Automating data wrangling

Sometimes we require a "one off" solution to a unique data analysis problem. In this situation, we write code to do a particular analysis on a particular data set. Then, if the analysis is part of a publication, we make the code and data publically available and... we're done.

Often, however, we require a ***reusable*** solution that operates on data of a given format even though some of the particulars, such as sample size or variable names, might change. In this case, we want our code to be "dynamic" in the sense that it should be able to handle any anticipated changes to the details of the input data.

Here, we'll tackle the same problem as last time – reformatting a data set from a cumbersome format into a more useful and "tidy" format.

Learning goals:

- write reusable code for a data wrangling problem
- create a function to make the code handy to use

Import pandas and look at the data from last time

In [1]:
```
import pandas as pd
```

Read in the data from last time.

In [2]:
```
my_input_data = pd.read_csv('datasets/017DataFile.csv')
```

Take a peek to remind ourselves of the data format.

In [3]:
```
1 my_input_data.head()
```

Out[3]:

	Male Mutant	Female Mutant	Male Wild Type	Female Wild Type
0	10.485451	8.250013	20.127063	25.946384
1	11.747948	8.453839	20.068147	23.464870
2	13.412580	9.706605	21.215148	22.989480
3	12.910095	9.522116	20.706416	25.324376
4	10.367770	8.583212	18.074795	22.607487

In this data set, there are two "independent variables", sex and genotype of laboratory rats, and one "dependent variable", response time. The data are formatted such that each column contains the data from a unique combination of the two independent variables, *i.e.* a "cell" of the experimental design. Like this:

	male	**female**
mutant	mm	fm
wildtype	mw	fw

This format might seem to make sense, but it's actually not very flexible. For analysis purposes, it's generally better to have data in a format that obeys a couple of rules:

- *each row should correspond to a single observation (measurement)*
- *each column should correspond to a single variable*

Data in this format are also referred to as "tidy".

So in this case, our goal is to take the above data and put it into a format like this:

response time	sex	genotype
rt value	male or female	wild or not

Once the data are in this format, we can easily use our tools to do things like compare wild to mutant, or compare wild to mutant only in females, etc.

Last time, we stacked the reaction time values into a single column using pandas functions. This relied on us knowing and "hard coding" the column names ("Male Mutant", etc.). If we're going to automate things, we want our code to be agnostic about these. One way would be to somehow read the column names into variables and work with them somehow...

But what about numpy arrays? We already know how to manipulate those and, since they are just numbers, there are no column names or pesky row indexes to worry about. So let's try using numpy!

In [4]:
```
import numpy as np
```

Pandas dataframes know how to convert themselves to numpy arrays. They have a `to_numpy()` method that will pull *just the numbers* out of our dataframe, ignoring the column labels and row indexs.

In [5]:
```
raw_data = my_input_data.to_numpy()
```

Let's take a look!

In [6]:
```
raw_data
```

Out[6]:
```
array([[10.48545088,  8.2500131 , 20.12706278, 25.94638414],
       [11.74794775,  8.45383932, 20.06814699, 23.46487013],
       [13.41258004,  9.70660484, 21.21514789, 22.98948034],
       [12.91009526,  9.52211638, 20.70641578, 25.32437595],
       [10.36777045,  8.58321246, 18.07479515, 22.60748688],
       [11.69842177,  9.83500171, 20.36762403, 23.05218737],
       [11.58315277, 10.53209602, 20.15252058, 25.3690367 ],
       [11.44734892,  9.39416641, 19.39247581, 23.37270897],
       [10.85227619,  8.73947266, 18.52434071, 25.21564644],
       [11.28589742, 10.89239399, 20.32502629, 24.99050453]])
```

Get some useful information from the original data

So far so good! Now we are going to put the data into the format we want. To automate this, we are going to get

- the number of observations in each group (which is the number of rows), and
- the number of groups (which is the number of columns)

and store them in variables.

In [8]:
```
obs_per_grp, grps = raw_data.shape
print("We have ", obs_per_grp, " observations per group and ", grps, " groups.")
# obs_per_grp = number of rows
# grps = number of columns
```

```
We have  10  observations per group and  4  groups.
```

Now we'll calculate the total number of observations, which is also how long we want our new data frame to be.

