FES 524 Winter 2022 Lab 1

Contents

Lab 1 is going to be a little bit different from all of the other labs for this class. Today we are working towards a simple two-sample analysis, which should be review for everyone. This gives us a chance to spend time doing some extra work in R, primarily covering a tiny bit about data manipulation in R. We will also spend a fair amount of time talking about R help - where you can find it and how to search for it. Today may be review for people in the class with a more extensive background in R, and those folks might decide to skip directly to the lab assignment.

Overall analysis goal

The overall analysis goal today is to compare mean respiration for "Cold" and "Hot" sites. Today "Cold" is defined as anything less than 8°C.

We have measurements of both respiration and temperature at different sites. However, these data are stored in three datasets instead of one, which I've provided to you on Canvas. A lot of your "R time" today will be spent reading the three datasets into R and combining them in preparation for a simple analysis.

The three datasets we will be working with today are:

`temp.txt`
`spring_87_resp.txt`
`fall_87_resp.txt`

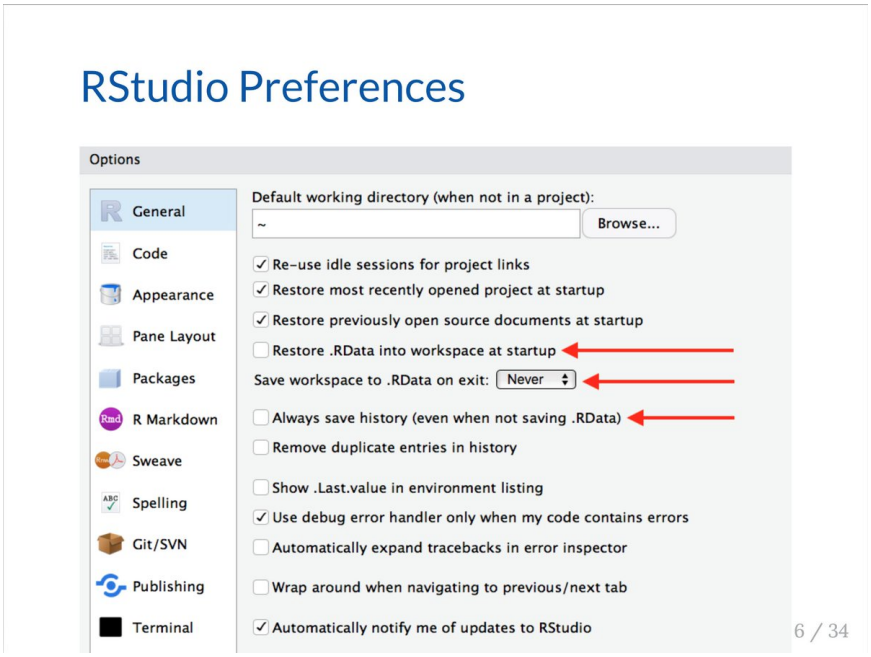
RStudio preferences for keeping a clean workspace

I recommend that you get into the habit of working in a fresh R process. To do this you'll need to make sure you never load a saved workspace in a new R session. The default behavior can be changed in RStudio.

To do this:
Go to `Tools > Global Options...` and

1. Unclick the box next to "Restore .RData into workspace at startup".
2. Change the "Save workspace to .RData on exit" option to "Never".
3. Unclick the box next to "Always save history (even when not saving .RData)"

The last one is less critical than the other two, depending on how if you have a need for the history for some reason. Here is a picture of the process from one of Alison Hill's slides.



Now you are guaranteed to get a clean session of R every time you reopen it. I use `Ctrl + Shift + F10` (Mac OS `Command + Shift + F10`) to restart R throughout the day to make sure I have a clean session as I switch from project to project.

See some discussion on twitter and read more about good coding practice in this blog post (with discussion).

R help

There are three main places where I go for help using R: the documentation within R, searching Stack Overflow/R mailing list archives/RStudio Community, and internet searches via a search engine.

Using the documentation within R

The first place to look for help is within R, in the R documentation. Every time I use a function for a first time or reuse a function after some time has passed, I spend time looking through the R documentation for that function. You can do this by typing `?functionname` into your Console and pressing enter, where "functionname" is the name of your function. This means you have to know the name of the function you want to use in advance. For example, if we wanted to take an average of some numbers with the `mean()` function, we would type `?mean` at the `>` in the R Console and look through the documentation to see how to use it. In RStudio, the documentation will open in the Help pane. If this is too small to read on your computer, click the "Show in a new window" button to open the documentation as a separate page.

The documentation for functions in R generally will look similar. A list of the arguments the function takes and the defaults to those arguments are the first sections. The documentation for a function usually contains example code at the very bottom, which you can copy and paste into R and run when you need to see how the function works. There can be a variety of other information between the arguments and the examples sections. While this is often useful information, I don't usually read through this on the first pass. Instead, I will revisit it when I'm really struggling with a new function and need further details on how it works or what it is doing.

For a more thorough dissection of a help page in R, see this blog post: https://aosmith.rbind.io/2020/04/28/r-documentation/

Using Stack Overflow and the R mailing list archives

When I'm looking for how to do something in R and I don't have a function name (or sometimes even if I do), I will search the R-tagged questions on help forums. I prefer the set-up of Stack Overflow and the RStudio Community, as the R mailing list archives is in a threaded format that seems harder to navigate to me. Stack Overflow is currently very active with R questions, and often you can find a solution to a problem you are having there.

For the R tagged Stack Overflow posts: http://stackoverflow.com/questions/tagged/r

The (somewhat) searchable R help archives is here: https://www.mail-archive.com/r-help@r-project.org/

A newer forum for R questions is the RStudio Community, which can be less intimidating than Stack Overflow if you are asking a question for the first time: https://community.rstudio.com/

Searching the internet

You can (and I often do) search the internet via a search engine in your browser. You need to include "R" or "Rstats" as a search term. This often works well for me, but can also work poorly because many, many pages include the term *R*.

The working directory

I use the working directory extensively when I use R. The working directory is where R, by default, will look for any datasets you load and will save anything you save. When I'm working on a relatively simple project, I save my R scripts and all files related to that project into a single folder that I set as my working directory. This makes it so I don't have to write out the whole directory path every time I want to load or save something while making these tasks repeatable by using code instead of relying on drop-down menus.

In RStudio, you can make a "Project" and use package **here** to streamline the working directory even more (see an example at https://aosmith.rbind.io/2018/10/29/an-example-directory-structure/). We won't be covering Projects in this class but these are a handy organizational tool that you may want to look into if you use R extensively. Many students learn to hard code the working directory with `setwd()`, which I discourage.

To see your default working directory, use the `getwd()` function to "get" your current working directory. You'll see my current working directory below.

```
getwd()
```

```
[1] "C:/Users/USER/Documents/Aosmith/FES 524/2022/Labs/Lab1"
```

You can set your working directory in a variety of ways. These days I tend to take advantage some of RStudio's features for this task. Most often I work with RStudio Projects or use RStudio's drop down menus.

If you've navigated to your directory in the "Files" pane, you can use the pane drop-down menus:

`More > Set As Working Directory`

If you've already opened the R script you'll be using, you can use the overall drop-down menus:

`Session > Set Working Directory > To Source File Location`

And, of course, you can always type out the path to your working directory using `setwd()`. However, I encourage you to learn package **here** instead of hard-coding a working directory with `setwd()`.

Important: You must either use single forward slashes or double backslashes in your directory path in R instead of the single backslash you may be used to.

The following example code will not run for you, and is simply to demonstrate how to write a directory path in R.

```
setwd("T:/Teach/Classes/FES524/Labs/Lab1")
setwd("T:\\Teach\\Classes\\FES524\\Labs\\Lab1")
```

Once you've set your working directory, you can check if you've made the change correctly by using `getwd()` as above.

The temperature dataset

Reading in a text file with `read.table()`

The respiration and temperature data are currently in three datasets that we need to combine into one for analysis. We'll start our work with the dataset that contains the temperature information, called `temp.txt`.

The temperature data are in a white-space-delimited text file, so we'll read this in using `read.table()`. You should make it a habit to check out the help files when you are using a function for the first time so you know what the default settings are and to see what settings you can control with different function arguments.

```
?read.table
```

We will need to tell R that our dataset contains columns names. This is commonly how we would store files, and it means that the very first row of our dataset has the variable names in it. We tell R that the first row are column names with the `header` argument. Per the R help page, the `header` argument is *a logical value indicating whether the file contains the names of the variables as its first line.* The default, shown on the help page, is `FALSE` in `read.table()` so we'll need to change it to `TRUE`.

We'll assign the name `temperature` to this dataset when we bring it in R. You will see today that assigning names to R *objects* is a key part of using R. I use `=` for assignment; the most common assignment operator you will see is `<-`. And right-hand assignment, `->` is becoming more common. Pick whichever you like in your work and stick with it.

```
temperature = read.table("temp.txt", header = TRUE)
```

Notice that we can now see an object named `temperature` in our RStudio Environment pane, so we have successfully imported the dataset.

You should name datasets whatever you like, although I personally recommend names that are easy to type. In R, datasets are called `data.frames`, and you could refer to `temperature` as a ***`data.frame object`***. I will be using the words dataset and data.frame interchangeably.

