CSCI 1430—Python Tutorial

1430 Course Staff

1 Installation

The course uses Python as our language. As software library installation and management can often be tricky, Python supports 'virtual environments' which collect and isolate a set of library packages at specific version numbers. CSCI 1430 has its own virtual environment to reduce configuration issues. You can use this virtual environment on the CS department computers, or you can recreate it on your personal machine. As a development environment, the course supports Visual Studio Code, which is a free cross-platform editor with debugging support.

1.1 Python version numbers

(Spring 2019) The class supports Python 3.5.3, which is the version installed on the CS department machines. Our Python virtual environment uses Python 3.5.3. Our Gradescope autograder uses the same virtual environment on Python 3.5.3.

Python 2.7 is not supported by the class.

If you are experienced with Python and virtual environments, and you can resolve your own versioning issues, then please feel free to use a more current Python version on your personal machine. But, the code that you submit still has to pass on our autograder.

1.2 Virtual Environment

Python with all dependencies needed for this course is already installed on the department machines. You can enable the virtual environment on department machines using the following command:

```
$ source /course/cs1430/cs1430_env/bin/activate
```

Deactivate the environment by using the following command.

```
$ deactivate
```

If you would like to work on your personal machine, there is a requirements.txt file at:

```
/course/cs1430/cs1430_env/requirements.txt
```

which you can use to install all needed libraries using the following command on your local computer (with python 3.5).

```
$ pip install -r requirements.txt
```

Pip is a package manager that allows us to easily install Python modules. In this scenario, the above command neatly reads a list of modules from requirements.txt and installs them all in a virtual environment. If you don't already have pip installed on your computer, you can follow the instructions here: https://pip.pypa.io/en/stable/installing/.

1.3 Common Errors

When installing the virtual environment via `pip`, if you already have Python 2.7 installed, then `pip` may refer to your Python 2.7 installation. Use `pip3` instead to refer to your Python 3.x installation.

```
$ pip3 install -r requirements.txt
```

Note that, if you do not have Python 2.7 installed, then `pip3` may not exist, and simply using `pip` is sufficient.

Likewise, when running `python` with Python 2.7 installed, then this may also refer to Python 2.7 and not Python 3.x. Instead, use the `python3` command:

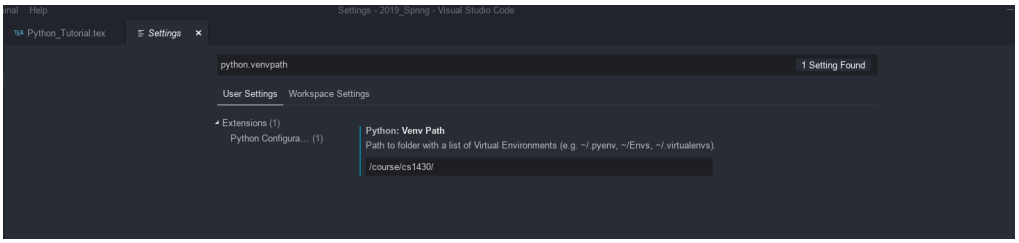
```
$ python3 yourfile.py
```

Another error you may run into, if you're running the virtual environment on Mac OS, is something like: "No module named '_tkinter"'. This seems to be an issue with the matplotlib module backend. If this happens, ensure that you include the following at the top of your Python code, when importing matplotlib.

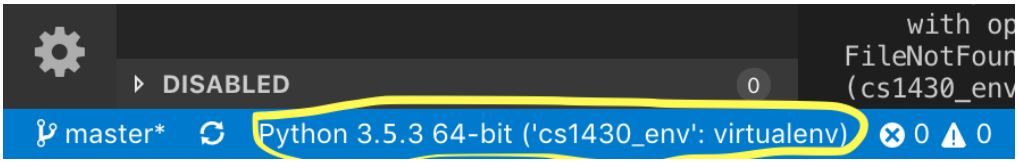
```
import matplotlib
matplotlib.use("TkAgg") # To run the virtual environment on MacOS. Otherwise,
                                    skip!
import matplotlib.pyplot as plt
```

1.4 VS Code

VS Code is a GUI-based text editor and development environment. VS Code is already installed on departmental machines. After starting it, make sure to set "Python.VenvPath" to "/course/cs1430/" in your VS Code settings so that it can correctly detect our custom python virtual environment.



