你对列表的属性和方法比较熟悉，可以将列表当作队列使用。此时会用到列表的 append()、pop()方法。这两个方法具体使用的例子如下：

```
>>> queue = ['a', 'b', 'c']
>>> queue.append('d')
>>> queue.append('e')
>>> queue
['a', 'b', 'c', 'd', 'e']
>>> queue.pop(0)
'a'
>>> queue
['b', 'c', 'd', 'e']
>>> queue.pop(0)
'b'
>>> queue
['c', 'd', 'e']
```

你也可以用课本所给出的 queue 类来完成此问题，类的定义如下：

```
#queue_class.py
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self,item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

参考代码：

```
#4-4.py
#利用队列实现重复密钥加密法
#队列的实现
……

#主函数入口
if __name__=='__main__':
    ……  #密钥列表
    ……  #输入待加密原文
    str_encode=""   #密文字符串
    str_decode=""   #解密字符串
    #将密钥值分别储存在加密队列和解密队列中
    ……

    #加密过程实现，包括从队列中取密钥值、进行加密和将用完的密钥值放入队列
    ……

    #解密过程实现，包括从队列中取密钥值、进行解密和将用完的密钥值放入队列
    ……

    #输出过程
    print("加密后的密文为：%s" %str_encode)
    print("解密后的原文为：%s" %str_decode)
```

注意：以上代码结构适用于利用编写好的队列类来实现队列。若选择用列表实现队列，则可直接在主程序中将列表用作队列，无需主程序前的队列实现部分。

程序运行结果如下：