Contents

R basics: a practical introduction to R

In today's workshop, we will be learning how to use R through a practical worked example. The goal today is a pretty standard one: we want to perform a simple statistical analysis on a set of data in R. As we work towards that goal, we will learn to read datasets into R and do basic data manipulations and some graphing. I'll take some time along the way to demonstrate some common coding techniques as well as some of the pitfalls that the R beginner faces.

We will spend a fair amount of time talking about R help - where you can find it, how to search for it, and, in particular, how to use the documentation within R. In my experience, knowing how to work with datasets in R and knowing where to look for R help can take you pretty far into the world of R.

We will not be spending time on topics such as reviewing the different types of R objects and their attributes, which are commonly taught in introductory R classes. If you start to use R regularly in your work for a wider variety of tasks, a deeper knowledge of the nuts and bolts of R will likely become more important. Once you are in that situation, a great place to start is the "Introduction to R" document on CRAN: http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf. There are tons of online workshops and classes and tutorials about R, as well, so spend some time exploring!

Overall analysis goal

The overall analysis goal today is to compare mean respiration for "Cold" and "Hot" sites. As often happens with data from real studies, the information we need to use for the analysis is currently stored in three separate datasets. I provided those three datasets to you earlier this week. We will spend most of today's workshop reading these three datasets into R and combining and manipulating them in preparation for analysis.

R help

The first thing I want to go over today is how to get R help, as I think learning this will help you more than anything else I can teach you. I will be revisiting R help throughout the workshop to show you how I use the help within R in my daily R work.

There are three main places I look for help when I run into trouble in R.

Documentation for a specific function

The first place to look for help is within R, in the R help pages. 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 help page for that function. You can do this by typing `?functionname` into your Console and pressing enter. 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 help page to see how to use it. A list of the arguments the function takes and the defaults to the arguments are the first topics covered in most R help pages. The help pages in R usually contain 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.

Search keywords in R

If you don't know the name of a function for a task you want to do, you can search the R help by key words with `??keyword` (note the two questions marks). Any page that contains that key word in R will come up and you can peruse through them to see if you can find a useful function to help you do whatever it is you are trying to do. You can try this out by typing `??merging` in your Console to search R help on merging.

Stack Overflow

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 the Stack Overflow site. I prefer the set-up of Stack Overflow, as the R mailing list archives is in a threaded format which seems harder to navigate to me. Stack Overflow is currently very active with R question, and very 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

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

The working directory

We'll begin our work in R by setting what's called the *working directory*. The working directory is where R, by default, will go to look for any datasets you load and is the place R will save anything you save. When working on a 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. This also helps me keep organized when working on a project.

Checking your current working directory

To see your default working directory, use the `getwd` function to *get* your current working directory. My default working directory is my "N" drive.

```
getwd()
```

```
[1] "N:/"
```

Setting your working directory in RStudio

You can set your working directory in a variety of ways. These days I often take advantage of RStudio's drop down menus for this.

If you've navigated to the folder where you've stored your files in the RStudio "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 in the RStudio Source pane, you can use the overall drop-down menus:

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

Code to set your working directory

And, of course, you can always type out the path to your working directory using the `setwd` function.

Important: You must either use single forward slashes or double backslashes in the directory path in R instead of the single backslashes. Below is an example (not run).

```
setwd("N:/docs/Classes/R workshops/R basics/2017_R_basics")
setwd("N:\\docs\\Classes\\R workshops\\R basics\\2017_R_basics")
```

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

```
getwd()
```

```
[1] "N:/docs/Classes/R workshops/R basics/2017_R_basics"
```

Reading data into R

Reading in a text file

The respiration and temperature data are currently in three datasets that we need to combine into a single dataset for analysis. I've purposefully made the three datasets different types of files so you will have a chance to see the different functions we can use to read datasets into R. We'll start with the dataset that contains the temperature information, called `temp.txt`.

The temperature data are in a whitespace-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 things you can control with different function arguments.

```
?read.table
```

We will need to tell R that our dataset contains column names. This is commonly how we would store files, and it means that the very first row of our dataset has all the variable names in it. We will tell R that 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`.

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 other common assignment operator you will see is `<-`. 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 throughout the workshop.

If your dataset isn't in your working directory, you need to write out the path to wherever the file is located. Again, you must either use forward slashes, like I demonstrate below, or double backslashes (code not run).

```
temperature = read.table("N:/docs/Classes/R workshops/R basics/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  : Factor w/ 7 levels "Cita","Fatima",..: 4 6 5 5 7 7 1 6 6 1 ...
 $ Temp  : num  4.5 4.5 4.5 4.5 5 5 5 5 5 5.5 ...
 $ DryWt : Factor w/ 57 levels ".","0.528","0.565",..: 4 6 7 8 10 11 14 15 17 3 ...
```

Uh-oh, I see a problem right away. The `str` function tells us what kind of variable each column in the dataset contains. `DryWt` should be numeric, but R read it in as a factor. A factor in R is a type of *classification* or *categorical* variable.

We need to figure out what's going on. Let's take a closer look at just 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 the variable we want and what object that variable is stored in. There are a variety of ways to do this, but a simple way that we will use today is to use dollar sign notation. In 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. Here we tell R that we want to use the `temperature` dataset and pull out the `DyrWt` column.

```
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  .     0.64  0.656 0.661
[17] 0.685 0.695 0.701 0.528 0.574 0.619 0.627 0.642 0.62  0.65  0.67  0.728 0.679 0.753 0.759 0.77
```

```
[33] 0.781 0.786 0.727 0.785 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
57 Levels: . 0.528 0.565 0.569 0.574 0.597 0.603 0.607 0.61 0.611 0.613 0.619 0.62 0.622 ... 0.859
```

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 categorical factor.

It turns out that this dataset was used in SAS at some point, and that 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 blank fields as missing. If you use a different character to indicate a missing value 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 R read as a factor. For your reference, there are two main reasons I've seen that cause this 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 simply 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.

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

How does the structure look now?

```r
str(temperature)
```

```
'data.frame':   59 obs. of  4 variables:
 $ Sample: int  18 20 22 19 31 30 28 32 29 24 ...
 $ Tech  : Factor w/ 7 levels "Cita","Fatima",..: 4 6 5 5 7 7 1 6 6 1 ...
 $ 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 ...
```

Initial exploration of a dataset

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.

```r
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 via the RStudio method.

You can look at only 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 first 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 and a tally of the number of observations in each category (aka *levels*) for factors.

```
summary(temperature)
```

```
     Sample            Tech         Temp            DryWt       
 Min.   :18.00   Cita    : 6   Min.   : 4.50   Min.   :0.5280  
```

```
 1st Qu.:33.50   Fatima  : 7   1st Qu.: 7.00   1st Qu.:0.6262
 Median :48.00   LaVerna : 3   Median :11.50   Median :0.7090
 Mean   :47.95   Mark    : 9   Mean   :10.81   Mean   :0.7086
 3rd Qu.:62.50   Nitnoy  : 9   3rd Qu.:14.25   3rd Qu.:0.7857
 Max.   :77.00   Raisa   :17   Max.   :19.00   Max.   :0.8590
                 Stephano: 8                   NA's   :1
```

You can see this could be hard to use effectively for a dataset that contains many variables.

The `summary` function can also be used on single columns of a dataset. Below we will summarize just the `Tech` variable.

