南开大学
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

软件工程

软件编码实现、分析和测试

穆禹宸　2012026

年级：2020 级

专业：信息安全、法学双学位班

指导教师：徐思涵

2023 年 5 月 16 日

目录

一、 问题描述及分析

(一) 问题描述

输入一个整数数组 nums，需要找出数组中乘积最大的非空连续子数组，并输出该子数组所对应的乘积。(子数组是指数组的连续子序列)。

示例：输入：nums = [2, 3, −2, 4]，输出：6。解释：子数组 [2, 3] 有最大乘积 6。

(二) 问题分析

解决该问题的一个较为朴素的思路是暴力枚举数组的每一个子数组，计算每个子数组的乘积，从而得到乘积最大的子数组，时间复杂度为 $O(n^2)$。考虑效率更高的算法，我们考虑数组的第 i 个元素 nums[i]，如果其是一个正数，我们希望以它前一个位置结尾的某个段的积也是个正数，并且希望该段的乘积尽可能地大；如果其是一个负数，那么我们希望以它前一个位置结尾的某个段的积也是个负数，这样就可以负负得正，并且我们希望该段的乘积尽可能小。显然，我们可以使用动态规划算法来解决该问题，用 fmax(i) 来表示以第 i 个元素结尾的乘积最大的子数组的乘积，用 fmin(i) 表示以第 i 个元素结尾的乘积最小的子数组的乘积。我们可以得到动态规划转移方程如下：

$$f_{max}(i) = max(f_{max}(i-1) \times nums[i], f_{min}(i-1) \times nums[i], nums[i]) \tag{1}$$

$$f_{min}(i) = min(f_{max}(i-1) \times nums[i], f_{min}(i-1) \times nums[i], nums[i]) \tag{2}$$

fmax(i) 中的最大值即为最终答案。

二、 原始代码

优化前的代码

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        if len(nums) == 0: return 0
        nums1 = nums[:]
        for i in range(1, len(nums)):
            if nums[i] == 0 or nums[i-1] == 0:
                continue
            nums[i] *= nums[i-1]

        for i in range(len(nums1)-2, -1, -1):
            if nums1[i] == 0 or nums1[i+1] == 0:
                continue
            nums1[i] *= nums1[i+1]
        maxList = [max(nums), max(nums1)]
        return max(maxList)
```

三、 Pylint 代码分析

通过使用 Pylint，可以得出对代码规范的静态分析，这里采用 PEP8 规范。

在命令行之中直接输入指令

代码指令

```
pylint filename.py
```

得到如下结果:

```
************* Module MaxPro
MaxPro.py:5:0: C0325: Unnecessary parens after 'while' keyword (superfluous-parens)
MaxPro.py:25:33: C0303: Trailing whitespace (trailing-whitespace)
MaxPro.py:26:18: C0303: Trailing whitespace (trailing-whitespace)
MaxPro.py:27:0: C0303: Trailing whitespace (trailing-whitespace)
MaxPro.py:33:0: C0303: Trailing whitespace (trailing-whitespace)
MaxPro.py:38:0: C0305: Trailing newlines (trailing-newlines)
MaxPro.py:1:0: C0114: Missing module docstring (missing-module-docstring)
MaxPro.py:1:0: C0103: Module name "MaxPro" doesn't conform to snake_case naming style (invalid-name)
MaxPro.py:1:0: C0115: Missing class docstring (missing-class-docstring)
MaxPro.py:4:4: C0116: Missing function or method docstring (missing-function-docstring)
MaxPro.py:14:4: C0116: Missing function or method docstring (missing-function-docstring)
MaxPro.py:14:4: C0103: Method name "maxProduct" doesn't conform to snake_case naming style (invalid-name)
MaxPro.py:19:12: C0103: Variable name "mx" doesn't conform to snake_case naming style (invalid-name)
MaxPro.py:20:12: C0103: Variable name "mn" doesn't conform to snake_case naming style (invalid-name)
MaxPro.py:28:0: C0116: Missing function or method docstring (missing-function-docstring)


Report
======
30 statements analysed.
```

图 1: 原结果

首次 pylint 检查分数为 5 分。具体原因主要有以下几点:

- 存在路径定义问题
- 缺少最后的换行符
- 函数没有编写描述
- 行尾有空格
- 函数名命名不符合 snake_case 规范

针对以上问题在经过修改以后，得到满分十分的结果，如下所示:

```
PS D:\MYCODE\Software-Engineering\乘积最大子数组\raw> pylint .\max_product.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

图 2: 优化后结果

我们通过更加复杂、更有针对性的命令，得到更加详细的分析结果。程序如下所示:

分析代码

```
import pylint.lint

pylint_opt = ['-ry', './max_product.py']
pylint.lint.Run(pylint_opt)
#c为convention
```

