CMPSCI 390A Python Instructions

Abstract

This document describes the recommended way to setup and use Python for the course. This document also contains links to additional Python resources.

1 Python Setup

It is possible to have many different versions of Python and Python packages installed on one computer. To make sure the code you develop for this course will be compatible with our grading machines, we will leverage Python virtual environments. A Python virtual environment is a Python installation separate from any other Python instance on your system. These virtual environments make it possible to have multiple versions of Python on a single computer, each with its own set of packages and dependencies. To set up this functionality, we will be using the Anaconda distribution of Python and its package manager. You can download Miniconda (a smaller version of Anaconda) from this link. You should download version 3.8 for 64-bit systems (unless your computer has a 32-bit CPU). Follow their instructions to install and set up your environment variables. We recommend using the default paths during the installation.

After installing Anaconda, you should open a terminal (or the Anaconda Prompt from the Start menu if you are using windows) and type `conda list`. This command will show the currently installed Anaconda packages. If you get an error saying `command 'conda' not recognized`, you probably need to reload your terminal for the path to be updated. If this error persists, your path is not set correctly, and you will need to update your path as specified in the installation instructions. Now with a terminal open and Anaconda correctly set up, we can create a new virtual environment. Execute the following command to create the virtual environment we will use for this course.

```
conda create --name cs390a python=3.8 numpy matplotlib
```

This command creates an environment called "cs390a" that runs Python version 3.8 and installs two packages: NumPy and Matplotlib. NumPy is a powerful library for easily manipulating arrays in Python. Matplotlib is a standard plotting package that we will use throughout the course. This environment can be activated with the command

```
conda activate cs390a
```

and then deactivated with the command

```
conda deactivate
```

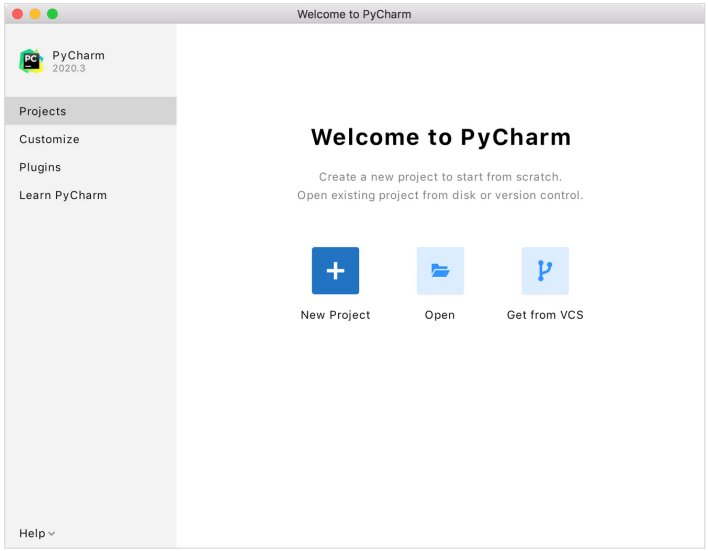
Once activated, any Python command will use the Python version and packages you installed in this environment. If you need to install a new package (we will tell you if you do), you can use the command

```
conda install <package_name>
```

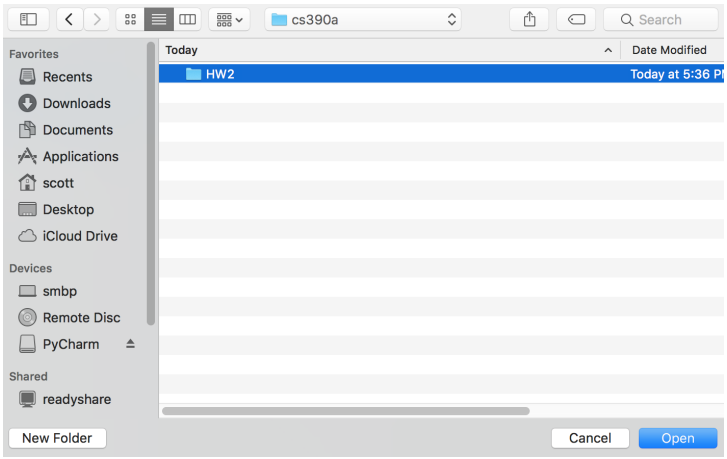
These commands, along with other useful conda commands, can be found in the Anaconda documentation. Note that you should not need to use sudo (on Mac or Linux) to install anything for this course, and we advise against it. You can now close the terminal/Anaconda Prompt.

