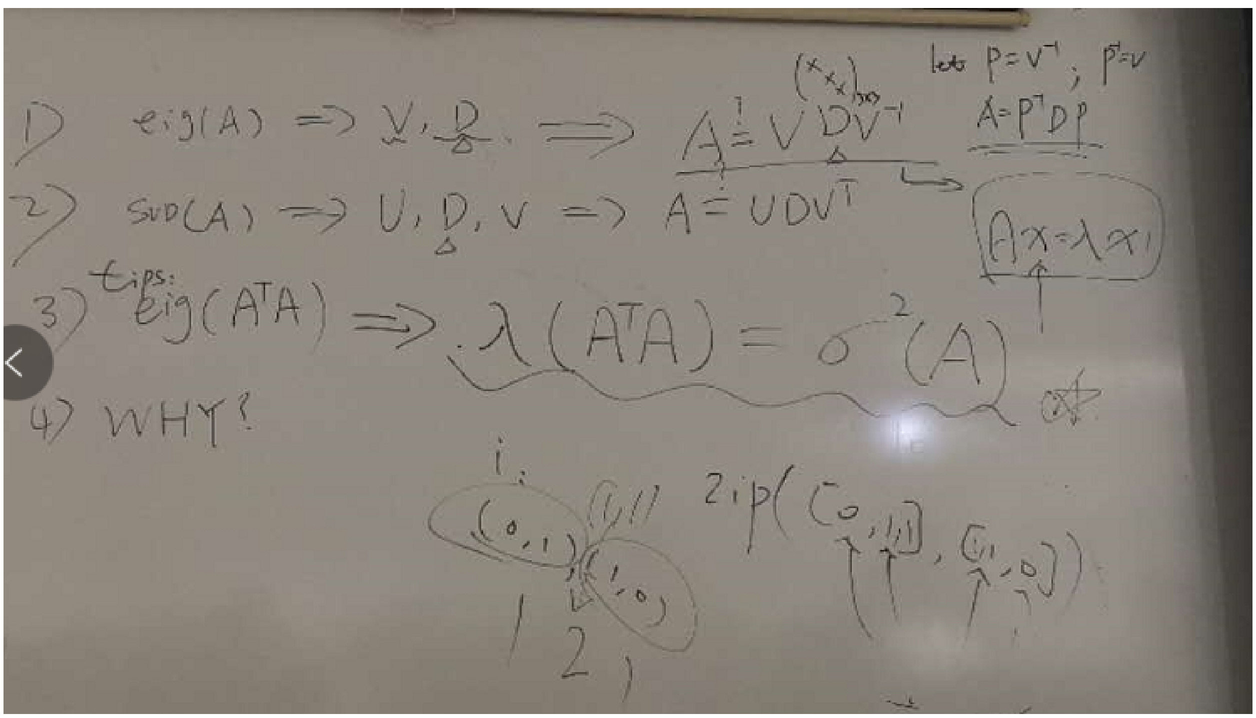
Week 3 - Task

- 杨辉三角代码 zip
- 特征值证明 numpy操作一遍
- 尝试迭代器 yield 杨辉三角
- 尽量列表迭代器 不要for
- 行维度 列维度 rank关系



λ(ATA) = σ^2(A)

