

FES 524 Lab 01

Winter 2024

Contents

Overall analysis goal	2
Setting up a workflow	2
Cloning a repository	2
General workflow	3
Keeping a clean workspace	3
R help	4
R man pages (documentation)	4
Stack Overflow	4
ChatGPT and internet searches	4
Coding best practices	4
The working directory	5
Using the <code>here</code> package	5
Getting to it	5
The temperature data	6
Reading in a text file with <code>readr::read_table()</code>	6
Using the <code>na =</code> argument to define missing value characters	8
Exploring a dataset in R	9
The respiration datasets	10
Stacking two datasets with <code>rbind()</code>	11
Merging two datasets	13
Finding values in one vector that are not in another	13
Joining two datasets with <code>inner_join()</code>	13
Creating new variables in a dataset based on existing variables	15
Working with missing values in R	17
The <code>na.rm</code> argument	18
Using <code>drop_na()</code> to remove rows with missing values	18
Saving a dataset	18
Data exploration	18
Summary statistics	18
Exploratory graphics	19
Analysis using a two-sample test	25
Wrapping up an analysis	26
Creating a graphic	26
Adding captions in RMarkdown	27
Making a summary table	27

```
knitr::opts_chunk$set(fig.show = "hold")
```

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

Overall analysis goal

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

We have measurements of both respiration and temperature at different sites. However, these data are stored in three datasets instead of one, which are provided to you in the template GitHub repository that you will get once you accept the assignment in GitHub Classroom. A lot of your R time today will be spent reading the three datasets into R and combining them in preparation for a simple analysis.

The three datasets we will be working with today are:

temp.txt

spring_87_resp.txt

fall_87_resp.txt

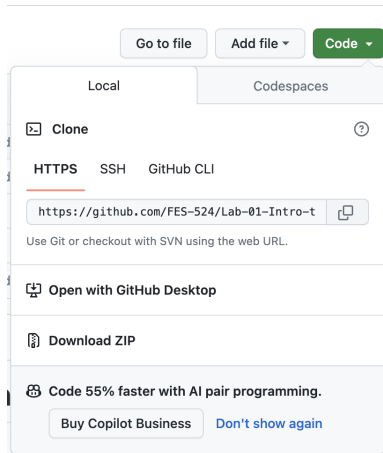
These data files are all found in your template repository in the /Data/ directory.


Setting up a workflow

I highly recommend using git and GitHub for version control and remote storage of code for all your projects. GitHub provides a place where the community of scientists who work with data can make their computer code public, and the “[best practices](#)” that go along with that provide a framework that can help keep you organized in any data analysis task. For example, all the code used to create the `ggplot2` package for R (maybe the most-used R package aside from the base package?) is hosted on GitHub [here](#). For a much less professional example, [here](#) is a repository I made public to go along with one of my PhD chapters. This includes documented code or all the analyses I performed so that, in theory, someone could reproduce my analysis. “Snapshots” of GitHub repositories can additionally be hosted on [Zenodo](#) to give your repository at the time of research publication or thesis defense a permanent DOI. I also recommend this for the sake of open and transparent research practices.


Cloning a repository

1. Once you accept the assignment by following the link posted on Canvas under Lab 01, a template repository will be created for you. You need to then *clone* this repository to your local machine so you can begin making changes and completing the assignment. To do this, find the green **code** dropdown menu and copy the link.





2. Open RStudio. Create a new project using the  button. Follow the prompt and click **Version Control** → **Git**, then paste the url into the dialog box. You can then use the **Browse** button to save the project (essentially just a folder with some nice properties) somewhere you can find it again. RStudio will then automatically open a new session and add a **Git** tab to the upper-right window.

General workflow

After cloning a repository (repo) from GitHub, it now has a reference condition (called a *HEAD*) that references a point in the history of the directory (the point at which you cloned it). Even though you just cloned the repo, you should get in the habit of starting every work session by *pulling* any changes down from the remote repository by clicking the blue “Pull” arrow ( Pull). This will ensure that you start working with the most up-to-date version of the repo in case you did some work from a different computer earlier in the week and are getting back to it on this computer now.

As you work in the version-controlled directory, adding, removing, or changing files, you will notice that files will be listed in the **Git** pane that was added to RStudio. This is sometimes called the *staging area*. You can *stage* the changes in any given file by clicking the check box next to the edited file.

Once you are ready to *commit* the changes to any of the staged files, click the commit button ( Commit). This will open a dialogue box into which you have to submit a *commit message*. This describes the changes made between the last tracked state of the repository to the current state you are committing. Make these descriptive! If you ever need to revert back to a previous state in the directory, you don’t want to be looking through a list of commits that say “commit 1, commit 2, ...”. You will never find the right state to go back to!

Once you commit to the changes, *push* ( Push) the changes up to the remote repository (the one hosted on GitHub). This will act as a backup for you and allow you to work from anywhere on different computers by cloning the repo to any local machine (for example, I host all my projects on GitHub and can pull changed R code down from GitHub onto the OSU computing cluster to run).

Keeping a clean workspace

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

Go to **Tools** → **Global Options**. Then,

1. Uncheck the box next to “Restore .RData into workspace at startup”.
2. Change the option for “Save workspace to .RData on exit” to “Never”.

3. Uncheck the box next to “Always save history (even when not saving .RData)”.

These options ensure that you don’t accidentally load objects from a previous session into your environment that might mess up your current work.

R help

There are three main places where I go for help using R: the documentation within R, searching Stack Overflow or the R mailing list archives, and ChatGPT or internet searches.

Note: ChatGPT has a certain coding style, so I will know if you rely on it too much for writing your code!

R man pages (documentation)

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

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 or the R mailing list archives. I prefer the set-up of Stack Overflow, as the R mailing list archives is in a threaded format that seems harder to navigate to me. Stack Overflow is currently very active with R questions, and often you can find a solution to a problem you are having there.

For the R tagged Stack Overflow posts:

The searchable R help forum is here:

A newer forum for R questions is the [RStudio Community](#), which can be less intimidating than Stack Overflow if you are asking a question for the first time.

