Bio724D

Paul M. Magwene and Gregory A. Wray

12/31/22

Table of contents

Preface

This book was designed for a Biology 724D: XXXX, a graduate level course offered in the Department of Biology at Duke University.

Part I

Computing Basics

1 Local computing

1.1 Mac OS

- Root = `/`
- Other disks (USB, etc.) mounted under `/Volumes`
- User directories under `/Users`; your home directory at `/Users/yourusername`
- System wide applications under `/Applications`; users can also install applications under `/Users/username/Applicatons`
- User specific config files under `/Users/username/Library` (hidden by default)
- Unix related executables under `/bin`, `/sbin`, etc.
- `PATH` is an environment variable, settable from command line or `~/.profile`

1.2 Windows 10/11

- Root = `C:\`
- Other disks mounted as `D:\`, `E:`, etc.
- User directories under `C:\Users`; your home directory at `C:\Users\yourusername`
- System wide applications under `c:\Program Files` (64 bit) and `C:\Program Files (x86)` (32 bit)
- `Path` variable settable from "System Properties" dialog

1.3 Both

- Standard shortcuts like "Desktop", "Documents", and "Downloads" are usually subdirectories of your home directory
- Hidden files and directories can be viewed by setting appropriate options in Finder / Explorer

1.4 Recommendations for file naming schemes

- Avoid spaces (or other non-printing characters) and punctuation other than dashes, underscores, and periods in file names
- File names that include dates should preferably follow the ISO-8601 formatting standard, which has the form "YYYY-MM-DD". For example, an experiment done of Jan 12, 2022 should be named something like "2022-01-12-Expt01.csv".
  - The advantage of this is it makes it easy to sort and search by date
- Try and be consistent in your naming schemes. I promise this will make your life (and/or that of your collaborators) easier at some point in your research career!
- See further recommendations from Iowa Stat University's Open Research Data Repository

1.5 Terminology

- A "terminal" is a text input/output environment for interacting with your computer. A terminal used to be a physical device attached to a computer. Strictly speaking, these days we run "terminal emulators" – graphical programs that emulate terminals.
- A "shell" is a program that runs in a terminal that processes and interprets the command you type. You can run different shells in the same terminal.

1.6 Mac OS

- Terminal accessed by running "Terminal" program (in the "Applications/Utilities" folder)
- Default shell is zsh; but bash also included by default (invoke at command line by typing `bash`)

1.7 Windows

- Default terminal is "Command Prompt" but new "Windows Terminal" is better choice
- Several options for shell but I recommend "PowerShell"

1.8 Commands for navigating the file system on Unix based systems

NOTE: The following commands also work in Windows PowerShell

- `pwd` – prints name of your "working" directory (i.e. the directory you're currently in)
- `ls` – lists the contents of the working directory
- `cd` – change directory
  - `cd Downloads`
  - `cd ~` (move to your home directory)
- `mkdir` – make a new directory
  - `mkdir foo_dir`
  - `mkdir bar_dir`
- `rmdir` – remove a directory (must be empty)
  - `rmdir bar_dir`
- `rm` – remove a file
  - `rm bar.txt`
- `cp` – copy a file
  - `cp foo.txt foo_copy.txt`
- `mv` – move a file or directory
  - `mv foo.txt baz.txt`
  - `mv foo_dir foo.dir`
- `cat` – write the contents of a file to the terminal:
  - `cat foo.txt`
- `more` – a pager program. View the contents of a file, one page at a time. Type `<space>` to advance pages, `q` to quit.

Side note: Why is it common to use names like `foo`, `bar`, and `baz` in examples? See https://en.wikipedia.org/wiki/Foobar

1.9 Shortcuts for working with the file system

- `~` – refers to the users home directory.
  - `cd ~` = change to my home directory
  - `ls ~/*.txt` = list all files ending with the prefix `.txt` in your home directory
- `.` – the current directory
  - `cd ./tmp` = move to the directory called `tmp` that is located in the directory the `cd` command is executed from.
- `..` – the directory above the current directory (if it exists)
  - `cd ..` = move up one level in the directory hierarchy. For example if `pwd` is `/home/jsmith` then `cd ..` moves you to `/home`
- `/` – the root directory of the file system
- `cd /data` = move to the `data` directory that is the subdirectory of the root
- `ls /` = list all files and directories in the root directory

2 Remote computing

2.1 Moving files to/from your VM

2.1.1 Using Cyberduck (or another SFTP client)

Cyberduck provides a graphical interface for moving files back and forth to/from a remote machine. We'll demonstrate how to use Cyberduck in class.

2.1.2 Using the command line

- `scp` – secure copy. Command line tool to copy files to or from a remote machine via SSH.
  - Move a local file (`foo.txt`) to your virtual machine:
    - `scp foo.txt netid@hostname:~`
  - Move a remote file (`~/data/bar.txt`) to your local machine (`~/bio724/data`):
    - `scp netid@hostname:~/data/bar.txt ~/bio724/data` (saves `bar.txt` under the local directory `~/bio724/data` assuming that directory already exists)
- `wget` – a command line program for downloading files from the web. You would typically use this to download files from a URL to a remote machine.

  To illustrate the use of `wget`, we'll download a file of interest from the NIH National Center for Biotechnology Information (NCBI), which hosts databases like Genbank, SRA, Pubmed, etc.

  - In your web browser, navigate to the NCBI SARS-CoV-2 Resources website. About half-way down the page are a set of blue buttons linking to information about the SARS-CoV-2 Genome Reference Sequence (NC_045512).
  - Right click the "Download Annotation" button and copy the URL link and then use `wget` to download the genome annotation file to your VM.

    ```
    wget https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/009/858/895/GCF_009858895.2_ASM98
    ```

- The `.gz` prefix indicates that this is a compressed file; compressed using a tool called `gzip`. To uncompress this file we can use the `gunzip` command as follows:

  ```
  gunzip GCF_009858895.2_ASM985889v3_genomic.gff.gz
  ```

  This will create the uncompressed file named `GCF_009858895.2_ASM985889v3_genomic.gff`.
- Let's create a directory for genome annotation files and move our file there:

  ```
  mkdir ~/genome_annotations
  ```

  ```
  mv GCF_009858895.2_ASM985889v3_genomic.gff ~/genome_annotations/
  ```
- GFF files are a commonly used format for genome annotations. This is a simple tab-delimited file format with nine columns. A full specification of the GFF format is provided here: https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md

Part II

R

3 Getting Started with R

3.1 What is R?

R is a statistical computing environment and programming language. It is free, open source, and has a large and active community of developers and users. There are many different R packages (libraries) available for conducting out a wide variety of different analyses, for everything from genome sequence data to geospatial information.

3.2 What is RStudio?

RStudio (http://www.rstudio.com/) is an open source integrated development environment (IDE) that provides a nicer graphical interface to R than does the default R GUI.

The figure below illustrates the RStudio interface, in it's default configuration. For the exercises below you'll be primarily entering commands in the "console" window. We'll review key parts of the RStudio interface in greater detail in class.

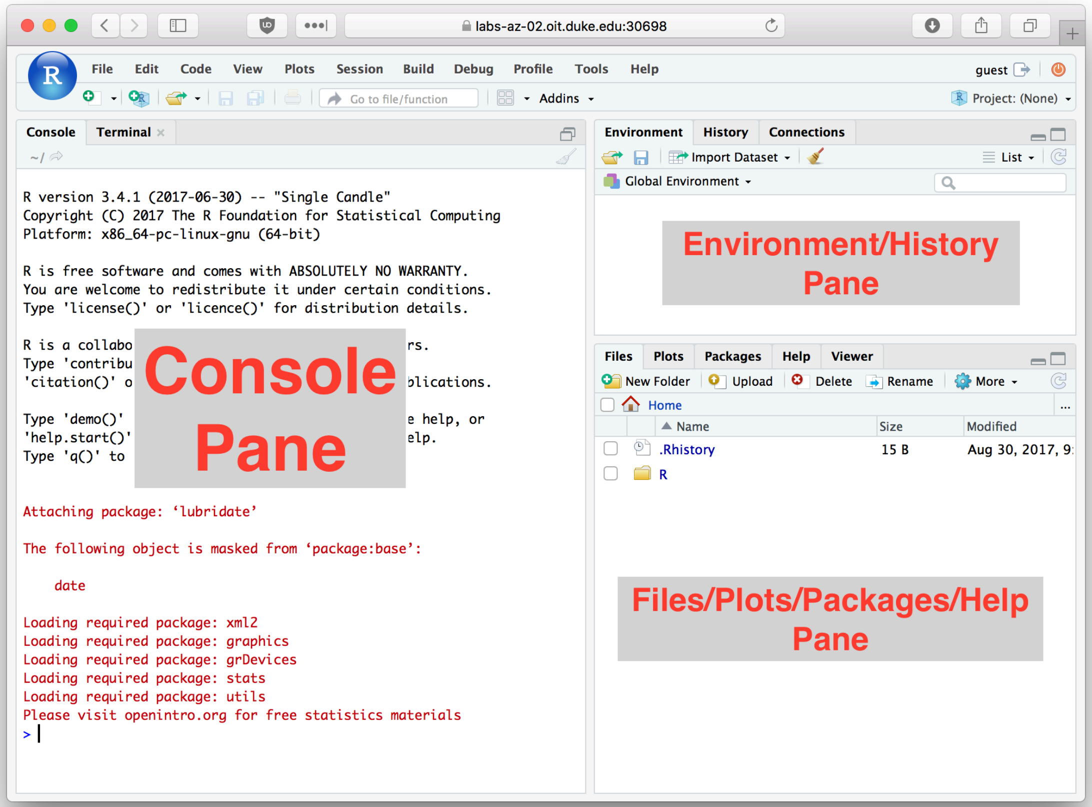
3.3 Entering commands in the console

You can type commands directly in the console. When you hit Return (Enter) on your keyboard the text you typed is evaluated by the R interpreter. This means that the R program reads your commands, makes sure there are no syntax errors, and then carries out any commands that were specified.

Try evaluating the following arithmetic commands in the console:

```
10 + 5
10 - 5
10 / 5
10 * 5
```

If you type an incomplete command and then hit Return on your keyboard, the console will show a continuation line marked by a `+` symbol. For example enter the incomplete statement `(10 + 5` and then hit Enter. You should see something like this.



```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collabo                                rs.
Type 'contribu
'citation()' o                                plications.

Type 'demo()'                                 e help, or
'help.start()'                                elp.
Type 'q()' to

Attaching package: 'lubridate'

The following object is masked from 'package:base':

    date

Loading required package: xml2
Loading required package: graphics
Loading required package: grDevices
Loading required package: stats
Loading required package: utils
Please visit openintro.org for free statistics materials
>
```

	Name	Size	Modified
	.Rhistory	15 B	Aug 30, 2017, 9:
	R		

Figure 3.1: RStudio window with the panes labeled

```
> (10 + 5
+
```

The continuation line tells you that R is waiting for additional input before it evaluates what you typed. Either complete your command (e.g. type the closing parenthesis) and hit Return, or hit the "Esc" key to exit the continuation line without evaluating what you typed.

3.4 Comments

When working in the R console, or writing R code, the pound symbol (`#`) indicates the start of a comment. Anything after the `#`, up to the end of the current line, is ignored by the R interpretter.

```
# This line will be ignored
5 + 4 # the first part of this line, up to the #, will be evaluated
```

Throughout this course I will often include short explanatory comments in my code examples.

When I want to display the output generated by an R statement typed at the console I will generally use a display convention in which I prepend the results with the symbols `##`.

```
5 + 4  # same as above but with output displayed
```

```
[1] 9
```

3.5 Using R as a Calculator

The simplest way to use R is as a fancy calculator. Evaluate each of the following statements in the console.

```
10 + 2 # addition
10 - 2 # subtraction
10 * 2 # multiplication
10 / 2 # division
10 ^ 2 # exponentiation
10 ** 2 # alternate exponentiation
pi * 2.5^2 # R knows about some constants such as Pi
10 %% 3 # modulus operator -- gives remainder after division
```

```
10 %/% 3 # integer division
```

Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing statements.

```
(10 + 2)/4-5   # was the output what you expected?
(10 + 2)/(4-5) # compare the answer to the above
```

Division by zero produces an object that represents infinite numbers. Infinite values can be either positive or negative

```
1/0
```

```
[1] Inf
```

```
-1/0
```

```
[1] -Inf
```

Invalid calculations produce a objected called `NaN` which is short for "Not a Number":

```
0/0  # invalid calculation
```

```
[1] NaN
```

3.5.1 Common mathematical functions

Many commonly used mathematical functions are built into R. Here are some examples:

```
abs(-3)   # absolute value
```

```
[1] 3
```

```
cos(pi/3) # cosine
```

```
[1] 0.5
```

```r
sin(pi/3) # sine
```

```
[1] 0.8660254
```

```r
log(10)   # natural logarithm
```

```
[1] 2.302585
```

```r
log10(10) # log base 10
```

```
[1] 1
```

```r
log2(10) # log base 2
```

```
[1] 3.321928
```

```r
exp(1)    # exponential function
```

```
[1] 2.718282
```

```r
sqrt(10)  # square root
```

```
[1] 3.162278
```

```r
10^0.5  # same as square root
```

```
[1] 3.162278
```

3.6 Variable assignment

An important programming concept in all programming languages is that of "variable assignment". Variable assignment is the act of creating labels that point to particular data values in a computers memory, which allows us to apply operations to the labels rather than directly to specific. Variable assignment is an important mechanism of abstracting and generalizing computational operations.

Variable assignment in R is accomplished with the assignment operator, which is designated as `<-` (left arrow, constructed from a left angular brack and the minus sign). This is illustrated below:

```r
x <- 10  # assign the variable name 'x' the value 10
sin(x)   # apply the sin function to the value x points to
```

```
[1] -0.5440211
```

```r
x <- pi  # x now points to a different value
sin(x)   # the same function call now produces a different result
```

```
[1] 1.224647e-16
```

```r
         # note that sin(pi) == 0, but R returns a floating point value very
         # very close to but not zero
```

3.6.1 Valid variable names

As described in the R documentation, "A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as '.2way' are not valid, and neither are the reserved words."

Here are some examples of valid and invalid variable names. Mentally evaluate these based on the definition above, and then evaluate these in the R interpetter to confirm your understanding :

```
x <- 10
x.prime <- 10
x_prime <- 10
my.long.variable.name <- 10
```

```
another_long_variable_name <- 10
_x <- 10
.x <- 10
2.x <- 2 * x
```

3.7 Data types

The phrase “data types” refers to the representations of information that a programming language provides. In R, there are three core data types representing numbes, logical values, and strings. You can use the function `typeof()` to get information about an objects type in R.

3.7.1 Numeric data types

There are three standard types of numbers in R.

1) “double” – this is the default numeric data type, and is used to represent both real numbers and whole numbers (unless you explicitly ask for integers, see below). “double” is short for “double precision floating point value”. All of the previous computations you’ve seen up until this point used data of type double.

   ::: {.cell}

   ```
   typeof(10.0)  # real number
   ```

   ::: {.cell-output .cell-output-stdout} `[1] "double"` :::

   ```
   typeof(10)  # whole numbers default to doubles
   ```

   ::: {.cell-output .cell-output-stdout} `[1] "double"` ::: :::

2) “integer” – when your numeric data involves only whole numbers, you can get slighly better performance using the integer data type. You must explicitly ask for numbers to be treated as integers.

   ::: {.cell}

   ```
   typeof(as.integer(10))  # now treated as an integer
   ```

   ::: {.cell-output .cell-output-stdout} `[1] "integer"` ::: :::

3) "complex" – R has a built-in data type to represent complex numbers – numbers with a "real" and "imaginary" component. We won't encounter the use of complex numbers in this course, but they do have many important uses in mathematics and engineering and also have some interesting applications in biology.

   ::: {.cell}

   ```
   typeof(1 + 0i)
   ```

   ::: {.cell-output .cell-output-stdout} `[1] "complex"` :::

   ```
   sqrt(-1)      # sqrt of -1, using doubles
   ```

   ::: {.cell-output .cell-output-stderr} `Warning in sqrt(-1): NaNs produced` :::

   ::: {.cell-output .cell-output-stdout} `[1] NaN` :::

   ```
   sqrt(-1 + 0i) # sqrt of -1, using complex numbers
   ```

   ::: {.cell-output .cell-output-stdout} `[1] 0+1i` ::: :::

3.7.2 Logical values

When we compare values to each other, our calculations no longer return "doubles" but rather `TRUE` and `FALSE` values. This is illustrated below:

```
10 < 9  # is 10 less than 9?
```

```
[1] FALSE
```

```
10 > 9  # is 10 greater than 9?
```

```
[1] TRUE
```

```
10 <= (5 * 2) # less than or equal to?
```

```
[1] TRUE
```

```
10 >= pi # greater than or equal to?
```

```
[1] TRUE
```

```r
10 == 10 # equals?
```

```
[1] TRUE
```

```r
10 != 10 # does not equal?
```

```
[1] FALSE
```

`TRUE` and `FALSE` objects are of "logical" data type (known as "Booleans" in many other languages, after the mathematician George Boole).

```r
typeof(TRUE)
typeof(FALSE)
x <- FALSE
typeof(x)  # x points to a logical
x <- 1
typeof(x)  # the variable x no longer points to a logical
```

When working with numerical data, tests of equality can be tricky. For example, consider the following two comparisons:

```r
10 == (sqrt(10)^2) # Surprised by the result? See below.
4 == (sqrt(4)^2) # Even more confused?
```

Mathematically we know that both $(\sqrt{10})^2 = 10$ and $(\sqrt{4})^2 = 4$ are true statements. Why does R tell us the first statement is false? What we're running into here are the limits of computer precision. A computer can't represent $\sqrt{10}$ exactly, whereas $\sqrt{4}$ can be exactly represented. Precision in numerical computing is a complex subject and a detailed discussion is beyond the scope of this course. However, it's important to be aware of this limitation (this limitation is true of any programming language, not just R).

To test "near equality" R provides a function called `all.equal()`. This function takes two inputs – the numerical values to be compared – and returns `TRUE` if their values are equal up to a certain level of tolerance (defined by the built-in numerical precision of your computer).

```r
all.equal(10, sqrt(10)^2)
```

```
[1] TRUE
```

Here's another example where the simple equality operator returns an unexpected result, but `all.equal()` produces the comparison we're likely after.

```
sin(pi) == 0
```

```
[1] FALSE
```

```
all.equal(sin(pi), 0)
```

```
[1] TRUE
```

3.7.2.1 Logical operators

Logical values support Boolean operations, like logical negation ("not"), "and", "or", "xor", etc. This is illustrated below:

```
!TRUE  # logical negation -- reads as "not x"
```

```
[1] FALSE
```

```
TRUE & FALSE # AND: are x and y both TRUE?
```

```
[1] FALSE
```

```
TRUE | FALSE # OR: are either x or y TRUE?
```

```
[1] TRUE
```

```
xor(TRUE,FALSE)  # XOR: is either x or y TRUE, but not both?
```

```
[1] TRUE
```

The function `isTRUE` can be useful for evaluating the state of a variable:

```r
x <- sample(1:10, 1) # sample a random number in the range 1 to 10
isTRUE(x > 5)  # was the random number picked greater than 5?
```

```
[1] TRUE
```

3.7.3 Character strings

Character strings (“character”) represent single textual characters or a longer sequence of characters. They are created by enclosing the characters in text either single our double quotes.

```r
typeof("abc")  # double quotes
```

```
[1] "character"
```

```r
typeof('abc')  # single quotes
```

```
[1] "character"
```

Character strings have a length, which can be found using the `nchar` function:

```r
first.name <- "jasmine"
nchar(first.name)
```

```
[1] 7
```

```r
last.name <- 'smith'
nchar(last.name)
```

```
[1] 5
```

There are a number of built-in functions for manipulating character strings. Here are some of the most common ones.

3.7.3.1 Joining strings

The `paste()` function joins two characters strings together:

```r
paste(first.name, last.name)  # join two strings
```

```
[1] "jasmine smith"
```

```r
paste("abc", "def")
```

```
[1] "abc def"
```

Notice that `paste()` adds a space between the strings? If we didn't want the space we can call the `paste()` function with an optional argument called `sep` (short for separator) which specifies the character(s) that are inserted between the joined strings.

```r
paste("abc", "def", sep = "")  # join with no space; "" is an empty string
```

```
[1] "abcdef"
```

```r
paste0("abc", "def") # an equivalent function with no space in newer version of R
```

```
[1] "abcdef"
```

```r
paste("abc", "def", sep = "|") # join with a vertical bar
```

```
[1] "abc|def"
```

3.7.3.2 Splitting strings

The `strsplit()` function allows us to split a character string into substrings according to matches to a specified split string (see `?strsplit` for details). For example, we could break a sentence into it's constituent words as follows:

```r
sentence <- "Call me Ishmael."
words <- strsplit(sentence, " ")  # split on space
words
```

```
[[1]]
[1] "Call"     "me"       "Ishmael."
```

Notice that `strsplit()` is the reverse of `paste()`.

3.7.3.3 Substrings

The `substr()` function allows us to extract a substring from a character object by specifying the first and last positions (indices) to use in the extraction:

```
substr("abcdef", 2, 5)  # get substring from characters 2 to 5
```

```
[1] "bcde"
```

```
substr(first.name, 1, 3) # get substring from characters 1 to
```

```
[1] "jas"
```

3.8 Packages

Packages are libraries of R functions and data that provide additional capabilities and tools beyond the standard library of functions included with R. Hundreds of people around the world have developed packages for R that provide functions and related data structures for conducting many different types of analyses.

Throughout this course you'll need to install a variety of packages. Here I show the basic procedure for installing new packages from the console as well as from the R Studio interface.

3.8.1 Installing packages from the console

The function `install.packages()` provides a quick and conveniet way to install packages from the R console.

3.8.2 Install the tidyverse package

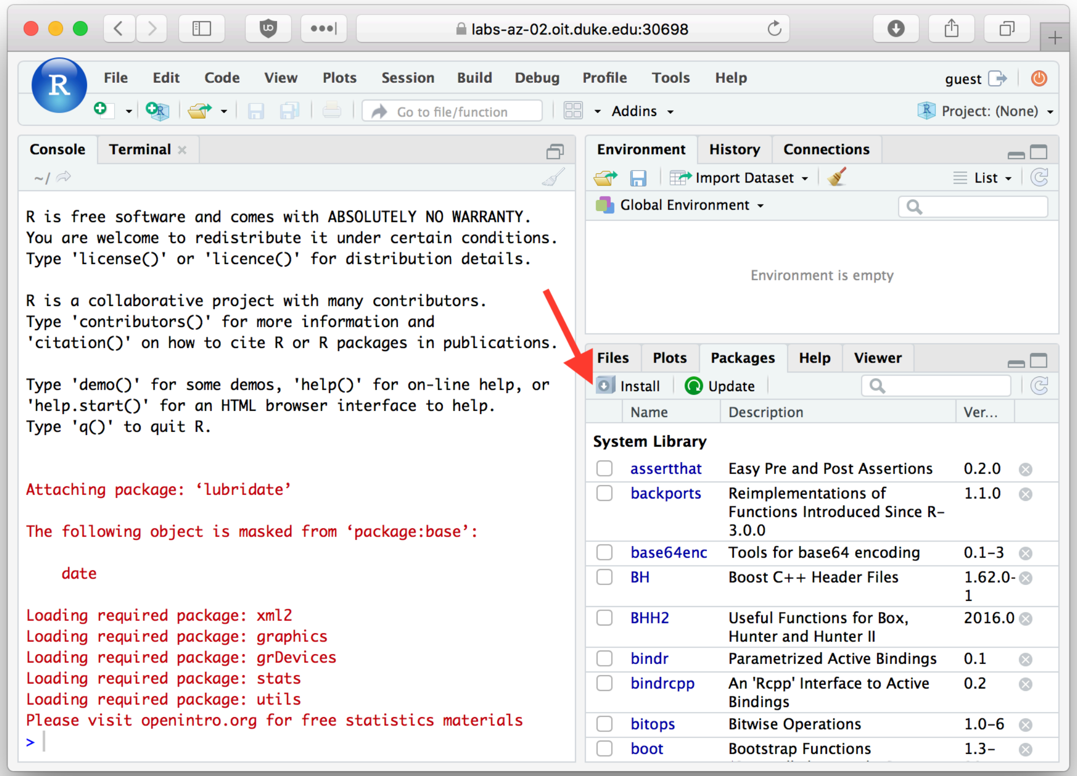
To illustrate the use of `install.packages()`, we'll install a collection of packages (a "meta-package") called the tidyverse. Here's how to install the tidyverse meta-package from the R console:

```
install.packages("tidyverse", dependencies = TRUE)
```

The first argument to `install.packages` gives the names of the package we want to install. The second argument, `dependencies = TRUE`, tells R to install any additional packages that tidyverse depends on.

3.8.3 Installing packages from the RStudio dialog

You can also install packages using a graphical dialog provided by RStudio. To do so pick the `Packages` tab in RStudio, and then click the `Install` button.


labs-az-02.oit.duke.edu:30698
File Edit Code View Plots Session Build Debug Profile Tools Help
guest
Go to file/function
Addins
Project: (None)
Console Terminal
~/
Environment History Connections
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
Install Update

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.


