Applied Linear Algebra - Lab 1

Introduction to NumPy, gaussian elimination, interpolation and color grading.

Contents:

Numpy

Datasets can be made of collections of images, sounds, videos, documents, numerical measurements, or, really anything. Despite the diversity, it will help us to think of all data fundamentally as arrays of numbers.

Data type	Arrays of Numbers?
Images	Pixel brightness across different channels
Videos	Pixels brightness across different channels for each frame
Sound	Intensity over time
Numbers	No need for transformation
Tables	Mapping from strings to numbers

Therefore, the efficient storage and manipulation of large arrays of numbers is really fundamental to the process of doing data science. Numpy is one of the libraries within the scientific stack that specialize in handling numerical arrays and data tables.

Numpy is short for *numerical python*, and provides functions that are especially useful when you have to work with large arrays and matrices of numeric data, like matrix multiplications.

The array object class is the foundation of Numpy, and Numpy arrays are like lists in Python, except that every thing inside an array must be of the same type, like int or float. As a result, arrays provide much more efficient storage and data operations, especially as the arrays grow larger in size. However, in other ways, NumPy arrays are very similar to Python's built-in list type, but with the exception of Vectorization.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
from IPython import display

# Global floating point precision
precision = 2
```

Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The shape of an array is a tuple of integers giving the size of the array along each dimension. A a one dimensional array (shape `(n,)`) corresponds to a vector.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

In [2]:
```
a = np.array([1, 2, 3])  # Create a 1 dimensional array i.e. a vector

print("a is type: ", type(a))
print("The shape of the vector a is: ", a.shape)
print(a[0], a[1], a[2])  # indexing
a[0] = 5  # Change an element of the array
print(a)
```

```
a is type:  <class 'numpy.ndarray'>
The shape of the vector a is:  (3,)
1 2 3
[5 2 3]
```

In [3]:
```
b = np.array([[1,2,3],[4,5,6]])   # Create a 2 dimensional array i.e. a matrix
print(b)
print("The shape of the matrix b is: ", b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
[[1 2 3]
 [4 5 6]]
The shape of the matrix b is:  (2, 3)
1 2 4
```

Array Arithmetic

Basic mathematical functions operate elementwise on arrays and matrices (which are just 2D arrays), and are available both as operator overloads and as functions in the numpy module:

In [ ]:
```
# Define two matrices
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

# Elementwise sum:
print(x + y)
print(np.add(x, y))
```

In [ ]:
```
# Elementwise difference:
print(x - y)
print(np.subtract(x, y))
```

In [ ]:
```
# Elementwise product:
#  * is elementwise multiplication, not matrix multiplication!
print(x * y)
print(np.multiply(x, y))
```

In [ ]:
```
# Elementwise square root:
print(np.sqrt(x))
```

In [ ]:
```
# Elementwise division:
print(x / y)
print(np.divide(x, y))
```

Exercise 1

Given two vectors v_1 and v_2, calculate v_3 where

$$v_3 = 12(v_1 - 2v_2)$$

In [ ]:
```
# define the two vectors
v1 = np.array([2,3,4,7])
v2 = np.array([6,1,0,3])

# calculate v3
v3 = ...
print(v3)
```

We use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

In [ ]:
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors.
print(v.dot(w))
print(np.dot(v, w))
print(v @ w)

# Matrix multiplication.
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)

x = np.matrix(x)
y = np.matrix(y)
x*y
```

We can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators +, -, * to use matrix algebra. Make sure the dimensions of the two matrices are compatible. You can use the `np.transpose()` function or the `T` method of NumPy vectors and matrices.

In [ ]:

```
# Create two row vectors
v = np.matrix(v)
w = np.matrix(w)

# Create two matrices
x = np.matrix(x)
y = np.matrix(y)

# Inner product of vectors.
print(v * w.T)

# Matrix multiplication.
print(x * y)

# Matrix Vector multiplication
print(x * v.T)
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the NumPy documentation. NumPy also supports compound data types.

Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in numpy that generate arrays of different forms. Some of the more common are:

In [ ]:
```
# We use these when the elements of the
# arrays are originally unknown but their size is known.

np.zeros((3,4))
```

In [ ]:
```
np.ones((2,3), dtype = np.int_)
```

In [ ]:
```
np.empty( (2,3) )
```

In [ ]:
```
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

In [ ]:
```
# Create a 3x3 Identity Matrix
np.eye(3)
```

Exercise 2

Given vector v, calculate the average of its elements using dot product.

