R Programming for Data Science

STSCI3040 - STSCI 5040

Fall 2023

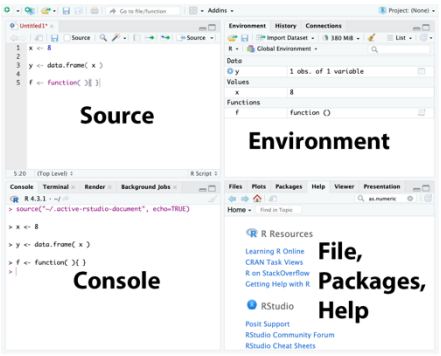
This page intentionally blank

Statistics courses usually use clean and well-behaved data, this leaves many unprepared for the messiness and chaos of data in the real world. This course aims to prepare students for dealing with data using the R programming language. The introduction will overview the basic R syntax, foundational R programming concepts such as data types, vectors arithmetic, and indexing, importing data into R from different file formats. The data wrangling topics include how to tidy data using the tidyverse to better facilitate analysis, string processing with regular expressions and with dates and times as file formats, web scraping, and text mining. Data visualization topics will cover visualization principles, the use of ggplot2 to create custom plots, and how to communicate data-driven findings.

Chapter 1: RStudio & R

To make use of **R** for data science, you need to install the **R** programming language on your computer. Versions of **R** can be found at https://cran.r-project.org**.** The installation comes with a program that acts as an **R**-language terminal. You type in a command and the command is executed. This program is all you need to write code in **R**. However, can be cumbersome to use. This is where **RStudio** come in. **RStudio** builds an environment around the basic **R** application. This environment consists of a series of panes that allow all the functionality of the *R* -programming language to be managed easier.

When using, **RStudio** will display four panes of information[1] These four panes are laid out as follows[2] :



Each of these panes can have several tabs. Not all of these tabs will be discussed.

Console

The **Console** pane will contain several different tabs. The most important of which is the **Console** tab. This tab acts as the stand-alone **R**-language terminal that was mentioned above. Essentially, it acts as a calculator. At the prompt, a command is typed in, once the enter/return key is pressed, the command is executed. This has the effect of giving you a very direct connection to execturing **R** code, but it is not very efficient. One command needs to be entered at a time[3]. However, if an error is made, it may be the case that each preceding line needs to be reentered. This is what make the **Source** pane very useful.

Source

The **Source** pane will hold as many tabs as you can create. For the most part, each tab will either be a text editor or a table viewer. The table viewer shows the content of a dataset that has been loaded. The text editors will be the most useful. A new tab (text editor) can be opened in the **Source** pane by selecting *File* and then *New File* in the drop-down menus. Under *New File*, a large selection of options are available.

The option that is chosen under *New File* depends upon the purpose of the work produced. Each of the options will produce a text with with a different file extension when saved[4]. The extension that is attached to these files has an effect on the options that are visible in **RStudio**, how the contents are displayed, and the structure of how the content is written.

The two options under *New File* that will be focused on here:

- R Script
- R Markdown…

R Script:
An **R Script** is simple a text document with different **R** commands written on different lines[5]. As was said before, using the **Console** directly, each command must be typed in one at a time. Which means that each time a collection of commands needs to be executed, they need to be retyped into the **Console** prompt. With code written into an **R Script**, a *Run* command can be used to send all, or a selected portion, of an **R Script** into the **Console**. This alleviated retyping a collection of commands into the Console. The **R Script** can be saved as a text file with a *.R* extention. This allows to the set of commands to be saved and possibly transported to other locations. It should be kept in mind, that the output of an

[1] There will be times you only see three panes. Usually, this the first time you open **RStudio** or after you have closed all tabs in the **Source** pane.
[2] The layout of these panes can be changed in the *Global Options* of **RStudio** (under the *Tools* dropdown menu).
[3] Multiples can be entered if they are seperated by a semicolon
[4] Even though these will not have a **.txt** extension, such files will be text files. They can be opened and edited in any text editor
[5] Choosing **R Script** indicates that **R** code will be used. Selecting one of the other script options would indicate that a different programming language is being used.

executed **R Script** does not become part of the **R Script**. Its output will appear in the **Console** or one of the other panes in **R**.

R Markdown…:
Selecting **R Markdown…** will produce a text document with a *.Rmd* extension. Unlike an **R Script**, creating an **R Markdown** file is only a portion of the created output. With an **R Script**, the ***.R** file is the output to be shared. An **R Markdown** file is only part of the output to be shared. An **R Markdown** file when properly processed can be used to produce a formatted Word document, a PDF, an HTML file for a report. Within an **R Markdown**, the plain text of a report can be written and using a special syntax **R** code can be included to perform computations and create graphics that are needed and incorporated into a report. All of this can be done with one document. There is no need to use one program to preform computations, a second program to make graphics, and a third to mix the two together to make a report. An **R Markdown** can contain all of these at once. Except for the datafiles, an **R Markdown** can contain all the information to create a report. The advantage of this is two fold:

- A reviewer only interested in the results can examine a version of the report (say a word document) with the code hidden (if desired).
- If a modification to the computations,graphics, or data is needed , the ***.Rmd** file can easily be changes or data updated, and the ***.Rmd** can be reprocessed to update the final report.

Environment

Before discussing the **Environment** pane, the idea of an **environment** needs to be explained. When using *R*, objects that are not built-into the system can be created. These objects may contain data, functions(collections of commands), or other information. These objects when created live in an **Environment** until that **environment** until they are removed from the environment or the environment is deleted. The interesting part of this, is that more than one **environment** can exist. The **Global Environment** is the most visible **environment**. Its contents are visible in the **Environment** tab in the **Environment** pane. The contents of the **Global Environment** are objects that are created when using the **Console** directly, running **R** code from an **R Script**, or directly running code within an **R Markdown** without processing then entire **R Markdown** to create a report[6].

File/Packages/Help

This last pane has many useful tabs:

1. Help: This tab gives searchable access to the documentation for all the functions that are available for use in your code that you have not specifically written. The documentation usually indicates what inputs and outputs are needed and give examples/vignettes of using the object.

2. Plots: This tab is used to display plots that you have created. Options are available for saving them using menu options. If viewing plots is important, care should be taken to not make this pane too small. Otherwise, errors will appear or odd graphics can appear.

3. Files: This tab gives a file browser for the computer being used.

4. Packages: When initially installed, the **R** programming language has a limited set of capabilities. Other **R** users have taken those basic capabilities and used them to create packages. A package is a collection of new commands that can be loaded, if desired, and made available for use. Additional packages are optional. If a new capability it not needed, there is no reason to make it available. Some packages extend the basic linear regression capabilities, others give more graphics options, some add capabilities for formatting text. The **packages** tab keeps track of which packages a user has downloaded, which packages are currently active, and give options for downloading new packages.[7]

---

[6] As was said, other **environments** can be made. A common way second **environment** is temporarily created is when an **R Markdown** is processed to create a report as a Word document, a PDF, or an HTML file. This second **environment** will not be visible to the user as the report is being created. However, it is important to be aware that it exists. As the report is being created, errors may occur because certain objects may exist in the **Global Environment** and not in the second **environment**. Because of this, it is important when writing **R Markdown** files, that all object that are used within it, are coded as part of the **R Markdown** file, before they are used

[7] It is important to keep in mind that some packages might overwrite existing commands. Depending upon you need, you may want to use the version from a package or you may want to use the existing version. This is why there is a capability to load and unload packages.

Chapter 2: Basic RMarkdown

An **R Markdown** file is text file that can be processed to produce a Word, a PDF, or an HTML document. To process one correctly several packages need to be installed and loaded. The most important are **rmarkdown** and **knitr**. In fact, the process of producing the finished Word, a PDF, or an HTML document is usually referred to as KNITTING.

An **R Markdown** file will consist of three types of text:

- YAML header,
- plain text, and
- code chunks.

A brief example can be seen on the next page.

YAML Header

The **YAML header** is the first thing found in most **R Markdown** documents. It will be located at the top of the document with its contents surrounded above and below by three dashes set on their own line. Between the dashed lines several options can be set. To do so, the options name is listed followed by a colon and a space. Then the value for that option is typed it. Common options are used are: title, subtitle, author, date, and output. When an **R Markdown** document is knitted together to produce a Word, a PDF, or an HTML document, the values for the first four options will appear on the first page of the resulting output. However, the exact appearance will differ based upon the output values starting with one of these word_document, pdf_document, html_document[8]. On lines after the output line, other options can be added.

Code Chunks

Code Chunks are sections of an **R Markdown** file that contain **R** code. They are sections of text surrounded by two separate lines that start with (at least) three tick marks. The initial set of three tick marks are followed immediately by a set of curly braces. The contents of the curly braces, separated by commas are:

- the programming language being used. The code contained in a chunk can be any of a number of different programming languages. Therefore, the language being used needs to be identified. As a course in *R*, the identifier is a lower case **r**[9].

- the name of the code chunk. This is optional. It is useful if a code chunk will be used more than once within a document. The name can be referenced later without recreating the entire code chunl. However, if two code chunks are named, they must be given different names.

- evaluation options

Three code chunks appear in the following example. They are *setup*, *cars*, and *pressure*. The *setup* code chunk is usually the first chunk that appears after the YAML header. Code within it is usually used to set default options for all subsequent codes. This is done by using the $knitr::opts\_chunk\$set(\ )$ command. The other two code chunks contain code to summarize and graph some data[10].

When the text of an R Markdown file is displayed in the source window, it will be highlighted. Additionally, in the upper right hand corner of each code chunk are some buttons. The gear gives you a menu of some options that can be set for the current chunk. The downward facing arrow will execute all previous code chunks. The rightward facing arrow will execute the current chunk[11]

---

[8] Typing in this line is not all that necessary. A **Knit** button appears are the top of the **Source** window when an **R Markdown** file is loaded. It has an option for the type of knitted output and it will fill it in automatically.
[9] Different coding languages can not be mixed inside a code chunk.
[10] The meaning of the code contained in these chunks will become apparent as the course progresses.
[11] When a R Markdown file is processed, code chunks chunks are executed from the top down. Without knitting your code, you can use the first arrow to reload previous chunks, if they have been changed, while you work on the current chunk.

Text

The remainder of the example is the text that would make up a report. Careful examination will make apparent that this is not simply text. There are lines that start with pound signs (#) and words surrounded by symbols like asterisks or angles.(*,< ,>). Usa of these symbols, and others, using the proper formatting will introduce headers, font formats, tables, and other items in the knitted result. In this example **Knit** indicates that the word Knit should be bold in the final result.

Example R Markdown Text

This text listed below is a basic template that appears when you create a new R Markdown file.

---
title: “Untitled”

author: “Jeremy Entner”

date: “8/25/2021”

output: word_document

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
```{r cars}
summary(cars)
```
```

Including Plots

You can also embed plots, for example:

```
```{r pressure, echo=FALSE}
plot(pressure)
```
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Chapter 3: Numerical Data

Numerical Data comes in THREE forms:

1. Integer
2. Floating Point/Double
3. Complex

We won't worry about complex numbers. An integer is any whole number, positive or negative. As an integer, it will be recognizable as a number followed by a capital L. For example, the integer three would appear as 3L.

A floating point/double number is any whole number or fractional number. The floating point/double version of the number three would simply appear as 3. The real difference between these two types is determined by how the computer stores the numbers. For the most part, we can ignore the difference between integers and floating point numbers.

Numerical Vectors

A **numerical vector** is a container in R that is used to hold numerical data. An individual numerical vector can contain zero or more numerical data values[12]. Collections of numerical vectors can be combined using the **combine function** *c*( ). This will produce a new vector whose elements are ordered as entered in *c*( ). Make sure to separate each numerical data value with a comma.

```
c(1, 4, -10, 3.5)

[1]   1.0   4.0 -10.0   3.5
```

Mathematic Operations

Many standard mathematical operations can be performed on numerical vectors. The most basic would be addition.

```
1 + 2.3

[1] 3.3
```

While it looks like we added two number, what really happened was that R added together two numerical vectors that contained a single value each, namely 1 and 2.3.[13] If we had wanted to add two longer vectors together, we could have done so

```
# Addition - ?Arithmetic
c(0, 2, 2.4) + c(1, -2, 0.6)

[1] 1 0 3
```

The plus sign + performs a **vectorized** addition. Corresponding pairs of values in each vector are added together. The first pair, 0 and 1, are added to give the first result, (1). The second pair, 2 and -2, are added to give the second result (0). Finally, the third pair, 2.4 and 0.6, result in (4).

Some basic mathematical operations that are vectorized are demonstrated below. [14].

```
# Subtraction - ?Arithmetic
c(0, 2, 2.4) - c(1, -2, 0.6)

[1] -1.0  4.0  1.8

# Multiplication - ?Arithmetic
c(0, 2, 2.4) * c(1, -2, 0.6)

[1]  0.00 -4.00  1.44

# Division - ?Arithmetic
c(0, 2, 2.4)/c(1, -2, 0.6)

[1]  0 -1  4
```

---

[12] at times each individual value may be called an **element**
[13] The combine function is not needed for single values.
[14] Multiple operations can be combined using parentheses.The order of operations can be described using PEMDAS: Parentheses/Grouping, Exponents, Multiplication, Division, Addition, Subtraction

Some more complex operations are also vectorized. However, for clarity, we will demonstrate them with single valued vectors.

```
# Absolute Value - ?MathFun
abs(-7.1)

[1] 7.1

# Square Root
sqrt(4)

[1] 2

# Exponents - ?Arithmetic
3^2

[1] 9

# Natural Log & Exponential- ?log
log(4.8)

[1] 1.568616

exp(1.569)

[1] 4.801844

# Trig functions - ?Trig
cos(2)

[1] -0.4161468

# Quotient and Remainder - ?Arithmetic
10%/%3

[1] 3

10%%3

[1] 1
```

Recycling

Some operations with two inputs **recycle** (repeat) elements the shorter vector, until it is long enough to perform the vectorized operation.

```
c(0, 2, 3.4, 10) + c(1, 0)

[1]  1.0  2.0  4.4 10.0
```

In the first line, R repeated the vector with two elements. The second vector was treated like c(1, 0, 1, 0).

```
c(0, 2, 3.4, 10) + c(1, 0, 7)

Warning in c(0, 2, 3.4, 10) + c(1, 0, 7): longer object length is not a multiple of shorter object length

[1]  1.0  2.0 10.4 11.0
```

In this example, a warning is given. It indicates that the longer vector is not a multiple of the shorter vector, in terms of length. Three does not divide evenly into four. The vector of length three was partially recycled as c(1, 0, 7, 1).

Two Interesting Facts

1. The number of digits displayed can be controlled. However, It doesn't change the representd value
2. The number π is stored in R. It can be accessed by typing *pi*.

```
options(digits = 20)  #?options
pi  #?Constants

[1] 3.1415926535897931116
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Using the help menu, determine the commands for computing the sine and tangent of a value.

2. Determine the following:

    a. $19^{2.2}$
    b. $\frac{19^{\ln 9}}{18} + 15$
    c. $\sqrt{24}$
    d. $\sin(6\pi)/tan(7\pi)$

3. Which produce equal/identical results.

    a. c($1^2, 2^3, 4^6, 8^{-1}$)
    b. c($1\pi, 2\pi, 3\pi, 10\pi$)
    c. c(1 ,2 ,4 , 8)^c(2, 3, 6, -1)
    d. c(12, 5, -4, 2) + c(1, 2)
    e. c(10, 2, 3, 1)*π
    f. c(12, 5, -4, 3) + c(1, 2, 1)

4. Convert these temperatures (32, 212, 100, -40) from Fahrenheit to Celsius. Do this with one line of code. (You should look up the conversion formula.)

5. Compute the following using two vectors:

    a. The quotient of 5 divided by 3. The quotient of 2 divided by 4. The quotient of 228 divided by 8.
    b. Two squared. Three cubed. Four to the forth. Five fifth.

6. Compute the following making use of recycling.(Shrink the given vectors)

    a. c(1, 2, 3, 4, 5, 6) * c(0, 4, 6, 0, 4, 6) - c(1, 1, 1, 1, 1, 1)
    b. c(1, 2, 3, 4, 5, 6) * c(0, 4, 6, 0, 4, 6) - c(1, 1, 1, 1, 1, 1, 1)
    c. c(1, 2, 3, 4, 5, 6) * c(0, 4, 6, 1, 9, 0) - c(1, 1, 1, 1, 1, 1)

7. Compute the following without using recycling.(Expand the given vectors as needed)

    a. c(1, 2) + c(0, 10, -1) * 20
    b. c(1, 2) + (c(0, 10, -1) * 20)
    c. c(1, 2) / c(1, 2, 1, 2) - c(1, -1, 1, -1, 1, -1)

This page will eventually be filled with problems also.

Chapter 4: Assignment Operator

The **assignment operator** $<-$ allows you to efficiently keep track of values used in computations and results produced by computations. Consider adding these vectors $c(0,2,2.4)$ and $c(1,-2,0.6)$

```
c(0, 2, 2.4) + c(1, -2, 0.6)

[1] 1 0 3
```

The result is $c(1,0,3)$. Suppose you wanted to save this result or perform further computations using it. The operator will allow you to do this. In particular, it will allow this without the need to retype the vector.

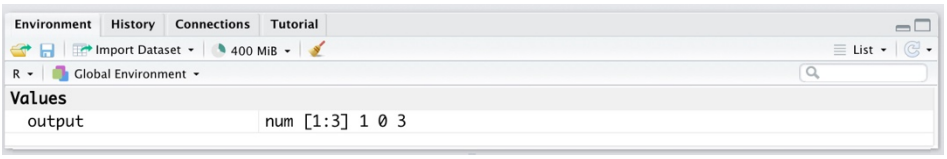
$c(1,0,3)$ is short, only three values. A result with 1000 values would not be very easy to retype.

The assignment operator can be used to attach a variable name to the result.

```
output <- c(0, 2, 2.4) + c(1, -2, 0.6)
```

To read this line of code, start on the right of the $<-$ sign. First, $c(0,2,2.4) + c(1,-2,0.6)$ is computed. Second, the assignment operator $<-$ assigns the result to a variable called *output*. Notice two things about this:

1. The result is not displayed.
2. A new value called *output* appeared in the environment pane.



Once a variable name has been assigned to a value, the two become synonymous. When executed, the variable *output* produces the values stored in it.

```
output

[1] 1 0 3
```

Operations can be performed on it, functions can be applied to it, as if the original vector of values was being used.

```
output + 1  # The vector 1 is being recycled.

[1] 2 1 4

sin(output)  # The sine function is vectorized.

[1] 0.841471 0.000000 0.141120
```

In fact, a variable can be used to update itself and add new values.

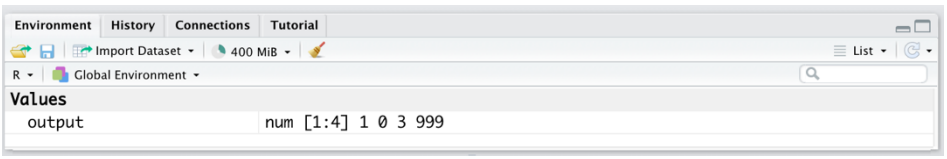
```
output

[1] 1 0 3

output <- c(output, 999)  # a fourth element is added to output
output

[1]   1   0   3 999
```

The right hand side of $<-$ takes the original values associated with the *output* variable, adds one to all of them, and then associates the result with the variable *output*. Effectively, the old values are erased, and replaced.



Several Variables and Multiple operation

Below, we assign variable names, perform some operations using the variables. The results have variable names assigned to them. Then, an operation is performed on the constructed results.

```
x <- c(0, 2, 2.4)  # Assign the variable name x to this vector
y <- c(1, -2, 0.6)  # Assign the variable name y to this vector
output.add <- x + y  # Add x and y together, and assign a name to the result.
output.difference <- x - y  # Subtract y from x, and assign a name to the result.
output.add * output.difference  # Multiply output.add and output.difference together

[1] -1.0  0.0  5.4
```

(Theses operations were selected for demonstration purposes only.)

Rules for Variable Names

There are some rules for variable names that must be followed:

1. The name must start with a letter.
2. The name can only contain letters, numbers, underscores, and periods.

There are also some general guidelines for variables names. They will make reading and updating code much easier.

1. If a value, (vector of values, or any object that you can construct) is going to be used multiple times, assign it a variable name. If you discover a mistake in that value, correct the mistake where the variable name is assigned. No need to search all of your code.

2. The variable name should be assigned **before** every instance of it being used in your code. R reads code from top to bottom.

```
test.variable  # This variable will not exist until the next line.

Error in eval(expr, envir, enclos): object 'test.variable' not found

test.variable <- 3
test.variable

[1] 3
```

3. Use a name that describes what the values represent. At a later time, the descriptive name makes it easier to understand the variables use.

4. Pick a convention for writing names and use is consistently.

```
variablename VARIABLENAME #all lower/UPPER case, words not separated, hard to read
variable.name #all lower case, words separated by a period, easy to read
variable_name #all lower case, words separated by an underscore, easy to read
variableName #Camel case, words not separated, Capitalize first letter of each word except possible the first, easier to read
```

5. Select names for your variables that do not already have some meaning in R.

```
pi  # This is a constant that is already defined in R.

[1] 3.141593

pi <- 10  # This assigns a new value to the name pi
pi

[1] 10
```

Other Assignment Operators

Other assignment operators exist.

1. The single equal sign =. This works like <-, but doesn't visually does not indicate what is occurring as well as <-.
2. -> assigns left-hand values into a variable listed on the right. If the value is long, it is hard to find the variable name.
3. <<-, ->> are assignment operators that **may** be used when making functions.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Create a single variable for each of the following vectors. Use a descriptive name. The naming convention to use is indicated.

    a. 12 (period separator)
    b. c(1, 3, 5, 7, 9, 11, 13) (All Capitals)
    c. c(1, 2, 3, 4, 5, 6, 7)* π (underscore separator)
    d. 1 / c(1, 2, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8) (Camel Case)

2. Create the following variables, and assign an appropriate vector to it. Modify the variable name to use the indicated naming convention

    a. sqrt.ofTWO (period separator)
    b. naturallogofonethroughten (Camel Case)
    c. sine_of_0_45_90____360_degrees (period separator)
    d. remaindersOfTen.DividedByZeroToTen (All Capitals)
    e. cUBe_ROOTof.3 (All Capitals)
    f. PO.WER_so.F5 (Camel Case)
    g. A._bsO1u_TeV4alues (period separator)

3. Create a vector with variable name **AssignmentData001**. Use functions of **AssignmentData001** to compute the vectors that follow. Assign variable names to each result. Display each result. The elements of **AssignmentData001**, in order, should be:

$$1\ 4\ 7\ 9\ 12\ 15\ 19\ 21\ 22\ 30$$

a. c(1, 4, 7, 9, 12, 15, 19, 21, 22, 30) * π
b. $1/c(3^1, 3^4, 3^7, 3^9, 3^{12}, 3^{15}, 3^{19}, 3^{21}, 3^{22}, 3^{30})$
c. c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)

4. Given Newton's Law of Gravitation $F = G\frac{m_1 m_2}{r^2}$ where the gravitational constant $G \approx 12.6$ (In Oan Units), m_1, m_2 are the masses of two bodies, r is the distance between them, and F is the gravitational force between them, determine the gravitational force between bodies with the following properties:

m_1	m_2	r
10	11	2
34	21	.5
4	39	19
884	73	98
4	34	23

5. Compute the first 20 numbers in the Fibonacci Sequence as a single vector. Store your result in a vector called Fib. Start with the number one.

Use the fact that the n^th value in the sequence F_n can be computed using

$$F_n = \frac{\psi^n - (-\psi)^{-n}}{\sqrt{5}}$$

where $\psi = \frac{1+\sqrt{5}}{2}$.

6. The following table provides the length of the hypotenuse and the length of another side of a right triangle. Determine the length of the third side for each of the following.

Hypotenuse	Other
5	4
13	12
41	40
29	21
65	63

7. Use the given variables to create the indicated vectors.

VariableName	Vector
ones	c(1,1,1)
evens	c(2,4,6, 8)
negatives	-c(10,20,30,120)
ninety_nine	99

   a. c(1, 1, 1, 1, 1, 1, 99)
   b. c(1, 1, 1, 2, 4, 6, 8, 10, 20, 30, 120)
   c. c(1,1,1,99,10,20,30,120,10,20,30,120,1,1,1,1,1,1)
   d. c(2,2,2,4,8,12,16)
   e. c(0, 0, 0, 5, 5, 5, 15, 0, 0, 0, 0)

Chapter 5: Character Data

Character Data appears in two forms. It will appear as text surrounded by:

1. "Double quotes" or,
2. 'Single quotes'.

Commonly, character data are referred to as **string** data, or **strings**. Strings can contain any character you can type on the keyboard. Strings can even contain no characters. This is indicated by a set of quotes without any characters or spaces in between them.

```
""  #This is an empty string

[1] ""

" "  #This string contains a space.

[1] " "

"This is a string. It contains words."

[1] "This is a string. It contains words."
```

Character Vectors

Collections of character data can be held in an ordered character (atomic) vector, in the same way as logical and numerical data. An ordered character vector can be created using the **concatenate function** $c(\quad)$. Make sure to surround each string with quotes and separate each string with a comma.

```
c("String", "Elements", "Are", "Wrapped", "In", "Quotes.", 9.7, 16)

[1] "String"   "Elements" "Are"      "Wrapped"  "In"       "Quotes."  "9.7"      "16"
```

Operating on Strings

The $paste(\quad)$ function is used to join character vectors together element by element. The $sep =$ argument indicates which string is used to separate the combined elements. $paste(\quad)$ produces a character vector.

```
paste(c("This", "a", "vector"), c("became", "single", "."), sep = "<:>")

[1] "This<:>became" "a<:>single"    "vector<:>."
```

Adding the $collapse =$ argument will combine all elements into a single string.

```
paste(c("This", "a", "vector"), c("became", "single", "."), sep = "<:>", collapse = " - - ")

[1] "This<:>became - - a<:>single - - vector<:>."
```

Printing Strings

When entered on its own, a string will automatically display itself. However, in some situation, you must force the display. This can be done using several functions.

```
print("This is a string.")

[1] "This is a string."

noquote("This is a string without printed quotes.")

[1] This is a string without printed quotes.
```

The $cat(\quad)$ function can be used to print a string without the line number printed and without quotes. It will also interpret formatting symbols.

```
cat("This line has a \t tab in it. \n Now, we have skipped a line and included ê.")

This line has a      tab in it.
 Now, we have skipped a line and included ê.
```

Modifying Strings

Strings can be modified in several ways. The case of the letters in string can be changed. (These functions may not give the same results on two different computers, if the underlying locales are set differently.)

```
tolower("ChanGE cAsE 1234")

[1] "change case 1234"

toupper("ChanGE cAsE 1234")

[1] "CHANGE CASE 1234"
```

Character replacement can be performed with *chartr*( ). The *old* argument gives the old values to be replaced. The *new* argument indicates what they are translated into. The final string is what need to be translated.

```
chartr(old = "EhC", new = "eHX", "ChanGE cAsE 1234")

[1] "XHanGe cAse 1234"

chartr(old = "abcX", new = "DEFx", "abcdefghijklmnopqrstuvwXyz")

[1] "DEFdefghijklmnopqrstuvwxyz"
```

Extracting Strings

It may occur that just a single string contains several pieces of information. You may want to separate these pieces or extract the single piece that you need.

To extract part of a string, use the *substr*( ) command. It requires that you know the numerical position of the characters that you want in the string (including spaces).

This code will extract the second, third, fourth and fifth value from the given string.

```
substring("abcdefghijklmnopqrstuvwxyz", first = 2, last = 5)

[1] "bcde"
```

In some cases, the portion that you want to extract might not have a known length. However, you may know that there will only be a certain number of characters, say 4, after it. Use the *nchar*( ) function to determine the number of characters in the string, and subtract 4 from this value.

```
string.with.equal.signs <- "=I Want This Part.==="
string.length <- nchar(string.with.equal.signs)
substring(text = string.with.equal.signs, first = 2, last = string.length - 3)

[1] "I Want This Part."
```

substring( ) can also be used to replace one part of one substring with another.

```
substring(text = string.with.equal.signs, first = 2, last = string.length - 3) <- "++++++"
string.with.equal.signs

[1] "=++++++ This Part.==="
```

If there is a regular pattern to how the information is given in the string, it can be split apart.

```
strsplit("Age , Height , Weight , Other", split = " , ")

[[1]]
[1] "Age"    "Height" "Weight" "Other"
```

Two things we should notice about this:

1. The string was split using " , " instead of ",". The first would leave a spaces on each resulting string.
2. The result has two sets of indexes [[1]] and [1]. This is because the result is not a vector, but a different object for holding data called a **list**.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Create a character vector called **blank**. It should have 5 elements. The middle three should be spaces. The outer two should be empty.

2. Create the following character vectors based on the given descriptions and assign it to a variable. Each individual bold item in the description should be its own element. The variable name should be descriptive of the content.

    a. vowels (5 elements) (You may need to look up what a vowel is.)
    b. The digits 0 through 9 (10 elements)
    c. First five **words** in US National Anthem
    d. First two **sentences** in A Tale of Two Cities.

3. Make a single string out of each of the following vectors.

    a. MoreCharacterData001; link using spaces
    b. MoreCharacterData002; link using underscores

4. Split this string CharacterData003 into a character vector with one number per element (9inches becomes 9, 12inches becomes 12, etc). Leave no extraneous spaces or characters. This actually creates a list that contains one character vector, but that is ok.

5. Make a character vector with 26 elements. The first element should be “I like the letter A.”. The second element should be “I like the letter B.” This should continue until the twenty sixth element “I like the letter Z.” Pay attention to the period. Make use of the vector CharacterData004.

6. Make a **single** string using the vectors CharacterData005 and CharacterData006. The final string should contain the ending of ‘MacArthur’s Farewell Address to Congress’. There should be no extraneous spaces at the end. It will start and end as follows:

   I am closing my ..... Good Bye.

7. A Caesar Cypher is a simple method for encoding a message. The method takes each letter in a given message and shifts it to a letter a fixed number of places to the right of it in the alphabet. Letters at the end of the alphabet wrap around to the beginning. For example, if the shift is 1, then A becomes B, B becomes C, Y becomes Z, and Z becomes A. Other shifts can be used. The messages below have been encoded with a Caesar Cipher. Decode them.

   a. CharacterData007
   b. CharacterData008
   c. CharacterData009
   d. CharacterData010

8. Extract the portion of each string that does not contain equal signs:

   a. CharacterData011
   b. CharacterData012
   c. CharacterData013

Chapter 6: More Character Operations - stringr

The *stringr* package adds a number of functions for manipulating strings. Functions built into the *stringr* package are all named beginning with "str_". Additionally, the arguments appear in a consistent order from function to function.

```
library(stringr)
```

Joining & Splitting Strings

The *str_c( )* function joins multiple strings into one.

```
str_c("Number: ", c(1, 2, 3, 4, 5, 6))

[1] "Number: 1" "Number: 2" "Number: 3" "Number: 4" "Number: 5" "Number: 6"

str_c(c("a", "b", "c", "d", "e", "f"), c(1, 2, 3, 4, 5, 6), sep = "_")

[1] "a_1" "b_2" "c_3" "d_4" "e_5" "f_6"

str_c(c("a", "b", "c", "d", "e", "f"), c(1, 2, 3, 4, 5, 6), "X", sep = "_", collapse = "^")

[1] "a_1_X^b_2_X^c_3_X^d_4_X^e_5_X^f_6_X"
```

The *str_c( )* function works in much the same way as *paste( )*.[15] *sep* is used to insert a string between the elements. *collapse* is used to combine all resulting elements into a single string.

The *str_flatten( )* function will collapse the separate elements of a vector into a single string.

```
str_flatten(string = c("pear", "orange", "berry", "nut"))

[1] "pearorangeberrynut"

str_flatten(string = c("pear", "orange", "berry", "nut"), collapse = " - ")

[1] "pear - orange - berry - nut"
```

The *collapse* argument indicates the string to be inserted between each element.

The *str_split( )* function breaks a string into several substrings based upon some given pattern.

```
str_split(string = "pear - orange - berry - nut", pattern = " - ", n = 3, simplify = FALSE)

[[1]]
[1] "pear"        "orange"      "berry - nut"
```

The *str_dup( )* function is a vectorized function duplicates and concatenates strings. Its second argument *times* indicates how many times each element of the given vector is duplicated.

```
str_dup(string = c("pear", "orange", "berry", "nut"), times = c(1, 2, 3, 4))

[1] "pear"            "orangeorange"    "berryberryberry" "nutnutnutnut"
```

The *str_length( )* function will return the number of characters in each element of a given vector. This is similar in function to *nchar( )*.

```
str_length(string = c("pear", "orange", "berry", "nut"))

[1] 4 6 5 3
```

Substrings - Extracting, Modifying & Padding

In the *stringr* package, the *str_sub( )* function extracts substrings from a character vector. The substrings will start with the character in the position indicated by *start* and end with the character in the position indicated by *end*. Failure to indicate either will be seen as an indication to select from the first character, or continue to the final character.

```
x <- c("abcdefg", "ABCDEFGH", "123456789")
str_sub(string = x, start = 2, end = 5)
```

[15] With the exception that it will only recycle vectors of with only a single element. Otherwise, vectors must have the same length. Notice the error in the second example.

```
[1] "bcde" "BCDE" "2345"

str_sub(string = x, end = -4)

[1] "abcd"   "ABCDE"  "123456"

str_sub(string = x, start = 2, end = -2) <- "A"
x

[1] "aAg" "AAH" "1A9"
```

A negative number in the *end* position will indicate that the selection should be made by counting backwards from the end of a given string.

The *str_pad*( ) function can be used to add characters to a string until the string attains a minimum length. The example below will add some characters ("Z") to both sides of these foods until they have a minimum length of 6 characters.

```
str_pad(string = c("pear", "orange", "berry", "nut"), width = 6, side = "both", pad = "Z")

[1] "ZpearZ" "orange" "berryZ" "ZnutZZ"
```

Characters can be added solely to the "left" or "right" sides of the given strings. The function is vectorized in the *pad* argument, which lists a vector of single characters used to extend the given strings.

Whitespace - Trimming & Squishing

When strings are split into substrings, extraneous white spaces are included in the included substrings.

The *str_trim*( ) function will remove extraneous white spaces white spaces on the "left" side, "right" side, or "both" sides of the given string depending upon the argument *side*.

```
x <- "  <--Extra Spaces Here-->       <--Here-->      "
str_trim(string = x, side = "both")

[1] "<--Extra Spaces Here-->       <--Here-->"

str_squish(string = x)

[1] "<--Extra Spaces Here--> <--Here-->"
```

The *str_squish*( ) function removes all extra white spaces on either end of, and inside of, a given string. In the examples, pay close attention to the placement of the quotation marks.

Truncation

The *str_trunc*( ) function will truncate a given character string, and pastes an ellipsis onto the result. The *width* argument determines the length of the overall result. The *width* minus the length of the *ellipsis* argument determines the number of characters kept from the original string. The *ellipsis* can be placed on the "right", "left", or "center".

```
x <- "This string has a lot of characters in it."
str_trunc(string = x, width = 11, side = "right", ellipsis = ".....")

[1] "This s....."
```

Sorting

The *str_sort*( ) function sorts a character vector. The *decreasing* argument determines whether the sorting is in decreasing or increasing order. *locale* defaults the sort order to English alphabet. This will affect the ordering used. *na. last* indicates where *NA* values fall in the ordering. *numeric* controls whether digits are treated as numbers or as strings.

```
str_sort(x = c("pear", "orange", NA, "berry", "nut"), decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE)

[1] "berry"  "nut"    "orange" "pear"   NA
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

Use the functions demonstrated from the *stringr* package to complete the following.

1. Make a character vector with 26 elements "Letter: A", "Letter: B", "Letter: C" etc. Make use of CharacterData004.

2. For each of the following, make a character vector with a single element. The single element should be a string containing all of the elements of the indicated vector connected as indicated.

    a. CharacterData001; link using spaces
    b. CharacterData002; link using underscores

3. Split these strings each into a character vector with one word per element. Leave no extraneous spaces or characters.

    a. MoreCharacter003
    b. MoreCharacter004
    c. MoreCharacter005
    d. MoreCharacter006

4. Make a character vector with 10 elements. The elements should indicate Santa Claus' fictional enthusiasm, as he makes his fictional trip around the world. Each of the following lines should be an element.

    Ho!Ho!Ho!Ho!Ho!Ho!Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!Ho!Ho!Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!Ho!Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!Ho!
    Ho!Ho!Ho!
    Ho!Ho!
    Ho!

5. Make a character vector with 26 elements. The first element should be "I like the letter A.". The second element should be "I like the letter B." This should continue until the twenty sixth element "I like the letter Z."

More To Come

Chapter 7: Logical Data

Logical Data takes in TWO values[16]:

1. TRUE (T)
2. FALSE (F)

Logical data can be used to encode the answer to a Yes/No question. Use TRUE for Yes, and FALSE for No, When using logical data, it must be entered in uppercase letters. Otherwise, R will no recognize it.

Collections of logical data can be held in an ordered (atomic) vector. A logical vector can be created using the **concatenate** function `c()`. Make sure to separate each logical data value with a comma.

```
c(TRUE, TRUE, FALSE, FALSE)
c(T, F, T, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F)
```

In fact, a single logical value on its own is a vector. It just happens to be a vector with one element.

Logical Operators

Computations using logical data can be performed.

The simplest operation is called **logical negation** or **NOT**. It turns a TRUE value into FALSE, and a FALSE value into TRUE.

```
!TRUE

[1] FALSE

!c(TRUE, FALSE)

[1] FALSE  TRUE
```

Other logical operations take two arguments(inputs) and return a logical value. A logical AND determines if both inputs are TRUE. A logical OR determines if at least one input is TRUE. The **exclusive OR**, **xor()**, determines if exactly one of the inputs is TRUE.

```
# Logical AND &&
TRUE && TRUE

[1] TRUE

# Logical OR ||
TRUE || TRUE

[1] TRUE

xor(TRUE, T)

[1] FALSE
```

The operations indicated by **&&** and **||** will only look at one element vectors.

These operations can be used in sequence (taking the result of one operation and plugging it into another). You just need to enclose the first operation inside parentheses.

```
(TRUE && !TRUE) || TRUE

[1] TRUE
```

[16] This is a slight lie. There is a third logical value that represents missing values

Vectorized Operations

If you want to perform these operations in a pairwise(element-wise) fashion, you want to use the **vectorized** version of these operators.

```
# Vectorized AND
c(TRUE, TRUE, FALSE, FALSE) & c(T, F, T, F)

[1]  TRUE FALSE FALSE FALSE

# Vectorized OR
c(TRUE, TRUE, FALSE, FALSE) | c(T, F, T, F)

[1]  TRUE  TRUE  TRUE FALSE

# exclusive logical OR
xor(c(TRUE, TRUE, FALSE, FALSE), c(T, F, T, F))

[1] FALSE  TRUE  TRUE FALSE
```

As you can see, xor() will work with single element vectors or vectors with more elements.

Recycling

For operations with two inputs, if one input has fewer elements than the other, R will **recycle** (repeat) the shorter vector, until it is long enough to perform the vectorized operation.

```
c(TRUE, TRUE, FALSE, FALSE) & c(T, F)

[1]  TRUE FALSE FALSE FALSE
```

In the first line, R repeated the vector with two elements. The second vector was treated like **c(T,F,T,F)**.

```
c(TRUE, TRUE, FALSE, FALSE) | c(T, F, T)

Warning in c(TRUE, TRUE, FALSE, FALSE) | c(T, F, T): longer object length is not a multiple of shorter object length

[1] TRUE TRUE TRUE TRUE
```

In this example, a warning is given. It indicates that the longer vector is not a multiple of the shorter vector, in terms of length. This is true. The longer vector had a length of 4, while the shorter had a length of three. In this case, the vector with length three was only partially recycled. To perform the computations, the second vector was treated like **c(T,F,T,T)**.

Any & All

Given a vector of logical data, you may wonder if **any** or **all** of the values are TRUE

```
any(c(F, T, F, F))

[1] TRUE

any(c(F, F, F, F))

[1] FALSE

all(c(T, T, T, T))

[1] TRUE

all(c(T, T, T, F))

[1] FALSE
```

One Interesting Fact

In certain contexts, R will treat logical data values as numbers. Namely, TRUE will be treated as one(1) and FALSE will be treated as zero(0).

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

Display any code you use to find your answer. In as few sentences as possible, state your answer.

1. Individually, determine the value of the following:

    a. FALSE or TRUE
    b. TRUE or TRUE
    c. FALSE and TRUE
    d. TRUE or FALSE
2. After creating two logical vectors, use a single line of code to determine the value of the following:

    a. FALSE or TRUE
    b. TRUE or TRUE
    c. FALSE or TRUE
    d. TRUE or FALSE
3. Translating these directly into the appropriate logical operations, determine which of these are TRUE:

    a. (T or F) and T
    b. (T and T) or (F and T)
    c. not (not T or not F)
    d. not not not TRUE
4. Compute the value of each without using recycling.

    a. c(T, T, F, F, TRUE) and T
    b. not c(F, T, T) and c(F, T, F, T, T, F)
    c. c(T, not FALSE, T, TRUE, T, T) or c(F, F, not T)
    d. not c(FALSE, not FALSE, T, TRUE, T, T, FALSE) or not c(not F, not F, not FALSE, T)
5. Using the results from question 4, determine if **any** and/or **all** of the resulting elements are TRUE.

6. Translating these directly into the appropriate logical operations, determine which of the following result in c(T, T, T, T, T, T)

    a. c(T, T, F, F, TRUE) or F
    b. c(T, T, F, F, T, F) or not c(F, T, F)
    c. c(T, not FALSE, T, TRUE, T, T) and not c(F, F, FALSE, not T)
    d. not c(T, not FALSE, T, TRUE, T, T) or c(not F, not F, not FALSE, T)

7. Perform a series of logical operations on the given vectors so that the result is a vector whose entries are all TRUE. All vectors in each part must be used at least once. Operations from the previous vectors can't be used. (Experiment)

    a. c(T, T, F, F) , c(T, F, T, F)
    b. c(T, T, T, T, F, F, F, F), c(T, T, F, F), c(T, F)

Chapter 8: Comparisons

Given two data values (logical, numeric, character) it will be important to determine how they relate two each other. Are these values the same? Is one greater than the other? Does one come before the other?

The comparison operators can be used to answer these questions. They take any two data values, and return a logical value indicating the validity of the comparison.

As with many of the operators we have seen before, these operators are vectorized, and will recycle shorter vectors.

Ordering (<, <=, >=, >)

To order numeric values, we ask if one value is less than < or greater than > another value. If we want to include equality, add on an equal sign. Consider the possible comparisons between the numbers 0 and 1.

```
# less than ?Comparison
0 < 1

[1] TRUE

1 < 0

[1] FALSE

# less than or equal to
0 <= 1

[1] TRUE

1 <= 0

[1] FALSE

# greater than ( or equal to)
c(0, 1) > 1

[1] FALSE FALSE

c(0, 1, 1, 0) >= c(1, 0)

[1] FALSE  TRUE  TRUE  TRUE
```

In the last line, realize that the $c(1,0)$ vector on the right hand side is being recycled. The comparisons that are being performed look (invisibly) like $c(0,1,1,0) >= c(1,0,1,0)$.

These ordering operators are not limited to comparing numeric data only. They can be used on any of the data types. Strings will use an alphabetic ordering according to the locale setup in your computer.

Equality

Checking whether two values, or vectors, are the same can be tricky. The obvious choice for checking equality would be the equal sign =. However, the equal sign on its own performs a different function. The appropriate operator is a double equal sign ==.

```
7 == 14/2

[1] TRUE

c("a", "b") == c("a", "b", "a", "b")

[1] TRUE TRUE TRUE TRUE

c("a", "b") == c("a", "B", "a", "Horse")

[1]  TRUE FALSE  TRUE FALSE
```

The double equal sign will return a logical value, or vector of logical values. It is vectorized and will recycle a shorter vector if needed. As we can see, it will compare character vectors. It will also compare logical vectors. In each case, it compares corresponding elements.

The double equal sign does have some things to be aware of:

1. == looks for exact equality in the computers representation of a value.

Because of how a computer stores numbers, two values that mathematically are the same, may not be the same when stored. Consider the square root of 2.

```
sqrt(2) * sqrt(2) == 2L

[1] FALSE

sqrt(2) * sqrt(2)

[1] 2
```

Theoretically, this should be TRUE. A computer approximates the square root of 2. When this approximation is squared, a small amount error is included. It will approximate 2. As a remedy, we might ask if two values are almost equal or nearly equal. This can be done using the *all.equal( )* function in base R or the *near( )* function included in the *dplyr* package. Both of these functions have a small tolerance for some difference between two values.

```
all.equal(sqrt(2) * sqrt(2), 2)

[1] TRUE

dplyr::near(sqrt(2) * sqrt(2), 2)

[1] TRUE
```

Both functions can be used on vectors of values, but the outputs are very different. *all.equal( )* will return TRUE if the difference in all pairs is within the given tolerance. However, it returns a string giving the average relative difference[17].

```
all.equal(c(1.000000004, 2.000000009), c(1.000004, 2.000000008))

[1] "Mean relative difference: 1.332333e-06"
```

dplyr::near( ) will return a logical vector containing the results for each pairing.

```
dplyr::near(c(1.000000007, 2.000000009), c(1, 2.000000008), tol = 5e-09)

[1] FALSE  TRUE
```

2. == does not compare the vectors as a whole.It compares corresponding elements

```
c("a", "b") == c("a", "b", "a", "b")

[1] TRUE TRUE TRUE TRUE
```

Allow for recycling of the shorter vector, the elements of these vectors match up. But, as a whole these vectors are not the same. Checking if any two R objects (not just vectors) are the same is done with the *identical( )* function.

```
identical(c("a", "b"), c("a", "b", "a", "b"))

[1] FALSE
```

This function will determine if two elements are exactly the same. It will check many things not visible to the eye.

```
identical(2, 2L)  #These are different types of numerical data.

[1] FALSE

identical(2, "2")  #These are different types of data

[1] FALSE

2 == "2"  #These are different types of data, but one is coerced into the other type.

[1] TRUE
```

[17] What is average relative distance?

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Using the following sets of vectors, determine how elements from the first vector are related to the corresponding elements from the second. Use the indicated comparison.

    a. 3, 3L (equality)
    b. c(1, 2, 3, 4, 5, 6), 4 (less than)
    c. c(1, 2, 3, 4, …., 25), c(sqrt(1), sqrt(2), …, sqrt(25))^2, (greater than or equal to)
    d. c(1, 2, 3, 4, …., 25), sqrt(c(1^2, 2^2, …, 25^2)), (less than or equal to)
2. Using the following sets of vectors, determine how elements from the first vector are related to the corresponding elements from the second. Use the indicated comparison.

    a. “a”, “A” (equality)
    b. “a”, “A” (less than)
    c. c(“a”, “b”, “c”), c(“b”, “c”, “d”) (greater than)
    d. c(1, 2, 3, 4), c(“1”, “2”, “3”, “4”) (equality)
3. Using the following sets of vectors, determine how elements from the first vector are related to the corresponding elements from the second. Assign a variable to the result. Determine if **any** of the results are $TRUE$. Use the indicated comparison.

    a. c(“the”, “dude”, “abides”), c(“thematic”, “duarchy”, “aberatted”) (less than)
    b. c(1, “2”, 4, 6), c(“one”, “two”, “three”, “nine”) (greater than or equal)
    c. c(TRUE, FALSE), c(1, 0) (equality)
    d. c(TRUE, FALSE), c(“1”, “0”) (equality)
4. Determine if the two given vectors are identical.

    a. c(“the”, “dude”, “abides”), c(“the”, “dude”, “abides”, “the”, “dude”, “abides”)
    b. c(“the”, “dude”, “abides”, “the”, “dude”, “abides”), c(“the”, “dude”, “abides”, “the”, “dude”, “abides”)
    c. c(TRUE, FALSE), c(1, 0)
    d. c(1, 2, 3, 4, …., 25), sqrt(c(1^2, 2^2, …, 25^2))

5. Determine if the two given vectors are identical, and if **all** of the paired components are equal (==). If there is a discrepancy, give an explanation.

    a. 9, “9”
    b. c(“a”, “aa”, “aaa”), c(“a”, “aa”, “aaa”, “a”, “aa”)
    c. c(1L, 2L, 3L, 4L, 5L, 6L, 7L), c(1L, 2, 3L, 4L, 5L, 6L, 7L)
    d. c(T, T, F, T), c(TRUE, TRUE, FALSE, TRUE)

Chapter 9: Vectors

We have encountered four types of (atomic) vectors depending on the type of data contained in it.

1. Logical
2. Numerical - Integer/Floating Point
3. Character

Given a vector, the type of data contained in the vector can be determined using the *typeof*( ) function.

```
typeof(c(1, 2, 3, 4))

[1] "double"

typeof(c("A", "B", "String"))

[1] "character"
```

Vector Attributes

Besides the data explicitly contained in a vector, other pieces of information can be attached to the vector. Names can be given to each data value. Names can be added immediately as the data is being entered.

```
exampleData <- c(data1 = "apple", data2 = "orange", data3 = "pineapple", data4 = "strawberry")
exampleData

      data1       data2       data3       data4
    "apple"    "orange"  "pineapple" "strawberry"
```

If the names need to be attached, or updated, this can be done using the *names*( ) function. It will require a character vector with a name for each piece of data. The names should appear in the same order as the data they represent.

```
names(exampleData) <- c("newName1", "newName2", "newName3", "newName4")
exampleData

    newName1     newName2     newName3     newName4
     "apple"     "orange"  "pineapple" "strawberry"
```

Comments can be added to describe the contents of the vector using the *comment*( ) function. Comments should be written as strings. The *comment*( ) function displays the comments also.

```
comment(exampleData) <- "This is a comment."
comment(exampleData)

[1] "This is a comment."
```

The *attributes*( ) function can be used to display a summary of these additional attributes.

```
attributes(exampleData)

$names
[1] "newName1" "newName2" "newName3" "newName4"

$comment
[1] "This is a comment."
```

Vector Length

The number of elements in a given vector can be determined with the *length*( ) function.

```
length(exampleData)

[1] 4
```

Numerical Sequences

A numerical vector containing a regular sequence of numbers is often needed. The most basic is a sequence of consecutive integers. This is generated with a colon separating the first and last integer in the sequence.

```
seqentialIntegers <- 3:8  #Non-Integers can be used, but the sequences increments by 1.
seqentialIntegers
```

```
[1] 3 4 5 6 7 8
```

For other regular sequences, you want to use the $seq(\ \ )$ function. It can by specifying three out of four arguments.

