Dirt Simple Matrix Inversion

For Insight and Future Research Tools

https://github.com/ThomIves/MatrixInverse

We are going to walk thru a brute force procedural method for inverting a matrix with pure Python. Why wouldn't we just use numpy? Great question. This blog is about tools that add efficiency ***AND*** clarity. I love numpy, pandas, sklearn, and all the great tools that the python data science community brings to us, but I have learned that the better I understand the "innards" of a thing, the better I know how to apply it. Plus, *tomorrows machine learning tools will be developed by those that understand the **innards** of the math and coding of today's tools.*

Also, once an efficient method of matrix inversion is understood, you are ~ 80% of the way to having your own Least Squares Solver and a component to many other personal analysis modules to help you better understand how all our great machine learning tools are built. Would I recommend that you use what we are about to develop for a real project? All those python modules mentioned above are lightening fast, so, usually, no. I would not recommend that you use your own such tools *UNLESS* you are working with smaller problems, ***OR*** you are investigating some new approach that requires slight changes to your personal tool suite. Thus, a statement above bears repeating: *tomorrows machine learning tools will be developed by those that understand the **innards** of the math and coding of today's tools.* I want to be part of, or at least foster, those that will make the next gen tools. Plus, if you are a geek, knowing how to code the inversion of a matrix is fun!

The way I was taught to inverse matrices, *in the dark ages that is*, was pure torture! If you go about it the way that you would program it, it is MUCH easier in my opinion. I would even think it's easier doing the method we will use when doing it by hand than the ancient teaching of how to do it. In fact, it is so easy that we will start with a 5x5 matrix to make it "clearer".

DON'T PANIC. The only really painful thing about it, is that, while it's very simple, it's a bit tedious and boring. However, compared to the ancient method, it's simple. Or, as one of my favorite mentors would commonly say, "It's simple. it's just not easy." We'll use python, to reduce the tedium, without losing any view to the insights of the methods.

We'll use python at first through a Jupyter notebook to clearly illustrate each step. Then, we'll be VERY ready to adapt those steps to build our own module. I will seek to be very pep8'ish. Please deviate from my style as you wish to make what we are doing your own and more clear to you. You'll be glad that you did.

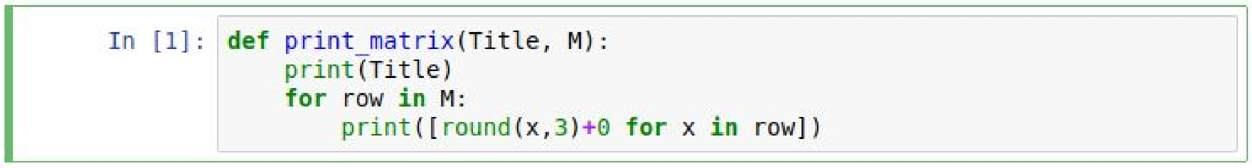
Let's start with the logo for the github repo that stores all this work, because it really says it all:

$$AX = IB$$
$$IX = A^{-1}B$$

Following the main rule of algebra (whatever we do to one side of the equal sign, we will do to the other side of the equal sign, in order to “stay true” to the equal sign), we will perform row operations to **A** in order to methodically turn it into an identity matrix while applying those same steps to what is “initially” the identity matrix on the right. When what was **A** becomes an identity matrix, what was **I** on the right will become **A**$^{-1}$.

If at some point, you have a big **“Ah HA!”** moment, try to work ahead on your own and compare to what we’ve done once you’ve finished or if you get stuck.

