Seaborn overview

We have seen before that `matplotlib` is powerful and flexible, but that power and flexibility means that there is also a lot to learn!

`Seaborn` is meant to provide a facilitatated access to plotting data. Just like `Pandas` sits on top of `Numpy` to facilitate object oriented data applications, `seaborn` sits on top of `Pandas` to accelarate data exploration via visualiation.

Learning goals

- `seaborn` - overview of functionality
- Plot distributions and histograms
- Explore data using visualization

Prerequisites

- Python and NumPy
- Pandas, DataFrames and Series

`Seaborn` was written to:

- make plots from `pandas` data frames
- create good looking plots "out of the box"

The `seaborn` package (which we've already used some) is a "high level" plotting package that calls various `matplotlib` functions for you while taking care of many details for you under the hood. The various `seaborn` functions are conceptually structured like this:

The three columns correspond to plot types: plots of relationships, plots of data distributions, and plots of categorical data.

For each plot type, there is a "figure level" function, `relplot()`, `displot()`, and `catplot()`. The main advantage of these is that they make it easy to create figures with multiple axes on them.

In addition to the figure level functions, there are specific "axes level" functions for making each specific kind of plot directly. Each of these returns an `axes` object, which you can then modify if necessary just as you would had you created it with `plt.plot()`.

Let's play around with these using some data we've played with before.

Preliminaries

First, let's import what we'll need:

