TIRF-NewAnalyzer

脚本教程

化学41 李梓劼

2014012318

[lizj14@outlook.com](mailto:lizj14@outlook.com)

1 程序及脚本简介

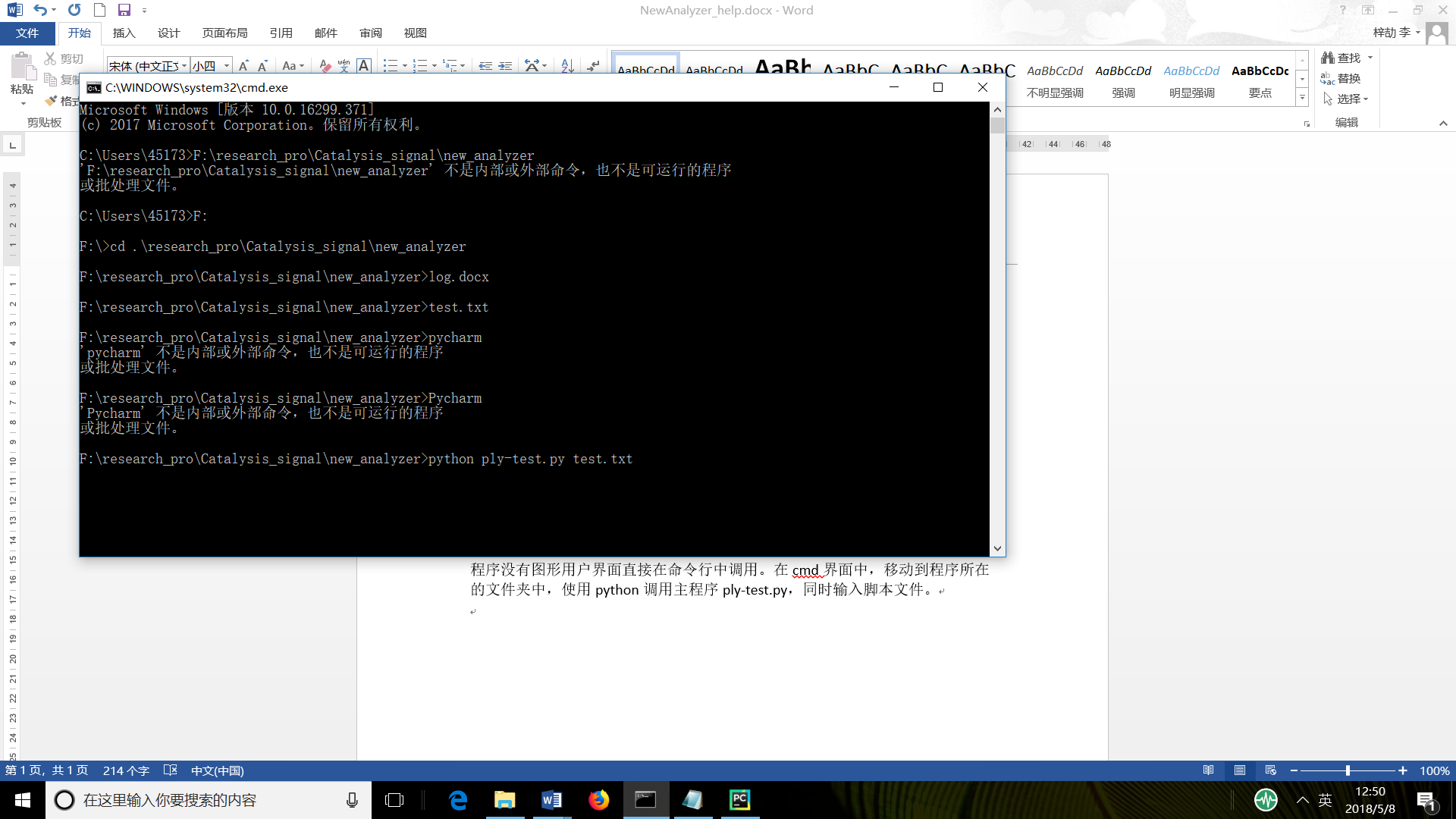
NewAnalyzer的设计目标是实现TIRF实验的数据处理工作，并且通过编写控制脚本的方式，使得使用者可以自行组织数据处理的流程、调用不同的功能、调整参数，而无需了解程序内部的结构。

