While you are waiting for the workshop to begin, please launch Spyder.

All code and slides covered today are on my Github:
https://github.com/atchen/python_workshop_part2

Introduction to Python for Biological Data Analysis: Part 2

Ann Chen
PhD student in BME
August 8, 2018

Yale SCHOOL OF ENGINEERING & APPLIED SCIENCE

Acknowledgements

Justin Bois
Lecturer, Division of Biology
and Biological Engineering at Caltech

This workshop is an *extremely* condensed version of the various Python courses he teaches.

I highly recommend visiting his website (bois.caltech.edu) and checking out BE/Bi/NB 203 and BE/Bi 103.

Most lessons are open-access.

Summary of part 1

Data type	Definition	Called by	Mutable?
integer	any whole number	an integer	no
float	any real, non-whole number	a float	no
string	a sequence of characters	single, double, or triple quotes	no
list	a sequence of objects	[ ]	yes
tuple	a sequence of objects	( )	no

Review of part I

```
my_list = [1, 3.49, 'apple', (1,'orange')]

print('the type of my_list is', type(my_list))
print('the type of my_list[0] is', type(my_list[0]))
print('the type of my_list[1] is', type(my_list[1]))
print('the type of my_list[2] is', type(my_list[2]))
print('the type of my_list[3] is', type(my_list[3]))

#lists are mutable
my_list[0] = 5
print(my_list)

#tuples are immutable
my_tuple = (1, 2)
my_tuple[0] = 2 #this will result in an error
my_list[3][0] = 2 #this will also result in an error
```


Review of part 1

```
def append_a(test_list):
    '''
    Adds the str 'a' after each element in a user-defined list.
    ex: input = [1, 2]. output = [1, 'a', 2, 'a']
    '''
    new_list = []
    for i in test_list:
        new_list.append(i)
        new_list.append('a')
    return new_list

print(append_a(my_list))
```


Today's agenda

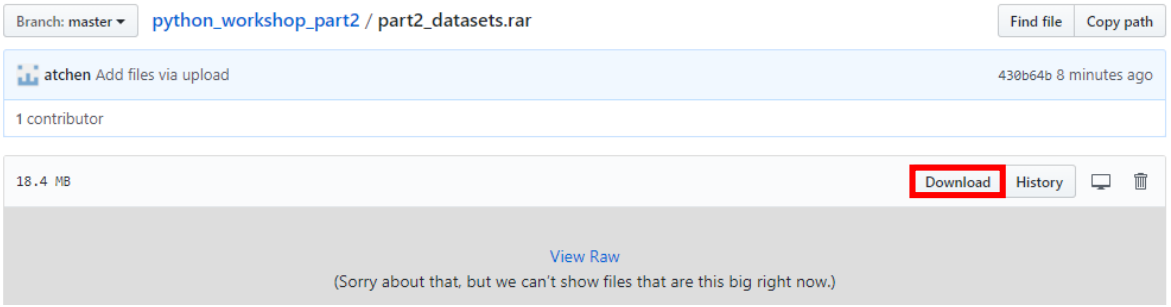
Today, we will use what we learned to analyze datasets from published articles using the Pandas package.

In addition, we will go over how to plot our data using two different packages: seaborn and matplotlib.

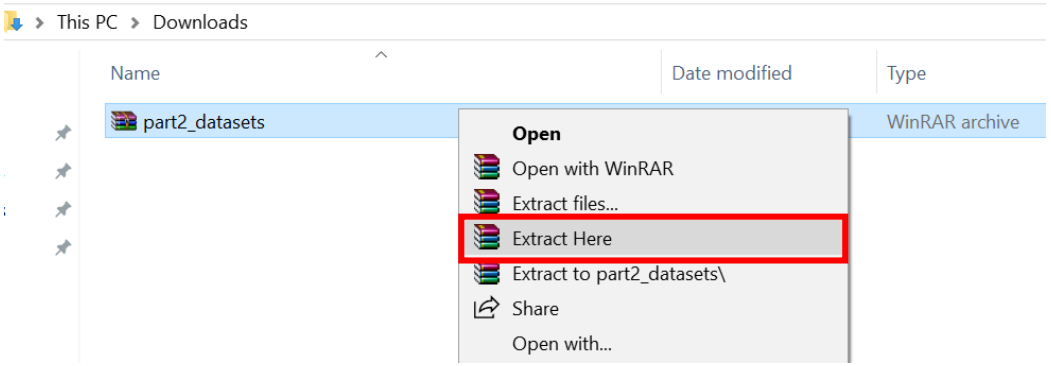
We will be covering three datasets from Grant PR, et al, Kleinteich T, et al., and Darmanis S, et al., respectively.

Download and extract files from Github

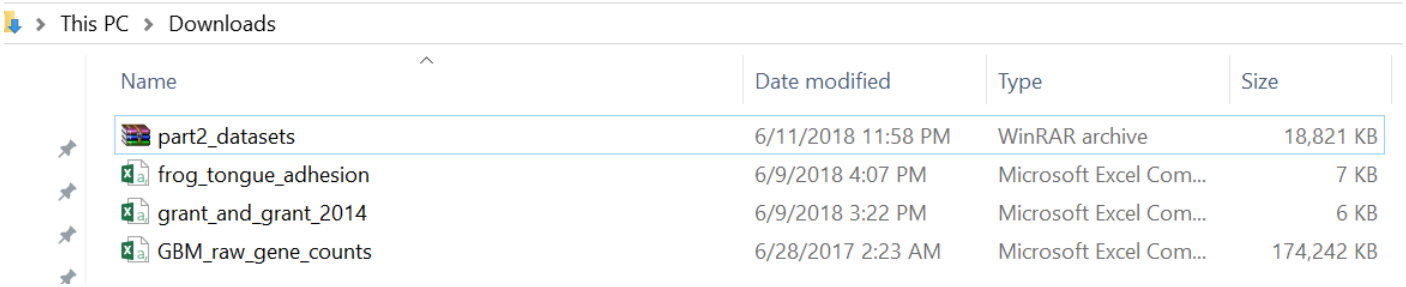
1.