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

# import your_module as ym or whatever if you want
```

If we don't have our module handy, we can copy our data loader and tidier function from before:

In [2]:
```
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, 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
```

And now we can load and tidy our data with one simple call.

In [3]:
```
our_data = tidyMyData("datasets/018DataFile2.csv")
```

Let's remind ourselves of what the data look like.

In [4]:
```
1 our_data
```

Out[4]:

	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

Figure level plots

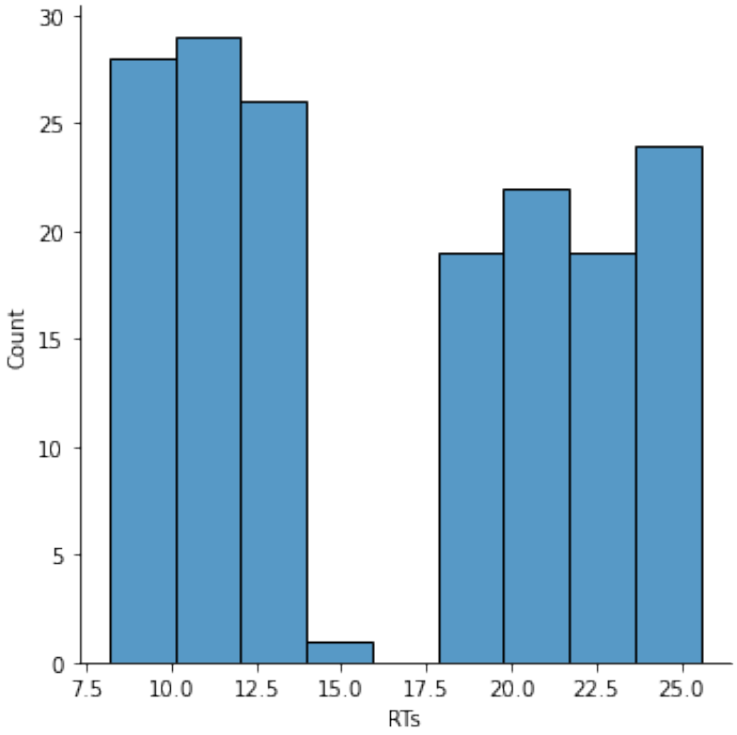
We'll start with some figure level plots.

Distribution plots

Histogram of the RTs

We'll start interrogating the data with a histogram of the lone numerical variable, the RTs

In [5]:
```
1 sns.displot(x = "RTs", data = our_data);
```



Okay, here we can see that there are two clumps of data. Let's see if they correspond to one or more of the categorical variables.

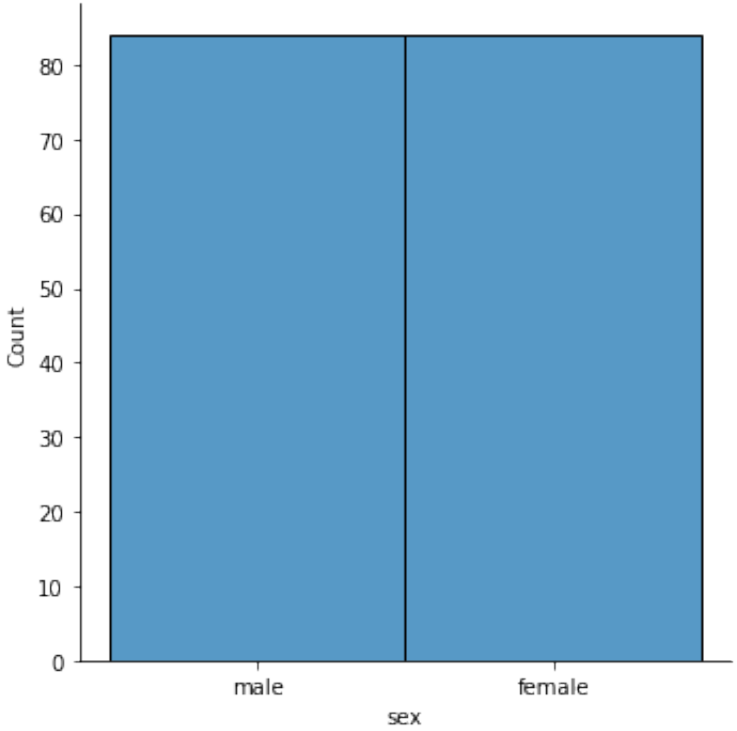
Complete the following exercise.

- Use the cell below to report one parameters that can be passed to `sns` . Explain what the parameter does using your own words.

There were many parameters that can be use with `sns.` such as `sns.algorithms` , `sns.barplot` , `sns.boxplot` , etc.

- Use the cell below to show an example application of the paramter use described above. Use `data` to shwo the example (Please, pick a parameter that is not the one used in the next example!):

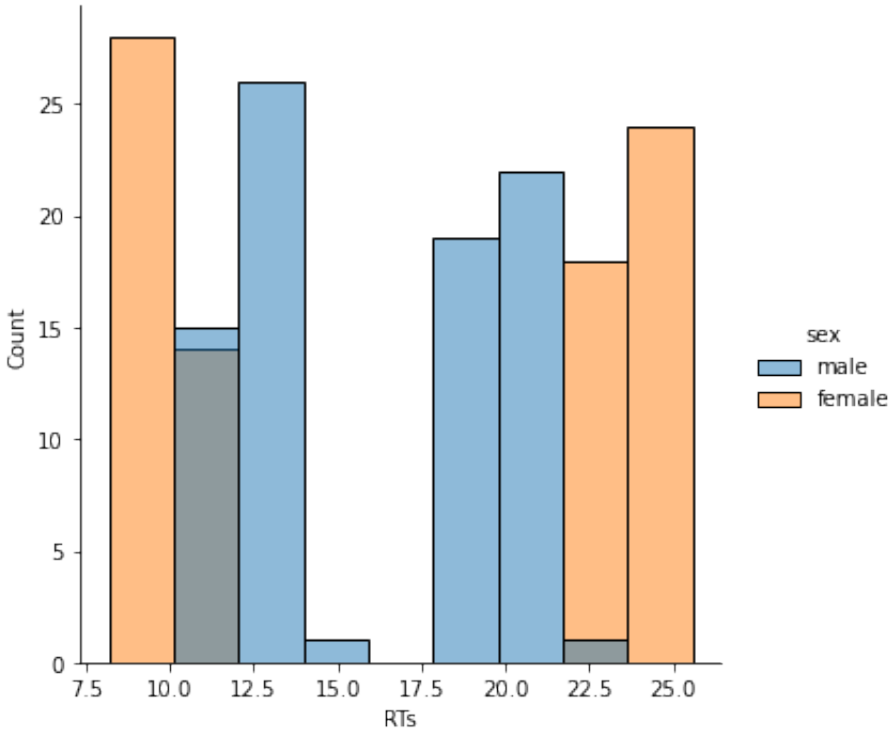
In [6]:
```
1 sns.displot(x = "sex", data = our_data)
```



Histogram of RTs by one of the categorical variables

We'll use color ("hue" in seaborn-speak) to code the categorical variable "sex".

In [7]:
```
1 sns.displot(x = "RTs", data = our_data, hue = "sex");
```

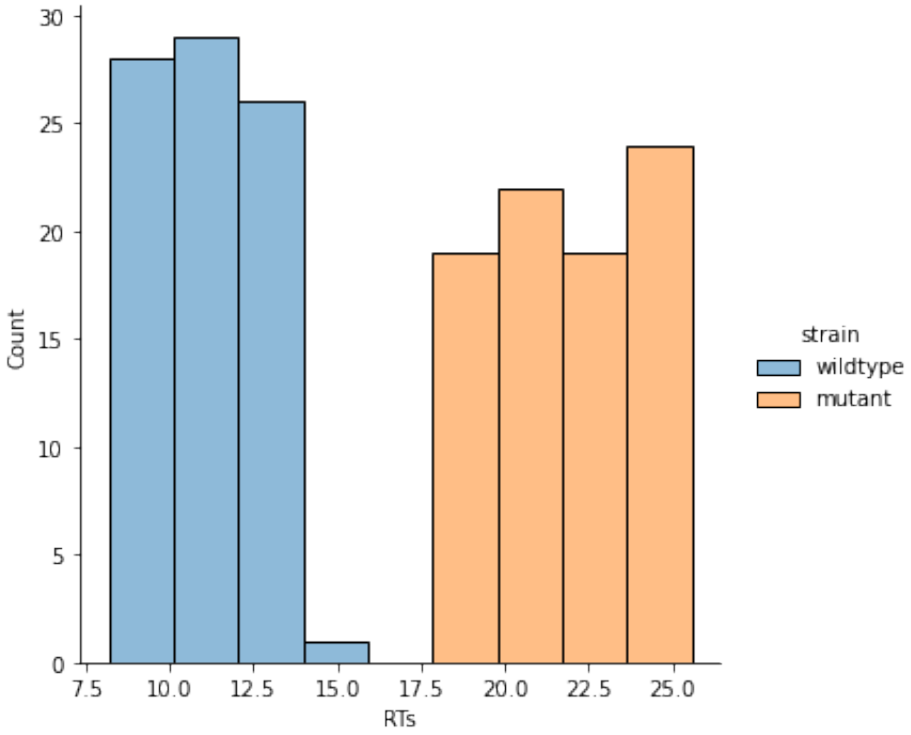