▼ 行秩 列秩 矩阵秩关系

- **行秩**: 把矩阵每一行看成一个向量，矩阵可被认为由这些行向量组成。矩阵的行向量的秩，被称为矩阵的行秩

- **列秩**: 把矩阵每一列看成一个向量，矩阵可被认为由这些列向量组成。矩阵的列向量的秩，被称为矩阵的列秩

初等行（列）变换不改变行秩：对换，向量不变；乘常数，向量本可以使用ka形式表示。

初等行（列）变换不改变列（行）秩

$\boxed{\text{定理：矩阵的行秩} = \text{矩阵的列秩}}$

证：任何矩阵A都可经过初等变换变为$\begin{pmatrix} E_r & 0 \\ 0 & 0 \end{pmatrix}$形式，

而它的行秩为r，列秩也为r。

又，初等变换不改变矩阵的行秩与列秩，

所以，A的行秩 = r = A的列秩

定义2：矩阵的行秩 = 矩阵的列秩，统称为矩阵的秩。

记为r(A),或rankA，或秩A。

$\boxed{\text{推论：矩阵的初等变换不改变矩阵的秩。}}$

行秩 = 列秩 = 矩阵秩

定理：n阶方阵A，

$$r(A)=n \Leftrightarrow \text{A的n个行（列）向量线性无关}$$
$$\Leftrightarrow |A| \neq 0, \text{即A为可逆矩阵（也称为满秩矩阵）}$$

$$r(A)<n \Leftrightarrow \text{A的n个行（列）向量线性相关}$$
$$\Leftrightarrow |A| = 0$$

▼ 特征值、特征向量

定义：

设A是n阶方阵，如果数λ和n维非零列向量x使关系式$Ax=\lambda x$成立，那么这样的数λ称为矩阵A特征值，非零向量x称为A的对应于特征值λ的特征向量。

求矩阵$A=\begin{pmatrix}1&0&0\\0&2&0\\0&0&3\end{pmatrix}$的特征值。

$$|A-\lambda I|=\begin{vmatrix}1-\lambda&0&0\\0&2-\lambda&0\\0&0&3-\lambda\end{vmatrix}=(1-\lambda)(2-\lambda)(3-\lambda)$$

$$\lambda_1=1,\lambda_2=2,\lambda_3=3$$

对角矩阵$\left\{\begin{matrix}\lambda_1&&&\\&\lambda_2&&\\&&\cdots&\\&&&\lambda_n\end{matrix}\right\}$的特征值就是主对角矩阵。

方阵A的对应于不同特征值的特征向量线性无关。

矩阵	特征值
A	λ
A^T	λ
kA	$k\lambda$
A^m	λ^m
A^{-1}	λ^{-1}

$$A = \begin{pmatrix} 4 & 2 & -5 \\ 6 & 4 & -9 \\ 5 & 3 & -7 \end{pmatrix}$$

计算：A的特征值和特征向量。

$$|\lambda E - A| = \begin{vmatrix} \lambda - 4 & -2 & 5 \\ -6 & \lambda - 4 & 9 \\ -5 & -3 & \lambda + 7 \end{vmatrix} = 0$$

则 $\lambda_1 = \lambda_2 = 0 \qquad \lambda_3 = 1$

$$\lambda_1 = 1 \Rightarrow (E - A)x = 0$$

$$E - A = \begin{pmatrix} -3 & -2 & 5 \\ -6 & -3 & 9 \\ -5 & -3 & 8 \end{pmatrix}$$

化简

$$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$

$$(E - A)x = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 0$$

令x=1，便可得出一个基础解系：

$$\xi_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

同理当$\lambda_2 = \lambda_3 = 0$时，得出：

$$(E - A)x = \begin{pmatrix} -2 & 0 & 1 \\ 0 & -2 & 3 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 0 \quad \begin{cases} -2x_1 + x_3 = 0 \\ -2x_2 + 3x_3 = 0 \end{cases}$$

同样可以得出特征向量：

$$\xi_2 = \xi_3 = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$$

▼ 特征值分解

如果说一个向量v是方阵A的特征向量，将一定可以表示成下面的形式： $Av = \lambda v$

这时候λ就被称为特征向量v对应的特征值，一个矩阵的一组特征向量是一组正交向量。特征值分解是将一个矩阵分解成下面的形式：

$$A = Q\Sigma Q^{-1}$$

其中Q是这个矩阵A的特征向量组成的矩阵，Σ是一个对角阵，每一个对角线上的元素就是一个特征值。我这里引用了一些参考文献中的内容来说明一下。

也就是之前说的：提取这个矩阵最重要的特征。总结一下，特征值分解可以得到特征值与特征向量，特征值表示的是这个特征到底有多重要，而特征向量表示这个特征是什么，可以将每一个特征向量理解为一个线性的子空间，我们可以利用这些线性的子空间干很多的事情。不过，特征值分解也有很多的局限，比如说变换的矩阵必须是方阵。

```
matrix=np.array([[8,1,6],[3,5,7],[4,9,2]])
print(matrix)
```