In [9]:
```
new_length = obs_per_grp*grps
print("We have ", new_length, " total observations.")
```

```
We have  40  total observations.
```

Complete the following exercise.

- Use the cell below and explain in your own words why we used Numpy Arrays in the previous cells. What was our final goal? Why did we dump the data into a Numpy Array?

We use `to_numpy()` to see the raw data without looking at the index from the original file. So, we can look at the overal data regardless of the index.

Build our response time (dependent variable) column

We could now play legos "by hand", stacking the columns of our numpy array on top of each other to make a new array (and we already know how to do that).

Or we could take advantage of the fact that one of the things numpy arrays know how to do – one of the methods they have – is to change their shape. So we'll take our `obs` by `cols` array and `numpy.reshape()` into a `new_length` by 1 array.

What this command does (effectively) is read out the data values from the original array one-by-one, and places them in the cells of a new array of a shape you specify. The only catch is that the total number of cells in the new array has to be the same as in the old array – in other words, each and every data value has to have one and only one place to go in the new array. Which makes sense.

In [12]:
```
values_col = np.reshape(raw_data, (new_length, 1))
```

I called it `values_col` because it will eventually become the values column of our new pandas data frame.

Let's see if that worked:

In [13]:
```
values_col
```

Out[13]:
```
array([[10.48545088],
       [ 8.2500131 ],
       [20.12706278],
```

```
[25.94638414],
[11.74794775],
[ 8.45383932],
[20.06814699],
[23.46487013],
[13.41258004],
[ 9.70660484],
[21.21514789],
[22.98948034],
[12.91009526],
[ 9.52211638],
[20.70641578],
[25.32437595],
[10.36777045],
[ 8.58321246],
[18.07479515],
[22.60748688],
[11.69842177],
[ 9.83500171],
[20.36762403],
[23.05218737],
[11.58315277],
[10.53209602],
[20.15252058],
[25.3690367 ],
[11.44734892],
[ 9.39416641],
[19.39247581],
[23.37270897],
[10.85227619],
[ 8.73947266],
[18.52434071],
[25.21564644],
[11.28589742],
[10.89239399],
[20.32502629],
[24.99050453]])
```

Nice! But let's make absolutely sure that worked. What we want is for the columns of the original data to be stacked on top of one another. Is that what we have?

Nope, it's not right. What happened is that the values got read out *left to right, top to bottom* (or row-wise) and placed into the new array one-by-one. But what we want is for the values to be read *top to bottom, left to right* (or columnwise). We can make this happen with the `order=` argument of `numpy.reshape()` .

In [14]:
```
values_col = np.reshape(raw_data, (new_length, 1), order = 'F')
```

Let's make sure that worked:

In [15]:
```
values_col
```

Out[15]:
```
array([[10.48545088],
       [11.74794775],
       [13.41258004],
       [12.91009526],
       [10.36777045],
       [11.69842177],
       [11.58315277],
       [11.44734892],
       [10.85227619],
       [11.28589742],
       [ 8.2500131 ],
       [ 8.45383932],
       [ 9.70660484],
       [ 9.52211638],
       [ 8.58321246],
       [ 9.83500171],
       [10.53209602],
       [ 9.39416641],
       [ 8.73947266],
       [10.89239399],
       [20.12706278],
       [20.06814699],
       [21.21514789],
       [20.70641578],
```

```
       [18.07479515],
       [20.36762403],
       [20.15252058],
       [19.39247581],
       [18.52434071],
       [20.32502629],
       [25.94638414],
       [23.46487013],
       [22.98948034],
       [25.32437595],
       [22.60748688],
       [23.05218737],
       [25.3690367 ],
       [23.37270897],
       [25.21564644],
       [24.99050453]])
```

Yay! It did!

Useless trivia: Two of Ye Olde Major Programming Languages are **C** (used mainly by programmers) and **Fortran** (used mainly by scientists). C (the language used to write Python) uses row-wise indexing, whereas Fortran uses columnwise indexing. That's why "F" is used to specify columnwise indexing above: the "F" is for "Fortran".

