

编码实现、分析和测试-乘积最大子数组

- 姓名：管昀玫
- 学号：2013750
- 专业：计算机科学与技术

1 题目

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1:输入: `nums=[2,3,-2,4]`，输出:6，解释子数组[2, 3]有最大乘积6

在此基础上，需要对代码进行分析：

- 使用pylint对代码进行分析
- 使用profile对代码进行性能分析
- 使用unittest对代码进行单元测试

2 代码分析

2.0 核心代码实现

2.0.1 MaxProduct类

初始化：

```
class MaxProduct:
    """
    乘积最大子数组
    """
    def __init__(self):
        """初始化"""
        self.num=[]
        self.nums=[]
        self.max_dp=0
        self.min_dp=0
        self.ans_dp=0
        self.shape=0
```

在这里，初始化6个成员变量。

- `self.nums` 用于存储待处理的数组
- `self.num` 为 `self.nums` 的辅助数组（多维数组按行处理）
- `self.max_dp`，`self.min_dp` 和 `self.ans_dp` 为动态规划的辅助变量
- `self.shape` 用于存储输入数组的维度。

2.0.2 成员函数set

用于输入数组并赋值，代码实现如下：

```
def set(self,in_num):
    """赋值输入数组"""
    self.nums=in_num    # 将输入的数组赋值给nums

    # np.array().ndim返回数组的维数，1代表一维数组，2代表二维数组
    self.shape=np.array(self.nums).ndim
```

2.0.3 成员函数calculate_list

用于计算成绩最大子数组的值。这里主要是使用动态规划的思想，分别记录当前的最大乘积和最小乘积，不断更新得到最大乘积。

```
def calculate_list(self):
    """计算乘积最大子数组的值"""
    # 如果num的长度小于等于1，那么ans_dp就是num[0]的值
    if len(self.num) <= 1:
        self.ans_dp=self.num[0]
    self.max_dp,self.min_dp,self.ans_dp = self.num[0],self.num[0],self.num[0]
    for i in range(1, len(self.num)):
        self.max_dp,self.min_dp= max(self.max_dp * self.num[i], self.num[i],
self.min_dp * self.num[i]),\
                                min(self.max_dp * self.num[i],
self.num[i],self.min_dp * self.num[i])
        self.ans_dp = max(self.ans_dp, self.max_dp)
    return self.ans_dp
```

函数首先对输入的数组 `self.num` 进行了长度判断，如果长度小于等于1，则最大乘积即为该元素本身。如果数组长度大于1，则需要进行动态规划处理，初始化时，将最大乘积、最小乘积和最终结果都赋值为数组的第一个元素，即 `self.max_dp=self.min_dp=self.ans_dp=self.num[0]`。

接下来，从数组的第二个元素开始，遍历整个数组，对于每个元素，将当前最大乘积和最小乘积与当前元素相乘得到两个中间结果，然后将这两个结果与当前元素本身进行比较，选出其中最大的和最小的结果。这里使用了Python内置的 `max()` 和 `min()` 函数实现。

最后，将当前的最大乘积与之前的最大乘积进行比较，选出其中更大的作为最终的结果，即 `self.ans_dp = max(self.ans_dp, self.max_dp)`。整个过程就是动态规划的过程，通过保存之前的结果，避免了重复计算，最终得到了最大乘积值所对应的非空连续子数组的结果。

2.0.4 成员函数calculate

这里主要对输入的数组进行分类处理。

- 如果数组是一维数组，那么num就等于nums
- 如果数组是二维数组，那么对每一行进行处理，即逐行调用 `calculate_list`，最后返回计算结果的最大值

```
def calculate(self):
    """对输入数列处理后计算MP"""
    # 如果数组是一维数组，那么num就等于nums
    if self.shape==1:
        self.num=self.nums
        print(self.num)
        return self.calculate_list() # 返回计算结果
    # 如果数组是二维数组，那么对每一行进行处理
    if self.shape==2:
        ans_list=[]
        for i in range(self.shape):
            self.num=self.nums[i]
            ans_list.append(self.calculate_list())
        return max(ans_list) # 返回计算结果的最大值
    print("Program error!")
    return TypeError
```

2.0.5 成员函数get_ans

这里直接返回 self.ans_dp 即可。

```
def get_ans(self):
    """返回计算结果"""
    return self.ans_dp
```

2.0.6 其他函数 pipeline

此函数即为计算流水线，调用 MaxProduct 类，set 其成员变量，返回 calculate 结果。

```
def pipeline(input_nums):
    """计算流水线"""
    my_mp=MaxProduct()
    my_mp.set(input_nums)
    return my_mp.calculate()
```

2.0.7 其他函数 get_list

输入要处理的集合，最后调用 pipeline 进行处理。

```
def get_list(the_set):
    """
    :param the_set: 要处理的集合
    :return:最大的数
    """
    return pipeline(the_set)
```

2.0.8 其他函数 get_in

控制程序的输入，实现由输入直接生成结果。

```
def get_in():
    """获取输入数组
    包括错误处理
    """
    inlist=[]
    try:
        inlist=literal_eval(input("please input a One-dim Or Two-dim array: "))
    except (SyntaxError, ValueError):
        print("error input! ")
    except TypeError:
        print("Format error!")
    if isinstance(inlist,tuple):
        inlist=list(inlist)
    elif isinstance(inlist,int):
        inlist=[inlist]
    elif isinstance(inlist, list):
        temp=np.array(inlist)
        if len(temp.shape) == 1:
            for index in inlist:
                try:
                    assert isinstance(index, (int, float))
                except AssertionError:
                    print("error input!")
        elif len(temp.shape)==2:
            for index_1 in inlist:
                for index_2 in index_1:
                    try:
                        assert isinstance(index_2, (int, float))
                    except AssertionError:
                        print("error input!")
        else:
            print("本程序仅支持一维数组和二维数组")
            return TypeError
    else:
        print("error input! ")
        return TypeError
    return inlist
```

在这个函数中，使用 `literal_eval()` 函数将用户输入的字符串转换为相应的Python对象，如字典、列表等，并将其赋值给 `inlist`。如果用户的输入不符合语法规则或者无法转换为Python对象，则会引发 `SyntaxError` 或 `ValueError` 异常，并打印出 "error input!" 的错误提示信息。如果 `literal_eval()` 函数无法识别用户输入的数据类型，则会引发 `TypeError` 异常，并打印出 "Format error!" 的错误提示信息。

然后，判断用户输入的数据类型，并对数据类型进行处理。

- 如果用户输入的是元组类型，则将其转换为列表类型。
- 如果用户输入的是整数类型，则将其转换为列表类型并将其作为列表的一个元素。

- 如果用户输入的是列表类型，则直接使用该列表。

最后，使用 `numpy` 库将列表转换为数组，并判断数组的形状。

- 如果数组是一维数组，则循环遍历数组中的元素，判断每个元素是否为整数或浮点数类型，如果不是则打印出 "error input!" 的错误提示信息。
- 如果数组是二维数组，则使用双重循环遍历每个元素，判断每个元素是否为整数或浮点数类型，如果不是则打印出 "error input!" 的错误提示信息。
- 如果数组的形状既不是一维数组也不是二维数组，则打印出 "本程序仅支持一维数组和二维数组" 的错误提示信息，并返回 `TypeError`。
- 如果用户输入的数据类型不是元组、整数或列表，则打印出 "error input!" 的错误提示信息，并返回 `TypeError`。最后，返回处理好的输入数组。

2.0.9 main函数

主要用于控制各个函数调用

```
if __name__ == '__main__':
    #获取一组数，用list存储
    nums=get_in()
    """执行计算"""
    print(pipeline(nums))
```

2.1 编程规范

2.1.1 编程规范解释

采用的是谷歌的编程规范，参考网址为：<https://google.github.io/styleguide/pyguide.html>

Type	Public	Internal
Packages	lower_with_under	
Modules	lower_with_under	_lower_with_under
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected)
Method Names	lower_with_under()	_lower_with_under() (protected)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

文件头规范：

Python文件头规范是指在Python脚本的第一行中添加特定格式的注释，用来指定脚本的解释器类型和版本，以便操作系统能够正确地解释执行该脚本。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

其中，第一行以`#!`开头，后面紧跟着解释器的路径。`/usr/bin/env`是一个通用的解释器路径，它会搜索系统PATH环境变量中的可执行文件路径，以找到正确的Python解释器。在Windows系统中，通常会使用`#!/usr/bin/python`或`#!/usr/bin/env python`作为文件头。

第二行是用来指定Python脚本的字符编码格式，这是非常重要的，尤其是在处理中文字符时。在Python 2中，默认的编码方式是ASCII编码，而在Python 3中，默认的编码方式是UTF-8编码。因此，如果在Python 2中使用中文字符，应该使用`# -*- coding: utf-8 -*-`来指定编码方式。

导入规范：

python有一些导入规范，简要来说如下所示：

1. 尽可能使用绝对导入。使用`import module`或`from package import module`的方式导入模块，而不是使用相对导入。相对导入可能会导致不同平台上的导入错误，因此应该尽量避免使用。
2. 避免使用通配符导入。不要使用`from module import *`的方式导入模块，这会导致命名空间污染和代码可读性降低。应该明确导入需要的模块或函数。
3. 将导入放在文件的开头。将所有的导入语句放在文件的开头，这样可以更清晰地表达模块之间的依赖关系，并且方便其他人阅读和维护代码。
4. 给导入的模块使用标准的别名。对于一些常用的模块，可以使用标准的别名来提高代码可读性。例如，可以将`numpy`模块的别名定义为`np`，将`pandas`模块的别名定义为`pd`等。
5. 在导入语句之间添加空行。为了提高代码的可读性，可以在导入语句之间添加一个空行。

命名规范：

1. 使用小写字母。变量、函数、模块和类的名称应该全部使用小写字母，单词之间使用下划线 `_` 分隔。
2. 用下划线分隔单词。当命名包含多个单词时，应该使用下划线 `_` 分隔单词，而不是使用驼峰命名法。
3. 不要使用保留字。不要使用Python语言中的保留字（如`if`、`else`、`for`等）作为变量、函数、模块和类的名称。
4. 遵循约定俗成的命名规则。在Python社区中，有一些约定俗成的命名规则，如使用`self`作为实例方法的第一个参数名，使用单一下划线 `_` 表示私有属性等。
5. 用名词表示变量和函数，用动词表示方法。变量和函数的名称应该用名词表示其含义，而方法的名称应该用动词表示其操作。
6. 类名使用大写字母开头。类的名称应该使用大写字母开头，多个单词之间使用驼峰命名法。

此类规范还有很多，此处不再一一列举。

2.1.2 pylint使用

在本地文件夹创建 `PylintConfig.conf` 的文件，来指定 Pylint 检查的样式，然后创建 `pylintTest.py` 文件来检查主程序的规范性，其内容如下：

```
import pylint.lint

pylint_opts = [ '--rcfile= PylintConfig.conf' , '-ry' , './ main.py']

pylint.lint.Run((pylint_opts))
```

运行该文件，可以看到 `main.py` 规范性基本符合 Google 的编程规范要求：

```
***** Module PylintConfig.conf
PylintConfig.conf:1:0: E0015: Unrecognized option found: profile, files-output,
comment, zope, required-attributes, bad-functions, ignore-iface-methods
(unrecognized-option)
```

Report

=====

78 statements analysed.

