Lab01

January 25, 2022

1 Lab 1

After Lab 0, your computer should be all set up for this course. Today we dive into the first concepts that will be the underpinning to everything that we do for the rest of term.

This lab's goals are: * Create a `.gitignore` file * Be able to import data in both `numpy` and `pandas` * Articulate the shape of data, specifying 'variables' and 'observations' * Understand the role of unit testing in our course and in programming development more broadly

1.0.1 Before starting...

Make sure that you have recently `pull`ed `course-materials`.

Just a bit more set-up Create a **sub-directory** under `course-materials` called `student-labs`. Each day after `pull`ing from `course-materials`, then put a copy of the lab folder in `student-labs`.

In `course-materials` on GitHub, you should see a file called `gitignore`. This file tells `git` what to `ignore` which in this case includes a folder called `student-labs`. This means anything in that folder will not create conflicts with the main directory.

Note that if a lab requires more than one file, it will be in its own folder. So you should copy the whole folder from `course-materials/Labs` into `student-labs`.

1.1 Creating `.gitignore`

The `.gitignore` file is a hidden file that tells `git` which files to ignore. In general, you should be creating one for each of your repositories. A few things that you will want to ignore each time include your `.ipynb` checkpoints.

Refering to the help file, the process for creating `.gitignore` file is as follows: 1. From your command line, navigate to the directory for your repository on your local machine 2. Next we use the `touch` command (which is similar to `cd` and `m`) to create the file.
Type `touch .gitignore`. This creates an empty file with that name 3. Now we need to add the files (or rather the file types) that we do not want to commit to git. To do this, we will use `vim`. Refering to this Simple Vim Workflow Example, we proceed as follows: * Type `vim .gitignore` Now you will see an empty file. If you try to type right now, nothing will happen until you hit the **i** key. This is confusing at first. But **i** is a special key in *vim* as we will discuss next. * Enter **insert mode** by typing `i`. Now you can type whatever you want. Since, we want to ignore our *.ipynb* checkpoints, add `.ipynb_checkpoints` to your *.gitignore* file. * Exiting a file in *vim* is not as simple as closing the window. First, we need to get out of *edit/insert* mode and back to command

mode. To do this hit `Esc`. * Now you are in *command* mode. To **save and exit** your file, type `:wq` and then `Enter`.

Unsolicited Advice - You will be doing this procedure a lot. I recommend either 1) creating a section in your machine learning notebook for **Repeated Procedures** and adding notes there about how to do this, 2) writing them on the inside cover of your notebook, or 3) tagging them with comments in your engagement journal.

1.2 Importing the necessary packages

The first coding component of any script or Jupyter Notebook is the list of imported packages. There are a few reasons for this: 1. **Programming reason** - Python executes in order of the given lines. This means that you need to import a package before you use an element from it 2. **Human reason** - Before running a notebook or a script, a new user would want to know immediately if they can run the script or not. Putting the import statements at the top of the file allows your user to check if they can run or not run your file.

Note: Failure to put your import statments at as the first lines of non-commented code will result in an *automatic loss of half of the assignment's total points.*

[ ]:
```
import numpy as np
import pandas as pd
```

Did you just throw an error? Happens all the time.

You might not be in the correct `conda` enviroment. Remember that in Lab 0, we only installed our required packages into one conda environment. So you may need to shut down the `jupyter` kernal, activate the correct environment, and then relaunch your `jupyter notebook`.

A few notes on imports You might be wondering why I imported these two packages this way. Partially this is due to what I've seen others do, but a better reason is the one articulated in 'Python for Data Analysis' on page 90: > … throughout the book, I use the standard NumPy convention of always using `import numpy as np`. You are, of course, welcome to put `from numpy import *` in your code to avoid having to write `np.`, but I advise against making a habit of this. The `numpy` namespace is large and contains a number of functions whose names conflict with built-in Python functions (like `min` and `max`).

You will even notice that the help files for numpy and pandas use these conventions.

1.3 Importing Data

When working with data, we need to first import that data. One can do this either using `pandas` or using `numpy`. In this lab, we will work with both methods for sake of completeness.