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

```
Evals, Evecs = np.linalg.eig(matrix)
print('Eigen values: \n', Evals)
#Q:
print('Eigen Vectors:\n', Evecs) #NOTICE: eigenvectors are stored in columns
print('AX , lambda*X:\n', matrix@Evecs[:,0], '\n', Evals[0]*Evecs[:,0])
```

```
Eigen values:
 [15.          4.89897949 -4.89897949]
Eigen Vectors:
 [[-0.57735027 -0.81305253 -0.34164801]
 [-0.57735027  0.47140452 -0.47140452]
 [-0.57735027  0.34164801  0.81305253]]
AX , lambda*X:
 [-8.66025404 -8.66025404 -8.66025404]
 [-8.66025404 -8.66025404 -8.66025404]
```

```
#Q^-1
Evecs_rev=np.linalg.inv(Evecs)
```

```
#sigma
D=np.diag(Evals)

print(D)
print(Evecs_rev)
print(Evecs)
```

```
[[15.          0.          0.        ]
 [ 0.          4.89897949  0.        ]
 [ 0.          0.         -4.89897949]]
[[-0.57735027 -0.57735027 -0.57735027]
 [-0.78656609  0.70710678  0.07945931]
 [-0.07945931 -0.70710678  0.78656609]]
[[-0.57735027 -0.81305253 -0.34164801]
 [-0.57735027  0.47140452 -0.47140452]
 [-0.57735027  0.34164801  0.81305253]]
```

```
#matrix A:
print(Evecs@D@Evecs_rev)
```

```
[[8. 1. 6.]
 [3. 5. 7.]
 [4. 9. 2.]]
```

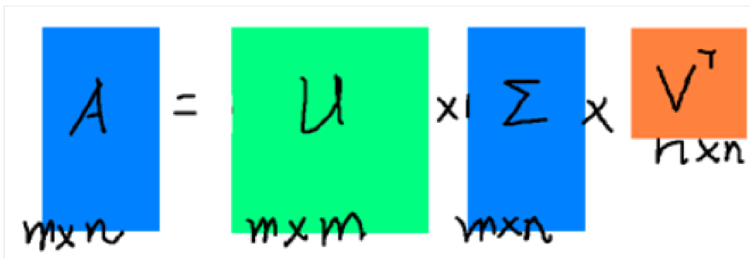
▼ SVD分解

任何一个矩阵A都可以分解成：

要特征呢？奇异值分解可以用来干这个事情，奇异值分解是一个能适用于任意的矩阵的一种分解的方法：

$$A = U\Sigma V^T$$

假设A是一个N * M的矩阵，那么得到的U是一个N * N的方阵（里面的向量是正交的，U里面的向量称为左奇异向量），Σ是一个N * M的矩阵（除了对角线的元素都是0，对角线上的元素称为奇异值），V'(V的转置)是一个N * N的矩阵，里面的向量也是正交的，V里面的向量称为右奇异向量），从图片来反映几个相乘的矩阵的大小可得下面的图片



U和V都是酉矩阵，即满足 $U^TU = I, V^TV = I$

其中U和V均为单位正交阵，即有$UU^T = I$和$VV^T = I$，U称为左奇异矩阵，V称为右奇异矩阵，Σ仅在主对角线上有值，我们称它为奇异值，其它元素均为0。

而奇异值分解的几何含义为：对于任何的一个矩阵，我们要找到一组两两正交单位向量序列，使得矩阵作用在此向量序列上后得到新的向量序列保持两两正交。

继续拿1.1节的例子进一步阐述，奇异值的几何含义为：这组变换后的新的向量序列的长度。

```
u,sigma,vt=np.linalg.svd(matrix)
print(u)
```

```
[[-5.77350269e-01  7.07106781e-01  4.08248290e-01]
 [-5.77350269e-01  5.91193761e-15 -8.16496581e-01]
 [-5.77350269e-01 -7.07106781e-01  4.08248290e-01]]
```

```
sigma=np.diag(sigma)
print(sigma)
```