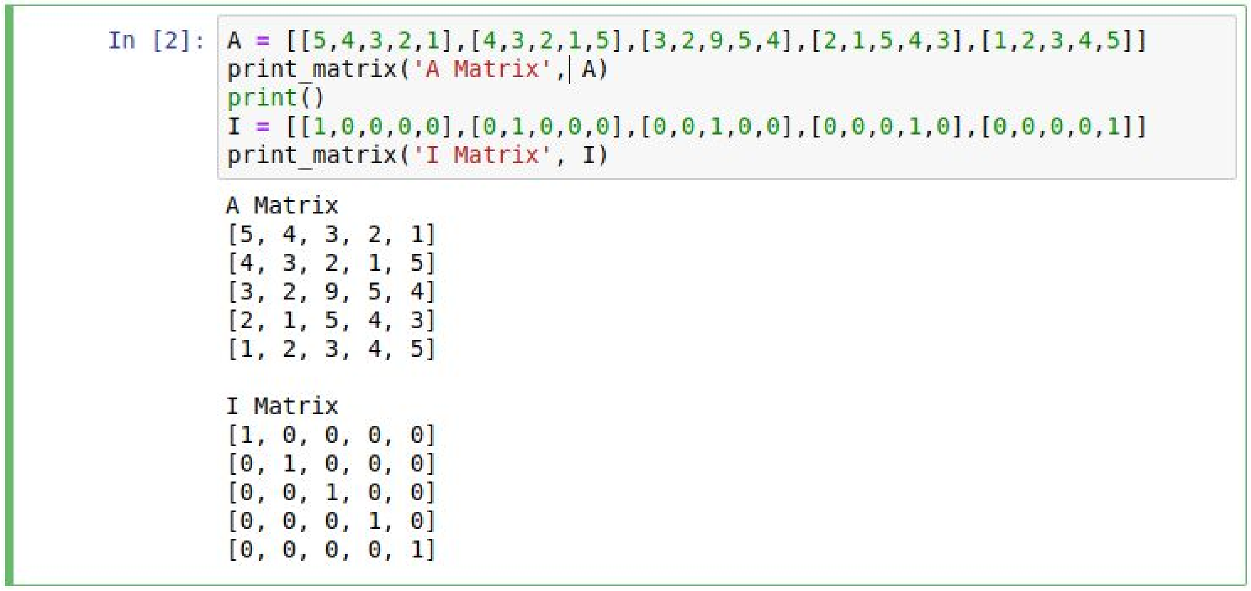
The Jupyter notebook called **MatrixInversion.ipynb** can be obtained from the github repo for this project. You don’t need to use Jupyter to follow along. I’ve also saved the cells as MatrixInversion.py in the same repo. Let’s first define a function that will give us a nice printout of the matrices we’ll be manipulating.



```
In [1]: def print_matrix(Title, M):
            print(Title)
            for row in M:
                print([round(x,3)+0 for x in row])
```

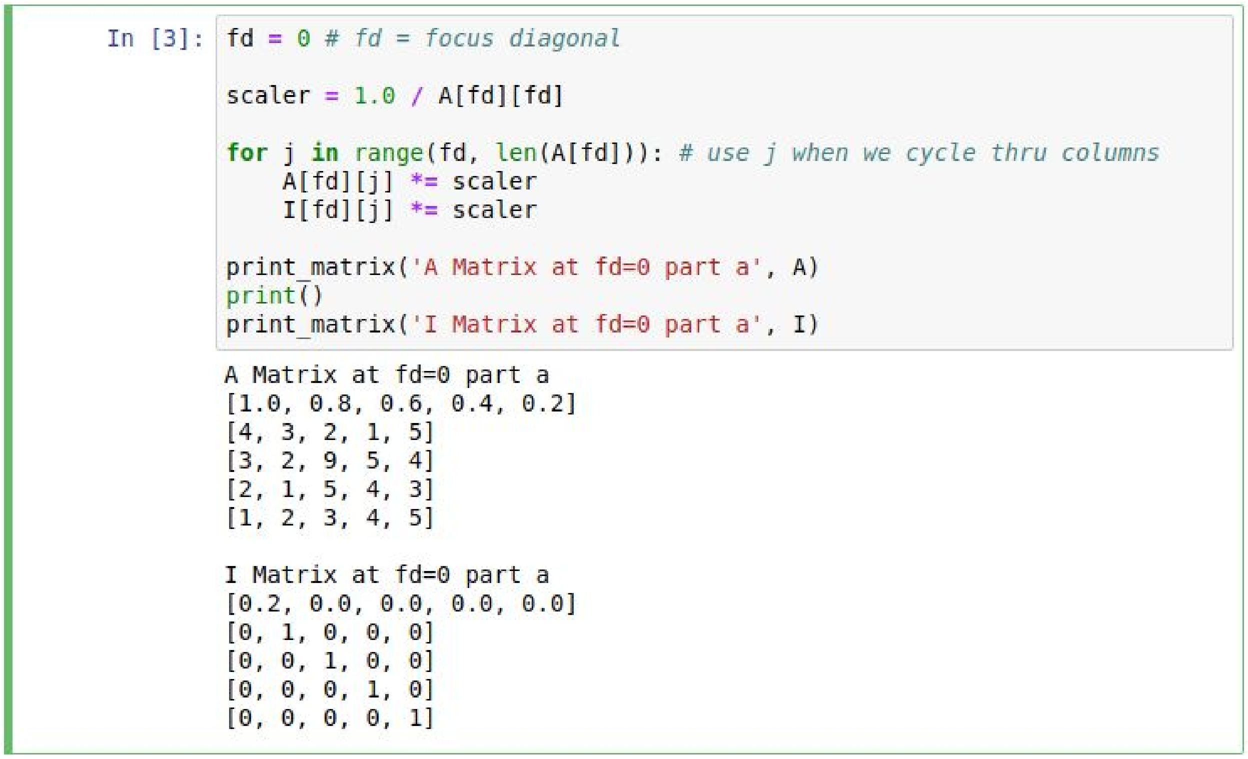
NOTE: The last print statement uses a trick to get rid of -0.0’s. Try it with and without the “+0” to see what I mean.

