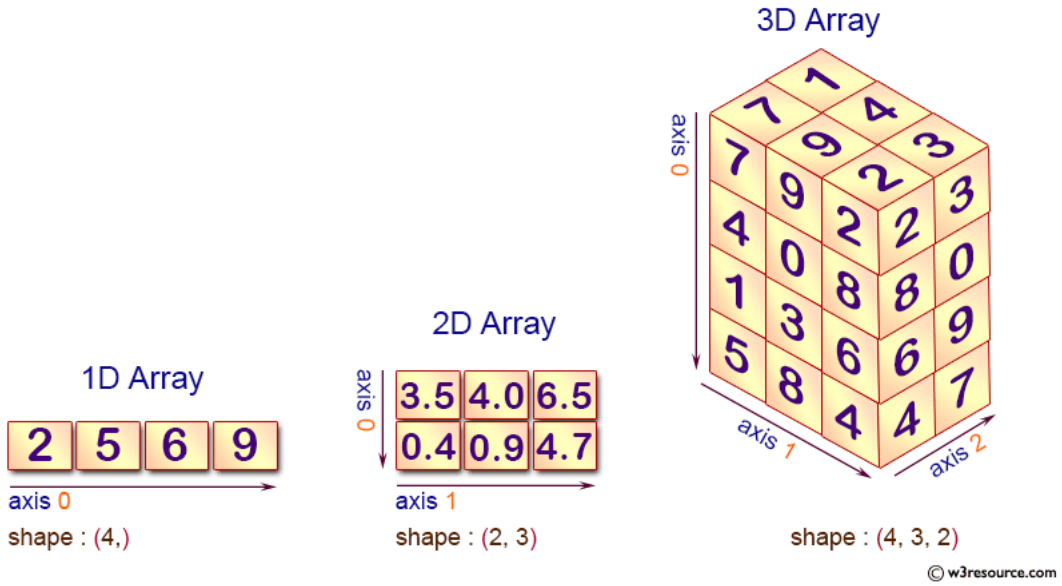
Matrices from the numpy Library
Mathematical Programming with Python

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

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/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);*
- *Factorizations:* `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 are familiar now with how to think about a mathematical vector as a `numpy()` list of numeric values, with an index $0 \leq i < m$. To extend such arrays to handle matrices, `numpy` defines an $m \times n$ matrix as an array, each of whose entries is an entire row of the matrix.

Here is one way to define such a matrix, and if you look closely at the square brackets, you can see that there is one big vector of $m = 5$ rows, with each row being itself a vector of length $n = 4$.

```
A = np.array ( [
  [  0,  1,  2,  3 ],  <-- row 0
  [ 10, 11, 12, 13 ],  <-- row 1
  [ 20, 21, 22, 23 ],  <-- row 2
  [ 30, 31, 32, 33 ],  <-- row 3
  [ 40, 41, 42, 43 ],  <-- row 4
  ] )
```

To access a particular entry, you use the notation `A[i,j]`, where i is the row and j the column. Thus, in the above example, `A[2,1]=21`. This differs from how we accessed entries in a list of lists, where a typical index would be `A[2][1]`.

Because the matrix is organized by rows, you can actually also use a single index to refer to all of a row. So `A[1]` represents the values `[10,11,12,13]`, although we could also use the standard notation `A[1,:]` as well. For columns, we can only use the standard two index notation, as in `A[:,0]`.

For a matrix formed as a `numpy()` array, the rows must all have the same number of elements, and the elements must share a common datatype, either logical or numeric.

2 Making matrices

We have already seen how, for small matrices, we can enter the values using the `np.array()` command, with the values given in a list of lists.