Minor annoying thing: (there is always at least one that pops up in any coding task, amirite?) `values_col` is a (40x1) 2-dimensional numpy array but, when we go to build our new data frame, we'll need it to be a 40 long (40,) 1-dimensional array.

This actually comes up so often that `numpy` has a `squeeze()` function to squeeze the dimension of length one into nothingness. It turns (n, 1) things into (n,) things.

Let's check the shape of our new array:

In [16]:
```
1 values_col.shape
```

Out[16]:
```
(40, 1)
```

Now let's squeeze the (uneeded and unwanted) column dimension into oblivion:

In [17]:
```
values_col = np.squeeze(values_col)
```

And check the shape again:

In [18]:
```
values_col.shape
```

Out[18]:
```
(40,)
```

Okay, that worked, now onto...

Complete the following exercise.

- Use the next cell to explaing what happened to the numpy array after the squeeze operation

It is now one long set of data arranged from each column: column 1 data -> column 2 data -> and so on.

- Type below code demonstrating how you could explore the help for the method `.shape()` to explore what it does:

In [20]:
```
.shape?
```

```
  Input In [20]
    .shape?abs
    ^
SyntaxError: invalid syntax
```

- Use the cell below to explain the use of the method `.reshape()`:

`.reshape()` changes the order of the data from the original data set.

Building the independent variable columns

What we require is that the levels our two independent variables repeat themselves in the right order down their respective columns. We could certainly type this in by hand, but that would be really annoying to change if we required new labels later on or something.

We could also use `for()` loops; they are designed for exactly such repetitive tasks after all. That might look something like this:

In [23]:
```
gen_var = list()                        # create a python list
for i in range(new_length) :            # loop through all observations
    if i < new_length/2 :               # for the first half, ...
        gen_var.append("wildtype")      # set to wildtype
    else :                              # otherwise...
        gen_var.append("mutant")        # set to mutant
```

In [24]:
```
print(gen_var)
```

```
['wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype'
, 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'wildtype
', 'wildtype', 'wildtype', 'wildtype', 'wildtype', 'mutant', 'mutant', 'mutant', 'mutant', 'mut
ant', 'mutant', 'mutant', 'mutant', 'mutant', 'mutant', 'mutant', 'mutant', 'mutant', 'mutant',
'mutant', 'mutant', 'mutant', 'mutant', 'mutant', 'mutant']
```

We'd have to get a little bit more fancy with our `if...` to create the sex variable, that'd be the idea.

But pandas provides easy ways to repeat and stack things (numpy does too), so let's try those. The two will use are

- `pandas.Series.repeat()`
- `pandas.concat()`

Note: When you see `pandas.Series.somefunction()` or `pandas.DataFrame.somefunction()` in the documentation, that means that all Series or DataFrames know how to do `somefunction()`. So if you had a Series named `Phred`, you would say `Phred.somefunction()` to use `somefunction()`.

Complete the following exercise.

- Use the cell below to explain what the variable `new_length` contain:

The numpy data without the `.reshape()` function.

- Use the cell below to explain the reason why we use `new_length/2` in combination with the `if, else`:

The `if, else` fucntion combine with `new_length/2` indicates that the first half of the data in `new_length` to set as `wild type` and the second half is `mutant`.

Make the genetic strain variable

In the way we have formatted the data, genetic strain is the "outer" variable, in that it only changes once as we go down the data set: all the wildtypes are on top, and all mutants are on the bottom. The sex variable is the "inner" variable, because it changes once within each value of strain, so it needs to three times as we go down the data set.

This is arbitrary and has nothing to do with the experimental design; we could have formatted the data such that the roles were reversed.

What we will do is

- make a short series containing the two levels of our variable
- repeat each value to make the long series
- deal with annoying index values (there's always something...)

In [25]:
```
strain = pd.Series(['wildtype', 'mutant'])  # make the short series
strain = strain.repeat(2*obs_per_grp)       # repeat each over two cell's worth of data
strain = strain.reset_index(drop=True)      # reset the series's index value
```

Complete the following exercise.

- Use the cell below to explain what is and what it is contained by the variable `strain` :

It contains the set of data from the `if, else` loop: wild type and mutant.