Let’s prepare some matrices to use.



```
In [2]: A = [[5,4,3,2,1],[4,3,2,1,5],[3,2,9,5,4],[2,1,5,4,3],[1,2,3,4,5]]
        print_matrix('A Matrix', A)
        print()
        I = [[1,0,0,0,0],[0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]]
        print_matrix('I Matrix', I)

        A Matrix
        [5, 4, 3, 2, 1]
        [4, 3, 2, 1, 5]
        [3, 2, 9, 5, 4]
        [2, 1, 5, 4, 3]
        [1, 2, 3, 4, 5]

        I Matrix
        [1, 0, 0, 0, 0]
        [0, 1, 0, 0, 0]
        [0, 0, 1, 0, 0]
        [0, 0, 0, 1, 0]
        [0, 0, 0, 0, 1]
```

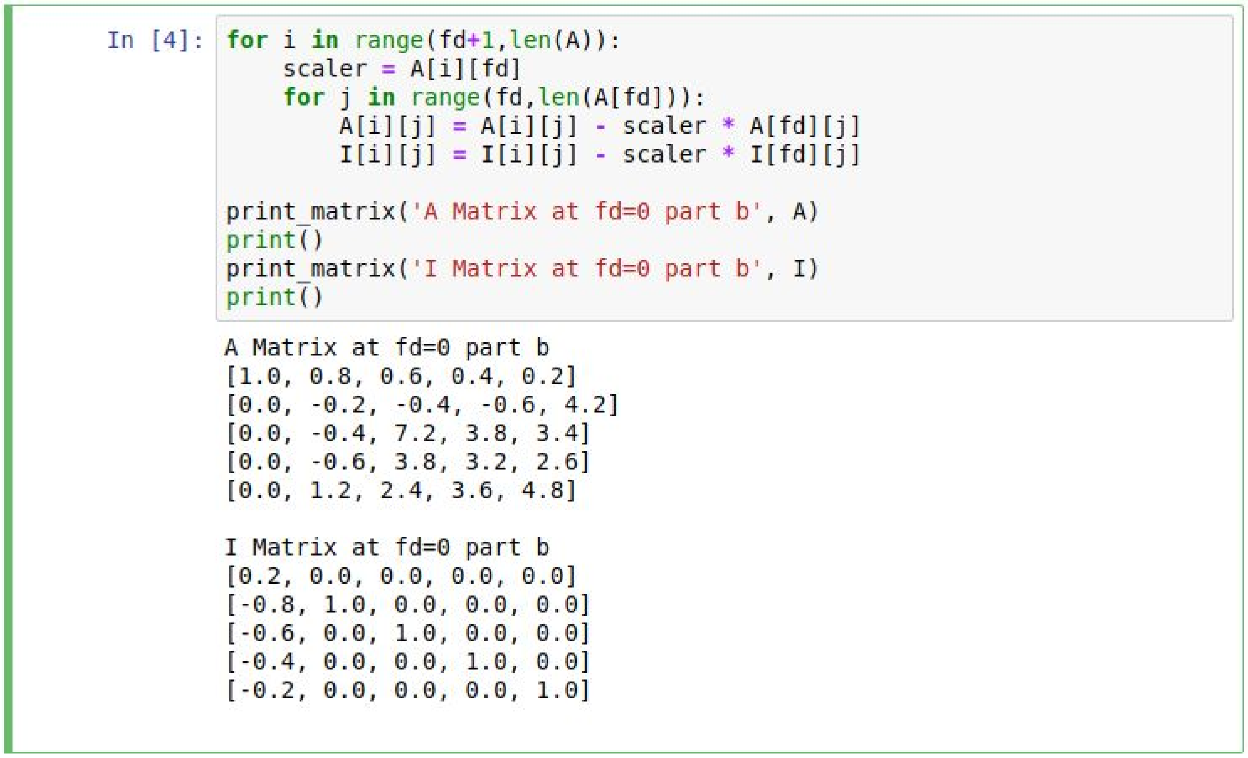
The first basic step, focusing on the first element of the diagonal of **A**, is to divide all elements of the first row by **5**. From here forward, all operations applied to **A** are applied to **I** also.



```
In [3]: fd = 0 # fd = focus diagonal

        scaler = 1.0 / A[fd][fd]

        for j in range(fd, len(A[fd])): # use j when we cycle thru columns
            A[fd][j] *= scaler
            I[fd][j] *= scaler

        print_matrix('A Matrix at fd=0 part a', A)
        print()
        print_matrix('I Matrix at fd=0 part a', I)
```

```
A Matrix at fd=0 part a
[1.0, 0.8, 0.6, 0.4, 0.2]
[4, 3, 2, 1, 5]
[3, 2, 9, 5, 4]
[2, 1, 5, 4, 3]
[1, 2, 3, 4, 5]

I Matrix at fd=0 part a
[0.2, 0.0, 0.0, 0.0, 0.0]
[0, 1, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 1, 0]
[0, 0, 0, 0, 1]
```