Okay, there might be something going on with females being both faster (left) and slower (right) than males, but there's still something going on here that "sex" isn't capturing. Let's see if "strain" does.

Histogram of RTs by the other categorical variable

Now we'll use color ("hue" in seaborn-speak) to code the categorical variable "strain".

In [8]:
```
1 sns.displot(x = "RTs", data = our_data, hue = "strain");
```



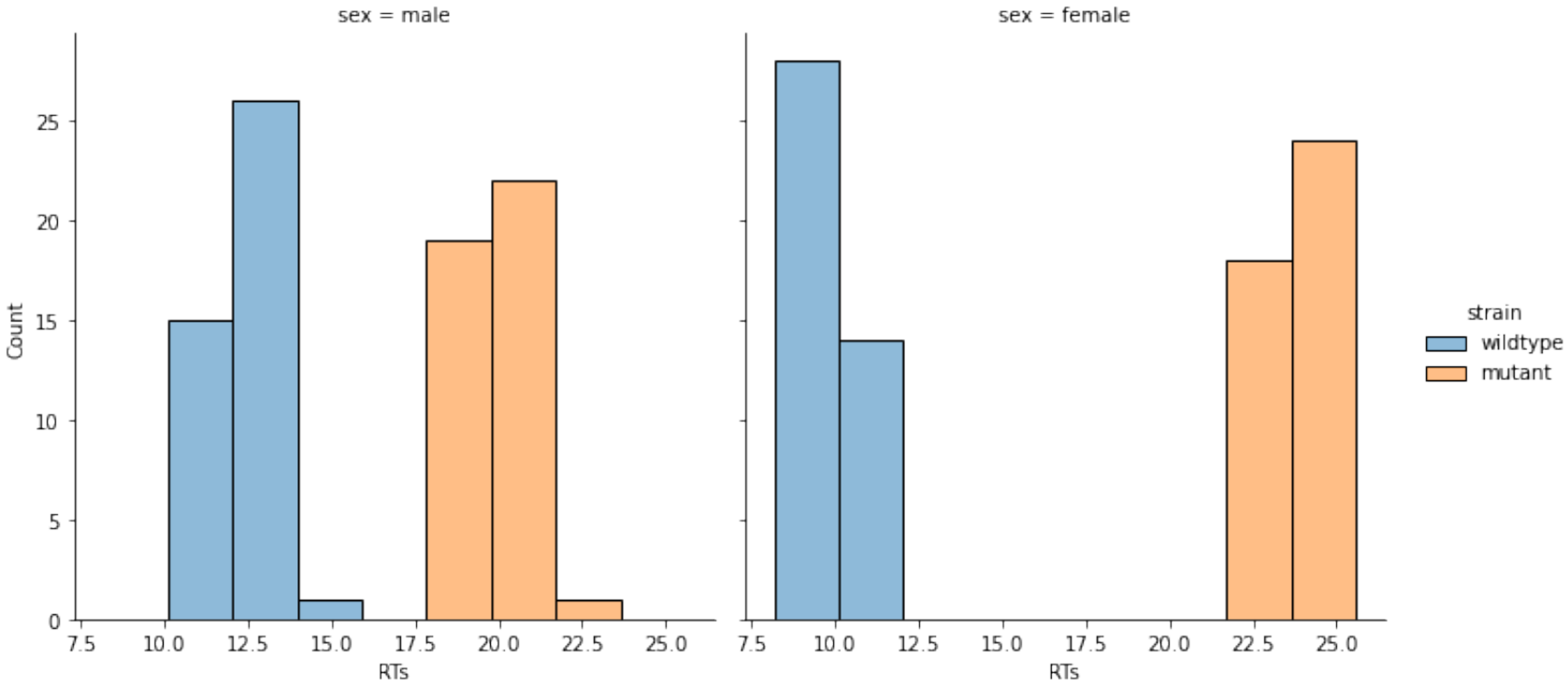
Aha! Gotcha – it looks like strain is doing a pretty good job of explaining the two clumps in the histogram. But the histogram of RT x Sex still did look a litte weird. Let's see if we can crack out both variables.

Creating a multi-axes figure with a figure level seaborn function

This is where the figure level `seaborn` functions are really handy. We can simply assign a categorical variable to be represented by the columns or rows of a multi-panel figure.

Let's assign "sex" to columns.

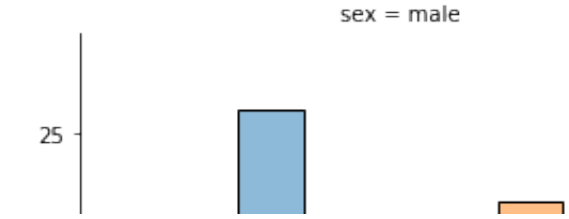
In [9]:
```
sns.displot(x = "RTs", data = our_data, hue = "strain", col="sex");
```

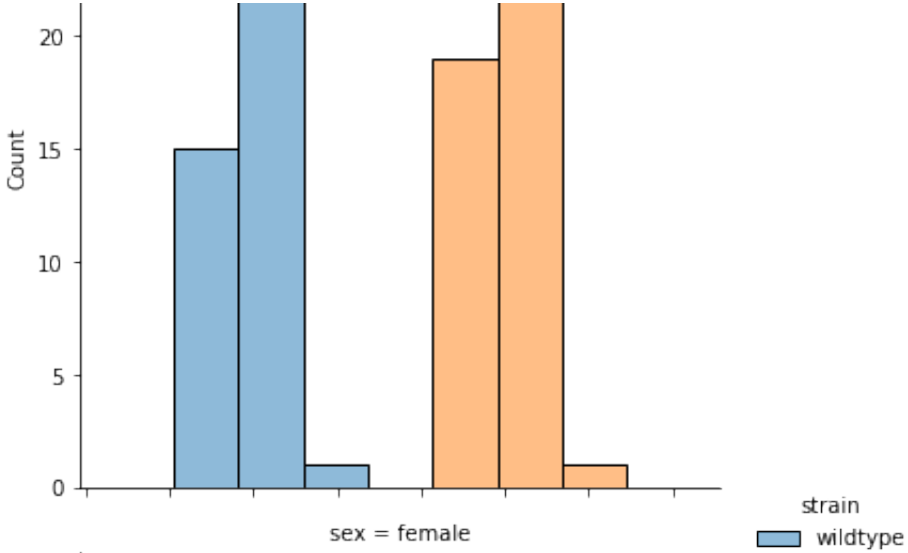


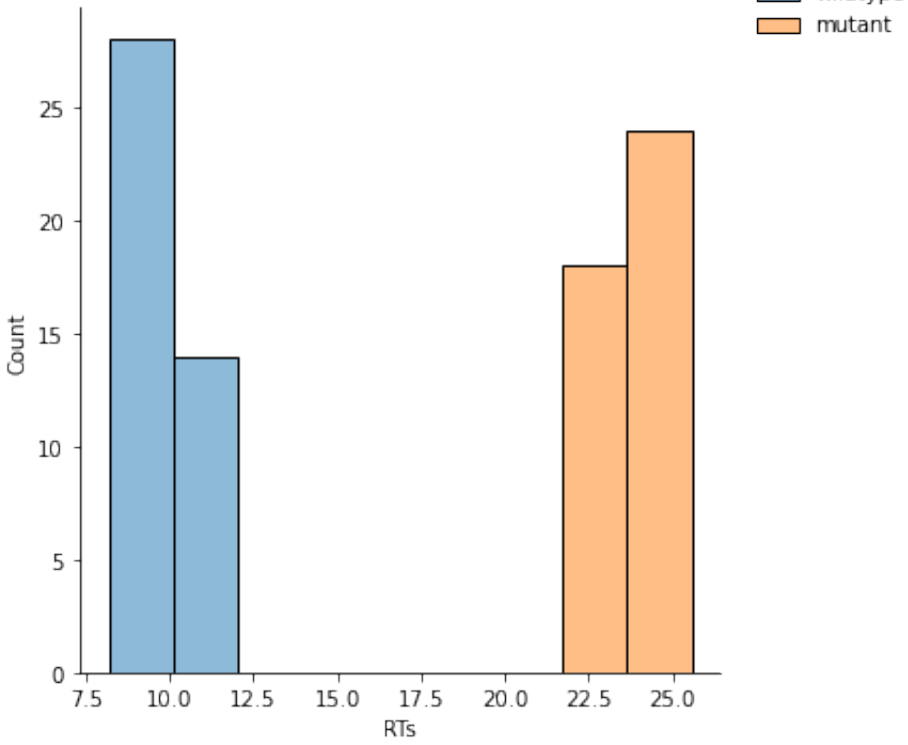
Okay, that's great. Now we have males on the left and females on the right. Also, `displot()` has done something really nice, which is to make the x-axis limits the same in the two plots. So the bigger gap in the female data isn't just a visual artifact of the axis scaling.

Still, since the data share a common x-axis, it would be nice to have the plots aligned vertically rather than horizontally. So let's assign sex to the rows rather than the columns.

In [10]:
```
sns.displot(x = "RTs", data = our_data, hue = "strain", row="sex");
```



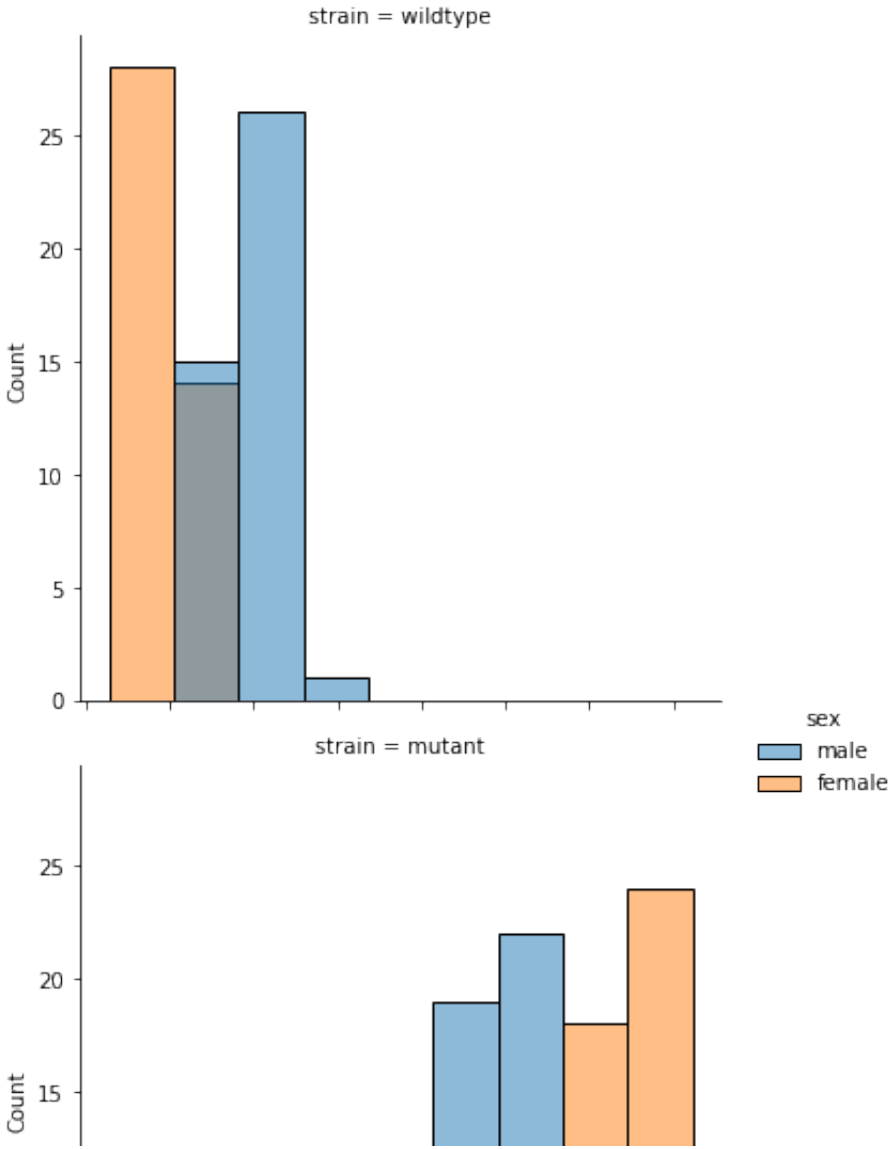

Count
20
15
10
5
0
sex = female
strain
wildtype
mutant

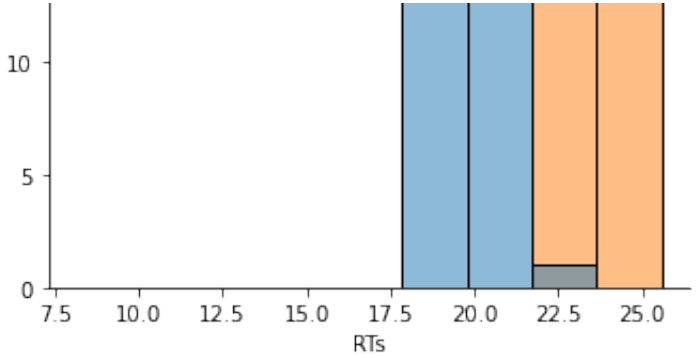

Count
25
20
15
10
5
0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
RTs

Complete the following exercise.

