Lecture3

红色标注的语句，为重点。
紫色加粗标注的语句，说明在附件中有对应的代码示例可供参考，建议实践并掌握。
蓝色下划线标注的语句，说明给出了参考阅读链接，可依兴趣阅读。

1. Python文件操作-基本

想要在日常研究与工作中使用Python，就一定会和各种格式的文件读写打交道。Python生态环境提供了Python内置的文件读写解决方案以处理普通读写，以及各式各样的模块（module）以专门处理不同特殊文件的读写。

本部分关注内置读写方案。**本部分的实际操作见附件6.**

1.1 打开/关闭文件

——*open( )*方法

Python提供了file对象及配套的方法来进行大部分的文件操作。file对象由open( )函数创建/实例化：

$$\text{file object} = \text{open}(\text{file\_name} [, \text{access\_mode}])$$

其中，file_name为想要读取的文件名的字符串形式；access_mode为打开的形式，可以控制打开后对文件可执行的操作，如只读、重写等。

一个例子如下：

$$\text{f} = \text{open}(\text{'/Users/fowillwly/reading.txt'},\text{'a+'})$$

在这个语句中，open( )函数接受了2个参数，第一个字符串参数指定了要打开的文件路径，第二个参数指定打开的模式为a+模式，在这个模式下我们可以进行读操作和写操作，且写操作的起始指针被设置在文件末尾，即新的写入被默认为追加内容。open( )函数会返回一个符合要求的，实例化后的file对象，并赋予给变量f。此后，变量f就可以调用file对象的属性和方法。

open( )函数提供了多种access_mode参数可选，选取常用部分介绍如下：

参数名	功能
t	文本模式（默认）
r	只读模式
r+	可以读写，文件指针（理解成光标）设置在文件开头
w	写模式，若文件已存在则抹去原内容重写；若不存在会自己创建一个。要小心重写掉关键信息！
w+	读写模式，若文件已存在则抹去原内容重写；若不存在会自己创建一个。同样小心重写掉关键信息！

参数名	功能
a	打开文件以追加内容，若文件存在，不会抹去原内容，而是把文件指针设置在文件末尾
a+	在追加的同时可以读，若文件存在，不会抹去原内容，而是把文件指针设置在文件末尾

显然可见，带有+号意味着，在原有的模式基础上，增加读的功能。一般情况下，为了方便操作，建议大家选取模式时都带上+号。

——*with*语句的*open( )*方法

在实际操作中，我们往往建议使用with语句来执行open( )：

```
with open('/Users/fowillwly/reading.txt','a+') as f:
```

该语句执行后的效果与open( )是一致的，变量f依然会被赋予指定路径的文件对象。

文件对象往往会占用大量内存空间，所以及时清理它们对提升程序效率有很大帮助。在用open( )函数打开文件进行操作时，如果读写出现了错误，程序会中断，无法执行后面的清理操作。with语句可以让Python创建一个独立的环境，在该环境内出现错误，也会自动关闭掉环境内的资源占用，这就让内存的释放变得自动而高效。

两种打开文件方法的区别：https://blog.csdn.net/weixin_45743420/article/details/102912567

1.2 文件对象的基本操作

使用open( )方法打开一个文件后，我们得到的就是一个文件对象。文件对象提供了多种属性让我们了解文件信息，也提供了多种方法来对其进行操作。

（下列表述均假设f是一个实例化后的文件对象）

——文件对象常见的属性

f.mode表示该文件的访问模式（即上面表格中的若干），f.name表示该文件的名称，f.closed表示该文件是否被关闭。

——文件对象常见的方法

f.close( )：刷新缓冲区内还未写入的信息（在这里暂不解释缓冲机制），关闭文件对象。虽然文件对象的引用被重新指定给另一个文件时，Python会自动关闭以前的文件节省内存空间，但用f.close( )方法关闭文件是一个很好的习惯。

f.read(size)：从文件中读取size个字节的信息，如果未指定size即读取全部内容。

f.readlines( )：从文件中读取所有行，并返回一个列表，列表中的每一项表示一行内容。

f.write(str)：向文件中写入str的内容（必须为字符串形式）。注意⚠️，每次f.write( )并不会空行，需要执行f.write('\n')后才能空一行。\n为转义后的换行符。

关于转义字符，可参考：https://www.runoob.com/python/python-strings.html

1.3 os模块

实际操作时，文件常常会存在于不同目录下，需要我们和不同文件夹打交道。Python提供了os模块来处理文件和目录。os模块为内置模块，直接使用import os命令导入即可。

