

Programming for Medical Physics

Neil Kirby, Ph.D.

Assistant Professor

The University of Texas Health Science Center San Antonio

Edited for Python by Noah Bice

Lecture 2: Scripting, Loops, and Conditionals

- 2.1 Interpreted vs. Compiled Implementations
- 2.2 Importance of Scripts
- 2.3 Running Python Scripts
- 2.4 Directory Navigation
- 2.5 For Loops
- 2.6 Logical and Conditional Statements
- 2.7 While Loops
- 2.8 Iterative Loop Alternatives
- 2.9 Image Import and Export

2.1 Interpreted vs. Compiled Implementations

From the previous lecture:

*Python is an interpreted language. Your commands are collected in a **script**, which is passed to the Python **interpreter** as a text file with a “.py” extension. The Python interpreter goes through your script and translates your prompts into machine code, which is then executed. The only thing one needs to run Python scripts is therefore a working Python interpreter and some knowledge about how to communicate with the interpreter, i.e. the Python language.*

The business of “translat[ing] your prompts into machine code” can be completed in many ways. The black and white options for translating scripts are **interpreting** and **compiling**. However, most current Python interpreters fall into a gray area. The basic difference between the two methods is that interpreting will execute the code line by line (like reading a script) with precompiled routines, whereas a compiling will first compile the entire code and then execute it. In general, the same computation will run faster with a compiled implementation.

As Python users, you have several options for interpreters. Each interpreter reads your code in a different way and has unique advantages. For the purpose of this course and most of the work you’ll ever do, you’ll use **CPython**. CPython is the interpreter that is downloaded with Anaconda and should

already be installed on your system. CPython can be viewed as both an interpreter and a compiler: it first reads commands from your Python script and compiles them to an “intermediate representation” called **bytecode**, which is then interpreted. This is significantly faster than interpreting your scripts line by line. CPython offers a **foreign function interface (FFI)** with C and other programming languages (hence the name), which allows one to utilize routines written in C. For instance, almost the entire NumPy package is written in C, and “wrapped” with Python for ease of use. Note that CPython is the reference implementation of Python, so almost every Python package you’ll want to use and every Python version will be compatible with it.

IronPython is another popular interpreter. **IronPython** has a built-in FFI with Microsoft’s .NET framework. This means that it can be easily used to build applications for Windows, and it will interact directly with some Microsoft applications like Excel. The treatment planning system RayStation by RaySearch supports scripting in Python and uses the IronPython interpreter. This means that some software libraries, like NumPy, will be incompatible with their Python interface. Also note that this interpreter only supports up to Python version 2.7.

2.2 Importance of Scripts

The previous lecture demonstrated the use of Python from the command line by entering calculations one line at a time. A script is a collection of these calculations in a file (.py) that a Python interpreter will execute. Whether you decide to create a script versus just perform a calculation on the command line is a question of calculation complexity. All computations take some sort of data in and then export another type back out. Sometimes these computations can be performed in a single line of code and other times it takes thousands of lines. As the number of lines increases, the probability that you make a mistake at some point also increases, which might require you to retype the computations over. You will still make these same mistakes in a script, but fixing the mistake in a script might be easy as changing a single character.

Beyond this, a script is also a way of documenting how a calculation is performed. This can be tremendously useful for recreating data or modifying it. For example, when I create figures for a manuscript, I save the scripts I used for this along with the figures.

Another major way to utilize scripts is to store classes and functions for use elsewhere. All the Python functions you use from different packages by importing are taken from scripts. You can store your own objects and actions using scripts and use them in different scripts or the command line. Python projects don’t often have one massive Python file with 5,000 lines. They will have directories and subdirectories filled with custom classes and functions, and the important computations will be performed in simple scripts that import from complex ones.

2.3 Running Python Scripts

You can create a new script in Spyder with the New File button located just under the File tab. This creates a new .py file, which is essentially a text file that contains a set of Python commands.

Once you create a new script and save it, you can run it using the green run button at the top of the script editor window or by pressing F5. Once you have run a script, the output will be displayed in Spyder's console window. Any variables created in your script will be accessible in Spyder's interactive Python prompt. Note that you can also run Python scripts from the Anaconda prompt by navigating to the script's directory and entering the command "python your_script.py".

