

## 1.Flowchart

```
1  def print_values(a,b,c):
2      a = float(input('请输入a的值: '))
3      b = float(input('请输入b的值: '))
4      c = float(input('请输入c的值: '))
5      if a > b:
6          if b > c:
7              print(a,b,c)
8          else:
9              if a > c:
10                 print(a,c,b)
11             else:
12                 print(c,a,b)
13     else:
14         if b > c:
15             if a > c:
16                 print(b,a,c)
17             else:
18                 print(b,c,a)
19         else:
20             print(c,b,a)
21
22 print_values(1,1,1)
23
```

Flowchart 的所有脚本

```
1  def print_values(a,b,c):
2      a = float(input('请输入a的值: '))
3      b = float(input('请输入b的值: '))
4      c = float(input('请输入c的值: '))
```

首先定义一个函数 print\_values，参数为 a, b, c。三者都为浮点数，需要让用户输入 a, b, c 的值。

```
5      if a > b:
6          if b > c:
7              print(a,b,c)
8          else:
9              if a > c:
10                 print(a,c,b)
11             else:
12                 print(c,a,b)
13     else:
14         if b > c:
15             if a > c:
16                 print(b,a,c)
17             else:
18                 print(b,c,a)
19         else:
20             print(c,b,a)
```

接着，进入了一个判断语句，如果结果为 True，则执行 if 后面的内容，否则执行 else 之后的内容。

假设用户输入的值  $a=3$ ， $b=2$ ， $c=1$ 。那么第一个  $a > b$  为 True，执行 if 之后的内容。接着判断 b 和 c 的大小，因为  $b > c$  为 True，所以执行 if 之后的内容，最后打印出 3.0 2.0 1.0 以此类推。

## 2. 矩阵乘法

### 2.1

```
8 from random import randint
9 import numpy as np
10 M1=np.random.randint(0,51,(5,10))
11 M2=np.random.randint(0,51,(10,5))
12 print(M1)
13 print(M2)
14
15
16 def Matrix_multip(M1,M2):
17     result = np.zeros((len(M1),len(M2[0])),int)
18     for i in range(len(M1)):
19         for j in range(len(M2[0])):
20             for k in range(len(M2)):
21                 result[i][j] += M1[i][k] * M2[k][j]
22     print(result)
23 Matrix_multip(M1,M2)
24
```

矩阵乘法的所有脚本

```
8 from random import randint
9 import numpy as np
```

首先，从模块中导入组成矩阵的随机数。

```
10 M1=np.random.randint(0,51,(5,10))
11 M2=np.random.randint(0,51,(10,5))
12 print(M1)
13 print(M2)
```

创建并打印矩阵 M1 和 M2，以 M1 为例：(0, 51) 代表取得 0 – 50 中的随机整数，(5, 10) 代表创建的矩阵为 5 行 10 列，M2 同理。

### 2.2

接着我们将两个矩阵相乘。

```
16 def Matrix_multip(M1,M2):
17     result = np.zeros((len(M1),len(M2[0])),int)
```

定义一个函数 Matrix\_multip，变量为 M1 和 M2，接着创建一个 5 行 5 列的矩阵 result：5 行代表 M1 的行数也就是 len(M1)，5 列代表 M2 的列数也就是 len(M2[0])，最后的输出结果为整数所以加上个 int。

```
18     for i in range(len(M1)):
19         for j in range(len(M2[0])):
20             for k in range(len(M2)):
21                 result[i][j] += M1[i][k] * M2[k][j]
22     print(result)
```

接着创建了 3 个 for 循环，第一个 for 循环的目的是迭代矩阵 result 的行。

第二个 for 循环的目的是迭代 result 的列。

第三个 for 循环的目的是迭代 M1 的列或者说迭代 M2 的行。

最后将 M1 和 M2 相乘的结果添加到 result 中。

调用函数，即可打印矩阵相乘的结果。

### 3. Pascal\_triangle 帕斯卡三角形

```
9 def Pascal_triangle(k):
10     blank = []
11     for i in range(k):
12         list1 = [1]
13         if i != 0:
14             for j in range(i - 1):
15                 list1.append(blank[j]+blank[j+1])
16             list1.append(1)
17         blank = list1
18
19     print(blank)
20     Pascal_triangle(100)
21     Pascal_triangle(200)
22
```

帕斯卡三角形的所有脚本

```
9 def Pascal_triangle(k):
10     blank = []
```

首先创建一个函数 `Pascal_triangle(k)`，在函数内创建一个空列表 `blank = []`，创建空列表的目的是用来暂时储存上一行两两相加得到的新列表。

```
11     for i in range(k):
12         list1 = [1]
```

