第十一周问答作业

朱士杭 231300027

2024 年 5 月 9 日

1 IPython

IPython 是一个开源的交互式计算和数据分析工具，提供了一个强大的交互式 shell 和一个 Jupyter Notebook 内核，可以增强 Python 交互能力，提供更好的交互式计算体验。

1.1 IPython 的基本功能

1.1.1 增强的交互式 shell

IPython 提供了一个比标准的 Python shell 更加强大和灵活的交互环境，支持语法高亮、代码补全、对象内省等功能，使得编写和调试代码更加高效。

1.1.2 查询功能

IPython 允许用户探索代码、对象和数据结构。通过使用问号?，可以查看对象的详细信息，包括文档字符串和函数签名。

1.1.3 历史记录

IPython 记录了用户在交互式会话中输入的所有命令，允许用户检索和重用以前的命令。通过将 In[ ] 所有的输入进去的代码按照顺序保存进一个列表当中去，将 Out[ ] 所有输出按照键值对存储进一个字典当中去

1.1.4 魔术命令

IPython 提供了一系列特殊的命令（Magic Function），这些命令可以执行各种有用的功能，如运行代码、计时、调试和执行系统命令等。一系列% 开头的辅助命令，用于控制 IPython 环境和系统行为等

1.1.5 多种输出格式

IPython 支持多种富媒体输出格式，如 HTML、LaTeX、SVG 和 JSON，这使得在交互式会话中显示图像、图表和其他多媒体内容变得容易。

1.2 如何演变成 Jupyter Notebook

Jupyter Notebook 是一个开源的 Web 应用程序，允许创建和共享包含实时代码、方程、可视化和解释性文本的文档。它是 Project Jupyter 的一部分，这个项目源自 IPython 项目，旨在支持多种编程语言。

1.2.1 交互式编程环境

Jupyter Notebook 允许用户在同一个环境中编写代码、运行代码和查看输出，这使得数据探索和实验变得更加直观。

1.2.2 支持多种编程语言

Jupyter Notebook 支持超过 40 种编程语言，包括 Python、R 和 Julia，用户可以为不同的编程语言创建笔记本。

1.2.3 富文本编辑

Jupyter Notebook 使用 Markdown 和 LaTeX 支持富文本编辑，使得创建格式化的文本、数学表达式和链接变得容易。

1.2.4 数据可视化

Jupyter Notebook 支持多种数据可视化库，如 Matplotlib、Plotly 和 Bokeh，使得创建交互式图表和图形变得简单。

1.2.5 共享和协作

Jupyter Notebook 可以轻松地共享和发布到互联网上，支持多人协作和实时编辑。

一言以蔽之，IPython 是一个强大的交互式计算工具，而 Jupyter Notebook 则是在 IPython 的基础上发展起来的一个更加完善的交互式数据科学和计算平台。

图 1: 在 IPython 中实现矩阵归一化代码

2 Magic Function 作用

Magic functions 是 IPython 和 Jupyter Notebook 中的一种特殊命令，以百分号% 或者%% 开头，用于执行与 Python 代码不同的特定操作，以及控制 IPython 环境和系统行为等，如运行代码、计时、调试和执行系统命令等，这些命令使得在交互式环境中工作更加高效和方便

2.1 常用的 Magic Function 及其作用

2.1.1 %timeit 与%time 作用

这两个命令用于测量代码片段的执行时间。它会自动运行多次代码，以得到一个更准确的平均执行时间。这对于性能分析和优化非常有用

2.1.2 %pwd 与%cd 与%ls 作用

%pwd 用于显示目前运行 IPython 时所处的文件夹路径

%cd 用于在系统命令行中切换文件夹，比如进入下一级文件夹/返回上一级文件夹

%ls 用于显示目前文件夹下存在的所有文件

2.1.3 %%writefile 命令与%run 命令

%%writefile 命令用于直接编写.py 文件的内容并将其保存进.py 文件当中然后再用%run 命令执行.py 文件（此时不用!python 命令系统执行）

3 %time 和%timeit 的区别和各自的使用场景

%time 用于简单地测量单次代码执行的时间，输出代码执行的时间和结果它适合于快速获取代码执行的近似时间，不需要精确的性能分析。

%time 只执行一次代码，因此它可能受到系统负载和其他偶然因素的影响，结果可能不够稳定。

%timeit 用于测量代码的执行时间，它会多次运行代码以计算一个平均时间。它自动选择合适的循环次数，以获得更稳定的测量结果，并最终取平均时间并输出，以及执行了多少次循环