2.



3.



Loading a CSV file

Download the file `grant_and_grant_2014.csv` and load it

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#use pd.read_csv() to read in the data and store in a DataFrame
df = pd.read_csv('C:/Users/Ann/Downloads/grant_and_grant_2014.csv', comment='#')

df.head()
```

The data is stored in a **dataframe**, a data type that Panda uses for convenient data analysis.

This dataset investigates the heritability of beak depth. It includes the maternal beak depth, paternal beak depth, and mean offspring beak depth.

Grant, PR, Grant, BR (2014) 40 years of evolution: Darwin's finches on Daphne Major Island. Princeton: Princeton University Press. http://www.worldcat.org/oclc/854285415

Dataframe columns

To access a column in a dataframe, use the following syntax

```
#slicing a column out of a dataframe by using the column name
df['Average offspring beak depth (mm)']
```

```
In [13]: df.head()
Out[13]:
   Average offspring beak depth (mm)  Paternal beak depth (mm)  \
0                              10.70                     10.90
1                               9.78                     10.70
2                               9.48                     10.70
3                               9.60                     10.70
4                              10.27                      9.85

   Maternal beak depth (mm)
0                       9.3
1                       8.4
2                       8.1
3                       9.8
4                      10.4
```

```
In [14]: df['Average offspring beak depth (mm)']
Out[14]:
0      10.70
1       9.78
2       9.48
3       9.60
4      10.27
```


Slicing dataframes with Booleans

Let's say we only want rows with average offspring beak depth ≥ 11 mm.
Pandas dataframes can be conveniently sliced with Booleans.

```
#only want indices with offspring beak depth >= to 11 mm
inds = df['Average offspring beak depth (mm)'] >= 11
inds
```

`inds` is an array of Booleans that is the same length of the dataframe.
It is `True` when the condition is met and `False` if it is not.

```
#slice out rows we want
df_big_offspring_bd = df.loc[inds]
df_big_offspring_bd
```

Indexing of rows is preserved.

Indexing with loc

Using `.loc` allows us to index by row. We put an array of Booleans and get back a dataframe containing rows associated with `True` in the arrays of Booleans.

Notice, the original indexing of rows is preserved.
For example, there is no index 2.

```
#this will result in an error
df_big_offspring_bd.loc[2]
```

To get around this, use `.iloc` which gives indexing with sequential integers.

```
#this will return the third row
df_big_offspring_bd.iloc[2]
```


Renaming columns

Pandas has a nice method for renaming column headers.
Let's rename the columns to

'Average offspring beak depth (mm)' → 'avg_offspring_bd'
'Paternal beak depth (mm) → 'paternal_bd'
'Maternal beak depth (mm) → 'maternal_bd'

```
rename_dict = {'Average offspring beak depth (mm)' : 'avg_offspring_bd', \
               'Paternal beak depth (mm)' : 'paternal_bd', \
               'Maternal beak depth (mm)' : 'maternal_bd'}

df = df.rename(columns=rename_dict)
df.head()
```


NumPy arrays from dataframe columns

```
offspring_bd = df['avg_offspring_bd'].values
paternal_bd = df['paternal_bd'].values
maternal_bd = df['maternal_bd'].values
```

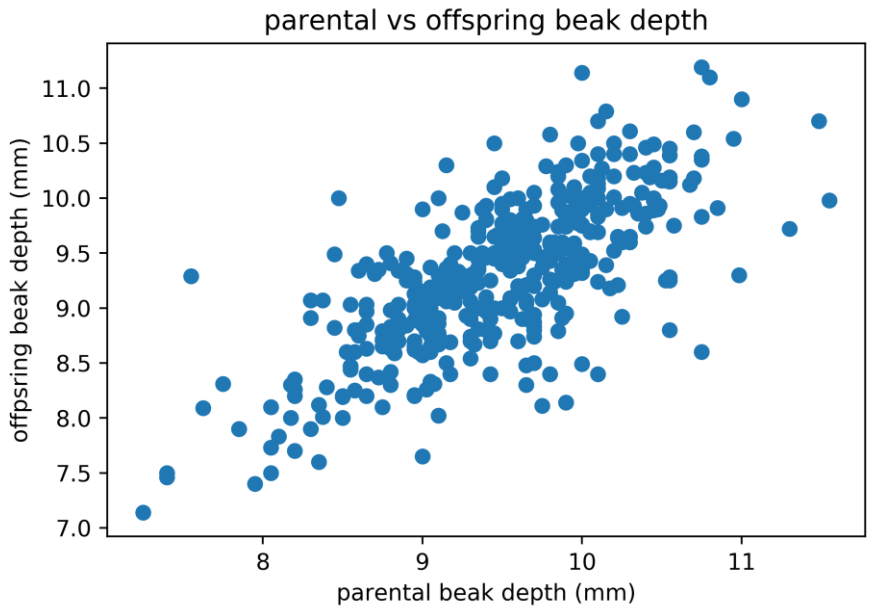
Now the values of the column are stored in a NumPy array (uses same notation for indexing as a list)

Let's calculate the mean parental beak depth. Conveniently, we don't need to use a for loop for each parent pair.

```
parental_bd = (paternal_bd + maternal_bd) / 2
```


Scatterplots

```
#plot mean parental beak depth vs mean offspring beak depth
plt.scatter(parental_bd, offspring_bd)
plt.xlabel('parental beak depth (mm)')
plt.ylabel('offpsring beak depth (mm)')
plt.title('parental vs offspring beak depth')
plt.savefig('scatter.png', dpi=500)
```



Linear regression with SciPy

Let's perform a linear regression to get the slope and intercept of the data. `scipy.optimize.curve_fit()` takes a set of data and returns the parameters of the model function that best describes the data.