In [ ]:
```
# define vector v
v = np.array([ 3, 5, -3, 7, 1 ])

# vector of ones
ones_vector = ...

# calculating the average using dot product
average = ...
average
```

Exercise 3

Matrices can *transform* the vectors that they are multiplied with. define vector $u = [3, -4]^T$ and matrix $A = \begin{bmatrix} 1 & -3 \\ 2 & -2 \end{bmatrix}$ then calculate and plot the result of Au which we'll call v.

Make sure that u is a 2×1 column vector.

In [ ]:
```
# define the vector. this is a row vector
u = ...

# define the 2x2 matrix
A = ...

# output vector is Av (convert v to column)
v = ...


# plotting
plt.plot([0,u[0]],[0,u[1]],label='u')
plt.plot([0,v[0]],[0,v[1]],label='Au')
plt.grid()
plt.axis((-2, 12, -6, 12))
plt.legend()
plt.show()
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x` , use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0` , `stop=` *`size of dimension`* , `step=1` . We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

In [ ]:
```
M = np.random.randint(100, size=(10, 12))
print("Initial matrix: ")
print(M)
```

In [ ]:
```
M[1,:] # second row
```

In [ ]:
```
M[:,1] # second column
```

In [ ]:
```
# assignment can also work for rows and columns. This is really powerful and fast!
M[1,:] = 0
M
```

In [ ]:
```
M[:,2] = -1
M
```

In [ ]:
```
M[::2] # step is 2, lower and upper defaults to the beginning and end of the array
```

In [ ]:
```
M[:3] # first three rows
```

In [ ]:
```
M[3:] # rows from row 3 to the end
```

In [ ]:
```
# slice a block from the original array
M[1:4, 1:4]
```

In [ ]:
```
# slice with different strides
M[::2, ::2]
```

Implementing Gaussian Elimination

In this exercise we'll try to implement gaussian elimination. First we create a couple of functions for elementry row operations.

Each of the elementary row operations is the result of matrix multiplication by an elementary matrix (on the left).

Exercise 4

Row Swap

For swapping row i with row j in a $m \times n$ matrix A, we multiply A by an $m \times m$ matrix E where E is equal to the identity matrix I_m except $E_{ii} = E_{jj} = 0$, and $E_{ij} = E_{ji} = 1$. (Equivalently, we can interchange i-th row and j-th row of the identity matrix I to get E.) For example, if A is a 3×3 matrix and we would like to swap row 1 with row 3, then E would be equal to:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Now try to define the `row_swap` function which take a matrix A and two indices i and j as its inputs and returns a matrix which is equal to A with its i-th row and j-th row swapped.

In [ ]:
```
def row_swap(A, i, j):
    "Swap row i and j in matrix A."
    m = ...
    E = np.eye(m)
    ...
    return np.around(E @ A, precision)
```

In [ ]:
```
A = np.array([[1,2,3],[4,-5,6],[7,-8,9]])
print("Before row exchange:")
print(A)
print("After row exchange:")
print(row_swap(A, 1, 2))
```

Exercise 5

Row Sum

For summing $k\times$ row i with row j in a $m \times n$ matrix A, we multiply A by the matrix E where E is equal to the identity matrix I_m except $E_{ji} = k$. For example, if A is 3 by 5 and we want to add -2 times 3 to row 1. then E would be equal to:

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now try to define the `row_sum` function which takes matrix A, scalar k and indices i, j as its inputs and returns the matrix resulting from adding k times row i to row j in the matrix A.

In [ ]:
```
# A simple exception for when the indices given to row_sum function are equal
class GERowSwapSameIndexException(Exception):
    def __init__(self):
        super().__init__("Error: indices given to the function must be different.")