ChatGPT and internet searches

Internet searches often provide good resources, but can sometimes take some digging before you find what you are looking for. On the other hand, ChatGPT is quite good at producing code for you and, with some tuning, can usually get it right. However, **do not rely too heavily on ChatGPT for this class**. While I don’t see it as much different from doing a search on Stack Overflow and copying and pasting the code from there, the code on Stack Overflow will often be specific to someone else’s problem and will therefore require (sometimes extensive) tweaking. This process of tweaking can be useful for understanding what the code is actually doing. While this is also true for code generated using ChatGPT, it is often to a lesser extent. I therefore request that you *always* go through any code you get from outside sources line-by-line to understand what is happening in each step and ensure that it is doing what you expect. ChatGPT has a distinct coding style (see the section on [Coding best practices](#)), so I will be able to tell if you rely too heavily on this tool to generate code for you.

Coding best practices

Throughout this course, you will be graded on your analyses and interpretation of results and not on your code (this is a stats course, not a coding course), but it will be much easier for me to provide feedback when something goes wrong if you have clean code that I can look at. For this reason, you should adopt an established coding “style”. For example, I generally try to follow the syntax used by the [tidyverse](#).


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

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

The working directory

The working directory trips people up more than anything else when folks are first learning to use R (or another programming language for that matter). This is another reason I suggest working within projects. Working within projects automatically sets a working directory that is the root of the tracked repo. This means that all the file paths can be relative to the root of the repo and not relative to the file structure on the particular machine you are using.

To see the difference:

1. Open a new R session by going to **Session** → **New Session**. Close the project in the new session using the drop-down menu in the upper-right next to the R project icon .
2. In both sessions in the console, type

```
getwd()
```

and hit **ENTER**.

Do you see the difference? In one case, R will automatically be looking for files within the git repo, in the other, we would first have to tell R where we placed the repo and to then look inside it. This will be different for every user, so paths relative to the root of the repo make for easier sharing of code and ideas and ensure that code doesn't *break* simply by changing machines (e.g., switching from a desktop to a home laptop to work on a project).

Using the **here** package

The **here** package makes relative paths even easier. I highly recommend using the **here** package. Documentation on using this can be found [here](#) (no pun intended), and a blog post to further convince you to use projects and the **here** package can be found [here](#).

Getting to it

We will generally try to use **tidyverse** syntax and commands in this class because that is the trend overall, but there are *base R* functions that do much of the same things but in what some consider to be a less intuitive way. To start, we need to load the collection of functions supplied in the **tidyverse** bundle. This includes functions from many commonly-used utility packages such as the plotting package **ggplot2** and the data manipulation and formatting package **dplyr**.

The **tidyverse** package should be installed on the lab machines, but in case you are using your own machine or for some reason it is not installed and the lines below complain to you about not knowing of a package called **tidyverse**, you can install this set of packages using:

```
# only needed to install and update  
install.packages("tidyverse")
```

Unlike package installation, you will 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()` versus `library()` well.

```
# needed every time you want to use these functions
library(tidyverse)
library(here)
```



The temperature data

Reading in a text file with `readr::read_table()`

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

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

```
?readr::read_table()
```

One thing to notice is that the `col_names` argument defaults to `TRUE`. This means that `read_table()` is expecting the first line in the file to define the names of the variables in the data. This is usually how data are stored, but you can change this argument to `FALSE` or `F` if you have data without column names.

We will assign the name `temperature` to this dataset when we bring it into R. You will see today that assigning names to R objects is a key part of using R. Working off the best practices outlined by the `tidyverse` style, I use *left-hand assignment* (`<-`) to give names to R objects. It is also possible to use `=` for assignment, or *right-hand assignment* `->`. I recommend using left-hand assignment, but pick whichever you like in your work and stick with it.

```
temperature <- read_table(  
  here("Data/temp.txt")  
)
```

```
##  
## -- Column specification -----  
## cols(  
##   Sample = col_double(),  
##   Tech = col_character(),
```

```
## Temp = col_double(),
## DryWt = col_character()
## )
```

Notice that we get a short summary of the data we loaded using the `tidyverse` function `read_table()` and we can now see an object named `temperature` in our RStudio Environment pane, so we have successfully imported the dataset (note that there is a base R version of this function, `read.table()`, which works similarly, but we will focus on using `tidyverse` functions in this class). You should name datasets whatever you like, although I personally recommend names that are easy to type. In R, datasets are called `data.frames` or `tibbles` (depending on some formatting options), and you could refer to `temperature` as a `data.frame` object. I will be using the words dataset, `tibble`, and `data.frame` interchangeably.

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)

## spc_tbl_ [59 x 4] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ Sample: num [1:59] 18 20 22 19 31 30 28 32 29 24 ...
## $ Tech : chr [1:59] "Mark" "Raisa" "Nitnoy" "Nitnoy" ...
## $ Temp : num [1:59] 4.5 4.5 4.5 4.5 5 5 5 5 5 5.5 ...
## $ DryWt : chr [1:59] "0.569" "0.597" "0.603" "0.607" ...
## - attr(*, "spec")=
## .. cols(
## .. Sample = col_double(),
## .. Tech = col_character(),
## .. Temp = col_double(),
## .. DryWt = col_character()
## .. )
```

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

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

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

```
# base R
temperature$DryWt

## [1] "0.569" "0.597" "0.603" "0.607" "0.611" "0.613" "0.622" "0.626" "0.634"
## [10] "0.565" "0.61" "0.62" "." "0.64" "0.656" "0.661" "0.685" "0.695"
## [19] "0.701" "0.528" "0.574" "0.619" "0.627" "0.642" "0.62" "0.65" "0.67"
## [28] "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"
## [46] "0.709" "0.732" "0.739" "0.749" "0.82" "0.836" "0.844" "0.848" "0.859"
## [55] "0.779" "0.801" "0.808" "0.828" "0.83"