Let's see if that worked:

In [28]:
```
print(strain)
```

```
0     wildtype
1     wildtype
2     wildtype
3     wildtype
4     wildtype
5     wildtype
6     wildtype
7     wildtype
8     wildtype
9     wildtype
10    wildtype
11    wildtype
12    wildtype
13    wildtype
14    wildtype
```

```
15    wildtype
16    wildtype
17    wildtype
18    wildtype
19    wildtype
20      mutant
21      mutant
22      mutant
23      mutant
24      mutant
25      mutant
26      mutant
27      mutant
28      mutant
29      mutant
30      mutant
31      mutant
32      mutant
33      mutant
34      mutant
35      mutant
36      mutant
37      mutant
38      mutant
39      mutant
dtype: object
```

Complete the following exercise.

- Use the cell below to explain why `mutants` appear at the bottom of the previous `Pandas Series`, who decided that order?

Because we set `new_length/2` (first half of the data) to be `wild type` and the second half to be `mutant`.

Make the sex variable

As the sex variable is the inner variable, we need it have `['male'..., 'female'...]` within each outer block of genotype. So what we'll do is make one block of `['male'..., 'female'...]` and then just stack two copies of that to make our variable. So the steps are

- make a short series containing the two levels of our variable (just like above)
- repeat it (just like above)
- stack two copies on top of each other (dropping the annoying indexes in the process)

In [29]:
```
sexes = pd.Series(['male', 'female'])              # make the short series
sexes = sexes.repeat(obs_per_grp)                  # repeat each over one cell's worth of data
sexes = pd.concat([sexes]*2, ignore_index=True)    # stack or "concatonate" two copies
```

In [30]:
```
print(sexes)
```

```
0       male
1       male
2       male
3       male
4       male
5       male
6       male
7       male
8       male
9       male
10    female
11    female
12    female
13    female
14    female
15    female
16    female
17    female
18    female
19    female
20      male
21      male
22      male
```

```
23      male
24      male
25      male
26      male
27      male
28      male
29      male
30    female
31    female
32    female
33    female
34    female
35    female
36    female
37    female
38    female
39    female
dtype: object
```

Complete the following exercise.

- Use the cell below to explain in your own words what happened in the previous cell:

We set the data to have male and female.

- Use the cell below to show your code to create a pandas series called `unicorns` comprising of 20 mistical equines half of which are `white` and half `pearl-white` in color (well ... what what do you want, they are unicorns):

In [34]:
```
unicorns = pd.Series(['white','pearl-white'])           # make the short series
unicorns = unicorns.repeat(obs_per_grp)                 # repeat each over one cell's worth
unicorns = pd.concat([unicorns], ignore_index=True)   # stack or "concatonate" two copies
```

In [35]:
```
unicorns
```

```
2           white
3           white
4           white
5           white
6           white
7           white
8           white
9           white
10    pearl-white
11    pearl-white
12    pearl-white
13    pearl-white
14    pearl-white
15    pearl-white
16    pearl-white
17    pearl-white
18    pearl-white
19    pearl-white
dtype: object
```

- Use the cell below to show your code to create a pandas series called `Three trees` comprising of 30 trees 1/3 of which are `Live Oaks` , 1/3 `White Oaks` and 1/3 `Red Oaks` :

In [37]:
```
trees = pd.Series(['Live Oaks', 'White Oaks', 'Red Oaks'])
trees = trees.repeat(obs_per_grp)
trees = pd.concat([trees], ignore_index=True)
```

In [38]:
```
1 trees
```

Out[38]:
```
0      Live Oaks
1      Live Oaks
2      Live Oaks
3      Live Oaks
4      Live Oaks
5      Live Oaks
6      Live Oaks
7      Live Oaks
8      Live Oaks
9      Live Oaks
10    White Oaks
11    White Oaks
12    White Oaks
13    White Oaks
14    White Oaks
15    White Oaks
16    White Oaks
17    White Oaks
18    White Oaks
19    White Oaks
20      Red Oaks
21      Red Oaks
22      Red Oaks
23      Red Oaks
24      Red Oaks
25      Red Oaks
26      Red Oaks
27      Red Oaks
28      Red Oaks
29      Red Oaks
dtype: object
```

Build our new data frame!

