Before you begin

- Make a new project for this lab
- Make `np_exercises.py`
- Browse through the Numpy quickstart page and familiarize yourself with some of the stuff you can do with Numpy. Pay particular attention to how it avoids using for loops in places you might have used them before.

Learning Objectives:

You should be able to do the following:

- Use vectorized numpy features to avoid using loops
- Compare numeric arrays
- Determine minimum/maximum values and the positions using numpy
- Draw large samples of random numbers in one line of code
- Filter and sort arrays using numpy

Thoughts (come back and read these after you've read the problems):

- Numpy is full of features which make your life convenient. This assignment asks you to take full advantage of them, which means that you will need to figure out how to actually do that. For instance, previously when you sampled random numbers, you would import a function from random, make an empty list, and keep appending to that list after calling your random number generator. You should think about what "the Numpy way" is to do that.
- Note that with Numpy arrays, if you try to do `array_a == array_b`, this does not return a value of `True` or `False`, but rather an array of `True` and `False` values. This means that the following lines of code will get you into trouble:

```
array_a = np.array([1,2])
if array_a == array_a:
    print('Hooray')
```

`ValueError`: The truth value of an array with more than one element is ambiguous. Use `a.any()` or `a.all()`.

If you see this error, it's because Numpy arrays cannot be treated as Booleans like regular Python lists. As the error says, you should use the `any()` or `all()` methods, which return a single Boolean value corresponding to whether any/all of the values in the array are `True`.

Grading Checkpoints [16 points]

For this assignment, you should not use loops! You should be using vectorized operations. Functional iterators are also not allowed. The following operations will be deducted: `for`, `while`, comprehensions (e.g. `[x for x in range(10)]`, `map()`, and `filter()`.

Please see associated footnotes for each item * .

- Assignment Grading Checkpoints (12 points total)
  - [1 points] `numpy_close()` works for base case *
  - [1 points] `numpy_close()` works for mismatched dimensions case *
  - [3 points] `simple_minimizer()` produces the correct values. *
  - [1 points] `simulate_dice_rolls()` exhibits the expected behavior *
  - [1 points] produces a reasonable histogram for `simulate_dice_rolls(5, 2000)`

*Indicates that the autograder will tell you if this problem is correct on our set of test cases and assign you credit.

  - [2 points] `is_transformation_matrix` returns the correct values *
  - [2 points] `nearest_neighbors()` returns the correct set of neighbors *
  - [1 points] `nearest_neighbors()` sorts neighbors in ascending order of distance *
  - [1 points] Code does not use any loops or functional iterators. **1 point off for each loop detected**, no lower bound *

- Standard Grading Checkpoints (4 points total)
  - [2 points] Code passes PEP8 checks with 10 errors max. *
  - [2 points] Code passes TA-reviewed style checks for cleanliness, layout, readability, etc.

Problem 1 Numpy Equals

For this assignment, create a new file called `np_exercises.py`. Put all functions in there.

- Write a function called `numpy_close` which takes in two Numpy arrays `a` and `b`, as well as a third optional argument, `tol`, which defaults to `1e-8`.
- It should return `True` if the arrays have the same shape and the absolute difference of each corresponding pair of elements less than `tol`.
- Your function should be able to work on arrays of arbitrary dimensions (they could be 1-D, 2-D, 8-D, etc.)

Problem 2 Minimization

Write a function `simple_minimizer` which takes in four arguments:

- A function reference `func`, which represents a 1-D function, e.g. `f(x) = x^2`. You can assume that it also will work on a Numpy array, so that if you call `func(x_values)`, you will get a corresponding array of y values.
- Two floats, `start` and `end`, which represent the region to search for a minimum. (Throw a `ValueError` if `start > end`.)
- An optional param, `num`, which defaults to 100.
- It should then evaluate `func` at `num` evenly-spaced points between `start` and `end` (endpoint `inclusive`) and **return two values**:
  - The `x` corresponding to the **minimum** value of `f(x)`
  - The minimum `f(x)`
- For example, if out of all the values you tested the minimum was `f(0.5) = 2.4`, you would return `(0.5, 2.4)`.

Example:

```
my_func = lambda x: x**2
simple_minimizer(my_func, -1.75, 2.25, num=5)
# Should return (0.25, 0.0625)
```

Problem 3 Dice roll simulation

- Write a function, `simulate_dice_rolls`, which takes in two params, `num_rolls` and `iterations`.
- It should return a 1-D Numpy array of length `iterations`, where each item is the sum of throwing a fair 6-sided die `num_rolls` times.
- It should also automatically output a histogram of the rolls to the file `dice_{0}_rolls_{1}.png`, where `{0}` is `num_rolls` and `{1}` is `iterations`. (No need for any labels, title, legends, etc.)
- Run your function for (`num_rolls = 5`, `iterations = 2000`) and include the plot with your submission

Problem 4: Transformation matrix

Recall that a transformation matrix `T` is a 4×4 matrix which can be used to represent the position and orientation of a coordinate frame relative to a base frame. A transformation matrix has the following form:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_1 \\ r_{21} & r_{22} & r_{23} & p_2 \\ r_{31} & r_{32} & r_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

p is a 3×1 matrix in $\mathbb{R}^3$, and R is a 3×3 **`rotation`** matrix. All rotation matrices have the property that the **`transpose`** of R is equal to its **`inverse`**, i.e. $R^T R = I$, where I is the 3×3 identity matrix.

Write a function **`is_transformation_matrix`** that takes in a 4×4 Numpy array and returns `True` if it's a valid transformation matrix, and `False` otherwise.

Example:

```
tf_valid = np.array([[0, 0, -1, 4], [0, 1, 0, 2.4], [1, 0, 0, 3], [0, 0, 0, 1]])
tf_invalid = np.array([[1, 2, 3, 1], [0, 1, -3, 4], [0, 1, 1, 1], [-0.5, 4, 0, 2]])
print(is_transformation_matrix(tf_valid)) # True
print(is_transformation_matrix(tf_invalid)) # False
```

Problem 5: Nearest neighbors

Write a function **`nearest_neighbors`** which takes in three arguments: a $N \times D$ Numpy array, a 1-dimensional array of length D **`point`**, and a distance threshold **`dist`**. It should return an $M \times D$ Numpy ($M <= N$) array of all points in the array which are within Euclidean distance **`dist`** of **`point`**. The output points should be sorted by distance from the corresponding **`point`**, with closest elements coming first. (You will receive partial credit if the correct points are output but not sorted.)

Example:

```
array = np.array([[1, 1, 1], [2, 3, 5], [0, 1, 1], [1.5, 1, 1], [10, 9, 9]])
target_pt = np.array([0, 1, 1])
nearest_neighbors(array, target_pt, 3.0)
# Should return [[0, 1, 1], [1, 1, 1], [1.5, 1, 1]]
```

Hint: You may be interested in the **`argsort`** function, which produces a Numpy array of indexes which will sort a given array.