然后得到如下分析结果:

```
PS D:\MYCODE\Software-Engineering\乘积最大子数组\fliter> pylint max_product.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 9.76/10, +0.24)

PS D:\MYCODE\Software-Engineering\乘积最大子数组\fliter> pylint -ry max_product.py


Report
======
41 statements analysed.

Statistics by type
------------------

+---------+-------+-----------+-----------+------------+---------+
|type     |number |old number |difference |%documented |%badname |
+=========+=======+===========+===========+============+=========+
|module   |1      |1          |=          |100.00      |0.00     |
+---------+-------+-----------+-----------+------------+---------+
|class    |1      |1          |=          |100.00      |0.00     |
+---------+-------+-----------+-----------+------------+---------+
|method   |4      |4          |=          |100.00      |0.00     |
+---------+-------+-----------+-----------+------------+---------+
|function |3      |3          |=          |100.00      |0.00     |
+---------+-------+-----------+-----------+------------+---------+



59 lines have been analyzed
```

图 3: 优化后结果

```
59 lines have been analyzed

Raw metrics
-----------

+---------+------+-----+--------+----------+
|type     |number |%    |previous |difference |
+=========+======+=====+========+==========+
|code     |43    |72.88 |NC      |NC        |
+---------+------+-----+--------+----------+
|docstring |9     |15.25 |NC      |NC        |
+---------+------+-----+--------+----------+
|comment  |7     |11.86 |NC      |NC        |
+---------+------+-----+--------+----------+
|empty    |0     |0.00  |NC      |NC        |
+---------+------+-----+--------+----------+


Duplication
-----------

+-------------------------+-----+--------+----------+
|                         |now  |previous |difference |
+----------+-------+-------+-----+--------+----------+
|type      |number |previous |difference |
+==========+=======+========+==========+
|convention |0     |0       |0         |
+----------+-------+--------+----------+
|refactor  |0     |0       |0         |
+----------+-------+--------+----------+
|warning   |0     |0       |0         |
+----------+-------+--------+----------+
|error     |0     |0       |0         |
+----------+-------+--------+----------+
```

图 4: 优化后结果

```
Messages
--------

+-----------+------------+
|message id |occurrences |
+===========+============+



-------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

图 5: 优化后结果

从结果中我们可以看出,程序代码已经完美符合了代码规范,各方面均取得了满分的成绩!(这里只展示最后的结果，并不展示修改的过程，因此最后满分十分是多次测试后的结果，末尾的提高分数为 0 分，因为已经取得了满分的成绩!)

四、 profile 性能分析

在这里使用 import profile 或 import cProfile 都可，通过结合所学内容以及查阅资料，本实验中采用 Profile 包进行性能分析。首先，对于原始程序，这里进行展示:

程序代码

```python
"""提供计算给定整形数组的乘积最大的非空连续子数组"""
import numpy as np
class Solution:
    """Solution Class"""
    def __init__(self) -> None:
        """初始化数组"""
        self.array = []
    def setarray(self):
        """输入数组, 如果输入数组不是int数组则要求重新输入"""
        while True:
            numsstr = input("please input an array, use space to split
                numbers:")
            temp = numsstr.split(' ')
            try:
                self.array = [int (x) for x in temp]
                break
            except(SyntaxError,ValueError):
                print("your input is illegal, please input again")
                continue
    def maxproduct(self) -> int:
        """考虑到乘以负数会反转符号, 所以要同时存储当前最大乘积与最小乘积"""
        res = self.array[0]
        max_f = self.array[0]#存储当前最大乘积
        min_f = self.array[0]#存储当前最小乘积
        for i in range(1, len(self.array)):
```

```
            tem_max = max_f
            tem_min = min_f
            #比较当前最大乘积, 当前坐标的值, 当前最小乘积与当前坐标值的乘积
            max_f = max(tem_max * self.array[i], self.array[i], tem_min *
                self.array[i])
            #比较当前最小乘积, 当前坐标的值, 当前最大乘积与当前坐标值的乘积
            min_f = min(tem_min * self.array[i], self.array[i], tem_max *
                self.array[i])
            res = max(res, max_f)
        return res
    def test_set_array(self,testnum:list[int]):
        "直接设置数组"
        self.array = testnum
def profile_test(scale):
    """测试性能"""
    test_array = [np.random.randint(-10, 10) for i in range(scale)]
    mxpd = Solution()
    mxpd.test_set_array(test_array)
    mxpd.maxproduct()
def unit_test(test_array:list)->int:
    """unit测试"""
    mxpd = Solution()
    mxpd.test_set_array(test_array)
    return mxpd.maxproduct()
def main():
    """主函数"""
    solution = Solution()
    solution.setarray()
    print("the result is: "+str(solution.maxproduct()))
if __name__ == '__main__':
    main()
```

可以编写如下的程序:

性能分析

```
import profile
import max_product_sub_array as mpsa

profile.run("mpsa.profile_test (1000000)")
```

然后我们观察我们优化以后的代码结果:

```
PS D:\MYCODE\Software-Engineering> & D:/anaconda3/python.exe d:/MYCODE/Software-Engineering/乘积最大子数组/raw/try_profile.py
         4000007 function calls in 4.062 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.062    4.062 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(len)
  1999998    0.797    0.000    0.797    0.000 :0(max)
   999999    0.469    0.000    0.469    0.000 :0(min)
  1000000    1.281    0.000    1.281    0.000 :0(randint)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    4.062    4.062 <string>:1(<module>)
        1    1.203    1.203    2.469    2.469 max_product.py:19(maxproduct)
        1    0.000    0.000    0.000    0.000 max_product.py:33(test_set_array)
        1    0.000    0.000    4.062    4.062 max_product.py:36(profile_test)
        1    0.312    0.312    1.594    1.594 max_product.py:38(<listcomp>)
        1    0.000    0.000    0.000    0.000 max_product.py:5(__init__)
        1    0.000    0.000    4.062    4.062 profile:0(max_product.profile_test(1000000))
        0    0.000             0.000          profile:0(profiler)