Statistics by type

type	number	old number	difference	%documented	%badname	
module	1	NC	NC	100.00	0.00	
class	1	NC	NC	100.00	0.00	
method	5	NC	NC	100.00	0.00	
function	3	NC	NC	100.00	0.00	

118 lines have been analyzed

Raw metrics

type	number	%	previous	difference	
code	82	69.49	NC	NC	
docstring	23	19.49	NC	NC	

comment	2	1.69	NC	NC	
empty	11	9.32	NC	NC	

Duplication

	now	previous	difference
nb duplicated lines	0	NC	NC
percent duplicated lines	0.000	NC	NC

Messages by category

type	number	previous	difference
convention	0	NC	NC
refactor	0	NC	NC
warning	0	NC	NC
error	0	NC	NC

Messages

message id	occurrences
------------	-------------

Your code has been rated at 10.00/10

2.2 错误和异常处理

错误与异常的处理主要在获取终端输入部分：控制程序的输入，实现由输入直接生成结果。定义合法的输入形式为数组、元组及整数，其中数组可以接受整数或浮点数的形式。当输入形如'[1,2,3,4]'、'(1,2,3,4)'、'1'时符合规范，其余形式均视为错误输入，需要进行错误处理。实现代码如下：

```
def get_in():
    """获取输入数组
    包括错误处理
    """
    inlist=[]
    try:
        inlist=literal_eval(input("please input a One-dim Or Two-dim array: "))
    except (SyntaxError, ValueError):
        print("error input! ")
    except TypeError:
        print("Format error!")
    if isinstance(inlist,tuple):
        inlist=list(inlist)
    elif isinstance(inlist,int):
        inlist=[inlist]
    elif isinstance(inlist, list):
        temp=np.array(inlist)
        if len(temp.shape) == 1:
            for index in inlist:
                try:
                    assert isinstance(index, (int, float))
                except AssertionError:
                    print("error input!")
        elif len(temp.shape)==2:
            for index_1 in inlist:
                for index_2 in index_1:
                    try:
                        assert isinstance(index_2,(int,float))
                    except AssertionError:
                        print("error input!")
        else:
            print("本程序仅支持一维数组和二维数组")
            return TypeError
    else:
        print("error input! ")
        return TypeError
    return inlist
```

2.3 两个原则

2.3.1 单一职责原则

在本次实验设计中，MaxProduct 类专门用来来进行数据的计算，不参与程序的读入数据或处理数据类型等问题的部分，变化的原因只有需要实现的求值功能的增删改才会对这一部分进行变化；pipeline 函数专门用于统筹计算流程的排布，当需要拓展功能时，只需添加相应模块，并在 pipeline 模块中重新排布计算流程即可；get_in 函数为获取终端输入的函数，该函数可对输入数据进行处理并进行错误处理。所有的变化的原因只有输入数据类型变化才会有变化，体现了单一职责原则。

2.3.2 开放-封闭原则

MaxProduct 模块的成员函数 calculate 的职责是计算乘积最大子数组，这一部分的内容是不必进行更改的，其他维度的最大子数组和都是依托于这一部分层层累积叠加，是可以扩展的，满足开放-封闭原则。

2.4 可扩展性

该实验在基础要求实现的情况下，增加了计算二维数组的乘积最大子数组的功能。且对于维度很大的数组，使用 list 类型的元素在理论上可以对数组维度和数组长度进行无限延伸，且 list 类型支持多种类型的数据混用，可以实现浮点数和整数的混合数组。在未来，可能会实现一维数组的求值过程的可视化打印显示。

更新后的代码如下所示：

```
def calculate(self):
    """对输入数列处理后计算MP"""
    if self.shape==1:
        self.num=self.nums
        print(self.num)
        return self.calculate_list()
    if self.shape==2:
        ans_list=[]
        for i in range(self.shape):
            self.num=self.nums[i]
            ans_list.append(self.calculate_list())
        return max(ans_list)
    print("Program error!")
    return TypeError

def get_in():
    """获取输入数组
    包括错误处理
    """
    inlist=[]
    try:
        inlist=literal_eval(input("please input a one-dim or two-dim array: "))
    except (SyntaxError, ValueError):
        print("error input! ")
    except TypeError:
        print("Format error!")
    if isinstance(inlist,tuple):
```

```

    inlist=list(inlist)
elif isinstance(inlist,int):
    inlist=[inlist]
elif isinstance(inlist, list):
    temp=np.array(inlist)
    if len(temp.shape) == 1:
        for index in inlist:
            try:
                assert isinstance(index, (int, float))
            except AssertionError:
                print("error input!")
    elif len(temp.shape)==2:
        for index_1 in inlist:
            for index_2 in index_1:
                try:
                    assert isinstance(index_2,(int,float))
                except AssertionError:
                    print("error input!")
    else:
        print("本程序仅支持一维数组和二维数组")
        return TypeError
else:
    print("error input! ")
    return TypeError
return inlist

```

其中，`calculate` 用于判断数组的维度，如果是一维则直接调用 `calculate_list` 函数，否则逐维调用 `calculate_list` 计算乘积最大子数组，放入数组中，最后返回数组中的最大值。

2.5 性能分析

2.5.1 算法设计

2.5.1.1 算法1

该算法需要考虑不同的情况，首先考虑包含 0 和不包含 0 的情况。

当数列中不包含 0 时，有以下三种情况：

1. 全为正数：直接所有数结果相乘即可
2. 偶数个负数：负负得正，直接所有数相乘即可
3. 奇数个负数：对于该情况，处理思路是给出两个数组，一个是正序数组，另一个是逆序数组，将两个数组均从左到右累乘，然后找出两个数组中的最大值，即为最大连乘结果。

当数列中包含 0 时，处理思路同奇数个负数，不同的是一旦遇到 0，那么下一个数就为其本身，表示区间从 0 处断开。

在算法实现上，新建一个 `nums` 的反顺序数组 `nums_reverse`，然后连乘 `nums[i-1]` or 1。当 `nums[i-1] != 0` 时，或运算结果就是 `nums[i-1]`，当 `nums[i-1] == 0` 时，或运算结果就是 1，这样就实现了 0 后的第一个数字不与 0 相乘。

```

def calculate(self):
    """
    计算乘积最大子数组的值
    """
    nums_reverse = self.num[::-1]
    for i in range(1, len(self.num)):
        self.num[i] *= self.num[i-1] or 1 # or 1的作用为，当nums[i-1]==0时，nums[i]
        乘等自身
        nums_reverse[i] *= nums_reverse[i-1] or 1
        self.ans_dp = max(max(self.num), max(nums_reverse))
    return self.ans_dp

```

2.5.1.2 算法2

该方法为动态规划。求数组中子区间的最大乘积，对于乘法，需要注意，负数乘以负数，会变成正数，所以需要维护两个变量，当前的最大值，以及最小值，最小值可能为负数，但没准下一步乘以一个负数，当前的最大值就变成最小值，而最小值则变成最大值了。

因此动态规划方程设计为以下形式：

$$\begin{aligned}
 max_{dp}[i+1] &= \max(max_{dp}[i] * A[i+1], A[i+1], min_{dp}[i] * A[i+1]) \\
 min_{dp}[i+1] &= \min(min_{dp}[i] * A[i+1], A[i+1], max_{dp}[i] * A[i+1]) \\
 dp[i+1] &= \max(dp[i], max_{dp}[i+1])
 \end{aligned}$$

还要注意元素为 0 的情况，如果 A[i] 为 0，那么 max_dp 和 min_dp 都为 0，需要从 A[i+1] 重新开始。

该逻辑的代码实现在 calculate 部分，如下：

```

def calculate(self):
    """
    计算乘积最大子数组的值
    """
    if len(self.num) <= 1:
        self.ans_dp=self.num[0]
    self.max_dp,self.min_dp,self.ans_dp = self.num[0],self.num[0],self.num[0]
    for i in range(1, len(self.num)):
        self.max_dp,self.min_dp= max(self.max_dp * self.num[i], self.num[i],
        self.min_dp * self.num[i]),\
                                min(self.max_dp * self.num[i],
        self.num[i],self.min_dp * self.num[i])
        self.ans_dp = max(self.ans_dp, self.max_dp)
    return self.ans_dp

```

2.5.2 性能对比

利用 yappi 工具对两种算法运行的时间进行测试，测试样例分别为 100000 个整数组成的数组和 100000 个浮点数组成的数组，测试结果如图3和4所示。

可以很明显看到执行 main.py 程序的开销集中在 calculate 函数，即计算乘积最大子数组。算法 2 相比算法 1 的加速比为 6.25。

分析两种算法的执行逻辑，发现两种算法的时间复杂度均为 $O(n)$ ，但两种算法的执行效率却是天差地别。为了探究其具体原因，我将算法一拆分成更小的执行模块，使用通过 yappi 工具查看各子模块的运行时间。

将其拆分成以下三个子模块：

```
def calculate(self):
    """
    计算乘积最大子数组的值
    """
    self.step1()
    self.step2()
    self.step3()
def step1(self):
    self.nums_reverse = self.num[: : -1]
def step2(self):
    for i in range(1, len(self.num)):
        self.num[i] *= self.num[i-1] or 1
        self.nums_reverse[i] *= self.nums_reverse[i-1] or 1
def step3(self):
    self.ans_dp = max(max(self.num), max(self.nums_reverse))
```

