This is a version of HW2 that aims to make the problems more readable.

> Please DO NOT use this file as the basis for your homework submission.

> Add your code to `hw2_template.py` to create the file that you submit.

1) This problem involves writing and using an alternate implementation of elemetary row operations.

The implementation in the Ch. 2 notebook uses a `for` loop to individually update each entry in the row being altered. Here you will write an alternate version supported by `numpy` that provides a way to refer to a portion of an array including multiple elements including an entire (or partial) row or column.

The related keywords in `numpy include "views", "slice", and "copy". Please read the materials at the following link that provides a nice description of the details:
https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html

1a) Write a new version of `row_op` that operates row-wise instead of element-wise.

1b) Previously we skipped one of the elementary row operations; i.e. swapping rows. Now is the time to fill in that gap. Write code for a row-wise implementation of `swap_rows(A, i0, i1)` that exchanges rows `i0` and `i1` in the array `A`.

2) This problem involves matrix iteration methods for computing eigenvalues, eigenvectors, and condition numbers.

2a) Implement `dominant_eigen_iteration` that executes the matrix iteration scheme for computing the "dominant" eigenvalue/eigenvector (associated with the eigenvalue with largest magnitude) of a square matrix. Here is one verion of a simple pseudo-code (that needs termination conditions)

$$
\begin{aligned}
&\text{Choose an initial vector } u0 \text{ such that } \|u0\| = 1 \\
&\text{for } k = 1, 2, \ldots \text{ do} \\
&\quad v^{(k)} = Au^{(k-1)} \\
&\quad u^{(k)} = v^{(k)} / \|v^{(k)}\| \\
&\text{end}
\end{aligned}
$$

2b) Implement `recessive_eigen_iteration` that the "inverse matrix iteration" scheme for computing the "recessive" eigenvalue/eigenvector pair (associated with the eigenvalue with smallest magnitude) of a square matrix. Here is a simple pseudo-code (again missing termination conditions)

$$
\begin{aligned}
&\text{Choose an initial vector } u0 \quad \text{ such that } \|u0\| = 1 \\
&\text{for } k = 1, 2, \ldots \text{ do} \\
&\quad \text{Solve } Av^{(k)} = u^{(k-1)} \\
&\quad u^{(k)} = v^{(k)} / \|v^{(k)}\| \\
&\text{end}
\end{aligned}
$$

Use `numply.linalg.solve` in your implementation.

2c) Use the functions you implemented in parts a and b to create a `condition` function that computes an estimate of the condition number of a square matrix.

3) This problem involves functions to support QR factorization.

3a) Implement a function `component_of_along` to compute the component of vector v along the direction of vector u.

3b) In class, we discussed QR factorization based on Gram Scmidt orthogonalization. In that approach, an orthogonal basis is constructed by subtracting from each new candidate basis vector the components along the direction of each already-computed entry in the (numerically) orthogonal set. That approach can run into precision issues (because candidate basis vectors that lie close to the space spanned by the existing basis vectors can lead to catastrophic cancellation when components are subtracted). That issue led to the creation of other approaches. The one that is the focus of this problem involves Householder reflections. The basic idea is to compute the vector that arises when an input vector v is reflected about the plane normal to a specified vector u.

Fill in code to implement the function `reflect` to compute the Householder reflection. This should be pretty straightforward if you use the function `component_of_along` that you implemented in 3a.

3c) Householder made good use of this basic reflection operation. First, he noted that the reflection operation corresponds to multiplication of the input vector `v` by an orthogonal matrix `Q_u`. Think about why that is true! Then, he figured out how to pick the mirror normal that would produce a reflected vector e0 that lies along the first coordinate axis; i.e. `Q_u(v) = norm(v)*e0`. Convince yourself that the choice `u = c*(v - norm(v)*e0)` works for any choice of the constant `c`. In particular, choose `c=1` so `u = v - norm(v)*e0`

> NOTE: The slightly more complicated choice u = v - sign(v[0])*norm(v)*e0 prevents the possibility of catastrophic cancellation errors.

Insert code to define the `reflect_to_e0` function that produces the orthogonal matrix that rotates the input vector to the e0 direction.

3d) Use the function you implemented for part 3c to implement the `Householder` function that does the first step of QR factorization by returning the factors Q and R_0 where $QR_0 = A$, Q is orthogonal, and the first column of R_0 has all zeros below the upper-left entry.

4) OPTIONAL:

Use the functions you wrote for problem 3 to implement `QR_Householder` that computes a full QR factorization based on Householder reflections.