```r
summary(temperature$Tech)
```

```
    Cita   Fatima  LaVerna     Mark   Nitnoy    Raisa Stephano 
       6        7        3        9        9       17        8 
```

Reading in comma-delimited files

Now that we've successfully read the temperature dataset into R, we'll move on to reading in the respiration data. The respiration data are stored in two different files, one with information from sampling in the spring (`spring resp.csv`) and one from fall sampling (`fall resp.xlsx`).

Let's read the *spring* dataset in first. This is a comma-delimited file, so column information is separated by commas. We could either use `read.table` again and define the variable separator as a comma with the `sep` argument, or use the convenience function `read.csv`. We'll do the latter.

There aren't any missing values in this dataset so we don't have to use the `na.strings` argument. If you look at the help page for `read.csv` (using `?read.csv`) you'll see the default setting for the `header` argument is `TRUE`, so we don't need to specify this in our code because our dataset does contain variable names in the first row.

We'll name the spring respiration dataset `respspring`.

```r
respspring = read.csv("spring resp.csv")
```

Reading in Excel spreadsheets

The fall respiration dataset is in an Excel file ending with `.xlsx`. Excel files can't be read into R with any built-in functions. However, there are many add-on packages that people have written and made available to make reading in data from Excel straightforward. Most file types can be read into R as long as you find and install the correct package.

Installing an add-on package

We'll load an add-on package called **readxl** to read the fall respiration dataset into R. If you've never used this package before on your computer, you will need to install it. You can install packages easily through RStudio's "Packages" pane and the Install button - you just need to know the package name that you want to install. You could also run the code `install.packages("readxl")` to do the same thing.

Packages only need to be installed one time onto a computer, and will be available in future R sessions. 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 a new R session.

```
library(readxl)
```

The warning message indicates that there is a newer version of R available - it is not an error.

Once a package is loaded, you can look at help pages for any functions the package contains in the usual way. We will be using the `read_excel` function today.

```
?read_excel
```

The main difference in loading Excel documents with `read_excel` compared to what we've done so far is that we'll need to tell R which worksheet we want to to read in. We can do this by giving either the index (`1` in this case, as it's the very first sheet) or name (`Sheet1`) of the sheet via the `sheet` argument. It seems easiest in this simple case to use the index. We'll name the new dataset `respfall`.

Notice that, much like `read.csv`, the default setting for the `col_names` argument in `read_excel` is `TRUE`. Therefore we won't need to include the `col_names` argument in our code because the first row of our dataset contains the variable names.

```
respfall = read_excel("fall resp.xlsx", sheet = 1)
```

Editing a variable in a dataset

Now that we have the two respiration datasets, let's check the structure of both of them.

```
str(respspring)
```

```
'data.frame':   30 obs. of  3 variables:
 $ Sample: int  26 23 25 27 24 19 20 22 18 21 ...
 $ Date  : Factor w/ 6 levels "1/1/1987","2/1/1987",..: 2 2 2 2 2 1 1 1 1 1 ...
 $ Resp  : num  0.057 0.085 0.159 0.266 0.368 0.074 0.089 0.117 0.135 0.287 ...
```

```
str(respfall)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   30 obs. of  3 variables:
 $ Sample: num  53 55 54 57 56 51 52 48 49 50 ...
 $ Date  : POSIXct, format: "1987-08-01" "1987-08-01" "1987-08-01" ...
 $ Resp  : num  0.093 0.111 0.143 0.205 0.224 0.058 0.081 0.089 0.106 0.119 ...
```

Hmm, the `Date` column in `respspring` is a **factor** but the same column is a **POSIXct** (aka date-time) variable in `respfall`. This is due to some differences in the `read_excel` function compared to `read.csv`. We are going to want to *stack* these two datasets together into one, but we will have difficulty if the columns with the same names contain very different variable types like this.

Working with dates in R

With that in mind, let's change the `Date` variable to a date in both datasets with the function `as.Date`. While we will not go into this in great detail, it will be good for you to see this. I've seen many people struggle with dates in R when they are first getting started.

```
?as.Date
```

We'll need to tell R what format our date is in with the `format` argument. This means we will tell R the order the months, days, and years are in our dataset as well as what separator is used between them. In `respspring`, our separator is a forward slash (`/`) and the order is month/day/year. Years are four digits.

Notice that I'm replacing the variable in `respspring` with the new variable by simply assigning it to the same name. If I didn't name this variable as I changed it from a factor to a date, these changes would *not* take place.

```
respspring$Date = as.Date(respspring$Date, format = "%m/%d/%Y")
```

We can do the same thing for `respfall`, but the date is in a different format. The separator is a hyphen and the order is year-month-day so we write the `format` differently. Years are still four digits.

```
respfall$Date = as.Date(respfall$Date, format = "%Y-%m-%d")
```

Now look at the structure of the two datasets again. The two datasets now have the same format (`Date`); problem solved.

```
str(respspring)
```

```
'data.frame':   30 obs. of  3 variables:
 $ Sample: int  26 23 25 27 24 19 20 22 18 21 ...
 $ Date  : Date, format: "1987-02-01" "1987-02-01" "1987-02-01" ...
 $ Resp  : num  0.057 0.085 0.159 0.266 0.368 0.074 0.089 0.117 0.135 0.287 ...
```

```
str(respfall)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   30 obs. of  3 variables:
 $ Sample: num  53 55 54 57 56 51 52 48 49 50 ...
 $ Date  : Date, format: "1987-08-01" "1987-08-01" "1987-08-01" ...
 $ Resp  : num  0.093 0.111 0.143 0.205 0.224 0.058 0.081 0.089 0.106 0.119 ...
```

Adding a new variable to a dataset

As I mentioned earlier, we want to combine these two datasets by putting one on top of the other. Let's add a column to each of them to represent `season` prior to combining them. This is easy. We define a new variable name in the dataset and assign whatever values we want to that new variable using dollar sign notation.

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 behavior of repeating a value to fill in all the rows of a dataset is called *recycling*, and can be very efficient. Be careful, though; recycling can also lead to mistakes if you are assigning more than one value to a new variable and the order doesn't match the order of the dataset.

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

```
  Sample       Date  Resp
1     26 1987-02-01 0.057
2     23 1987-02-01 0.085
3     25 1987-02-01 0.159
4     27 1987-02-01 0.266
5     24 1987-02-01 0.368
6     19 1987-01-01 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 1987-02-01 0.057 spring
2      23 1987-02-01 0.085 spring
3      25 1987-02-01 0.159 spring
4      27 1987-02-01 0.266 spring
5      24 1987-02-01 0.368 spring
6      19 1987-01-01 0.074 spring
```

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

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

```
# A tibble: 6 x 4
  Sample       Date  Resp season
   <dbl>     <date> <dbl>  <chr>
1      53 1987-08-01 0.093   fall
2      55 1987-08-01 0.111   fall
3      54 1987-08-01 0.143   fall
4      57 1987-08-01 0.205   fall
5      56 1987-08-01 0.224   fall
6      51 1987-07-01 0.058   fall
```

Stacking two datasets with `rbind`

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

```r
?rbind
```

The function `rbind` stacks all the rows in the datasets based on matching names, which you would see if you delved deeply into the "Details" section of the help file. Do our variable names match between datasets?

```r
names(respspring)
```

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

```r
names(respfall)
```

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

Changing the column names

We made our names all the same, which avoids any problems when using `rbind`. What if we hadn't? We can change column names by simply *assigning* new ones. Below we will change the name for the `season` column in `respfall` to `Season` (with a capital "S"). We essentially replace the four original column names with new ones.