Data frames are created in pandas by handing it data it can make sense of. There are various ways to accomplish this, and one handy one is to hand it data in a "column label 1 : data 1, column label 2 : data 2, ..." format.

We can accomplish this with a python "dictionary" (remember those?). A python `dict` associates a label (the "word") with a value or set of values or whatever (the "definition"). They are very useful, so let's take a look at a simple example before we use one to build out data frame. You create a dictionary using curly braces, and then use colons to bind each word or `key` with its definition or `value`. Commas separate each key-value pair.

In [39]:
```
myData = {"name": "Larry", "rank": "full", "years": 30, "bikes": 5, "motorcycles": 2, "teslas
```

In [40]:
```
myData["name"]
```

Out[40]:
```
'Larry'
```

In [41]:
```
myData["bikes"]
```

Out[41]:
```
5
```

Complete the following exercise.

- Use the cell below to build a `dict()` describing a student, with a name, with a student ID, a GPA and a major, make up all the values but use the lables as described here:

In [43]:
```
myData2 = {"Name": "Jay", "UT EID": "JT43227", "GPA": 4.00, "Major": "B.S, Psychology"}
```

In [44]:
```
myData2
```

Out[44]:
```
{'Name': 'Jay', 'UT EID': 'JT43227', 'GPA': 4.0, 'Major': 'B.S, Psychology'}
```

So a dictionary associates a label with data values. **Perfect!**

Time to build our data frame!

In [45]:
```
my_tidy_data = pd.DataFrame(      # invoke creation
    {                             # start the dictionary with a {
        "RTs": values_col,        # assign each variable to a label
        "sex": sexes,
        "strain": strain
    }                             # end the dictionary with a }
)                                 # end of creation
```

Note that the formatting above is just to make the columns we're creating more obvious and human-readable. This will work too:

In [46]:
```
my_tidy_data = pd.DataFrame({"RTs": values_col, "sex": sexes, "strain": strain})
```

It's just not as pretty.

Let's look at our creation!

In [47]:
```
my_tidy_data
```

Out[47]:

	RTs	sex	strain
0	10.485451	male	wildtype
1	11.747948	male	wildtype
2	13.412580	male	wildtype
3	12.910095	male	wildtype
4	10.367770	male	wildtype
5	11.698422	male	wildtype
6	11.583153	male	wildtype
7	11.447349	male	wildtype
8	10.852276	male	wildtype
9	11.285897	male	wildtype

10	8.250013	female	wildtype
11	8.453839	female	wildtype
12	9.706605	female	wildtype
13	9.522116	female	wildtype
14	8.583212	female	wildtype
15	9.835002	female	wildtype
16	10.532096	female	wildtype
17	9.394166	female	wildtype
18	8.739473	female	wildtype
19	10.892394	female	wildtype
20	20.127063	male	mutant
21	20.068147	male	mutant
22	21.215148	male	mutant
23	20.706416	male	mutant
24	18.074795	male	mutant
25	20.367624	male	mutant
26	20.152521	male	mutant
27	19.392476	male	mutant
28	18.524341	male	mutant
29	20.325026	male	mutant
30	25.946384	female	mutant
31	23.464870	female	mutant
32	22.989480	female	mutant
33	25.324376	female	mutant
34	22.607487	female	mutant
35	23.052187	female	mutant

36	25.369037	female	mutant
37	23.372709	female	mutant
38	25.215646	female	mutant
39	24.990505	female	mutant

Yay! We win!

Important point: Crucially, *the above code doesn't rely on us knowing much about the input data ahead of time*. As long as it's a pandas data frame that contains numerical values, the code will run. It's automatic.

Look at new data with more observations with same code

We'll make this code self-contained, so it can be run without running anything above. We'll also add comments, so that future-us can read the code more easily without having to wade through the notebook text above.