def row_sum(A,k,i,j):
    "Add k times row i to row j in matrix A."
    if (i == j):
        raise GERowSwapSameIndexException

    m = ...
    E = ...
    ...
    return np.around(E @ A, precision)
```

In [ ]:
```
A = np.array([[1,2,3],[4,-5,6],[7,-8,9]])
print("Before row sum:")
print(A)
print("After row sum:")
print(row_sum(A, 2, 1, 2))
```

Exercise 6

Row Scale

For summing $k\times$ row i with row j in a $m \times n$ matrix A, we multiply A by the matrix E where E is equal to the identity matrix I_m except $E_{ii} = k$. For example, if A is 4 by 3 and we want to multiply row 3 by -4 then E would be equal to:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The implementation of a function that scales a row of a matrix must be trivial by now. It take a matrix A, a scalar k and an index i and returns the matrix that results from k times row i in the matrix A.

In [ ]:
```
def row_scale(A,k,i):
    "Multiply row i by k in matrix A"
    m = ...
    E = ...
    ...
    return np.around(E @ A, precision)
```

In [ ]:
```
A = np.array([[1,2,3],[4,-5,6],[7,-8,9]])
print("Before row scale:")
print(A)
print("After row scale:")
print(row_scale(A, -3, 1))
```

Exercise 7

Gaussian Elimination

Having defined the elementary row operations, now let's implement gaussian elimination.

Define the function `eliminate` which takes a matrix A and a vector b and applies gaussian elimination on them and then returns the resulting matrix and vector.

In [ ]:
```
# A simple exception for when the indices given to row_sum function are equal
class GEEliminationErr(Exception):
    def __init__(self, msg):
        super().__init__(msg)

def is_zero(x, epsilon=1e-10):
    return abs(x) < epsilon

def gaussian_elimination(A):
    number_rows = A.shape[0]
    try:
        for i in range(number_rows):
            pivot = ...

            if is_zero(pivot):
                swap_index = (np.where(A[i+1:,i] != 0)[0][0]) + i + 1
                A = ...

            # Eliminate elements below the pivot
            loopRange = ...
            for j in range(loopRange):
                multiplier = ...
                A = ...
                print("Row operation: ")
                print(A)

        if A[number_rows-1, number_rows-1] == 0:
            raise GEEliminationErr("Matrix is not full rank. There are no unique solution.")

    except IndexError as exc:
        raise GEEliminationErr("Pivot not found. There are no unique solutions.") from exc
    return A
```

Exercise 8

Next, implement the `back_substitution` function. This function solves a linear system of equations that has been transformed into reduced row-echelon form. It takes a matrix A and a vector b in their rref forms and returns the solution of $Ax = b$.

In [ ]:
```
def back_substitution(A,b):
    n = A.shape[0]
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        tmp = b[i]
        for j in range(n-1, i, -1):
            ...
        ...
    return np.around(np.matrix(x), precision)
```

In [ ]:
```
# define an example matrix
A = np.matrix([[1,2,3], [1,2,5], [3,9,7]], dtype = float)
print("Original Matrix")
print(A)

# define vector b
b = np.mat([2,3,1], dtype = float)
b = b.reshape(3,1)
```

```
Aug = np.concatenate((A, b), axis=1)
print("Augmented Matrix")
print(Aug)

# Applying the elimination
ref_Aug = gaussian_elimination(Aug)
# Extract everything BUT the last column of Aug
ref_A = ref_Aug[:, :-1].copy()
# Extract the last column of Aug
ref_b = ref_Aug[:, -1].copy()

z = back_substitution(ref_A, ref_b)
print("Final solution of system of equations is ")
print(z.T)
```

Exercise 9

Finding the inverse

Implement the `reduced_row_echelon_form(A)` function which also eliminates elements that lie above the pivots and also changes the pivots to 1s (the corresponding rows of those pivots must be scaled respectfully). It should return a matrix of the form $[IM]$.

In [ ]:
```
def reduced_row_echelon_form(A):
    number_rows = A.shape[0]
    try:
        for i in range(number_rows):
            pivot = ...

            if is_zero(pivot):
                swap_index = (np.where(A[i+1:,i] != 0)[0][0]) + i + 1
                A = ...