控制脚本由多条命令组成，程序将读取脚本，按照脚本顺序逐个执行指令，并且会在执行前检查脚本中是否有错误。

2 操作概述

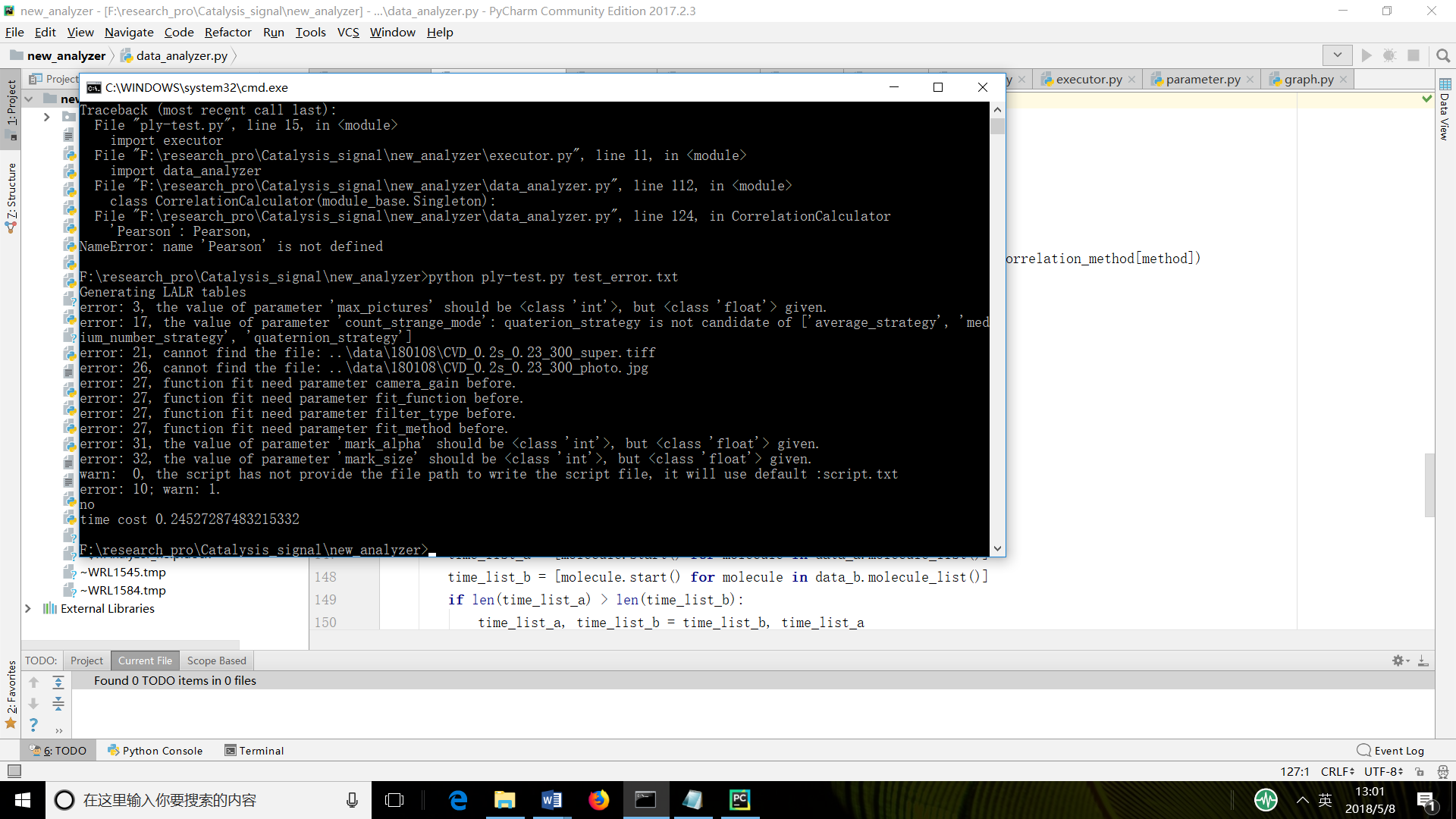
程序使用前，需要保证计算机中安装python以及需要的python库，具体可以在命令行界面中通过python和pip工具（pip list命令检查已经安装的python库）检查，详见文末。