# tidyverse
temperature %>%
```



```
pull(., DryWt)
```

```
## [1] "0.569" "0.597" "0.603" "0.607" "0.611" "0.613" "0.622" "0.626" "0.634"
## [10] "0.565" "0.61" "0.62" "." "0.64" "0.656" "0.661" "0.685" "0.695"
## [19] "0.701" "0.528" "0.574" "0.619" "0.627" "0.642" "0.62" "0.65" "0.67"
## [28] "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"
## [46] "0.709" "0.732" "0.739" "0.749" "0.82" "0.836" "0.844" "0.848" "0.859"
## [55] "0.779" "0.801" "0.808" "0.828" "0.83"
```

Note that in the above I use *pipes* (`%>%`). This is basically a “feeding” function that takes the result of the previous call and feeds it to the next function. In the above, I fed the `temperature` dataset to the `pull()` function, which takes `.data` as its first argument. Any argument with `.` in front will be replaced by the “piped in” object when using *tidyverse*. So, `temperature`, since it was fed in via the pipe, was interpreted as the first argument, `.data`, by the `pull()` function.

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

Using the `na =` argument to define missing value characters

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

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

For your later reference, there are two main reasons I have seen to cause the “numeric read as character” problem. First, like in this example, is missing values stored as some miscellaneous character value, such as `na` or `n/a` or `N/A`. The second situation I have 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.

```
temperature <- read_table(
  here("Data/temp.txt"),
  na = "."
)
```

```
##
## -- Column specification -----
## cols(
##   Sample = col_double(),
##   Tech = col_character(),
##   Temp = col_double(),
##   DryWt = col_double()
## )
```

We can see now by the summary from `read_table()` that the column `DryWt` was read in as a “double”, or a numeric vector, this time.

Exploring a dataset in R

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

If we just run the name of this dataset, the whole dataset will print into the R Console. This isn't that useful unless the dataset is small.

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

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

```
# print the first 10 rows
head(temperature, n = 10)
```

```
## # A tibble: 10 x 4
##   Sample Tech      Temp DryWt
##   <dbl> <chr>    <dbl> <dbl>
## 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    0.611
## 6     30 Stephano   5    0.613
## 7     28 Cita      5    0.622
## 8     32 Raisa     5    0.626
## 9     29 Raisa     5    0.634
## 10    24 Cita      5.5  0.565
```

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 from "a"). Be sure to watch out for this when working with categorical variables and names.

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

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

```
dim(temperature)
```

```
## [1] 59  4
```

```
nrow(temperature)
```

```
## [1] 59
```

```
ncol(temperature)
```

```
## [1] 4
```

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

```
summary(temperature)
```

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

The respiration datasets

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

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

```
(
  resp_spring <- read_table(
    here("Data/spring_87_resp.txt")
  )
)
```

```
##
## -- Column specification -----
## cols(
##   Sample = col_double(),
##   Date = col_character(),
##   Resp = col_double()
## )
## # A tibble: 30 x 3
##   Sample Date    Resp
##   <dbl> <chr> <dbl>
## 1     26 Feb-87 0.057
## 2     23 Feb-87 0.085
## 3     25 Feb-87 0.159
## 4     27 Feb-87 0.266
## 5     24 Feb-87 0.368
## 6     19 Jan-87 0.074
## 7     20 Jan-87 0.089
## 8     22 Jan-87 0.117
## 9     18 Jan-87 0.135
## 10    21 Jan-87 0.287
## # i 20 more rows
```

```
(
  resp_fall <- read_table(
    here("Data/fall_87_resp.txt")
  )
)

##
## -- Column specification -----
## cols(
##   Sample = col_double(),
##   Date = col_character(),
##   Resp = col_double()
## )
## # A tibble: 30 x 3
##   Sample Date    Resp
##   <dbl> <chr>  <dbl>
## 1     53 Aug-87 0.093
## 2     55 Aug-87 0.111
## 3     54 Aug-87 0.143
## 4     57 Aug-87 0.205
## 5     56 Aug-87 0.224
## 6     51 Jul-87 0.058
## 7     52 Jul-87 0.081
## 8     48 Jul-87 0.089
## 9     49 Jul-87 0.106
## 10    50 Jul-87 0.119
## # i 20 more rows
```

Stacking two datasets with rbind()

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

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

```
# base R
resp_spring$season <- "spring"
resp_fall$season <- "fall"

# tidyverse
resp_spring <- resp_spring %>%
  mutate(.,
    season = "spring"
  )
resp_fall <- resp_fall %>%
  mutate(.,
    season = "fall"
  )
```

```
# check the results
head(resp_spring)
```

```
## # A tibble: 6 x 4
##   Sample Date    Resp season
##   <dbl> <chr>    <dbl> <chr>
## 1    26 Feb-87  0.057 spring
## 2    23 Feb-87  0.085 spring
## 3    25 Feb-87  0.159 spring
## 4    27 Feb-87  0.266 spring
## 5    24 Feb-87  0.368 spring
## 6    19 Jan-87  0.074 spring
```

```
head(resp_fall)
```

```
## # A tibble: 6 x 4
##   Sample Date    Resp season
##   <dbl> <chr>    <dbl> <chr>
## 1    53 Aug-87  0.093 fall
## 2    55 Aug-87  0.111 fall
## 3    54 Aug-87  0.143 fall
## 4    57 Aug-87  0.205 fall
## 5    56 Aug-87  0.224 fall
## 6    51 Jul-87  0.058 fall
```

Now we can combine these two datasets into a single dataset using the `rbind()` function. The “r” in `rbind()` stands for row. This is an example of a help page that is harder to navigate, but still contains useful information if you delve in far enough.

```
?rbind()
```

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

```
all.equal(
  names(resp_spring),
  names(resp_fall)
)
```

```
## [1] TRUE
```

We made our names all the same on purpose. What if we hadn’t? We can change column names by assigning new ones. Below we will change the name for the season column in `resp_fall` to `Season` (note the capital “S”).