1. $from/to$ - the starting / end values, at least one of these is needed.
2. by - the size of the increment between values
3. $length.out$ - the number of values to be produced.

```
seq(from = 4, to = 6, length.out = 9)
seq(from = 4, to = 6.1, by = 0.25)
seq(from = 4, by = 0.25, length.out = 9)
seq(to = 6, by = 0.25, length.out = 9)
```

All of these produce the same result.

```
[1] 4.00 4.25 4.50 4.75 5.00 5.25 5.50 5.75 6.00
```

An integer sequence that from one to the length of an R object is useful. This can be done using colon notation, or $seq(\ \ )$, along with the $length(\ \ )$ function. Alternatively, $seq\_along(\ \ )$ acommplishes this with a single function.

```
seq_along(exampleData)

[1] 1 2 3 4
```

Replicating Vectors

Repeating patterns can be created with the $rep(\ \ )$ function. A vector containing the basic parts of the pattern is needed. The arguments control the method of replication.

```
givenVector <- c("a", 9)
rep(givenVector, times = 3)

[1] "a" "9" "a" "9" "a" "9"

rep(givenVector, each = 3, times = 2)

 [1] "a" "a" "a" "9" "9" "9" "a" "a" "a" "9" "9" "9"

rep(givenVector, each = 3, times = 2, length.out = 10)

 [1] "a" "a" "a" "9" "9" "9" "a" "a" "a" "9"
```

Combining Vectors

New vectors can be created by concatenation using variable names and explicitly written vectors.

```
x <- 7:4  #Sequential Vector
y <- rep(9, times = 3)  #Replicated Vector
newVector <- c(x, y, 10, x)
newVector

 [1]  7  6  5  4  9  9  9 10  7  6  5  4
```

This code chunk creates a vector x containing 7 down to 4, a vector y replicating the number 9 three times. Finally, the vector newVector is created with elements from x, then from y, followed by 10, and finishing with the elements from x.

Placeholder Vectors

When computations are repeated with different values, the results will need to be stored. If you know how many different sets of computations are to be performed, it can be useful to have a vector that can be filled in with the results.

```
resultsVector1 <- vector(mode = "logical", length = 5)
resultsVector2 <- rep(NA, length = 5)  # NA stands for Not Available. It is used for missing values.
resultsVector1
resultsVector2

[1] FALSE FALSE FALSE FALSE FALSE
[1] NA NA NA NA NA
```

The mode in the $vector(\ \ )$ function can be replaced with integer, double, or character.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Determine the type of vector for each of the following: integer, double, character, logical

    a. c(1, 2L, 3L, 4L, 5L, 6L, 7L, 8L)
    b. c(T, T, T, T, "F")
    c. c(T, 1)
    d. c("a", T, 10)
2. For each set of attributes, create an appropriate vector.(Spaces are not allowed in names)

    a. names, in order: one squared, two squared, three squared, four squared, …, ten squared
    b. comment: These are the fourth roots of the integers 2 through 10.
    c. names, in order: lower a, upper B, lower c, upper D, …, upper J comment: Letters a-j alternating case.
3. Create the following vectors and assign a variable to the result

    a. c(1, 2, 3, 4, … , 1000)
    b. c(sqrt(1), sqrt(2), sqrt(3), … , sqrt(1000))
    c. c(2, 4, 6, 8, …, 1000)
    d. c(1, 3, 5, 7, …, 999)
    e. c(10, 9, 8, 7, … 0, -1, -2, …, -50)
4. Create the following vectors and assign a variable to the result

    a. c(1, 1.5, 2, 2.5, 3, 3.5, …, 50)
    b. c(2.2, 3.2, 4.2, 5.2, 6.2, …, 100.2)
    c. c(-1, -.6, -.2, .2, .6, 1.0, …, 5.4)
    d. c(ln(8.7), ln(9), ln(9.3), ln(9.6), ln(9.9), ln(10.2), …, ln(15))
5. Create a vector with the following properties:

    a. Nine evenly spaced elements starting at -0.123 and ending at 68
    b. Nineteen elements starting at 90 and decreasing by 12.
    c. Elements starting at 9, increasing by .3, no larger than 21.2
    d. Square root of all multiples of 10 between 1 and 2001.
    e. Twenty-five more than the square root of all multiples of 82 between 1 and 2001.

6. Use the vectors listed below, or functions of them, to create the indicated vector:

VectorsA, VectorsGirl, VectorsOne, VectorsInts, VectorsZero

```
a.  c(1, 4, 5, 6, 7)
b.  A vector with 100 elements. The values alternate between "a" and "girl" with the first element being "a"
c.  c("a", "1", "a", "girl", "4", "5", "6", "7", a)
d.  A numerical vector with 200 elements. Each element is a 1.
e.  c("girl", "girl", "girl", "girl", "girl", "girl", "girl", "girl", "girl", "girl", "girl", "girl", "girl")
f.  c("girl", "girl", "girl", "girl", "girl", "a", "girl", "girl", "girl", "girl", "girl", "a", "girl", "girl", "girl", "girl",
"girl", "a", "girl", "girl", "girl", "girl", "girl", "a" )
g.  c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 )
h.  c(4, 5, 6, 7, 8, 10, 12, 14, 12, 15, 18, 21, 16, 20, 24, 28)
```

7. Determine the number of elements in the following vectors.

   a. seq(from = 156, to = 10000, by = pi)
   b. seq(from = 203, to = -56, by = -2.6)
   c. c(14.6, 15.1, 15.6, 16.1, …., 200.1)
   d. Elements start at -7.2, increasing by .3, no larger than 56.8

Chapter 10: Filtering & Subsetting Vectors

Given any type of vector with any number of elements, at times you will want to work with only smaller portion of the elements in the given vector. This can be done numerical values or with logical values.

Numerical Subsetting

To indicate which portion of a given vector you want to use, *R* needs to be given the **indexes** or **positions** of the desired elements need. The index is the numerical position as you count through the given vector.

```
givenVector <- c("a", "b", "c", "d", "e", "F", "G", "H", "I", "J", "k", "l", "m", "n", "o", "P", "q", "r", "s", "T", "a", "b",
"c", "a")
givenVector

 [1] "a" "b" "c" "d" "e" "F" "G" "H" "I" "J" "k" "l" "m" "n" "o" "P" "q" "r" "s" "T" "a" "b" "c" "a"
```

To determine the index of a value, count which element it is. *R* helps with this. At the head of each line, an integer is in square brackets. That integer indicates the index of the first element in that line. The capital T has an index of [20].

To select a single element from a vector, follow a straight forward syntax. Start with the name of the vector, followed by a set of square brackets that contain the index of the desired element.

```
# Selecting the 20th element
givenVector[20]

[1] "T"
```

A subset does not need to be restricted a single element. A numerical vector containing multiple values can be used.

```
# Selecting the 20th element
givenVector[c(1, 2, 20, 23, 24)]

selectThese <- c(1, 2, 20, 23, 24)
givenVector[selectThese]

[1] "a" "b" "T" "c" "a"
```

It may be that you want to select all elements from a given vector **except** for a given set. Use the same procedure as before, but use negative indexes

```
# Selecting the 20th element
givenVector[-c(1, 2, 20, 23, 24)]

givenVector[-selectThese]
```

Either of these will produce:

```
 [1] "c" "d" "e" "F" "G" "H" "I" "J" "k" "l" "m" "n" "o" "P" "q" "r" "s" "a" "b"
```

The *which*( ) function can be used to determine the indexes of all elements that satisfy a certain condition.

```
# Determine the index where 'a' elements are located.
which(givenVector == "a")

[1]  1 21 24

LOCATEa <- which(givenVector == "a")
givenVector[LOCATEa]

[1] "a" "a" "a"
```

Subsetting by Name

When names are assigned, a character vector inserted inside square brackets selects values.

```
exampleData <- c(data1 = 2.2, data2 = 9, data3 = -1)
exampleData[c("data2", "data3")]

data2 data3
    9    -1
```

Reordering by filtering

The values in the index vector do not have to be arranged in an increasing order. They can be used to reorder all, or a subset of a vector.

```
givenVector <- c("z", "y", "x", "a", "b", "c")

givenVector[c(3, 2, 1)]  #List the third, then second, then first value

[1] "x" "y" "z"
```

The *order*( ) function will reshuffle the indexes of all elements, so that the elements can be put in order

```
order(givenVector)

[1] 4 5 6 3 2 1

alphabeticalOrder <- order(givenVector)
givenVector[alphabeticalOrder]

[1] "a" "b" "c" "x" "y" "z"
```

Logical Subsetting

When using logic to subset/filter a vector, you must provide a $TRUE$ or $FALSE$ value for each element in the given vector.

```
givenVector <- 1:6
givenVector

[1] 1 2 3 4 5 6

selectThese <- c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)
givenVector[selectThese]

[1] 1 3 5
```

Using this logical vector, only the odd numbers were selected from the numbers. The same result can be produced if we set up a comparison that will produce the same vector of $TRUE$ and $FALSE$ values. This comparison will determine f the remainder is 1 when each given value is divided by two.

```
givenVector%%2 == 1

[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE

selectOdd <- givenVector%%2 == 1
givenVector[selectOdd]

[1] 1 3 5
```

A more readable version of this can be seen if you use the *subset*( ) function.

```
# Argument \#1 is the vector to subset. Argument \#2 is the subset criteria.
subset(givenVector, givenVector%%2 == 1)

[1] 1 3 5
```

Replacing elements by filtering

When some elements in a vector need updating, filtering can be used to identify the elements, and then new values can be assigned.

```
dat <- c(1, 2, 3, NA, NA, 6, NA)  # NA represents missing values in positions 4,5, and 7.
dat

[1]  1  2  3 NA NA  6 NA

dat[c(4, 5, 7)] <- c(104, 105, 107)  # Replace the NA values
dat

[1]   1   2   3 104 105   6 107
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Extract the indicated element from the given vector.
    a. The fifth element of **CharacterData004**.
    b. The last element of **CharacterData005**.
    c. The thirty-seventh element of **Filtering001**.
2. Extract the indicated elements from the given vector. Your answer for each should be a single vector with multiple elements. The elements should appear in the same order that as the vector they were extracted from. Each should be completed with a single line of code.
    a. The tenth through the fifteen elements of **CharacterData001**.
    b. The last five elements of **CharacterData006**.
    c. The first twenty elements of **Filtering001**.
3. Extract the indicated elements from the given vector. The elements should appear in the same order that as the vector they were extracted from. Your answer for each should be a single vector with multiple elements. Each should be completed with at most two lines of code.
    a. The elements in the odd indexes(positions) of **CharacterData004**
    b. The elements of **CharacterData005** that have an index position that is a multiple of 4.
    c. The elements **CharacterData001** whose index(position) is equal to a value in the Fibonacci Sequence of numbers.
4. Extract the indicated elements from the given vector. The elements should appear in the same order that as the vector they were extracted from. Your answer for each should be a single vector with multiple elements. Each should be completed with at most two lines of code.
    a. The elements in **Filtering001** that are greater than or equal to 9.9.
    b. The elements in **Filtering001** that are between 0 and 0.025.
    c. The elements of **CharacterData001** that are either the string **if** or the string **this**.
5. For the indicated vector, determine the indices of the elements with the given properties.
    a. The elements in **Filtering001** that are less than -9.95.
    b. The elements of **CharacterData001** that start with a “W” or “w”.
    c. The elements of **CharacterData002** whose first character is a lower case a.
6. Extract the indicated elements from the given vector. The elements should appear in the same order that as the vector they were extracted from. Your answer for each should be a single vector with multiple elements. Each should be completed with at most two lines of code.
    a. The elements of **CharacterData001** that are after the letter U in the dictionary.
    b. The elements of **CharacterData002** whose second character is the letter M (either case).
    c. The elements of **CharacterData002** whose start with a W followed by an H. (either case)
7. Create the following vectors. These should not be created by inspecting the indicated vectors.
    a. A vector with two elements. The two elements should be the smallest and largest elements found in **Filtering001**. (Smallest, Largest)
    b. A vector with four elements. It should contain the four middle values (in terms of size) of **Filtering001**. (Smallest —– Largest)

8. Create the following vectors. These should not be created by inspecting the indicated vectors.

    a. A vector with two elements. The two elements should be the smallest and largest elements found in **Filtering001**. (Smallest, Largest)
    b. A vector with four elements. It should contain the four middle values (in terms of size) of **Filtering001**. (Smallest —– Largest)

Chapter 11: Matrices & Arrays

Matrices and Arrays are more general versions of vectors. If you think of a vector as a set of numbers written on a line, then think of a matrix as a set written out on a grid. If you start stacking matrices, then a three dimensional array is produced. However, from *R* point of view, a matrix or an array is a vector with a dimension attribute. The dimension information indicates how the values are indexed.

Matrices

To Construct a matrix, the **matrix()** function is used. Its first argument is the *data* to be included in the form of a vector. The next two arguments *nrow* and *ncol* indicate the number of rows and number of columns included. Only one if these is required, the other can be found by looking at the length of the data vector. The last argument *byrow* indicates how the matrix is filled in.

```
dat <- 1:24
mat.byrow.FALSE <- matrix(data = dat, nrow = 4, ncol = 6, byrow = FALSE)
mat.byrow.FALSE

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

In this example, *byrow* is set to *FALSE*. Therefore, the columns are filled first from top to bottom, then moving left.

Looking at the margins of the matrix, the square bracketed values indicate the two part index for each value. An individual value is indicated by its row index, followed by a column index. The values should be separated by a comma. Vectors of multiple values can be used to filter a matrix.

```
# Values found in rows 1 and 4 AND columns 2, 3, 4, and 5.
mat.byrow.FALSE[c(1, 4), 2:5]

     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    8   12   16   20

# No values from rows 2 or 3 are include. Neither are values from column 1.
```

If all values in a subset of rows are desired, the column indices don't need to be specified. However, the comma can not be omitted. If a subset of columns is desired, the row values can be omitted.

```
# Rows 1 and 4, all columns
mat.byrow.FALSE[c(1, 4), ]

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    4    8   12   16   20   24

# Columns 2 and 3, all rows
mat.byrow.FALSE[, c(2, 3)]

     [,1] [,2]
[1,]    5    9
[2,]    6   10
[3,]    7   11
[4,]    8   12
```

If the *byrow* argument is set to *TRUE*, the rows are filled from left to right, starting with the top row and moving down. A final argument *dimnames* allows names to be attached to the rows and/or columns. The names need to be added as a list. Lists have not been described yet, they will be in the future.

```
rowlabel <- c("row1", "row2", "row3", "row4")
collabel <- c("column1", "column2", "column3", "column4", "column5", "column6")
mat.labels <- list(row = rowlabel, col = collabel)
dat <- 1:24
mat.byrow.TRUE <- matrix(data = dat, nrow = 4, ncol = 6, byrow = TRUE, dimnames = mat.labels)
mat.byrow.TRUE
```

```
      col
row    column1 column2 column3 column4 column5 column6
  row1       1       2       3       4       5       6
  row2       7       8       9      10      11      12
  row3      13      14      15      16      17      18
  row4      19      20      21      22      23      24
```

To use the row or column names to filter elements from a matrix, use two character vectors whose elements list the names of the desired rows and columns.

```
mat.byrow.TRUE[c("row1", "row4"), c("column2", "column3", "column4", "column5")]
```

```
      col
row    column2 column3 column4 column5
  row1       2       3       4       5
  row4      20      21      22      23
```

As it was said earlier, a matrix is a vector with an additional dimension attribute. The dimension of a matrix is a vector with the number of rows followed by the number of columns. Both matrices that were constructed have 4 rows and 6 columns.

```
dim(mat.byrow.FALSE)

[1] 4 6

dim(mat.byrow.TRUE)

[1] 4 6
```

Looking at the structure of either of these, you will see the vector of data listed as a vector with the indices running from 1 to 4 for the rows and 1 to 6 for the columns as well as an explicit indication of the dimension.

```
str(mat.byrow.TRUE)

 int [1:4, 1:6] 1 7 13 19 2 8 14 20 3 9 ...
 - attr(*, "dimnames")=List of 2
  ..$ row: chr [1:4] "row1" "row2" "row3" "row4"
  ..$ col: chr [1:6] "column1" "column2" "column3" "column4" ...
```

As with a vector, a matrix can only hold one type of data: logical, numeric, or character. From the method of construction, this should be clear. A matrix is built from a vector, which can only hold one type of data.

Arrays

A matrix is a two dimensional example of an array. Arrays can have as many dimensions as you like. However, after three dimensions, its impossible to picture their structure in higher dimensions. A three dimensional array can be pictured as at least two *rxc* matrices stacked on top of each other. With this in mind, each element would be described by its row and column in one of these matrices. A third value would be needed to describe which matrix in the stack.

Creating an array, is similar to creating a matrix. A vector of data values is needed. This is passed to the *array*( ) function. A secondary argument *dim* indicates the three dimensions. Each matrix is filled from the first row to the last, starting with the first column and moving left. Once the first matrix is filled, the second matrix in the stack if filled in the same way. Care must be taken to make sure the original data is in the desired order.

```
dat <- c(1:20, 101:120, 1001:1020)
arr <- array(data = dat, dim = c(4, 5, 3))
arr
```

```
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]  101  105  109  113  117
[2,]  102  106  110  114  118
[3,]  103  107  111  115  119
[4,]  104  108  112  116  120

, , 3

     [,1] [,2] [,3] [,4] [,5]
[1,] 1001 1005 1009 1013 1017
[2,] 1002 1006 1010 1014 1018
[3,] 1003 1007 1011 1015 1019
[4,] 1004 1008 1012 1016 1020
```

When displayed, each 'stack' is indicated by ' , , stack number'. With enough data, this can become very unmanagable to look through.

Filtering elements from an array works as you would for a matrix. In three dimensions, you list row, column, and then the stack. Leaving out a set of indices, indicates that all indices are desired. Depending upon the selection, *R* may return a vector, a matrix, or an array. Whichever form of data is simplest.

```
arr[1:2, 2:4, c(1, 3)]  # An Array
```

```
, , 1

     [,1] [,2] [,3]
[1,]    5    9   13
[2,]    6   10   14

, , 2

     [,1] [,2] [,3]
[1,] 1005 1009 1013
[2,] 1006 1010 1014
```

```
arr[, , 3]  # A matrix
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 1001 1005 1009 1013 1017
[2,] 1002 1006 1010 1014 1018
[3,] 1003 1007 1011 1015 1019
[4,] 1004 1008 1012 1016 1020
```

```
arr[1:3, 4, 2]  # A vector
```

```
[1] 113 114 115
```

The *str*( ) and *dim*( ) functions play the same roll they did for matrices. Names can be added, if desired. However, they could become very difficult to work with.

```
str(arr)

 int [1:4, 1:5, 1:3] 1 2 3 4 5 6 7 8 9 10 ...

dim(arr)

[1] 4 5 3
```

As with a vector and a matrix, an array can only hold one type of data: logical, numeric, or character.

An example of a four dimensional array is given below. Think of it as at least two three dimensional arrays stacked.

```
dat1 <- c(1:20, 101:120, 1001:1020)
dat2 <- c(21:40, 221:240, 2021:2040)
arr.4d <- array(data = c(dat1, dat2), dim = c(4, 5, 3, 2))
arr.4d

, , 1, 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20

, , 2, 1

     [,1] [,2] [,3] [,4] [,5]
[1,]  101  105  109  113  117
[2,]  102  106  110  114  118
[3,]  103  107  111  115  119
[4,]  104  108  112  116  120

, , 3, 1

     [,1] [,2] [,3] [,4] [,5]
[1,] 1001 1005 1009 1013 1017
[2,] 1002 1006 1010 1014 1018
[3,] 1003 1007 1011 1015 1019
[4,] 1004 1008 1012 1016 1020

, , 1, 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   21   25   29   33   37
[2,]   22   26   30   34   38
[3,]   23   27   31   35   39
[4,]   24   28   32   36   40

, , 2, 2

     [,1] [,2] [,3] [,4] [,5]
[1,]  221  225  229  233  237
[2,]  222  226  230  234  238
[3,]  223  227  231  235  239
[4,]  224  228  232  236  240

, , 3, 2

     [,1] [,2] [,3] [,4] [,5]
[1,] 2021 2025 2029 2033 2037
[2,] 2022 2026 2030 2034 2038
[3,] 2023 2027 2031 2035 2039
[4,] 2024 2028 2032 2036 2040
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Create the following **matrix** that is filled with **letters**:

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] "a"  "b"  "c"  "d"  "e"  "f"
[2,] "g"  "h"  "i"  "j"  "k"  "l"
[3,] "m"  "n"  "o"  "p"  "q"  "r"
[4,] "s"  "t"  "u"  "v"  "w"  "x"
```

2. Create the matrix displayed below.

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    9    3   12    5   15    7
[2,]    3   12    5   15    7    1    9
[3,]    5   15    7    1    9    3   12
[4,]    7    1    9    3   12    5   15
```

3. Extract the fourth row of the matrix **fruitMatrix**.

4. Extract the fifth column of the matrix **fruitMatrix**.

5. Create a new matrix by extracting a portion of the matrix **fruitMatrix**. It should consist of the elements of **fruitMatrix** that are in middle four rows of **fruitMatrix** AND the middle three columns of **fruitMatrix**.

6. Create a new vector by extracting a portion of the matrix **fruitMatrix**. It should consist of the elements in the fifth column of **fruitMatrix** that are also in the second, third, and fifth row of **fruitMatrix**.

7. Create a new matrix by extracting some columns from the matrix **numberMatrix**. The columns that are extracted should be those whose last element are between 15(included) and 60(excluded).

8. Create a new matrix by extracting some rows from the matrix **numberMatrix**. The rows that are extracted should be those whose first element is less than or equal to 25.

This page left intentionally blank for more problems

Chapter 12: Constants Values and Vectors

Several useful values and vectors are built into the base *R* package. These relieve you from needing to type them in manually, or loading them from outside source.

Pi

The numerical constant π can be obtained by using the *pi* variable.

```
pi

[1] 3.141593
```

Letters

The 26 letters of the Roman alphabet are loaded in either upper or lower case form into two vectors.

```
letters

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

LETTERS

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Months

The months of the year are also accesible.

```
month.abb

 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

month.name

 [1] "January"   "February"  "March"     "April"     "May"       "June"      "July"      "August"    "September" "October"   "N
ovember"  "December"
```

Sometimes, it is useful to have a larger collections of strings to experiment on. By loading the *stingr* package you are given access to several interesting

Fruit, Words, and Sentences

fruit gives access to a character vector containing 80 names of fruit.

```
library(stringr)
fruit[1:10]

 [1] "apple"        "apricot"      "avocado"      "banana"       "bell pepper"  "bilberry"     "blackberry"   "blackcurrant" "b
lood orange" "blueberry"
```

words gives access to a character vector containing 980 words.

```
words[1:10]

 [1] "a"        "able"     "about"    "absolute" "accept"   "account"  "achieve"  "across"   "act"      "active"
```

Sentences

sentences gives access to a character vector containing 980 words.

```
sentences[1:10]

 [1] "The birch canoe slid on the smooth planks."  "Glue the sheet to the dark blue background." "It's easy to tell the depth o
f a well."      "These days a chicken leg is a rare dish."    "Rice is often served in round bowls."
 [6] "The juice of lemons makes fine punch."       "The box was thrown beside the parked truck." "The hogs were fed chopped cor
n and garbage." "Four hours of steady work faced us."         "A large size in stockings is hard to sell."
```

A Lorem Ipsum Generator

The *stri_rand_lipsum*( ) is a Lorem Ipsum generator included in the *stringi* package. It will produce a string of dummy text. Its main purpose is to produce text that is useful for considering page layouts, without being distracted by the actual content.

```
library(stringi)
stri_rand_lipsum(n_paragraphs = 3, start_lipsum = TRUE)

[1] "Lorem ipsum dolor sit amet, odio ipsum arcu posuere nec sit tristique. Proin at ac maecenas platea libero, sapien at mi. S
ed orci a enim. Tristique, ac, mus per magna mauris nec nascetur et porta nec faucibus. Id at, nascetur sodales id vitae vel da
pibus. Metus mi, magna nunc eu nulla blandit. Faucibus tristique sapien curae sed hendrerit eu. Maximus, integer ac per accumsa
n odio ex id nulla ut vel facilisis. Eros, quisque netus congue aenean ut senectus."
[2] "Mauris blandit in sed ipsum, at leo tincidunt. Primis, luctus facilisi nisl, facilisi taciti curae. Gravida per, amet puru
s sem at tempus sociosqu non litora felis inceptos aptent facilisis. Ac donec tellus turpis congue. Risus, a eu in magnis ultri
cies nulla nibh. Metus vitae diam amet, curae pulvinar dolor ipsum sit. Erat aliquam eget urna congue magna ut, purus phasellus
dictum euismod. Vitae ante cras vulputate tellus eu. Velit sodales nullam mauris nunc tortor curae in feugiat sit sit elementum
, sed."
[3] "Felis rutrum, odio blandit aenean ultricies, diam! Proin euismod mus sed molestie ac at, pellentesque duis nam nisi. Orci
lectus hendrerit ligula eros accumsan. Tempor sociosqu per fringilla nisl duis vitae vestibulum fusce vel etiam ut porttitor, s
ed ad cubilia. Scelerisque donec sed amet placerat, aptent libero non sed. Eros in pretium nec velit, nisi. Commodo quam commod
o dictum aliquam fusce at curabitur habitant. Sed ut nunc sed in massa, rhoncus integer blandit. Dapibus id a elit in. In biben
dum quis curabitur sit lacus ante. Lorem purus condimentum phasellus arcu vehicula mi lobortis commodo dui dui."
```

The first argument determines the number of paragraphs to generate. The second indicates whether the text should start with 'Lorem ipsum dolor sit amet'.

Colors

While not a vector, the *colors*( ) function returns vector containing all the named colors in *R*. When producing graphics, these names can be used to set the color of a graphical object. There are 657 available to choose from.

```
clrs <- colors()
clrs[1:20]

 [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1" "antiquewhite2" "antiquewhite3" "antiquewhite4" "aquamarin
e"    "aquamarine1"   "aquamarine2"   "aquamarine3"   "aquamarine4"   "azure"         "azure1"        "azure2"
[16] "azure3"        "azure4"        "beige"         "bisque"        "bisque1"
```

Time Zones

Like the *colors*( ) function, the *OlsonNames*( ) function returns a vector. This vector returns a vector of time zones available in *R*

```
tz <- OlsonNames()
tz[1:10]

 [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa" "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
"Africa/Bamako"      "Africa/Bangui"      "Africa/Banjul"      "Africa/Bissau"
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

These should not be completed by inspecting the given vectors and typing in the answer.

1. What is the cube root of the fifth power of pi?

2. Make a character vector with a single element. The element should be a string with all the letters of the English alphabet in reverse alphabetical order. Both upper case and lower case letters should appear. Each lower case letter should be immediately followed by the upper case version. Letters should not be individually typed.

3. For each fruit that begins with the letter B in the **fruit** vector, determine how many characters are in its name. The result should be a single numberical vector.

4. TRUE/FALSE There is at least one sentence in **sentences** whose last letter is the letter z(either case). The result should be a logical vector of length one.

5. TRUE/FALSE All the words in **words** are at most 11 characters long. The result should be a logical vector of length one.

6. How many Olson time zones indicate in their text Europe?

7. For each fruit that ends with the letter r in the **fruit** vector, determine how many characters are in its name. The result should be a single numerical vector.

8. TRUE/FALSE There is at least one sentence in **sentences** whose last two letters, in any order are r and e (either case). The result should be a logical vector of length one.

9. TRUE/FALSE All the words in **words** are at least 2 characters long. The result should be a logical vector of length one.

10. How many Olson time zones indicate in their text **G**reenwich **M**ean **T**ime?

This page left blank for more questions to be added in the future.

Chapter 13: Infinity and Undefined Values

Some computations result in outcomes that are too large to *R* to store or are undefined mathematically.

Infinity

In situations, the result of a computation is larger (in absolute value) than can be stored. *R* will store this value as $\pm Inf$, either a positive or negative infinity, depending on the case.

```
2^9999

[1] Inf

-10^1000

[1] -Inf
```

While dividing a non-zero value by zero is undefined in some contexts. It is often useful to assign an infinite value to this result. Depending on the numerator, *R* will indicate this with either a positive or negative infinity.

```
1/0

[1] Inf

-pi/0

[1] -Inf
```

As far as *R* is concerned, these infinities are numerical values. Standard arithmetic operations can be performed with them, and a consistent result will be given.

```
typeof(Inf)

[1] "double"

Inf + 1

[1] Inf

10/Inf

[1] 0
```

Testing for an infinity can be done a few ways. The `==` operator will detect either a positive or negative infinity. $is.infinite(\quad)$ will detect either type.

```
x <- c(1, -1)/0
x == Inf

[1]  TRUE FALSE

x == -Inf

[1] FALSE  TRUE

is.infinite(x)

[1] TRUE TRUE
```

Not a Number

Some operations that are undefined should not produce a meaningful numeric result. This can occur when you are subtracting infinite values, or multiplying/dividing infinite values and zero. When the result should not be a meaningful numeric result, *R* indicates that the result is 'Not a Number'.

```
Inf - Inf

[1] NaN

Inf * 0

[1] NaN

0/0

[1] NaN

Inf/Inf

[1] NaN
```

While NaN is not a meaningful numerical value, *R* will still allow arithmetic operations to be performed on it. The result will also be NaN.

```
typeof(NaN)

[1] "double"

NaN + 1

[1] NaN
```

However, identifying a NaN value can not be done using the usual == double equal sign. Instead, the $is.nan(\quad)$ function must be used.

```
NaN == NaN

[1] NA

is.nan(NaN)

[1] TRUE
```

If you want to identify NaN values or missing values NA, $is.na(\quad)$ can be used.

```
NaNNA <- c(NaN, NA, 2)
is.na(NaNNA)

[1]  TRUE  TRUE FALSE
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

These should not be completed by inspecting the given vectors and typing in the answer.

1. How many **PossiblyInfinite** values are infinite?

2. How many **PossiblyNAN** values are not a number?

This page left blank for more questions to be added in the future.

Chapter 14: Missing Values & Vectors

When a data value is missing for some reason, it still needs to be recorded that some values should be listed. In R, NA can be used to indicate a missing data value. It stands for 'Not Available'.

```
NA

[1] NA

typeof(NA)

[1] "logical"
```

As you can see, NA on its own is actually a third type of logical value. The other two being $TRUE$ and $FALSE$. However, when NA is used in vector containing some other data type, it will be coerced/changed into the appropriate data type, be it numeric or character data.

```
givenVector <- c(1, 2, 3, NA, NA)
typeof(givenVector[4])

[1] "double"
```

Operations with Missing Values

Performing operations with NA values can be tricky. The NA value will infect the operations, and generally lead to a result of NA. This is something to be aware of.

```
givenVector + 1

[1]  2  3  4 NA NA

0 * NA

[1] NA
```

Many functions will have an logical argument $na.rm$. If this argument is set to TRUE, missing values will be ignored.

Detecting Missing Values

The == double equal sign operator can't be used to detect NA values. It will return a value of NA.

```
NA == NA

[1] NA
```

As a workaround, two functions are provided that can logically check for NA values. $is.na(\ \ )$ is a vectorized function that will check each element of a vector to see if it is NA and return $TRUE/FALSE$ for each. $anyNA(\ \ )$ will return a single logical value. If at least one value is NA, $TRUE$ will be returned.

```
is.na(NA)

[1] TRUE

is.na(givenVector)

[1] FALSE FALSE FALSE  TRUE  TRUE

anyNA(givenVector)

[1] TRUE
```

Replacing Missing Values

Certain situation require replacing NA values with another value. If all NA values will be replaced with the same value, the replacement is done with a short line of code.

```
# Replace the NA values in givenVector with -100.
givenVector

[1]  1  2  3 NA NA

givenVector[is.na(givenVector)] <- -100
givenVector

[1]    1    2    3 -100 -100
```

The first line of code displays givenVector. The last two elements are NA. The second line of code does the replacement. As it is written, it can be tricky to read. $is.na(givenVector)$ produces a vector of $TRUE/FALSE$ values. Placing that vector inside the square brackets filters the values in givenVector corresponding to TRUE. Finally, the assignment operator assigns -100 to those two values.

NULL

NA is used to indicate that a value within a vector is absent. The $NULL$ value is used to indicate the absence of a vector. Sometimes, $NULL$ will be the result of an undefined computation. Sometimes, $NULL$ will be used to erase some information.

```
length(NULL)

[1] 0

typeof(NULL)

[1] "NULL"
```

Detecting a $NULL$ value is done with the $is.null(\quad)$ function.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Determine if the given vectors contain any missing values. Your answer should be a single element logical vector for each. This should be indicated by a TRUE value if the given vector contains any missing values. Otherwise, FALSE. This should not be done by inspection.

    a. MissingValues01
    b. MissingValues02
    c. MissingValues03
2. Determine if the given vectors contain any missing values. Your answer should be a single element logical vector for each. This should be indicated by a TRUE value if the given vector contains any missing values. Otherwise, FALSE. This should not be done by inspection.

    a. firstNames
    b. lastNames
    c. idNumbers
3. Determine the indices of all missing values in the given vectors. Your answer should be a single vector for each of the following.

    a. MissingValues01
    b. MissingValues02
    c. MissingValues03
    d. firstNames
    e. lastNames
    f. idNumbers
4. Create a vector that contains all the non-missing values of the vector indicated below. (In order to not cause problems in future questions, the new vector should have the same name as the vector indicated below with the letter X appended to the end.)

    a. MissingValues01
    b. MissingValues03
    c. firstNames
    d. lastNames
5. For each pair of vectors, display the elements of the first vector that correspond to the missing elements of the second vector.

    a. firstNames; lastNames
    b. lastNames; firstNames
    c. idNumbers; firstNames
    d. idNumbers; lastNames
    e. lastNames; idNumbers
    f. A vector of both corresponding first and last names, last name first, seperated by a comma; idNumbers
6. For each vector below, replace the missing values with the indicated quantity/string. Do not display the result.

    a. firstNames; Kobalt
    b. lastNames; 000000

c. idNumbers; G-Ma

Waiting for more problems

Chapter 15: Coercion

The data type of a vector is determined by the values entered into it. However, **R** has two methods for coercing (changing) the data type of a vector: implicit and explicit coercion.

Implicit coercion occurs when a vector is created using a mixture of data types. In situations where an attempt is made to mix data types into a single vector, **R** will coerce the different data types into a common data type. A hierarchy exist for this implicit coercion with logical data at the bottom, character data at the top, and numeric data in the middle. When data types are mixed, lower types will move as high up as possible.

Explicit coercion occurs when a function is applied to a vector to change its data type. This does not need to be a vector that was created with mixed data types. Explicit coercion can be used to move vectors either up or down the hierarchy. However, care must be taken as vectors are moved down the hierarchy as some values can not be coerced 'down' and will be converted into *NA* values.

Implicit Coercion - Mixed Data Types

Numeric Data

As numeric data falls in the middle of the hierarchy, when mixed with other types it will either remain its own type of become character data. In this example, several numeric values are written into a vector that contains a character element.

```
Numeric_Char <- c("STRING", 0, 1, 2)
Numeric_Char

[1] "STRING" "0"      "1"      "2"
```

Each numerical value in the output is now surrounded by quotation marks. This is an indication that each numeric value is regarded as a single character now. These characters have lost their numeric value and mathematical operations can no longer be performed on them. Errors will result.

```
"0" * 1

Error in "0" * 1: non-numeric argument to binary operator
```

Logical Data

At the bottom of the hierarchy, logical data can be coerced into either numeric or character data depending upon what other data types are mixed in a given vector. This mix also affects the output.

A mix of logical and numeric data will become numeric data. The coercion process will turn TRUE values into the number one and FALSE values into the number zero.

```
Logical_Num <- c(9, TRUE, T, FALSE, F)
Logical_Num

[1] 9 1 1 0 0

typeof(Logical_Num)

[1] "double"
```

Additionally, if mathematical operations are performed on a set of vectors that mix such values, the same numerical replacement will occur.

```
T + 10

[1] 11
```

Mixing Logical data with Character data will result in a direct conversion into character data, even if numerical values are present.

```
Logical_Char <- c("STRING", TRUE, T, FALSE, F, 10)
Logical_Char

[1] "STRING" "TRUE"   "TRUE"   "FALSE"  "FALSE"  "10"
```

The existence of a character string determines that the resulting vector will be a character vector. The logical values are coerced into string values of either "TRUE" or "FALSE" even if T and F are used[18].

Explicit Coercion

With the use of additional functions, the specifics of the coercion can be better controlled.

Becoming Character Data

The $as.character(\ \ )$ function can be used to explicitly coerce a vector into a character data.

```
allNumeric <- as.character(c(1, 2, 3))
allLogical <- as.character(c(TRUE, FALSE, T, F))
allNumeric

[1] "1" "2" "3"

allLogical

[1] "TRUE"  "FALSE" "TRUE"  "FALSE"
```

With the surrounding quotes, it should be clear that these vectors contain character data even though they were written as numeric/logical withing the $as.character(\ \ )$ function.

Becoming Numeric Data

The $as.numeric(\ \ )$ function can be used to explicitly coerce a vector into a numeric vector. $as.numeric(\ \ )$ will coerce numeric characters into numeric values. This allows mathematical computations to be performed on them. However, strings that can not be interpreted as numeric values will be converted in to NA values[19].

```
Char_to_Num <- as.numeric(c("4", "-100.2", "Ten"))
Char_to_Num

[1]    4.0 -100.2     NA
```

Similarly, $as.numeric$ can be used to explicitly coerce logical data into numeric data. The numeric conversion of TRUE becoming a one and FALSE becoming a zero occurs.

```
Logical_to_Num <- as.numeric(c(T, F, TRUE, FALSE))
Logical_to_Num

[1] 1 0 1 0
```

Becoming Logical Data

The $as.logical(\ \ )$ function can be used to explicitly coerce a vector into a logical vector data. As with $as.numeric(\ \ )$, only certain values can be coerced into logical values. All others will become NA values.

```
Char_to_Logi <- as.logical(c("TRUE", "FALSE", "T", "F ", "0", "1", "Character"))
Char_to_Logi

[1]  TRUE FALSE  TRUE    NA    NA    NA    NA
```

The only coercible strings are "TRUE","FALSE", "T", "F"[20] . Adding even so much as a space to any of these will result in an NA value as is clear from the string "F".

When using $as.logical(\ \ )$ on a numeric vector, every numeric value will be converted into TRUE except for the value zero. Zeros will be converted into FALSE.

```
Num_to_Logi <- as.logical(c(0, 1, 10.9, -2))
Num_to_Logi

[1] FALSE  TRUE  TRUE  TRUE
```

[18] Notice that the presense of the numeric value 10 does not cause the logical values to become zeros and ones before becoming character strings.
[19] When this happens, a warning can be displayed
[20] "True", "true". , "False", and "false" will also be coerced to TRUE and FALSE.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Coerce the indicated vector into the indicated type. Additionally, produce a logical value where it is TRUE if the resulting vector contains missing values (FALSE otherwise).

    a. Coerce01; logical
    b. Coerce02; logical
    c. Coerce03; character
    d. Coerce04; logical
    e. Coerce01; numerical
    f. Coerce05; numerical
2. Coerce the indicated vector into the indicated type insuch a way that missing values are not introduced.

    a. Coerce01; logical
    b. Coerce02; logical
    c. Coerce03; character
    d. Coerce04; logical
    e. Coerce04; numerical
    f. Coerce05; character with numeric characters.
3. Using some/all of vectors Coerce01, Coerce02, and Coerce03 (at most once) create a vector with at least six elements of the indicated type. (Implicit coercion only should be used.) The vectors should be used with their names in increasing order.

    a. character
    b. numeric
    c. logical
4. Using some/all of vectors Coerce01, Coerce02, and Coerce03 (at most once) create a vector with at most three elements of the indicated type. Explicit coercion should be used. The resulting vector should not have any missing values.

    a. character
    b. numeric
    c. logical
5. Combine three different vectors whose name starts **Coerce** in such a way that the resulting vector does not have any missing values and is of the following type. Both implicit and explicit coercion can be used. Display the vector you created and its type. The vectors should be used with thier names in increasing order.

    a. character
    b. numeric
    c. logical

More to come

Chapter 16: Lists

Lists provide another way to store data. Lists differ from vectors in that lists can hold many different types of data.

Making a List

As a basic example, three vectors are created, each of a different type and different length.

```
numbers <- 1:10
characters <- letters[1:6]
logicals <- rep(c(T, F), times = 2)
```

To create a list containing these three vectors, use the *list*( ) function and type in the names of the created vectors. As *R* creates the list, it actually coerces/changes these vectors into sublists.

```
newList <- list(numbers, characters, logicals)
newList

[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]
[1] "a" "b" "c" "d" "e" "f"

[[3]]
[1]  TRUE FALSE  TRUE FALSE
```

As you examine printed list, you will see two sets of indexes. The double square brackets [[]] indicate which sublist being displayed. The single bracket [] are used to index the vectors themselves.

List Attributes

Besides the data explicitly contained in a list, other pieces of information can be attached to it. Names can be given to each sublist. This can be done in two different ways.

Names can be added, modified, and retrieved, and using the *names*( ) function[21]. It requires a character vector with a name for each sublist. The names appearing in the same order as the sublists.

```
names(newList) <- c("LN", "LC", "LL")
```

Alternatively, the names can be added as the list is being created. Notice that the [[]] are replaced by dollar sign and a name with a named list.

```
namedList <- list(LN = numbers, LC = characters, LL = logicals)
namedList

$LN
 [1]  1  2  3  4  5  6  7  8  9 10

$LC
[1] "a" "b" "c" "d" "e" "f"

$LL
[1]  TRUE FALSE  TRUE FALSE
```

Comments can be added to describe the contents of the list using the comment() function. Comments should be written as strings. The comment() function displays the comments also.

```
comment(namedList) <- "This is a comment on a list."
comment(namedList)

[1] "This is a comment on a list."
```

The attributes() function can be used to display a summary of these additional attributes.

```
attributes(namedList)

$names
[1] "LN" "LC" "LL"
```

[21] The same as is can with vectors.

```
$comment
[1] "This is a comment on a list."
```

List Structure

When trying to access the information in a list, it is useful to look at its underlying structure using the *str*( ) function. The *str*( ) function will list the names of each component as well as a simplified summary of the information contained in that component.

```
str(newList)

List of 3
 $ LN: int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ LC: chr [1:6] "a" "b" "c" "d" ...
 $ LL: logi [1:4] TRUE FALSE TRUE FALSE

str(namedList)

List of 3
 $ LN: int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ LC: chr [1:6] "a" "b" "c" "d" ...
 $ LL: logi [1:4] TRUE FALSE TRUE FALSE
 - attr(*, "comment")= chr "This is a comment on a list."
```

The *str*( ) function is very useful with extracting information from lists. It can be very easy to extract information in a form that is different than you desired.

List within Lists

As was said before, the components of a list can be another list. Creating one is done in the same way we have already created a list.

```
listOfLists <- list(newList, "New Data")
listOfLists

[[1]]
[[1]]$LN
 [1]  1  2  3  4  5  6  7  8  9 10

[[1]]$LC
[1] "a" "b" "c" "d" "e" "f"

[[1]]$LL
[1]  TRUE FALSE  TRUE FALSE


[[2]]
[1] "New Data"

str(listOfLists)

List of 2
 $ :List of 3
  ..$ LN: int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ LC: chr [1:6] "a" "b" "c" "d" ...
  ..$ LL: logi [1:4] TRUE FALSE TRUE FALSE
 $ : chr "New Data"
```

Notice that there are two levels to the this list. It is not that easy to see when displaying the list. However, looking at the *str*( ) of the list, we can see that *listOFLists* is a list made up of two components. The first component, listed after the first dollar sign, is a list with three components. The second component, listed after the second dollar sign, is a single piece of character data.

Vectors & Lists

We have been rather loose with the term vector. In *R*, vectors come in two flavors: atomic vectors and lists. Atomic vectors are those vectors that contain one type of data. Lists are vectors that can contain many types. We have a special word, lists, for the second type, it won't cause a problem referring to atomic vectors as 'vectors'.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Make a list called **Lists001**. It should have five components. None of the components should have names. Display **ONLY** the structure of **Lists001**. The content of each component is as follows:
    a. CharacterData001
    b. Coerce05
    c. MissingValues01
    d. VectorsZero
    e. PossiblyNAN
2. Make a list called **Lists002**. It should have three components. The components should be named. Display **ONLY** the structure of **Lists002**. The content of each component is as follows:
    a. Chopper: a vector with length 47 that starts at -23 and ends at -178. The values should be evenly spaced.
    b. Hera: a vector of length 30 with all TRUE values.
    c. UpperBackwards: a single string with the upper case alphabet recorded in reverse alphabetical order.
3. Make a list called **Lists003**. It should have two components. Only the first component should be named. Display **ONLY** the structure of **Lists003**. The content of each component is as follows:
    a. fruityFoods: The matrix fruitMatrix
    b. A character vector with 10 elements. All but the first are empty strings. The first value is missing.
4. Make a list called **Lists004**. It should have four components. The components should be named. Display **ONLY** the structure of **Lists004**. The content of each component is as follows:
    a. InsideList: A list containing two unnamed components.
        i. A single string of the character “a”.
        ii. A numerical vector with two values: 0, 1.
    b. InsideMatrix: a matrix with 2 rows and three columns. All the values should be ones.
    c. InsideMissing: A vector with length 10. All elements should be missing. It should have type character.

{*MoreListssometime*}

Chapter 17: Modifying Lists

Data From Lists

When extracting a multiple components from a list, the result can be **preserved** as a list, or can be possibly **simplified** into a 'simpler' data type. The choice of which method to use will depend on how the result will be used.

```
namedList

$LN
 [1]  1  2  3  4  5  6  7  8  9 10

$LC
[1] "a" "b" "c" "d" "e" "f"

$LL
[1]  TRUE FALSE  TRUE FALSE
```

Preservation

To preserve the data as a list, use the name of the list followed by a single set of square brackets. Inside the square brackets, include a vector with the indices or the names of the desired components. A single sublist is extracted with just a numerical index or a sublist name.

```
namedList[1]
namedList["LN"]
```

The result of extracting a single sublist, using a single set of square brackets will be a list.

```
$LN
 [1]  1  2  3  4  5  6  7  8  9 10

[1] "list"
```

Operations and functions that perform perfectly well on vectors, might not work on lists.

```
namedList[1] + 1

Error in namedList[1] + 1: non-numeric argument to binary operator
```

If desired, multiple components can be filtered out. This is done using a vector of indexes or a vector of names.

```
namedList[c(1, 3)]
namedList[c("LN", "LL")]

$LN
 [1]  1  2  3  4  5  6  7  8  9 10

$LL
[1]  TRUE FALSE  TRUE FALSE
```

Simplification

Another way to extract the information from a list is to extract and attempt to simplify it. When *R* simplifies a list, if attempts to coerce the list into a 'simpler' form, from *R*s point of view. Simplification, of a single sublist, is attempted either by using double square brackets or a dollar sign followed by the component name.

```
namedList[[3]]
namedList$LL

[1]  TRUE FALSE  TRUE FALSE

typeof(namedList$LL)

[1] "logical"
```

In either case, for this example, the result is just a logical vector.

The *unlist*( ) function provides a way to collapse the components of a list into a single vector regardless of the number of components.

```
unlist(namedList[c(1, 3)])

 LN1  LN2  LN3  LN4  LN5  LN6  LN7  LN8  LN9 LN10  LL1  LL2  LL3  LL4
   1    2    3    4    5    6    7    8    9   10    1    0    1    0
```

Modifying List Components

Modifying list components can be done by assigning a new set of values to a particular component. This can be done either with names or numerical index.

```
namedList[2] <- TRUE
namedList$LC <- TRUE
namedList[[c(1, 1)]] <- 2
str(namedList)

List of 3
 $ LN: num [1:10] 2 2 3 4 5 6 7 8 9 10
 $ LC: logi TRUE
 $ LL: logi [1:4] TRUE FALSE TRUE FALSE
 - attr(*, "comment")= chr "This is a comment on a list."
```

Adding components

Adding Components can be done three ways. Two use the assignment operator. In either of these cases, you assign your new data to a component that doesn't already exist. Either refer to the next component index, or assign a new name using the dollar sign.

```
newDataIndex <- length(namedList)
namedList[newDataIndex + 1] <- "New Data"
namedList$ExtraData <- "Newer Data"
str(namedList)

List of 5
 $ LN       : num [1:10] 2 2 3 4 5 6 7 8 9 10
 $ LC       : logi TRUE
 $ LL       : logi [1:4] TRUE FALSE TRUE FALSE
 $          : chr "New Data"
 $ ExtraData: chr "Newer Data"
 - attr(*, "comment")= chr "This is a comment on a list."
```

When referring to the next component index, it is important to check the length of the list. Otherwise, several empty list components can be added.