- The previosu example used `sex` to separate the data into histogram plots. Can you use the `strain` to create two separate hisograms? If yes, show the code in the next cell, if not, use the following MarkDown Cell to explain why not.

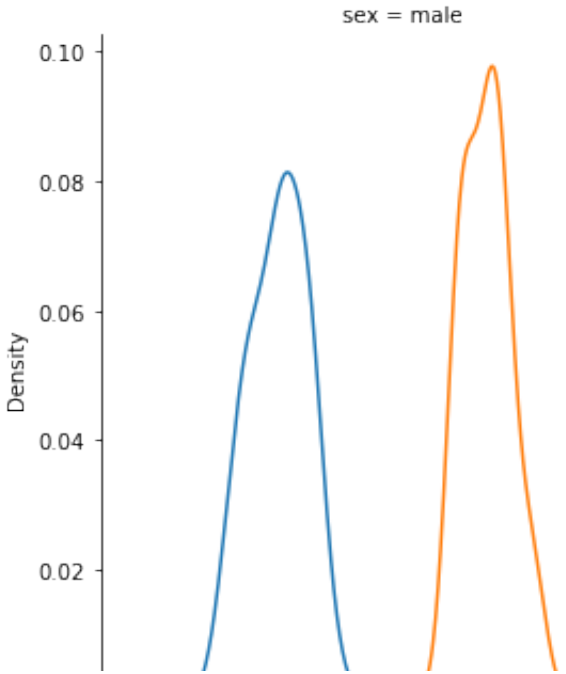
In [11]:
```
1 sns.displot(x = "RTs", data = our_data, hue = "sex", row="strain");
```

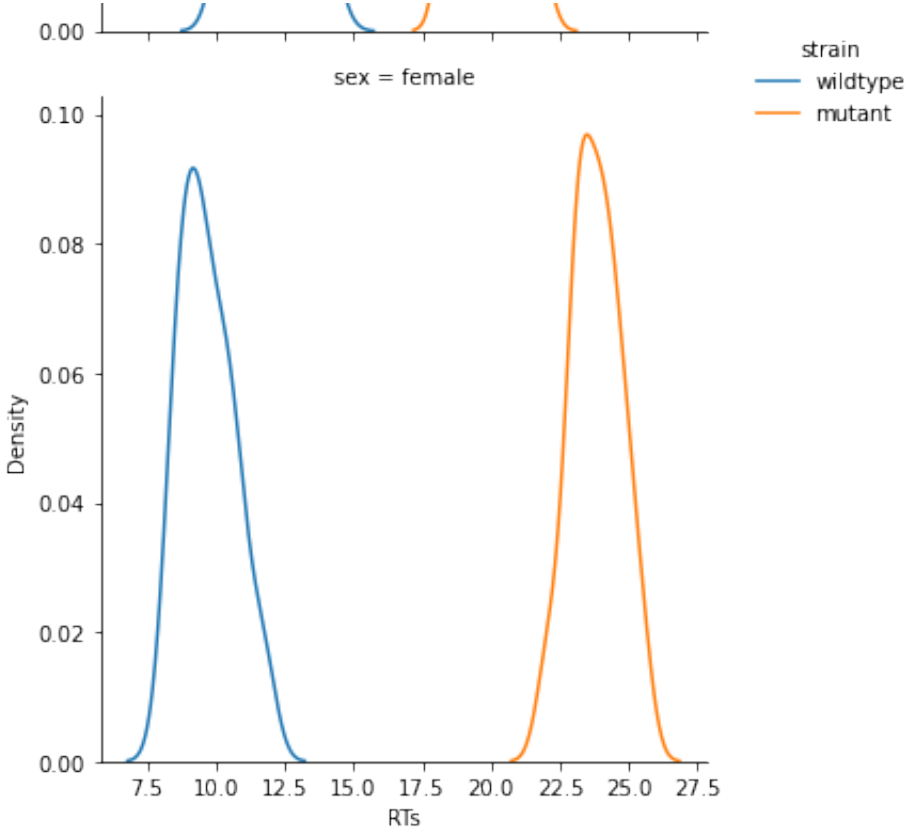




Okay, that last plot we made above was much better in terms of making a visual comparison between the sexes. Still, these histrograms are a bit ugly. We could improve that by playing around with the bins. Or we could just ask `displot()` to give us kernel density estimates instead.

In [12]:
```
1 sns.displot(x = "RTs", data = our_data, hue = "strain", row="sex", kind="kde");
```





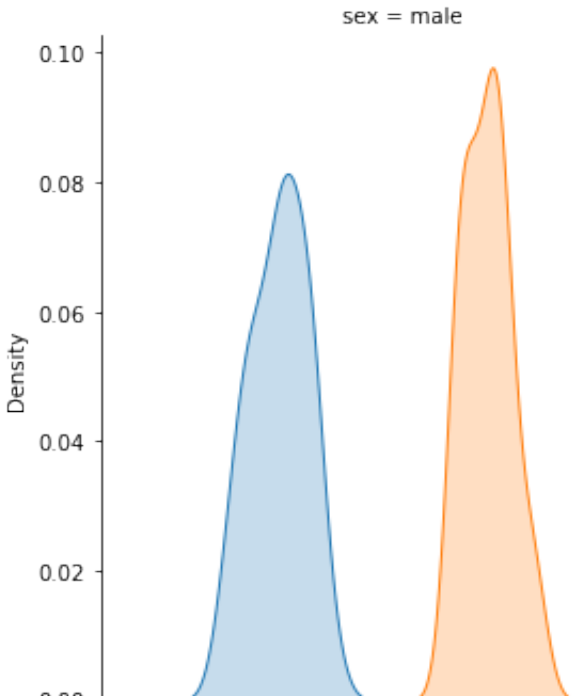
Much better. Visually, however, filled KDE's are a bit nicer. Since these are probability densities, it's the area that's important anyway, and having them filled emphasizes the area rather than the height.

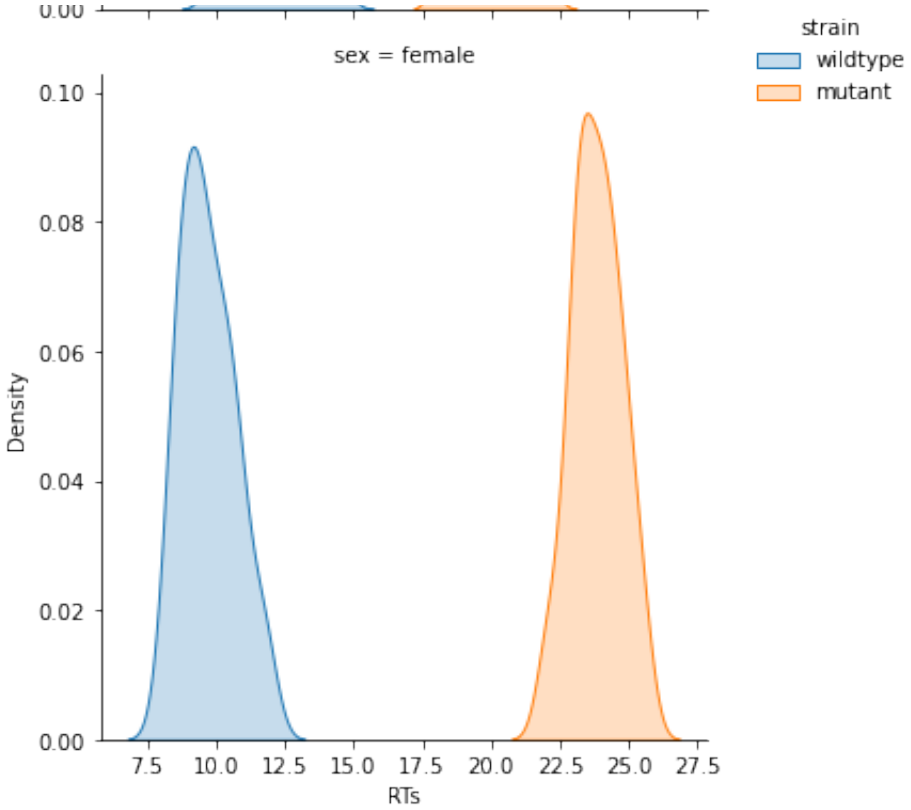
We can easily do this by setting a `fill` argument to `True`. Strictly speaking, however, `fill` is not a valid argument to `displot()`. However, what displot will do is pass any named argument (called a "keywork arguement" or "kwarg" in Python) to the underlying axes level function.

The only catch with these `**kwargs` is that they won't appear in the documentation for the figure level plots, only in the documentation for the axes level plots. The documentation for the figure level plot (https://seaborn.pydata.org/generated/seaborn.displot.html#seaborn.displot), like `displot()` does helpfully tell us this at least:

So now let's plot with `fill=True` and see if that works.

In [13]:
```
1 sns.displot(x = "RTs", data = our_data, hue = "strain", row="sex", kind="kde", fill=True);
```



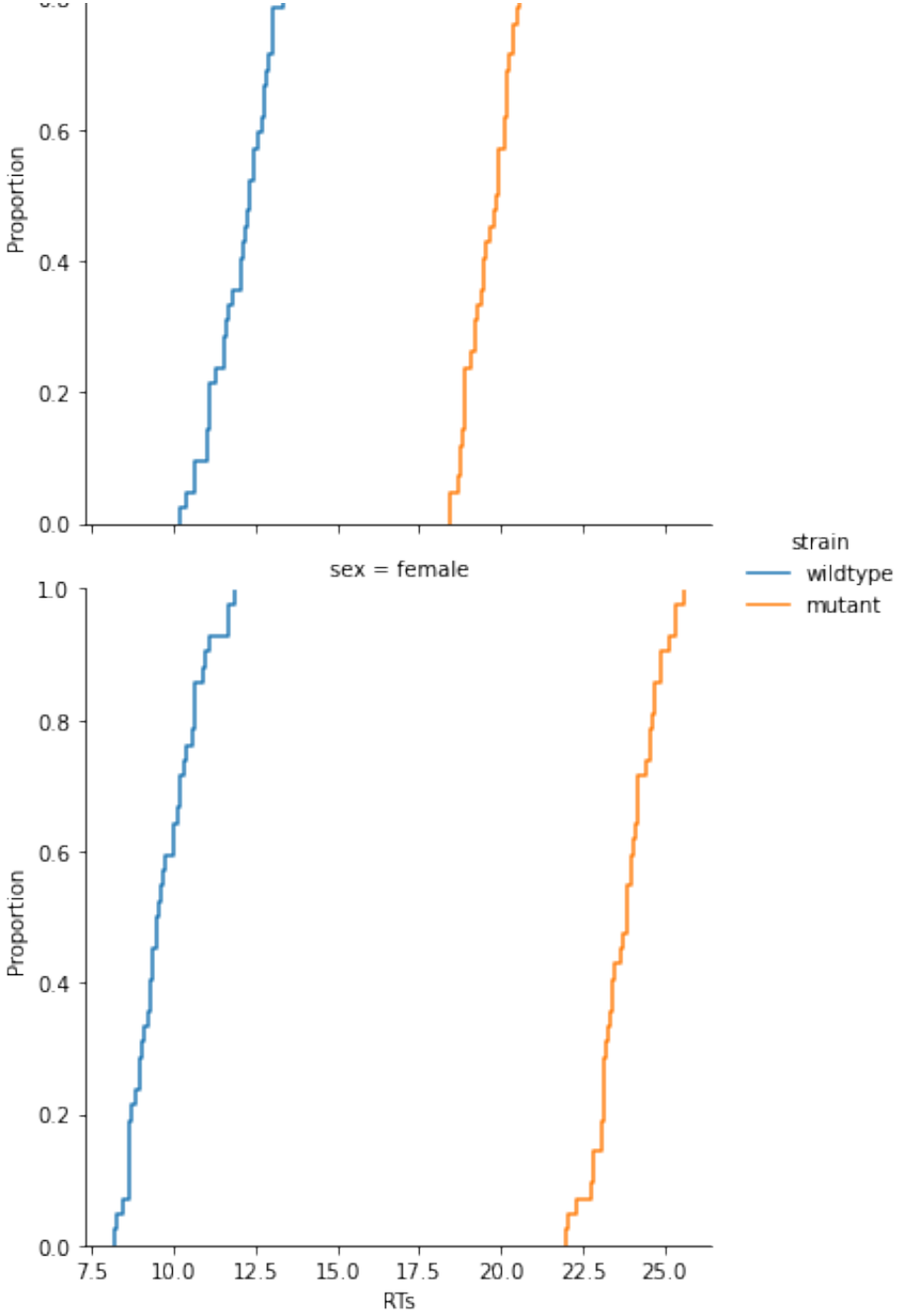


Ah, much better!

Complete the following exercise.

- Use the cell below to show an example plot that uses the data above in combination with the `seaborn` parameter `ecdf`.

