data analysis with Pandas

- so far, we've seen a few different ways of storing data in Python
    - collections
        - 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 not very user friendly
    - Pandas is a library built on top of numpy that deals with these annoyances, designed to make it easier to handle 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: array, can be 1, 2, 3 or more dimensions
- pandas has two basic object types: Series & DataFrame, 1 and 2 dimensions respectively
- customary name for pandas import is `pd`: `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, others
    - 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.array(np.random.random( 10))
    t = np.arange(0, 1, 0.1) # timestamps, in seconds
    ```

        - 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. trim the data down to just the first 5 data points:

        ```
        trim_fl = fl[:5]
        trim_t = t[:5]
        ```

- another annoyance: say you want to get fluorescence value at a specific timepoint, like t=0.2 seconds
- 2 step process:

```
idx = t == 0.2 # find where t is 0.2, save boolean array to idx
v = fl[idx] # use idx as index into fl, get a float array with one entry
# or in one line:
v = fl[t == 0.2] # also a float array with one entry
v[0] # in either case, to get 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)` - kwarg 'index' means row labels
- now if you want to trim the Series, it's a single command:
- `s.iloc[:5]` for the 1st 5 data points - `.iloc` stands for "integer location"
  - same as `s.head()`
  - `s.tail()` returns last 5 data points
- to get fluorescence at t=0.2 sec, it's a single user-friendly command:
  - `s[0.2]`, or equivalently, `s.loc[0.2]` - `.loc` stands for "location"
- can also slice data directly between non-integer indices:
  - `s[:0.2]` returns all values from start to t=0.2, inclusive
  - `s[0.3:0.7]` does what you'd expect
  - Series slices always return another Series. To get the actual data values out, use `.values`
    - `s[0.3:0.7].values` - returns a normal array of just fluorescence values
- can do vectorized math operations on Series, just like on arrays:
  - `s - 5`
- 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[:0.5].plot()`
  - `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, 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([ 0.5,  0.7,  0.4,  0.2,  0.1,  0.8,  0.9,  0.3,  0. ,  0.6])
s2 = pd.Series( data=fl, index=t2)
s2[0.7:0.1]
```

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

- `pd.DataFrame`
  - like a 2D numpy array, but both row and column indices can be non-integers

- looks and feels a lot like a spreadsheet
- again, indices are more like labels, can be ints, floats, strings
- e.g., short segment of neural EEG voltage data on 3 channels

```
v = np.array(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=v, index=t, columns=chans) # 'index' is rows
```

- `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
- `df.loc[50]` returns a single row at t=50 ms, also a series
- if we want a specific voltage value at a specific channel and timestamp:
  - `df['Fz'][50]` - specify column, then row, opposite of numpy, but same as spreadsheet indexing (i.e., cell A2, C7, etc.)
  - or if you prefer (row, column) indexing: `df.loc[50]['Fz']` gives same result
  - 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
  - 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 file header to label each column in the DataFrame
  - notice the data types differ across columns, but are consistent within column
  - what might happen if we try `exp1.plot()`?
    - plots numerical columns as a function of trials
    - `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 `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', sheetname='exp1')`
  - `exp2 = pd.read_excel('exp.xlsx', sheetname='exp2')`

- can also save a DataFrame to .csv and .xslx files using `.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()`
- `.groupby()` is amazing!
  - 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()`
  - `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, 6],
         [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,)`, i.e. this is just a one dimensional list
    - `a.ndim` is `1`
    - `a[:, 0]` gives an IndexError
    - this is no different from a list of lists, i.e. can't index into columns, even though it looks almost like a 2D array
  - so, missing data can't simply be left out when creating numpy arrays
  - to represent missing data in numpy, can use a placeholder called `np.nan`
  - nan = "not a number"

```
nand = [[1, 2, 3],
        [4, np.nan, 6],
        [7, 8, 9]]
```

  - now converting to an array is useful again:
    - `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)`
        - any stats exclude missing data

- see both pandas cheat sheets