运行结果如下：

算法1的运行结果：

```
6 function calls in 50.828 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      16.516    16.516    60.172    60.172  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:46(pipeline)
1       0.000     0.000    60.172    60.172  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:53(get_list)
1      34.312    34.312    43.656    43.656  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:26(MaxProduct.caculate)
1       0.000     0.000     0.000     0.000  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:22(MaxProduct.set)
1       0.000     0.000     0.000     0.000  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:42(MaxProduct.get_ans)
1       0.000     0.000     0.000     0.000  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:15(MaxProduct.__init__)

12 function calls in 50.922 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
2      16.531     8.266    60.266    30.133  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:46(pipeline)
2       0.000     0.000    60.266    30.133  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:53(get_list)
2      34.391    17.195    43.734    21.867  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:26(MaxProduct.caculate)
2       0.000     0.000     0.000     0.000  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:22(MaxProduct.set)
2       0.000     0.000     0.000     0.000  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:42(MaxProduct.get_ans)
2       0.000     0.000     0.000     0.000  C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:15(MaxProduct.__init__)
```

算法2的运行结果：

```

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    8.375    8.375 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:40(pipeline)
1      0.000    0.000    8.375    8.375 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:47(get_list)
1      7.969    7.969    8.375    8.375 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:26(MaxProduct.caculate)
1      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:36(MaxProduct.get_ans)
1      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:22(MaxProduct.set)
1      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:15(MaxProduct.__init__)

12 function calls in 8.406 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
2      0.000    0.000    9.016    4.508 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:40(pipeline)
2      0.000    0.000    9.016    4.508 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:47(get_list)
2      8.406    4.203    9.016    4.508 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:26(MaxProduct.caculate)
2      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:36(MaxProduct.get_ans)
2      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:22(MaxProduct.set)
2      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:15(MaxProduct.__init__)

```

分布执行运行时间：

```

1      0.016    0.016   45.141   45.141 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:27(MaxProduct.caculate)
1     25.281   25.281   25.281   25.281 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:41(MaxProduct.step2)
1      0.016    0.016   19.844   19.844 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:45(MaxProduct.step3)
1      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:50(MaxProduct.get_ans)
1      0.000    0.000    0.000    0.000 C:\Users\civilizwa\Desktop\软件工程\乘积最大子数组\main.py:39(MaxProduct.step1)

```

由此可以看出，时间主要开销在 step2 和 step3，其中 step2 的理论时间复杂度为 $O(n)$ ，step3 用来求最大值。合理推测虽然两种算法的时间复杂度均为 $O(n)$ ，但是当数据量过大时，其 n 前的系数 k 也会影响实际执行效率。

通过这两个算法的比较发现，如果想要提高一个程序的执行效率，可以通过优化核心代码对其进行优化。并且可以看出不同的算法其执行效率差异巨大。因此在软件开发中具备优秀的算法能力将会提高开发软件的整体性能

2.6 单元测试

2.6.1 测试设计

对该程序进行黑盒测试。首先对于数据的正负值，设计如下：

- 全为正数（最大子数组是其本身）
- 全为负数（最大子数组是其负值的最大值）
- 以及正值负值均包含的测试样例

对于整数和浮点数，设计如下：

- 全为整数的测试样例
- 全为浮点数的测试样例
- 整数和浮点数混合的测试样例

测试包含了可能出现的几种数据类型；于此同时，我们也进行了输入样例的测试包括上面提到的各种输入类型，如：

- 普通的一位一维 list

- 多个一位 tuple
- 单独的数字、set 类型的输入

同时也进行了错误输入的测试，比如：

- 输入的元素非要求的列表类型
- 列表类型中的元素非整数或浮点数（如 string）
- 矩阵的维度为二维以上等。

2.6.2 测试代码

具体测试样例如下：

```
class Testing(unittest.TestCase):
    """
    测试
    """

    def test_pipeline(self):
        """一维"""
        self.assertEqual(main.pipeline([-1, 2, 3, -4]),24)
        self.assertEqual(main.pipeline([1, 2, 3, 4]),24)
        self.assertEqual(main.pipeline([-1, 2, -5, 3, -4]),120)
        self.assertEqual(main.pipeline([-1, 20, 0, 30, -4]),30)
        self.assertEqual(main.pipeline([-2, -3, -5, -1, -9]),135)
        self.assertEqual(main.pipeline([-2.0, -3.0, -5.0, -1.0, -9.0]),135.0)
        self.assertEqual(main.pipeline([-2.0, -3.0, -5.0, -1, -9]),135.0)
        """二维"""
        self.assertEqual(main.pipeline([[1,2,3],[0,1,2]]),6)
        self.assertEqual(main.pipeline([[1, 2, 3,4], [0, 1, 2,4]]), 24)
        self.assertEqual(main.pipeline([[-1.0,-2.0,7],[1,2,0]]),14.0)

    @patch('builtins.input')
    def test_in(self, mock_input):
        mock_input.return_value = '1'
        self.assertEqual(main.get_in(), [1])

        mock_input.return_value = '[1, 2]'
        self.assertEqual(main.get_in(), [1, 2])

        mock_input.return_value = '[1, 2, 0]'
        self.assertEqual(main.get_in(), [1, 2, 0])

        mock_input.return_value = '[[1, 2,-3],[0,1,2]]'
        self.assertEqual(main.get_in(), [[1, 2,-3],[0,1,2]])

        mock_input.return_value = '(1, 2)'
        self.assertEqual(main.get_in(), [1, 2])

    @patch('builtins.input')
```


2.6.4 测试覆盖率

而对于测试覆盖率，我使用 `coverage` 来完成。输入 `coverage run unittestTest.py` 和 `coverage report`，即可得到以下结果：

```
PS D:\Archive\大三下\软件工程\最大子数组\new> coverage report
Name                Stmts   Miss  Cover
-----
main.py              78      13    83%
unittestTest.py      47       0   100%
-----
TOTAL                125      13    90%
PS D:\Archive\大三下\软件工程\最大子数组\new>
```

总体覆盖率为90%，整体语句基本都有覆盖。

在main.py中未能覆盖的语句如下所示：

```
52 |         print("Program error!")
53 |         return TypeError
54 |
55 |
56 |     def get_ans(self):
57 |         """返回计算结果"""
58 |         return self.ans_dp
59 |
60 | def pipeline(input_nums):
61 |     """计算流水线"""
62 |     my_mp=MaxProduct()
63 |     my_mp.set(input_nums)
64 |     return my_mp.calculate()
65 |
66 | def get_list(the_set):
67 |     """
68 |     :param the_set: 要处理的集合
69 |     :return:最大的数
70 |     """
71 |     return pipeline(the_set)
72 |
```

```

82     except TypeError:
83         print("Format error!")
84     if isinstance(inlist,tuple):
85         inlist=list(inlist)
86     elif isinstance(inlist,int):
87         inlist=[inlist]
88     elif isinstance(inlist, list):
89         temp=np.array(inlist)
90         if len(temp.shape) == 1:
91             for index in inlist:
92                 try:
93                     assert isinstance(index, (int, float))
94                 except AssertionError:
95                     print("error input!")
96     elif len(temp.shape)==2:
97         for index_1 in inlist:
98             for index_2 in index_1:
99                 try:
100                     assert isinstance(index_2,(int,float))
101                 except AssertionError:
102                     print("error input!")
103     else:
104         print("本程序仅支持一维数组和二维数组")
105         return TypeError
106     else:
107         print("error input! ")
108         return TypeError
109     return inlist

112 if __name__=='__main__':
113     #获取一组数, 用list存储
114     nums=get_in()
115     """执行计算"""
116     print(pipeline(nums))

```

可以看到，基本为 `TypeError` 语句。而 `main` 函数中并未得到调用，覆盖的意义不大。

2.6.5 缺陷报告

缺陷标题：浮点型舍入误差

缺陷编号：1

所在模块：MaxProduct类

缺陷标题：浮点型舍入误差

缺陷描述：当输入为浮点型时，可能存在舍入误差使得计算结果与真实值不一致

发现者：Yunmei Guan

日期：2023/5/14

缺陷状态：Assigned

指派给：Yunmei Guan

版本：1.0

严重程度：high（由于浮点功能非基础功能部分要求，此处不影响交付）

优先级：high

测试环境 python 3.9.11

重现方法：向程序中输入数组 [1,2.3,3], 预期结果为 6.9，实际结果为 6.899999999999995

建议解决方法：使用 Python 中的 decimal 模块来进行浮点数计算，该模块提供了更高精度的计算功能，可以有效避免精度不足的问题

```
please input a One-dim Or Two-dim array: [1, 2.3, 3]  
[1, 2.3, 3]  
6.899999999999995
```

3 扩展部分

3.1 数据

3.1.1 二维数组

从数据的维度扩展方面考虑，本程序已经扩展了处理二维数组的能力。该处理的核心逻辑是逐行计算最大子数组的乘积，然后在每行的最大子数组中筛选出最大的成绩，具体代码如下：

```
def calculate(self):  
    """对输入数列处理后计算MP"""  
    if self.shape==1:  
        self.num=self.nums  
        print(self.num)  
        return self.calculate_list()  
    ### 扩展部分：二维数组  
    if self.shape==2:  
        ans_list=[]  
        for i in range(self.shape):  
            self.num=self.nums[i]  
            ans_list.append(self.calculate_list())  
        return max(ans_list)  
    print("Program error!")
```

```
return TypeError
```

3.2.2 高精度计算

从数据本身的属性扩展方面出发，考虑当输入数据或者计算中出现数据过大，可能出现溢出错误。本程序实现时，直接调用的函数是 `input`，而 `input` 函数自带高精度算法，直接可以处理输入数据过大的情况。一个高精度计算的运行结果如下所示：

```
please input a One-dim Or Two-dim array: [12124436323234, 23423234, -2341.2342, 3244.333]
[12124436323234, 23423234, -2341.2342, 3244.333]
283993509117209618756
```

3.2.3 算法应用领域扩展

从数据的其他属性扩展方面出发，考虑其具体应用。计算最大连续子数组乘积的算法在很多实际场景中都有应用，比如：

1. 金融分析领域：在股票交易分析中，可以利用最大连续子数组乘积算法来寻找最优的交易策略。通过对历史数据的分析，可以识别出最佳的买入和卖出时机，以获得最大的收益。这种算法还可以用于评估不同的投资组合，以确定哪些股票或资产应该被包括在投资组合中，以最大限度地提高收益率。
2. 在图像处理中，我们通常利用最大连续子数组乘积算法来寻找图像中的最大连通区域。引入抽象，将输入的数据修改为不同图像这一概念，可以扩展的处理不同类图像的寻找最大连通区域问题。
2. 在自然语言处理领域，我们通常利用最大连续子数组乘积算法，寻找具有最大影响力或重要性的短语或单词序列，以便计算机能够理解和处理。比如文本分类、关键词提取、文本摘要、命名体识别等，都可以用该算法识别文本中最相关/最重要的单词序列。
3. 工程优化领域：在工程优化中，可以使用最大连续子数组乘积算法来确定最优的参数组合。例如，在自动化制造中，可以使用该算法来确定最佳的生产参数，以获得最大的生产效率。此外，该算法还可以应用于能源管理，以确定最佳的能源利用方式，以减少能源浪费。
4. 机器学习领域：在机器学习中，最大连续子数组乘积算法可以用于特征选择和数据降维。特征选择是指从原始数据中选择最相关的特征以提高模型性能。该算法可以通过识别最相关的特征子集来实现。此外，该算法还可以用于数据降维，以减少数据集的维度并提高模型的计算效率。