If your dataset isn't in your working directory, you'd need to write out the path to wherever the file is located. Again, you must either use forward slashes, like I did here, or double backslashes.

The following code will not run for you, and is simply to demonstrate how to write out the whole directory path.

```
temperature = read.table("T:/Teach/Classes/FES524/Labs/Lab1/temp.txt",
                         header = TRUE)
```

The first thing to do after reading in a dataset is to take a look at it to make sure everything looks the way you expect it to. We can check the basic *structure* of the dataset with the `str()` function. In RStudio, we can click on the arrow next to the object name in the Environment pane to see the structure of the dataset, as well.

```
str(temperature)
```

```
'data.frame':   59 obs. of  4 variables:
 $ Sample: int  18 20 22 19 31 30 28 32 29 24 ...
 $ Tech  : chr  "Mark" "Raisa" "Nitnoy" "Nitnoy" ...
 $ Temp  : num  4.5 4.5 4.5 4.5 5 5 5 5 5 5.5 ...
 $ DryWt : chr  "0.569" "0.597" "0.603" "0.607" ...
```

Uh-oh, I see a problem right away. The structure lists what kind of variable each column in the dataset contains and `DryWt` was read as a character but should be numeric. A character variable in R is one type of *classification* or *categorical* variable.

We need to figure out what's going on. Let's take a closer look at that single column. We can do this by printing out the column as a vector of values into the Console.

To work directly with a single column from a dataset, we need to indicate to R both which variable we want and where that variable is stored. There are a variety of ways to do this, but a simple way is to tell R that the variable `DryWt` is located in the `temperature` dataset using dollar sign notation. We write out the name of the `data.frame` the variable is in, a dollar sign (`$`), and the name of the variable we are interested in.

If you learned to use `attach()` in your earlier statistics classes I strongly recommend you break that habit. I'll provide some additional alternatives later in this lab.

```
temperature$DryWt
```

```
 [1] "0.569" "0.597" "0.603" "0.607" "0.611" "0.613" "0.622" "0.626" "0.634" "0.565" "0.61"  "0.62"
[13] "."     "0.64"  "0.656" "0.661" "0.685" "0.695" "0.701" "0.528" "0.574" "0.619" "0.627" "0.642"
[25] "0.62"  "0.65"  "0.67"  "0.728" "0.679" "0.753" "0.759" "0.77"  "0.781" "0.786" "0.727" "0.785"
[37] "0.787" "0.793" "0.795" "0.709" "0.765" "0.768" "0.791" "0.804" "0.694" "0.709" "0.732" "0.739"
[49] "0.749" "0.82"  "0.836" "0.844" "0.848" "0.859" "0.779" "0.801" "0.808" "0.828" "0.83"
```

Can you see that one of the values is a period, `.`, all by itself? A period by itself is a character, not a number, and so when R found a character in that column it defaulted to making the whole column a character variable.

Using `na.strings` to define missing value characters

It turns out that this dataset was used in SAS at some point, and the period represents a missing value. We will need to tell R that `.` means `NA` so it reads the dataset correctly. We do this by taking advantage of the the argument `na.strings` in `read.table()`. You see in the help page that, by default, R considers fields that contain `NA` or blanks as missing; if you use something else you have to be sure to tell R.

I didn't tell you about the `.` earlier because I wanted you to see this happen. This is a common hurdle for people when they first start to use R. If you look around online you'll see many people asking questions that boil down to a numeric variable that was read as a character. I recommend figuring out why this is happening by addressing it when you are first reading the data in rather than trying to change variable types later.

For your later reference, there are two main reasons I've seen to cause the "numeric read as character" problem. First, like in this example, is missing values stored as some miscellaneous character value, such as as `na` or `n/a` or `N/A`. The second situation I've commonly seen is when folks have stored their large numbers with commas in them like, e.g, `1,112` instead of `1112`. The easiest way to avoid the second is to not store numbers like that, but if you do there is help online to show you what to do.

Let's read in the dataset again, this time using the `na.strings` argument to indicate that missing values are represented by `"."`. We will name the object *temperature* again, replacing the previous version with the new one.

```
temperature = read.table("temp.txt", header = TRUE, na.strings = ".")
```

How does the structure look now in your Environment pane?

```
'data.frame':   59 obs. of  4 variables:
 $ Sample: int  18 20 22 19 31 30 28 32 29 24 ...
 $ Tech  : chr  "Mark" "Raisa" "Nitnoy" "Nitnoy" ...
 $ Temp  : num  4.5 4.5 4.5 4.5 5 5 5 5 5 5.5 ...
 $ DryWt : num  0.569 0.597 0.603 0.607 0.611 0.613 0.622 0.626 0.634 0.565 ...
```

Exploring a dataset in R

Now that things look better, let's look at some more options for exploring a dataset.

If we just run the name of this dataset, the whole dataset will print into the R Console.

```
temperature
```

```
   Sample     Tech Temp DryWt
1      18     Mark  4.5 0.569
2      20    Raisa  4.5 0.597
3      22   Nitnoy  4.5 0.603
4      19   Nitnoy  4.5 0.607
5      31 Stephano  5.0 0.611
6      30 Stephano  5.0 0.613
7      28     Cita  5.0 0.622
8      32    Raisa  5.0 0.626
9      29    Raisa  5.0 0.634
10     24     Cita  5.5 0.565
11     25   Fatima  5.5 0.610
```

```
12     27    Raisa  5.5 0.620
13     23   Fatima  5.5    NA
14     26     Mark  5.5 0.640
15     74    Raisa  7.0 0.656
16     77   Nitnoy  7.0 0.661
17     76    Raisa  7.0 0.685
18     73  LaVerna  7.0 0.695
19     75    Raisa  7.0 0.701
20     33   Nitnoy  8.0 0.528
21     36    Raisa  8.0 0.574
22     37     Cita  8.0 0.619
23     35 Stephano  8.0 0.627
24     34  LaVerna  8.0 0.642
25     44 Stephano 10.5 0.620
26     43     Mark 10.5 0.650
27     46    Raisa 10.5 0.670
28     45 Stephano 10.5 0.728
29     47     Cita 10.5 0.679
30     71    Raisa 11.5 0.753
31     72     Mark 11.5 0.759
32     68    Raisa 11.5 0.770
33     69     Mark 11.5 0.781
34     70   Fatima 11.5 0.786
35     50     Mark 13.0 0.727
36     51 Stephano 13.0 0.785
37     48     Mark 13.0 0.787
38     49    Raisa 13.0 0.793
39     52    Raisa 13.0 0.795
40     57   Nitnoy 14.0 0.709
41     53   Nitnoy 14.0 0.765
42     54 Stephano 14.0 0.768
43     56   Fatima 14.0 0.791
44     55 Stephano 14.0 0.804
45     41    Raisa 14.5 0.694
46     42   Nitnoy 14.5 0.709
47     39   Fatima 14.5 0.732
48     38   Nitnoy 14.5 0.739
49     40    Raisa 14.5 0.749
50     65   Fatima 16.0 0.820
51     67     Cita 16.0 0.836
52     64  LaVerna 16.0 0.844
53     63    Raisa 16.0 0.848
54     66     Mark 16.0 0.859
55     60   Fatima 19.0 0.779
56     58   Nitnoy 19.0 0.801
57     59     Cita 19.0 0.808
58     62     Mark 19.0 0.828
59     61    Raisa 19.0 0.830
```

This isn't that useful unless the dataset is small.

If you click on the `temperature` object in your RStudio Environment pane, you can see the dataset in your Source pane. You cannot edit from here, but this is another way that you can get a sense of what the dataset looks like. You can also do some basic filtering and sorting.

You can look at just the first or last six rows of your dataset by using `head()` or `tail()`, respectively, to get an idea of what a dataset looks like without printing the whole thing into the Console. This can be useful for long datasets.

```
head(temperature)
```

```
  Sample     Tech Temp DryWt
```

```
1     18     Mark  4.5 0.569
2     20    Raisa  4.5 0.597
3     22   Nitnoy  4.5 0.603
4     19   Nitnoy  4.5 0.607
5     31 Stephano  5.0 0.611
6     30 Stephano  5.0 0.613
```

```
tail(temperature)
```

```
   Sample   Tech Temp DryWt
54     66   Mark   16 0.859
55     60 Fatima   19 0.779
56     58 Nitnoy   19 0.801
57     59   Cita   19 0.808
58     62   Mark   19 0.828
59     61  Raisa   19 0.830
```

If you need to check the names of the variables (which I invariably forget), you can see the column names with `names()`. This is useful, as the column names are often used when working with data in R.

```
names(temperature)
```

```
[1] "Sample" "Tech"   "Temp"   "DryWt"
```

Speaking of names, it's important that you recognize that R is case sensitive. This means that it reads upper and lower case letters differently (e.g., "A" is different than "a"). Be sure to watch out for this when working with categorical variables and names.

Let's take a look at the dataset dimensions. This is another easy check to do at early on to make sure the dataset was read in correctly. A `data.frame` in R has two dimensions, rows and columns. A `data.frame` can't be *ragged*, but instead is always rectangular. This means each column is the same length as every other column and each row is the same width as every other row. If your real dataset is not rectangular, any blanks will be filled in with missing values.