```
import scipy.optimize

def linear_fun(x, slope, intercept):
    return slope * x + intercept

#compute the curve fit (guess is unit slope and zero intercept)
popt, covar = scipy.optimize.curve_fit(linear_fun, parental_bd, offspring_bd,
                                       p0=[1,0])
```

scipy.optimize.curve_fit(model function, x-data, y-data, guess for parameters). Read the documentation for more information about this function.

Linear regression with SciPy

`scipy.optimize.curve_fit()` returns two arrays. The first contains the optimal set of parameters for the curve fit. The second is a 2D array with the covariance of the parameters.

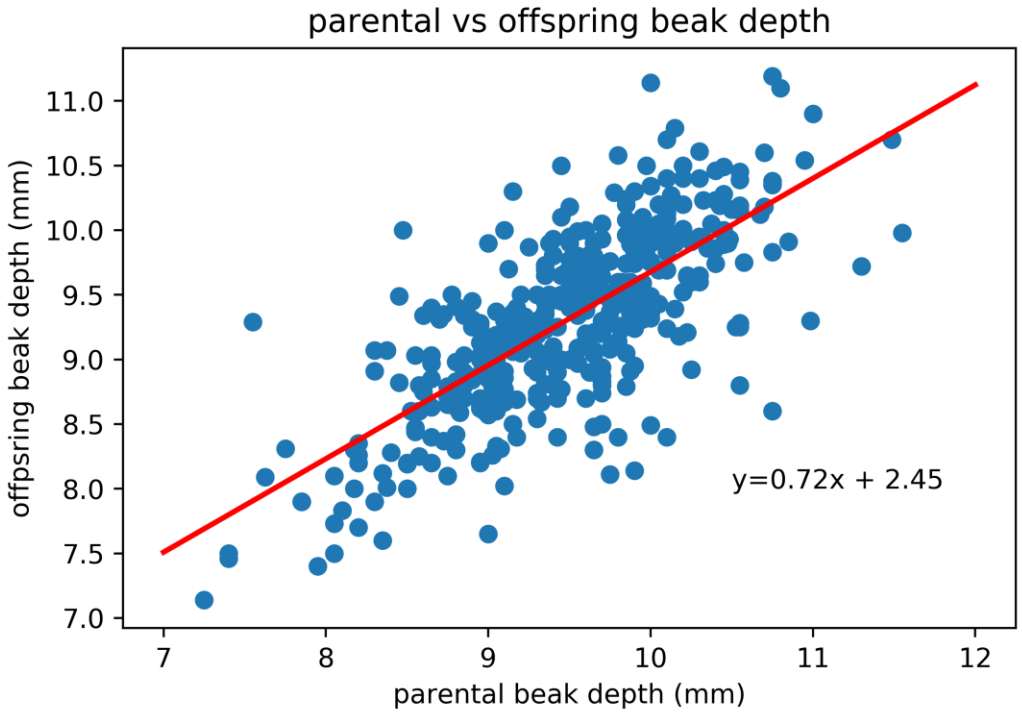
```
#parse the results
slope, intercept = popt

#print the results
print('slope =', slope)
print('intercept = ', intercept, 'mm')
```

slope = 0.722905188545
intercept = 2.44841830979 mm

Scatterplot with line of best fit

```
#define range for line
x = np.array([7, 12])
y = linear_fun(x, slope, intercept)

plt.scatter(parental_bd, offspring_bd)
plt.plot(x, y, color='red', linewidth=2)
plt.xlabel('parental beak depth (mm)')
plt.ylabel('offpsring beak depth (mm)')
plt.title('parental vs offspring beak depth')
plt.text(10.5, 8, 'y=%.2fx + %.2f' %(slope, intercept), fontsize=10)
plt.savefig('scatter_w_line.png', dpi=500)
```


Scatterplot with line of best fit



Making a dataframe

```
#use a dictionary to make a dataframe
data_dictionary = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data=data_dictionary, index=[49, 48, 1])
```

More practice with `loc` and `iloc`:

```
#returns the second row
df.iloc[1]

#returns the row with the index 1 (third row in this dataframe)
df.loc[1]
```

```
In [165]: df
Out[165]:
    A  B  C
49  1  4  7
48  2  5  8
1   3  6  9
```

```
In [166]: df.iloc[1]
Out[166]:
A    2
B    5
C    8
Name: 48, dtype: int64
```

```
In [167]: df.loc[1]
Out[167]:
A    3
B    6
C    9
Name: 1, dtype: int64
```


Frog tongue adhesion

In this dataset, the authors looked at the adhesive strength of the tongue for South American horned frogs. Each row is an experiment where they recorded different metrics.

```
#read dataset and store in dataframe
df = pd.read_csv('C:/Users/Ann/Downloads/frog_tongue_adhesion.csv', comment='#')

df.head()
```

	date	ID	trial number	impact force (mN)	impact time (ms)	impact force / body weight	adhesive force (mN)	time frog pulls on target (ms)	adhesive force / body weight	adhesive impulse (N-s)	total contact area (mm2)	contact area without mucus (mm2)	contact area with mucus / contact area without mucus	contact pressure (Pa)	adhesive strength (Pa)
0	2013_02_26	1	3	1205	46	1.95	-785	884	1.27	-0.290	387	70	0.82	3117	-2030
1	2013_02_26	1	4	2527	44	4.08	-983	248	1.59	-0.181	101	94	0.07	24923	-9695
2	2013_03_01	1	1	1745	34	2.82	-850	211	1.37	-0.157	83	79	0.05	21020	-10239
3	2013_03_01	1	2	1556	41	2.51	-455	1025	0.74	-0.170	330	158	0.52	4718	-1381
4	2013_03_01	1	3	493	36	0.80	-974	499	1.57	-0.423	245	216	0.12	2012	-3975

Kleinteich T, Gorb SN (2014) Tongue adhesion in the horned frog Ceratophrys sp. Sci Rep. 2014 Jun 12;4:5225. doi: 10.1038/srep05225.

Data extraction

Let's extraction the experiment with index 42.