3.2 需求

3.2.1 从文件读入

其实在大部分实际运用的过程中，读入数据都需要从文件中读入，而非用户手动输入。基于此，我增加了从文件中读入输入的扩展。修改IO类的输入函数 `get_input`，增加从文件读入数据的方式。

```

# get_input()函数中
method = input("please choose the method to input files: read from
files(INPUT'R') or type manually(INPUT'D)")
if method == "R":
    self.get_intput_from_file()
else:
    self.get_intput_from_console()

# get_input_from_file()函数
def get_intput_from_file(self):
    """从文件读入数组"""
    filename = input("请输入文件名: ")
    with open(filename, "w", encoding='utf-8') as file:
        self.nums = literal_eval(file.read())

```

3.2.2 重复计算

有时，我们需要利用一个程序做好多次重复的工作，而频繁运行文件又会带来另外的系统开销，因此我们希望这个程序能够像计算器一样重复进行计算。

需要在main函数中添加一个循环接收用户操作指令，使得一次调用可以允许多次计算。其代码逻辑为：进入循环后，先使用 IO类获取用户输入，然后将输入交给计算类并调用其计算函数，最后通过 IO 类输出计算结果。如果用户根据提示输入“more”，就进行下一轮获取输入、计算乘积的过程；如果输入“exit”，程序退出。

具体代码如下所示：

```

if __name__ == '__main__':
    DOEXIT = False
    while not DOEXIT:
        nums=get_in()
        """执行计算"""
        print(pipeline(nums))
        # 循环获取用户命令
        while True:
            cont = input("是否继续？请输入“more”以继续计算，输入“exit”以结束程序：")
            if cont == 'more':
                break
            if cont == 'exit':
                DOEXIT = True
                print("感谢您的支持，再见！")
                break
            print("请输入合法命令！")
            continue

```

3.3 用户

3.3.1 语言界面

对于用户来说，用户可能使用不同的语言，因此可以为该程序提供CN和EN两种扩展。在 `main` 函数中让用户进行选择，之后便可根据不同的语言打印不同的提示信息；

```
# main函数中的语言选择语句
if __name__ == "__main__":
    ...
    LANGUAGE = input("请选择语言 (CN/EN)，CN代表中文，EN代表英文:")
    while not DOEXIT:
        ...
# 根据语言选择变量打印不同提示信息的示例
if LANGUAGE == "EN":
    print("This program only support one- or two-dimentional array")
else:
    print("本程序仅支持一维数组和二维数组")
```

2.3.2 错误处理

对于代码运行中出现的错误，必须进行必要的错误处理，即打印对应的信息告诉用户错误类型，以便让他们理解本程序的用法。一个示例为：

```
temp=np.array(inlist)
if len(temp.shape) == 1:
    for index in inlist:
        try:
            assert isinstance(index, (int, float))
        except AssertionError:
            print("error input!")
elif len(temp.shape)==2:
    for index_1 in inlist:
        for index_2 in index_1:
            try:
                assert isinstance(index_2,(int,float))
            except AssertionError:
                print("error input!")
```

3.4 软件构建

3.4.1 多平台使用

从软件构建扩展方面考虑，由于 Python 为解释性语言，在安装了相同版本解释器的机器后，Python 程序应当是平台无关的。python支持的平台有：

1. Windows：Python提供了Windows平台的安装包，可以在Windows上运行Python程序。
2. macOS：Python也提供了macOS平台的安装包，可以在Mac上运行Python程序。
3. Linux：Python是Linux系统的一部分，因此Linux平台上通常都已经预装了Python。此外，Python也可以在其他Linux发行版上使用，如Ubuntu、Debian等。
4. Web：Python可以用于Web开发，如Django和Flask等框架可以在Web服务器上运行。

5. 移动设备：Python可以用于开发移动应用程序，如Kivy等框架可用于Android和iOS。

总之，Python几乎可以在任何平台上使用。而本程序在编写过程中设计了MaxProduct类，便于在任何的平台开发和扩展。

4 附录

该部分为我个人在尝试建立一个成功的按照代码规范书写代码过程中的失败案例。这些案例可以运行，且一些算法拥有较好的性能，但是他们并未按照软件工程的规范书写，因此最终没有采用。尽管如此，我将其作为一个教训，让自己之后书写代码时刻谨记规范与其他要求，不能再按照以前的那种随心所欲的模式书写代码了。篇幅较长，可不看。

4.1 暴力法

暴力搜索，即在 $C(n,2)$ 种可能的方法中寻找出乘积最大的一种。

4.1.1 代码实现

```
def max_product_subarray(nums):
    """计算乘积最大的子数组

    Args:
        nums (List[int]): 整数数组

    Returns:
        Tuple[List[int], int]: 最大子数组和其乘积
    """
    max_product = nums[0]
    max_left = 0
    max_right = 0
    n = len(nums)
    if n == 0:
        return None
    for i in range(n):
        for j in range(i, n):
            arr_product = 1
            for k in range(i, j + 1):
                arr_product *= nums[k]
            if arr_product > max_product:
                max_product = arr_product
                max_left = i
                max_right = j
    return nums[max_left:max_right + 1], max_product


def print_max_product_subarray(nums):
    """打印乘积最大的子数组和其乘积

    Args:
        nums (List[int]): 整数数组
    """
    max_subarray, max_product = max_product_subarray(nums)
```

```
print("最大子数组是: {0} , 最大子数组乘积为: {1}".format(max_subarray, max_product))
```

```
if __name__ == "__main__":  
    nums = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]  
    print_max_product_subarray(nums)
```

4.1.2 pylint测试

使用pylint对原始代码进行测试

```
PS D:\Archive\大三下\软件工程\最大子数组\max-subarray> pylint .\force.py  
***** Module force  
force.py:38:0: C0304: Final newline missing (missing-final-newline)  
force.py:1:0: C0114: Missing module docstring (missing-module-docstring)  
force.py:1:25: W0621: Redefining name 'nums' from outer scope (line 37) (redefined-outer-name)  
force.py:13:4: C0103: Variable name "n" doesn't conform to snake_case naming style (invalid-name)  
force.py:26:31: W0621: Redefining name 'nums' from outer scope (line 37) (redefined-outer-name)  
force.py:33:10: C0209: Formatting a regular string which could be a f-string (consider-using-f-string)  
  
-----  
Your code has been rated at 7.14/10
```

添加参数 `-ry` , 尝试生成详细报告。

```
***** Module force  
force.py:38:0: C0304: Final newline missing (missing-final-newline)  
force.py:1:0: C0114: Missing module docstring (missing-module-docstring)  
force.py:1:25: W0621: Redefining name 'nums' from outer scope (line 37) (redefined-outer-name)  
force.py:13:4: C0103: Variable name "n" doesn't conform to snake_case naming style (invalid-name)  
force.py:26:31: W0621: Redefining name 'nums' from outer scope (line 37) (redefined-outer-name)  
force.py:33:10: C0209: Formatting a regular string which could be a f-string (consider-using-f-string)
```

Report

=====

21 statements analysed.

Statistics by type

type	number	old number	difference	%documented	%badname
module	1	1	=	0.00	0.00
class	0	NC	NC	0	0
method	0	NC	NC	0	0
function	2	2	=	100.00	0.00

40 lines have been analyzed

Raw metrics

+-----+					
type	number	%	previous	difference	
+=====+					
code	23	57.50	NC	NC	
+-----+					
docstring	13	32.50	NC	NC	
+-----+					
comment	0	0.00	NC	NC	
+-----+					
empty	4	10.00	NC	NC	
+-----+					

Duplication

+-----+				
	now	previous	difference	
+=====+				
nb duplicated lines	0	0	0	
+-----+				
percent duplicated lines	0.000	0.000	=	
+-----+				

Messages by category

+-----+			
type	number	previous	difference
+=====+			
convention	4	4	4
+-----+			
refactor	0	0	0
+-----+			
warning	2	2	2
+-----+			
error	0	0	0
+-----+			

Messages

```
+-----+
|message id          |occurrences |
+=====+
|redefined-outer-name|2          |
+-----+
|missing-module-docstring|1          |
+-----+
|missing-final-newline|1          |
+-----+
|invalid-name        |1          |
+-----+
|consider-using-f-string|1          |
+-----+
```

Your code has been rated at 7.14/10 (previous run: 7.14/10, +0.00)

通过观察pylint的报告，我们可以依据要求作如下修改：

1. 在代码最后添加一个空行，以符合C0304规则。
2. 添加一个模块(docstring)级别的文档字符串，以符合C0114规则。
3. 将变量名n改为 snake_case 风格，以符合C0103规则。
4. 修改 max_child 函数的参数名 nums 为 array，以避免和全局变量重名，符合W0621规则。
5. 使用 f-string 格式化输出语句，以符合C0209规则。

```
def max_child(array):
    """
    计算给定数组中乘积最大的子数组
    :param array: 一个整数数组
    :return: 一个元组，包含最大子数组和其对应的乘积
    """
    max_product = 0
    left = 0
    right = 0
    n = len(array)
    if n == 0:
        return None
    for i in range(0, n):
        for j in range(i, n):
            subarray_product = 1
            for k in range(i, j+1):
                subarray_product = subarray_product * array[k]
            if subarray_product > max_product:
                max_product = subarray_product
```

```

        left = i
        right = j
    return array[left:right + 1], max_product

if __name__ == '__main__':
    input_array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4,
7]

    max_subarray, max_product = max_child(input_array)
    print(f"最大子数组是: {max_subarray} , 最大子数组之积为: {max_product}")

```

在此之后我们再次尝试生成pylint报告，如下所示：

```

Your code has been rated at 7.14/10 (previous run: 7.14/10, +0.00)

PS D:\Archive\大三下\软件工程\最大子数组\max-subarray> pylint .\force.py
***** Module force
force.py:11:4: W0621: Redefining name 'max_product' from outer scope (line 28) (redefined-outer-name)
force.py:14:4: C0103: Variable name "n" doesn't conform to snake_case naming style (invalid-name)

Your code has been rated at 8.95/10 (previous run: 7.14/10, +1.80)

PS D:\Archive\大三下\软件工程\最大子数组\max-subarray> |