We can see the dimensions of our object in our RStudio Environment pane, or check the dimensions using `dim`, `nrow`, and `ncol`.

```
dim(temperature)
```

```
[1] 59  4
```

```
nrow(temperature)
```

```
[1] 59
```

```
ncol(temperature)
```

```
[1] 4
```

While we're in the data exploration stage, let's get summary information on the whole dataset with `summary()`. This returns summary statistics for each numeric column, the total column length for character variables, and a tally of the number of observations in each category (aka *levels*) if you have factors. We'll talk more about factors in lab 2.

```
summary(temperature)
```

```
     Sample          Tech                Temp           DryWt        
 Min.   :18.00   Length:59          Min.   : 4.50   Min.   :0.5280  
 1st Qu.:33.50   Class :character   1st Qu.: 7.00   1st Qu.:0.6262  
 Median :48.00   Mode  :character   Median :11.50   Median :0.7090  
 Mean   :47.95                      Mean   :10.81   Mean   :0.7086  
 3rd Qu.:62.50                      3rd Qu.:14.25   3rd Qu.:0.7857  
 Max.   :77.00                      Max.   :19.00   Max.   :0.8590  
                                                    NA's   :1       
```

The `summary()` function can also be used on single columns of a dataset. This tends to be most useful for numeric variables. Below we summarize just the `Temp` variable.

```
summary(temperature$Temp)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   4.50    7.00   11.50   10.81   14.25   19.00 
```

The respiration datasets

Now we'll read in the two respiration datasets, `fall_87_resp.txt` and `spring_87_resp.txt`, using `read.table()`. There are no missing values in these dataset, so we don't need the `na.strings` argument. The extra pair of parentheses around the function prints the dataset to the console. This is a tool I primarily use for teaching purposes.

Once we've read these datasets, we'll check the structure of each in the Environment pane.

```
( respspring = read.table("spring_87_resp.txt", header = TRUE) )
```

```
   Sample   Date  Resp
1      26 Feb-87 0.057
2      23 Feb-87 0.085
3      25 Feb-87 0.159
4      27 Feb-87 0.266
5      24 Feb-87 0.368
6      19 Jan-87 0.074
7      20 Jan-87 0.089
8      22 Jan-87 0.117
9      18 Jan-87 0.135
10     21 Jan-87 0.287
11     29 Mar-87 0.063
12     32 Mar-87 0.064
13     31 Mar-87 0.073
14     30 Mar-87 0.074
15     28 Mar-87 0.093
16     41 May-87 0.092
17     40 May-87 0.097
18     38 May-87 0.178
19     42 May-87 0.267
20     39 May-87 0.302
21     35 Apr-87 0.097
22     34 Apr-87 0.105
23     36 Apr-87 0.116
24     33 Apr-87 0.176
25     37 Apr-87 0.185
26     45 Jun-87 0.043
27     47 Jun-87 0.122
28     44 Jun-87 0.175
29     43 Jun-87 0.207
30     46 Jun-87 0.523
```

```
( respfall = read.table("fall_87_resp.txt", header = TRUE) )
```

```
   Sample   Date  Resp
1      53 Aug-87 0.093
2      55 Aug-87 0.111
3      54 Aug-87 0.143
4      57 Aug-87 0.205
5      56 Aug-87 0.224
6      51 Jul-87 0.058
7      52 Jul-87 0.081
8      48 Jul-87 0.089
9      49 Jul-87 0.106
10     50 Jul-87 0.119
```

```
11     68 Nov-87 0.050
12     71 Nov-87 0.070
13     70 Nov-87 0.080
14     72 Nov-87 0.094
15     69 Nov-87 0.114
16     67 Oct-87 0.063
17     63 Oct-87 0.065
18     66 Oct-87 0.072
19     65 Oct-87 0.086
20     64 Oct-87 0.107
21     61 Sep-87 0.080
22     59 Sep-87 0.085
23     62 Sep-87 0.121
24     60 Sep-87 0.207
25     58 Sep-87 0.274
26     75 Dec-87 0.023
27     76 Dec-87 0.052
28     74 Dec-87 0.055
29     73 Dec-87 0.069
30     77 Dec-87 0.076
```

The structure of the spring dataset:

```
'data.frame':   30 obs. of  3 variables:
 $ Sample: int  26 23 25 27 24 19 20 22 18 21 ...
 $ Date  : chr  "Feb-87" "Feb-87" "Feb-87" "Feb-87" ...
 $ Resp  : num  0.057 0.085 0.159 0.266 0.368 0.074 0.089 0.117 0.135 0.287 ...
```

The structure of the fall dataset:

```
'data.frame':   30 obs. of  3 variables:
 $ Sample: int  53 55 54 57 56 51 52 48 49 50 ...
 $ Date  : chr  "Aug-87" "Aug-87" "Aug-87" "Aug-87" ...
 $ Resp  : num  0.093 0.111 0.143 0.205 0.224 0.058 0.081 0.089 0.106 0.119 ...
```

Stacking two datasets with `rbind()`

We want to combine these two datasets by stacking one on top of the other. For the sake of organization, let's add a column to each dataset to represent `season` before we combine them. This is actually relatively straightforward in R. We'll need to define a new variable in the dataset and assign whatever values we want to that variable.

In this case, we'll make a new variable called `season` with a value of `spring` in the `respspring` dataset. R handily repeats the value of spring for all rows of the dataset. This is referred to as *recycling*, and can be very efficient. Be careful, though, as recycling can also lead to mistakes if you are assigning more than one value to a new variable.

```
head(respspring)   # The original dataset only has 3 variables
```

```
  Sample   Date  Resp
1     26 Feb-87 0.057
2     23 Feb-87 0.085
3     25 Feb-87 0.159
4     27 Feb-87 0.266
5     24 Feb-87 0.368
6     19 Jan-87 0.074
```

```
respspring$season = "spring"   # Add the column "season" with the category of "spring"
head(respspring)   # Now there is a 4th variable names "season"
```

```
  Sample   Date  Resp season
1     26 Feb-87 0.057 spring
2     23 Feb-87 0.085 spring
3     25 Feb-87 0.159 spring
4     27 Feb-87 0.266 spring
```

```
5     24 Feb-87 0.368 spring
6     19 Jan-87 0.074 spring
```

Next we'll add the `season` variable to `respfall` with a value of `fall`.

```
respfall$season = "fall"
head(respfall)
```

```
  Sample   Date  Resp season
1     53 Aug-87 0.093   fall
2     55 Aug-87 0.111   fall
3     54 Aug-87 0.143   fall
4     57 Aug-87 0.205   fall
5     56 Aug-87 0.224   fall
6     51 Jul-87 0.058   fall
```

Now we can combine these two datasets into a single dataset using the `rbind()` function. The `r` in `rbind()` stands for row.

This is an example of a help page that is harder to navigate, but still contains useful information if you delve in far enough.

```
?rbind
```

There is important information buried deep in the "Data frame methods" section. The function `rbind()` stacks all the rows in the datasets based on *matching names*, not on position. Do our names match between datasets?

```
names(respspring)
```

```
[1] "Sample" "Date"   "Resp"   "season"
```

```
names(respfall)
```

```
[1] "Sample" "Date"   "Resp"   "season"
```

We made our names all the same on purpose. What if we hadn't? We can change column names by *assigning* new ones. Below we will change the name for the `season` column in `respfall` to `Season` (note the capital "S"). We replace the four old names with four new ones.

```
names(respfall) = c("Sample", "Date"  , "Resp" ,  "Season")
names(respfall)
```

```
[1] "Sample" "Date"   "Resp"   "Season"
```

If you want to change just one name without having to write all the names out, you can use the extract function `[`. In R, brackets represent the extract function.

```
?"["
```

We want to *extract* and change just the fourth name in `respfall`.

```
names(respfall)
```

```
[1] "Sample" "Date"   "Resp"   "Season"
```

```
names(respfall)[4]  # the 4th column name only
```

```
[1] "Season"
```

To change just the fourth column name, we extract it and assign a new name.

```
names(respfall)[4] = "season" # assign the 4th name back to lower case
names(respfall)
```

```
[1] "Sample" "Date"   "Resp"   "season"
```

Now let's finally stack the two respiration datasets together with `rbind()`. We'll name our new dataset `respall`. Here I list `respspring` first within the function, but it really doesn't matter.

```
respall = rbind(respspring, respfall)
summary(respall)
```

```
    Sample          Date                Resp            season
Min.   :18.00   Length:60          Min.   :0.02300   Length:60
1st Qu.:32.75   Class :character   1st Qu.:0.07375   Class :character
Median :47.50   Mode  :character   Median :0.09550   Mode  :character
Mean   :47.50                      Mean   :0.12935
3rd Qu.:62.25                      3rd Qu.:0.16300
Max.   :77.00                      Max.   :0.52300
```