```
#extract experiment with index 42
df.loc[42]
```

We can also use Boolean slicing to extract data.

Let's say we want to look at trial number 3 on May 27, 2013 of frog III. This is a more common and meaningful case.

```
#set up boolean slicing
date = df['date'] == '2013_05_27'
trial = df['trial number'] == 3
ID = df['ID'] == 'III'

#slice out the row
df_slice = df.loc[date & trial & ID]
```

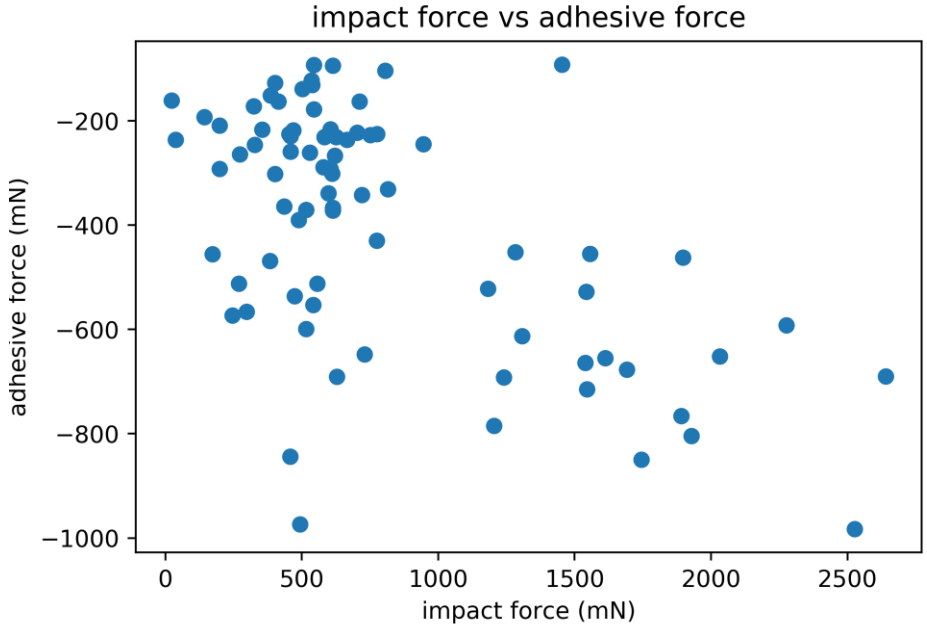

Computing with dataframes and inserting columns

Let's make a new column with impact force in Newtons. We can divide the `impact force (mN)` column elementwise by 1000, just as with arrays.

```
#add a new columns with impact force in units of newtons
df['impact force (N)'] = df['impact force (mN)'] / 1000

df.head()
```


Plotting how impact force correlates with other metrics

```
plt.scatter(df['impact force (mN)'], df['adhesive force (mN)'])
plt.xlabel('impact force (mN)')
plt.ylabel('adhesive force (mN)')
plt.title('impact force vs adhesive force')
plt.savefig('frog_scatter.png', dpi=500)
```



Making subplots

Let's say we want to plot the adhesive force, total contact area, impact time, and contact pressure against impact force.