Attaching package: ‘lubridate’

The following object is masked from ‘package:base’:

    date

Loading required package: xml2
Loading required package: graphics
Loading required package: grDevices
Loading required package: stats
Loading required package: utils
Please visit openintro.org for free statistics materials
>
```

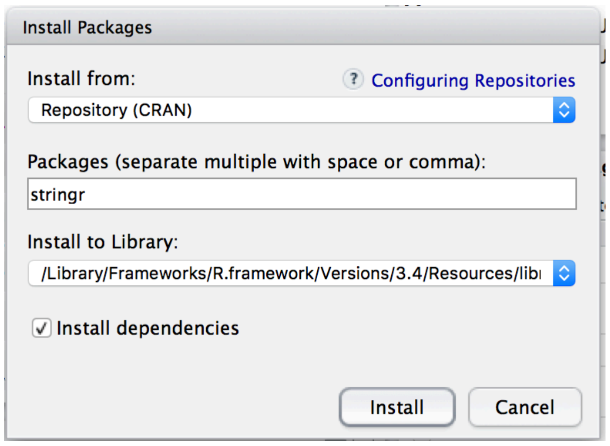
Name	Description	Ver...
System Library		
assertthat	Easy Pre and Post Assertions	0.2.0
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.0
base64enc	Tools for base64 encoding	0.1-3
BH	Boost C++ Header Files	1.62.0-1
BHH2	Useful Functions for Box, Hunter and Hunter II	2016.0
bindr	Parametrized Active Bindings	0.1
bindrcpp	An 'Rcpp' Interface to Active Bindings	0.2
bitops	Bitwise Operations	1.0-6
boot	Bootstrap Functions	1.3-

Figure 3.2: The Packages tab in RStudio

In the packages entry box you can type the name of the package you wish to install. Let's install another useful package called "stringr". Type the package name in the "Packages" field, make sure the "Install dependencies" check box is checked, and then press the "Install" button.

3.8.4 Loading packages with the `library()` function

Once a package is installed on your computer, the package can be loaded into your R session using the `library` function. To insure our previous install commands worked correctly, let's load one of the packages we just installed.


Install Packages

Install from: Repository (CRAN)

Packages (separate multiple with space or comma): stringr

Install to Library: /Library/Frameworks/R.framework/Versions/3.4/Resources/libı

[✓] Install dependencies

Figure 3.3: Package Install Dialog

```
library(tidyverse)
```

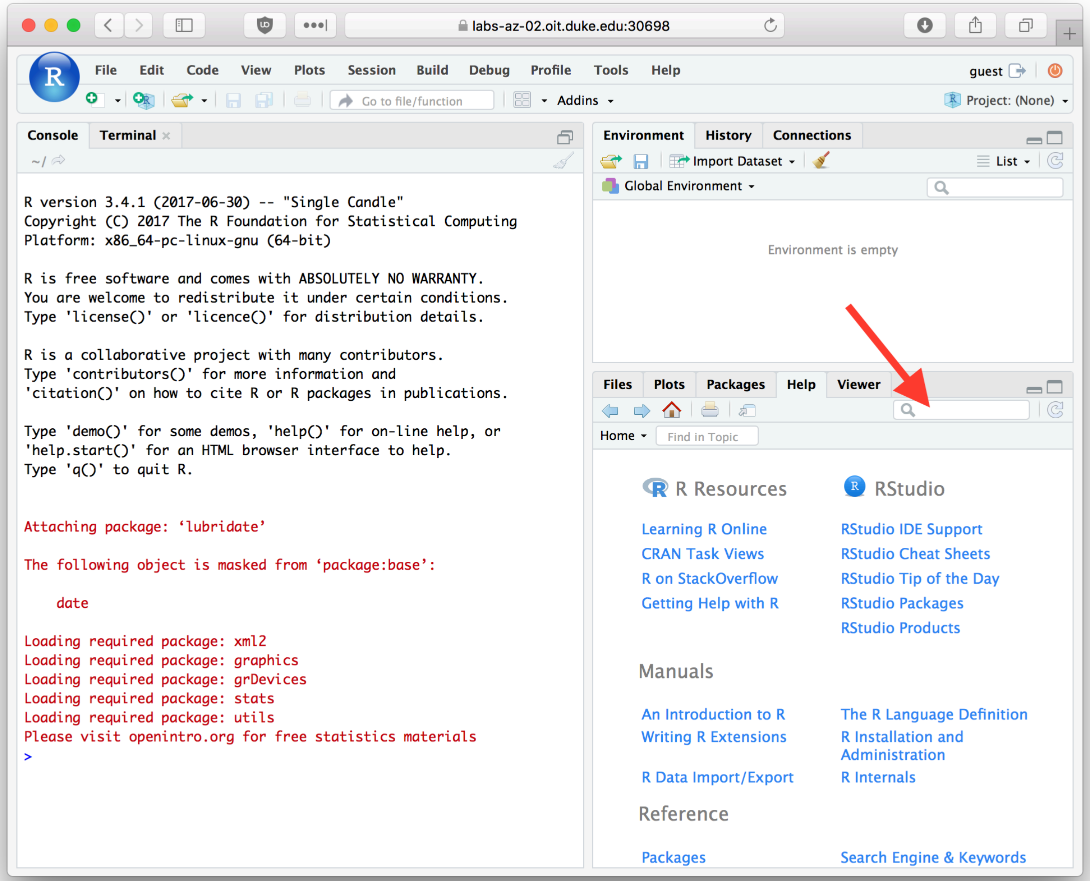
Since the tidyverse pacakge is a "meta-package" it provides some additional info about the sub-packages that got loaded.

When you load tidyverse, you will also see a message about "Conflicts" as several of the functions provided in the dplyr package (a sub-package in tidyverse) conflict with names of functions provided by the "stats" package which usually gets automically loaded when you start R. The conflicting funcdtions are `filter` and `lag`. The conflicting functions in the stats package are `lag` and `filter` which are used in time series analysis. The `dplyr` functions are more generally useful. Furthermore, if you need these masked functions you can still access them by prefacing the function name with the name of the package (e.g. `stats::filter`).

We will use the "tidyverse" package for almost every class session and assignment in this class. Get in the habit of including the `library(tidyverse)` statement in all of your R documents.

3.9 The R Help System

R comes with fairly extensive documentation and a simple help system. You can access HTML versions of the R documentation under the Help tab in Rstudio. The HTML documentation also includes information on any packages you've installed. Take a few minutes to browse through the R HTML documentation. In addition to the HTML documentation there is also a search box where you can enter a term to search on (see red arrow in figure below).



```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.


Attaching package: ‘lubridate’

The following object is masked from ‘package:base’:

    date

Loading required package: xml2
Loading required package: graphics
Loading required package: grDevices
Loading required package: stats
Loading required package: utils
Please visit openintro.org for free statistics materials
>
```

Figure 3.4: The RStudio Help tab

3.9.1 Getting help from the console

In addition to getting help from the RStudio help tab, you can directly search for help from the console. The help system can be invoked using the `help` function or the `?` operator.

```
help("log")
?log
```

If you are using RStudio, the help results will appear in the "Help" tab of the Files/Plots/Packages/Help/Viewer (lower right window by default).

What if you don't know the name of the function you want? You can use the `help.search()` function.

```
help.search("log")
```

In this case `help.search("log")` returns all the functions with the string `log` in them. For more on `help.search` type `?help.search`.

Other useful help related functions include `apropos()` and `example()`, `vignette()`. `apropos` returns a list of all objects (including variable names and function names) in the current session that match the input string.

```
apropos("log")
```

```
 [1] "as.data.frame.logical" "as.logical"            "as.logical.factor"
 [4] "dlogis"                "is.logical"            "log"
 [7] "log10"                 "log1p"                 "log2"
[10] "logb"                  "Logic"                 "logical"
[13] "logLik"                "loglin"                "plogis"
[16] "qlogis"                "rlogis"                "SSlogis"
```

`example()` provides examples of how a function is used.

```
example(log)
```

```
log> log(exp(3))
[1] 3

log> log10(1e7) # = 7
[1] 7
```

```
log> x <- 10^-(1+2*1:9)

log> cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
         x
 [1,] 1e-03 9.995003e-04 9.995003e-04 1.000500e-03 1.000500e-03
 [2,] 1e-05 9.999950e-06 9.999950e-06 1.000005e-05 1.000005e-05
 [3,] 1e-07 1.000000e-07 1.000000e-07 1.000000e-07 1.000000e-07
 [4,] 1e-09 1.000000e-09 1.000000e-09 1.000000e-09 1.000000e-09
 [5,] 1e-11 1.000000e-11 1.000000e-11 1.000000e-11 1.000000e-11
 [6,] 1e-13 9.992007e-14 1.000000e-13 9.992007e-14 1.000000e-13
 [7,] 1e-15 1.110223e-15 1.000000e-15 1.110223e-15 1.000000e-15
 [8,] 1e-17 0.000000e+00 1.000000e-17 0.000000e+00 1.000000e-17
 [9,] 1e-19 0.000000e+00 1.000000e-19 0.000000e+00 1.000000e-19
```

The `vignette()` function gives longer, more detailed documentation about libraries. Not all libraries include vignettes, but for those that do it's usually a good place to get started. For example, the stringr package (which we installed above) includes a vignette. To read it's vignette, type the following at the console

```
vignette("stringr")
```

4 R Markdown and R Notebooks

RStudio comes with a useful set of tools, collectively called R Markdown, for generating "literate" statistical analyses. The idea behind literate statistical computing is that we should try to carry out our analyses in a manner that is transparent, self-explanatory, and reproducible. Literate statistical computing helps to ensure your research is reproducible because:

1. The steps of your analyses are explicitly described, both as written text and the code and function calls used.
2. Analyses can be more easily checked for correctness and reproduced from your literate code.
3. Your literate code can serve as a template for future analyses, saving you time and the trouble of remembering all the gory details.

As we'll see, R Markdown will allow us to produce statistical documents that integrate prose, code, figures, and nicely formatted mathematics so that we can share and explain our analyses to others. Sometimes those "others" are advisors, supervisors, or collaborators; sometimes the "other" is you six months from now. For the purposes of this class, you will be asked to complete problem sets in the form of R Markdown documents.

R Markdown documents are written in a light-weight markup language called Markdown. Markdown provides simple plain text "formatting" commands for specifying the structured elements of a document. Markdown was invented as a lightweight markup language for creating web pages and blogs, and has been adopted to a variety of different purposes. This chaptern provides a brief introduction to the capabilities of R Markdown. For more complete details, including lots of examples, see the R Markdown Website.

4.1 R Notebooks

We're going to create a type of R Markdown document called an "R Notebook". The R Notebook Documentation describes R Notebooks as so: "An R Notebook is an R Markdown document with code chunks that can be executed independently and interactively, with output visible immediately beneath the input."

4.2 Creating an R Notebook

To create an R Notebook select `File > New File > R Notebook` from the files menu in RStudio.

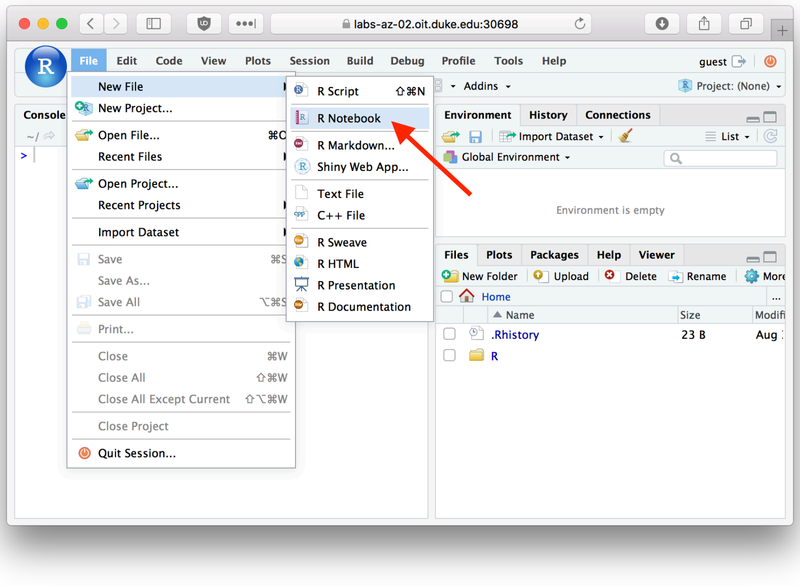


Figure 4.1: Using the File menu to create a new R Notebook.

4.3 The default R Notebook template

The standard template that RStudio creates for you includes a header section like the following where you can specify document properties such as the title, author, and change the look and feel of the generated HTML document.

```
---
title: "R Notebook"
output: html_notebook
---
```

The header is followed by several example sections that illustrate a few of the capabilities of R Markdown. Delete these and replace them with your own code as necessary.

4.4 Code and Non-code blocks

R Markdown documents are divided into code blocks (also called “chunks”) and non-code blocks. Code blocks are sets of R commands that will be evalauted when the R Markdown document is run or “knitted” (see below). Non-code blocks include explanatory text, embedded images, etc. The default notebook template includes both code and non-code blocks.

4.4.1 Non-code blocks

The first bit of text in the default notebook template is a non-code block that tells you how to use the notebook:

```
This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook.
When you execute code within the notebook, the results appear
beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk
or by placing your cursor inside it and pressing *Cmd+Shift+Enter*.
```

The text of non-code blocks can include lightweight markup information that can be used to format HTML or PDF output generated from the R Markdown document. Here are some examples:

```
# Simple textual formatting

This is a paragraph with plain text.  Nothing fancy will happen here.

This is a second paragraph with *italic*, **bold**, and `verbatim` text.

# Lists

## Bullet points lists

This is a list with bullet points:

  * Item a
  * Item b
  * Item c

## Numbered lists
```

```
This is a numbered list:

  1. Item 1
  #. Item 2
  #. Item 3

## Mathematics

R Markdown supports mathematical equations, formatted according to LaTeX
conventions. Dollar signs ($) are used to offset mathematics
like so:  $x^2 + y^2 = z^2$.
```

Notice from the example above that R Markdown supports LaTeX style formatting of mathematical equations. For example, `$x^2 + y^2 = z^2$` appears as $x^2 + y^2 = z^2$.

4.4.2 Code blocks

Code blocks are delimited by matching sets of three backward ticks ("‘). Everything within a code block is interpretted as an R command and is evaluated by the R interpretter. Here's the first code block in the default notebook template:

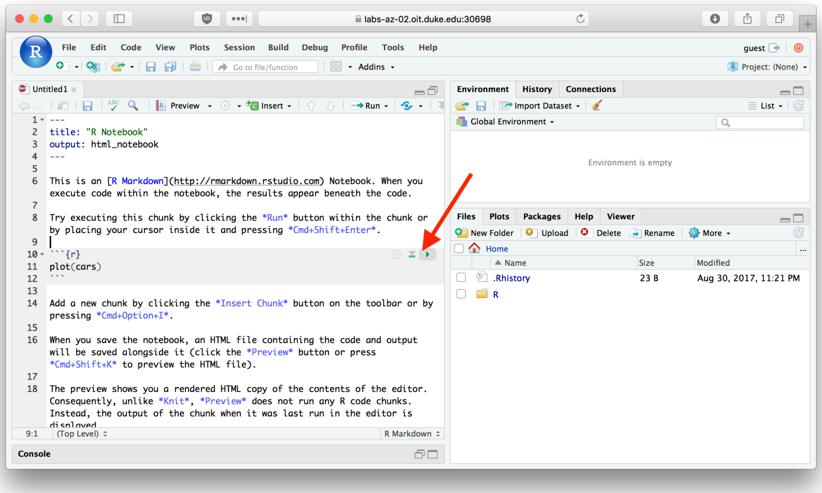
~~~
```
{r}
plot(cars)
```
~~~

4.5 Running a code chunk

You can run a single code block by clicking the small green "Run" button in the upper right hand corner of the code block as shown in the image below.

If you click this button the commands within this code block are executed, and any generated output is shown below the code block.

Try running the first code block in the default template now. After the code chunk is executed you should see a plot embedded in your R Notebook as shown below:



~~~~
---
title: "R Notebook"
output: html_notebook
---

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you
execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or
by placing your cursor inside it and pressing *Cmd+Shift+Enter*.

```{r}
plot(cars)
```

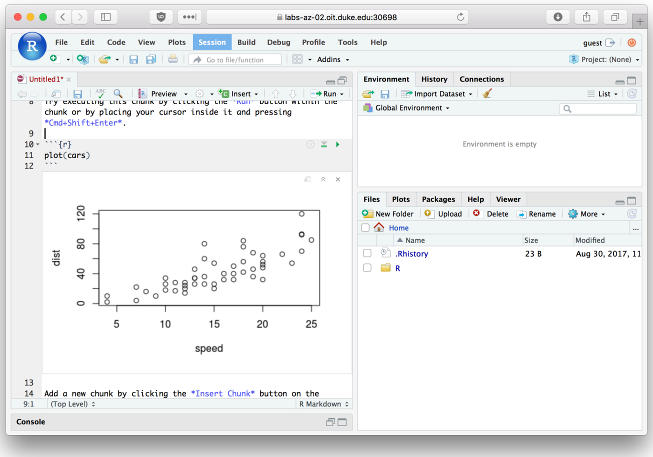
Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by
pressing *Cmd+Option+I*.

When you save the notebook, an HTML file containing the code and output
will be saved alongside it (click the *Preview* button or press
*Cmd+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor.
Consequently, unlike *Knit*, *Preview* does not run any R code chunks.
Instead, the output of the chunk when it was last run in the editor is
displayed.
~~~~

Name	Size	Modified
.Rhistory	23 B	Aug 30, 2017, 11:21 PM
R		

Figure 4.2: Click the Run button to execute a code chunk.



~~~~
Try executing this chunk by clicking the *Run* button within the
chunk or by placing your cursor inside it and pressing
*Cmd+Shift+Enter*.

```{r}
plot(cars)
```

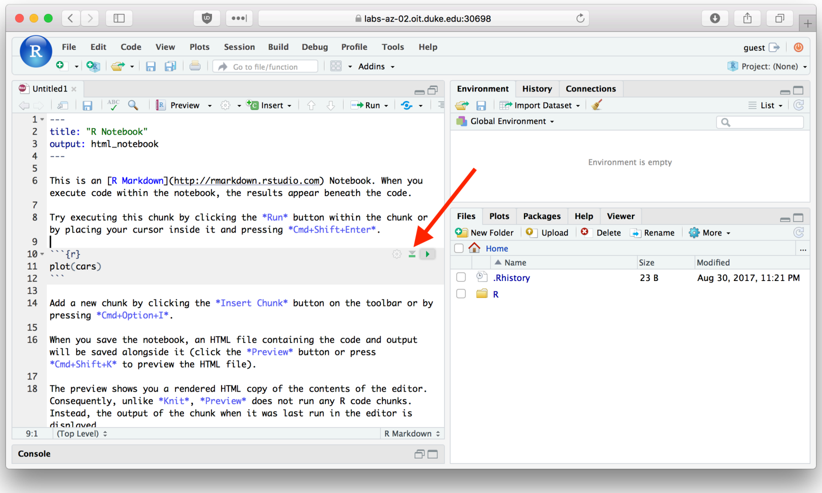
Add a new chunk by clicking the *Insert Chunk* button on the
~~~~

Name	Size	Modified
.Rhistory	23 B	Aug 30, 2017, 11
R		

Figure 4.3: An R Notebook showing an embedded plot after executing a code chunk.

4.6 Running all code chunks above

Next to the "Run" button in each code chunk is a button for "Run all chunks above" (see figure below). This is useful when the code chunk you're working on depends on calculations in earlier code chunks, and you want to evaluated those earlier code chunks prior to running the focal code chunk.



~~~
---
title: "R Notebook"
output: html_notebook
---

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you
execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or
by placing your cursor inside it and pressing *Cmd+Shift+Enter*.

```{r}
plot(cars)
```

Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by
pressing *Cmd+Option+I*.

When you save the notebook, an HTML file containing the code and output
will be saved alongside it (click the *Preview* button or press
*Cmd+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor.
Consequently, unlike *Knit*, *Preview* does not run any R code chunks.
Instead, the output of the chunk when it was last run in the editor is
displayed
~~~

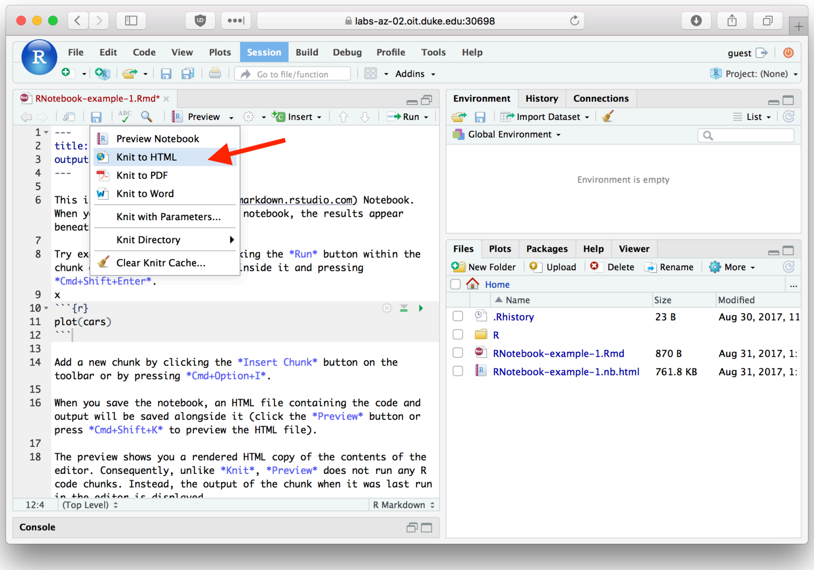
Figure 4.4: Use the 'Run all chunks above' button to evaluate all previous code chunks.

4.7 "Knitting" R Markdown to HTML

Save your R Notebook as `first_rnotebook.Rmd` (RStudio will automatically add the `.Rmd` extension so you don't need to type it). You can generate an HTML version of your notebook by clicking the "Preview" menu on the Notebook taskbar and then choosing "Knit to HTML" (see image below).

When an RMarkdown document is "knit", all of the code and non-code blocks are executed in a "clean" environment, in order from top to bottom. An output file is generated (HTML or one of the other available output types) that shows the results of executing the notebook. By default RStudio will pop-up a window showing you the HTML output you generated.

Knitting a document is a good way to make sure your analysis is reproducible. If your code compiles correctly when the document is knit, and produces the expected output, there's a good chance that someone else will be able to reproduce your analyses independently starting with your R Notebook document (after accounting for differences in file locations).



~~~
---
title:
output
---

This i                                   markdown.rstudio.com) Notebook.
When y                                   notebook, the results appear
beneat

Try ex                                   king the *Run* button within the
chunk                                    inside it and pressing
*Cmd+Shift+Enter*.
x
```{r}
plot(cars)
```


Add a new chunk by clicking the *Insert Chunk* button on the
toolbar or by pressing *Cmd+Option+I*.

When you save the notebook, an HTML file containing the code and
output will be saved alongside it (click the *Preview* button or
press *Cmd+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the
editor. Consequently, unlike *Knit*, *Preview* does not run any R
code chunks. Instead, the output of the chunk when it was last run
in the editor is displayed
~~~

Name	Size	Modified
.Rhistory	23 B	Aug 30, 2017, 11
R		
RNotebook-example-1.Rmd	870 B	Aug 31, 2017, 1:
RNotebook-example-1.nb.html	761.8 KB	Aug 31, 2017, 1:

Figure 4.5: Use the 'Knit to HTML' menu to generate HTML output from your R Notebook

4.8 Sharing your reproducible R Notebook

To share your R Notebook with someone else you just need to send them the source R Markdown file (i.e. the file with the `.Rmd` extension). Assuming they have access to the same source data, another user should be able to open the notebook file in RStudio and regenerate your analyses by evaluating the individual code chunks or knitting the document.

In this course you will be submitting homework assignments in the form of R Notebook markdown files.

5 Introduction to `dplyr`

In today's class we introduce a new package, `dplyr`, which, along with ggplot2 will be used in almost every class session. We will also introduce the `readr` package, for reading tabular data.

5.1 Libraries

Both `readr` and `dplyr` are members of the tidyverse, so a single invocation of `library()` makes the functions defined in these two packages available for our use:

```r
library(tidyverse)
```

5.2 Reading data with the `readr` package

The `readr` package defines a number of functions for reading data tables from common file formats like Comma-Separated-Value (CSV) and Tab-Separated-Value (TSV) files.

The two most frequently used `readr` functions we'll use in this class are `read_csv()` and `read_tsv()` for reading CSV and TSV files respectively. There are some variants of these basic function, which you can read about by invoking the help system (`?read_csv`).

5.2.1 Reading Excel files

The tidyverse also includes a package called `readxl` which can be used to read Excel spreadsheets (recent versions with `.xls` and `.xlsx` extensions). Excel files are somewhat more complicated to deal with because they can include separate "sheets". We won't use `readxl` in this class, but documentation and examples of how `readxl` is used can be found at the page linked above.

5.2.2 Example data: NC Births

For today's hands on session we'll use a data set that contains information on 150 cases of mothers and their newborns in North Carolina in 2004. This data set is available at the following URL:

- https://github.com/Bio723-class/example-datasets/raw/master/nc-births.txt

The births data is a TSV file, so we'll use the `read_tsv()` function to read it:

```r
births <- read_tsv("https://github.com/Bio723-class/example-datasets/raw/master/nc-births.
```

```
Rows: 150 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: "\t"
chr (3): premature, sexBaby, smoke
dbl (6): fAge, mAge, weeks, visits, gained, weight

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Notice that when you used `read_tsv()` the function printed information about how it "parsed" the data (i.e. the types it assigned to each of the columns).

The variables in the data set are:

- father's age (`fAge`),
- mother's age (`mAge`),

- weeks of gestation (`weeks`)
- whether the birth was premature or full term (`premature`)
- number of OB/GYN visits (`visits`)
- mother's weight gained in pounds (`gained`)
- babies birth weight (`weight`)
- sex of the baby (`sexBaby`)
- whether the mother was a smoker (`smoke`).

Notice too that we read the TSV file directly from a remote location via a URL. If instead, you wanted to load a local file on your computer you would specify the "path" – i.e. the location on your hard drive where you stored the file. For example, here is how I would load the same file if it was stored in the Downloads directory on my Mac laptop:

```r
# load the data from a local file
births <- read_tsv("/Users/pmagwene/Downloads/nc-births.txt")
```

5.3 A note on "tibbles"

You may have noticed that most of the functions defined in tidyverse related packages return not data frames, but rather something called a "tibble". You can think about tibbles as light-weight data frames. In fact if you ask about the "class" of a tibble you'll see that it includes `data.frame` as one of it's classes as well as `tbl` and `tbl_df`.

```r
class(births)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

There are some minor differences between data frame and tibbles. For example, tibbles print differently in the console and don't automatically change variable names and types in the same way that standard data frames do. Usually tibbles can be used wherever a standard data frame is expected, but you may occasionally find a function that only works with a standard data frame. It's easy to convert a tibble to a standard data frame using the `as.data.frame` function:

```r
births.std.df <- as.data.frame(births)
```

For more details about tibbles, see the Tibbles chapter in R for Data Analysis.

5.4 Data filtering and transformation with `dplyr`

`dplyr` is powerful tool for data filter and transformation. In the same way that `ggplot2` attempts to provide a "grammar of graphics", `dplyr` aims to provide a "grammar of data manipulation". In today's material we will see how `dplyr` complements and simplifies standard data frame indexing and subsetting operations. However, `dplyr` is focused only on data frames and doesn't completely replace the basic subsetting operations, and so being adept with both `dplyr` and the indexing approaches we've seen previously is important. If you're curious about the name "dplyr", the package's originator Hadley Wickham says it's supposed to invoke the idea of pliers for data frames (Github: Meaning of dplyrs name)

5.5 dplyr's "verbs"

The primary functions in the `dplyr` package can be thought of as a set of "verbs", each verb corresponding to a common data manipulation task. Some of the most frequently used verbs/functions in `dplyr` include:

- `select` – select columns
- `filter` – filter rows
- `mutate` – create new columns
- `arrange`– reorder rows
- `summarize` – summarize values
- `group_by` – split data frame on some grouping variable. Can be powerfully combined with `summarize`

All of these functions return new data frames rather than modifying the existing data frame (though some of the functions support in place modification of data frames via optional arguments).We illustrate these below by example using the NC births data.

5.5.1 `select`

The `select` function subsets the columns (variables) of a data frame. For example, to select just the weeks and weight columns from the births data set we could do:

```
# note I'm prefixing select with the package name (dplyr)
# to avoid name clashes with built-in select function
wks.weight <- dplyr::select(births, weeks, weight)
dim(wks.weight)  # dim should be 50 x 2
```

```
[1] 150   2
```

```
head(wks.weight)
```

```
# A tibble: 6 x 2
  weeks weight
  <dbl>  <dbl>
1    39   6.88
2    39   7.69
3    40   8.88
4    40   9
5    40   7.94
6    40   8.25
```

The equivalent using standard indexing would be:

```
wks.wt.alt <- births[c("weeks", "weight")]
dim(wks.wt.alt)
```

```
[1] 150   2
```

```
head(wks.wt.alt)
```

```
# A tibble: 6 x 2
  weeks weight
  <dbl>  <dbl>
1    39   6.88
2    39   7.69
3    40   8.88
4    40   9
5    40   7.94
6    40   8.25
```

Notes:
* The first argument to all of the `dplyr` functions is the data frame you're operating on

- When using functions defined in `dplyr` and `ggplot2` variable names are (usually) not quoted or used with the `$` operator. This is a design feature of these libraries and makes it easier to carry out interactive analyes because it saves a fair amount of typing.

5.5.2 `filter`

The `filter` function returns those rows of the data set that meet the given logical criterion.

For example, to get all the premature babies in the data set we could use filter as so:

```
premies <- filter(births, premature == "premie")
dim(premies)
```

```
[1] 21  9
```

The equivalent using standard indexing would be:

```
premies.alt <- births[births$premature == "premie",]
```

The `filter` function will work with more than one logical argument, and these are joined together using Boolean AND logic (i.e. intersection). For example, to find those babies that were premature *and* whose mothers were smokers we could do:

```
smoking.premies <- filter(births, premature == "premie", smoke == "smoker")
```

The equivalent call using standard indexing is:

```
# don't forget the trailing comma to indicate rows!
smoking.premies.alt <- births[(births$premature == "premie") & (births$smoke == "smoker"),
```

`filter` also accepts logical statements chained together using the standard Boolean operators. For example, to find babies who were premature *or* whose moms were older than 35 you could use the OR operator `|`:

```
premies.or.oldmom <- filter(births, premature == "premie" | fAge > 35)
```

5.5.3 `mutate`

The `mutate` function creates a new data frame that is the same as input data frame but with additional variables (columns) as specified by the function arguments. In the example below, I create two new variables, `weight.in.kg` and a `mom.smoked`:

```
# to make code more readable it's sometime useful to spread out
# function arguments over multiple lines like I've done here
births.plus <- mutate(births,
                      weight.in.kg = weight / 2.2,
                      mom.smoked = (smoke == "smoker"))

head(births.plus)
```

```
# A tibble: 6 x 11
   fAge  mAge weeks premature visits gained weight sexBaby smoke weigh~1 mom.s~2
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>   <dbl> <lgl>
1    31    30    39 full term     13      1   6.88 male    smok~    3.13 TRUE
2    34    36    39 full term      5     35   7.69 male    nons~    3.50 FALSE
3    36    35    40 full term     12     29   8.88 male    nons~    4.04 FALSE
4    41    40    40 full term     13     30   9    female  nons~    4.09 FALSE
```

```
5    42    37    40 full term     NA     10    7.94 male    nons~    3.61 FALSE
6    37    28    40 full term     12     35    8.25 male    smok~    3.75 TRUE
# ... with abbreviated variable names 1: weight.in.kg, 2: mom.smoked
```

The equivalent using standard indexing would be to create a new data frame from `births`, appending the new variables to the end as so:

```
births.plus.alt <- data.frame(births,
                              weight.in.kg = births$weight / 2.2,
                              mom.smoked = (births$smoke == "smoker"))
```

5.5.4 `arrange`

Arrange creates a new data frame where the rows are sorted according to their values for one or more variables. For example, to sort by mothers age we could do:

```
young.moms.first <- arrange(births, mAge)
head(young.moms.first)
```

```
# A tibble: 6 x 9
   fAge  mAge weeks premature visits gained weight sexBaby smoke
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
1    18    15    37 full term     12     76   8.44 male    nonsmoker
2    NA    16    40 full term      4     12   6    female  nonsmoker
3    21    16    38 full term     15     75   7.56 female  smoker
4    26    17    38 full term     11     30   9.5  female  nonsmoker
5    17    17    29 premie         4     10   2.63 female  nonsmoker
6    20    17    40 full term     17     38   7.19 male    nonsmoker
```

The equivalent to `arrange` using standard indexing would be to use the information returned by the `order` function:

```
young.moms.first.alt <- births[order(births$mAge),]
head(young.moms.first.alt)
```

```
# A tibble: 6 x 9
   fAge  mAge weeks premature visits gained weight sexBaby smoke
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
1    18    15    37 full term     12     76   8.44 male    nonsmoker
```

```
2    NA    16    40 full term      4     12   6    female  nonsmoker
3    21    16    38 full term     15     75   7.56 female  smoker
4    26    17    38 full term     11     30   9.5  female  nonsmoker
5    17    17    29 premie         4     10   2.63 female  nonsmoker
6    20    17    40 full term     17     38   7.19 male    nonsmoker
```

When using `arrange`, multiple sorting variables can be specified:

```
sorted.by.moms.and.dads <- arrange(births, mAge, fAge)
head(sorted.by.moms.and.dads)
```

```
# A tibble: 6 x 9
   fAge  mAge weeks premature visits gained weight sexBaby smoke
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
1    18    15    37 full term     12     76   8.44 male    nonsmoker
2    21    16    38 full term     15     75   7.56 female  smoker
3    NA    16    40 full term      4     12   6    female  nonsmoker
4    17    17    29 premie         4     10   2.63 female  nonsmoker
5    20    17    40 full term     17     38   7.19 male    nonsmoker
6    26    17    38 full term     11     30   9.5  female  nonsmoker
```

If you want to sort in descending order, you can combing `arrange` with the `desc` (=descend) function, also defined in `dplyr`:

```
old.moms.first <- arrange(births, desc(mAge))
head(old.moms.first)
```

```
# A tibble: 6 x 9
   fAge  mAge weeks premature visits gained weight sexBaby smoke
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
1    NA    41    33 premie        13      0   5.69 female  nonsmoker
2    41    40    40 full term     13     30   9    female  nonsmoker
3    33    40    36 premie        13     23   7.81 female  nonsmoker
4    40    40    38 full term     13     38   7.31 male    nonsmoker
5    46    39    38 full term     10     35   6.75 male    smoker
6    NA    38    32 premie        10     16   2.19 female  smoker
```

5.5.5 `summarize`

`summarize` applies a function of interest to one or more variables in a data frame, reducing a vector of values to a single value and returning the results in a data frame. This is most often used to calculate statistics like means, medians, count, etc. As we'll see below, this is powerful when combined with the `group_by` function.