Merging two datasets

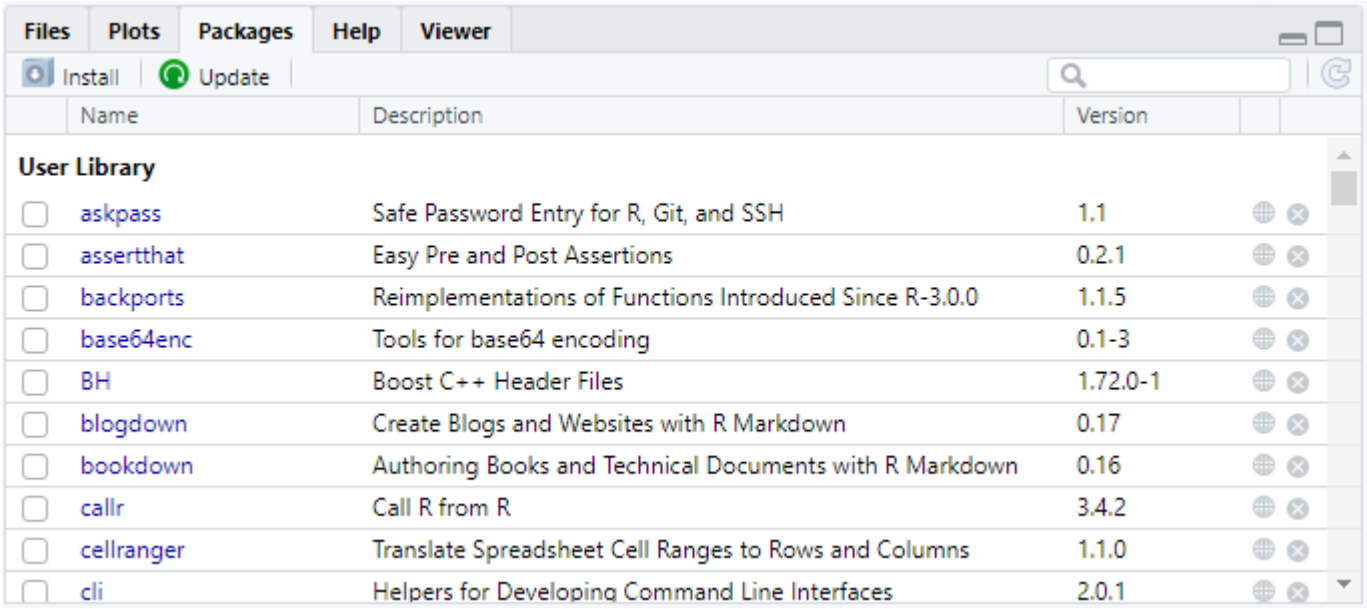
Now we have all of our respiration information in `respall` and all of our temperature information in `temperature`. We want these in one dataset for analysis, so we'll need to merge these two datasets together. The unique identifier for each sample taken is called `Sample`, and is in both datasets. The unique identifier is how we match the rows in one dataset to the rows in the other dataset during the merging process.

Check your Environment pane, though. The `temperature` dataset has one less row than `respall`. Which `Sample` is missing from `temperature`? Let's check.

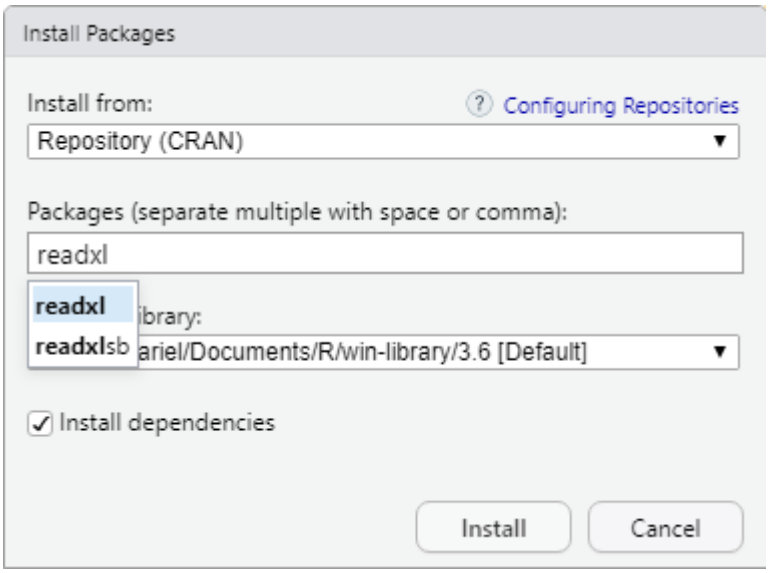
Finding values in one dataset that are not in another

If we were working in Excel, we might order the dataset by `Sample` and then scan through until we found a missing value. This isn't very efficient in R, though. A better way would be to use the handy function `%in%`, which involves *matching*. However, the easiest way to do this I've found is to use the `anti_join()` function from package **dplyr** to do this.

Installing an add-on package Package **dplyr** is already installed on the computers in the lab classroom but you may need to install it if you are working on this on your own computer. You can install packages through RStudio's `Packages` pane. Note the `Install` button in the picture below.



You need to know the package name that you want to install and type it in the **Packages** field in the pop-up window that comes up after clicking `Install`. The picture below shows installing package **readxl**.



You could also run the code `install.packages("dplyr")` in the Console to do the same thing.

Packages only need to be installed one time onto a computer, and will be available in future R sessions. There is absolutely no need to go through the trouble of reinstalling it every time you open R. For that reason, you shouldn't include `install.packages()` code in a working script.

Loading an add-on package Once the package is installed, you can load a package into R using the `library()` function.

Unlike package installation, you do need to load add-on packages each time you use then in a new R session. Here's a nice picture by Dianne Cook that explains `install.packages()` vs `library()` well.

Let's load **dplyr**.

```
library(dplyr) # using v. 1.0.7 in winter 2022
```

```
?anti_join
```

We can see from the `anti_join()` help page that the function returns all rows from the first dataset (the x dataset) that are **NOT** in the second dataset (the y dataset). We want to see which value of `Sample` is in the `respall` dataset that is NOT in the `temperature` dataset. This means `respall` will be our x dataset in `anti_join()`.

We'll also use the `by` argument to define which variable in the two datasets we want to match on (more about this in the next section).

```
anti_join(respall, temperature, by = "Sample")
```

```
  Sample   Date  Resp season
1     21 Jan-87 0.287 spring
```

We can see that the `temperature` dataset is missing sample 21. In a real analysis, we would spend some time investigating why there was no temperature value taken for that sample.

Joining two datasets with `inner_join()`

We can join all the temperature and respiration information into a single dataset using some of the other `join` functions from package **dplyr**.

Per the documentation, an *inner join* will:

> return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

This tells us that we will end up with only the samples that are in both datasets after joining.

Let's see what that looks like, using the respiration dataset as the "x" dataset and the temperature dataset as the "y" dataset. We'll join on `Sample` like we did above with the anti join. I name the new object `resptemp` and take a look at the result.

```
resptemp = inner_join(x = respall, y = temperature, by = "Sample")
head(resptemp)
```

```
  Sample   Date  Resp season   Tech Temp DryWt
1     26 Feb-87 0.057 spring   Mark  5.5 0.640
2     23 Feb-87 0.085 spring Fatima  5.5    NA
3     25 Feb-87 0.159 spring Fatima  5.5 0.610
4     27 Feb-87 0.266 spring  Raisa  5.5 0.620
5     24 Feb-87 0.368 spring   Cita  5.5 0.565
6     19 Jan-87 0.074 spring Nitnoy  4.5 0.607
```

```
str(resptemp)
```

```
'data.frame':   59 obs. of  7 variables:
 $ Sample: int  26 23 25 27 24 19 20 22 18 29 ...
 $ Date  : chr  "Feb-87" "Feb-87" "Feb-87" "Feb-87" ...
 $ Resp  : num  0.057 0.085 0.159 0.266 0.368 0.074 0.089 0.117 0.135 0.063 ...
 $ season: chr  "spring" "spring" "spring" "spring" ...
 $ Tech  : chr  "Mark" "Fatima" "Fatima" "Raisa" ...
 $ Temp  : num  5.5 5.5 5.5 5.5 5.5 4.5 4.5 4.5 4.5 5 ...
 $ DryWt : num  0.64 NA 0.61 0.62 0.565 0.607 0.597 0.603 0.569 0.634 ...
```

The above works if the names in the two datasets are the same. What if they are different? Let's test by making a second temperature dataset called `temp2`, and change the name of `Sample` to `Samplenum`. Note that `Sample` is the first column in the dataset.

This is good practice on using the extract brackets and assigning new names.

```
temp2 = temperature
names(temp2)[1] = "Samplenum"
```

To join datasets when the matching variable has different names in the two different datasets, we must list both names to `by`. The first listed is for the "x" dataset and the second is for the "y" dataset. You can see what the code looks like below.

I print only the first six lines of the result in this document. There, you can see the name of the column in the joined dataset comes from the "x" dataset.

```
inner_join(x = respall, y = temp2, by = c("Sample" = "Samplenum") )
```

```
  Sample   Date  Resp season   Tech Temp DryWt
1     26 Feb-87 0.057 spring   Mark  5.5 0.640
2     23 Feb-87 0.085 spring Fatima  5.5    NA
```

```
3     25 Feb-87 0.159 spring Fatima  5.5 0.610
4     27 Feb-87 0.266 spring  Raisa  5.5 0.620
5     24 Feb-87 0.368 spring   Cita  5.5 0.565
6     19 Jan-87 0.074 spring Nitnoy  4.5 0.607
```

Getting back to our joined dataset, did you notice there are only 59 rows in the joined dataset `resptemp` we made above?

```
nrow(resptemp)
```

```
[1] 59
```

This is because an inner join drops any rows with samples that aren't in both datasets. Since we are missing sample 21 in the temperature dataset, R dropped this row from the joined dataset. We can use a different kind of join to keep rows that are missing from one or both datasets. In this case, let's use `left_join()`.

