第一章 预备知识

一、Python基础

1. 列表推导式与条件赋值

在生成一个数字序列的时候，在 `Python` 中可以如下写出：

```
In [1]: L = []

In [2]: def my_func(x):
   ...:     return 2*x
   ...:

In [3]: for i in range(5):
   ...:     L.append(my_func(i))
   ...:

In [4]: L
Out[4]: [0, 2, 4, 6, 8]
```

事实上可以利用列表推导式进行写法上的简化：`[* for i in *]`。其中，第一个 `*` 为映射函数，其输入为后面 `i` 指代的内容，第二个 `*` 表示迭代的对象。

```
In [5]: [my_func(i) for i in range(5)]
Out[5]: [0, 2, 4, 6, 8]
```

列表表达式还支持多层嵌套，如下面的例子中第一个 `for` 为外层循环，第二个为内层循环：

```
In [6]: [m+'_'+n for m in ['a', 'b'] for n in ['c', 'd']]
Out[6]: ['a_c', 'a_d', 'b_c', 'b_d']
```

除了列表推导式，另一个实用的语法糖是条件赋值，其形式为 `value = a if condition else b`：

```
In [7]: value = 'cat' if 2>1 else 'dog'

In [8]: value
Out[8]: 'cat'
```

等价于如下的写法：

```
a, b = 'cat', 'dog'
condition = 2 > 1 # 此时为True
if condition:
    value = a
else:
    value = b
```

下面举一个例子，截断列表中超过5的元素：

```
In [9]: L = [1, 2, 3, 4, 5, 6, 7]

In [10]: [i if i <= 5 else 5 for i in L]
Out[10]: [1, 2, 3, 4, 5, 5, 5]
```

2. 匿名函数与map方法

有一些函数的定义具有清晰简单的映射关系，例如上面的 `my_func` 函数，这时候可以用匿名函数的方法简洁地表示：

```
In [11]: my_func = lambda x: 2*x

In [12]: my_func(3)
Out[12]: 6

In [13]: multi_para_func = lambda a, b: a + b

In [14]: multi_para_func(1, 2)
Out[14]: 3
```

但上面的用法其实违背了“匿名”的含义，事实上它往往在无需多处调用的场合进行使用，例如上面列表推导式中的例子，用户不关心函数的名字，只关心这种映射的关系：

```
In [15]: [(lambda x: 2*x)(i) for i in range(5)]
Out[15]: [0, 2, 4, 6, 8]
```

对于上述的这种列表推导式的匿名函数映射， Python 中提供了 map 函数来完成，它返回的是一个 map 对象，需要通过 list 转为列表：

```
In [16]: list(map(lambda x: 2*x, range(5)))
Out[16]: [0, 2, 4, 6, 8]
```

对于多个输入值的函数映射，可以通过追加迭代对象实现：

```
In [17]: list(map(lambda x, y: str(x)+'_'+y, range(5), list('abcde')))
Out[17]: ['0_a', '1_b', '2_c', '3_d', '4_e']
```

3. zip对象与enumerate方法

zip函数能够把多个可迭代对象打包成一个元组构成的可迭代对象，它返回了一个 zip 对象，通过 tuple, list 可以得到相应的打包结果：

```
In [18]: L1, L2, L3 = list('abc'), list('def'), list('hij')

In [19]: list(zip(L1, L2, L3))
Out[19]: [('a', 'd', 'h'), ('b', 'e', 'i'), ('c', 'f', 'j')]

In [20]: tuple(zip(L1, L2, L3))
Out[20]: (('a', 'd', 'h'), ('b', 'e', 'i'), ('c', 'f', 'j'))
```

往往会在循环迭代的时候使用到 zip 函数：

```
In [21]: for i, j, k in zip(L1, L2, L3):
   ....:     print(i, j, k)
   ....:
a d h
b e i
c f j
```

enumerate 是一种特殊的打包，它可以在迭代时绑定迭代元素的遍历序号：

```
In [22]: L = list('abcd')

In [23]: for index, value in enumerate(L):
   ....:     print(index, value)
   ....:
0 a
1 b
2 c
3 d
```