%timeit 适合于需要进行精确性能分析的场景，尤其是需要优化代码或比较不同代码片段的性能时

如果只需要一个快速的估计，可以使用%time。如果需要进行更精确的性能分析，尤其是对于执行时间很短的代码片段，应该使用%timeit

4 NumPy 数组在存储和内存使用上的优势

4.1 内存连续性

NumPy 数组在内存中是连续存储的，这意味着数组中的元素都紧密地排列在一起。这种存储方式允许 NumPy 快速地进行元素访问和迭代，因为数组中的元素可以通过简单的数学运算来定位。而 Python 列表是动态的，它们在内存中可能是非连续的，因此访问和迭代可能需要更复杂的操作。

4.2 固定类型

NumPy 数组是同质的，即它们只能存储一种类型的元素（例如，只能是整数或浮点数）

这种类型的一致性允许 NumPy 在内存中更高效地存储数组，因为不需要为每个元素存储额外的类型信息。

相比之下，Python 列表可以包含不同类型的元素，这使得它们在内存中更加灵活但效率较低。

4.3 向量化操作

NumPy 数组支持向量化操作，这意味着可以对整个数组或数组的切片执行操作，而不需要显式循环。这些操作通常由底层 C 或 Fortran 代码实现，因此速度比 Python 循环快得多。

而 Python 列表不支持向量化操作，需要使用 Python 循环来实现相同的功能，这通常会更慢。

4.4 内存使用效率

由于 NumPy 数组的内存连续性和固定类型，它们通常比 Python 列表更节省内存。

就类似于 C++ 中的 struct 数组类型会进行数据内存的对齐导致很多内存的浪费，而在 NumPy 当中由于元素类型是一样的因此可以大大节省内存

4.5 并行处理和向量化

NumPy 数组的存储和处理方式使其易于与并行处理库（如 OpenMP）集成，以及利用现代 CPU 的 SIMD（单指令多数据）功能，使得 NumPy 数组在执行大规模数值计算时非常高效。

5 %debug 这个 Magic Function 的功能以及使用方式

5.1 %debug 功能

%debug 是 Jupyter Notebook 中的一个 Magic Function，它启动一个交互式调试器，允许用户在代码执行出错时深入检查和调试代码。

在 Jupyter Notebook 中运行代码时，如果遇到错误，使用%debug 可以进入调试模式。

在这种模式下，你可以查看变量的值、执行单步执行、检查调用栈、设置断点等，从而帮助你理解代码的行为并找出问题所在。

5.2 使用方式

1. 打开 JupyterNotebook 并启用一个.ipynb 文件开始写代码

2. 在一个 cell 中写好代码之后运行发现出错

3. 在下一个 cell 中写入%debug，会跳出 ipdb（带有提示符的交互式环境）在其中输交互即可

4. 在这个环境中，你可以使用多种命令来调试代码，例如：

p <variable>：打印变量的值

n：执行下一行代码

s：进入当前行的函数或方法
c：继续执行代码直到下一个断点或异常
q：退出调试器
l：列出当前行的代码及其周围代码
b <line>：在指定行设置断点
cl：清除所有断点
disable <breakpoint>：禁用指定的断点
enable <breakpoint>：启用指定的断点