From the documentation, a left join will:

> return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

In this case we keep all rows of the *left* (i.e., "x") dataset regardless if there is a match in the second dataset. Since `respall` is the dataset with all 60 rows, we list that one first.

You can see that the new object `resptemp`, made with a left join, has 60 rows instead of 59.

```
resptemp = left_join(x = respall, y = temperature, by = "Sample")
nrow(resptemp)
```

```
[1] 60
```

If we look at the first 10 rows of the dataset you can see that the temperature data for sample 21 are all filled with `NA`. Note the `n` argument in `head()`.

```
head(resptemp, n = 10)
```

```
   Sample   Date  Resp season   Tech Temp DryWt
1      26 Feb-87 0.057 spring   Mark  5.5 0.640
2      23 Feb-87 0.085 spring Fatima  5.5    NA
3      25 Feb-87 0.159 spring Fatima  5.5 0.610
4      27 Feb-87 0.266 spring  Raisa  5.5 0.620
5      24 Feb-87 0.368 spring   Cita  5.5 0.565
6      19 Jan-87 0.074 spring Nitnoy  4.5 0.607
7      20 Jan-87 0.089 spring  Raisa  4.5 0.597
8      22 Jan-87 0.117 spring Nitnoy  4.5 0.603
9      18 Jan-87 0.135 spring   Mark  4.5 0.569
10     21 Jan-87 0.287 spring   <NA>   NA    NA
```

We finally have all our data in a single dataset to work with, which means we've made good practice. Now we can focus a little more on working the variables in this dataset to learn more about how R works.

Creating new variables in a dataset based on existing variables

Now that we have a single dataset to work with, let's practice creating a new variable in a dataset that is based on existing variables. We'll first calculate temperature in degrees Fahrenheit from temperature in degrees Celsius and add it to the `resptemp` dataset with the name `tempf`.

The dollar sign notation can get tedious once you start adding variables to datasets. R has several built-in functions to help with this while still avoiding the `attach()` function, including `with()` and `transform()`. The `mutate()` function from **dplyr** is also available for this. We will be using `mutate()` today; note **dplyr** must be loaded to use `mutate()`.

In `mutate()`, the first argument is the dataset we want to add variables to. We then create one or more new variables, assigning variable names and using existing variables to create the new ones. We don't need any dollar sign notation, as `mutate()` allows us to both assign new variable names and refer to existing variables without it.

When using `mutate`, I generally name the *mutated* dataset (the one with the new column in it) the same as the original dataset. While we won't see it today, we can make multiple new variables at once in `mutate` by separating the new variables

with commas.

```
resptemp = mutate(resptemp, tempf = 32 + ( (9/5)*Temp) )
```

Take a look at `resptemp` with the new variable in it.

```
head(resptemp)
```

```
  Sample   Date  Resp season   Tech Temp DryWt tempf
1     26 Feb-87 0.057 spring   Mark  5.5 0.640  41.9
2     23 Feb-87 0.085 spring Fatima  5.5    NA  41.9
3     25 Feb-87 0.159 spring Fatima  5.5 0.610  41.9
4     27 Feb-87 0.266 spring  Raisa  5.5 0.620  41.9
5     24 Feb-87 0.368 spring   Cita  5.5 0.565  41.9
6     19 Jan-87 0.074 spring Nitnoy  4.5 0.607  40.1
```

Remember that our question of interest is about differences in mean respiration between two temperature categories. Right now we have a quantitative variable for temperature (`Temp`) instead of a categorical one. We can create a categorical variable based on `Temp` using `ifelse()`. In our case, if temperature in Celsius is less than 8 degrees the row will be placed in the `Cold` category, otherwise the row will be put in the `Hot` category.

```
?ifelse
```

In `ifelse`, we list the *condition* we want to test first. If the result of the test is `TRUE` for a row in the dataset, the first value given is assigned to that row. If the result of the test is `FALSE`, the second value given is assigned.

Example code, combined with `mutate()` to add the new variable to the `resptemp` dataset, is below. The condition we use is that the value of `Temp` is less than 8°.

```
resptemp = mutate(resptemp, tempgroup = ifelse(Temp < 8, "Cold", "Hot") )
resptemp$tempgroup
```

```
 [1] "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" NA     "Cold" "Cold" "Cold"
[14] "Cold" "Cold" "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"
[27] "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"
[40] "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"  "Hot"
[53] "Hot"  "Hot"  "Hot"  "Cold" "Cold" "Cold" "Cold" "Cold"
```

```
str(resptemp)
```

```
'data.frame':   60 obs. of  9 variables:
 $ Sample   : int  26 23 25 27 24 19 20 22 18 21 ...
 $ Date     : chr  "Feb-87" "Feb-87" "Feb-87" "Feb-87" ...
 $ Resp     : num  0.057 0.085 0.159 0.266 0.368 0.074 0.089 0.117 0.135 0.287 ...
 $ season   : chr  "spring" "spring" "spring" "spring" ...
 $ Tech     : chr  "Mark" "Fatima" "Fatima" "Raisa" ...
 $ Temp     : num  5.5 5.5 5.5 5.5 5.5 4.5 4.5 4.5 4.5 NA ...
 $ DryWt    : num  0.64 NA 0.61 0.62 0.565 0.607 0.597 0.603 0.569 NA ...
 $ tempf    : num  41.9 41.9 41.9 41.9 41.9 40.1 40.1 40.1 40.1 NA ...
 $ tempgroup: chr  "Cold" "Cold" "Cold" "Cold" ...
```

Working with missing values in R

If we look at the `summary()` of `resptemp`, we can see we have some missing values, represented in R as `NA`.

```
summary(resptemp)
```

```
     Sample          Date                Resp            season             Tech
 Min.   :18.00   Length:60          Min.   :0.02300   Length:60          Length:60
 1st Qu.:32.75   Class :character   1st Qu.:0.07375   Class :character   Class :character
 Median :47.50   Mode  :character   Median :0.09550   Mode  :character   Mode  :character
 Mean   :47.50                      Mean   :0.12935
 3rd Qu.:62.25                      3rd Qu.:0.16300
 Max.   :77.00                      Max.   :0.52300
```

```
      Temp           DryWt            tempf        tempgroup
 Min.   : 4.50   Min.   :0.5280   Min.   :40.10   Length:60
 1st Qu.: 7.00   1st Qu.:0.6262   1st Qu.:44.60   Class :character
 Median :11.50   Median :0.7090   Median :52.70   Mode  :character
 Mean   :10.81   Mean   :0.7086   Mean   :51.46
 3rd Qu.:14.25   3rd Qu.:0.7857   3rd Qu.:57.65
 Max.   :19.00   Max.   :0.8590   Max.   :66.20
 NA's   :1       NA's   :2        NA's   :1
```

R treats missing values differently than other software packages you may have used, so we'll spend a couple minutes talking about them. For example, look what happens if we take the mean of the variable `DryWt` with the `mean()` function. The `DryWt` variable contains a missing value.

```
mean(resptemp$DryWt)
```

```
[1] NA
```

A missing value is something that we have no value for. In R logic, if we try to average something that has no value (I think of this as something that doesn't exist) with some actual values, the result is impossible to calculate and so returns `NA`. When you have missing values in R, you will need to specifically decide what you want to do with them as R isn't going to just ignore them for you.

The `na.rm` argument

Many functions have the argument `na.rm` for dealing with missing values. This stands for "NA remove", and tells the function to remove any missing values before applying the function. This is true for `mean()`, which you can see in the help page (`?mean`).

```
mean(resptemp$DryWt, na.rm = TRUE)
```

```
[1] 0.7086379
```

Using `na.omit` to remove rows with missing values

If we didn't want any rows that had missing values anywhere in our dataset, we could remove them all with `na.omit()`. Here we could make a new dataset called `resptemp2` that contains no missing values. You can see it has two less rows (58) than `resptemp` when we look in our RStudio Environment pane.

```
resptemp2 = na.omit(resptemp)

nrow(resptemp2)
```

```
[1] 58
```

Other functions for working with `NA`

There are other functions to use when working with missing values, including `is.na()` and `complete.cases()`. You should check out the help pages for those if you are interested. We'll see an example of using `is.na()` in a few minutes, but not in any great detail.

Saving a dataset

We just went to the trouble of making a single dataset from the three original datasets. Right now, it only exists within our current R session unless we save the `.RData`. While we could always recreate it because we have all of our R code saved in a script, sometimes it's worth saving a dataset you've created. Let's save the combined dataset as a comma-delimited file called `combined_resp_and_temp_data.csv` using the `write.csv()` function. If you wanted to save the file somewhere other than our working directory you'd need to write out the path to that directory.

We'll use the `row.names` set to `FALSE` so the row names that R makes won't be written into the file.

```
?write.csv

write.csv(x = resptemp,
```

```
          file = "combined_resp_and_temp_data.csv",
          row.names = FALSE)
```

Data exploration

Before embarking on an analysis, we'll need to spend time exploring the dataset. This usually involves calculating useful data summaries and creating exploratory graphics to understand the dataset. This process allows us to check for oddities in the data as well as to start thinking about whatever assumptions we might make need to make in our statistical model.