用 zip 对象也能够简单地实现这个功能：

```
In [24]: for index, value in zip(range(len(L)), L):
   ....:     print(index, value)
   ....:
0 a
1 b
2 c
3 d
```

当需要对两个列表建立字典映射时，可以利用 zip 对象：

```
In [25]: dict(zip(L1, L2))
Out[25]: {'a': 'd', 'b': 'e', 'c': 'f'}
```

既然有了压缩函数，那么 Python 也提供了 * 操作符和 zip 联合使用来进行解压操作：

```
In [26]: zipped = list(zip(L1, L2, L3))

In [27]: zipped
Out[27]: [('a', 'd', 'h'), ('b', 'e', 'i'), ('c', 'f', 'j')]

In [28]: list(zip(*zipped)) # 三个元组分别对应原来的列表
Out[28]: [('a', 'b', 'c'), ('d', 'e', 'f'), ('h', 'i', 'j')]
```

二、Numpy基础

1. np数组的构造

最一般的方法是通过 array 来构造：

```
In [29]: import numpy as np

In [30]: np.array([1,2,3])
Out[30]: array([1, 2, 3])
```

下面讨论一些特殊数组的生成方式：

【a】等差序列： np.linspace, np.arange

```
In [31]: np.linspace(1,5,11) # 起始、终止（包含）、样本个数
Out[31]: array([1. , 1.4, 1.8, 2.2, 2.6, 3. , 3.4, 3.8, 4.2, 4.6, 5. ])

In [32]: np.arange(1,5,2) # 起始、终止（不包含）、步长
Out[32]: array([1, 3])
```

【b】特殊矩阵： zeros, eye, full

```
In [33]: np.zeros((2,3)) # 传入元组表示各维度大小
Out[33]:
array([[0., 0., 0.],
       [0., 0., 0.]])

In [34]: np.eye(3) # 3*3的单位矩阵
Out[34]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In [35]: np.eye(3, k=1) # 偏移主对角线1个单位的伪单位矩阵
Out[35]:
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])

In [36]: np.full((2,3), 10) # 元组传入大小，10表示填充数值
Out[36]:
array([[10, 10, 10],
       [10, 10, 10]])

In [37]: np.full((2,3), [1,2,3]) # 通过传入列表填充每列的值
Out[37]:
array([[1, 2, 3],
       [1, 2, 3]])
```

【c】随机矩阵： np.random

最常用的随机生成函数为 rand, randn, randint, choice ， 它们分别表示0-1均匀分布的随机数组、标准正态的随机数组、随机整数组和随机列表抽样：

```
In [38]: np.random.rand(3) # 生成服从0-1均匀分布的三个随机数
Out[38]: array([0.69055559, 0.96144845, 0.63623802])

In [39]: np.random.rand(3, 3) # 注意这里传入的不是元组，每个维度大小分开输入
Out[39]:
array([[0.00549291, 0.02592448, 0.26250209],
       [0.90704431, 0.02531117, 0.58160792],
       [0.14875349, 0.73999109, 0.491112  ]])
```

对于服从区间 a 到 b 上的均匀分布可以如下生成：

```
In [40]: a, b = 5, 15

In [41]: (b - a) * np.random.rand(3) + a
Out[41]: array([ 5.76111065, 14.1489915 ,  7.78442235])
```

`randn` 生成了 $N(\mathbf{0},\mathbf{I})$ 的标准正态分布：

```
In [42]: np.random.randn(3)
Out[42]: array([-0.36455115, -0.02718377,  0.31660922])

In [43]: np.random.randn(2, 2)
Out[43]:
array([[ 0.81948272,  1.34361096],
       [-0.17046773, -0.91937857]])
```

对于服从方差为 σ^2 均值为 μ 的一元正态分布可以如下生成：

```
In [44]: sigma, mu = 2.5, 3

In [45]: mu + np.random.randn(3) * sigma
Out[45]: array([2.21173398, 1.39806909, 6.16357612])
```