我们首先回顾下特征值和特征向量的定义如下：

$$Ax = \lambda x$$

其中A是一个$n \times n$的实对称矩阵，是一个维向量，则我们说是矩阵A的一个特征值，而是矩阵A的特征值所对应的特征向量。

```
[[15.          0.          0.        ]
 [ 0.          6.92820323  0.        ]
 [ 0.          0.          3.46410162]]
```

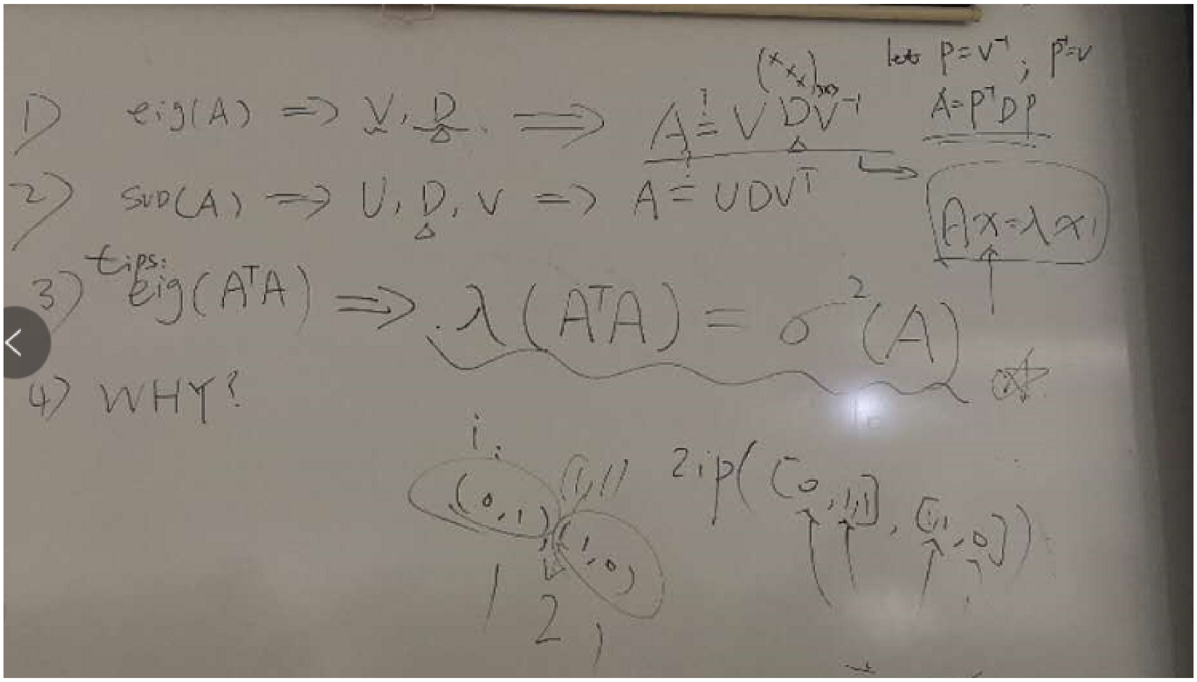
```
print(vt)
```

```
[[-5.77350269e-01 -5.77350269e-01 -5.77350269e-01]
 [ 4.08248290e-01 -8.16496581e-01  4.08248290e-01]
 [ 7.07106781e-01 -1.16573418e-14 -7.07106781e-01]]
```

```
#matrix A:
print(u@sigma@vt)
```

```
[[8. 1. 6.]
 [3. 5. 7.]
 [4. 9. 2.]]
```