We can also create many special matrices by commands like:

```
I = np.identity ( 3 )       <-- returns the 3x3 identity matrix.
I = np.eye ( 3 )            <-- also returns the 3x3 identity matrix.
O = np.ones ( [ 3, 2 ] )
R = np.random.random ( [ 3, 2 ] )
Z = np.zeros ( [ 3, 2 ] )
```

The same new `numpy` attributes we saw for vectors are also available for matricesl:

- `A.ndim` tells us that A is a 2-dimensional array;
- `A.shape` statement returns (5,4);
- `A.shape[0]` returns 5, `A.shape[1]` returns 4;
- `A.size` returns 20 (total number of entries);
- `A.dtype` returns int or int64;

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
A[2:4,1:3] = [ [21,22], [31,32] ] #  A submatrix
```

3 Arithmetic, Operators and Functions

A matrix stored as a numpy array has a data type, which is usually logical integer, float, or complex. The user can specify the datatype of a matrix by adding the `dtype = value` argument when creating an array. Otherwise, Python will assume the lowest datatype consistent with the input. In some cases, a matrix initially of one data type may be "promoted" to a nigher data type in order for an operation to be carried out smoothly.

As we say with vectors, a matrix represented by a numpy array can be involved with the operators +, -, *, /, **, and %. These operators can in general pair a matrix and a scalar, as in $A**2$ or $7.5 * B$. Two matrices can also be comgined with these operators; in that case, the matrices must have the same number of rows and columns, and the operations will be carried out elementwise.

Similarly, most numpy functions like `np.cos()` or `np.sqrt()` can be applied to a matrix, but the functions will be applied element by element, resulting in a new matrix of the same shape.

However, there are some exceptions, sometimes called "reduction operators", in which the result of the operation is a single number, or a vector. These functions have an optional input keyword `axis=?`. If it is omitted, or specified as None, then a single scalar result is returned. If it is 0, then a vector is returned with a result for each column; an axis value of 1 returns results for each row.

Consider the following table which suggests how one `numpy()` function can be used in three different ways:

Axis Result size	None 1	axis = 0 [1 per column]	axis = 1 [1 per row]
np.argmax()	array argmax	column argmax	row argmax
np.argmin()	array argmax	column argmax	row argmax
np.max()	array max	column max	row max
np.mean()	array mean	column mean	row mean
np.min()	array min	column min	row min
np.prod()	array prod	column prod	row prod
np.std()	array std	column std	row std
np.sum()	array sum	column sum	row sum
np.var()	array var	column var	row var

Here are four ways to call `np.sum(A)` for our example matrix:

```
np.sum(A)
>>> 430

np.sum(A, axis=None)
>>> 430

np.sum(A, axis=0)
array([100, 105, 110, 115])

np.sum(A, axis=1)
array([  6,  46,  86, 126, 166])
```

We will need to understand how to call `np.sum()` when we look at the magic matrix.

4 Rearranging a matrix

There are many `numpy()` functions for rearranging what's inside a matrix, to extract a chunk of a matrix, or to stack two matrices together to make a bigger one. They are often a little tricky to use, so you should always check the documentation and be willing to make a few mistakes when starting to use them.

```
B = np.reshape ( A,  ( 2, 10 ) )
```

makes a new version of the entries of A as a 2×10 array.

```
C = A.flatten ( )  # Not "np.flatten(A)"
```

makes a new version of the entries of A as a "flattened" array (vector) of 20 elements.

```
D = np.flipud ( A )
```

"flips" up and down, making an upside down version of A. There is also a `np.fliplr()` function which flips left and right.

```
E = np.rot90 ( A )
```

makes a 4×5 copy of A in which the matrix has been rotated counterclockwise 90 degrees.

```
F = np.roll ( A, shift, axis = None/0/1 )
```

moves each element by `shift` positions, along all axes, or the column axis, or the row axis.

```
G = np.transpose ( A )
H = A.T
```

Two different commands which both return the transpose of the matrix, that is, flipping it around its main diagonal.