图 2: 在 Jupyter 中出现报错

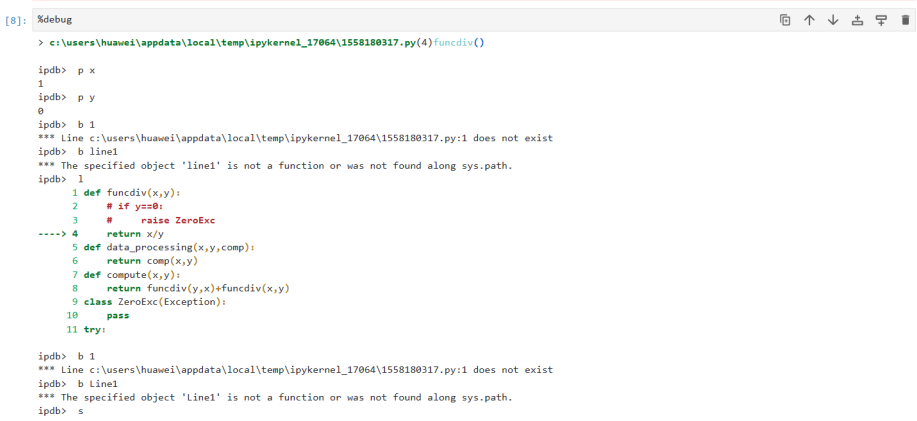

图 3: 在 Jupyter 当中使用%debug 模式

6 NumPy 如何处理和存储高维数组

在内部，NumPy 数组（np.ndarray 方式）是通过一个称为“数据缓冲区”的连续内存块来存储的, 这种存储方式允许 NumPy 高效地处理和操作大量数据

NumPy 最终仍旧是将一整块高维数组展平成一条一维数组,通过传入 strides(n1,n2,……) 通过 n1,n2,……的值来记录需要访问高维数组中某个元素的时候 x1,x2,……各自需要跳转多少个字节

这就可以使得当需要数组进行转置的时候不需要对内存块进行复制转移,可以直接通过对 strides 修改从而改变其访问方式就可以实现对数组的转置，实际内部存储方式并未发生改变，极大提高了效率

另外，NumPy 还可以通过向量化操作、广播（自动扩展较小的数组与较大数组的形状匹配，从而允许不同形状的数组算术运算）以及并行处理等方式提升高维数组计算的效率

7 NumPy 进行数据操作的优点

7.1 性能优化

NumPy 的底层使用 C 语言编写，这使得它在执行数组运算时非常快速。NumPy 的数组操作经过高度优化，可以利用底层硬件的并行性，比如 SIMD 指令集。

7.2 丰富的函数库

NumPy 包含了大量的数学函数，涵盖了线性代数、概率统计、傅里叶变换等多个数学领域，这些函数可以无缝地在 NumPy 数组上工作。

7.3 广播（Broadcasting）

NumPy 的广播机制允许用户在不同大小的数组之间进行算术运算，这在处理数组运算时非常强大且节省时间。

7.4 兼容性

NumPy 是许多其他科学计算和数据分析库的基础，如 SciPy、Pandas、Matplotlib、Scikit-learn 等，这使得 NumPy 成为了 Python 科学计算生态系统的核心。

7.5 跨平台

NumPy 可以在多种操作系统上运行，包括 Windows、macOS 和 Linux，可用性和灵活性更高，应用生态更好

7.6 社区支持

NumPy 拥有一个活跃的社区，不断有新的功能和改进被加入，同时，社区也为用户遇到的各种问题提供了丰富的文档和支持。

7.7 适用于大规模数据

尽管 NumPy 在处理非常大规模的数据集时可能会受到内存限制，但它仍然提供了有效的数据分块和操作机制，使得用户能够处理比内存中能容纳的更大的数据集。由于这些优点，NumPy 成为了数据科学、机器学习以及更广泛的科学计算领域中不可或缺的工具。

8 讨论不同类型的 Magic Function

8.1 行 magic（line）和单元格 magic（cell）

行 magic 一般执行的是一行的命令，用% 开头命令；单元格 magic 一般执行的是一整块的代码命令，用%% 开头

比如说要测试代码运行时间性能，%timeit [i*10 for i in range(10000)] 就是行 magic 执行的是一条语句的时长

而%%timeit for i in range(10000):print(i*10) 就是 cell magic 执行一整块的代码

8.2 别名 magic、模块库 magic、系统命令 magic

别名 magic 指的是那些允许用户为常用的命令或代码片段定义别名的命令，以便于快速访问。例如，%alias 用于定义别名。

实用 magic 命令提供了一些实用的功能，如%matplotlib 用于集成 Matplotlib 图形库，%load 用于加载外部脚本。

系统命令 magic 允许用户在笔记本中执行系统命令。! 开头的命令将被解析为调用系统 shell 命令，比如!pwd、!cd 等等

9 借用 NumPy 思想实现 N 维矩阵类

```
import copy
8 个用法
class N_matric:
    def __init__(self,array):
        self.array=array
        self.cal_dim()
        self.to_one_array()#将高维数组展平
        self.cal_trace()
    1 个用法
    def to_one_array(self):...
    21 个用法 (4 个动态)
    def visit_element(self,*args):...
    5 个用法
    def Reshape(self,new_shape):...
    4 个用法
    def transpose(self):...
    def __add__(self, other):...
    def __sub__(self, other):...
    1 个用法
    def dotmul(self,other):...
    def __mul__(self, other):...
    1 个用法
    def cal_trace(self):...
```

