Randomly Generate Matrices of MxN sizes

March 1, 2020

1 Installing Anaconda

This section will walkthrough how to install Anaconda and Python. Anaconda is a free and open-source distribution of Python that focuses on data science, machine learning and data analysis.

Anaconda uses a package management system *conda*. Anaconda can be installed on Windows, Linux and MacOS.

1.1 Download Anaconda

- Download Anaconda

1.2 Installation Documentation:

- Anaconda Installation Guide

1.3 Running Jupyter Notebook

- Jupyter Notebook documentation

1.4 numpy Documentation

- numpy Official Documentation

I'll refer to the relevant sections of the numpy documentation throughout this tutorial as we work through these examples.

1.4.1 Running Jupyter Notebook Cells

Jupyter notebook is a great way to code iteratively. You can run a cell by clicking on the "Run" button above or you can use the shortcut by holding both **"Shift + Enter"** or **"Shift + Return"** if you are on a mac.

1.5 Programming in Python & numpy

Python is a high-level general purpose programming language. It comes with several great built in libraries and built in functions that make it easy to get started programming quickly. The syntax is also very human readable.

When working with matrices using a scientific library like numpy can be helpful. Below we will start writing code for our randomly generated matrix problem.

Here on line 7 is an example of an `import` statement. An `import` statement allows you to bring in other existing code into your program. This existing code is called a library. There is a bit more to importing code but is outside the scope of this project. For now, I will leave this link if you want to learn more about it on your own.

[2]:
```
# I'll be using a lot of comments in this notebook.
# In this first cell we are going to import the libraries to use for
# creating our matrix.

# numpy is imported as 'np' as a convention.

import numpy as np
```

Python lists is one of the most basic data structures in Python. Here is a quick run down on some list basics.

Link to the Python Official Documentation on Lists

[3]:
```
# There are a couple data structures that we will rely on. As you may
# know the basic data structure is a numpy array (vector).
# To create an array we need to leverage an existing Python data structure: list
# Here we are going to use a list (denoted by square brackets) to create an␣
 ↪array.

# Creating list mylist
mylist = [1,2,3,4]

# Transforming mylist into an array:
myarray = np.array(mylist)

# Reviewing the output of myarray:
print(myarray)

# Still looks like a list but it is really a numpy array. Let's
# take a closer look.
```

```
[1 2 3 4]
```

`type` is a Python built-in (pre-packaged) function. More information about `type` is available here

[4]:
```
# Reviewing the data type of myarray:
print(type(myarray))

# Notice that the array is called an ndarray. This is short for
# n-dimensional array. We can quickly find the shape (number of
# rows vs columns) by using a numpy method.
```

```
<class 'numpy.ndarray'>
```

1.5.1 Documentation for Shape

- https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html

[5]:
```
# This means we have an array with four entries. We will use this later
# when verifying the shape of our randomly generated matrix.
myarray.shape
```

[5]:
```
(4,)
```

2 Generating a Random Number

Relevant Doc: Generating Randomness

numpy provides a random class that generates real or integer numbers for us automatically. Let's take a look:

[6]:
```
# How to generate a single real random number. By default it will
# generate a real number between 0 and 1.

np.random.random()
```

[6]:
```
0.5056698802908643
```

[7]:
```
# How to generate a single integer number. Here, the input determines
# what kind of data is returned to us.

# If we enter one digit it will randomly select an integer from 0 to
# one less the provided number. For example, if we enter 3 into our
# randint method numpy will generate a number between 0 and 2. This
# means it excludes the number we provided. Everything up to but not
# including the input will be randomly generated.

np.random.randint(3)
```

[7]:
```
1
```

3 Generating a Random Matrix

Relevant doc: Random Integers

This is helpful in explaining the parameters for the randint method.

[8]:
```
# Creating a matrix of shape 1000 x 1000 with randomly generated
# integers from 1 to 9 (included).

data = np.random.randint(low=1, high=10, size=(1000,1000))
```

```
# This randint method is nice because it does the work of creating
# the randomly generated matrix with lower and upper thresholds for
# the element values. Plus, it defines the size of the matrix for you.
```

[9]:
```
# Preview of the data. Too big to display all of it. What jupyter does
# is it gives you a preview of the first few records and then skip
# to the tail (or end) and show you the last few records in the
# matrix.
data
```

[9]:
```
array([[2, 1, 5, …, 6, 7, 2],
       [7, 4, 8, …, 9, 5, 9],
       [3, 8, 2, …, 3, 4, 8],
       …,
       [6, 5, 1, …, 3, 6, 7],
       [2, 4, 4, …, 4, 5, 9],
       [9, 8, 6, …, 1, 7, 2]])
```

[10]:
```
# The shape method returns a tuple of the (rows, columns) for
# the data array.

data.shape
```

[10]:
```
(1000, 1000)
```

[11]:
```
# Checking the maximum value in the matrix. We would expect 9 since
# we defined our high parameter as 10. Recall that it is up to but not
# including the high number.

data.max()
```

[11]:
```
9
```

[12]:
```
# Verifying that the lowest integer in our matrix matches the
# parameters above.

data.min()
```

[12]:
```
1
```

Slicing and indexing arrays is a quick notation for splitting arrays and matrices into smaller parts. Please refer to the documentation for more information.

[13]:
```
# This shows the first two arrays in the data matrix.
data[:2]
```

[13]:
```
array([[2, 1, 5, …, 6, 7, 2],
       [7, 4, 8, …, 9, 5, 9]])
```

4 Finding Inverse of a Matrix

Here, we will use the Linear Algebra class in the numpy library.