```r
names(respfall)
```

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

```r
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 column names out, you can use the *extract* function. In R, brackets (`[`) represent the extract function. We will not be using these much today, but if you start coding in R regularly you will likely start using these more at some point.

```
?"["
```

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

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

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

To change just the fourth column name, we can extract it and assign a new name, effectively replacing only the fourth column name with a new name.

```
names(respfall)[4] = "season" # replace the 4th name
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   Min.   :1987-01-01   Min.   :0.02300   Length:60         
 1st Qu.:32.75   1st Qu.:1987-03-24   1st Qu.:0.07375   Class :character  
 Median :47.50   Median :1987-06-16   Median :0.09550   Mode  :character  
 Mean   :47.50   Mean   :1987-06-16   Mean   :0.12935                     
 3rd Qu.:62.25   3rd Qu.:1987-09-08   3rd Qu.:0.16300                     
 Max.   :77.00   Max.   :1987-12-01   Max.   :0.52300                     
```

Joining 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 *join* these two datasets together. The unique identifier for each sample taken is called `Sample`, and is in both datasets. Having a unique identifier is key to merging/joining in any programming language, including R.

Check your Environment pane, though. The `temperature` dataset has one less row than `respall`. It turns out there is a missing `Sample` in the `temperature` dataset, and we'll need to keep that in mind during the joining process.

Using `merge` for joining

We will be using the `merge` function today, part of base R, to join datasets. There are other joining functions available in different add-on packages. With `merge`, and most other joining functions, you can only join two datasets together at a time. If you have more than two datasets to join you will need to join them iteratively.

```
?merge
```

By default, the `merge` function joins two datasets on all columns that have the same name. We'll start by using this default behavior, as our datasets have only a single shared column name, `Sample`. We'll name the merged dataset `resptemp`, and check out the first six lines and structure of the result to see what it looks like.

```r
resptemp = merge(respall, temperature)
head(resptemp)
```

```
  Sample       Date  Resp season   Tech Temp DryWt
1     18 1987-01-01 0.135 spring   Mark  4.5 0.569
2     19 1987-01-01 0.074 spring Nitnoy  4.5 0.607
3     20 1987-01-01 0.089 spring  Raisa  4.5 0.597
4     22 1987-01-01 0.117 spring Nitnoy  4.5 0.603
5     23 1987-02-01 0.085 spring Fatima  5.5    NA
6     24 1987-02-01 0.368 spring   Cita  5.5 0.565
```

```r
str(resptemp)
```

```
'data.frame':   59 obs. of  7 variables:
 $ Sample: num  18 19 20 22 23 24 25 26 27 28 ...
 $ Date  : Date, format: "1987-01-01" "1987-01-01" "1987-01-01" ...
 $ Resp  : num  0.135 0.074 0.089 0.117 0.085 0.368 0.159 0.057 0.266 0.093 ...
 $ season: chr  "spring" "spring" "spring" "spring" ...
 $ Tech  : Factor w/ 7 levels "Cita","Fatima",..: 4 5 6 5 2 1 2 4 6 1 ...
 $ Temp  : num  4.5 4.5 4.5 4.5 5.5 5.5 5.5 5.5 5.5 5 ...
 $ DryWt : num  0.569 0.607 0.597 0.603 NA 0.565 0.61 0.64 0.62 0.622 ...
```

Defining the columns to merge on

You can choose which column names to merge on using the `by` argument. I tend to do this because it makes my code easier to understand when I run through it again. It can also help me avoid mistakes if I have columns with the same name in the datasets that I don't want to use in the joining.

```r
merge(respall, temperature, by = "Sample")
```

```
   Sample       Date  Resp season     Tech Temp DryWt
1      18 1987-01-01 0.135 spring     Mark  4.5 0.569
2      19 1987-01-01 0.074 spring   Nitnoy  4.5 0.607
3      20 1987-01-01 0.089 spring    Raisa  4.5 0.597
4      22 1987-01-01 0.117 spring   Nitnoy  4.5 0.603
5      23 1987-02-01 0.085 spring   Fatima  5.5    NA
6      24 1987-02-01 0.368 spring     Cita  5.5 0.565
7      25 1987-02-01 0.159 spring   Fatima  5.5 0.610
8      26 1987-02-01 0.057 spring     Mark  5.5 0.640
9      27 1987-02-01 0.266 spring    Raisa  5.5 0.620
10     28 1987-03-01 0.093 spring     Cita  5.0 0.622
11     29 1987-03-01 0.063 spring    Raisa  5.0 0.634
12     30 1987-03-01 0.074 spring Stephano  5.0 0.613
13     31 1987-03-01 0.073 spring Stephano  5.0 0.611
14     32 1987-03-01 0.064 spring    Raisa  5.0 0.626
15     33 1987-04-01 0.176 spring   Nitnoy  8.0 0.528
16     34 1987-04-01 0.105 spring  LaVerna  8.0 0.642
17     35 1987-04-01 0.097 spring Stephano  8.0 0.627
18     36 1987-04-01 0.116 spring    Raisa  8.0 0.574
19     37 1987-04-01 0.185 spring     Cita  8.0 0.619
20     38 1987-05-01 0.178 spring   Nitnoy 14.5 0.739
21     39 1987-05-01 0.302 spring   Fatima 14.5 0.732
22     40 1987-05-01 0.097 spring    Raisa 14.5 0.749
23     41 1987-05-01 0.092 spring    Raisa 14.5 0.694
```

```
24      42 1987-05-01 0.267 spring   Nitnoy 14.5 0.709
25      43 1987-06-01 0.207 spring     Mark 10.5 0.650
26      44 1987-06-01 0.175 spring Stephano 10.5 0.620
27      45 1987-06-01 0.043 spring Stephano 10.5 0.728
28      46 1987-06-01 0.523 spring    Raisa 10.5 0.670
29      47 1987-06-01 0.122 spring     Cita 10.5 0.679
30      48 1987-07-01 0.089   fall     Mark 13.0 0.787
31      49 1987-07-01 0.106   fall    Raisa 13.0 0.793
32      50 1987-07-01 0.119   fall     Mark 13.0 0.727
33      51 1987-07-01 0.058   fall Stephano 13.0 0.785
34      52 1987-07-01 0.081   fall    Raisa 13.0 0.795
35      53 1987-08-01 0.093   fall   Nitnoy 14.0 0.765
36      54 1987-08-01 0.143   fall Stephano 14.0 0.768
37      55 1987-08-01 0.111   fall Stephano 14.0 0.804
38      56 1987-08-01 0.224   fall   Fatima 14.0 0.791
39      57 1987-08-01 0.205   fall   Nitnoy 14.0 0.709
40      58 1987-09-01 0.274   fall   Nitnoy 19.0 0.801
41      59 1987-09-01 0.085   fall     Cita 19.0 0.808
42      60 1987-09-01 0.207   fall   Fatima 19.0 0.779
43      61 1987-09-01 0.080   fall    Raisa 19.0 0.830
44      62 1987-09-01 0.121   fall     Mark 19.0 0.828
45      63 1987-10-01 0.065   fall    Raisa 16.0 0.848
46      64 1987-10-01 0.107   fall  LaVerna 16.0 0.844
47      65 1987-10-01 0.086   fall   Fatima 16.0 0.820
48      66 1987-10-01 0.072   fall     Mark 16.0 0.859
49      67 1987-10-01 0.063   fall     Cita 16.0 0.836
50      68 1987-11-01 0.050   fall    Raisa 11.5 0.770
51      69 1987-11-01 0.114   fall     Mark 11.5 0.781
52      70 1987-11-01 0.080   fall   Fatima 11.5 0.786
53      71 1987-11-01 0.070   fall    Raisa 11.5 0.753
54      72 1987-11-01 0.094   fall     Mark 11.5 0.759
55      73 1987-12-01 0.069   fall  LaVerna  7.0 0.695
56      74 1987-12-01 0.055   fall    Raisa  7.0 0.656
57      75 1987-12-01 0.023   fall    Raisa  7.0 0.701
58      76 1987-12-01 0.052   fall    Raisa  7.0 0.685
59      77 1987-12-01 0.076   fall   Nitnoy  7.0 0.661
```

The above works if the names in the two datasets are the same. What if they are different? Let's check by making a second temperature dataset called `temp2`, and change the name of `Sample` to `Samplenum`. This uses some skills we learned earlier for replacing column names.

