第四章 Numpy 内存布局与 ASC II 文件的读写

4.1 Numpy 数组内存布局与访问性能

小节概述 在 NumPy 中，数组的内存布局会影响访问速度。数组可以是**行优先或列优先**。不同的内存布局会影响数组的遍历速度和存储方式。

4.1 数组内存布局概念

4.1.1 行优先 C 风格

特性

① 在行优先，数组**每行**中的元素是连续存储的。
② 遍历顺序从每行的第一个元素开始，依次遍历行中后续的元素。
③ 连续读取速度最快，因此需要从 3 开始。 对于$a(2,3)$而言
④ **Python 和 C 语言**默认使用行优先内存布局。

示例

```
a=np.array([[1,2,3],[4,5,6]],order='C')
#内存顺序：1，2，3，4，5，6
```

4.1.2 列优先 Fortran 风格

特性

① 在列优先数组中，数组**每列**中的元素是连续存储的。
② 遍历顺序是从每一列的第一个元素开始，依次遍历列中后续的元素。
③ **Fortran 语言**默认使用列优先内存布局。

示例

```
b=np.array([[1,2,3],[4,5,6]],order='F')
#内存顺序：1，4，2，5，3，6
```

4.2 访问性能

概述 对于大规模数组，访问数据的顺序（按行或按列）会显著影响性能。下面的代码通过 **timeit** 模块来测量按行和按列访问数据的时间差异。

分析

① **行优先数组：按行访问更快**，因为数据是按行连续存储的，符合 CPU 缓存的存取模式。
② **列优先数组：按列访问更快**，因为数据是按列连续存储的（适用于列优先的存储方式）。

分析代码

```
rows,cols =2000,2000
data = np.random.rand(rows,cols)

#按行访问数组
def access_row(data):
    row_sum = 0
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            row_sum += data[i,j]
    return row_sum

#按列访问数组
def access_col(data):
    col_sum = 0
    for j in range(data.shape[1]):
        for i in range(data.shape[0]):
            col_sum += data[i,j]
    return col_sum

time_row = timeit.timeit('access_row(data)', globals=globals(), number=10)
time_col = timeit.timeit('access_col(data)', globals=globals(), number=10)
print(f"Accessing by rows took {time_row:.4f} seconds.")
print(f"Accessing by columns took {time_col:.4f} seconds.")
#Accessing by rows took 5.0795 seconds.
#Accessing by columns took 5.2440 seconds.
```

4.3 性能优化：使用向量化操作代替循环

概述 在数据科学中，向量化操作更加高效。NumPy 提供了很多**向量化函数**，可避免手动编写嵌套循环。

示例 向量化计算总和：

```
# 创建一个较大的随机数组        rows, cols = 2000, 2000       data = np.random.rand(rows, cols)
# 使用向量化操作计算总和（按行或按列求和）
def vectorized_sum(data):
    return np.sum(data)
# 测试向量化方法的性能
time_vectorized = timeit.timeit('vectorized_sum(data)', globals=globals(), number=10)
print("向量化求和时间: {:.5f} 秒".format(time_vectorized))        #向量化求和时间: 0.03580 秒
```

4.2 ASC II 文件读写

4.2.1 基本思想

通用程序 ① 文件**存在/不存在**
② **文件编码**与指定编码不符合：导致编码失败、指定的编码名称不正确
③ 类型转换的**数据类型**问题

提供参数 ① 路径、文件名 ② 列、行范围 ③ 文件头范围 ④ 分隔符符号 ⑤ 数据类型转换

4.2.2 基本语法与参数

4.2.2.1 open 方法

基本语法

```
file_object = open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

参数

`mode` 文件打开模式，常见模式有 `r` 只读、`w` 写入、`a` 追加、`b` 二进制模式等

`encoding` 指定编码格式，如 `utf-8`、`gbk` 等，确保正确解码文本文件。

4.2.2.2 常用读取方法

read() 一次性读取整个文件，返回一个字符串。

readline() 逐行读取，每次读取一行。

readlines() 一次性读取所有行，返回一个列表，每行作为列表的一个元素。

示例

```
with open('test.txt','r',encoding='utf-8') as f:
    content=f.read()   #一次性读取全部内容
    line = f.readline() #检查是否正确
    while line:
        print(line.strip()) #去除换行符
        line = f.readline()
    lines = f.readlines()   #一次性读取多行，返回列表，推荐使用
```

4.2.2.3 异常处理

异常类型 FileNotFoundError：文件不存在。
UnicodeDecodeError：文件编码与指定编码不符，导致解码失败。
LookupError：指定的编码名称不正确。