            # Eliminate elements below the pivot
            loopRange = ...
            for j in range(loopRange):
                multiplier = ...
                A = ...
                print("Row operation :")
                print(A)

            # Eliminate elements above the pivot
            loopRange = ...
            for j in range(loopRange):
                multiplier = ...
                A = ...
                print("Row operation :")
                print(A)

            # Scale the pivot to 1
            A = ...
            print("Row operation :")
            print(A)

        if A[number_rows-1, number_rows-1] == 0:
            raise GEEliminationErr("Matrix is not full rank. There are no unique solution.")

    except IndexError as exc:
        raise GEEliminationErr("Pivot not found. There are no unique solutions.") from exc
    return A
```

In [ ]:
```
# define an example matrix
A = np.matrix([[1,2,3], [1,2,5], [3,9,7]], dtype = float)
I = np.eye(A.shape[0])
print("Original Matrix")
print(A)

# define vector b
b = np.mat([2,3,1], dtype = float)
b = b.reshape(3,1)

Aug = np.concatenate((A, I), axis=1)
print("Augmented Matrix")
print(Aug)

# Applying the elimination
rref_Aug = reduced_row_echelon_form(Aug)
print("Reduced Augmented Matrix:")
print(rref_Aug)

# Extract the last A.shape[0] columns of Aug
M = rref_Aug[:, A.shape[0]:].copy()
print("The inverse matrix with a precision of {} decimals is:".format(precision))
print(M)

print("Final solution of system of equations is ")
print(M @ b)
```

Polynomial Interpolation

Polynomial interpolation is a procedure for modeling a set of precise data points using a polynomial function, $p(x)$, that fits the data exactly (passes through all provided data points). The data points are normally obtained from a complicated mathematical model, $f(x)$, of an engineering or scientific system derived from physical principles. Once an

interpolation polynomial is computed, it can be used to replace the complicated mathematical model for the purpose of analysis and design. For instance, the interpolating polynomial can be used to estimate the value of the function at a new point x', as $f(x') \cong p(x')$. The solution for the coefficients of the interpolation polynomial $p(x)$ can be determined by solving an associated system of linear equations, or can be computed using formulas.

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

$$p(x_i) = y_i \qquad \text{for all } i \in \{0, 1, \ldots, n\}\,.$$

$$P\vec{a} = \vec{y} \rightarrow \begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \ldots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \ldots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \ldots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

We have to solve this system for ak to construct the interpolant p(x). The matrix on the left is commonly referred to as a Vandermonde matrix.

Exercise 10

Find a degree 3 polynomial that best fits the points:

$$(1, 3),\ (2, -2),\ (3, -5),\ (4, 0)$$

(use the `gaussian_elimination` function that you implemented above)

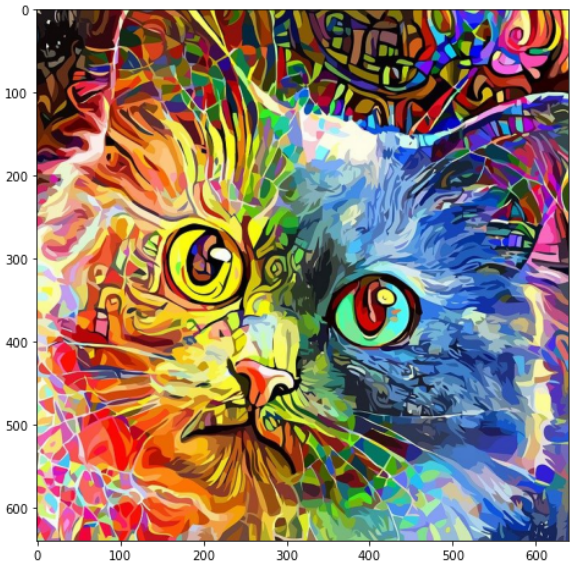
In [ ]:
```
P = ...
a = ...