常用os模块命令如下：

os.mkdir(name)：在当前工作目录下创建名为name的新文件夹。

os.getcwd( )：获取当前工作目录。

os.rmdir(path)：删除路径为path的目录。

os.chdir(path)：将当前工作目录变更为path。

os.listdir(path)：返回path文件夹包含的文件名组成的列表。

注意⚠️：这里提到的path可以为相对路径和绝对路径。在实际操作中，如果想写出通用性高的程序（即不仅仅在你的个人设备上运行），建议大家尽量使用相对路径，避免程序交给他人使用时因为路径不正确而报错。

绝对路径和相对路径的区别可参考：https://blog.csdn.net/hgd613/article/details/8041662

2. Python文件操作-扩展

Python内置的文件读写方案往往只能处理简单的文字或者图片文件，但实际操作中往往会与Excel、SQL、PDF等文件打交道。第三方开发者为我们提供了一系列模块来解决这样的问题。下面给出一些实例，分别展示了Excel、z金融接口数据的读取。

理论上来说，Python可以通过扩展模块读取绝大部分类型的文件。所以有需求的话，百度或者在Github上查找一下往往能找到合适的模块，这也是Python生态系统强大的体现。

本部分关注扩展读写。**由于该部分无理论内容，完全以实践导向，因此主要内容见附件7。**

3. Scipy入门——Numpy篇

处理文件本质上是为了获取、处理数据，而这正是Python的强项。第三方开发者提供了名为Scipy的强大Ecosystem，服务于学术、工程各界，为矩阵运算、科学计算、符号计算、数据处理、图像绘制提供了有力支持。

如上图，我们主要介绍这套Ecosystem中的三个模块：Numpy模块提供数组类型扩展，以及对矩阵的支持，是其他模块的基础；Pandas模块提供对数据的结构化处理和分析功能；Matplotlib模块负责画图功能。

Lecture3

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2-D plotting

IP[y]: IPython
Enhanced interactive console

SymPy
Symbolic mathematics

pandas
Data structures & analysis

这一部分中，我们主要关注Numpy。Numpy是为了处理大量矩阵运算设计，为Python提供了扩展的数据结构——数组（原生Python没有数组这一数据类型，只有类似的列表List），并且进行了加速设计。Lecture1提到过，Python是一门动态语言，为开发者带来方便的同时，也造成了内存空间的占用和运行效率的低下，原生Python不适合进行大批量数据的处理。Numpy采用C语言编写底层，提供了大量预编译的函数供Python语句调用，这才使得Python能在数据时代大放光彩。

想要理解Numpy的逻辑，需要进行大量代码示例，**建议结合附件8阅读和操作。**

3.1 Numpy数据的基本单元——ndarray

在Numpy中，数据主要以ndarray对象进行存储，该对象也是Numpy主要操作的执行者。

ndarray被官方称为homogenous multidimensional array（同数据类型的多维数组），ndarray中的数据必须为同一类型，这是出于节省内存、加速运算效率考虑的；ndarray中的每个维度称为axe(轴)；对于ndarray的每个维度，该维度中的元素都由一系列非负数字索引。

一个常见的ndarray可能类似这样：

```
import numpy as np
```

```
a = np.array([[1,0,1],
              [0,1,1]])
```

如图，在导入numpy时，为了方便后续调用，我们往往将其约定俗成地命名为np。我们使用np.array( )函数来实例化一个ndarray对象。由图可见，a是一个ndarray对象，在这里是一个二维数组，因此具有两条轴，即两个axis，axis0的长度为2，指具有2个子数组；axis1的长度为3，指每个子数组中具有3个元素。

axes（轴）的概念非常重要，目前需要记住的是axes的数量等于ndarray的维度数，axes的长度等于该维度的元素数量。

值得注意的是，我们使用成对的[ ]来划分数组的上下级关系，一个三维ndarray对象示意如下：

```
b = np.array([
              [[1,0,1],
               [0,0,1]],

              [[0,1,1],
               [1,1,1]],
                      ])
```

b是一个三维的ndarray，可以看到，b就是在a的基础上再增加了一条axes（轴），第三条轴axis2的长度为2。

ndarray的一些常见属性如下：

属性名	指代
ndarray.ndim	该ndarray的维度，即axes（轴）的数量
ndarray.shape	该ndarray每个轴的长度
ndarray.size	该ndarray包含的元素数量，等于每个轴的长度乘积
ndarray.dtype	该ndarray中元素的类型，一个ndarray中所有元素属性相同

