# Automating data wrangling

## Introduction

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

Read in the data from last time.

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

Take a peek to remind ourselves of the data format.

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

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

Let's take a look!

In [ ]:
```
raw_data
```

# 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 [ ]:
```
obs_per_grp, grps = raw_data.shape
print("We have ", obs_per_grp, " observations per group and ", grps, " grou
```

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

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

# 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 [ ]:
```
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 [ ]:
```
values_col
```

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

Let's make sure that worked:

In [ ]:
```
values_col
```

**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 [ ]:
```
values_col.shape
```

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

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

And check the shape again:

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

Okay, that worked, now onto...

## 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 [ ]:
```
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 male
    else :                              # otherwise...
        gen_var.append("mutant")        # set to female
```

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

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()`.

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

Let's see if that worked:

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

## 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 [ ]:
```
sexes = pd.Series(['male', 'female'])              # make the short series
sexes = sexes.repeat(obs_per_grp)                  # repeat each over one ce
sexes = pd.concat([sexes]*2, ignore_index=True)    # stack or "concatonate"
```

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

## 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", which is a thing 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 [ ]:
```
myData = {"name": "Larry", "rank": "full", "years": 30, "bikes": 5, "motorc
```

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

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

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

Time to build our data frame!

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

It's just not as pretty.

Let's look at our creation!

In [ ]:
```
my_tidy_data
```

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 [1]:
```
# import our libraries
import pandas as pd
import numpy as np

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

raw_data = my_input_data.to_numpy()                        # convert to numpy

obs, grps = raw_data.shape                                 # get the number o
```

Check the size of the new data real quick:

In [2]:
```
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 [3]:
```
new_length = obs*grps                                   # compute total nu

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

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

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

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

In [4]:
```
my_new_tidy_data
```

Out[4]:

| | 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 [7]:
```
def tidyMyData() :
    import pandas as pd
    import numpy as np

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

    raw_data = my_input_data.to_numpy()                          # convert to n

    obs, grps = raw_data.shape                                   # get the numb

    new_length = obs*grps                                        # compute tota

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

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

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

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            "RTs": values_col,                                   # make a colum
            "sex": sexes,                                        # ditto for se
            "strain": strain                                     # and for gene
        }
    )

    return my_new_tidy_data
```

In [8]:
```
datFromFun = tidyMyData()
```

In [9]:
```
datFromFun
```

Out[9]:

| | 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 [10]:
```
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 n

    obs, grps = raw_data.shape                                   # get the numb

    new_length = obs*grps                                        # compute tota

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

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

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

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            "RTs": values_col,                                   # make a colum
            "sex": sexes,                                        # ditto for se
            "strain": strain                                     # and for gene
        }
    )

    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 [11]:
```
newDataFromFun = tidyMyData("018DataFile2.csv")
```

In [12]:
```
newDataFromFun
```

Out[12]:

| | 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 [13]:
```
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 n

    obs, grps = raw_data.shape                                   # get the numb

    new_length = obs*grps                                        # compute tota

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

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

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

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            "RTs": values_col,                                   # make a colum
            "sex": sexes,                                        # ditto for se
            "strain": strain                                     # and for gene
        }
    )

    return my_new_tidy_data
```

In [14]:
```
help(tidyMyData)
```

```
Help on function tidyMyData in module __main__:

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

# Coding Challenge!

Modify our function to make it even more flexible. Let the user specify the output column headers to be whatever they want.

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.

Have at it!

In [25]:
```
def tidyMyData(filename,colnames) :
    '''
    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

    colnames = colnames

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

    raw_data = my_input_data.to_numpy()                          # convert to n

    obs, grps = raw_data.shape                                   # get the numb

    new_length = obs*grps                                        # compute tota

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

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

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

    # construct the data frame
    my_new_tidy_data = pd.DataFrame(
        {
            colnames[0]: values_col,                                 # make a
            colnames[1]: sexes,                                      # ditto
            colnames[2]: strain                                    # and for g
        }
    )

    return my_new_tidy_data
```

In [27]:
```
newDataFromFu = tidyMyData("018DataFile2.csv",["Times", "Gender", "Genotype
```

In [28]:
```
newDataFromFu
```

Out[28]:

| | Times | Gender | 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