```
# set figure = fig and subplots = ax
fig, ax = plt.subplots(4, sharex=True)
plt.xlabel('impact force (mN)')
ax[0].scatter(df['impact force (mN)'], df['impact time (ms)'])
ax[1].scatter(df['impact force (mN)'], df['adhesive force (mN)'])
ax[2].scatter(df['impact force (mN)'], df['total contact area (mm2)'])
ax[3].scatter(df['impact force (mN)'], df['contact pressure (Pa)'])
fig.savefig('frog_subplots.png',dpi=500)
```


Making subplots

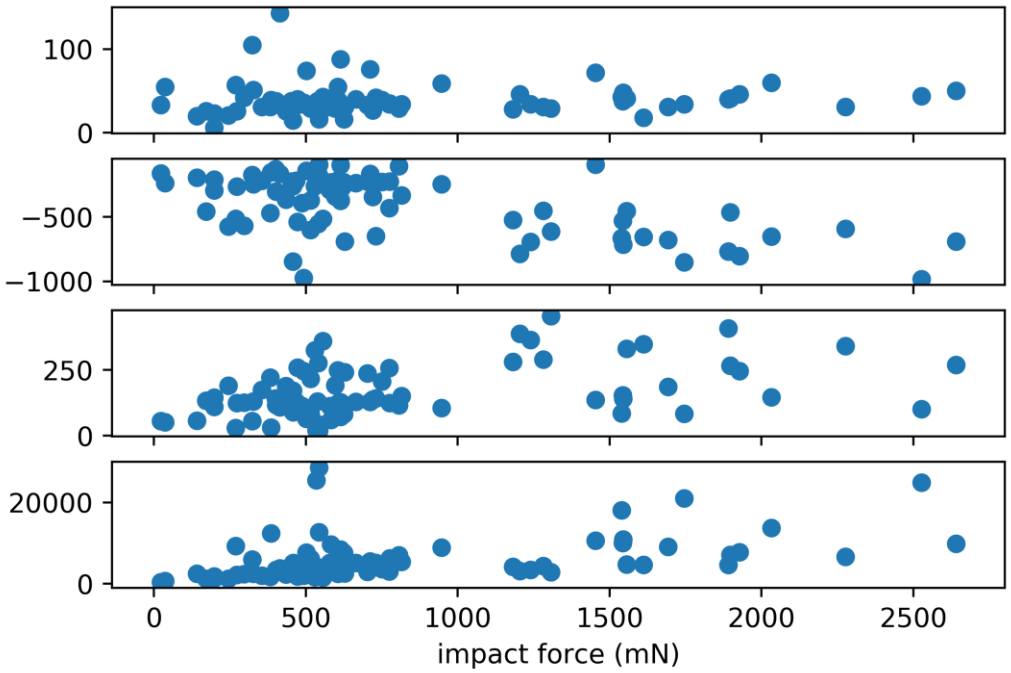
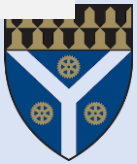


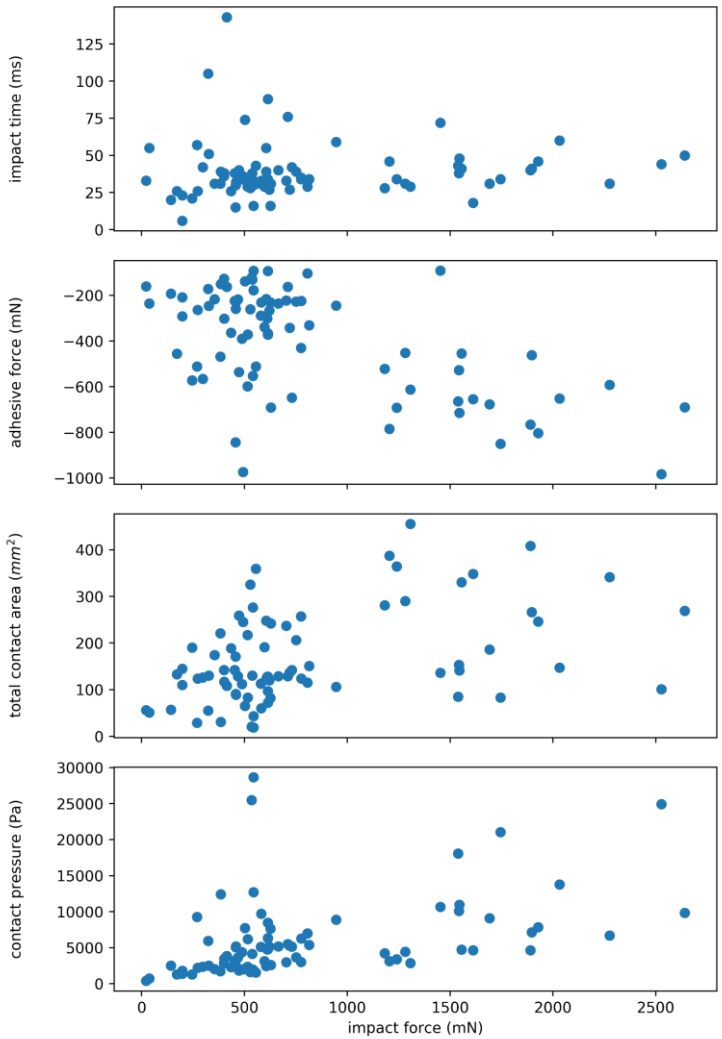
Figure seems a bit stretched out, and there are no y-axis labels.

Subplot parameters

```
fig, ax = plt.subplots(4, sharex=True, figsize=(7,10))

plt.xlabel('impact force (mN)')

ax[0].scatter(df['impact force (mN)'], df['impact time (ms)'])
ax[0].set_ylabel('impact time (ms)')
ax[0].get_yaxis().set_label_coords(-0.15, 0.5)

ax[1].scatter(df['impact force (mN)'], df['adhesive force (mN)'])
ax[1].set_ylabel('adhesive force (mN)')
ax[1].get_yaxis().set_label_coords(-0.15, 0.5)

ax[2].scatter(df['impact force (mN)'], df['total contact area (mm2)'])
ax[2].set_ylabel('total contact area ($mm^2$)')
ax[2].get_yaxis().set_label_coords(-0.15, 0.5)

ax[3].scatter(df['impact force (mN)'], df['contact pressure (Pa)'])
ax[3].set_ylabel('contact pressure (Pa)')
ax[3].get_yaxis().set_label_coords(-0.15, 0.5)

plt.tight_layout()
fig.savefig('frog_subplots.png',dpi=500)
```

Subplot parameters



Summary of what was changed:

1. Set y-axis labels for subplots
2. Change figure size
3. Align y-axis labels for subplots
4. Improve spacing with plt.tightlayout()

`enumerate()` as a tool for plotting

```
subplot_list = ['impact time (ms)', 'adhesive force (mN)', \
                'total contact area (mm2)', 'contact pressure (Pa)']

#enumerate function example
for i,e in enumerate(subplot_list):
    print(i)
    print(e)
```

It allows us to loop over something and have an automatic counter.

Subplots with `enumerate()`

```
#same as before
fig, ax = plt.subplots(4, sharex=True, figsize=(7,10))

plt.xlabel('impact force (mN)')

#instead of typing out each subplot individually, use a for loop
for i,e in enumerate(subplot_list):
    ax[i].scatter(df['impact force (mN)'], df[e])
    ax[i].set_ylabel(e)
    ax[i].get_yaxis().set_label_coords(-0.15, 0.5)