▼ 奇异值、特征值关系推导



SVD也是对矩阵进行分解，但是和特征分解不同，SVD并不要求要分解的矩阵为方阵。假设我们的矩阵A是一个$m \times n$的矩阵，那么我们定义矩阵A的SVD为：

$$A = U\Sigma V^T$$

其中U是一个$m \times m$的矩阵，是一个$m \times n$的矩阵，除了主对角线上的元素以外全为0，主对角线上的每个元素都称为奇异值，V是一个$n \times n$的矩阵。U和V都是酉矩阵，即满足$U^TU = I, V^TV = I$。下图可以很形象的看出上面SVD的定义：

如果我们将A的转置和A做矩阵乘法，那么会得到$n \times n$的一个方阵A^TA。既然A^TA是方阵，那么我们就可以进行特征分解，得到的特征值和特征向量满足下式：

$$(A^TA)v_i = \lambda_i v_i$$

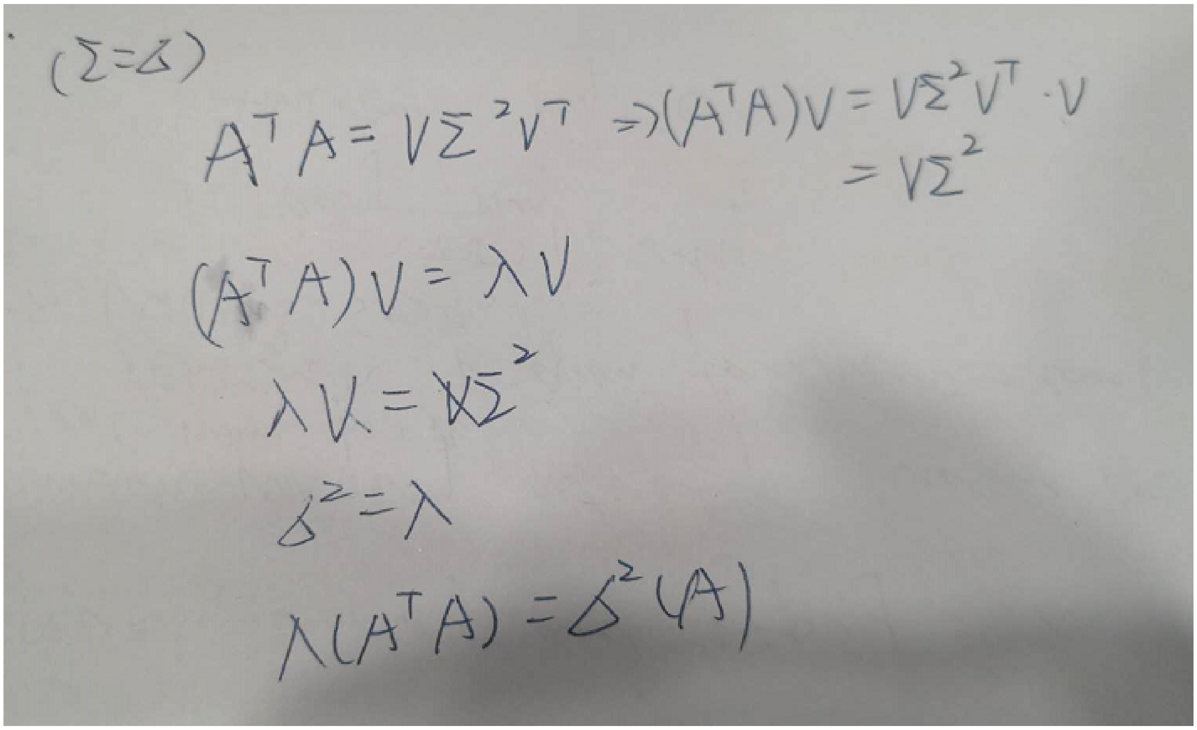
这样我们就可以得到矩阵A^TA的n个特征值和对应的n个特征向量了。将A^TA的所有特征向量张成一个$n \times n$的矩阵V，就是我们SVD公式里面的V矩阵了。一般我们将V中的每个特征向量叫做A的右奇异向量。

