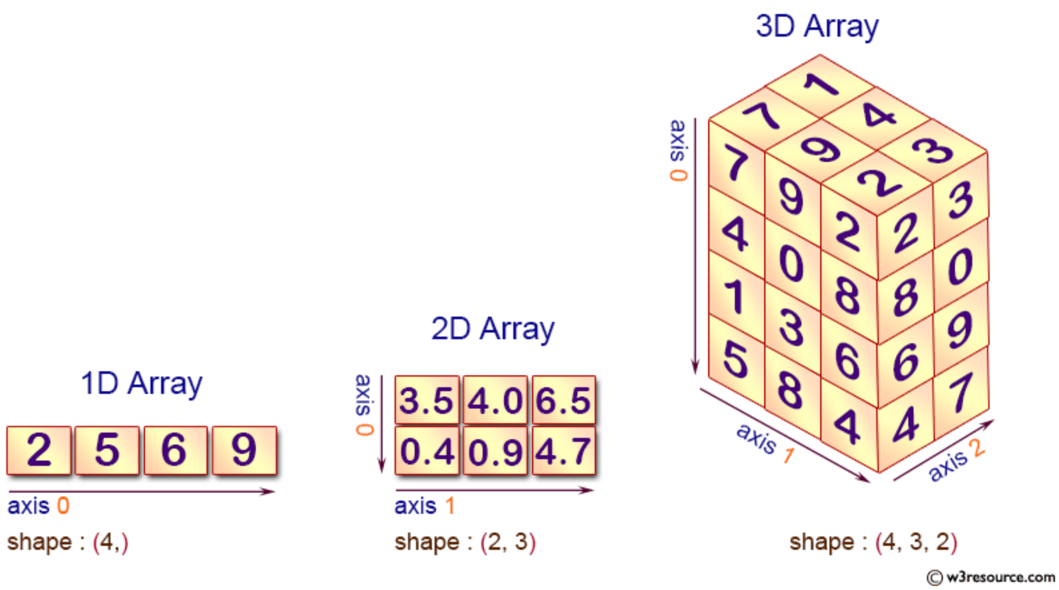
Matrices from the numpy Library
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/matrices/matrices.pdf



The numpy Library

- numpy() *defines a matrix as an array of arrays;*
- *Matrices represent linear transformations of vectors;*
- *Initialize a matrix with data, or with zeros, ones, or random values;*
- *Access an entry by double index, like* A[i,j]*;*
- *Multiplication A*x=b using* np.dot()*;*
- *Solve linear system by x=np.linalg.solve(A,b);*
- *Factoriations:* L,U=np.lu(A)*, or QR, or SVD;*
- *Matrix eigenvalues:* L = np.linalg.eigvals(A)*;*

1 A numpy matrix is an array of arrays

We know that to numpy(), an m-vector is simply a list of numeric values, with an index $0 \leq i < m$. Let's write it out to look like a column vector:

```
v = np.array ( [
  0,
  1,
  2,
```

```
  ...
  m-1 ] )
```

To create an $m \times n$ matrix, we can simply specify that the i-th entry of the array is itself an array of values, that is, the values of row i, something like this:

```
A = np.array ( [
  [row 0],
  [row 1],
  [row 2],
  ...
  [row m-1]
  ] )
```

where each row will be a vector of `n` values.

If we use a single index to refer to the array, then `A[i]` represents the entire `i`-th row of values, whereas, `A[i,j]` is the j-th item of the i-th row. Note that rows have a special status here. In order to reference all the entries of the `j`-th column, we have to use two indices: `A[:,j]`.

For a matrix formed as a `numpy()` array, the rows must all have the same number of elements, and the elements must be numeric.

2 Making matrices

Now it's time to move to two dimensions, and see how `numpy` arrays can be used to create, modify and analyze matrices.

An $m \times n$ mathematical matrix can be represented by a `numpy` array of dimensions (m, n). We can create matrices by commands like:

```
E = np.empty ( [ 3, 2 ] )
I = np.identity ( 3 )
O = np.ones ( [ 3, 2 ] )
R = np.random.rand ( 3, 2 )    # Does not bracket the dimensions!
Z = np.zeros ( [ 3, 2 ] )
```

For small matrices we can enter the values in a list of lists. Suppose our mathematical matrix is:

$$A = \begin{bmatrix} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \\ 40 & 41 & 42 & 43 \end{bmatrix}$$

Then we can enter the Python commands:

```
A = np.array ( [ \
  [  0,  1,  2,  3 ], \
  [ 10, 11, 12, 13 ], \
  [ 20, 21, 22, 23 ], \
  [ 30, 31, 32, 32 ], \
  [ 40, 41, 42, 43 ] ] )
```

Some new `numpy` array attributes are available as well:

- `A.ndim` tells us that A is a 2-dimensional array;

- `A.shape` statement returns (5,4);
- `A.shape[0]` returns 5;
- `A.size` returns 20 (total number of entries);

To index the item in row `i`, column `j`, we write `numpy` arrays use the more familiar `A[i,j]`.