```
# tidyverse renaming
resp_spring <- resp_spring %>%
  rename(.,
    Season = season
  )
```

Now, let’s see the base R version of renaming to change the name of the `season` column in `resp_fall` to match the `Season` column in `resp_spring`. To do this, we need to use *vector indexing*, or *extract* a certain element within a list or vector of data. Base R uses square brackets, `[]`, for this. Try `?"["`.

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

```
# base R renaming
names(resp_fall)[4] <- "Season"
names(resp_fall)
```

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

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

```
resp_all <- rbind(
  resp_fall, resp_spring
)
```

Merging two datasets

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

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

Finding values in one vector that are not in another

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

```
?dplyr::anti_join()
```

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

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

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

```
## # A tibble: 1 x 4
##   Sample Date    Resp Season
##   <dbl> <chr>  <dbl> <chr>
## 1     21 Jan-87 0.287 spring
```

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

Joining two datasets with `inner_join()`

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

Per the documentation, an inner join will:

return all rows from `x` where there are matching values in `y`, and all columns from `x` and `y`. If there are multiple matches between `x` and `y`, all combinations 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 will join on `Sample` like we did above with the anti join. I name the new object `resp_temp` and take a look at the result.

```
# inner join
resp_temp <- inner_join(
  x = resp_all,
  y = temperature,
  by = "Sample"
)

# take a look
str(resp_temp)

## tibble [59 x 7] (S3: tbl_df/tbl/data.frame)
## $ Sample: num [1:59] 53 55 54 57 56 51 52 48 49 50 ...
## $ Date : chr [1:59] "Aug-87" "Aug-87" "Aug-87" "Aug-87" ...
## $ Resp : num [1:59] 0.093 0.111 0.143 0.205 0.224 0.058 0.081 0.089 0.106 0.119 ...
## $ Season: chr [1:59] "fall" "fall" "fall" "fall" ...
## $ Tech : chr [1:59] "Nitnoy" "Stephano" "Stephano" "Nitnoy" ...
## $ Temp : num [1:59] 14 14 14 14 14 13 13 13 13 13 ...
## $ DryWt : num [1:59] 0.765 0.804 0.768 0.709 0.791 0.785 0.795 0.787 0.793 0.727 ...
```

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.

```
# using pipes and tidyverse
temp2 <- temperature %>%
  rename(., Samplenum = Sample)

# using base R and indexing
temp2 <- temperature
names(temp2)[1] <- "Samplenum"
```

To join datasets when the matching variable has different names in the two different datasets, we must provide both names to the `by` argument. 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.

```
# view what the first six lines would look like
head(
  inner_join(
    x = resp_all,
    y = temp2,
    by = c("Sample" = "Samplenum")
  )
)

## # A tibble: 6 x 7
##   Sample Date    Resp Season Tech      Temp DryWt
##   <dbl> <chr>   <dbl> <chr>  <chr>   <dbl> <dbl>
## 1     53 Aug-87 0.093 fall  Nitnoy     14 0.765
## 2     55 Aug-87 0.111 fall  Stephano    14 0.804
```

```
## 3      54 Aug-87 0.143 fall   Stephano      14 0.768
## 4      57 Aug-87 0.205 fall   Nitnoy        14 0.709
## 5      56 Aug-87 0.224 fall   Fatima        14 0.791
## 6      51 Jul-87 0.058 fall   Stephano      13 0.785
```

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

```
nrow(resp_temp)
```

```
## [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 `resp_all` is the dataset with all 60 rows, we list that one first. You can see that the new object `resp_temp`, made with a left join, has 60 rows instead of 59.

```
# left join instead
resp_temp <- left_join(
  x = resp_all,
  y = temperature,
  by = "Sample"
)

# sanity check
nrow(resp_temp)
```

```
## [1] 60
```

Note: In the above code block I have written over our previous object called `resp_temp`. The old one with 59 rows produced from the inner join no longer exists. This is something you need to be careful of in your coding. Overwriting can sometimes cause problems.

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 that I use the function `which()` to find the row of the dataset in which Sample 21 exists and then extract it using row indexing, `[row, column]`.

```
resp_temp[which(resp_temp$Sample == 21), ]
```

```
## # A tibble: 1 x 7
##   Sample Date    Resp Season Tech    Temp DryWt
##   <dbl> <chr>   <dbl> <chr>  <chr> <dbl> <dbl>
## 1     21 Jan-87 0.287 spring <NA>    NA     NA
```

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

Creating new variables in a dataset based on existing variables

Now that we have a single dataset to work with, let's practice creating a new variable in a dataset that is based on existing variables. We will first calculate temperature in degrees Fahrenheit from temperature in

degrees Celsius and add it to the `resp_temp` 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 that, as with any function in add-on packages, the package `dplyr` must be loaded to use `mutate()`.

In `mutate()`, the first argument is the dataset we want to add variables to. I will pipe the dataset in to this argument below. 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.

```
# add new column
resp_temp <- resp_temp %>%
  mutate(.,
    tempf = 32 + (9 / 5) * Temp
  )

head(resp_temp)
```

```
## # A tibble: 6 x 8
##   Sample Date   Resp Season Tech      Temp DryWt tempf
##   <dbl> <chr>   <dbl> <chr>  <chr>   <dbl> <dbl> <dbl>
## 1     53 Aug-87 0.093 fall  Nitroy     14 0.765  57.2
## 2     55 Aug-87 0.111 fall  Stephano    14 0.804  57.2
## 3     54 Aug-87 0.143 fall  Stephano    14 0.768  57.2
## 4     57 Aug-87 0.205 fall  Nitroy     14 0.709  57.2
## 5     56 Aug-87 0.224 fall  Fatima      14 0.791  57.2
## 6     51 Jul-87 0.058 fall  Stephano    13 0.785  55.4
```

Note that in the above `mutate()` command, I was able to refer to the `Temp` column by name without pointing first to the dataset because `mutate` looks within the dataset provided to the first argument.