```
summarize(births,
          mean.wt = mean(weight),
          median.wks = median(weeks))
```

```
# A tibble: 1 x 2
  mean.wt median.wks
    <dbl>      <dbl>
1    7.05         39
```

You'll need to be diligent if your data has missing values (NAs). For example, by default the `mean` function returns NA if any of the input values are NA:

```
summarize(births,
          mean.gained = mean(gained))
```

```
# A tibble: 1 x 1
  mean.gained
        <dbl>
1          NA
```

However, if you read the `mean` docs (`?mean`) you'll see that there is an `na.rm` argument that indicates whether NA values should be removed before computing the mean. This is what we want so we instead call summarize as follows:

```
summarize(births,
          mean.gained = mean(gained, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  mean.gained
        <dbl>
1        32.5
```

5.5.6 `group_by`

The `group_by` function implicitly adds grouping information to a data frame.

```r
# group the births by whether mom smoked or not
by_smoking <- group_by(births, smoke)
```

The object returned by `group_by` is a "grouped data frame":

```r
class(by_smoking)
```

```
[1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

Some functions, like `count()` and `summarize()` (see below) know how to use the grouping information. For example, to count the number of births conditional on mother smoking status we could do:

```r
count(by_smoking)
```

```
# A tibble: 2 x 2
# Groups:   smoke [2]
  smoke         n
  <chr>     <int>
1 nonsmoker   100
2 smoker       50
```

`group_by` also works with multiple grouping variables, with each added grouping variable specified as an additional argument:

```r
by_smoking.and.mAge <- group_by(births, smoke, mAge > 35)
```

5.5.7 Combining grouping and summarizing

Grouped data frames can be combined with the `summarize` function we saw above. For example, if we wanted to calculate mean birth weight, broken down by whether the baby's mother smoked or not we could call `summarize` with our `by_smoking` grouped data frame:

```r
summarize(by_smoking, mean.wt = mean(weight))
```

```
# A tibble: 2 x 2
  smoke     mean.wt
  <chr>       <dbl>
```

```
1 nonsmoker    7.18
2 smoker       6.78
```

Similarly to get the mean birth weight of children conditioned on mothers smoking status and age:

```
summarize(by_smoking.and.mAge, mean(weight))
```

```
`summarise()` has grouped output by 'smoke'. You can override using the
`.groups` argument.
```

```
# A tibble: 4 x 3
# Groups:   smoke [2]
  smoke     `mAge > 35` `mean(weight)`
  <chr>     <lgl>                <dbl>
1 nonsmoker FALSE                 7.17
2 nonsmoker TRUE                  7.26
3 smoker    FALSE                 6.83
4 smoker    TRUE                  6.30
```

5.5.8 Scoped variants of `mutate` and `summarize`

Both the `mutate()` and `summarize()` functions provide "scoped" alternatives, that allow us to apply the operation on a selection of variables. These variants are often used in combination with grouping. We'll look at the `summarize` versions – `summarize_all()`, `summarize_at()`, and `summarize_if()`. See the documentation (`?mutate_all`) for descriptions of the `mutate` versions.

5.5.8.1 `summarize_all()`

`summarize_all()` applies a one or more functions to all columns in a data frame. Here we illustrate a simple version of this with the iris data:

```
# group by species
by_species <- group_by(iris, Species)
# calculate the mean of every variable, grouped by species
summarize_all(by_species, mean)
```

```
# A tibble: 3 x 5
  Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
  <fct>             <dbl>       <dbl>        <dbl>       <dbl>
1 setosa             5.01        3.43         1.46       0.246
2 versicolor         5.94        2.77         4.26       1.33
3 virginica          6.59        2.97         5.55       2.03
```

Note that if we try and apply `summarize_all()` in the same way to the grouped data frame `by_smoking` we'll get a bunch of warning messages:

```
summarize_all(by_smoking, mean)
```

```
# A tibble: 2 x 9
  smoke      fAge  mAge weeks premature visits gained weight sexBaby
  <chr>     <dbl> <dbl> <dbl>     <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
1 nonsmoker    NA  26.9  38.6        NA   NA       NA   7.18      NA
2 smoker       NA  26    38.5        NA   10.8     NA   6.78      NA
```

Here's an example of one of these warnings:

```
Warning messages:
1: In mean.default(premature) :
  argument is not numeric or logical: returning NA
```

This message is telling us that we can't apply the `mean()` function to the data frame column `premature` because this is not a numerical or logical vector. Despite this and the other similar warnings, `summarize_all()` does return a result, but the means for any non-numeric values are replaced with NAs, as shown below:

```
# A tibble: 2 x 9
  smoke      fAge  mAge weeks premature visits gained weight sexBaby
  <chr>     <dbl> <dbl> <dbl>     <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
1 nonsmoker    NA  26.9  38.6        NA   NA       NA   7.18      NA
2 smoker       NA  26.0  38.5        NA   10.8     NA   6.78      NA
```

If you examine the output above, you'll see that there are several variables that are numeric, however we still got NAs when we calculated the grouped means. This is because those variables contain NA values. The `mean` function has an optional argument, `na.rm`, which tells the function to remove any missing data before calculating the mean. Thus we can modify our call to `summarize_all` as follows:

```r
# calculate mean of all variables, grouped by smoking status
summarize_all(by_smoking, mean, na.rm = TRUE)
```

```
Warning in mean.default(premature, na.rm = TRUE): argument is not numeric or
logical: returning NA

Warning in mean.default(premature, na.rm = TRUE): argument is not numeric or
logical: returning NA

Warning in mean.default(sexBaby, na.rm = TRUE): argument is not numeric or
logical: returning NA

Warning in mean.default(sexBaby, na.rm = TRUE): argument is not numeric or
logical: returning NA
```

```
# A tibble: 2 x 9
  smoke      fAge  mAge weeks premature visits gained weight sexBaby
  <chr>     <dbl> <dbl> <dbl>     <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
1 nonsmoker  29.8  26.9  38.6        NA   11.9   32.5   7.18      NA
2 smoker     29.7  26    38.5        NA   10.8   32.3   6.78      NA
```

Note that the non-numeric data columns still lead to NA values.

5.5.8.2 summarize_if()

`summarize_if()` is similar to `summarize_all()`, except it only applies the function of interest to those variables that match a particular predicate (i.e. are TRUE for a particular TRUE/FALSE test).

Here we use `summarize_if()` to apply the `mean()` function to only those variables (columns) that are numeric.

```r
# calculate mean of all numeric variables, grouped by smoking status
summarize_if(by_smoking, is.numeric, mean, na.rm = TRUE)
```

```
# A tibble: 2 x 7
  smoke      fAge  mAge weeks visits gained weight
  <chr>     <dbl> <dbl> <dbl>  <dbl>  <dbl>  <dbl>
1 nonsmoker  29.8  26.9  38.6   11.9   32.5   7.18
2 smoker     29.7  26    38.5   10.8   32.3   6.78
```

5.5.8.3 `summarize_at()`

`summarize_at()` allows us to apply functions of interest only to specific variables.

```r
# calculate mean of gained and weight variables, grouped by smoking status
summarize_at(by_smoking, c("gained", "weight"), mean, na.rm = TRUE)
```

```
# A tibble: 2 x 3
  smoke     gained weight
  <chr>      <dbl>  <dbl>
1 nonsmoker   32.5   7.18
2 smoker      32.3   6.78
```

All three of the scoped summarize functions can also be used to apply multiple functions, by wrapping the function names in a call to `dplyr::funs()`:

```r
# calculate mean and std deviation of
# gained and weight variables, grouped by smoking status
summarize_at(by_smoking, c("gained", "weight"), funs(mean, sd), na.rm = TRUE)
```

```
Warning: `funs()` was deprecated in dplyr 0.8.0.
i Please use a list of either functions or lambdas:

# Simple named list: list(mean = mean, median = median)

# Auto named with `tibble::lst()`: tibble::lst(mean, median)

# Using lambdas list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
```

```
# A tibble: 2 x 5
  smoke     gained_mean weight_mean gained_sd weight_sd
  <chr>           <dbl>       <dbl>     <dbl>     <dbl>
1 nonsmoker        32.5        7.18      15.2      1.43
2 smoker           32.3        6.78      16.6      1.60
```

`summarize_at()` accepts as the the argument for variables a character vector of column names, a numeric vector of column positions, or a list of columns generated by the `dplyr::vars()` function, which can be be used as so:

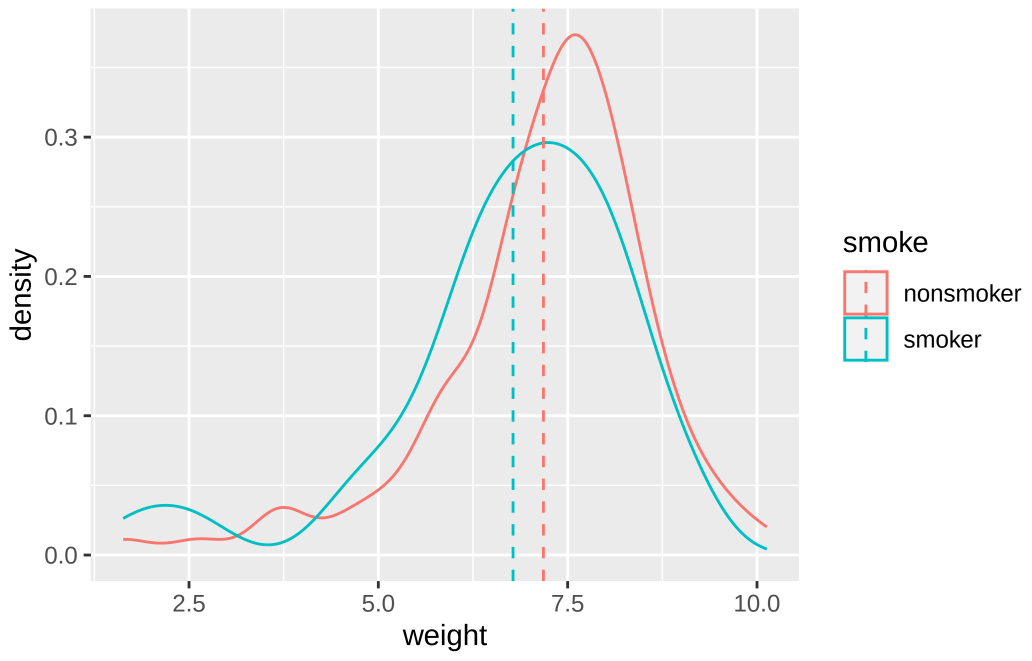
```
# reformatted to promote readability of arguments
summarize_at(by_smoking,
             vars(gained, weight),
             funs(mean, sd),
             na.rm = TRUE)
```

```
# A tibble: 2 x 5
  smoke     gained_mean weight_mean gained_sd weight_sd
  <chr>           <dbl>       <dbl>     <dbl>     <dbl>
1 nonsmoker        32.5        7.18      15.2      1.43
2 smoker           32.3        6.78      16.6      1.60
```

5.5.9 Combining summarize with grouping aesthetics in `ggplot2`

We've already seen an instance of grouping (conditioning) when we used aesthetics like color or fill to distinguish subgroups in different types of statistical graphics. Below is an example where we integrate information from a `group_by`/`summarize` operation into a plot:

```
# calculate mean weights, conditioned on smoking status
wt.by.smoking <-
  summarize(by_smoking, mean_weight = mean(weight, na.rm = TRUE))

# create density plot for all the data
# and then use geom_vline to draw vertical lines at the means for
# each group
ggplot(births) +
  geom_density(aes(x = weight, color = smoke)) + # data drawn from births
  geom_vline(data = wt.by.smoking,  # note use of different data frame!
             mapping = aes(xintercept = mean_weight, color = smoke),
             linetype = 'dashed')
```



5.6 Pipes

`dplyr` includes a very useful operator available called a pipe available to us. Pipes are powerful because they allow us to chain together sets of operations in a very intuitive fashion while minimizing nested function calls. We can think of pipes as taking the output of one function and feeding it as the *first argument* to another function call, where we've already specified the subsequent arguments.

Pipes are actually defined in another packaged called `magrittr`. We'll look at the basic pipe operator and then look at a few additional "special" pipes that `magrittr` provides.

5.6.1 Install and load `magrittr`

In magrittr in not already installed, install it via the command line or the RStudio GUI. Having done so, you will need to load magrittr via the `library()` function:

```
library(magrittr)
```

5.6.2 The basic pipe operator

The pipe operator is designated by `%>%`. Using pipes, the expression `x %>% f()` is equivalent to `f(x)` and the expression `x %>% f(y)` is equivalent to `f(x,y)`. The documentation on pipes (see `?magrittr`) uses the notation `lhs %>% rhs` where `lhs` and `rhs` are short for "left-hand side" and "right-hand side" respectively. I'll use this same notation in some of the explanations that follow.

```
births %>% head()   # same as head(births)
```

```
# A tibble: 6 x 9
   fAge  mAge weeks premature visits gained weight sexBaby smoke
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
1    31    30    39 full term     13      1   6.88 male    smoker
2    34    36    39 full term      5     35   7.69 male    nonsmoker
3    36    35    40 full term     12     29   8.88 male    nonsmoker
4    41    40    40 full term     13     30   9    female  nonsmoker
5    42    37    40 full term     NA     10   7.94 male    nonsmoker
6    37    28    40 full term     12     35   8.25 male    smoker
```

```
births %>% head     # you can even leave the parentheses out
```

```
# A tibble: 6 x 9
   fAge  mAge weeks premature visits gained weight sexBaby smoke
  <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
1    31    30    39 full term     13      1   6.88 male    smoker
2    34    36    39 full term      5     35   7.69 male    nonsmoker
3    36    35    40 full term     12     29   8.88 male    nonsmoker
4    41    40    40 full term     13     30   9    female  nonsmoker
5    42    37    40 full term     NA     10   7.94 male    nonsmoker
6    37    28    40 full term     12     35   8.25 male    smoker
```

```
births %>% head(10) # same as head(births, 10)
```

```
# A tibble: 10 x 9
    fAge  mAge weeks premature visits gained weight sexBaby smoke
   <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
 1    31    30    39 full term     13      1   6.88 male    smoker
 2    34    36    39 full term      5     35   7.69 male    nonsmoker
 3    36    35    40 full term     12     29   8.88 male    nonsmoker
 4    41    40    40 full term     13     30   9    female  nonsmoker
 5    42    37    40 full term     NA     10   7.94 male    nonsmoker
 6    37    28    40 full term     12     35   8.25 male    smoker
 7    35    35    28 premie         6     29   1.63 female  nonsmoker
 8    28    21    35 premie         9     15   5.5  female  smoker
 9    22    20    32 premie         5     40   2.69 male    smoker
10    36    25    40 full term     13     34   8.75 female  nonsmoker
```

Multiple pipes can be chained together, such that `x %>% f() %>% g() %>% h()` is equivalent to `h(g(f(x)))`.

```
# equivalent to: head(arrange(births, weight), 10)
births %>% arrange(weight) %>% head(10)
```

```
# A tibble: 10 x 9
    fAge  mAge weeks premature visits gained weight sexBaby smoke
   <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
 1    35    35    28 premie         6     29   1.63 female  nonsmoker
 2    NA    18    33 premie         7     40   1.69 male    smoker
 3    NA    38    32 premie        10     16   2.19 female  smoker
 4    17    17    29 premie         4     10   2.63 female  nonsmoker
 5    22    20    32 premie         5     40   2.69 male    smoker
 6    38    37    26 premie         5     25   3.63 male    nonsmoker
 7    25    22    34 premie        10     20   3.75 male    nonsmoker
 8    NA    24    38 full term     16     50   3.75 female  nonsmoker
 9    30    25    35 premie        15     40   4.5  male    smoker
10    19    20    34 premie        13      6   4.5  male    nonsmoker
```

When there are multiple piping operations, I like to arrange the statements vertically to help emphasize the flow of processing and to facilitate debugging and/or modification. I would usually rearrange the above code block as follows:

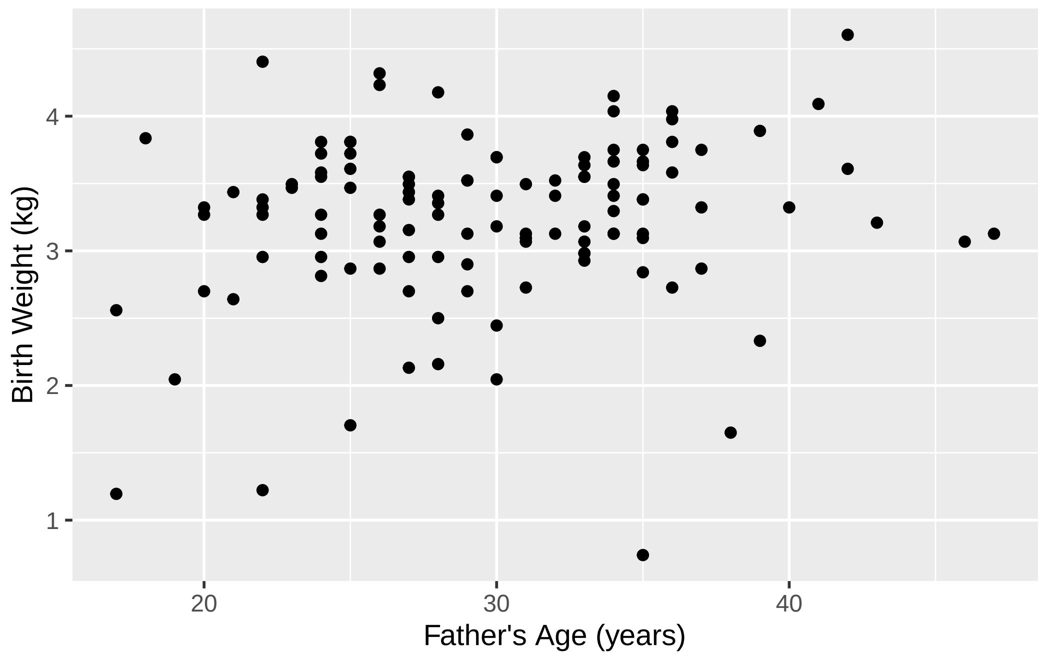
```
births %>%
  arrange(weight) %>%
  head(10)
```

```
# A tibble: 10 x 9
    fAge  mAge weeks premature visits gained weight sexBaby smoke
   <dbl> <dbl> <dbl> <chr>      <dbl>  <dbl>  <dbl> <chr>   <chr>
 1    35    35    28 premie         6     29   1.63 female  nonsmoker
 2     NA     18     33 premie          7      40    1.69 male     smoker
 3     NA     38     32 premie         10      16    2.19 female   smoker
 4     17     17     29 premie          4      10    2.63 female   nonsmoker
 5     22     20     32 premie          5      40    2.69 male     smoker
 6     38     37     26 premie          5      25    3.63 male     nonsmoker
 7     25     22     34 premie         10      20    3.75 male     nonsmoker
 8     NA     24     38 full term      16      50    3.75 female   nonsmoker
 9     30     25     35 premie         15      40    4.5  male     smoker
10     19     20     34 premie         13       6    4.5  male     nonsmoker
```

5.6.3 An example without pipes

To illustrate how pipes help us, first let's look at an example set of analysis steps without using pipes. Let's say we wanted to explore the relationship between father's age and baby's birth weight. We'll start this process of exploration by generating a bivariate scatter plot. Being good scientists we want to express our data in SI units, so we'll need to converts pounds to kilograms. You'll also recall that a number of the cases have missing data on father's age, so we'll want to remove those before we plot them. Here's how we might accomplish these steps:

```r
# add a new column for weight in kg
births.kg <- mutate(births, weight.kg = weight / 2.2)

# filter out the NA fathers
filtered.births <- filter(births.kg, !is.na(fAge))

# create our plot
ggplot(filtered.births, aes(x = fAge, y = weight.kg)) +
  geom_point() +
  labs(x = "Father's Age (years)", y = "Birth Weight (kg)")
```



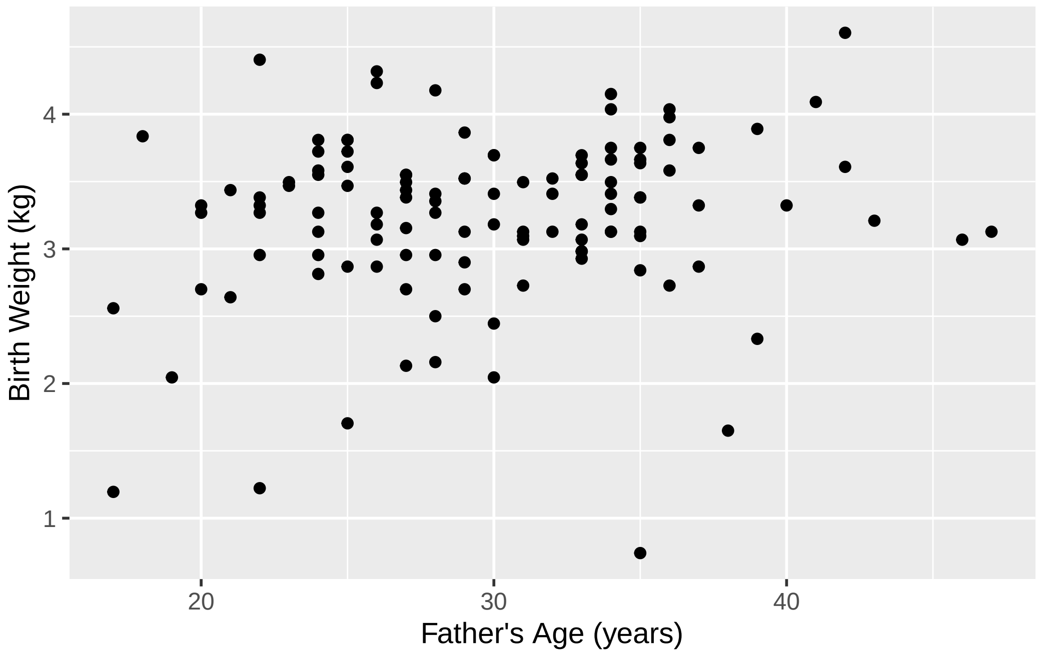
Notice that we created two "temporary" data frames along the way – `births.kg` and `filtered.births`. These probably aren't of particular interest to us, but we needed to generate them to build the plot we wanted. If you were particularly masochistic you could avoid these temporary data frames by using nested functions call like this:

```
# You SHOULD NOT write nested code like this.
# Code like this is hard to debug and understand!
ggplot(filter(mutate(births, weight.kg = weight / 2.2), !is.na(fAge)),
       aes(x = fAge, y = weight.kg)) +
  geom_point() +
  labs(x = "Father's Age (years)", y = "Birth Weight (kg)")
```

5.6.4 The same example using pipes

The pipe operator makes the output of one statement (`lhs`) as the first input of a following function (`rhs`). This simplifies the above example to:

```
births %>%
  mutate(weight.kg = weight / 2.2) %>%
  filter(!is.na(fAge)) %>%
  ggplot(aes(x = fAge, y = weight.kg)) +
    geom_point() +
    labs(x = "Father's Age (years)", y = "Birth Weight (kg)")
```



In the example above, we feed the data frame into the `mutate` function. `mutate` expects a data frame as a first argument, and subsequent arguments specify the new variables to be created. `births %>% mutate(weight.kg = weight / 2.2)` is thus equivalent to `mutate(births, weight.kg = weight / 2.2))`. We then pipe the output to `filter`, removing NA fathers, and then pipe that output as the input to ggplot.

As mentioned previously, it's good coding style to write each discrete step as its own line when using piping. This make it easier to understand what the steps of the analysis are as well as facilitating changes to the code (commenting out lines, adding lines, etc)

5.6.5 Assigning the output of a statement involving pipes to a variable

It's important to recognize that pipes are simply a convenient way to chain together a series of expression. Just like any other compound expression, the output of a series of pipe statements can be assigned to a variable, like so:

```
stats.old.moms <-
  births %>%
  filter(mAge > 35) %>%
  summarize(median.gestation = median(weeks),
            mean.weight = mean(weight))

stats.old.moms
```

```
# A tibble: 1 x 2
  median.gestation mean.weight
             <dbl>       <dbl>
1               38        6.94
```

Note that our summary table, `stats.old.moms`, is itself a data frame.