`randint` 可以指定生成随机整数的最小值最大值和维度大小：

```
In [46]: low, high, size = 5, 15, (2,2)

In [47]: np.random.randint(low, high, size)
Out[47]:
array([[6, 6],
       [6, 9]])
```

`choice` 可以从给定的列表中，以一定概率和方式抽取结果，当不指定概率时为均匀采样，默认抽取方式为有放回抽样：

```
In [48]: my_list = ['a', 'b', 'c', 'd']

In [49]: np.random.choice(my_list, 2, replace=False, p=[0.1, 0.7, 0.1 ,0.1])
Out[49]: array(['d', 'b'], dtype='<U1')

In [50]: np.random.choice(my_list, (3,3))
Out[50]:
array([['c', 'a', 'c'],
       ['b', 'b', 'b'],
       ['a', 'c', 'c']], dtype='<U1')
```

当返回的元素个数与原列表相同时，等价于使用 `permutation` 函数，即打散原列表：

```
In [51]: np.random.permutation(my_list)
Out[51]: array(['b', 'd', 'a', 'c'], dtype='<U1')
```

最后，需要提到的是随机种子，它能够固定随机数的输出结果：

```
In [52]: np.random.seed(0)

In [53]: np.random.rand()
Out[53]: 0.5488135039273248

In [54]: np.random.seed(0)

In [55]: np.random.rand()
Out[55]: 0.5488135039273248
```

2. np数组的变形与合并

【a】转置： `T`

```
In [56]: np.zeros((2,3)).T
Out[56]:
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

【b】合并操作： `r_, c_`

对于二维数组而言， `r_`和`c_`分别表示上下合并和左右合并：

```
In [57]: np.r_[np.zeros((2,3)),np.zeros((2,3))]
Out[57]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])

In [58]: np.c_[np.zeros((2,3)),np.zeros((2,3))]
Out[58]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

一维数组和二维数组进行合并时，应当把其视作列向量，在长度匹配的情况下只能够使用左右合并的`c_`操作：

```
In [59]: try:
   ....:     np.r_[np.array([0,0]),np.zeros((2,1))]
   ....: except Exception as e:
   ....:     Err_Msg = e
   ....:

In [60]: Err_Msg
Out[60]: ValueError('all the input arrays must have same number of dimensions, but the
array at index 0 has 1 dimension(s) and the array at index 1 has 2 dimension(s)')

In [61]: np.r_[np.array([0,0]),np.zeros(2)]
Out[61]: array([0., 0., 0., 0.])

In [62]: np.c_[np.array([0,0]),np.zeros((2,3))]
Out[62]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

【c】维度变换： `reshape`

`reshape`能够帮助用户把原数组按照新的维度重新排列。在使用时有两种模式，分别为`C`模式和`F`模式，分别以逐行和逐列的顺序进行填充读取。

```
In [63]: target = np.arange(8).reshape(2,4)

In [64]: target
Out[64]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

In [65]: target.reshape((4,2), order='C') # 按照行读取和填充
Out[65]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])

In [66]: target.reshape((4,2), order='F') # 按照列读取和填充
Out[66]:
array([[0, 2],
       [4, 6],
       [1, 3],
       [5, 7]])
```

特别地，由于被调用数组的大小是确定的，reshape允许有一个维度存在空缺，此时只需填充-1即可：

```
In [67]: target.reshape((4,-1))
Out[67]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

下面将 n*1 大小的数组转为1维数组的操作是经常使用的：

```
In [68]: target = np.ones((3,1))

In [69]: target
Out[69]:
array([[1.],
       [1.],
       [1.]])

In [70]: target.reshape(-1)
Out[70]: array([1., 1., 1.])
```

3. np数组的切片与索引

数组的切片模式支持使用 slice 类型的 start:end:step 切片，还可以直接传入列表指定某个维度的索引进行切片：

```
In [71]: target = np.arange(9).reshape(3,3)

In [72]: target
Out[72]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [73]: target[:-1, [0,2]]
Out[73]:
array([[0, 2],
       [3, 5]])
```