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

```
?ifelse()
```

In the `ifelse()` function, 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 `resp_temp` dataset is below. The condition we use is that the value of `Temp` is less than 8 °C.

```
# add categorical variable
resp_temp <- resp_temp %>%
  mutate(.,
    tempgroup = ifelse(Temp < 8, "Cold", "Hot")
  )
```

```
# view just the new column
resp_temp %>% pull(., tempgroup)

## [1] "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot"
## [11] "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot"
## [21] "Hot" "Hot" "Hot" "Hot" "Hot" "Cold" "Cold" "Cold" "Cold" "Cold"
## [31] "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" "Cold" NA
## [41] "Cold" "Cold" "Cold" "Cold" "Cold" "Hot" "Hot" "Hot" "Hot" "Hot"
## [51] "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot" "Hot"
```

Working with missing values in R

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

```
summary(resp_temp)

##      Sample      Date      Resp      Season
##  Min.   :18.00   Length:60   Min.    :0.02300   Length:60
##  1st Qu.:32.75   Class :character   1st Qu.:0.07375   Class :character
##  Median :47.50   Mode  :character   Median :0.09550   Mode  :character
##  Mean    :47.50                      Mean    :0.12935
##  3rd Qu.:62.25                      3rd Qu.:0.16300
##  Max.    :77.00                      Max.    :0.52300
##
##      Tech      Temp      DryWt      tempf
##  Length:60   Min.    : 4.50   Min.    :0.5280   Min.    :40.10
##  Class :character   1st Qu.: 7.00   1st Qu.:0.6262   1st Qu.:44.60
##  Mode  :character   Median :11.50   Median :0.7090   Median :52.70
##                      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
##  tempgroup
##  Length:60
##  Class :character
##  Mode  :character
##
##
##
```

R treats missing values differently from other software packages you may have used, so we will 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(resp_temp$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(resp_temp$DryWt, na.rm = T)
```

```
## [1] 0.7086379
```

Using `drop_na()` 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 `tidyverse` function `drop_na()`. Here we could make a new dataset called `resp_temp2` that contains no missing values. You can see it has two less rows (58) than `resp_temp` when we look in our RStudio Environment pane.

```
resp_temp2 <- drop_na(resp_temp)
nrow(resp_temp2)
```

```
## [1] 58
```

There are other functions to use when working with missing values, including `is.na()`, `complete.cases()`, and `na.omit()`. You should check out the help pages for those if you are interested. We will 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` (don’t do this!). 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 from the `tidyverse` package `readr`. Because we are working in an R project, the file will be saved to the root of the project if you do not specify a different file path. I recommend keeping your projects organized with subdirectories that make sense to you. For now, let’s save the file to the `/Data/` directory in the project.

```
# check out the man pages first
?readr::write_csv()

# write our formatted dataset from memory to the hard drive
write_csv(
  resp_temp,
  file = here("Data/combined_resp_and_temp_data.csv")
)
```

Data exploration

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

Summary statistics

In this class we will be making group summaries using functions `group_by()` and `summarise()` from package `dplyr`. Today you will see the code, but we will not discuss it in detail until lab 2. RStudio has a nice data transformation cheat sheet to download (<https://github.com/rstudio/cheatsheets/raw/master/data->

[transformation.pdf](#)) for those who want to get started understanding data wrangling with `dplyr`. If you are like me, you will end up using the functions in `dplyr` quite often in any pre-analysis data wrangling.

We want separate summary statistics of `Resp` for each `tempgroup`. We will calculate the range (minimum and maximum) as well as the median, mean, and standard deviation for each group. Note that because there is a missing value in `tempgroup` we get a third set of summary statistics. I will show the code for doing this using both pipes (my personal preference) and nesting. You can choose whichever makes the most sense to you when using these data wrangling functions. Don't worry too much about the different syntax for now. We will see lots of examples throughout the course and discuss this syntax in greater detail in week 2.

```
# group summaries using pipes
resp_temp %>% group_by(., tempgroup) %>%
  summarise(.,
    across(
      .cols = "Resp",
      .fns = list(
        min = min,
        mean = mean,
        max = max,
        sd = sd
      )
    )
  )
```

```
## # A tibble: 3 x 5
##   tempgroup Resp_min Resp_mean Resp_max Resp_sd
##   <chr>      <dbl>    <dbl>    <dbl>    <dbl>
## 1 Cold      0.023     0.105     0.368    0.0826
## 2 Hot       0.043     0.137     0.523    0.0898
## 3 <NA>      0.287     0.287     0.287    NA
```