```

可以看出目前pylint得分已有显著提升。

4.1.3 profile测试

```

import profile

# 测试force.py
import force
profile.run("force.max_child([2, 3, -2, 4])")

```

```

PS D:\Archive\大三下\软件工程\最大子数组\max-subarray> & D:/Python39/python.exe d:/Archive/大三下/软件工程/最大子数组/max-subarray/try_profile.py
6 function calls in 0.016 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.000    0.000    0.000    0.000  :0(exec)
1      0.000    0.000    0.000    0.000  :0(len)
1      0.016    0.016    0.016    0.016  :0(setprofile)
1      0.000    0.000    0.000    0.000  <string>:1(<module>)
1      0.000    0.000    0.000    0.000  force.py:5(max_child)
1      0.000    0.000    0.016    0.016  profile:0(force.max_child([2, 3, -2, 4]))
0      0.000    0.000    0.000    0.000  profile:0(profile)

```

profile的每个指标解释如下：

- ncalls：函数被调用的次数。
- tottime：函数总运行时间，除去函数中调用子函数的运行时间。
- percall：平均每次函数调用的运行时间，等于tottime/ncalls。
- cumtime：函数总运行时间，包括函数中调用子函数的运行时间。
- percall：平均每次函数调用的运行时间，等于cumtime/ncalls。
- filename: lineno(function)：函数名称及其在源代码中的行号。

在这里我们可以看到，`max_child` 函数只被调用了1次，该函数的总运行时间为0.016秒，平均每次函数调用的运行时间也是0.016秒。同时，该函数中没有调用任何其他子函数，因此tottime和cumtime的值相等，为0.016秒。在这里只有一个函数，且`max_child` 函数本身耗时极少。

4.1.4 unittest测试

测试用例设计思路如下：

- 1. 空数组作为输入，期望函数返回一个空数组和None。
- 2. 数组只有一个元素作为输入，期望函数返回原数组和该元素本身。
- 3. 数组中所有元素均为正数，期望函数返回原数组和所有元素的积。
- 4. 数组中所有元素均为负数，期望函数返回原数组中的乘积最大子数组和他们的积。
- 5. 数组中既有正数也有负数，期望函数返回最大的子数组和其对应的乘积。
- 6. 数组中存在0，期望函数返回原数组中的乘积最大子数组和他们的积。
- 7. 数组中存在多个最大子数组，期望函数返回其中任意一个最大子数组和其对应的乘积。
- 8. 数组中正数、负数不规则交替分布，期望函数返回最大的子数组和其对应的乘积。
- 9. 数组中正数、负数、0不规则交替分布且较长，期望函数返回最大的子数组和其对应乘积。

输入	期望输出
[]	[], None
[1]	[1], 1
[1, 2, 3]	[1, 2, 3], 6
[-1, -2, -3]	[-2, -3], 6
[2, -1, 3, -2, 4, -1]	[2, -1, 3, -2, 4], 48
[1, 0, 2, 0, 3]	[3], 3
[1, 2, 3, 0, 1, 2, 3]	[1, 2, 3], 6
[1, 2, -3, 2, -3, 2, -5]	[1, 2, -3, 2, -3, 2], 72
[2, -3, 1, 2, -1, 5, -3, 2, -4, 3, 0, 1, 2]	[2, -3, 1, 2, -1, 5, -3, 2, -4, 3], 4320

构造的单元测试代码如下：

```
import unittest
from force import max_child

class TestMaxChild(unittest.TestCase):

    def test_max_child(self):
        # 测试用例设计思路：
        # 1. 空数组作为输入，期望函数返回一个空数组和0。
        # 2. 数组只有一个元素作为输入，期望函数返回原数组和该元素本身。
        # 3. 数组中所有元素均为正数，期望函数返回原数组和所有元素的积。
        # 4. 数组中所有元素均为负数，期望函数返回原数组中的乘积最大子数组和他们的积。
        # 5. 数组中既有正数也有负数，期望函数返回最大的子数组和其对应的乘积。
        # 6. 数组中存在0，期望函数返回原数组中的乘积最大子数组和他们的积。
```

```

# 7. 数组中存在多个最大子数组，期望函数返回其中任意一个最大子数组和其对应的乘积。
# 8. 数组中正数、负数不规则交替分布，期望函数返回最大的子数组和其对应的乘积。
# 9. 数组中正数、负数、0不规则交替分布且较长，期望函数返回最大的子数组和其对应乘积。

# 测试用例
test_cases = [
    {'input': [], 'output': None},
    {'input': [1], 'output': ([1], 1)},
    {'input': [1, 2, 3], 'output': ([1, 2, 3], 6)},
    {'input': [-1, -2, -3], 'output': ([-2, -3], 6)},
    {'input': [2, -1, 3, -2, 4, -1], 'output': ([2, -1, 3, -2, 4], 48)},
    {'input': [1, 0, 2, 0, 3], 'output': ([3], 3)},
    {'input': [1, 2, 3, 0, 1, 2, 3], 'output': ([1, 2, 3], 6)},
    {'input': [1, 2, -3, 2, -3, 2], 'output': ([1, 2, -3, 2, -3, 2], 72)},
    {'input': [2, -3, 1, 2, -1, 5, -3, 2, -4, 3, 0, 1, 2], 'output': ([2,
-3, 1, 2, -1, 5, -3, 2, -4, 3], 4320)}
]

for i, test_case in enumerate(test_cases):
    with self.subTest(f'test_{i}'):
        input_array = test_case['input']
        expected_output = test_case['output']
        self.assertEqual(max_child(input_array), expected_output)

if __name__ == '__main__':

    with open('./TestResult.txt', 'w') as file:
        runner = unittest.TextTestRunner(stream=file, verbosity=2)
        unittest.main(testRunner=runner)

```

运行后，其结果为：

```

1 test_max_child (__main__.TestMaxChild) ... ok
2
3 -----
4 Ran 1 test in 0.001s
5
6 OK
7

```

说明代码成功通过了所有的测试用例。

4.1.5 回答问题

1. 是否遵守编程规范，参考的哪个规范、如何检查是否遵守编程规范的？

答：在编程过程中，已尽量遵守python语言的PEP8编程规范。使用pylint来检查代码，第一次检查发现违反了C0304, C0114, W0621, C0103, W0621, C0209 规则；在对代码进行修改和重构后，代码得分为8.95/10，基本符合编程规范。

2. 是否考虑算法的可扩展性(已扩展了什么功能、未来还可以如何扩展)?

答：在这份代码中，已考虑返回值为同时返回子数组和它们的乘积（题目只要求返回乘积），更多的扩展并未考虑；因为它只需要提供了一种计算给定数组中最大子数组的乘积的算法，且该算法为暴力求解，而没有提供支持其他操作或变量类型的功能。如果要扩展该算法以支持其他操作或变量类型，需要对代码进行重构。例如，可以将算法与其他类型的数据结构结合使用，以支持更复杂的查询和操作。

3. 是否遵守了两个原则，代码的哪个部分是后续扩展功能也可以重复使用、不会修改的基础功能？

答：这个代码符合单一职责原则，因为它仅执行一项任务：找到乘积最大的子数组。它也符合开放-封闭原则，因为它是可扩展的，可以通过继承和重载来扩展其功能，而无需修改原始代码。

4. 考虑了哪些错误与异常处理

答：最初的代码没有显式处理任何错误或异常，也没有使用任何语言结构来处理可能发生的错误或异常，如try-except语句。如果数组为空，函数返回None，但并没有向调用者发出警告或错误消息。为了考虑错误和异常处理，我进行了以下尝试：

```
def max_child(array):
    """
    计算给定数组中乘积最大的子数组
    :param array: 一个整数数组
    :return: 一个元组，包含最大子数组和其对应的乘积
    """
    if not isinstance(array, list):
        raise TypeError("参数应该是一个整数数组")
    if len(array) == 0:
        return None, 0
    max_product = array[0]
    max_product_left = 0
    max_product_right = 0
    n = len(array)
    for i in range(0, n):
        if not isinstance(array[i], int):
            raise TypeError("数组元素应该是整数")
        product = array[i]
        if product > max_product:
            max_product = product
            max_product_left = i
            max_product_right = i
        for j in range(i + 1, n):
            if not isinstance(array[j], int):
                raise TypeError("数组元素应该是整数")
            product *= array[j]
            if product > max_product:
                max_product = product
                max_product_left = i
                max_product_right = j
    return array[max_product_left:max_product_right + 1], max_product

if __name__ == '__main__':
    try:
        # input_array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22,
        # 15, -4, 7]
        input_array = [2, -3, 1, 2, -1, 5, -3, 2, -4, 3, 0, 1, 2]
```

```
max_subarray, max_product = max_child(input_array)
print(f"最大子数组是: {max_subarray} , 最大子数组之积为: {max_product}")
except Exception as e:
    print(f"发生了错误: {e}")
```

更改后，能够考虑以下的错误与异常：

1. 参数类型错误：如果传入的参数不是整数数组，则会抛出 `TypeError` 异常，并提示用户传入正确的参数类型。
2. 数组元素类型错误：如果传入的整数数组中包含了非整数类型的元素，则会抛出 `TypeError` 异常，并提示用户传入正确的元素类型。
5. 算法复杂度是多少？如何对代码性能进行分析？分析的结果如何？你是如何进行优化的？
答：很明显，因为使用了三个 `for` 循环，该暴力求解法的复杂度为 $O(n^3)$ 。为了对代码进行性能分析，我使用了 `profile` 工具，详见 `profile` 部分。该代码可以进一步优化的地方有：
 - 三个 `for` 循环造成了 $O(n^3)$ 的复杂度，这将成为代码中最耗时的地方。为了优化这个问题，我尝试优化了算法，降低了整体的复杂度，详见后面的动态规划法等方法。
6. 单元测试的用例设计思路、覆盖率(选择一种覆盖指标)、测试通过率分别是多少？
答：单元测试的设计思路详见 `unittest` 部分。测试通过率为100%。

4.2 动态规划法

为了输出最大乘积的子数组，我们需要在动态规划的过程中记录下每个最大值和最小值的来源。使用两个数组 `max_dp` 和 `min_dp`，其中 `max_dp[i]` 表示以 `nums[i]` 结尾的子数组中乘积最大的值，`min_dp[i]` 表示以 `nums[i]` 结尾的子数组中乘积最小的值。此外，还需要记录下每个最大值和最小值对应的子数组的起点和终点。因此使用两个数组 `max_range` 和 `min_range`，其中 `max_range[i]` 表示以 `nums[i]` 结尾的子数组中乘积最大的值对应的子数组的起点和终点，`min_range[i]` 表示以 `nums[i]` 结尾的子数组中乘积最小的值对应的子数组的起点和终点。对于 `nums[i]` 来说，有以下两种情况：