We have already seen some examples of how Python indexing works. For our sample matrix `A`,

```
A[1,2] = 12
A[0,:] = [ 0, 1, 2, 3]            # Row 0
A[:,1] = [ 1, 11, 21, 31, 41 ]    # Column 1
A[2:4,1] = [ 21, 31 ]             # Rows 2 and 3 of column 1
```

3 Operators: transpose(), dot(), matmul()

For a matrix, we have the `np.transpose()` operator:

```
B = np.transpose(A)
[  [ 0, 10, 20, 30, 40 ],
   [ 1, 11, 21, 31, 41 ],
   [ 2, 12, 32, 32, 42 ],
   [ 3, 13, 23, 33, 43 ] ]
```

which can also be written as

```
B = A.T
```

Given two vectors `u` and `v` of the same length, we can compute their dot product

```
  udotv = np.dot ( u, v )
```

`A` is an $m \times n$ matrix and `x` is a vector of length n, we can use the `np.dot()` operator to carry out matrix-vector multiplication

```
  b = np.dot ( A, x )
```

```
  x = [ 1, 2, 3, 4 ]
  b = np.dot ( A, x )
  [ 20, 120, 220, 320, 420 ]
```

If `A` is an $m \times n$ matrix and `B` is an $n \times k$ matrix, we can compute the matrix-vector product using `matmul()`:

```
  B = A.T                     # B is now an nxm matrix
  C = np.matmul ( A, B )      # C is an mxm matrix
```

4 Plotting Temperature Data

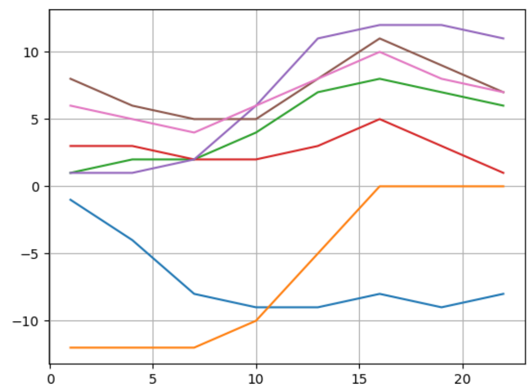
Some `numpy` nfunctions can be applied to a matrix in a variety of ways. To start with, consider the `np.max()` function. Let's take as our data an array T that actually measures the temperature every 3 hours, over a week.

```
T = np.array ( [ \
  [ -1, -4, -8, -9, -9, -8, -9, -8 ], \
  [-12,-12,-12,-10, -5,  0,  0,  0 ], \
  [  1,  2,  2,  4,  7,  8,  7,  6 ], \
```

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

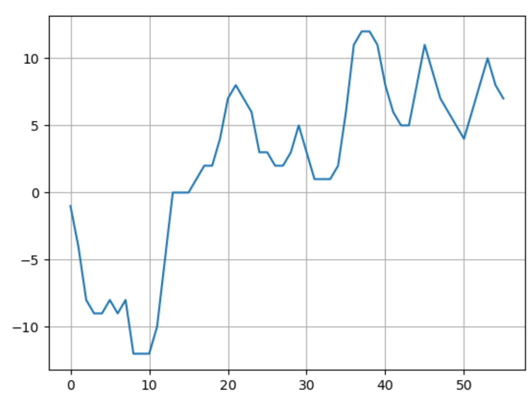
We could look at this data day by day, and plot it that way:

```
  h = np.linspace ( 1, 22, 8 )   # 24 hour time
  plt.clf ( )
  for day in range ( 0, 7 ):
    plt.plot ( h, T[day,:] )
  plt.grid ( True )
  plt.show ( )
  plt.close ( )
```



If we want a single plot over the whole week, we need to “flatten” the matrix, that is, to make a vector by stringing the rows together:

```
Tweek = T.flatten ( )
plt.plot ( Tweek )
plt.show ( )
```



5 Analyzing Temperature Data

Now that we have our temperature data, we might want to ask for the minimum, average, and maximums

- for each day
- for each measured hour;
- over the whole week.

```
min_day = np.min ( T, axis = 0 )
min_hour = np.min ( T, axis = 1 )
min_week = np.min ( T )
```

and our results are:

```
min(T) daily  =  [-12 -12 -12 -10  -9  -8  -9  -8]
min(T) hourly =  [ -9 -12   1   1   1   5   4]
min(T) weekly =  -12
```

You should see that `axis=0` computes the minimum value for each row, while `axis=1` does the same for columns, and with no axis specified, the minimum is over the whole set of data.