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

Now we'll need to define the variable to merge on in the first dataset (called the `x` dataset) and in the second dataset (called the `y` dataset) by using both the `by.x` and `by.y` arguments.

```
merge(respall, temp2, by.x = "Sample", by.y = "Samplenum")
```

```
   Sample       Date  Resp season     Tech Temp DryWt
1      18 1987-01-01 0.135 spring     Mark  4.5 0.569
2      19 1987-01-01 0.074 spring   Nitnoy  4.5 0.607
3      20 1987-01-01 0.089 spring    Raisa  4.5 0.597
4      22 1987-01-01 0.117 spring   Nitnoy  4.5 0.603
5      23 1987-02-01 0.085 spring   Fatima  5.5    NA
6      24 1987-02-01 0.368 spring     Cita  5.5 0.565
```

7	25	1987-02-01	0.159	spring	Fatima	5.5	0.610
8	26	1987-02-01	0.057	spring	Mark	5.5	0.640
9	27	1987-02-01	0.266	spring	Raisa	5.5	0.620
10	28	1987-03-01	0.093	spring	Cita	5.0	0.622
11	29	1987-03-01	0.063	spring	Raisa	5.0	0.634
12	30	1987-03-01	0.074	spring	Stephano	5.0	0.613
13	31	1987-03-01	0.073	spring	Stephano	5.0	0.611
14	32	1987-03-01	0.064	spring	Raisa	5.0	0.626
15	33	1987-04-01	0.176	spring	Nitnoy	8.0	0.528
16	34	1987-04-01	0.105	spring	LaVerna	8.0	0.642
17	35	1987-04-01	0.097	spring	Stephano	8.0	0.627
18	36	1987-04-01	0.116	spring	Raisa	8.0	0.574
19	37	1987-04-01	0.185	spring	Cita	8.0	0.619
20	38	1987-05-01	0.178	spring	Nitnoy	14.5	0.739
21	39	1987-05-01	0.302	spring	Fatima	14.5	0.732
22	40	1987-05-01	0.097	spring	Raisa	14.5	0.749
23	41	1987-05-01	0.092	spring	Raisa	14.5	0.694
24	42	1987-05-01	0.267	spring	Nitnoy	14.5	0.709
25	43	1987-06-01	0.207	spring	Mark	10.5	0.650
26	44	1987-06-01	0.175	spring	Stephano	10.5	0.620
27	45	1987-06-01	0.043	spring	Stephano	10.5	0.728
28	46	1987-06-01	0.523	spring	Raisa	10.5	0.670
29	47	1987-06-01	0.122	spring	Cita	10.5	0.679
30	48	1987-07-01	0.089	fall	Mark	13.0	0.787
31	49	1987-07-01	0.106	fall	Raisa	13.0	0.793
32	50	1987-07-01	0.119	fall	Mark	13.0	0.727
33	51	1987-07-01	0.058	fall	Stephano	13.0	0.785
34	52	1987-07-01	0.081	fall	Raisa	13.0	0.795
35	53	1987-08-01	0.093	fall	Nitnoy	14.0	0.765
36	54	1987-08-01	0.143	fall	Stephano	14.0	0.768
37	55	1987-08-01	0.111	fall	Stephano	14.0	0.804
38	56	1987-08-01	0.224	fall	Fatima	14.0	0.791
39	57	1987-08-01	0.205	fall	Nitnoy	14.0	0.709
40	58	1987-09-01	0.274	fall	Nitnoy	19.0	0.801
41	59	1987-09-01	0.085	fall	Cita	19.0	0.808
42	60	1987-09-01	0.207	fall	Fatima	19.0	0.779
43	61	1987-09-01	0.080	fall	Raisa	19.0	0.830
44	62	1987-09-01	0.121	fall	Mark	19.0	0.828
45	63	1987-10-01	0.065	fall	Raisa	16.0	0.848
46	64	1987-10-01	0.107	fall	LaVerna	16.0	0.844
47	65	1987-10-01	0.086	fall	Fatima	16.0	0.820
48	66	1987-10-01	0.072	fall	Mark	16.0	0.859
49	67	1987-10-01	0.063	fall	Cita	16.0	0.836
50	68	1987-11-01	0.050	fall	Raisa	11.5	0.770
51	69	1987-11-01	0.114	fall	Mark	11.5	0.781
52	70	1987-11-01	0.080	fall	Fatima	11.5	0.786
53	71	1987-11-01	0.070	fall	Raisa	11.5	0.753
54	72	1987-11-01	0.094	fall	Mark	11.5	0.759
55	73	1987-12-01	0.069	fall	LaVerna	7.0	0.695
56	74	1987-12-01	0.055	fall	Raisa	7.0	0.656
57	75	1987-12-01	0.023	fall	Raisa	7.0	0.701
58	76	1987-12-01	0.052	fall	Raisa	7.0	0.685
59	77	1987-12-01	0.076	fall	Nitnoy	7.0	0.661

Missing values while merging

There are only 59 rows in `resptemp`, because `merge` drops any unmatched row by default. To change this, we can set the `all` argument to `TRUE` to leave all rows in regardless of if they have a match in both datasets. Missing values are filled in with `NA`.

```
head(resptemp)  # You can see that sample 21 is missing
```

```
  Sample       Date  Resp season   Tech Temp DryWt
1     18 1987-01-01 0.135 spring   Mark  4.5 0.569
2     19 1987-01-01 0.074 spring Nitnoy  4.5 0.607
3     20 1987-01-01 0.089 spring  Raisa  4.5 0.597
4     22 1987-01-01 0.117 spring Nitnoy  4.5 0.603
5     23 1987-02-01 0.085 spring Fatima  5.5    NA
6     24 1987-02-01 0.368 spring   Cita  5.5 0.565
```

```
resptemp = merge(respall, temperature, by = "Sample", all = TRUE)
head(resptemp)  # Now sample 21 is here, with NA in the Tech, Temp, and DryWt columns
```

```
  Sample       Date  Resp season   Tech Temp DryWt
1     18 1987-01-01 0.135 spring   Mark  4.5 0.569
2     19 1987-01-01 0.074 spring Nitnoy  4.5 0.607
3     20 1987-01-01 0.089 spring  Raisa  4.5 0.597
4     21 1987-01-01 0.287 spring   <NA>   NA    NA
5     22 1987-01-01 0.117 spring Nitnoy  4.5 0.603
6     23 1987-02-01 0.085 spring Fatima  5.5    NA
```

Working with factors in R

Let's spend some time talking more about factors in R. Knowing how to work with factors in R becomes important when we want to make graphs using categorical variables or we want to fit a linear model with factors (like ANOVA) and would like to control what the output looks like.