In [49]:
```
my_input_data = pd.read_csv('datasets/018DataFile.csv')  # read the data

raw_data = my_input_data.to_numpy()                      # convert to numpy array

obs, grps = raw_data.shape                               # get the number of rows and columns
```

Check the size of the new data real quick:

In [50]:
```
print("We have ", obs, " observations per group and ", grps, " groups.")
```

```
We have  20  observations per group and  4  groups.
```

And now run the "meat" of the code:

In [63]:

```
new_length = obs*grps                                   # compute total number of observatio

values_col = np.reshape(raw_data, (new_length, 1),
                        order = 'F')                    # reshape the array
values_col = np.squeeze(values_col)                     # squeeze to make 1D

# construct the inner grouping variable
sexes = pd.Series(['male', 'female'])                   # define the levels
sexes = sexes.repeat(obs)                               # make one cycle of the levels
sexes = pd.concat([sexes]*2, ignore_index=True)         # and repeat the cycle, ditching the

# construct the outer grouping variable
strain = pd.Series(['wildtype', 'mutant'])              # define the levels
strain = strain.repeat(2*obs)                           # make the one cycle
strain = strain.reset_index(drop=True)                  # drop the pesky index

# construct the data frame
my_new_tidy_data = pd.DataFrame(
    {
        "RTs": values_col,                              # make a column named RTs and put th
        "sex": sexes,                                   # ditto for sex
        "strain": strain                                # and for genetic strain
    }
)
```

In [64]:
```
1 my_new_tidy_data
```

Out[64]:

	RTs	sex	strain
0	12.333785	male	wildtype
1	11.675152	male	wildtype
2	12.029059	male	wildtype
3	12.126430	male	wildtype
4	10.307197	male	wildtype
...	...	...	...
75	24.886821	female	mutant
76	24.475663	female	mutant
77	21.935896	female	mutant
78	23.852748	female	mutant
79	25.515138	female	mutant

80 rows × 3 columns

Success!

Making the code even more functional

Now we have a chunk of code that seems handy and re-usable. How could we make it ever more handy?

If we make it into a ***function***, then we can run the whole entire thing just by typing one command – no copying, no pasting, fewer ways to make mistakes.

Defining a function

Since we already have all the code, we can literally just indent it and throw a `def...` in front of it!

In [65]:

```
def tidyMyData() :
    import pandas as pd
    import numpy as np

    my_input_data = pd.read_csv('datasets/018DataFile.csv')  # read the data

    raw_data = my_input_data.to_numpy()                      # convert to numpy array

    obs, grps = raw_data.shape                               # get the number of rows and co

    new_length = obs*grps                                    # compute total number of obser

    values_col = np.reshape(raw_data, (new_length, 1),
                            order = 'F')                     # reshape the array
    values_col = np.squeeze(values_col)                      # squeeze to make 1D

    # construct the inner grouping variable
    sexes = pd.Series(['male', 'female'])                    # define the levels
    sexes = sexes.repeat(obs)                                # make one cycle of the levels
    sexes = pd.concat([sexes]*2, ignore_index=True)     # and repeat the cycle, ditching the

    # construct the outer grouping variable
    strain = pd.Series(['wildtype', 'mutant'])               # define the levels
    strain = strain.repeat(2*obs)                            # make the one cycle
    strain = strain.reset_index(drop=True)                   # drop the pesky index

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            "RTs": values_col,                               # make a column named RTs and pu
            "sex": sexes,                                    # ditto for sex
            "strain": strain                                 # and for genetic strain
        }
    )

    return my_new_tidy_data
```

In [54]:

```
datFromFun = tidyMyData()
```

In [55]:
```
1 datFromFun
```

Out[55]:

	RTs	sex	strain
0	12.333785	male	wildtype
1	11.675152	male	wildtype
2	12.029059	male	wildtype
3	12.126430	male	wildtype
4	10.307197	male	wildtype
...	...	...	...
75	24.886821	female	mutant
76	24.475663	female	mutant
77	21.935896	female	mutant
78	23.852748	female	mutant
79	25.515138	female	mutant

80 rows × 3 columns

Defining a function with an argument

A common (very common) scenario in data analysis is wanting to run the same code – like the code we just wrote – on different files. So one really nice addition to this function would be to add the ability for the user to specify a filename to tell the function which data file to read.