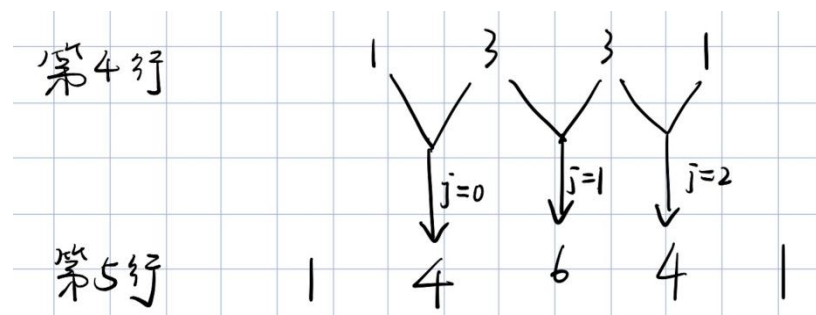
接着创建一个 `for` 循环，创建 `for` 循环的目的是使帕斯卡三角形执行到第  $k$  行（例如， $k = 5$ ，则会执行到第 5 行）。在 `for` 循环之后创建一个列表 `list1 = [1]`，这一步的目的是创建一个初始列表作为帕斯卡三角形的第一行。

```
17         blank = list1
18
19     print(blank)
```

如果  $k = 1$  则  $i = 0$ ，那么 `blank` 就会等于 `list1 = [1]`，那我们输出 `blank` 就会等于 `[1]`。

```
13         if i != 0:
14             for j in range(i - 1):
15                 list1.append(blank[j]+blank[j+1])
16             list1.append(1)
17         blank = list1
```

如果  $i$  不等于 0，则会执行以上代码。我们创建一个 `for` 循环，这个 `for` 循环的目的是使上一行两两相加得到新列表并储存在 `blank` 中。当我们需要算第  $k$  行时， $i$  能取得的最大值为  $(k-1)$ ，我们需要确定第  $(k-1)$  行的数两两相加并添加进 `list1` 中，所以 `list1.append(blank[j] + blank[j+1])`。第  $(k-1)$  行一共有  $i$  个数字所以  $j$  的取值范围为  $(i-1)$ 。最后只需在列表的末尾加上一个元素 1 就得到列表 `blank`。



以第  $k = 5$  为例，如上图所示。

## 4. Least\_moves

```
9  def least_moves(x):
10     count = 0
11     while x > 1:
12         if x%2 == 0:
13             x = x/2
14         else:
15             x = x - 1
16             count = count + 1
17     print(count)
18
19     least_moves(10)
20
```

Least\_moves 的全部脚本

```
9  def least_moves(x):
10     count = 0
```

首先定义函数 least\_moves (x) 参数为 x，创建一个变量 count = 0，count 的目的是用来计算从 1 走到 x 一共用了多少步。

```
11     while x > 1:
12         if x%2 == 0:
13             x = x/2
14         else:
15             x = x - 1
16             count = count + 1
17     print(count)
```

我采用了逆向思维，从 1 走到 x 所需的步数和从 x 走到 1 的步数是一样的。从 x 到 1 的最短步数是：当 x 为偶数时  $x = x/2$ ，当 x 为奇数时  $x = x - 1$ 。

创建一个循环， $x > 1$  的目的是当 x 走到 1 循环结束。接着进行判断，当 x 为偶数时执行 if 之后的语句，否则执行 else 之后的语句，每次循环 count 都会+1 用来计算一共循环了多少次，最后输出 count 即可得到走了多少步。

## 5. Dynamic programming

```
9 from itertools import product
10
11 def expr(p):
12     return "{}1{}2{}3{}4{}5{}6{}7{}8{}9".format(*p)
13
14 def gen_expr():
15     op = ['+', '-', '']
16     return [expr(p) for p in product(op, repeat=9) if p[0] != '+']
17
18 def all_exprs():
19     values = {}
20     for expr in gen_expr():
21         val = eval(expr)
22         if val not in values:
23             values[val] = 1
24         else:
25             values[val] += 1
26
27     return values
28
29
30 def Find_expression(val):
31     counts = 0
32     for s in filter(lambda x: x[0] == val, map(lambda x: (eval(x), x), gen_expr())):
33         print(s)
34
35 Find_expression(50)
36
37 Total = []
38 for expr in gen_expr():
39     val = eval(expr)
40     if val in range(1,101):
41         Total += [val]
42
43 Total_solutions = []
44 for i in range(1,101):
45     Total_solutions = Total_solutions + [Total.count(i)]
46
47 positionmax = Total_solutions.index(max(Total_solutions))+1
48 positionmin = Total_solutions.index(min(Total_solutions))+1
49 print('在0-100中数字中: ', positionmax, '产生的解决方案最多, 一共有: ', max(Total_solutions), '种')
50 print('在0-100中数字中: ', positionmin, '产生的解决方案最少, 一共有: ', min(Total_solutions), '种')
```