In [23]:
```
1 sns.displot(x = "RTs", data = our_data, hue = "strain", row="sex", kind="ecdf");
```


Proportion
0.0
0.2
0.4
0.6
sex = female
1.0
0.8
0.6
0.4
0.2
0.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
RTs
strain
wildtype
mutant

- If you belive the parameter cannot be used in combination with the data at hand, use the cell below to explain the reason why the parameter cannot be properly used in this case.

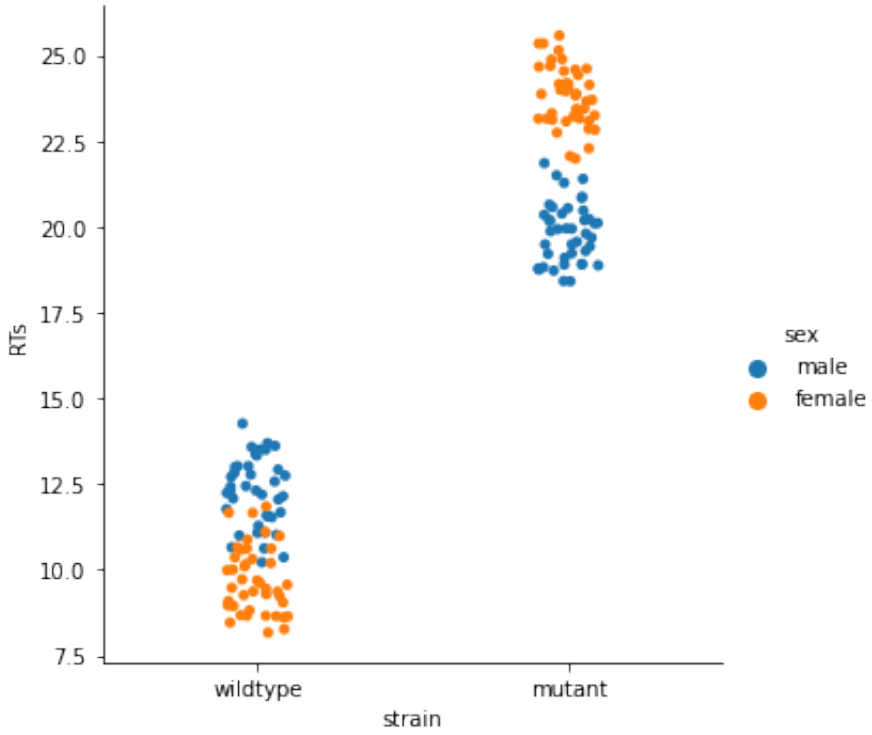
The parameter does work with the data, however, we need to delete `fill=True` because `ecdf` does not have `fill` options.

Categorical plots

The categorical plots are nice because they allow us to separate both of our categorical variables within a single plot.

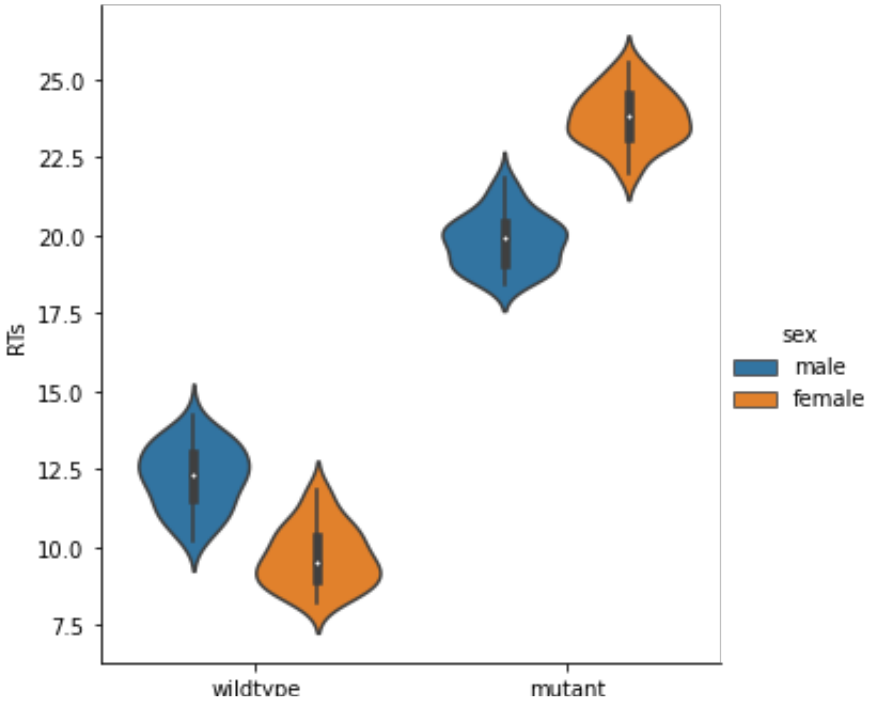
Let's try playing with `catplot()`

In [24]:
```
1 sns.catplot(y = "RTs", x="strain", data = our_data, hue = "sex");
```



So a `stripplot` is the default axes-level plot (and notice that the default axes-level plots are the first ones listed under their corresponding figure-level counterparts. But we can have it call `boxplot()` for us by telling it that we want `kind="box"` .

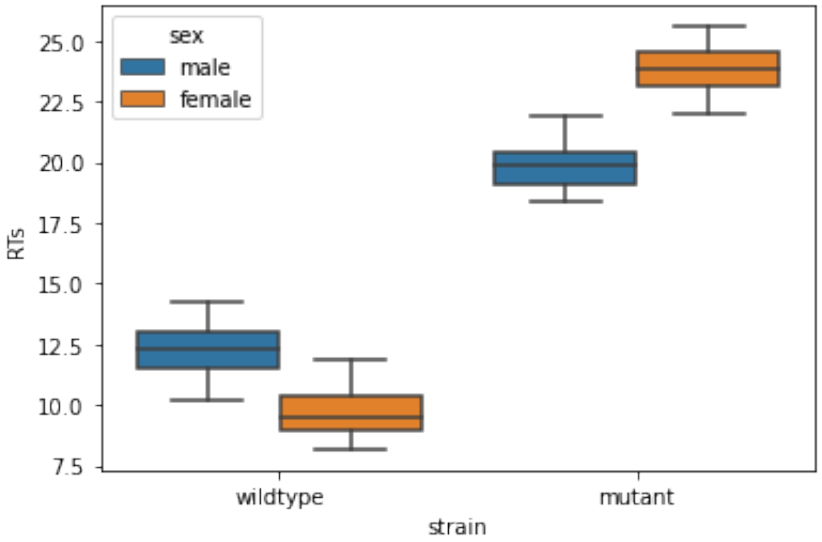
In [25]:
```
1 sns.catplot(y = "RTs", x="strain", data = our_data, hue = "sex", kind="violin");
```



Axis level plots

We can call any of the axis-level functions directly, without going through the corresponding figure-level function. This gives us more control over single-panel plots should we need it.

In [26]:
```
1 ax = sns.boxplot(y = "RTs", x="strain", data = our_data, hue = "sex");
```



Notice that we've assigned the output of `sns.boxplot()` to `ax`. So we have an `axes` object (named `ax`), and thus have access to all the things an `axes` knows how to do. So if we type "ax." and a tab, we'll see something like this:

So let's do that:

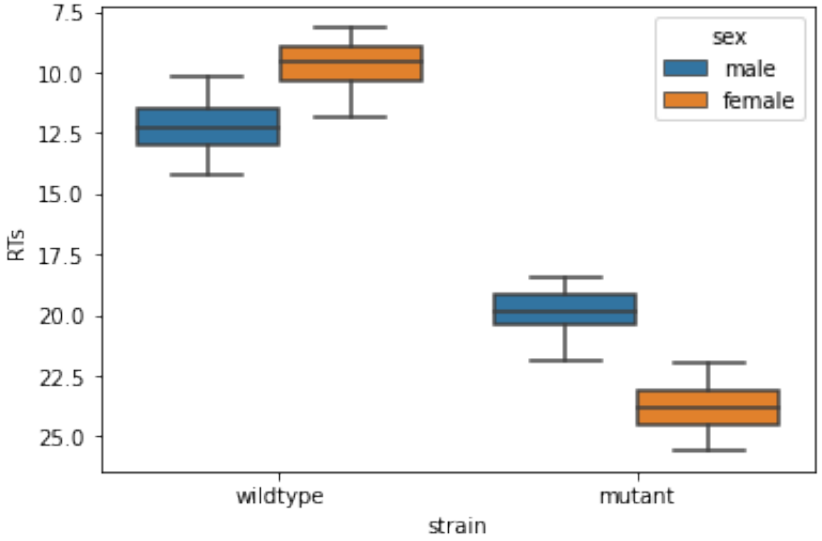
In [27]:
```
1 ax.
```