5.6.6 Compound assignment pipe operator

A fairly common operation when working interactively in R is to update an existing data frame. `magrittr` defines another pipe operator – `%<>%` – called the "compound assignment" pipe operator, to facilitate this. The compound assignment pipe operator has the basic usage `lhs %<>% rhs`. This operator evaluates the function on the `rhs` using the `lhs` as the first argument, and *then* updates the `lhs` with the resulting value. This is simply shorthand for writing `lhs <- lhs %>% rhs`.

```
stats.old.moms %<>%  # note compound pipe operator!
  mutate(mean.weight.kg = mean.weight / 2.2)
```

5.6.7 The dot operator with pipes

When working with pipes, sometimes you'll want to use the `lhs` in multiple places on the `rhs`, or as something other than the first argument to the `rhs`. `magrittr` provides for this situation by using the dot (`.`) operator as a placeholder. Using the dot operator, the expression `y %>% f(x, .)` is equivalent to `f(x,y)`.

```
c("dog", "cakes", "sauce", "house") %>%  # create a vector
  sample(1) %>% # pick a random single element of that vector
  str_c("hot", .)  # string concatenate the pick with the word "hot"
```

```
[1] "hotsauce"
```

5.6.8 The exposition pipe operator

`magrittr` defines another operator called the “exposition pipe operator”, designed `%$%`. This operator exposes the names in the `lhs` to the expression on the `rhs`.

Here is an example of using the exposition pipe operator to simply return the vector of weights:

```
births %>%
  filter(premature == "premie") %$%  # note the different pipe operator!
  weight
```

```
 [1] 1.63 5.50 2.69 6.50 7.81 4.75 3.75 2.19 6.81 4.69 6.75 4.50 5.94 4.50 5.06
[16] 5.69 1.69 6.31 2.63 5.88 3.63
```

If we wanted to calculate the minimum and maximum weight of premature babies in the data set we could do the following (though I’d usually prefer `summarize()` unless I needed the results in the form of a vector):

```
births %>%
  filter(mAge > 35) %$%  # note the different pipe operator!
  c(min(weight), max(weight))
```

```
[1]  2.19 10.13
```

6 Introduction to ggplot2

Pretty much any statistical plot can be thought of as a **mapping** between data and one or more visual representations. For example, in a scatter plot we map two ordered sets of numbers (the variables of interest) to points in the Cartesian plane (x,y-coordinates). The representation of data as points in a plane can be thought of as a type of **geometric mapping**. In a histogram, we divide the range of a variable of interest into bins, count the number of observations in each bin, and represent those counts as bars. The process of counting the data in bins is a type of **statistical transformation** (summing in this case), while the representation of the counts as bars is another example of a geometric mapping. Both types of plots can be further embellished with additional information, such as coloring the points or bars based on a categorical variable of interest, changing the shape of points, etc. These are examples of **aesthetic mappings**. An additional operation that is frequently useful is **faceting** (also called conditioning), in which a series of subplots are created to show particular subsets of the data.

The package `ggplot2` is based on a formalized approach for building statistical graphics as a combination of geometric mappings, aesthetic mappings, statistical transformations, and faceting (conditioning). In ggplot2, complex figures are built up by combining layers – where each layer includes a geometric mapping, an aesthetic mapping, and a statistical transformation – along with any desired faceting information.

Many of the key ideas behind ggplot2 (and its predecessor,"ggplot") are based on a book called "The Grammar of Graphics" (Leland Wilkinson, 1985). The "grammar of graphics" is the "gg" in the ggplot2 name.

6.1 Loading ggplot2

`ggplot2` is one of the packages included in the `tidyverse` meta-package we installed during the previous class session (see the previous lecture notes for instruction if you have not installed `tidyverse`). If we load the tidyverse package, ggplot2 is automatically loaded as well.

```r
library(tidyverse)
```

However if we wanted to we could load only ggplot2 as follows:

```r
library(ggplot2)  # not necessary if we already loaded tidyverse
```

6.2 Example data set: Anderson's Iris Data

To illustrate ggplot2 we'll use a dataset called `iris`. This data set was made famous by the statistician and geneticist R. A. Fisher who used it to illustrate many of the fundamental statistical methods he developed (Recall that Fisher was one of the key contributors to the modern synthesis in biology, reconciling evolution and genetics in the early 20th century). The data set consists of four morphometric measurements for specimens from three different iris species (*Iris setosa*, *I. versicolor*, and *I. virginica*). Use the R help to read about the iris data set (`?iris`). We'll be using this data set repeatedly in future weeks so familiarize yourself with it.

The iris data is included in a standard R package (`datasets`) that is made available automatically when you start up R. As a consequence we don't need to explicitly load the iris data from a file. Let's take a few minutes to explore this iris data set before we start generating plots:

```r
names(iris) # get the variable names in the dataset
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```r
dim(iris)   # dimensions given as rows, columns
```

```
[1] 150   5
```

```r
head(iris)  # can you figure out what the head function does?
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
tail(iris)  # what about the tail function?
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

6.3 Template for single layer plots in ggplot2

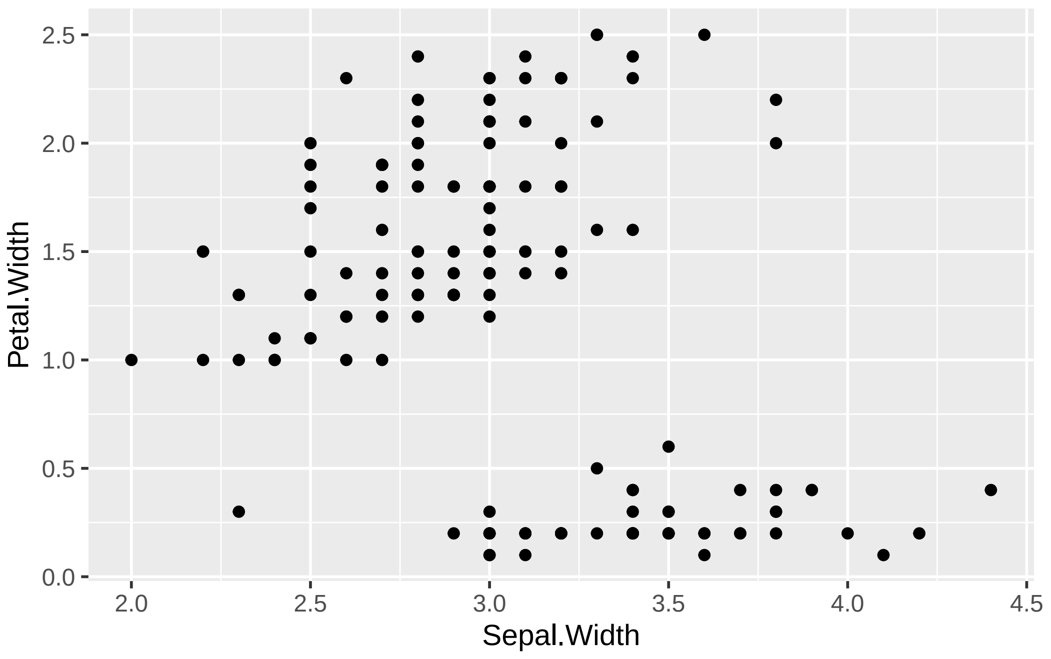
A basic template for building a single layer plot using `ggplot2` is shown below. When creating a plot, you need to replace the text in brackets (e.g. `<DATA>`) with appropriate objects, functions, or arguments:

```
# NOTE: this is pseudo-code. It will not run!

ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The base function `ggplot()` is responsible for creating the coordinate system in which the plot will be display. To this coordinate system we add a geometric mapping (called a "geom" for short) that specifies how data gets mapped into the coordinate system (e.g. points, bars, etc). Included as an input to the geom function is the aesthetic mapping function that specifies which variables to use in the geometric mapping (e.g. which variables to treat as the x- and y-coordinates), colors, etc.

For example, using this template we can create a scatter plot that show the relationship between the variables `Sepal.Width` and `Petal.Width`. To do so we subsitute `iris` for `<DATA>`, `geom_point` for `<GEOM_FUNCTION>`, and `x = Sepal.Width` and `y = Petal.Width` for `<MAPPINGS>`.

```
ggplot(data = iris) +
  geom_point(mapping = aes(x = Sepal.Width,  y = Petal.Width))
```



If we were to translate this code block to English, we might write it as "Using the iris data frame as the source of data, create a point plot using each observeration's Sepal.Width variable for the x-coordinate and the Petal.Width variable for the y-coordinate."

6.4 An aside about function arguments

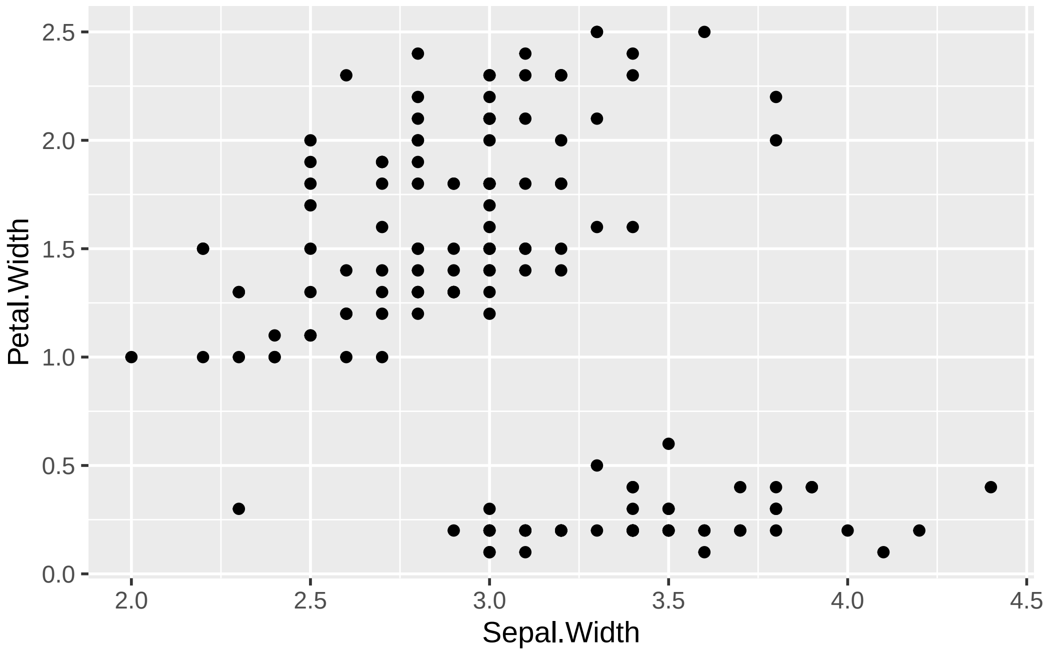
The inputs to a function are also known as "arguments". In R, when you call a function you can specify the arguments by keyword (i.e. using names specified in the function definition) or by position (i.e. the order of the inputs).

In our bar plot above, we're using using keyword arguments. For example, in the line `ggplot(data = iris)`, iris is treated as the "data" argument. Similarly, in the second line, `aes(x = Sepal.Width, y = Petal.Width)` is the "mapping" argument to `geom_bar`. Note that `aes` is itself a function (see `?aes`) that takes arguments that can be specified positionally or with keywords.

If we wanted to, we could instead use position arguments when calling a function, by passing inputs to the function corresponding to the order they are specified in the function definition. For example, take a minute to read the documentation for the `ggplot` function (`?ggplot`). Near the top of the help page you'll see a description of how the function is called under "Usage". Reading the Usage section you'll see that the the "data" argument is the first positional argument to `ggplot`. Similarly, if you read the docs for the `geom_point` function you'll see that mapping is the first positional argument for that function.

The equivalent of our previous example, but now using positional arguments is:

```
ggplot(iris) +   # note we dropped the "data = " part
  # note we dropped the "mapping = " part from the geom_point call
  geom_point(aes(x = Sepal.Width,  y = Petal.Width))
```



The upside of using positional arguments is that it means less typing, which is useful when working interactively at the console (or in an R Notebok). The downside to using positional arguments is you need to remember or lookup the order of the arguments. Using positional arguments can also make your code less "self documenting" in the sense that it is less explicit about how the inputs are being treated. While the argument "x" is the first argument to the `aes` function, I chose to explicitly include the argument name to make it clear what variable I'm plotting on the x-axis.

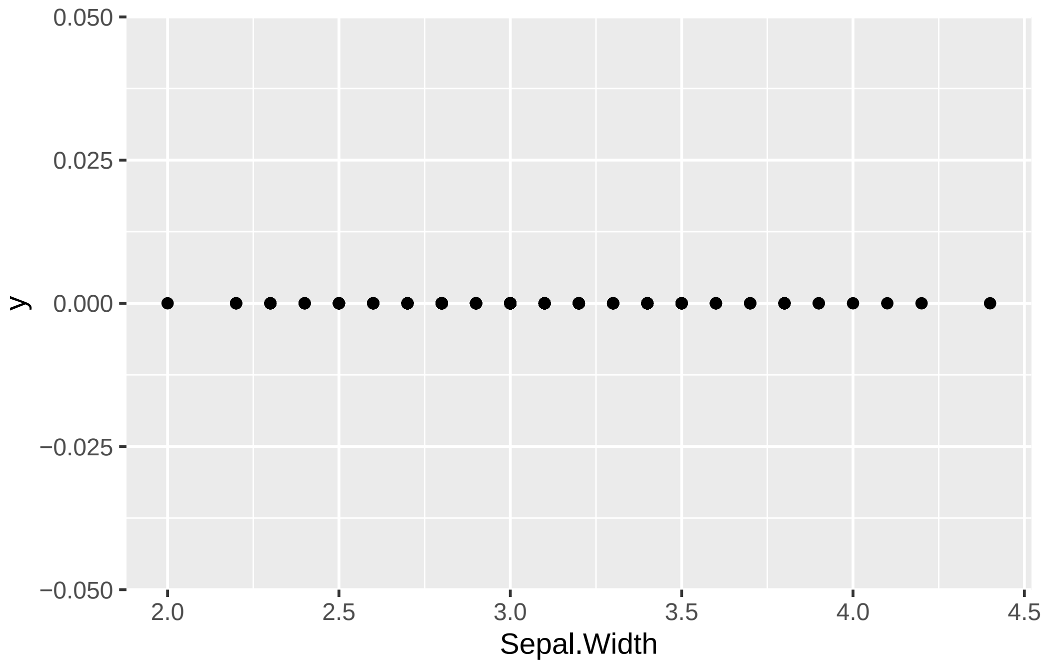
We will cover function arguments in greater detail a class session or two from now, when we learn how to write our own functions.

6.5 Strip plots

One of the simplest visualizations of a continuous variable is to draw points along a number line, where each point represent the value of one of the observations. This is sometimes called a "strip plot".

First, we'll use the `geom_point` function as shown below to generate a strip plot for the `Sepal.Width` variable in the iris data set.

```r
ggplot(data = iris) +
  geom_point(aes(x = Sepal.Width, y = 0))
```



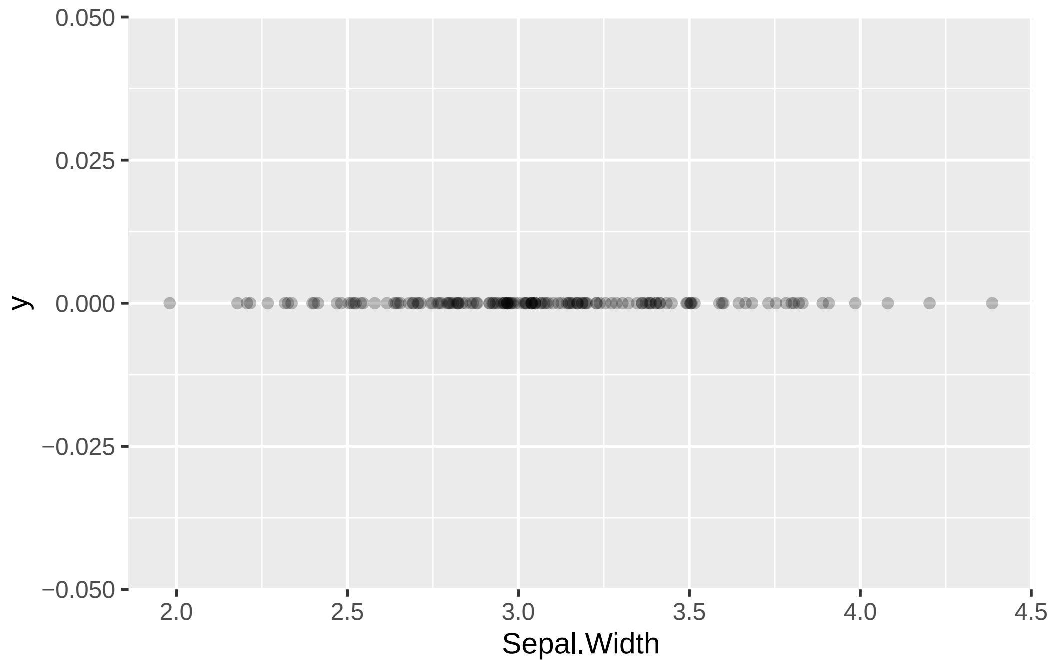
6.5.1 Jittering data

There should have been 150 points plotted in the figure above (one for each of the iris plants in the data set), but visually it looks like only about 25 or 30 points are shown. What's going on? If you examine the iris data, you'll see that the all the measures are rounded to the nearest tenth of a centimer, so that there are a large number of observations with identical values of `Sepal.Width`. This is a limitation of the precision of measurements that was used when generating the data set.

To provide a visual clue that there are multiple observations that share the same value, we can slightly "jitter" the values (randomly move points a small amount in either in the vertical or horizontal direction). Jittering is used solely to enhance visualization, and any statistical analyses you carry out would be based on the original data. When presenting your data to someone else, should note when you've used jittering so as not to misconvey the actual data.

Jittering can be accomplished using `geom_jitter`, which is derived from `geom_point`:

```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = 0),
              width = 0.05, height = 0, alpha = 0.25)
```



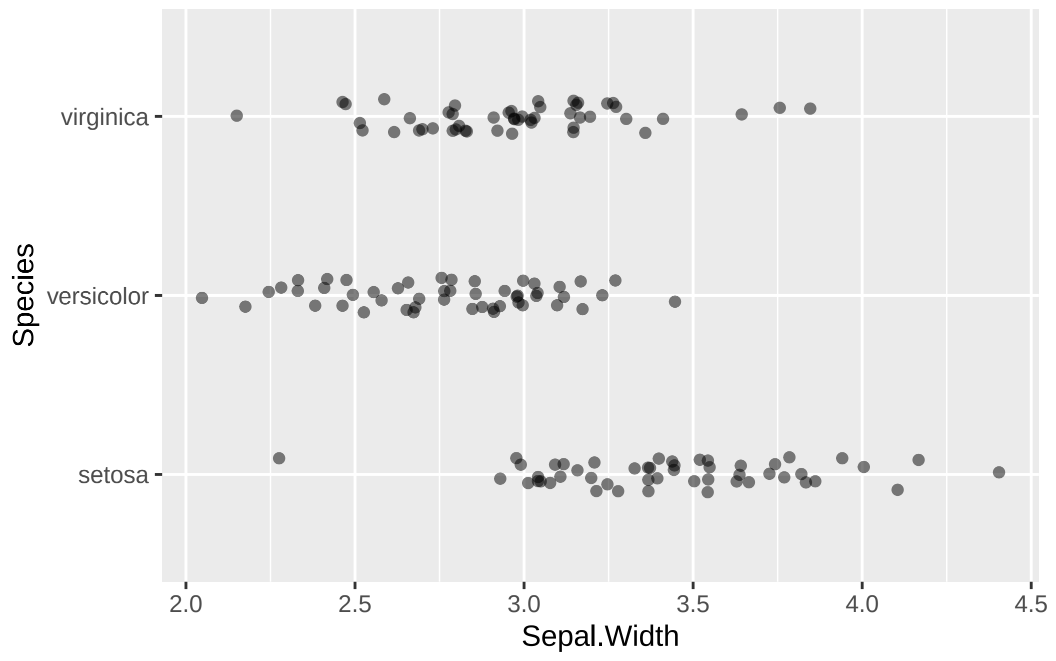
The `width` and `height` arguments specify the maximum amount (as fractions of the data) to jitter the observed data points in the horizontal (width) and vertical (height) directions. Here we only jitter the data in the horizontal direction. The `alpha` argument controls the transparency of the points – the valid range of alpha values is 0 to 1, where 0 means completely transparent and 1 is completely opaque.

Within a geom, arguments outside of the `aes` mapping apply uniformly across the visualization (i.e. they are fixed values). For example, setting 'alpha = 0.25' made all the points transparent.

6.5.2 Adding categorical information

Recall that are three different species represented in the data: Iris setosa, I. versicolor, and I. virginica. Let's see how to generate a strip plot that also includes a breakdown by species.

```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = Species),
            width=0.05, height=0.1, alpha=0.5)
```

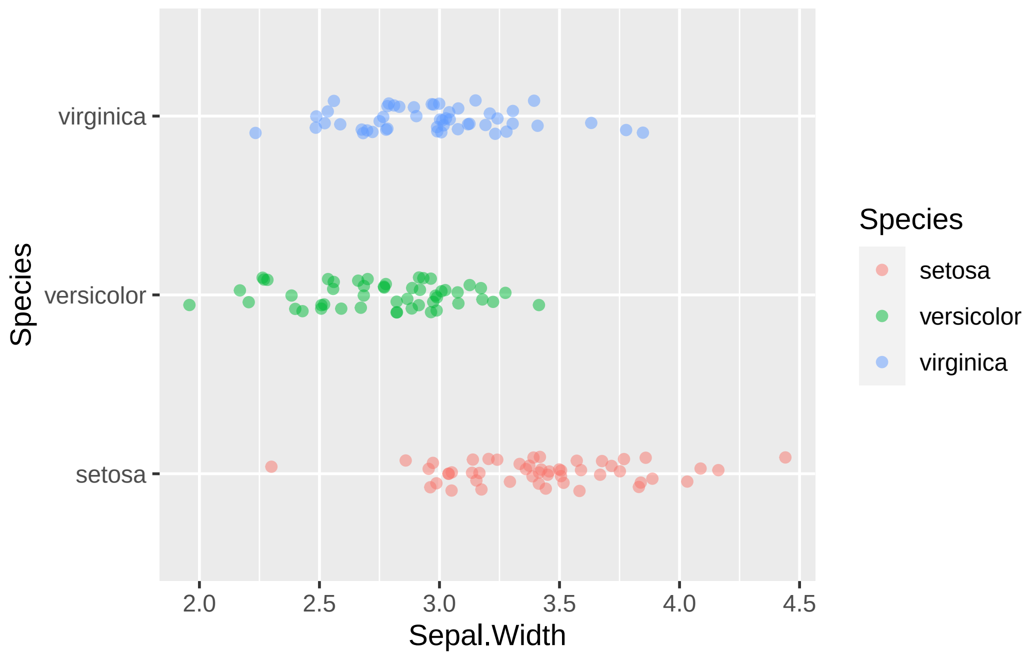


That was easy! All we had to do was change the aesthetic mapping in `geom_jitter`, specifying "Species" as the y variable. I also added a little vertical jitter as well to better separate the points.

Now we have a much better sense of the data. In particular it's clear that the *I. setosa* specimens generally have wider sepals than samples from the other two species.

Let's tweak this a little by also adding color information, to further emphasize the distinct groupings. We can do this by adding another argument to the aesthetic mapping in `geom_jitter`.

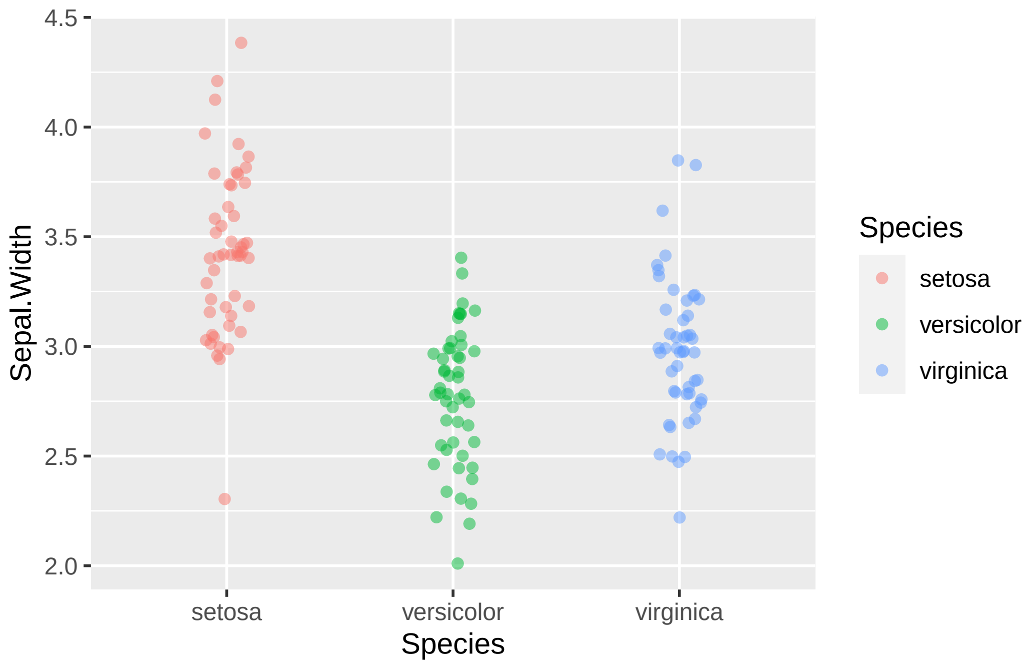
```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = Species, color=Species),
            width=0.05, height=0.1, alpha=0.5)
```



6.5.3 Rotating plot coordinates

What if we wanted to rotate this plot 90 degrees, depicting species on the x-axis and sepal width on the y-axis. For this example, it would be easy to do this by simpling swapping the variables in the `aes` mapping argument. However an alternate way to do this is with a coordinate transformation function. Here we use `coord_flip` to flip the x- and y-axes:

```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = Species, color=Species),
              width=0.05, height=0.1, alpha=0.5) +
  coord_flip()
```



We'll see other uses of coordinate transformations in later lectures.

6.6 Histograms

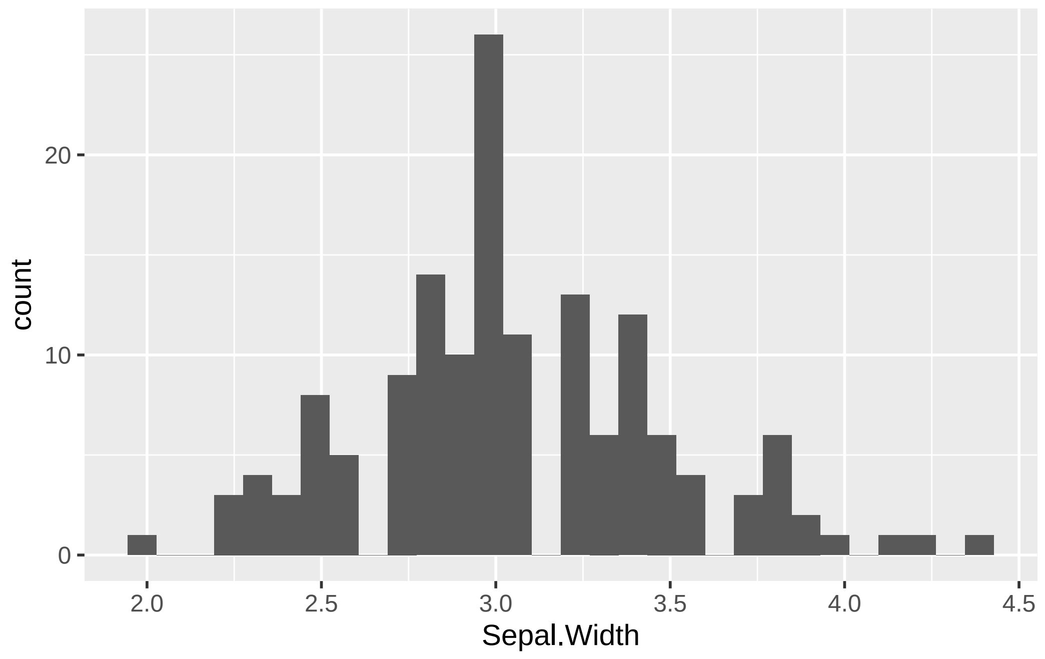
Histograms are probably the most common way to depict univariate data. In a histogram rather than showing individual observations, we divide the range of the data into a set of bins, and use vertical bars to depict the number (frequency) of observations that fall into each bin. This gives a good sense of the intervals in which most of the observations are found.