Relevant doc: https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

[14]:
```
# Let's create a matrix called A using the randint method.

A = np.random.randint(low=1, high=10, size=(1000,1000))
```

[15]:
```
# Creates the inverse of matrix A.
inv_A = np.linalg.inv(A)
```

[16]:
```
# Let's preview A.
inv_A
```

[16]:
```
array([[-0.00395897,  0.01104517, -0.00735188, …, -0.00420944,
         0.00535848,  0.01468699],
       [-0.00164451, -0.00168302,  0.00707375, …,  0.00842495,
        -0.008797  ,  0.00139026],
       [-0.03033788,  0.00230603, -0.00883095, …, -0.00010514,
        -0.02156271,  0.04427837],
       …,
       [-0.00173637,  0.01069032, -0.01116593, …,  0.00131012,
         0.003028  ,  0.01177586],
       [-0.00459489, -0.01272178, -0.00370096, …,  0.00551177,
        -0.00533818,  0.00112759],
       [ 0.01833599, -0.00069322,  0.0105715 , …,  0.00739902,
         0.00172418, -0.01744882]])
```

[17]:
```
#  Okay, but how do we know that this actually generated the inverse
# of matrix A? We know that the dot product of a matrix and its inverse
# results in an identity matrix.

np.dot(inv_A, A)
```

[17]:
```
array([[ 1.00000000e+00,  7.06656955e-14, -1.09912079e-14, …,
         1.29674049e-13,  1.90958360e-14,  4.37705427e-14],
       [-8.99280650e-14,  1.00000000e+00,  5.88418203e-15, …,
        -4.78506124e-14, -4.37427872e-14, -4.52415883e-14],
       [-1.68975944e-13, -5.04041253e-14,  1.00000000e+00, …,
        -9.32587341e-14, -1.91846539e-13, -1.10578213e-13],
       …,
       [-1.21569421e-14, -5.55111512e-16, -1.46549439e-14, …,
         1.00000000e+00, -6.99440506e-15, -1.11022302e-16],
       [ 5.32907052e-15, -1.77635684e-15,  6.66133815e-15, …,
         2.00950367e-14,  1.00000000e+00,  4.32986980e-15],
       [ 1.82354132e-14, -2.14828155e-14,  1.87072580e-14, …,
```

```
       6.21724894e-15,  1.85962357e-15,  1.00000000e+00]])
```

[18]:
```
# Looks a bit odd but this is because the numbers are very small.
# Let's use round and abs methods to clean this up.

identity_matrix = np.dot(inv_A, A)

# Absolute Value of rounded dot_product
identity_matrix = np.abs(identity_matrix.round())
```

[19]:
```
identity_matrix
```

[19]:
```
array([[1., 0., 0., …, 0., 0., 0.],
       [0., 1., 0., …, 0., 0., 0.],
       [0., 0., 1., …, 0., 0., 0.],
       …,
       [0., 0., 0., …, 1., 0., 0.],
       [0., 0., 0., …, 0., 1., 0.],
       [0., 0., 0., …, 0., 0., 1.]])
```

[25]:
```
# Ok, what about a matrix of size 10,000 x 10,000? We will reuse
# the same code from above but change the size. Recall, the size
# defines the (rows, columns) shape of your matrix.

# For this example, I am being explicit with the parameters.
new_matrix = np.random.randint(low=1, high=10, size=(10000,10000))
```

[21]:
```
new_matrix
```

[21]:
```
array([[5, 1, 6, …, 5, 6, 7],
       [8, 2, 4, …, 4, 4, 5],
       [5, 7, 5, …, 4, 5, 3],
       …,
       [5, 6, 8, …, 9, 5, 6],
       [4, 5, 3, …, 1, 6, 9],
       [2, 2, 5, …, 2, 7, 5]])
```

[22]:
```
new_matrix.shape
```

[22]:
```
(10000, 10000)
```

5 Timing Code Execution

There are a few factors that come into play when timing your code. One method is taking the starting time and subtracting the ending time. The other is using the `%timeit` special function which runs the code several times and takes an average of the execution time. The second method may be more accurate but less practical if you are just looking at how long it takes the code to run.

5.1 Method 1: Using time Module

- time documentation

[31]:
```
# Let's use the time module
import time
```

[38]:
```
# Returns the current time in seconds since the Epoch.
print(time.time())
```

```
1583081506.3349175
```

[39]:
```
# Set starting time to t0
t0 = time.time()

# Run our matrix code
np.random.randint(low=1, high=10, size=(10000,10000))

# set our ending time to t1
t1 = time.time()

# The difference between t1-t0 is the time it took our matrix
# code to execute.
total_time = t1-t0

# Let's print that here.
print("The total time to run was ", total_time, "second(s).")
```

```
The total time to run was  1.4613189697265625 second(s).
```

5.2 Method 2: Using %timeit

`%timeit` only can be run in iPython or Jupyter Notebooks. It is what we call a *magic command* that is provided by the IPython kernel. In other words, `%timeit` would not run in pure Python. Nonetheless, it is great for giving a more accurate execution time of your code. This is usually for determining the performance of your code.

[42]:
```
# Simply add a '%' in front of timeit to run this magic command.

%timeit np.random.randint(low=1, high=10, size=(1000,1000))
```

```
16.4 ms ± 1.43 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In this case we are returned with the time it took to run the underlying code in milliseconds. The `%timeit` output also returns the mean and standard deviation of the runs.