data analysis with Pandas

- so far, we've seen a few different ways of storing data in Python
  - sequences: tuples `()` and lists `[]`
  - mappings: dictionaries `{}`
  - numpy arrays: `np.array()` - best for handling large multidimensional datasets, fast, memory efficient, vectorized math, matrix math, lots of builtin analyses
- not all experimental data can fit seamlessly into a normal numpy array
  - indexing with integers isn't always ideal
    - sometimes it's nicer to use more meaningful labels, like strings, such as in a dictionary
  - missing data isn't necessarily handled automatically in numpy
  - different number of data points for different subjects/trials, which requires multiple arrays, each of different length
  - heterogenous data types that you want to keep together in the same data object
    - possible with `numpy.recarray()`, but a bit tricky to use
- **Pandas** is a library built on top of numpy that deals with these annoyances
  - designed to make it easier to handle (mostly tabular) real-world data
  - quickly calculate and plot simple analyses
  - Pandas also has the ability to load/save data directly from/to text files, Excel files, as well as databases
  - why the name pandas? comes from "panel data", economics term
- numpy has one basic object type: `numpy.ndarray`, can be 1, 2, 3 or more dimensions
- pandas has two basic object types: `Series` & `DataFrame`, 1 and 2 dimensions respectively, though DataFrames can be optionally made > 2D
- customary name for pandas import is `pd`, i.e. `import pandas as pd`

`pd.Series`

- like a 1D numpy array, but more flexible in that indices don't have to be integers
  - indices are more like labels, can be ints, floats, strings...
  - e.g. time series data of fluorescence intensity of some ROI vs. time
  - with numpy, you'd need two arrays of the same length to properly describe this data: one for fluorescence, and another to store the corresponding timestamps of each measurement

```
fl = np.random.random(20) # fake fluorescence data
t = np.arange(0, 400, 20) # fake timestamps, in ms
```

  - a bit awkward: one data set represented by two separate arrays, with two different names
  - if you want to manipulate this data set, you have to remember to do the manipulation on both arrays, not just one of them!
  - e.g. grab a subset of data, just the first 5 data points:

```
flsub = fl[:5]
tsub = t[:5]
```

- another annoyance: say you want to extract a single fluorescence value at a specific timepoint, like t=60 ms
- 2 step process:

```
i = t == 60 # find where t is 60, save bool array to i
v = fl[i] # use i as index into fl, get a float array with one entry
# or in one line:
v = fl[t == 60] # also a float array with one entry
v[0] # in either case, to extract the actual float value out - tedious!
```

- pandas lets you combine fluorescence data and timestamps into a single pandas data series:
- `s = pd.Series(data=fl, index=t)` - the `index` keyword arg indicates row labels
- select by integer position
  - `len(s)` works as you'd expect
  - `s.iloc[4]` - select the 4th entry
  - `.iloc` stands for "integer location", works just like indexing into a 1D array
  - now if you want to get a subset of this time series, it's a single command:
  - `s.iloc[:5]` for the 1st 5 data points, slices both the data (`fl`) and the index (`t`) at the same time
    - same as `s.head()` which defaults to 5, but you can specify e.g. `s.head(10)` to get the first 10
    - `s.tail()` returns last 5 data points, `s.tail(7)` returns the last 7, etc.
  - `s.iloc[3:7]` does what you'd expect, note that it's end exclusive, just like a numpy array (compare with `fl[3:7]`)
- select by label: the real benefit of a Series
  - as opposed to above numpy example, can get fluorescence at t=60 ms in a single step:
  - `s.loc[60]` - `.loc` stands for "location" (by label), compare with `.iloc`
  - can also slice data directly between labels
  - `s.loc[:60]` returns all values from start to t=60
  - NOTE: slicing by label is end **inclusive**, while slicing by integer position is end **exclusive** (like for lists, tuples and arrays)
  - `s.loc[30:70]` does what you'd expect
  - Series slices always return another Series. To get the actual underlying numpy data values out, use `.values`
    - `s.loc[30:70].values` - returns a normal array of just fluorescence values
  - NOTE: indexing directly into a series, e.g. `s[60]` works the same as `s.loc[60]` , i.e, doing so indexes by label
    - strangely, `s[30:70]` doesn't work the same as `s.loc[30:70]` :(
    - slicing directly into a series without the `.loc` does the same as `s.iloc[30:70]` , which returns nothing because it's out of range
    - better example: `s[5:10]` returns the same as `s.iloc[5:10]`