程序没有图形用户界面直接在命令行中调用。在cmd界面中，移动到程序所在的文件夹中，使用python调用主程序ply-test.py，同时输入脚本文件。



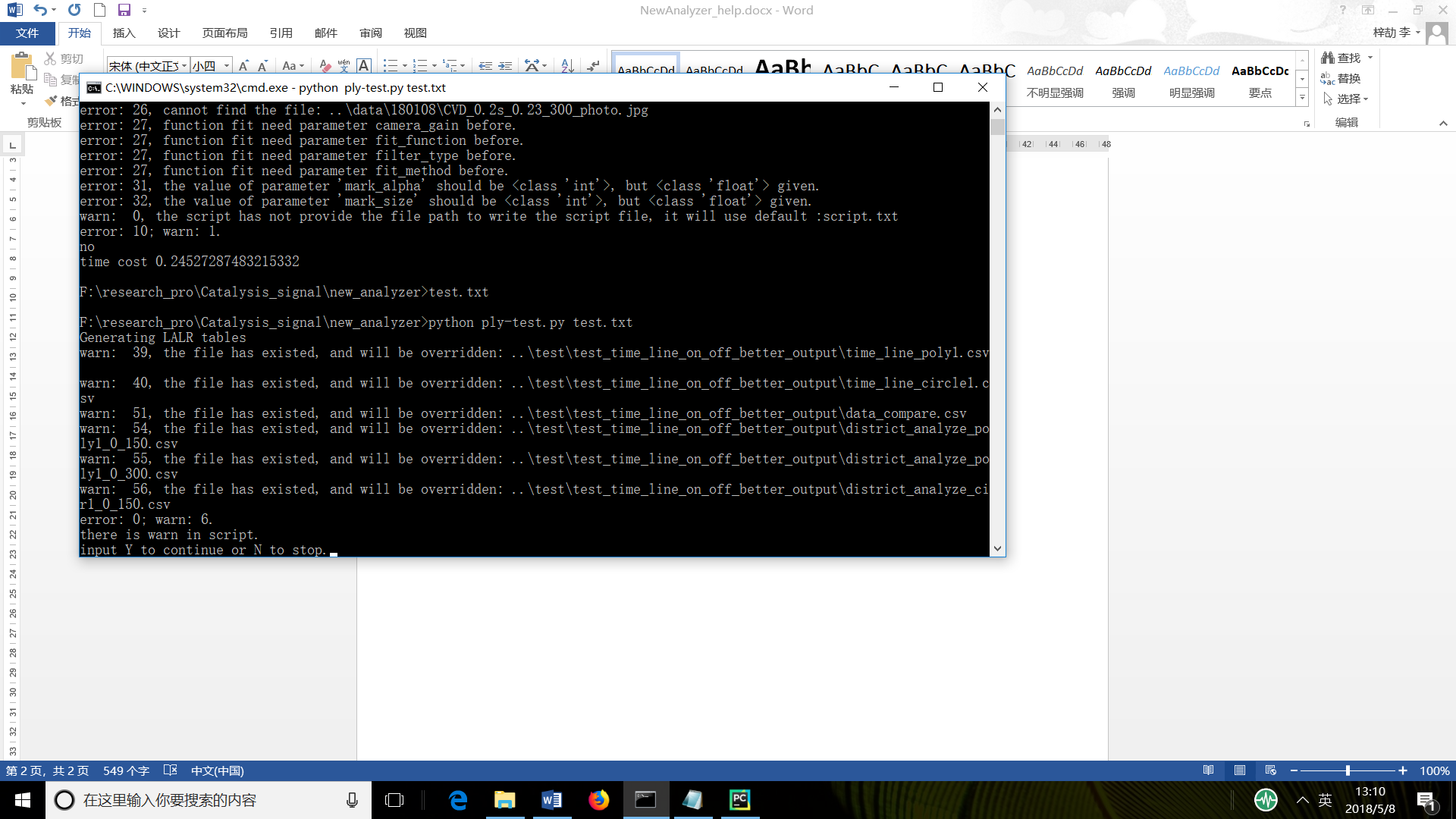
脚本文件需要输入前完成，输入之后不能再做修改。

之后程序将自动进行脚本检查（因为脚本遵循语法规范，对于不符合的情况无法正确执行），如果坚持到错误，会自动显示错误信息，并且终止程序：



如图，test\_error.txt是一个故意设计的错误脚本。输入之后，程序首先显示Generating LALR tables，这是程序在为解析脚本做准备，因为本程序所用脚本的语法分析是LALR方法。之后，程序显示了检查出来的各个错误，开头的error表明这是一个错误，后面的数字表示脚本中所在行号，之后表示错误的提示信息。而有一个是warn，这表示是一个警告，很可能是一个错误。之后程序显示error和warn的总数，并输出no，表示脚本有误，不能正确执行。如果脚本无误，则会显示yes，之后程序开始执行。