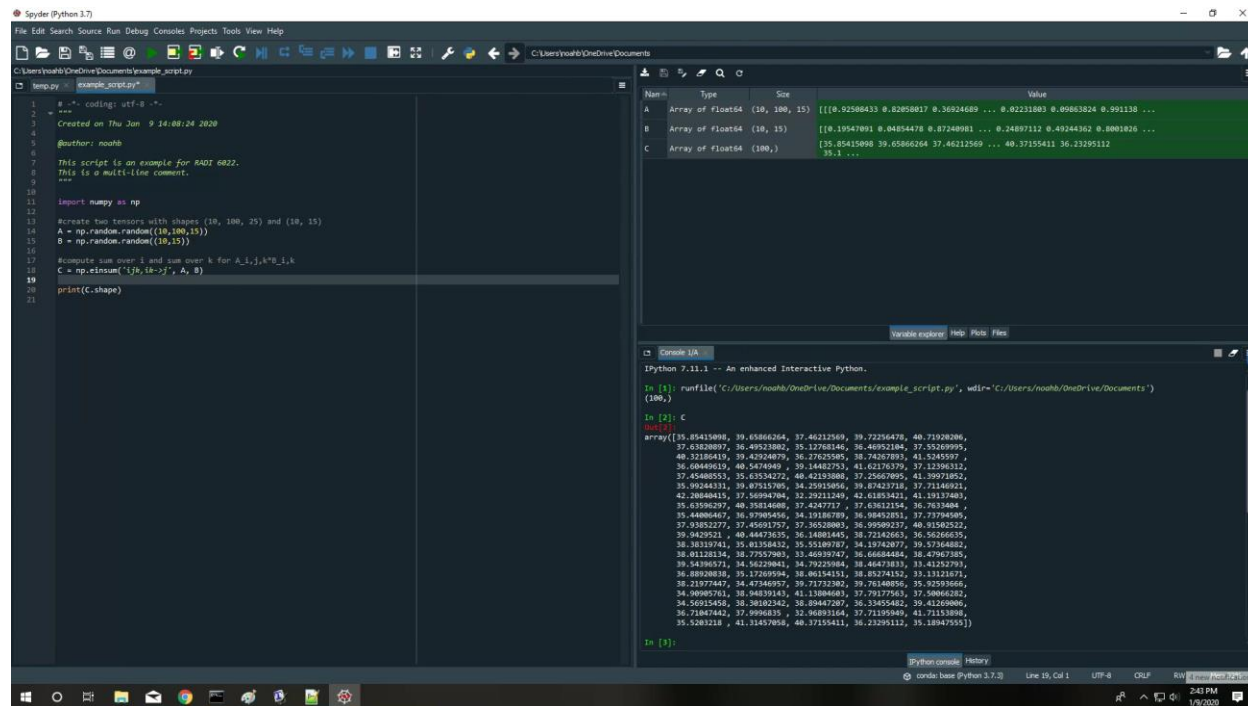


FIG 1. Executing a simple script in Spyder.

Notice in the example script that I have made use of comments. As mentioned, creating scripts is a great way to document how you've performed a calculation. Commenting your code can be just as important. Months (or if you're like me, days) after you have created a script, you will forget what it does, so it's useful to leave a few clues.

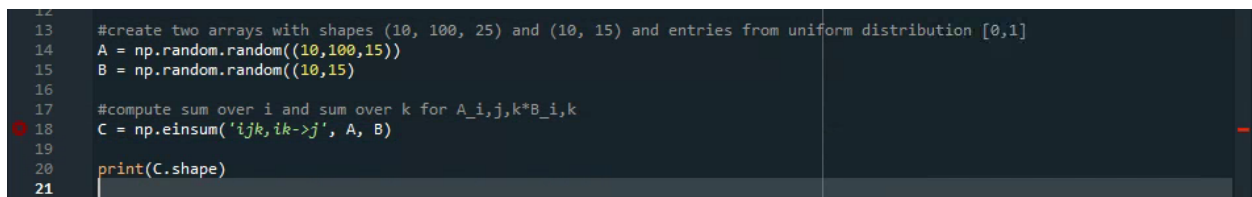
There are two levels of commenting: comments at the top of your code and those dispersed throughout it. The ones at the top are meant to summarize what your code does. These can help you decide whether you are opening the correct code. You can open and close multi-line comments with three quotation marks. You can also make comments throughout the code to help you navigate through the details of the calculation with a "#". Any text following a "#" in a line will be ignored by the interpreter. I will often comment when I first define variables. At the time of creating a script, the variable names will make perfect sense, but months after they might not.