2 IDE Setup

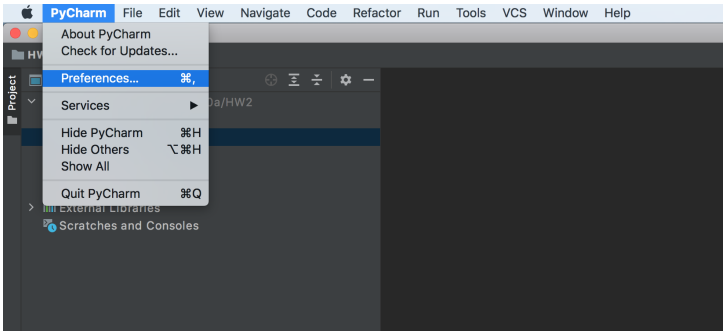
You may also find it helpful to use an IDE to develop and debug your programs. While there are many good IDEs, we recommend and will minimally support the use of PyCharm. You can use either the free community edition or sign up for a free professional license as a student. Download and install PyCharm now. Next unpack the `cs390a_hw.zip` file. Inside this archive, there is a folder called `HW2`; move this folder to where ever you want to store and edit your code for this course. Once installed, run PyCharm for the first time following these instructions. You will get a Welcome window similar to the one below.



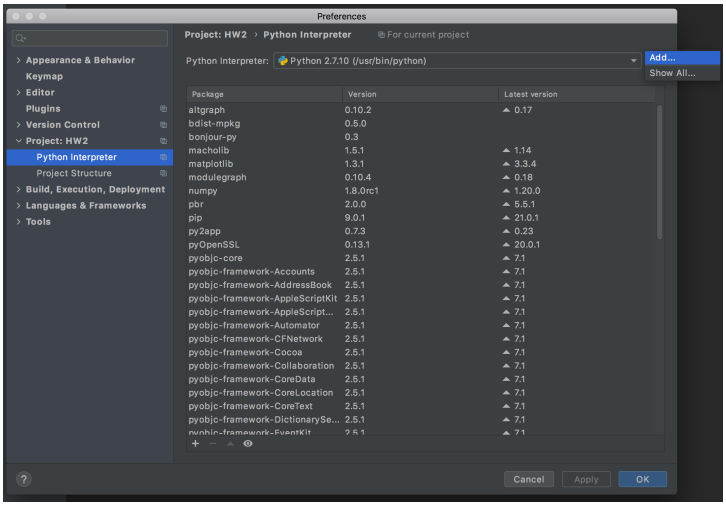
We will now create a project for homework 2. Select Open in the window and navigate to where you put the `HW2` directory and open that directory in PyCharm.



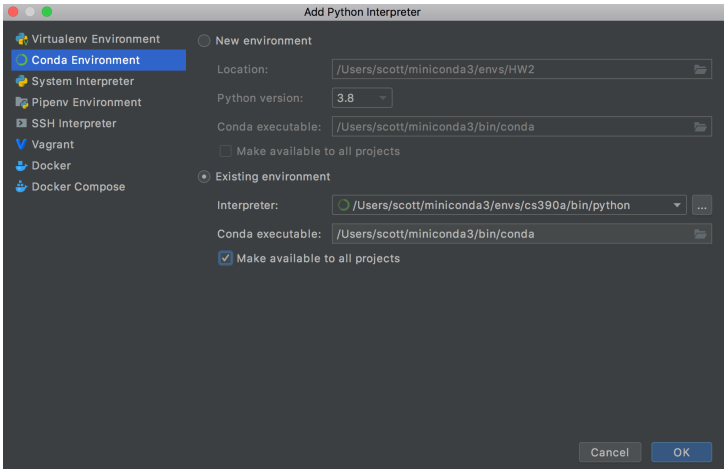
You now have a project created for homework 2 and can repeat this step for future assignments. Next, we will set up the Python interpreter to use the Anaconda version we installed above. From the PyCharm menu, select Preferences/Settings.



