

Contents

R basics: a practical introduction to R	2
Overall analysis goal	2
Before you start the workshop	3
Getting started with RStudio	3
Console pane	4
Source pane	5
Global options	5
Change workspace options	7
R help	9
R documentation for a specific function	9
Stack Overflow	11
RStudio Community	12
The working directory	12
Checking your current working directory	12
Setting your working directory in RStudio	12
Code to set your working directory	13
Reading data into R	13
Reading in a text file	13
Initial exploration of a dataset	17
Reading in comma-delimited files	20
Reading in Excel spreadsheets	20
Installing an add-on package	20
Loading an add-on package	21
Editing a variable in a dataset	23
Working with dates in R	24
Adding a new variable to a dataset	26
Stacking two datasets with <code>rbind()</code>	27
Changing the column names	27
Joining two datasets	28
Finding values in one dataset that are not in another	29
Joining two datasets with <code>inner_join()</code>	30
Working with factors in R	32
Setting the order of the categories	33
Changing the labels of the categories	33
Creating new variables in a dataset based on existing variables	34

Working with missing values in R	36
The <code>na.rm</code> argument	37
Using <code>na.omit</code> to remove rows with missing values	37
Other functions for working with NA	37
Saving a dataset	37
Graphical data exploration	37
Exploratory graphics	38
Scatterplot	38
Using <code>is.na()</code> to remove missing values	38
Boxplot	39
Adding the mean to a boxplot	41
Histogram	41
Density plot	42
Analysis using a two-sample test (finally!)	43

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 scientific goal today is a pretty standard one: we want to perform a statistical analysis on a set of data in R. As we work towards that goal, we will learn to read datasets into R and then do basic data manipulations and 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 become more important. Once you are in that situation, a 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. We will spend most of today’s workshop reading these three datasets into R and combining and manipulating them in preparation for the analysis.

Before you start the workshop

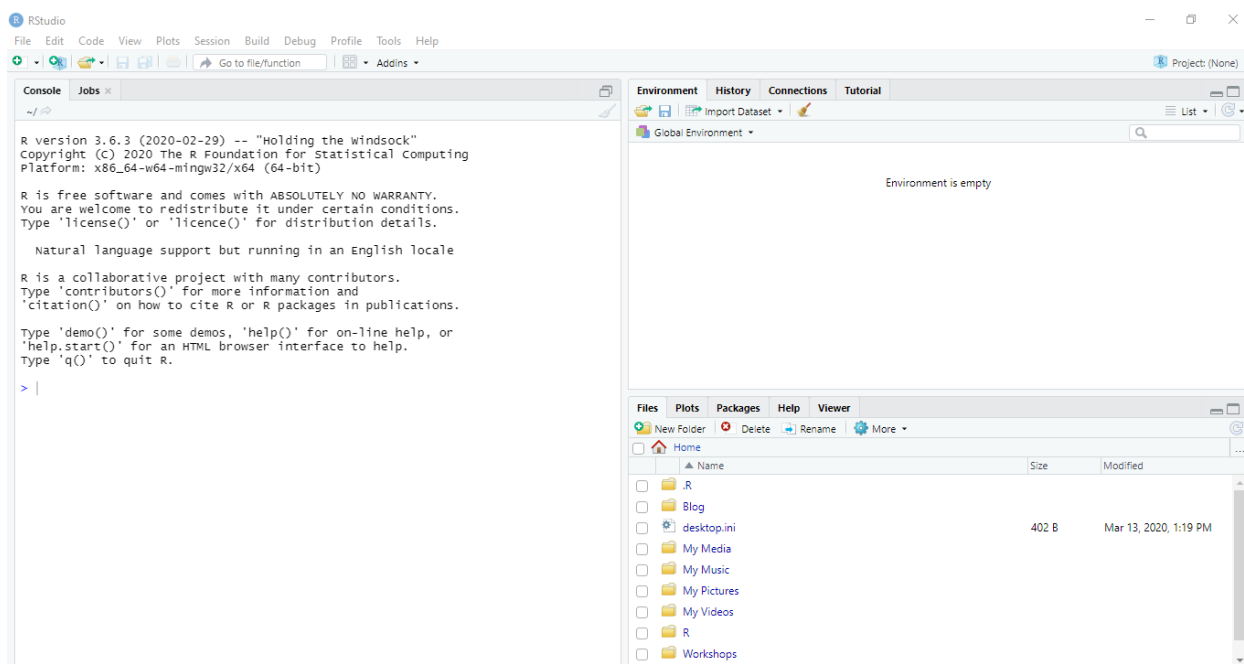
As of April 24, 2020, the current R version is 4.0.0 and the current RStudio version is 1.2.5042. You should download current versions of both these programs and install them on your computer. You can download R from [CRAN](https://cran.r-project.org/) and the free version of RStudio from [their site](https://www.rstudio.com/).

You will also need to download and save the three data files we will work with along with the R script. Save all four of these files into a single folder on your computer. If working through this on your own, you can get all of these files from my website here, <https://ariel.rbind.io/workshop/rbasics/>.

Getting started with RStudio

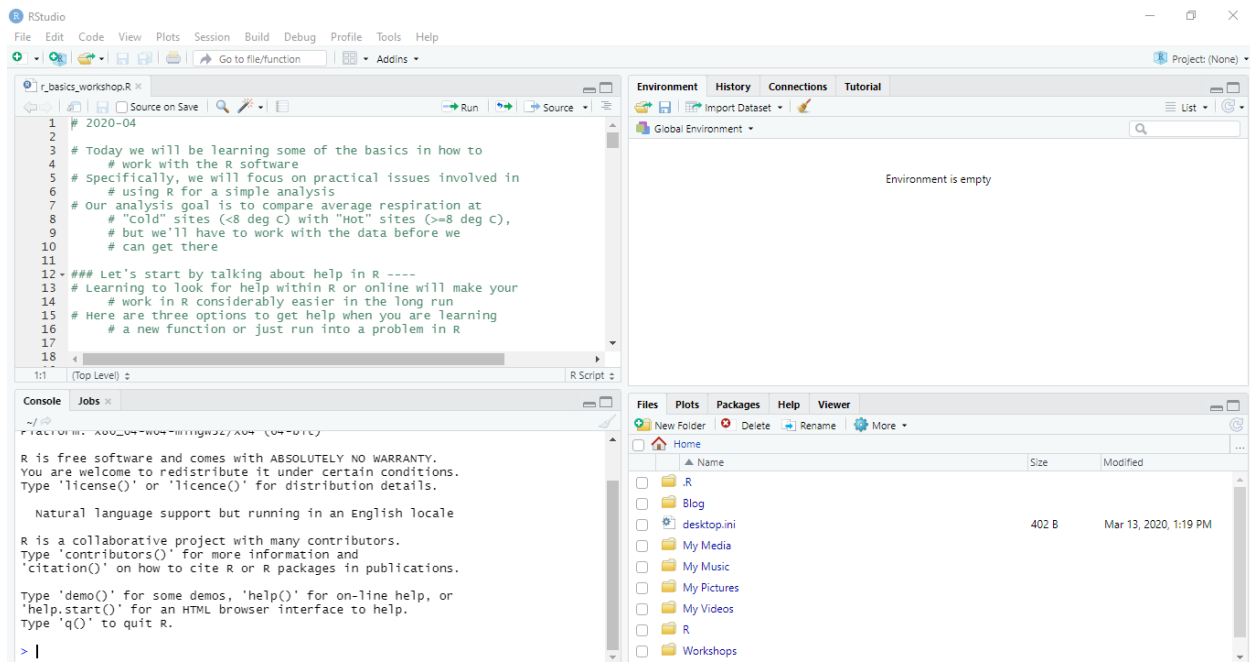
We will be working with R through RStudio. If working along on your own computer, make sure you have RStudio open and ready to go. If you already have a basic familiarity with RStudio, you might want to skip to the [next section on R help](#).

The very first time you open RStudio you will see three of the four RStudio “panes”. The panes are different sections that make up the main RStudio window. You can see the **Console** pane on the left, the **Environment** pane on the upper right, and the **Files** pane (along with **Plots**, **Packages** and **Help**) on the lower right.



When you open an R script I provided, which I named `r_basics_workshop.R` for the example here, you’ll see the fourth pane show up. Open up the R script I provided now to see this. This is the **Source** pane.

You can navigate to open the R script you saved as you normally would on your computer and then open it, selecting RStudio if you need to choose a program to open with. Alternatively, you can navigate through the RStudio **Files** pane, which is the lower right pane in the picture above. To use the latter option you may need the `...` button on the far right, depending on what your default directory is.