As with many things, there is the need for balance in the amount of commenting. If you leave too little commenting, you'll waste time trying to understand your code. If you leave too much, you'll waste time commenting. As an example, before the last line of the script in Fig. 1, I could have put "display the shape of the output in the console". However, you can read that directly from the code, so there is no need for this comment. Beyond this, the balance in your commenting will change based on the amount of collaboration. If you're sharing code, you should leave more comments than if it is just for your own eyes.

As soon as you begin creating scripts, you'll also begin creating errors. Thence, you need to know how to debug your code. The Spyder scripting interface is fairly efficient at debugging. For example, if I modify the script in Figure 1 to say "B = np.random.random((10,15))" (dropping the last parenthesis), the code displays the following error upon running.

```
File "C:\Users\noahb\OneDrive\Documents\example_script.py", line 18
C = np.einsum('ijk,ik->j', A, B)
^
SyntaxError: invalid syntax
```

The specific error that it shows may not make sense, but it will tell you the line of your code where the error occurs. Note, however, the line it tells you is many times not exactly the same line as the error, as can be seen in Fig. 2. The error actually occurred on line 15, but the code did not run into problems until it tried to run line 18. As can be seen in Fig. 2, you may not even need to run a code to see that there is an error with it. The right hand side of the editor shows red and orange bars for parts of the code with errors and warnings, respectively. For particularly tricky errors, you may need to run the code line by line to find the error. You can do this with the "Run current cell" button to the right of the "Run" button in Spyder.



```
12
13 #create two arrays with shapes (10, 100, 25) and (10, 15) and entries from uniform distribution [0,1]
14 A = np.random.random((10,100,15))
15 B = np.random.random((10,15))
16
17 #compute sum over i and sum over k for A_i,j,k*B_i,k
18 C = np.einsum('ijk,ik->j', A, B)
19
20 print(C.shape)
21
```

FIG. 2. Illustration of a script error.

2.4 Directory Navigation

If you are running Python scripts with a user-friendly IDE like Spyder, you should be able to run all of your programs using the Run button. However, IDEs are not necessary for Python use. If you choose, you can create Python scripts using a generic text editor and save them as .py files. In this case, you can run Python scripts using the Anaconda or operating system command prompt, but you will have to navigate to your scripts. The following commands are useful for navigating with the command prompt:

- cd – change directory. Use "cd .." to go up a directory

- ls – list files in the current directory
- rm – delete a file
- mkdir – create a directory
- rmdir – remove a directory
- cp – copy a directory or file to a new location

You can also change the drive you are working in by typing the letter of the destination drive followed by a colon. Suppose I've created a Python script in notepad and saved it in G:\Users\Noah\Documents\script.py. I might navigate to it and run it with the following commands.

```
> C:\Users\noahb>G:
> G:\>cd users/
> G:\Users>ls
'All Users'  Default 'Default User'  Noah  Public  desktop.ini
> G:\Users>cd noah/documents
> G:\Users\Noah\Documents>python script.py
```

I can create an example directory, copy script.py into it, and subsequently delete what I've created like so:

```
> G:\Users\Noah\Documents>cp script.py example
> G:\Users\Noah\Documents>cd example
> G:\Users\Noah\Documents\example>ls
script.py
> G:\Users\Noah\Documents\example>rm script.py
> G:\Users\Noah\Documents\example>cd ..
> G:\Users\Noah\Documents>rmdir example
```

If for whatever reason you need to interact with the environment from the inside of a script, you can use the `os` module. The `os` module implements the commands we've discussed and many more as Python functions. I find `os.listdir()` to be especially handy for reading data from several files in the same script, as you'll see in an example in Section 2.5.

2.5 For Loops

One of the main purposes of programming is to automate a calculation or data manipulation that you would take you a long time to do by hand. Iterative loops have a similar purpose for the act of coding itself. They loop through a section of code that would otherwise need to be written many times. Let us begin with the example of a **for loop**.

```

23 #Example 1: calculate n!
24 n = 10
25 output = 1.
26
27 for i in range(n):
28     output *= (i + 1)
29
30 print(output) #3628800.0
31

```

FIG. 2. Calculating $n!$ with a for loop.