This is actually fairly straightforward. All we have to do as add an **argument** to our function, and then replace the hardcoded filename in the function with the **variable** created by the function argument.

In [68]:

```
def tidyMyData(filename) :
    import pandas as pd
    import numpy as np

    my_input_data = pd.read_csv(filename)  # read the data

    raw_data = my_input_data.to_numpy()                          # convert to numpy array

    obs, grps = raw_data.shape                                   # get the number of rows and co

    new_length = obs*grps                                        # compute total number of obser

    values_col = np.reshape(raw_data, (new_length, 1),
                            order = 'F')                         # reshape the array
    values_col = np.squeeze(values_col)                          # squeeze to make 1D

    # construct the inner grouping variable
    sexes = pd.Series(['male', 'female'])                        # define the levels
    sexes = sexes.repeat(obs)                                    # make one cycle of the levels
    sexes = pd.concat([sexes]*2, ignore_index=True)     # and repeat the cycle, ditching the

    # construct the outer grouping variable
    strain = pd.Series(['wildtype', 'mutant'])                   # define the levels
    strain = strain.repeat(2*obs)                                # make the one cycle
    strain = strain.reset_index(drop=True)                       # drop the pesky index

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            "RTs": values_col,                                   # make a column named RTs and pu
            "sex": sexes,                                        # ditto for sex
            "strain": strain                                     # and for genetic strain
        }
    )

    return my_new_tidy_data
```

Now we can call the function and specify whatever data files exist. Let's try it with "datasets/018DataFile2.csv"!

In [69]:
```
newDataFromFun = tidyMyData("datasets/018DataFile2.csv")
```

In [70]:
```
newDataFromFun
```

Out[70]:

	RTs	sex	strain
0	12.577226	male	wildtype
1	12.778183	male	wildtype
2	13.389130	male	wildtype
3	12.747877	male	wildtype
4	13.615121	male	wildtype
...	...	...	...
163	24.539374	female	mutant
164	23.877924	female	mutant
165	23.161896	female	mutant
166	24.426455	female	mutant
167	21.990136	female	mutant

168 rows × 3 columns

Adding help

It's always a good idea to **heavily comment your code!**

When writing fuctions, it's also a good idea to add a documentation string, called a `docstring`, to your function. This way people can get help on your function with the `help()` function. Like `help(tidyMyData)`.

In [73]:
```
def tidyMyData(filename) :
    '''
    tidyMyData() Takes one-column-per-cell rat reaction time data as input.
    Returns tidy one-column-per-variable data.
```

```
    User specifies a filename string.
    '''

    import pandas as pd
    import numpy as np

    my_input_data = pd.read_csv(filename)  # read the data

    raw_data = my_input_data.to_numpy()                          # convert to numpy array

    obs, grps = raw_data.shape                                   # get the number of rows and co

    new_length = obs*grps                                        # compute total number of obser

    values_col = np.reshape(raw_data, (new_length, 1),
                            order = 'F')                         # reshape the array
    values_col = np.squeeze(values_col)                          # squeeze to make 1D

    # construct the inner grouping variable
    sexes = pd.Series(['male', 'female'])                        # define the levels
    sexes = sexes.repeat(obs)                                    # make one cycle of the levels
    sexes = pd.concat([sexes]*2, ignore_index=True)              # and repeat the cycle, ditchin

    # construct the outer grouping variable
    strain = pd.Series(['wildtype', 'mutant'])                   # define the levels
    strain = strain.repeat(2*obs)                                # make the one cycle
    strain = strain.reset_index(drop=True)                       # drop the pesky index

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            "RTs": values_col,                                   # make a column named RTs and pu
            "sex": sexes,                                        # ditto for sex
            "strain": strain                                     # and for genetic strain
        }
    )

    return my_new_tidy_data
```

In [72]:
```
help(tidyMyData)
```

```
Help on function tidyMyData in module __main__:

tidyMyData(filename)
    tidyMyData() Takes one-column-per-cell rat reaction time data as input.
    Returns tidy one-column-per-variable data.
    User specifies a filename string.
```

Complete the following exercise.

- Use the cell below to show how you would modify the previous function so as to make it even more flexible. Let the user specify the output column headers to be whatever they want.