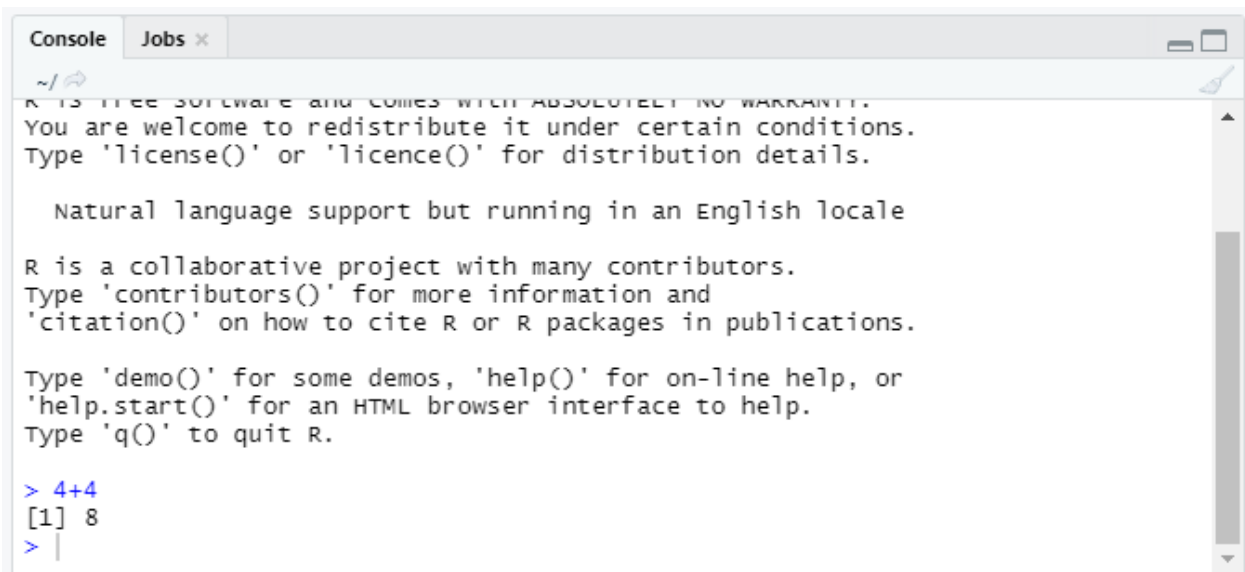


Everything in green in the script in the picture above is a comment. You make a comment in R by writing text after the pound sign, #. This script has extensive comments because it is for other people. However, most scripts will contain comments that you write to explain to your future self exactly what you did.

Console pane

The **Console** pane is on the lower left of the main window by default. This is where you will see R output. You can also type in and run code from here if you don't want to save it.

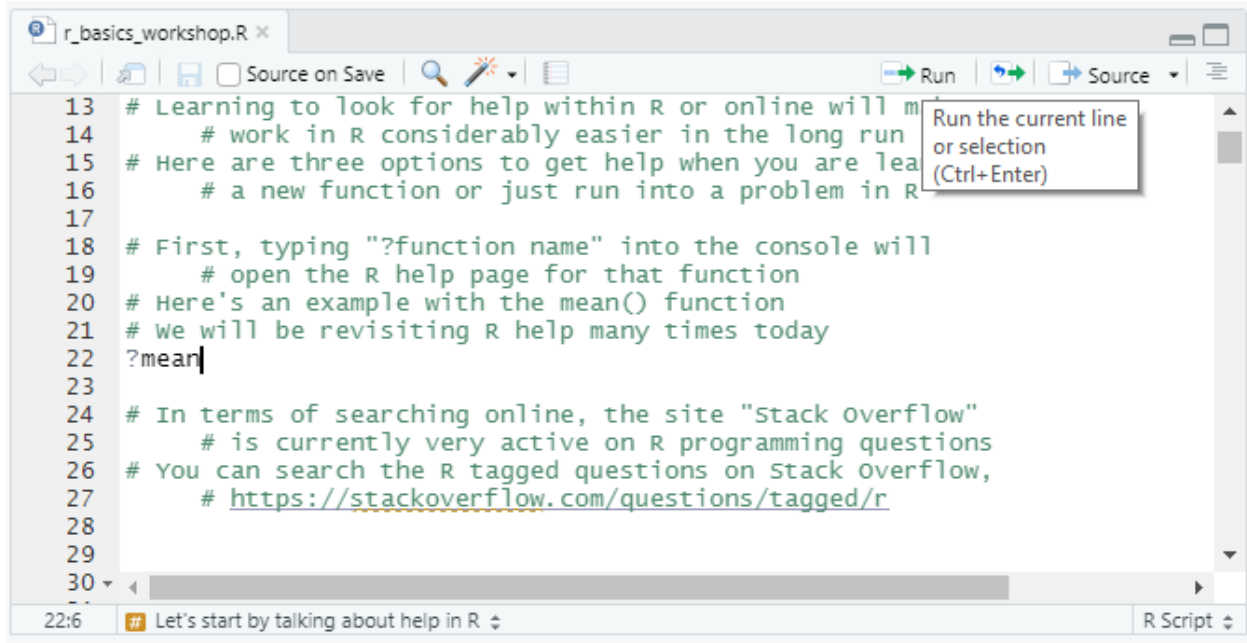
Here you can see I typed in 4+4 next to the > and pushed enter. The output is shown on the next line.



Source pane

The **Source** pane is where your R scripts will open. It is on the upper left above the **Console** by default. Anytime you are working on code you'll want to save (which is most of the time), you'll write the code in an R script saved with `.R`. Today I have written the script with code for us to run together. To run code, put your cursor on the line you want to run and push the **Run** button or use **Ctrl+Enter** (MacOS **Cmd+Enter**).

In the screenshot below I'm getting ready to run line 22 of the script.



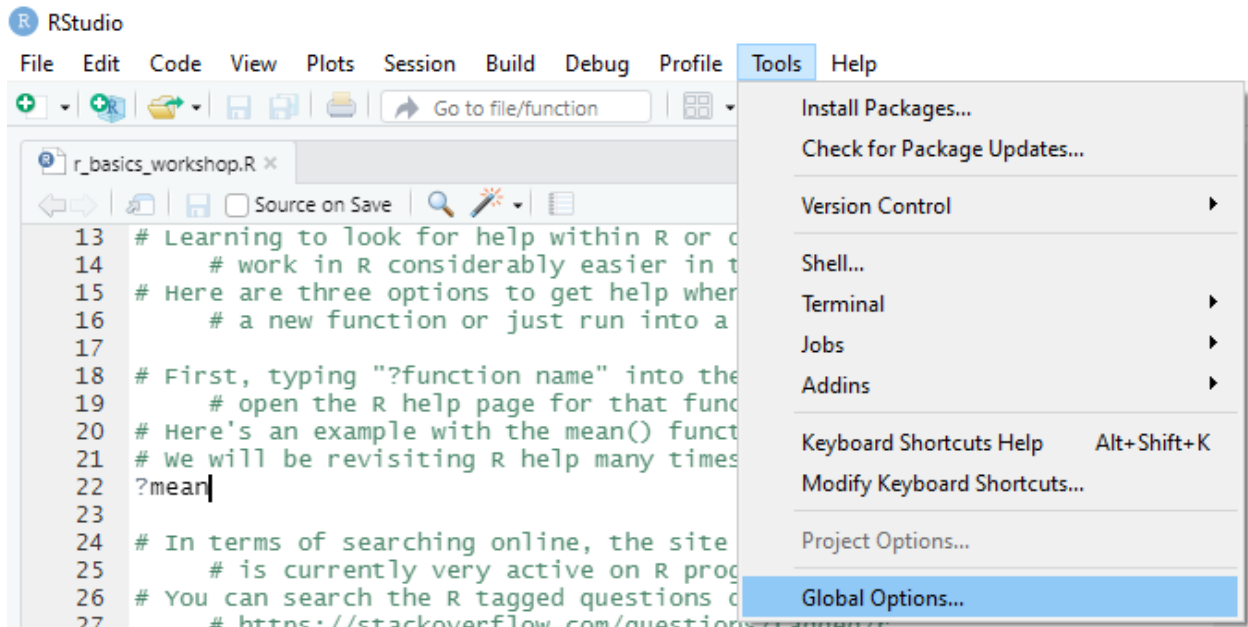
We'll talk about some features of other RStudio panes later in the workshop. Note you can minimize or maximize panes using the squares in the top right corner of each pane.