First, I should say there are easier ways to calculate a factorial. This does, however, serve the purpose of illustration well. Notice that I have made use of the `range()` function, which creates an iterable “list” (not the built-in data type) of numbers from 0 to $n-1$. The `for` loop tells the interpreter to go through the range item by item and evaluate the indented code for every item. Remember that you can create ranges that do not begin at zero with the `range()` function. To use a `for` loop to parse through an object, the object needs to be iterable. Lists, strings, and dictionaries are some examples of iterable objects.

```

>>> range(2,10,3)
[2, 5, 8]
>>> a = 'a string'
>>> for character in a:
...     print(character)
a

```

```

s
t
r
i
n
g

```

```

>>> my_dict = {'key0': 'value0', 'key1': 'value1'}
>>> for key in my_dict:
...     print(key)
key1
key0
>>> for (key, value) in my_dict.items(): #creates a list of “tuples”
...     print(value)
value1
value0

```

The basic syntax when creating a `for` loop is “`for item in object`”, where the object might be a list, string, or dictionary, for example. If the computations are to be completed in separate lines, the object should be followed with a colon and the following lines should begin with an indent or 4 spaces. The Python interpreter will accept either tabs or spaces so long as you use the same convention throughout your script. The official Python style guide recommends using spaces for the sake of making the same code

compatible for between different editors. Most IDEs (including Spyder) will automatically recognize tab presses as 4 spaces.

Note that `for` loops do not absolutely have to take up multiple lines. You can use `for` loops to create lists in a single line like so:

```
>>> x = [i**2 for i in range(3,7)]
>>> x
[9, 16, 25, 36]
```

Aside: The Single Underscore. Note that if you just want some lines of code to be repeated some number of times, you can use “_”.

```
>>> y = ['repeated item' for _ in range(5)]
>>> y
['repeated item', 'repeated item', 'repeated item', 'repeated item', 'repeated item']
```

The single underscore has a special meaning in Python. It acts as a filler to satisfy syntax rules but claims that we do not actually care about the value in that place. Values replaced with an underscore will not be stored in our RAM. You can use the underscore to ignore an output from functions with multiple outputs. For example, suppose we have a function `rectangle_properties()` that takes some variable `our_rectangle` and returns its height, width and area, in that order. If we only care about the rectangle's area, which we'll call “Area”, we can ignore the other outputs with

```
>>> _, _ Area = rectangle_properties(our_rectangle)
```

I want to conclude this section by stating that you will often have many choices for how to implement an idea. Suppose that you have data saved as NumPy arrays in ten files that are alone in an example directory: “example/0.npy, example/1.npy, ... , example/9.npy”. You want to read the data from these files into an array called `my_array`. Following the initialization of the NumPy array

```
>>> import numpy as np
>>> my_array = np.zeros((10), dtype='object')
```

you can read the data using a `for` loop over integers in a range object.

```
>>> for i in range(10):
...     data_i = np.load('./example/' + str(i) + '.npy')
...     my_array[i] = data_i
```

As an alternative, we could use the `os.listdir()` function to retrieve the file names of all the data in the directory `“./example/”`.

```
>>> import os
>>> file_names = os.listdir('example')
>>> file_names
['0.npy', '1.npy', '2.npy', '3.npy', '4.npy', '5.npy', '6.npy', '7.npy', '8.npy', '9.npy']
```

then exploit the built-in `enumerate()` function to write the data to `my_array`. `enumerate()` takes an iterable object and adds a counter, so the items in the list are numbered in the order that they appear. This method is useful for reading data that do not have convenient numerical names.

```
>>> for i, name in enumerate(file_names):
...     my_array[i] = np.load('./example/' + name)
```

Both methods for reading this data are completely viable.

2.6 Logical and Conditional Statements

Logical (or Boolean) statements evaluate to true or false. The most common syntax I use for these statements are the following.