图 4: N 维矩阵类的框架

```
1个用法
def cal_dim(self):
    '''计算self.array的维数大小、shape形状以及strides元组大小'''
    dim=1
    a=self.array
    num=1
    while type(a[0])==list:
        num*=len(a)
        dim+=1
        a=a[0]
    print("数组的维数为：",dim)
    self.dim=dim#存储矩阵维数大小
    num*=len(a)
    self.num=num#存储数组拥有的元素个数
    print("数组拥有的元素个数为：",num)
    '''接下来模仿numpy当中操作计算strides序列'''
    strides=[0]*self.dim
    shape=[0]*self.dim
    i=0
    a = self.array
    while type(a[0])==list:
        shape[i]=len(a)
        a = a[0]
        strides=[j*len(a) for j in strides]
        strides[i] = len(a)
        i+=1
    strides[self.dim-1]=1
    shape[i]=len(a)
    self.strides=strides#存储数组的strides访问时移动的个数
    self.shape=shape#存储数组的shape形状大小
    print("数组的形状大小为：",self.shape)
    print("数组的strides为：",strides)
1个用法
```


图 5: 类中计算数组维度、形状以及 strides 大小

```python
def to_one_array(self):
    '''将高维数组展成一维数组'''
    def one_array(array):
        ''''''
        if type(array[0])≠list:
            return array
        else:
            total=[]
            for sub_arr in array:
                total+=one_array(sub_arr)
            return total
    self.array=one_array(self.array)
    print("将数组展平之后为：",self.array)
```

图 6: 将高维数组展平成一维数组

```python
21 个用法 (4 个动态)
def visit_element(self,*args):
    '''元素访问，传入未知个数元组，表示各个维度的索引'''
    if len(args)>self.dim:
        print("注意！您访问的元素超出了维度限制")
        return None
    ...
    ele=0
    for i in range(len(args)):
        ele+=self.strides[i]*args[i]
    return self.array[ele]
```

图 7: 利用 strides 访问数组元素

```
5个用法
def Reshape(self,new_shape):
    '''注意new_shape传进来的是一个元组'''
    multiplication=1
    for i in new_shape:
        multiplication*=i
    if multiplication!=self.num:...
    strides=[0]*self.dim
    for i in range(1,len(new_shape)):
        strides[i-1]=1
        strides=[j*new_shape[i] for j in strides]
    strides[len(new_shape)-1]=1
    self.strides=strides
    self.shape=list(new_shape)
    print(f"接下来进行数组形状重塑为{new_shape}操作")
    print("新的数组形状大小为：",new_shape)
    print("新的strides为：",self.strides)
```

图 8: 类的 Reshape 方法重塑数组形状

```
def transpose(self):
    '''对于更高维的数组直接倒序即可'''
    # self.Reshape(tuple(reversed(self.shape)))
    self.shape=list(reversed(self.shape))
    self.strides=list(reversed(self.strides))
```

图 9: 类的转置 transpose 方法改变访问方式

```python
def __add__(self, other):
    '''矩阵加法，对应元素直接相加即可，并且要求数组是二维的'''
    if self.shape≠other.shape:...
    if self.dim≠2:...
    new_array = [[0] * self.shape[1] for i in range(self.shape[0])]
    for i in range(self.shape[0]):
        for j in range(self.shape[1]):
            new_array[i][j] = self.visit_element( *args: i, j) + other.visit_element(i, j)
    return N_matric(new_array)
```

图 10: 操作符重载实现多维数组直接相加

```python
def __mul__(self, other):
    if self.shape[1]≠other.shape[0]:
        ...return
    if self.dim≠2:...
    new_array = [[0]*other.shape[1] for i in range(self.shape[0])]
    for i in range(self.shape[0]):
        for j in range(other.shape[1]):
            sum=0
            for h in range(self.shape[1]):
                sum+=self.visit_element( *args: i,h)*other.visit_element(h,j)
            new_array[i][j]=sum
    return N_matric(new_array)
```

图 11: 操作符重载实现矩阵之间的乘法

```python
def cal_trace(self):
    '''求方阵的迹'''
    if self.shape[0]≠self.shape[1]:...
    sum=0
    for i in range(self.shape[0]):
        sum+=self.visit_element( *args: i,i)
    self.trace=sum
    return self.trace
```