1. `nums[i]` 为正数：此时乘积最大的子数组可以由前一个乘积最大的子数组和 `nums[i]` 组成，或者由前一个乘积最小的子数组和 `nums[i]` 组成。因此，有 `max_dp[i] = max(max_dp[i-1] * nums[i], nums[i], min_dp[i-1] * nums[i])`，并且如果 `max_dp[i]` 等于 `nums[i]`，则 `max_range[i]` 为 `(i, i)`，否则 `max_range[i]` 为 `max_range[i-1]` 对应的子数组的起点和终点。
2. `nums[i]` 为负数：此时乘积最大的子数组可以由前一个乘积最小的子数组和 `nums[i]` 组成，或者由前一个乘积最大的子数组和 `nums[i]` 组成。因此，有 `max_dp[i] = max(max_dp[i-1] * nums[i], nums[i], min_dp[i-1] * nums[i])`，并且如果 `max_dp[i]` 等于 `nums[i]`，则 `max_range[i]` 为 `(i, i)`，否则 `max_range[i]` 为 `min_range[i-1]` 对应的子数组的起点和终点。

最后，乘积最大的子数组所对应的乘积即为 `max(max_dp)`，对应的子数组即为 `max_range` 中所记录子数组。

4.2.1 代码如下

动态规划的详细代码如下：

```
def maxProduct(nums):
    n = len(nums)
    max_dp = [0] * n
```

```

min_dp = [0] * n
max_range = [(0, 0)] * n
min_range = [(0, 0)] * n
max_dp[0] = min_dp[0] = nums[0]
max_range[0] = min_range[0] = (0, 0)

for i in range(1, n):
    if nums[i] > 0:
        max_dp[i] = max(max_dp[i-1] * nums[i], nums[i])
        min_dp[i] = min(min_dp[i-1] * nums[i], nums[i])
        if max_dp[i] == nums[i]:
            max_range[i] = (i, i)
        else:
            max_range[i] = (max_range[i-1][0], i)
        if min_dp[i] == nums[i]:
            min_range[i] = (i, i)
        else:
            min_range[i] = (min_range[i-1][0], i)
    else:
        max_dp[i] = max(min_dp[i-1] * nums[i], nums[i])
        min_dp[i] = min(max_dp[i-1] * nums[i], nums[i])
        if max_dp[i] == nums[i]:
            max_range[i] = (i, i)
        else:
            max_range[i] = (min_range[i-1][0], i)
        if min_dp[i] == nums[i]:
            min_range[i] = (i, i)
        else:
            min_range[i] = (max_range[i-1][0], i)

ans = float('-inf')
range_ans = ()
for i in range(n):
    if max_dp[i] > ans:
        ans = max_dp[i]
        range_ans = max_range[i]
return ans, nums[range_ans[0]:range_ans[1]+1]

if __name__ == '__main__':
    array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]
    ans, subarray = max_product(array)
    print("最大子数组是: {0} , 最大子数组乘积为: {1}".format(subarray, ans))

```

4.2.2 pylint测试

输入 `pylint -ry ./dp.py` , 得到以下输出:

```

***** Module dp
dp.py:33:0: C0303: Trailing whitespace (trailing-whitespace)
dp.py:44:0: C0304: Final newline missing (missing-final-newline)
dp.py:1:0: C0114: Missing module docstring (missing-module-docstring)

```



```

dp.py:1:0: C0116: Missing function or method docstring (missing-function-docstring)
dp.py:1:0: C0103: Function name "maxProduct" doesn't conform to snake_case naming style (invalid-name)
dp.py:34:4: W0621: Redefining name 'ans' from outer scope (line 43) (redefined-outer-name)
dp.py:2:4: C0103: Variable name "n" doesn't conform to snake_case naming style (invalid-name)
dp.py:1:0: R0912: Too many branches (13/12) (too-many-branches)
dp.py:44:6: C0209: Formatting a regular string which could be a f-string (consider-using-f-string)

```

Report

=====

36 statements analysed.

Statistics by type

type	number	old number	difference	%documented	%badname	
module	1	NC	NC	0.00	0.00	
class	0	NC	NC	0	0	
method	0	NC	NC	0	0	
function	1	NC	NC	0.00	100.00	

46 lines have been analyzed

Raw metrics

type	number	%	previous	difference	
code	42	91.30	NC	NC	
docstring	0	0.00	NC	NC	
comment	0	0.00	NC	NC	
empty	4	8.70	NC	NC	

Duplication

	now	previous	difference
nb duplicated lines	0	NC	NC
percent duplicated lines	0.000	NC	NC

Messages by category

type	number	previous	difference
convention	7	NC	NC
refactor	1	NC	NC
warning	1	NC	NC
error	0	NC	NC

Messages

message id	occurrences
invalid-name	2
trailing-whitespace	1
too-many-branches	1
redefined-outer-name	1
missing-module-docstring	1
missing-function-docstring	1
missing-final-newline	1
consider-using-f-string	1

Your code has been rated at 7.50/10

根据pylint的报告，可以做如下修改：

1. Trailing whitespace (trailing-whitespace) 修改：将行末的空格删除。
2. Final newline missing (missing-final-newline) 修改：在文件末尾添加一个空白行。
3. Missing module docstring (missing-module-docstring) 修改：在模块的开头添加文档字符串。
4. Missing function or method docstring (missing-function-docstring) 修改：在函数或方法的开头添加文档字符串。
5. Function name "maxProduct" doesn't conform to snake_case naming style (invalid-name) 修改：将函数名修改为snake_case命名风格，即使用小写字母和下划线分隔单词。
6. Redefining name 'ans' from outer scope (line 43) (redefined-outer-name) 修改：在函数内部使用另一个变量名。
7. Variable name "n" doesn't conform to snake_case naming style (invalid-name) 修改：将变量名修改为snake_case命名风格，即使用小写字母和下划线分隔单词。
8. Too many branches (13/12) (too-many-branches) 修改：考虑使用函数封装和提取公共部分等方式来减少分支语句。
9. Formatting a regular string which could be a f-string (consider-using-f-string) 修改：将字符串格式化方式修改为f-string。

尝试修改后的代码如下：

```
"""
求乘积最大子数组
"""

def max_product(nums):
    """
    求乘积最大子数组

    :param nums: 整数数组
    :type nums: List[int]
    :return: 最大乘积、最大子数组
    :rtype: Tuple[int, List[int]]
    """
    n = len(nums)
    max_dp = [0] * n
    min_dp = [0] * n
    max_range = [(0, 0)] * n
    min_range = [(0, 0)] * n
    max_dp[0] = min_dp[0] = nums[0]
    max_range[0] = min_range[0] = (0, 0)
```

```

for i in range(1, n):
    if nums[i] > 0:
        max_dp[i] = max(max_dp[i-1] * nums[i], nums[i])
        min_dp[i] = min(min_dp[i-1] * nums[i], nums[i])
        if max_dp[i] == nums[i]:
            max_range[i] = (i, i)
        else:
            max_range[i] = (max_range[i-1][0], i)
        if min_dp[i] == nums[i]:
            min_range[i] = (i, i)
        else:
            min_range[i] = (min_range[i-1][0], i)
    else:
        max_dp[i] = max(min_dp[i-1] * nums[i], nums[i])
        min_dp[i] = min(max_dp[i-1] * nums[i], nums[i])
        if max_dp[i] == nums[i]:
            max_range[i] = (i, i)
        else:
            max_range[i] = (min_range[i-1][0], i)
        if min_dp[i] == nums[i]:
            min_range[i] = (i, i)
        else:
            min_range[i] = (max_range[i-1][0], i)

ans = float('-inf')
range_ans = ()
for i in range(n):
    if max_dp[i] > ans:
        ans = max_dp[i]
        range_ans = max_range[i]
return ans, nums[range_ans[0]:range_ans[1]+1]

if __name__ == '__main__':
    array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]
    ans, subarray = max_product(array)
    print("最大子数组是: {0} , 最大子数组乘积为: {1}".format(subarray, ans))

```

之后再使用pylint进行测试:

```

***** Module dp
dp.py:46:4: W0621: Redefining name 'ans' from outer scope (line 56) (redefined-outer-name)
dp.py:14:4: C0103: Variable name "n" doesn't conform to snake_case naming style (invalid-name)
dp.py:5:0: R0912: Too many branches (13/12) (too-many-branches)
dp.py:57:6: C0209: Formatting a regular string which could be a f-string (consider-using-f-string)

-----
Your code has been rated at 8.89/10 (previous run: 7.50/10, +1.39)

```

可以看到代码的得分有一定的提高。

4.2.3 profile测试

```

import profile

# 测试dp.py
import dp
profile.run("dp.max_product([2, 3, -2, 4])")

```

测试后，结果如下所示：

```

PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray> & D:/Python39/python.exe d:/Archieve
/大三下/软件工程/最大子数组/max-subarray/dp.py
12 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    0.000    0.000  :0(exec)
1      0.000    0.000    0.000    0.000  :0(len)
3      0.000    0.000    0.000    0.000  :0(max)
3      0.000    0.000    0.000    0.000  :0(min)
1      0.000    0.000    0.000    0.000  :0(setprofile)
1      0.000    0.000    0.000    0.000  <string>:1(<module>)
1      0.000    0.000    0.000    0.000  dp.py:5(max_product)
1      0.000    0.000    0.000    0.000  profile:0(dp.max_product([2, 3, -2, 4]))
0      0.000    0.000    0.000    0.000  profile:0(profiler)

PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray>

```

尝试更换更长的数组：

```

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000   0.000   0.000   0.000 :0(exec)
      1   0.000   0.000   0.000   0.000 :0(len)
     15   0.000   0.000   0.000   0.000 :0(max)
     15   0.000   0.000   0.000   0.000 :0(min)
      1   0.000   0.000   0.000   0.000 :0(setprofile)
      1   0.000   0.000   0.000   0.000 <string>:1(<module>)
      1   0.000   0.000   0.000   0.000 dp.py:5(max_product)
      1   0.000   0.000   0.000   0.000 profile:0(dp.max_product([13, -3, -25, 20, -3,
-16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]))
      0   0.000   0.000   0.000   0.000 profile:0(profiler)

PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray>