Now we can *easily* scale the first row's values by the value in the first column of all the rows below it and subtract that scaled first row from each row below it to leave zeros below the first diagonal element. If you're as big a geek as me, you have chills now.


```
In [4]: for i in range(fd+1,len(A)):
            scaler = A[i][fd]
            for j in range(fd,len(A[fd])):
                A[i][j] = A[i][j] - scaler * A[fd][j]
                I[i][j] = I[i][j] - scaler * I[fd][j]

        print_matrix('A Matrix at fd=0 part b', A)
        print()
        print_matrix('I Matrix at fd=0 part b', I)
        print()
```

```
A Matrix at fd=0 part b
[1.0, 0.8, 0.6, 0.4, 0.2]
[0.0, -0.2, -0.4, -0.6, 4.2]
[0.0, -0.4, 7.2, 3.8, 3.4]
[0.0, -0.6, 3.8, 3.2, 2.6]
[0.0, 1.2, 2.4, 3.6, 4.8]

I Matrix at fd=0 part b
[0.2, 0.0, 0.0, 0.0, 0.0]
[-0.8, 1.0, 0.0, 0.0, 0.0]
[-0.6, 0.0, 1.0, 0.0, 0.0]
[-0.4, 0.0, 0.0, 1.0, 0.0]
[-0.2, 0.0, 0.0, 0.0, 1.0]
```

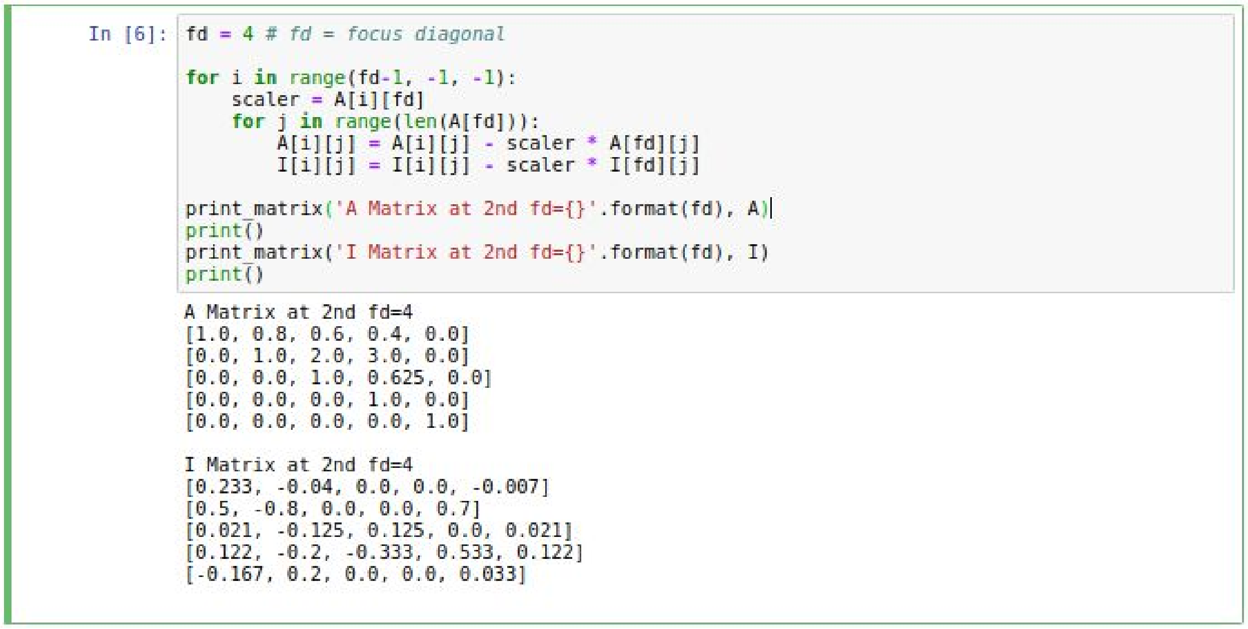
Repeating this process will make the lower triangle of the matrix all zeros. We focus on the next element on the diagonal of our morphing **A** matrix, and repeat the above steps. Thus, we divide the second row by -0.2. Like before, but starting from the second row now, we again scale the second row's values by the value in the second column of all the rows below the second row and subtract that scaled second row from each row below it to leave zeros below the second diagonal element, which is now 1.0. A new outermost **for** loop controls the focus diagonal.