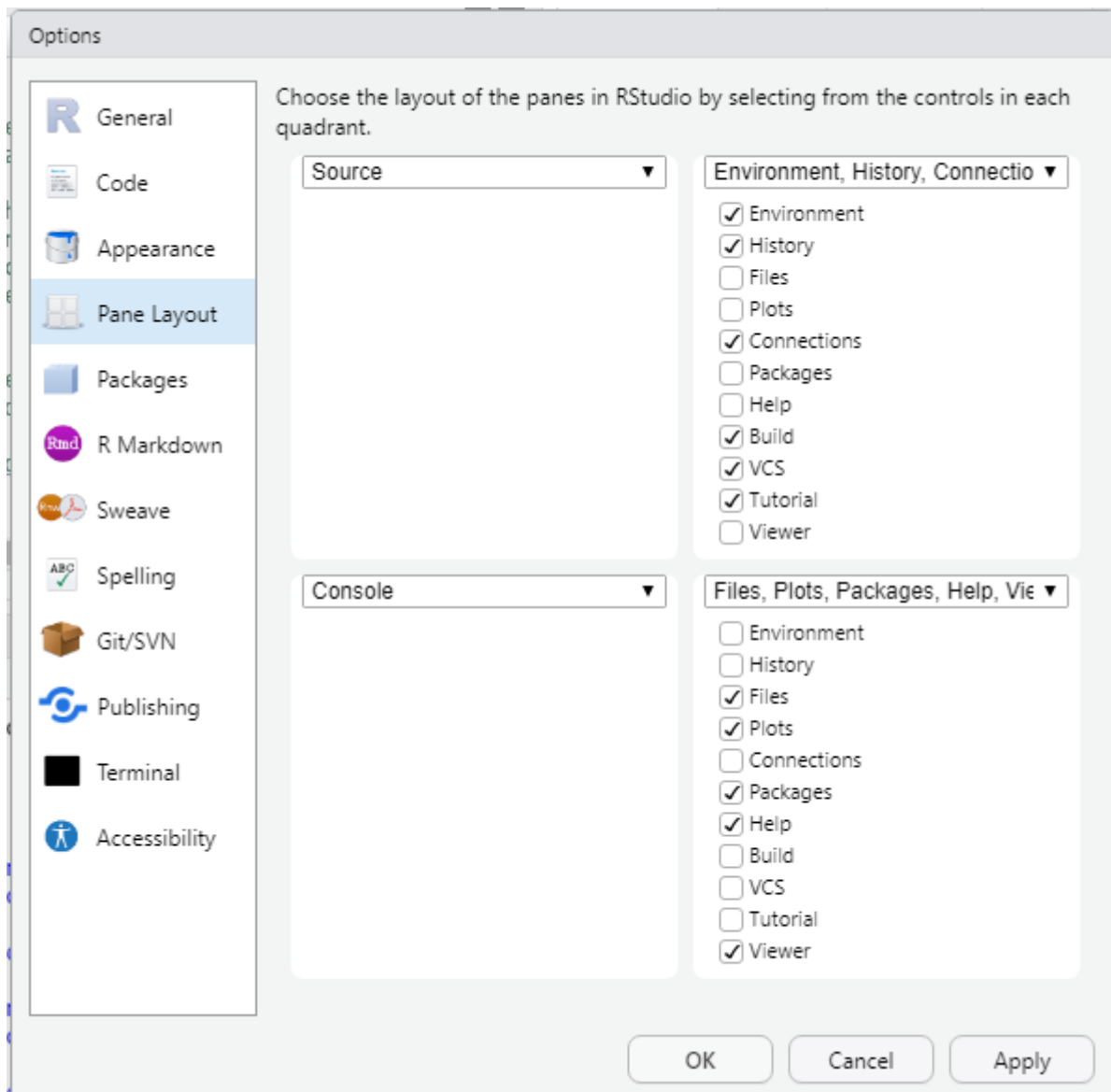
You can also make them bigger or smaller by hovering between two panes and then clicking and dragging.

Global options

My impression is that many people start out using the default settings in RStudio. That's an absolutely fine thing to do. However, there are many options you can set to personalize your RStudio version. For example, I like to have my **Console** and **Source** panes side by side rather than stacked. You can change this (and much more) in **Global Options...** from the **Tools** menu.



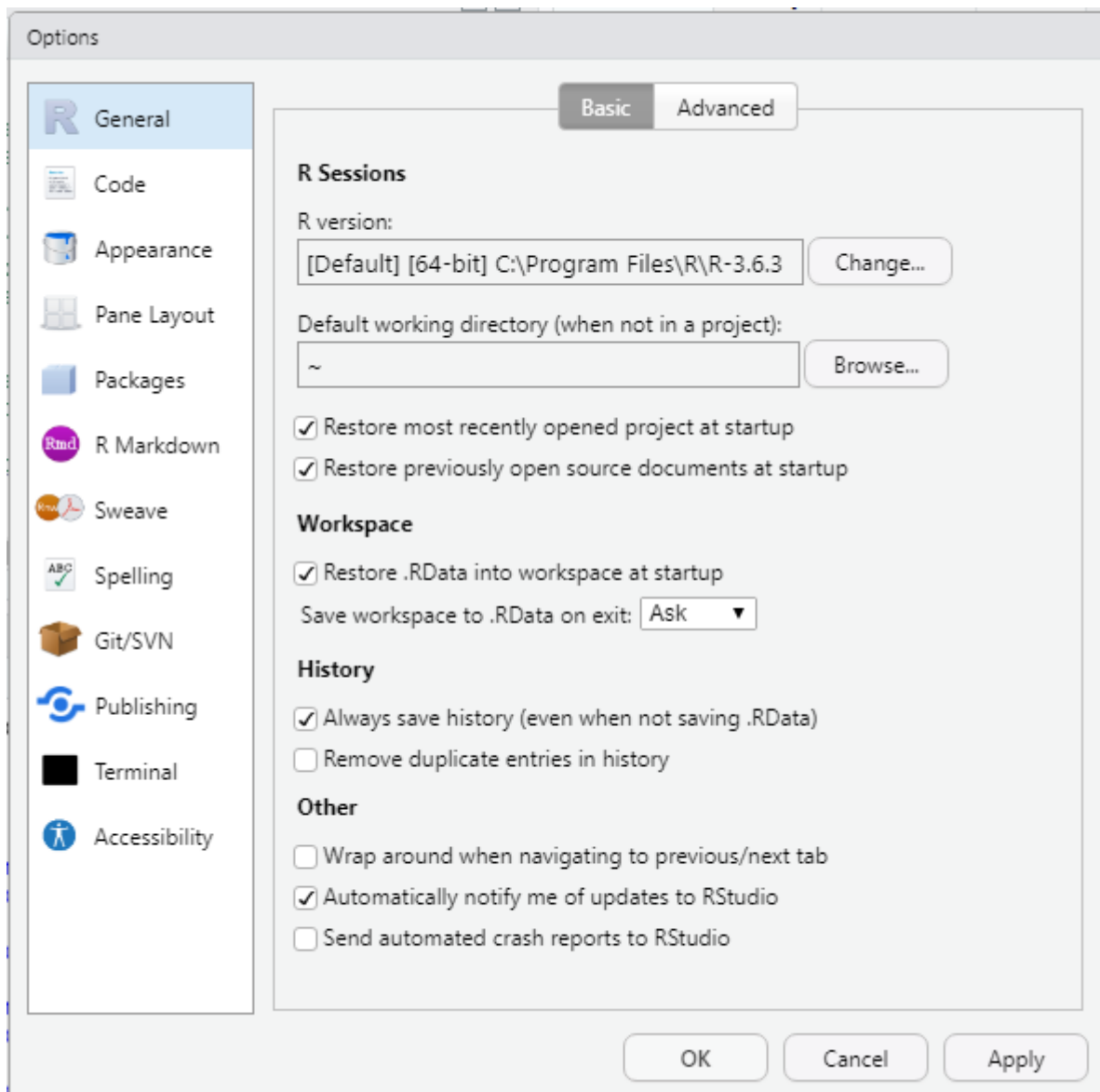
The fourth option down is where you can control the panes layout, shown in the picture below with the defaults. If you wanted you could change the pane positions. You should feel free to explore more of the global options, such as **Appearance**, and see what you like.



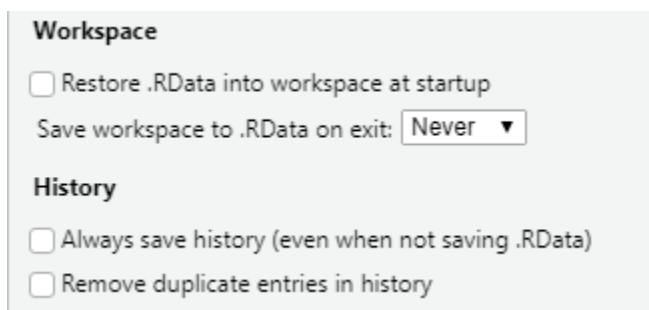
Change workspace options

Before going on to the next section, let's talk for a minute about the global options involving **Workspace** and **History**. I believe it is best practice to start in a clean version of R every time you open a fresh session in RStudio. However, by default the work you did last time is saved and then loaded. If you are new to R this is a good time to change away from the default.

In **Global Options...**, the **General** option will look something like this when you open it with all the default settings. In particular note the current state of the check boxes under the titles **Workspace** and **History**.



If loading your workspace when you start R is an important part of your current R workflow, don't follow these next steps. I recommend you unclick the boxes in **Workspace** and **History** and choose **Never** from the drop-down menu. Those changes will look like:



Click Apply and OK to save these as your global options.

R help

One important thing to talk about as you are first learning R is how to get R help. I believe learning about getting help is the most useful thing I can teach you.