PS D:\MYCODE\Software-Engineering>
```

图 6: 性能分析

可以看到运行时间集中在 profile_test 函数，即我们计算一维数组最大乘积子数组的函数。我们应该集中优化该函数，具体的优化思路在文章开头已经给出，即使用时间复杂度为 O(N) 的动态规划算法替代原来的时间复杂度为 $O(N^3)$ 的暴力算法。当然，我们这里已经是优化后的结果，就是动态规划的了。进一步优化的话可能可以考虑使用一些并行算法来加速该函数的运行。除此之外，我们可以使用滚动数组来优化动态规划算法的内存占用。

因此，我们尝试进一步降低复杂度，得到一个 $O(n)$ 的程序:

程序代码

```python
"""提供计算给定整形数组的乘积最大的非空连续子数组"""
import numpy as np
class Solution:
    """Solution Class"""
    def __init__(self) -> None:
        """初始化数组"""
        self.array = []
    def setarray(self):
        """输入数组, 如果输入数组不是int数组则要求重新输入"""
        while True:
            #输入数组
            numsstr = input("please input an array, use space to split
                numbers:")
            #将输入的字符串数组转换为int数组
            temp = numsstr.split(' ')
            #判断输入的数组是否为int数组
            try:
                self.array = [int (x) for x in temp]
                break
            except(SyntaxError,ValueError):
                print("your input is illegal, please input again")
                continue
    def maxproduct(self) -> int:
        """考虑到乘以负数会反转符号, 所以要同时存储当前最大乘积与最小乘积"""
        maxval = float('-inf')#存储当前最大乘积
```

```
25          imax, imin = 1, 1#存储当前最小乘积
26          for i in range(0,len(self.array)):
27              #如果当前坐标的值小于0，则最大乘积与最小乘积交换
28              if self.array[i] < 0:
29                  #比较当前最大乘积，当前坐标的值，当前最小乘积与当前坐标值的乘
                        积
30                  imax, imin = imin, imax
31              #比较当前最小乘积，当前坐标的值，当前最大乘积与当前坐标值的乘积
32              imax = max(imax * self.array[i], self.array[i])
33              imin = min(imin * self.array[i], self.array[i])
34              #比较当前最大乘积与最大乘积
35              maxval = max(maxval, imax)
36          return maxval
37      def test_set_array(self,testnum:list[int]):
38          "直接设置数组"
39          self.array = testnum
40  def profile_test(scale):
41      """测试性能"""
42      test_array = [np.random.randint(-10, 10) for i in range(scale)]
43      mxpd = Solution()
44      mxpd.test_set_array(test_array)
45      mxpd.maxproduct()
46  def unit_test(test_array:list)->int:
47      """unit测试"""
48      mxpd = Solution()
49      mxpd.test_set_array(test_array)
50      return mxpd.maxproduct()
51  def main():
52      """主函数"""
53      solution = Solution()
54      solution.setarray()
55      print("the result is: "+str(solution.maxproduct()))
56  if __name__ == '__main__':
57      main()
```

再次进行测试，如下所示:

```
PS D:\MYCODE\Software-Engineering> & D:/anaconda3/python.exe d:/MYCODE/Software-Engineering/乘积最大子数组/fliter/try_profile.py
         4000010 function calls in 4.031 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.031    4.031 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(len)
  2000000    0.859    0.000    0.859    0.000 :0(max)
  1000000    0.312    0.000    0.312    0.000 :0(min)
  1000000    1.406    0.000    1.406    0.000 :0(randint)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.016    0.016    4.031    4.031 <string>:1(<module>)
        1    1.188    1.188    2.359    2.359 max_product.py:22(maxproduct)
        1    0.000    0.000    0.000    0.000 max_product.py:37(test_set_array)
        1    0.000    0.000    4.016    4.016 max_product.py:40(profile_test)
        1    0.250    0.250    1.656    1.656 max_product.py:42(<listcomp>)
        1    0.000    0.000    0.000    0.000 max_product.py:5(__init__)
        1    0.000    0.000    4.031    4.031 profile:0(max_product.profile_test(1000000))
        0    0.000             0.000          profile:0(profiler)