图 12: 计算矩阵的迹（主对角线元素之和）

```
arr=[[[1,2,3],
      [4,5,6]],
     [[7,8,9],
      [10,11,12]],
     [[13,14,15],
      [16,17,18]]]
arr1=[[[[1,2,3],
      [4,5,6]],
     [[7,8,9],
      [10,11,12]],
     [[13,14,15],
      [16,17,18]]]]
arr2=[[1,2,3],
      [4,5,6]]
arr3=[[0,1,2],
      [3,4,5]]
print("先展示二维数组：")
print("arr2数组为：",arr2)
arr2=N_matric(arr2)
print("arr2坐标(2,3)的元素为：",arr2.visit_element( *args: 1,2))
arr2.transpose()
print("将arr2转置后坐标(1,2)的元素为：",arr2.visit_element( *args: 0,1))
print("将arr2转置后坐标(3,1)的元素为：",arr2.visit_element( *args: 2,0))
arr2.Reshape((1,6))
print("将arr2重塑为行向量后坐标(1,6)的元素为：",arr2.visit_element( *args: 0,5))
arr2.Reshape((6,1))
print("将arr2重塑为列向量后坐标(6,1)的元素为：",arr2.visit_element( *args: 5,0))

print("")
print("接下来再展示三维数组：")
arr=N_matric(arr)
print("arr坐标(2,2,2)的元素为：",arr.visit_element( *args: 1,1,1))
arr.transpose()
print("将arr转置后坐标(2,2,2)的元素为：",arr.visit_element( *args: 1,1,1))
arr.Reshape((2,3,3))
print("将arr重塑为(2,3,3)后坐标(1,2,3)的元素为：",arr.visit_element( *args: 0,1,2))
arr.Reshape((3,3,2))
print("将arr重塑为(3,3,2)后坐标(1,2,3)的元素为：",arr.visit_element( *args: 0,1,2))
#注意这里因为超出元素访问界限因此会有提示！！！
print("将arr重塑为(3,3,2)后坐标(1,2,1)的元素为：",arr.visit_element( *args: 0,1,0))

print("")
print("接下来再展示四维数组：")
arr1=N_matric(arr1)
print("arr1坐标(2,2,2,2)的元素为：",arr1.visit_element( *args: 1,1,1,1))
arr.transpose()
print("将arr转置后坐标(1,2,1,2)的元素为：",arr1.visit_element( *args: 0,1,0,1))

print("")
print("接下来展示矩阵运算操作")
arr2.Reshape((2,3))
arr3=N_matric(arr3)
print("")
print("矩阵加法结果为：",(arr2+arr3).array)
print("")
print("矩阵减法结果为：",(arr2-arr3).array)
print("")
print("矩阵点乘结果为：",arr2.dotmul(arr3).array)
print("")
arr2.transpose()
three_dim_phalanx=(arr2*arr3)
print("arr2*arr3矩阵乘法结果为：",three_dim_phalanx.array)
print("arr2*arr3为方阵其迹为：",three_dim_phalanx.trace)
```

图 13: 人为编写一些测试数据

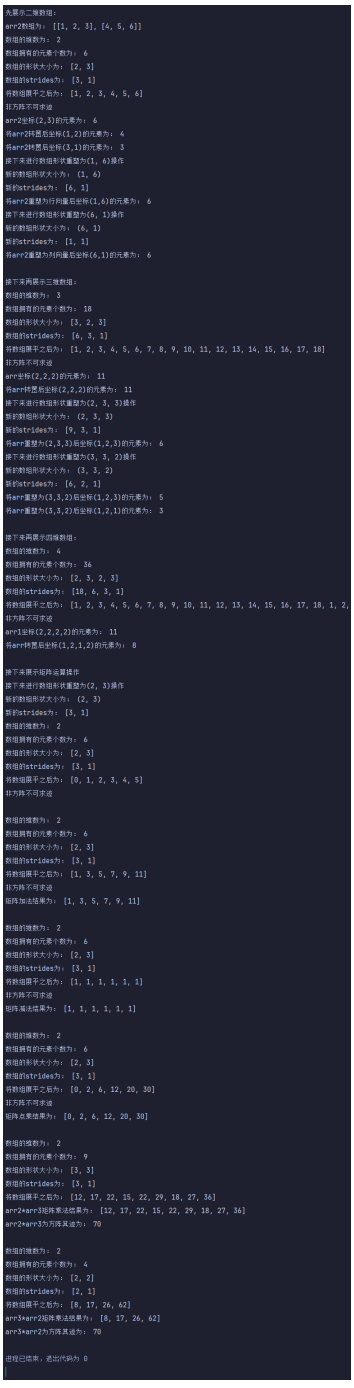

图 14: 最终运行测试数据结果