plt.tight_layout()
plt.xlabel('impact force (mN)')
fig.savefig('frog_subplots_looped.png',dpi=500)
```

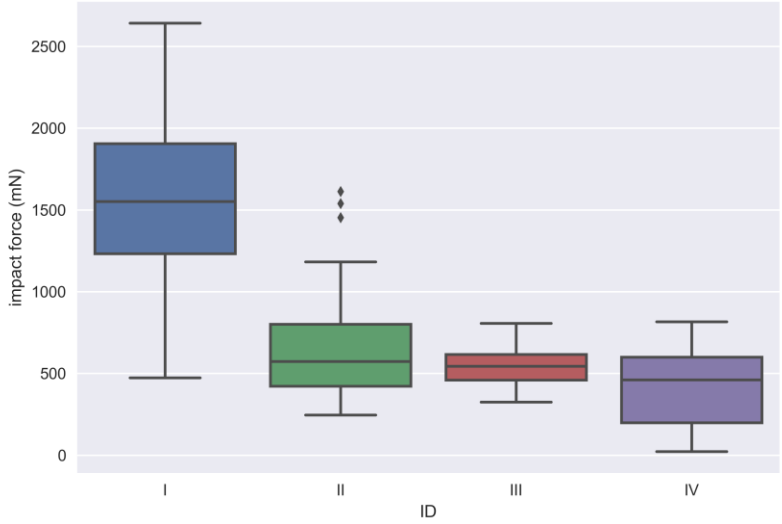
```
ax[0].scatter(df['impact force (mN)'], df['impact time (ms)'])
ax[0].set_ylabel('impact time (ms)')
ax[0].get_yaxis().set_label_coords(-0.15, 0.5)

ax[1].scatter(df['impact force (mN)'], df['adhesive force (mN)'])
ax[1].set_ylabel('adhesive force (mN)')
ax[1].get_yaxis().set_label_coords(-0.15, 0.5)

ax[2].scatter(df['impact force (mN)'], df['total contact area (mm2)'])
ax[2].set_ylabel('total contact area ($mm^2$)')
ax[2].get_yaxis().set_label_coords(-0.15, 0.5)

ax[3].scatter(df['impact force (mN)'], df['contact pressure (Pa)'])
ax[3].set_ylabel('contact pressure (Pa)')
ax[3].get_yaxis().set_label_coords(-0.15, 0.5)
```

```
for i,e in enumerate(subplot_list):
    ax[i].scatter(df['impact force (mN)'], df[e])
    ax[i].set_ylabel(e)
    ax[i].get_yaxis().set_label_coords(-0.15, 0.5)
```

Seaborn: a high level plotting package for Matplotlib

Let's make a box plot of the impact force sorted by frog ID.

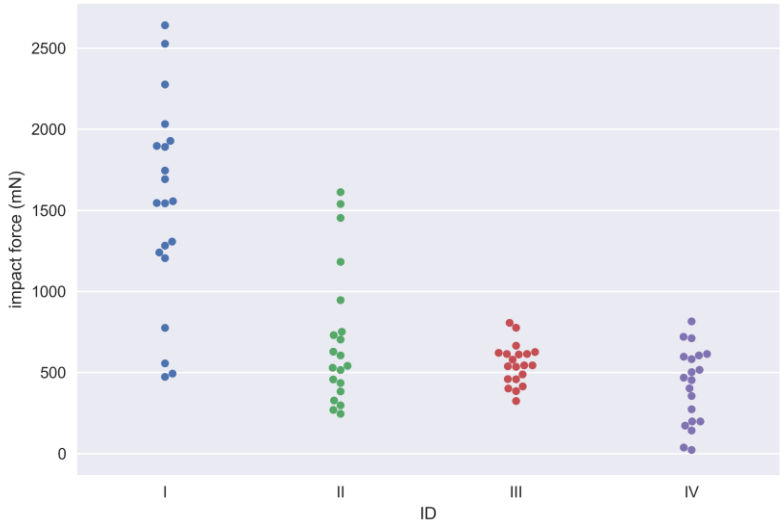
```
import seaborn as sns

#box plot
sns.boxplot(data=df, x='ID', y='impact force (mN)')
plt.savefig('boxplot.png', dpi=500)
```



Seaborn: a high level plotting package for Matplotlib

Now, let's make a beeswarm plot of the impact force sorted by frog ID.

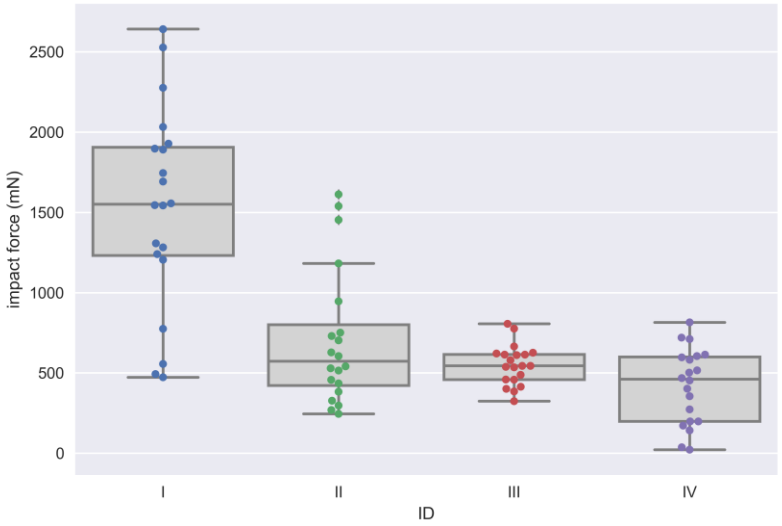
```
#beeswarm plot
sns.swarmplot(data=df, x='ID', y='impact force (mN)')
plt.savefig('beeswarm.png', dpi=500)
```



Seaborn: a high level plotting package for Matplotlib

Lastly, let's overlay the boxplot and the beeswarm plot.

```
#overlay box plot and beeswarm plot
sns.boxplot(data=df, x='ID', y='impact force (mN)', color='lightgray')
sns.swarmplot(data=df, x='ID', y='impact force (mN)')
plt.savefig('boxbeeplot.png', dpi=500)
```



Working with gene expression datasets

Pandas is useful for analyzing extremely large datasets – like single-cell RNA-seq data.

Load in the gene expression matrix. In this case, it is space-delimited.
Be patient. It will take a few seconds to load.