PS D:\MYCODE\Software-Engineering>
```

图 7: 性能分析

可以看到在 profile_test 函数和总的执行时间上都有了一定的改善，有了毫秒级的提高。当然，我们可以看到，函数调用最多的是集中在 python 的内置函数 max 和 min 上，这是由于我们维护了已知的最大和最小值，同时不断更新这两个值造成的。这一点如果还需要进行提速，可行的办法是采取 C++ 重写底层代码的方式进行加速，不过那已经超出了我们这个实验的范畴。

五、 unittest 单元测试

对我们的代码进行黑盒测试。设计样例考虑如下:

- 首先对于数据的正负值，我们设计了全为正数、全为负数以及正数和负数混合的测试样例;
- 对于数据类型，我们设计了全为整数、全为浮点数以及整数和浮点数混合的测试样例;
- 对于数组长度，我们设计了数组长度为 1 的极端数据情况;
- 设计了对从标准输入读入数组功能和从文件读入数组功能的测试样例;
- 同时我们也进行了错误输入的测试，如输入的不是列表、输入格式错误、数组元素不是 int/float、数组为空、输入的数组维度大于 1 等。

具体测试样例和代码如下:

单元测试

```python
class TestMaxProduct(unittest.TestCase):
    """单元测试类"""
    def test_common_case(self):
        #数组长度为1
        self.assertEqual(mxpd.unit_test([2]),2)
        self.assertEqual(mxpd.unit_test([-2]),-2)
        #全正整数
        self.assertEqual(mxpd.unit_test([3,4,5]),60)
        #全负整数
        self.assertEqual(mxpd.unit_test([-3,-4,-5]),20)
        #正负整数混合
```

```
        self.assertEqual(mxpd.unit_test([2,3,-4,5]),6)
        #全浮点数
        self.assertEqual(mxpd.unit_test([2.0,-3.0,4.0,5.0]),20)
        #整数浮点数混合
        self.assertEqual(mxpd.unit_test([2,-3.0,4,5.0]),20)
        #带0
        self.assertEqual(mxpd.unit_test([-2,0,-1]),0)

    @patch('builtins.input')
    def test_read_from_stdin(self, mock_input):
        #数组长度为1
        mock_input.return_value = '[2]'
        self.assertEqual(mxpd.unit_test([2]),2)
        mock_input.return_value = '[-2]'
        self.assertEqual(mxpd.unit_test([-2]),-2)
        #全正整数
        mock_input.return_value = '[3, 4, 5]'
        self.assertEqual(mxpd.unit_test([3,4,5]),60)
        #全负整数
        mock_input.return_value = '[-3,-4,-5]'
        self.assertEqual(mxpd.unit_test([-3,-4,-5]),20)
        #正负整数混合
        mock_input.return_value = '[2,3,-4,5]'
        self.assertEqual(mxpd.unit_test([2,3,-4,5]),6)
        #全浮点数
        mock_input.return_value = '[2.0,-3.0,4.0,5.0]'
        self.assertEqual(mxpd.unit_test([2.0,-3.0,4.0,5.0]),20)
        #整数浮点数混合
        mock_input.return_value = '[2,-3.0,4,5.0]'
        self.assertEqual(mxpd.unit_test([2,-3.0,4,5.0]),20)
        #带0
        mock_input.return_value = '[-2,0,-1]'
        self.assertEqual(mxpd.unit_test([-2,0,-1]),0)

if __name__ == '__main__':
    suite = unittest.TestSuite()
    tests = [TestMaxProduct("test_common_case"), TestMaxProduct("
        test_read_from_stdin")]
    suite.addTests(tests)
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

测试用例表:

编写一个测试类进行测试，同时与之前的步骤相同，给出详细的代码注释，遵守代码规范。

得到如下的结果:

序号	测试样例描述	输入参数	期望输出
1	数组大小为 3，输入值全正	2，3，4	24
2	数组大小为 3，输入值全负	-2，-3，-4	12
3	数组大小为 5，输入值为正负数混合	2，-3，4，-2，-1	48
4	数组大小为 3，输入值为小数整数混合	0.5，1.5，2	3
5	数组大小为 4，输入值含 0	0，2，3，0	6

表 1: 测试样例

```
PS D:\MYCODE\Software-Engineering\乘积最大子数组\raw> coverage run .\try_unittest.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
PS D:\MYCODE\Software-Engineering\乘积最大子数组\raw> coverage html -d my_coverage_result
Wrote HTML report to my_coverage_result\index.html
PS D:\MYCODE\Software-Engineering\乘积最大子数组\raw>
```

图 8: 单元测试

首先我们可以看到测试的结果全部通过了，这证明了我们程序的正确性。

六、 代码覆盖率

然后，我们通过`coverage`得到代码覆盖率结果，输入如下命令

单元测试

```
coverage run try_unittest.py
coverage html -d my_coverage_result
```

生成代码覆盖率报告。

Coverage for **max_product.py**: 60%

40 statements | 24 run | 16 missing | 0 excluded

« prev ^ index » next coverage.py v7.2.5, created at 2023-05-16 00:02 +0800

```
"""提供计算给定整形数组的乘积最大的非空连续子数组"""
import numpy as np
class Solution:
    """Solution Class"""
    def __init__(self) -> None:
        """初始化数组"""
        self.array = []
    def setarray(self):
        """输入数组，如果输入数组不是int数组则要求重新输入"""
        while True:
            #输入数组
            numsstr = input("please input an array, use space to split numbers:")
            #将输入的字符串数组转换为int数组
            temp = numsstr.split(' ')
            #判断输入的数组是否为int数组
            try:
                self.array = [int (x) for x in temp]
                break
            except(SyntaxError,ValueError):
                print("your input is illegal, please input again")
                continue
```