The geom, `geom_histogram`, takes care of both the geometric representation and the statistical transformations of the data necessary to calculate the counts in each binn.

Here's the simplest way to use `geom_histogram`:

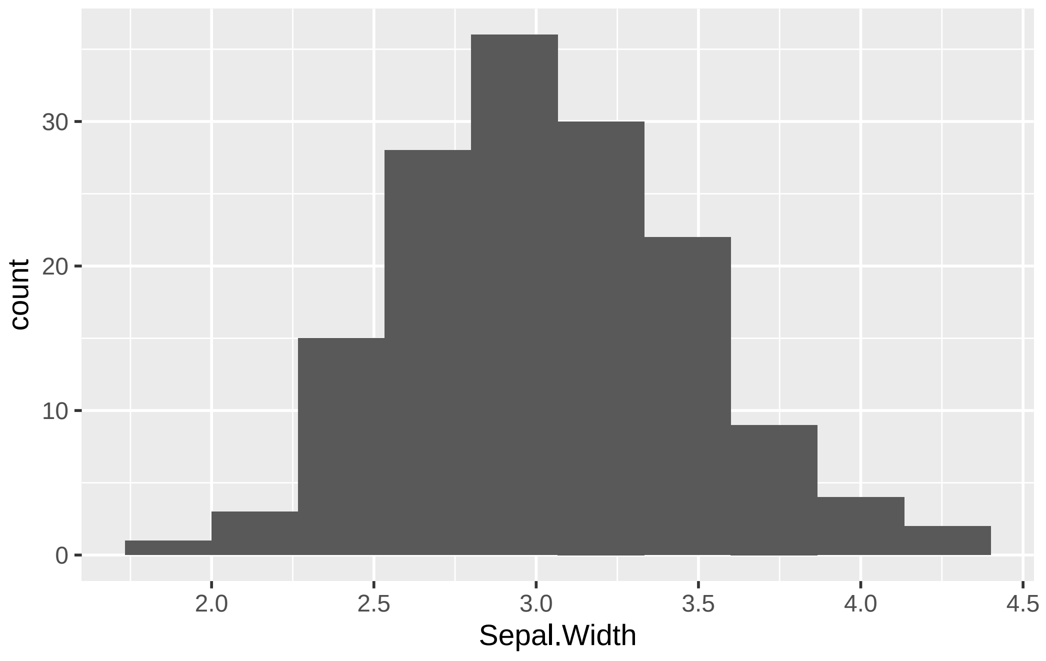
```r
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width))
```

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

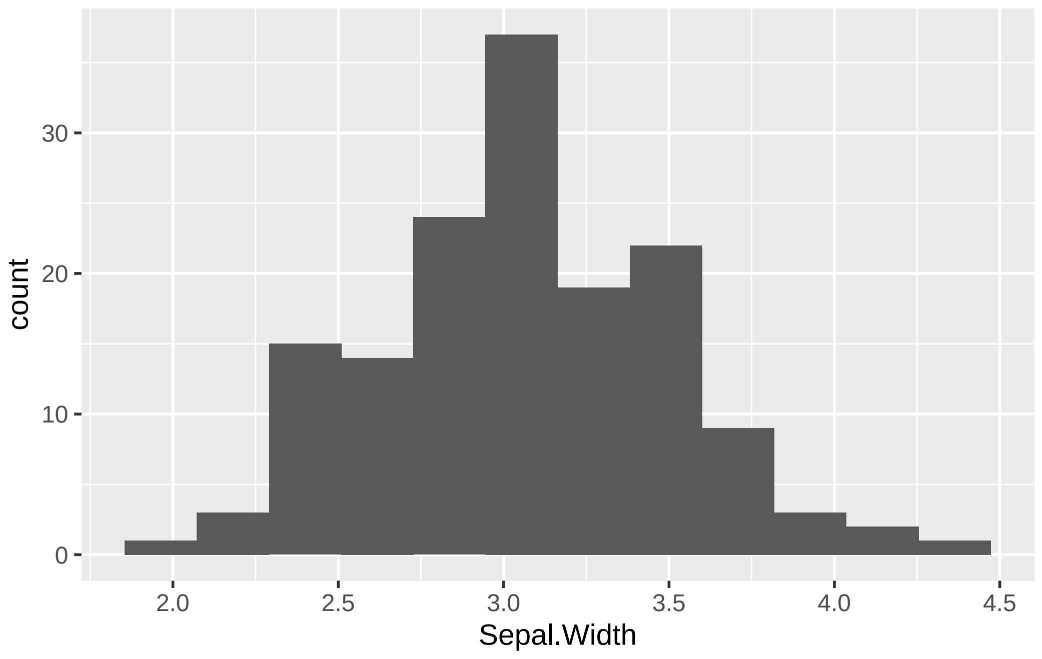


The default number of bins that `geom_histogram` uses is 30. For modest size data sets this is often too many bins, so it's worth exploring how the histogram changes with different bin numbers:

```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width), bins = 10)
```



```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width), bins = 12)
```



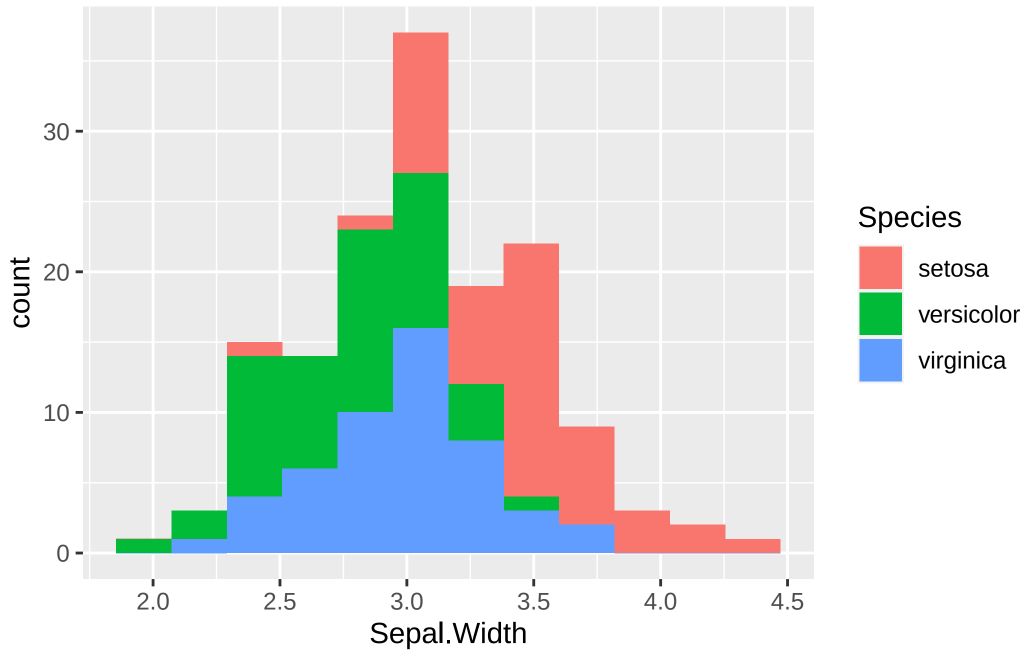
One important thing to note when looking at these histograms with different numbers of bins is that the number of bins used can change your perception of the data. For example, the number of peaks (modes) in the data can be very sensitive to the bin number as can the perception of gaps.

6.6.1 Variations on histograms when considering categorical data

As before, we probably want to break the data down by species. Here we're faced with some choices about how we depict that data. Do we generate a "stacked histogram" to where the colors indicate the number of observations in each bin that belong to each species? Do we generate side-by-side bars for each species? Or Do we generate separate histograms for each species, and show them overlapping?

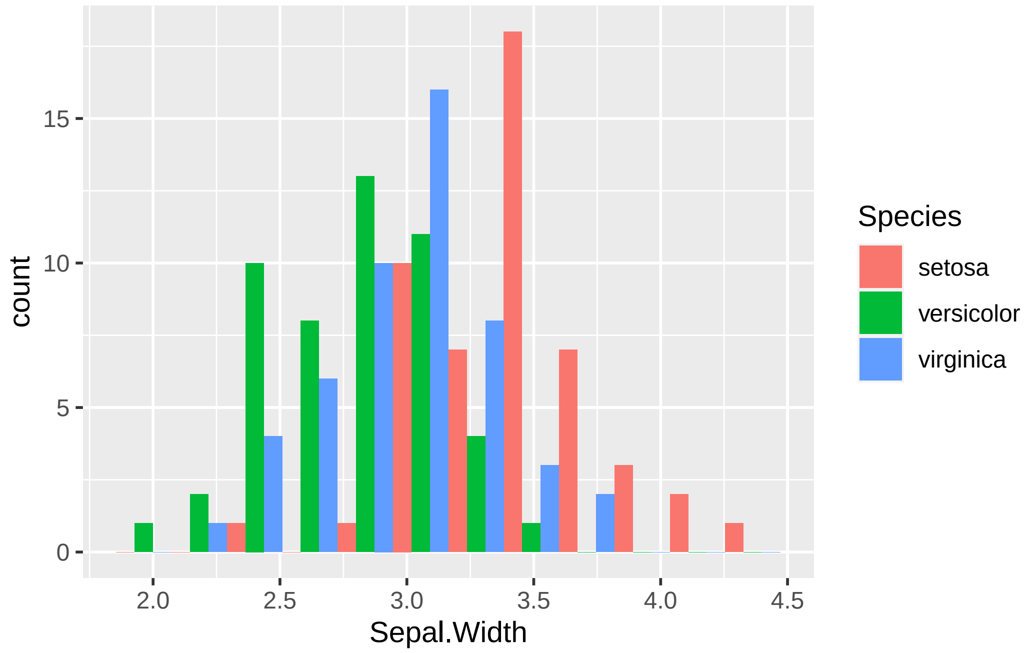
Stacked histograms are the default if we associate a categorical variable with the bar fill color:

```r
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species),
                 bins = 12)
```



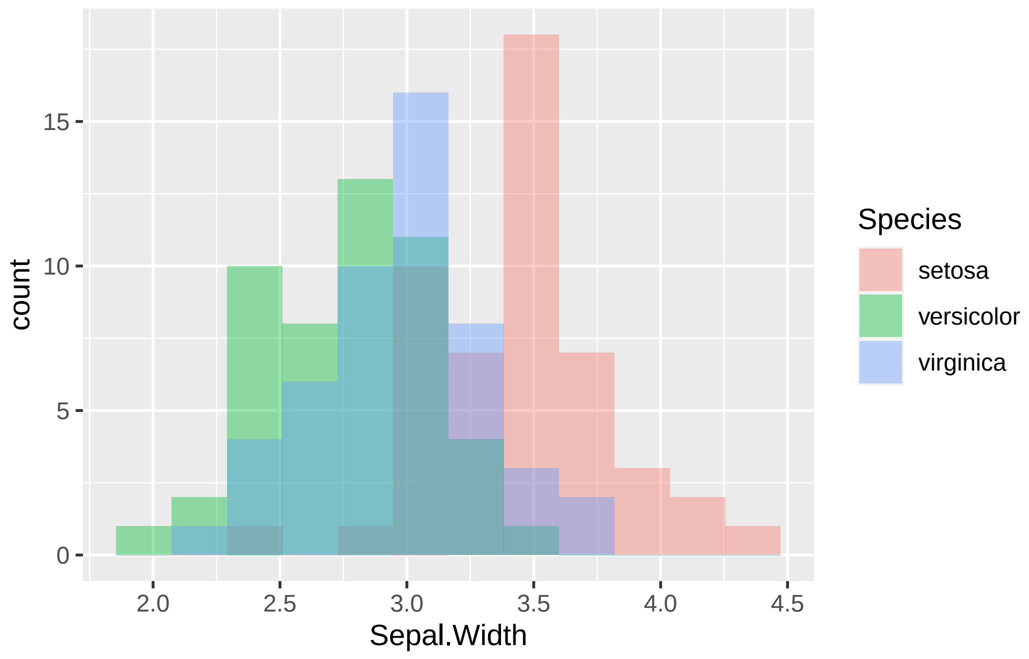
To get side-by-side bars, specify "dodge" as the `position` argument to `geom_histogram`.

```r
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species),
                 bins = 12, position = "dodge")
```



If you want overlapping histograms, use `position = "identity"` instead. When generating overlapping histograms like this, you probably want to make the bars semi-transparent so you can can distinguish the overlapping data.

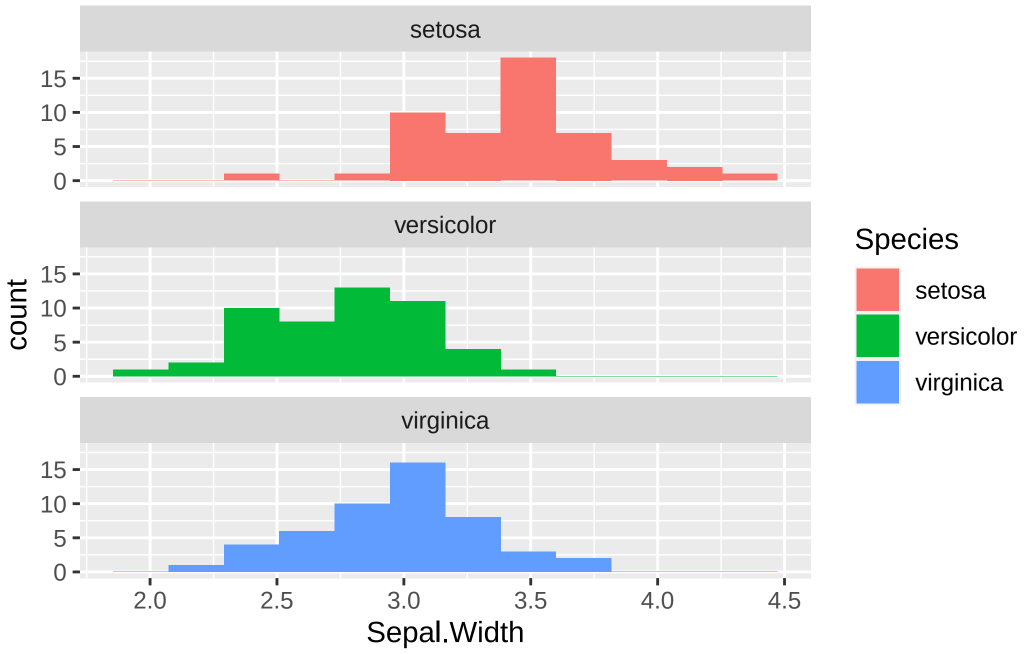
```r
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species),
                 bins = 12, position = "identity", alpha = 0.4)
```



6.7 Faceting to depict categorical information

Yet another way to represent the histograms for the three species is to using faceting, the create subplots for each species. Faceting is the operation of subsetting the data with respect to a discrete or categorical variable of interest, and generating the same plot type for each subset. Here we use the "ncol" argument to the `facet_wrap` function to specify that the subplots should be drawn in a single vertical column to facilitate comparison of the distributions.

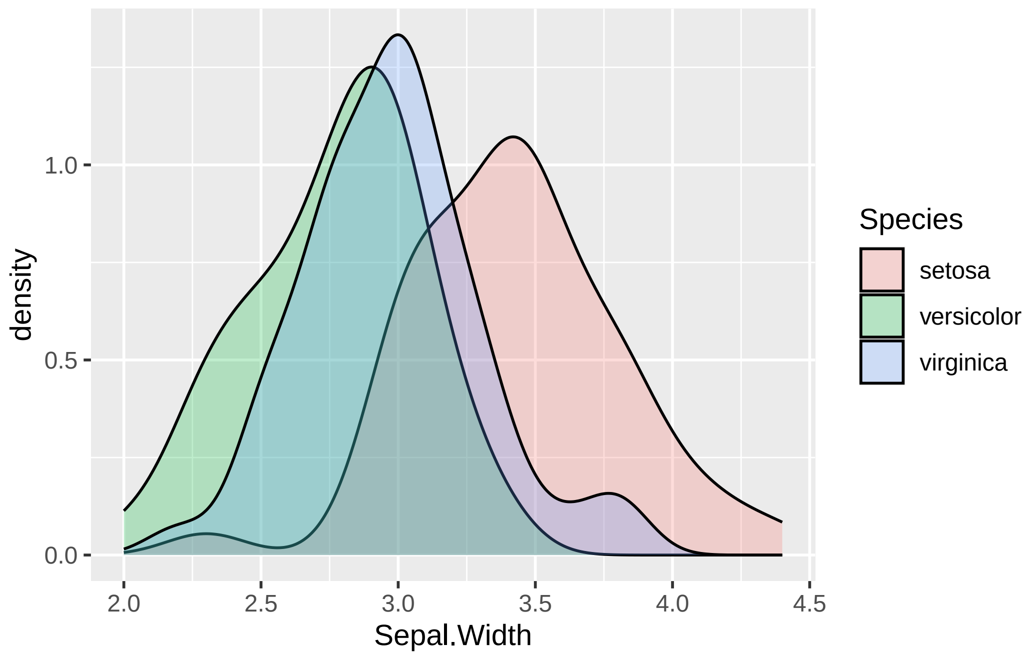
```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species), bins = 12) +
  facet_wrap(~Species, ncol = 1)
```



6.8 Density plots

One shortcoming of histograms is that they are sensitive to the choice of bin margins and the number of bins. An alternative is a "density plot", which you can think of as a smoothed version of a histogram.

```
ggplot(iris) +
  geom_density(aes(x = Sepal.Width, fill = Species), alpha=0.25)
```



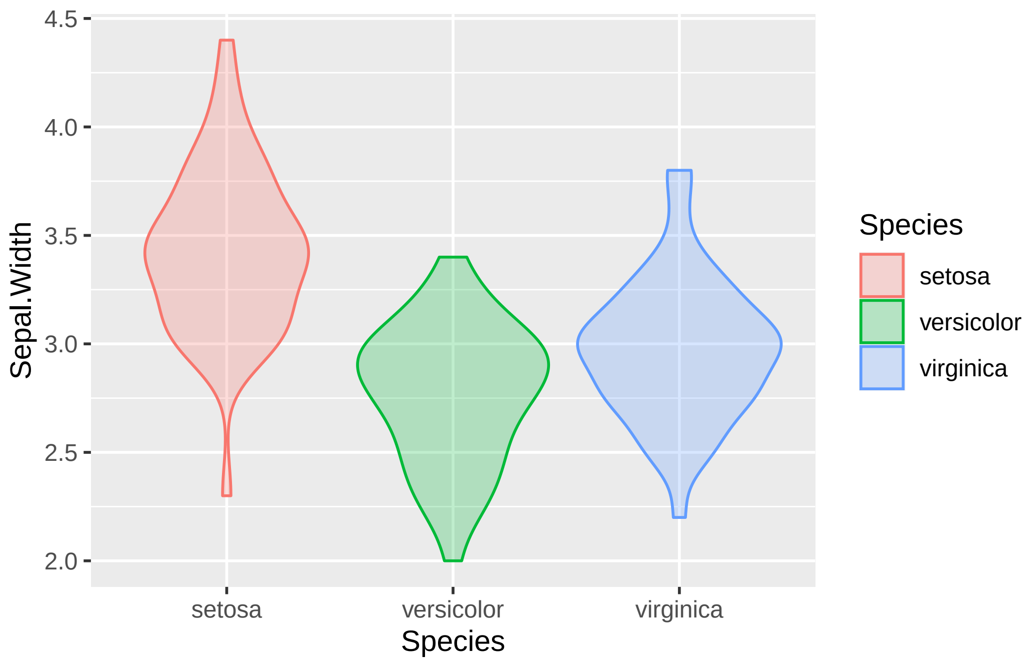
Density plots still make some assumptions that affect the visualization, in particular a "smoothing bandwidth" (specified by the argument `bw`) which determines how course or granular the density estimation is.

Note that the vertical scale on a density plot is no longer counts (frequency) but probability density. In a density plot, the total area under the plot adds up to one. Intervals in a density plot therefore have a probabilistic intepretation.

6.9 Violin or Beanplot

A violin plot (sometimes called a bean plot) is closely related to a density plot. In fact you can think of a violin plot as a density plot rotated 90 degress and mirrored left/right.

```
ggplot(iris) +
  geom_violin(aes(x = Species, y = Sepal.Width, color = Species, fill=Species),
              alpha = 0.25)
```

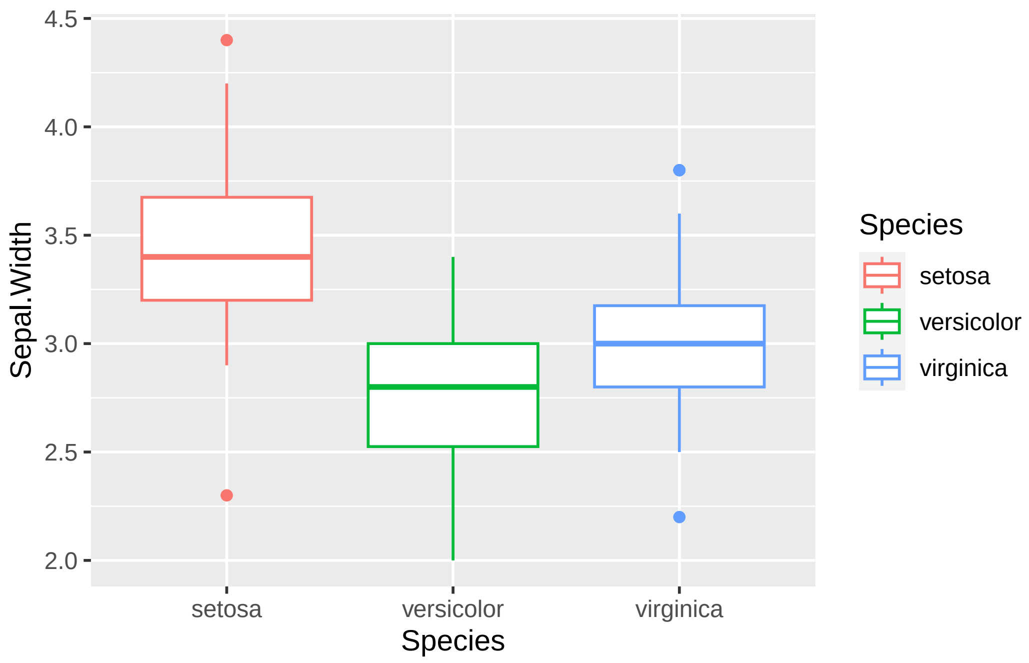


6.10 Boxplots

Boxplots are another frequently used univariate visualization. Boxplots provide a compact summary of single variables, and are most often used for comparing distributions between groups.

A standard box plot depicts five useful features of a set of observations: 1) the median (center most line); 2 and 3) the first and third quartiles (top and bottom of the box); 4) the whiskers of a boxplot extend from the first/third quartile to the highest value that is within 1.5 * IQR, where IQR is the inter-quartile range (distance between the first and third quartiles); 5) points outside of the whiskers are usually consider extremal points or outliers. There are many variants on box plots, particularly with respect to the "whiskers". It's always a good idea to be explicit about what a box plot you've created shows.

```
ggplot(iris) +
  geom_boxplot(aes(x = Species, y = Sepal.Width, color = Species))
```



Boxplots are most commonly drawn with the cateogorical variable on the x-axis.

6.11 Building complex visualizations with layers

All of our ggplot2 examples up to now have involved a single geom. We can think of geoms as “layers” of information in a plot. One of the powerful features of plotting useing ggplot2 is that it is trivial to combine layers to make more complex plots.

The template for multi-layered plots is a simple extension of the single layer:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION1>(mapping = aes(<MAPPINGS>)) +
  <GEOM_FUNCTION2>(mapping = aes(<MAPPINGS>))
```

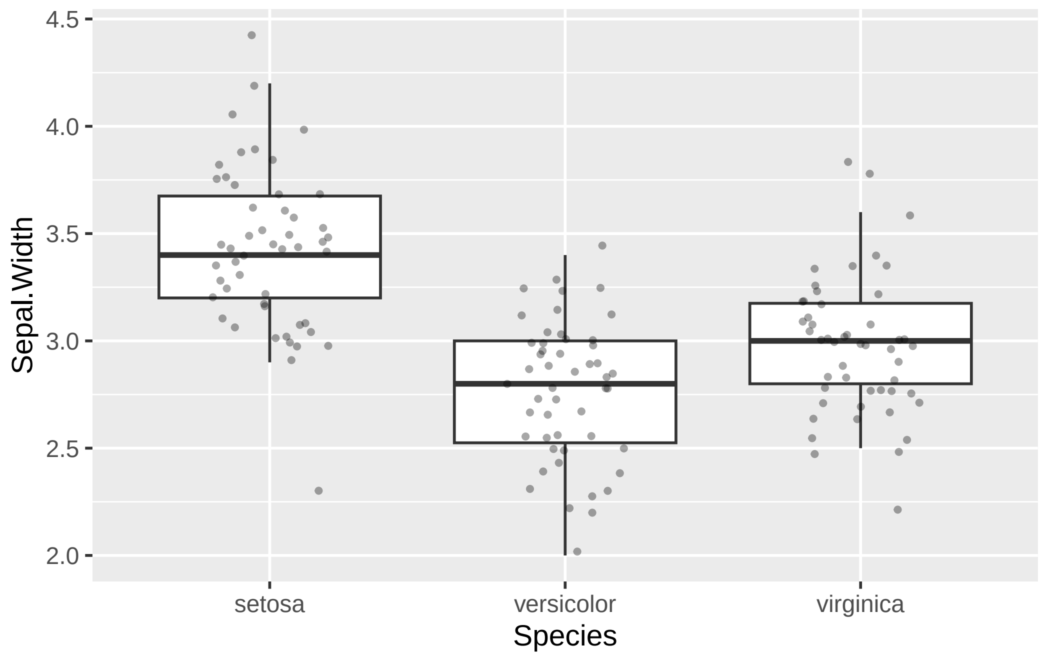
6.12 Useful combination plots

Boxplot or violin plots represent visual summaries/simplifications of the underlying data. This is useful but sometimes key information is lost in the process of summarizing. Combining these plots with a strip plot give you both the “birds eye view” as well as granular information.

6.12.1 Boxplot plus strip plot

Here's an example of combining box plots and strip plots:

```
ggplot(iris) +
  # outlier.shape = NA suppresses the depiction of outlier points in the boxplot
  geom_boxplot(aes(x = Species, y = Sepal.Width), outlier.shape = NA) +
  # size sets the point size for the jitter plot
  geom_jitter(aes(x = Species, y = Sepal.Width), width=0.2, height=0.05, alpha=0.35, size=
```

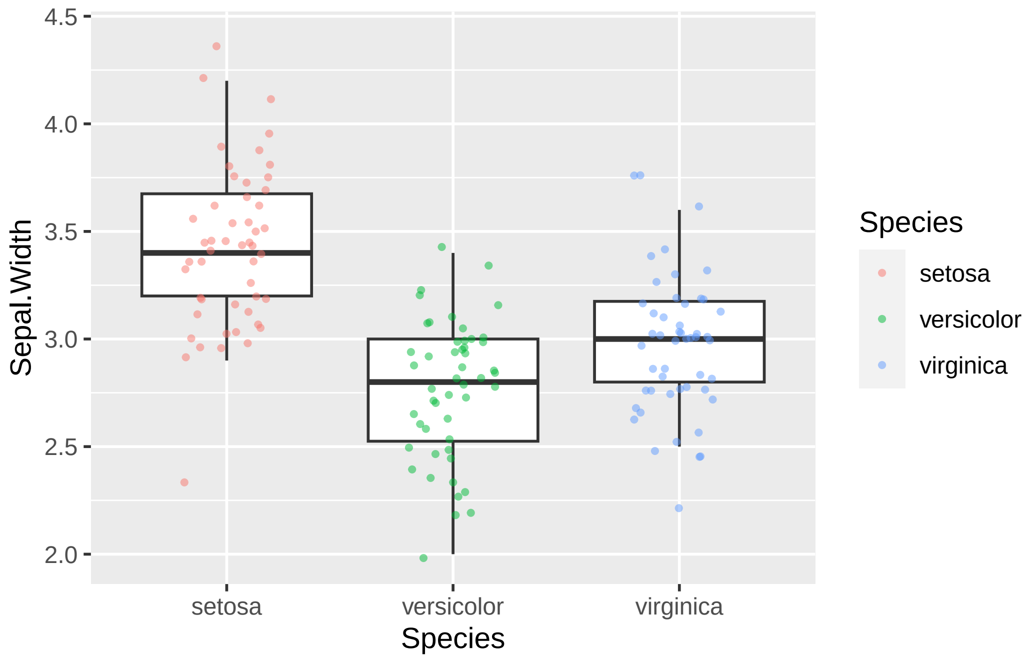


Note that I suppressed the plotting of outliers in `geom_boxplot` so as not to draw the same points twice (the individual data are drawn by `geom_jitter`).

6.12.2 Setting shared aesthetics

The example above works well, but you might have noticed that there's some repetition of code. In particular, we set the same aesthetic mapping in both `geom_boxplot` and `geom_jitter`. It turns out that creating layers that share some of the same aesthetic values is a common case. To deal with such cases, you can specify *shared aesthetic mappings* as an argument to the `ggplot` function and then set additional aesthetics specific to each layer in the individual geoms. Using this approach, our previous example can be written more compactly as follow.