The third method for adding a component to a list uses the *append*( ) function. It requires the list you have, and the components you want at add to it. This method requires that you assign you result to a variable, otherwise the change will not take place. The previous methods update the existing list.

```
namedList <- append(namedList, "Even More New Data")
str(namedList)

List of 6
 $ LN       : num [1:10] 2 2 3 4 5 6 7 8 9 10
 $ LC       : logi TRUE
 $ LL       : logi [1:4] TRUE FALSE TRUE FALSE
 $          : chr "New Data"
 $ ExtraData: chr "Newer Data"
 $          : chr "Even More New Data"
```

Removing Components

To remove a component, assign the value *NULL* to that component. This example will remove all but the first sublist.

```
namedList[2:6] <- NULL
str(namedList)

List of 1
 $ LN: num [1:10] 2 2 3 4 5 6 7 8 9 10
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Determine the structure of the following:

    a. ModifyLists001
    b. ModifyLists002
    c. ModifyLists003
2. If possible, extract the indicated component(s) from each of the following lists. The result should remain a list. Display the structure of the result and not the result itself.

    a. The first component of ModifyLists001
    b. The last component of ModifyLists002
    c. The third component of ModifyLists003
3. If possible, extract the indicated component(s) from each of the following lists. The result should remain a list.

    a. The first two components of ModifyLists001
    b. The last three components of ModifyLists002
    c. The even components of ModifyLists003
4. If possible, extract the indicated component(s) from each of the following lists. The result should be a vector.

    a. The first component of ModifyLists001:
    b. The burger component of ModifyLists002
    c. The graflex component of ModifyLists003
    d. The first two components of ModifyLists001
    e. The last five component of ModifyLists002
    f. The even components of ModifyLists003
5. If possible, extract the indicated values(s) from each of the following lists.

    a. The last two elements of the third component of ModifyLists001.
    b. The first five **wild** elements of ModifyLists002.
    c. The second through fourth element of **feet** in ModifyLists003. The result should be a character vector.
    d. The elements of **feet** in ModifyLists003 that are less than 10000 numerically. The result should be a character vector and not missing.
6. Remove the unnamed elements of ModifyLists003. Display the structure of the result. Do not modify the original list. Make a copy of it before you remove the elements.

7. Create a new list **CreatedList001** with three components. The components should be ModifyLists001, ModifyLists002, and ModifyLists003.

8. Create a new list **CreatedList002** with all components of the lists ModifyLists001, ModifyLists002, and ModifyLists003.

Lists

Chapter 18: DataFrames

When performing data analysis, most likely you will be using a **dataframe** to hold your data. A dataframe is a list where each component has the same length. To visualize a dataframe, think of a matrix whose columns can hold different types of data. The columns correspond to the components of a list. Generally, each column represents a single variable.

Creating a dataframe

Dataframes can be created many ways. Importing data is one way. But, the simplest way to make a dataframe is from a collection of vectors.

```
numbers <- 1:26
characters <- letters[1:26]
logicals <- rep(c(T, F), times = 13)
```

Using the $data.frame(\ \ )$ function, list the vectors that you want to include, separated by commas. Names can be included, if desired.

```
dat <- data.frame(numbers, charName = characters, 101:126, `2 name` = logicals)
# `2 name` is an incorrectly formatted name. The back-ticks 'allow' it.
head(dat, n = 4)  # Change 6 to number of lines you want see.

  numbers charName X101.126 X2.name
1       1        a      101    TRUE
2       2        b      102   FALSE
3       3        c      103    TRUE
4       4        d      104   FALSE
```

Above, you can see the structure of a dataframe. You should notice, that the values in the *numbers* and *characters* vectors each have 26 elements. The dataframe as displayed only shows the first four rows. The $head(\ \ )$ function was used to display just the beginning of the dataframe. Entering dat alone, without using $head(\ \ )$, would print all 26 lines. In this case, that is not much to look at, but consider a data frame with 10,000 rows. The output can be overwhelming. There is a $tail(\ \ )$ function. It displays the end of a dataframe. The $dim(\ \ )$ function will display the number of rows and columns.

If we continue to look at the structure of the dataframe, we will notice some behavior of $data.frame(\ \ )$.

```
str(dat)

'data.frame':   26 obs. of  4 variables:
 $ numbers : int  1 2 3 4 5 6 7 8 9 10 ...
 $ charName: chr  "a" "b" "c" "d" ...
 $ X101.126: int  101 102 103 104 105 106 107 108 109 110 ...
 $ X2.name : logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

First, $data.frame(\ \ )$ assigns a name to each component that is added. (Remember, these components are each columns.) The name for each column will come from either

1. the named variable (numbers)
2. a name given prior to naming the variable (charName)
3. some convention *R* uses for making up a name, if the first two types of names are not available.

Secondly, notice that $data.frame(\ \ )$ changed the name of a component. An *X* was appended to `2 name` because it starts with number. Spaces are replaced with a period. Basically, improperly formatted names are 'fixed'. Improperly formatted names are common in imported data sets.

Finally, if we check the attributes of a dataframe, we can see that besides names, it has two other attributes namely *class* and *row.names*.

```
attributes(dat)

$names
[1] "numbers"  "charName" "X101.126" "X2.name"

$class
[1] "data.frame"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

Subsetting Dataframes

Elements of a dataframe are identified by both the row and column they are in. Using numerical vectors to select a single element, identify the row index and the column index. The "d" is in the fourth row and second column of the dataframe **dat**. To select "d", type the name of the dataframe with a set of square brackets. Inside the brackets, enter the row index, a comma, and the column index.

```
dat[4, 2]

[1] "d"
```

If you want to restrict yourself to specific columns, at least two methods exist.

1. In the square brackets, enter a vector of column indexes or names. (This will return a dataframe.)
2. Replace the square brackets with a dollar sign, followed by the column name. (This will return a vector.)

```
dat[, c(1, 3)]  # The leading comma is not needed when subsetting columns
dat[, c("numbers", "X101.126")]
dat$numbers
```

To replace a single value, a set of values, entire rows, or entire columns, use the assignment operator and reference the

Subsetting Entire Rows

Subsetting specific rows can be done in several different ways. After the vector name, and surrounded by square brackets

1. enter a numeric vector of desired rows, followed by a comma. (A vector of negative values will remove rows)
2. enter a logical vector followed by a comma. $TRUE$ values will be kept.
3. enter a comparison, based on a column, followed by a comma. The comparison is usually based on values in a specific column.

As with vectors, the $subset(\ \ )$ function can also be used. The first argument is a dataframe, the second is the subset selection condition(s).

```
dat[c(1, 3), ]
dat[c(T, F, T, rep(F, times = 23)), ]
dat[dat$charName == "a" | dat$charName == "c", ]
subset(dat, dat$charName == "a" | dat$charName == "c")
```

Each of these filters out the same two rows.

```
  numbers charName X101.126 X2.name
1       1        a      101    TRUE
3       3        c      103    TRUE
```

Missing Values

Determining which rows of a dataframe have missing values can be important. $complete.cases(\ \ )$ will return a logical vector where $FALSE$ will indicate rows with missing values. $na.omit(\ \ )$ will return a dataframe with incomplete rows removed. However, columns of interest could be complete. In that case, removing rows is a bad idea.

Extending a dataframe

New columns can be added to a dataframe in several ways:

1. Use $data.frame(\ \ )$ listing the original dataframe with any new columns. Assign the result to a variable.
2. Using the dollar sign, add a new name to dataframe and assign the new data to it.
3. Use $cbind(\ \ )$ to bind a column to the existing data frame. Assign the result to a variable.

```
dat <- data.frame(dat, newNumbers1 = 1:26)
dat$newNumbers2 <- 11:36
dat <- cbind(dat, newNumbers3 = 21:46)
head(dat, n = 1)

  numbers charName X101.126 X2.name newNumbers1 newNumbers2 newNumbers3
1       1        a      101    TRUE           1          11          21
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Construct a single dataframe that has ALL of the following properties. The construction of each should not be done by brute force (listing out elements).

    a. Name: example.dataframe.1,
    b. Four variables with 10 observations each.
    c. variable.1 contains ten replications of the string 1. This should be caled variable.1.
    d. variable.2 contains numbers 1 - 10 in increasing order. This should be caled variable.2.
    e. variable.3 contains alternating logical values starting with $TRUE$. This should be caled variable.3
    f. variable.4 contains the first 10 colors from the colors function. This should be caled variable.4.
2. Construct a single dataframe that has ALL of the following properties. The construction of each should not be done by brute force (listing out elements).

    a. Name: example.dataframe.2,
    b. Three variables with 78 observations each.
    c. variable.1 contains the letters of the lowercase alphabet each repeated 3 times consecutively.
    d. variable.2 contains numbers 1 - 78 in decreasing order.
    e. variable.3 contains the last 78 colors in the colors() function. Starting with the last in the list alphabetically.
3. How many rows and columns do these dataframes have? This should be done with a function, not by inspection.

    a. ChickWeight
    b. Formaldehyde
    c. Loblolly
4. Extract the following from the indicated dataframe: (This should not be done by inspection. The result should be a data.frame.)

    a. Loblolly: The age variable
    b. Formaldehyde: The optical density variable
    c. ChickWeight: The weight variable
5. Extract the following from the indicated dataframe: (The result should be a vector.)

    a. Loblolly: The age of all Loblolly pine trees with a height over 55 ft.
    b. Formaldehyde: The optical density when the Carbohydrate is greater than 0.4 ml
    c. ChickWeight: The weight of chicks at 4 days old.
6. Extract the following from the indicated dataframe: (This should not be done by inspection. The result should be a dataframe)

    d. Loblolly: Rows with heights over 60 ft
    e. Formaldehyde: Rows with carbs over 0.5 ml
    f. ChickWeight: Rows at age 0 days.

Questions. Questions. Questions. I need more questions

Chapter 19: Tibbles

A **tibble** is a modified version of a **dataframe**[22]. The *tibble* package adds the necessary functionality to *R*.

```
library(tibble)
```

Creating a tibble

As with dataframes, tibbles can be created in different ways. They are the result of importing data. They are the result of using the *tibble*( ) function in a manner analogous to using *data.frame*( ) to bind vectors together. For comparison, a tibble and a dataframe are constructed from the same set of vectors. As with a dataframe, list the vectors, possibly with names, separated by commas[23].

```
numbers <- 1:26
characters <- letters[1:26]
logics <- rep(c(T, F), length = 26)
# Building a dataframe
dat <- data.frame(numbers, charName = characters, 101:126, `2 name` = logics)
# Building a tibble.
tib <- tibble(numbers, charName = characters, 101:126, `2 name` = logics)
```

A tibble can also be built by directly typing in the values row by row. The is done with the *tribble*( ) function. To do so, list the column names separated by a comma in the first row of entered data. Immediately precede each name with a tilde. One subsequent lines enter the data row by row separating values with a comma.

```
tribble.example <- tribble(~var.1, ~var.2, 1, "q", 2, "w", 30, "e")
tribble.example

# A tibble: 3 × 2
  var.1 var.2
  <dbl> <chr>
1     1 q
2     2 w
3    30 e
```

A tibble can also be made direct from a dataframe, vector, or matrix. The *as_tibble*( ) function will attempt to coerce these into a tibble. Going the other direct, the *as.data.frame*( ) function can be used to coerce a tibble into a data.frame[24].

Differences between tibble and dataframe

For comparison, the tibble **tib** and dataframe **dat** are displayed below:

```
tib

# A tibble: 26 × 4
   numbers charName `101:126` `2 name`
     <int> <chr>        <int> <lgl>
 1       1 a              101 TRUE
 2       2 b              102 FALSE
 3       3 c              103 TRUE
 4       4 d              104 FALSE
 5       5 e              105 TRUE
 6       6 f              106 FALSE
 7       7 g              107 TRUE
 8       8 h              108 FALSE
 9       9 i              109 TRUE
10      10 j              110 FALSE
# ℹ 16 more rows

# `2 name` is an incorrectly formatted name. The back-ticks 'allow' it.
head(dat, n = 4)

  numbers charName X101.126 X2.name
1       1        a      101    TRUE
2       2        b      102   FALSE
3       3        c      103    TRUE
4       4        d      104   FALSE
```

[22] Most of this information is drawn from _vignette("tibble").
[23] For tibbles, *tibble*( ) will only recycle vectors of length one.
[24] Older functions may not recognize tibbles

Things to notice about these two:

1. The tibble did not change the name of any of the columns. Spaces were not turned into periods. No X was appended to the start of the improperly named columns.

2. No extra command was needed to restrict the values displayed by the tibble. By default, a tibble will only display 10 rows, and as many columns as will fit nicely on the screen, or page.

3. At the head of each column, the tibble displays the data type.

Something not visible in the printout; the type of data entered into $tibble(\ )$ does not change. In older versions of $data.frame(\ )$ is would change some data types[25].

Subsetting a tibble

Selecting an element, or set of elements, from a tibble follows the same process as selecting from a dataframe with a few tweaks[26].

1. If a column has an incorrectly formatted name, the name needs to be surrounded by back tick to use it.

2. When using a $ followed by a name, a data frame does not require the entire name to be included. It will look for a partial match. A tibble requires the entire name.

```
tib$`2 name`

 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  T
RUE FALSE  TRUE FALSE  TRUE FALSE

dat$num

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

tib$num

NULL
```

An added behavior

When creating a tibble, one column can refer to another column, and perform operations on it to construct another column.

```
x <- 11:13  # OUTSIDE of tibble()
z <- 101:103  # OUTSIDE of tibble()
tib <- tibble(x = 1:3, y = x^2, z)
tib

# A tibble: 3 × 3
      x     y     z
  <int> <dbl> <int>
1     1     1   101
2     2     4   102
3     3     9   103
```

Look at the two vector constructed before the tibble. The $tibble(\ )$ function ignores the vector x outside of the $tibble(\ )$ function in favor of the one created inside. However, there is no vector z created inside the $tibble(\ )$ function. So, the vector z outside the $tibble(\ )$ function is used.

[25] The latest version of $data.frame(\ )$ does not to this. The argument $stringsAsFactor$ defaults to FALSE.
[26] Knowing the dimensions of a tibble is useful when subsetting. The $dim(\ )$ function can be used for this just as it could with dataframes, matrices, and vectors.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Construct a single tibble that has ALL of the following properties. The construction of each should not be done by brute force (listing out elements).

    a. Name: example.tibble.1,
    b. Four variables with 10 observations each.
    c. variable.1 contains ten replications of the string 1. This should be caled variable.1.
    d. variable.2 contains numbers 1 - 10 in increasing order. This should be caled variable.2.
    e. variable.3 contains alternating logical values starting with $TRUE$. This should be caled variable.3
    f. variable.4 contains the first 10 colors from the colors function. This should be caled variable.4.
2. Construct a single tibble that has ALL of the following properties. The construction of each should not be done by brute force (listing out elements).

    a. Name: example.tibble.2,
    b. Three variables with 78 observations each.
    c. variable.1 contains the letters of the lowercase alphabet each repeated 3 times consecutively.
    d. variable.2 contains numbers 1 - 78 in decreasing order.
    e. variable.3 contains the last 78 colors in the colors() function. Starting with the last in the list alphabetically.
3. How many rows and columns do these tibbles have? This should be done with a function, not by inspection.

    a. ChickWeight.tib
    b. Formaldehyde.tib
    c. Loblolly.tib
4. Extract the following from the indicated tibble: (This should not be done by inspection. The result should be a tibble.)

    a. Loblolly.tib: The age variable
    b. Formaldehyde.tib: The optical density variable
    c. ChickWeight.tib: The weight variable
5. Extract the following from the indicated tibble: (The result should be a vector.)

    a. Loblolly.tib: The age of all Loblolly pine trees with a height over 55 ft.
    b. Formaldehyde.tib: The optical density when the Carbohydrate is greater than 0.4 ml
    c. ChickWeight.tib: The weight of chicks at 4 days old.
6. Extract the following from the indicated tibble: (This should not be done by inspection. The result should be a tibble)

    d. Loblolly.tib: Rows with heights over 60 ft
    e. Formaldehyde.tib: Rows with carbs over 0.5 ml
    f. ChickWeight.tib: Rows at age 0 days.
7. Create a tibble **alpha.tib** where the first component **forwardAlpha** is a vector of all the lowercase letters in the English alphabet. The second component **backwardAlpha** should be the Uppercase letters of the English alphabet in reverse alphabetical order. The third component **bothAlpha** should be a function of the first two components where each element is a character string of length two consisting of the corresponding elements of the first two components.

Questions. Questions. Questions. I need more questions

Chapter 20: Functions

When coding, you may find that with minor modifications, blocks of code are being reused. This will be most apparent when a chunk of code is copied, pasted, and a few values are changed. Once this occurs, a function should be made out of this code. A **function** is a named set of commands. This has at least two advantages:

1. Code is easier to read. Repetitive sets of commands are replaced with a function call and its arguments.

2. Code is easier to correct. Without making a function, an error in a repeated chunk of code must be corrected in each replication. With a function, the correction only needs to be made once, where the function is written.

Example

In previous sections, many examples of functions have been encountered. Some of these include: $any(\quad), print(\quad), c(\quad)$.

With the exception of the last, these functions come built into the basic R installation. The last is added using the $dplyr$ package.

Even if we can not see them, each of these follows a set of commands, and outputs a result.

Basic Form

This section will deal with creating new functions[27]. When creating a new functions, it will have this basic form:

```
functionName <- function( arguments = defaults ){
      steps to be executed
      return( optional output to be returned )
}
```

It starts with *functionName*[28]. This is the name of the function. This name will be used whenever you want to execute the steps contained within the vector. The function name is followed by the assignment operator, and then the **function()** command[29]. This command indicates to R that a function is being built. Within the round parentheses, list any arguments for the function.

Arguments

Arguments are variables used to customize how a function works. Most, but not all, functions that are written will require some arguments. If needed, some of the arguments can be assigned default values. These default values will be used within the function unless otherwise specified. A default is made by setting the argument equal to the default value

$$\text{argument} = \text{default value}$$

Body of a Function

Following the round parentheses, that contain the arguments, will be a set of curly braces. Within the curly braces will be the code the function is meant to execute. The last line inside the curly braces is a $return(\quad)$ statement. The $return(\quad)$ statement will specifically indicate what the output will be for the function.

As an example, the function $close(\quad)$ is given below.

```
# Determines if two values x and y are within tol of each other.
close <- function(x, y, tol = 1e-04) {
    abs.diff <- abs(x - y)  #step 1
    less.than.tol <- abs.diff < tol  #step #2
    return(less.than.tol)
}
```

As the name, and comment, indicate, $close(\quad)$ determines if two values are close to each other.

Step #1 Determine the absolute value of the difference of x and y and assign this value to a variable $abs.diff$.

Step #2 Determine if $abs.diff$ is less than the variable tol. Assign the logical result to the variable $less.than.tol$. The last line returns the value for $less.than.tol$.

---

[27] New functions need to be created in the code BEFORE it is used in the code.
[28] Anonymous functions are functions where no *functionName* is assigned.
[29] **function()** can be abbreviated with **\()**

The description of the function leaves x, y and tol unspecified. These are the arguments. The x and y arguments will specify the values being compared. The tolerance tol specifies what it means for two values to be close.

In $close(\quad)$, tol's default is set to 0.0001. Meaning that unless tol is explicitly changed, 0.0001 will be used.

```
first.Vector <- 1.01
second.Vector <- 1.02

close(x = first.Vector, y = second.Vector)

[1] FALSE

close(x = first.Vector, y = second.Vector, tol = 0.1)

[1] TRUE
```

Calling a Function

When using a function, the arguments can be listed in several ways.

1. When using argument names, the arguments can be listed in any order, as long as they are separated by a comma.
2. When not using argument names, the values need to be entered in the same order as the argument.

These would run through the same computations:

```
close(first.Vector, second.Vector)
close(x = first.Vector, y = second.Vector, 0.1)
```

However, this would not.

```
close(first.Vector, 0.1, second.Vector)
```

In this case, the x argument takes values from $first.Vector$, the y argument is set to 0.1, and tol is set to $second.Vector$.

Multiple Outputs

Functions produce only a single result. The result can be any data type: a vector, a list, a dataframe, or nothing at all.

A modified $close(\quad)$ function, returns a dataframe containing the absolute difference, the tolerance, and the determination.

```
# Determines if two values x and y are within tol of each other.
close.2 <- function(x, y, tol = 1e-04) {
    abs.diff <- abs(x - y)  #step 1
    less.than.tol <- abs.diff < tol  #step #2
    output <- data.frame(difference = abs.diff, tolerance = tol, closeTF = less.than.tol)
    return(output)
}

close.2(first.Vector, second.Vector)

  difference tolerance closeTF
1       0.01     1e-04   FALSE
```

Functional Programming

R is a functional programming language. This means that a functions argument can be another function. For example, the function $fun(\quad)$ has two arguments fn and x.

```
fun <- function(fn, x) {
    computed <- fn(x)
    return(computed)
}
```

Examining the code, it can be seen that $fun(\quad)$ applies the function $fn(\quad)$ to the value x. The example below shows $fun(\quad)$ applying $mean(\quad)$ and $min(\quad)$ to the vector $c(2,1,3)$.

```
fun(fn = mean, x = c(2, 1, 3))
fun(fn = min, x = c(2, 1, 3))

[1] 2
[1] 1
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Make a function **squarer** that takes a single numerical argument x and returns the square of the value of x. Assume that only numerical values for x will be supplied.

2. Make a function **FtoC** that takes a single numerical argument $Fahr$ and returns the Celsius equivalent. Assume that only numerical values for $Fahr$ will be supplied.

3. Make a function **rootOf100** that takes a single numerical argument n and returns the n-th root of 100. Assume that only numerical values for n will be supplied. The default for n should be one.

4. Make a function **CtoF** that takes a single numerical argument $Celc$ and returns the Celsius equivalent. Assume that only numerical values for $Celc$ will be supplied. The default for $Celc$ should be 0.

5. Make a function **Bigger.Logical** that takes two numerical arguments x and y. Assume that only numerical values for x and y will be supplied. It should return a logical value of TRUE if x is larger then y. It should return a FALSE otherwise.

6. Make a function **Bigger.Numeric** that takes two numerical arguments x and y. Assume that only numerical values for x and y will be supplied. It should return a numerical value of 1 if x is larger then y. It should return a zero otherwise.

7. Make a function **Smaller** that takes two numerical arguments x and y. Assume that only numerical values for x and y will be supplied. It should return the smaller of the values x and y.

8. Make a function **totalOfTen** that a single numerical argument x. Assume that only numerical values of length ten will be supplied. **totalOfTen** should return a vector with a single element in it. The value of that single element should be the total of all ten values in x.The default value for x should be a vector of ten zeros.

Questions. Questions. Questions. I need more questions

Chapter 21: Controls - Decisions

If

An if statement takes the form

```
if( condition ){
  Steps to execute IF the condition is TRUE
}
```

The statement will check a condition. The condition, contained in round parentheses, should evaluate to a single logical value. If that logical value is $TRUE$, the code inside the curly braces will be executed. If that logical value is $FALSE$, the code inside the curly braces will be skipped.

Looking over these two snippets of code, both if statements determine if all values in a given vector are large than zero. If they are, R will print out a message. Otherwise, R will do nothing.

```
numbers = c(1, 8)

if (all(numbers > 0)) {
    print("The numbers are all positive.")
}
```

```
[1] "The numbers are all positive."
```

```
numbers = c(-1, 1)

if (all(numbers > 0)) {
    print("The numbers are all positive.")
}
```

As can be seen, only the first snippet produces a result.

If...Else...

An $if \ldots else \ldots$ statement expands on the if statement. It allows alternatives outcomes to be produced. An $if \ldots else \ldots$ strings along a series of if statements. The first time R finds an if statement with a $TRUE$ condition, it will execute the code in the accompanying curly braces. Once R executes that particular set of commands, it will not check anymore conditions in the $if \ldots else \ldots$, and will skip paste any more accompanying code.

```
numbers = c(-1, -8)

if (all(numbers > 0)) {
    print("The numbers are all positive.")
} else if (all(numbers < 0)) {
    print("The numbers are all negative.")
} else if (all(numbers < 0)) {
    # An intentional mistake.
    print("The numbers are all zero.")
} else {
    print("The numbers are a mess.")
}
```

```
[1] "The numbers are all negative."
```

In this snippet, R determines if all the numbers are negative. Since the first condition fails, R checks the second. It is TRUE, therefore R prints "The numbers are all negative." At this point R does not check any further conditions. This can be seen by looking at the third condition. It is also a $TRUE$ statement, but the accompanying statement is not printed.

For both an if statement, and an $if \ldots else \ldots$, multiple lines of code can be included for each condition. A single line was used in the example for brevity.

ifelse

The $ifelse(\ \ )$ function is a vectorized version of a single $if\ldots else\ldots$ statement. The syntax for $ifelse(\ \ )$ is

$$ifelse(logicalcondition,\text{result.T, result.F}).$$

$ifelse(\ \ )$ will evaluate the condition, and return $result.T$ if the condition is $TRUE$, otherwise it will return $result.F$.

```
x <- c(-1, 0, 3)
ifelse(x > 0, "Positive", "Not Positive")

[1] "Not Positive" "Not Positive" "Positive"
```

Several conditions can be evaluated using $ifelse(\ \ )$ by nesting an $ifelse(\ \ )$ inside of itself.

$$ifelse\big(condition1, result.T1, ifelse(condition2, result.T2, result.F)\big)$$

```
x <- c(-1, 0, 3)
ifelse(x > 0, "Positive", ifelse(x < 0, "Negative", "Zero"))

[1] "Negative" "Zero"     "Positive"
```

However, this can be very hard to read. Readability will also be very difficult if the results are several lines of code.

switch

$switch(\ \ )$ can be used to select from multiple results. The basic form is

$$switch(EXPR, result1, result2, \ldots).$$

$switch(\ \ )$ uses its first argument $EXPR$ to select the result from the successive arguments. The results are listed in one of two distinct ways depending whether the first argument evaluates to a character string or a number.

In this first example, $EXPR$ evaluates to a number. The result is determined by counting through the listed results.

```
x <- 2
switch(EXPR = x, "a", "b", "c", 100)  #The second result is 'b'.

[1] "b"

x <- 4
switch(EXPR = x, "a", "b", "c", 100)  #The fourth result is 100.

[1] 100
```

Take note that the results are a list of values, but they are not collected into a $list(\ \ )$ or a vector.

In this second example, $EXPR$ evaluates to a string. For this to evaluate correctly, each result, with one exception, needs to be given a name. R will select the result whose name matches $EXPR$.

```
x <- "b"
switch(EXPR = x, a = 1, b = 2, c = "c", 100)  # 'b' matches with the result labeled b.

[1] 2

x <- "e"
switch(EXPR = x, a = 1, b = 2, c = "c", 100)  #EXPR = 'e' has no match. The unnamed result is used..

[1] 100
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Write an if-else statement that will print the string, "Controls001 has a number eight", if the vector Controls001 contains the number eight, otherwise it will print the string "Controls001 does not have a number eight"

2. Write an if-else statement that will print the string, "Controls001 has a negative number", if the vector Controls001 contains a negative number, otherwise it will print the string "Controls001 does not have a negative number"

3. Write an if-else statement that will print the string:

    a. "Controls001 has all positive numbers", if the vector Controls001 contains all positive numbers,
    b. "Controls001 has all negative numbers", if the vector Controls001 contains all negative numbers numbers,
    c. "Controls001 has all zeros", if the vector Controls001 contains all zeros,
    d. "Controls001 is a mess" for anything else

4. Write an if-else statement that will print the string:

    a. "The first element of Controls002 is the smallest", if this is true
    b. "The second element of Controls002 is the smallest", if this is true
    c. "The third element of Controls002 is the smallest", if this is true
    d. "The fourth element of Controls002 is the smallest", if this is true

5. Create a function **Bigger** that takes two arguments: x and y. It should return the larger of the two values. If they are the same, it should return x. Assume that x and y will only have vectors of length one passed to each of them. This should make use of an if/else statement.

6. Create a function **BiggerSquaredOrHalf** that takes two arguments: x and y. It should return the the square of x if x is greater. Otherwise, it should return one-half of y Assume that x and y will only have vectors of length one passed to each of them. This should make use of an if/else statement.

7. Add two variables to the **LifeCycleSavingsUpdate** dataframe: **age.ratio** and **broken.up**

    a. **age.ratio** should be the ratio of the **pop15** variable to the **pop75** variable.
    b. **broken.up** should take one of three values: "UP", "DOWN", or "OVER". Use "DOWN" if **age.ratio** is less than 10 and **ddpi** is less than 3. Use "UP" if **age.ratio** is greater than 20 and **ddpi** is greater than 4. For everything else, use "OVER".

8. Using the **oldFaithful** dataframe, print out one of the following strings:

    a. "Old faithful has XXX more eruptions that last at least four minutes than eruptions that last less than three minutes."
    b. "Old faithful has XXX more eruptions that last less than four minutes than eruptions that last at least than three minutes."
    c. "Old faithful has the same number of eruptions that last less than four minutes as eruptions that last at least than three minutes. There are XXX each."
    d. XXX should be replaced by the positive difference in the number of eruptions that last at least four minutes versus those that last less than four minutes. If they are the same, XXX should be replaced by zero.

9. Make a function $tempChange(\quad)$ that meets the following requirements:

    a. It takes two arguments: $givenTemp$ and $to.C$.
    b. $givenTemp$ should have no default. $to.C$ should default to $TRUE$.

c. If $to.C$ is $TRUE$, then $tempChange(\ )$ should convert $givenTemp$ from a Fahrenheit temperature to its equivalent temperature in Celcius.

d. If $to.C$ is $FALSE$, then $tempChange(\ )$ should convert $givenTemp$ from a Celcius temperature to its equivalent temperature in Fahrenheit.

e. $tempChange(\ )$ should return a list with three named components: $givenTemp$, $to.C$, and $convertedTemp$. $convertedTemp$ is the converted temperature. Demonstrate the usage of your function by applying it to the **changeTheseTempsVector** using $to.C$ set to both options.

Questions. Questions. Questions. I need more questions

Chapter 22: Controls - Loops

A **loop** is a piece of code that will be repeat the commands contained inside of it. Writing a loop can make your code shorter, since the code is not repeated, and it will make it easier to read. Situations where a loop will be useful can be broken down into two cases:

1. The number of repetitions/iterations can be determined before the looping process begins.

2. The number of repetitions/iterations can not be determined until after the loop begins.

(In reality, both cases can be handled by either of the functions that will be outlined. But, it is easier to think of them as separate cases)

for() Loop

A $for(\quad)$ loop is a looping method used when the number of repetitions/iteration can be determined before the looping procedure begins. As an example, a $for(\quad)$ loop will be constructed to print each element in a given vector. It will be shown once without using a $for(\quad)$ loop, and once with.

```
print(letters[1])
print(letters[2])
print(letters[3])
print(letters[4])
print(letters[5])

for (index in 1:5) {
    print(letters[index])
}

[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
```

Both pieces of code produce the same output: the first five characters in $letters$. The second is much shorter. In fact, if the goal was to print all 26 characters in $letters$, the first method would require 26 lines of code. The second method would need to change the 5 to a 26.

The form of a $for(\quad)$ loop,

```
givenVector <- c(element1, element2, ...,  element.Final)

# The for() loop begins here.
for(eachElement in givenVector){
    # perform the given steps
}
```

makes it relatively straight forward to conceptualize the process.

> For each element in a given vector, complete the given steps.

The syntax can be hard to conceptualize. It begins with the $for(\quad)$ command. Inside the round parentheses are three objects, in order:

1. an indexing variable
2. the word ‘in’
3. a given vector of any type.

The word ‘in’, in a certain sense, acts as an assignment operator. For each pass through the loop, it assigns the next value in the vector to the indexing variable. Since the given vector has a finite number of elements, the loop will only run a finite number of times.

Indexing Variable

The indexing variable does not need to be used explicitly in the steps contained in a $for(\quad)$ loop, but it can be. When it is not used, it acts to count the repetitions. When it is used, in addition to counting the repetitions, it also customizes the code for each repetition.

Skipping Iterations - next/break

Under certain conditions, skipping a single repetition, or all remaining repetitions, will be the proper course of action. To do so, the body of the loop should include an $if(\ )$ statement whose condition, when $TRUE$, tells R, to advance the indexing variable to the next value, or leave the loop altogether. The command following the $if(\ )$ statement will be $next$, or $break$, respectively.

```
# The for() loop begins here.
for(eachElement in givenVector){
  if( skipping.condition ){
    next
  }
    # perform the given steps, if the skipping condition is FALSE
}
```

while() Loop

A $while(\ )$ loop is a looping method used when the number of repetitions can't be determined before the looping procedure begins. The basic form is as follows:

```
while( given.Condition == TRUE){
    # perform the given steps
    # update check.Value
}
```

A $while(\ )$ loop will execute a given set of commands as long as a given condition is $TRUE$. Once the given condition becomes $FALSE$, R will skip past the code in the loop and continue with any subsequent code.

As an example, the $sample(\ )$ function will select a random value from 1 to 100. A $while(\ )$ loop will print a sentence about the random value, and then select a new one. As long as the random value is less than 90, R will continue to print and select new values. Since each successive number is unknown, it can not be determined ahead of time how many values will be printed.

```
randomValue <- sample(x = 1:100, size = 1)
while (randomValue < 90) {
    sentence <- paste("The current random value is ", randomValue, ".", sep = "")
    print(sentence)
    randomValue <- sample(x = 1:100, size = 1)
}

[1] "The current random value is 42."
[1] "The current random value is 83."
[1] "The current random value is 31."

randomValue

[1] 92
```

The last value that is printed is a 92. But, it is not part of sentence as the other results are. This is because once the 92 was selected, the condition was no longer $TRUE$. When the condition is $FALSE$, R dropped out of the loop, and executed the command that displayed just the number 92.

Care should be taken when using a $while(\ )$ loop. If it is not written properly, it is possible for the loop to continue infinitely. This code, if run, will never stop. Replacing $randomValue < 90$ with $randomValue < 1000$ will give a condition that will never be FALSE. Theoretically, a loop with the second condition will run forever.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Use a $for(\quad)$ loop to add together the values in the vector Loops001.

2. Use a $for(\quad)$ loop to add together the values in the vector Loops001. The result should be a vector with the same length as Loops001. The i^{th} value of the result should be the total of the first i values in Loops001.

3. Using a $for(\quad)$ loop, create a vector with 40 elements. The first element should be -4. The second should be 3. Subsequent elements should be the total of the previous two elements.

4. Using a $for(\quad)$ loop, create a vector with 40 elements. The first three elements should all be the number one. Subsequent elements should be the total of the previous three elements.

5. Using a $for(\quad)$ loop, create a vector with 40 elements. The first element should the number one. Subsequent elements should be the negative of two times the previous.

6. Use a $for(\quad)$ loop, LoopsMascots, LoopsPunc, and LoopsPhrase to recreate these vectors of length one.

```
[1] "Once upon a time, my mascot was a Pink Turtle.  I was five."
[1] "Once upon a time, my mascot was a Blue Devil! mehh."
[1] "Once upon a time, my mascot was a Tiger. ROAR!!!"
[1] "Once upon a time, my mascot was a Golden Eagle. Wow, I can fly."
[1] "Once upon a time, my mascot was a Fuzzy Orange? Yup, a Fuzzy Orange. "
[1] "Once upon a time, my mascot was a Big Red Bear. Really, there is no mascot."

[1] "Once upon a time, my mascot was a Pink Turtle.  I was five."                "Once upon a time, my mascot was a Blue Devil
! mehh."                         "Once upon a time, my mascot was a Tiger. ROAR!!!"
[4] "Once upon a time, my mascot was a Golden Eagle. Wow, I can fly."            "Once upon a time, my mascot was a Fuzzy Oran
ge? Yup, a Fuzzy Orange. "       "Once upon a time, my mascot was a Big Red Bear. Really, there is no mascot."
```

7. Create a function called **FibonMatrix()**. **FibonMatrix()** will take five arguments: **rowNumber**, **colNumber**, **firstNumber**, **secondNumber**, and logical value **acrossRow**. **FibonMatrix()** should create a matrix with **rowNumber** rows and **colNumber** columns. The values of the matrix should be filled across the rows first if **acrossRow** is set to TRUE. Otherwise, the matrix should be filled down the columns first.(Similar to using the **matrix()** function.) The first value to fill in the matrix will be first **firstNumber**. The second value to fill in the matrix will be **secondNumber**. Subsequent values that are filled in should be the sum of the previous two. The matrix should be returned. Demonstrate that your function works by creating a 5 x 7 matrix where the first two values filled in down the first column are four and then three.

8. Create a function called **sqeezeIt**. **sqeezeIt()** will take two arguments: **startValue** and **tolerance**. **sqeezeIt** will determine the lowest power of **startValue** that is less than or equal to **tolerance** in absolute value. If **startValue** has an absolute value of one or larger and **startValue** is less than or equal to **tolerance**, **sqeezeIt** should return 1. If **startValue** has an absolute value of one or larger and **startValue** is greater than or equal to **tolerance** in absolute value, **sqeezeIt** should return NA. Otherwise, **sqeezeIt** should return the appropriate power. Demonstrate your function by determining the lowest power of -.99 whose absolute values is less than 0.01. Make use of a while loop.

Questions. Questions. Questions. I need more questions

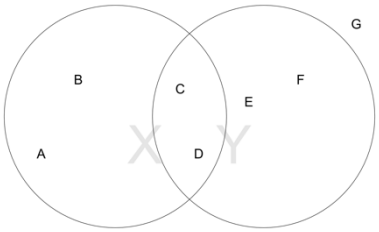
Chapter 23: Functions - Set Operations

A vector is a collection of elements. Usually, the order of the elements in the vector matters. We can state what the first, second, and third (and so on) elements are. However, in some instances, we are only interested in knowing what elements are included in a vector, or a collection of vectors. In this case, we the vectors can be thought of as sets.

In our example, we will define three character vectors.

```
x <- c("A", "B", "C", "D", "A", "A", "A")
y <- c("C", "D", "E", "F", "F", "E")
```

Here we have a visual illustration of these vectors when thought of as sets.

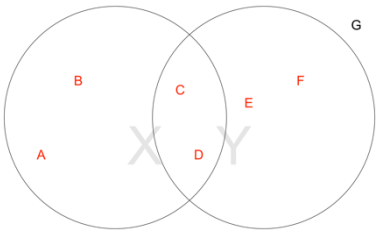


Union

The union of two sets is all elements that occur in at least one of the two sets. Looking at the union of vector x and y, the union is all elements except for "G".

```
union(x, y)  #elements in at least one of x or y

[1] "A" "B" "C" "D" "E" "F"
```



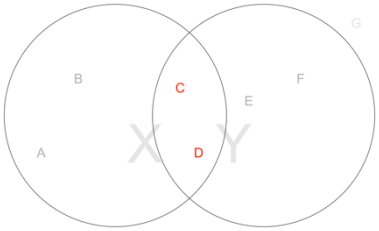
Notice that the union of these two sets produces a vector that contains all the elements of vector x listed first. That is, with the exception of the final three "A"s in vector x. Then, elements from vector y are listed in the order that they appear in y, provided they are not already listed.

Intersection

The intersection of two sets is all elements that occur in both two sets. Looking at the intersection of vector x and y, the intersection is "C" and "D".

```
intersect(x, y)  #elements in both x and y

[1] "C" "D"
```



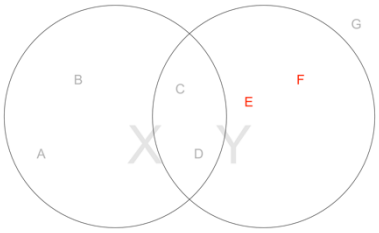
It will be the case that the intersection of two sets will always be included in the union.

Set Difference

The set difference is like subtraction for sets. Take the first set, and remove from it all elements that are in the second. The remaining elements are the set difference. As with subtraction, the order of the subtraction will matter. In this case, removing all element of x that fall into y leaves a set difference of "E" and "F".

```
setdiff(y, x)  #elements in y but not in x

[1] "E" "F"
```



Switching the order around, the set difference would be "A" and "B", if y was subtracted from x.

Set Equality

Set equal can be tricky. It checks for the existence of each element of one vector in the other, and vice versa. It is not checking that there are the same number of each, or that they appear in the same order. It only checks for existence.

The elements in x and y are not the same.

```
setequal(x, y)

[1] FALSE
```

Here, a new vector z is created. It is shorter the x but contains the same elements. Therefore, as sets, the two vectors are equal.

```
z <- c("A", "D", "B", "C")

setequal(x, z)

[1] TRUE
```

Recall, to determine if two vectors have the same elements, in the same order, use the identical function.

```
identical(x, z)

[1] FALSE
```

Is this %in% in the set?

The $%in%% operate determines if elements of one vector are *in* another. It will return a logical value for each element in vector listed on the right hand side of %*in*%.

```
"A" %in% x  #is this element in this vector

[1] TRUE

"A" %in% y

[1] FALSE

c("A", "E", "A") %in% x

[1]  TRUE FALSE  TRUE
```

Along with a few other commands, the %*in*% operator can be used in a similar manner as *setequal*( ).

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Determine the union of the values in **Set001** and **Set002**.
2. Determine the union of the values in **Set003** and **Set004**.
3. Determine the union of the values in **Set001** and **Set003**.
4. Determine the intersection of the values in **Set001** and **Set002**.
5. Determine the intersection of the values in **Set003** and **Set004**.
6. Determine the intersection of the values in **Set001** and **Set003**.
7. Determine the which values are in **Set001** but not in **Set002**.
8. Determine the which values are in **Set004** but not in **Set003**.
9. Is “A” an element of **Set001**? Indicate TRUE if this is the case. Otherwise FALSE.
10. Are “A” or “B” elements of **Set001**? Indicate TRUE if this is the case. Otherwise FALSE.
11. Is 1 an element of **Set003**? Indicate TRUE if this is the case. Otherwise FALSE.
12. Are 1 or 2 elements of **Set004**? Indicate TRUE if this is the case. Otherwise FALSE.

Questions. Questions. Questions. I need more questions

Chapter 24: Functions - Mathematical Operations

Sums & Products

: The $sum(\quad)$ function totals the values given to it. The basic structure for using the function is

$$sum(x, na.rm = FALSE)$$

where x is a vector of numeric or logical values to be summed. The second argument $na.rm =$ determines whether missing values should be removed or not. By default, they are included in the computation. When missing values are included, the result will be a missing value. If there are no missing values, the argument can be ignored.

```
x <- c(1, 5, 3, 2, NA, 4, NA)
sum(x, na.rm = FALSE)

[1] NA

sum(x, na.rm = TRUE)

[1] 15
```

The sum over multiple vectors can be taken. To do so, list all desired vectors in the $sum(\quad)$ function, separating them with commas.

```
y <- c(0, 1)
sum(x, y, na.rm = TRUE)

[1] 16
```

Logical values can also be summed. Numerically, $TRUE$ equals 1 and $FALSE$ equals 0. If $TRUE$ represents the occurrence of an event, summing a logical vector determines the total number of occurrences.

```
logical.x <- c(TRUE, TRUE, FALSE, TRUE)
sum(logical.x)

[1] 3
```

Cumulative Sums
The $cumsum(\quad)$ function computes the cumulative sums of a vector. $cumsum(\quad)$ returns a vector with the same length as the original vector. The first output element is the first element of the original vector. The second element outputted is the sum of the first two elements of the original vector. The third element outputted is the sum of the first three elements of the original vector. This repeats for each element of the original vector. $cumsum(\quad)$ does not have an argument that deals with missing values. Once a missing value appears, the remaining cumulative sums are missing.

```
cumsum(x)

[1]  1  6  9 11 NA NA NA
```

Differences:
The $diff(\quad)$ function takes different of vector elements whose indices are a fixed distance apart. This distance is called **lag**.

```
x <- c(1, 22, 333, 44, 5, 66, NA)
diff(x, lag = 2)

[1]  332   22 -328   22   NA
```

With a lag of 2, the difference is taken of the the first and third elements (lag = 3 - 1 = 2), the second and fourth elements (lag = 4-2 = 2), and so on. The vector of differences will be shorter than the one original vector.

Products:
The $prod(\quad)$ function multiplies the values given to it. The basic structure for using the function is

$$prod(x, na.rm = FALSE)$$

where x is the vector of numeric or logical values to be summed. The second argument $na.rm =$ determines whether missing values should be removed or not. By default, they are included in the computation. When missing values are included, the result will be a missing value. If there are no missing values, the argument can be ignored.

```
x <- c(1, 5, 3, 2, NA, 4, NA)
prod(x, na.rm = FALSE)

[1] NA

prod(x, na.rm = TRUE)

[1] 120
```

The product can be taken over more than one vector. To do so, list all desired vectors in the $prod(\quad)$ function, separating them with commas.

```
y <- c(0, 1)
prod(x, y, na.rm = TRUE)

[1] 0
```

As with addition, Logical values can also be multiplied. If $TRUE$ represents the occurrence of an event, multiplying all elements of a logical vector determines whether the event always occurred or if it failed to occur at least once.

```
logical.x <- c(TRUE, TRUE, FALSE, TRUE)
prod(logical.x)

[1] 0
```

Cumulative Products:
The $cumprod(\quad)$ function computes the cumulative products of a vector. $cumprod(\quad)$ returns a vector with the same length as the original vector. The first output element is the first element of the original vector. The second output element is the product of the first two elements of the original vector. The third output element is the product of the first three elements of the original vector. This repeats for each element of the original vector. $cumprod(\quad)$ does not have an argument that deals with missing values. Once a missing value appears, the remaining cumulative sums are missing.

```
cumprod(x)

[1]  1  5 15 30 NA NA NA
```

Factorials:
The $factorial(\quad)$ function takes the product of the first n positive integers.

```
factorial(5)

[1] 120

prod(1:5)

[1] 120
```

Binomial Coefficient:
Built from the factorials is the $choose(\quad)$ function. It computes binomial coefficients, namely

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

where n, m are both non-negative integers with n greater than or equal to m.

```
choose(5, 2)

[1] 10

numerator <- factorial(5)
denominator <- factorial(2) * factorial(5 - 2)
numerator/denominator

[1] 10
```

Maximums

The $max(\quad)$ function finds the maximum value within the vector given to it. The basic structure for using the function is

$$max(xna.rm = FALSE)$$

where x is a vector of numeric or logical values. The second argument $na.rm =$ determines whether missing values should be removed or not. By default, they are included in the computation. When missing values are included, the result will be a missing value. If there are no missing values, the argument can be ignored.

```
x <- c(30, NA, 12, 100, 43, 67, 100)
max(x, na.rm = FALSE)

[1] NA

max(x, na.rm = TRUE)

[1] 100
```

The maximum can be taken over more than one vector. To do so, list all desired vectors in the $max(\quad)$ function, separating them with commas.

```
y <- c(0, 1000, 15, 20, 11, 0, 0)
max(x, y, na.rm = TRUE)

[1] 1000
```

The maximum of a logical vector can be found. R coerces(changes) $TRUE$ to a value of 1, and $FALSE$ as a 0. If $TRUE$ represents the occurrence of an event. If the maximum of a vector equals 1, this implys that the event occurred at least once.

```
logical.x <- c(TRUE, TRUE, FALSE, TRUE)
max(logical.x)

[1] 1
```

Parallel Maximum :

The $pmax(\quad)$ function is used to determine parallel maximum over a collection of vectors. $pmax(\quad)$ will return a vector that contains the maximum of the first elements, the maximum of the second elements, and so on. The basic structure for the pairwise maximum is

$$pmax(\ldots, na.rm = FALSE)$$

[30] where .. are the given vectors separted by a comma. The argument $na.rm =$ determines whether missing values should be ignored or not. By default, they are not ignored in the computation. When missing values are included, the result will be a missing value. If there are no missing values, the argument can be ignored.

```
# This example only uses
x <- c(30, NA, 12, 100, 43, 67, 100)
y <- c(0, 1000, NA, 20, 11, 0, 12)
z <- 17
pmax(x, y, z, na.rm = FALSE)

[1]  30  NA  NA 100  43  67 100

pmax(x, y, z, na.rm = TRUE)

[1]   30 1000   17  100   43   67  100
```

Notice that when $na.rm = TRUE$, the missing value is ignored. If the vectors represents by .. do not have the same length, the shorter will be recycled. If more than two vectors are used, list them, separating them with commas. This will return the parallel maximum.

[30] The 'p' in $pmax(\quad)$ actually stands for parallel.

Index of Maximum:
The $which.max(\quad)$ determines the index of the *first* occurrence of the maximum of a vector.

```
which.max(x)

[1] 4
```

Minimums:
The $min(\quad)$ function finds the minimum value within the vector given to it. The basic structure for using the function is

$$min(xna.rm = FALSE)$$

where x is a vector of numeric or logical values.

```
x <- c(30, NA, 12, 100, 43, 67, 100)
min(x, na.rm = FALSE)

[1] NA

min(x, na.rm = TRUE)

[1] 12
```

The minimum can be taken over more than one vector. To do so, list all desired vectors in the $min(\quad)$ function, separating them with commas.

```
y <- c(0, 1000, 15, 20, 11, 0, 0)
min(x, y, na.rm = TRUE)

[1] 0
```

The minimum of a logical vector can be found. Using the usual interpretation of $TRUE$ and $FALSE$, if the minimum of a vector equals 0, this implies that the event never occurred.

```
logical.x <- c(TRUE, TRUE, FALSE, TRUE)
min(logical.x)

[1] 0
```

Parralel Minimum: The $pmin(\quad)$ function is used to determine pairwise minimum over a pair of vectors. $pmin(\quad)$ determines the corresponding minimums. The basic structure for the pairwise maximum is

$$pmax(x, y, na.rm = FALSE)$$

where x, y are the given vector of numeric or logical values. The third argument $na.rm =$ determines whether missing values should be ignored or not. By default, they are not ignored in the computation.

```
x <- c(30, NA, 12, 100, 43, 67, 100)
y <- c(0, 1000, 15, 20, 11, 0, 0)
pmin(x, y, na.rm = FALSE)

[1]  0 NA 12 20 11  0  0

pmin(x, y, na.rm = TRUE)

[1]    0 1000   12   20   11    0    0
```

Index of Minimum: The $which.min(\quad)$ determines the index of the *first* occurrence of the minimum of a vector.

```
which.max(x)

[1] 4
```

Rounding

R has several method for rounding output. Most of the methods given here change the computed value as opposed to just changing the displayed value.

The $round(\ \ )$ function is the basic rounding function. Beyond the given vector x of values to be rounded, its other argument indicates the number of digits. When rounding a number that ends in a 5, it may not round exactly as you expect. The operating system of the computer and the exact representation of the number can affect the result. Depending on the pairing, $round(0.15, digits = 1)$ could be 0.1 or 0.2.

```
x <- c(-1.24, -2.75, 1.24, 2.54, 0.15)
round(x, digits = 1)

[1] -1.2 -2.8  1.2  2.5  0.1
```

The $floor(\ \ )$ and $ceiling(\ \ )$ functions round to an integer value. $floor(\ \ )$ rounds to the largest integer less than the given value (towards negative infinity). $ceiling(\ \ )$ rounds to the smallest integer greater than the given value (towards positive infinity).

```
floor(x)

[1] -2 -3  1  2  0

ceiling(x)

[1] -1 -2  2  3  1
```

The $trunc(\ \ )$ function returns an integer by dropping all decimal places.

```
trunc(x)

[1] -1 -2  1  2  0
```

Ranking & Sorting

Sorting:

Given a vector whose elements can be ordered, the $sort(\ \ )$ function can be used to rearrange the vector from 'smallest' to 'largest', or the other way around. If the results need to be used subsequently, the result needs to be assigned to a variable.

```
x <- c(30, 20, 50, NA, 10, 40)
sort(x, decreasing = FALSE, na.last = NA)

[1] 10 20 30 40 50
```

$decreasing$ is a logical argument. When set to $FALSE$, the values are rearranged in increasing order. $na.last$ determines what becomes of NA values. When set to NA, they are removed. Resulting in a shorter vector than one started with. When set to $TRUE$ they are put at the end of the returned vector. When set to $FALSE$, they go to the beginning.