```
# the same summaries using nesting
summarise(
  group_by(resp_temp, tempgroup),
  across(
    .cols = "Resp",
    .fns = list(
      min = min,
      median = median,
      mean = mean,
      max = max,
      sd = sd
    )
  )
)
```

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

Exploratory graphics

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

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

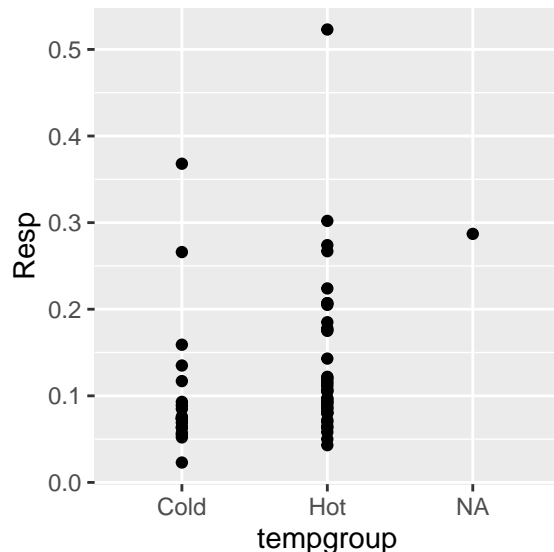
The use of `qplot()` for exploratory graphics is likely review for most people in the class.

You can load package `ggplot2` using the `library()` function, but if you loaded `tidyverse` at the beginning of the session, `ggplot2` is already loaded.

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

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

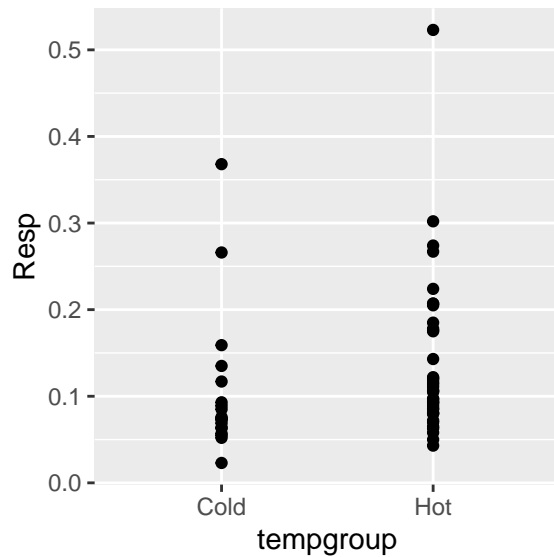
```
## Warning: `qplot()` was deprecated in ggplot2 3.4.0.  
## This warning is displayed once every 8 hours.  
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was  
## generated.
```



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

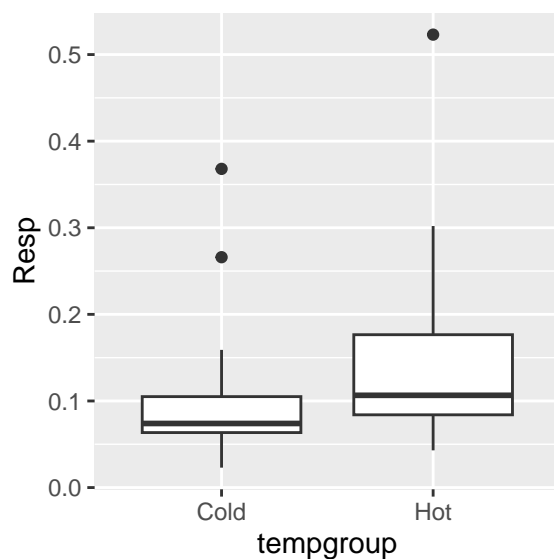
One option for this is to use the `filter()` function from package `dplyr` with `is.na()`. Because we want to remove all the rows where `tempgroup` is NOT NA, so we use `is.na()` with `!` (i.e., the not operator). This is a nice first example of how the `filter()` function is useful for subsetting rows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

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



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

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



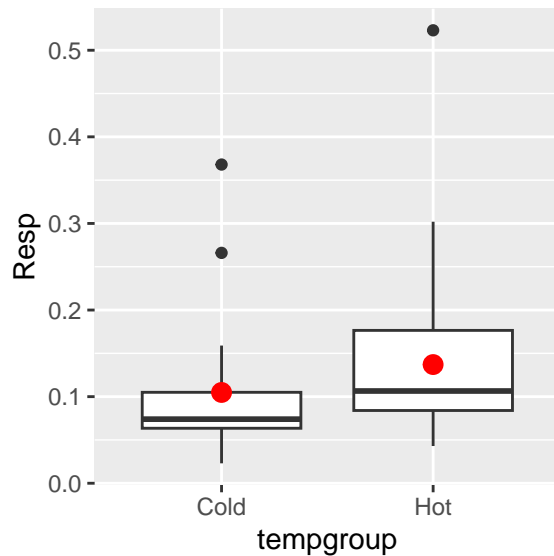
At this point I decided to make a new dataset without that missing `tempgroup` value. We won't be using that value in any analyses today, and it seems like a nuisance. I name the new dataset `resp_temp2`.

```
# remove missing tempgroup
resp_temp2 <- resp_temp %>%
  filter(., !is.na(tempgroup))
```

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

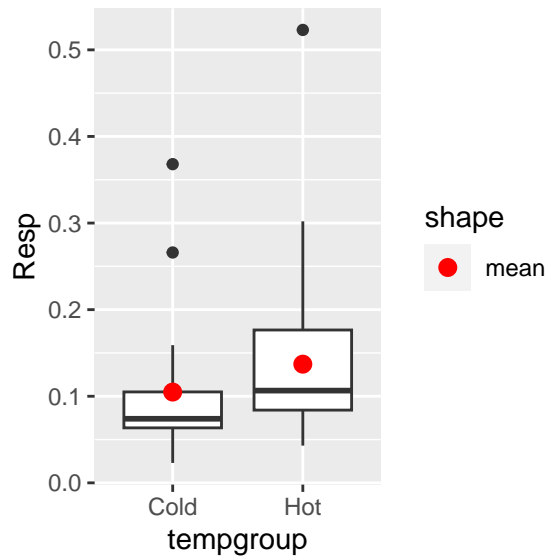
can explore later on your own.

```
qplot(
  x = tempgroup,
  y = Resp,
  data = resp_temp2,
  geom = "boxplot"
) +
stat_summary(
  fun = mean,
  geom = "point",
  color = "red",
  size = 3
)
```



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

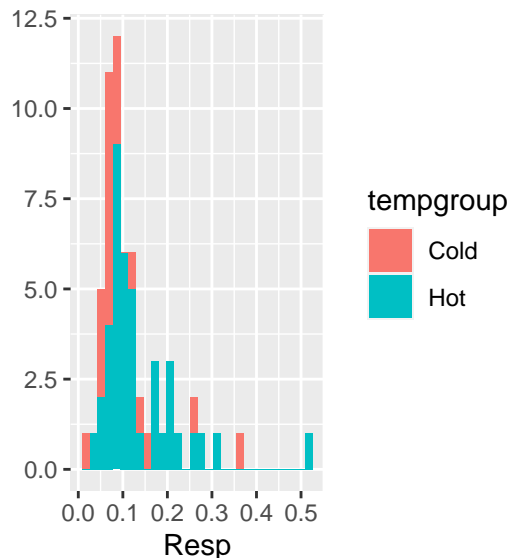
```
qplot(
  x = tempgroup,
  y = Resp,
  data = resp_temp2,
  geom = "boxplot"
) +
stat_summary(
  fun = mean,
  aes(shape = "mean"),
  geom = "point",
  color = "red",
  size = 3
)
```

Histograms and density plots are other commonly used exploratory plots. Here we make two histograms, one for each group, by setting different colors for each temperature group using the `fill` argument. The variable of interest in a histogram is on the x axis, so we put `Resp` on the x axis. Notice we get an informative message from R about the default number of bins used.