```
G = np.stack ( ( row1, row2 ) )
```

creates a new matrix whose rows are row1, row2, (and row3, etc). The vectors must have the same shape. This command can also manipulate more complicated objects, and has related vertical and horizontal versions `vstack()` and `hstack()`.

5 Magic squares

A magic square of order n is an $n \times n$ array of numbers such that all rows, all columns, and both diagonals have the same sum. It is not obvious at first how to construct a magic square, and hence they are often considered to be lucky charms with magic properties.

When programmers need a "random" matrix to illustrate some procedure, they frequently use a magic matrix, since the entries are distinct integers, and the matrix has some interesting properties. MATLAB, for example, has a built-in `magic(n)` function to create a magic matrix of any order (except 2!).

Here are magic matrices of order $n = 3, 5, 7$.

The 3×3 example:

$$A_3 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

The 5×5 example:

$$A_5 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The 7×7 example:

$$A_7 = \begin{bmatrix} 30 & 39 & 48 & 1 & 10 & 19 & 28 \\ 38 & 47 & 7 & 9 & 18 & 27 & 29 \\ 46 & 6 & 8 & 17 & 26 & 35 & 37 \\ 5 & 14 & 16 & 25 & 34 & 36 & 45 \\ 13 & 15 & 24 & 33 & 42 & 44 & 4 \\ 21 & 23 & 32 & 41 & 43 & 3 & 12 \\ 22 & 31 & 40 & 49 & 2 & 11 & 20 \end{bmatrix}$$

Can we use Python to create a new magic square, by following the steps of the magic square algorithm?

6 Magic matrix algorithm

As long as the matrix order n is odd, the following algorithm can be used to fill an $n \times n$ grid with the integers 1 through n^2, with constant row, column and diagonal sums.

1. Start in the middle of the top row, and let $k = 1$.
2. Write k in the current grid position;
3. If $k = n^2$, the grid is complete, so stop;
4. Else, set $k = k + 1$;
5. Plan to move diagonally up and right, "mod n", that is, wrap to first column or last row if necessary.
6. If this cell is already filled, move vertically down one space instead;
7. Return to step 2.

7 Magic matrix in Python

```
def magic ( n ):

  A = np.zeros ( [ n, n ] )

  k = 1
  i = 0
  j = n // 2

  while ( k <= n**2 ):

    A[i,j] = k

    k = k + 1
#
#  Go UP one row, and RIGHT one column, mod n
#
    new_i = ( i - 1 ) % n
    new_j = ( j + 1 ) % n

    if ( A[new_i,new_j] ):     # True if A[new_i,new_j] is already set
      i = i + 1
    else:
      i = new_i
      j = new_j

  return A
```

Listing 1: magic_matrix.py

What does "`if ( A[new_i,new_j] ):`" mean? Remember that, to Python, 0 plays the role of `False` and 1, or in fact, any nonzero value, is understood as `True`. So all we are saying here is `if ( A[new_i,new_j] has been set):`.

8 Verify the Magic

Let's try to verify that our matrix is magic. Instead of using many individual `sum()` commands, or multiplying by a vector of 1's, we can use the fact that the `sum()` command can return the sum of rows or columns of a matrix if we specify the "axis":

```
  np.sum ( A )            # sums all the entries
  np.sum ( A, axis = 0 )  # sums the columns
  np.sum ( A, axis = 1 )  # sums the rows
```

It turns out that we can compute the diagonal sum because there is a linear algebra function called `trace()` that sums the diagonal elements of a matrix, and `numpy()` implements it:

```
np.trace ( A )
```

How do we get the antidiagonal sum? There are a number of `numpy()` functions for flipping a matrix. In particular, `flipud()` flips up and down, with the result that what was the antidiagonal of the matrix is now the diagonal.

```
np.trace ( np.flipud ( A ) )
```

In any case, if we have done our work correctly, all these operations should return the same magic value.

9 meshgrid(): Making a 2D grid

We have already seen how the `np.linspace()` function creates an evenly spaced grid of points in one dimension, which is a big help in plotting, and later in approximation and partial differential equations. Often we need the same kind of regular grid in two dimensions, usually as the product of simple separate grids in x and y.
For example, suppose we have 1D grids:

```
x = np.linspace ( 4, 8, 3 ) = [ 4, 6, 8 ]
y = np.linspace ( 10, 20, 2 ) = [ 10, 20 ]
```

To describe a 2D grid from these components, we call

```
X, Y = np.meshgrid ( x, y )
```

resulting in:

```
X = [ [ 4, 6, 8 ],
      [ 4, 6, 8 ] ]