At the moment, the `season` variable is a *character* variable instead of a factor. We can see this when we print it in the Console because the categories are not listed as *levels*. We can also see it if we look at the structure of the dataset in the Environment pane.

```
resptemp$season
```

```
 [1] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[11] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[21] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[31] "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"
[41] "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"
[51] "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"   "fall"
```

We can turn this into a factor via the `factor` function. This is particularly useful when you, say, have stored a categorical variable as an integer (e.g., 1, 2, 3) and R doesn't know you meant it to be a categorical.

```
factor(resptemp$season)
```

```
 [1] spring spring spring spring spring spring spring spring spring spring spring spring spring
[14] spring spring spring spring spring spring spring spring spring spring spring spring spring
[27] spring spring spring spring fall   fall   fall   fall   fall   fall   fall   fall   fall
[40] fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall
[53] fall   fall   fall   fall   fall   fall   fall   fall
Levels: fall spring
```

We just made `season` a factor and printed it to the Console. But have we changed the dataset?

```
str(resptemp)
```

```
'data.frame':	60 obs. of  7 variables:
 $ Sample: num  18 19 20 21 22 23 24 25 26 27 ...
 $ Date  : Date, format: "1987-01-01" "1987-01-01" "1987-01-01" ...
 $ Resp  : num  0.135 0.074 0.089 0.287 0.117 0.085 0.368 0.159 0.057 0.266 ...
 $ season: chr  "spring" "spring" "spring" "spring" ...
 $ Tech  : Factor w/ 7 levels "Cita","Fatima",..: 4 5 6 NA 5 2 1 2 4 6 ...
 $ Temp  : num  4.5 4.5 4.5 NA 4.5 5.5 5.5 5.5 5.5 5.5 ...
 $ DryWt : num  0.569 0.607 0.597 NA 0.603 NA 0.565 0.61 0.64 0.62 ...
```

Nope, using `factor` doesn't change anything unless we assign a name. Let's give the new variable the same name, `season`, so the factor variable will replace the original character variable in the dataset.

```
resptemp$season = factor(resptemp$season)
```

Now the variable has been appropriately changed and is in the dataset.

```
str(resptemp)
```

```
'data.frame':	60 obs. of  7 variables:
 $ Sample: num  18 19 20 21 22 23 24 25 26 27 ...
 $ Date  : Date, format: "1987-01-01" "1987-01-01" "1987-01-01" ...
 $ Resp  : num  0.135 0.074 0.089 0.287 0.117 0.085 0.368 0.159 0.057 0.266 ...
 $ season: Factor w/ 2 levels "fall","spring": 2 2 2 2 2 2 2 2 2 2 ...
 $ Tech  : Factor w/ 7 levels "Cita","Fatima",..: 4 5 6 NA 5 2 1 2 4 6 ...
 $ Temp  : num  4.5 4.5 4.5 NA 4.5 5.5 5.5 5.5 5.5 5.5 ...
 $ DryWt : num  0.569 0.607 0.597 NA 0.603 NA 0.565 0.61 0.64 0.62 ...
```

Let's talk more about the levels of a factor. The order of the levels can be important, and changing the order can, for example, change what a graph you are making looks like.

Setting the order of the categories

By default, R sets the factor levels alphanumerically. This means numbers come before letters and "A" comes before "B". We can change the order of the levels with the `levels` argument of `factor`.

```
?factor
```

To set the order of the levels, we list all the categories present in the variable in the order we want them in a vector and put it as the `levels` argument.

```
factor(resptemp$season, levels = c("spring", "fall") )
```

```
 [1] spring spring spring spring spring spring spring spring spring spring spring spring spring
[14] spring spring spring spring spring spring spring spring spring spring spring spring spring
[27] spring spring spring spring fall   fall   fall   fall   fall   fall   fall   fall   fall  
[40] fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall  
[53] fall   fall   fall   fall   fall   fall   fall   fall  
Levels: spring fall
```

Typos matter here, so be careful. Look what happens if we don't write the categories in the `levels` argument exactly as they appear in the dataset (I put a capital "S" on "spring" in the code below). It's definitely important to check what's happening as you go along to avoid these sorts of mistakes.

```
factor(resptemp$season, levels = c("Spring", "fall") )
```

```
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[20] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> fall fall fall fall fall fall fall fall
[39] fall fall fall fall fall fall fall fall fall fall fall fall fall fall fall fall fall fall fall
[58] fall fall fall
Levels: Spring fall
```

Changing the labels of the categories

If we want to make the names of the categories look nicer for graphing, we can change them with the `labels` argument.

```
factor(resptemp$season, levels = c("spring", "fall"),
      labels = c("Spring", "Fall") )
```

```
 [1] Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring
[14] Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring
[27] Spring Spring Spring Spring Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall
[40] Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall
[53] Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall
Levels: Spring Fall
```

The order of the categories in `labels` must be the same as in `levels` to avoid an error that will drastically change your dataset and lead to mistakes in all the rest of your work. Look at the results when I get the order of the `labels` incorrect. R does what I ask it to do, but now my `spring` and `fall` data are mislabeled.

```
factor(resptemp$season, levels = c("spring", "fall"),
      labels = c("Fall", "Spring") )
```

```
 [1] Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall
[14] Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall   Fall
[27] Fall   Fall   Fall   Fall   Spring Spring Spring Spring Spring Spring Spring Spring Spring
[40] Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring Spring
[53] Spring Spring Spring Spring Spring Spring Spring Spring
Levels: Fall Spring
```

As we've been going along practicing with `factor`, I haven't assigned anything we've been doing to a variable name. Let's assign the reordered factor to `season` before moving on.

```
resptemp$season = factor(resptemp$season, levels = c("spring", "fall"),
                    labels = c("Spring", "Fall") )
```

Creating new variables in a dataset from 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 Fahrenheit from temperature in Celsius and add it to the `resptemp` dataset with the name `tempf`.

The dollar sign notation can start to get tedious once you start adding variables to datasets because of all of the typing. R has several built-in functions to help with this, including `with`, `within`, and `transform`. I'll be showing you the `transform` function today so you can see how it works.

The first argument of `transform` is the dataset you want to use variables from. Defining the dataset means in the rest of the calculation we can use the column names from this dataset directly without the dollar sign. We assign names to variables as we create them, also without dollar sign notation. A nice thing about `transform` is that multiple new variables can be created at one time (which we won't see today).

When using `transform`, I generally name the *transformed* dataset (the one with the new column in it) the same as the original dataset. If I don't assign a name to the transformed dataset, the original dataset will not be changed.

```
resptemp = transform(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     18 1987-01-01 0.135 Spring   Mark  4.5 0.569  40.1
2     19 1987-01-01 0.074 Spring Nitnoy  4.5 0.607  40.1
3     20 1987-01-01 0.089 Spring  Raisa  4.5 0.597  40.1
4     21 1987-01-01 0.287 Spring   <NA>   NA    NA    NA
5     22 1987-01-01 0.117 Spring Nitnoy  4.5 0.603  40.1
6     23 1987-02-01 0.085 Spring Fatima  5.5    NA  41.9
```

Remember that our question of interest is about mean respiration differences 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.

It looks like this (combined with `transform` to add the new variable to the `resptemp` dataset), using the condition that the value of `Temp` is less than 8 degrees.

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

```
 [1] Cold Cold Cold <NA> Cold Cold Cold Cold Cold Cold Cold Cold Cold Cold Cold Hot  Hot  Hot  Hot 
[20] Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot 
[39] Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Hot  Cold Cold
[58] Cold Cold Cold
Levels: Cold Hot
```

```
str(resptemp)
```

```
'data.frame':   60 obs. of  9 variables:
 $ Sample   : num  18 19 20 21 22 23 24 25 26 27 ...
 $ Date     : Date, format: "1987-01-01" "1987-01-01" "1987-01-01" ...
 $ Resp     : num  0.135 0.074 0.089 0.287 0.117 0.085 0.368 0.159 0.057 0.266 ...
 $ season   : Factor w/ 2 levels "Spring","Fall": 1 1 1 1 1 1 1 1 1 1 ...
 $ Tech     : Factor w/ 7 levels "Cita","Fatima",..: 4 5 6 NA 5 2 1 2 4 6 ...
 $ Temp     : num  4.5 4.5 4.5 NA 4.5 5.5 5.5 5.5 5.5 5.5 ...
 $ DryWt    : num  0.569 0.607 0.597 NA 0.603 NA 0.565 0.61 0.64 0.62 ...
 $ tempf    : num  40.1 40.1 40.1 NA 40.1 41.9 41.9 41.9 41.9 41.9 ...
 $ tempgroup: Factor w/ 2 levels "Cold","Hot": 1 1 1 NA 1 1 1 1 1 1 ...
```