图: dynamic programming 的所有代码

```
8
9 from itertools import product
10
```

第一步, 调用模块

```
12 def expr(p):
13     return "{}1{}2{}3{}4{}5{}6{}7{}8{}9".format(*p)
```

第二步, 创建一个函数 expr (p): p 为参数 (后面会用到), 并将返回值返回给函数。

"{}1{}2{}3{}4{}5{}6{}7{}8{}9"的目的是在 1-9 之间的任意位置插入 '+'、'-' and ""。

```
15 def gen_expr():
16     op = ['+', '-', '']
17     return [expr(p) for p in product(op, repeat=9) if p[0] != '+']
```

第三步, 创建一个函数 gen\_expr():, 在函数内创建一个列表 op = ['+', '-', '']。

Iterator	Arguments	Results
<code>product()</code>	<code>p, q, ...</code> <code>[repeat=1]</code>	cartesian product, equivalent to a nested for-loop

Examples	Results
<code>product('ABCD', repeat=2)</code>	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD

<https://docs.python.org/3/library/itertools.html>

上图是 product 的用法，我从中得到启发。product (op, repeat=9) 就为 '+'、'-' 或 'or' 的任意排列组合，例如：-----，-----+，-----++。一共有  $3^9$  种不同的排列组合。

这里创建一个 for 循环，使 p 为其中一种情况，此时为了避免重复需要判断第一位是否为 '+'，因为第一位为 '+' 的情况和 '-' 是一样的结果，因此引入 if 语句筛选出第一位不为 '+' 的排列。将其结果返回到 expr (p) 函数中，将结果返回即得到所有情况的表达式。

所以，如果我们 print (gen\_expr ()) 则会产生所有表达式，例如 1+2+3+4+5+6+7+8+9，1+2+3+4+5+6+7+8-9 等等。

```

20 def all_exprs():
21     values = {}
22     for expr in gen_expr():
23         val = eval(expr)
24         if val not in values:
25             values[val] = 1
26         else:
27             values[val] += 1
28
29     return values
30

```

第四步，创建一个函数 all\_exprs() 目的是来计算函数 gen\_expr() 中的表达式得到的结果并统计该结果出现的次数有几次。

在函数内创建一个空字典 values = {}。

创建一个 for 循环逐个计算 gen\_expr() 中的表达式并赋值给 val。

### Python eval() 函数

 Python 内置函数

#### 描述

eval() 函数用来执行一个字符串表达式，并返回表达式的值。

#### 语法

以下是 eval() 方法的语法：

```
eval(expression[, globals[, locals]])
```

<https://www.runoob.com/python/python-func-eval.html>

此处，我从上方链接得到 eval 的用法。

接着创建一个 if 语句，如果 val 不在 values 中，执行 if 语句，那就会添加一个新的 key : values。否则执行 else 语句就会变成 key: values+1。这一步的目的是判断该数字一共有几个表达式。

```
{43: 34, 25: 36, 115: 15,
```

如上图是 print(values) 的部分截图，得数等于 43 的一共有 34 种不同的表达式，得数等于 25 的一共有 36 种表达式，以此类推。



```
32 def Find_expression(val):
33     counts = 0
34     for s in filter(lambda x: x[0] == val, map(lambda x: (eval(x), x), gen_expr())):
35
36         print(s)
37
38 Find_expression(50)
39 |
```