Ordering by Index:

The $order(\ \ )$ function does not sort a vector. However, its result can be used to sort a given vector. $order(\ \ )$ returns a rearrangement of the indexes (a permutation) of the given vector x that can be used to sort the given vector.

Suppose the first value in the given vector x is the fifth smallest value in x. The result from $order(\ \ )$ will have a 1 in the fifth position. Next, suppose the second value in x is the smallest value in x. $order$ will place a 2 in the first position of its result.

```
order(x)

[1] 5 2 1 6 3 4

x[order(x)]

[1] 10 20 30 40 50 NA
```

This is useful when it is desired to sort a dataframe based upon one variable, or a few variables. In this first case, the result of *order*( ) can be used to rearrange the rows in the dataframe according to the values in *dat*$*V*2 alone.

```
dat <- data.frame(V1 = LETTERS[1:6], V2 = c(7, 1, 6, 4, 1, 7), V3 = c(F, T, T, T, F, T))
dat

  V1 V2    V3
1  A  7 FALSE
2  B  1  TRUE
3  C  6  TRUE
4  D  4  TRUE
5  E  1 FALSE
6  F  7  TRUE

order.1.var <- order(dat$V2)
order.1.var

[1] 2 5 4 3 1 6

dat

  V1 V2    V3
1  A  7 FALSE
2  B  1  TRUE
3  C  6  TRUE
4  D  4  TRUE
5  E  1 FALSE
6  F  7  TRUE

LETTERS[1:6]

[1] "A" "B" "C" "D" "E" "F"

V2 = c(7, 1, 6, 4, 1, 7)
```

Notice that the tied rows are taken in the order they appear in the dataframe. However, if a second vector datV3 $is added to *order*( ), it is used to adjust the output so that the tied rows have an order. Additional vectors can be added if needed. The *order*( ) function has an *na.last* and *decreasing*.

```
order.2.var <- order(dat$V2, dat$V3)
order.2.var

[1] 5 2 4 3 1 6

dat[order.2.var, ]

  V1 V2    V3
5  E  1 FALSE
2  B  1  TRUE
4  D  4  TRUE
3  C  6  TRUE
1  A  7 FALSE
6  F  7  TRUE
```

Ranking:
The result of the *rank*( ) function can't be used to sort a vector, or the rows of a dataframe. Instead, it provides the position of each value in each given value, if the values were sorted. In other words, it tells you the placement of each value in an ordered list.

```
x <- c(30, 20, 50, NA, 10, 40, 40)
sort(x)

[1] 10 20 30 40 40 50

rank(x, na.last = TRUE, ties.method = "average")

[1] 3.0 2.0 6.0 7.0 1.0 4.5 4.5
```

The argument *ties.method* determines how to rank ties. The default is to average them. In this example, forty is the forth and fifth value in the ordered list. This averages to 4.5. Other options are "first", "last", "random", "max", and "min". The first two put tied ranks into increasing or decreasing order, respectively. "max" uses the maximum rank for each set of tied values. While "min" uses the minimum.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. If possible, determine the total of the values in the vector **Math001**.

2. If possible, determine the total of the non-missing values in the vector **Math002**.

3. Create a vector with the same number of elements as **Math002**. Its i^{th} element should be the total of the first i values of **Math002**.

4. Create a vector from **Math002**. Its i^{th} element should be the total of the first i non-missing values of **Math002**.

5. Create a vector from **Math001**. Its values should correspond to the difference of consecutive values from **Math001$

6. If possible, determine the product of the values in the vector **Math003**.

7. If possible, determine the total of the non-missing values in the vector **Math002**.

8. Determine the value of a binomial coefficient where $m = 8$ and $n = 3$

9. Determine the value of the binomial coefficients where $n = 20$ and $m = 0,1,2, \ldots ,20$

10. If possible, determine the maximum value within the vector $Math001$.

11. If possible, determine the maximum non-missing value among all values within the vectors $Math001$ and $Math002$.

12. If possible, determine the maximum of all binomial coefficients where $n <= 1000$ and $m = 3$.

13. For each possible $i = 1,2,3, \ldots ,1000$, determine the minimum of the i^{th} values taken the vectors **Math001, Math002, Math003, Math004** . Display only the first 20 results.

14. Consider the minimum i^{th} values taken the vectors **Math001, Math002, Math003, Math004**. How many of these values are non-negative? Ignore missing values.

15. Reorder the rows of the dataframe **Math005**. They should be ordered based on the values of the variable **comp2**. Store the result as **MathOrdered**. Display the first 10 rows of the result.

Questions. Questions. Questions. I need more questions

Chapter 25: Functions - Probability Distributions

R contains several functions pertaining to probability distributions for many types on random variables X. These functions relate to discrete random variables with finite and infinite sample spaces, and continuous random variables. Most of the functions discussed here come in sets of four that follow a specific naming pattern. each function in a set is starts with either a d,p,q, or r, followed by an abbreviation for the distributions name. (xxx is a placeholder for the abbreviation.) Each function has arguments that are specific to the distribution, and arguments that are common.

a. **dxxx** - density or mass function with common argument (x). This function determines the value of a density, or mass, function at the values$x$.

b. **pxxx** - distribution function with common arguments $(q, lower.tail, lop.p)$. This functions determines cumulative probability $P(X \leq q)$ at q when the $lower.tail$ is set to $TRUE$. When, the $lower.tail$ argument is set to $FALSE$, the complementary probabilities $P(X > q)$ at q are determined. $log.p$ controls whether a probability is returned $(FALSE)$ or its log is returned $(TRUE)$.

c. **qxxx** - quantile function with common arguments $(p, lower.tail, lop.p)$. For each value in p, this function returns a quantile q such that the cumulative probability at p is q when $lower.tail$ is set to $TRUE$. In other words, this functions solves $P(X \leq q) = p$ for q. When $lower.tail$ is set to $FALSE$, the complement of the cumulative probability is used. The equation $P(X > q) = p$ is solved for q. $log.p$ indicates whether a probability p is given $(FALSE)$ or if its log is given $(TRUE)$.

d. **rxxx** - random number generator with common argument (n). The random number generator produces a sample of n observations from the indicated probability distribution.

Regardless of the distribution, each type of function has some common elements. The first element q for the distribution function is a vector of quantiles.

Continuous Distributions

Several different continuous distributions have functions in *R*.

For the **normal** distribution, the functional names are $dnorm(\ \ )$, $pnorm(\ \ )$, $qnorm(\ \ )$, and $rnorm(\ \ )$. The parameters for a normal distribution are indicated by the $mean$ and sd (standard deviation) arguments. The default is a standard normal distribution.

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

For the **chi-square** distribution, the functional names are $dchisq(\ \ )$, $pchisq(\ \ )$, $qchisq(\ \ )$, and $rchisq(\ \ )$. The parameters are indicated by the df (degrees of freedom) and ncp (non-centrality parameter) arguments.ncp defaults to 0.

```
dchisq(x, df, ncp = 0, log = FALSE)
pchisq(q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisq(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rchisq(n, df, ncp = 0)
```

For the **Student's t** distribution, the functional names are $dt(\ \ )$, $pt(\ \ )$, $qt(\ \ )$, and $rt(\ \ )$. The parameters are indicated by the df (degrees of freedom) and ncp (non-centrality parameter) arguments. The ncp argument was omitted to indicate that the centralized t distribution is used.

```
dt(x, df, log = FALSE)
pt(q, df, lower.tail = TRUE, log.p = FALSE)
qt(p, df, lower.tail = TRUE, log.p = FALSE)
rt(n, df)
```

For the **uniform** distribution, the functional names are $dunif(\ \ )$, $punif(\ \ )$, $qunif(\ \ )$, and $runif(\ \ )$. The parameters are indicated by the min (minimum) and max (maximum) arguments. As defaults, these are set to zero and one, respectively.

```
dunif(x, min = 0, max = 1, log = FALSE)
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
runif(n, min = 0, max = 1)
```

Discrete Distributions

For the **binomial** distribution, the functional names are $dbinom(\ \ )$, $pbinom(\ \ )$, $qbinom(\ \ )$, and $rbinom(\ \ )$. The parameters indicated by the *size* (number of trials) and *prob* (probability of success on an individual trial) arguments.

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

For the **poisson** distribution, the functional names are $dpois(\ \ )$, $ppois(\ \ )$, $qpois(\ \ )$, and $rpois(\ \ )$. The sole parameter is indicated by the *lambda* (mean) argument.

```
dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)
```

In the help menu, searching for "distributions" finds a listing of many more distributions that are included included in *base R*.

Sample Function

While the random number generators in the previous two functions will generate values from two 'named' distributions, the $sample(\ \ )$ function can be used to generate values from any discrete random variable with a finite sample space. To do so, a few arguments must be set. x indicate all the possible values in the sample space. *prob* indicates the corresponding probabilities for each value in x. *size* indicates the number of elements of x to be sampled. *replace* is a logical argument. Setting *replace* to $TRUE$ indicates that values taken from x should be replaced after each selection. This will simulate an independent and identically distributed sample from x. Setting *replace* to $FALSE$ indicates that each value in x will not be allowed to be selected more than once. Effectively, when $replace = FALSE$, *size* can not be any larger than the number of elements in x.

```
possible.values <- c(5, 7, 69, 80)
related.probabilities <- c(0.1, 0.2, 0.3, 0.4)
sample.size <- 10

sample(x = possible.values, size = sample.size, replace = TRUE, prob = related.probabilities)

 [1] 80  7 80  5 80 69 80 80  7 80
```

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Determine the cumulative probability that a Normal Distribution with mean equal to 10 and standard deviation equal to 2 is less than or equal to 13.8.

2. Determine the cumulative probability that a Normal Distribution with mean equal to 10.5 and standard deviation equal to 1.8 is greater than 11.9.

3. For each whole number mean between 0 and 15, determine the cumulative probability that a Normal Distribution is less than or equal to 10 when the standard deviation equals 5.2. This should be done with a single command.

4. Determine the cumulative probability that a t Distribution with degrees of freedom equal to 10 is less than or equal to 0.8.

5. Determine the cumulative probability that a t Distribution with degrees of freedom equal to 20 is greater than -1.0.

6. For each multiple of 10 degrees of freedom between 1 and 10, determine the cumulative probability that a t is less than or equal to 1.3 This should be done with a single command.

7. Sample three values from the first ten upper case LETTERS. Each possibility should have the same chance of being picked as the other possibilities. The same letter can be selected more than once.

8. Sample five values from the first one-thousand positive integers. Each possibility should have the same chance of being picked as the other possibilities. The same letter can not be selected more than once.

Questions. Questions. Questions. I need more questions

Chapter 26: Basic Statistics Functions

Sample Mean

The *mean*( ) function computes the average of all values in a vector[31]. *mean*( ) is an example of a function with an *na.rm* logical argument. Its default setting is *FALSE*. This can lead to an *NA* result, if there are missing values in the vector being averaged.

```
x <- c(1, 5, 5, 5, 5, NA, 5, 5, 5, 9)
mean(x)

[1] NA

mean(x, na.rm = TRUE)

[1] 5
```

Sample Variance & Sample Standard Deviation

The *var*( ) computes the (unbiased) **sample variance** of a given vector. This is the sample variance whose denominator is one less than the number of data values[32]. Corresponding to this is the *sd*( ) function that computes the sample **standard deviation**. As with *mean*( ), missing values *NA* can affect the result. As such, *na.rm* is an argument for these two functions.

```
var(x, na.rm = TRUE)

[1] 4

sd(x, na.rm = TRUE)

[1] 2
```

Sample Quantiles

Sample quantiles are computed using *quantile*( ). Its first argument is a vector of values. The *probs* argument is a numerical vector identifying the quantiles to be computed. The default is to produce a five-number summary. *na.rm* is available to handle missing values. The *type* argument is used to determine the method used for interpolating between consecutive order statistics.

```
x <- c(1:10, NA)
quantile(x, probs = c(0, 0.2, 0.5, 0.76, 1), na.rm = TRUE, type = 6, names = TRUE)

   0%   20%   50%   76%  100%
 1.00  2.20  5.50  8.36 10.00
```

Sample Range

The *range*( ) function produces a two element vector containing the largest and smallest values of its given arguments[33]. The *na.rm* argument is available to handle missing values. The *finite* determines if infinite values should be ignored.

```
range(1:10, NA, Inf, na.rm = TRUE)  #multiple arguments are given

[1]   1 Inf

range(1:10, NA, Inf, na.rm = TRUE, finite = TRUE)

[1]  1 10
```

Character data can also be used.

```
range(letters, NA, Inf, na.rm = TRUE)  #multiple arguments are given

[1] "a" "z"
```

---

[31] And many other R objects
[32] If n is the length of the vector, the denominator is $n-1$.
[33] vectors, dataframes, matrices, etc.

Frequencies

The *table*( ) will produce a contingency table for the elements of the objects[34] given to it. With a single argument, it produces a one-way table of frequencies

```
x <- c(1, 2, 4, 3, 2, 3, 4, 3, 2, 1, 2, 3)
table(x)

x
1 2 3 4
2 4 4 2
```

When two arguments are given, a two-way table of frequencies is produced.

```
y <- c("a", "b", "b", "a", "c", "b", "a", "b", "a", "c", "b", "b")
table(x, y)

   y
x   a b c
  1 1 0 1
  2 1 2 1
  3 1 3 0
  4 1 1 0
```

It is important to look at the structure of the result. In this case, a **table** is returned. Values within this result can be retrieved by treating this result as an array.

```
str(table(x, y))

 'table' int [1:4, 1:3] 1 1 1 1 0 2 3 1 1 1 ...
 - attr(*, "dimnames")=List of 2
  ..$ x: chr [1:4] "1" "2" "3" "4"
  ..$ y: chr [1:3] "a" "b" "c"
```

Data Summaries

The *summary*( ) function is generic function whose results will depend on the type of *R* object is given. This example makes use of a dataframe.

```
numbers <- 1:10
logicals <- c(T, T, T, T, F, F, T, F, F, T)
dat <- data.frame(numbers, logicals)
summary(dat)

    numbers       logicals
 Min.   : 1.00   Mode :logical
 1st Qu.: 3.25   FALSE:4
 Median : 5.50   TRUE :6
 Mean   : 5.50
 3rd Qu.: 7.75
 Max.   :10.00
```

This produced the mean and the five-number summary of the numerical data, and the frequencies of the logical data.

[34] Any objects that can viewed as factors.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Determine the average of the values in the following vectors.
    a. StatFunc001
    b. Math001
    c. Math002
    d. Math003
    e. Math004
2. Determine the average of the non-missing values in the following vectors.
    a. StatFunc001
    b. Math001
    c. Math002
    d. Math003
    e. Math004
3. Determine the variance of the values in the following vectors.
    a. StatFunc001
    b. Math001
    c. Math002
    d. Math003
    e. Math004
4. Determine the average of the non-missing values in the following vectors.
    a. StatFunc001
    b. Math001
    c. Math002
    d. Math003
    e. Math004
5. Determine the 0, .25, .5, .75, and 1 quantile for each of the following randomly generated sets of 10000 numbers:(Use type 6 quantiles.)
    a. Uniform Distribution with minimum equal to zero and maximum equal to 4.
    b. Normal Distribution with mean equal to 0 and standard deviation equal to 5.
6. Determine the frequency of each unique element in StatFunc002.
7. Summarize the contents of the following dataframes.
    a. Loblolly
    b. MissingValues02
    c. ModifyLists001

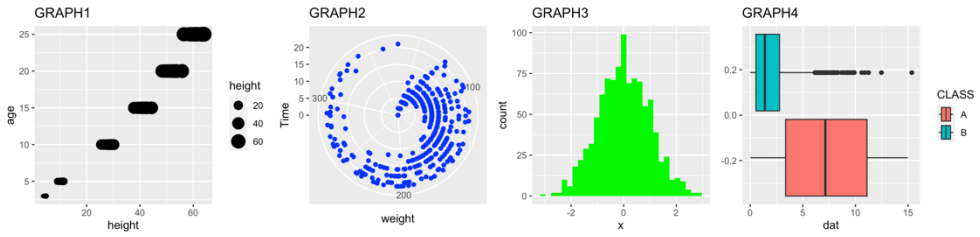
Questions. Questions. Questions. I need more questions

Chapter 27: Patchwork

patchwork is a package used for combining graphs into a single image and controlling thier layout. *patchwork* is mainly used in conjunction with graphs produced by a package called *ggplot2*[35].

```
library(patchwork)
```

When graphs are to be layed out using *patchwork*, they will have already been created using *ggplot2* and each graph will have had a variable name assigned to is. For the following examples, these four graphs will be used. Their variable name is noted in the upper left hand corner of each graph[36]

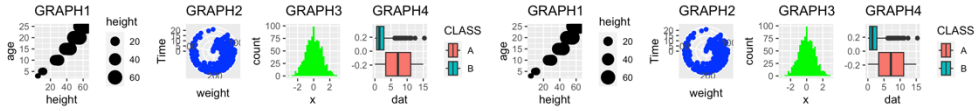


Basic Operations

A great deal of control over the layout of multiple graphs comes from three basic operators: "|", "/" and "()". The "|" places

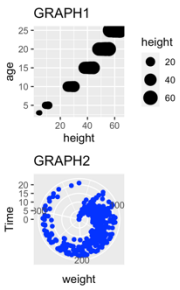
The "|" sign place graphs side-by-side. The "|" was used to make the first graph on this page. This example places the four graphs in a row. Each is scaled down.

```
graph1 | graph2 | graph3 | graph4 | graph1 | graph2 | graph3 | graph4
```



The "/" sign is used to stack graphs on top of one another.

```
graph1/graph2
```

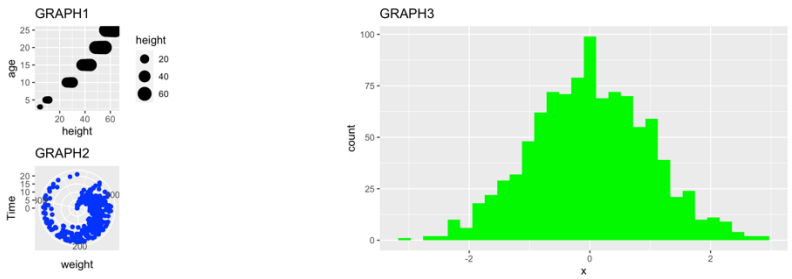


The parenthesis are used to group graphs together when the "|" and "/" commands are used together. With one grouping, *graph*1 and *graph*2 are stacked. Their stacked image is place next two *graph*3.

---

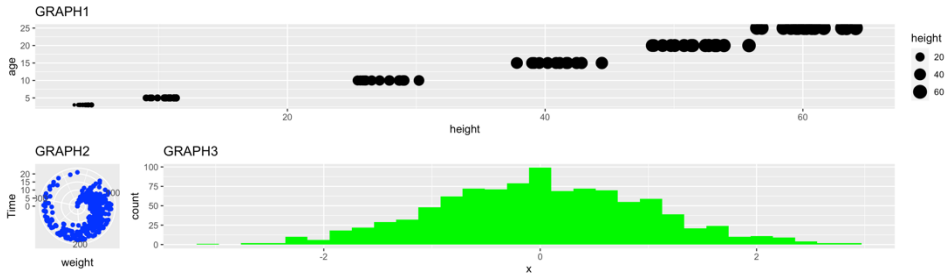
[35] The *ggplot2* package will be covered in the next chapter.
[36] For consistency, each set of graphs will be displayed in a region that is the same height and width as this first graph. This may make some graphs look odd. However, the layout is the current concern.

```
(graph1/graph2) | graph3
```



While another grouping places *graph*1 above a side-by-side pairing of *graph*2 and *graph*3.

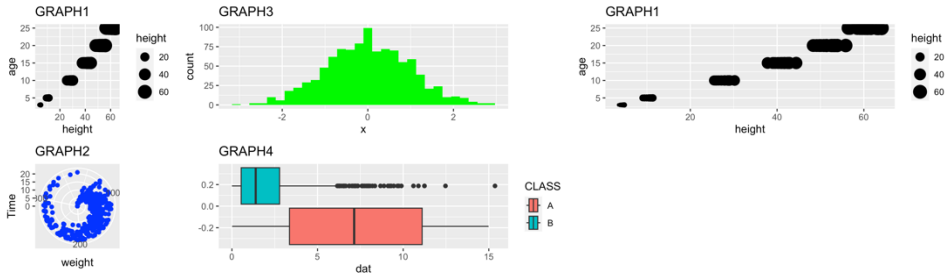
```
graph1/(graph2 | graph3)
```



Other Operations

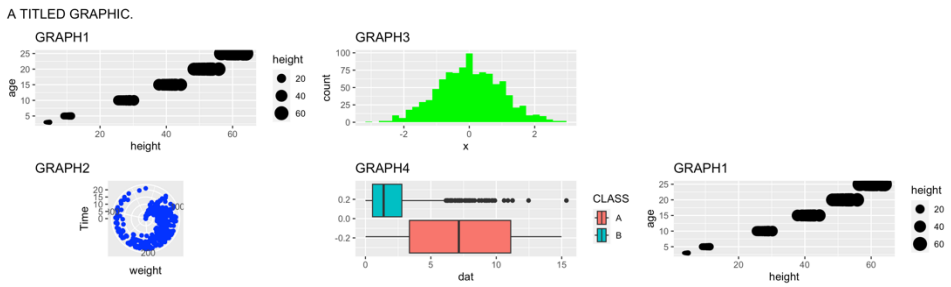
Other operations can be used. The "+" can be used to make a grid of graphics. When used in conjunction with the *plot_layout*( ) function the dimension and fill of the grid can be controlled[37].

```
graph1 + graph2 + graph3 + graph4 + graph1 + plot_layout(nrow = 2, ncol = 3, byrow = FALSE)
```



The *plot_spacer*( ) command can be used to skip a graphing position. While adding a *widths* argument to *plot_layout* will even out the widths of the graphs, if possible. *plot_annotation*( ) can add an overall title.

```
graph1 + graph2 + graph3 + graph4 + plot_spacer() + graph1 + plot_layout(nrow = 2, ncol = 3, byrow = FALSE, widths = 1) + plot_
annotation(title = "A TITLED GRAPHIC.")
```



[37] This is much the same as the *matrix*( ) function.

Exercises

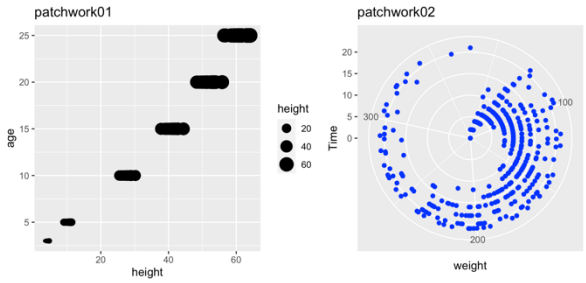
The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

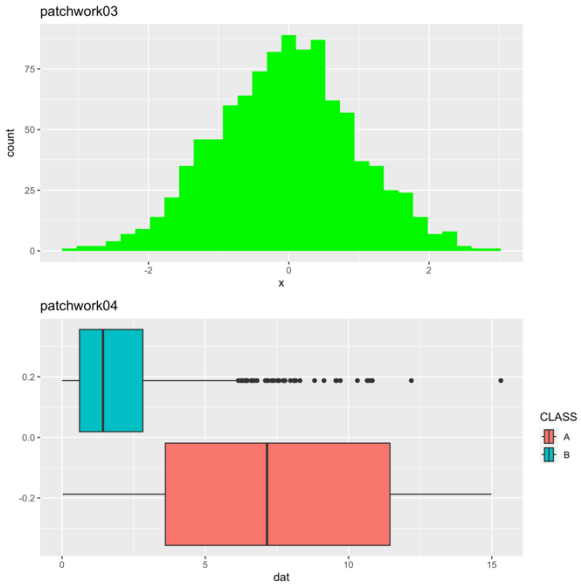
Unless directed otherwise, a result should always be displayed.

Recreate each of these graphics using the graphs named: patchwork01, patchwork02, patchwork03, patchwork04.

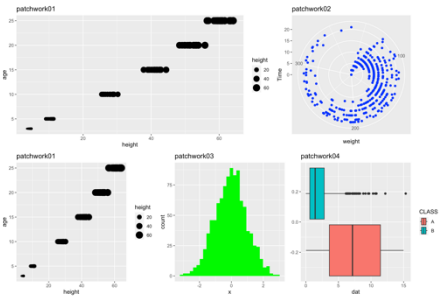
1. Graphic #01



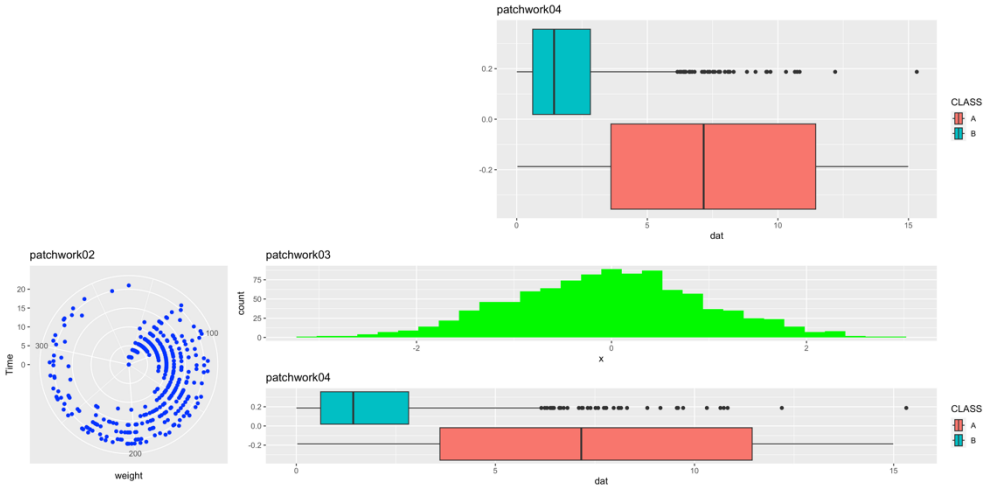
2. Graphic #02



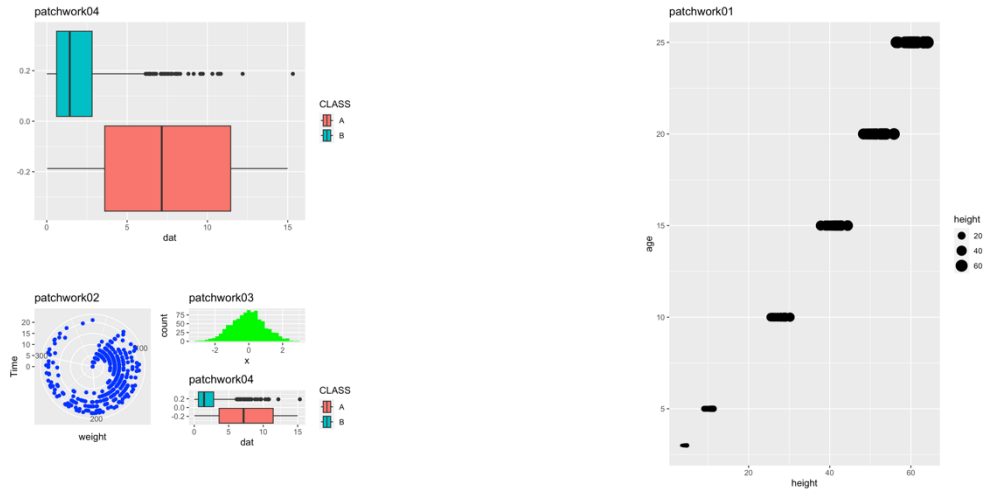
3. Graphic #03



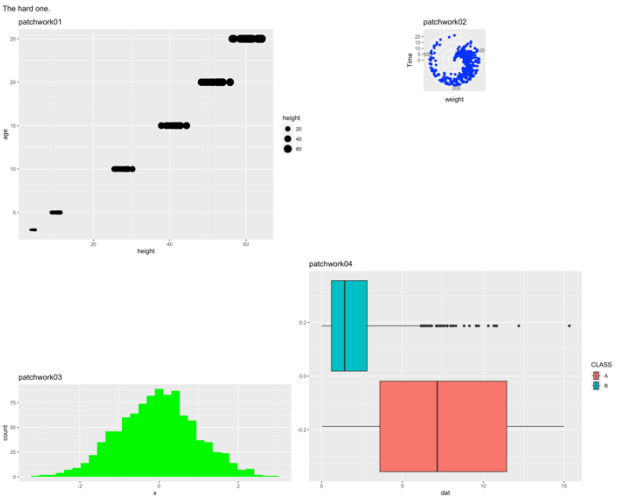
4. Graphic #04



5. Graphic #05



6. Graphic #06



Chapter 28: ggplot2

*ggplot*2 is a package for building graphics. The **gg** in the name stands for `grammar of graphics`.

```
library(ggplot2)
```

Each graphic made using *ggplot*2 will require, at a minimum, four components to be identified.

1. At least one data set
2. The *ggplot*( ) function to initialize/start up a plot.
3. A set of aesthetic mappings *aes*( ) that will map variables to aesthetic (visual) properties.
4. A set of geometric objects **geoms** that will each create a layer in the actual graphic. (Wickham 2016)

Other components can be added to further customize the graphic, but these four will be the current focus.

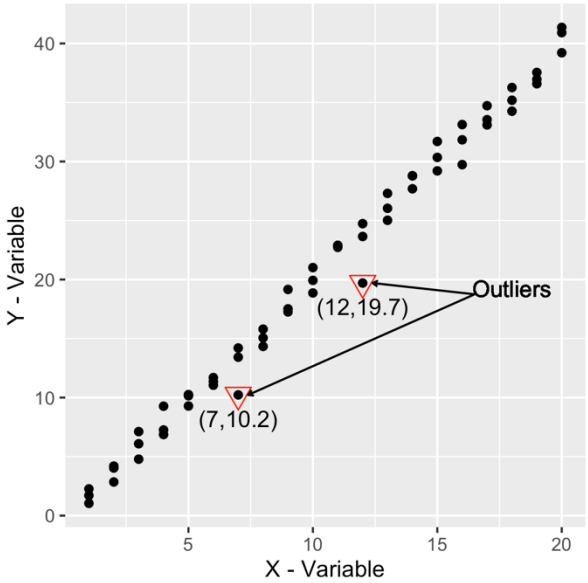
Data

To illustrate the basic parts of a graphic generated by *ggplot*( ), a contrived (made-up) data set, called **linear.data**, will be used. Seeing the data constructed, makes it easier to see the function of the various plotting options.

```
var.1 <- rep(1:20, each = 3)
var.2 <- 2 * var.1 + rnorm(60)
var.2[c(19, 36)] <- var.2[c(19, 36)] - 5
group.var.1 <- rep(c("A", "B", "C"), times = 20)
group.var.2 <- rep(c("V", "W", "X", "Y", "Z"), each = 12)
linear.data <- data.frame(var.1, var.2, group.var.1, group.var.2)
head(linear.data)
```

```
  var.1    var.2 group.var.1 group.var.2
1     1 1.038067           A           V
2     1 1.707474           B           V
3     1 2.258788           C           V
4     2 2.847868           A           V
5     2 4.195783           B           V
6     2 4.030124           C           V
```

The *ggplot*( ) function is designed to use a *tidy* data set. This is a data set where each row represents a single observation and each column a single variable. By design, **linear.data** is a tidy data set.



linear. data consist of four variables: two numeric, and two categorical. It exhibits a very strong linear trend with the nineteenth and thirty-sixth observations set relatively far outside the over all linear pattern.

ggplot()

On its own, the $ggplot(\quad)$ function sets up a blank canvas. The grey box is that canvas[38].

```
ggplot()  #The basic ggplot window
```


It should make sense that nothing of interest appeared. In order for a function to deliver any type of information about a data set, it needs to know about the data set. In this next graph, the **data** argument in $ggplot(\quad)$ will be set to **linear.data**. Additionally, the graphic will be assigned to a variable[39].

```
basic.plot <- ggplot(data = linear.data)  #The basic ggplot window
basic.plot
```


This returns a grey box, also. Two questions should come to mind:

a. Why did the graph not change?
b. What did $ggplot(\quad)$ do with the data that was passed to it?

The answer to the first is that nothing in the $ggplot(data = linear.data)$ command indicates how the data should be represented. Without that information, nothing new can be shown.

To answer the second, the data is stored as a default. The **basic** structure for creating a graphic is to take the grey box and add geometric components to it[40],

$$ggplot(\ldots) + geoms(\quad).$$

These geometric functions, **geoms** need a data set to produce geometric objects. When they are not supplied with one directly, they inherit the data set supplied to the $ggplot(\quad)$ command.

In fact, $ggplot(\quad)$ has a second argument, **mapping**, that can also be inherited by subsequent **geom** functions. Again, this only occurs if they are not directly supplied as an argument to a subsequent **geom**.

[38] This is the default. It can be changed using themes
[39] This makes it easy to execute the graphic at any time, or to add to it.
[40] The plus sign + plays a special role when constructing graphics using *ggplots*. Geoms, and other functions, are added to the graphic one at a time using a plus sign.

aesthetic Mappings

The mapping argument is set take the results of the aesthetic $aes(\ \ )$ function. The $aes(\ \ )$ function produces a mapping between data values and visual properties of a graph. The first argument of $aes(\ \ )$ is x. This argument identifies which variable will be mapped to the horizontal axis [41]. The second argument y identifies which variable is mapped to the vertical axis. Depending upon the type of graph being built, the y argument does not always need to be explicitly provided.

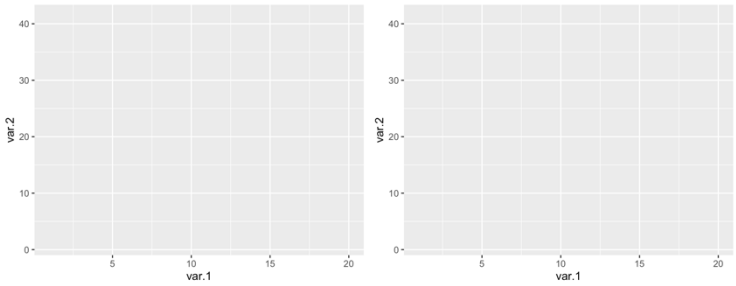
a. Histogram do not require y to be stated. The vertical axis of a histogram is a count based on the horizontal data values.
b. Scatterplots do require y to be assigned. A point can't be plotted without all its components.

If a default mapping is desired between the variable values and a visual property of a graph then additional arguments can be supplied to $aes(\ \ )$. If a default mapping is not desired for a particular visual property, a mapping can be made outside of $gglpot(\ \ )$ and inside the **geom** where it is desired.

```
# color is assigned to a variable inside aes(). This will map the values of group.var.1 to a set of colors.
aes.plot.1 <- ggplot(data = linear.data, mapping = aes( x = var.1, y = var.2, color = group.var.1))

# No color aesthetic mapping is done at this stage. Color will either be mapped or set once the geometric objects are added on.
aes.plot.2 <- ggplot(data = linear.data, mapping = aes( x = var.1, y = var.2))

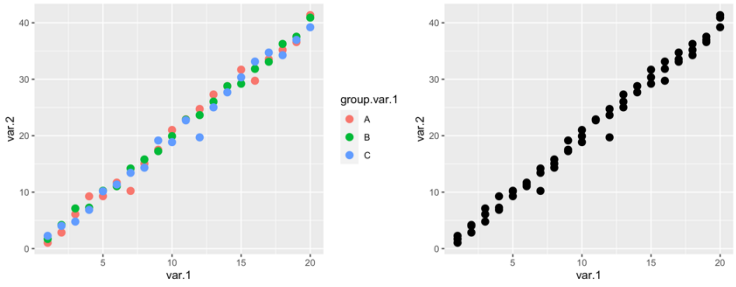
# ( aes.plot.1 | aes.plot.2 )
```



These two graphs now have a coordinate system. This is something that can be assigned once the horizontal and vertical variables have been assigned. There is still not indication of the data because there is nothing that assigns a geometric object to the data.

A **color** aesthetic mapping[42] was added left hand graph. This has no visual effect for the same reason. Nothing indicates what the geometric objects are that will appear in the graphic.

However, looking at **aes.plot.1**, the color of the objects will have some relationship to the values of **group.var.1**. This is because a default mapping is being produced by $aes(\ \ )$. In **aes.plot.2**, at this point, the color of any geometric object will have not relationship to any of the given variables. The default color for each geometric object will be used.



The graph on the left is one example of a graph that could be made using **aes.plot.1** to start a plot and set some default aesthetics. The key thing to notice is that the color is not always the same. In fact the colors correspond to values of **group.var.1**. The legend indicates the mapping. For the graph on the right, **aes.plot.2**, black, the default, was used.

[41] It is assumed that a cartesian coordinate system is being used. This is the default coordinate system. When a different coordinate system is used, an appropriate mapping of the x and y arguments in the $aes(\ \ )$ function will be used.
[42] Other aesthetics exist. Some are **alpha**(transparency), **fill**, **group**, **shape**, **size**, and **stroke**.

geoms

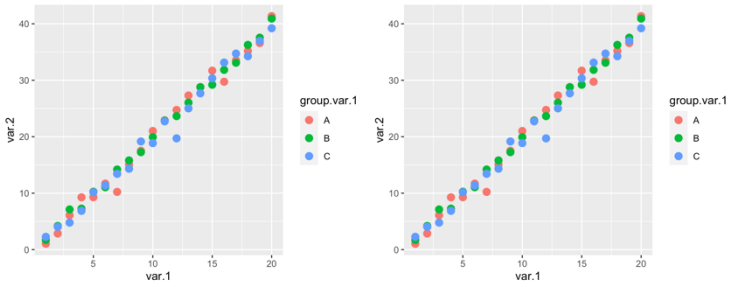
A **geom** function will add a geometric object, or set of geometric objects, to a plot. Each **geom** creates a layer that can be applied to the basic *ggplot*( ) grid. If no data or aesthetic arguments[43] are specified in the **geom** function, it will inherit the defaults set in the *ggplot*( ) function. Otherwise, a **geom** will override the defaults with those that are specifically specified within it. Three things should be noticed.

1. Default data and aesthetic mappings do not need to be specified in *ggplot*( ). The appropriate arguments can be set within each **geom**.

```
# No default data or aesthetic mappings are set in ggplot()
example.plot.1 <- ggplot( ) +
  geom_point(data = linear.data , mapping =  aes(x = var.1, y = var.2, color = group.var.1),  size=3 )

# No data or aesthetic mappings are given in geom_point()
example.plot.2 <- ggplot( data = linear.data, mapping =  aes(x = var.1, y = var.2, color = group.var.1)  ) +
  geom_point( size=3 )

# (example.plot.1 | example.plot.2)
```



2. It is not an all or nothing situation. In a specific **geom** some arguments can be overridden and others can be left to the default values.

3. The data set used by in a **geom** can vary from the default. A graphic can be made from multiple data sets.

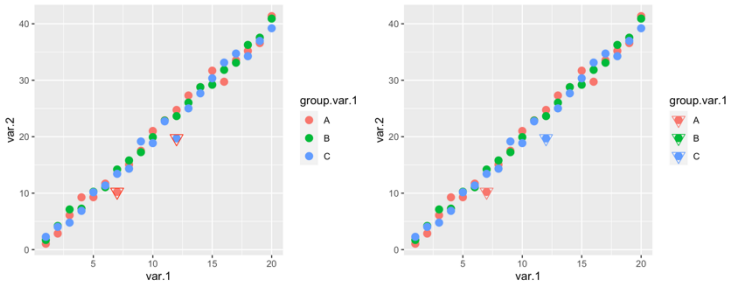
```
# outlying.data contains the coordinates of the two outlying data points.
outlying.data

   var.1    var.2 group.var.1
19     7 10.22431           A
36    12 19.70552           C

example.plot.3 <- example.plot.2 +
  geom_point( data = outlying.data, size = 4, color = "red", shape = 6)

example.plot.4 <- example.plot.2 +
  geom_point( data = outlying.data, size = 4, shape = 6)

# (example.plot.3 | example.plot.4)
```



Notice that in both plots, the **outlying.data** points are plotted with with an aesthetic triangle shape. The left hand graph explicitly set the color to red. The default aesthetic color mapping was overridden. No color was specified in the right-hand plot. The default aesthetic color mapping was used.

[43] Not all aesthetics are applicable to all geoms. Check the geoms help documentation.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

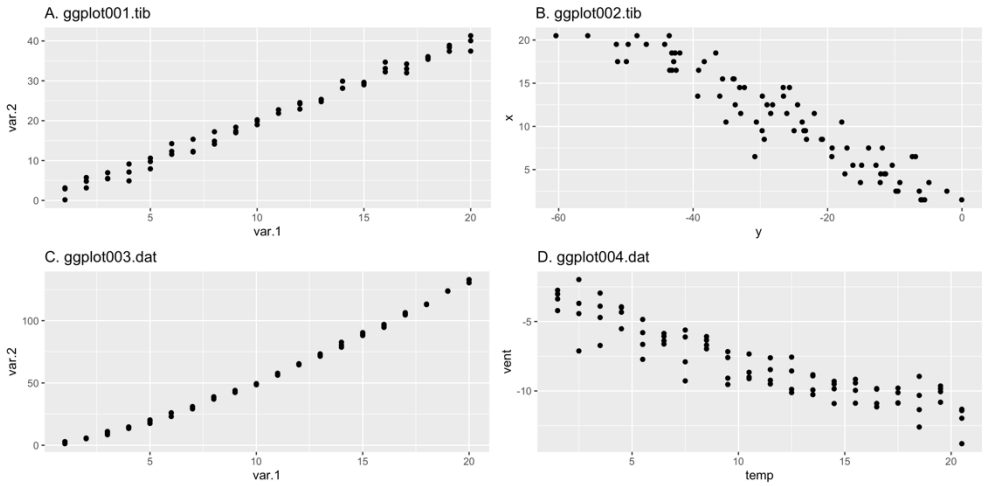
When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

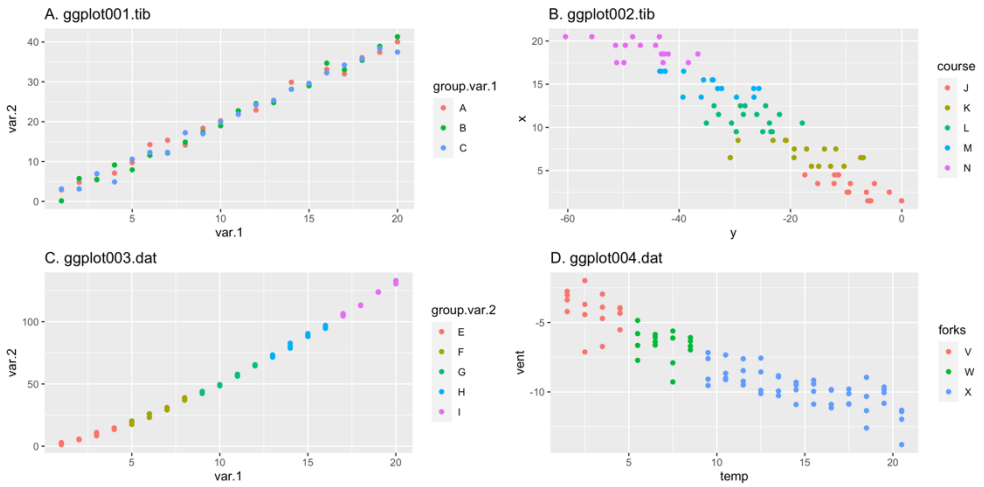
For the following, recreate the indicated graphs. Use the tibbles and dataframes named ggplot001.tib, ggplot002.tib, ggplot003.dat, and ggplot004.dat. Some graphs will indicate which dataset is used. Others will leave that for you to figure out. Do not try to recreate the title of the graph.

A listing of shapes can be found in the help menu under points.

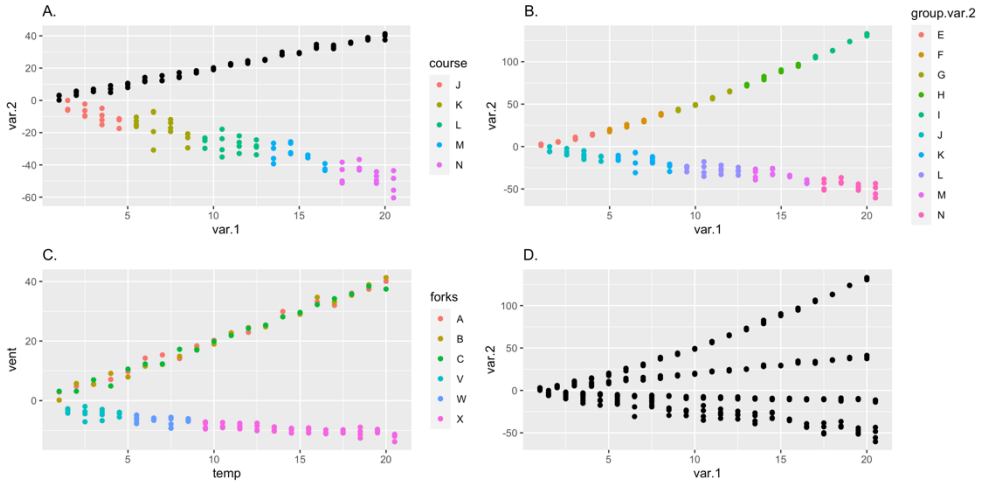
1. Use the data.frame/tibble listed in the title to recreate the following graphs:



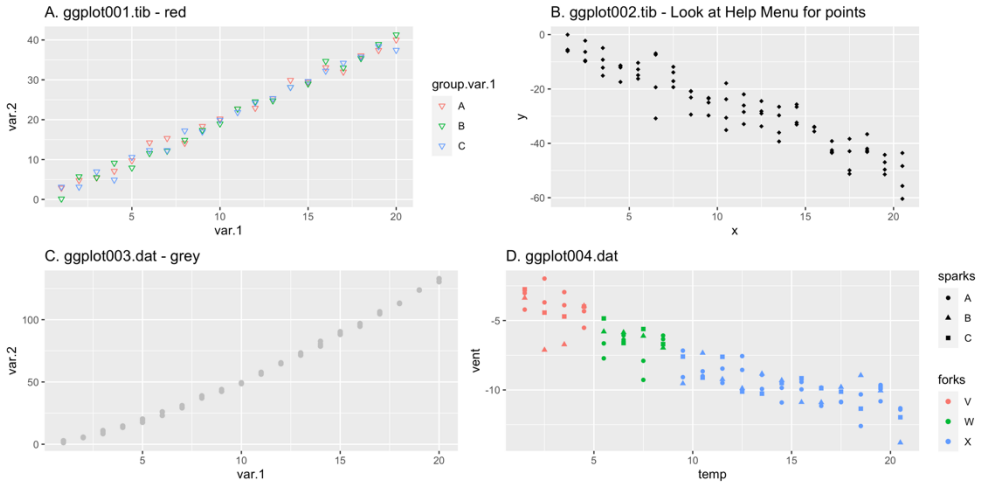
2. Use the data.frame/tibble listed in the title to recreate the following graphs:



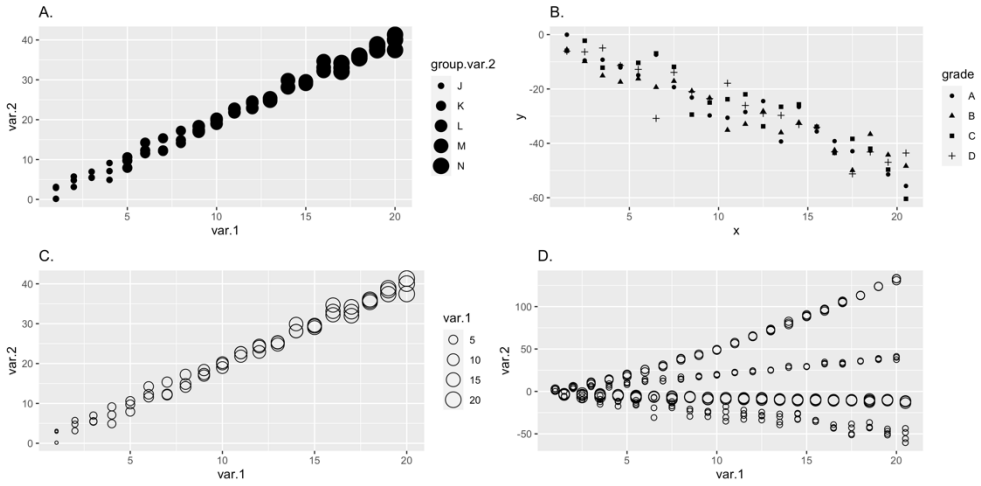
3. Recreate the following graphs:



4. Use the data.frames/tibbles listed in the title to recreate the following graphs:



5. Recreate the following graphs:



Chapter 29: ggplot2 - Coordinate Systems

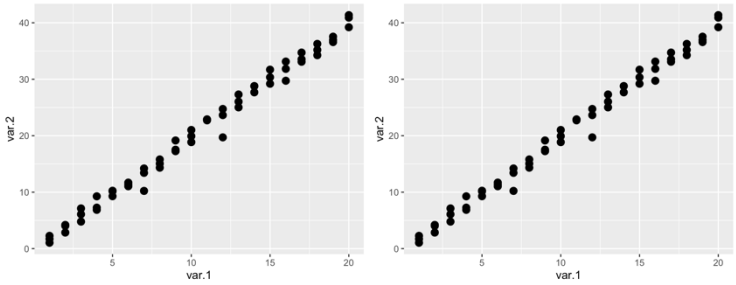
A Cartesian coordinate system is the default for most geoms. This can be overridden by adding a **coord** function to a plot.