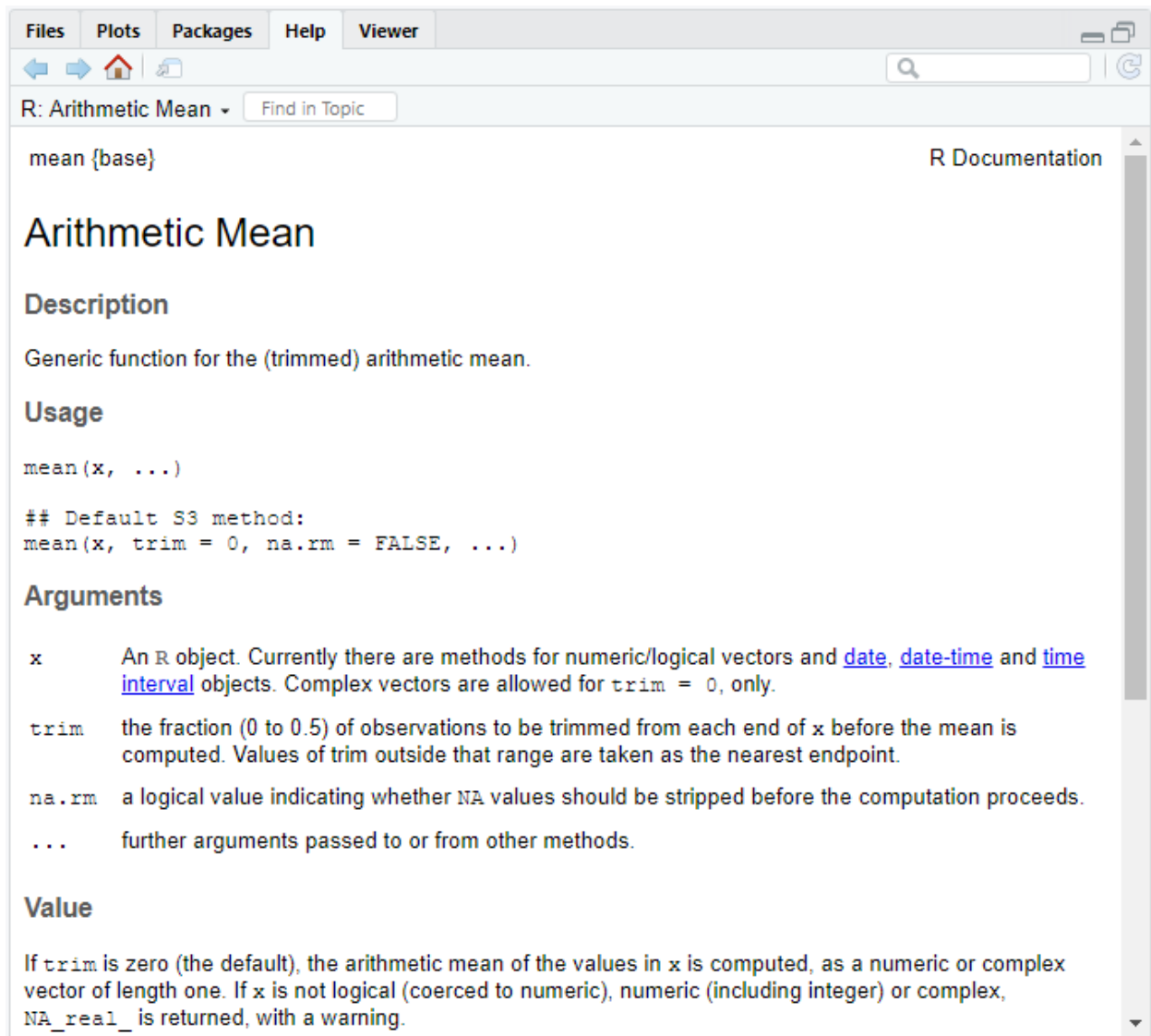
There are three main places I look for help when I run into trouble in R, which I'll go through below.

R documentation for a specific function

The first place to look for help is within R, in the R help page 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 help page for that function. You can do this by typing `?functionname` into your Console and pressing enter, where `functionname` is some R function you want to use. 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 pressing Enter. The help page for the function opens in the **Help** pane on the lower right of your RStudio window.

Here's what the top of the help page for `mean()` looks like.



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)
```

Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments

x An R object. Currently there are methods for numeric/logical vectors and [date](#), [date-time](#) and [time interval](#) objects. Complex vectors are allowed for `trim = 0`, only.

trim the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of `trim` outside that range are taken as the nearest endpoint.

na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.

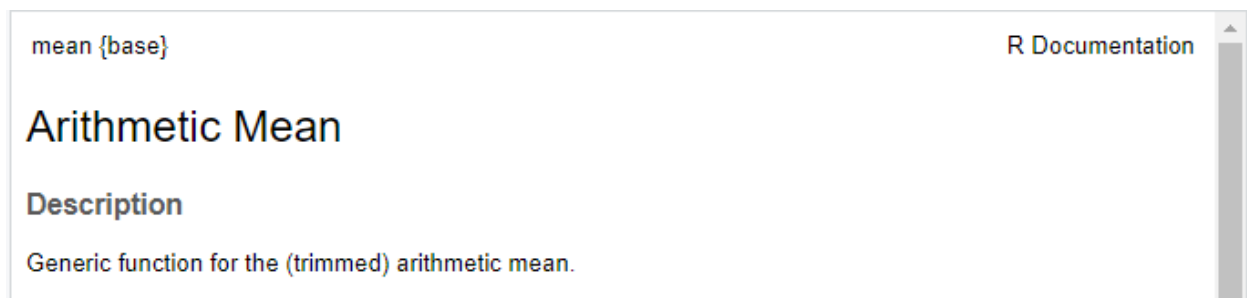
... further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

Every help page has the same basic set-up.

At the top you'll see the function name, followed by the package the function is in surrounded by curly braces. This is followed by a basic description of the function.



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

I often spend time in the **Usage** section, since this is where defaults to argument are given. The function *arguments* are the labels shown within the function. You might note here that the `na.rm` argument in `mean()` defaults to `FALSE`, which we'll talk more about later in the workshop.

Usage

```
mean(x, ...)  
  
## Default S3 method:  
mean(x, trim = 0, na.rm = FALSE, ...)
```

The arguments the function takes and a description of those arguments is given in the **Arguments** section. This is a section I often spend time in, figuring out what arguments do and the options available for each argument.

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date , date-time and time interval objects. Complex vectors are allowed for <code>trim = 0</code> , only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of <code>x</code> before the mean is computed. Values of <code>trim</code> outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

If you scroll to the very bottom of a help page you will find the **Examples** section. This gives examples of how a function works, which you can copy and paste into your **Console** to run the code. You can also highlight the code and run it directly from the help pane with **Ctrl+Enter** (MacOS **Cmd+Enter**).

After looking at **Usage** and **Arguments** I often jump right down to the **Examples** section to see an example using the function. The **Examples** section for `mean()` is pretty sparse, but you'll find that these are quite extensive for some functions.

Examples

```
x <- c(0:10, 50)  
xm <- mean(x)  
c(xm, mean(x, trim = 0.10))
```

There can be a variety of different information after **Arguments** and before **Examples**, such as a description of what is returned, mathematical notation, references in support of what the function does, etc. This can be extremely valuable information, but I often don't read it until I run into trouble using the function or need more information to understand exactly what the function does.

I will also be revisiting the R documentation throughout the workshop to show you how I use it in my daily R work.

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. Stack Overflow is currently very active with R questions and answers, and very often you can find a solution to a problem you are having there based on a question someone else asked.

You can see the R tagged Stack Overflow posts here: <http://stackoverflow.com/questions/tagged/r>

RStudio Community

The RStudio Community at <https://community.rstudio.com/> is another question-answer site that can be useful. It's a little newer and can feel a little friendlier compared to Stack Overflow. This makes it a great place to start if you want to ask your first question about an R problem you are having.

You can (and I often do) search the internet via a search engine in your browser, as well, including “R” or “rstats” as part of the search term.

The working directory

We'll begin 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 files you save. When working on a 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. This also helps me keep organized.

Checking your current working directory

To see your default working directory, use the `getwd()` function to *get* your current working directory. Let's say my default working directory is my “C” drive. This is what shows up when I use `getwd()` to see where the working directory is currently set to.

```
getwd()
```

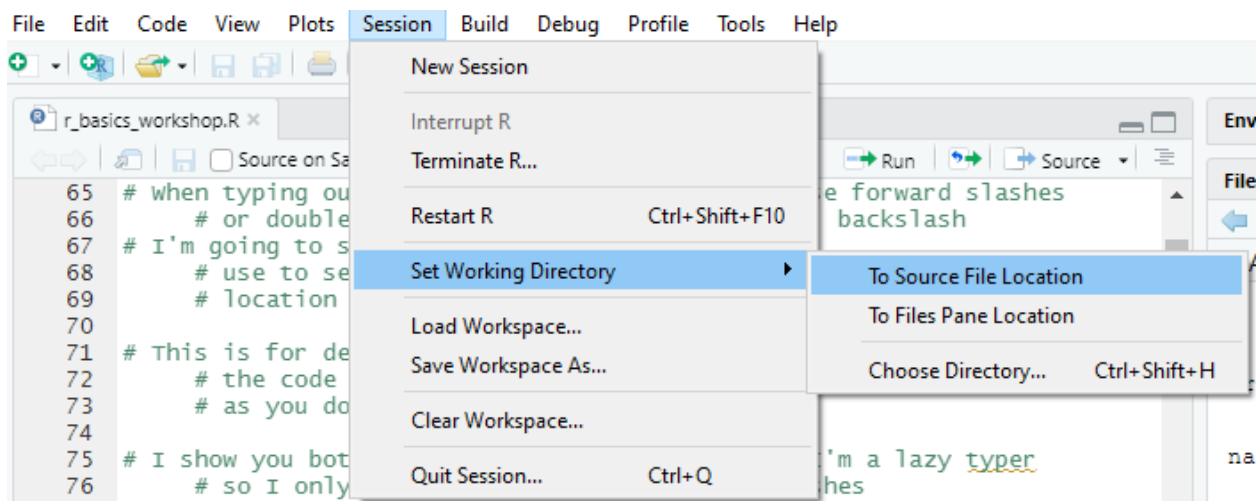
```
[1] "C:/"
```

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.

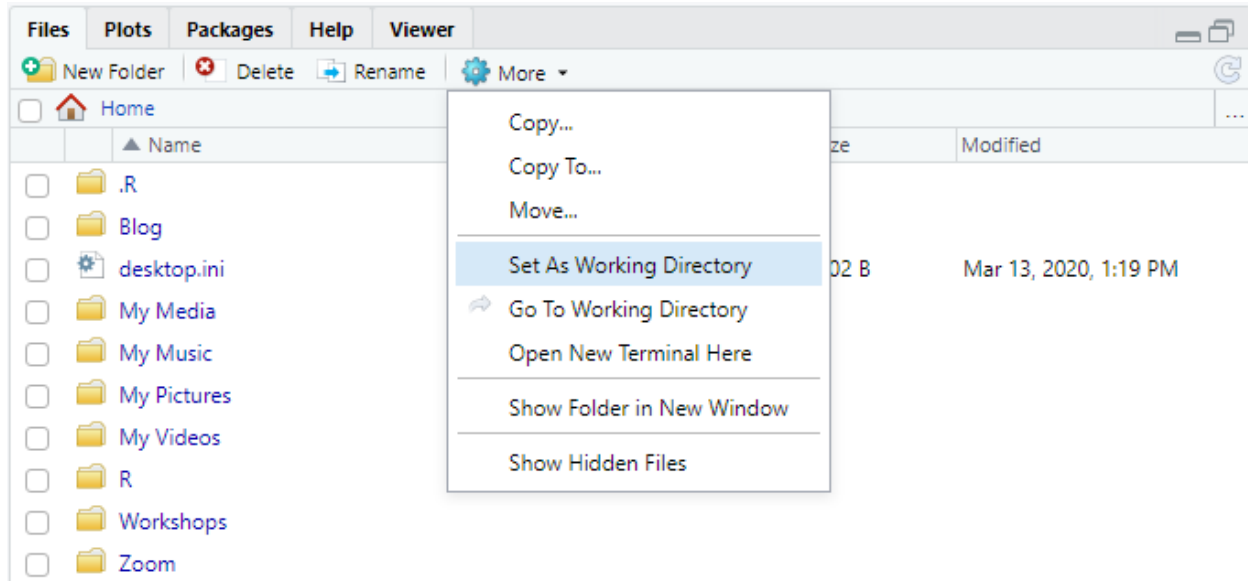
If you've already opened the R script you'll be using in the RStudio **Source** pane, as you have today, you can use the overall drop-down menus to set the directory to wherever your script is stored:

Session > Set Working Directory > To Source File Location



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 to set the working directory to that folder:

More > Set As Working Directory



Code to set your working directory

You can always type out the path to your working directory using the `setwd()` function directly.

Important: You must either use single forward slashes or double backslashes in the directory path in R instead of the single backslashes. If you work in Windows this will not be what you are used to.

Below is an example (not run).