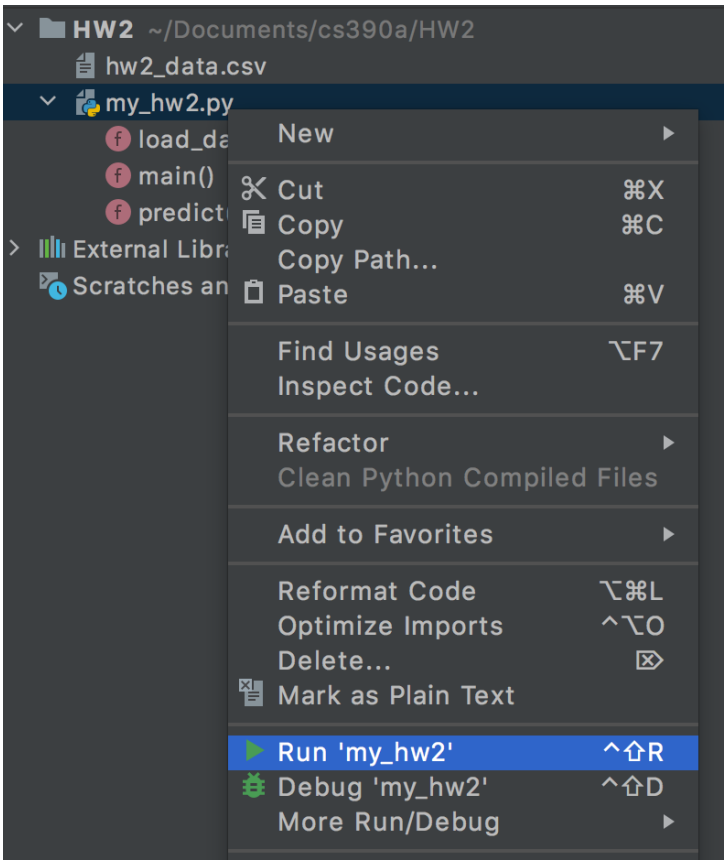
On the left-hand side, navigate to the Project: HW2 item, expand that list and select Python Interpreter. Then click the gear icon next to the interpreter path line and select add.



Then select the Conda Environment option on the left-hand side. Then choose the Existing environment option. You need to enter/select the python file in the Conda virtual environment for the interpreter box. This path is most likely `/path/to/miniconda3/envs/cs390a/bin/python`. Next, check the box "Make available to all projects", so you can easily select this environment for future homework assignments. Then click OK. Then click OK again in the Preferences/Settings menu to apply these changes and close the window.



You are now set up and can run Python files in this project. To run the file `my_hw2.py`, first, open the file by double-clicking it. Then right-click the file and select Run.



If successful, you should see print statements appear in the bottom half of the window.

Next time, you can click the green arrow or the green debug symbol in the top right corner of PyCharm.

To use the debugger, you first need to set a breakpoint. Click just right of the line number 21 to set a breakpoint. If there are no line numbers, this line corresponds to the line in the image below, and you can click where the red dot is. To remove the breakpoint, click it again.

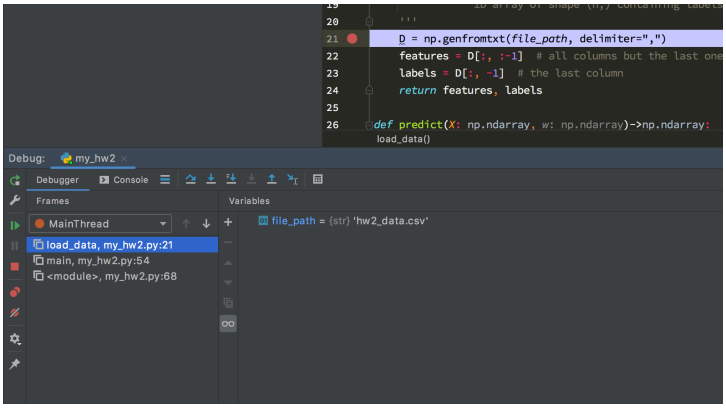
```python
def load_data(file_path: str)->Tuple[np.ndarray, np.ndarray]:
    '''
    This function loads and parses text file separated by a ',' character and
    returns a data set as two arrays, an array of features, and an array of labels.

    Parameters
    ----------
    file_path : str
                path to the file containing the data set

    Returns
    -------
    features : ndarray
                2D array of shape (n,m) containing features for the data set
    labels : ndarray
                1D array of shape (n,) containing labels for the data set
    '''
    D = np.genfromtxt(file_path, delimiter=",")
    features = D[:, :-1]  # all columns but the last one
    labels = D[:, -1]  # the last column
    return features, labels

def predict(X: np.ndarray, w: np.ndarray)->np.ndarray:
    '''
```

With the breakpoint set, click the green debug symbol in the top right corner to run the file in debugging mode. The program will run until it hits the line where the breakpoint is. It will also open up the "debug" view, where the line that is about to be executed is highlighted, and a control window pops up in the bottom half of the screen.