图 9: 代码覆盖率

```
    def maxproduct(self) -> int:
        """考虑到乘以负数会反转符号，所以要同时存储当前最大乘积与最小乘积"""
        maxval = float('-inf')#存储当前最大乘积
        imax, imin = 1, 1#存储当前最小乘积
        for i in range(0,len(self.array)):
            #如果当前坐标的值小于0，则最大乘积与最小乘积交换
            if self.array[i] < 0:
                #比较当前最大乘积，当前坐标的值，当前最小乘积与当前坐标值的乘积
                imax, imin = imin, imax
            #比较当前最小乘积，当前坐标的值，当前最大乘积与当前坐标值的乘积
            imax = max(imax * self.array[i], self.array[i])
            imin = min(imin * self.array[i], self.array[i])
            #比较当前最大乘积与最大乘积
            maxval = max(maxval, imax)
        return maxval
    def test_set_array(self,testnum:list[int]):
        "直接设置数组"
        self.array = testnum
```

图 10: 代码覆盖率

```
def unit_test(test_array:list)->int:
    """unit测试"""
    mxpd = Solution()
    mxpd.test_set_array(test_array)
    return mxpd.maxproduct()
def main():
    """主函数"""
    solution = Solution()
    solution.setarray()
    print("the result is: "+str(solution.maxproduct()))
if __name__ == '__main__':
    main()
```

图 11: 代码覆盖率

Coverage report: 79%

coverage.py v7.2.5, created at 2023-05-16 00:02 +0800

Module	*statements*	*missing*	*excluded*	*coverage*
max_product.py	41	16	0	61%
try_unittest.py	37	0	0	100%
Total	**78**	**16**	**0**	**79%**

图 12: 代码覆盖率

代码中未被覆盖的语句主要为用于性能测试的 profile_test 函数以及 main 函数，这两个函数并不需要被测试，在单元测试时也没有被调用，因此未被覆盖。其余部分的代码大致均已被覆盖，当然还有一些输入输出的语句，并没有呗覆盖到。因为在测试时，我们的样例是编写好的。只有几条为了后续扩展预留的代码没有被覆盖。

七、 编程规范

代码规范是一套约定俗成的准则，用于指导程序员在编写代码时遵循统一的风格和结构。它的目的是提高代码的可读性、可维护性和可扩展性，以便多个人能够协同开发并理解代码。在 Python 中，PEP8（Python Enhancement Proposal 8）是一份广泛接受的代码规范，它提供了关于代码布局、命名约定、注释等方面的建议。

PEP8 规范主要包括以下几个方面的内容：

- 缩进：使用四个空格作为缩进的标准，不要使用制表符（Tab）。
- 行长度：每行代码应尽量保持在 79 个字符以内，超过这个限制时应进行换行处理。对于长的表达式或注释，可以使用括号或反斜杠进行换行，以提高可读性。
- 命名规范：变量、函数、类等命名应具有描述性，并采用小写字母和下划线的组合。类名应采用驼峰命名法（每个单词首字母大写），模块名应尽量短小，并避免与 Python 标准库模块重名。

- 空格使用：在运算符、逗号、冒号、分号等符号周围应添加适当的空格，以增加代码的可读性。例如，赋值操作符前后应添加空格，函数参数之间应添加逗号和空格。
- 导入规范：每个导入语句应独占一行，按照标准库、第三方库和本地库的顺序进行分组。每个分组之间应留有一个空行。避免使用通配符导入（如 from module import *），而是明确导入所需的模块或函数。
- 注释规范：使用注释来解释代码的功能、实现细节等重要信息。注释应该清晰明了，并且应该与代码保持同步更新。对于非常重要或复杂的功能，可以使用多行注释进行详细说明。
- 函数和类的定义：函数和类的定义之间应留有两个空行，类中的方法定义之间应留有一个空行。函数和方法的参数列表应放在括号内，并在逗号后添加一个空格。
- 其他规范：避免使用多个语句或表达式在同一行上，除非是在列表、字典等简单结构中。对于条件语句和循环语句，要使用适当的缩进，并在必要时使用括号来明确代码块的范围。

为了编写符合 PEP8 规范的 Python 代码，可以使用一些工具来自动检查和格式化代码，例如 pylint 等。这些工具可以帮助发现不符合规范的代码，并提供修复建议。当然，最重要的是我们要养成良好的习惯。

八、 程序可扩展性

程序的可扩展性是指程序在面对变化和增长时，能够以一种简单、灵活和可持续的方式进行修改、扩展和维护的能力。可扩展性是一个重要的软件设计目标，它确保程序在需求变化和系统演化的情况下能够保持高效、可靠和易于维护。一个可扩展的程序具有以下特征：