```

可以看到，`len` 和 `max` 方法的调用次数变多了（`ncalls`），但是由于更换了复杂度更低的动态规划算法，整体的用时比暴力求解法有了很大的下降，基本时间都是0.000。

4.2.4 unittest测试

在这里，我们使用与之前相同的测试用例，测试用例设计思路见暴力法部分的unittest测试。

使用unittest测试后的结果如下：

```

PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray> & D:/Python39/python.exe d:/Archieve/大三下/软件工程/最大子数组/max-subarray/try_unittest.py
=====
ERROR: test_max_child (__main__.TestMaxChild) [test_0]
-----
Traceback (most recent call last):
  File "d:\Archieve\大三下\软件工程\最大子数组\max-subarray\try_unittest.py", line 36, in test_max_child
    self.assertEqual(max_product(input_array), expected_output)
  File "d:\Archieve\大三下\软件工程\最大子数组\max-subarray\dp.py", line 19, in max_product
    max_dp[0] = min_dp[0] = nums[0]
IndexError: list index out of range
=====
FAIL: test_max_child (__main__.TestMaxChild) [test_2]
-----
Traceback (most recent call last):
  File "d:\Archieve\大三下\软件工程\最大子数组\max-subarray\try_unittest.py", line 36, in test_max_child
    self.assertEqual(max_product(input_array), expected_output)
AssertionError: Tuples differ: ([2, 3], 6) != ([1, 2, 3], 6)

self.assertEqual(max_product(input_array), expected_output)
AssertionError: Tuples differ: ([2, -3, 2, -3, 2], 72) != ([1, 2, -3, 2, -3, 2], 72)

First differing element 0:
[2, -3, 2, -3, 2]
[1, 2, -3, 2, -3, 2]

- ([2, -3, 2, -3, 2], 72)
+ ([1, 2, -3, 2, -3, 2], 72)
?   +++

-----
Ran 1 test in 0.002s

FAILED (failures=3, errors=1)
PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray>

```

我们发现，测试用例test_6和test_7都没有通过。

这个问题的根源在于，最后返回的最大子数组部分使用的是`nums[range_ans[0]:range_ans[1]+1]`，但是这样返回的只是从`range_ans[0]`到`range_ans[1]`的切片，可能不是最大子数组。正确的返回方式是使用`nums[range_ans[0]:range_ans[1]+1]`中的元素构成的最大子数组，即与最大乘积所对应的最大子数组。因此，需要作以下修改：

```

max_sub_array = nums[range_ans[0]:range_ans[1]+1]
if not max_sub_array:
    max_sub_array = [0]
return max_sub_array, ans

```

4.3 线性方法

还有时间复杂度更好的方法。用 $[i,j]$ 表示一段子数组，如果 $[i,j]$ 对应最大子数组，那么必有 $\text{product}([i,k]) \geq 0$ ， $k=i+1, i+2, \dots, j$ 。利用这个特性，我们可以设计出时间复杂度更好的算法。

4.3.1 代码实现

详细代码如下所示：

```
import sys

def linear_find_max_subarray(array):
    """
    线性时间复杂度查找最大子数组
    """
    left = 0
    right = 1
    i = left
    j = right
    if not isinstance(array, list):
        raise TypeError("Input parameter must be a list.")
    if len(array) == 0:
        raise ValueError("Input list cannot be empty.")
    max_sum = -sys.maxsize - 1
    sum_now = array[0]
    while j < len(array):
        sum_now *= array[j]
        if sum_now > max_sum: # 元素积增大，调整相关变量记录当前元素构成的子数组
            max_sum = sum_now
            left = i
            right = j
        if sum_now <= 0: # 如果元素积小于0，那么[i:j]就不属于最大子数组的一部分
            i = j + 1
            if j != len(array) - 1:
                sum_now = array[i]
                j += 1
            else:
                sum_now = array[-1]
                j += 1
        j += 1
    return array[left:right + 1], max_sum

def maxProduct(nums):
    n = len(nums)
    if not isinstance(nums, list):
        raise TypeError("Input parameter must be a list.")
    if n == 0:
        return None
    left = 0
    curr_product = max_product = min_product = nums[0]

    for i in range(1, n):
```

```

        prev_max_product, prev_min_product = max_product, min_product
        max_product = max(nums[i], prev_max_product * nums[i], prev_min_product *
nums[i])
        min_product = min(nums[i], prev_max_product * nums[i], prev_min_product *
nums[i])
        curr_product = max(curr_product, max_product)

    if nums[i] == 0:
        curr_product = max(0, curr_product)
        left = i + 1

    return curr_product

if __name__ == '__main__':
    # input_array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4,
7]
    array = [1, 2, -3, 2, -3, 2]
    # array = [2, 3, -2, 4]
    ans = maxProduct(array)
    print(ans)

```

4.3.2 pylint测试

使用pylint进行测试，结果如下：

```

PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray> pylint -ry .\linear.py
***** Module linear
linear.py:55:0: C0304: Final newline missing (missing-final-newline)
linear.py:1:0: C0114: Missing module docstring (missing-module-docstring)
linear.py:3:29: W0621: Redefining name 'array' from outer scope (line 52)
(redefined-outer-name)
linear.py:30:0: C0116: Missing function or method docstring (missing-function-
docstring)
linear.py:30:0: C0103: Function name "maxProduct" doesn't conform to snake_case
naming style (invalid-name)
linear.py:31:4: C0103: Variable name "n" doesn't conform to snake_case naming style
(invalid-name)
linear.py:34:4: W0612: Unused variable 'left' (unused-variable)

```

Report

=====

42 statements analysed.

Statistics by type

+	+	+	+	+	+	+
type	number	old number	difference	%documented	%badname	
+	+	+	+	+	+	+
module	1	NC	NC	0.00	0.00	

class	0	NC	NC	0	0	
method	0	NC	NC	0	0	
function	2	NC	NC	50.00	50.00	

57 lines have been analyzed

Raw metrics

type	number	%	previous	difference	
code	45	78.95	NC	NC	
docstring	3	5.26	NC	NC	
comment	2	3.51	NC	NC	
empty	7	12.28	NC	NC	

Duplication

	now	previous	difference	
nb duplicated lines	0	NC	NC	
percent duplicated lines	0.000	NC	NC	

Messages by category

type	number	previous	difference	
convention	5	NC	NC	
refactor	0	NC	NC	

warning	2	NC	NC	
+-----+				
error	0	NC	NC	
+-----+				

Messages

```

+-----+
|message id          |occurrences |
+=====+
|invalid-name        |2           |
+-----+
|unused-variable     |1           |
+-----+
|redefined-outer-name|1           |
+-----+
|missing-module-docstring|1         |
+-----+
|missing-function-docstring|1       |
+-----+
|missing-final-newline  |1         |
+-----+

```

Your code has been rated at 8.33/10

根据pylint的报告，尝试进行以下修改：

1. 在文件末尾添加一个空白行，修复 C0304 错误（Final newline missing）。
2. 在文件开头添加一个模块(doc)字符串来描述该模块的作用，修复 C0114 错误（Missing module docstring）。
3. 更改函数中的变量名，将第52行的 `array` 改为 `arr`，修复 W0621 错误（Redefining name from outer scope）。
4. 为函数 `maxProduct` 添加函数(doc)字符串，修复 C0116 错误（Missing function or method docstring）。
5. 将函数名 `maxProduct` 改为 `max_product_pointer` 以符合蛇形命名规则，修复 C0103 错误（Function/variable name doesn't conform to snake_case naming style）。
6. 将变量名 `n` 改为 `num_len` 以符合蛇形命名规则，修复 C0103 错误（Function/variable name doesn't conform to snake_case naming style）。
7. 在函数 `max_product` 中删除未使用的变量 `left`，修复 W0612 错误（Unused variable）。

```
import sys
```

```

def linear_find_max_subarray(array):
    """
    线性时间复杂度查找最大子数组
    """
    left = 0
    right = 1
    i = left
    j = right
    max_sum = -sys.maxsize - 1
    if not isinstance(array, list):
        raise TypeError("Input parameter must be a list.")
    if len(array) == 0:
        raise ValueError("Input list cannot be empty.")
    sum_now = array[0]
    while j < len(array):
        sum_now *= array[j]
        if sum_now > max_sum: # 元素积增大, 调整相关变量记录当前元素构成的子数组
            max_sum = sum_now
            left = i
            right = j
        if sum_now <= 0: # 如果元素积小于0, 那么[i:j]就不属于最大子数组的一部分
            i = j + 1
            if j != len(array) - 1:
                sum_now = array[i]
                j += 1
            else:
                sum_now = array[-1]
                j += 1
        j += 1
    return array[left:right + 1], max_sum


def max_product_linear(nums):
    """
    计算一个整数数组中连续元素积的最大值
    """
    length = len(nums)
    if not isinstance(nums, list):
        raise TypeError("Input parameter must be a list.")
    if length == 0:
        return None
    curr_product = max_product = min_product = nums[0]
    for i in range(1, length):
        prev_max_product, prev_min_product = max_product, min_product
        max_product = max(nums[i], prev_max_product * nums[i], prev_min_product *
nums[i])
        min_product = min(nums[i], prev_max_product * nums[i], prev_min_product *
nums[i])
        curr_product = max(curr_product, max_product)
        if nums[i] == 0:
            curr_product = max(0, curr_product)

```

```

        left = i + 1
    return curr_product

if __name__ == '__main__':
    # input_array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4,
    7]
    array = [1, 2, -3, 2, -3, 2]
    # array = [2, 3, -2, 4]
    ans = max_product_linear(array)
    print(ans)

```

再次之后，再次使用pylint进行打分，

```

***** Module linear
linear.py:1:0: C0114: Missing module docstring (missing-module-docstring)
linear.py:3:29: W0621: Redefining name 'array' from outer scope (line 58)
(redefined-outer-name)
linear.py:52:12: W0612: Unused variable 'left' (unused-variable)

-----
Your code has been rated at 9.36/10 (previous run: 9.02/10, +0.34)

```

可以看到，在多次修改后，代码得分已达到9.36分，使人满意。

4.3.3 profile测试

使用和之前同样的测试文件，测试结果如下所示

```

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    0.000    0.000  :0(exec)
1      0.000    0.000    0.000    0.000  :0(isinstance)
1      0.000    0.000    0.000    0.000  :0(len)
6      0.000    0.000    0.000    0.000  :0(max)
3      0.000    0.000    0.000    0.000  :0(min)
1      0.000    0.000    0.000    0.000  :0(setprofile)
1      0.000    0.000    0.000    0.000  <string>:1(<module>)
1      0.000    0.000    0.000    0.000  linear.py:35(max_product_linear)
1      0.000    0.000    0.000    0.000  profile:0(linear.max_product_linear([2, 4, -3, 2]))
0      0.000    0.000    0.000    0.000  profile:0(profiler)