In [5]:

```
for fd in range(1,5): # fd = focus diagonal

    scaler = 1.0 / A[fd][fd]

    for j in range(len(A[fd])): # use j when we cycle thru columns
        A[fd][j] *= scaler
        I[fd][j] *= scaler

    for i in range(fd+1,len(A)):
        scaler = A[i][fd]
        for j in range(len(A[fd])):
            A[i][j] = A[i][j] - scaler * A[fd][j]
            I[i][j] = I[i][j] - scaler * I[fd][j]

    print_matrix('A Matrix at fd={}'.format(fd), A)
    print()
    print_matrix('I Matrix at fd={}'.format(fd), I)
    print()
```

```
A Matrix at fd=1
[1.0, 0.8, 0.6, 0.4, 0.2]
[0.0, 1.0, 2.0, 3.0, -21.0]
[0.0, 0.0, 8.0, 5.0, -5.0]
[0.0, 0.0, 5.0, 5.0, -10.0]
[0.0, 0.0, 0.0, 0.0, 30.0]

I Matrix at fd=1
[0.2, 0.0, 0.0, 0.0, 0.0]
[4.0, -5.0, 0.0, 0.0, 0.0]
[1.0, -2.0, 1.0, 0.0, 0.0]
[2.0, -3.0, 0.0, 1.0, 0.0]
[-5.0, 6.0, 0.0, 0.0, 1.0]

A Matrix at fd=2
[1.0, 0.8, 0.6, 0.4, 0.2]
[0.0, 1.0, 2.0, 3.0, -21.0]
[0.0, 0.0, 1.0, 0.625, -0.625]
[0.0, 0.0, 0.0, 1.875, -6.875]
[0.0, 0.0, 0.0, 0.0, 30.0]

I Matrix at fd=2
[0.2, 0.0, 0.0, 0.0, 0.0]
[4.0, -5.0, 0.0, 0.0, 0.0]
[0.125, -0.25, 0.125, 0.0, 0.0]
[1.375, -1.75, -0.625, 1.0, 0.0]
[-5.0, 6.0, 0.0, 0.0, 1.0]

A Matrix at fd=3
[1.0, 0.8, 0.6, 0.4, 0.2]
[0.0, 1.0, 2.0, 3.0, -21.0]
[0.0, 0.0, 1.0, 0.625, -0.625]
[0.0, 0.0, 0.0, 1.0, -3.667]
[0.0, 0.0, 0.0, 0.0, 30.0]

I Matrix at fd=3
[0.2, 0.0, 0.0, 0.0, 0.0]
[4.0, -5.0, 0.0, 0.0, 0.0]
[0.125, -0.25, 0.125, 0.0, 0.0]
[0.733, -0.933, -0.333, 0.533, 0.0]
[-5.0, 6.0, 0.0, 0.0, 1.0]

A Matrix at fd=4
[1.0, 0.8, 0.6, 0.4, 0.2]
[0.0, 1.0, 2.0, 3.0, -21.0]
[0.0, 0.0, 1.0, 0.625, -0.625]
[0.0, 0.0, 0.0, 1.0, -3.667]
[0.0, 0.0, 0.0, 0.0, 1.0]

I Matrix at fd=4
[0.2, 0.0, 0.0, 0.0, 0.0]
[4.0, -5.0, 0.0, 0.0, 0.0]
[0.125, -0.25, 0.125, 0.0, 0.0]
[0.733, -0.933, -0.333, 0.533, 0.0]
[-0.167, 0.2, 0.0, 0.0, 0.033]
```