Returning to a simple scatterplot of **linear.data** on a Cartesian coordinate system we have the following graphs. The second plot is constructed by using the *coord_cartesian*( ) function. There should not be, and there is no, difference between the two.

```
example.plot.1 <- basic.plot +
  geom_point( mapping =  aes(x = var.1, y = var.2), size=3)

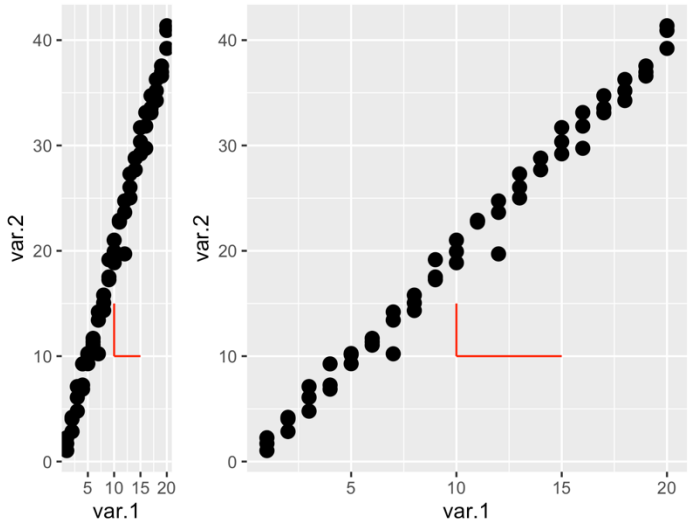
example.plot.cartesian <- example.plot.1 +
  coord_cartesian()

# ( example.plot.1 | example.plot.cartesian )
```



The *coord_fixed*( ) function fixes the displayed ratio of units on each axis. Setting its *ratio* argument to r indicates that the displayed length of one unit on the horizontal axis is r times the displayed length of one unit on the vertical. In these example the ratio is set to 1/2 and 2. Line segments of length five are plotted on each. The ratio of the displayed length can more easily be seen.

```
example.plot.fixed.times.two <- example.plot.1 +
  coord_fixed(ratio = 2)
example.plot.fixed.times.half <- example.plot.1 +
  coord_fixed(ratio = 1/2)
# ( example.plot.fixed.times.two | example.plot.fixed.times.half  )
```

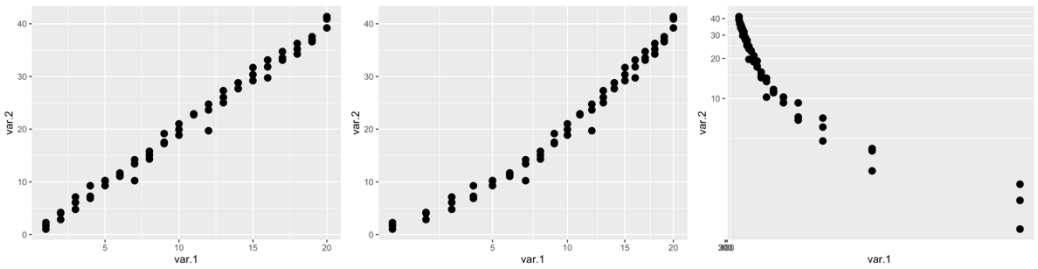


The narrowness of the first graph is a direct result of the fixed ratio. Looking at the range of each variable, the vertical runs from 0 to 40 approximately, and the horizontal from 0 to 20. Because of the fixed ratio, a displayed vertical distance of 20 is twice the displayed horizontal displayed distance of 20. Since the vertical range is twice the horizontal range, the graph would need to be four times taller than it is wide.

The *coord_flip*( ) function flips the x and y axis. Nothing exciting about it.

The *coord_trans*( ) function can be used to transform the values on the each axis. This is done by supplying a transformation to either, or both, of its **x** or **y** arguments. Notice that this changes the pattern of the plotted points. This is because on these scales the distance from zero is proportional to a selected function of a data value.

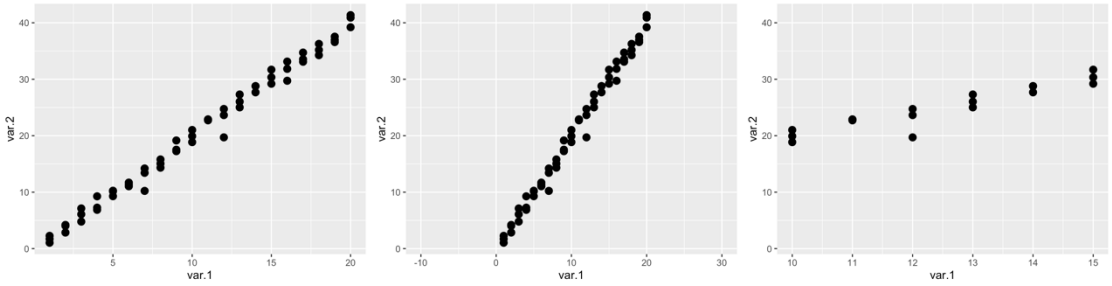
```
library(scales)
example.plot.sqrt.x <- example.plot.1 + coord_trans(x = sqrt_trans() )
example.plot.sqrt.x.log10.y <- example.plot.1 + coord_trans(x = "reciprocal", y = "log" )
# ( example.plot.1 | example.plot.sqrt.x  | example.plot.sqrt.x.log10.y )
```



Several different transformation functions can be used. These can be found by searching for "_trans" in the help menu. Taking the portion of their name that appears before "_trans" as a string will allow thier direct use[44] However, some will require additional arguments. *boxcox_trans*( ) requires p, a transformation exponent. In this case, the transformation needs to be explicitly written into *coord_trans*( ) as $boxcox\_trans(p = 2)$[45]

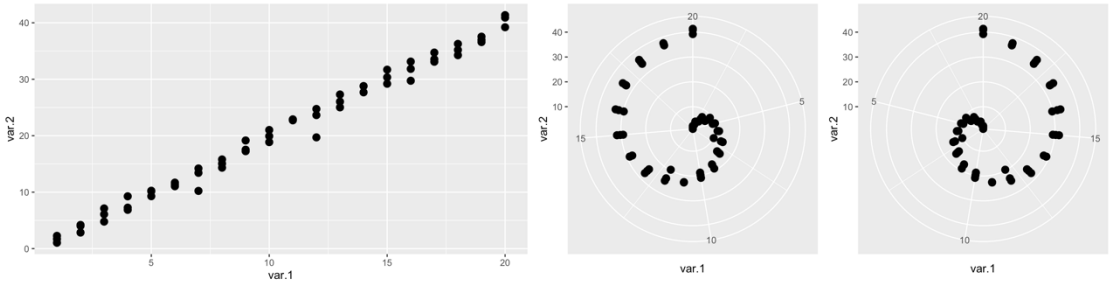
Each of these functions has two arguments **xlim** and **ylim**. These can be used to restrict or enlarge the range of the visible axis.

```
example.plot.expanded.xlim <- example.plot.1 + coord_cartesian(xlim = c(-10,30))
example.plot.restricted.xlim <- example.plot.1 + coord_cartesian(xlim = c(10,15))
# ( example.plot.1 | example.plot.sqrt.x  | example.plot.sqrt.x.log10.y )
```



The *coord_polar*( ) function sets up polar coordinates. The **theta** argument indicates which position variable will be mapped to an angle. The remaining position variable will be mapped to a radius. The **start** argument indicates, in radians, where all angles are measured from. The default zero angle is 12:00. The **direction** argument indicates a clockwise (1) or counter-clockwise (-1) measurement of angles.

```
example.polar <- example.plot.1 + coord_polar()
example.polar.with.arguments <- example.plot.1 + coord_polar( theta = "x", start = 0, direction = -1 )
# ( example.plot.1 | example.polar | example.polar.with.arguments )
```



[44] In the example, "reciprocal" was used in place of *reciprocal_trans*( ).
[45] When accessing the "_trans" function directly, the *scales* library needs to be used.

Exercises

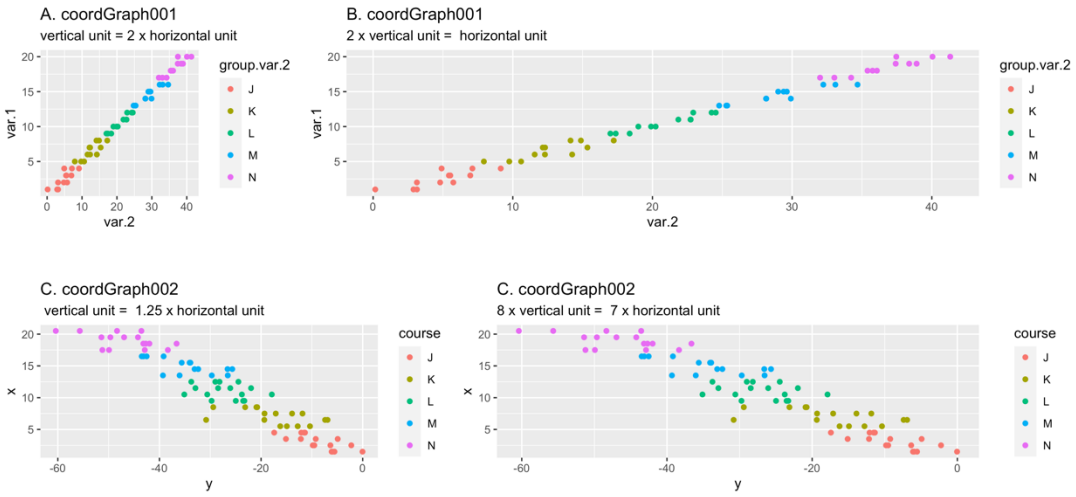
The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

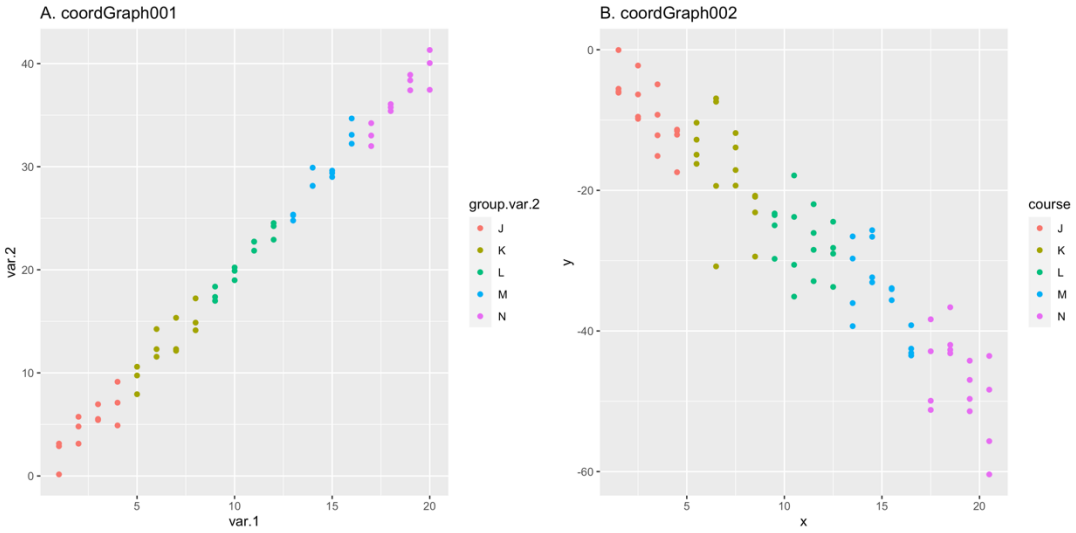
Unless directed otherwise, a result should always be displayed.

Questions with a * attached to them indicate that a transformation was used that is not explicitly covered in this section. You should reference the help menu to find the transformation and then use it.

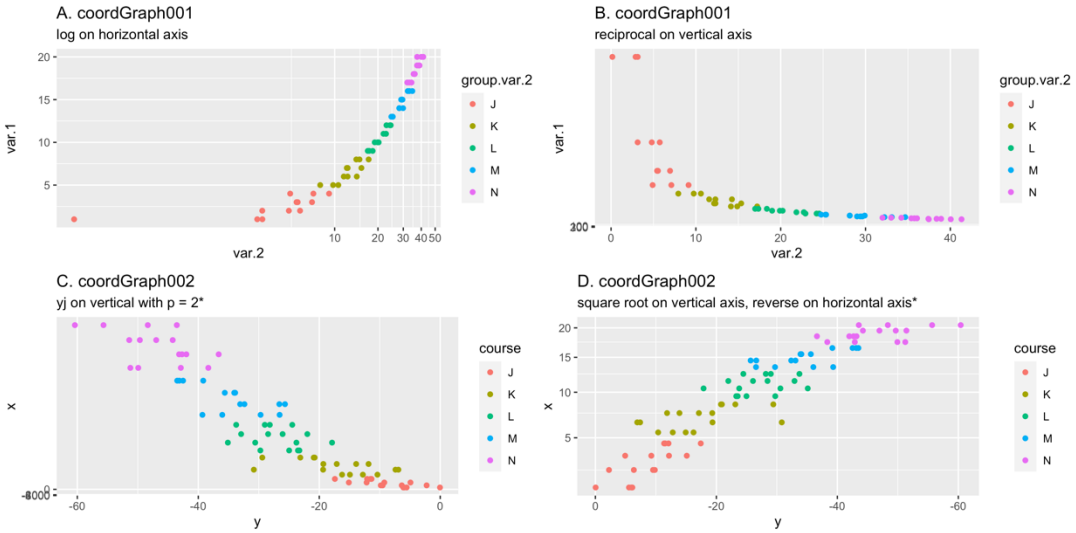
1. Using the graph named in the title, recreate the following graph by fixing the ratio of the coordinate system as indicated.



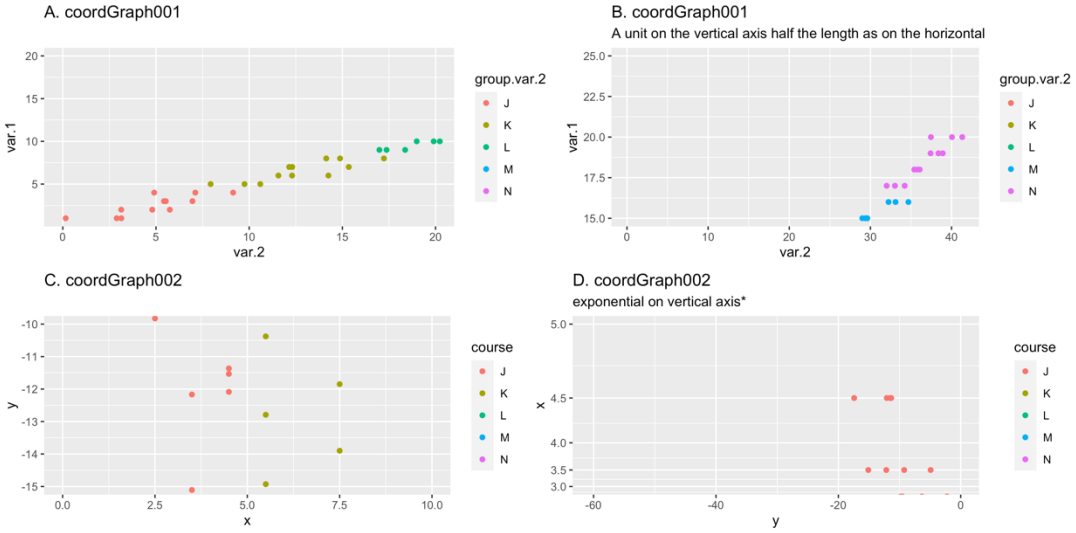
2. Using the graph named in the title, recreate the following graph.



3. Using the graph named in the title, recreate the following graph by using the indicated transformed coordinate system.



4. Using the graph named in the title, recreate the following graph by using the indicated transformed coordinate system with the respective limits.



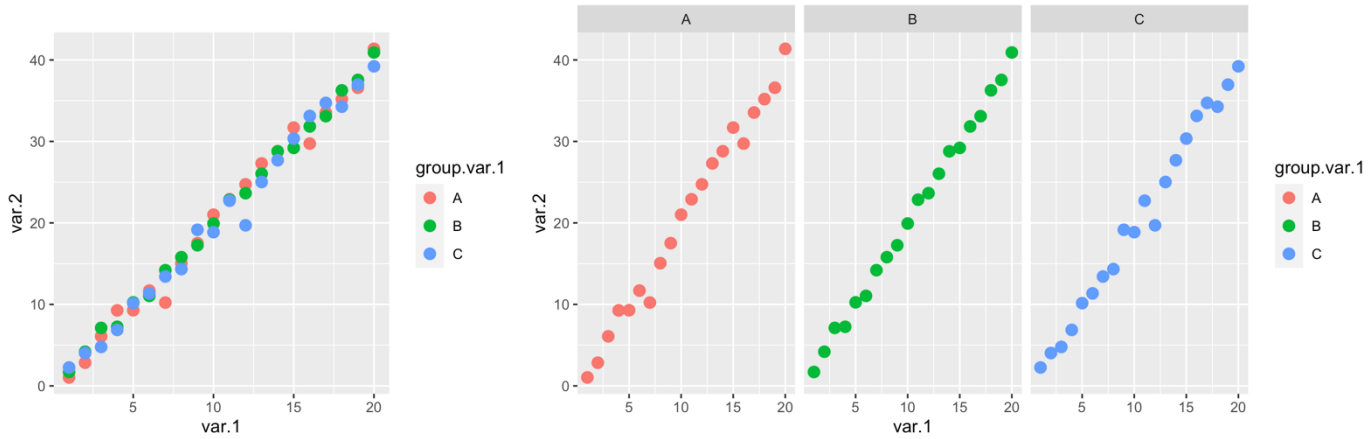
Chapter 30: ggplot2 - Faceting

The faceting functions provide a quick way to create multiple plots each of which focuses on a subset of the data set.

The first graph on the left displays **linear.data** with the colors determined by **group.var.1**.

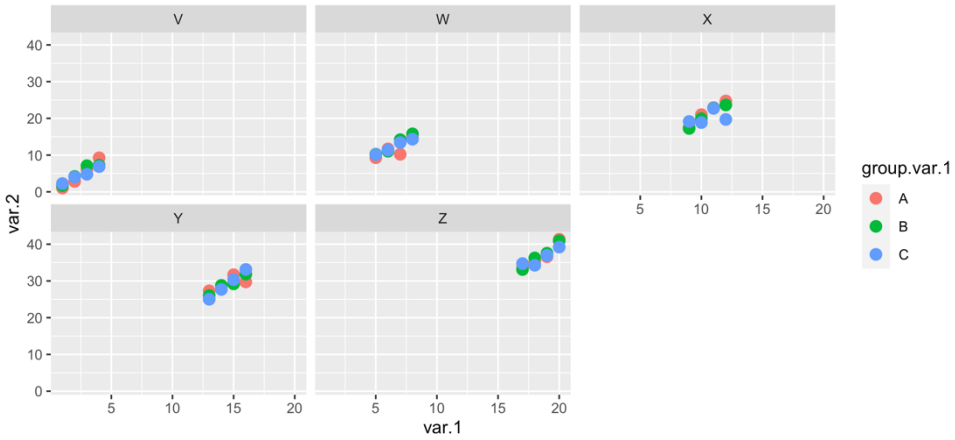
The *facet_wrap( )* function creates the remaining three graphs. Each graph focuses on data that takes on a single value from **group.var.1**. Since each graph contains only one type of data, from **group.var.1**, and the color is determined by the type, each graph contains only one color of point.

```
example.facet.1 <- example.plot.2 +
                facet_wrap( ~ group.var.1)
```



Using the variable **group.var.2**, the data set if split into 5 subsets. Each subset contains various values from **group.var.2**. Therefore, each graph contains multiple colors.

```
example.facet.2 <- example.plot.2 +
                facet_wrap( facets = ~group.var.2)
```



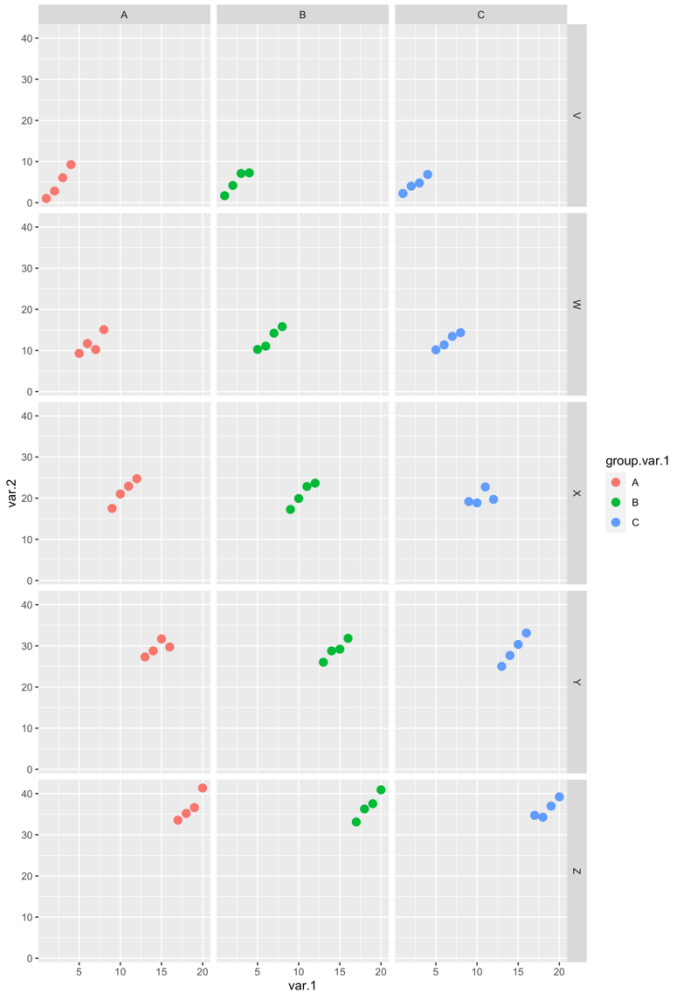
facet_wrap( ) creates graphs in a row, and then wraps down to subsequent rows, if needed.

Pay special attention to the notation for identifying the faceting variable. There is a tilde ~ followed by the name.

The *facet_grid*( ) function is used to form a matrix of plots. Each plot will focus on a subset build from pairing values from two variables. One variables values will define the rows and the other will define the columns of the matrix plot. The notation is

row variable ~ column variable.

```
example.facet.3 <- example.plot.2 +
                   facet_grid( group.var.2 ~ group.var.1)
```



As can be see, care should be used when faceting. If a variable takes too many values, it can lead to an excessively large number of graphs. Faceting with continuous data could lead to plots with at most single point each. Creating a variable, or set of variables, that bins the data would be advisable.

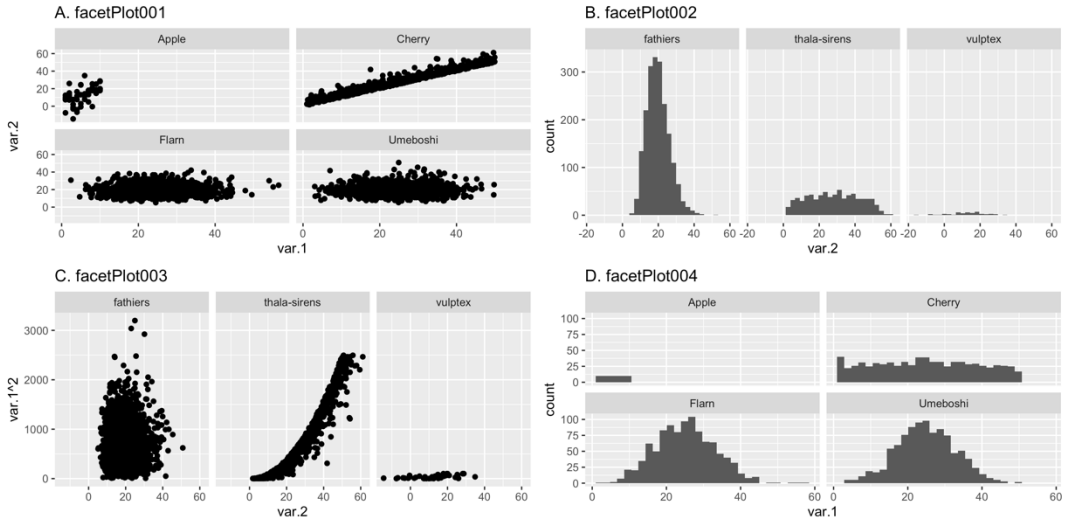
Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

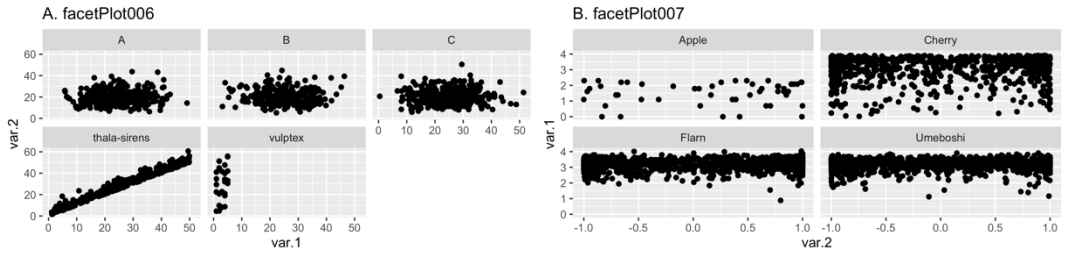
When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

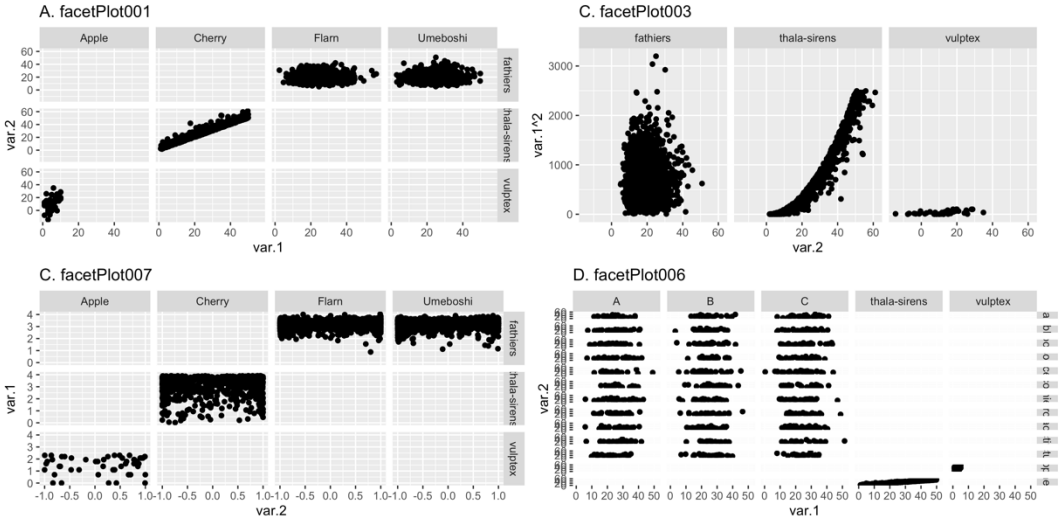
1. Using the graph in the title, and referring **ggplot005.dat**, recreate the following.



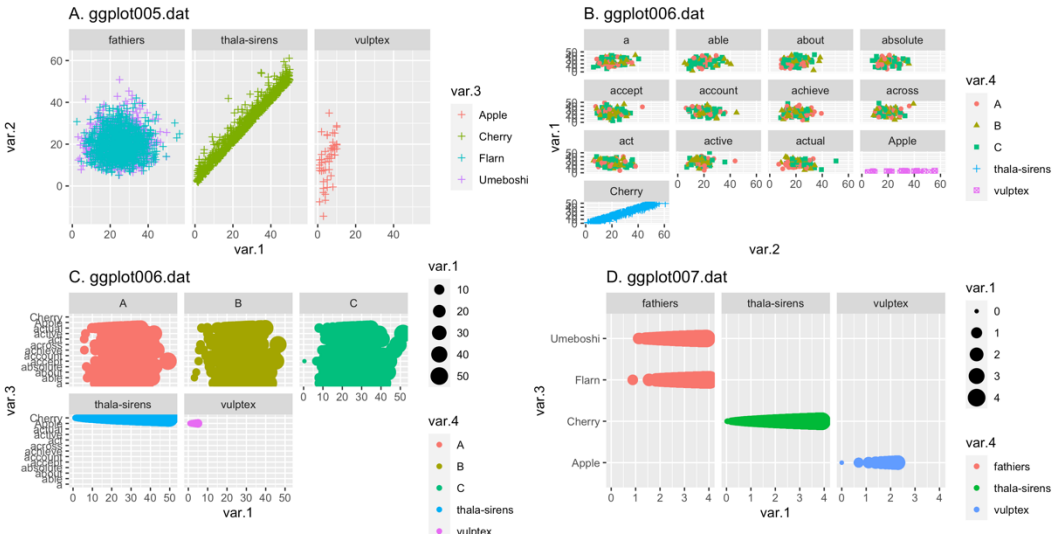
2. Using the graph in the title, and refer to **ggplot005.dat**, recreate the following.



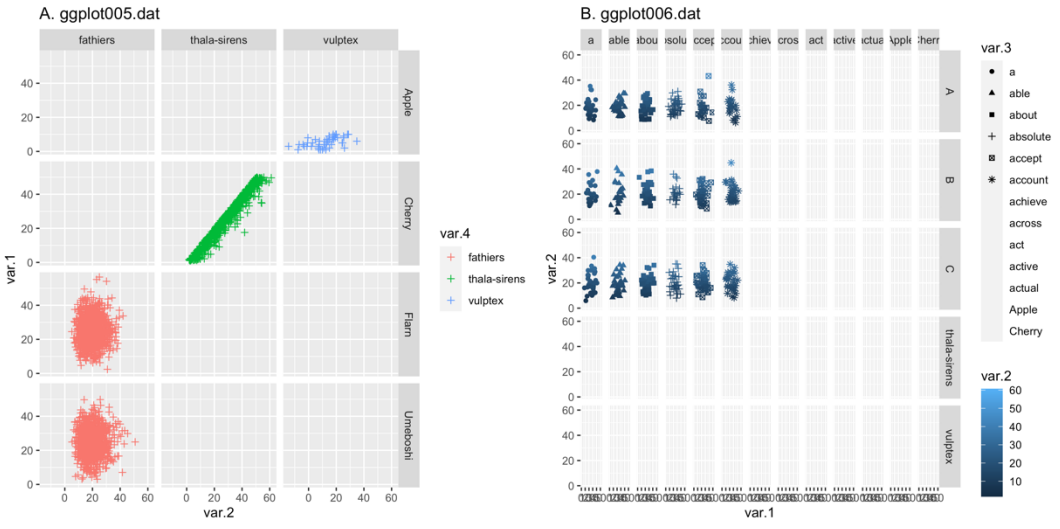
3. Using the graph in the title, and refer to **ggplot006.dat** or **ggplot007.dat**, recreate the following.



4. Using the dataframe named in the title, recreate the following graph.



5. Using the dataframe named in the title, recreate the following graph.



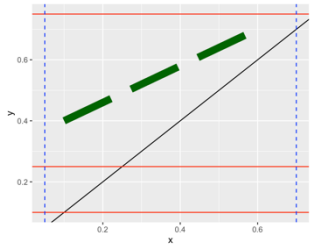
Chapter 31: ggplot2: Basic Geometric Objects

ggplot( ) has several geoms for constructing layers that consist of basic geometric objects. First, a few geoms that can build lines will be examined, then a single data set will be used to construct other geometric objects.

Lines

Constructing a line with an arbitrary intercept and slope is done with the *geom_abline*( ) function. Its arguments are the intercept and slope. While *geom_abline*( ) can be used to build a horizontal line, *geom_hline*( ) does the same thing. Only a y-intercept needs to be provided. A vertical line is made with *geom_vline*( ). These three function all build lines. As such, the line will extend to the edges of the graph. To construct a line segment instead, use *geom_segment*( ). The starting (x, y) and ending $(xend, yend)$ points must be given.

```
ggplot() +
  geom_abline( mapping = aes( intercept = 0, slope = 1)) +
  geom_hline( mapping = aes( yintercept = c(.1, .25, .75) ), color = "red" ) +
  geom_vline( mapping = aes( xintercept = c(.05, .7) ), color =  "blue", linetype = 2) +
  geom_segment( mapping = aes( x = .1, y = .4, xend = .6, yend = .7  ), color = "darkgreen", linetype = 5, linewidth = 4)
```



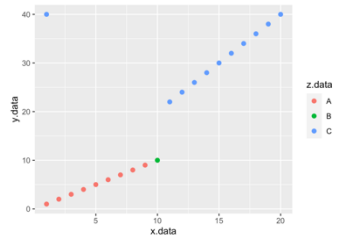
Points

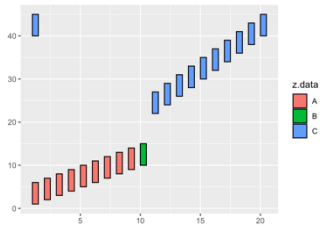
The example data set being used consist of two numeric variable and a single character variable. The data was built to resemble two line segments with different slopes. For all except the last point, the **x.data** values are increasing. The last **x.data** value repeats the first.

```
x.data <- c(1:10, 10, 10, 11:20, 1)
y.data <- c(1:10, 10, 10, 2 * (11:20), 40)
z.data <- c(rep("A", 10), "B", "B", rep("C", 11))
dat <- data.frame(x.data, y.data, z.data)
```

Using *geom_point*( ) a scatter plot can be built. The character data was mapped to a color aesthetic. As indicated, the red points and majority of the blue each follow a linear pattern. The slope of the blue being greater than for the red.

```
ggplot(data = dat) +
  geom_point( mapping = aes(x = x.data, y = y.data, color = z.data), size = 2)
```





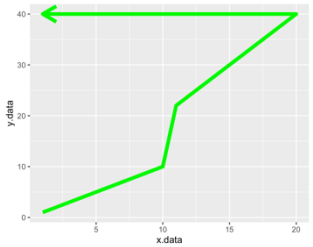
The *geom_rect*( ) can create rectangles. The length of the horizontal edge is controlled by the **xmin**, and **xmax** aesthetics For the vertical, use **ymin**, and **ymax**. In the above graph, a 'scatterplot of rectangles' was constructed.

```
ggplot(data = dat) +
  geom_rect( mapping = aes( xmin = x.data, xmax = x.data + .5, ymin = y.data, ymax = y.data + 5,  fill = z.data), color = "blac
k")
```

Paths

Using *geom_path( )* will connect the data points, in the order that they are provided, with straight lines. As can be seen, for this set of data, the indicated path would not indicate a functional relationship.

```
ggplot(data = dat) +
  geom_path( mapping = aes(x = x.data, y = y.data), size = 2, arrow = arrow(), color = "green")
```



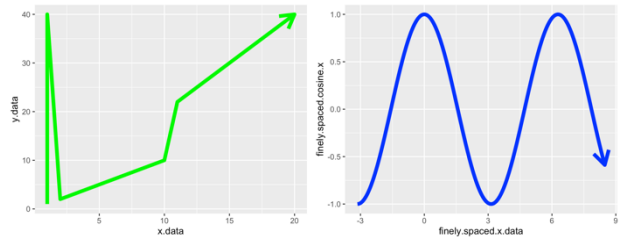
Similar to *geom_path( )*, *geom_line( )* will connect the data values. However, *geom_line( )* connects the points in an increasing order for the variable mapped to the *x* aesthetic. The *geom_line( )* only connects points with straight lines. If the connecting lines are small enough in length, the illusion of a smooth, non-straight, curve curve is achieved. The second graph is of the cosine function. It was produced with *geom_line( )* using 200 short line segments

```
curve.1 <- ggplot(data = dat) +
  geom_line( mapping = aes(x = x.data, y = y.data), size = 2, arrow = arrow(), color = "green")

finely.spaced.x.data <- seq(from = -pi, to = 2.7*pi, length.out = 201)
finely.spaced.cosine.x <- cos( finely.spaced.x.data  )
cos.dat <- data.frame( finely.spaced.x.data , finely.spaced.cosine.x )

curve.2 <- ggplot(data = cos.dat) +
  geom_line( mapping = aes(x = finely.spaced.x.data , y = finely.spaced.cosine.x), size = 2, arrow = arrow(), color = "blue")

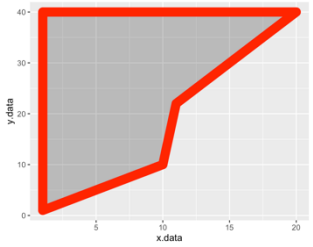
( curve.1 | curve.2 )
```



Polygons

The *geom_polygon( )* function connects the data points in the order that they are given, in a manner similar to *geom_path( )*. In the end, *geom_polygon( )* connects the last data point back to the first, creating a closed shape. The outer path of a polygon can be colored using the **color** aesthetic, similar to *geom_path( )*, but the inner portion can be filled in using the **fill** aesthetic.

```
ggplot(data = dat ) +
  geom_polygon( mapping = aes(x = x.data, y = y.data), color = "red", fill = "black", size = 5, alpha = .25)
```



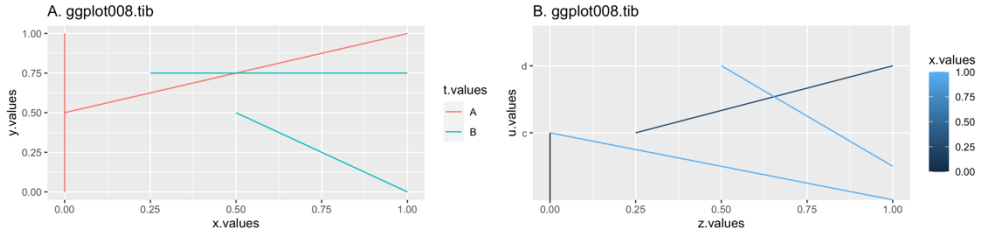
Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

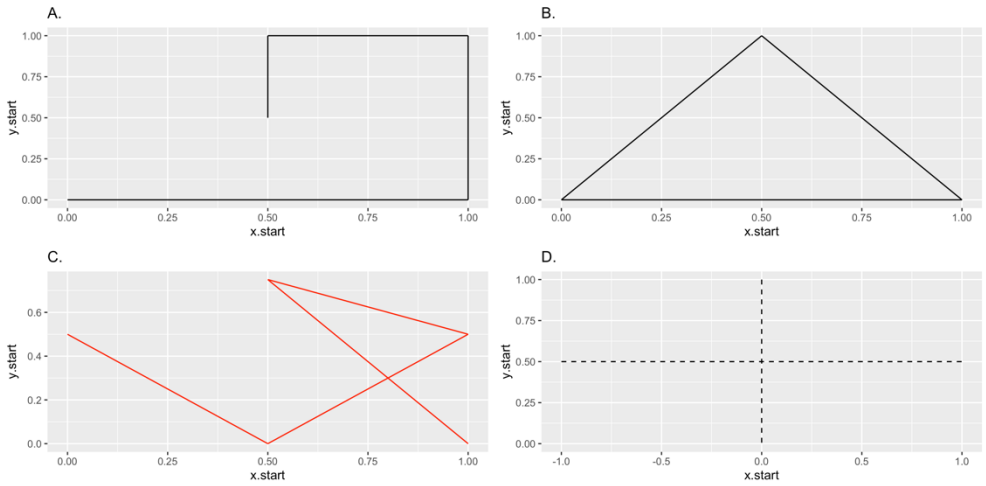
When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

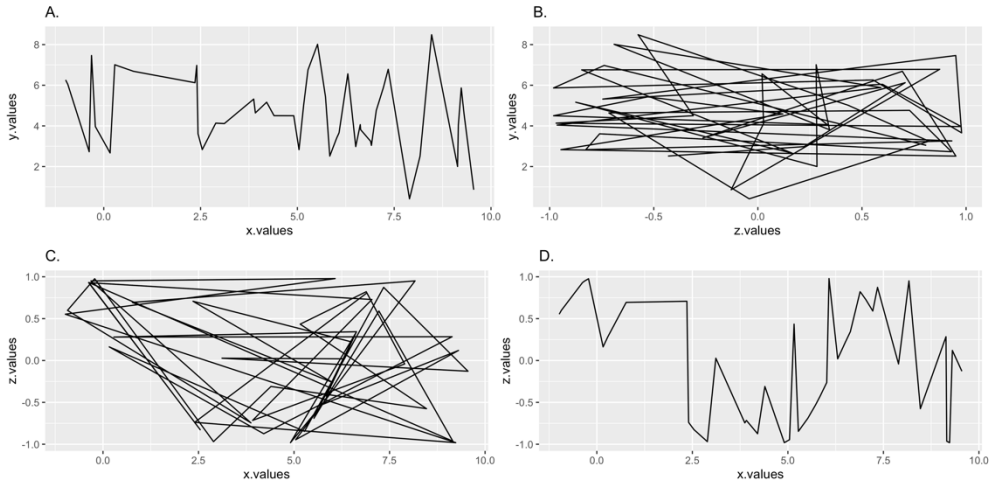
1. Using the tibble named in the title, recreate the following graphs.



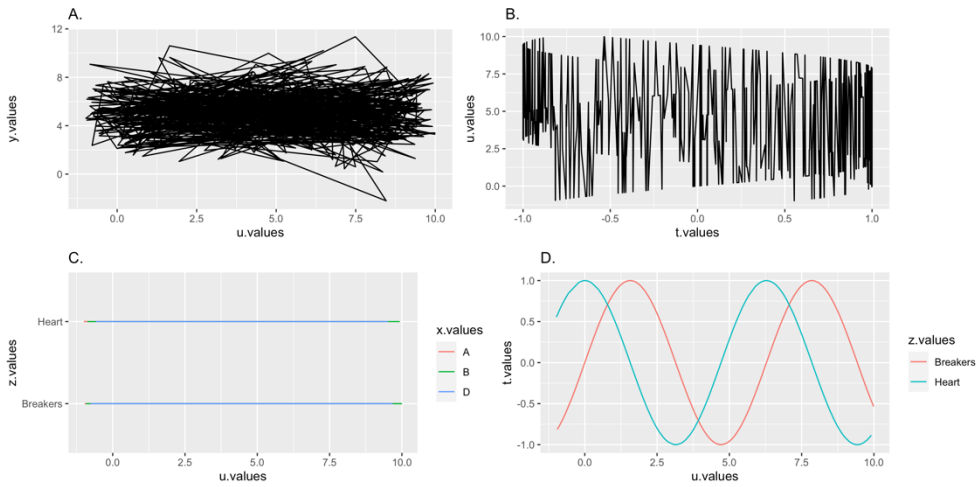
2. Recreate the following graphs: (You may want to create a dataframe/tibble to do so.)



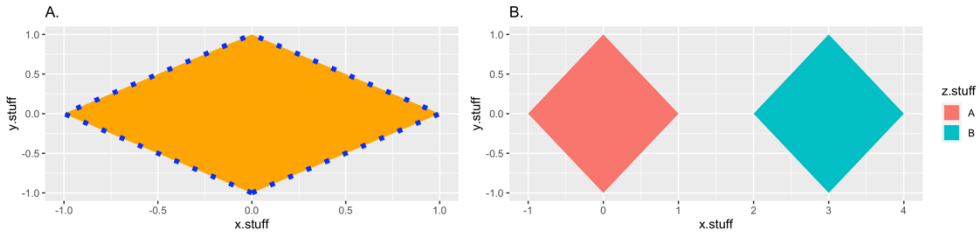
3. Recreate the following graphs using **ggplot009.tib**:



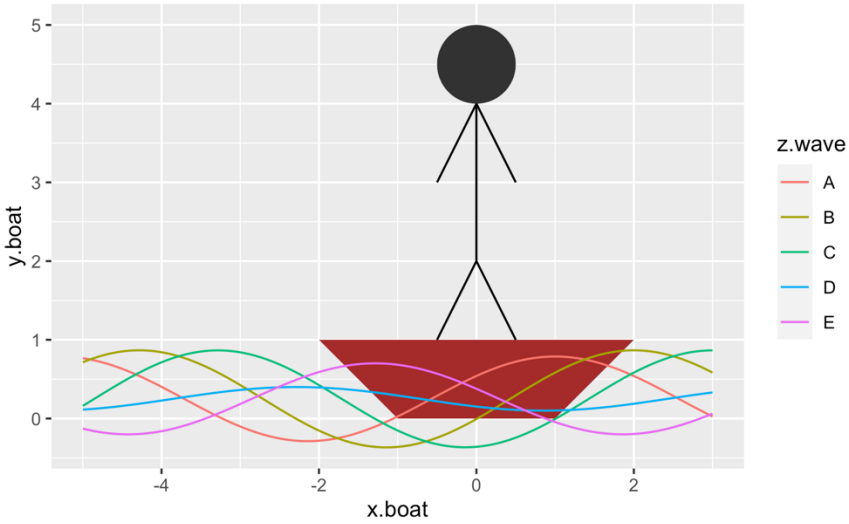
4. Recreate the following graphs using **ggplot010.tib**:



5. Recreate the following graphs:



6. Recreate the following graph: (The exact placement of the waves is not important.)



Chapter 32: ggplot2: One Variable

Several different plots can be made using a single variable of data. To make a 2-dimensional plot a second variable is needed. For the single variable geoms, a statistical function provides the values for the second variable needed. Counts, densities, and quantiles are computed from the single given variable.

Discrete Data

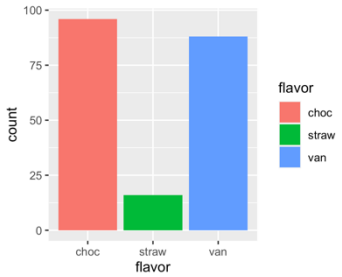
When looking at a single discrete variable, a bar chart can be an appropriate graphic. This example has a random selection of flavors stored in a dataframe called favorites.

```
flavor <-  sample( c( "choc", "van", "straw"), size = 200, replace = T, prob = c(.5, .4, .1))
locale <- rep( c( "dairy barn", "purity"), times = 100 )
favorites <- data.frame( flavor , locale )
head(favorites, n = 5)

  flavor     locale
1   choc dairy barn
2    van     purity
3   choc dairy barn
4    van     purity
5   choc dairy barn
```

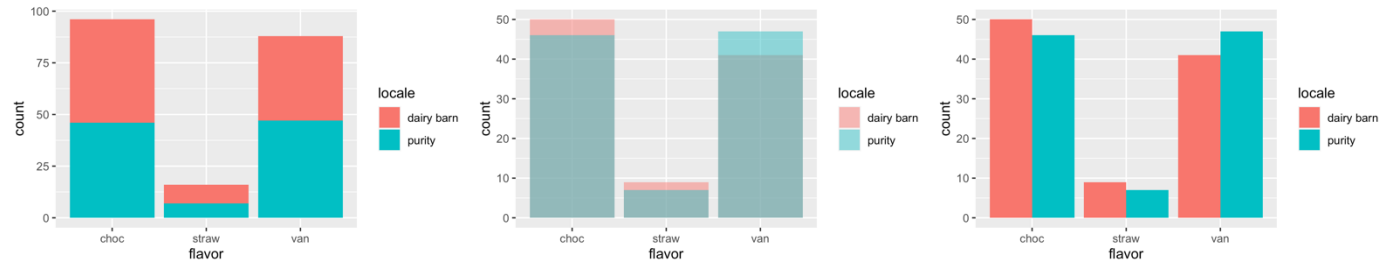
The *geom_bar( )* function creates bar charts. By default, it expects data that still needs to be tallied. For tallied data, both an *x* and *y* aesthetic must be mapped. One to the categories, the other to the frequencies. Additionally, a **stat = "identity"** argument need to be added to *geom_bar( )*.

```
ggplot(data = favorites, mapping = aes( x = flavor, fill = flavor)) +
  geom_bar( )
```



Using **fill** to add **locale**s to the aesthetic mappings, a second variable can be introduced. The display of that information will be controlled by a position argument in the *geom_bar( )* function. **stack** will fill each bar according to each locales contribution. **identity** will make a separate bar for each locale[46]. **dodge** will make a bar for each locale, and place them side by side.

```
ggplot(data = favorites, mapping = aes( x = flavor,  fill = locale)) +
  geom_bar(    position = "stack")
ggplot(data = favorites, mapping = aes( x = flavor,  fill = locale)) +
  geom_bar(   position = "identity", alpha = .5)
ggplot(data = favorites, mapping = aes( x = flavor,  fill = locale)) +
  geom_bar(    position = "dodge")
```



[46] The bars will overlap. The transparency **alpha** was adjusted to make each bar visible

Continuous Data

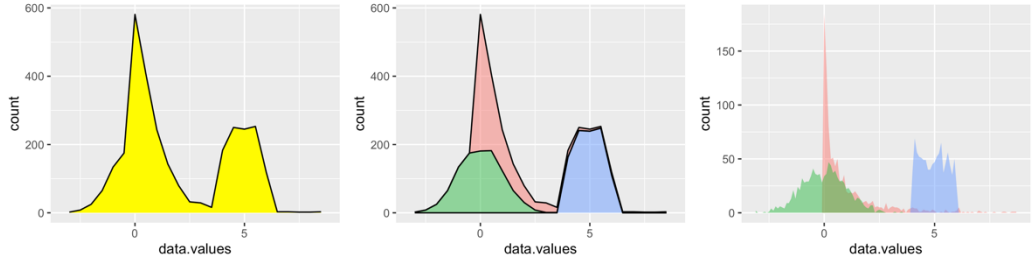
Several different types of graphics can be made from a single continuous variable. The simulated data here comes from three different probability distributions. All three types are stacked in a single column of the dataframe **dat**. A secondary variable indicates the source distribution for each.

```
sample.size <- 1000
x.data <- rnorm(n = sample.size, mean =  0, sd = 1)
y.data <- rchisq(n = sample.size, df = 1)
z.data <- runif(n =  sample.size, min = 4, max = 6)
data.values <- c(x.data, y.data, z.data)
source.distribution <- rep(c("Normal","Chi-Square","Uniform"), each = sample.size)
dat <- data.frame( data.values, source.distribution)
```

The *geom_area*( ) function will create a frequency polygon, and fill in the enclosed regions with a color. If the **fill** option is used as part of the aesthetic mapping, the frequency of each type of data is graphed in the vertical direction. It should be noticed, that the overall shape of the first two graphs is exactly the same. This is because the accumulated frequencies in each group are stacked on top of each other. The **position** argument controls this behavior. Its default setting is "stacked". To compare the individual distributions in the same graphic, the position argument need to change. Setting it to "identity" and reducing the transparency (**alpha**) allows the individual distributions to be seen. The **binwidth** argument controls the size of the data groupings.