- 模块化设计：程序应该被分解为独立的、可重用的模块，每个模块都专注于特定的功能。模块化设计使得在需要扩展或修改程序时，可以更容易地理解和操作代码，而无需改动整个程序。
- 松耦合：模块之间应该尽可能地解耦，即减少模块之间的依赖关系。通过定义明确的接口和使用适当的抽象层，可以降低模块之间的耦合性，使得对一个模块的修改不会对其他模块产生不必要的影响。
- 可替换性：程序应该提供一种简单的方式来替换或添加新的功能模块，而无需修改现有代码。这可以通过接口的设计和依赖注入等技术来实现。可替换性使得在需求变化时，可以快速地引入新的功能或替换旧的实现，而不会对整个程序产生破坏性的影响。
- 可配置性：程序应该提供一些配置选项，使得用户能够根据需求和环境来自定义程序的行为。通过将程序的行为参数化，可以更容易地适应不同的使用场景和需求变化。
- 扩展点：程序应该提供一些明确的扩展点，允许其他开发者或用户通过插件、模块或扩展来增加新的功能。这些扩展点应该具有良好的接口定义和规范，以便其他人能够简单地添加新功能而无需修改核心代码。

针对性本次实验程序的可扩展性，主要采取了以下措施：

- 数据扩展：数据方面，我们主要考虑了数据来源的扩展、数据规模的扩展以及数据形式的扩展。

- 数据来源：考虑我们如何得到数据，我们可以从用户输入、文件、网络、数据库等来源处得到我们需要处理的数组，因此这里我们将得到数据的这一行为封装成 set_array 成员函数，该函数根据传入参数的不同调用不同的函数，从不同的源头处获得数据。如代码实现部分所展示，我们目前实现了从用户输入、文件、函数参数处获取待处理的数组，之后如果想对获取数组这一行为进行扩展，我们只需要实现新的相应的数据读取函数，并将该函数添加到 set_array 函数的一个新的分支中即可。
- 数据规模：这里为了支持对任意长度的数组以及多维数组的处理，我们选择使用 list 类型来存储数组，使用 list 类型在理论上可以对数组的维度和数组长度进行无限延伸，从而支持不同数据规模的处理。
- 算法实现方式扩展:
- 考虑到在未来，我们还有可能对最大乘积子数组算法的具体实现方式进行更改，我们定义成员函数提供统一的抽象，当我们修改或者添加新的算法实现方式的时候，我们只需要在函数中调用新的处理函数即可，而不需要更改其他的函数的代码。
- 结果呈现扩展: 除了返回数组的最大乘积子数组的乘积的值之外，我还实现了得到该最大乘积子数组的具体内容，用户调用函数便可以得到子数组内容以及乘积。未来可以考虑将结果可视化地呈现给用户

九、 两个原则

单一职责原则(Single Responsibility Principle,SRP)和开放封闭原则(Open-Closed Principle, OCP）是面向对象设计中的两个重要原则，它们有助于构建灵活、可维护和可扩展的软件系统。

(一) 单一职责原则（SRP）

单一职责原则指的是一个类或模块应该只有一个单一的责任或功能。换句话说，一个类应该只有一个引起它变化的原因。如果一个组件有多个职责，则会导致代码耦合度增加，难以维护和扩展。相反，如果一个组件只负责一个职责，则可以使其更加专注、可测试和易于修改。单一职责原则是许多其他原则和设计模式的基础，如依赖反转原则、工厂模式等。这样做的好处是:

- 高内聚：类的职责单一，类内部的各个方法和属性都围绕着同一个关注点。这样可以提高类的内聚性，减少代码的复杂性。
- 易于理解和维护：由于类的职责明确，代码的功能和行为更加清晰，使得其他开发者更容易理解和维护。
- 可扩展性：当需求变化时，只需要修改与该职责相关的类，而不影响其他类。这样可以减少代码的修改范围，降低引入错误的风险。

要遵守和应用单一职责原则，可以采取以下措施:

- 分析类的职责：仔细分析每个类的职责，并确保每个类都专注于一个单一的责任。如果一个类承担了多个职责，应该考虑将其拆分为多个类。
- 提取通用功能：当发现多个类有相似的功能时，可以将这些功能提取出来，形成一个单独的类或模块，以避免功能重复。

- 使用组合和委托：当一个类需要实现多个职责时，可以使用组合和委托的方式，将职责委托给其他类来处理。这样可以保持类的单一职责，同时协同工作完成多个功能。

针对我们的程序来说我对 MaxProductSubArray 类的成员函数的设计符合单一职责原则。如 set_array 函数只负责读取数组；array_is_legal 函数只负责判断数组是否合法；solve 函数只负责数组求解。且对于不同数组读取方式以及不同的问题求解算法也编写到了不同的函数中，从而保证对某一个功能的修改只会导致单一函数的代码发生改动，符合单一职责原则。

(二) 开放封闭原则（OCP）

开放封闭原则指的是软件实体（类、模块、函数等）应该对扩展开放，对修改封闭。换句话说，软件实体应该通过扩展来改变其行为，而不是通过修改其源代码来实现, 而是应该通过增加新代码来实现。这样可以避免对原有功能产生影响，并且可以更好地维护和测试代码。为了实现开放封闭原则，可以使用接口、抽象类、继承、多态等技术。开放封闭原则也是面向对象设计中最重要的原则之一。这样做的好处是：

- 可扩展性：通过扩展现有的代码，可以引入新的功能和行为，而无需修改已经稳定的代码。这样可以减少对现有功能的影响和风险。
- 可维护性：由于不修改现有的代码，所以不会破坏已经测试和验证过的功能。这样可以降低维护成本，提高系统的稳定性。