```
qplot(
  x = Resp,
  fill = tempgroup,
  data = resp_temp2,
  geom = "histogram"
)
```

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



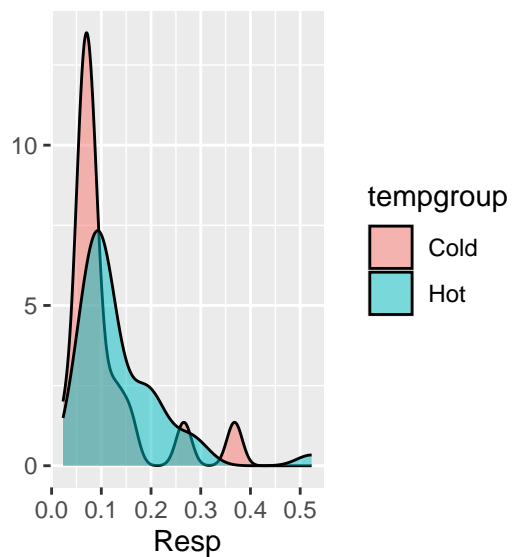
A density plot is essentially a smoothed version of a histogram. Here are the density plots for each group, distinguished by fill color. To make the fill color more transparent instead of totally opaque we can set `alpha` to less than 1.

```
qplot(
  x = Resp,
```

```

fill = tempgroup,
data = resp_temp2,
geom = "density",
alpha = I(0.5) # This syntax removes alpha from the legend
)

```



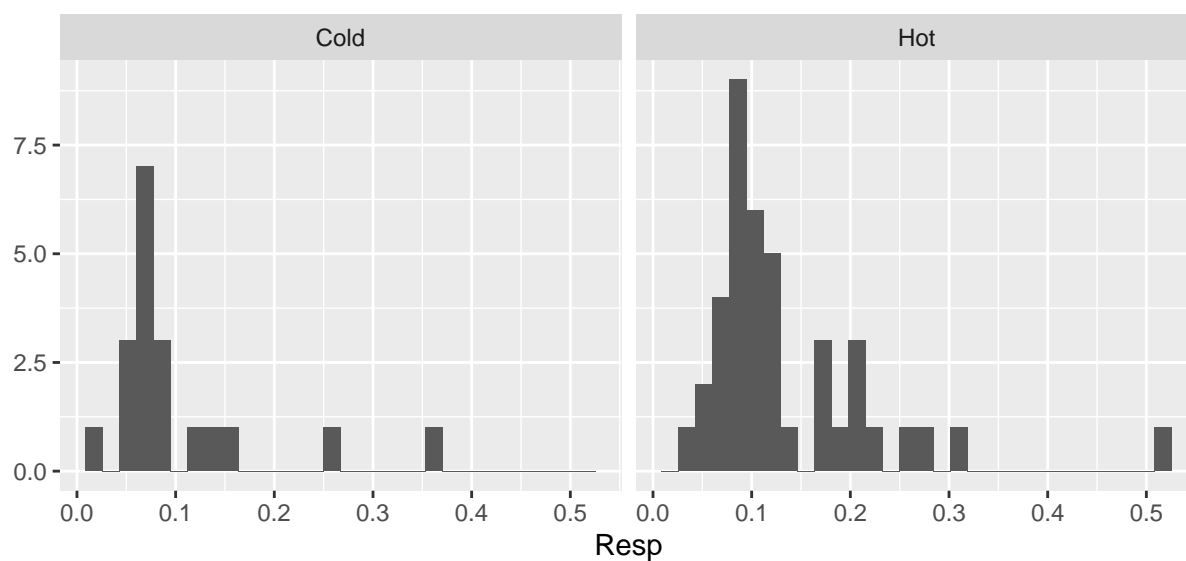
We can plot each group's histogram as a separate plot by using what are called facets. Here are the histograms using `tempgroup` as a facet instead of as fill color.

```

qplot(
  x = Resp,
  data = resp_temp2,
  geom = "histogram",
  facets = ~ tempgroup
)

```

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



Analysis using a two-sample test

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

Each of us would have to decide if the assumptions are reasonably met for a two-sample t-test, a Welch's two-sample t-test if we don't want to assume the variances are equal, or possibly a Wilcoxon rank-sum test if data are extremely skewed. Note, however, that the distributional assumptions are on the sampling distribution of the means, which will tend towards a normal distribution with large sample sizes (remember the Central Limit Theorem?). Thus, a t-test may still be appropriate even with skewed data distributions *if* we have a large sample size.

We will focus just on the function for t-tests in lab today but you can use any analysis you think appropriate on assignments. In R, there is a built-in function `t.test()`, with many different variants of a t-test available.

```
?t.test()
```

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

```
# test with unequal variances assumed
(resp_uneq <- t.test(Resp ~ tempgroup, data = resp_temp2))

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

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

```
# equal variances assumed
(resp_eq <- t.test(Resp ~ tempgroup, var.equal = T, data = resp_temp2))

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

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

Wrapping up an analysis

At the end of an analysis we need to decide how to display the results of any test in our report. This will often involve creating a high-quality graphic and/or a nice summary table. The results from a two-sample t-test can easily be written in the text of a document, and making a graphic of the results from a single test may be overkill. Because of this, I decided to create a graphic with a boxplot for each group with the observed respiration measurements overlaid as a dotplot (i.e., a histogram of dots) and the group means shown as a separate point.

Creating a graphic

Throughout this course (and likely in your work beyond) we will use the `tidyverse` package `ggplot2()` to make my high quality graphics. I add layers to improve how the graphic looks, including changing the background color, dot color, removing some of the grid lines, and adding appropriate units to the axis titles. I also used `base_size` to increase the size of all plot text.

We will see many plots using `ggplot2` this quarter, but won't have time to learn the package in any detail. See the [ggplot2 cheat sheet from RStudio](#) if you are interested in learning more `ggplot2` details.