```
  Input In [27]
    ax.
       ^
SyntaxError: invalid syntax
```

and now we can scroll around to find useful things.

Let's try inverting the y axis so that faster times plot visually higher!

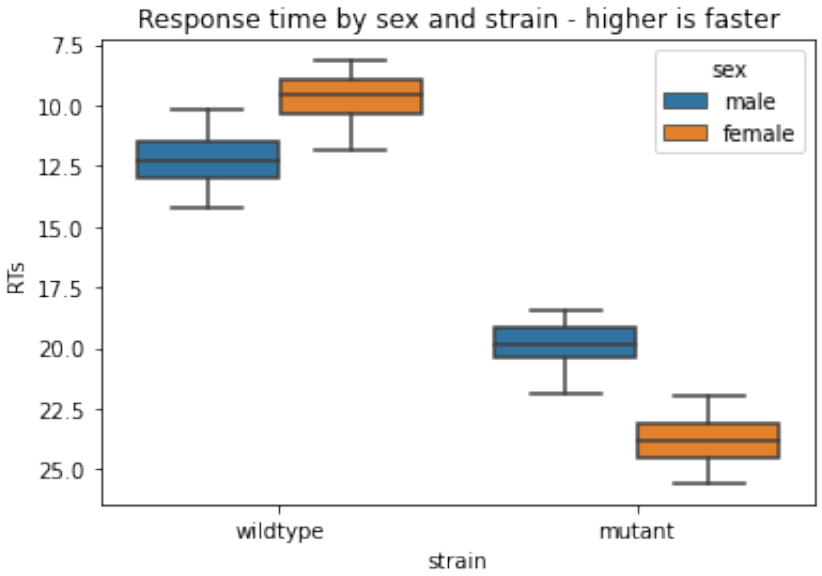
In [28]:
```
ax = sns.boxplot(y = "RTs", x="strain", data = our_data, hue = "sex");
ax.invert_yaxis();
```



And we can see a lot of useful stuff by typing `ax.set` and a tab. Among those is `ax.set_title` which we can use to... wait for it... add a title!

In [29]:

```
ax = sns.boxplot(y = "RTs", x="strain", data = our_data, hue = "sex");
ax.invert_yaxis();
ax.set_title("Response time by sex and strain - higher is faster");
```



Summary

So `seaborn` is a nice way to make plots of data from `pandas` data frames. Its default values make good looking plots. It has two main kinds of plotting functions:

- figure level functions that are handy for making multi axes panel figures
- axes level functions that return an axes object handle to you, allowing for fine control over the plot's appearance

---

Complete the following exercise.

Write a function that uses `seaborn` to visualize data as we need it!

To flex both our plotting and function writing muscles, let's write a function to do some plotting! Your function should:

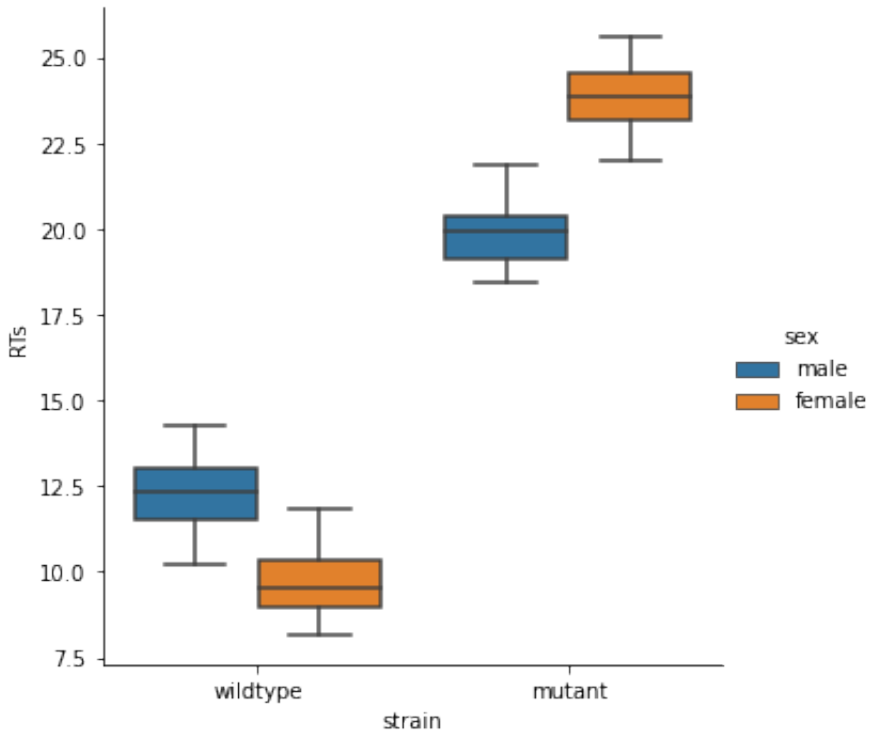
- take as input a data frame as produced by the function above
- allow the user to choose between a strip, violin, or box plot
- set one of the above three be the default
- have a docstr so users can get help() on it
- produce the plot requested by the user (of course!)
- provide a meanignful help

Write your function here:

In [86]:
```
def Plot(our_data, plot_type = 'box') :
    '''