```
setwd("C:/Users/Aosmith/R workshops/r-basics-workshop")
setwd("C:\\Users\\Aosmith\\R workshops\\r-basics-workshop")
```

Once you've set your working directory, you can check if you've successfully made the change using `getwd()` as above. You won't generally need to check this every time you change your working directory, but it is always an option if you are having trouble reading a dataset and need to figure out what your working directory is.

In my case you can see I have a rather long working directory file path.

```
getwd()
```

```
[1] "C:/Users/Owner/Documents/Aosmith/R workshops/R basics/r-basics-workshop"
```

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. These should be saved into the folder you've chosen to be your working directory.

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 the dataset 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
```

In the `read.table()` documentation, take a look at the `header` argument. In the **Usage** section you can see that the default is `header = FALSE`.

The description for `header` in **Arguments** is:

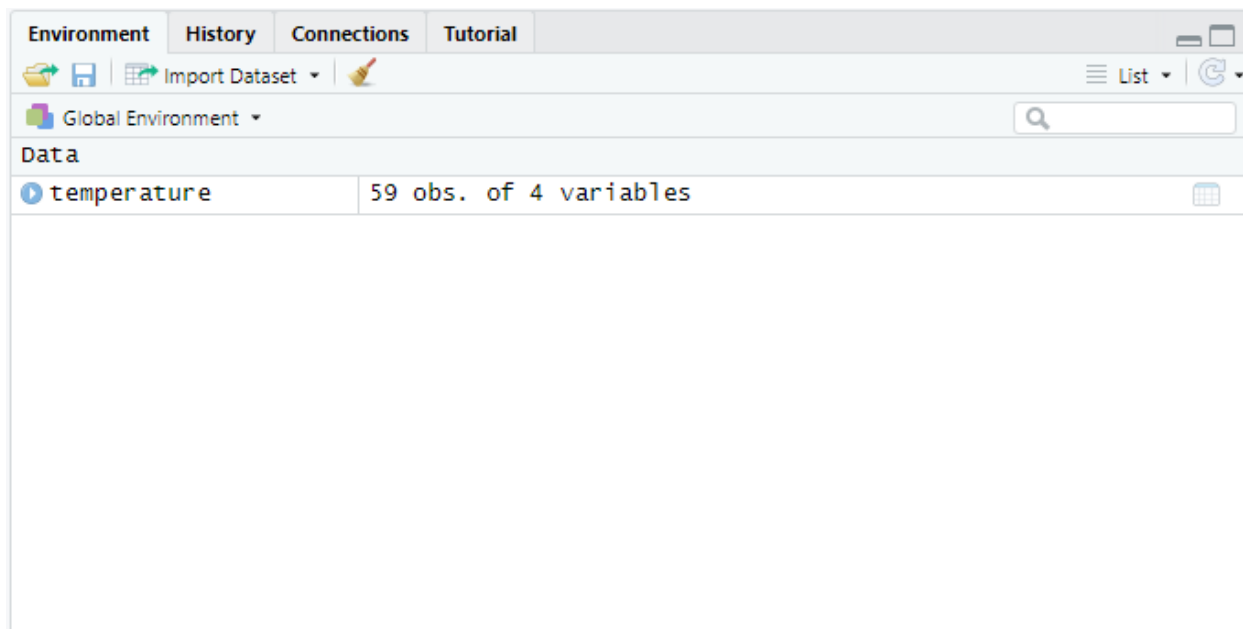
a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains one fewer field than the number of columns.

What this all means is that we will need to tell R that our dataset contains column names via the `header` argument. This is commonly how we would store data in a file, and it means that the very first row of our dataset has all the variable names in it.

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 you can now see an object named `temperature` in the RStudio **Environment** pane in the upper right. This means you 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 current 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("C:/Users/Aosmith/R workshops/r-basics-workshop/temp.txt",
                        header = TRUE)
```

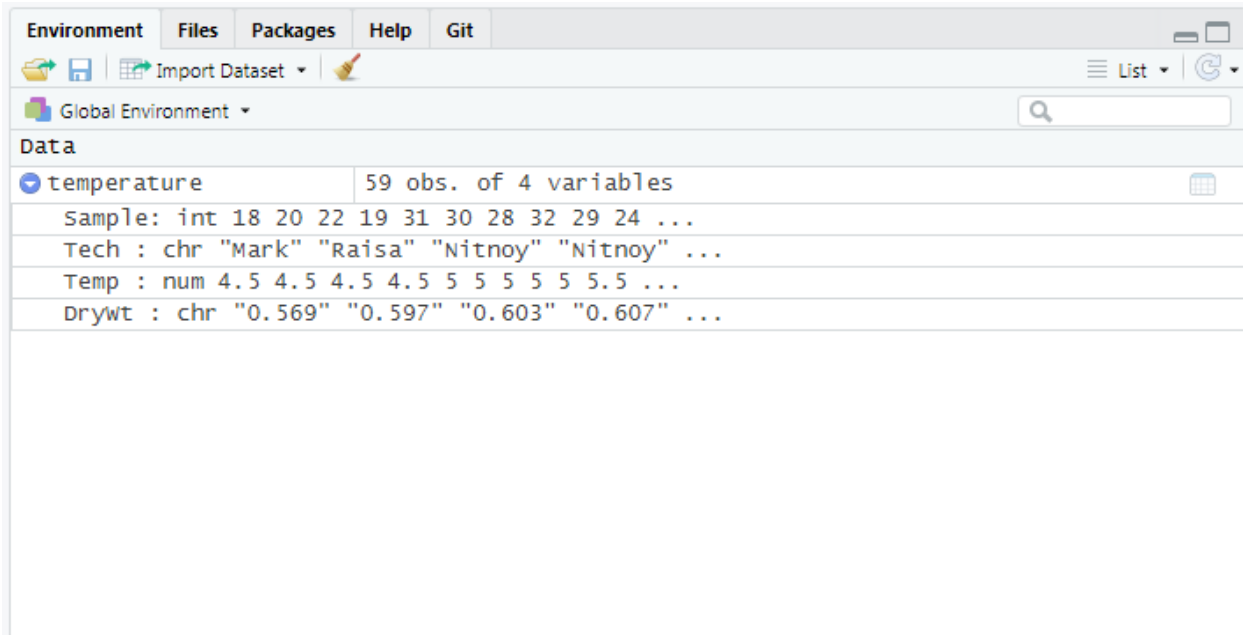
The first thing to do after reading in a dataset is to take a look at it to make sure everything looks the way you expect it to. We can check the basic *structure* of the dataset with the `str()` function. In RStudio, we can click on the arrow next to the object name in the **Environment** pane to see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

Here's what running the `str()` function in the **Console** would look like. You can see the size of the dataset and the names and type of each variable.

```
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" ...
```

And here's clicking the arrow in the **Environment** pane, which shows the same information.



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 character. In pre-4.0.0 versions of R it would be read as a factor. Character and factor variables in R are both types of *categorical* variables.

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 `DryWt` 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"  
[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 categorical variable.

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 the description in the documentation:

a character vector of strings which are to be interpreted as NA values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields. Note that the test happens after white space is stripped from the input, so `na.strings` values may need their own white space stripped in advance.

This indicates that R considers NA or blank fields as missing values. If you use anything else to indicate a missing value you need to tell R what it is via `na.strings`.

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 categorical.

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 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? You can see `DryWt` is now numeric so that problem is fixed.