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`.

```r
summary(resptemp)
```

```
     Sample           Date                 Resp             season         Tech         Temp      
 Min.   :18.00   Min.   :1987-01-01   Min.   :0.02300   Spring:30   Raisa   :17   Min.   : 4.50  
 1st Qu.:32.75   1st Qu.:1987-03-24   1st Qu.:0.07375   Fall  :30   Mark    : 9   1st Qu.: 7.00  
 Median :47.50   Median :1987-06-16   Median :0.09550               Nitnoy  : 9   Median :11.50  
 Mean   :47.50   Mean   :1987-06-16   Mean   :0.12935               Stephano: 8   Mean   :10.81  
 3rd Qu.:62.25   3rd Qu.:1987-09-08   3rd Qu.:0.16300               Fatima  : 7   3rd Qu.:14.25  
 Max.   :77.00   Max.   :1987-12-01   Max.   :0.52300               (Other) : 9   Max.   :19.00  
                                                                    NA's    : 1   NA's   :1      
     DryWt            tempf        tempgroup 
 Min.   :0.5280   Min.   :40.10   Cold:19   
 1st Qu.:0.6262   1st Qu.:44.60   Hot :40   
 Median :0.7090   Median :52.70   NA's: 1   
 Mean   :0.7086   Mean   :51.46             
 3rd Qu.:0.7857   3rd Qu.:57.65             
 Max.   :0.8590   Max.   :66.20             
 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. Look what happens if we take the mean of the variable `DryWt` with the `mean` function. Remember that the `DryWt` variable contains missing values.

```r
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.

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`.

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

```
[1] 0.7086379
```

If we didn't want any rows that contain missing values in our dataset, we could remove them all with `na.omit`. For example, we could make a new dataset called `resptemp2` that has no missing values. You can see it has less rows than `resptemp` when we look in our RStudio Environment pane.

```r
resptemp2 = na.omit(resptemp)
```

It is key to remember that `na.omit` will remove any row with a missing value in it anywhere. This is not going to always be what you want or need. 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 with `write.csv`

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. While we could always recreate it when we save our R code in a script, sometimes it's worth saving a dataset you've created for later use. For practice, let's save the combined dataset `resptemp` as a comma-delimited file called `combined_resp_and_temp_data.csv` using `write.csv`.

If we wanted to save the file somewhere other than our working directory we'd need to write out the path to that directory.

We'll set the `row.names` argument 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 want to spend time exploring the dataset. This usually involves calculating interesting data summaries and creating exploratory graphics to understand the dataset. This gives us a chance to find mistakes and learn what the variables of interest look like as we start thinking about what statistical tool will help us answer our question of interest. We won't be going into great detail on this today, but I will show you how to do some basic data exploration tasks.

Subset a dataset with `subset`

We can make summaries separate by group once we learn how to use the `subset` function.

```
?subset
```

Here is an example, where we print a subset of the dataset into the R Console only when `tempgroup` is `Cold`. We use the `==` to *test for equality*. R will test every row to see if the value of `tempgroup` is `Cold`. If it is, the row will be kept; if not, it will be discarded.

```
subset(resptemp, tempgroup == "Cold")
```

```
   Sample       Date  Resp season     Tech Temp DryWt tempf tempgroup
1      18 1987-01-01 0.135 Spring     Mark  4.5 0.569  40.1      Cold
2      19 1987-01-01 0.074 Spring   Nitnoy  4.5 0.607  40.1      Cold
3      20 1987-01-01 0.089 Spring    Raisa  4.5 0.597  40.1      Cold
5      22 1987-01-01 0.117 Spring   Nitnoy  4.5 0.603  40.1      Cold
6      23 1987-02-01 0.085 Spring   Fatima  5.5    NA  41.9      Cold
7      24 1987-02-01 0.368 Spring     Cita  5.5 0.565  41.9      Cold
8      25 1987-02-01 0.159 Spring   Fatima  5.5 0.610  41.9      Cold
9      26 1987-02-01 0.057 Spring     Mark  5.5 0.640  41.9      Cold
10     27 1987-02-01 0.266 Spring    Raisa  5.5 0.620  41.9      Cold
11     28 1987-03-01 0.093 Spring     Cita  5.0 0.622  41.0      Cold
12     29 1987-03-01 0.063 Spring    Raisa  5.0 0.634  41.0      Cold
13     30 1987-03-01 0.074 Spring Stephano  5.0 0.613  41.0      Cold
14     31 1987-03-01 0.073 Spring Stephano  5.0 0.611  41.0      Cold
15     32 1987-03-01 0.064 Spring    Raisa  5.0 0.626  41.0      Cold
56     73 1987-12-01 0.069   Fall  LaVerna  7.0 0.695  44.6      Cold
57     74 1987-12-01 0.055   Fall    Raisa  7.0 0.656  44.6      Cold
58     75 1987-12-01 0.023   Fall    Raisa  7.0 0.701  44.6      Cold
59     76 1987-12-01 0.052   Fall    Raisa  7.0 0.685  44.6      Cold
60     77 1987-12-01 0.076   Fall   Nitnoy  7.0 0.661  44.6      Cold
```

We can make a subset of only a few columns of a dataset by adding in the `select` argument. Here, we *select* the `Resp` and `tempgroup` columns only.

```
subset(resptemp, tempgroup == "Cold", select = c("Resp", "tempgroup") )
```

```
    Resp tempgroup
```

```
1  0.135      Cold
2  0.074      Cold
3  0.089      Cold
5  0.117      Cold
6  0.085      Cold
7  0.368      Cold
8  0.159      Cold
9  0.057      Cold
10 0.266      Cold
11 0.093      Cold
12 0.063      Cold
13 0.074      Cold
14 0.073      Cold
15 0.064      Cold
56 0.069      Cold
57 0.055      Cold
58 0.023      Cold
59 0.052      Cold
60 0.076      Cold
```

Summary statistics for variables of interest

Now we can use the `summary` function on our data subsets. We're really interested in the `Resp` variable, so we'll get a summary for each group for only this variable.

This is a very basic way to get group summaries. You will want different methods as you start working with more complicated datasets with many groups.

```
summary( subset(resptemp, tempgroup == "Cold", select = "Resp") )
```

```
      Resp
 Min.   :0.0230
 1st Qu.:0.0635
 Median :0.0740
 Mean   :0.1048
 3rd Qu.:0.1050
 Max.   :0.3680
```

```
summary( subset(resptemp, tempgroup == "Hot", select = "Resp") )
```

```
      Resp
 Min.   :0.0430
 1st Qu.:0.0840
 Median :0.1065
 Mean   :0.1371
 3rd Qu.:0.1765
 Max.   :0.5230
```

Exploratory graphics

Much of the data exploration I do is with graphics. Today we'll be using the function `qplot` from package **ggplot2** to make simple exploratory graphics. The "q" in `qplot` stands for "quick", so this is the function that we can use to make quick exploratory plots.

If we wanted to make more polished graphics for publication or presentation, we would want to switch to using the `ggplot` function. Making nice graphics with `ggplot` is a topic for another workshop.

Let's load the **ggplot2** package. You would need to install the package if you didn't already have it installed.