Summary statistics

In this class we'll be making group summaries using functions `group_by()` and `summarise()` from package **dplyr**. Today you will see the code, but we will not discuss it in detail until lab 2. RStudio has a nice data transformation cheat sheet to download (https://raw.githubusercontent.com/rstudio/cheatsheets/main/data-transformation.pdf) for those who want to get started understanding data wrangling with **dplyr**.

We want separate summary statistics of `Resp` for each `tempgroup`. We will calculate the range (minimum and maximum) as well as the median, mean, and standard deviation for each group. Note that because there is a missing value in `tempgroup` we get a third set of summary statistics.

```
summarise( group_by(resptemp, tempgroup),
           across(.cols = "Resp",
                  .fns = list(min = min,
                              median = median,
                              mean = mean,
                              max = max,
                              sd = sd) ) )
```

```
# A tibble: 3 x 6
  tempgroup Resp_min Resp_median Resp_mean Resp_max Resp_sd
  <chr>        <dbl>       <dbl>     <dbl>    <dbl>   <dbl>
1 Cold         0.023       0.074     0.105    0.368  0.0826
2 Hot          0.043       0.106     0.137    0.523  0.0898
3 <NA>         0.287       0.287     0.287    0.287 NA
```

Exploratory graphics

Most of the data exploration I do is with graphics. We'll be making a variety of exploratory graphics here so you can get an idea for the type of plot that you might like to use in your own work.

Today we'll be using the function `qplot()` from package **ggplot2** to make simple exploratory graphics. The *q* in `qplot` stands for quick, so we use this function for exploratory graphics because we don't need to spend time making exploratory graphics look nice.

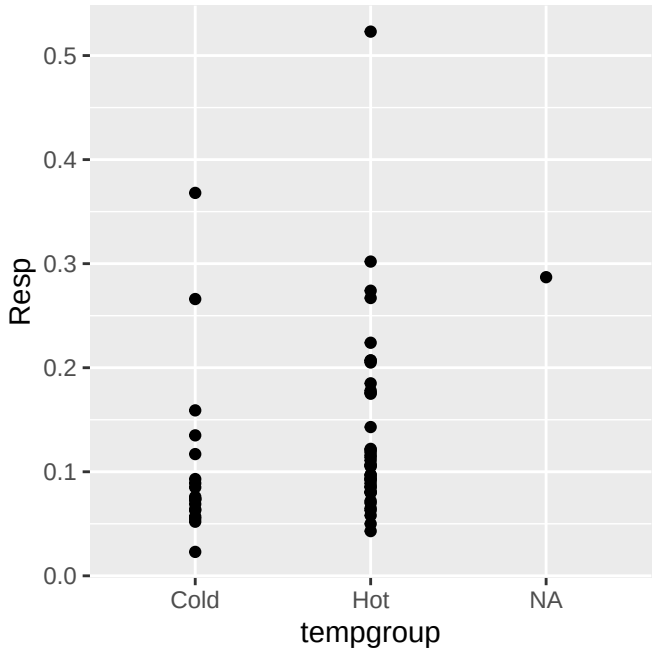
The use of `qplot()` for exploratory graphics is likely review for most people in the class.

Load package **ggplot2** using the `library()` function. This package is already installed on the lab computers.

```
library(ggplot2) # v. 3.3.5 in 2022
```

We'll start with a scatterplot of `Resp` by `tempgroup`, with `Resp` on the y axis and `tempgroup` on the x axis. We will use the `data` argument to define the dataset that contains all of our variables.

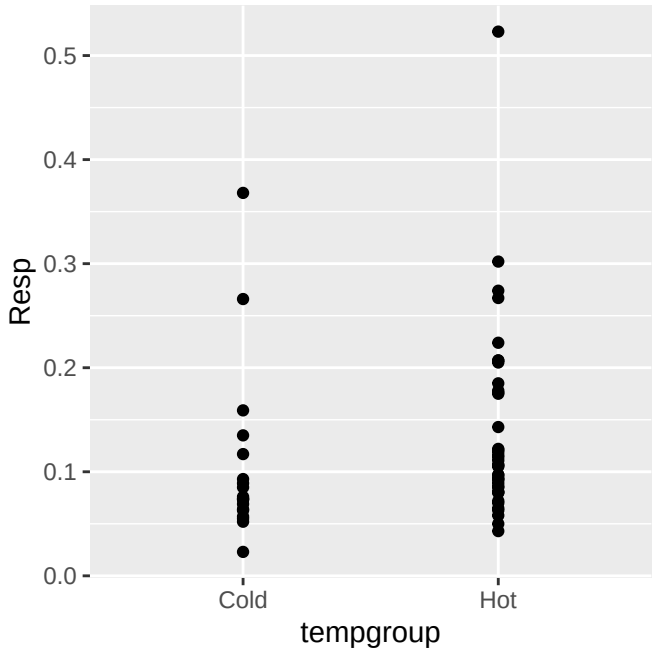
```
qplot(x = tempgroup, y = Resp, data = resptemp)
```



Notice we have a missing value on the x axis. We don't want to remove all the rows in the whole dataset that are missing with `na.omit()`, but we can remove the rows missing `tempgroup` by filtering them out of the dataset.

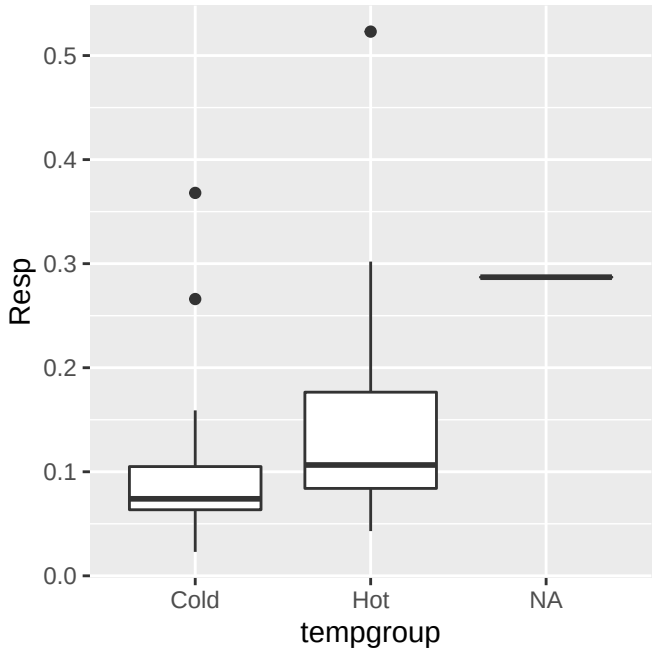
One option for this is to use the `filter()` function from package **dplyr** with `is.na()`. Because we want to remove all the rows where `tempgroup` is NOT `NA`, so we use `is.na()` with `!` (i.e., the *not* operator). This is a nice first example of how the `filter()` function is useful for subsetting rows of datasets, although we don't have time to talk about it in detail today.

```
qplot(x = tempgroup, y = Resp,
      data = filter(resptemp, !is.na(tempgroup) ) )
```



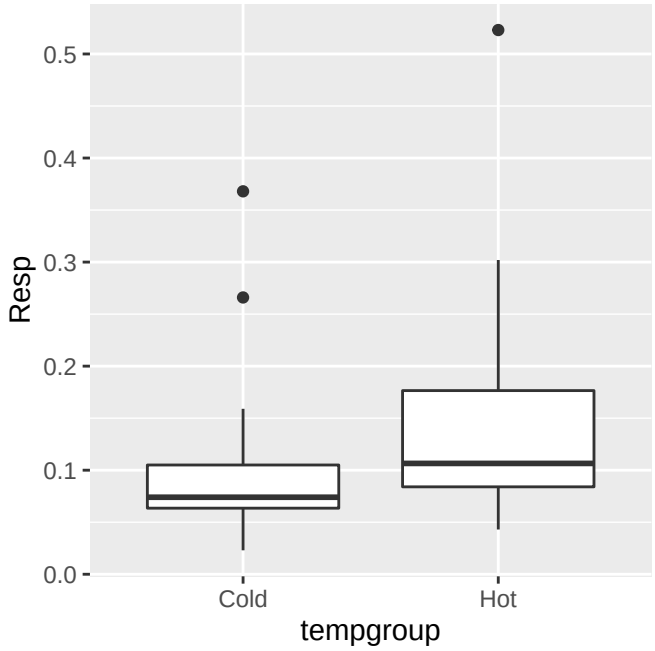
We can make a boxplot instead of a scatterplot by adding the `geom` argument.

```
qplot(x = tempgroup, y = Resp,
      data = resptemp,
      geom = "boxplot")
```



And, again, we might want to take out that missing value

```
qplot(x = tempgroup, y = Resp,
      data = filter(resptemp, !is.na(tempgroup) ),
      geom = "boxplot")
```