此外，还可以利用 np.ix_ 在对应的维度上使用布尔索引，但此时不能使用 slice 切片：

```
In [74]: target[np.ix_([True, False, True], [True, False, True])]
Out[74]:
array([[0, 2],
       [6, 8]])

In [75]: target[np.ix_([1,2], [True, False, True])]
Out[75]:
array([[3, 5],
       [6, 8]])
```

当数组维度为1维时，可以直接进行布尔索引，而无需 np.ix_ ：

```
In [76]: new = target.reshape(-1)

In [77]: new[new%2==0]
Out[77]: array([0, 2, 4, 6, 8])
```

4. 常用函数

为了简单起见，这里假设下述函数输入的数组都是一维的。

【a】 where

where 是一种条件函数，可以指定满足条件与不满足条件位置对应的填充值：

```
In [78]: a = np.array([-1,1,-1,0])

In [79]: np.where(a>0, a, 5) # 对应位置为True时填充a对应元素，否则填充5
Out[79]: array([5, 1, 5, 5])
```

【b】 nonzero, argmax, argmin

这三个函数返回的都是索引，`nonzero` 返回非零数的索引，`argmax, argmin` 分别返回最大和最小数的索引：

```
In [80]: a = np.array([-2,-5,0,1,3,-1])

In [81]: np.nonzero(a)
Out[81]: (array([0, 1, 3, 4, 5], dtype=int64),)

In [82]: a.argmax()
Out[82]: 4

In [83]: a.argmin()
Out[83]: 1
```

【c】`any, all`

`any` 指当序列至少 存在一个 `True` 或非零元素时返回 `True`，否则返回 `False`

`all` 指当序列元素 全为 `True` 或非零元素时返回 `True`，否则返回 `False`

```
In [84]: a = np.array([0,1])

In [85]: a.any()
Out[85]: True

In [86]: a.all()
Out[86]: False
```

【d】`cumprod, cumsum, diff`

`cumprod, cumsum` 分别表示累乘和累加函数，返回同长度的数组，`diff` 表示和前一个元素做差，由于第一个元素为缺失值，因此在默认参数情况下，返回长度是原数组减1

```
In [87]: a = np.array([1,2,3])

In [88]: a.cumprod()
Out[88]: array([1, 2, 6], dtype=int32)

In [89]: a.cumsum()
Out[89]: array([1, 3, 6], dtype=int32)

In [90]: np.diff(a)
Out[90]: array([1, 1])
```

【e】统计函数

常用的统计函数包括 `max, min, mean, median, std, var, sum, quantile`，其中分位数计算是全局方法，因此不能通过 `array.quantile` 的方法调用：

```
In [91]: target = np.arange(5)

In [92]: target
Out[92]: array([0, 1, 2, 3, 4])

In [93]: target.max()
Out[93]: 4

In [94]: np.quantile(target, 0.5) # 0.5分位数
Out[94]: 2.0
```

但是对于含有缺失值的数组，它们返回的结果也是缺失值，如果需要略过缺失值，必须使用 `nan*` 类型的函数，上述的几个统计函数都有对应的 `nan*` 函数。

```
In [95]: target = np.array([1, 2, np.nan])

In [96]: target
Out[96]: array([ 1.,  2., nan])

In [97]: target.max()
Out[97]: nan

In [98]: np.nanmax(target)
Out[98]: 2.0

In [99]: np.nanquantile(target, 0.5)
Out[99]: 1.5
```

对于协方差和相关系数分别可以利用 `cov, corrcoef` 如下计算：

```
In [100]: target1 = np.array([1,3,5,9])

In [101]: target2 = np.array([1,5,3,-9])

In [102]: np.cov(target1, target2)
Out[102]:
array([[ 11.66666667, -16.66666667],
       [-16.66666667,  38.66666667]])

In [103]: np.corrcoef(target1, target2)
Out[103]:
array([[ 1.        , -0.78470603],
       [-0.78470603,  1.        ]])
```