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

```
>>> 2 == 2
True
>>> 2 < 1
False
```

We can expand on these simple tests with the self-explanatory **Boolean operators** `and`, `or`, and `not`.

```
>>> not 2 == 2
False
>>> (2 >= 2) or (2 < 1)
True
>>> (2 >= 2) and (2 < 1)
False
```


Note that `or` is “inclusive,” meaning that if both statements are true, the collective statement is still true. You can also check the equality of non-numeric data types with logical statements.

```
>>> (True != False) or ('a' == 'a')
True
```

We can use the **conditional statements** `if`, `elif`, and `else` to execute code based on whether logical statements evaluate to true or false. This is useful for implementing scripts with different options and for changing the flow of a computation based on the data input. The syntax for conditionals is a section of indented code preceded by a conditional statement, a logic statement, and a colon. The code following an `if` statement will run if the logic statement next to the `if` is true. A modification to the `if` statement can be made by adding an `elif` or `else`. `elif` acts as a secondary if statement and executes code if some special case occurs. If neither the `if` statement nor any `elif` statements evaluate to true, the code section after `else` will run. For example, we can write a block of code that will perform some computation to the variable “x” if the variable “compute” evaluates to true.

```
>>> compute = True
>>> mode = 'multiply'
>>> x = 5
>>> if compute:
...     if mode == 'add':
...         print(x + x)
...     elif mode == 'multiply':
...         print(x * x)
...     else:
...         print('Variable mode is invalid.')
25
```

2.7 While Loops

`For` loops are useful for cases where you know the number of passes that you need at the beginning of a calculation. This is not always the case. A **while loop** can be used for this purpose. These loops require the usage of logical statements.

The syntax for a while loop is a section of indented code preceded by a `while` statement and a logic statement. As long as that logic statement is true, the while loop will continue to run through the code section.

```
>>> while x < 4:
...     print(x)
...     x += 1
0
1
2
3
```

When we are using `while` or `for` loops, the statements `pass`, `break`, and `continue` can be helpful. `pass` is like the single underscore; it acts as a placeholder to satisfy syntax, but nothing is actually executed. The `continue` statement tells the interpreter to end the current iteration of the loop, but to continue iterating. Finally, `break` can be used to terminate the iterative process and move on to the rest of the script.

```
>>> for x in range(5):
...     if x != 3:
...         pass
...     else:
...         print(x)
3

>>> x = 0
>>> while True:
...     if x == 1:
...         print('x equals one!!')
...         x += 1
...         continue
...     print(x)
...     x += 1
...     if x == 4:
...         break
0
x equals one!!
2
3
```

If you're a dummy like me, you will probably occasionally create a `while` loop with no termination condition. When this happens, you can terminate the script by force with the "Stop the current command" button in top right of the Spyder console, or by using Control+C (Windows, Linux). In the following example, I create `while` loop that will never terminate. In this code I make use of the `time` module. `time.sleep()` can be used to delay the execution of your code, and `time.time()` can be used to get the current time in seconds since January 1, 1970, known as **Unix time**.

```
>>> import time
>>> t0 = time.time()
>>> while True:
...     print(time.time() - t0)
...     time.sleep(2)
10.9750001431
12.9760000706
14.9779999256
16.9779999256
```

18.9790000916

Traceback (most recent call last):

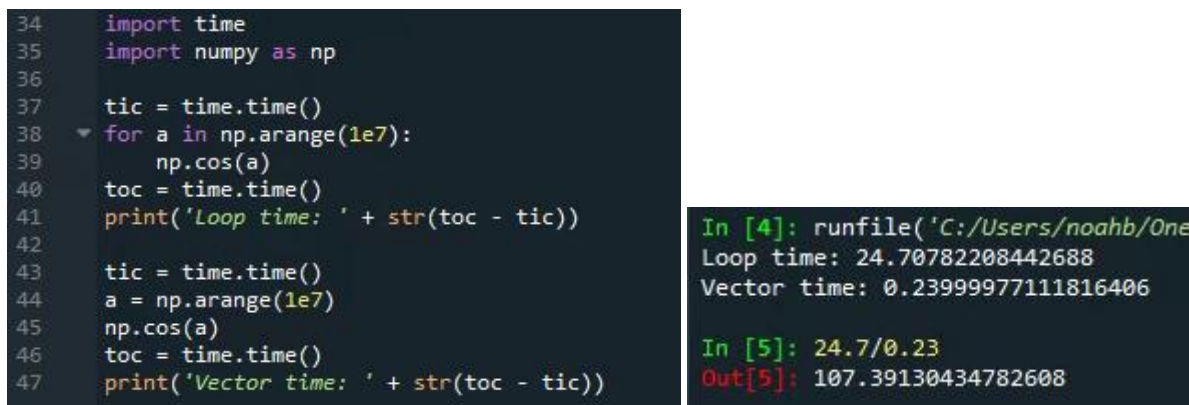
File "<stdin>", line 3, in <module>

KeyboardInterrupt

After 5 iterations of the loop, I used Control+C to terminate the computation.