- can also use float values as the index, which is convenient for e.g. timestamps in seconds, but it's dangerous:

```
tfloat = np.arange(0, 2, 0.1) # float timestamps, in sec
sfloat = pd.Series(data=fl, index=tfloat)
```

  - `sfloat.loc[0.0]` and `sfloat.loc[0.1]` and `sfloat.loc[0.2]` work, but `sfloat.loc[0.3]` doesn't. Why?
  - float roundoff error! 0.3 isn't exactly represented (on my laptop at least), not ideal for use as a label
  - use `sfloat.index` to see the precise values used for the indices
  - `sfloat.loc[0.30000000000000004]` works, but isn't very useful...
- can do vectorized math operations on Series, just like on arrays:
  - `s - 5`
  - `s < 0.5`
  - to get data exceeding some threshold: `s[s > 0.5]`
    - get time values with `s[s > 0.5].index`
    - get fluorescence values with `s[s > 0.5].values`
  - math operations work on the data, not on the indices
- you can plot immediately using Series methods, without having to specify x and y args!
  - `s.plot()` - line plot to current MPL axes, or creates new one if none exist
  - use `f, ax = plt.subplots` to prevent overwriting existing figures
    - don't forget to `import matplotlib.pyplot as plt`
  - `s.plot()`, returns an axes, which you can capture with `ax = s.plot()`
    - then you can do the usual axes stuff, like labelling: `ax.set_xlabel()`, etc.
  - besides calling `s.plot()` for a line plot, `s.plot` is also a way to access some other kinds of plots:
    - `s.plot.hist()`, `s.plot.bar()`, `s.plot.area()`, and others
- simple stats as Series methods:
  - `s.min()`, `s.max()`, `s.sum()`, `s.mean()`, `s.median()`, `s.std()`
  - `s.describe()` returns nice summary of several stats
- pandas can handle dates, and date ranges, which can then be used as indices:
  - `dr = pd.date_range('2017-06-01', periods=10, freq='D')`
  - `s3 = pd.Series(data=fl[:10], index=dr)` - fluorescence as a f'n of time in days!
- NOTE: numeric indices need not be in numerical order, they're just a label:

```
t2 = np.array([ 50,  70,  40,  20,  10,  80,  90,  30,  0,  60])
s2 = pd.Series(data=fl[:10], index=t2)
s2.loc[70:10]
```

- indices don't even have to be unique! but that's a bit weird

Series exercise:

1. Load data from two separate numpy arrays : `t.npy` and `V.npy`. `t` is time in sec, and `V` is voltage in mV.

2. Combine them into a pandas Series. Use `t` as the index and `V` as the data.
3. Plot the series as a function of time. Label the x and y axes.
4. Spikes! Choose a voltage threshold (in mV) that separates the spikes from the noise. Apply the threshold to the series to get a printout of only the subset of the data which falls above your threshold.
5. Extract the spike times from the series, and save them to an array called `st`. Save the array to a file named `spike_times.npy`. Now load that file back in, just to make sure it worked.
6. Highlight the spike peaks in your existing figure by plotting over top of them (remember, adding a plot to an existing figure automatically uses a different colour). Highlight just the points, i.e. don't connect them with a line. Hint: use `.plot(ls='', marker='.')` to turn off the line and turn on dot markers (`ls` = `linestyle`). Save the figure to a file named `spikes.png`. If you don't have a save button on your figure, use `ax.get_figure().savefig()`

`pd.DataFrame`

- like a 2D numpy array, but both row and column indices can be non-integers
- other big difference from a 2D array: each column can have its own data type
  - looks and feels a lot like a spreadsheet
  - as for Series, DataFrame row and column indices are really like labels, can be ints, floats, strings...
  - e.g., short segment of (fake) neural EEG voltage data on 3 channels

```
eeg = np.random.random((20, 3)) # 2D array of voltages
t = np.arange(0, 20*50, 50) # timestamps, in ms
chans = ['Fz', 'Cz', 'Pz'] # scalp electrode labels
df = pd.DataFrame(data=eeg, index=t, columns=chans) # label rows with t, column
```