5. 向量与矩阵的计算

【a】向量内积： `dot`

$$\mathbf{a} \cdot \mathbf{b} = \sum_{\mathrm{i}} \mathrm{a_i}\,\mathrm{b_i}$$

```
In [104]: a = np.array([1,2,3])

In [105]: b = np.array([1,3,5])

In [106]: a.dot(b)
Out[106]: 22
```

【b】向量范数和矩阵范数： `np.linalg.norm`

在矩阵范数的计算中，最重要的是 `ord` 参数，可选值如下：

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
‘fro’	Frobenius norm	–
‘nuc’	nuclear norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

```
In [107]: martix_target =  np.arange(4).reshape(-1,2)

In [108]: martix_target
Out[108]:
array([[0, 1],
       [2, 3]])

In [109]: np.linalg.norm(martix_target, 'fro')
Out[109]: 3.7416573867739413

In [110]: np.linalg.norm(martix_target, np.inf)
Out[110]: 5.0

In [111]: np.linalg.norm(martix_target, 2)
Out[111]: 3.702459173643833
```

```
In [112]: vector_target =  np.arange(4)

In [113]: vector_target
Out[113]: array([0, 1, 2, 3])

In [114]: np.linalg.norm(vector_target, np.inf)
Out[114]: 3.0

In [115]: np.linalg.norm(vector_target, 2)
Out[115]: 3.7416573867739413

In [116]: np.linalg.norm(vector_target, 3)
Out[116]: 3.3019272488946263
```

【c】矩阵乘法：@

$$[\mathbf{A}_{\mathrm{m}\times\mathrm{p}}\mathbf{B}_{\mathrm{p}\times\mathrm{n}}]_{\mathrm{ij}}=\sum_{\mathrm{k}=1}^{\mathrm{p}}\mathbf{A}_{\mathrm{ik}}\mathbf{B}_{\mathrm{kj}}$$

```
In [117]: a = np.arange(4).reshape(-1,2)

In [118]: a
Out[118]:
array([[0, 1],
       [2, 3]])

In [119]: b = np.arange(-4,0).reshape(-1,2)

In [120]: b
Out[120]:
array([[-4, -3],
       [-2, -1]])

In [121]: a@b
Out[121]:
array([[ -2,  -1],
       [-14,  -9]])
```

三、练习

Ex1：改进矩阵计算的性能

设 Z 为 $m\times n$ 的矩阵， B 和 U 分别是 $m\times p$ 和 $n\times p$ 的矩阵， B_i 为 B 的第 i 行， U_j 为 U 的第 j 行，下面定义 $R=\sum_{i=1}^{m}\sum_{j=1}^{n}\|B_i-U_j\|_2^2 Z_{ij}$ ，其中 $\|\mathbf{a}\|_2^2$ 表示向量 $\mathbf{a}$ 的分量平方和 $\sum_i a_i^2$ 。

现有某人根据如下给定的样例数据计算 R 的值，请充分利用 Numpy 中的函数，基于此问题改进这段代码的性能。

```
In [122]: np.random.seed(0)

In [123]: m, n, p = 100, 80, 50

In [124]: B = np.random.randint(0, 2, (m, p))

In [125]: U = np.random.randint(0, 2, (n, p))

In [126]: Z = np.random.randint(0, 2, (m, n))
```

```
In [127]: L_res = []

In [128]: for i in range(m):
   .....:     for j in range(n):
   .....:         norm_value = ((B[i]-U[j])**2).sum()
   .....:         L_res.append(norm_value*Z[i][j])
   .....:

In [129]: sum(L_res)
Out[129]: 101107
```

Ex2：连续整数的最大长度

输入一个整数的 Numpy 数组，返回其中连续整数子数组的最大长度。例如，输入 [1,2,5,6,7]，[5,6,7]为具有最大长度的连续整数子数组，因此输出3；输入[3,2,1,2,3,4,6]，[1,2,3,4]为具有最大长度的连续整数子数组，因此输出4。请充分利用 Numpy 的内置函数完成。（提示：考虑使用 nonzero, diff 函数）