2.8 Iterative Loop Alternatives

There are many instances where iterative loops are required for calculations. However, there are also many problems where using these loops drastically slows down a calculation. Many functions available in CPython, specifically those offered by NumPy, are vectorized, meaning that they can map vectors of function arguments to vectors of answers. Consider the case of performing a series of cosine calculations for integers from 0 to 9,999,999. We can complete the computation with a [for](#) loop, or with NumPy's vector operations.



The screenshot shows a Jupyter Notebook with two code cells. The first cell (In [4]) uses a for loop to calculate the cosine of integers from 0 to 10,000,000, taking approximately 24.7 seconds. The second cell (In [5]) uses NumPy's vectorized cosine function to calculate the same, taking only 0.23 seconds. The output of the second cell shows the ratio of the two times is approximately 107.

```
34 import time
35 import numpy as np
36
37 tic = time.time()
38 for a in np.arange(1e7):
39     np.cos(a)
40 toc = time.time()
41 print('Loop time: ' + str(toc - tic))
42
43 tic = time.time()
44 a = np.arange(1e7)
45 np.cos(a)
46 toc = time.time()
47 print('Vector time: ' + str(toc - tic))
```

In [4]: runfile('C:/Users/noahb/One
Loop time: 24.70782208442688
Vector time: 0.23999977111816406

In [5]: 24.7/0.23
Out[5]: 107.39130434782608

FIG. 3. Vectorized cosine computation is faster than iterative method.

Performing the calculation with a vector argument reduced the computation time in this case by a factor of 107. Notice that the syntax is also shorter and more convenient.

Another important consideration with vectorization occurs for multiplication and division. NumPy was built to be efficient for matrix calculations. If I have two matrices and I want to multiply them, I can do this with `np.matmul()`.

```
>>> import numpy as np
>>> A = np.array([[1,2],[2,1]])
>>> B = np.array([[0,1],[1,0]])
>>> np.matmul(A,B)
array([[2, 1],
       [1, 2]])
```

If I wanted to perform an element-by-element multiplication so that $C_{i,j} = A_{i,j} * B_{i,j}$, I can do this with the usual multiplication and division syntax. NumPy arrays also support operations with numbers. For example, the operation $A + 3$ will add 3 to every element of the matrix A.

```
>>> A*B
array([[0, 2],
       [2, 0]])
>>> A + 3
array([[4, 5],
       [5, 4]])
```

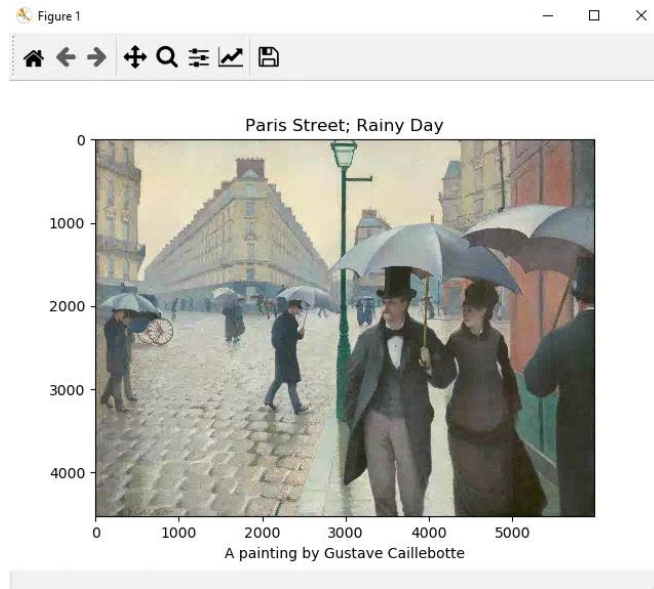
Notice that when I created A, I made a list of lists of integers and called `np.array()` on it. When I divide elements of A by 3, despite the real solutions being non-integers, NumPy rounds new elements to maintain the old integer data type. You can change an array's data type with the NumPy array's `astype()` method.

```
>>> A / 3
array([[0, 0],
       [0, 0]])
>>> A.astype(np.float) / 3.
array([[ 0.33333333,  0.66666667],
       [ 0.66666667,  0.33333333]])
```

