Lab 2: Importing and plotting data

Data Science for Biologists • University of Washington • BIOL 419/519 • Winter 2019

Course design and lecture material by Bingni Brunton (https://github.com/bwbrunton) and Kameron Harris (https://github.com/kharris/). Lab design and materials by Eleanor Lutz (https://github.com/eleanorlutz/), with helpful comments and suggestions from Bing and Kam.

Table of Contents

Helpful Resources

- Python Data Science Handbook (http://shop.oreilly.com/product/0636920034919.do) by Jake VanderPlas
- Python Basics Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/e30fbcd9-f595-4a9f-803d-05ca5bf84612) by Python for Data Science
- Jupyter Notebook Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/48093c40-5303-45f4-bbf9-0c96c0133c40) by Python for Data Science
- Matplotlib Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/28b8210c-60cc-4f13-b0b4-5b4f2ad4790b) by Python for Data Science
- Numpy Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/e9f83f72-a81b-42c7-af44-4e35b48b20b7) by Python for Data Science

Data

- The data in this lab was downloaded from Kaggle (https://www.kaggle.com/uciml/iris) (originally from Fisher 1936 (http://rcs.chemometrics.ru/Tutorials/classification/Fisher.pdf)) and was edited for teaching purposes.

Lab 2 Part 1: Review of Numpy arrays

In lecture this week we used Numpy arrays to generate random numbers, look at data, and make patterns. In this first lab section we'll review how to create, access, and edit parts of a Numpy array.

To use the Numpy library we need to first import it using the command `import numpy as np` . We'll also import Matplotlib in this same code block, since we'll use this library later in the lab. It's good practice to import all of your libraries at the very beginning of your code file, so that anyone can quickly see what external libraries are necessary to run your code.

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt

        # Magic command to turn on in-line plotting
        # (show plots within the Jupyter Notebook)
        %matplotlib inline
```

Creating a Numpy array from existing data

To review some important concepts about Numpy arrays, let's make a small 3x3 array called `alphabet_data` , filled with different letters of the alphabet:

```
In [ ]: row_A = ["A", "B", "C"]
        row_D = ["D", "E", "F"]
        row_G = ["G", "H", "I"]

        alphabet_data = np.array([row_A, row_D, row_G])
        print(alphabet_data)
```

We can use the `print` command to look at the entire `alphabet_data` Numpy array. But often we'll work with very large arrays full of data, and we'll want to pick small subsets of the data to look at. Therefore, it's useful to know how to ask Python to give you just a section of any Numpy array.

Selecting subsets of Numpy arrays

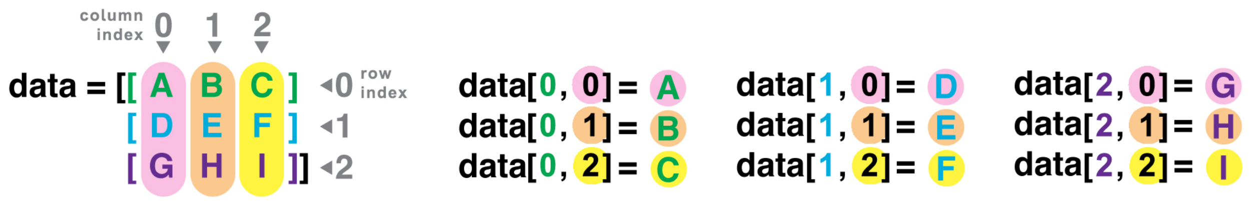
In lab 1, we talked about how index values describe where to find a specific item within a Python list or array. For example, the variable `example_list` is a list with one row, containing three items. To print the first item in the list we would print `example_list[0]` , or *the value in the variable example_list at index 0*. Remember that the first item in a Python list corresponds to the *index* 0.

In [ ]:
```
example_list = ["avocado", "tomato", "onion"]