```
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 : 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 **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 the dataset 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 this RStudio method.

	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

Showing 1 to 11 of 59 entries, 4 total columns

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 check to do 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 as shown earlier, or check the dimensions using `dim()`, `nrow()`, and/or `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 later.

```
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

```

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 documentation for `read.csv()` (using `?read.csv`) you'll see in `Usage` that the default setting for the `header` argument is `TRUE`.

```
read.csv(file, header = TRUE, sep = ",", quote = "\"",
        dec = ".", fill = TRUE, comment.char = "", ...)
```

This means, unlike `read.table()`, we don't need to specify this in our code since our dataset has variable names as the first row.

We'll name the spring respiration dataset `respspring`. We don't need to write out the file path because this dataset is also in our working directory.

```
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 through RStudio's `Packages` pane. Note the `Install` button in the picture below.

Files

Plots

Packages

Help

Viewer

Install

Update

Name	Description	Version		
User Library				
<input type="checkbox"/> askpass	Safe Password Entry for R, Git, and SSH	1.1		
<input type="checkbox"/> assertthat	Easy Pre and Post Assertions	0.2.1		
<input type="checkbox"/> backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.5		
<input type="checkbox"/> base64enc	Tools for base64 encoding	0.1-3		
<input type="checkbox"/> BH	Boost C++ Header Files	1.72.0-1		
<input type="checkbox"/> blogdown	Create Blogs and Websites with R Markdown	0.17		
<input type="checkbox"/> bookdown	Authoring Books and Technical Documents with R Markdown	0.16		
<input type="checkbox"/> callr	Call R from R	3.4.2		
<input type="checkbox"/> cellranger	Translate Spreadsheet Cell Ranges to Rows and Columns	1.1.0		
<input type="checkbox"/> cli	Helpers for Developing Command Line Interfaces	2.0.1		

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**.

Install Packages

Install from:

?

Configuring Repositories

Repository (CRAN)

Packages (separate multiple with space or comma):

readxl

readxl

library:

readxlb

ariel/Documents/R/win-library/3.6 [Default]

☒ Install dependencies

Install

Cancel

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. 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 them in a new R session. Here's a nice picture by [Dianne Cook](#) that explains `install.packages()` vs `library()` well.

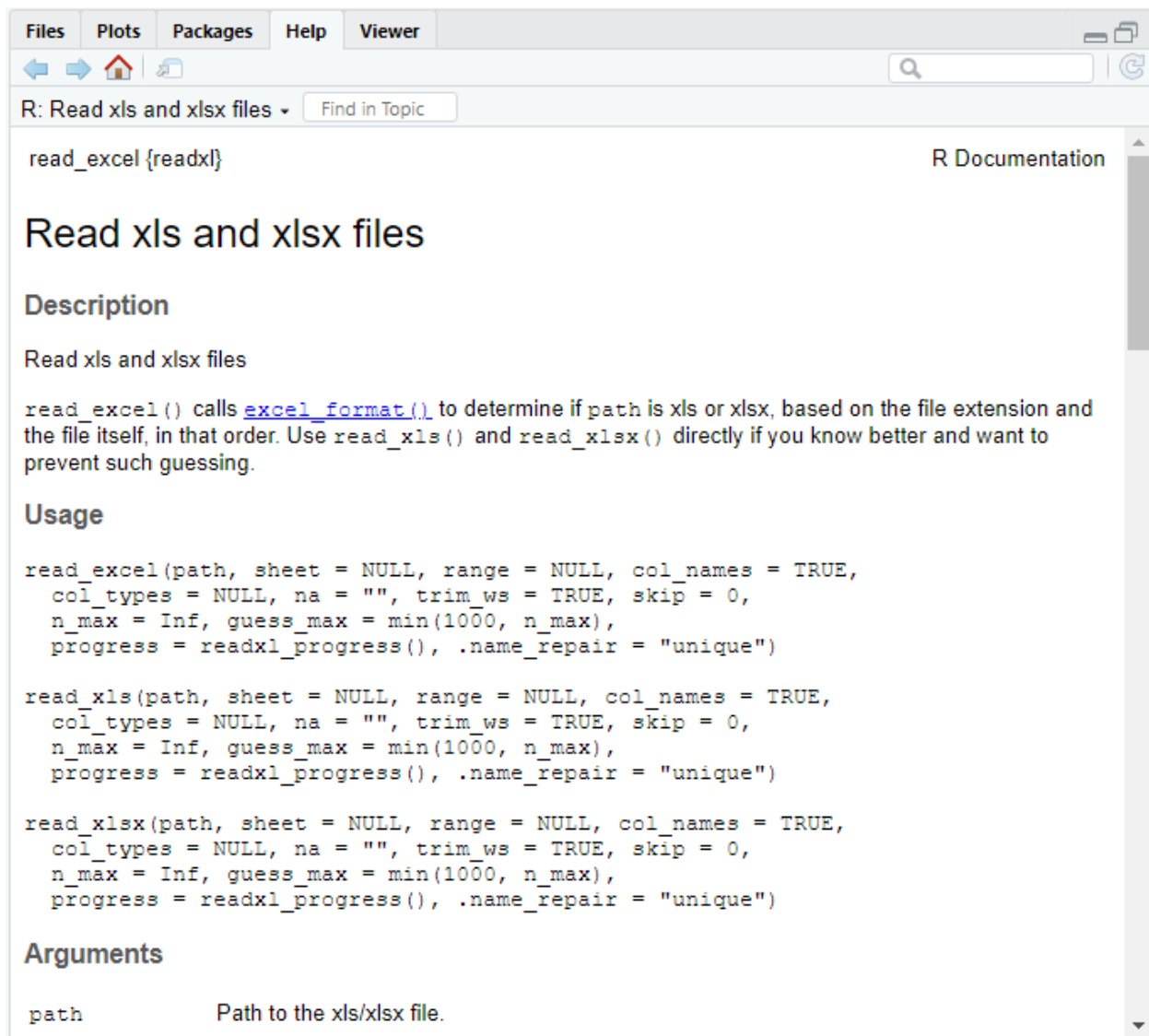


Let's load `readxl`.

```
library(readxl)
```

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
```



A primary 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 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, where there's only one worksheet, to use the index. We'll name the new dataset `respfall`.

Notice that, much like `read.csv()` and `header`, the default setting for the `col_names` argument in `read_excel` is `TRUE` (scroll back up to the documentation screenshot above to see this). We don't need to include the `col_names` argument in the 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  : chr  "2/1/1987" "2/1/1987" "2/1/1987" "2/1/1987" ...
 $ Resp  : num  0.057 0.085 0.159 0.266 0.368 0.074 0.089 0.117 0.135 0.287 ...
```

```
str(respfall)
```

```
tibble [30 x 3] (S3: tbl_df/tbl/data.frame)
 $ Sample: num [1:30] 53 55 54 57 56 51 52 48 49 50 ...
 $ Date  : POSIXct[1:30], format: "1987-08-01" "1987-08-01" "1987-08-01" ...
 $ Resp  : num [1:30] 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 categorical 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 eventual stacking 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, I do think it is good for you to see this. I've seen many people struggle with dates in R when they are first getting started.

```
?as.Date
```


The screenshot shows the R Documentation window for 'as.Date'. The title bar includes 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. The address bar shows 'R: Date Conversion Functions to and from Character'. The main content area is titled 'Date Conversion Functions to and from Character' and includes sections for 'Description', 'Usage', and 'Arguments'.

Description
Functions to convert between character representations and objects of class "Date" representing calendar dates.

Usage

```
as.Date(x, ...)
## S3 method for class 'character'
as.Date(x, format, tryFormats = c("%Y-%m-%d", "%Y/%m/%d"),
        optional = FALSE, ...)
## S3 method for class 'numeric'
as.Date(x, origin, ...)
## S3 method for class 'POSIXct'
as.Date(x, tz = "UTC", ...)

## S3 method for class 'Date'
format(x, ...)

## S3 method for class 'Date'
as.character(x, ...)
```

Arguments

x an object to be converted.

format [character](#) string. If not specified, it will try `tryFormats` one by one on the first non-NA element and give an error if none works. Otherwise, the processing is via `strptime`.

We 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 assigning it to the same name. If I didn't name this variable as I changed it to a date, these changes would *not* take place. Instead the output would show in the `Console` with the original dataset would be unchanged.

```
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)
```

```
tibble [30 x 3] (S3: tbl_df/tbl/data.frame)
 $ Sample: num [1:30] 53 55 54 57 56 51 52 48 49 50 ...
 $ Date  : Date[1:30], format: "1987-08-01" "1987-08-01" "1987-08-01" ...
 $ Resp  : num [1:30] 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 stacking one on top of the other. Let's add a column to each of them to represent **season** prior to combining them. It turns out this isn't too hard. 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**.

```
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()`

The next task is to combine these two datasets into a single dataset using the `rbind()` function. The *r* in `rbind()` stands for *row*.

The function `rbind()` stacks all the rows in the datasets based on matching names. You would only learn that it stacks based on names and not column position if you delved deeply into the **Details** section of the help file at `?rbind`. This is a time where reading the information between the **Arguments** and the **Examples** was useful to me.

Data frame methods

The `cbind` data frame method is just a wrapper for `data.frame(..., check.names = FALSE)`. This means that it will split matrix columns in data frame arguments, and convert character columns to factors unless `stringsAsFactors = FALSE` is specified.