Then, restart vscode. Once done, open a Python file, look to the bottom right, and click the 'Python x.x.x' button. A list of Python environments installed on your machine should appear—select the 'cs1430_env' one. Once done, it should look like this:



For personal laptops, please install VS Code from official Microsoft download page: https://code.visualstudio.com/. Install the official Python extension from within VS Code (menu 'View' then 'Extensions', then search for 'python') to allow code highlighting and debugging. Make sure to set "Python.VenvPath" in settings to the location where you recreated the CSCI 1430 virtual environment.

1.5 Optional: Easier Virtual Environment Activation

You may wish to create an alias for the environment activation command to make it easier to type and remember. To do this, open your `.bashrc` file (located in your home directory) and enter the following at the bottom:

```
alias cs1430_env="source /course/cs1430/cs1430_env/bin/activate"
```

Note that the `.bashrc` file may be hidden from the file explorer depending on your settings, but can still be opened with a command-line text editor like `nano`. The alias will not take effect until you restart (close and open) the terminal.

2 Official Python Introduction

If you are new to Python, we recommend starting with official introduction to Python programming:
https://docs.python.org/3/tutorial/

In general, Python's documentation is excellent, so use it liberally.:
https://docs.python.org/3/index.html

The rest of this document contains concepts we will assume you know. Please become familiar with them, try them out, and let us know if you have any questions.

3 NumPy Introduction

NumPy makes manipulating matrices and vectors in python significantly easier and faster. If you are new to NumPy, we recommend starting with this introduction to programming with NumPy:
https://engineering.ucsb.edu/ shell/che210d/numpy.pdf

4 Syntax

4.1 Whitespace

Please be aware of whitespace rules! Python doesn't require brackets to separate methods and classes; control flow is instead declared with indentation. The following code sample will not work, due to the indentation error in the if statement code block.

```
x = 1
y = 2
if x < y:
x=3 # indentation error!
```

but this one will:

```
x = 1
y = 2
if x < y:
    x=3 # Fixed by adding whitespace!
```

Indentation level is usually either indicated by a tab, or by 4 spaces. We can debate about which is better, but all that really matters is that your choice is consistent.

5 Arrays

5.1 Python Lists vs. NumPy Arrays

Python lists and NumPy arrays are two different data structures. A Python list can contain multiple data types, whereas all elements of a NumPy array must be the same data type. NumPy arrays must be pre-allocated, whereas Python lists do not need pre-allocation. NumPy arrays can be used in vector or matrix multiplication and other linear algebra operations, whereas Python lists cannot. Note that you can convert from Python lists to NumPy arrays and vice versa, but we recommend doing so as little as possible since doing so incurs a computational overhead.

5.2 Indexing

To access or modify the first element of an array, the element at position 0 in the array is called. To see how this works, try the following exercise. In a Python interpreter, create a 5-element array A, with integers 1 through 5 as follows:

```
import numpy as np
A = np.arange(1, 6)
```

A should now be equal to the array $[1, 2, 3, 4, 5]$. Now, type $A[0]$ into the command window and press enter. Note that 1 is the return value. Now try modifying the first element of the array by setting its value to 5; type $A[1] = 5$ into the command window and press enter. The value of the array should now be $[5, 2, 3, 4, 5]$.

6 Reading and Displaying Images

The *skimage.io.imread* function is used to load images. The *matplotlib.pyplot.imshow* function takes in a NumPy array representation of an image as a parameter and displays the image.

6.1 Types

There is no type declaration in Python; the language handles this automatically. However, NumPy arrays *do* have types listed here, and it is critical to understand how this works to use various image processing functions. Try downloading an image from the internet and reading it like so:

```
from skimage import io
image = io.imread('yourimage.jpg')
```

The variable `image` now contains the color information of 'yourimage.jpg' within a numpy array. If your chosen image has color, this array will be of size (height, width, 3), the 3 representing the 3 color channels. Try printing out one of the values in the array (ex: print(image[200,35,1]). You should notice that it consists of integers from 0 to 255.

At times, you will want to alter the image in ways such that some of the entries become non-integer values, and thus you will want to convert the image to floating point format by using the numpy function. You might find it tempting to cast types using *astype*. However, astype() violates typical assumptions made *in other functions* about data type ranges, and this can lead to confusion when using built-in scikit-image processing functions.