```
# A plot that summarises the data
# Don't forget to add a caption in your write-up
g1 <- ggplot(resp_temp2, aes(x = tempgroup, y = Resp)) + # define plot axes
  geom_boxplot() + # Add boxplots for each group
  geom_dotplot( # Dotplot by group
    binaxis = "y",
    stackdir = "center",
    dotsize = .5,
    fill = "grey64", colour = "grey64"
  ) +
  stat_summary( # Add means to the plot
    fun = mean,
    geom = "point", shape = 18, size = 4
  ) +
  theme_bw(base_size = 14) + # Change theme (remove grey background), increase text size
  theme(
    panel.grid.major.x = element_blank(), # Remove gridlines from x axis
    panel.grid.minor.y = element_blank() # Remove minor gridlines from y
  ) +
  labs(
    # Change y axis labels with complex units,
    y = expression(
      paste(
        "Soil Respiration ",
        "(g ", "C ", m^-2, " ", day^-1, ")",
        sep = ""
      )
    ),
    # remove x axis labels
    x = NULL
  ) +
  scale_y_continuous(
    breaks = seq(0, .5, by = .1), # Add more breaks
    limits = c(-.05, .55) # Change the limits of the y axis
  )
g1
```

```
## Bin width defaults to 1/30 of the range of the data. Pick better value with
## `binwidth`.
```

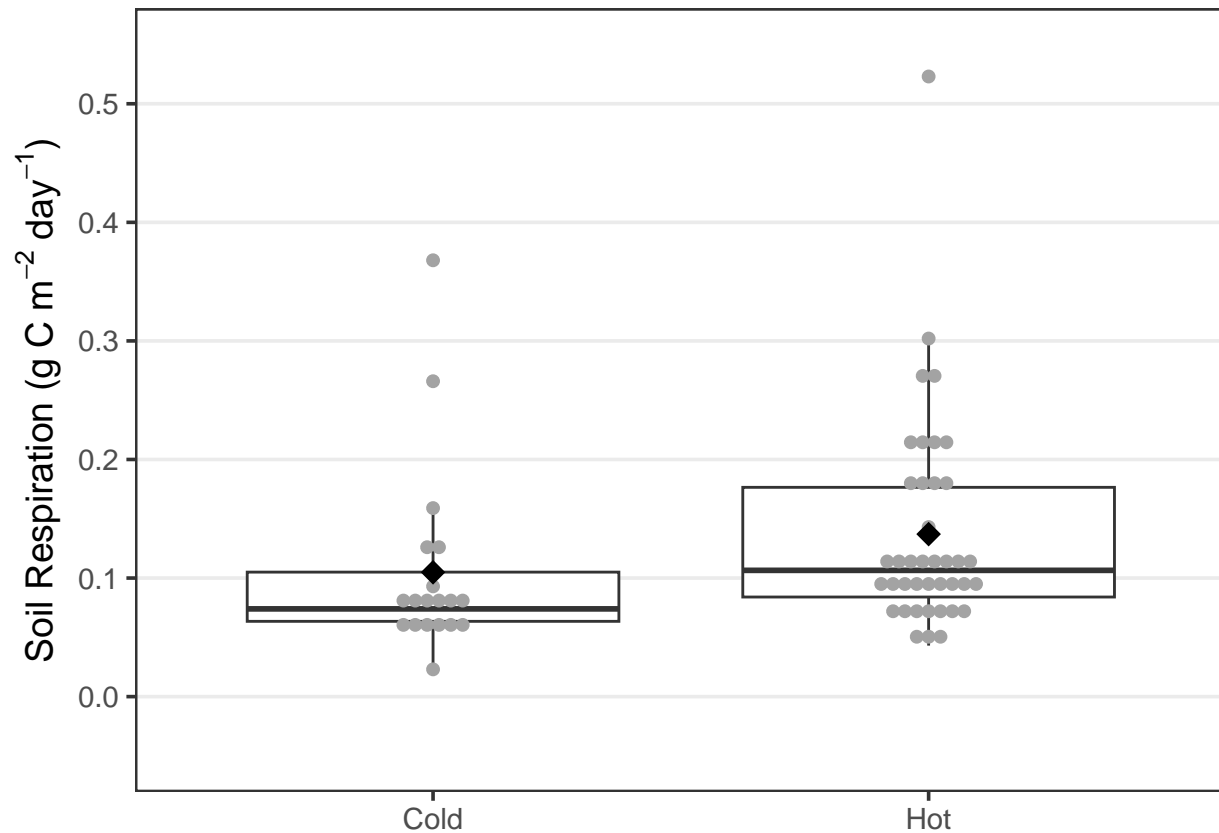


Figure 1: Caption goes here.

Adding captions in RMarkdown

To add a caption to your plot, you can add the `fig.cap` argument to the code block like so:

```
{r fig.cap="Write caption here."}
```

Making a summary table

I made a summary table of descriptive statistics by group as well, including the sample size, mean, and standard deviation. I used function `summarise()` from package `dplyr` to make the summary table. We will learn more about how to use `summarise()` in later lab examples. Note it is up to you to decide what to put in your report. However, if you put something in it must be referred to in the text at some point. If you do use a summary table, don't forget to round to a reasonable number of significant digits (I used three here but it depends on the variable).

```
# Make a quick summary table using package dplyr
# You need to load package if tidyverse or dplyr is not already loaded
# library(dplyr)
sumresp <- resp_temp2 %>%
  group_by(., tempgroup) %>%
  summarise(
    n = length(Resp),
    Mean = round(mean(Resp), 3),
```

Table 1: Summary statistics of respiration from each temperature group.

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

```
SD = round(sd(Resp), 3)
)
# Change column name
names(sumresp)[1] <- "Temperature"
sumresp
```

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

There are a number of ways to export `data.frames` and `tibbles` to other formats directly from R. When using RMarkdown, the easiest way to print a nice table is using the `kable()` function from `kableExtra`. Let's see an example.

```
# load library
library(kableExtra)

# make table
report_tab <- kable(
  x = sumresp,
  format = "latex",
  booktabs = T,
  caption = "Summary statistics of respiration from each temperature group.",
  align = "c",
  digits = 3
)

report_tab
```