2.9 Image Import and Export

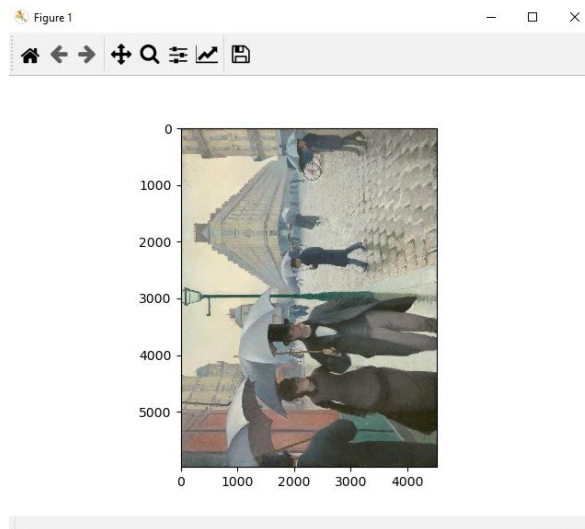
The homework set this week will require that you work with images. For this reason, the data format for images in Python, as well as import and export, is discussed here now. There are various packages with functions for reading and writing image data. One popular package is the **PyPlot** module from **Matplotlib**. **PyPlot** has an `imread()` function that is built on top of the **Pillow** package. This function can be used to read images as NumPy arrays. You simply set a variable equal to this function with the image location as an argument. The image data is then loaded into a NumPy array, which can be displayed with **PyPlot**'s `imshow()` and `show()` functions. `imshow()` creates an `AxesImage` object at some location in memory, which can be further augmented (by the `title()` and `xlabel()` methods) before it is displayed with the `show()` function. Calling the `show()` function will open a separate window.

```
>>> import matplotlib.pyplot as plt
>>> img = plt.imread('./pictures/paris.jpg')
>>> img.shape
(4531, 5982, 3)
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at 0x000002B5320D4080>
>>> plt.title('Paris Street; Rainy Day')
Text(0.5, 1.0, 'Paris Street; Rainy Day')
>>> plt.xlabel('A painting by Gustave Caillebotte')
Text(0.5, 0, 'A painting by Gustave Caillebotte')
>>> plt.show()
```



Note that once an image has been loaded with the `imread()` function, it is a NumPy array, and therefore can be manipulated with NumPy array methods like `swapaxes()`.

```
>>> rotated = img.swapaxes(0,1)
>>> plt.imshow(rotated)
<matplotlib.image.AxesImage object at 0x000002B532E81390>
>>> plt.show()
```



In this image, first two dimensions are the numbers of rows and columns for the image and the third encodes the RGB (red-green-blue) data for a given pixel. More specifically, `img[:, :, 0]`, `img[:, :, 1]`, and `img[:, :, 2]` are the red, green, and blue components of the image. We can tamper with the data for color channels as in the following examples. Note that if the output of `imread()` is read-only, you can simply make a copy of the array with the `copy()` method.

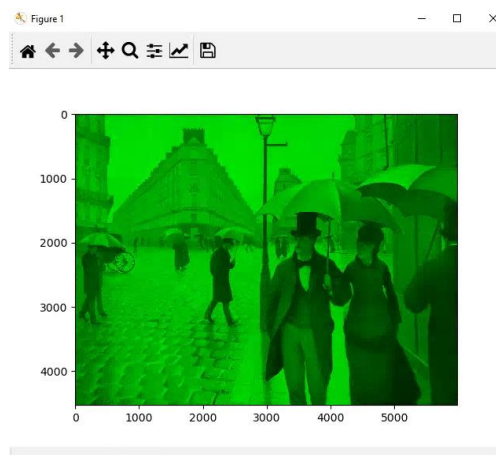
```
>>> img[:, :, 0] = 0
```

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: assignment destination is read-only

```
>>> img = img.copy()
>>> img[:, :, 0] = 0
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at 0x000002B533189320>
>>> plt.show()
```



```
>>> img[:, :, 2] = 0
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at 0x000002B533486B38>
>>> plt.show()
```



`matplotlib.pyplot.imsave()` can then be used to write a NumPy array back to an image. If I wanted to save my green image to a new file in the jpg format I would use the following operation.

```
>>> plt.imsave('green image.jpg', img)
```