The `rbind` data frame method first drops all zero-column and zero-row arguments. (If that leaves none, it returns the first argument with columns otherwise a zero-column zero-row data frame.) It then takes the classes of the columns from the first data frame, and matches columns by name (rather than by position).

Do our variable names match between datasets?

```
names(respspring)
```

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

```
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.

```
names(respfall)
```

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

```
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. You can get to the documentation using `?[]`.

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 `respsall`. Here I list `respspring` first within the function, but it really doesn't matter.

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

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 `respsall` 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 part of both datasets. The unique identifier is how we match the rows in one dataset to the rows in the other dataset during the joining process.

Check your Environment pane, though. You can see the `temperature` dataset has one less row than `respall` (60 vs 59).

Global Environment	
Data	
▶ respall	60 obs. of 4 variables
▶ respfall	30 obs. of 4 variables
▶ respspring	30 obs. of 4 variables
▶ temperature	59 obs. of 4 variables

Which Sample is missing from `temperature`? Let's check.

Finding values in one dataset that are not in another

If I were working in Excel, I might order the dataset by `Sample` and then scan through until I found a missing value. This isn't very efficient, though, particularly in R. 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`.

If you haven't used package `dplyr` before you may need to install it. You saw how to install packages [earlier in this workshop](#) using drop down menus or `install.packages("dplyr")`. Once it is installed, load the package into R via the `library()` function

```
library(dplyr)
```

If you go to the documentation with `?anti_join` and scroll down we can see the an anti join will:

return all rows from x where there are not matching values in y, keeping just columns from x.

In other words, the `anti_join()` returns rows from the first dataset ("x") listed that are **NOT** in the second dataset listed ("y"). 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 use the `by` argument to define which variable in the two datasets we want to match on. Since `Sample` uniquely identifies where each sample came from, this is our `by` variable.

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

```
Sample      Date Resp season
1      21 1987-01-01 0.287 spring
```

We can see that the `temperature` dataset is missing sample 21. If we didn't know this already we would probably 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	1987-02-01	0.057	spring	Mark	5.5	0.640
2	23	1987-02-01	0.085	spring	Fatima	5.5	NA
3	25	1987-02-01	0.159	spring	Fatima	5.5	0.610
4	27	1987-02-01	0.266	spring	Raisa	5.5	0.620
5	24	1987-02-01	0.368	spring	Cita	5.5	0.565
6	19	1987-01-01	0.074	spring	Nitnoy	4.5	0.607

```
str(resptemp)
```

```
'data.frame':  59 obs. of  7 variables:
 $ Sample: num  26 23 25 27 24 19 20 22 18 29 ...
 $ 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 0.063 ...
 $ 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 = respass, y = temp2, by = c("Sample" = "Samplenum"))
```

	Sample	Date	Resp	season	Tech	Temp	DryWt
1	26	1987-02-01	0.057	spring	Mark	5.5	0.640
2	23	1987-02-01	0.085	spring	Fatima	5.5	NA
3	25	1987-02-01	0.159	spring	Fatima	5.5	0.610
4	27	1987-02-01	0.266	spring	Raisa	5.5	0.620
5	24	1987-02-01	0.368	spring	Cita	5.5	0.565
6	19	1987-01-01	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 `respass` 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 = respass, 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	1987-02-01	0.057	spring	Mark	5.5	0.640
2	23	1987-02-01	0.085	spring	Fatima	5.5	NA
3	25	1987-02-01	0.159	spring	Fatima	5.5	0.610
4	27	1987-02-01	0.266	spring	Raisa	5.5	0.620
5	24	1987-02-01	0.368	spring	Cita	5.5	0.565
6	19	1987-01-01	0.074	spring	Nitnoy	4.5	0.607
7	20	1987-01-01	0.089	spring	Raisa	4.5	0.597
8	22	1987-01-01	0.117	spring	Nitnoy	4.5	0.603
9	18	1987-01-01	0.135	spring	Mark	4.5	0.569
10	21	1987-01-01	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.

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 an ANOVA) and would like to control what the output looks like.

At the moment, the `season` variable is a *character* variable. 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; the variable type is character, `chr`.

```
resptemp$season
```

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

We can turn variables 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 spring
[14] spring 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   fall
[40] fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall   fall
[53] fall   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   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 ...
 $ 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 ...
```

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  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 ...
 $ season: Factor w/ 2 levels "fall","spring": 2 2 2 2 2 2 2 2 2 2 ...
 $ 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 ...
```

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()`. You can see the description of the arguments in the documentation at `?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 fall
[40] fall fall 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>
[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 Spring
[14] Spring 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 Fall
[40] Fall Fall Fall Fall Fall Fall Fall Fall Fall Fall Fall Fall Fall Fall
[53] Fall 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 Fall
[14] Fall 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 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 via `library()` to use `mutate()`. We've already done this.

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	1987-02-01	0.057	Spring	Mark	5.5	0.640	41.9
2	23	1987-02-01	0.085	Spring	Fatima	5.5	NA	41.9
3	25	1987-02-01	0.159	Spring	Fatima	5.5	0.610	41.9
4	27	1987-02-01	0.266	Spring	Raisa	5.5	0.620	41.9
5	24	1987-02-01	0.368	Spring	Cita	5.5	0.565	41.9
6	19	1987-01-01	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
```

Conditional Element Selection

Description

`ifelse` returns a value with the same shape as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

Usage

```
ifelse(test, yes, no)
```

Arguments

`test` an object which can be coerced to logical mode.

`yes` return values for true elements of `test`.

`no` return values for false elements of `test`.

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" "Cold" NA      "Cold" "Cold" "Cold"
[14] "Cold" "Cold" "Hot"  "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 : num 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 ...
 $ season : Factor w/ 2 levels "Spring","Fall": 1 1 1 1 1 1 1 1 1 1 ...
 $ 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	Min. :1987-01-01	Min. :0.02300	Spring:30	Length:60
1st Qu.:32.75	1st Qu.:1987-03-24	1st Qu.:0.07375	Fall :30	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		

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. You can see this argument in the documentation, `?write.csv`.

```
write.csv(x = resptemp, file = "combined_resp_and_temp_data.csv",  
          row.names = FALSE)
```

Graphical 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.

I will only be focusing on graphical data exploration today. However, if you use R regularly you will want to learn how to make summaries for groups and other data manipulation tasks. If you want to learn more about data manipulation in R you can check out, e.g., my data manipulation with **dplyr** workshop materials [here](#).

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 may need to install the package if you don't already have it installed (`install.packages("ggplot2")`).

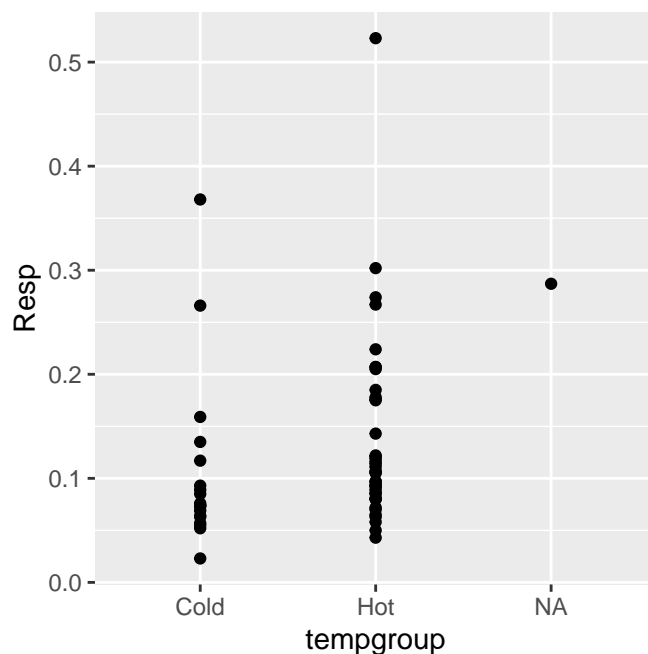
```
library(ggplot2)
```

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)
```

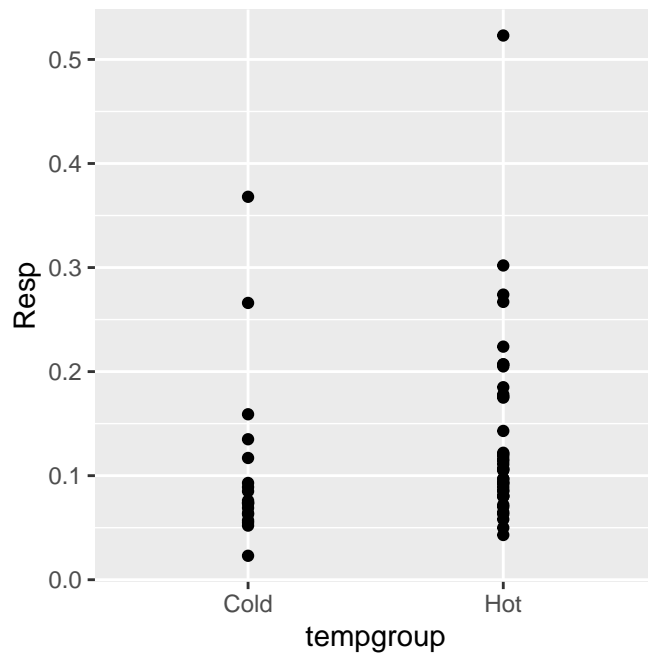


Using `is.na()` to remove missing values

Notice we have a missing value in `tempgroup`, which shows up on the x axis. 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 the `subset()` function with `is.na()`. The `subset()` function is new to you. Basically, we use `is.na()` to test if the `tempgroup` value on each row 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 (`!`). If the value is *not* NA, we keep it. Using `subset()` we can now plot only the existing temperature groups.