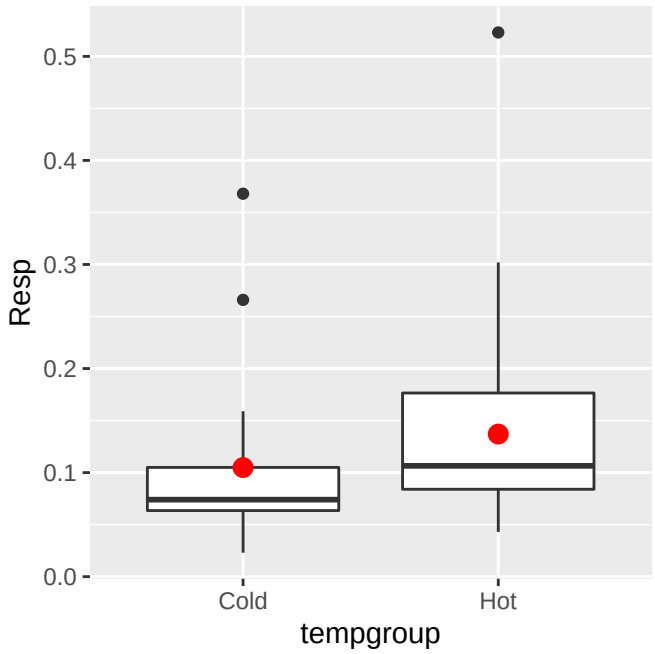


At this point I decided to make a new dataset without that missing `tempgroup` value. We won't be using that value in any analyses today, and it seems like a nuisance. I name the new dataset `resptemp2`.

```
resptemp2 = filter(resptemp, !is.na(tempgroup) )
```

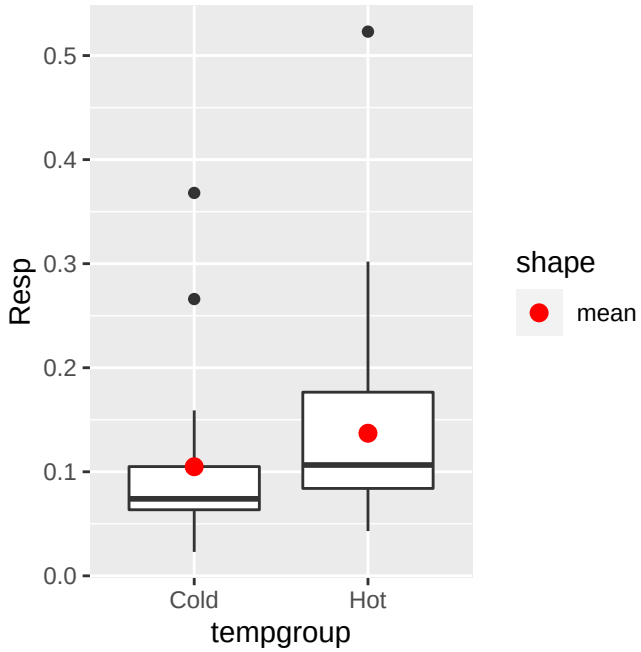
The boxplot shows the medians, range (minimum and maximum), and interquartile range (25% and 75% of data) but not the means. I'll add a *layer* to the graphic to add the mean of each group on top of the boxplot as a red dot using `stat_summary()`. We will not be going through this code in detail, but I wanted you to have some example code of doing this that you can explore later on your own.

```
qplot(x = tempgroup, y = Resp,
      data = resptemp2,
      geom = "boxplot") +
     stat_summary(fun = mean, geom = "point",
                  color = "red", size = 3)
```



To include a legend to indicate that the mean is a point, we can add `aes()` to `stat_summary()`.

```
qplot(x = tempgroup, y = Resp,
      data = resptemp2,
      geom = "boxplot") +
     stat_summary(fun = mean, aes(shape = "mean"),
                  geom = "point", color = "red", size = 3)
```

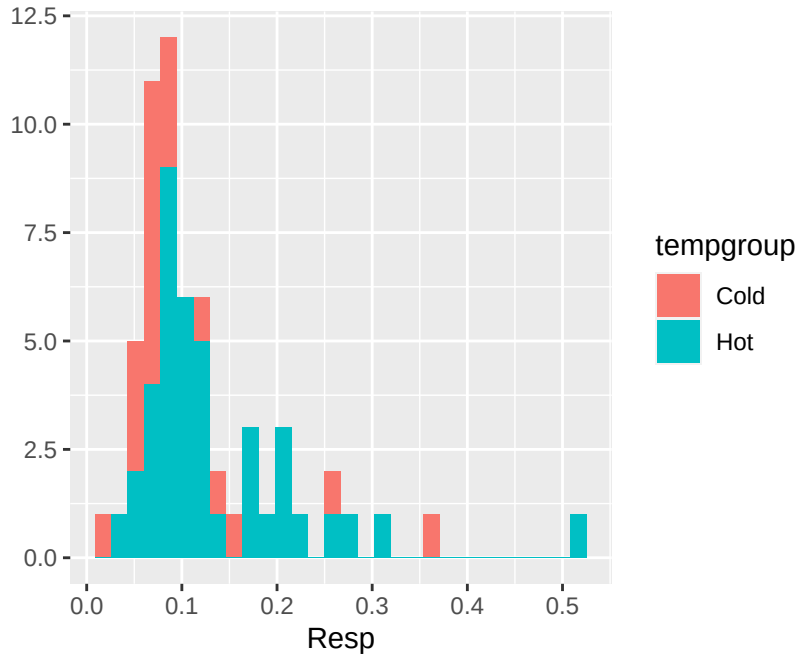


Histograms and density plots are other commonly used exploratory plots. Here we make two histograms, one for each group, by setting different colors for each temperature group using the `fill` argument. The variable of interest in a histogram is on

the x axis, so we put `Resp` on the x axis. Notice we get an informative message from R about the default number of bins used.

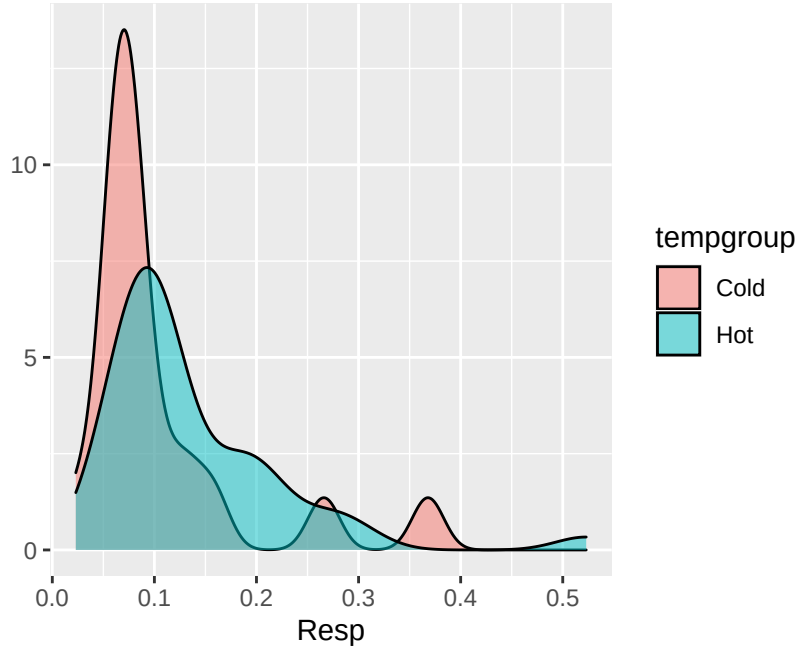
```
qplot(x = Resp, fill = tempgroup,
      data = resptemp2,
     geom = "histogram")
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



A density plot is essentially a smoothed version of a histogram. Here are the density plots for each group, distinguished by `fill` color. To make the fill color more transparent instead of totally opaque we can set `alpha` to less than 1.

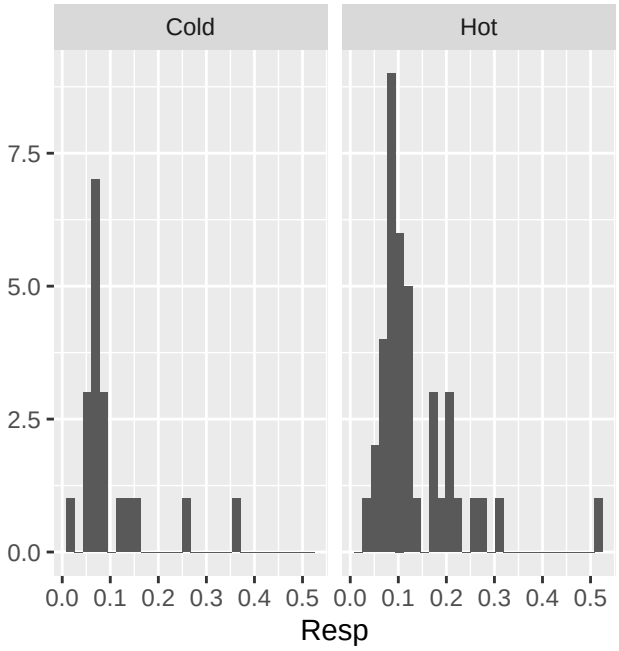
```
qplot(x = Resp, fill = tempgroup,
      data = resptemp2,
      geom = "density",
      alpha = I(.5) )
```



We can plot each group's histogram as a separate plot by using what are called *facets*. Here are the histograms using

`tempgroup` as a facet instead of as fill color.