1.3.1 Importing with `numpy`

Importing with `numpy` will bring in your data as a `numpy array`. The most straightforward way to do this is using `genfromtxt()`. To learn more about this function, read its help file.

[ ]:
```
ffire_np = np.genfromtxt("forestfires.csv", delimiter=',')
```

Taking this apart piece by piece, let's look at what we just did: 1. `ffire_np` is a variable 2. `np.` tells us to reach into the `numpy` library 3. `genfromtxt` is the specific method that we want to call 4. The first argument is the name of the file, which we have placed within the same directory. 5. The second argument `delimiter=` tells us what gaps to look for between data information. It is safe to use a comma as our separation since `csv` stands for `c`omma `s`eparated `v`aribles.

Let's take a peek at what our variable looks like:

[ ]:
```
print(ffire_np)
```

Is this what we expect to see? Open the datafile using your favorite spreadsheet viewer.

What do you see? Does it match the above?

There is a third argument that we can use with `genfromtxt` to ignore the column headers: `skip_header=1`. In the below code block use `genfromtxt` with three arguments to re-import the `forestfires` data and print the result.

[ ]:
```
# Import forestfire data here
ffire_np =

# Take a look at the result here
```

Again, is this what you expect to see? Compare this with the open datafile in your favorite spreadsheet viewer.

Limitations in `numpy` `numpy` can have a hard time with non-numerical data. When `numpy` doesn't know what to do with cells that are not numbers, it replaces those cells with `nan` or `n`ot `a` `n`umber.

Take a look at both of your outputs, and comparing them again to the open datafile in your favorite spreadsheet viewer. Which rows and colums have `nan`s?

What? Why? This doesn't feel particularly useful given that so much data contains information that is non-numerical. However, a closer examination of what `numpy` stands for makes it a bit clearer why `numpy` does this. With each reference to `numpy`, I've used the `code` formatting, but the name of this package is **NumPy** or **Num**erical **Py**thon. This package is for the fast processing of numerical data leveraging tricks from linear algebra.

`pandas` can offer us a bit more flexibility. Let's try importing data with it.

1.3.2 Importing with `pandas`

`pandas` has more functionality when it comes to data with non-numerical values than `numpy`. While our book doesn't delve into the root of the name `panda`, according to this site, `pandas` is short for `P`ython `Data` `A`nalysis. (Clearly I'm not sure how they got 'pandas' from those three words, so if you would like to instead just imagine a group of cuddly pandas, that's fine with me.)

Returning to how we can import using `pandas`, the most common method that we will use is `read_csv()`. To learn more about this function, read its help file.

[ ]:
```
ffire_pd = pd.read_csv("forestfires.csv", sep=',')
```

Taking this apart piece by piece, let's look at what we just did: 1. `ffire_pd` is again just a variable 2. `pd.` tells us to reach into the `pandas` library 3. `read_csv` is the specific method that we want to call 4. The first argument is the name of the file (just like in `genfromtxt()` from above).
5. The second argument `sep=` is the pandas version of `delimiter=` telling us what gaps to look for between data information. (Why are we using a comma here for denoting the gaps?)

Let's take a peek at what our variable looks like. Note that we're going to do this both using and not using the `print()` method.

[ ]:
```
ffire_pd
```

[ ]:
```
print(ffire_pd)
```

Before we go on:

1. What do you see in both of these views of the output `ffire_pd`?
2. What are the differences between these views?
3. How does `ffire_pd` compare to the datafile in your favorite spreadsheet viewer?
4. How do `ffire_pd` and `ffire_np` differ?

1.3.3 Data Frames and Series

Data types are something that we pay a lot of attention to in computer science. So before moving much further,we should ask ourselves: what kinds of objects did we just create? (How do we check the **type** of an object in python?)

[ ]:
```
# Print the data types for ffire_np and ffire_pd here
print(____(ffire_np))

____(____(_____))
```

We'll turn our attention to the `pandas` version: `ffire_pd`. This variable is a `pandas` ***DataFrame***, which are comprised of ***Series***. DataFrames look like spreadsheets that we are used to seeing, and they can contain data with both numerical and non-numerical values. Each row of the DataFrame is a Series.