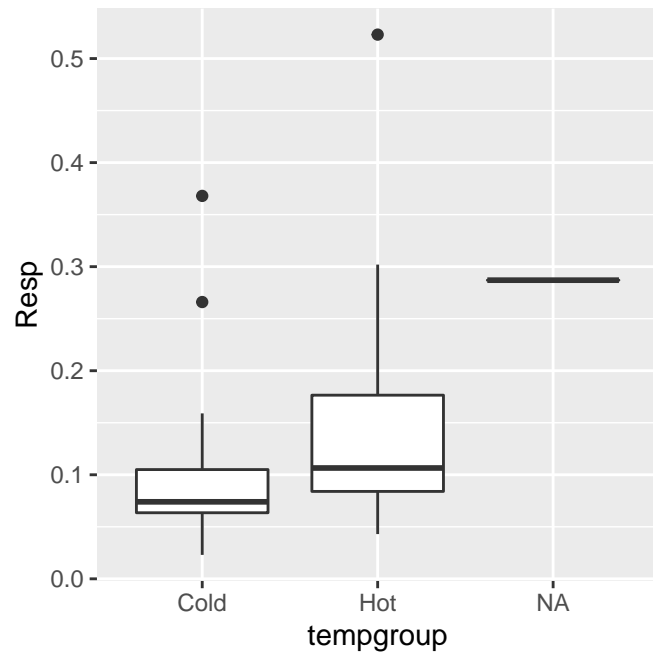
```
qplot(x = tempgroup, y = Resp,  
      data = subset(resptemp, !is.na(tempgroup)) )
```



Boxplot

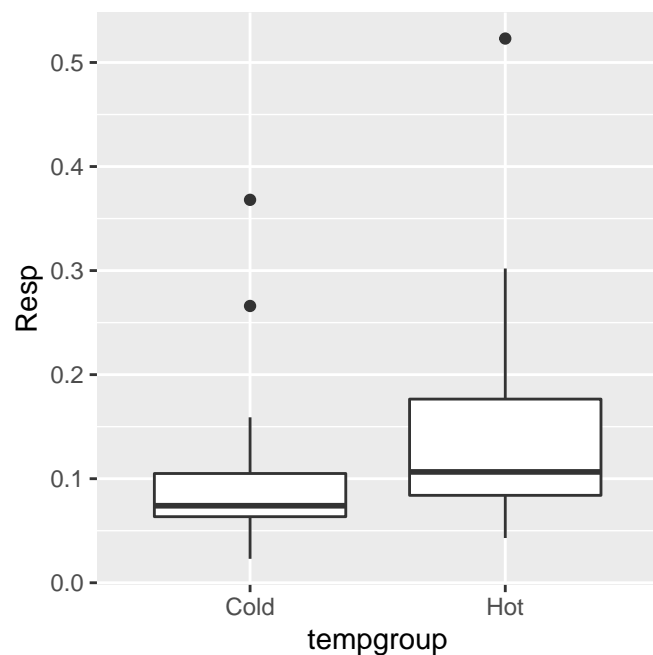
We can make a boxplot instead of a scatterplot by using the `geom` argument to define the plot type. For a boxplot we use `"boxplot"`.

```
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`.

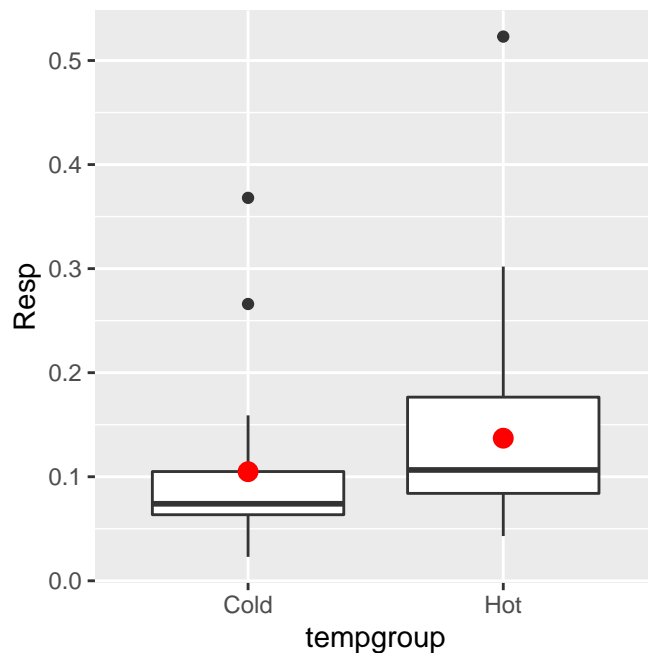

```
resptemp2 = subset(resptemp, !is.na(tempgroup) )
```

Adding the mean to a boxplot

A boxplot shows the median, range, and interquartile range, but not the mean. Since the focus of analysis is 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()`. 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 = mean, geom = "point",  
              color = "red", size = 3)
```

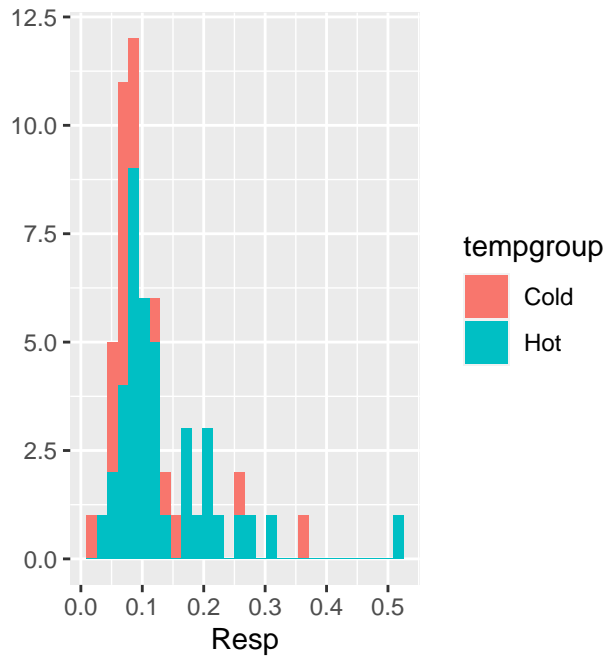


Histogram

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")
```

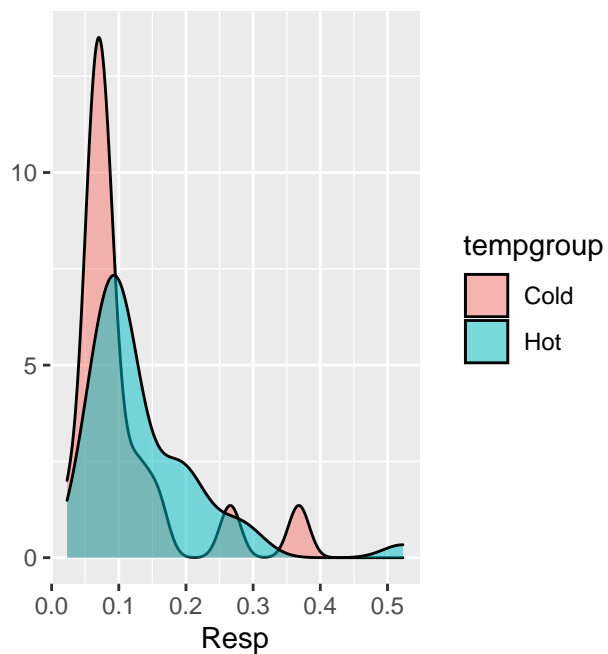
``stat_bin()`` using ``bins = 30``. Pick better value with ``binwidth``.



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 density plot for each group, which 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) )
```

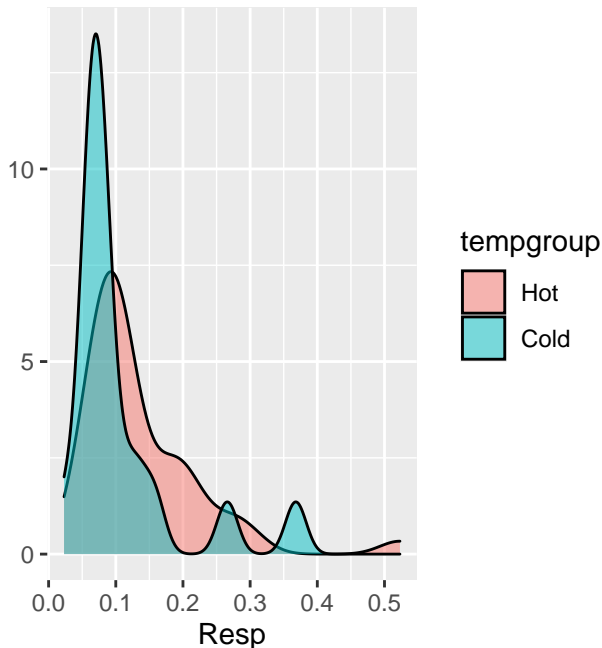


Now let's change the order of the levels of the factor `tempgroup` like we learned how to do earlier, so Hot will come before Cold instead of vice versa.

```
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`.

```
( resptemp2 = 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(0.092, 0.097, 0.178, 0.267, 0.302, : 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.