```
matrix A : [[1]
 [2]
 [4]
 [5]
 [7]
 [8]]
dimensions of A : (6, 1)
matrix B : [[1 4 7]
 [2 5 8]]
dimensions of B : (2, 3)
```

Another useful expression is `np.c_` and `np.r_`, which allow us to stack column (and row) vectors with other vectors (and matrices).

```
In [ ]: np.c_[-np.identity(3), np.zeros(3)]
```

```
Out[ ]: array([[-1., -0., -0.,  0.],
               [-0., -1., -0.,  0.],
               [-0., -0., -1.,  0.]])
```

```
In [ ]: np.r_[np.identity(3), np.zeros((1,3))]
```

```
Out[ ]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.],
               [0., 0., 0.]])
```

6.2. Zero and identity matrices

Zero matrices. A zero matrix of size $m \times n$ is created using `np.zeros((m,n))`. Note the double brackets!

```
In [ ]: np.zeros((2,2))
```

```
Out[ ]: array([[0., 0.],
               [0., 0.]])
```

Identity matrices. Identity matrices in Python can be created in many ways, for example, by starting with a zero matrix and then setting the diagonal entries to one. You can also use the `np.identity(n)` function to create an identity matrix of dimension n.

```
In [ ]: np.identity(4)
```

```
Out[ ]: array([[1., 0., 0., 0.],
               [0., 1., 0., 0.],
               [0., 0., 1., 0.],
               [0., 0., 0., 1.]])
```

Ones matrix. In VMLS we do not use special notation for a matrix with all entries equal to one. In Python, such a matrix is given by `np.ones((m,n))`

Diagonal matrices. In standard mathematical notation, $\mathbf{diag}(1,2,3)$ is a diagonal 3×3 matrix with diagonal entries $1,2,3$. In Python, such a matrix can be created using `np.diag()`.

```
In [ ]: x = np.array([[0, 1, 2],
                      [3, 4, 5],
                      [6, 7, 8]])
        print(np.diag(x))
```

```
[0 4 8]
```

```
In [ ]: print(np.diag(np.diag(x)))
```

```
[[0 0 0]
 [0 4 0]
 [0 0 8]]
```

Here we can see that, when we apply the `np.diag()` function to a matrix, it extracts the diagonal elements as a vector (array). When we apply the `np.diag()` function to a vector (array), it constructs a diagonal matrix with diagonal entries given in the vector.

Random matrices. A random $m\times n$ matrix with entries distributed uniformly between 0 and 1 is created using `np.random.random((m,n))`. For entries that have a standard normal distribution, we can use `np.random.normal((m,n))`.

```
In [ ]: np.random.random((2,3))
```

```
Out[ ]: array([[0.13371842, 0.28868153, 0.6146294 ],
               [0.81106752, 0.10340807, 0.02324088]])
```

```
In [ ]: np.random.randn(3,2)
```

```
Out[ ]: array([[ 1.35975208, -1.63292901],
               [-0.47912321, -0.4485537 ],
               [-1.1693047 , -0.05600474]])
```

Sparse matrices. Functions for creating and manipulating sparse matrices are contained in the `scipy.sparse` module, which must be installed and imported. Sparse matrices are stored in a special format that exploits the property that most of the elements are zero. The `scipy.sparse.coo_matrix()` function create a sparse matrix from three arrays that specify the row indexes, column indexes, and values of the nonzero elements. The following code creates a sparse matrix

$$A = \begin{bmatrix} -1.11 & 0 & 1.17 & 0 & 0 \\ 0.15 & -0.10 & 0 & 0 & 0 \\ 0 & 0 & -0.3 & 0 & 0 \\ 0 & 0 & 0 & 0.13 & 0 \end{bmatrix}$$

```
In [ ]: from scipy import sparse
        I = np.array([ 0, 1, 1, 0, 2, 3 ]) # row indexes of nonzeros
        J = np.array([ 0, 0, 1, 2, 2, 3 ]) # column indexes
        V = np.array([ -1.11, 0.15, -0.10, 1.17, -0.30, 0.13 ]) # values
        A = sparse.coo_matrix((V,(I,J)), shape=(4,5))
        A
```

```
Out[ ]: <4x5 sparse matrix of type '<class 'numpy.float64'>'
          with 6 stored elements in COOrdinate format>
```

```
In [ ]: A.nnz
```

```
Out[ ]: 6
```

Sparse matrices can be converted to regular non-sparse matrices using the `todense()` method.

```
In [ ]: A.todense()
```

```
Out[ ]: matrix([[-1.11,  0.  ,  1.17,  0.  ,  0.  ],
                [ 0.15, -0.1 ,  0.  ,  0.  ,  0.  ],
                [ 0.  ,  0.  , -0.3 ,  0.  ,  0.  ],
                [ 0.  ,  0.  ,  0.  ,  0.13,  0.  ]])
```

A sparse $m \times n$ zero matrix is created with `sparse.coo_matrix((m, n))`. To create a sparse $n \times n$ identity matrix in Python, use `sparse.eye(n)`. We can also create a sparse diagonal matrix (with different offsets) using

```
In [ ]: diagonals = [[1, 2, 3, 4], [1, 2, 3], [1, 2]]
        B = sparse.diags(diagonals, offsets=[0,-1,2])
        B.todense()
```

```
Out[ ]: matrix([[1., 0., 1., 0.],
                [1., 2., 0., 2.],
                [0., 2., 3., 0.],
                [0., 0., 3., 4.]])
```

A useful function for creating a random sparse matrix is `sparse.rand()`. It generates a sparse matrix of a given shape and density with uniformly distributed values.

```
In [ ]: matrix = sparse.rand(3, 4, density=0.25, format='csr',
        ↪  random_state=42)
        matrix
```

```
Out[ ]: <3x4 sparse matrix of type '<class 'numpy.float64'>'
          with 3 stored elements in Compressed Sparse Row format>
```

6.3. Transpose, addition, and norm

Transpose. In VMLS we denote the transpose of an $m \times n$ matrix A as A^T. In Python, the transpose of `A` is given by `np.transpose(A)` or simply `A.T`.

```
In [ ]: H = np.array([[0,1,-2,1], [2,-1,3,0]])
        H.T
```

```
Out[ ]: array([[ 0,  2],
               [ 1, -1],
               [-2,  3],
               [ 1,  0]])
```

```
In [ ]: np.transpose(H)
```

```
Out[ ]: array([[ 0,  2],
               [ 1, -1],
               [-2,  3],
               [ 1,  0]])
```