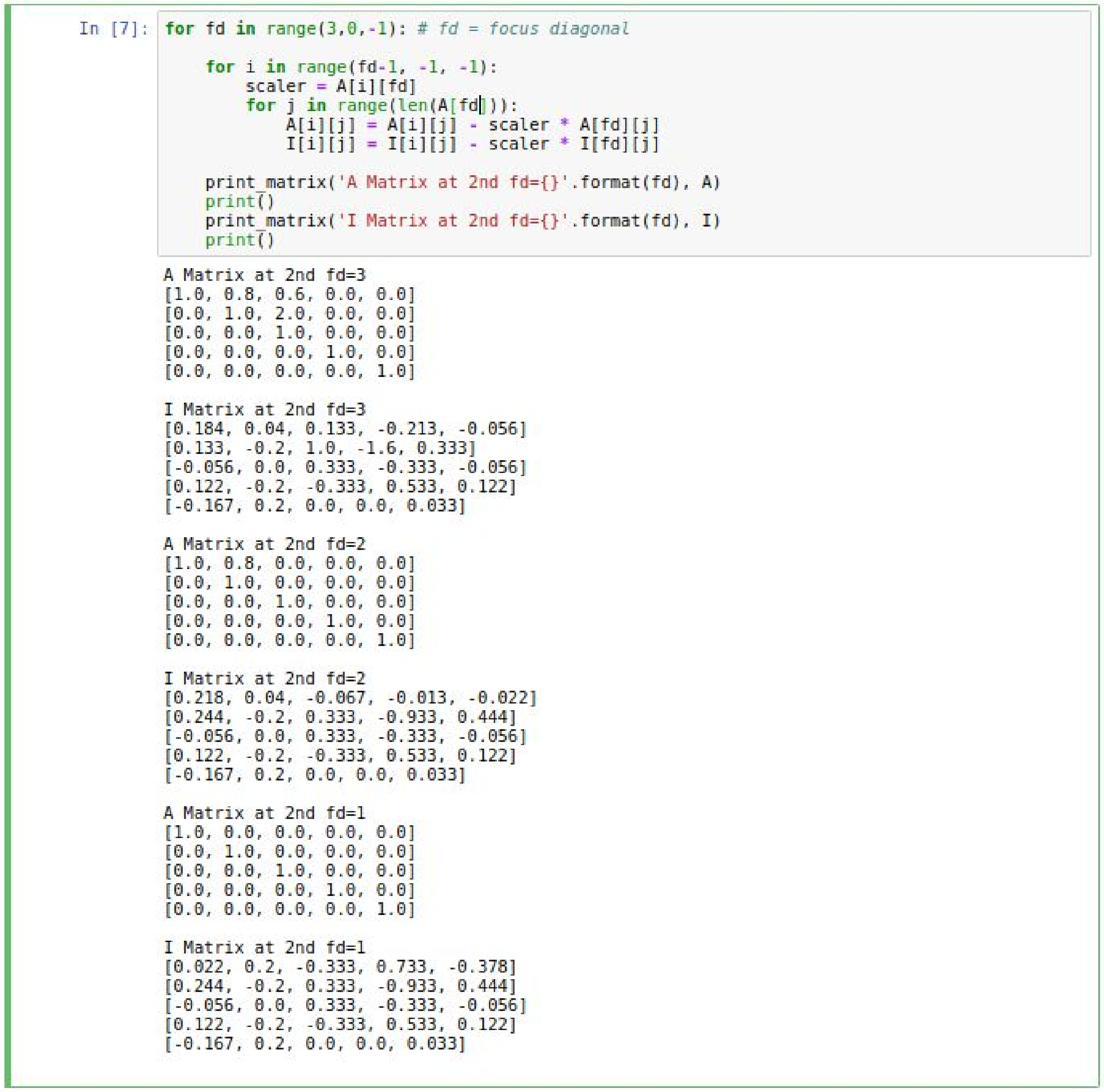
Now, we need to make the upper triangular matrix all zeros. We will work from the bottom of the matrix up this time for convenience. We have less work on the way up from the bottom, because we don't need to scale each row so that the diagonals are 1.0. We've already done that. So the first step is to scale row 5 by the last element of row 4 and subtract that from row 4 to create a zero in the last element of row 4. Remember, EVERYTHING that we're doing to the morphing **A** matrix must be done to the morphing **I** matrix too. If we don't, the **I** matrix will not become the inverse of **A**.



```
In [6]: fd = 4 # fd = focus diagonal

        for i in range(fd-1, -1, -1):
            scaler = A[i][fd]
            for j in range(len(A[fd])):
                A[i][j] = A[i][j] - scaler * A[fd][j]
                I[i][j] = I[i][j] - scaler * I[fd][j]

        print_matrix('A Matrix at 2nd fd={}'.format(fd), A)
        print()
        print_matrix('I Matrix at 2nd fd={}'.format(fd), I)
        print()
```

```
A Matrix at 2nd fd=4
[1.0, 0.8, 0.6, 0.4, 0.0]
[0.0, 1.0, 2.0, 3.0, 0.0]
[0.0, 0.0, 1.0, 0.625, 0.0]
[0.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 1.0]

I Matrix at 2nd fd=4
[0.233, -0.04, 0.0, 0.0, -0.007]
[0.5, -0.8, 0.0, 0.0, 0.7]
[0.021, -0.125, 0.125, 0.0, 0.021]
[0.122, -0.2, -0.333, 0.533, 0.122]
[-0.167, 0.2, 0.0, 0.0, 0.033]
```