Series It is possible to just individual `series` and then compile them into a dataframe. For example, you might want to store information on your drink order at each coffee shop in Northampton, noting the `name` and `size` of your favorite drink using `.Series`:

```
woodstar = pd.Series(['woodstar', 'cafe vanilla', 'med'], index = ['restaurant',
'name', 'size'])
```

Notice that the specific information for my order are in the first part of the `.Series` command and the general information name is in the second part.

Use `.Series` to store your order for the Campus cafe and for Neilson:

[ ]:
```
# Code block for your drink orders
```

Once we have a group of `series`, we want to combine them into **one** data frame. To do this, we

- Create a list of our series: `drinks = [woodstar, campus, neilson]`
- Transform our list into a dataframe: `drinks_df = pd.DataFrame(drinks).set_index('restaurant')`

Copy the above code below and then print `drinks` and `drinks_df`. What do you notice?

[ ]:
```
# Create `drinks` and `drinks_df`


# Print both
```

What happens if you change the input for `.set_index` to `name`?

[ ]:
```
# Code block for you
```

Based on this, what do you think `.set_index` does?

Markdown block for you!

1.4 Shape of Data

Data, like spreadsheets, have a shape. Most data that we will work with in this class will have a shape that can be described by the number of rows and the number of columns. When refering to data, we think of ***Observations*** and ***Variables***. * Each observation is a row. Think of the observations as an object, person, or item that we have a set of information on. * The variables are stored as columns. The variables details the kind of information that we have stored on each observation.

Looking at your dataframe and refering to the data's code book, what are the observations in this dataset? What are the variables? *(Note: A code book is a document that provides details on a dataset.)*

Finding the shape of your data Once you have the data loaded, you can quickly generate a number of facts about your data: 1. The dimensions of your data (i.e. the number of observations and variables) using `.shape` 2. The names of the variables using `.columns` 3. Create a quick glance of your data using `.head`

[ ]:
```
# 1. Using `.shape` on ffire_pd
print(ffire_pd.shape)

#2. Using `.columns` on ffire_pd
var_names = ffire_pd.columns
print(var_names)

# 3. Using `.head` on ffire_pd
glance = ffire_pd.head()
print(glance)
```

Taken these pieces apart: 1. `.shape` tells us the number of observations followed by the number of variables. We call data *tall* if there are substantially more observations than variables. We say data is *wide* if there are substantially more variables than observations. 2. `.columns` gives us the list of *variables* which `pandas` considers as an index. 3. `.head` gives us a snippet of the data. The

default is 5 rows, but we can actually specify the number of rows that we want to see by supplying a number between the parentheses.

Your Turn Check out fivethirtyeight's data repository on GitHub and pick a dataset of interest. Use python to import the data and describe key features about the data set. Practice your machine learning communication skills and write a short paragraph about your data based on the code that you ran.

```
[ ]: # Code Block for you!
```

(Markdown block for you!)

1.5 Powerhouse packages

As `python` and data science have grown and evolved, several packages – including `numpy`, `pandas`, `scipy`, `matplotlib` – as critical to the practice of machine learning. In this course, we will use these packages, but with an eye towards deep understand of each method that we employ.

This course has twin themes: **carpentry** and **creativity**. Most of the assignments in the course focus on the former in service of the latter. Think of this course as a kind of cookng class where we will spend a considerable amount of time on each ingredient and the most basic of recipes, with the ultimate goal of creating a glorious meal mixing and blending the ingredients in unexpected ways with each other and with new ingredients.

Carpentry: We are going to focus on the interior elements of classic machine learning algorithms, building each one from scratch. We will compare our results to already optimized versions in packages like `scikit-learn`. The goal of carpentry is *deep* understanding of each algorithm. The course ***Homework*** and ***Labs*** are designed to build your toolkit.

Creativity: While the majority of class time and homework assignments are focused on carpentry, we are building towards you asking "what if I applied X to Y in this way?" and "what happens if we remove this assumption from Z?" The goal of creativity is applying these algorithms to problems and data that you care about in *new* and *exciting* ways. The course ***Projects*** are designed to engage you creatively.