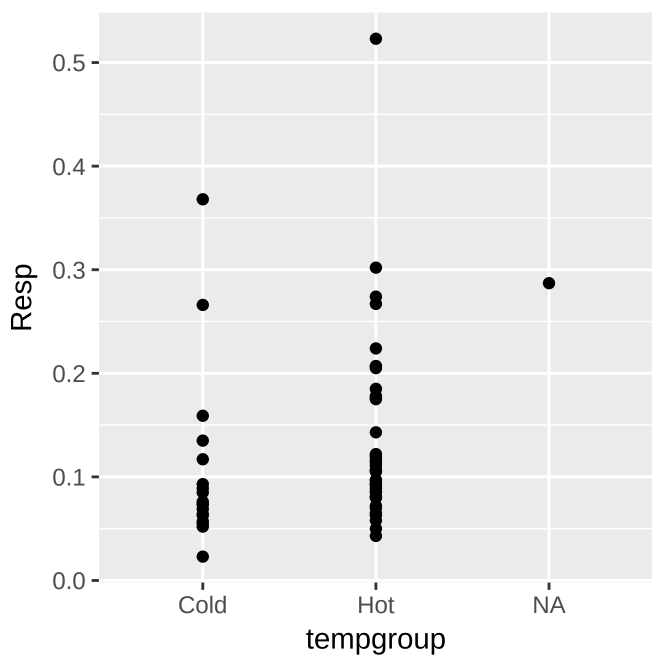
```
library(ggplot2)
```

Scatterplot

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 the variables we want to graph. This makes it so we don't have to (and shouldn't!) use dollar sign notation.

The scatterplot is the default plot type in `qplot`.

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

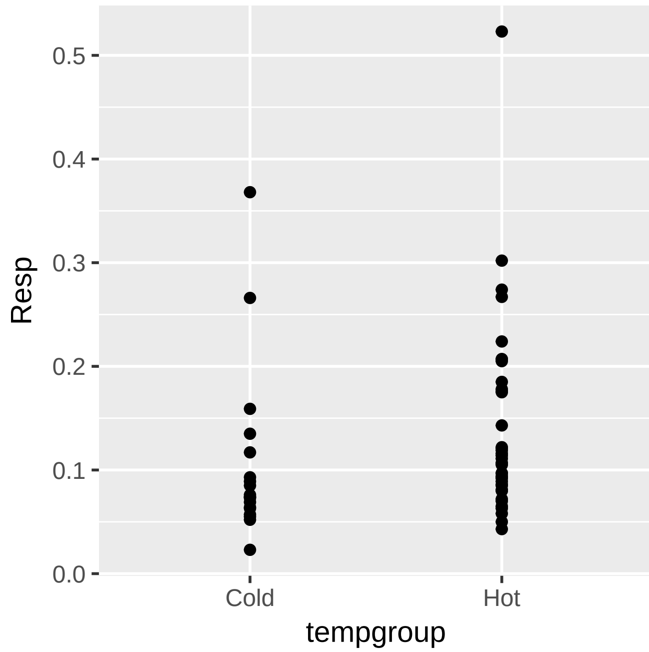


Using `is.na` to remove missing values

Notice we have a missing value. We don't want to remove all the rows in the whole dataset that are missing because we have two rows with missing values but only 1 row that is missing `tempgroup`.

We can remove only the missing value in `tempgroup` using `subset` and `is.na`. We use `is.na` to test if a value is `NA` or not. Because we want to remove all the rows where `tempgroup` is **not** `NA`, we use `is.na` with the not operator `!`.

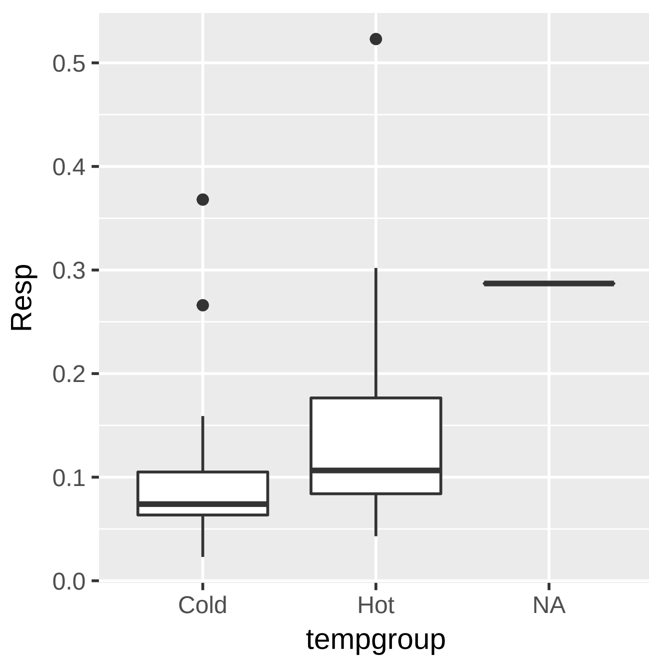
```
qplot(x = tempgroup, y = Resp, data = subset(resptemp, !is.na(tempgroup) ) )
```



Boxplot

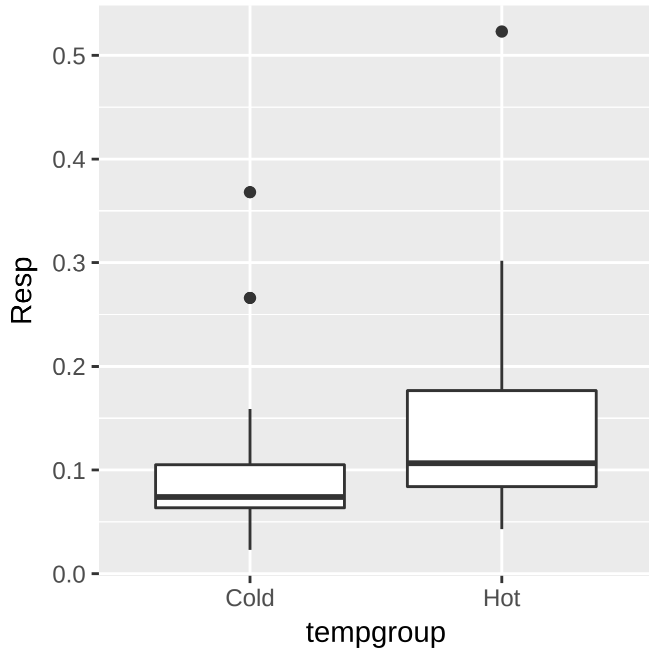
We can make a boxplot instead of a scatterplot by using the `geom` argument to define the plot type. For a boxplot we use `"boxplot"`.

