# Python Tips for MEAM 620

Below you'll find some tips for writing Python code that is easy to read and debug. There may be something to learn here both for those new to Python and those who have used Python in the past but are missing out on modern features. These examples are pulled from real MEAM 620 project code.

## Avoid looping over items in an array.

Python is a high level language. If your data is a vector, you probably don't need to write loops over it's individual elements.

**Example: Replace all negative values in an array with zero.**

**Input:**

```
x = np.array([1, 4, -3, 7])
```

**Avoid using loops like this:**

This works, but it disguises what you're trying to accomplish and creates more places bugs may hide.

```
for (index, value) in enumerate(x):
    if value < 0:
        x[index] = 0
```

**Option #1 using numpy functions:**

Idea: "I want x to be the maximum of each value in x and 0."

```
x = np.maximum(x, 0)
```

**Option #2 using 'logical indexing.'**

Idea: "Everywhere x is less than zero, replace it with zero."

```
x[x < 0] = 0
```

**Example: Test if two points are not the same.**

**Input:**

```
start = np.array([3, 7, 9])
goal  = np.array([3, 7, 9])
```

**Avoid manually comparing elements.**

```
if start[0] != goal[0] or start[1] != goal[1] or start[2] != goal[2]:
   # Points are not the same.
```

**Option #1:** Idea: Two points are the same if the distance between them is zero.

```
distance = np.linalg.norm(goal-start)
if distance > 0:
    # Point are not the same.
```

**Option #2** Idea: Google the phrase “numpy check if two arrays are equal.”

```
if not np.array_equal(start, goal):
    # Points are not the same.
```

## Use Modern Language Features

### Multiply matrices using the ‘@’ symbol.

There is typically no need to explicitly use the the np.matmul function.

```
R = Rotation.from_euler('z', np.pi/4).as_matrix()
x = np.array([1,0,0])
y = R @ x
```

Note that typically you want 3D vectors to have shape=(3,). There is only one possible way to interpret the shape of x in y=Rx or y=xR. There is usually no need to add an extra dimension to x to “force” it to look like a row or column vector.

### Print values using “f-strings”

```
x = np.array([1,2,3])
y = np.sum(x)
print(f"The sum of {x} is {y}.")
```

```
Output: The sum of [1 2 3] is 6.
```

Notice that the variable names ‘x’ and ‘y’ are automatically recognized and printed in a pretty format.

### Use assertions to check your assumptions.

You can check assumptions inside your code with an ‘assertion.’ In this example, my program assumes that y is a real number. I’ve noticed that sometimes, randomly, y becomes ‘nan’ and I want to figure out why.

```
while True:
    # Sample Gaussian distribution with mean 1 and stdev 1.
    x = np.random.normal(loc=1.0, scale=1.0)
    y = np.sqrt(x)
    assert not np.isnan(y), f'Square root of {x} is {y}'
```

When the assertion fails, the program will stop with an error and output a traceback showing exactly where the error happened:

```
AssertionError                            Traceback (most recent call last)
<ipython-input-12-e2ff553d6dde> in <module>
      3     x = np.random.normal(loc=1.0, scale=1.0)
      4     y = np.sqrt(x)
----> 5     assert not np.isnan(y), f'Square root of {x} is {y}.'
      6

AssertionError: Square root of -0.23754037331081346 is nan.
```

Assertions are like a safety test and a code comment all wrapped into one. They tell the reader what you expect to happen, and they give you debugging information if something ever goes wrong.

Here's an example of reading a traceback.

```
======================================================================
FAIL: test_unit_traj_end_loc (proj1_1.util.test.TestBase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/autograder/source/meam620-2020/proj1_1/util/test.py", line 229, in test_unit_traj_e
    self.assertEqual(flat_outputs['x'].shape, (3,))
AssertionError: Tuples differ: (3, 1) != (3,)

First tuple contains 1 additional elements.
First extra element 1:
1

- (3, 1)
?    --

+ (3,)
```

We see that the code expected the variable flat_outputs['x'] to have a shape of (3,) like in the provided code template, but instead flat_outputs['x'] has a shape of (3,1).

**Take advantage of basic data types.**

- A 'np.array' is great if you want to do math.

- A ‘tuple’ is great for storing a short, constant ordered collections of things, like a shape or index for a numpy array.
- A ‘list’ is great for storing an ordered collection of things that you want to add, remove, or modify.
- A ‘dict’ is great for storing an unordered collections of things (‘values’) that you can access by name (‘key’).
- A ‘set’ is great for storing an unordered collection of unique things, and testing if a particular thing is already in your set.