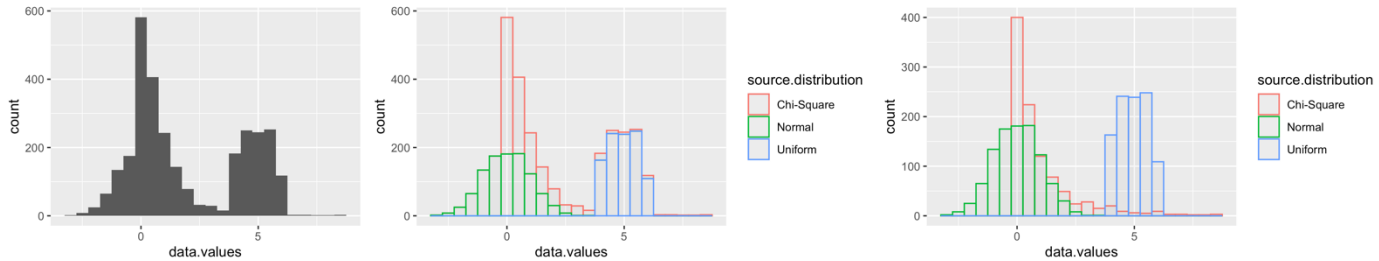
```
ggplot(data = dat) +
  geom_area( mapping =  aes(x = data.values), stat= "bin", binwidth = .5, color = "black", fill = "yellow")
ggplot(data = dat) +
    geom_area( mapping =  aes(x = data.values, fill = source.distribution), stat= "bin", binwidth = .5, alpha = .5, position =
"stack", show.legend = FALSE, color = "black")

ggplot(data = dat) +
    geom_area( mapping =  aes(x = data.values, fill = source.distribution), stat= "bin", binwidth = .1, alpha = .5, position =
"identity", show.legend = FALSE)
```



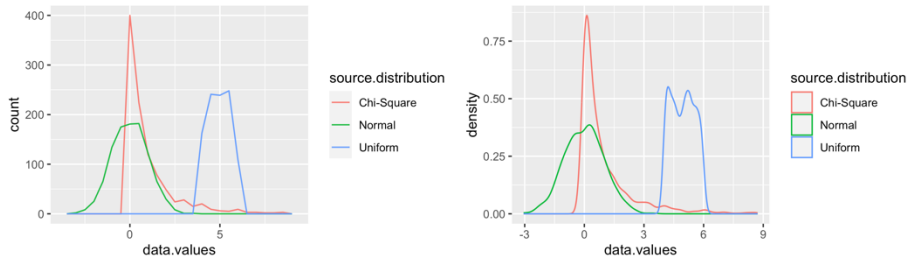
The *geom_histogram*( ) function will create a histogram. The graph on the left treats all the data as one set, regardless of **source.distribution**. The middle graph separated the data by **source.distribution** using the **color** aesthetic (inside an aethetic mapping). However, the **position** aesthetic is set to **stack**. When viewing the graphic, the **color** argument indicate each source distributions contribution to the overall distribution of the data. Setting **position** to **identity** produce a single graph that indicates the distribution of each source. The **binwidth** argument controls the groupings. **alpha** controls transparency. The *geom_freqpoly*( ) function creates a frequency polygon. It has many of the same arguments as *geom_histogram*( ).

```
ggplot(data = dat) +
  geom_histogram( aes(x = data.values),  binwidth = .5)
ggplot(data = dat) +
  geom_histogram( aes(x = data.values, color = source.distribution ), alpha = .05 , binwidth = .5, position = "stack")
ggplot(data = dat) +
  geom_histogram( aes(x = data.values, color = source.distribution ), alpha = .05 , binwidth = .5, position = "identity")
```



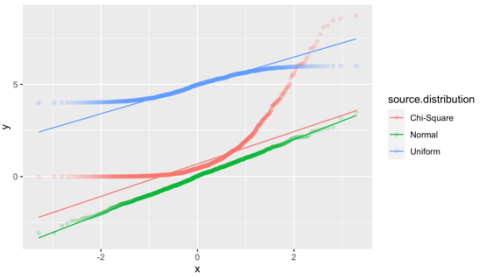
Two other graphics closely related to the histogram are frequency polygons and density plots. Respectively, they are made using the *geom_freqpoly( )* and *geom_density( )* functions.

```
ggplot(data = dat) + geom_freqpoly(aes(x = data.values, color = source.distribution), alpha = 0.95, binwidth = 0.5)
ggplot(data = dat) + geom_density(aes(x = data.values, color = source.distribution), alpha = 0.95)
```



Quantile - Quantile plots are created using the *geom_qq( )* function. *geom_qq_line( )* will add a reference line for a selected distribution. The default is a normal distribution. The **distribution** and **dparams** arguments can be used to look at other distributions. The **distribution** should be set to a quantile function in the **stats** package.

```
ggplot(data = dat, aes(sample = data.values, color = source.distribution )) +
  geom_qq(  alpha = .2 , distribution = stats::qnorm, dparams = list( mean= 0, sd = 1 )) +
  geom_qq_line( distribution = stats::qnorm, dparams = list( mean= 0, sd = 1 ) )
```

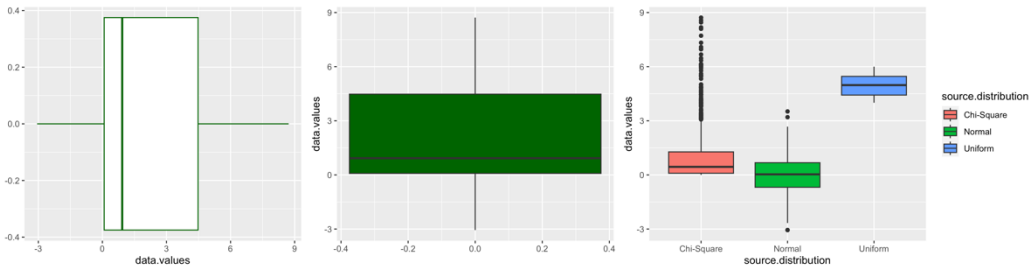


A boxplot is one take on viewing the distribution of a single variable. The *geom_boxplot( )* function will create a boxplot. The direction is determined by the choice of x or y aesthetic. Mapping your main data to the x aesthetic will result in a horizontal boxplot. Verticals result from mapping your main data to y. By mapping a discrete variable to the unused aesthetic(x or y) will create multiple boxplots across the graph.

```
ggplot( data = dat, mapping =  aes( x = data.values)) +
  geom_boxplot( color = "darkgreen")

ggplot( data = dat, mapping =  aes( y = data.values)) +
  geom_boxplot( fill = "darkgreen")

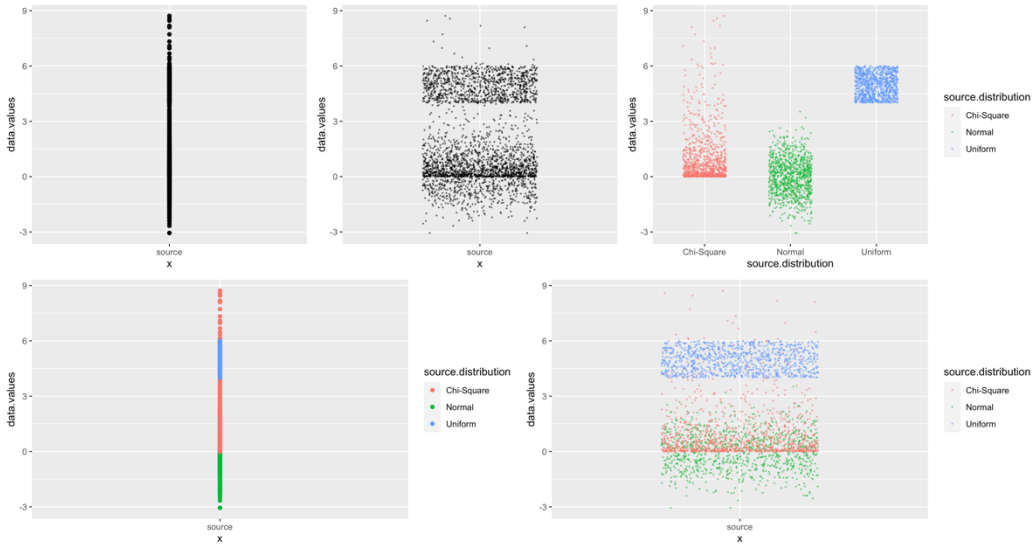
ggplot( data = dat, mapping =  aes(x = source.distribution , y = data.values)) +
  geom_boxplot(  mapping = aes( fill = source.distribution ))
```



With a single continuous variable of interest, the data values could be plotted at corresponding locations on a number line. However, it the data set was large or there were (nearly) repeated values, the distribution of the values may be lost do the printing limitations.

For the first two graphs, two geoms are used that are meant for two variable data sets. They need both and x and y aesthetic. For the immediate purposes, one of these aesthetics x is set to a single value. Effectively, this sets the horizontal coordinate for all data points to the same value. Looking at the first graph, using *geom_point( )* the data values are plotted, but the vast majority of plotted points overlap. This makes it difficult to identify the concentration of points. The second graphic uses *geom_jitter( )*. This functions adds a small random amount to each data value. The **width** argument limits the horizontal spread around the **source** vertical axis. Had **source** been mapped to a horizonta axis, the spread would be controlled by a **height** argument. Without the color aesthetic in the second graphic, all points would have the same color. The third plot maps the x aesthetic to an actual data variable. This splits the data set into three groups.

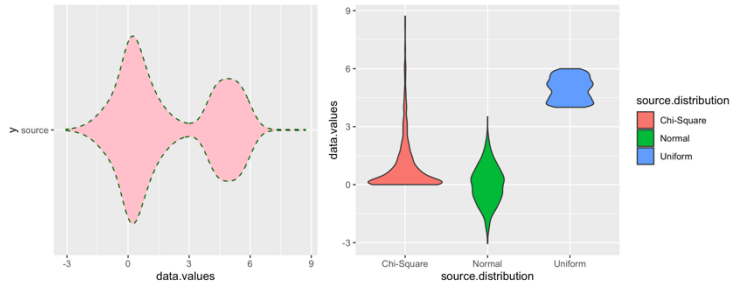
```
# First Row
ggplot( data = dat, mapping =  aes(x="source" , y = data.values)) +
  geom_point( )
ggplot( data = dat, mapping =  aes(x= "source" , y = data.values)) +
  geom_jitter(  width =.25, size = .25, alpha = .5)
ggplot( data = dat, mapping =  aes(x= "source" , y = data.values)) +
  geom_jitter(  aes( color = source.distribution),width =.25, size = .25, alpha = .5)
# Second Row
ggplot( data = dat, mapping =  aes(x="source" , y = data.values)) +
  geom_point( aes(color = source.distribution ))
ggplot( data = dat, mapping =  aes(x = source.distribution , y = data.values)) +
  geom_jitter( aes( color = source.distribution ), width =.25, size = .25, alpha = .5)
```



A violin plot takes a density plot and reflects it over a line. It is meant to give an indication of a single variables distribution in much the same way as a boxplot or the jitted plot above. A violin plot is made with the *geom_violin( )* function.

```
ggplot( data = dat, mapping =  aes(y = "source" , x = data.values)) +
  geom_violin( color = "darkgreen", fill = "pink", linetype = 2 )

ggplot( data = dat, mapping =  aes(x = source.distribution , y = data.values)) +
  geom_violin( aes( fill = source.distribution ))
```



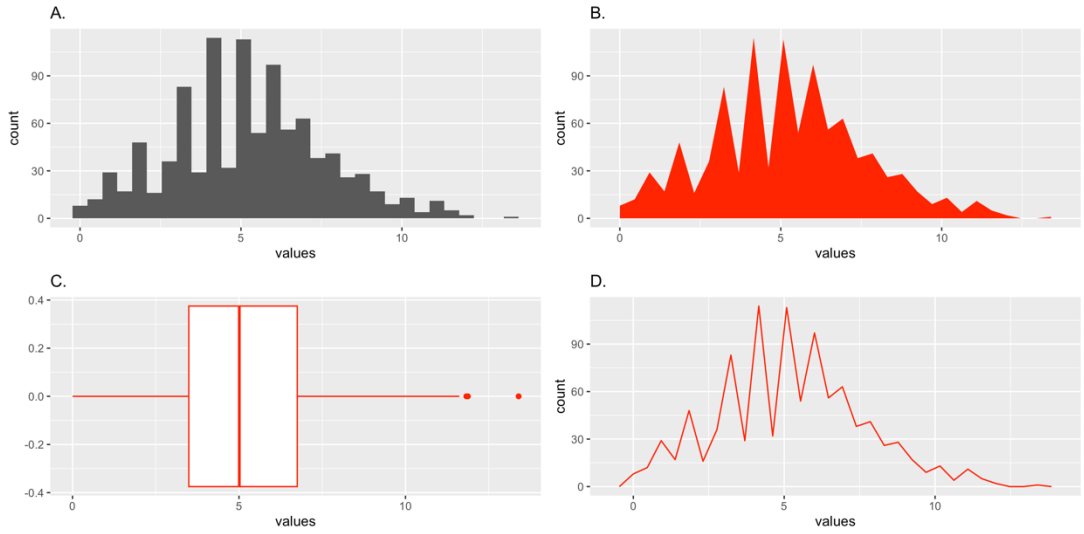
Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

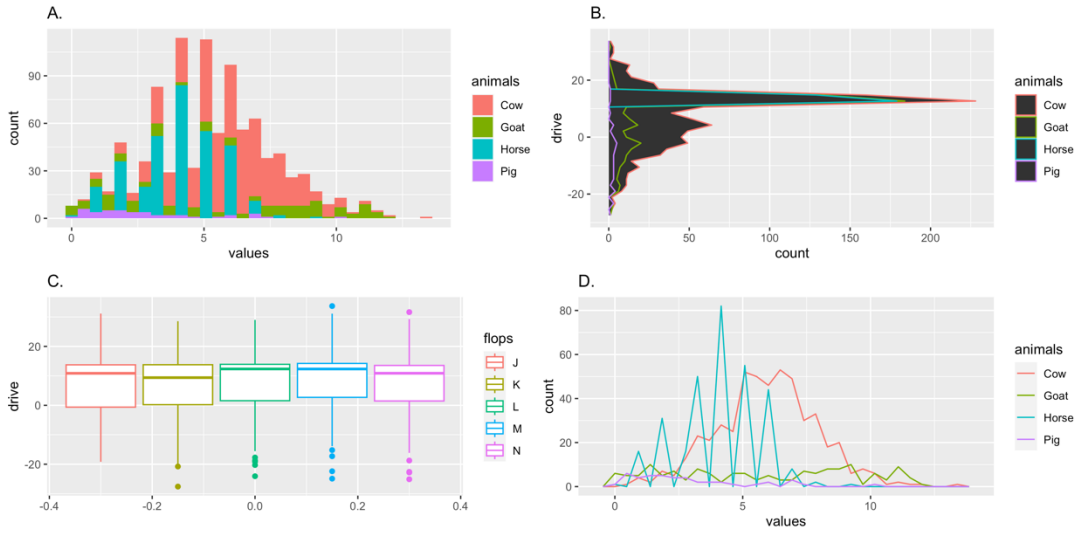
When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

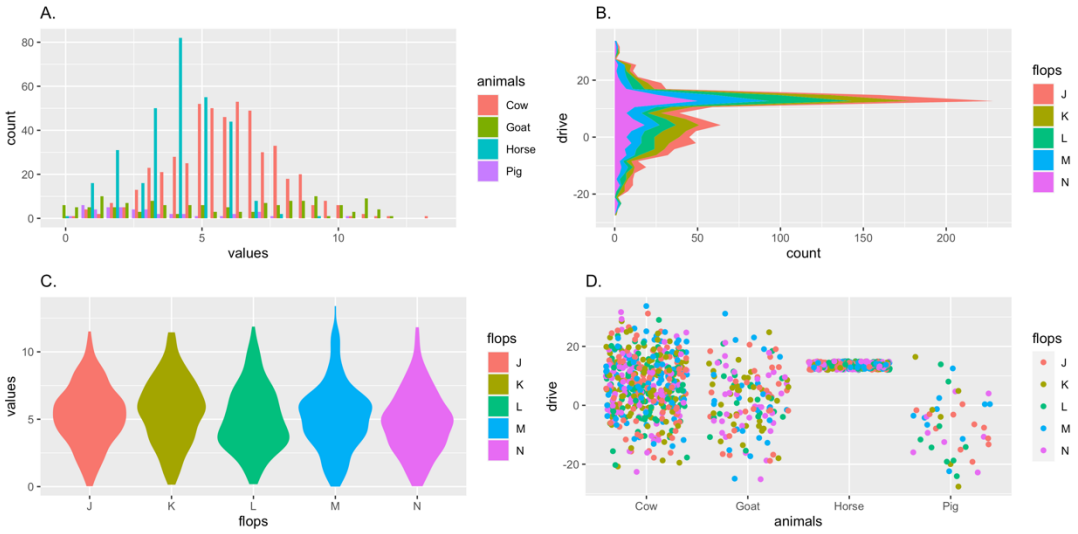
1. Use **oneVariable001.dat** to recreate the following graphs.



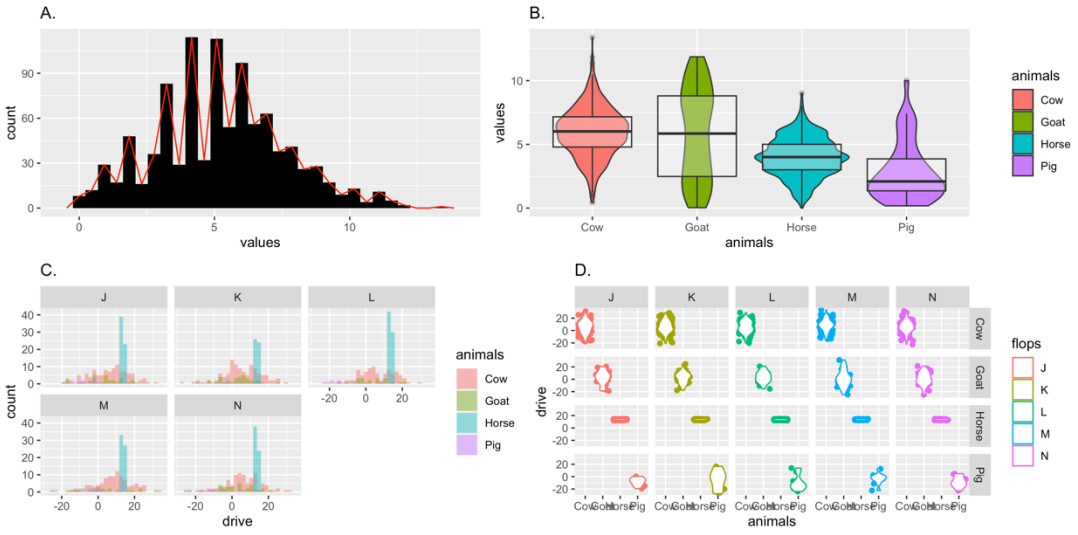
2. Use **oneVariable001.dat** to recreate the following graphs.



3. Use **oneVariable001.dat** to recreate the following graphs.



4. Use **oneVariable001.dat** to recreate the following graphs.



Chapter 33: ggplot2: Two Variables

When looking at at least two numerical variables, several options are available for augmenting a scatterplot. The simulated dataset contains four variables. Two are meant to illustrate a string linear relationship. The third, **dist.from.origin**, is an measure of a points distance from the origin. THe final variable is used to break the data points into groups.

```
x.data <- c(1:10, 3:7, 4:6)
y.data <- 2 * x.data
dist.from.origin <- sqrt(x.data^2 + y.data^2)
dat <- data.frame(x.data, y.data, dist.from.origin, AB = rep(c("A", "B"), times = 9))
head(dat, n = 5)

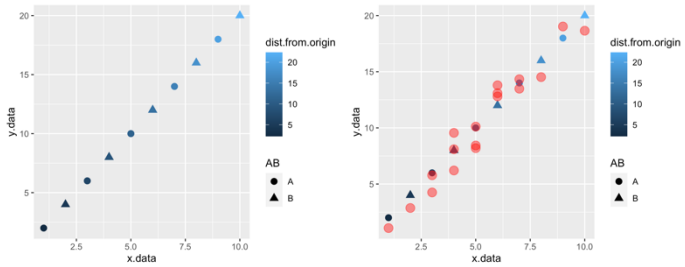
  x.data y.data dist.from.origin AB
1      1      2         2.236068  A
2      2      4         4.472136  B
3      3      6         6.708204  A
4      4      8         8.944272  B
5      5     10        11.180340  A
```

The first modification of a scatterplot come from using the *geom_jitter*( ) function. The funcion will add a small random shift to each data point. The can be handing when there multiple observations of the same point. *geom_jitter*( ) can make the multiplicity of points visible. The **width** and **height** arguments put limits on the range of the jitter.

In the simulated data, the points with x-coordinate four, five and six, are repeated three times. In the graph on the left, there is no indication of this. The graph on the right uses *geom_jitter*( ) makes this repetition visible.

```
ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_point( mapping = aes( color = dist.from.origin, shape = AB  ), size = 4 )

ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_jitter(color = "red", width = 0, height = 1, size = 4, alpha = .5)
```

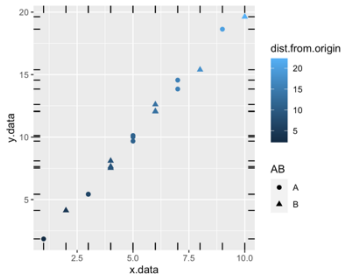


As constructed, the simulate data set would not illustrate the rest of these scatterplot modifications very well. A small amount of randomness will be added to **y.data**.

```
dat$y.data <- y.data + rnorm(18, mean = 0, sd = 0.5)
```

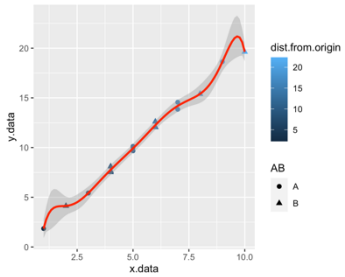
The *geom_rug*( ) function will adds a set of ticks to each margin of the scatterplot. These are meant to give an indication of the marginal distribution of each variable. The **sides** argument controls where the ticks appear. Either on the **l**eft, **r**ight, **t**op, or **b**ottom of the graph.

```
ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_point( mapping = aes( color = dist.from.origin, shape = AB  ), size = 2 ) +
  geom_rug( sides="bltr" )
```



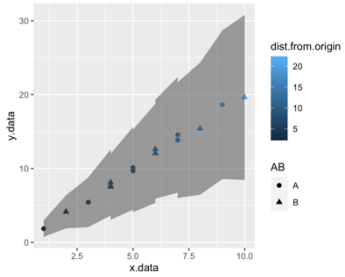
The *geom_smooth*( ) function will add a smooth fitted curve. Several **method**s for producing the smoothed curve are available: "lm", "glm", "gam", "loess". In this example, an eight degree polynomial was fit. Additionally, confidence intervals can be added using the **se** argument. The confidence level is set using the **level** argument.

```
ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_point( mapping = aes( color = dist.from.origin, shape = AB  ), size = 2 ) +
  geom_smooth( method = "lm", formula = y ~ poly(x, 8), se = TRUE, color = "red" , level = .99)
```



The *geom_ribbon*( ) function adds an interval around each point. Then ends of adjacent intervals are then linearly connected to create the ribbon. The width of the ribbon is determined by the **ymin** and **ymax** arguments. The **xmin** and **xmax** arguments can be used to control the ribon in the horizontal direction. In this example, the width is dependent on the distance from the origin. Larger distances translate into a wider ribbon.

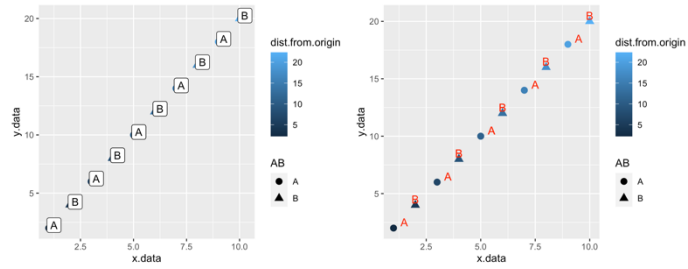
```
ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_point( mapping = aes( color = dist.from.origin, shape = AB  ), size = 2 ) +
  geom_ribbon(mapping = aes( ymin = y.data-dist.from.origin/2, ymax = y.data+ dist.from.origin/2  ), alpha =.5)
```



Individual points can be labeled using either the *geom_label*( ) or the *geom_text*( ) functions. There are two arguments to nudge the labels position: **nudge_x** and **nudge_y**. *geom_label*( ) draws a rectangle around the label to make it easier to read. These functions do not need to label every data point. If provided with a different dataset, a different set of data points, or only a select few would be labelled.

```
ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_point( mapping = aes( color = dist.from.origin, shape = AB  ), size = 2 ) +
  geom_label(mapping = aes( label = AB), nudge_x = .25,  nudge_y = .25 )

ggplot(data = dat, mapping = aes(x = x.data, y = y.data) ) +
  geom_point( mapping = aes( color = dist.from.origin, shape = AB  ), size = 2 ) +
  geom_text(mapping = aes( label = AB), color = "red", nudge_x = c(.5,0),  nudge_y = c(.5,.5), size = 4 )
```



Exercises

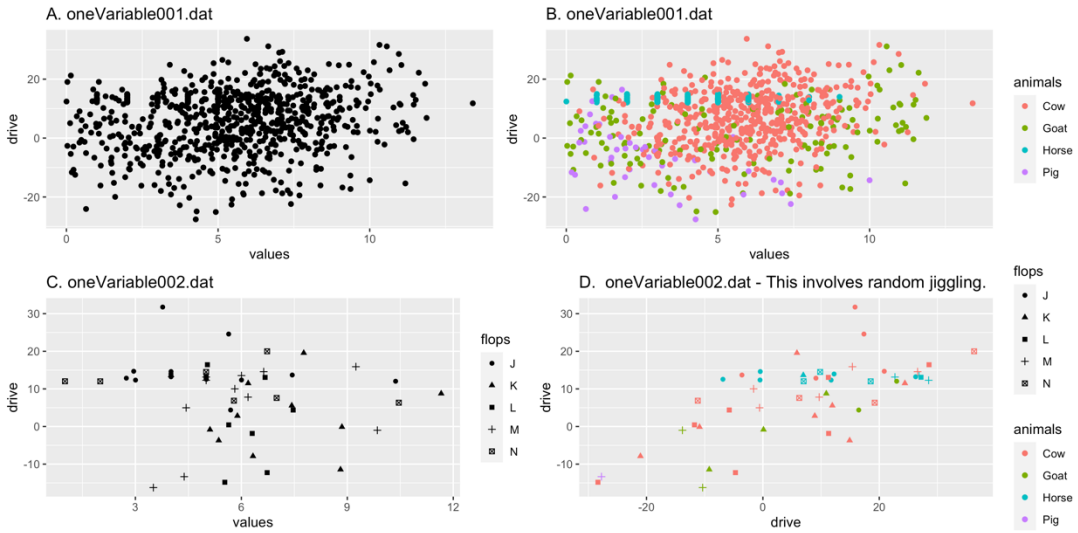
The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

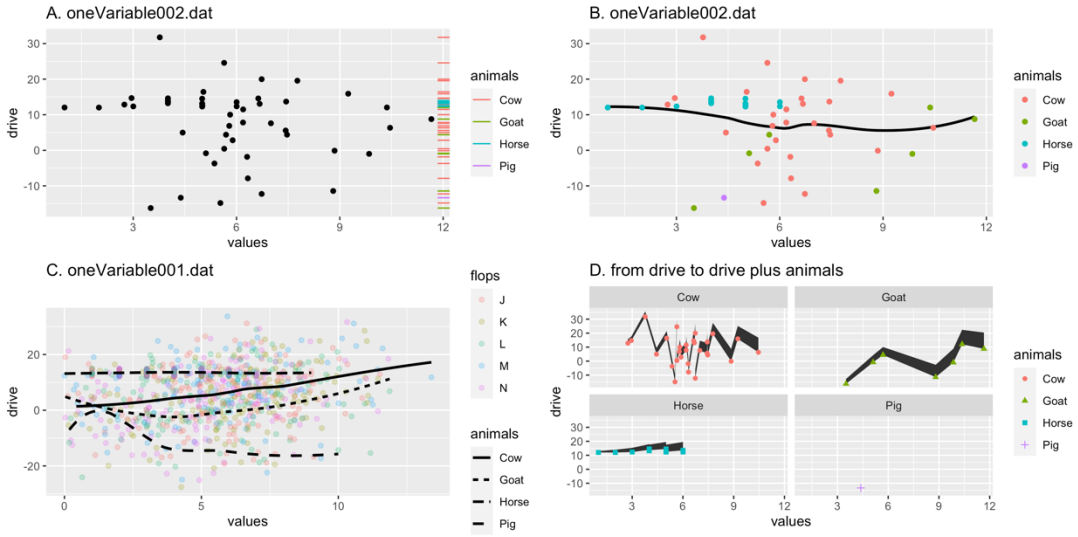
Unless directed otherwise, a result should always be displayed.

Only the "lm" and "loess" methods will be used. The level will always be set to .95. The transparancey will either be 0.25, .5, or 1.

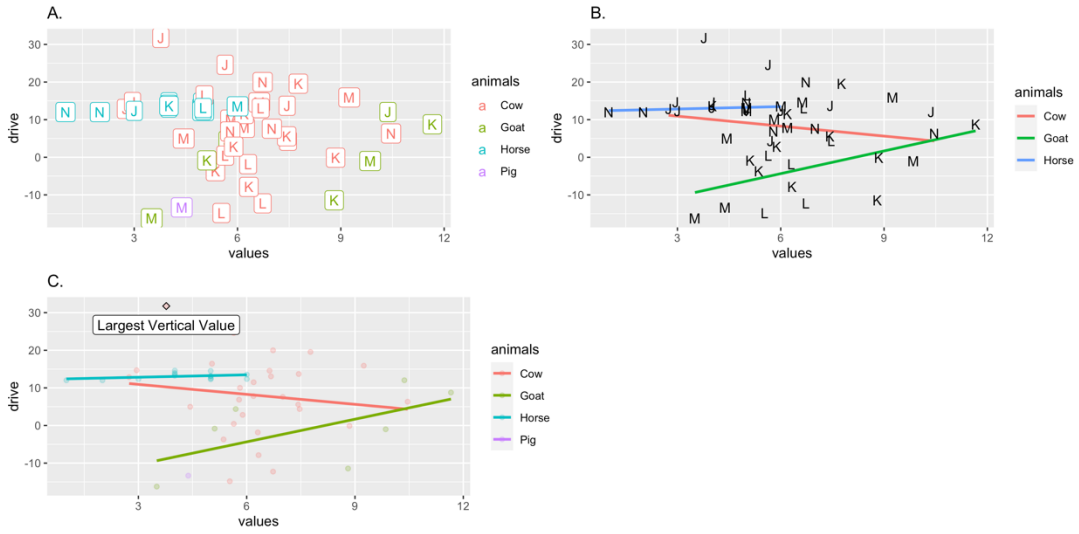
1. Use the dataframe listed in the title to recreate the following graphs.



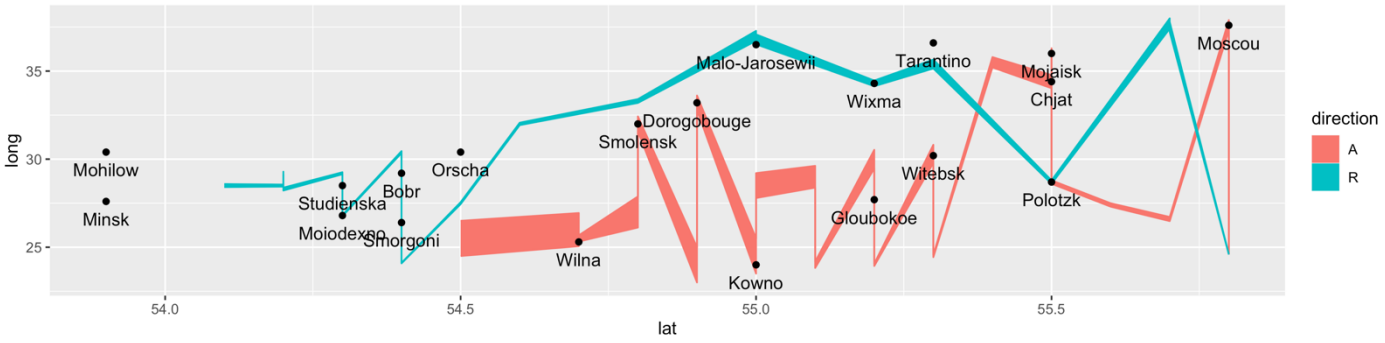
2. Unless otherwise indicated in the title, use the dataframe oneVariable002.dat to recreate the following graphs.



3. Unless otherwise indicated in the title, use the dataframe oneVariable002.dat to recreate the following graphs.



4. Use the Minard dataframes in the HistData package to recreate the following graph. Scale the width of the ribbon to be the proportion of survivors over the maximum number of survivors.



Chapter 34: ggplot2 - Discrete Scales

Scale functions are used to control the mapping of data values to aesthetics. If a scale function is not specified, a default will be used. However, using a scale function, or more than one, overrides the defaults.

Scale functions are named using the following convention: scale_**(aesthetic parameter)**_**(type of scale/mapping)**. The **(aesthetic parameter)** is the aesthetic that is being remapped: x, color, fill, linetype, size, fill, etc. **(type of scale/mapping)** gives an indication of the type of scale being used or the mapping strategy being used.

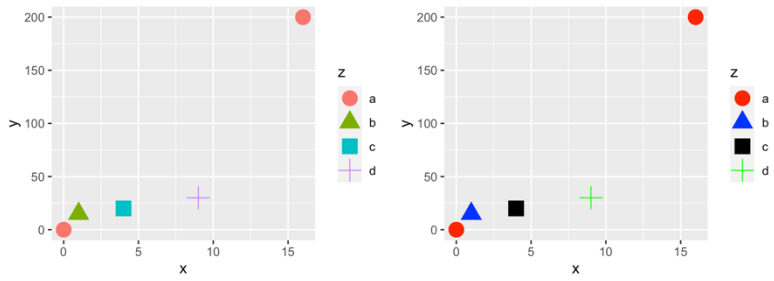
Aesthetics for Discrete Data

To begin, scale functions that affect discrete data will be looked at. These functions work in a similar fashion and have similar arguments. As a first example, a data set is needed. This contains two numerical vector and one character vector. The character vector will be the discrete component of this data set that is used.

```
x <- c(0, 1, 4, 9, 16)
y <- c(0, 15, 20, 30, 200)
z <- c(letters[1:4], "a")
dat <- data.frame(x, y, z)
```

Two scatterplots will be created. The first does not explicitly use a scale function. The default mappings for color and shape will be used. The other will use a scale function to affect the color of the plotted points.

```
p1 <- ggplot(data = dat) + geom_point(mapping = aes( x = x, y = y, color = z, shape = z), size =5)
p2 <- p1 + scale_color_manual( breaks = c("a", "b", "c", "d"), values = c("red","blue", "black", "green"))
(p1|p2)
```



Color

The *scale_color_manual*( ) function will be used to do this. As the name suggests, the aesthetic being manipulated is **color**. The mapping will be a manual one. The mapping between colors and discrete data values will manually be entered. Using the **breaks** argument, discrete data levels are listed. The **values** argument lists the corresponding colors that the **breaks** should be mapped to. In this case, "a" is mapped to "red", "b" is mapped to "blue", "c" is mapped to "black", and "d" is mapped to "green"[47].

Shape[48]

The *scale_shape_manual*( ) function can be used to create a manual mapping between the data and the shape that is used to display it. The **breaks** and **values** arguments will be used to do this in the same way it was done with the color aesthetic.

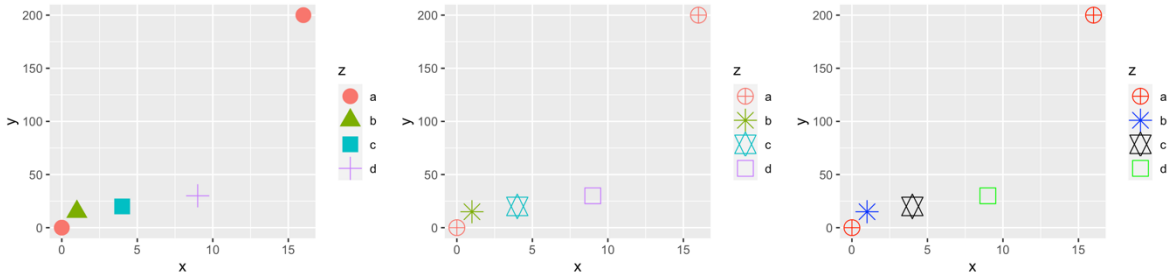
The original (default) graph is on the left. The middle graph remaps the shape aesthetic using *scale_shape_manual*( ). The third graph demonstrates that **scale** functions can be used together. Both the shape and color aesthetics have had new maps used. Both make use of the **breaks** and **values** arguments.

```
p3 <- p1 + scale_shape_manual(  breaks = c("a","b", "c", "d"),  values = c(10, 8, 11,0) )

p4 <- p1 + scale_shape_manual(  breaks = c("a","b", "c", "d"),  values = c(10, 8, 11,0) ) +
           scale_color_manual( breaks = c("a", "b", "c", "d"), values = c("red","blue", "black", "green"))
(p1|p3|p4)
```

---

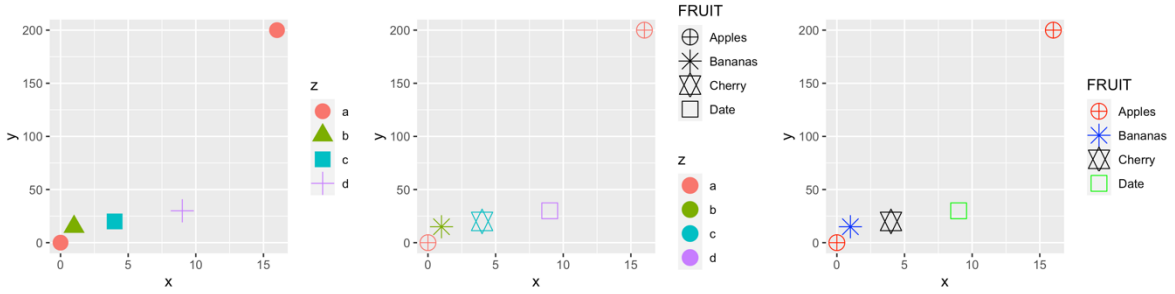
[47] The **breaks** argument is not required. If it had been left out, a listing of unique values in the vector *z* would have been determined and used.
[48] Other aesthetics have similar functions. These include alpha, fill, linetype, linewidth, and size.



Remapping Labels and Multiple Scales

Other arguments are available to remap data. Specifically, these can be used to clean up the appearance of raw data values that may not be aesthetically pleasing or informative to look at. The **labels** argument can be used to map the data values to new labels that will appear in the legend. The **name** argument will map the variable name represented in the legend to a more informative one. Again, for comparison the original "default" graph is displayed on the left.

```
p4 <- p1 + scale_shape_manual( name = "FRUIT", breaks = c("a","b", "c", "d"),
                               values = c(10, 8, 11,0), labels = c("Apples", "Bananas", "Cherry", "Date") )
p5 <- p4 + scale_color_manual( name = "FRUIT", breaks = c("a","b", "c", "d"),
                               values = c("red","blue", "black", "green"), labels = c("Apples", "Bananas", "Cherry", "Date"))
(p1|p4|p5)
```

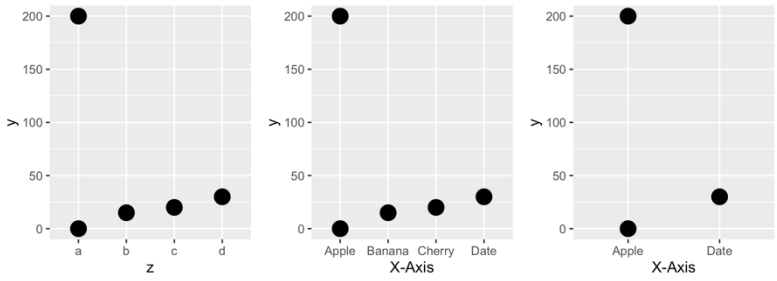


The middle graph uses both the **name** and **labels** argument. The name of the variable has been mapped to "FRUIT". Data values ("a", "b", "c", "d") are mapped to new label ("Apples", "Bananas", "Cherry", "Date"). The middle graph has two legends because the new mapping only applies to the shape aesthetic. Hence, two legends are needed. The third graph adds these mappings to the color aesthetic. Now, the two legends to be merged.

Horizontal & Vertical Axes

The *scale_x_discrete*( ) (*scale_y_discrete*( )) function sets a mapping of discrete data to the horizontal(vertical) axis of a graph.

```
p6 <- ggplot(data = dat, mapping = aes(x = z, y = y)) +  geom_point( size = 5)
p7 <- p6 + scale_x_discrete( name = "X-Axis", breaks = c("a", "b", "c", "d"), labels = c("Apple", "Banana", "Cherry", "Date") )
p8 <- p6 + scale_x_discrete( name = "X-Axis", labels = c("Apple", "Date"), limits = c("a", "d"))
(p6|p7|p8)
```



The **breaks**, **name**, and **labels** arguments play the same role as before on the axis. The **values** argument has no role here. The **limits** argument is used to restrict which values are displayed when the default **breaks** value is used[49].

[49] **limits** is an argument for the other scales as well

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

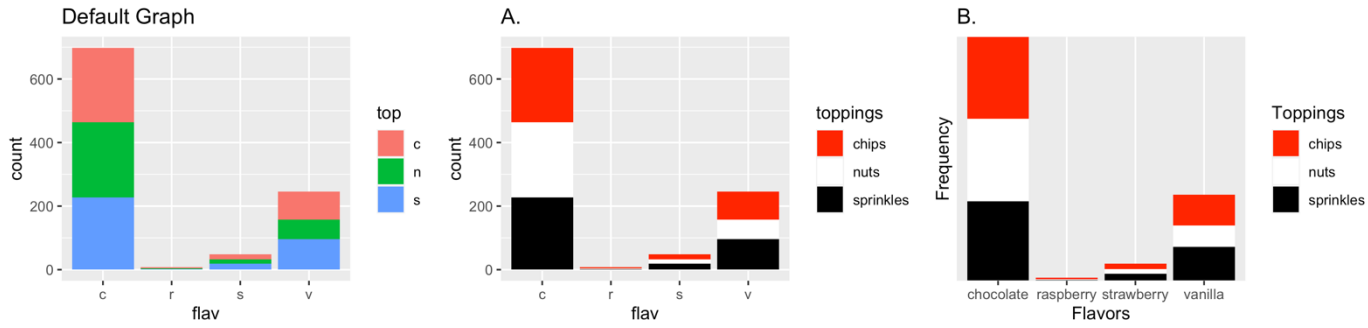
When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

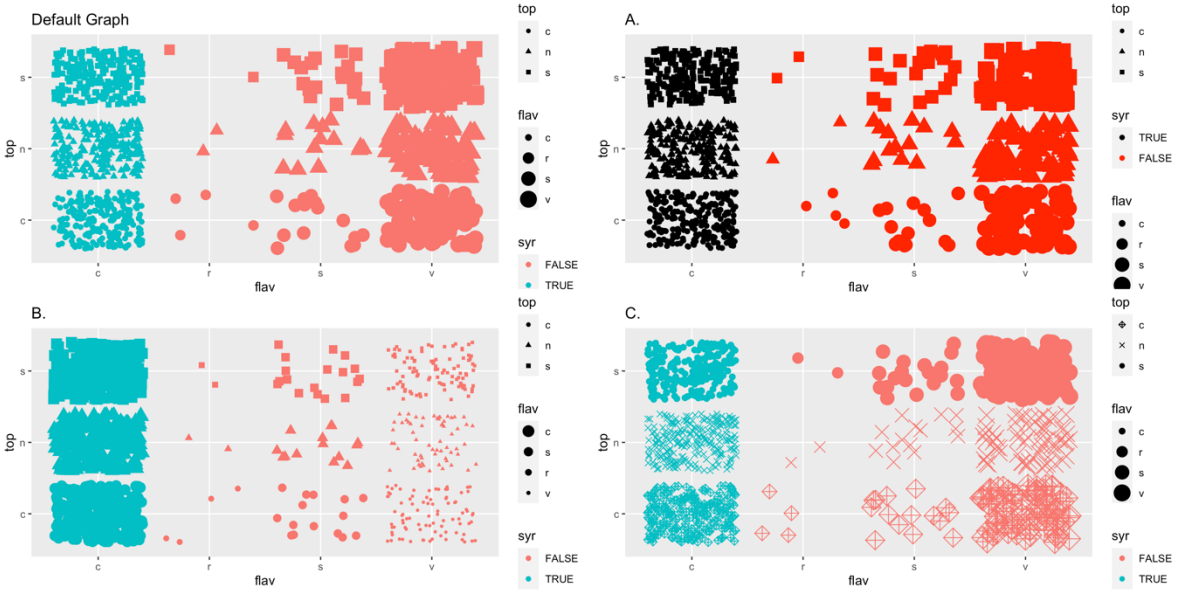
The variables in the given data set are **top**pings, **flav**or, **syr**up, **cal**ories, and weight. The levels for most of these variables are recorded using thier first letter as follows:

flavor	**toppings**	**syrup**
chocolate	**c**hips	TRUE
vanilla	**n**uts	FALSE
strawberry	**s**prinkles	
raspberry		

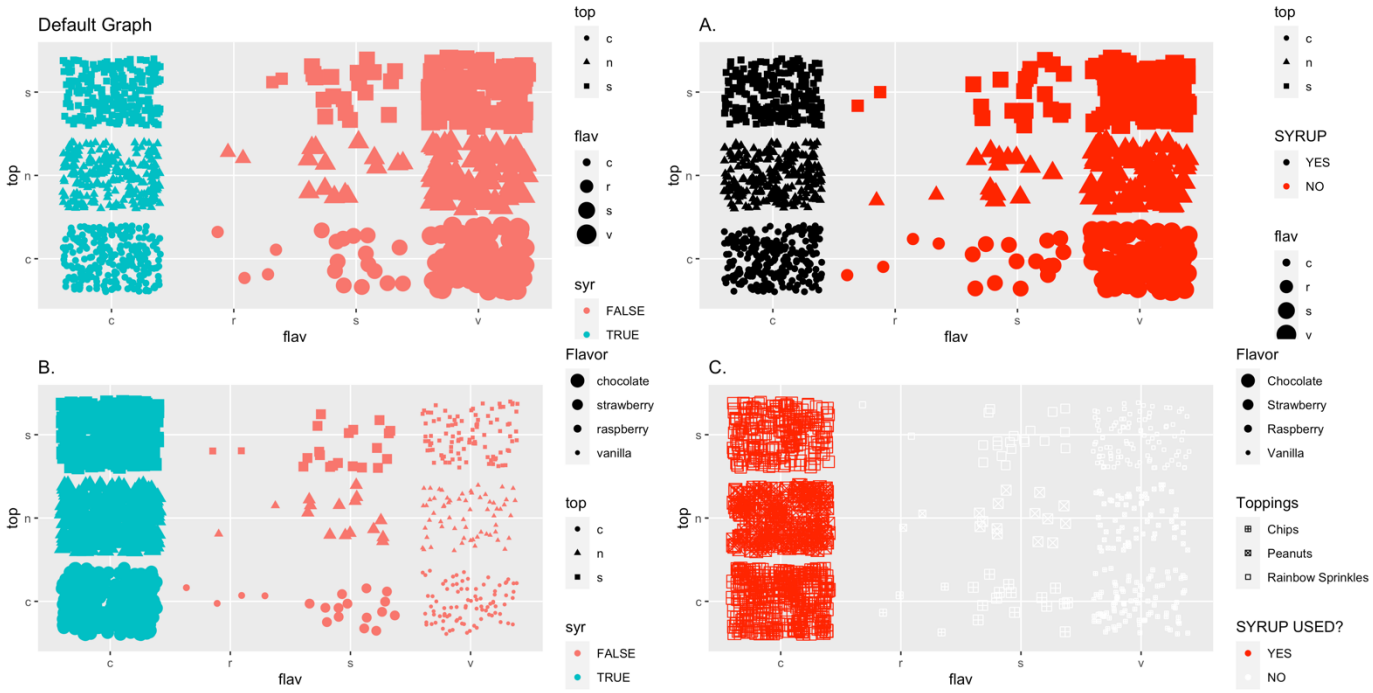
1. Using the graph **scalesGraph000**, recreate the following graphs.