  - having row and column labels, like a spreadsheet, is nice!
  - `df.iloc[:5]` - returns another dataframe of first five rows, same as `df.head()`
  - `df.iloc[0, 0]` - returns entry in 1st row and 1st column, just like 2D array
  - `df.iloc[-1, -1]` - returns entry in last row and last column
  - `df['Fz']` returns a single column, this time as a series, because it's only 1D
  - `df.Fz` can also be used as a shortcut, but is **strongly discouraged** by the developers
  - `df.loc[50]` returns a single row at t=50 ms, also a series
    - in this case, what were column labels in the DataFrame become row labels in the Series
    - now to get a single value, index again using the chan name: `df.loc[50]['Fz']`
      - `.loc` in a DataFrame allows (row, column) indexing similar to numpy
    - direct DataFrame indexing without `.loc`:
      - `df['Fz'][50]` - specify column, then row, opposite of numpy, but same as spreadsheet indexing (i.e., cell A2, C7, etc.)
    - think of a row as an observation, and a column as a variable

- DataFrames can handle more heterogenous data than the above EEG example
    - load some behavioural trial data from a .csv text file into a DataFrame
    - csv = comma separated values
    - open `exp1.csv` in a plain text editor, and then in a spreadsheet progam (LibreOffice Calc, Excel, etc.)
    - each line of text is a row, commas separate the columns
    - first line can be treated as a "header" of column labels
    - `exp1 = pd.read_csv('exp1.csv')`
    - pandas automatically uses the first line as a header to label each column in the DataFrame
    - notice the data types differ across columns, but are consistent within column
    - notice that the rows don't have any particular label, just integers starting from 0
    - what might happen if we try `exp1.plot()`?
        - plots numerical columns as a function of (default) row labels
        - `exp1.plot.hist()` - plots all histograms on top of each other
        - `exp1.hist()` - plots separate histograms
    - let's load a 2nd experiment:
    - `exp2 = pd.read_csv('exp2.csv')`
    - concatenating DataFrames: collect all your data into a single DataFrame
        - very similar to `np.concatenate()` in numpy, but called `pd.concat()` instead
        - vertically (default): `exps = pd.concat([exp1, exp2])`
        - horizontally by using the kwarg `hexps = pd.concat([exp1, exp2], axis=1)` ("across columns")
    - now that we have more data, scatter plot trial start and end times:
        - `exps.plot.scatter('start_time', 'end_time')`
        - compute correlations between all numeric columns: `exps.corr()`
    - sort a DataFrame by values according to a column: `exps.sort_values('start_time')`
- can also load directly from .xlsx files
    - pandas relies on another library for this called `xlrd`, which comes with Anaconda
    - can handle multiple sheets:
    - `exp1 = pd.read_excel('exp.xlsx', sheet_name='exp1')`
    - `exp2 = pd.read_excel('exp.xlsx', sheet_name='exp2')`
- can also save a DataFrame to .csv and .xslx files using methods `.to_csv()` and `.to_excel()`
- DataFrame has same simple stats methods as Series, but now calculated separately for each numerical column:
    - `exps.min()`, `exps.max()`, `exps.sum()`, `exps.mean()`, `exps.median()`, `exps.std()`
    - `exps.describe()` returns separate stats summary for each column
    - `.nunique()` counts number of unique values of a column or Series:
        - `exps['subject'].nunique()`
- Split-Apply-Combine

- Many statistical summaries are in the form of split along some property, then apply a function to each subgroup and finally combine the results into some object. This is known as the 'split-apply-combine' pattern and implemented in Pandas via `groupby()` and a function that can be applied to each subgroup.
- http://people.duke.edu/~ccc14/sta-663/UsingPandas.html
- https://pandas.pydata.org/pandas-docs/stable/groupby.html