```

可以看到，在这里函数总体用时极少，几乎都是0.000，尝试更复杂的数组进行测试：

```

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    0.000    0.000  :0(exec)
1      0.000    0.000    0.000    0.000  :0(isinstance)
1      0.000    0.000    0.000    0.000  :0(len)
30     0.000    0.000    0.000    0.000  :0(max)
15     0.000    0.000    0.000    0.000  :0(min)
1      0.000    0.000    0.000    0.000  :0(setprofile)
1      0.000    0.000    0.000    0.000  <string>:1(<module>)
1      0.000    0.000    0.000    0.000  linear.py:35(max_product_linear)
1      0.000    0.000    0.000    0.000  profile:0(linear.max_product_linear([13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]))
0      0.000    0.000    0.000    0.000  profile:0(profiler)

```

在这里，虽然输出的数组更长，`max` 和 `len` 函数也被调用了更多次（30次和15次），但实际上总耗时几乎还是没变，说明该代码已经较为简洁。

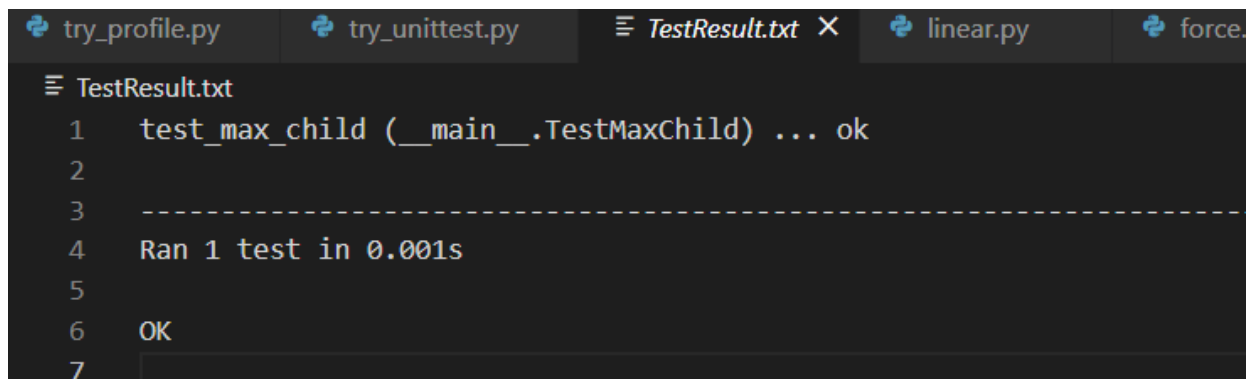
4.3.4 unittest测试

在这里，我们使用与之前相同的测试用例，测试用例设计思路见暴力法部分的unittest测试。

测试用例如下所示：

```
test_cases2 = [
    {'input': [], 'output': None},
    {'input': [1], 'output': (1)},
    {'input': [1, 2, 3], 'output': (6)},
    {'input': [-1, -2, -3], 'output': (6)},
    {'input': [2, -1, 3, -2, 4, -1], 'output': (48)},
    {'input': [1, 0, 2, 0, 3], 'output': (3)},
    {'input': [1, 2, 3, 0, 1, 2, 3], 'output': (6)},
    {'input': [1, 2, -3, 2, -3, 2], 'output': (72)},
    {'input': [2, -3, 1, 2, -1, 5, -3, 2, -4, 3, 0, 1, 2], 'output': (4320)}
]
```

测试结果如下：



```
try_profile.py try_unittest.py TestResult.txt X linear.py force.
TestResult.txt
1 test_max_child (__main__.TestMaxChild) ... ok
2
3 -----
4 Ran 1 test in 0.001s
5
6 OK
7
```

该代码通过全部的测试，测试通过率100%。

4.4 递归法

这里的递归方法是一种类似于双指针的做法，从左边和右边同时开始遍历数组，分别维护当前位置左边和右边的乘积，取两者的最大值即可。由于子数组必须是连续的，因此不需要进行子数组的判断和拼接。

4.4.1 代码实现

```
# 递归法求最大子数组

import sys

def maxProduct(nums):
    n = len(nums)
    if not isinstance(nums, list):
        raise TypeError("Input parameter must be a list.")
    if n == 0:
        return 0
    if n == 1:
        return nums[0]
    left_product = right_product = 1
```

```

max_product = float('-inf')

for i in range(n):
    left_product *= nums[i]
    right_product *= nums[n - i - 1]
    max_product = max(max_product, left_product, right_product)

return max_product if max_product > 0 else 0

if __name__ == '__main__':
    array = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]
    sub_array = maxProduct(array)
    print("最大子数组之和为: {}".format(sub_array))

```

4.4.2 pylint测试

使用pylint进行测试，测试结果如下所示：

```

PS D:\Archieve\大三下\软件工程\最大子数组\max-subarray> pylint -ry .\pointer.py
***** Module pointer
pointer.py:1:0: C0114: Missing module docstring (missing-module-docstring)
pointer.py:5:0: C0116: Missing function or method docstring (missing-function-
docstring)
pointer.py:5:0: C0103: Function name "maxProduct" doesn't conform to snake_case
naming style (invalid-name)
pointer.py:6:4: C0103: Variable name "n" doesn't conform to snake_case naming style
(invalid-name)
pointer.py:26:10: C0209: Formatting a regular string which could be a f-string
(consider-using-f-string)
pointer.py:3:0: W0611: Unused import sys (unused-import)

```

Report

=====

20 statements analysed.

Statistics by type

type	number	old number	difference	%documented	%badname	
module	1	1	=	0.00	0.00	
class	0	NC	NC	0	0	
method	0	NC	NC	0	0	
function	1	1	=	0.00	100.00	

28 lines have been analyzed

Raw metrics

type	number	%	previous	difference	
code	22	78.57	NC	NC	
docstring	0	0.00	NC	NC	
comment	1	3.57	NC	NC	
empty	5	17.86	NC	NC	

Duplication

	now	previous	difference	
nb duplicated lines	0	0	0	
percent duplicated lines	0.000	0.000	=	

Messages by category

type	number	previous	difference	
convention	5	5	5	
refactor	0	0	0	
warning	1	1	1	
error	0	0	0	

Messages

```

-----

+-----+-----+
|message id          |occurrences |
+=====+=====+
|invalid-name        |2           |
+-----+-----+
|unused-import       |1           |
+-----+-----+
|missing-module-docstring |1           |
+-----+-----+
|missing-function-docstring |1           |
+-----+-----+
|consider-using-f-string   |1           |
+-----+-----+

-----

Your code has been rated at 7.00/10 (previous run: 7.00/10, +0.00)

```

需要修改的内容如下所示：

- 添加了模块注释和函数注释，以满足 C0114 和 C0116 的要求。
- 将函数名 `maxProduct` 改为 `max_product_pointer`，符合 `snake_case` 的命名规范。
- 将变量名 `n` 改为 `num_length`，符合 `snake_case` 的命名规范。
- 将字符串格式化改为使用 f-string，符合 C0209 的要求。
- 删除了无用的 `sys` 模块导入语句，以满足 W0611 的要求。

多次修改之后，重新进行测试，结果显示：

```

***** Module pointer
pointer.py:12:4: C0103: Variable name "n" doesn't conform to snake_case naming style
(invalid-name)

-----

Your code has been rated at 9.47/10 (previous run: 8.95/10, +0.53)

```

代码的评分达到了9.47，令人满意。

4.4.3 profile测试

所使用的测试文件和之前相同，结果如下所示：


```

11 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000   0.000   0.000   0.000 :0(exec)
      1   0.000   0.000   0.000   0.000 :0(isinstance)
      1   0.000   0.000   0.000   0.000 :0(len)
      4   0.000   0.000   0.000   0.000 :0(max)
      1   0.000   0.000   0.000   0.000 :0(setprofile)
      1   0.000   0.000   0.000   0.000 <string>:1(<module>)
      1   0.000   0.000   0.000   0.000 pointer.py:5(max_product_pointer)
      1   0.000   0.000   0.000   0.000 profile:0(pointer.max_product_pointer([2, 4, -3, 1]))
      0   0.000   0.000   0.000   0.000 profile:0(profiler)

```

从这里可以看出，整体代码用时相当的短，几乎都是0.000。尝试使用更复杂的数组，如下所示：

```

23 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000   0.000   0.000   0.000 :0(exec)
      1   0.000   0.000   0.000   0.000 :0(isinstance)
      1   0.000   0.000   0.000   0.000 :0(len)
     16   0.000   0.000   0.000   0.000 :0(max)
      1   0.000   0.000   0.000   0.000 :0(setprofile)
      1   0.000   0.000   0.000   0.000 <string>:1(<module>)
      1   0.000   0.000   0.000   0.000 pointer.py:5(max_product_pointer)
      1   0.000   0.000   0.000   0.000 profile:0(pointer.max_product_pointer([13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]))
      0   0.000   0.000   0.000   0.000 profile:0(profiler)

PS D:\Archive\大三下\软件工程\最大子数组\max-subarray>

```

在这里，只有 `max` 函数调用次数增多了，达到了16次，然而整体用时仍然相当短，说明代码性能较优。

4.4.4 unittest测试

在这里，我们使用与之前相同的测试用例，测试用例设计思路见暴力法部分的unittest测试。

测试用例如下所示：

```

test_cases2 = [
    {'input': [], 'output': None},
    {'input': [1], 'output': (1)},
    {'input': [1, 2, 3], 'output': (6)},
    {'input': [-1, -2, -3], 'output': (6)},
    {'input': [2, -1, 3, -2, 4, -1], 'output': (48)},
    {'input': [1, 0, 2, 0, 3], 'output': (3)},
    {'input': [1, 2, 3, 0, 1, 2, 3], 'output': (6)},
    {'input': [1, 2, -3, 2, -3, 2], 'output': (72)},
    {'input': [2, -3, 1, 2, -1, 5, -3, 2, -4, 3, 0, 1, 2], 'output': (4320)}
]

```

测试结果如下：

```

try_profile.py  try_unittest.py  TestResult.txt  linear.py  pointer.py
TestResult.txt
1  test_max_child (__main__.TestMaxChild) ... ok
2
3  -----
4  Ran 1 test in 0.000s
5
6  OK
7  |

```

在这里，测试通过率为100%。