如果我们将A和A的转置做矩阵乘法，那么会得到$m \times m$的一个方阵AA^T。既然AA^T是方阵，那么我们就可以进行特征分解，得到的特征值和特征向量满足下式：

$$(AA^T)u_i = \lambda_i u_i$$

这样我们就可以得到矩阵AA^T的m个特征值和对应的m个特征向量了。将AA^T的所有特征向量张成一个$m \times m$的矩阵U，就是我们SVD公式里面的U矩阵了。一般我们将U中的每个特征向量叫做A的左奇异向量。

$$A = U\Sigma V^T \Rightarrow A^T = V\Sigma^T U^T \Rightarrow A^T A = V\Sigma^T U^T U\Sigma V^T = V\Sigma^2 V^T$$




```
matrix
```

```
array([[8, 1, 6],
       [3, 5, 7],
       [4, 9, 2]])
```

```
matrix.T
```

```
array([[8, 3, 4],
       [1, 5, 9],
       [6, 7, 2]])
```

```
A=matrix@matrix.T
print(A)
```

```
[[101  71  53]
 [ 71  83  71]
 [ 53  71 101]]
```

```
Evals1, Evecs1 = np.linalg.eig(A)
print(Evals1)
print(Evecs1)
```

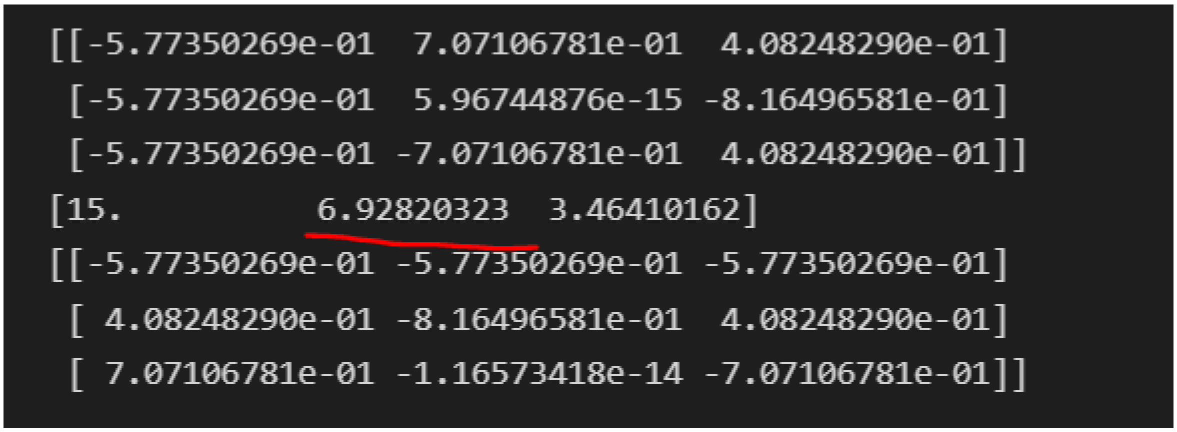
```
[225.  48.  12.]
[[-5.77350269e-01 -7.07106781e-01  4.08248290e-01]
 [-5.77350269e-01  1.69222025e-16 -8.16496581e-01]
 [-5.77350269e-01  7.07106781e-01  4.08248290e-01]]
```

```
u,sigma,vt=np.linalg.svd(matrix)
print(u)
print(sigma)
print(vt)
```

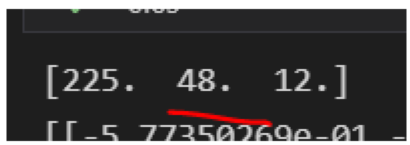
```
[[-5.77350269e-01  7.07106781e-01  4.08248290e-01]
 [-5.77350269e-01  5.96744876e-15 -8.16496581e-01]
 [-5.77350269e-01 -7.07106781e-01  4.08248290e-01]]
[15.          6.92820323  3.46410162]
[[-5.77350269e-01 -5.77350269e-01 -5.77350269e-01]
 [ 4.08248290e-01 -8.16496581e-01  4.08248290e-01]
 [ 7.07106781e-01 -1.16573418e-14 -7.07106781e-01]]
```

```
6.9282**2
```

```
47.999955240000006
```



```
[[-5.77350269e-01  7.07106781e-01  4.08248290e-01]
 [-5.77350269e-01  5.96744876e-15 -8.16496581e-01]
 [-5.77350269e-01 -7.07106781e-01  4.08248290e-01]]
[15.          6.92820323  3.46410162]
[[-5.77350269e-01 -5.77350269e-01 -5.77350269e-01]
 [ 4.08248290e-01 -8.16496581e-01  4.08248290e-01]
 [ 7.07106781e-01 -1.16573418e-14 -7.07106781e-01]]
```



```
[225.  48.  12.]
```