For example, when using `matplotlib.pyplot.imshow`, Matplotlib assumes that the image pixel values are between 0 and 255 if the image is in integer format; if the image is in floating-point format, it is assumed that the pixel values are in between 0 and 1. Thus, as a general rule, when you convert an image to floating-point format, you should normalize the image so that all its entries are between 0 and 1; this helps prevent potential confusion.

This is easiest done by using particular scikit-image functions that will convert safely between datatypes: *img_as_float32* to go from 8-bit unsigned byte [0,255] to 32-bit floating point; this function converts an array to floating-point format and then normalizes the array so that all of its values are between 0 and 1. OR, use *img_as_ubyte* to go the other way: from a floating point [0,1] image back to an 8-bit unsigned byte [0,255]. For instance, to convert just before saving an image to disk.

You can read more about this issue here.

To illustrate these concepts, try using these functions yourself. First, try to cast the image that you read from a file to floating-point format using `astype()` and displaying it as follows.

```
import matplotlib
matplotlib.use("TkAgg") #You'll need this line if you're running the virtual
                                    environment on MacOS. Otherwise, feel
                                    free to skip it!
import matplotlib.pyplot as plt
import numpy as np
from skimage import io
image = io.imread('myimage.jpg')
image = image.astype(np.float32)
plt.imshow(image)
plt.show()
```

The displayed image should look very strange; it should hardly resemble an image. This is because it has not been normalized. Now, try converting the image to floating-point format and normalize it from 0 to 1; then display the image:

```
from skimage import img_as_float
image = io.imread('myimage.jpg')
floatImage = img_as_float32(image)
plt.imshow(floatImage)
plt.show()
```

The displayed image should now look normal. Also, note that if you examine any of the entries in the floatImage array, you should find that all entries have values between 0 and 1.

If ever you want to know the type of a numpy array, you can look at the 'dtype' variable of the array, e.g., `print(a.dtype)`.

Note: The `img_as_float` function will covert to a double-wide 64-bit floating point number (also called a float64). scikit-image provides the function `img_as_float32` to convert to the more efficient single-wide 32-bit floating point number (float32) format.

6.2 Multidimensional Arrays / Matrices

Multidimensional arrays in Numpy are an extension of the two-dimensional matrix. One of the most common usages of multidimensional arrays in computer vision is to represent images with multiple channels. For instance, an RGB image has three channels, and can be represented as a 3-D array. Each of these channels can be accessed independently. Let us create an RGB image. To begin, let us create a 300x400x3 array and initialize it to zeros. This can be done as follows:

```
import numpy as np
image = np.zeros((300, 400, 3))
```

Now, we assign a mid red to the first hundred columns and a bright red to the following hundred columns:

```
image[:,0:100,0] = 0.5 # 'half' red
image[:,100:200,0] = 1 # 'full' red
```

The colon ':' indexes all elements in a particular dimension. Finally, we can assign green randomly to the first 100 rows:

```
image[0:100,:,2] = np.random.randint(2, size=(100,400))
# choose random integer in range [0,2)
```

To view the image, type the following into the command window:

```
import matplotlib.pyplot as plt
plt.imshow(image)
plt.show()
```

6.3 Color images vs. Grayscale

Color images are often built of several stacked color channels, each of them representing value levels of the given channel. For example, RGB images are composed of three independent channels for red, green and blue primary color components. In contrast, a grayscale image (aka black and white image) is one in which the value of each pixel is a single sample, that is, it carries only intensity information.

In Python, it is easy to convert an RGB image to grayscale. This can be achieved using scikit-image's *rgb2gray* function.

We can also access individual color channels of a color image. This is illustrated in the code snippet below.

```
# Read in original RGB image.
import numpy as np
from skimage import io, color
import matplotlib.pyplot as plt
rgbImage = io.imread('yourimage.jpg')
(m,n,o) = rgbImage.shape
# Extract color channels.
```

```
redChannel = rgbImage[:,:,0] # Red channel
greenChannel = rgbImage[:,:,1] # Green channel
blueChannel = rgbImage[:,:,2] # Blue channel
# Create an all black channel.
allBlack = np.zeros((m, n), dtype=np.uint8)
# Create color versions of the individual color channels.
justRed = np.stack((redChannel, allBlack, allBlack), axis=2)
justGreen = np.stack((allBlack, greenChannel, allBlack),axis=2)
justBlue = np.stack((allBlack, allBlack, blueChannel),axis=2)
# Recombine the individual color channels to create the original RGB image
                                        again.
recombinedRGBImage = np.stack(( redChannel, greenChannel, blueChannel),axis=2)
plt.imshow(recominedRGBImage)
plt.show()
```

Try to view the various results using *imshow*. You may have to call matplotlib.pyplot.imshow(image) first, followed by matplotlib.pyplot.show(). See the above code block for an example.