- `.groupby()` can be very handy
  - give it column name to "group by", and it finds all the unique values in that column
  - returns a groupby object, with all the same simple stats methods, including `.describe()`, but now tabulated according to the unique values of the chosen column
  - `exps.groupby('outcome').mean()` - returns a DataFrame, which means you can index into its columns or rows like any other
    - e.g. `exps.groupby('outcome').mean()['start_time']`
  - `exps.groupby('outcome').describe()`
  - how can you calculate the duration of each trial?
    - `exps['end_time'] - exps['start_time']`
  - if you want examine trial outcome vs trial duration, need to add duration as a new column:
    - `exps['duration'] = exps['end_time'] - exps['start_time']`
    - `exps.groupby('outcome').mean()` will now show duration as well

- missing data:
  - say you have 2D data, and one data point is missing
  - if you simply leave it out, like this:

```
missd = [[1, 2, 3],
         [4, 5],
         [7, 8, 9]]
```

  - what kind of object is this? try `type(missd)`
  - what happens if you try to convert this list of variable length lists to an array?
    - `a = np.array(missd)`
    - not all the rows are the same length, converting to an array doesn't have any benefit
    - the hint that something is wrong is that `dtype=object` instead of say `dtype=int`
    - `a.shape` is `(3,)` and `a.ndim` is `1`, i.e. this is just a 1D array
    - `a[:, 0]` gives an IndexError, again because it isn't 2D
    - `a` is no better than a list of lists, i.e. can't index into columns, even though `missd` almost looks like a 2D array
  - so, missing data can't simply be left out when creating numpy arrays
  - to represent missing data in numpy, use a placeholder called `np.nan`
  - nan = "not a number"

```
nand = [[1, 2, 3],
        [4, 5, np.nan],
        [7, 8, 9]]
```

    - now converting to an array is useful:
        - `a = np.array(nand)`
        - `a.shape` is `(3, 3)` and `a.ndim` is `2`
        - can index into columns: `a[:, 0]` works
        - but notice that the dtype isn't integer, it's float:
            - `a.dtype` gives `dtype('float64')`
            - this is because `np.nan` is itself a special float value
            - a single `np.nan` forces the whole array to become float, even though all the real values it was given were integers
    - pandas DataFrame deals better with missing data
        - `pd.DataFrame(missd)` and `pd.DataFrame(nand)`
        - now only the one column with the NaN is of type float, the rest remain int
        - any stats exclude missing data, e.g. `pd.DataFrame(missd).mean()`

- ADVANCED: you can generate DataFrames with a hierarchical set of indices using by assigning it a `MultiIndex` instead of just a normal single index:
    - https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html

```
levels = [[-180, -90, 0, 90, 180], [-45, 0, 45]] # set of valid labels for 2 levels
names = ['longitude', 'latitude']
mi = pd.MultiIndex.from_product(levels, names=names)
columns = ['elevation', 'temperature']
df = pd.DataFrame(index=mi, columns=columns) # empty, complex structure ready to be
```

- see course website for two different pandas cheat sheets
- here's a handy 10 min text tutorial
- here's a dense 10 min video tour by Pandas author Wes McKinney

DataFrame exercise: Galton study

This dataset is taken from Francis Galton's 1885 study and explores the relationship between the heights of adult children and the heights of their parents. Found at http://www.math.uah.edu/stat/data/Galton.html

Every row of the csv file `Galton.csv` represents a single child, and the columns describe their parents and family.

1. Load the data in `Galton.csv` into a DataFrame (maybe call it `gdf`). Is there redundant data in this data set?
2. How many children are in this dataset? What variables (columns) are there in this dataset? Are the number of children in the dataset equal to the number of children across all of the families?
3. The column named `Height` describes each child's height. What is the mean child height?
4. Plot a distribution of child heights.

5. How many families are there in this data?

6. Notice there's a weird family named `136A`. Extract the subset of the DataFrame that *excludes* this family. Hint: Use what you know about boolean fancy indexing in numpy and apply it to the DataFrame.

7. What is the mean Father height? The mean Mother height? Note: because parents are repeated in this data, it makes sense to first group the data by family, otherwise parents with more children will be weighed more heavily than parents with fewer children. Hint: I think you have to ask for two means: once within each family, and once more across all families.

8. Is there a relationship between Father and Mother height? Plot them against each other in a scatter plot, and check for a correlation.

9. Is there a relationship between parent height (Father or Mother) and the average height of their children?

Homework 4 (final one), will be due before next class (class 10) on June 23