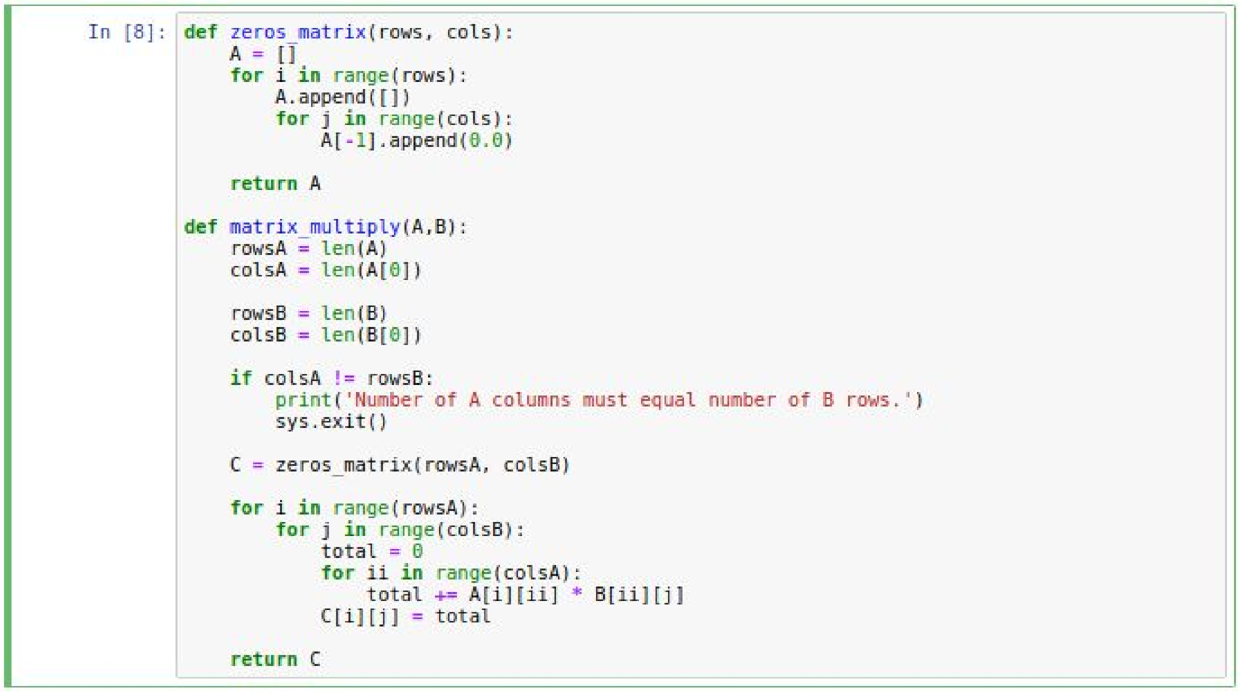
We repeat this step similar to the way we did before for the above diagonal elements, but, we're doing it from the bottom up. Also, we add an outer for loop to control which diagonal we're on.



```
for fd in range(3,0,-1): # fd = focus diagonal

    for i in range(fd-1, -1, -1):
        scaler = A[i][fd]
        for j in range(len(A[fd])):
            A[i][j] = A[i][j] - scaler * A[fd][j]
            I[i][j] = I[i][j] - scaler * I[fd][j]

    print_matrix('A Matrix at 2nd fd={}'.format(fd), A)
    print()
    print_matrix('I Matrix at 2nd fd={}'.format(fd), I)
    print()
```