此外，还有一种情况：

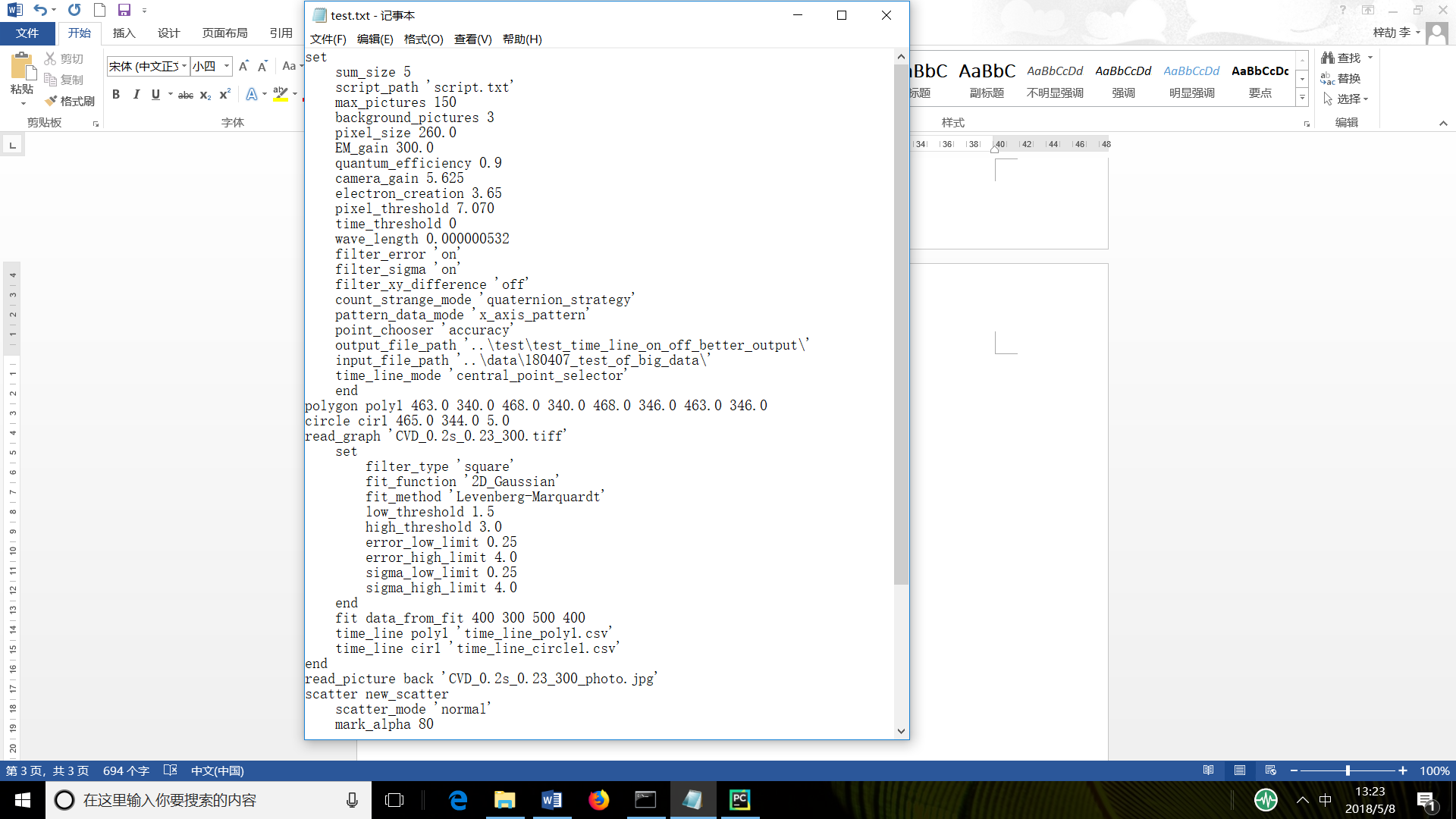


这种情况下，脚本中只有warn，而没有error，因此使用者可以自行决定，是执行程序还是修改脚本。查看提示：input Y to Continue or N to stop。按提示执行即可。

具体的各error、warn的类型、出现原因会在后文中详细介绍，下一部分，我们先介绍程序脚本。

3 程序脚本编写规则

3.1 脚本示例与格式说明



如图，程序脚本可以直接使用记事本编写、修改。每个脚本都有多条命令组成，程序会由上而下依次执行脚本。一般脚本中会有换行和缩进，换行和缩进并不会影响脚本的执行，但是可以方便使用者阅读、修改，因为换行和缩进可以体现脚本的层次关系。

3.2 脚本语句组成

每个脚本由多个依次执行的语句组成，语句有以下组成部分组成：

关键字：如set，fit，polygon等，一般在语句的开头，告诉语句这条语句的类型、功能。关键字在程序内部确定，用户不能修改。

参数名：数据处理过程中用到的参数的名称，如：wave\_length，荧光波长。在程序中往往后跟其值。和关键字一样，在程序内部确定，用户不可修改。

常量：程序中使用的各种常量，如read\_picture语句中的文件路径名。NewAnalyzer的脚本中一共有三种常量，分别是整数、浮点数（小数）、字符串。需要注意的是，整数和浮点数不能彼此替代，字符串起始需要加单引号。常量并没有名字，在程序中也只会在书写处使用一次。