```
ggplot(iris, mapping = aes(x = Species, y = Sepal.Width)) +
  geom_boxplot(outlier.shape = NA) +
  # note how we specify a layer specific aesthetic in geom_jitter
  geom_jitter(aes(color = Species), width=0.2, height=0.05, alpha=0.5, size=0.75)
```



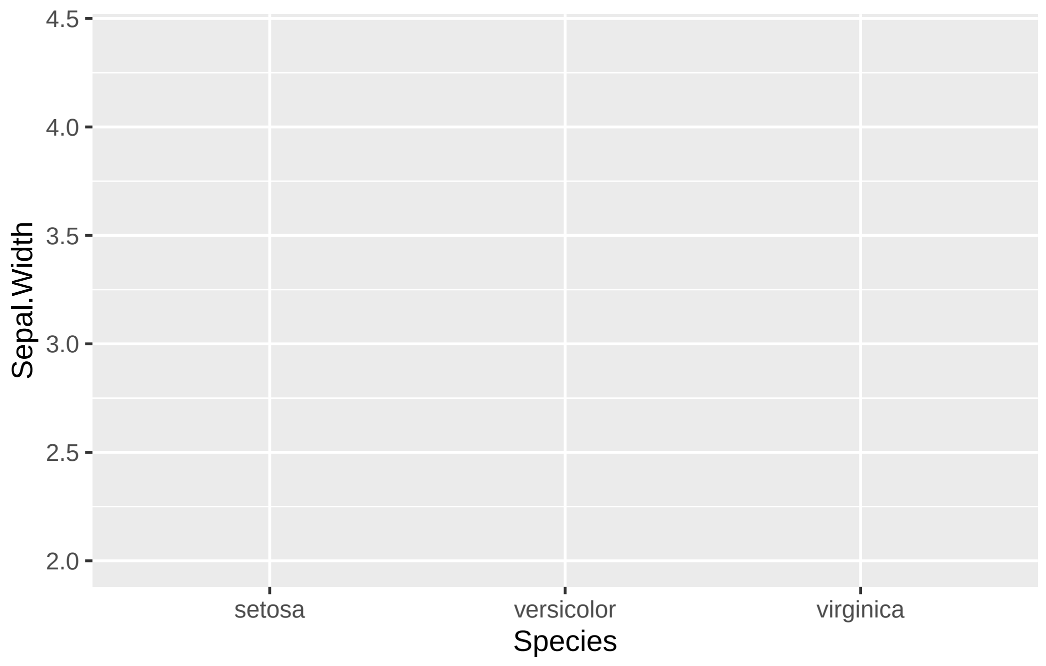
6.13 ggplot layers can be assigned to variables

The function `ggplot()` returns a "plot object" that we can assign to a variable. The following example illustrates this:

```
# create base plot object and assign to variable p
# this does NOT draw the plot
p <- ggplot(iris, mapping = aes(x = Species, y = Sepal.Width))
```

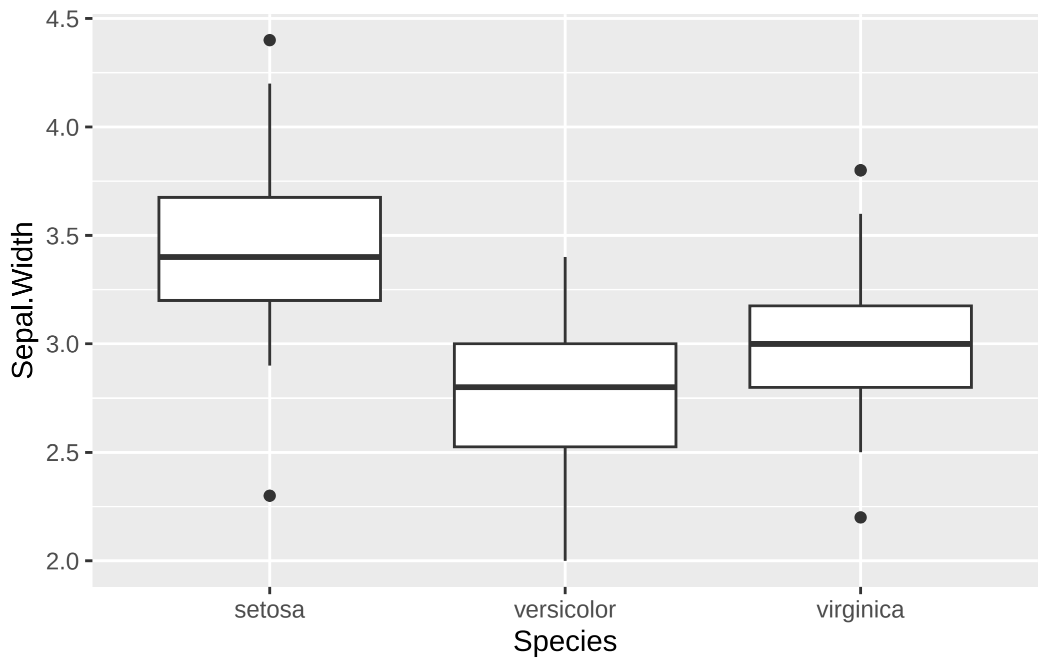
In the code above we created a plot object and assigned it to the variable `p`. However, the plot wasn't drawn. To draw the plot object we evaluate it as so:

```
p  # try to draw the plot object
```



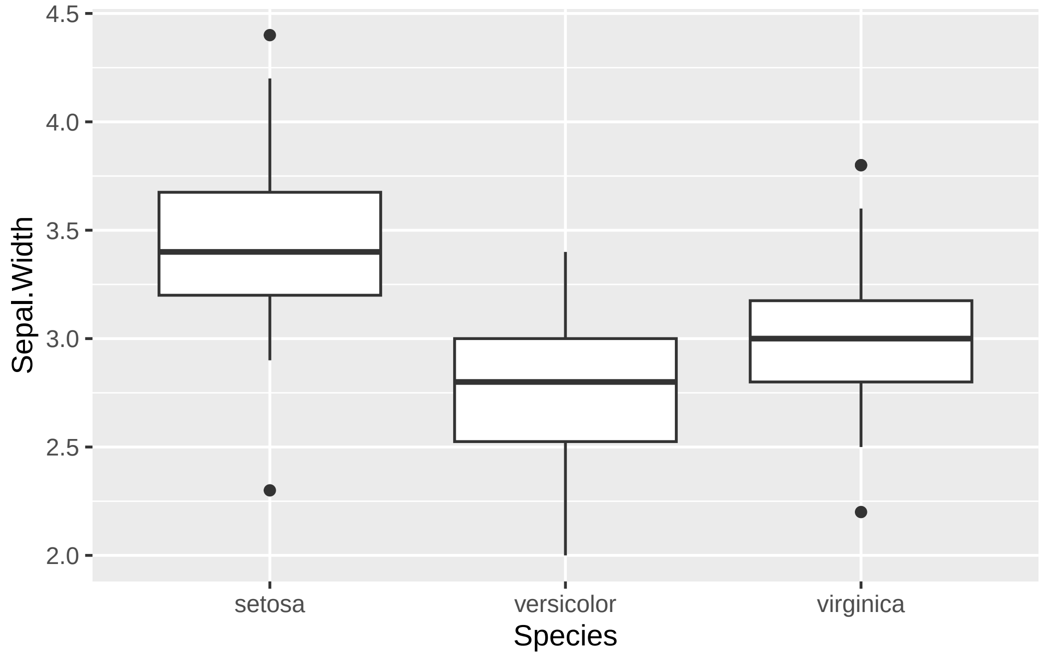
The code block above didn't generate an image, because we haven't added a geom to the plot to determine how our data should be drawn. We can add a geom to our pre-created plot object as so:

```
# add a point geom to our base layer and draw the plot
p + geom_boxplot()
```



If we wanted to we could have assigned the geom to a variable as well:

```
box.layer <- geom_boxplot()
p + box.layer
```

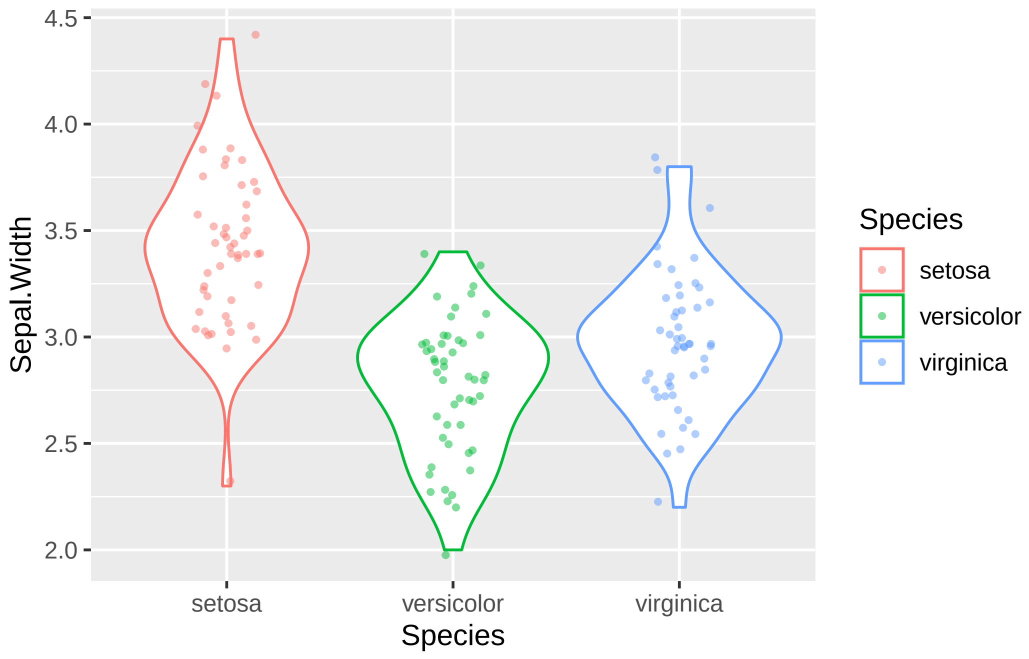


In this case we don't really gain anything by creating an intermediate variable, but for more complex plots or when considering different versions of a plot this can be very useful.

6.13.1 Violin plot plus strip plot

Here is the principle of combining layers, applied to a combined violin plot + strip plot. Again, we set shared aesthetic mappings in `ggplot` function call and this time we assign individual layers of the plot to variables.

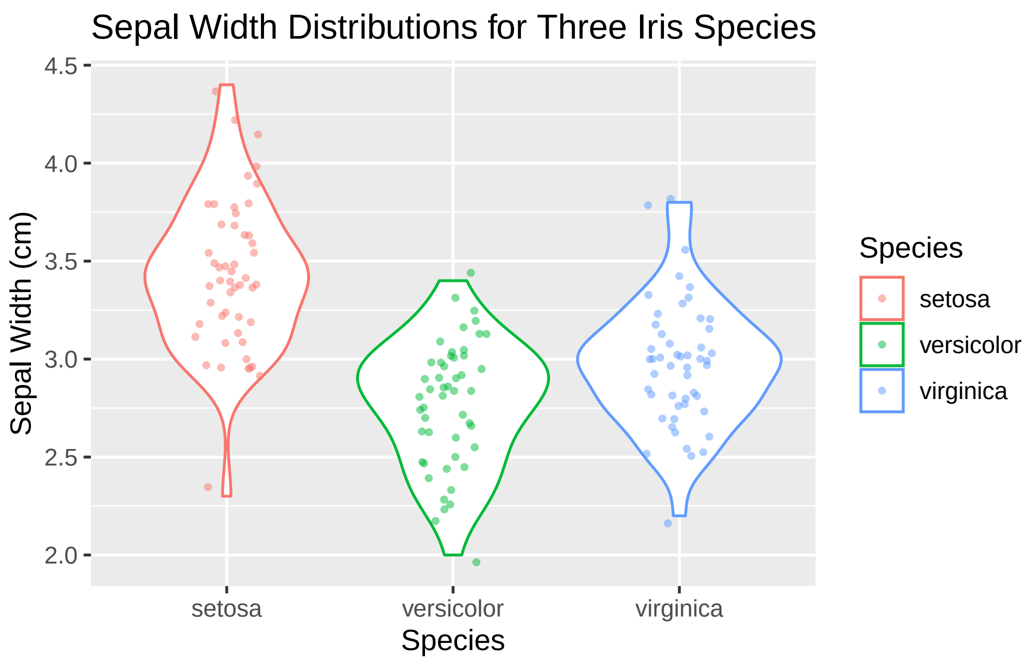
```
p <- ggplot(iris, mapping = aes(x = Species, y = Sepal.Width, color = Species))

violin.layer <- geom_violin()
jitter.layer <- geom_jitter(width=0.15, height=0.05, alpha=0.5, size=0.75)

p + violin.layer + jitter.layer # combined layers of plot and draw
```



6.14 Adding titles and tweaking axis labels

ggplot2 automatically adds axis labels based on the variable names in the data frame passed to `ggplot`. Sometimes these are appropriate, but more presentable figures you'll usually want to tweak the axis labs (e.g. adding units). The `labs` (short for labels) function allows you to do so, and also let's you set a title for your plot. We'll illustrate this by modifying our previous figure. Note that we save considerable amounts of re-typing since we had already assigned three of the plot layers to variables in the previous code block:

```
p + violin.layer + jitter.layer +
  labs(x = "Species", y = "Sepal Width (cm)",
       title = "Sepal Width Distributions for Three Iris Species")
```



6.15 ggplot2 themes

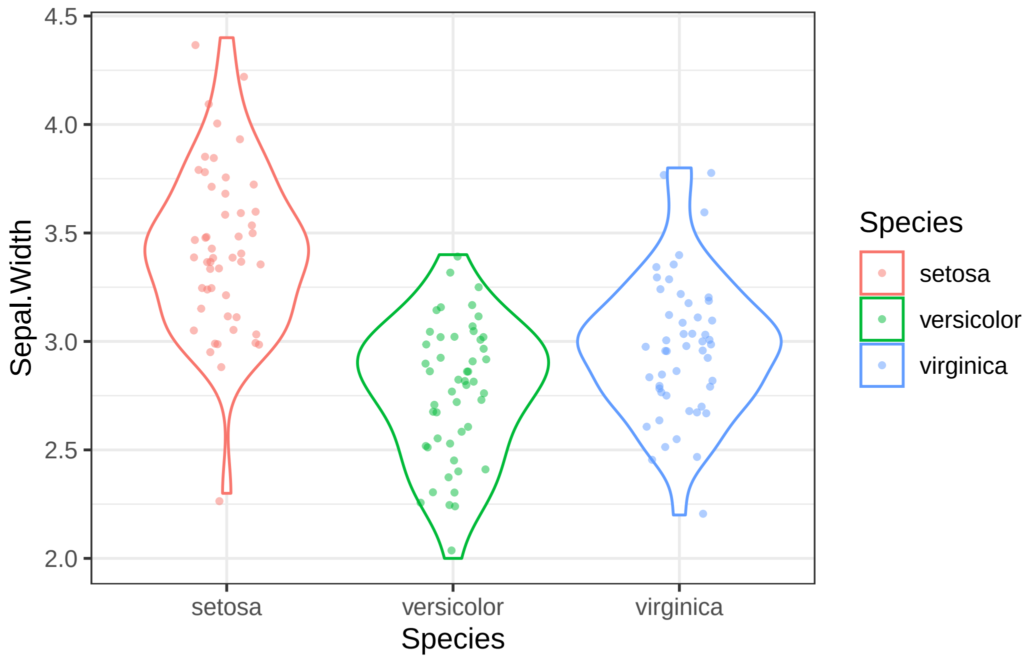
By now you're probably familiar with the default "look" of plots generated by ggplot2, in particular the ubiquitous gray background with a white grid. This default works fairly well in the context of RStudio notebooks and HTML output, but might not work as well for a published figure or a slide presentation. Almost every individual aspect of a plot can be tweaked, but ggplot2 provides an easier way to make consistent changes to a plot using "themes". You can think of a theme as adding another layer to your plot. Themes should generally be applied after all the other graphical layers are created (geoms, facets, labels) so the changes they create affect all the prior layers.

There are eight default themes included with ggplot2, which can be invoked by calling the corresponding theme functions: `theme_gray`, `theme_bw`, `theme_linedraw`, `theme_light`, `theme_dark`, `theme_minimal`, `theme_classic`, and `theme_void` (See http://ggplot2.tidyverse.org/reference/ggtheme.html for a visual tour of all the default themes)

For example, let's generate a boxplot using `theme_bw` which get's rid of the gray background:

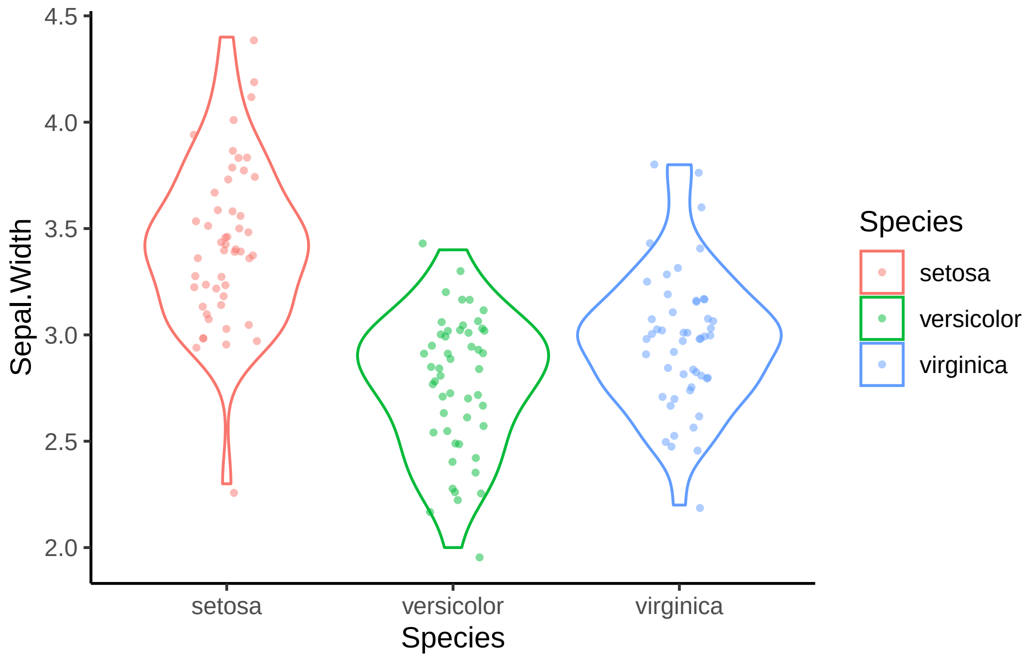
```
# create another variable to hold combination of three previous
# ggplot layers. I'm doing this because I'm going to keep re-using
# the same plot in the following code blocks
```

```r
violin.plus.jitter <- p + violin.layer + jitter.layer

violin.plus.jitter + theme_bw()
```



Another theme, `theme_classic`, remove the grid lines completely, and also gets rid of the top-most and right-most axis lines.

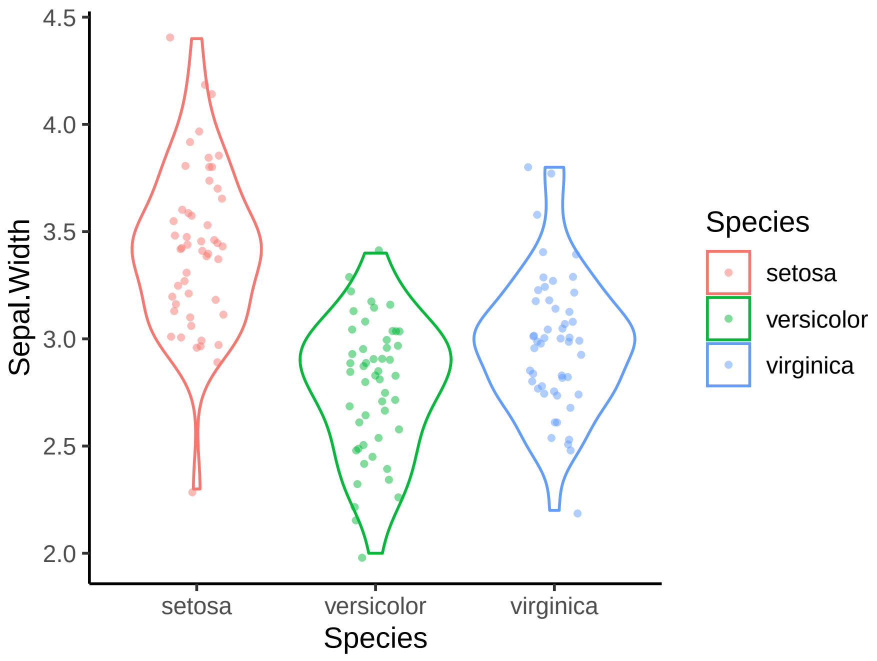
```r
violin.plus.jitter + theme_classic()
```



6.15.1 Further customization with `ggplot2::theme`

In addition to the eight complete themes, there is a `theme` function in ggplot2 that allows you to tweak particular elements of a theme (see `?theme` for all the possible options). For example, to tweak just the aspect ratio of a plot (the ratio of width to height), you can set the `aspect.ratio` argument in `theme`:

```
violin.plus.jitter + theme_classic() + theme(aspect.ratio = 1)
```

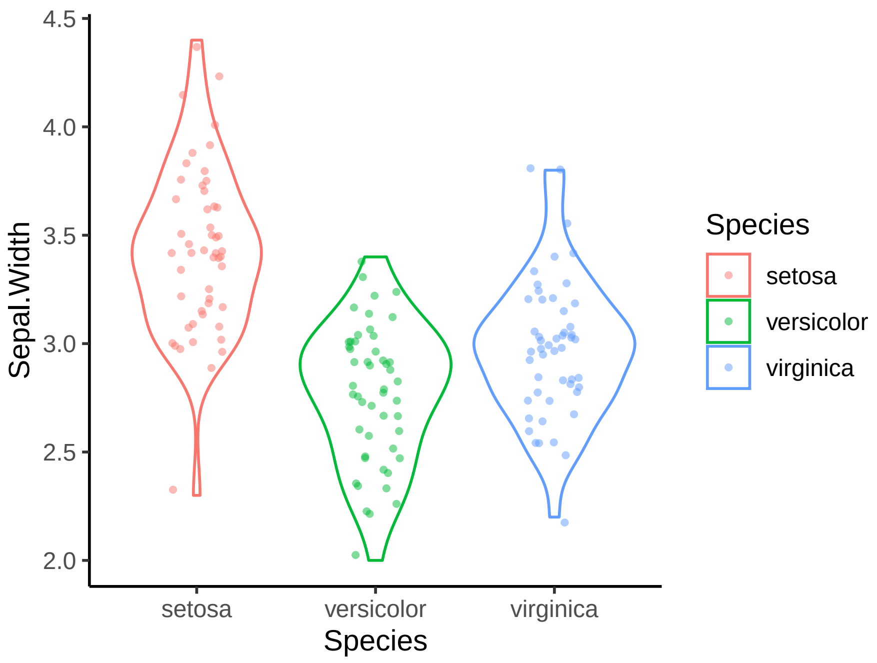


Theme related function calls can be combined to generate new themes. For example, let's create a theme called `my.theme` by combining `theme_classic` with a call to `theme`:

```
my.theme <- theme_classic()  + theme(aspect.ratio = 1)
```

We can then apply this theme as so:

```
violin.plus.jitter + my.theme
```



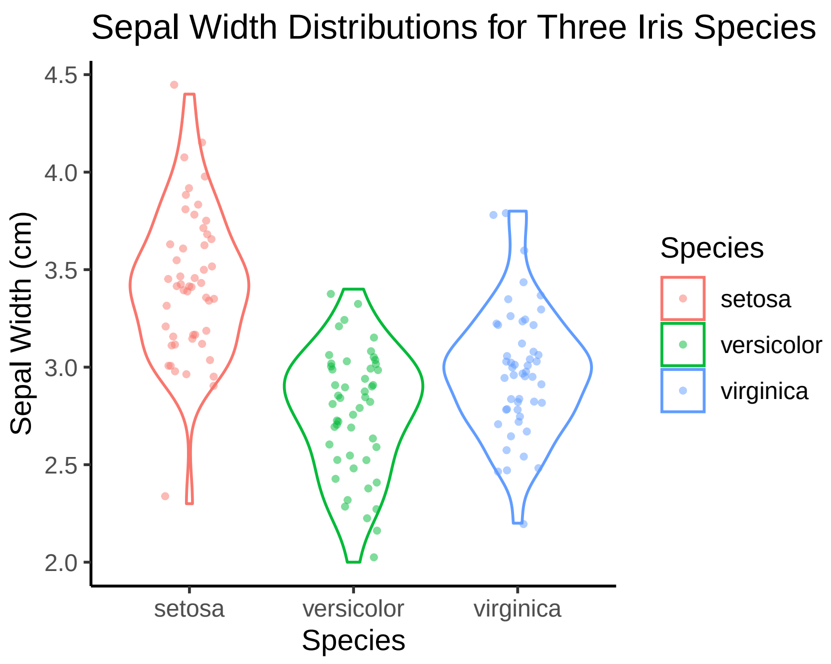
6.16 Other aspects of ggplots can be assigned to variables

Plot objects, geoms and themes are not the only aspects of a figure that can be assigned to variables for later use. For example, we can create a label object:

```
my.labels <- labs(x = "Species", y = "Sepal Width (cm)",
                  title = "Sepal Width Distributions for Three Iris Species")
```

Combining all of our variables as so, we generate our new plot:

```
violin.plus.jitter + my.labels + my.theme
```



6.17 Bivariate plots

Now we turn our attention to some useful representations of bivariate distributions.

For the purposes of these illustrations I'm initially going to restrict my attention to just one of the three species represented in the iris data set – the I. setosa specimens. This allows us to introduce a vary useful base function called `subset()`. `subset()` will return subsets of a vector or data frames that meets the specified conditions. This can also be accomplished with conditional indexing but `subset()` is usually less verbose.