```
#read in the scrna-seq dataset. this will take a few seconds
df = pd.read_csv('C:/Users/Ann/Downloads/GBM_raw_gene_counts.csv', sep=' ')
```

This dataset is 25,000x larger than the other datasets we've been using!

It is a 23386 x 3589 dataframe. Each row is a gene, and each column is a cell. so this dataset contains information for 23386 genes in 3589 cells.

Darmanis, S., et al. Single-Cell RNAseq analysis of infiltrating neoplastic cells at the migrating front of human glioblastoma (2017). Cell Rep. 2017 Oct 31;21(5):1399-1410.

Non-integer index column

```
#let's look at the index column
df.index
```

For this dataset, the index column contains the gene names

```
#df.loc must contain a gene name
df.loc['A2MP1']
```

If you want an integer index column, can use the function `df.reset_index()` or set `index_col=False` in `read_csv()`.

Checking if a gene is in the dataset

```
#can do this manually
'TP53' in df.index
```

Will return `True` if in the gene is in the dataset and `False` if it is not.

What if we have a list of genes we want to check? We can write a function.

```
def gene_in_df(gene_list):
    """
    This function takes in a list of gene names and
    returns the names of genes that are not in the dataset.
    """
    genes_in_df = []
    for i in gene_list:
        if i in df.index:
            genes_in_df.append(i)

    print('Gene(s) not in dataset', set(gene_list) - set(genes_in_df))
    return set(gene_list) - set(genes_in_df)
```


Checking if a gene is in the dataset

Let's test the function out.

```
gene_list = ['ETNPPL', 'FGFR3', 'AQP4', 'GJA1', 'AGT', 'MGST1', 'SLC39A12', \
             'SLC25A18', 'GPR98', 'SLCO1C1', 'SDC4', 'GPR37L1', 'ACSBG1', \
             'SFXN5', 'BMPR1B', 'ATP13A4', 'RANBP3L', 'GJB6', 'GFAP', \
             'PRODH', 'SLC4A4', 'TMEM130', 'GABRB2', 'VSNL1', 'GABRA1', \
             'SYNPR', 'THY1', 'CAMK2A', 'MEG3', 'GABRG2', 'CKMT1B', 'CCK', \
             'CHGB', 'SCG2', 'DNM1', 'MAP7D2', 'CELF4', 'CIT', 'UNC80', \
             'NRXN3', 'SCN2A', 'SNAP25']

gene_in_df(gene_list)
```


Gene expression correlation using Pearsonr

`scipy.stats.pearsonr(x,y)`

Calculate a Pearson correlation coefficient and its corresponding p-value

The Pearson correlation coefficient measures the linear relationship between two datasets. Coefficient will vary between -1 and +1, with 0 implying no correlation and -1/+1 implying an exact linear relationship.

```
from scipy.stats import pearsonr

#measure the correlation between TP53 and DERL1
coef, pval = pearsonr(df.loc['TP53'], df.loc['DERL1'])
print('the coefficient is %0.3f' %coef)
print('the p-value is %0.3f' %pval)
```

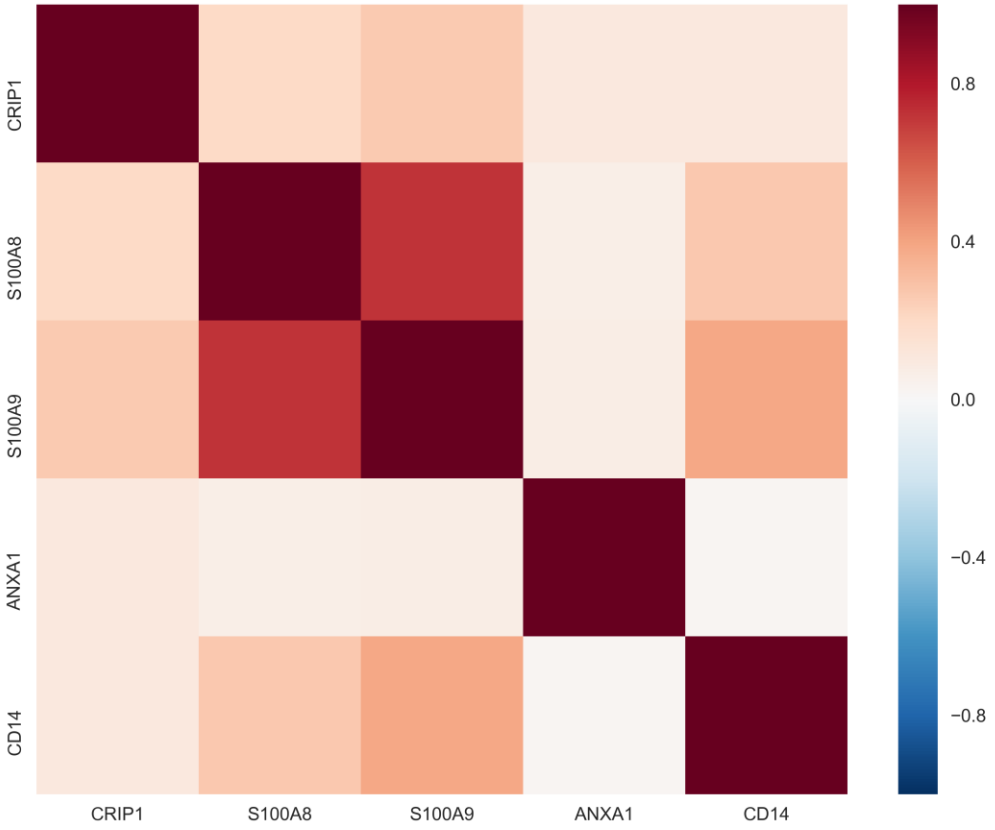
Gene correlation matrix

`pandas.DataFrame.corr`

Computes pairwise correlation of **columns**. Default is Pearsonr correlation.