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



And, again, we might want to take out that missing value in `tempgroup`.

```
qplot(x = tempgroup, y = Resp, data = subset(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 was a nuisance to have to remove it for each plot. I name the new dataset `resptemp2`.

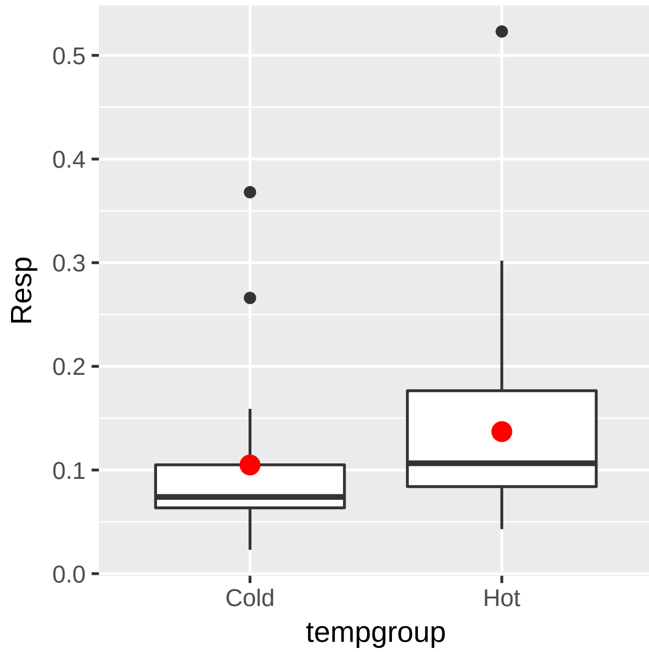
```
resptemp2 = subset(resptemp, !is.na(tempgroup) )
```

Adding the mean to a boxplot

A boxplot shows the median, range, and interquartile range, but not the mean. 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`. Layers are added with plus signs, `+`.

We will not be going through this code in detail. This examples is so you have an example of doing this that you can explore later on your own if you wish.

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

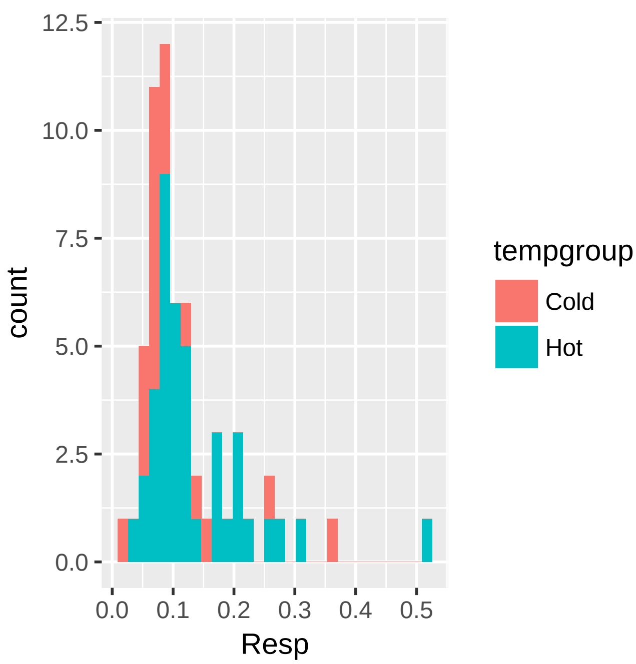


Histogram

Some people like histograms to check their dataset for skew and symmetry. Here we make two histograms, one for each group, by setting the `fill` color of the histograms by `tempgroup`. Notice the variable of interest in a histogram (`Resp` in our case) is on the x axis.

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

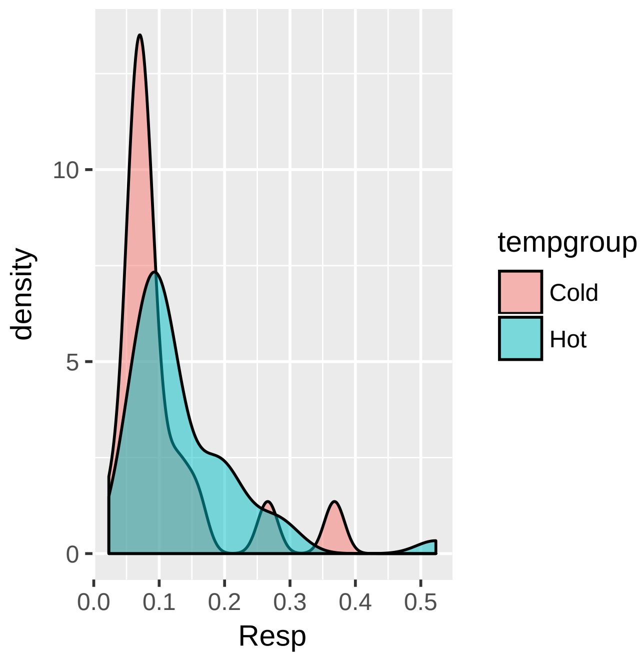
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Density plot

The last example of plotting I give is primarily to show you how changing a factor can change your graphical output. Let's make a quick density plot (a density plot is kind of like a smoothed histogram). We'll use `tempgroup` for the `fill` color again and use `alpha` to make the colors more transparent.

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

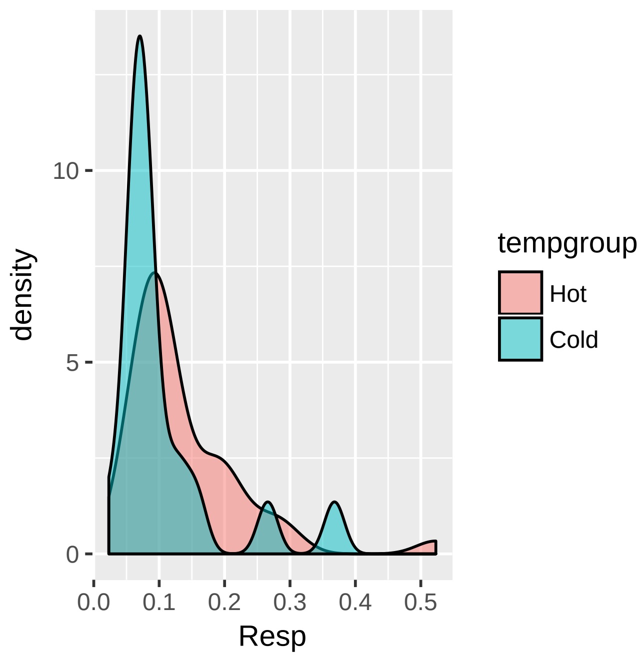


Now let's change the order of the levels of the factor `tempgroup` like we learned how to do earlier, so `Hot` will come before `Cold` instead of vice versa.

```
resptemp2$tempgroup = factor(resptemp2$tempgroup, levels = c("Hot", "Cold") )
```

Look at how both the plot and the legend changed because we changed the order level of the factor variable.

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



Analysis using a two-sample test (finally!)

Let's compare the respiration rate between temperature groups with a two-sample test. Each of us would have to decide if the assumptions are reasonable for a two-sample t-test, a Welch two-sample t-test if the variances are unequal, or possibly a Wilcoxon rank-sum test if there is extreme skewness. In R, there is a built-in function `t.test` for doing t-tests and `wilcox.test` for rank-sum and signed-rank tests.

Here is an example of two different t-tests and a Wilcoxon rank-sum test. I name each test, but also print the results to the Console using an extra pair of parentheses.

I am writing the code 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, and so I avoid using dollar sign notation.

```
( 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 is not equal to 0
95 percent confidence interval:
 -0.01570194  0.08011773
sample estimates:
 mean in group Hot mean in group Cold
         0.1370500          0.1048421
```

Notice the unequal variances t-test is the default, so if the variances seem reasonably 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 is not equal to 0
95 percent confidence interval:
 -0.01665544  0.08107123
sample estimates:
 mean in group Hot mean in group Cold
         0.1370500          0.1048421
```

The `wilcox.test` function is similar, but does the nonparametric Wilcoxon rank-sum test instead of a t-test. In this case we get a useful warning (which is not an error). See the `exact` argument in the documentation to learn more about this warning.

```
( respwilcox = wilcox.test(Resp ~ tempgroup, data = resptemp2) )
```

```
Warning in wilcox.test.default(x = c(0.176, 0.105, 0.097, 0.116, 0.185, : cannot compute exact p-
value with ties


    Wilcoxon rank sum test with continuity correction

data:  Resp by tempgroup
W = 522, p-value = 0.02169
alternative hypothesis: true location shift is not equal to 0
```

To be clear, you wouldn't do all of these tests. Instead, you would have chosen one based on how well you'd met any assumptions. I show all three to give you a couple of additional examples of working with functions in R.

With the analysis finished, our R work is done. In a real analysis, we would spend time making a final graphic or table of results. That is beyond the scope of this workshop, however, so we'll end here knowing there is more to learn in R but with a good start on the basics.