The left-hand side has the main control buttons. The green bar with an arrow is the continue button that will execute the code until the next breakpoint is hit. The red square aborts the program. The Frames section shows the program's current stack trace, and you can jump to different positions in the stack. The variables section shows you the current variables available and their values. The *step into* and *step over* buttons are the ones with blue arrows. You can now click the continue button and remove the breakpoint. You are now set up to finish the first homework.

3 Python Resources

If you are unfamiliar with Python, there are many tutorials on the internet that cover the syntax. Python.org has many tutorials covering the basics of Python, see `https://docs.python.org/3/tutorial/`. One of the most common errors in writing Python code is in handling indents. Since Python has no curly braces or end statements to enclose functions, loops, and other code segments, indentation is the only way to signify which block a line of code belongs to. If for indentation tab characters and spaces are mixed, then the program might silently fail by running code differently than you intended. The PEP8 standard[1] for Python specifies that indentation should be four spaces and use the space character instead of a tab. Most editors can automatically convert tabs to spaces for a specified tab width. This error usually only occurs when switching between editors without reformatting the code.

You may have noticed that in the function definitions for homework 2, each variable has a : then a *type* specified. These types are really type *hints* meaning that an IDE or editor uses them to reason about the code and possibly identify errors before it is run. Python does not have typed variables, and no checks on types will be performed unless you code for them.

4 NumPy Arrays

Manipulating lists, arrays, vectors, and matrices, will be a common component in the programming assignments for this course. NumPy is a powerful package for manipulating arrays and performing linear algebra (e.g., dot products, matrix-

[1] `https://www.python.org/dev/peps/pep-0008/`

vector multiplication). Most importantly, NumPy makes it easy to index and slice (select sections) arrays. In this section, we provide a small overview of using NumPy arrays. To use the NumPy package, you can add the following line to the top of the file.

```
import numpy as np
```

This command imports the NumPy package and aliases it to use the abbreviation `np`, this way anytime you want to call a function in the NumPy package, you only have to write `np.<function_name>`. There are many ways to create and interact with NumPy arrays. Here we list a few common ones.

```
### 1D Arrays ###

# creation
x = [1,2,3,4]  # a normal list in Python
x = np.array(x)  # x is now a numpy array.

n = 3
x = np.zeros(n)  # creates an array of length n containing all zeros
x = np.ones(n)  # creates an array of length n containing all ones
x = np.arange(n)  # creates an array containing the integers 0,1,...,(n-1)

# typing the elements
x= np.ones(n, dtype=np.int64) # creates an array of ones that are all 64-bit integers
# dtype is an optional argument on most NumPy functions that create arrays
# default type is np.float64
# other useful types: np.int32, np.float32, np.bool
# Python types of int, float, and bool are also acceptable

# indexing
x = np.array([1,2,3,4])
x[0]  # first element (returns 1)
x[3]  # last element returns 4
x[-1]  # also last element
x[-2]  # second to last element
x[1:4] # gets an array containing the elements at indexes 1,2,3
x[1:] # gets an array containing the elements starting at index 1
x[0:-1]  # all but the last element
x[:-1]  # the 0 index is assumed
x[:]  # all elements

### 2D Arrays ###

# creation

n,m = 4,3
x = np.zeros((n,m))  # creates a 2D array with n rows and m columns

x.shape  # returns the dimensions/shape of the array as a tuple, in this case (4,3)
x.shape[0]  # gets the number of rows
x.shape[1]  # gets the number of columns
x.size  # gets the total number of elements in the array

x[0, 1]  # selects the element in the first row and second column
x[0, :]  # selects the first row
x[:, 2]  # selects the third column
x[:2, :]  # selects the first two rows
x[:, :2]  # selects the first two columns
x[-2:, :2]  # selects the last two rows and first two columns
x[:, :]  # selects all rows and columns

a = np.ones(m)
```

```
x[0,:] = a  # sets the first row to be equal to array a
x[:,0] = 1  # sets the first row to be equal to one

### element-wise operations ###

a = np.arange(3)
b = np.array([2,0,3])
c = a + b  # creates an array where each element in a is added by the element in b
# c = [0+2, 1+2, 2+3]
a - b  # element wise subtraction
a * b  # element wise multiplication
a / b  # element wise division
# vectors must be same size
# this also works for matrices of the same size
```