```
gene_list = ['CRIP1', 'S100A8', 'S100A9', 'ANXA1', 'CD14']

#make a new dataframe with your genes of interest
coef_df = df.loc[gene_list]
coef_df.head()

#transpose the dataframe first so genes are in columns
#then calculate pairwise correlation
corr = coef_df.transpose().corr()

#set color palette where low coefs are blue and high coefs are red
#center the color bar at 0
plt.subplots(figsize=(10, 8))
sns.heatmap(corr, cmap='RdBu_r', center=0)
plt.savefig('correlation_heatmap.png', dpi=500)
```


Gene correlation matrix



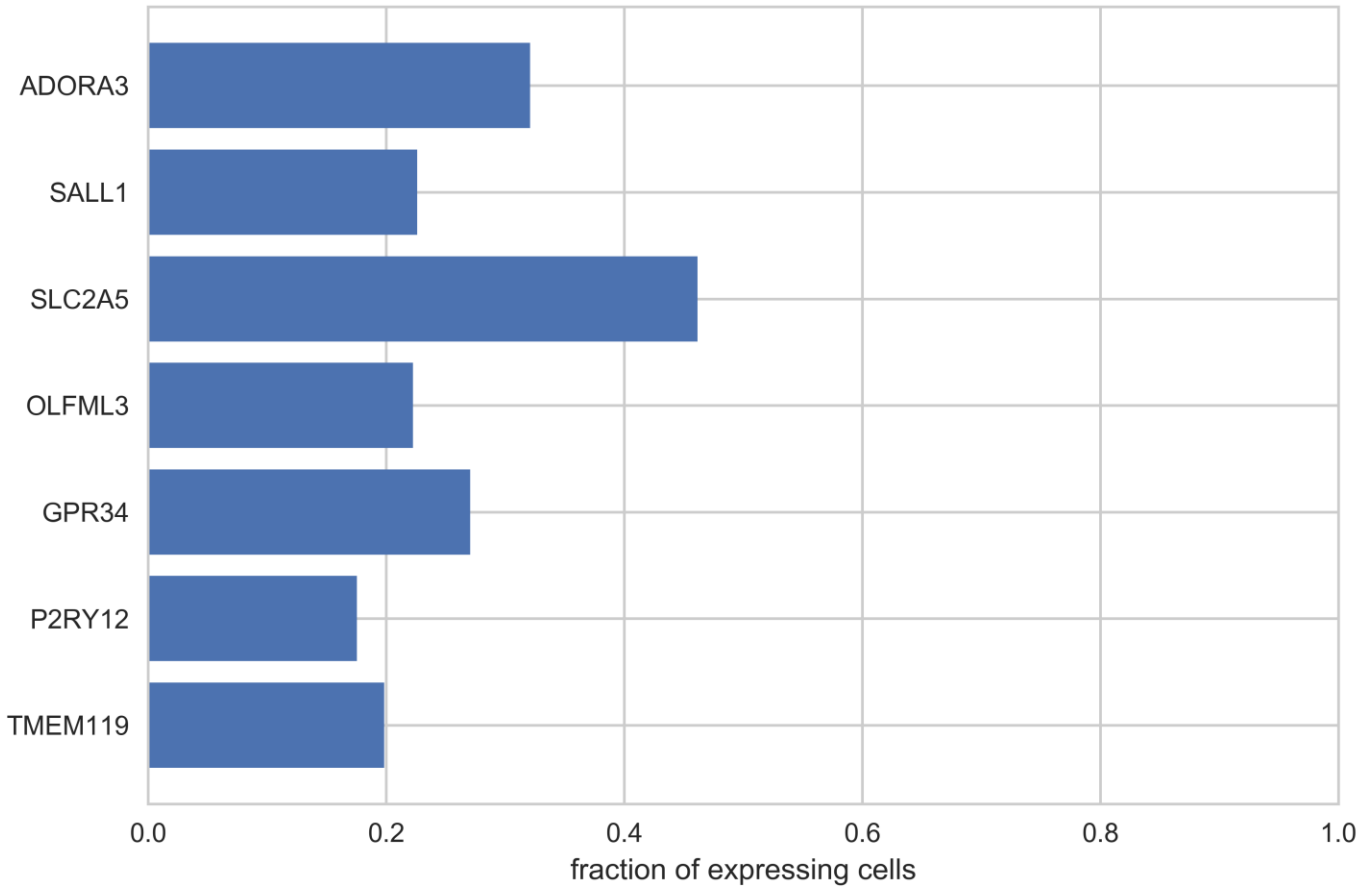
Fraction of cells expressing a gene

```
gene_list = ['TMEM119', 'P2RY12', 'GPR34', 'OLFML3', 'SLC2A5', 'SALL1',
'ADORA3']

def gene_frac_df(gene_list):
    '''
    This function takes in a list of gene names and returns the fraction of
    cells that express each gene.
    '''
    #list to store the fraction for each gene
    gene_frac_list = []
    for i in gene_list:
        if i in df.index:
            #if the gene is in the dataset
            #sum the number of cells that have 0 expression
            gene_0 = (df.loc[i]==0).sum()
            #(total number of cells - cells with 0 expression) / total
            gene_frac = (len(df.columns) - gene_0) / len(df.columns)

            #add the fraction to the gene_frac_list
            gene_frac_list.append(gene_frac)
        else:
            #if the gene is not in the dataset, append 0 to the list
            gene_frac_list.append(0)
    return gene_frac_list
```

Fraction of expressing genes

```
gene_frac_list = gene_frac_df(gene_list)

#array containing the number of genes
y_pos = np.arange(len(gene_list))

#barh = horizontal bar graph
plt.barh(y_pos, gene_frac_list)
plt.xlim(0,1)
plt.yticks(y_pos, tuple(gene_list))
plt.xlabel('fraction of expressing cells')
plt.savefig('frac_gene.png', facecolor='white', dpi=500)
```


Fraction of expressing genes



4 general rules to follow

1. You can never over-comment your code.
2. Write functions that do simple tasks.
3. There might be a package for it. It doesn't hurt to check.
4. If you have a question, someone has probably asked it online.

Thank you for attending!