Aug = ...
print("Augmented Matrix")
print(Aug)

# Applying the elimination
ref_Aug = ...
# Extract everything BUT the last column of Aug
ref_P = ...
# Extract the last column of Aug
ref_a = ...

z = ...
print("Final solution of system of equations is ")
print(z.T)
```

In [ ]:
```
# Plotting the polynomial
def polynomial_coefficients(xs, coeffs):
    order = len(coeffs)
    print(f'# This is a polynomial of order {order - 1}.')
    ys = np.zeros(len(xs))  # Initialise an array of zeros of the required length.
    for i in range(order):
        ys += coeffs[i] * xs ** i
    return ys


xs = np.linspace(-1, 4, 100)  # Change this range according to your needs. Start, stop, number of steps.
coeffs = z.flatten().tolist()[::-1]

plt.plot(xs, polynomial_coefficients(xs, coeffs))
plt.axhline(y=0, color='r')  # Show xs axis
plt.axvline(x=0, color='r')  # Show y axis
points = [(1, 3), (2, -2), (3, -5), (4, 0)]
for p in points:
    plt.plot(p[0], p[1], 'o', color='black');
plt.plot(1, 3)
plt.show()
```

Color Grading

The RGB color model is a method of describing colors. In this model each color is represented as a mixture of three basic colors: red, green, and blue. By varying intensities of these components a variety of colors can be obtained.

In [11]:
```
# import image
cat = plt.imread("/home/arashsm79/Playground/python-jupyter/notebooks/dist/student/cat.jpg")
plt.figure(figsize=(8,8))
plt.imshow(cat)
plt.show()
```



The `color_grader` function works as follows. It takes as its arguments a 3x3 matrix A and an array representing an image. For each image pixel it takes the vector

$$\mathbf{v} = \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

with RGB coordinates of the pixel, and replaces it with the vector $A\mathbf{v}$, which specifies the new pixel color. Then it displays the image with colors given by the vectors $A\mathbf{v}$. Since valid RGB values are integers between 0 and 255, coordinates of each vector $A\mathbf{v}$ are rounded to the nearest integer in this range. In particular, if $A\mathbf{v}$ has negative coordinates they are rounded up to 0, and if it has coordinates exceeding 255 they are rounded down to 255.

Here is the implementation of the `color_grader` function:

In [ ]:
```
def color_grader(A, img, width=8, height=8):
    A = np.array(A).astype(float)

    if img.dtype == 'uint8':
        img = img.astype(float)/255
    new_img = np.transpose(np.dot(A, np.transpose(img, axes = (0, 2, 1))), axes = (1, 2, 0))
    new_img[new_img < 0] = 0
    new_img[new_img > 1] = 1
    new_img = (255*new_img).astype('uint8')
    plt.figure(figsize=(width,height))
    plt.imshow(new_img)
    plt.show()
```

In [ ]:
```
A = np.array([[1 , 0, 0.1], [1, 1.5, 0.1], [0.1, 0.3, 1]])
print(A)
color_grader(A, cat)
```

In each of the cases below find a 3×3 matrix A which transforms colors of image pixels as indicated. Use the function `color_grader` to display the resulting image of the balloon.

Exercise 11

The matrix leaves the red component unchanged and sets the other components to 0:

$$A \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \\ 0 \end{bmatrix}$$

In [ ]:
```
A = ...
print("Matrix A:")
print(A)
color_grader(A, cat)
print("The chances of you being killed by this cat is low, but never zero...")
```

Exercise 12

The matrix interchanges the red component with the blue component:

$$A \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} b \\ g \\ r \end{bmatrix}$$

In [ ]:
```
A = ...
```

```
color_grader(A, cat)
```

Exercise 13

The matrix divides all components by 2:

$$A \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} \frac{1}{2}r \\ \frac{1}{2}g \\ \frac{1}{2}b \end{bmatrix}$$

In [ ]:
```
A = ...
color_grader(A, cat)
```

Exercise 14

The matrix replaces all components by their average:

$$A \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} \frac{r+g+b}{3} \\ \frac{r+g+b}{3} \\ \frac{r+g+b}{3} \end{bmatrix}$$

In [ ]:
```
A = ...
color_grader(A, cat)
```

> This document was compiled, gathered and coded by the teaching assistant team and may be used only for educational purposes. The authors would like to thank the many projects and educational material that made their source code freely available on the internet, especially otter-grader that made the generation and sanitization of the notebook easier.