More specifically how would you allow passing in the three labels, `sex`, `RTs` and `strain`, instead of having them 'hard coded' inside the code. This means that instead of using labels such as `sex`, `RTs` and `strain`, we will want to pass paramters for each one of the labels and use the parameters in the function. For example, instead of `sex`, `RTs` and `strain` we will want to pass others say, `s`, `ReactionTime` or `type` or any three combinations of lables, always three but that can change everytime we call the function.

You would do this with arguments (obviously). But you could do it with multiple arguments, so users would call it like:

```
tidyMyData("datasets/018DataFile2.csv", "Times", "Gender", "Genotype")
```

or you could do it with one additional arguments, so the user would call it by either:

```
tidyMyData("datasets/018DataFile2.csv", ["Times", "Gender", "Genotype"])
```

or

```
colNames = ["Times", "Gender", "Genotype"]

tidyMyData("datasets/018DataFile2.csv", colNames)
```

Pro tip: The function would probably be most handy if there were *default* values for the column names, so that user could just type something like

```
myTidyData = tidyMyData("datasets/018DataFile2.csv")
```

if they didn't want to specify custom column headers.

In [74]:
```
1 myTidyData2 = tidyMyData("datasets/018DataFile2.csv")
```

In [75]:
```
1 myTidyData2.head
```

Out[75]:
```
<bound method NDFrame.head of           RTs     sex    strain
0    12.577226    male  wildtype
1    12.778183    male  wildtype
2    13.389130    male  wildtype
3    12.747877    male  wildtype
4    13.615121    male  wildtype
..         ...     ...       ...
163  24.539374  female    mutant
164  23.877924  female    mutant
165  23.161896  female    mutant
166  24.426455  female    mutant
167  21.990136  female    mutant

[168 rows x 3 columns]>
```

In [77]:

```
# Rename the columns
df1 = myTidyData2.rename(columns={'RTs': 'Reaction Times', 'sex': 'Genders','strain':'Genotyp
df1
```

Out[77]:

	Reaction Times	Genders	Genotype
0	12.577226	male	wildtype
1	12.778183	male	wildtype
2	13.389130	male	wildtype
3	12.747877	male	wildtype
4	13.615121	male	wildtype
...	...	...	...
163	24.539374	female	mutant
164	23.877924	female	mutant
165	23.161896	female	mutant
166	24.426455	female	mutant
167	21.990136	female	mutant

168 rows × 3 columns

In [96]:

```
def tidyMyData(df1) :
    '''
    tidyMyData() Takes one-column-per-cell rat reaction time data as input.
    Returns tidy one-column-per-variable data.
    User specifies a filename string.
    '''

    import pandas as pd
    import numpy as np

    my_input_data = df1                                    # read the data

    raw_data = my_input_data.to_numpy()                    # convert to numpy array
```

```
    obs, grps = raw_data.shape                                  # get the number of rows and co
                                                                
    new_length = obs*grps                                       # compute total number of obser
                                                                
    values_col = np.reshape(raw_data, (new_length, 1),          
                            order = 'F')                        # reshape the array
    values_col = np.squeeze(values_col)                         # squeeze to make 1D
                                                                
    # construct the inner grouping variable                     
    sexes = pd.Series(['male', 'female'])                       # define the levels
    sexes = sexes.repeat(obs)                                   # make one cycle of the levels
    sexes = pd.concat([sexes]*2, ignore_index=True)             # and repeat the cycle, ditchin
                                                                
    # construct the outer grouping variable                     
    strain = pd.Series(['wildtype', 'mutant'])                  # define the levels
    strain = strain.repeat(2*obs)                               # make the one cycle
    strain = strain.reset_index(drop=True)                      # drop the pesky index
                                                                
    # construct the data frame                                  
    my_new_tidy_data2 = pd.DataFrame(                           
        {                                                       
            "Reaction Times": values_col,                        # make a column named RTs and 
            "Gender": sexes,                                     # ditto for sex
            "Genotype": strain                                   # and for genetic strain
        }                                                       
    )                                                           
                                                                
    return my_new_tidy_data2
```