第五步，创建一个函数 Find\_expression(val)，参数 val 代表用户想要结果，例如 val 等于 50，则会输出所有得数等于 50 的表达式。如下图所示：

```
In [344]: runfile('/Users/gong/Desktop/ESE5023/assignment/assignment1/  
Find_expression.py', wdir='/Users/gong/Desktop/ESE5023/assignment/assignment1')  
(50, '-1+2+3-4+56+7-8+9')  
(50, '-1+2-3+4+56+7+8-9')  
(50, '-1-2-34-5+6+7+89')  
(50, '-1+23-4+56+7-8-9')  
(50, '-1-2+3+4+56+7-8-9')  
(50, '-1-2+34+5+6+7-8+9')  
(50, '-1-23+4-5+6+78-9')  
(50, '-12+3+4+5+67-8-9')  
(50, '-12+3+45+6+7-8+9')  
(50, '-12+3-4-5+67-8+9')  
(50, '-12-3+4-5+67+8-9')  
(50, '-1+2+3+4-56+7+89')  
(50, '-1+2+3-4+56+7+8-9')  
(50, '-1+2+34-5-6+7+8+9')  
(50, '-1+2+34-56+78-9')  
(50, '-1+2-3+4+56+7-8-9')  
(50, '-1+2-34+5-6-7+89')  
(50, '-1-2+3-45+6+78+9')  
(50, '-1-2+34+5+6+7+8-9')  
(50, '-1-2+34-5-67+89')  
(50, '-1-2-3+4+56+7-8+9')  
(50, '-1-2-3-4-5-6+78-9')  
(50, '-1-2-34-5-6+7+89')  
(50, '-1-23+4+5-6+78-9')  
(50, '-1-23-4-5-6+78+9')  
(50, '12+3+4-56+78+9')  
(50, '12-3+45+6+7-8-9')  
(50, '12-3-4-5+67-8-9')
```

<https://www.runoob.com/python/python-func-filter.html>

<https://www.runoob.com/python3/python3-func-map.html>

[https://blog.csdn.net/weixin\\_41656968/article/details/79598754](https://blog.csdn.net/weixin_41656968/article/details/79598754)

<https://towardsdatascience.com/lambda-functions-with-practical-examples-in-python-45934f3653a8>

我在以上 4 个链接中了解了 filter, map, lambda 的用法, 以及 filter 和 map 的区别。

图中 `map(lambda x: (eval(x), x), gen_expr())` 函数用于计算 `gen_expr()` 中所有表达式的值并返回

回 x, 此时 x 为一个元组, 例如 x = (100, '-1+2-3+4+5+6+78+9')。

lambda x: x[0] == val 用于比较用户输入的值 val 是否与 x[0]相等，如果相等则打印出结果以及表达式。

所以此处创建了一个 for 循环用来核对表达式计算的结果和用户输入的值是否匹配。

最后，调用函数 `Find expression (50)` 就可以执行所有结果等于 50 的解决方案。

## 5.2

```
40 Total = []
41 for expr in gen_expr():
42     val = eval(expr)
43     if val in range(1,101):
44         Total += [val]
```

创建一个空列表 `Total = []`。

创建一个 for 循环用来判断计算 `gen_expr()` 中表达式计算的结果是否在 1-101 之内，如果在 1-101 之内则将结果添加进列表 `Total` 中，否则不添加。`Total` 的部分结果如下图：

```
[43, 25, 27, 9, 88, 29, 11, 13, 97, 79, 81, 63, 31, 13, 15, 94, 76, 17, 89, 1, 35, 88, 70, 72, 54, 74, 56, 58, 40, 33, 15, 17, 96, 78, 19, 1, 91, 3, 87, 69, 71, 53, 21, 3, 93, 5, 84, 66, 7, 79, 25, 48, 39, 21, 34, 79, 61, 63, 45, 65, 47, 49, 31, 99, 27, 67, 49, 51, 33, 53, 35, 37, 19, 71, 35, 17, 19, 1, 98, 80, 21, 3, 93]
```

所有数字都在 1-101 之内。

```
45 Total_solutions = []
46 for i in range(1,101):
47     Total_solutions = Total_solutions + [Total.count(i)]
48     #print(Total_solutions)
49     positionmax = Total_solutions.index(max(Total_solutions))+1
50     positionmin = Total_solutions.index(min(Total_solutions))+1
51     print('在0-100中数字中: ',positionmax,'产生的解决方案最多, 一共有: ',max(Total_solutions),'种')
52     print('在0-100中数字中: ',positionmin,'产生的解决方案最少, 一共有: ',min(Total_solutions),'种')
53
```

接着创建一个空列表 `Total_solutions = []`。

创建一个 for 循环，目的是判断数字 `i` 在列表 `Total` 中一共出现了几次，出现了几次也就代表一共有几种解决方案。并将出现的次数附加到列表 `Total_solutions` 中。例如当 `i = 100` 时，

`Total.count(i) = 12`，那就会将 12 附加到列表 `Total_solutions` 中。

因此，列表 `Total_solutions` 中储存了 1-101 各个数字解决方案的数量。

创建变量 `positionmax` 用来储存数字出现解决方案最大值的位置，用 `max()`，就能找出最大值，+1 是因为列表从 0 开始计数。`positionmin` 同理。

最后，只需打印出哪个数字出现解决方案最多以及有多少种即可，如下图所示：

```
在0-100中数字中:  9 产生的解决方案最多, 一共有:  46 种
在0-100中数字中:  100 产生的解决方案最少, 一共有:  12 种
```