You can get similar results using `np.max()`, `np.mean()`, and `np.sum()`.

6 Making X and Y Spatial Matrices for Plotting

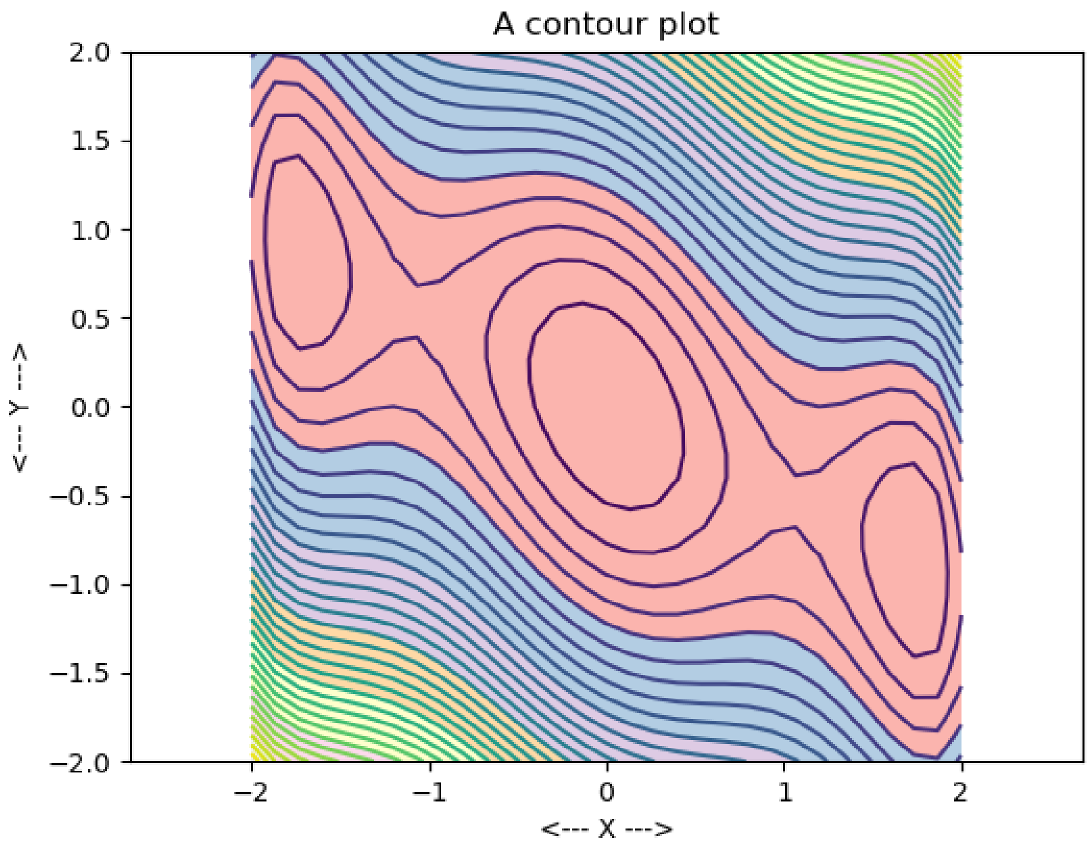
A standard way of sampling a function $z = f(x, y)$ is to define a grid of m equally spaced points over the x range, and n equally spaced points over the y range, evaluate the function $z_{i,j} = f(x_i, y_j)$ and somehow create a visual display of this information.

The `numpy` library allows us to write such a process in an efficient way. Here, we would like to sample the function $f(x, y) = 2x^2 + 1.05x^4 + x^6/6 = xy + y^2$ over the square $-2 \leq x, y \leq +2$ and then make a contour plot.

```
xvec = np.linspace ( -2.0, 2.0, 31 )
yvec = np.linspace ( -2.0, 2.0, 31 )

X, Y = np.meshgrid ( xvec, yvec )
Z = 2 * X**2 - 1.05 * X**4 + X**6 / 6 + X * Y + Y**2

plt.clf ( )
plt.contourf ( X, Y, Z )    # filled regions
plt.contour ( X, Y, Z, levels = 35 )  # contour lines
plt.show ( )
```


A contour plot
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2
−1
0
1
2
<--- X --->
<--- Y --->