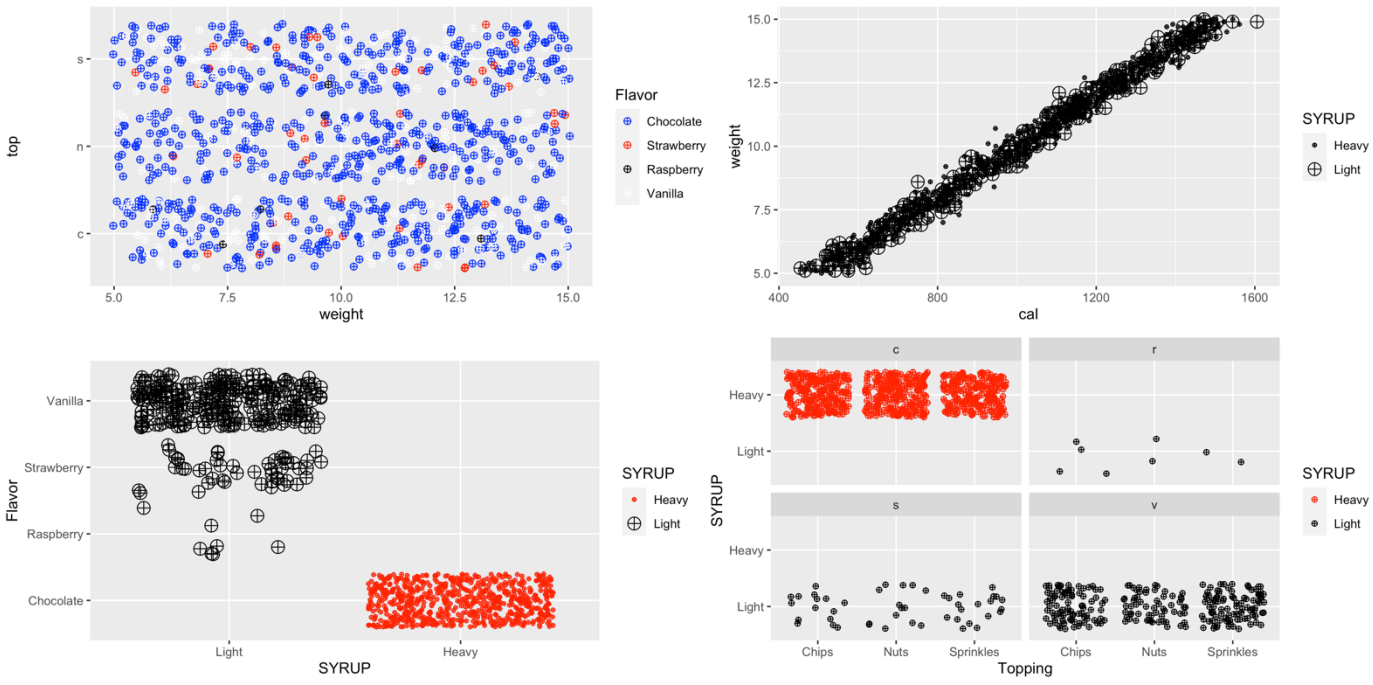
2. Using the graph **scalesGraph001**, recreate the following graphs. Some randomization is involved in the default graph. You should not try to match exact point positions, just point characteristics.



3. Using the graph **scalesGraph001**, recreate the following graphs. Some randomization is involved in the default graph. You should not try to match exact point positions, just point characteristics.



4. Using the dataframe **scaled002.dat**, recreate the following graphs. Some randomization will be needed in the placement of the points. You should not try to match exact point positions, just point characteristics.



Chapter 35: ggplot2 - Continuous Scales

Aesthetics for Continuous Data

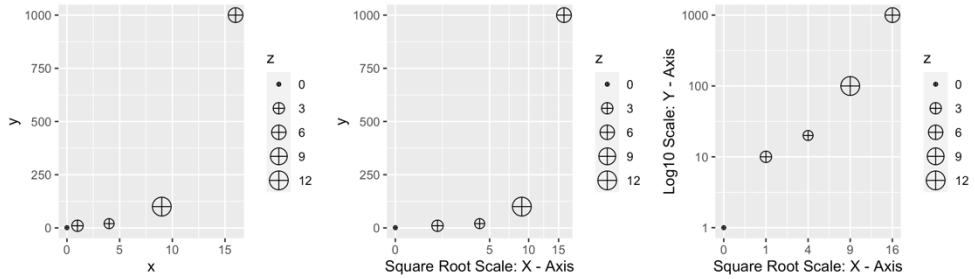
Scale functions are used to control the mapping of data values to aesthetics. When the data being used is continuous, the scale functions can't explicitly list out the mappings being used as was done with discrete data. Scale functions for continuous data work in a similar fashion and have similar arguments. As a first example, a data set is needed. This contains three numerical vectors.

```
x <- c(0, 1, 4, 9, 16)
y <- c(1, 10, 20, 100, 1000)
z <- c(0, 3, 2, 12, 6.3)
dat <- data.frame(x, y, z)
```

Horizontal & Vertical Axes

Mapping continuous data to the horizontal(vertical) axis makes use of one of the $scale\_c\_(typeof scale/mapping)(\ \ )$ functions. This first graph shows a scatter plot of this data with the default "identity" scaling. Applying the $scale_{x_s} qrt(\ \ )$ in the second graph, and the x values from the data set are mapped to their square roots of the x aesthetic(x-axis) of the graph. The x values 0, 1, 4, 9, and 16 are mapped to a distance equal to their square root on the x-axis.

```
p1 <- ggplot(data = dat, mapping = aes( x = x,  y = y, size = z)) + geom_point( shape = 10)
p2 <- p1 + scale_x_sqrt( name = "Square Root Scale: X - Axis")
p3 <- p1 + scale_x_sqrt( name = "Square Root Scale: X - Axis", breaks = c(0, 1, 4, 9, 16)) +
          scale_y_log10( name = "Log10 Scale: Y - Axis")
(p1|p2|p3)
```

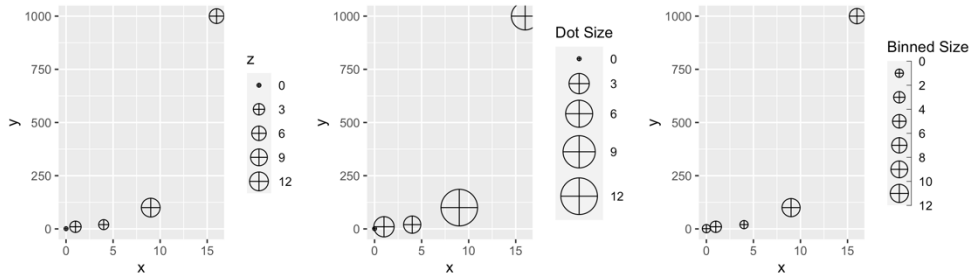


The third graph displayed here uses $scale\_y\_log10(\ \ )$ to map the y values of the dataframe to a distance equal to their log(base-10) on the y-axis. Additionally, the **breaks** is added to the $scale\_x\_sqrt(\ \ )$ function to indicate which tick marks should be displayed. Other scalings can be made to the x and y aesthetics. This is done by using the $scale\_x\_continuous(\ \ )$ and/or $scale\_y\_continuous(\ \ )$ functions. With these scale functions, the **trans** argument needs to be set to the desired scaling[50].

Size

The **size** aesthetic can be mapped few different ways. The first graph produced indicates the default mapping of a continuous variable to point sizes.

```
p4 <- p1 + scale_size(name = "Dot Size", range = c(1, 12) )
p5 <- p1 + scale_size_binned(name = "Binned Size", breaks = seq(from = 0, to = 12, by = 2))
(p1|p4|p5)
```



[50] A listing of **trans** values can be found in the help documentation. Additionally, custom scales can be created by way of the $trans_n ew(\ \ )$ function.

The following graphs both use the **name** argument in their scale functions to set legend name. The *scale_size*( ) function is used in the second graph to set a **range** of radii. The smallest value is mapped to the lower end of the range. The largest value to the upper end of the range. Other values are interpolated throughout that range. The *scale_size_binned*( ) functions by binning the data values. The **breaks** argument is used to set the endpoints for the bins. This sets a finite set of radii in the second graph. Theoretically, *scale_size*( ) could produce an infinite number of sizes.

Shape

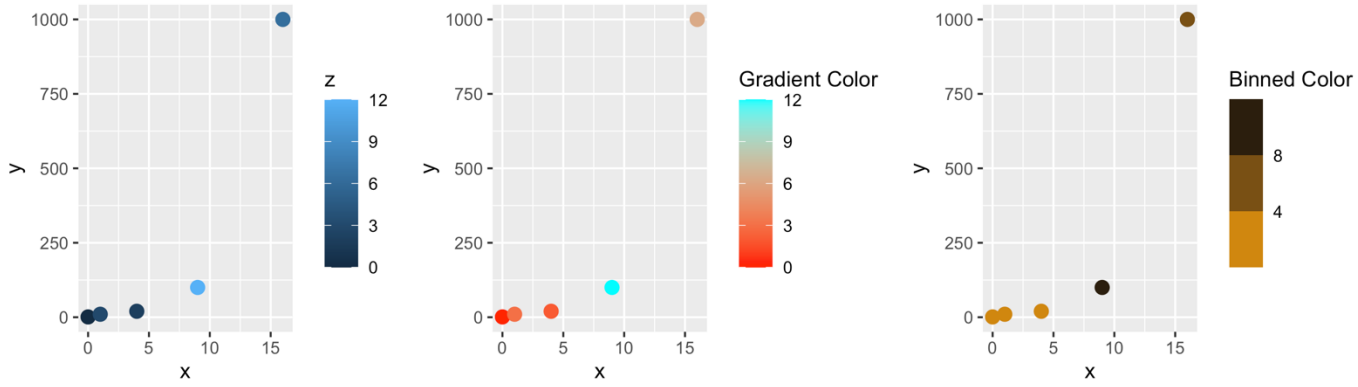
A continuous variable can not be mapped directly to the shape aesthetic. Theoretically, a mapping of an infinite number of values to a finite number of shapes would need to be made. However, the *scale_shape_binned*( ) function can group values from a continuous variable into bins and then provide mapping to the shape aesthetic. It will work in a manner similar to *scale_size_binned*( ).

Color[51]

```
p4 <- ggplot(data = dat, mapping = aes( x = x,  y = y,  color = z)) + geom_point( size = 3)

p5 <- p4 + scale_color_gradient(name = "Gradient Color", low = "red", high = "cyan",
  space = "Lab")
p6 <- p4 + scale_color_steps(name = "Binned Color", n.breaks = 4,low = "orange",high = "black")

(p4|p5|p6)
```



The first graph indicates the default coloring scale. Behind the scenes, it is making use of *scale_color_continuous*( ) whose default **type** argument is "gradient". This **type** has been changed in the second graph. The gradient of colors can be customized using *scale_color_gradient*( ). It require two color arguments to be set: **low** and **high**. The scale of colors will then run from one to the other. *scale_color_step*( ) is used in the fourth graph. **n.breaks** creates bins using the indicated number of endpoints. In this case, three bins are created and mapped to three colors. Its two color arguments are similar to those of *scale_color_gradient*( ).

[51] The fill aesthetic has similar functions.

Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

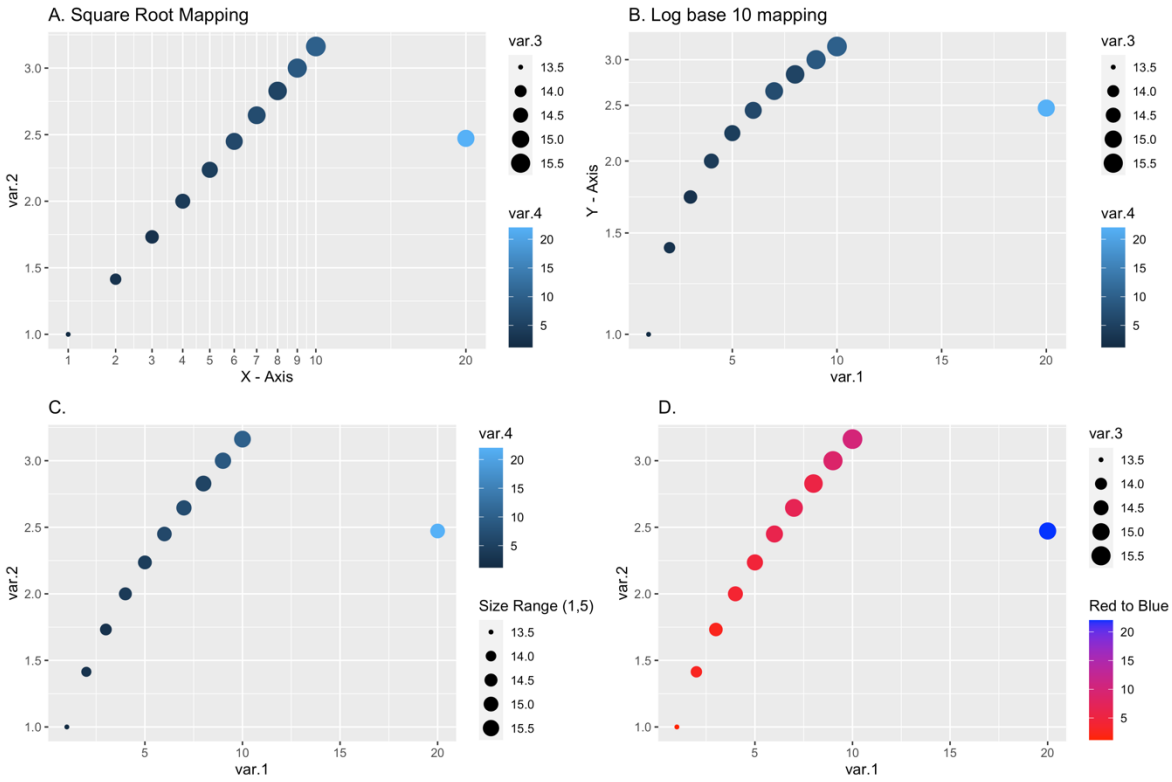
Unless directed otherwise, a result should always be displayed.

The graph **scalesGraph002** and **scalesGraph003** used in the exercises below, are built using the tibble **scaled001.tib**. The variables in the given data set are **top**pings, **flav**or, **syr**up, **cal**ories, and **weight**. The levels for most of these variables are recorded using their first letter as follows:

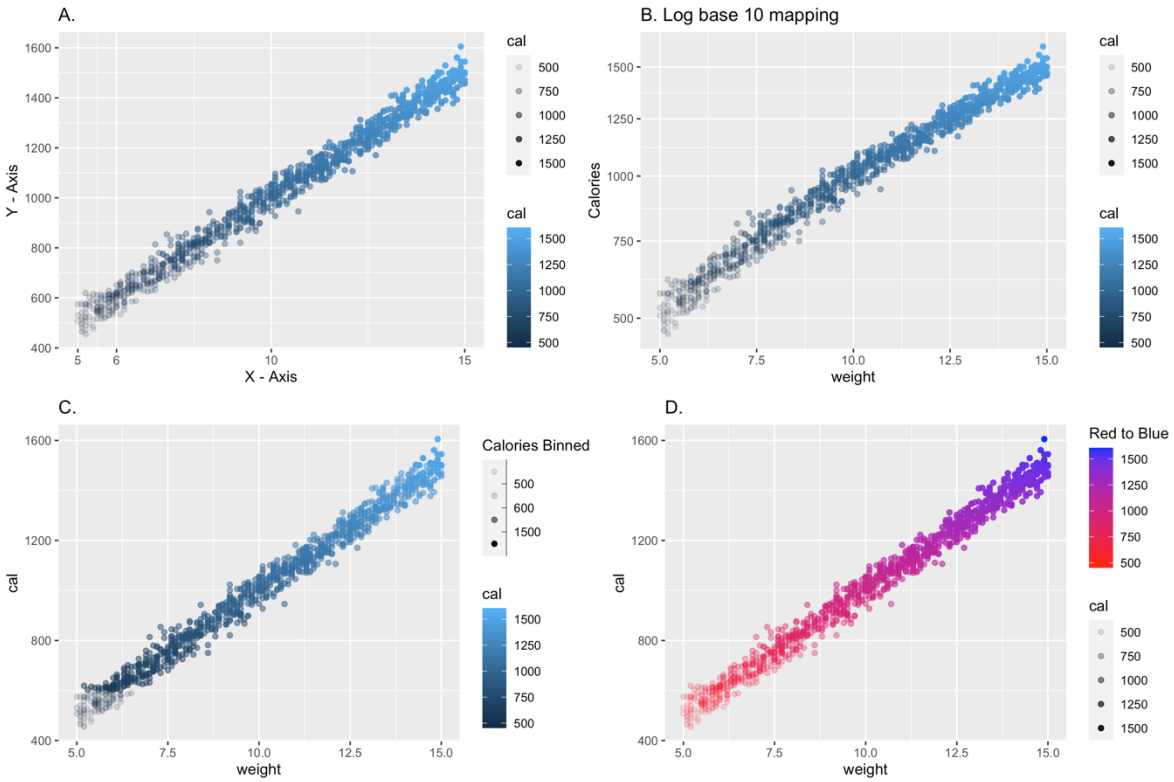
flavor	**toppings**	**syrup**
chocolate	**c**hips	TRUE
vanilla	**n**uts	FALSE
strawberry	**s**prinkles	
raspberry		

For these problems, any scale_(aesthetic)_(type of scale/mapping) function may be used, regardless of whether it was demonstrated.

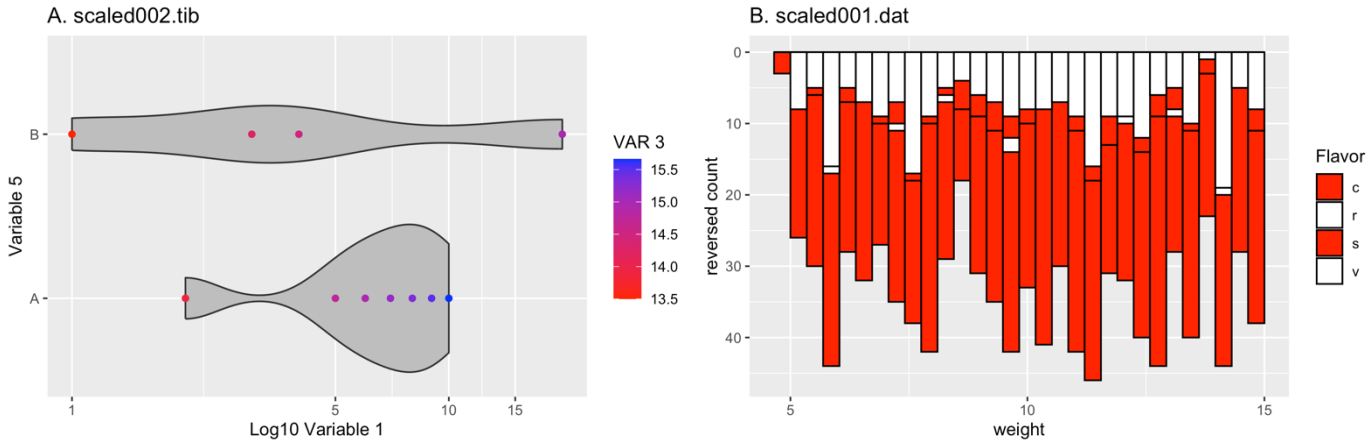
1. Using the graph **scalesGraph002**, recreate the following graphs.



2. Using the graph **scalesGraph003**, recreate the following graphs using any function whose name is of the form scale_(aesthetic)_(type of scale/mapping).



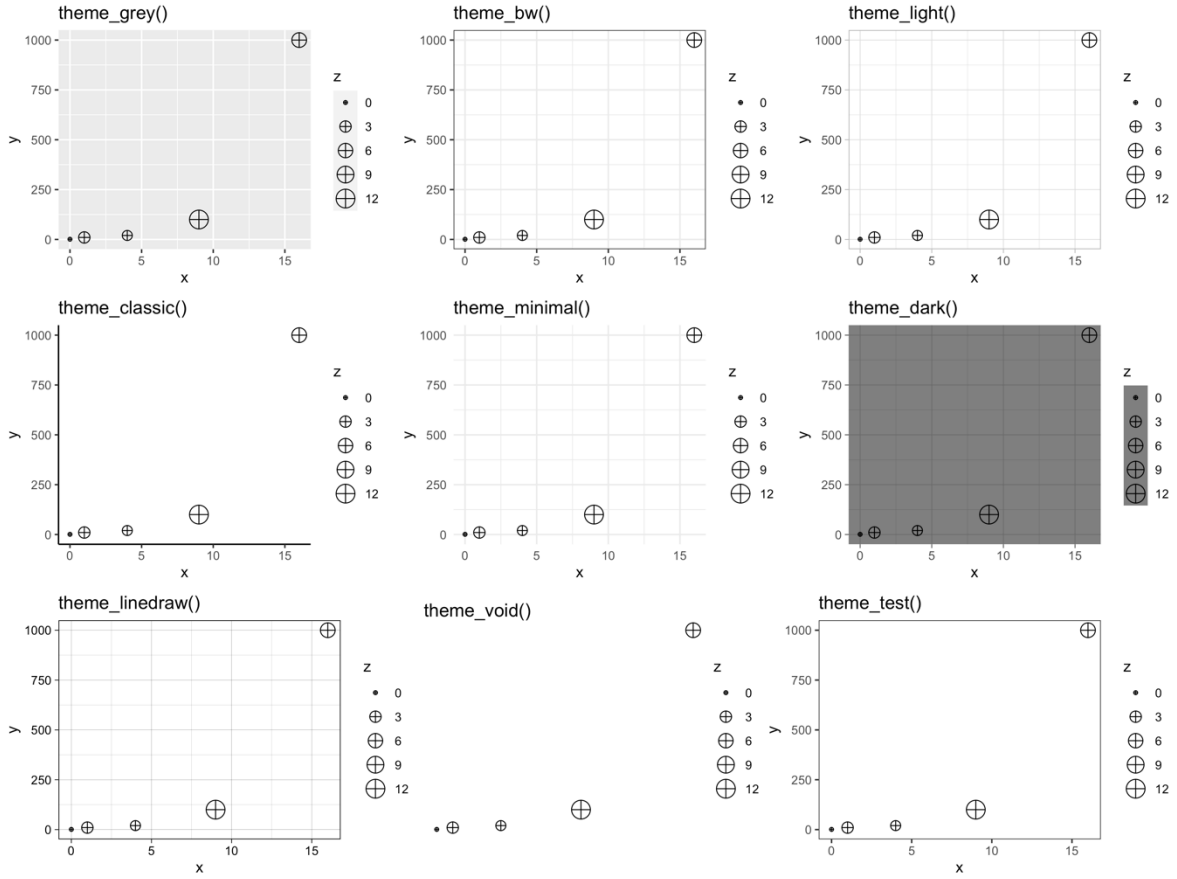
3. Using the dataframe indicated in the title to recreate the following graphs using any function whose name is of the form scale_(aesthetic)_(type of scale/mapping).



Chapter 36: ggplot2 - Themes

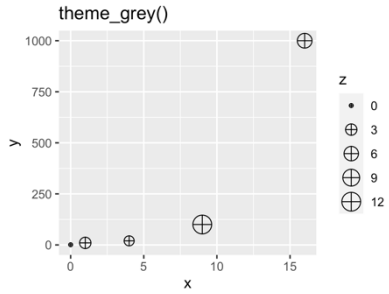
All the functions detailed so far control the display of the data. The **theme** functions all control of all non-data related aspects of the graph. This includes the fonts, backgrounds, margins, gridlines, and many many other aspects of a graphic. The *theme_grey*( ) function is the default theme. Other built in theme functions are displayed below.

```
theme.grey <- p1 + labs(title = "theme_grey()")
theme.bw <- p1 + labs(title = "theme_bw()") + theme_bw()
theme.light <- p1 + labs(title = "theme_light()") + theme_light()
theme.dark <- p1 + labs(title = "theme_dark()") + theme_dark()
theme.minimal <- p1 + labs(title = "theme_minimal()") + theme_minimal()
theme.classic <- p1 + labs(title = "theme_classic()") + theme_classic()
theme.linedraw <- p1 + labs(title = "theme_linedraw()") + theme_linedraw()
theme.void <- p1 + labs(title = "theme_void()") + theme_void()
theme.test <- p1 + labs(title = "theme_test()") + theme_test()
```

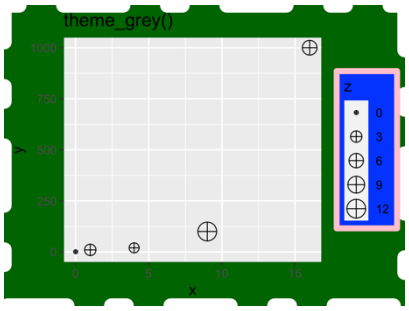


The *theme*( ) function can be used to create a custom theme or customize an existing theme. The arguments of the *theme*( ) function control the non-data aspects at a granular level. This means that the list of usable arguments is extremely long. However, they can be broken down into four types: **line**s, **rect**angle, **text** and **margins**. The **line**s type controls all line elements such as ticks, axis lines, ans panel grid. Any **line** related argument uses the $element_line()$ function to set aspects of the specified line. The **rect**angle type controls all rectangular elements which are usually related to backgrounds. Any **rect**angle related argument uses the $element_rect()$ function to set aspects. The **text** type controls all aspects of the texts appearance font, alignment, angle. Any **text** related argument uses the $element_text()$ function to set aspects of the specified text. Only the appeance of the text is affected. The actual words are determined with other functions when constructing the graph.

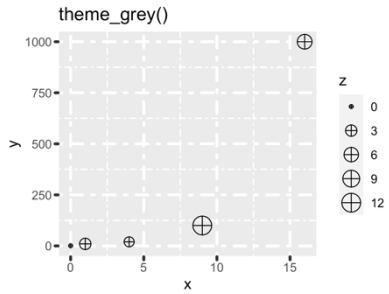
```
theme.grey.margin <- theme.grey + theme(plot.margin = margin(t = 5, r = 2, b = 3, l = 0, unit = "cm"))
theme.grey
```




```
theme.rect <- theme.grey + theme(rect = element_rect(fill = "darkgreen", color = "red", size = 5, linetype = 4), legend.backgro
und = element_rect(fill = "blue",
    color = "pink", size = 2, linetype = 1), legend.box.background = element_rect(fill = "yellow", color = "red", size = 2, lin
etype = 1))
theme.rect
```

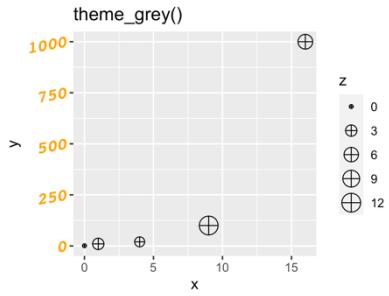



```
theme.line <- theme.grey + theme(line = element_line(color = "red", size = 1, linetype = 4, lineend = "round"))
theme.line
```




```
theme.text <- theme.grey + theme(axis.text.y = element_text(family = "mono", face = "bold.italic", colour = "orange", size = 12
, angle = 10))

theme.text
```



Exercises

The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

More sometime

Chapter 37: Factors

A **factor** is a type of vector used for storing categorical data. The different values being refered to as categories or levels. Construction of one makes use of the $factor(\ \ )$ function.

```
cat <- c("D", "A", "A", "B", "A", "A", "D", "D")
factor.cat <- factor(cat)
factor.cat

[1] D A A B A A D D
Levels: A B D
```

By default, the levels of the factor will only include values that exist in the data set. This is reflected by looking a table of values found in **factor.cat**.

```
table(factor.cat)

factor.cat
A B D
4 1 3
```

If there were other possible categories/levels that were not included in the **factor.cat** the table makes not reference to them. In this example, suppose that "C" was possible, but was not just not one of the values included. By making use of the **levels** argument, this unseen extra possibility can be recorded.

```
expanded.cat <- factor(cat, levels = c("A", "B", "C", "D"))
expanded.cat
table(expanded.cat)  # 'C' appears with a zero count.

[1] D A A B A A D D
Levels: A B C D
expanded.cat
A B C D
4 1 0 3
```

Functions for manipulating factors are found in the **forcats** library. All of these functions are named with the $fct\_$ prefix. The $fct\_expand(\ \ )$ function can be used to attach extra levels after a factor has been created. Additional levels are listed after the factor. The **after** argument indicates index after which the new levels are placed.

```
library(forcats)
expanded.cat.forcat <- fct_expand(f = factor.cat, "E", "F", after = 1)
expanded.cat.forcat  # 'E', 'F' are placed after level 1 = 'A'

[1] D A A B A A D D
Levels: A E F B D
```

The $fct\_drop(\ \ )$ function removes levels that do not appear in the dataset.

When looking at these factors, it may appear that a factor is a character vector. However, the letters are not surrounded by quotes. Therefore, they are not strings. $factor(\ \ )$ has actually created an integer vector with extra attributes. The $unclass(\ \ )$ function will display the integer equivalents.

```
unclass(factor.cat)  # 'A' <-> 1. 'B' <-> 2.  'D' <-> 3.

[1] 3 1 1 2 1 1 3 3
attr(,"levels")
[1] "A" "B" "D"
```

By default, the mapping between the levels and the integers is determined alphabetically. Two different mapping occur here due to the addition of "C" as a level in **expanded.cat**. For a different mapping, the **levels** argument should be used.

```
new.mapping.cat <- factor(cat, levels = c("D", "C", "A", "B"))
unclass(new.mapping.cat)  # 'D' <-> 1. 'C' <-> 2.  'A' <-> 3. 'B' <-> 4.

[1] 1 3 3 4 3 3 1 1
attr(,"levels")
[1] "D" "C" "A" "B"
```

The integer mapping doesn't imply an ordering or importance to the levels. However, some categorical data is naturally ordered. If so, the levels should be set in an increasing order and the **ordered** argument should be set to $FALSE$.

```
ordered.cat <- factor(cat, levels = c("D", "A", "B", "C"), ordered = TRUE)
ordered.cat  # D is purposely set to the 'least' in this ordering

[1] D A A B A A D D
Levels: D < A < B < C
```

In other situation, the level mapped to 1 will represent a *reference* level[52]. In this case, the factor should be set up with the *reference* level first. Alternatively, the $fct\_relevel(\ )$ allows a remapping of levels.

```
new.mapping.cat

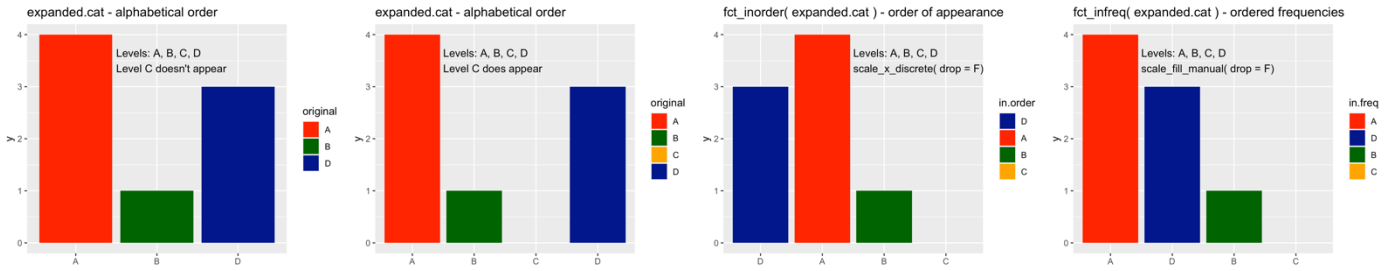
[1] D A A B A A D D
Levels: D C A B

fct_relevel(.f = new.mapping.cat, "C", "A")  #Unnamed levels get moved to the end retaining thier original order.

[1] D A A B A A D D
Levels: C A D B
```

The integer mappings can control how levels are displayed. In particular, three functions in the **forcats** package can quickly change the mapping. $fct\_inorder(\ )$ will set a mapping that indicates the order which levels appeared.[53] $fct\_infreq(\ )$ sets a mapping based upon frequencies; largest first. If the levels are numeric, $fct\_inseq(\ )$ will arrange the levels based on the numeric values.

```
expanded.original.cat <- data.frame(original = expanded.cat)
inorder.cat <- data.frame(in.order = fct_inorder(expanded.cat))  # Order of first appearance
infreq.cat <- data.frame(in.freq = fct_infreq(expanded.cat))  # Frequency of each level - Largest First
```



If the labels need to be changed, the $fct\_recode(\ )$ function will change labels but not the integer mapping[54].

```
expanded.cat <- fct_recode(expanded.cat, d = "D", a = "A", b = "B", c = "C")
# Notice the difference in syntax. levels needs 'oldName' = 'NewName'. fct\_recode uses `NewName` = 'OldName'

expanded.cat

[1] D A A B A A D D
Levels: A B C D

unclass(expanded.cat)

[1] 4 1 1 2 1 1 4 4
attr(,"levels")
[1] "A" "B" "C" "D"
```

The $fct\_recode(\ )$ function can be used to collapse a levels together. “a” and “e” are being replaced with “vowel” and the rest are becoming “consonant”.

```
expanded.cat <- fct_recode(.f = expanded.cat, consonant = "d", vowel = "a", consonant = "b", consonant = "c")

expanded.cat

[1] D A A B A A D D
Levels: A B C D
```

[52] This occurs with some logistic regression models.
[53] An error will occur is any unobserved layers have been added
[54] Using $fct\_recode(\ )$ requires storing the result in a variable. That is not done here.

Exercises

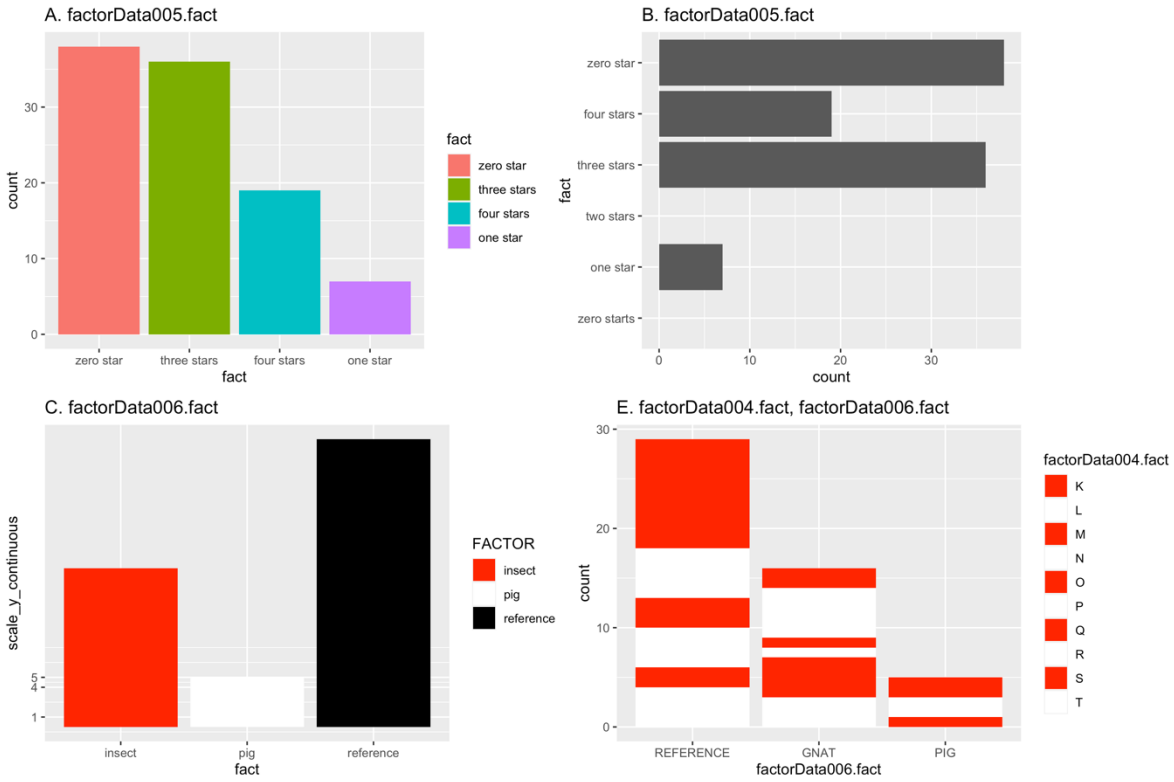
The exercises given below should be completed using only information given in the current section and previous sections. The **help** menu should not be used as a window into all possible functions unless it is specifically allowed. Similarly, only those packages that have been introduced in the current or previous sections may be used. Any computations that need to be completed should be done making use of R and not in your head. For example, if asked to find the sum of one and two, you should type in 1+2, and not simply give the answer of 3.

When told to use a specific number of lines of code, this should be interpreted as one line is one complete command. Some commands can be very long and necessary will wrap around. In some instances, a command is purposely wrapped around several lines. This insures readability.

Unless directed otherwise, a result should always be displayed.

1. Use the indicated vector to create a factor with an alphabetical assignment of integers to levels. Display the result and the integer version.
    a. factorData001
    b. factorData002
    c. factorData002
2. Use the indicated vector to create a factor with an alphabetical assignment of integers to levels. Listed after the vector is a list of possible levels. Make sure they all recorded in the factor. Table the contents of the factor.
    a. factorData001 - “a” “b” “c” “d” “e” “f” “g” “h” “i” “j”
    b. factorData002 - “better” “best” “good” “bad” “awful”
    c. factorData003 - “reference” “cow” “apple” “leaf”
3. Use the indicated factor to create a factor with the specified levels. The first level listed should be mapped to 1, the second to 2, and so on. Table the contents of the created factor.
    a. factorData004.fact - “K” “L” “M” “N” “O” “P” “Q” “R” “S” “T”
    b. factorData005.fact - “four stars” “three stars” “two stars” “one star” “zero star”
    c. factorData006.fact - “reference” “giraffe” “gnat” “pig”
4. Recode the indicated factors as described below. Table the contents of the created factor.
    a. factorData004.fact - “vowels” “consonant” (Look up what these are and you will know the recoding.)
    b. factorData005.fact - “no stars” “one,two, or three stars” “four stars”
    c. factorData006.fact - “reference” “not reference”

5. Use the indicated vector/factor/dataframe/tibble tor recreate the following graphs. Table the contents of the created factor.



Chapter 38: Dates & (Dates & Times)

The **lubridate** package is used to handle two types of data: date and datetime. **lubridate** has several functions for reformatting character and numeric vectors whose elements are meant to represent these data types. These functions attempt to reformat single values or combine values from several vectors into a date related format. First, data containing date information only will be considered. Once that is sufficiently completed, time will be added to the mix.

```
library(lubridate)
```

When attempting to reformat a date value, it is important to know the order of the calendar periods: year, month, and day. The order of the calendar periods will decide on the function to be used. Luckily, the name of the relevant function and order of the calendar periods are linked. The function's names are based on the first letter of **y**ear, **m**onth, and **d**ay, placed in the appropriate order.

For example, "2021-10-31", would make use of the *ymd*( ) function. While, "10-31-21" would make use of the *mdy*( ) function[55].

```
ymd("2021-10-31")  # Year - Month - Day

[1] "2021-10-31"

mdy("10-31-21")  # Month - Day - Year

[1] "2021-10-31"
```

Based on the output, it appears that these functions produce a character vector. A closer inspection indicates otherwise.

```
attributes(ymd("2021-10-31"))

$class
[1] "Date"

typeof(mdy("10-31-21"))

[1] "double"
```

The result of these functions is a numeric value of a "Date" class. In actuality, a date is stored as the number of days before or after January 1, 1970. Positive(Negative) values represent times after(before) that date. However, as class "Date" *R* will display the corresponding date as a string.

In examples given, a minus sign was used to separate the components. This does not need to be the case. The functions will examine the values given to them, attempt to determine the relevant information.

```
# Some of the different Possible Date Formats
dmy(311021)
dmy(31102021)
dmy("3110 21")
dmy("31.10.2021")
dmy("31October21")

dmy("Today is 31 Oct 21.")

[1] "2021-10-31"
```

In the preceding code, several different date representations are given, and the *dmy*( ) function was able to extract the date information. This will be true for the other variations of this function. However, on occasion, it may not be possible to parse the given information. In which case, *NA* is returned with a warning.

```
dmy("31Octber21")
```

When time (**h**ours, **m**inutes, **s**econds) is part of the date data, variations of the existing function names are used. Some of these are *dmy_h*( ), *dmy_hm*( ), or *dmy_hms*( ) . The first three letters are rearranged depending upon the date portion of the data. When times are involved, the data is recorded as POSIXct class. The data is still numerical; it is the number of seconds since the beginning of 1970.

```
ymd_h("2021-10-31 19")
```

[55] Additionally, two function exist for dealing with only a **y**ear and **m**onth.

```
[1] "2021-10-31 19:00:00 UTC"

str(ymd_hm("2021-10-31 19:06", tz = "MST"))  #Time zones can be included. OlsonNames() lists timezones.

 POSIXct[1:1], format: "2021-10-31 19:06:00"

typeof(ymd_hms("2021-10-31 19:06:44.9", tz = "EST"))

[1] "double"
```

When the date data is held in different vectors, the *make_date*( ) function can be used to pull it together.

```
days.data <- c(31, 3, 5)
month.data <- c(10, 11, "Oct")
year.data <- c(2021, 21, 2021)
make_date(year = year.data, month = month.data, day = days.data)

[1] "2021-10-31" "0021-11-03" NA
```

With *make_date*( ), the years can not be shortened and the months must be specified numerically. Otherwise, an *NA* value will result with a warning. If a time is also involved, the *make_datetime*( ) function is used. Additional **hour**, **min**ute, and **sec**ond arguments are used.

```
make_datetime(year = 2021, month = 10, day = 31, hour = 23, min = 58, sec = 59)

[1] "2021-10-31 23:58:59 UTC"
```

Once a data value has been reformatted, extracting a piece of the data may be necessary. The *year*( ), *month*( ), *day*( ), *wday*( ), *hour*( ), *minute*( ), and *second*( ) functions are some of these.

```
ex.date <- make_date(year = 2021, month = 1, day = 11)  # Date Class
year(ex.date)

[1] 2021

day(ex.date)  # day of the year

[1] 11

ex.date.time <- make_datetime(year = 2021, month = 1, day = 11, hour = 12, min = 34, sec = 12)  # POSIXct Class
# label and abbr arguments control a name being displayed or a numeric equivalent
month(ex.date.time, label = FALSE, abbr = TUE)  #1 = Jan, 2 = Feb, etc

[1] 1

wday(ex.date.time, label = TRUE, abbr = FALSE)  # 1 - Sun, 2 - Mon, etc

[1] Monday
Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < Friday < Saturday

second(ex.date.time)

[1] 12
```

These functions can also be used to modify a component.

```
year(ex.date.time) <- 2022
minute(ex.date.time) <- minute(ex.date.time) + 30  #Notice that the hour rolls over
ex.date.time

[1] "2022-01-11 13:04:12 UTC"
```

Special Dates and Times

Determining the current date and/or time can be done using *Sys.Date*( ) function from base *R* and the *now*( ) and *today*( ) functions from the **lubridate** package. *now*( ) returns a POSIXct value. The others return Date class data.

Exercises

Lab

Chapter 39: Fixed Pattern Matching - base

Several functions exist that will search a string, or a vector of strings, for a given pattern of characters. The pattern of characters could be an explicitly defined string, like “ab”, or it could be a more general pattern.

A more general pattern may look for strings of characters that start with “a” and end with “b” and have at least one letter between them. These functions would search for strings that look like “aab”, “aodib”, “a98765qb”, etc. These more general patterns are described with strings called **regular expressions**. The functions detailed here can use an explicitly defined pattern or a regular expression. When demonstrating them, an explicit pattern “ab” will be used. Creating a regular expression will be covered later.

A few things to note about these functions.

a. These functions are found in base *R* and in the *stringr* package. There is some overlap in functionality. When there is, they will be grouped together.
b. Functions from the *stringr* package will be listed with *stringr*::. This is not truly necessary when the *stringr* package is explicitly loaded, but it will make it easier to identify the functions associated with that package.

```
library(stringr)
```

c. The functions in the stringr package all start with ‘str_’, and the arguments are consistent in naming and ordering.
d. For most functions, more arguments exist than will be discussed. In those cases, the behavior of the function will be describe in terms of the defaults for the remaining arguments.

For each function, a vector of strings will be searched. The vector *search.this* will be used. The substring being searched for will be the two character word “ab”, in all lower case letters.

```
search.this <- c("e", "ab", "abab", "aBc", "aBCdab")
```

grep( )

The *grep( )* function will search the given string *search.this* for the *pattern*, “ab” in this case. When the argument *value* is set to *FALSE*, *grep( )* will return the index of the elements of *search.this* that contains the pattern. If *value* is set to *TRUE*, the elements of the vector *search.this* are displaced. When *ignore.case* is set to FALSE, it is a case-sensitive search, the case does not matter.

```
grep(pattern = "ab", x = c("e", "ab", "abab", "aBc", "aBCdab"), ignore.case = FALSE, value = FALSE)

[1] 2 3 5

grep(pattern = "ab", x = c("e", "ab", "abab", "aBc", "aBCdab"), ignore.case = TRUE, value = TRUE)

[1] "ab"     "abab"   "aBc"    "aBCdab"
```

The *stringr* function *str_subset( )* serves the same purpose as *grep* when *value* = *TRUE*.

```
str_subset(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

[1] "ab"     "abab"   "aBCdab"
```

The functions *grepl( )* & *str_detect( )* detect the existence of a given pattern, like *grep( )*, however the returned value is logical vector. *TRUE* indicates the presence of the pattern.

```
grepl(pattern = "ab", x = c("e", "ab", "abab", "aBc", "aBCdab"), ignore.case = TRUE)  # Is set to be case-insensitive.

[1] FALSE  TRUE  TRUE  TRUE  TRUE

stringr::str_detect(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")  # Looks the the exact string.

[1] FALSE  TRUE  TRUE FALSE  TRUE
```

str_detect( ) does not have the option *ignore.case*. When regular expressions are used to define the pattern being searched for, case sensitivity can be included there, if desired.

Substring Replacement

The functions *sub*( ) & *str_replace*( ) will look for the first instance of the pattern in each string, and replace it with another string.

```
sub(pattern = "ab", replacement = "ZZZ", x = c("e", "ab", "abab", "aBc", "aBCdab"), ignore.case = FALSE)

[1] "e"       "ZZZ"     "ZZZab"   "aBc"     "aBCdZZZ"

stringr::str_replace(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab", replacement = "ZZZ")

[1] "e"       "ZZZ"     "ZZZab"   "aBc"     "aBCdZZZ"
```

The functions *gsub*( ) and *str_replace_all*( ) replace every instance of the pattern

```
gsub(pattern = "ab", replacement = "Z", x = c("e", "ab", "abab", "aBc", "aBCdab"), ignore.case = FALSE)

[1] "e"     "Z"     "ZZ"    "aBc"   "aBCdZ"

stringr::str_replace_all(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab", replacement = "ZZZ")

[1] "e"       "ZZZ"     "ZZZZZZ"  "aBc"     "aBCdZZZ"
```

Extracting Substrings

The *str_extract*( ) and *str_extract_all*( ) functions are used to extract substrings based on the given pattern. When the given pattern is a fixed string, the functions will only return substrings that look exactly like the fixed string. When used with a regular expression, different substrings will be returned, as long as they fit the pattern. *str_extract*( ) will only extract the first instance of a pattern in each string, whereas *str_extract_all*( ) will extract all instances of the pattern. *str_extract_all*( ) has a third argument that controls the form of the output. When simplify is set to *TRUE*, a *matrix* is returned instead of a *list*.

```
stringr::str_extract(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

[1] NA   "ab" "ab" NA   "ab"

str_extract_all(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab", simplify = FALSE)

[[1]]
character(0)

[[2]]
[1] "ab"

[[3]]
[1] "ab" "ab"

[[4]]
character(0)

[[5]]
[1] "ab"
```

str_count( )

The *str_replace_count*( ) function will count the number of instances of the given pattern in a string.

```
stringr::str_count(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

[1] 0 1 2 0 1
```

Pattern Location

The *str_locate*( ) and *str_locate_all*( ) will return information on the position of a pattern within a given string. They will list the start position and the ending position. If the pattern does not exist, *NA* will be returned.

```
stringr::str_locate(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

     start end
[1,]    NA  NA
[2,]     1   2
[3,]     1   2
[4,]    NA  NA
[5,]     5   6
```

The output is a matrix. The first row of the result indicates that “ab” does not exist in the string “e”. The second row indicates that “ab” can be found for the first time in the string “ab” starting with the first character, and finishing with the second character. The last row indicates that “ab” can be found in “aBCdab” starting with the fifth character and ending with the sixth. When looking at this result, it may seem that the second column of ending positions is not needed. This is true when looking for a fixed explicit pattern like “ab”. But, when looking for a more general pattern, it can be the case that one instance of the pattern is three characters long, while another instance is ten characters long.

When looking at *str_locate_all*( ), fhe main difference from *str_locate*( ) will return information on all disjoint matches instead of just the first.

```
stringr::str_locate_all(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

[[1]]
     start end

[[2]]
     start end
[1,]     1   2

[[3]]
     start end
[1,]     1   2
[2,]     3   4

[[4]]
     start end

[[5]]
     start end
[1,]     5   6
```

The output here is in the form of a list. Each double-square [[ ]] bracketed section will reference a new element. [[3]] references “abab”. Its first row indicate that the first occurence of “ab” start in position 1 and ends in position 2. The second row indicates the positioning of the second instance.

Base *R* has similar functions *regexpr*( ), *gregexpr*( ),and *regexec*( ).

str_starts( ) & *str_ends*( )

The functions *str_starts*( ) & *str_ends*( ) return a logical vector indicating whether the pattern occurs at the start or end of the given strings.

```
stringr::str_ends(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

[1] FALSE  TRUE  TRUE FALSE  TRUE

stringr::str_starts(string = c("e", "ab", "abab", "aBc", "aBCdab"), pattern = "ab")

[1] FALSE  TRUE  TRUE FALSE FALSE
```

Splitting Strings Apart

Several pieces of information could be contained in a single string. The *strsplit( )* and *str_split( )* functions make it relatively simple to separate these pieces of information, if there is a regular pattern used to separate them.

```
strsplit("Age , Height , Weight , Other", split = " , ")

[[1]]
[1] "Age"    "Height" "Weight" "Other"

stringr::str_split(string = "Age , Height , Weight , Other", pattern = " , ", n = 3, simplify = FALSE)

[[1]]
[1] "Age"            "Height"         "Weight , Other"
```

str_split( ) has an argument *n* that determines the number of pieces the string will be split into. The *simplify* argument indicates where a list of a matrix will result.

One final note about these functions: The string " , " was used instead of ",". The second would have left extra spaces around the split text.

Chapter 40: Regular Expressions

A **regular expression** is a string that represents a pattern of characters symbolically.

It could be as simple as “the”. This is a regular expression that would be used to identify string that contained the string “the”. It could be used to find

“**the**”

“**the**re is a cat in the hall”

“And **the**n it ran.”

More generally, a regular expression could be used to identify strings that contain words that begin and end with the same letter. The string

\\b([[:alpha:]])[[:alpha:]]+\\1([^[:alpha:]]|\\b)

would identify strings such as

“The source of the huge **river** is the clear spring.”,

“The fruit peel was cut in thick **slices**.”,

“**bob**”,

but not strings such as

“garbage”,

“A severe storm tore down the barn.” .