▼ 杨辉三角

yield:

- 列表所有数据都在内存中，如果有海量数据的话将会非常耗内存。
- 如：仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。
- 如果列表元素按照某种算法推算出来，那我们就可以在循环的过程中不断推算出后续的元素，这样就不必创建完整的list，从而节省大量的空间。
- 又想要得到庞大的数据，又想让它占用空间少，那就用生成器

```
def triangles():
    L = [1]
    while True:
        yield L
        L = [sum(i) for i in zip([0]+L, L+[0])]
```

```
a = triangles()
for i in range(7):
    ss = " ".join([str(d) for d in next(a)])
    print("{0:^25}".format(ss))
```

首先，如果你还没有对yield有个初步分认识，那么你先把yield看做“return”，这个是直观的，它首先是个return，普通的return是什么意思，就是在程序中返回某个值，返回之后程序就不再往下运行了。看做return之后再把它看做一个是生成器（generator）的一部分（带yield的函数才是真正的迭代器），好了，如果你对这些不明白的话，那先把yield看做return,然后直接看下面的程序，你就会明白yield的全部意思了：

```
def foo():
    print("starting...")
    while True:
        res = yield 4
        print("res:",res)
g = foo()
print(next(g))
print("*"*20)
print(next(g))
```

就这么简单的几行代码就让你明白什么是yield，代码的输出这个：

```
starting...
4
********************
res: None
4
```

我直接解释代码运行顺序，相当于代码单步调试：

1.程序开始执行以后，因为foo函数中有yield关键字，所以foo函数并不会真的执行，而是先得到一个生成器g(相当于一个对象)

2.直到调用next方法，foo函数正式开始执行，先执行foo函数中的print方法，然后进入while循环

3.程序遇到yield关键字，然后把yield想想成return,return了一个4之后，程序停止，并没有执行赋值给res操作，此时next(g)语句执行完成，所以输出的前两行（第一个是while上面的print的结果,第二个是return出的结果）是执行print(next(g))的结果，

4.程序执行print("*"*20)，输出20个*

5.又开始执行下面的print(next(g)),这个时候和上面那个差不多，不过不同的是，这个时候是从刚才那个next程序停止的地方开始执行的，也就是要执行res的赋值操作，这时候要注意，这个时候赋值操作的右边是没有值的（因为刚才那个是return出去了，并没有给赋值操作的左边传参数），所以这个时候res赋值是None,所以接着下面的输出就是res:None,

6.程序会继续在while里执行，又一次碰到yield,这个时候同样return 出4，然后程序停止，print函数输出的4就是这次return出的4.

到这里你可能就明白yield和return的关系和区别了，带yield的函数是一个生成器，而不是一个函数了，这个生成器有一个函数就是next函数，next就相当于“下一步”生成哪个数，这一次的next开始的地方是接着上一次的next停止的地方执行的，所以调用next的时候，生成器并不会从foo函数的开始执行，只是接着上一步停止的地方开始，然后遇到yield后，return出要生成的数，此步就结束。

join:

举个例子：

```
','.join('abc')
```

上面代码的含义是“将字符串abc中的每个成员以字符','分隔开再拼接成一个字符串”，输出结果为：

```
'a,b,c'
```

```
def triangles():
    L = [1]
    while True:
        yield L
        L = [sum(i) for i in zip([0]+L, L+[0])]

a = triangles()
for i in range(7):
    ss = " ".join([str(d) for d in next(a)])
    print("{0:^25}".format(ss))
#居中对齐^
```