7 Performance Improvements

Since NumPy is an extension for Python that is written in C, NumPy operations are faster than their corresponding Python equivalents. For example, performing matrix multiplication of two NumPy arrays is faster than iterating through two Python lists representing arrays and multiplying the correct elements. As such, we recommend doing as much of your calculations in NumPy as possible. It is best to avoid using for loops whenever possible; one can attain significant performance improvements through vectorization and logical indexing.

7.1 Pre-allocation

Numpy does not support dynamic array allocation; it is necessary to allocate space for an array before making assignments. For example, suppose you want to create a 10 element array such that every element is the integer 5. This could be done as follows:

```
import numpy as np
A = np.zeros(10)
for i in range(10):
    A[i]=5
```

As we can see, the array A must be initialized with 10 elements before we can change those elements. To increase the size of the array, we can use methods like numpy.append().

```
import numpy as np
A = np.array([5]) # create an array that is too small...
for i in range(9):
    A = np.append(A, [5]) # ...that we resize every time
```

However, when using resizing methods like append() instead of pre-allocation, elements need to be re-copied every time the size of the array is increased. For a small list like this, the impact is not noticeable, but for long lists it will quickly become a bottleneck. Thus, you should always pre-allocate if possible.

7.2 Vectors as function parameters

Most Numpy functions support passing vectors or matrices as parameters. This prevents you having to apply the function to individual elements as a way of improving performance. It is best illustrated with a few examples: Suppose you have a 10-element array A. You want to take the sine of each element and store the results in another array B. A naive method would use a for loop as follows:

```
import numpy as np
B=np.zeros(10)
for i in range(10):
    B[i]=np.sin(A[i])
```

The same operation can be accomplished as follows:

```
B=np.sin(A)
```

Similar operations can be completed if one wishes to raise every element in A to a certain power. For example, suppose we want to square every element in A and store the result in B. This can be done as follows:

```
import numpy as np
B=np.power(A,2)
```

7.3 Logical Indexing

Suppose we have an $m \times n$ 2D array, and we want to set every element in the array that has a value greater than 100 to 255. This can be done as follows with a for loop:

```
import numpy as np
import matplotlib.pyplot as plt
m = 400
n = 400
A = np.random.randint( 255, size=(m,n))
for i in range(m):
    for j in range(n):
        if A[i,j] > 100:
            A[i,j] = 255
plt.imshow( A );
plt.show()
```

A more efficient method uses logical indexing:

```
B = A > 100
A[B] = 255
```

B is now a binary logical array, where for all i, j, $B[i][j] = 1$ if and only if $A[i][j]) > 100$; otherwise, $B[i][j] = 0$. Then we do the following: $A[B] = 255$. An element-wise assignment is then performed; the result of A the same as it would be using the for loop method. A appears brighter, as more pixels are set to their maximum value. Of course, removing the temporary variable $A[A > 100] = 255$ will work just as well.

Note: Logical indexing also works across multi-channel images! Even though we can split out each of our color channels and index into them individually, we can also logically index directly into the three-channel RGB image.

7.4 Evaluating Performance

We can evaluate time performance using the Python package time. It is used as follows:

```
import time
start = time.time()
# Perform some operation
end = time.time()
print(end - start)
```

The elapsed time between the variables start and end is then printed out. You should try doing several of the examples above, and note the performance differences between using for loops and using the more efficient methods.

8 Debugging

There are a few debuggers that exist for Python. We support using VS Code with the Python extension. To use this debugger, download the Python extension from the following page: https://marketplace.visualstudio.com/items?itemName=ms-python.python. This extension also provides linting (syntax checking) and autocomplete.

Please see the following VS Code page for information on how to enter the debugger, and how to set and navigate breakpoints: VS Code Debugger

VS Code should automatically detect the configuration and run properly when you press "F5" when editing your main file. If it does not work, check the version of python that VS Code has selected by looking at the bottom left corner. It should be the "cs1430_env" virtualenvironment. You can change the version by clicking on version displayed in the bottom left and selecting a different version using the drop down menu. If this setup does not work, please come to TA hours.

9 Project 0 questions

Made it this far? Good. Now, please attempt the three Project 0 questions on the course Webpage.