print("example_list is:", example_list)
print("example_list[0] is:", example_list[0])
```

Selecting a single value in a Numpy array

`alphabet_data` is a little more complicated since it has rows *and* columns, but the general principle of indexing is still the same. Each value in a Numpy array has a unique index value for its row location, and a separate unique index value for its column location. We can ask Numpy to give us just the value we want by using the syntax `alphabet_data[row index, column index]`.



Exercise 1: Use indexing to print the second item in the first row of `alphabet_data`:

In [ ]:
```
```

Selecting a range of values in a Numpy array

In addition to selecting just one value, we can use the syntax `lower index range : upper index range` to select a range of values. Remember that ranges in Python are *exclusive* - the last index in the range is not included. Below is an example of range indexing syntax used on `example_list`:

In [ ]:
```
example_list = ["avocado", "tomato", "onion"]

print("example_list is:", example_list)
print("example_list[0:2] is:", example_list[0:2])
```

We can use exactly the same notation in a Numpy array. However, since we have both row *and* column indices, we can declare one range for the rows and one range for the columns. For example, the following code prints all rows from index 0 to index 3, and all columns from index 0 to index 2. Note that index 3 doesn't actually exist - but since the upper index range is not included in a Python range, we need to use an index of 3 to print everything up to index 2.

In [ ]:
```
print(alphabet_data[0:3, 0:2])
```

Exercise 2: Print the first two rows of the first two columns in `alphabet_data`.

In [ ]:
```
```

Exercise 3: Print the last two rows of the last two columns in `alphabet_data`.

In [ ]:
```
```

Once we know how to select subsets of arrays, we can use this knowledge to *change* the items in these selections. For example, in a list we can assign a value found at a specific index to be something else. In this example we use indexing to reference the first item in `example_list`, and then change it.

In [ ]:
```
example_list = ["avocado", "tomato", "onion"]
print("before assignment, the example_list is:", example_list)

example_list[0] = "banana"
print("after assignment, the example_list is:", example_list)
```

Similarly, we can change items in a Numpy array using indexing:

In [ ]:
```
print("before assignment, alphabet_data is:")
print(alphabet_data)

alphabet_data[0, 0] = "Z"
print("after assignment, alphabet_data is:")
print(alphabet_data)
```

Exercise 4: Replace the item in the third row and second column of `alphabet_data` with `"V"`.

In [ ]:
```
```

Exercise 5: Replace the entire second row of `alphabet_data` with a new row: `["X", "Y", "X"]`

In [ ]:
```
```

Lab 2 Part 2: Importing data from a file into a Numpy array

Let's apply these principles of Numpy arrays to some real biological data. In the `Lab_02` data folder there are three data files:

- `./data/Lab_02/Iris_setosa_data.csv`
- `./data/Lab_02/Iris_versicolor_data.csv`
- `./data/Lab_02/Iris_virginica_data.csv`

These files contain data collected by botanist Edgar Anderson in the Gaspe Peninsula for three species of irises: *Iris setosa*, *Iris versicolor*, and *Iris virginica*. Anderson was interested in investigating the morphologic variation in these three related species. He collected 50 individual flowers for each species, and recorded four different measurements for each flower: the length and width of the sepals and petals.

The data is formatted as a large table, with one file for each species. The files contain 50 rows, each representing one individual, and four columns, which represent sepal length, sepal width, petal length, and petal width. For example, `Iris_setosa_data.csv` corresponds to the column and row labels shown below:

Sample ID	Sepal length (cm)	Sepal width (cm)	Petal length (cm)	Petal width (cm)
Individual 1	5.1	3.5	1.4	0.2
Individual 2	4.9	3	1.4	0.2
Individual 3	4.7	3.2	1.3	0.2
...	...	...	...	...
Individual 50	5	3.3	1.4	0.2

We'll use the Numpy command `loadtxt` to read in our first file, `Iris_setosa_data.csv`. We will save this data in a Numpy array called `iris_data`.

In [ ]:
```
# Load our file data from "filename" into a variable called iris_data
filename = "./data/Lab_02/Iris_setosa_data.csv"
iris_data = np.loadtxt(fname=filename, delimiter=",")
```

The data description above tells us that `iris_data` should contain 50 rows and 4 columns, so let's use the Numpy `shape` command to double check that's the case. Numpy `shape` will print two numbers in the format `(number of rows, number of columns)`.

Exercise 6: Right now, the code below prints a warning if we don't have the expected 50 rows. Edit the code so that the warning is also printed if the number of columns is not 4.

In [ ]:
```
# Print the shape of the loaded dataset
data_shape = iris_data.shape
print("Iris data shape is:", data_shape)