    '''

    import pandas as pd
    import numpy as np
    import seaborn as sns

    return sns.catplot(y = "RTs", x="strain", data = our_data, hue = "sex", kind=plot_type);
```

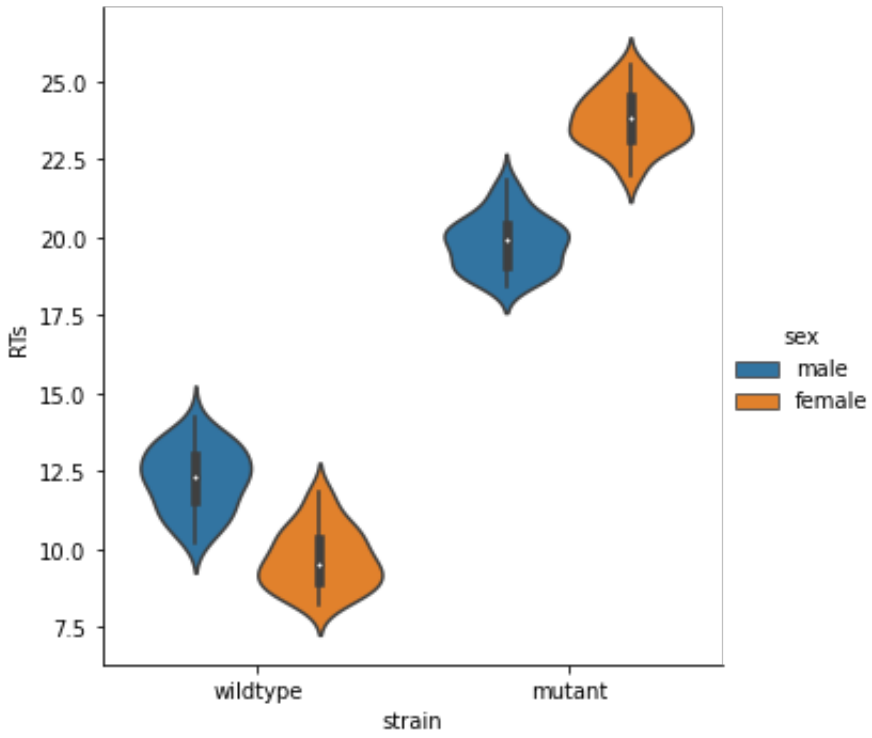
In [90]:
```
1 Plot(our_data)
```

Out[90]: <seaborn.axisgrid.FacetGrid at 0x7fc6873de0a0>



In [89]:
```
1 Plot(our_data, 'violin')
```

Out[89]: <seaborn.axisgrid.FacetGrid at 0x7fc687671790>



In [91]:
```
1 Plot(our_data, 'strip')
```

Out[91]: <seaborn.axisgrid.FacetGrid at 0x7fc6882f2400>