对象：脚本中声明一个名字，将一个对象（可能是数据、图形）与之关联；当再次使用该名字时，程序将自动调用与之关联的对象。一般编程语言中，变量赋值后，可以再次赋值，变量将与新值关联；但是本脚本中的名字与对象关联后，将不能再次将该名字与其他对象关联。关于对象的具体介绍，请参见下文。程序中的每个对象都会有明确的类型，具体可以在后文检查。

这一部分略为抽象，接下来我们将一步步介绍脚本如何编写。

4 脚本编写教程

4.1 set语句：参数语句及参数系统

作为一个数据处理程序，程序中必然要使用大量的参数。我们需要通过脚本告诉计算机，我们都使用了什么参数。而为了做到这件事情，我们需要使用set语句。

Ex 4.1.1

set

out\_file\_path ‘..\test\’

end

set为关键字，表示set语句开始。out\_file\_path 为参数名称，告诉程序我们要设置out\_file\_path这个参数。后面的’..\test\’是一个常量，告诉程序这个参数的值我们设置成了这个值。而最后的end表明这条语句的终结。那么为什么要有end呢？（很快我们就会看到，大部分语句并不需要end作为结尾）这是因为我们可以在一条set语句中设置多组参数，每组即一个参数名称和相应的参数值。那么，接下来我们要看复杂一些的情况，来看程序是如何调用参数的。

Ex 4.1.2

set

out\_file\_path ‘..\test\’

script\_path ‘script.txt’

end

X

set

out\_file\_path ‘..\case\’

input\_file\_path ‘.\example\’

end

Y ‘asdflkj.jpg’