# Print a warning if the data shape is not what we expect
if data_shape[0] != 50:
    print("Unexpected data shape!")
else:
    print("Correct data shape of 50 rows, 4 columns!")
```

It looks like our `iris_data` Numpy array is the shape we expect. Now let's look at a subset of data to see what kind of data we're working with.

Exercise 7: Use Python array indexing to print the first three rows, first four columns of `iris_data`. Check to make sure that the printed data matches what is given to you in the data description above.

In [ ]:
```
```

Lab 2 Part 3: Examining and plotting data in a Numpy array

Calculate interesting characteristics of a Numpy array

Now that we have loaded our Iris data into a Numpy array, there are several interesting commands we can use to find out more about our data. First let's look at the sepal length column for *Iris setosa* (the first column in the dataset). Using array indexing, we will put this entire first column into a new variable called `sepal_lengths`. When indexing between a range of values, leaving the upper range bound blank causes Python to include everything until the end of the array:

In [ ]:
```
# put the sepal lengths for this dataset in a variable called sepal_leng
sepal_lengths = iris_data[0:, 0]
print("The sepal lengths in this dataset is:")
print(sepal_lengths)
```

Numpy contains many useful functions for finding out different characteristics of a dataset. The code below shows some examples:

In [ ]:
```
# Print some interesting characteristics of the data
print("Mean:", np.mean(sepal_lengths))
print("Standard deviation:", np.std(sepal_lengths))
print("Median:", np.median(sepal_lengths))
print("Minimum:", np.min(sepal_lengths))
print("Maximum:", np.max(sepal_lengths))
```

We can use our `sepal_lengths` variable and the useful characteristics we found above to make a histogram of our data. In the below code we've created a histogram, and added a line that shows where the mean of the dataset is.

Exercise 8: Edit the code block below to plot the maximum and minimum data values as two additional vertical lines.

In [ ]:
```
# Create a histogram with an opacity of 50% (alpha=0.5)
plt.hist(sepal_lengths, alpha=0.5)

# Add a vertical line to the plot showing the mean.
plt.axvline(np.mean(sepal_lengths), label="mean")
# Your code here!

# Don't forget to label the axes!
plt.xlabel("Sepal length (cm)")
plt.ylabel("Frequency (number of individuals)")

# Add a legend to the plot
plt.legend()

# Show the plot in our jupyter notebook
plt.show()
```

Review of for loops using indexing

Last week in lab we went over an example of a `for` loop that uses indices to loop through a list. Let's pretend that in this *Iris setosa* dataset, we have marked in our lab notebook that the first, 12th, 26th, and 44th irises we sampled seemed suspiciously small. Let's use a `for` loop to print out the sepal length of each of these irises.

In [ ]:
```
# First let's make a list of all of the indexes where we can find suspic
interesting_indices = [0, 11, 25, 43]

# Now we'll look at every single index in the list of suspicious indices
for index in interesting_indices:

    # Because we are looking at indices, we need to use indexing to find
    # value in sepal_lengths that we're interested in.
    sepal = sepal_lengths[index]

    print("The sepal length at index", index, ":", sepal)
```

Exercise 9: Instead of using a `for` loop to look at just the indices in `interesting_indices`, use a `for` loop to look at *all* indices in the `sepal_lengths` dataset. Remember that you can use the command `len(sepal_lengths)` to find out how many values are in the data. Print the sepal length and index if the sepal length is larger than the mean sepal length.

In [ ]:
```

```

So far we've only looked at the sepal lengths in this dataset. Let's use a `for` loop to also look at the sepal widths, petal lengths, and petal widths. Remember that the columns in this dataset stand for:

In [ ]:
```
sepal_lengths = iris_data[0:, 0]
sepal_widths = iris_data[0:, 1]
petal_lengths = iris_data[0:, 2]
petal_widths = iris_data[0:, 3]