```
# create a new data frame composed only of the I. setosa samples
setosa.only <- subset(iris, Species == "setosa")
```

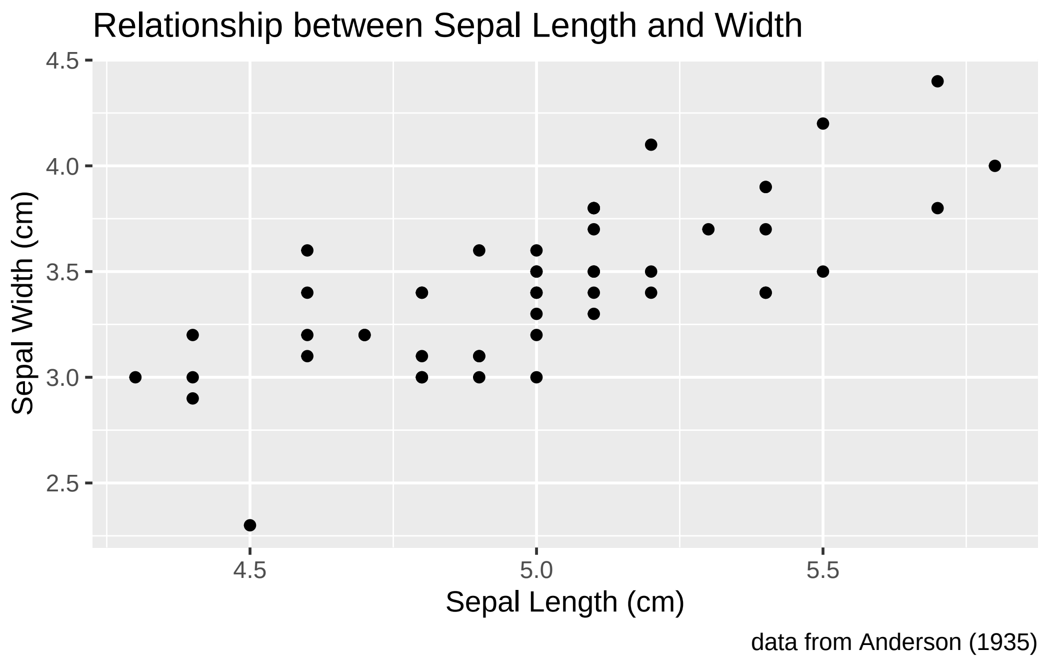
In the examples that follow, I'm going to illustrate different ways of representing the same bivariate distribution – the joint distribution of Sepal Length and Sepal Width – over and over again. To avoid repitition, let's assign the base ggplot layer to a variable as we did in our previous examples. We'll also pre-create a label layer.

```
setosa.sepals <- ggplot(setosa.only,
                        mapping = aes(x = Sepal.Length, y = Sepal.Width))

sepal.labels <- labs(x = "Sepal Length (cm)", y = "Sepal Width (cm)",
                     title = "Relationship between Sepal Length and Width",
                     caption = "data from Anderson (1935)")
```

6.17.1 Scatter plots

A scatter plot is one of the simplest representations of a bivariate distribution. Scatter plots are simple to create in ggplot2 by specifying the appropriate X and Y variables in the aesthetic mapping and using `geom_point` for the geometric mapping.

```
setosa.sepals  + geom_point() + sepal.labels
```



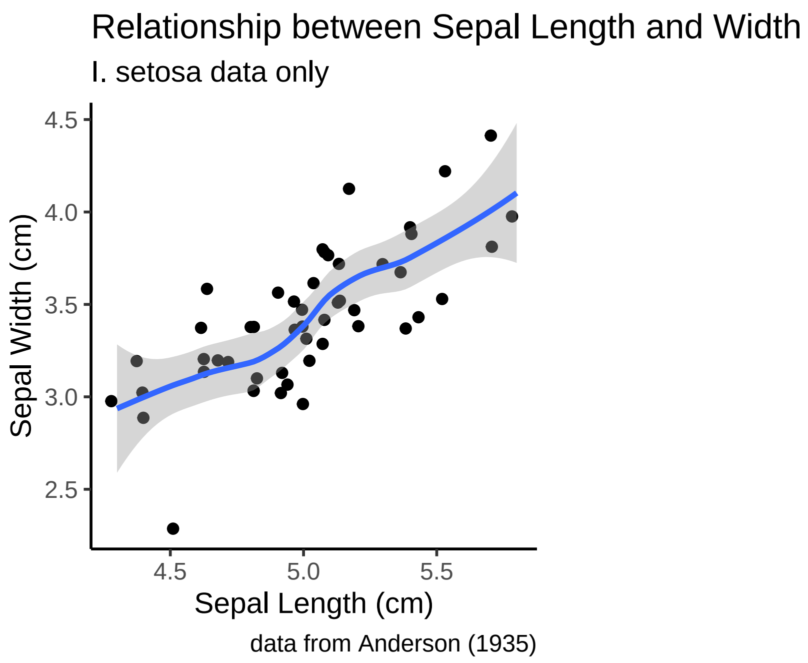
6.17.2 Adding a trend line to a scatter plot

ggplot2 makes it easy to add trend lines to plots. I use "trend lines" here to refer to representations like regression lines, smoothing splines, or other representations mean to help visualize the relationship between pairs of variables. We'll spend a fair amount of time exploring the mathematics and interpetation of regression lines and related techniques in later lectures, but for now just think about trends lines as summary representations for bivariate relationships.

Trend lines can be created using `geom_smooth`. Let's add a default trend line to our *I. setosa* scatter plot of the Sepal Width vs Sepal Length:

```
setosa.sepals +
  geom_jitter() +  # using geom_jitter to avoid overplotting of points
  geom_smooth() +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```

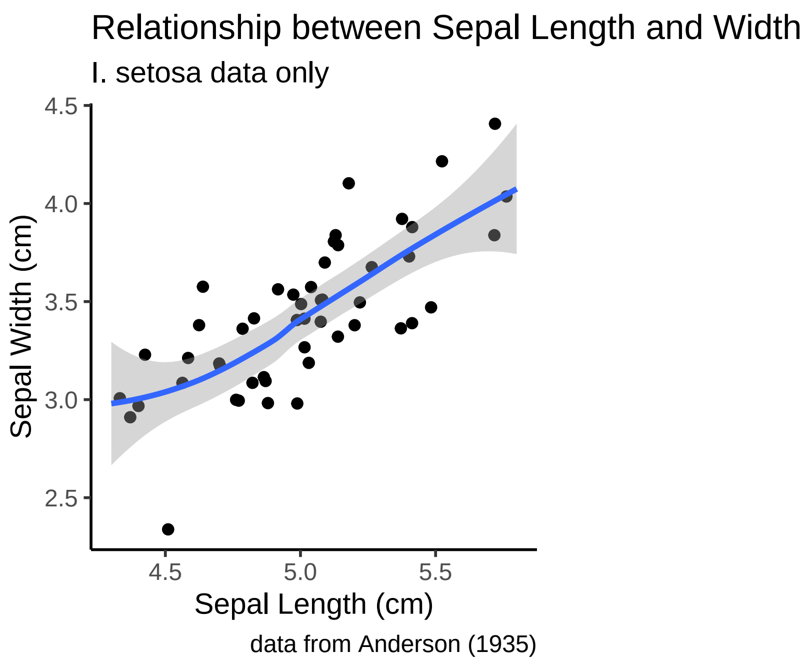
```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



The defaul trend line that `geom_smooth` fits is generated by a technique called "LOESS regression". LOESS regression is a non-linear curve fitting method, hence the squiggly trend line we see above. The smoothness of the LOESS regression is controlled by a parameter called `span` which is related to the proportion of points used. We'll discuss LOESS in detail in a later lecture, but here's an illustration how changing the span affects the smoothness of the fit curve:

```
setosa.sepals +
  geom_jitter() +  # using geom_jitter to avoid overplotting of points
  geom_smooth(span = 0.95) +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

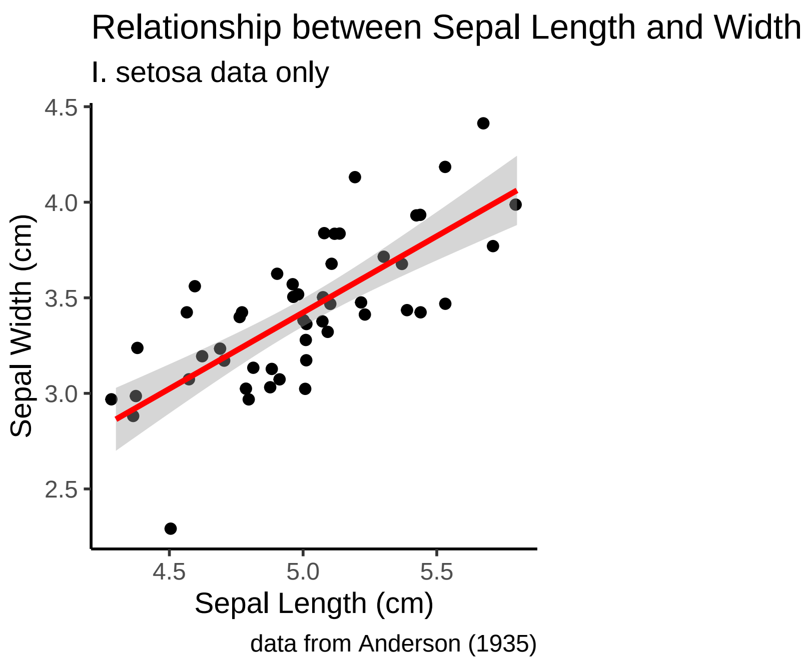


6.17.2.1 Linear trend lines

If instead we want a straight trend line, as would typically be depicted for a linear regression model we can specify a different statistical method:

```
setosa.sepals +
  geom_jitter() +  # using geom_jitter to avoid overplotting of points
  geom_smooth(method = "lm", color = "red") + # using linear model ("lm")
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```

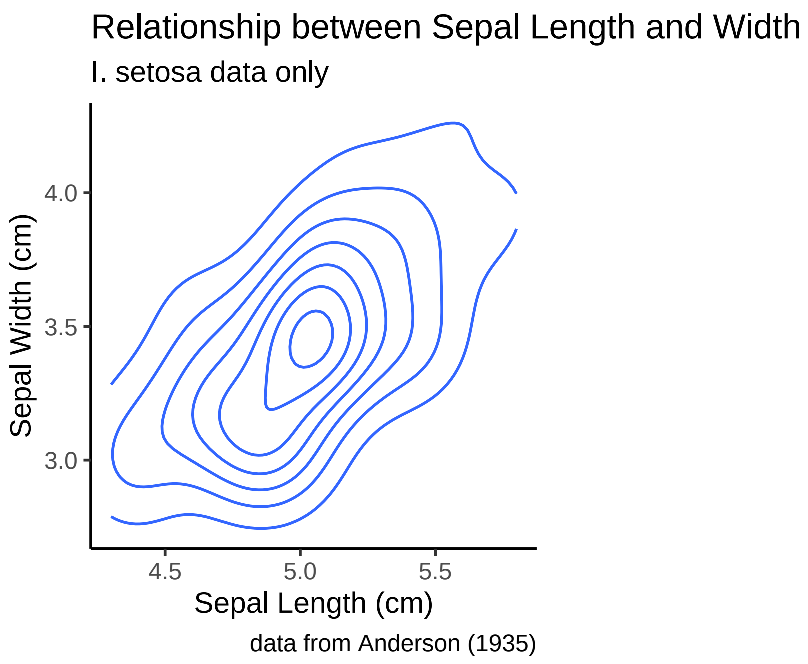
```
`geom_smooth()` using formula = 'y ~ x'
```



6.18 Bivariate density plots

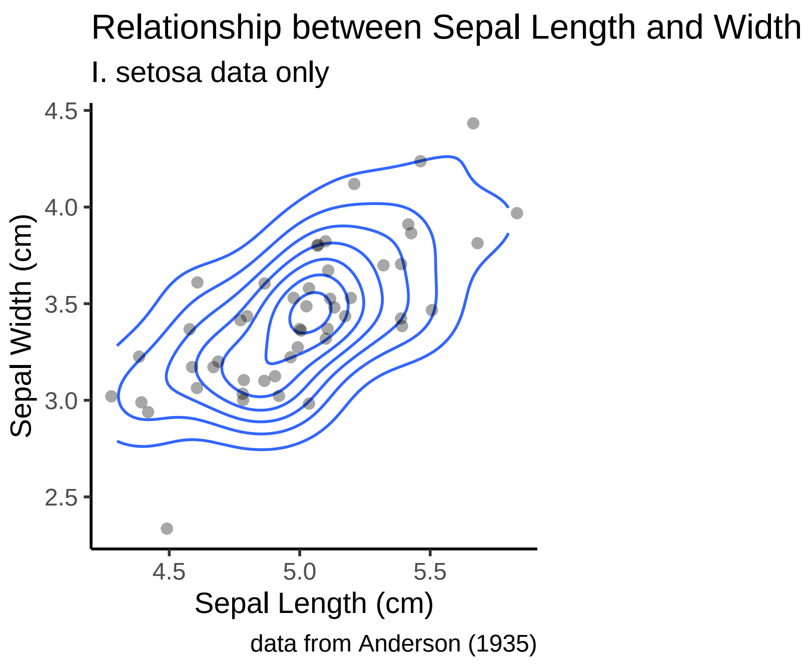
The density plot, which we introduced as a visualization for univariate data, can be extended to two-dimensional data. In a one dimensional density plot, the height of the curve was related to the relatively density of points in the surrounding region. In a 2D density plot, nested contours (or contours plus colors) indicate regions of higher local density. Let's illustrate this with an example:

```
setosa.sepals +
  geom_density2d() +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



The relationship between the 2D density plot and a scatter plot can be made clearer if we combine the two:

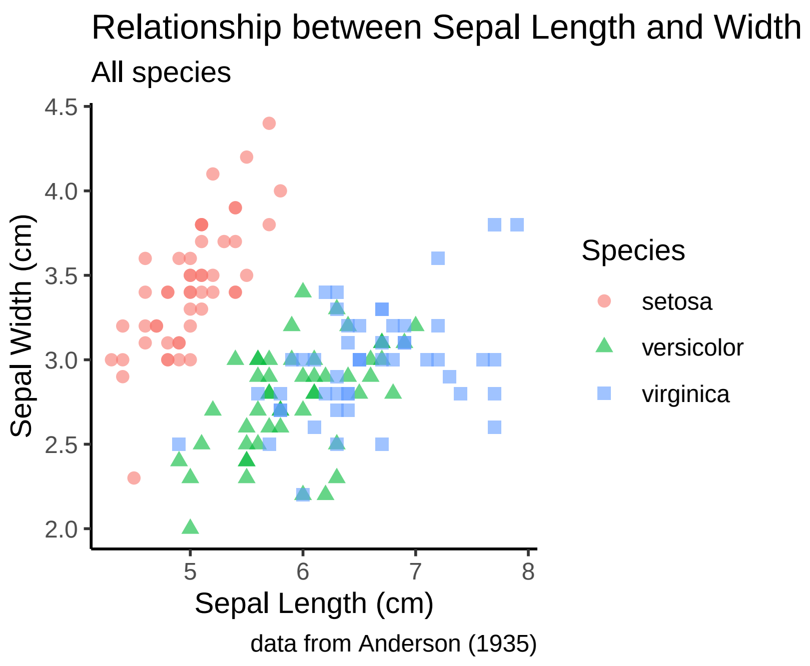
```
setosa.sepals +
  geom_density_2d() +
  geom_jitter(alpha=0.35) +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



6.19 Combining Scatter Plots and Density Plots with Categorical Information

As with many of the univariate visualizations we explored, it is often useful to depict bivariate relationships as we change a categorical variable. To illustrate this, we'll go back to using the full iris data set.

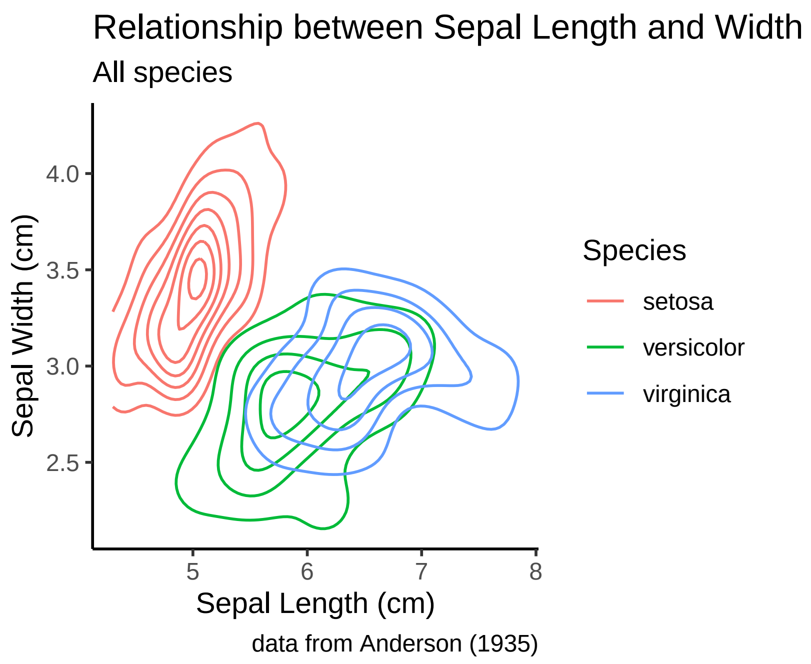
```r
all.sepals <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))

all.sepals +
  geom_point(aes(color = Species, shape = Species), size = 2, alpha = 0.6) +
  sepal.labels + labs(subtitle = "All species") +
  my.theme
```



Notice how in our aesthetic mapping we specified that both color and shape should be used to represent the species categories.

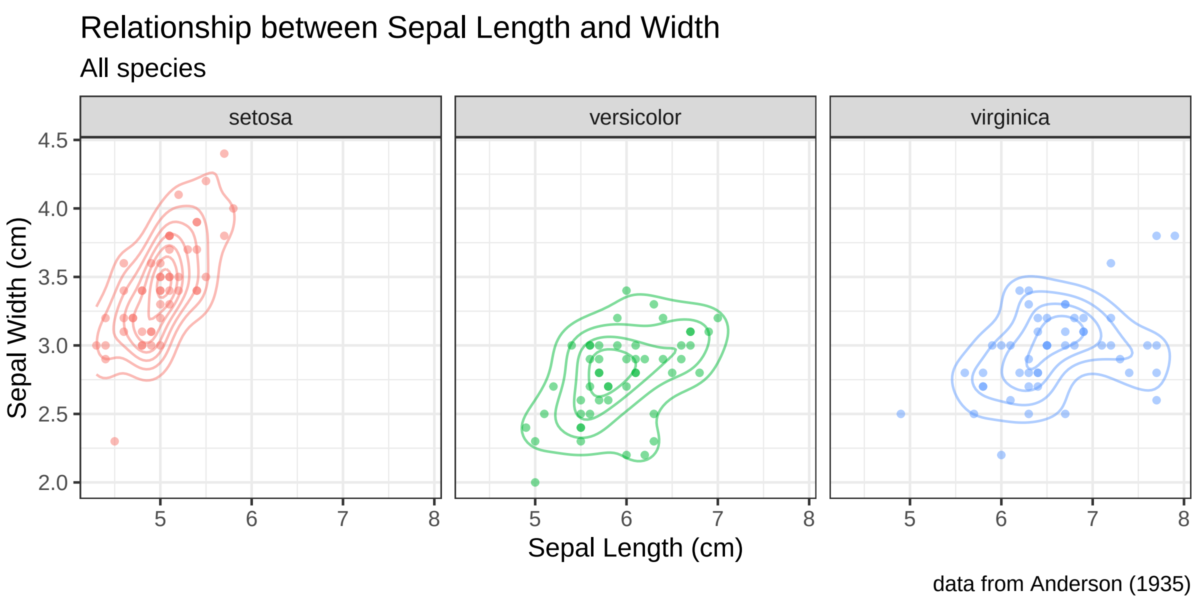
The same thing can be accomplished with a 2D density plot.

```
all.sepals +
  geom_density_2d(aes(color = Species)) +
  sepal.labels + labs(subtitle = "All species") +
  my.theme
```



As you can see, in the density plots above, when you have multiple categorical variables and there is significant overlap in the range of each sub-distribution, figures can become quite busy. As we've seen previously, faceting (conditioning) can be a good way to deal with this. Below a combination of scatter plots and 2D density plots, combined with faceting on the species variable.

```
all.sepals +
  geom_density_2d(aes(color = Species), alpha = 0.5) +
  geom_point(aes(color = Species), alpha=0.5, size=1) +
  facet_wrap(~ Species) +
  sepal.labels + labs(subtitle = "All species") +
  theme_bw() +
  theme(aspect.ratio = 1, legend.position = "none")  # get rid of legend
```



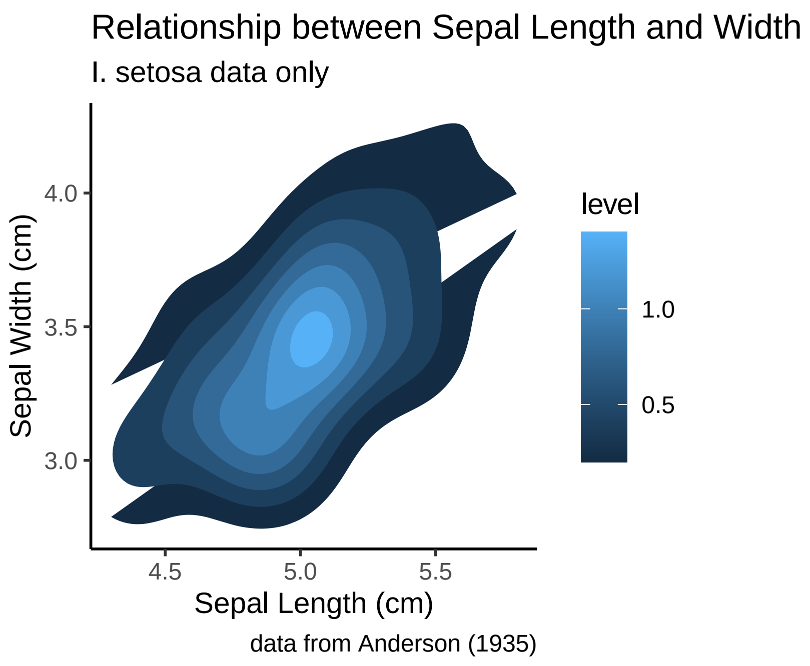
In this example I went back to using a theme that includes grid lines to facilitate more accurate comparisons of the distributions across the facets. I also got rid of the legend, because the information there was redundant.

6.20 Density plots with fill

Let's revisit our earlier single species 2D density plot. Instead of simply drawing contour lines, let's use color information to help guide the eye to areas of higher density. To draw filled contours, we use a sister function to `geom_density_2d` called `stat_density_2d`:

```
setosa.sepals +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```

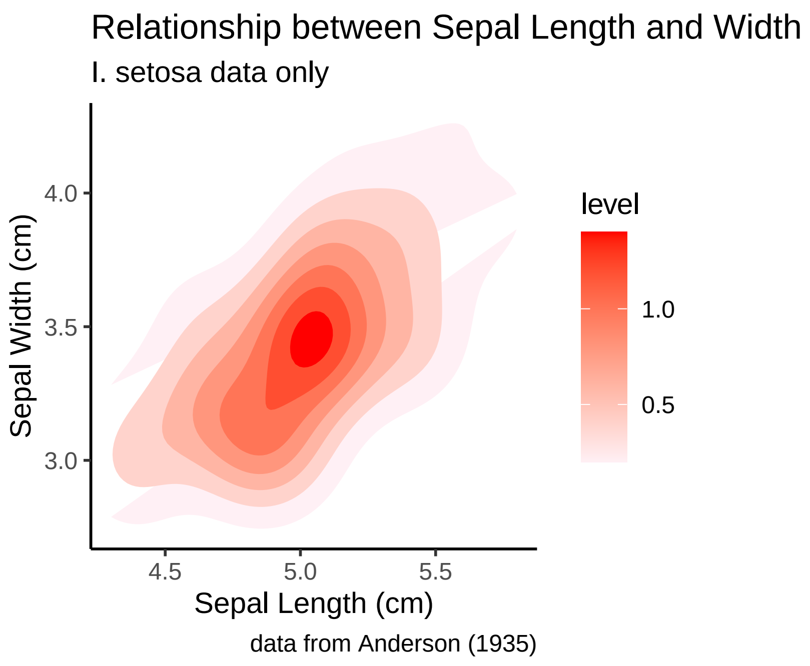
```
Warning: The dot-dot notation (`..level..`) was deprecated in ggplot2 3.4.0.
i Please use `after_stat(level)` instead.
```



Using the default color scale, areas of low density are drawn in dark blue, whereas areas of high density are drawn in light blue. I personally find this dark -to-light color scale non-intuitive for density data, and would prefer that darker regions indicate area of higher density. If we want to change the color scale, we can use the a scale function (in this case `scale_fill_continuous`) to set the color values used for the low and high values (this function we'll interpolate the intervening values for us).

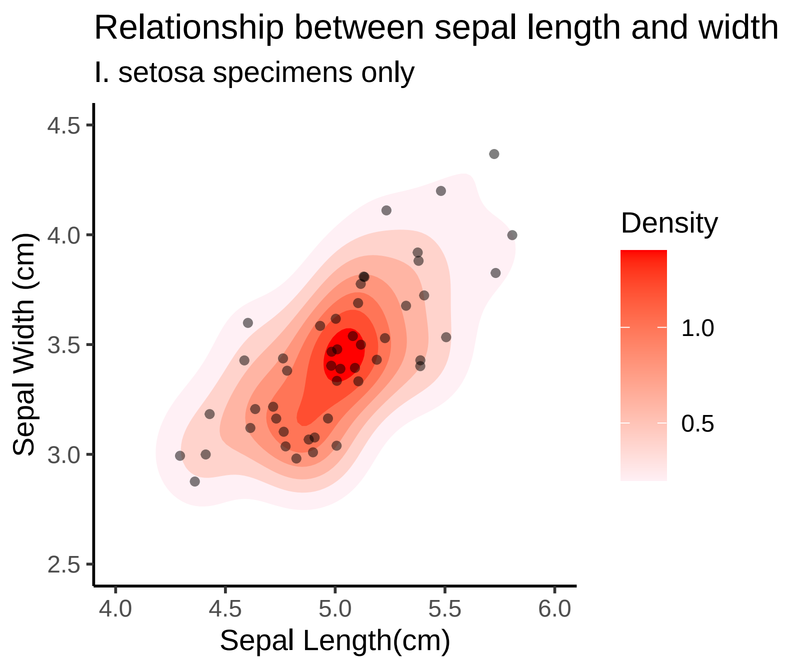
NOTE: when specifying color names, R accepts standard HTML color names (see the Wikipedia page on web colors for a list). We'll also see other ways to set color values in a later class session.

```r
setosa.sepals +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  # lavenderblush is the HTML standard name for a light purplish-pink color
  scale_fill_continuous(low="lavenderblush", high="red") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



The two contour plots we generated looked a little funny because the contours are cutoff due to the contour regions being outside the limits of the plot. To fix this, we can change the plot limits using the `lims` function as shown in the following code block. We'll also add the scatter (jittered) to the emphasize the relationship between the levels, and we'll change the title for the color legend on the right by specifying a text label associated with the fill arguments in the `labs` function.