3.2 ndarray的创建

——最常见的创建方法：使用*np.array( )*函数

np.array( )函数接受一个序列（可以是列表List或者元组Tuple），并自动识别该序列的嵌套情况，如果是简单的一层序列，则形成一维ndarray，如果是两层嵌套，如[[1,2,3,4],[5,6,7,8]]，则形成二维ndarray，以此类推。

——特殊创建方法

为了解决一些特殊ndarray的生成，Numpy提供了其他方法：

方法名	效果
np.zeros(shape)	生成指定shape的全部由0组成的数组
np.ones(shape)	生成指定shape的全部由1组成的数组
np.empty(shape)	生成指定shape的空数组（每个元素均为随机生成）
np.arange(start,end,step)	生成一维数组，数组内的元素是从start开始，间隔为step，到end为止的序列。（不包含end，见实例）
np.linspace(start,end,num)	生成一维数组，数组内的元素是从start开始，到end为止，共有num个数字的等间隔序列。（包含end）

——函数创建方法

有时我们需要创建一个每个元素按照规则生成的特殊数组，例如，在一个二维数组中，每一个元素（x,y）的值是x，y的函数。Numpy提供了np.fromfunction( )函数来生成符合要求的数组。

```
func = lambda x,y : 10*x+y
c = np.fromfunction(func,(5,4),dtype=int)
print(c)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
```

如图，使用lambda语句（见Lecture2）快速定义了函数func后，我们将其和shape、想要生成的ndarray中元素属性dtype传入np.fromfunction( )即可。

注意⚠️：在这个例子中，可以看到函数的参数可以是另一个函数，这是个有趣的应用。

3.3 ndarray的基本操作

——基本数学运算

不同的ndarray之间可以进行加减运算，例如：

```
a = np.array([1,2,3,4],dtype='int')
b = np.ones((1,4),dtype='int')
a+b
```

```
array([[2, 3, 4, 5]])
```

但要注意区分乘法（*）和点乘（@）的区别，前者是两个数组之间的元素相乘，后者是线性代数中的点乘：

```
a = np.array([[1,1],
              [1,1]])
b = np.array([[1,1],
              [1,1]])
print(a*b)
print(a@b)
```

```
[[1 1]
 [1 1]]
[[2 2]
 [2 2]]
```

与数字计算类似，ndarray计算也提供+=，-=，*=的符号，a+=b 等同于 a=a+b，会将加分计算的结果直接赋予给a。但要注意，使用类似+=的运算时，要注意ndarray内元素的数据类型是固定不能改变的，违反了这一原则会导致报错：

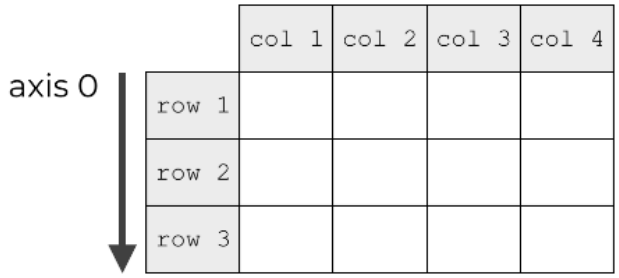
```
a = np.array([1,2,3,4],dtype='int')
b = np.ones((1,4),dtype='float')
b+=a
print(b)
a+=b
```

```
[[2. 3. 4. 5.]]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-bcd968844f99> in <module>
      3 b+=a
      4 print(b)
----> 5 a+=b

TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with casting rule 'same_kind'
```