1.6 Unit Testing: When are we right?

We will be using unit testing in our course a lot. In fact, each homework assignment will have unit tests that you can use (or not). Quoting McFee and Kell:

> Tests make your life easier and give you confidence in your code. Software is complicated, and tests allow you to be sure that you've not broken your project when you change something.
>
> Writing tests slows you down in the short term, but speeds things up in the long term. Further, as someone who is writing scientific code, tests make sure that your work can be easily reproduced.
>
> Philosophically, each test should test one thing. This makes it easier to ensure that we can see what has broken when a test breaks.

In the next few blocks, we will go over: 1. Using tests 2. Writing tests

Unit tests in our course For many of our course's homework assignments, you will be given tests. You can use these tests to check your code and see if it is working how I expect it to. In the case of projects, you may be asked to design your own tests. Even if you are not explicitly asked to design a test, it is strongly recommended that you get in the habit of building tests.

A few notes: 1. When grading your homework, The first thing I will do is check to see if the tests pass. 2. I personally am just getting in the habit of writing and using unit-tests.

1.6.1 Using tests

We will use `pytest` to test our work. In this course, the convention for these files will to have a file name that begins with `test`. In fact, according to this post, `pytest` expects our test files to either use this preamble or end with `_test.py`. In this section, I am again editing from Part 5 of McFee and Kell's tutorial:

To run the tests, from your command line, type: `pytest -v`

Note: 1. My test files will assume that you have followed all directions in the homework assignment, including and especially naming files correctly. Please pay careful attention to these details. 2. You cannot test functions inside a `jupyter notebook`. You will need to create a `.py` file.

Your turn I have written a set of 3 tests for this lab in a file called `test_Lab1.py`. Open this file in your favorite code editor and then run the tests from the command line. What happens?

This space is intentionally left blank ####
####
####
####
####
####

They failed?! Oh no! This was intentional. Tests fail a lot. It's ok.

Let's figure out what is going on. (Remember that **I** wrote these tests. You have done nothing wrong.)

Read the error message that you got. What seems to be the issue?

Test your theory by opening `labone.py`. Make **2** minor changes to this file and run the tests again. (When they pass, take a moment to celebrate. Happy dances are encouraged.)

Note about imports in testing You will notice that we have three odd lines in the test file: 1. `import os` 2. `path, _ = os.path.split(os.path.abspath(__file__))` 3. `fname = os.path.join(path,"forestfires.csv")`

When testing our files, GitHub Actions (we'll learn more about this next time) is building a virtual machine to test our files. We need the paths to be such that GitHub Actions can build them. The first line imports the `os` package that allows us to do path manipulations. The second line extracts the path for the test file and splits it such that the file name (in this case `test_Lab1.py`) from the directory name (i.e. where the test file lives). Then the third line connects our directory with the data file.

We'll learn more about GitHub Actions next time…

1.6.2 Writing tests

To write your own tests, you need to 1. Create a python file whose name starts with `test` 2. `import` the python file that you are testing 3. Write out what you expect to happen 4. Write an assert statement to test the results of your code against your expectations

Your turn Edit `test_importdata` to test the function `importdata` on importing your dataset from earlier.

1.6.3 Final Thoughts

To finish up this lab, create a post to **#lab01_submission** channel on slack sharing a screenshot of all your tests passing.

If your have questions from this lab, post them to #lab_questions with the same preamble (i.e. starting with **Lab1**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Resources consulted to build this lab:

1. *Python for Data Analysis*
2. Open Source and Reproducible MIR Research
3. NumPy Tutorial: Data analysis with Python on Dataquest
4. A Quick Introduction to the "Pandas" Python Library
5. Testing Python Applications with Pytest
6. Find current directory and file's directory [duplicate]
7. Travis CI tests (Python 3.5, 3.6) failing on reproducibility tests
8. Ignoring Files and Directories in Git with .gitignore
9. ignoring any 'bin' directory on a git project