```
setosa.sepals +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  scale_fill_continuous(low="lavenderblush", high="red") +
  geom_jitter(alpha=0.5, size = 1.1) +

  # customize labels, including legend label for fill
  labs(x = "Sepal Length(cm)", y = "Sepal Width (cm)",
       title = "Relationship between sepal length and width",
       subtitle = "I. setosa specimens only",
       fill = "Density") +

  # Set plot limits
  lims(x = c(4,6), y = c(2.5, 4.5)) +
  my.theme
```

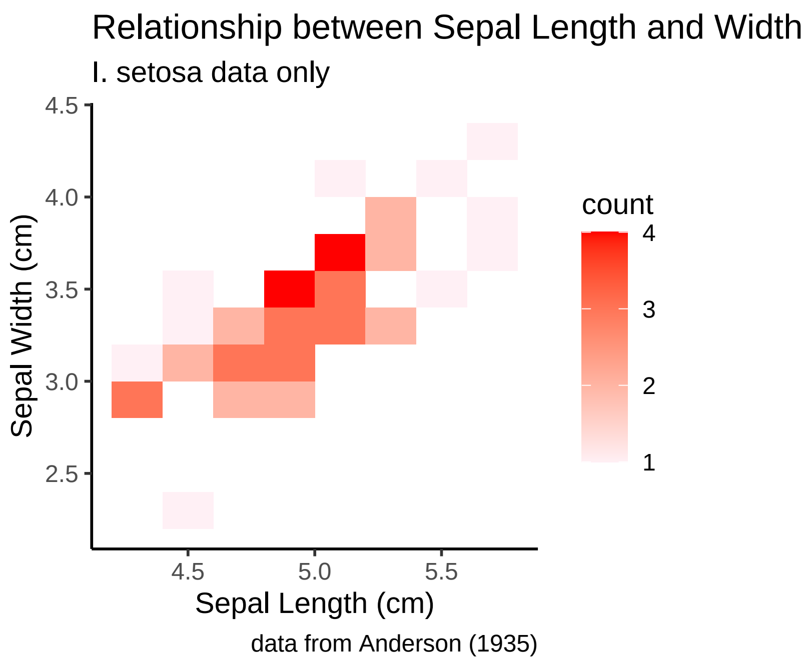


6.21 2D bin and hex plots

Two dimensional bin and hex plots are alterative ways to represent the joint density of points in the Cartesian plane. Here are examples of to generate these plot types. Compare them to our previous examples.

A 2D bin plot can be tought of as a 2D histogram:

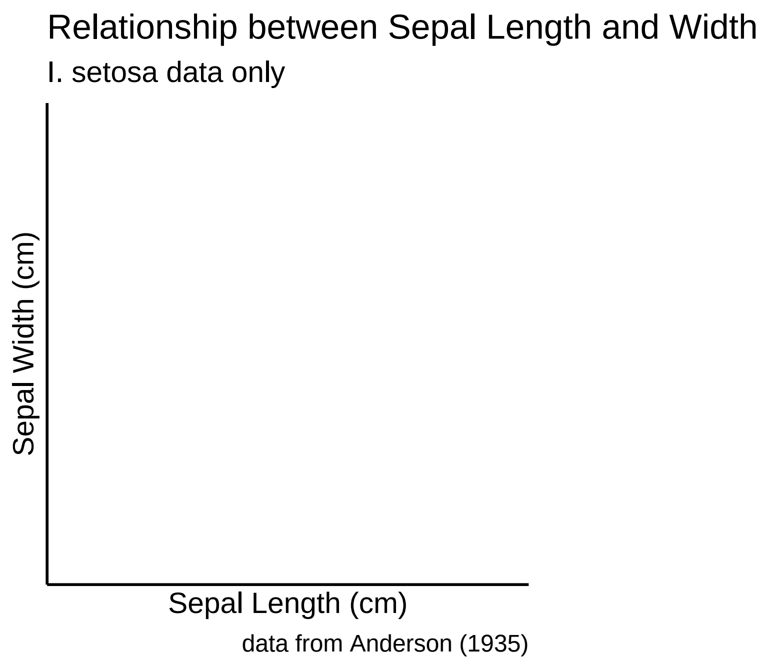
```
setosa.sepals +
  geom_bin2d(binwidth = 0.2) +
  scale_fill_continuous(low="lavenderblush", high="red") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



A hex plot is similar to a 2D bin plot but uses hexagonal regions instead of squares. Hexagonal bins are useful because they can avoid visual artefacts sometimes apparent with square bins:

```
setosa.sepals +
  geom_hex(binwidth = 0.2) +
  scale_fill_continuous(low="lavenderblush", high="red") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```

```
Warning: Computation failed in `stat_binhex()`
Caused by error in `compute_group()`:
! The package `hexbin` is required for `stat_binhex()`
```



6.22 The `cowplot` package

A common task when preparing visualizations for scientific presentations and manuscripts is combining different plots as subfigures of a larger figure. To accomplish this we'll use a package called `cowplot` that compliments the power of ggplot2. Install cowplot either via the command line or the R Studio GUI (see Section @ref(packages)).

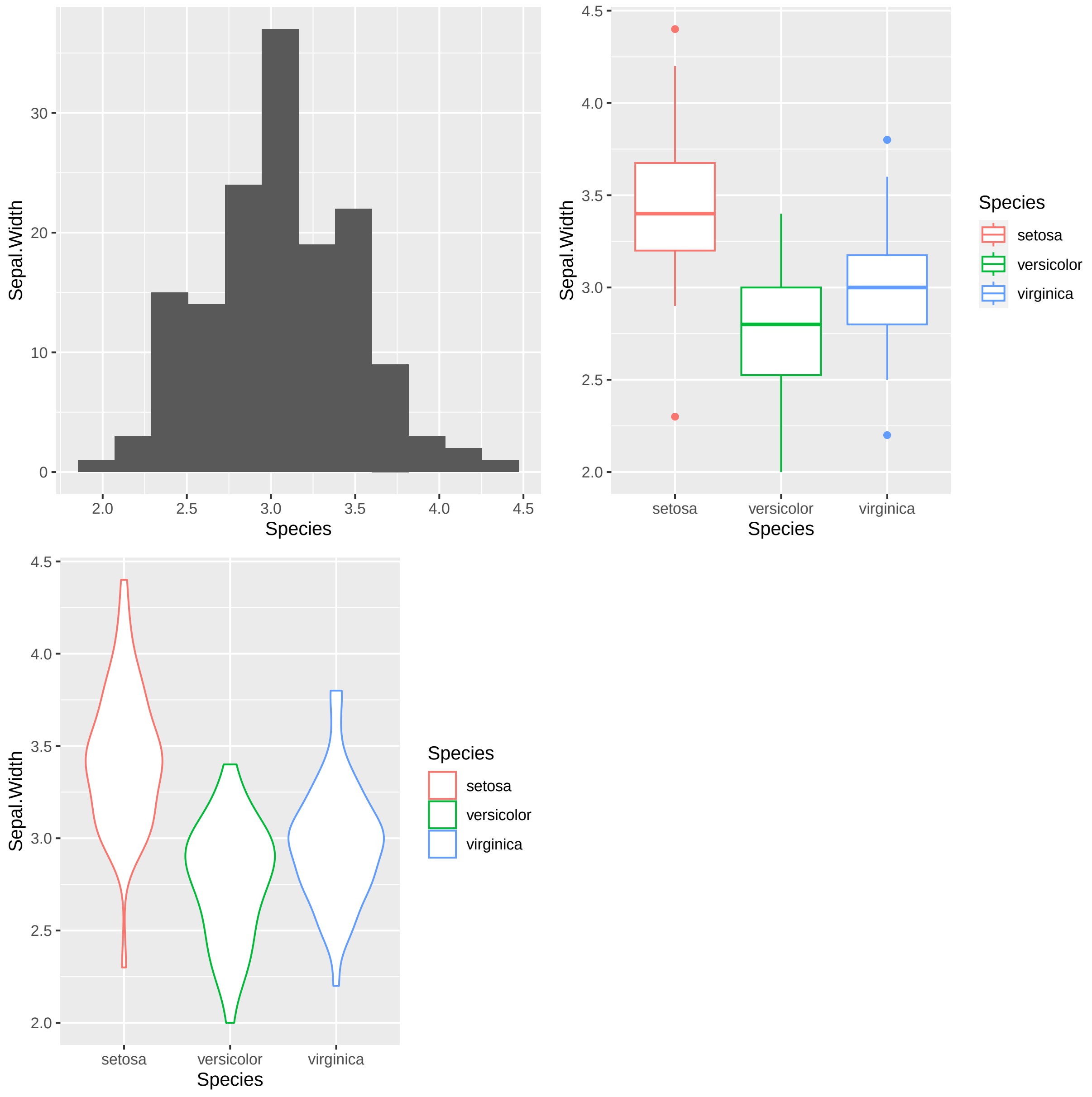
```
library(cowplot) # assumes package has been installed
```

`cowplot` allows us to create individual plots using ggplot, and then arrange them in a grid-like fashion with labels for each plot of interest, as you would typically see in publications. The core function of `cowplot` is `plot_grid()`, which allows the user to layout the sub-plots in an organized fashion and add labels as necesary.

To illustrate `plot_grid()` let's create three different representations of the distribution of sepal width in the irisu data set, and combine them into a single figure:

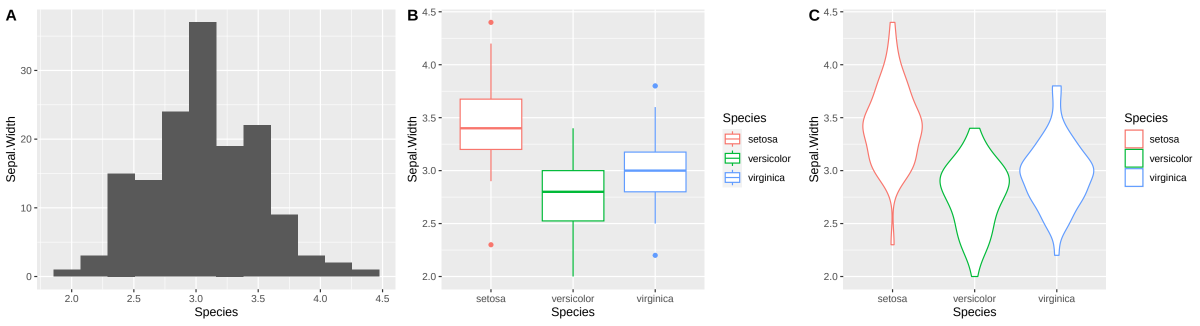
```
p <- ggplot(iris,
            mapping = aes(x = Species, y = Sepal.Width, color = Species))

# for the histogram we're going to override the mapping because
# geom_histogram only takes an x argument
```

```
plot.1 <- p +
  geom_histogram(bins=12,
                 mapping = aes(x = Sepal.Width), inherit.aes = FALSE)
plot.2 <- p + geom_boxplot()
plot.3 <- p + geom_violin()

plot_grid(plot.1, plot.2, plot.3)
```



If instead, we wanted to layout the plots in a single row we could change the call to `plot_grid` as so:

```r
plot_grid(plot.1, plot.2, plot.3,
          nrow = 1, labels = c("A", "B", "C"))
```



Notice we also added labels to our sub-plots.

References

Part III

Unix

7 Bash scripts

7.1 Writing shell scripts

Shell scripts are small programs written in the language of the shell; they're convenient for tying together a series of commands that you might otherwise type by hand into a repeatable and documented set of operations. Bash scripts also allow us to write new commands or programs that can be used in a manner similar to other "built-in" programs we've been using.

7.2 Bash scripts are picky

One annoying feature of bash scripting is that it's particularly picky about white space around commands and arguments, because essentially it tries to execute each line as if it was written at the command line itself (see). When following the examples below I recommend you be careful to follow the spacing I've used to avoid hard to diagnose syntax errors. For this reason, bash is *not* my preferred scripting environment. However it's so ubiquitous in bioinformatics that it's worth learning the basics of bash.

7.3 Simple script examples

7.3.1 Simple shell script 01

Let's start with the simplest possible shell script – one that simply writes a message to `stdout`.

Put the following code into a file named `tut01.sh`:

```
# tut01.sh
echo "Hello, shell world!"
```

Now you can call your shell script as so from the command line (assuming it's in your current working directory)

```
bash tut01.sh
```

This one line script simply called the `echo` command line tool to write text to `stdout`.

7.3.2 Simple shell script 02

That's a pretty boring shell script. Let's create a new version that takes as input an argument telling it what to print after the initial greeting. `$1` is how we designate the first argument from the command line.

```
# tut02.sh
echo "Hello, $1"
```

This expects an argument from the command line, so run this script as follows, substituting `Paul` as appropriate:

```
bash tut02.sh Paul
```

7.3.3 Simple shell script 03

Shell scripts can create variables to hold the output of commands. This is illustrated below in which we create a variable `input_reversed` to hold the value of a computation done on the first argument to the script:

```
# tut03.sh
# note that rev command reverses lines of input
input_reversed=$(echo $1 | rev)
echo "$1 reversed is $input_reversed"
```

And we again run it as:

```
bash tut03.sh "Twas brillig and the slithey toves..."
```

Notice that I wrapped the text in quotes. What happens if you don't include the quotes?

7.3.4 Simple shell script 04

We can get multiple arguments from the command line:

```
# tut04.sh
echo "The first argument was: $1"
echo "The second argument was: $2"
```

Or we can check the number of arguments:

```
# tut04a.sh
nargs=$#

if [[ $nargs -ne 2 ]]
then
    echo "I need two arguments!"
    exit
fi

echo "The first argument was: $1"
echo "The second argument was: $2"
```

7.4 Different ways of using Bash scripts

7.4.1 Reproducibility

- Did your analysis require a bunch of different steps? Could you reproduce what you did? Could you teach someone else to do it?
- Put each of the steps of your analysis into a bash script and save the file with a `.sh` ending. You or someone else can replicate your analysis by running `bash yourscriptname.sh`. Consider saving all such scripts you create for reproducibility purposes to a git repository.

Example:

```
# yeast_feature_counts.sh
#
# Purpose: count feature types in yeast genome annotation GFF file
# Output: a CSV table sorted by freq of feature

wget -q https://ftp.ncbi.nlm.nih.gov/genomes/refseq/fungi/Saccharomyces_cerevisiae/referen

gunzip GCF_000146045.2_R64_genomic.gff.gz

awk -F"\t" -v OFS="," \
    'NF == 9 {cts[$3] += 1}
    END {for (ftr in cts)
            print ftr, cts[ftr] }' \
    GCF_000146045.2_R64_genomic.gff |
sort -t, -k 2 -nr
```

You can execute this script by calling it as an argument to bash:

```
bash yeast_feature_counts.sh > yeast_features.csv
```

7.4.2 Reusability

- You developed a pipeline to solve one instance of a problem. Is this a type of computation you do repeatedly or are likely to do in the future on different inputs?
- Create a bash script "template" by re-writing the script to include variables for specifying input and output file name, key parameters, etc.

```
# feature_count_template.sh
#
# Purpose: count feature types in any GFF annotation
# output a CSV table sorted by freq of feature

# --- REPLACE THESE VARIABLES --- #

GFF_URL=""  # RefSeq URL Location
OUT_DIR="output"  # Directory for output, will be created if necessary

# -- END USER DEFINED VARIBLES -- #

# Create output directory if it doesn't exist
if [ ! -e ${OUT_DIR} ]; then
    mkdir -p ${OUT_DIR}
fi

# intermediate files written to OUT_DIR
gff_gz=${OUT_DIR}/gff.gz
gff_file=${OUT_DIR}/features.gff

# download and unzip the data if
if [ ! -f ${gff_file} ]; then
```

```
    wget ${GFF_URL} -O ${gff_gz}
    gunzip ${gff_gz} -c > ${gff_file}
fi

# run the analysis sending the results to stdout
awk -F"\t" -v OFS="," \
    'NF == 9 {cts[$3] += 1}
    END {for (ftr in cts)
            print ftr, cts[ftr] }' \
    ${gff_file}  |
sort -t, -k 2 -nr
```

If you, or someone you gave the script to, wanted to use this to run the analysis on another organism you would make a copy:

```
cp feature_count_template.sh ecoli_feature_count.sh
```

modify the `GFF_URL` and `OUT_DIR` lines appropriately:

```
# ecoli_feature_count.sh
#
# Purpose: count feature types in E coli genome
# Output: a CSV table sorted by freq of feature

GFF_URL="https://ftp.ncbi.nlm.nih.gov/genomes/refseq/bacteria/Escherichia_coli/reference/G

OUT_DIR="Escherichia_coli"

... <rest of template stays the same...>
```

and then run the script:

```
bash ecoli_feature_count.sh
```

7.4.3 Encapsulation and Abstraction

- You developed a pipeline that carries out a useful computation that you want to integrate into other analyses or pipelines
- Use a bash script to turn your pipeline into a command that can be integrated into other pipelines, following the Unix philosophy. From the user's perspective, you can think of this as "abstracting away" the details of how the analysis works. They no longer have

to understand the details of how the program or command works, they simply have to know what data to call it with and the form of the output.

Here we achieve this in a very simple way – by providing a way for our script to process arguments provided at the command line.

```bash
#!/usr/bin/env bash
# feature_count.sh -- download a GFF file from RefSeq and
#   count the feature types

nargs=$#

if [ $nargs -ne 2 ]; then
    echo "usage: $0 URL OUTDIR"
    exit 1
fi

GFF_URL=$1  # RefSeq URL Location
OUT_DIR=$2  # Directory for output, will be created if necessary

# Create output directory if it doesn't exist
if [ ! -e ${OUT_DIR} ]; then
    mkdir -p ${OUT_DIR}
fi

# intermediate files written to OUT_DIR
gff_gz=${OUT_DIR}/gff.gz
gff_file=${OUT_DIR}/features.gff

# download and unzip the data if
if [ ! -f ${gff_file} ]; then
    wget ${GFF_URL} -O ${gff_gz}
    gunzip ${gff_gz} -c > ${gff_file}
fi

# run the analysis sending the results to stdout
awk -F"\t" -v OFS="," \
    'NF == 9 {cts[$3] += 1}
    END {for (ftr in cts)
            print ftr, cts[ftr] }' \
    ${gff_file}  |
sort -t, -k 2 -nr
```

Make this script executable (`chmod +x feature_count.sh`). You can call it directly from your working directory or add it to your `PATH`. Having done this you can use this script by specifying a URL and a directory name to write the output files to. Since the URL is long I assign it to a variable:

```
celegans_url=https://ftp.ncbi.nlm.nih.gov/genomes/refseq/invertebrate/Caenorhabditis_elega
```

And then call the bash script

```
feature_count.sh $celegans_url celegans
```

7.4.3.1 Back to reproducibility

By encapsulating our pipeline we've made it reusable and generalizable to analyses of arbitrary GFF annotation files. However, we've lost an element of reproducibility because we're no longer documenting the data sources used when we invoke it at the command line. Luckily it's easy to restore that with – you guessed it – another bash script!

To illustrate this, let's assume we wanted to compare feature counts across multiple genomes. Let's put species names and corresponding RefSeq URLs in a CSV file called `genomelist.csv`:

```
Saccharomyces_cerevisiae,https://ftp.ncbi.nlm.nih.gov/genomes/refseq/fungi/Saccharomyces_c
Escherichia_coli,https://ftp.ncbi.nlm.nih.gov/genomes/refseq/bacteria/Escherichia_coli/ref
Caenorhabditis_elegans,https://ftp.ncbi.nlm.nih.gov/genomes/refseq/invertebrate/Caenorhabd
```

Now we can generate feature counts for each genome by creating a bash script called `multi_species_count.sh` that includes a call to `parallel`:

```
# multi_species_count.sh

parallel --colsep="," 'mkdir -p {1}; ./feature_count.sh {2} {1} > {1}/ftrcounts.csv' ::::
```

and run this as:

```
bash multi_species_count.sh
```

For each species name in your input list, there should now be a corresponding directory that contains a `ftrcounts.csv` file with the respective counts:

```
$ ls -1 */ftrcounts.csv
Caenorhabditis_elegans/ftrcounts.csv
Escherichia_coli/ftrcounts.csv
Saccharomyces_cerevisiae/ftrcounts.csv
```

In terms of reproducibility, our full analysis now requires three files:

1. The `feature_count.sh` script
2. The list of genomes we're processing: `genomelist.csv`
3. Our `multi_species_count.sh` that processes the information in (2) through the pipeline in (1)

There is a slight increase in complexity versus our single species script, but with the following advantages:

1. It's trivial to extend our analysis to include new species/genomes of interest – we simply add additional species names and URLs to `genomelist.csv`

2. We can run this analysis efficiently for many genomes at once by adding the argument `--jobs=n` to the call to `parallel` where `n` is the number of threads we want to run simultaneously (assuming that you have access to a multi-CPU machine or a compute cluster)

7.5 Other resources

The above is a short intro to bash scripting. Some other useful resources include:

- https://linuxconfig.org/bash-scripting-tutorial-for-beginners – provides a good overview of bash
- https://www.shellscript.sh/ – an in-depth tutorial on bash scripting

8 Awk

Awk (`awk`) is a programming language designed for processing structured text files. You can use it to write short one liners or to write full blown programs. We'll use Awk to illustrate how you might transitions from simple command line usage into slightly more complicated scripts.

8.1 Simple examples

One simple thing we can do with Awk is to use it to re-order fields in a structured data file:

```
awk '{print $2, $1, $3}' columns.txt
```

The text in quotes represents the Awk program which we're apply to the file `columns.txt`. The dollar signs followed by numbers refer to the fields of the file. With it's default setting `awk` operates line by line, so you can interpret the above statement as saying:

> "for each line, print the fields 2, 1, and 3".

8.2 Awk delimiters

By default, Awk recognizes any non printing character including spaces to delineate fields/columns. If we want to be specific about the character used as a delimiter, we can set the `-F` or `--field-separator` option. For example to use tabs as the delimiter we could do:

```
awk -F'\t' '{print $2, $1, $3}' columns.txt
```

8.3 `pattern {action}` syntax

The basic syntax of `awk` is often depicted in the form `pattern {action}`. The above command only specified an action, so it was applied to every line. By contrast, in the example below we specify both a pattern and an action. The pattern can be read as – "if the 2nd field is 'chromosome' ". For all lines that match that pattern the corresponding action is applied; in this case the action is "print fields 1 and 4" (the chromosome name and its length):

```
awk '$2=="chromosome" {print $1, $4}' yeast_features.txt
```

Here's another `pattern {action}` pair that shows how we could find all gene features with length less than 300 and count them:

```
awk '$2 == "gene" && ($4 - $3 + 1) < 300 {print $0 }' yeast_features.txt | wc -l
```

`&&` is the AND operator. Read this as "if the 2nd field is 'gene' AND the length of the feature is less than 300." (Note: we add one to the calculation of length because the feature coordinates are inclusive of both the start and end coordinate).

In this last example we add one more condition – we use a regular expression against the 6th field (`$6 ~ /../`) to look for the string "orf_classification=Dubious". The results indicate that about a third of these small gene features have been classified as "dubious" (i.e. may not actually code for proteins).

```
awk '$2 == "gene" && ($4 - $3 + 1) < 300 && $6 ~ /orf_classification=Dubious/ {print $0 }'
```

8.4 Multiple rules and Awk scripts

So far our `pattern {action}` rules have matched focused on single patterns. However, you can specify multiple matching rules and do different computations depending on which one matches. For example, if we wanted to count genes and mRNAs simultaneously in an annotation file and then print out the counts at the end, we can construct a awk call with three rules:

```
awk '$2 == "gene" {gene_ct += 1} $2 == "mRNA" {mRNA_ct += 1} END {print gene_ct, mRNA_ct}'
```

At this point the command line is getting a little out of control so it would be better to put the rules into a file we can call. In your editor create a file called `count_genes_mRNAs.awk` with the following contents:

```
$2 == "gene" {
    gene_ct += 1
}

$2 == "mRNA" {
    mRNA_ct += 1
}
```

```
## called once all lines have been processed
END {
    print gene_ct, mRNA_ct
}
```

This file works exactly like our command line version, but it has the advantages that it's easier to read and thus easier to debug or modify. To call this Awk program from the command line we use the `-f` option to specify the script:

```
awk -f count-genes-mRNAs.awk yeast_features.txt
```

8.5 Variables in Awk

The prior example introduced the use of "variables" in Awk. Variables store values; in Awk that are only two variable types, strings and numbers.

8.5.1 Numerical variables

In the example above `gene_ct` and `mRNA_ct` are numerical variables. In Awk, when a numerical value is first created (initialized) it's automatically assigned the value zero. In this example we used these variables in lines that look like `gene_ct += 1`; the `+=` operator is a short-hand way of saying "add the value on the right to the value of the variable on the left" so this statement is equivalent to the slightly wordier `gene_ct = gene_ct + 1`.

8.5.2 String variables

String variables are created by wrapping characters in double quotes.

```
awk 'BEGIN {a = "Hello"; b = "World"; print a b}'
```

You can get the length of a string using the `length()` function:

```
awk 'BEGIN {a = "Hello"; print "The length of " a " is " length(a)}'
```

The GNU Awk implementation (`gawk`) provides a wide variety of string functions. Some of the most useful include: `substr()`, `split()`, and `gsub()/sub()` . Here's an example using `substr()`:

```
awk '
BEGIN {
  s = "Jabberwocky"
  print substr(s, 4, 7) # arguments are string, start, length
}'
```

8.6 Arrays in Awk

Most programming languages have a core set of "data structures" which specify different ways of storing and accessing values. In Awk the core data structure is called an array. It has some superficial similarities to arrays (sometimes called lists) in other languages, but also has some features that are fairly unique to Awk.

An array can be thought of being made up of a contiguous set of slots or positions where we can store values. Each slot has an associated label that we call its index (plural indices). In most languages array indices are integers, with the first position indexed by successive integer values (some languages start indexing the 0, others with 1). In Awk, array indices can be either numbers or strings. This makes arrays in Awk a little like a combination between what other languages might call an array (or list) and a dictionary (hash map).

Here are some illustrations of using arrays in Awk:

- The `split()` function splits a string based on a delimiter and returns the substrings in an array that is indexed by integer position

  ```
  awk '
  BEGIN {
    name = "kebab-case-name"
    split(name, parts, "-")  # parts array is created
    print "part 1: " parts[1]
    print "part 2: " parts[2]
    print "part 3: " parts[3]
  }'
  ```

  Typically you'd use a for-loop to iterate over the indices of `parts`

  ```
  awk '
  BEGIN {
    name = "kebab-case-name"
    # split() return the number of substrings it creates
    n = split(name, parts, "-")
  ```

```
  # for-loop iterating over integer indices
  for (i = 1; i <= n; ++i )
    print "part " i ": " parts[i]
}'
```

- uniqct.awk – a program that outputs a table listing the unique set of lines in the input, and the number of times that each line appears

  ```
  # this rule applies to every line, $0 returns the whole line
  { counts[$0] += 1 }

  END{
    # for loop with implicit indices
    for (idx in counts)
      print idx, counts[idx]
  }
  ```

  This code might be used like so:

  ```
  cut -f 2 yeast_features.txt | awk -f uniqct.awk
  ```

For a full exposition on Awk's arrays, read the Gawk manual on the Basics of Arrays. Here I illustrate some basic uses of arrays.

8.7 Some useful Awk constructs

- `BEGIN {action}` – do the specified action before reading any input
- `END {action}` – do the specified action after reading all input
- `next` – the next statement causes Awk to stop processing the current record (line) and move onto the next record.
- `exit` – the exit statement immediately stops execution of any rules and jumps to the `END` rule. Useful for terminating processing based on a condition.
- Record and field counts:
  - `FNR` and `NR` – `FNR` gives the current record number in the current file (awk can process multiple files simultaneously); `NR` gives the total number of records seen so far. When processing a single file, `FNR` == `NR`
  - `NF` – gives the number of fields in the current record(line)

```
awk -F "\t" `{print "There are", NF, "fields in line", FNR}` yeast_features.txt
```

- Field separators – both the input field separator (`FS`) and the output field separator (`OFS`) can be specified in an Awk program. This is usually done in a `BEGIN` rule:
  - `firstlast_csv.awk` – outputs first and last fields of input file in CSV separated fields

    ```
    BEGIN {
      OFS = ","
    }

    {print $1, $NF} # applies to every line
    ```

- Matching and extracting information using regular expressions
  - `match(field, regex)` – this function find the specified regex in the given field. Per the gawk manual: "Returns the character position (index) at which that substring begins (one, if it starts at the beginning of string). If no match is found, return zero."

    The following example returns all features in a GFF file for which the attribute field (column 9) includes an ID designation.

    ```
    awk 'match($9, "ID=([^;]+)") {print $0}' yeast.gff
    ```

  - `gawk` provides a more powerful version of `match` where the matches can take an optional array as the third argument. As described in the Gawk manual:

    > If array is present, it is cleared, and then the zeroth element of array is set to the entire portion of string matched by regexp. If regexp contains parentheses, the integer-indexed elements of array are set to contain the portion of string matching the corresponding parenthesized subexpression.

    Here's an example of using the gawk `match` function to extract the ID attribute for every gene feature and create a new table giving ID, landmark, and start and end coordinates.

    ```
    $3 == "gene" {
        id = ""
        if (match($9, "ID=([^;]+);", matchvar))
            id = matchvar[1]
        print id, $1, $4, $5
    ```

```
}
```

If we put this in `idgenes.awk` we can call it as:

```
awk -f idgenes.awk yeast.gff
```