（由于还没有讲到其他语句，因此X和Y是语句例子，先不用管他具体是干什么的。）

本程序中，脚本处理参数使用的是职责链机制。简单来讲，当一个语句需要调用一个参数时，他会向上查找，检查上一条set语句是否有该参数；如果没有，则一直向前查找，直到脚本起始，如果仍没有，则会报错（详见5章节，具体错误类型。）举例来讲，假如Y需要使用script\_path参数，程序会先搜索第二条set语句中是否有该参数；显然是没有的，所以程序会再向上查找，搜索第一个set语句，他找到了’script.txt’，所以在执行过程中，这个参数的值就被视为了这个值。但是，如果Y想使用input\_file\_path，程序就会在第二条set语句中找到它，然后使用该值。那么问题来了，如果Y想使用out\_file\_path，而第一条和第二条set中都有这个参数，那么程序会选择哪一个呢？考虑一下我们之前介绍的搜索顺序，不难想到，程序会使用’..\case\’。

而考虑X语句，它只能使用第一条set语句中的参数，因为程序永远不会向下查找参数。

这样我们就知道了参数系统的基本设定，不过之后我们还会再介绍其他的一些参数调用机制。下面我们先讲一下本节中出现的三个参数的含义。我们会在各个章节分别讲解各个参数的含义。

script\_path: 程序正常执行结束后，会自动保存脚本文件，这个参数设置的是自动保存的脚本文件的文件名。这个参数是可以不设置的，程序会使用默认值，并且弹出一个warn。

out\_file\_path：输出文件路径。程序执行过程中，我们要把结果保存在文件里，这个参数指定保存的路径。实际执行过程中，每次调用输出文件路径是，会自动在前面加上当前位置（忘了的同学复习本章前面的部分！）的输出文件路径参数值。例如，out\_file\_path为’../case/’，输出文件名为’script.txt’，则输出的脚本文件的完整路径名就是’../case/script.txt’。

input\_file\_path：输入文件路径，与输出文件路径一致，只不过是输入文件的路径前缀。

Set语句格式:

set

参数名称 参数值

…（以上重复若干组，可以为0）

end

4.2 read\_molecule语句 & write\_molecule语句：对象机制

接下来两个语句是读取和写入分子信息（拟合结果）的语句。（由于拟合语句涉及较多，我们稍后再介绍。）通过这两条语句，理解对象机制的作用。

Ex 4.2.1

read\_molecule data\_1 ‘data.csv’

write\_molecule data\_1 ‘data\_copy.csv’

这两条语句的含义是，从data.csv这个文件中读取分子信息（这个文件也是本程序生成的，具有一定格式），然后存储到data\_1为名称的对象中；然后再从名称为data\_1的对象中提取数据信息，写入到名为data\_copy.csv的文件中。

这两个语句的格式是：

read\_molecule 分子信息对象名（新声明） 数据源文件路径名

write\_molecule 分子信息对象名 保存文件路径名

在Ex 4.2.1 中，两条语句中的data\_1都是对象名，都指一组数据。那么程序中为什么要引入对象这个概念呢？因为这样同一组脚本中，同一类型的数据可以有很多组。以分子对象信息为例，引入对象概念后，我们可以在同一个脚本中进行多组拟合，然后对不同组的数据分别给予不同的对象名；这样，当我们需要在数据分析功能中调用数据时，使用对象名就可以区分不同组的数据。回到Ex 4.2.1中，第一条语句中，data\_1是一个新声明的对象名，而在第二条中，使用data\_1是为了使用data\_1为名的对象中保存的数据。

Ex 4.2.2

read\_molecule data\_2 ‘data.csv’

write\_molecule data\_3 ‘data\_new.csv’

read\_molecule data\_2 ‘da.csv’

在上例中，后两条语句都是非法的。第二行中，data\_3这个对象名之前并没有被声明，并没有与数据关联；而第三行中，read\_molecule 语句会读取数据，并关联到指定的对象名上；但是data\_2这个对象名已经被使用，如果再次使用，则之后使用data\_2时无法判断使用哪一组数据。

4.3