morphologies = [sepal_lengths, sepal_widths, petal_lengths, petal_widths

for morphology in morphologies:
    # Create a histogram
    plt.hist(morphology)
    # Show the plot in our jupyter notebook
    plt.show()
```

Notice that the code in the above box is doing the same action for every column in the array. So instead of re-assigning every column in the array to a new variable called `sepal_lengths` , `sepal_widths` , etc, let's use array indexing to loop through the data instead. Notice that the only thing changing when looking at different columns is the *column index*.

Exercise 10: Change the following code so that it creates a histogram for all columns in *Iris setosa*, like in the previous block. However, instead of making a new variable for each column called `sepal_lengths` , `sepal_widths` , etc, use indexing instead.

In [ ]:

```
column_indices = [0, 1, 2, 3]

for index in column_indices:
    data_subset = # Your code here
    # Create a histogram
    plt.hist(data_subset)
    # Show the plot in our jupyter notebook
    plt.show()
```

Putting it all together: Using a for loop to load, analyze, and plot multiple data files

We've now found some interesting things about *Iris setosa*. But our original dataset included three different species - *Iris setosa*, *Iris versicolor*, and *Iris virginica*. We probably want to run these exact same analyses for each species, and this is a great opportunity to use a `for` loop to make our lives easier. Because all three of our datasets are exactly the same shape and format, we can reuse all of our code that we've already written.

In [ ]:

```
# First, make a list of each filename that we're interested in analyzing
filenames = ["./data/Lab_02/Iris_setosa_data.csv",
             "./data/Lab_02/Iris_versicolor_data.csv",
             "./data/Lab_02/Iris_virginica_data.csv"]
```

Now that we have a list of filenames to analyze, we can turn this into a `for` loop that loads each file and then runs analyses on the file. The code block below has started the process - for each filename, we load in the file data as a variable called `iris_data` . Note that we're not actually doing anything with the data yet, so we don't see many interesting things being printed.

In [ ]:

```
for filename in filenames:
    # Load our file data from "filename" into a variable called iris_dat
    iris_data = np.loadtxt(fname=filename, delimiter=",")
    print("NOW ANALYZING DATASET: ", filename)
```

The data loading doesn't seem to have caused any errors, so we'll continue to copy and paste the code we've already written to work with the data. Note that everything we've copied and pasted is code we've already written - but now we're asking Python to run this same code on *all* the data files, instead of just *Iris setosa*. For the purposes of this exercise, we'll analyze just the sepal lengths of the dataset, so that we end up with a manageable number of output plots.

In [ ]:
```
for filename in filenames:
    # Load our file data from "filename" into a variable called iris_dat
    iris_data = np.loadtxt(fname=filename, delimiter=",")
    print("----")
    print("NOW ANALYZING DATASET: ", filename)

    # Print the shape of the loaded dataset
    data_shape = iris_data.shape
    print("Iris data shape is:", data_shape)

    # Print a warning if the data shape is not what we expect
    if data_shape[0] != 50:
        print("Unexpected data shape!")
    else:
        print("Correct data shape of 50 rows, 4 columns!")

    # put the sepal lengths for this dataset in a variable called sepal_
    sepal_lengths = iris_data[0:, 0]
    print("The sepal lengths in this dataset is:")
    print(sepal_lengths)
```

Exercise 11: Similarly, add in the code you've already written to print the interesting characteristics of the data (mean, median, max, etc.) and create a histogram for each data file that includes the mean and median. Run your final for loop. Which iris species has the longest mean sepal length? Smallest minimum sepal length?

In [ ]:
```
```

Lab 2 Bonus exercise

Bonus Exercise 1: Now take the above code and edit it so that we analyze all of the 4 flower morphology variables, for all of the species of plants. Label the plot axis and title with the appropriate information (flower species for title, and the morphological variable on the x axis).

In [ ]:
```
```

```
In [ ]:
```