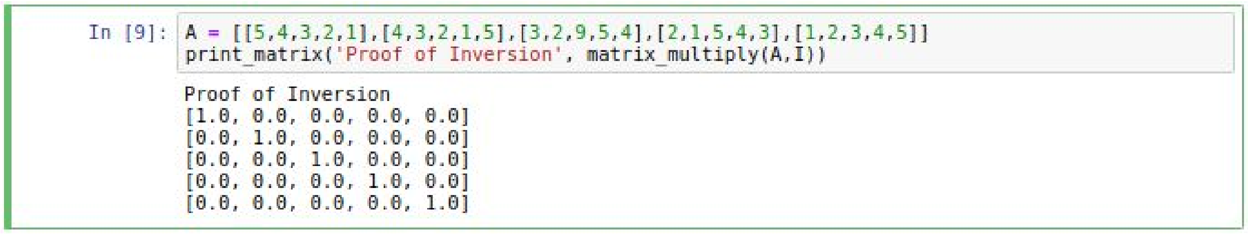
```
A Matrix at 2nd fd=3
[1.0, 0.8, 0.6, 0.0, 0.0]
[0.0, 1.0, 2.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 1.0]

I Matrix at 2nd fd=3
[0.184, 0.04, 0.133, -0.213, -0.056]
[0.133, -0.2, 1.0, -1.6, 0.333]
[-0.056, 0.0, 0.333, -0.333, -0.056]
[0.122, -0.2, -0.333, 0.533, 0.122]
[-0.167, 0.2, 0.0, 0.0, 0.033]

A Matrix at 2nd fd=2
[1.0, 0.8, 0.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 1.0]

I Matrix at 2nd fd=2
[0.218, 0.04, -0.067, -0.013, -0.022]
[0.244, -0.2, 0.333, -0.933, 0.444]
[-0.056, 0.0, 0.333, -0.333, -0.056]
[0.122, -0.2, -0.333, 0.533, 0.122]
[-0.167, 0.2, 0.0, 0.0, 0.033]

A Matrix at 2nd fd=1
[1.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 1.0]

I Matrix at 2nd fd=1
[0.022, 0.2, -0.333, 0.733, -0.378]
[0.244, -0.2, 0.333, -0.933, 0.444]
[-0.056, 0.0, 0.333, -0.333, -0.056]
[0.122, -0.2, -0.333, 0.533, 0.122]
[-0.167, 0.2, 0.0, 0.0, 0.033]
```

Success! **A** has morphed into an Identity matrix, and **I** has become the inverse of **A**. Yay! And yes, I am easily entertained. Now, two more helper functions to help us prove that we've achieved success.



```
In [8]: def zeros_matrix(rows, cols):
            A = []
            for i in range(rows):
                A.append([])
                for j in range(cols):
                    A[-1].append(0.0)

            return A

        def matrix_multiply(A,B):
            rowsA = len(A)
            colsA = len(A[0])

            rowsB = len(B)
            colsB = len(B[0])

            if colsA != rowsB:
                print('Number of A columns must equal number of B rows.')
                sys.exit()

            C = zeros_matrix(rowsA, colsB)

            for i in range(rowsA):
                for j in range(colsB):
                    total = 0
                    for ii in range(colsA):
                        total += A[i][ii] * B[ii][j]
                    C[i][j] = total

            return C
```

Let’s apply these functions to our proof.



```
In [9]: A = [[5,4,3,2,1],[4,3,2,1,5],[3,2,9,5,4],[2,1,5,4,3],[1,2,3,4,5]]
        print_matrix('Proof of Inversion', matrix_multiply(A,I))

        Proof of Inversion
        [1.0, 0.0, 0.0, 0.0, 0.0]
        [0.0, 1.0, 0.0, 0.0, 0.0]
        [0.0, 0.0, 1.0, 0.0, 0.0]
        [0.0, 0.0, 0.0, 1.0, 0.0]
        [0.0, 0.0, 0.0, 0.0, 1.0]
```

Yes! When we multiply the original A matrix on our Inverse matrix we do get the identity matrix.

I do love Jupyter notebooks, but I want to use this in scripts now too. See below.

```
def inverse_matrix(A):
    check_squareness(A) # a separate function in the repo

    n = len(A)
    AM = copy_matrix(A) # this function is also in the repo
    Inv = identity_matrix(n) # this function is also in the repo

    # Make the lower triangular matrix all 0’s and make all diagonal elements 1’s
    for fd in range(n): # fd is for focus diagonal
        # The next for loop is started one row lower each time in accordance with the current
fd
        for i in range(fd,n):
            if i == fd: # if current row = fd, the scale is the inverse of the fd for the fd’s row ...
                scaler = 1.0 / AM[i][fd]
        else: # ... otherwise the scaler is in the element of the col = fd that we are in
            scaler = AM[i][fd]
        for j in range(n):
            if i == fd: # scale the row when = the fd to make diagonals 1
                AM[i][j] *= scaler
                Inv[i][j] *= scaler
            else: # scale the row = the fd by the number if the column = the fd for this row
                AM[i][j] = AM[i][j] - scaler * AM[fd][j]
                Inv[i][j] = Inv[i][j] - scaler * Inv[fd][j]

    # Do the above steps from the bottom up and from right to left
    #     to make the upper triangular matrix all 0's. All diagonal elements are 1 now.
    for fd in range(n-1,0,-1):
        for i in range(fd-1,-1,-1):
            scaler = AM[i][fd]
            for j in range(n):
                AM[i][j] = AM[i][j] - scaler * AM[fd][j]
                Inv[i][j] = Inv[i][j] - scaler * Inv[fd][j]

    return Inv
```

In future posts, we will start from here to see first hand how this can be applied to basic machine learning and how it applies to other techniques beyond basic least squares linear regression.