```
qplot(x = Resp,
      data = resptemp2,
      geom = "histogram",
      facets = ~tempgroup)
```



Analysis using a two-sample test (finally!)

Let's compare the mean respiration rate between temperature groups with a two-sample test. Because we are working with a two sample test, we checked our variance and distributional assumptions by exploring the observed data. This is very unusual; as you know, we will usually be checking assumptions based on the *residuals*, not the observed data.

Each of us would have to decide if the assumptions are reasonably met for a two-sample t-test, a Welch two-sample t-test if we don't want to assume the variances are equal, or possibly a Wilcoxon rank-sum test if data are extremely skewed. We will focus just on the function for t-tests in lab today but you can use any analysis you think appropriate on assignments. In R, there is a built-in function `t.test()`.

```
?t.test
```

Below is an example of two different t-tests. I name each test, but also print the results to the Console using an extra pair of parentheses. I am writing these in the *formula* format, with the response variable listed first and the explanatory variable after the tilde. I also define the dataset the variables are in with the `data` argument.

```
( respunequal = t.test(Resp ~ tempgroup, data = resptemp2) )
```

```
    Welch Two Sample t-test

data:  Resp by tempgroup
t = -1.3605, df = 38.324, p-value = 0.1816
alternative hypothesis: true difference in means between group Cold and group Hot is not equal to 0
95 percent confidence interval:
 -0.08011773  0.01570194
sample estimates:
mean in group Cold  mean in group Hot
         0.1048421           0.1370500
```

The unequal variances t-test is the default test, as you can see in the documentation, so if you wanted to assume the variances were equal you need to change the `var.equal` argument to `TRUE`.

```
( respequal = t.test(Resp ~ tempgroup, data = resptemp2, var.equal = TRUE) )
```

```
    Two Sample t-test

data:  Resp by tempgroup
t = -1.3199, df = 57, p-value = 0.1921
alternative hypothesis: true difference in means between group Cold and group Hot is not equal to 0
95 percent confidence interval:
 -0.08107123  0.01665544
sample estimates:
mean in group Cold  mean in group Hot
         0.1048421          0.1370500
```

For a Wilcoxon rank-sum test, see the `wilcox.test()` function.

Wrapping up an analysis

At the end of an analysis we need to decided how to display the results of any test in our report. This will often involve creating a high-quality graphic and/or a nice summary table.

The results from a two-sample t-test can easily be written in the text of a document, and making a graphic of the results from a single test is overkill. Because of this, I decided to create a graphic with a boxplot for each group with the observed respiration measurements overlaid as a dotplot (i.e., a histogram of dots) and the group means shown as a separate point.

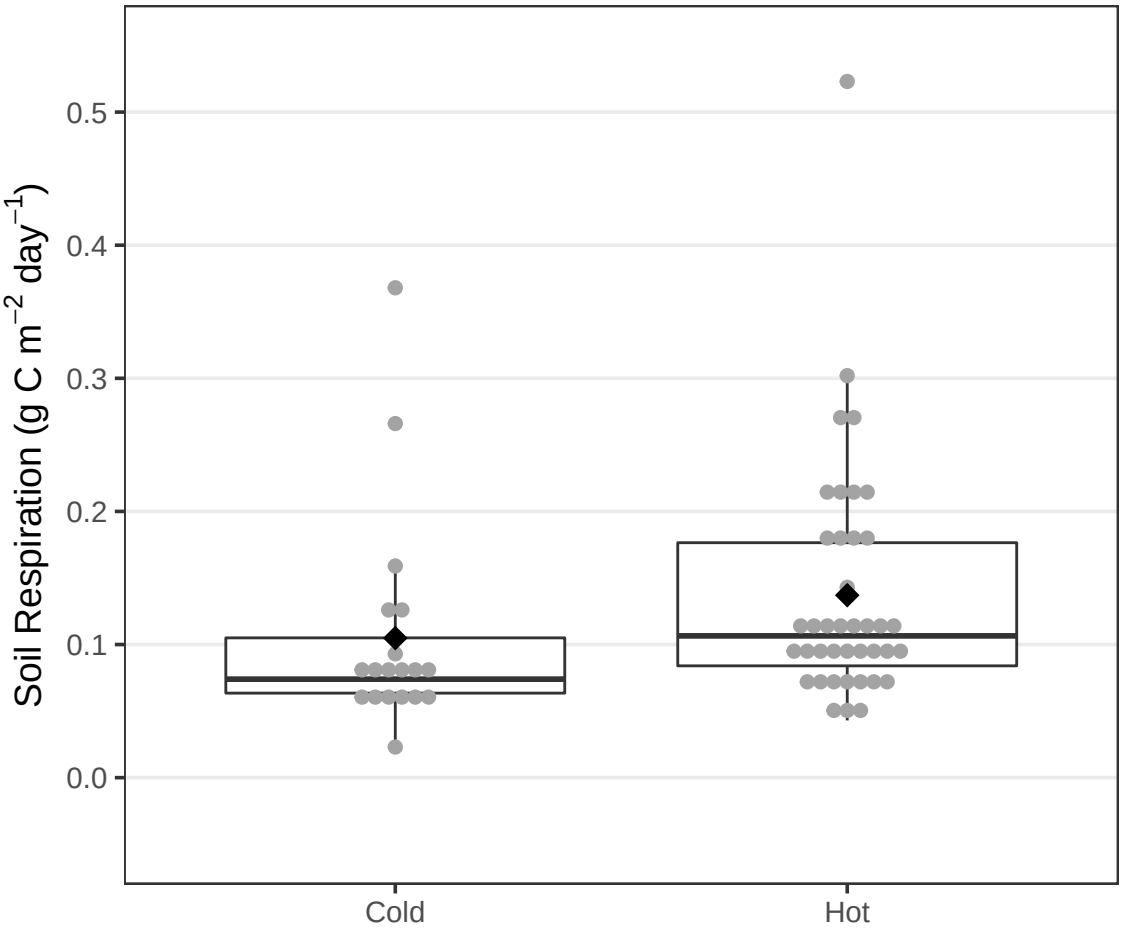
Creating a final graphic

I will switch to using the `ggplot()` function from the package **ggplot2** to make my high quality graphic. I add layers to improve how the graphic looks, including changing the background color, dot color, removing some of the grid lines, and adding appropriate units to the axis titles. I also used `base_size` to increase the size of all plot text.

We will see many plots using **ggplot2** this quarter, but won't have time to learn the package in any detail. Download the **ggplot2** cheat sheet from RStudio (https://raw.githubusercontent.com/rstudio/cheatsheets/main/data-visualization.pdf) if you are interested in learning more **ggplot2** details.

The plot is printed here at 6 inches X 5 inches.

```
# A plot that summarises the data
# Don't forget to add a caption in your write-up
g1 = ggplot(resptemp2, aes(x = tempgroup, y = Resp) ) + # define plot axes
     geom_boxplot() + # Add boxplots for each group
     geom_dotplot(binaxis = "y", stackdir = "center",
                  dotsize = .5, fill = "grey64", colour = "grey64") + # Dotplot by group
     stat_summary(fun = mean, geom= "point", shape = 18, size = 4) + # Add means to the plot
     theme_bw(base_size = 14) + # Change theme (remove grey background), increase text size
     theme(panel.grid.major.x = element_blank(), # Remove gridlines from x axis
           panel.grid.minor.y = element_blank() ) + # Remove minor gridlines from y
     labs(y = expression(paste("Soil Respiration ",
                               "(g ", "C ", m^-2, " ", day^-1, ")",
                               sep = "" )),
          x = NULL) + # Change y axis labels with complex units,
                      # remove x axis labels
     scale_y_continuous(breaks = seq(0, .5, by = .1), # Add more breaks
                        limits = c(-.05, .55) ) # Change the limits of the y axis
g1
```



Making a table of summary statistics

I made a summary table of descriptive statistics by group, as well, including the sample size, mean, and standard deviation. I used function `summarise()` from package **dplyr** to make the summary table. We will learn more about how to use `summarise()` in later lab examples. If we hadn't loaded **dplyr** already we could do it here; if the package is already loaded then loading it again doesn't do anything.

Note it is up to you to decide what to put in your report. However, if you put something in it must be referred to in the text at some point. If you do use a summary table, don't forget to round to a reasonable number of significant digits (I used three here but it depends on the variable).

```
# Make a quick summary table using package dplyr
# You need to load package only if you didn't load it earlier
library(dplyr)
sumresp = summarise(group_by(resptemp2, tempgroup),
                    n = length(Resp),
                    Mean = round(mean(Resp), 3),
                    SD = round(sd(Resp), 3) )
# Change column name
names(sumresp)[1] = "Temperature"
sumresp
```

```
# A tibble: 2 x 4
  Temperature     n  Mean    SD
  <chr>       <int> <dbl> <dbl>
1 Cold           19 0.105 0.083
2 Hot            40 0.137 0.09
```

At your current level of R, copying and pasting this table into Excel is likely the easiest way to make the table look nice

enough to include in your write-up. If you use **rmarkdown** and **knitr** to write a report the `kable()` function in R makes nice tables fairly easy to make. The table below is made with `kable()`.

Temperature	n	Mean	SD
Cold	19	0.105	0.083
Hot	40	0.137	0.090

Once your graphics and tables are made, you are ready to work on your assignment. Remember to only include information you use refer to in the text; just because we make something in lab doesn't mean everyone will want to include it in their assignment.

Example write-ups in report form will be posted on Canvas each week after you turn in your assignments. Use these examples to help improve your lab assignments, as needed.

For bonus code for making a graphic that combines the statistical test results, summary statistics, and plot, see `lab1.example.bonusgraphics.handout.pdf`.