要遵守和应用开放封闭原则，可以采取以下措施：

- 使用抽象和接口：通过定义抽象类和接口，可以定义稳定的协议，供其他模块或类来扩展和实现。这样可以保持现有代码的稳定性，同时允许引入新的实现。
- 使用多态性：通过利用多态性，可以将扩展的具体实现替换为抽象的接口，而不影响现有代码的调用。这样可以实现新的功能，同时不修改已有的代码。
- 使用设计模式：一些设计模式，如装饰器模式、策略模式等，提供了一种实现开放封闭原则的方法。通过应用适当的设计模式，可以使系统更加灵活和可扩展。
- 编写可插拔的代码：通过使用插件、扩展点和配置文件等机制，使得系统的功能可以通过外部的组件进行扩展。这样可以在不修改现有代码的情况下，添加新的功能。

针对我自己的程序, MaxProductSubArray 类的成员函数的作用是求出一维数组的最大子数组乘积，这一部分内容是不必进行更改的，对于更高维度的数组的求解可以通过调用该函数来实现，是可以扩展的，满足开放-封闭原则。

十、 错误处理

错误是导致程序崩溃的问题，例如 Python 程序的语法错误 (解析错误) 或者未捕获的异常 (运行错误) 等. 异常是运行时期检测到的错误，即使一条语句或者表达式在语法上是正确的，当试图执行它时也可能会引发错误。异常处理是用于管理程序运行期间错误的一种方法，这种机制将程序中的正常处理代码和异常处理代码显式地区别开来，提高了程序的可读性。

良好的错误和异常处理对于构建健壮、可靠的软件系统至关重要。以下是在 Python 中实现良好错误和异常处理的一些关键概念和技术：

- 异常类和异常处理语句：Python 中的异常是通过抛出异常对象来表示的。异常对象是由内置的异常类或自定义的异常类实例化得到的。通过使用 raise 语句抛出异常，可以将异常传递给调用者或处理该异常的代码块。在处理异常时，可以使用 try-except 语句来捕获和处理特定类型的异常。try 块中包含可能引发异常的代码，而 except 块则定义了如何处理捕获到的异常。可以使用多个 except 块来处理不同类型的异常，也可以使用 else 块定义在 try 块中没有发生异常时执行的代码，还可以使用 finally 块定义无论是否发生异常都必须执行的代码。
- 异常处理的层级结构：Python 中的异常类是通过形成层级结构来组织的，这样可以根据异常的类型进行更精细的处理。所有的异常类都是 BaseException 类的子类。常见的内置异常类包括 Exception、TypeError、ValueError 等。可以根据具体情况选择捕获特定类型的异常或捕获更通用的异常。
- 适当捕获异常：捕获和处理可能发生的异常，避免程序因为异常而中断或崩溃。根据具体情况选择捕获特定类型的异常或捕获更通用的异常。
- 提供有意义的错误信息：在捕获异常时，可以使用 except 块来记录或打印有意义的错误信息，以便于调试和排查问题。错误信息应该清晰明确，指示发生了什么问题和如何解决。
- 避免捕获过宽的异常：避免捕获过于宽泛的异常，因为这可能会隐藏潜在的问题或导致错误的处理。尽可能捕获特定的异常类型，以便有针对性地处理和恢复。
- 不要忽略异常：避免将异常处理代码
- 留空或简单地忽略异常。至少应该记录异常，以便追踪和排查问题。根据具体情况，可以选择重新抛出异常、执行备选操作或中断程序的执行。
- 使用 finally 块进行清理操作：finally 块中的代码总是会被执行，无论是否发生异常。这可以用于进行清理操作，如关闭文件、释放资源等。
- 定义异常：在 Python 中，可以通过继承内置的异常类来创建自定义的异常类。自定义异常类可以根据具体的应用场景和需求来定义，以便更好地表示特定的异常情况。通过自定义异常类，可以提供更具体和有意义的错误信息，以及针对特定异常的处理逻辑。

针对我们的程序，首先我们应该规定要处理的数组的格式，我们规定数组应该以列表的形式输入，数组元素为 int 或 float 类型。同时目前只支持一维数组的求解，对于多维数组应当给出报错提示。首先，我们在 ___read_array_from_stdin 和 ___read_array_from_file 这两个成员函数中会通过 literal_eval 函数来初步判断读入的数据是否是 python 中的合法的数据类型，如果判断失败则会打印提示。但是凭借 literal_eval 只能在一定程度上对输入数组进行检测，我们还需要调用 array_is_legal 函数来进一步判断输入数组的合法性。在该函数中，我们首先判断数组是否为 list 格式，如果不是则提示错误。然后判断数组是否为空，若为空同样提示错误。最后，判断数组的维度，如果维度大于一，由于程序还没有实现对高维数组的处理，这里也提示错误；如果数组维度为一，则遍历数组每个元素判断元素类型是否为 int/float，若不是，则提示错误。最终我们根据 array_is_legal 的返回值来赋值 self.legal。当 self.legal 为 False 时，调用类的 solve 函数将会提示错误。