Y = [ [ 10, 10, 10 ],
      [ 20, 20, 20 ] ]
```

so that the (i,j) point in our grid has coordinates `( X[i,j], Y[i,j] )`. We will see how useful this idea is in a simple graphics example:

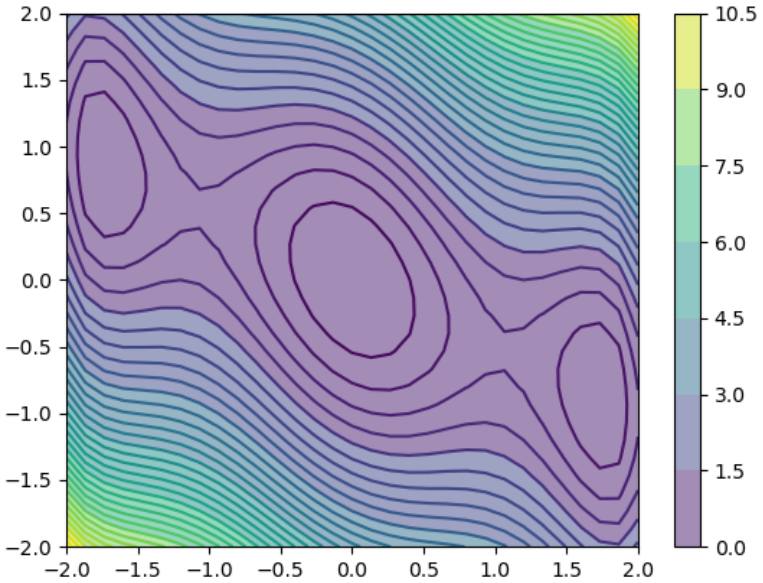
A standard way of sampling a function $z = f(x, y)$ is to define a grid of m equally spaced points over the x range, and n equally spaced points over the y range, evaluate the function $z_{i,j} = f(x_i, y_j)$ and somehow create a visual display of this information.

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.

```
def meshgrid_example ( ):

  import matplotlib.pyplot as plt
  import numpy as np

  x = np.linspace ( -2.0, 2.0, 31 )
  y = np.linspace ( -2.0, 2.0, 31 )

  X, Y = np.meshgrid ( x, y )

  Z = 2 * X**2 - 1.05 * X**4 + X**6 / 6 + X * Y + Y**2

  plt.contourf ( X, Y, Z, alpha = 0.5 ) # contours by color
  plt.colorbar ( )
```

```
  plt.contour ( X, Y, Z, levels = 35 )  # contour by lines
  plt.show ( )
```

Listing 2: meshgrid_example.py



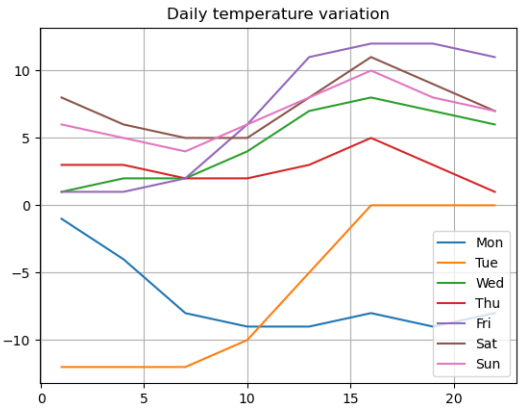
10 Plotting Temperature Data

Suppose we have a set of temperature measurements, taken every 3 hours, over a week. The data is arranged in a matrix, with each row representing a particular day, and each column a particular time in the range 1am, 4am, ..., 10pm.

```
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:

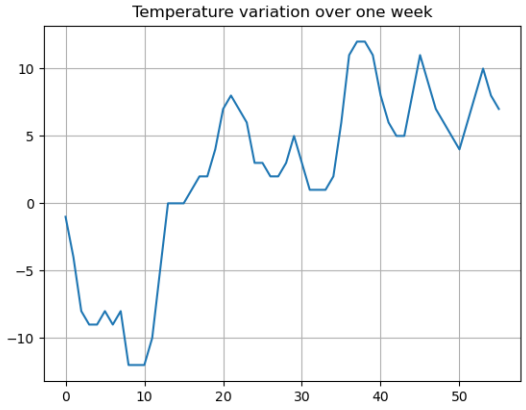
```
  h = np.linspace ( 1, 22, 8 )  # 24 hour time
  for day in range ( 0, 7 ):
    plt.plot ( h, T[day,:] )
  plt.grid ( True )
  plt.show ( )
```



Daily temperature variation for 7 days.

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 one after another.

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



Temperature variation over a week.

11 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 time;
- over the whole week.

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

and our results are:

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

You should see that `axis=0` computes the minimum value for each column (time), while `axis=1` does the same for rows (days), 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()`.