——基于轴的运算

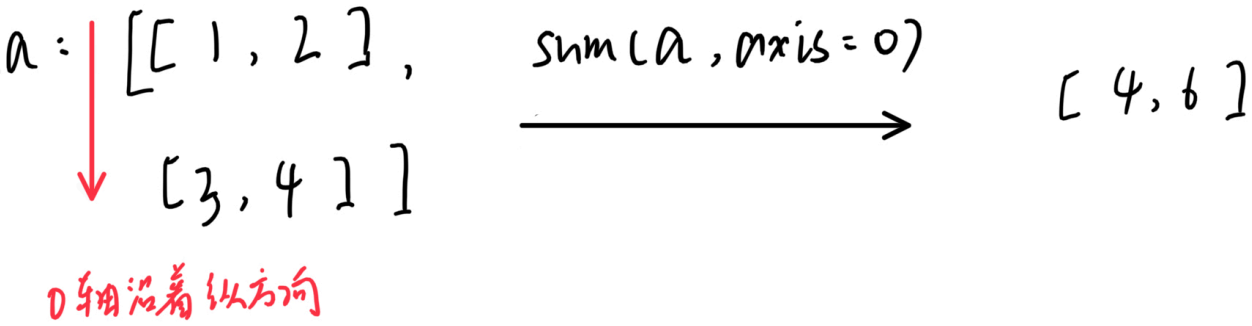
ndarray还可以很方便地进行最大值、最小值、求和的运算，但显而易见的是，ndarray往往会是多维数组，所以求最值、求和都必须指定按照哪一个维度来计算。

此前提到的axes（轴）就起到了指明方向的作用。我们以二维数组为例，阐释axes（轴）的功能：



在二维数组中，axis0轴，也就是第1条轴，对应的是纵方向；axis1轴，也就是第2条轴，对应的是横方向。大家可以用“纵横”这个词来进行记忆。注意，这里是将“轴”和“方向”对应起来的，也就是说，轴不仅仅表明了第几个维度，还代表着方向！

因此，在求最值、求和时，轴会被当作重要的参数传入，告诉Numpy应该按照哪个方向执行：



如果用更加抽象的词汇来描述，可能可以这么理解：告诉函数轴的编号，代表着执行函数时，该编号代表的维度会被折叠掉，数组会被降低一个维度。

对axes（轴）的详细解释：https://www.jianshu.com/p/f4e9407f9f9d

3.4 ndarray的索引和切片

Numpy毕竟还是面对Python编写的模块，因此对ndarray的索引和切片与对str、List的切片非常类似。在这里不再赘述，**参考附件7即可。**

3.5 ndarray的shape操作

Ndarray中的元素可以被很方便地重组，即改变不同轴的长度。就像24个小正方体，可以一会儿拼成2*2*6，一会儿拼成2*3*4一样。

Numpy提供了两种方法来进行重组，ndarray.reshape( )与ndarray.resize( )，两者的最大区别就是，reshape( )会返回一个全新的重组后的ndarray，不对原数组产生影响；resize( )会直接在原数组上进行维度重组：

```
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])
a.reshape(2,6)
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])
a.resize(2,6)
print(a)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

3.6 ndarray的堆叠与拆分

函数	效果
np.vstack([a,b])	将a和b垂直叠放，a上b下
np.hstack([a,b])	将a和b水平拼接，a左b右
np.vsplit(a,num1[,num2])	将a在第num1列（第num2列等）后进行拆分
np.hsplit(a,num1[,num2])	将a在第num1行（第num2行等）后进行拆分

3.7 ndarray的拷贝操作

实际操作中，我们往往会公用同样的一批数据，为了避免同一批数据在不同场景被修改造成损失，我们必须对Python和Numpy的拷贝机制有基本的了解。

——未拷贝

Numpy中，将一个变量对应的ndarray对象赋予给另一个变量并不构成拷贝。

```
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])

b = a

print(b is a)

a.resize(2,6)

print(b)
```

```
True
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

如图，这种情况下，仅仅是a和b这两个变量都指向了同一个ndarray对象而已，在内存空间中，这两个指针指向的地址相同，因此对a做操作，b也会发生变化。

——浅拷贝

Numpy中，使用ndarray.view( )函数进行浅拷贝，浅拷贝仅仅进行元素内容的共享，对元素的组成方式（如2*2*6或2*3*4）不进行共享。

```
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])
b = a.view()
print(b is a)
a.resize(2,6)
print(b)
b[0,0] = 10000
print(a)
```

```
False
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[10000     2     3     4     5     6]
 [    7     8     9    10    11    12]]
```

因此，b浅拷贝了a后，a进行的resize操作不会影响到b，而b进行的修改元素的操作可以影响到a。

——深拷贝

Numpy中，使用ndarray.copy( )函数进行深拷贝，深拷贝是完全的复制，不进行任何共享。

```
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])
b = a.copy()
print(b is a)
a.resize(2,6)
print(b)
b[0,0] = 10000
print(a)
```

```
False
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

显而易见，a和b咫尺天涯，毫无联系。