十一、 完整程序代码

最终代码

```python
"""提供计算给定整形数组的乘积最大的非空连续子数组"""
import numpy as np
class Solution:
    """Solution Class"""
    def __init__(self) -> None:
        """初始化数组"""
        self.array = []
    def setarray(self):
        """输入数组, 如果输入数组不是int数组则要求重新输入"""
        while True:
            #输入数组
            numsstr = input("please input an array, use space to split numbers:")
            #将输入的字符串数组转换为int数组
            temp = numsstr.split(' ')
            #判断输入的数组是否为int数组
            try:
                self.array = [int (x) for x in temp]
                break
            except(SyntaxError,ValueError):
                print("your input is illegal, please input again")
                continue
    def maxproduct(self) -> int:
        """考虑到乘以负数会反转符号, 所以要同时存储当前最大乘积与最小乘积"""
        maxval = float('-inf')#存储当前最大乘积
        imax, imin = 1, 1#存储当前最小乘积
        for i in range(0,len(self.array)):
            #如果当前坐标的值小于0, 则最大乘积与最小乘积交换
            if self.array[i] < 0:
                #比较当前最大乘积, 当前坐标的值, 当前最小乘积与当前坐标值的乘积
                imax, imin = imin, imax
            #比较当前最小乘积, 当前坐标的值, 当前最大乘积与当前坐标值的乘积
            imax = max(imax * self.array[i], self.array[i])
            imin = min(imin * self.array[i], self.array[i])
            #比较当前最大乘积与最大乘积
            maxval = max(maxval, imax)
        return maxval
    def test_set_array(self,testnum:list[int]):
        "直接设置数组"
        self.array = testnum
def profile_test(scale):
    """测试性能"""
    test_array = [np.random.randint(-10, 10) for i in range(scale)]
    mxpd = Solution()
    mxpd.test_set_array(test_array)
    mxpd.maxproduct()
```

```
def unit_test(test_array:list)->int:
    """unit测试"""
    mxpd = Solution()
    mxpd.test_set_array(test_array)
    return mxpd.maxproduct()
def main():
    """主函数"""
    solution = Solution()
    solution.setarray()
    print("the result is: "+str(solution.maxproduct()))
if __name__ == '__main__':
    main()
```

代码覆盖率

```
coverage run try_unittest.py
coverage html -d my_coverage_result
```

静态分析

```
import pylint.lint

pylint_opt = ['-ry', './max_product.py']
pylint.lint.Run(pylint_opt)
```

性能分析

```
import profile
import max_product
profile.run("max_product.profile_test(1000000)")
```

单元测试

```
import unittest
import max_product as mxpd
from unittest.mock import patch
class TestMaxProduct(unittest.TestCase):
    """单元测试类"""
    def test_common_case(self):
        #数组长度为1
        self.assertEqual(mxpd.unit_test([2]),2)
        self.assertEqual(mxpd.unit_test([-2]),-2)
        #全正整数
        self.assertEqual(mxpd.unit_test([3,4,5]),60)
        #全负整数
        self.assertEqual(mxpd.unit_test([-3,-4,-5]),20)
        #正负整数混合
        self.assertEqual(mxpd.unit_test([2,3,-4,5]),6)
        #全浮点数
        self.assertEqual(mxpd.unit_test([2.0,-3.0,4.0,5.0]),20)
```

```
        #整数浮点数混合
        self.assertEqual(mxpd.unit_test([2,-3.0,4,5.0]),20)
        #带0
        self.assertEqual(mxpd.unit_test([-2,0,-1]),0)

    @patch('builtins.input')
    def test_read_from_stdin(self, mock_input):
        #数组长度为1
        mock_input.return_value = '[2]'
        self.assertEqual(mxpd.unit_test([2]),2)
        mock_input.return_value = '[-2]'
        self.assertEqual(mxpd.unit_test([-2]),-2)
        #全正整数
        mock_input.return_value = '[3, 4, 5]'
        self.assertEqual(mxpd.unit_test([3,4,5]),60)
        #全负整数
        mock_input.return_value = '[-3,-4,-5]'
        self.assertEqual(mxpd.unit_test([-3,-4,-5]),20)
        #正负整数混合
        mock_input.return_value = '[2,3,-4,5]'
        self.assertEqual(mxpd.unit_test([2,3,-4,5]),6)
        #全浮点数
        mock_input.return_value = '[2.0,-3.0,4.0,5.0]'
        self.assertEqual(mxpd.unit_test([2.0,-3.0,4.0,5.0]),20)
        #整数浮点数混合
        mock_input.return_value = '[2,-3.0,4,5.0]'
        self.assertEqual(mxpd.unit_test([2,-3.0,4,5.0]),20)
        #带0
        mock_input.return_value = '[-2,0,-1]'
        self.assertEqual(mxpd.unit_test([-2,0,-1]),0)

if __name__ == '__main__':
    suite = unittest.TestSuite()
    tests = [TestMaxProduct("test_common_case"), TestMaxProduct("
        test_read_from_stdin")]
    suite.addTests(tests)
    runner = unittest.TextTestRunner()
    runner.run(suite)
```