Metacharacters

When creating a regular expression, certain characters are given special significance.

The metacharacters are

\ $ | . ( ) [{ * + ?

When used as part of a regular expression they will each indicate something about the pattern that is being search for. However, there will be times where these symbols will be the object of the search. In such times, \\ should be placed in front of each desired symbol individually. To search for the literal symbol and not the metacharacter, you would search for

\\ \\$ \\| \\. \\(\\) \\[\\{ * \\+ \\?

This means that a search for \\ requires four back slashes \\\\.

Search Method

Consider the literal regular expression “bob” and the following two strings

“boy! bonnie bounced away”

“boy! bob bounced away”.

When deciding if there is a match withing either string, the “b” in boy would be considered the start of a match. Therefore, a search for the next character in “bob” would begin in the second position of these strings. Since a match occurs in both strings, the third character of each string is compared to the third character in “bob”. Neither is a match.

The process begins again with the first “b” in “bob”, and now that “b” is compared to the “o” in “boy”. No match. The first “b” in “bob” is now compared to the “y”. No match. Now, it is compared to the “!”. No match.

Next it is compared to the space between “!” and the second “b” in either string. Now, we move on to comparing the “o” in “bob” to the seventh character in each string. Match! The second “b” in “bob” is now compared to the eighth character in both strings. There is no match for the first string at this point. But, there is a match for the second. The second search is stopped.

However, for the first string. The search begins anew. The first “b” in “bob” is compared to the “o” in “bonnie”. Since “bob” does not occur in the first string, only the first “b” in “bob” will eventually be compared to each letter in the string. The “o” in “bob” will only be compared to the “o” in “bounced”, and the second “b” in “bob” will only be compared to the “u” in “bounced”

Matching characters, the search would look something like. The vertical lines are included to make matching easier

```
boy! bonnie bounced away            boy! bob bounced away
bob^ ^ ^                            bob^ ^ ^
 b | | |                             b | | |
  b| | |                              b| | |
   b | |                               b | |
    b| |                                b| |
     bob                                 bob  stop
      b
       b etc.
```

Character Classes

A **character class** is a step beyond searching for an exact character match. A character class defines a group of characters that are all an equally valid match. Based on the example, a character class can be defined to search for “b” or “B” as the first letter. Essentially, the class would allow a search for “bob” or “Bob”.

To indicate a character class, wrap a set of acceptable characters in square brackets. The regular expression used to search for “bob” or “Bob” would be “[bB]ob”. The order of the characters inside the square brackets does not matter.

Sometimes, the number of elements in the desired character class will be easier to describe by indicating what is not desired. In such situations, a carat “^” should be inserted after the first square bracket. “[^bB]ob” would search for “Aob”, ” ob”, “xob” as few examples. A string whose last two characters, in order, are “o” and “b”, and first character is not “b” or “B”.

A set of shortcuts for certain classes are listed below:

Class	Description	POSIX CLASS
[Einr]	Exactly one of “E”, “i”, “n”, or “r”	
[^Einr]	Anything but “E”, “i”, “n”, or “r”	
[A-Z]	An Uppercase Letter From A to Z	[[:upper:]]
[a-z]	A Lowercase Letter From a to z	[[:lower:]]
[a-zA-Z]	An Uppercase or Lowercase Letter	[[:alpha:]]
[0-9]	A Digit from 0 to 9	[[:digit:]]
[a-zA-Z0-9]	A Digit or A Letter - either case	[[:alphanum:]]
[^a-zA-Z0-9]	Not A Digit or A Letter - either case	[^[:alphanum:]]
	Blank Characters (space & tab)	[[:blank:]]
	Control Characters	[[:cntrl:]]
	Punctuation !#%&*()“:;’+-.	[[:punct:]]
	Space Characters:tab,newline,space…	[[:space:]]
		(Sanchez 2013)

This regular expression will search for strings that start with “b”, followed by a vowel.

```
my_pattern = "b[aeiou]"
possible_strings <- c("b", "B", "bu", "Bu", "ban", "a happy bunny", "bpo")
str_subset(string = possible_strings, pattern = my_pattern)

[1] "bu"           "ban"          "a happy bunny"
```

Keep in mind that regardless of the number of characters in the character class, only one character is being checked for a match.

Quantifiers

In certain searches, a character class needs to be used to check several consecutive positions. In our previous example, a search for a "b" followed by three vowels might be desired. The regular expression for this would look like

$$b[aeiou][aeiou][aeiou]$$

While three repetitions is not many, imagine how unwieldy a longer search criteria may become. Imagine a case where the number of repetitions could not be specified. This is where quantifiers can be used. Quantifiers will indicate the number of repetitions of the previous character class, or literal character. A quantifier will follow the character class that it is applied to. For each quantifier in this table, the quantifier will indicate that the preceding class, or character, is

Quantifier	Description
?	optional and matched at most once
*	is matched zero or more times
+	is matched at least one time
{n}	is matched exactly n times
{n,}	is matched at least n times
{n,m}	is matched at least n times and at most m times
	(Sanchez 2013)

Using the question mark, almost all strings are returned. The search is for a "b" that is **possibly** followed by a vowel. Neither "B" or "Bu" satisfy this, and are not selected.

```
possible_strings <- c("b", "B", "bu", "Bu", "ban", "a happy bunny", "bpo", "bue", "a happy bounty", "beautiful", "beady", "beeee")
my_pattern = "b[aeiou]?"
str_subset(string = possible_strings, pattern = my_pattern)

 [1] "b"              "bu"             "ban"            "a happy bunny"  "bpo"            "bue"            "a happy bounty"
 [8] "beautiful"      "beady"          "beeee"
```

Using the plus sign, fewer strings are returned. The search is for a "b" that **is** followed by at least one vowel. In addition to the previously excluded are "b" (not followed by anything) and "bpo" (not followed by a vowel).

```
my_pattern = "b[aeiou]+"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "bu"             "ban"            "a happy bunny"  "bue"            "a happy bounty" "beautiful"      "beady"
[8] "beeee"
```

Replacing the plus sign with a {2}, the results are fewer. Only strings with "b" followed by at least two vowels are selected.

```
my_pattern = "b[aeiou]{2}"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "bue"            "a happy bounty" "beautiful"      "beady"          "beeee"
```

Changing {2} to {2,3}, {2,4} or {2,10} won't produce a different result unless another character is placed after the quantifier. Adding [^aeiou] will indicate that a non vowel should appear after 2-3 vowels have appeared.

```
my_pattern = "b[aeiou]{2,3}[^aeiou]"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "a happy bounty" "beautiful"      "beady"
```

Anchors

Anchors are a different type of search indicator. Instead of searching for a specific set of characters, the search for a position in a string. An anchor may force a match at the beginning of a string, or the end. It may consider only possible matches that occur at the boundary between what could be considered the start and end of a word.

Anchor	Description
^	Match occurs at the start of a string
$	Match occurs at the end of the string
\b	Match occurs on boundary of a word
\B	Match does not occur on boundary of a word

The carat looks for a match only at the very beginning of a string. In this example, it is looking for the first character to be "d", absent that, no match could be found, even though "dog" appears in all strings.

```
possible_strings <- c(" dogs are fun", "dogs are fun", "a dog is fun", "all dogs are fun", "funny dog", "what is adog")
my_pattern <- "^dog"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "dogs are fun"
```

After a match for "dog" has been found, the dollar sign at the end of the pattern indicates that the "g" needs to be the last character in the string being searched.

```
my_pattern <- "dog$"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "funny dog"    "what is adog"
```

A word boundary is indicated by "\\b". A word boundary is a position either at the beginning of a string or the end of the string where the first or last character of the string is identified as a word character. Additionally, a word boundary could occur with in a string at a position between a non-word character and a word character. A **word character** is any letter, digit or underscore in the current locale. In this example, "what is adog" is not selected because the character before "d" is a letter. Letters are considered word characters.

```
my_pattern <- "\\bdog"
str_subset(string = possible_strings, pattern = my_pattern)

[1] " dogs are fun"    "dogs are fun"     "a dog is fun"     "all dogs are fun" "funny dog"
```

Using the word boundary on both sides of "dog" says that the string must be surrounded by non-word characters. For "a dog is fun" that is spaces. For "funny dog", it is a space on one end, and the end of the string on the other.

```
my_pattern <- "\\bdog\\b"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "a dog is fun" "funny dog"
```

Using the non-word boundary, only "what is that adog" is selected. The "d" is not at the start or end of a string of letters (or non-word characters) and it is followed by "o"and "g".

```
my_pattern <- "\\Bdog"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "what is adog"
```

Period

The period "." is used to match any character. It is a placeholder for when there is no preference for what character occurs in a particular position, but a preference exists for those before and after. With a quantifier, it can be optional or repeated.

```
possible_strings <- c("dag", "deg", "ddg", "dog", "d g", "dx", "dogs")
my_pattern <- "d.g"
str_subset(string = possible_strings, pattern = my_pattern)

[1] "dag"  "deg"  "ddg"  "dog"  "d g"  "dogs"
```

Grouping

Using a set of round parentheses around a sequence of characters in a pattern creates a group. With a quantifier added on, the grouped portion of the search pattern can be made optional or repeated.

```
possible_strings <- c("The bad dog.", "the bad bad  dog", "THe bad bad bad dog", "tHE bad tin dog", "The dog", "Oh! the dog ran
")
my_pattern <- "(bad )+dog"
str_extract(string = possible_strings, pattern = my_pattern)

[1] "bad dog"         NA                "bad bad bad dog" NA                NA                NA
```

With the plus attached to "(bad)+" the string "bad" is searched for, once found the search continues provided that "bad" is found next or is completes if "dog" is found next.

With an anchor, then entire set of search parameters can be forced to look at only certain positions.

Used in conjunction with the **alternation** "|" symbol, alternative strings can be searched for in a given position.

```
possible_strings <- c("The bad dog.", "the good dog", "THe muddy dog", "the dog", "the dog", "Oh! the dog ran", "the good muddy
dog.")
my_pattern1 <- "(good |bad |muddy )+dog"
str_extract(string = possible_strings, pattern = my_pattern1)

[1] "bad dog"        "good dog"       "muddy dog"      NA               NA               NA               "good muddy dog"

my_pattern2 <- "(good|bad|muddy)+ dog"
str_extract(string = possible_strings, pattern = my_pattern2)

[1] "bad dog"   "good dog"  "muddy dog" NA          NA          NA          "muddy dog"
```

As an interesting case, two patterns were used to make the same selection of strings. The difference is in the placement of a space. Notice *str_extract( )* will indicate exactly what the matching characters are. Looking at the last string, in one case the match is good muddy dog" while in the other it is "muddy dog".

Backreferences

Used in conjunction with grouping, a back reference can be used to identify previously constructed strings. They are identifies by two backslashes followed by a number. The first grouping used in a regular expression would be identified with \\1. The second grouping used would be identified with \\2.

```
possible_strings <- c("to be or not to be ", "to be or not be to")
my_pattern1 <- "(..) (..).*\\1 \\2"
str_extract(string = possible_strings, pattern = my_pattern1)

[1] "to be or not to be" NA

my_pattern2 <- "(..) (..).*\\2 \\1"
str_extract(string = possible_strings, pattern = my_pattern2)

[1] NA                   "to be or not be to"
```

With bother patterns, "(..) (..)" defines a pattern of two characters followed by a space followed by two characters. When searching the possible strings, these "(..) (..)" become "(to) (be)". Assigned to "(to)" is back reference "\\1" and "(be)" is assigned "\\2". They are followed by ".*" which is just a placeholder used to extend the search out for the back references. Using *my_pattern1*, the backreferences together look for "to be". With *my_pattern2*, the backreferences look for "be to".

Exercises

Lab

Chapter 41: dplyr - Groups & Summaries

The **dplyr** package provides functions that are exclusively meant for manipulating dataframes/tibbles.

```
library(dplyr)
```

Many of the operations can be performed using base *R* code. The first set of **dplyr** functions to be examined perform some operations on the rows of a dataframe/tibble. To demonstrate these functions the following example dataframe will be used.

```
num.var1 <- 10:1
num.var2 <- c(100, 60, 80, 70, 90, 10, 20, 50, 30, 40)
char.var3 <- c("A", "B", "A", "B", "C", "A", "A", "C", "B", "C")
logi.var4 <- c(F, F, T, T, F, T, F, T, F, F)
dat <- data.frame(num.var2, num.var1, char.var3, logi.var4)
```

Grouping

The *group_by*( ) is a function that does not produce an obvious change in a given dataframe/tibble beyond possibly converting a dataframe into a tibble and listing the variables that constitute a group[56].

```
dat.grouped.char.logi <- group_by(dat, char.var3, logi.var4)  # First argument is a dataframe/tibble, subsequent arguments are grouping variables.
dat.grouped.char.logi

# A tibble: 10 × 4
# Groups:   char.var3, logi.var4 [6]
   num.var2 num.var1 char.var3 logi.var4
      <dbl>    <int> <chr>     <lgl>
 1      100       10 A         FALSE
 2       60        9 B         FALSE
 3       80        8 A         TRUE
 4       70        7 B         TRUE
 5       90        6 C         FALSE
 6       10        5 A         TRUE
 7       20        4 A         FALSE
 8       50        3 C         TRUE
 9       30        2 B         FALSE
10       40        1 C         FALSE
```

The reverse of the *group_by*( ) function is the *ungroup*( ) function. It will remove selected variables from a grouping withing a tibble[57].

```
dat.grouped.char <- ungroup(dat.grouped.char.logi, logi.var4)  # Only the grouping due to logi.var is removed
dat.grouped.char[1:3, ]

# A tibble: 3 × 4
# Groups:   char.var3 [2]
  num.var2 num.var1 char.var3 logi.var4
     <dbl>    <int> <chr>     <lgl>
1      100       10 A         FALSE
2       60        9 B         FALSE
3       80        8 A         TRUE
```

The usefulness of adding a grouping to a tibble, is that certain operations can be applied to tibble, and the internal grouping will cause those operations to be performed on each group.

Summarizing Data

The *summarize*( ) function is a function that will apply summarizing functions to a tibble by each group in the tibble. Some functions that can be applied *summarize*( ) are

Location	Spread	Range	Position	Count	Logical
mean( )	*sd*( )	*min*( )	*first*( )	*n*( )	*any*( )
median( )	*IQR*( )	*max*( )	*last*( )	*n_distinct*( )	*all*( )
quantile( )			*nth*( )		(Wickham et al. 2021)

[56] To my knowledge, a dataframe can not hold grouping information. This necessitates the need to convert to a tibble.
[57] By not specifying any grouping variables to be removed, all grouping variables will be removed from the grouping.

To observe the functionality of *summarize*( ), it is first applied to **dat**, the tibble with no groupings. Each argument after **dat** indicates the name for a column in the resulting tibble, a function to be applied, and a variable[58].

```
summarize(dat, mean.var1 = mean(num.var1), sd.var2 = sd(num.var2), count = n())

  mean.var1 sd.var2 count
1       5.5 30.2765    10
```

Since **dat** did not contain any groupings, each function was applied to the indicated variable within **dat**.

By swapping out **dat** for **dat.grouped.char.logi**, a tibble with a grouping, the interaction of *summarize*( ) and *group_by*( ) becomes visible.

```
summarize(dat.grouped.char.logi, mean.var1 = mean(num.var1), sd.var2 = sd(num.var2), count = n())

# A tibble: 6 × 5
# Groups:   char.var3 [3]
  char.var3 logi.var4 mean.var1 sd.var2 count
  <chr>     <lgl>         <dbl>   <dbl> <int>
1 A         FALSE           7      56.6     2
2 A         TRUE            6.5    49.5     2
3 B         FALSE           5.5    21.2     2
4 B         TRUE            7      NA       1
5 C         FALSE           3.5    35.4     2
6 C         TRUE            3      NA       1
```

The first two columns of the resulting tibble, taken as pairs, indicate the different possible groupings of values in each grouping variable. The remaining columns provide the appropriate value for each function listed in the *summariz*( ) command applied to only those values in the indicated group. The value of 3.5 in the *mean. var*1 column represents the mean of all values in the **num.var1** column whose corresponding **char.var3** value was "C" and **logi.var4** value was *FALSE*. The last column contains the number of elements from the original tibble that fall into teach group. The two values of 1 explain why there are *NA* values for some computed standard deviations[59].

Applying the same summarize function to the tibble with only one grouping variable results in the following

```
summarize(dat.grouped.char, mean.var1 = mean(num.var1), sd.var2 = sd(num.var2), count = n())

# A tibble: 3 × 4
  char.var3 mean.var1 sd.var2 count
  <chr>         <dbl>   <dbl> <int>
1 A              6.75    44.3     4
2 B              6       20.8     3
3 C              3.33    26.5     3
```

Given only three unique values in **char.var3**, there are only three groups to be summarized in **dat.group.char**.

Two Important Facts

1. Once the tibble is named within either *summarize*( ) or *group_by*( ), it is not necessary to use the $-notation to reference any of the enclosed variables. Simply stating a variables name suffices.

2. Neither *summarize*( ) nor *group_by*( ) modify the original dataframe/tibble. If that is the desire, an assignment must be made.

[58] Except for the function *n*( ).
[59] At least two values are needed to compute a sample standard deviation.

Exercises

Lab

Chapter 42: dplyr - Rows

Several functions within the **dplyr** package are used for sub-setting ,or ordering, a given dataframe/tibble. To demonstrate these functions, the following dataframe will be used.

```
num.var1 <- 10:1
num.var2 <- c(100, 60, 80, 70, 90, 10, 20, 50, 90, 40)
char.var3 <- c("A", "B", "A", "B", "C", "A", "A", "C", "A", "C")
logi.var4 <- c(F, F, T, T, F, T, F, T, F, F)
dat <- data.frame(num.var2, num.var1, char.var3, logi.var4)
```

Filtering

The *filter*( ) function is used to subset, or extract, rows from a given dataframe/tibble. Its first argument is the dataframe/tibble to be filtered. Subsequently, should be a series of logical comparisons that characterize the rows that should be selected. In this example, only one condition is given.

```
filter(dat, char.var3 == "B")  # Select all rows with char.var3 value equal to 'B'

  num.var2 num.var1 char.var3 logi.var4
1       60        9         B     FALSE
2       70        7         B      TRUE
```

This command is equivalent to two base *R* commands.

```
dat[dat$char.var3 == "B", ]
subset(dat, char.var3 == "B")
```

More than one logical condition can be given. Only those rows that satisfy all the logical conditions will be selected.

```
# Select all rows with char.var3 value equal to 'B'
filter(dat, char.var3 == "B", num.var2 > 50)

  num.var2 num.var1 char.var3 logi.var4
1       60        9         B     FALSE
2       70        7         B      TRUE
```

Arranging

The *arrange*( ) function is used to sort the rows of a dataframe. Its first argument is the dataframe/tibble to be rearranged. Subsequently, the variables used to determine the sorting are listed. The first variable listed determines the overall order. Variables that come next determine how ties are broken.

```
arrange(dat, char.var3, num.var1)

   num.var2 num.var1 char.var3 logi.var4
1        90        2         A     FALSE
2        20        4         A     FALSE
3        10        5         A      TRUE
4        80        8         A      TRUE
5       100       10         A     FALSE
6        70        7         B      TRUE
7        60        9         B     FALSE
8        40        1         C     FALSE
9        50        3         C      TRUE
10       90        6         C     FALSE
```

Thus, the order in which sorting variables are listed is important. To reverse the order for a particular variable, wrap its name with *desc*( )

```
arrange(dat, num.var1, desc(char.var3))

   num.var2 num.var1 char.var3 logi.var4
1        40        1         C     FALSE
2        90        2         A     FALSE
3        50        3         C      TRUE
4        20        4         A     FALSE
5        10        5         A      TRUE
6        90        6         C     FALSE
7        70        7         B      TRUE
8        80        8         A      TRUE
9        60        9         B     FALSE
10      100       10         A     FALSE
```

Slicing

The slicing functions are used to select rows based on their index, and not a specific quality of its elements. The *slice*( ) function is used for extracting, or filtering out, rows specifically by index. Its first argument is the dataframe/tibble to be sliced. The second argument is a vector of row indices[60].

```
slice(dat, c(1, 3, 4))  # dat[ c(1, 3, 4), ] is comparable.

  num.var2 num.var1 char.var3 logi.var4
1      100       10         A     FALSE
2       80        8         A      TRUE
3       70        7         B      TRUE
```

The *slice_head*( ) and *slice_tail*( ) functions select rows at the top(head) or bottom(tail). The argument **n** indicates the number of rows. Alternatively, the argument **prop** is the proportion to be selected.

```
slice_head(dat, n = 2)  # head(dat, n = 2 ) is comparable.

  num.var2 num.var1 char.var3 logi.var4
1      100       10         A     FALSE
2       60        9         B     FALSE
```

The *slice_max*( ) function will select the rows with the **n** largest values found in the **order_by** column. By default, the **with_ties** argument is set to *TRUE*. This could result in more than **n** rows being selected.

```
slice_max(dat, order_by = num.var2, n = 2, with_ties = TRUE)  # prop can be used instead of n

  num.var2 num.var1 char.var3 logi.var4
1      100       10         A     FALSE
2       90        6         C     FALSE
3       90        2         A     FALSE
```

The *slice_min*( ) function performs in a manner similar to *slice_max*( ), except it looks at the minimums.

Sampling

The *slice_sample*( ) function is used to generate a random sampling of the rows of a dataframe/tibble. Its first argument is the dataframe/tibble. Subsequently, either a sample size **n**, or a **prop**ortion to be selected is indicated. Sampling with(*TRUE*) or without(*FALSE*) replacement is indicated by the **replace** argument. If the row selection is to be equally likely, the **weight_by** argument is ignored. Otherwise, the length of the**weight_by** vector is equal to the number of rows[61].

```
slice_sample(dat, prop = 0.2, weight_by = 1:10, replace = TRUE)

  num.var2 num.var1 char.var3 logi.var4
1       90        6         C     FALSE
2       40        1         C     FALSE
```

slice_sample( ) could be considered equivalent to using the sample function to select from a vector of integers, and then using those integers as indices to be selected[62].

```
dat[sample(1:10, size = 2, replace = TRUE, prob = 1:10), ]

  num.var2 num.var1 char.var3 logi.var4
5       90        6         C     FALSE
9       90        2         A     FALSE
```

Two Important Facts

1. Once the dataframe/tibble is named within any of these **dplyr** functions, it is not necessary to use the $-notation to reference any of the enclosed variables. Simply stating a variables name suffices.

2. These **dplyr** functions don't modify the original dataframe/tibble. If the resulting dataframe/tibble is to be kept, an assignment is needed.

---

[60] Positive values are kept. Negative values are dropped
[61] Fractional values do not need to be given. The vector will automatically be standardized to sum to 1.
[62] These results are different because a random sample is taken. If a seed had been set just prior to each, the results would be the same.

Exercises

Lab

Chapter 43: dplyr - Columns

The **dplyr** package has several functions for selecting and manipulating the columns of a dataframe/tibble. To demonstrate these functions, the following tibble will be used.

```
num.var1 <- 1
num.var2 <- 10
char.var3 <- "A"
logi.var4 <- F
fact.var5 <- factor("Worst")
NANA.var6 <- NA
dat <- data.frame(num.var1, num.var2, char.var3, logi.var4, fact.var5, NANA.var6)
```

Selecting columns

The *select*( ) function is used for selecting columns. Its first argument is a dataframe/tibble. The remaining arguments are some representation of the columns to be selected. The most basic usage would be to select a single column.

```
select(dat, num.var1)  # Using the variable name

select(dat, 1)  # Using the column index

  num.var1
1        1
```

Notice that a dataframe is returned, and not a vector. As such, the *select*( ) is an analogue to $dat[1]$ or *dat["num.var1"]* and not $dat\$num.var1$ or $dat[[1]]$. The latter of which return a vector. As with the base *R* selections when using square brackets, a vector of indices can be used to make a selection. In addition to using the standard indexing of the columns, some specialized notation and/or helper functions can be used with *select*( ).

A consecutive series of variables can be selected by placing a colon between the first and last variables names[63]. The complement of a set of variables can be selected using an exclamation mark.

```
select(dat, num.var2:fact.var5)  # Replaces dat[2:5]

  num.var2 char.var3 logi.var4 fact.var5
1       10         A     FALSE     Worst

select(dat, !num.var2:fact.var5)  # Replaces dat[ -(2:5)]

  num.var1 NANA.var6
1        1        NA
```

The $last\_col(offset = 0L)$ function helps select the last column, or the column **offset** steps from the right.

```
select(dat, last_col(offset = 2))

  logi.var4
1     FALSE
```

The $starts\_with(match)$ helper function helps select columns with names that start with **match**. $ends\_with(match)$ is another version for matching the end of variable names.

```
select(dat, starts_with(match = "NU", ignore.case = TRUE))

  num.var1 num.var2
1        1       10
```

The $matches(match)$ helper function helps select columns that match the regular expression **match**[64]

```
select(dat, matches(match = "(m|A)\\."))

  num.var1 num.var2 NANA.var6
1        1       10        NA
```

[63] It may be the case that variable names are easier to detemine than the variables index.
[64] Other helpers are *num_range*( ), *all_of*( ), *any_of*( ), and *where*( ) can be viewed under the *select*( ) function's documentation.

Renaming Columns

Columns can be renamed with the *rename*( ) function. The first argument is the dataframe/tibble to be altered. Subsequent arguments set the new name equal to the old name as if they are variables, as opposed to using strings.

```
renamed.dat <- rename(dat, Empty.var6 = NANA.var6, FACT.var5 = fact.var5)
renamed.dat

  num.var1 num.var2 char.var3 logi.var4 FACT.var5 Empty.var6
1        1       10         A     FALSE     Worst         NA
```

Reordering Columns

The *relocate*( ) function is used to reorder the columns of a dataframe/tibble. The first argument is the dataframe/tible to be altered. The next arguments are the columns to be moved. Finally, a **.before** or **.after** argument indicates where the selected columns are to be moved. The helper functions that were used with *select*( ) can be used to set the value of **.before** or **.after**.

```
relocate(dat, num.var1, num.var2, .after = matches(match = "i\\."))

  char.var3 logi.var4 num.var1 num.var2 fact.var5 NANA.var6
1         A     FALSE        1       10     Worst        NA
```

Mutations

The *mutate*( ) function is used to create, delete, or modify columns of a dataframe/tibble. The first argument is the dataframe/tible to be altered. Subsequent arguments are used to create a new column, create multiple columns, or remove columns.

```
a.vector = "New"
mutate(dat, new.7 = a.vector, square.8 = num.var2^2)

  num.var1 num.var2 char.var3 logi.var4 fact.var5 NANA.var6 new.7 square.8
1        1       10         A     FALSE     Worst        NA   New      100
```

The **new.7** column was created from an external variable. The **square.8** column was created as a function of an existing variable[65] . It should be noted, that if a tibble with groupings is used, the groupings can affect some functions being applied to an existing column.

To attach one dataframes/tibble to another, set the second argument as the additional dataframe/tibble.

```
new.tib <- tibble(new.var1 = "!", new.var2 = "@")
mutate(dat, new.tib)

  num.var1 num.var2 char.var3 logi.var4 fact.var5 NANA.var6 new.var1 new.var2
1        1       10         A     FALSE     Worst        NA        !        @
```

Removing columns occurs with their name is set to NULL.

```
mutate(dat, num.var2 = NULL)

  num.var1 char.var3 logi.var4 fact.var5 NANA.var6
1        1         A     FALSE     Worst        NA
```

Two Important Facts

1. Once the dataframe/tibble is named within any of these **dplyr** functions, it is not necessary to use the $-notation to reference any of the enclosed variables. Simply stating a variables name suffices.

2. **dplyr** functions don't modify the original dataframe/tibble. To keep the resulting dataframe/tibble an assignment is needed.

[65] Some other functions that might be used: *lead*( ), *lag*( ), *dense_rank*( ), *cumsum*( ), *cummean*( ), *cummin*( ), *cummax*( ), and *if_else*( ).

Exercises

Lab

Chapter 44: dplyr - Joins

The **dplyr** package has several functions used for joining two dataframes/tibbles together. For example, information regarding a collection of people could be held in two different dataframes. Making a single new dataframe that contains information from the two existing ones can make analyzing the information easier. Two dataframes have been constructed to demonstrate these **join**ing functions: **dat.A** and **dat.B**.

```
dat.A

    NAMES A.Camera A.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3   Vesta  Graflex     3
4    Erin  Yashica     4

dat.B

    NAMES B.Camera B.Logic B.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3   Vesta  Yashica   FALSE           11
4  Ashley    Panon    TRUE           99
```

Examining these two dataframes, it should be apparent that there are a few ways that could be used to **join** these data sets together. These would match rows base upon

1. columns with the same name that represent the same variable in both dataframes. The **NAMES** variable is such a variable.
2. columns with different names that represent the same variable. The **A.Camera** and **B.Camera** are an example of this.
3. a mixture of columns from each of the previous two types.

In all three cases, the columns that match hold data that represents the same type of measurements, it was possibly stored under a different name in its particular dataframe. These columns that are used to produce matchs are called **keys**.

Decisions also need to be made about handling a value for a **key** variable does not have a match in both dataframes. *Erin* in the **NAMES** variable and $Graflex$ in the **A.Camera** variable are two such examples. Possible solutions include:

1. Keep only those rows that have a match in both dataframes. (**Inner Join**)
2. Base the match on one dataframe. All rows from first dataframe would be kept and infomation from the second dataframe would be added on to them. Any rows in the second dataframe that do not have a match in the first are lost. (**Left or Right Join**)
3. Make all the matches possible, and include the all the information that is not matched. (**Full Join**)

Inner Joins

The $inner\_join(\ \ )$ function joins two dataframes/tibble together combining those rows that have a match in both dataframes. Its first two arguments are **x** and **y** are the two dataframes/tibbles to be joined together[66]. By default, $inner\_join(\ \ )$ will build matches based upon columns that have matching names. In these dataframes, the columns **NAMES** appears in both. These columns (or this variable) will be used to produce the matches.

```
inner_join(x = dat.A, y = dat.B)

Joining with `by = join_by(NAMES)`

    NAMES A.Camera A.Int B.Camera B.Logic B.Double.Int
1 Minerva    Canon     1    Canon    TRUE           33
2    Odin     Fuji     2     Fuji    TRUE           22
3   Vesta  Graflex     3  Yashica   FALSE           11
```

The $inner\_join(\ \ )$ will produce a message listing all variables that were used for matches. In this example, the value "Erin" did not appear in both **NAMES** columns. So, $inner\_join(\ \ )$ omitted it from the result.

[66] Refer to **x** as the $left$ dataframe, and **y** as the right dataframe.

If matches should be produced by using columns that have different names, the **by** argument should be used. Pairing together a set of columns is done by writing each columns name as a string, and setting them equal to each other. Care should be taken to make sure that the first name is from the $left$ dataframe **x**, the second is from the $right$ dataframe **y**, and the entire statement is wrapped in the $c(\ \ )$ function. In this case, the matches are based upon the **Camera** values.

```
inner_join(x = dat.A, y = dat.B, by = c(A.Camera = "B.Camera"), suffix = c(".A", ".B"))

  NAMES.A A.Camera A.Int NAMES.B B.Logic B.Double.Int
1 Minerva    Canon     1 Minerva    TRUE           33
2    Odin     Fuji     2    Odin    TRUE           22
3    Erin  Yashica     4   Vesta   FALSE           11
```

In this case, "Graflex" does not have a match in both dataframes. Therefore, its information is omitted. Since the **NAMES** columns were not used to make matches, both columns were carried over to the final result. The **suffix** argument is a character vector that customizes columns from different dataframes, not used to make matches, that have the same name[67].

Joining two dataframes/tibbles can also be done by making matches based upon more than one set of columns. To do so, the pairs should be listed out, separated by a comma, using the **by** argument. When a name appears in both dataframes, it is enough to list the single name as a string.

```
inner_join(x = dat.A, y = dat.B, by = c(A.Camera = "B.Camera", "NAMES"))

    NAMES A.Camera A.Int B.Logic B.Double.Int
1 Minerva    Canon     1    TRUE           33
2    Odin     Fuji     2    TRUE           22
```

Using more than one matching variable, create more conditions on the matches. Therefore, joined dataframes made with more matching criteria will never have more rows that another with some of those criteria removed, provided the values in the key columns are unique. If there are repeats, this will not be the case.

(Antomym for Unique) Key Values

The values in a key variable (column) do not have to be unique. The following dataframes has multiple listings under the **NAMES** variable for "Odin".

```
dat.C

    NAMES C.Camera C.Logic C.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3    Odin  Yashica   FALSE           11
4   Vesta   Pentax   FALSE           99

dat.D

    NAMES D.Camera D.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3    Odin  Graflex     3
4    Odin  Yashica     4
5    Erin    Panon     5
```

Since the $inner\_join(\ \ )$ function keeps all rows that have a match, each row from $dat.C$ with "Odin" will be matched with a row from $dat.D$ that contains "Odin". This will result in Resulting in six "Odin" rows in the result. In general, if there are n occurrences of a value in a key column for the left dataframe and m for the right, the result will have $n \times m$ rows in the result for that specific value. One resulting row for each pairing of left and right rows.

```
inner_join(x = dat.C, y = dat.D, by = "NAMES")

    NAMES C.Camera C.Logic C.Double.Int D.Camera D.Int
1 Minerva    Canon    TRUE           33    Canon     1
2    Odin     Fuji    TRUE           22     Fuji     2
3    Odin     Fuji    TRUE           22  Graflex     3
4    Odin     Fuji    TRUE           22  Yashica     4
5    Odin  Yashica   FALSE           11     Fuji     2
6    Odin  Yashica   FALSE           11  Graflex     3
7    Odin  Yashica   FALSE           11  Yashica     4
```

[67] The default is to add ".x" to the left dataframe and ".y" to the right dataframe.

Exercises

Lab

Chapter 45: dplyr - Full Joins

Full Joins

The **dplyr** package has several functions used for joining two dataframes/tibbles together. One of these function is the $full\_join(\ )$ The $full\_join(\ )$ function keeps all the information from both dataframes. To demonstrate the $full\_join(\ )$ function, the following dataframes will be used: **L.dat** and **R.dat**.

```
L.dat

    NAMES L.Camera L.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3   Vesta  Graflex     3
4    Erin  Yashica     4

R.dat

    NAMES R.Camera R.Logic R.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3   Vesta  Yashica   FALSE           11
4  Ashley    Panon    TRUE           99
```

When joining two dataframes together, at least one pair of columns must be selected; One from each dataframe. A selected pair of columns, possibly with different names, should contain measurements on the same variable. Each column will have values that match at least one value in its paired column. There will also be some values that do not have matches. Unlike the $inner\_join(\ )$, which only maintains information from rows that have a match, the $full\_join(\ )$ keeps all the information whether is in a row with a match or not.

As was indicated, The $full\_join(\ )$ function joins two dataframes/tibbles together combining rows whether they have a match or not. Its first two arguments are **x** and **y** are the two dataframes/tibbles to be joined together[68]. By default, $full\_join(\ )$ will build matches based upon columns that have matching names. In these dataframes, the columns **NAMES** appears in both. This pair of columns (or this variable) will be used to produce the matches.

```
full_join(x = L.dat, y = R.dat)

    NAMES L.Camera L.Int R.Camera R.Logic R.Double.Int
1 Minerva    Canon     1    Canon    TRUE           33
2    Odin     Fuji     2     Fuji    TRUE           22
3   Vesta  Graflex     3  Yashica   FALSE           11
4    Erin  Yashica     4     <NA>      NA           NA
5  Ashley     <NA>    NA    Panon    TRUE           99
```

The first three rows of the result represent **NAMES** values that had a match in both dataframes. The last two rows are the interesting ones. They did not have matches. “Erin” only appeared in **L.dat** and “Ashley” only appeared in **R.dat**. In this cases, when the non-key columns were brought together, there was no values to put in certain places. In **L.dat**, there is no **R.Camera** value for “Erin”. So the **R.Camera** column in the joined result has no value to put in this spot. In situation like this, $full\_join(\ )$ will fill in such places with NA. (It will label them as missing values.) Notice that the order of the values in the resulting **NAMES** column matches those of the left dataframe **x = L.dat**, followed by unique values from the right dataframe *y* = R.dat

If matches should be produced by using columns that have different names, the **by** argument should be used. Pairing together a set of columns is done by writing each columns name as a string, and setting them equal to each other. Care should be taken to make sure that the first name is from the $left$ dataframe **x**, the second is from the $right$ dataframe **y**, and the entire statement is wrapped in the $c(\ )$ function. In this example, the matches are based upon the **Camera** values.

```
full_join(L.dat, R.dat, by = c(L.Camera = "R.Camera"), suffix = c(".A", ".B"))

  NAMES.A L.Camera L.Int NAMES.B R.Logic R.Double.Int
1 Minerva    Canon     1 Minerva    TRUE           33
2    Odin     Fuji     2    Odin    TRUE           22
3   Vesta  Graflex     3    <NA>      NA           NA
4    Erin  Yashica     4   Vesta   FALSE           11
5    <NA>    Panon    NA  Ashley    TRUE           99
```

[68] Refer to **x** as the $left$ dataframe, and **y** as the right dataframe.

In this example, the **L.Camera** and **R.Camera** columns were merged together under the **L.Camera** name. The initial four values are from the original **L.Camera** column, and those that follow after are the unmatched values from **R.Camera**. The corresponding row information was used to fill out the new dataframe. Two **NAMES** columns exist with different suffixes. This is because the **NAMES** columns were not joined into one. Their new distinct names come from the **suffix** argument set equal to a character vector of length two[69].

Joining two dataframes/tibbles can also be done by making matches based upon more than one set of columns. To do so, the pairs should be listed out, separated by a comma, using the **by** argument. When a name appears in both dataframes, it is enough to list the single name as a string.

```
full_join(x = L.dat, y = R.dat, by = c(L.Camera = "R.Camera", "NAMES"))

    NAMES L.Camera L.Int R.Logic R.Double.Int
1 Minerva    Canon     1    TRUE           33
2    Odin     Fuji     2    TRUE           22
3   Vesta  Graflex     3      NA           NA
4    Erin  Yashica     4      NA           NA
5   Vesta  Yashica    NA   FALSE           11
6  Ashley    Panon    NA    TRUE           99
```

Adding on the extra matching criteria, made it harder to find a match. This will produce more rows with missing values NA.

(Antomym for Unique) Key Values

The values in a key variable (column) do not have to be unique. The following dataframes has multiple listings under the **NAMES** variable for "Odin".

```
dat.C

    NAMES C.Camera C.Logic C.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3    Odin  Yashica   FALSE           11
4   Vesta   Pentax   FALSE           99

dat.D

    NAMES D.Camera D.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3    Odin  Graflex     3
4    Odin  Yashica     4
5    Erin    Panon     5
```

Since the $full\_join(\ \ )$ function keeps all rows, each row from $dat.C$ with "Odin" will be matched with a row from $dat.D$ that contains "Odin". This will result in Resulting in six "Odin" rows in the result. In general, if there are n occurrences of a value in a key column for the left dataframe and m for the right, the result will have $n \times m$ rows in the result for that specific value. One resulting row for each pairing of left and right rows. Additionally, in the resulting dataframe will be those rows without a match with missing values filled in as NA.

```
full_join(x = dat.C, y = dat.D, by = "NAMES")

    NAMES C.Camera C.Logic C.Double.Int D.Camera D.Int
1 Minerva    Canon    TRUE           33    Canon     1
2    Odin     Fuji    TRUE           22     Fuji     2
3    Odin     Fuji    TRUE           22  Graflex     3
4    Odin     Fuji    TRUE           22  Yashica     4
5    Odin  Yashica   FALSE           11     Fuji     2
6    Odin  Yashica   FALSE           11  Graflex     3
7    Odin  Yashica   FALSE           11  Yashica     4
8   Vesta   Pentax   FALSE           99     <NA>    NA
9    Erin     <NA>      NA           NA    Panon     5
```

[69] The default is to add ".x" to the left dataframe and ".y" to the right dataframe.

Exercises

Lab

Chapter 46: dplyr - Left Joins

The **dplyr** package has several functions used for joining two dataframes/tibbles together. For example, information regarding a collection of people could be held in two different dataframes. Making a single new dataframe that contains information from the two existing ones can make analyzing the information easier. Two dataframes have been constructed to demonstrate these **join**ing functions: **L.dat** and **R.dat**.

```
L.dat

    NAMES L.Camera L.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3   Vesta  Graflex     3
4    Erin  Yashica     4

R.dat

    NAMES R.Camera R.Logic R.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3   Vesta  Yashica   FALSE           11
4  Ashley    Panon    TRUE           99
```

The $left\_join(\ \ )$ and $right\_join(\ \ )$ functions first two arguments are dataframes. The first argument **x** will be called the left dataframe. The sec argument **y** will be called the right dataframe.

Left Joins

When joining two dataframes together, at least one pair of columns must be selected; One from each dataframe. A selected pair of columns, possibly with different names, should contain measurements on the same variable. Each column will have values that match at least one value in its paired column. There will also be some values that do not have matches. The $left\_join(\ \ )$ function finds rows in the right dataframe that are matched with the left. Then it extends the existing rows in the left dataframe with data taken from the right or with NA values when no match exists. Thus the left dataframe remains intact, while unmatched information in the right is lost.

By default, $left\_join(\ \ )$ will build matches based upon columns that have matching names. In these dataframes, the columns **NAMES** appears in both. This pair of columns (or this variable) will be used to produce the matches.

```
left_join(L.dat, R.dat)

    NAMES L.Camera L.Int R.Camera R.Logic R.Double.Int
1 Minerva    Canon     1    Canon    TRUE           33
2    Odin     Fuji     2     Fuji    TRUE           22
3   Vesta  Graflex     3  Yashica   FALSE           11
4    Erin  Yashica     4     <NA>      NA           NA
```

The **NAMES** column in the result only contains values already existing in the left dataframe. The difference from other joins ($inner\_join(\ \ )$ or $full\_join(\ \ )$) is that:

1. if an $inner\_join(\ \ )$ had been used, "Erin" would not have been included. It did not occur in both **NAMES** columns.
2. if a $full\_join(\ \ )$ had been used, both "Erin" and "Ashley" would have been included. They occured in at least one of the **NAMES** columns.

Notice, that the "Erin" row is extended with NA values. Without a match in the right dataframe, there are no values to fill in for the remaining columns.

If matches should be produced by using columns that have different names, the **by** argument should be used. Pairing together a set of columns is done by writing each columns name as a string, and setting them equal to each other. Care should be taken to make sure that the first name is from the $left$ dataframe **x**, the second is from the $right$ dataframe **y**, and the entire statement is wrapped in the $c(\ \ )$ function. In this example, the matches are based upon the **Camera** values.

```
left_join(x = L.dat, y = R.dat, by = c(L.Camera = "R.Camera"), suffix = c(".A", ".B"))

  NAMES.A L.Camera L.Int NAMES.B R.Logic R.Double.Int
1 Minerva    Canon     1 Minerva    TRUE           33
2    Odin     Fuji     2    Odin    TRUE           22
3   Vesta  Graflex     3    <NA>      NA           NA
4    Erin  Yashica     4   Vesta   FALSE           11
```

In this example, first, **L.dat** was copied over. Second, each value in **L.Camera** was mapped to any matching values in **R.Camera**. Thirds, rows in **R.dat** that have a match are appended to the end of their partner in the left dataframe[70]. Lastly, any empty places are filled in NA[71].

Pay attention to the fact that two **NAMES** columns exist with different suffixes. This is because the **NAMES** columns were not joined into one. Their new distinct names come from the **suffix** argument set equal to a character vector of length two[72].

Joining two dataframes/tibbles can also be done by making matches based upon more than one set of columns. To do so, the pairs should be listed out, separated by a comma, using the **by** argument. When a name appears in both dataframes, it is enough to list the single name as a string.

```
left_join(x = L.dat, y = R.dat, by = c(L.Camera = "R.Camera", "NAMES"))

    NAMES L.Camera L.Int R.Logic R.Double.Int
1 Minerva    Canon     1    TRUE           33
2    Odin     Fuji     2    TRUE           22
3   Vesta  Graflex     3      NA           NA
4    Erin  Yashica     4      NA           NA
```

This result has fewer columns than the previous, because there were more pairings included. Also, **L.dat** is clearly visible in the first three columns.

(Antomym for Unique) Key Values

The values in a key variable (column) do not have to be unique. The following dataframes has multiple listings under the **NAMES** variable for "Odin".

```
dat.C

    NAMES C.Camera C.Logic C.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3    Odin  Yashica   FALSE           11
4   Vesta   Pentax   FALSE           99

dat.D

    NAMES D.Camera D.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3    Odin  Graflex     3
4    Odin  Yashica     4
5    Erin    Panon     5
```

Since the $left\_join(\quad)$ function keeps only rows from the left dataframe, each row from $dat.D$ with "Odin" will be matched to each row in $dat.C$ that contains "Odin". This will result in six "Odin" rows. In general, if there are n occurrences of a value in a key column for the left dataframe and m for the right, the result will have $n \times m$ rows in the result for that specific value. One resulting row for each pairing of left and right rows.

```
left_join(x = dat.C, y = dat.D, by = "NAMES")

    NAMES C.Camera C.Logic C.Double.Int D.Camera D.Int
1 Minerva    Canon    TRUE           33    Canon     1
2    Odin     Fuji    TRUE           22     Fuji     2
3    Odin     Fuji    TRUE           22  Graflex     3
4    Odin     Fuji    TRUE           22  Yashica     4
5    Odin  Yashica   FALSE           11     Fuji     2
6    Odin  Yashica   FALSE           11  Graflex     3
7    Odin  Yashica   FALSE           11  Yashica     4
8   Vesta   Pentax   FALSE           99     <NA>    NA
```

[70] The non-key variables from **R.dat** are added onto the copy of **L.dat**.
[71] This is not how the code for the function works, but it will give you an intuition for what is happening.
[72] The default is to add ".x" to the left dataframe and ".y" to the right dataframe.

Exercises

Lab

Chapter 47: dplyr - Right Joins

The **dplyr** package has several functions used for joining two dataframes/tibbles together. For example, information regarding a collection of people could be held in two different dataframes. Making a single new dataframe that contains information from the two existing ones can make analyzing the information easier. Two dataframes have been constructed to demonstrate these **join**ing functions: **L.dat** and **R.dat**.

```
L.dat

    NAMES L.Camera L.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3   Vesta  Graflex     3
4    Erin  Yashica     4

R.dat

    NAMES R.Camera R.Logic R.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3   Vesta  Yashica   FALSE           11
4  Ashley    Panon    TRUE           99
```

The $right\_join(\quad)$ and $left\_join(\quad)$ functions first two arguments are dataframes. The first argument **x** will be called the left dataframe. The second argument **y** will be called the right dataframe.

Right Joins

When joining two dataframes together, at least one pair of columns must be selected; one from each dataframe. A selected pair of columns, possibly with different names, should contain measurements on the same variable. Each column will have values that match at least one value in its paired column. There will also be some values that do not have matches. The $right\_join(\quad)$ function finds rows in the left dataframe that are matched with the right. Then it expands the existing rows in the right dataframe with data taken from the left or with NA values when no match exists. Thus the right dataframe remains intact, while unmatched information in the left is lost.

By default, $right\_join(\quad)$ will build matches based upon columns that have matching names. In these dataframes, the columns **NAMES** appears in both. This pair of columns (or this variable) will be used to produce the matches.

```
right_join(L.dat, R.dat)

    NAMES L.Camera L.Int R.Camera R.Logic R.Double.Int
1 Minerva    Canon     1    Canon    TRUE           33
2    Odin     Fuji     2     Fuji    TRUE           22
3   Vesta  Graflex     3  Yashica   FALSE           11
4  Ashley     <NA>    NA    Panon    TRUE           99
```

The **NAMES** column in the result only contains values already existing in the left dataframe. The difference from other joins ($inner\_join(\quad)$ or $full\_join(\quad)$) is that:

1. if an $inner\_join(\quad)$ had been used, “Ashley” would not have been included. It did not occur in both **NAMES** columns.
2. if a $full\_join(\quad)$ had been used, both “Ashley” and “Erin” would have been included. They occurred in at least one of the **NAMES** columns.

Notice, that the “Ashley” row is expanded with NA values. Without a match in the left dataframe, there are no values to fill in for these extra columns.

If matches should be produced by using columns that have different names, the **by** argument should be used. Pairing together a set of columns is done by writing each columns name as a string, and setting them equal to each other. Care should be taken to make sure that the first name is from the $left$ dataframe **x**, the second is from the $right$ dataframe **y**. The entire statement is wrapped in the $c(\quad)$ function. In this example, the matches are based upon the **Camera** values.

```
right_join(L.dat, R.dat, by = c(L.Camera = "R.Camera"), suffix = c(".A", ".B"))

  NAMES.A L.Camera L.Int NAMES.B R.Logic R.Double.Int
1 Minerva    Canon     1 Minerva    TRUE           33
2    Odin     Fuji     2    Odin    TRUE           22
3    Erin  Yashica     4   Vesta   FALSE           11
4    <NA>    Panon    NA  Ashley    TRUE           99
```

In this example, first, **R.dat** was copied over. Second, each value in **R.Camera** was mapped to any matching values in **L.Camera**. Third, rows in **L.dat** that have a match are paste merged with their partner in the right dataframe[73]. Lastly, any empty places are filled in NA[74].

Pay attention to the fact the names of some of columns. Two **NAMES** columns exist with different suffixes. This is because the **NAMES** columns were not joined into one. Their new distinct names come from the **suffix** argument set equal to a character vector of length two[75]. Even though only rows from the right dataframe are to be kept, the name of the column that performed the match was taken from the left matrix. The ordering of the columns is an attempt to maintain the order of columns from the left dataframe on the left and the right dataframe on the right.

Joining two dataframes/tibbles can also be done by making matches based upon more than one set of columns. To do so, the pairs should be listed out, separated by a comma, using the **by** argument. When a name appears in both dataframes, it is enough to list the single name as a string.

```
right_join(x = L.dat, y = R.dat, by = c(L.Camera = "R.Camera", "NAMES"))

    NAMES L.Camera L.Int R.Logic R.Double.Int
1 Minerva    Canon     1    TRUE           33
2    Odin     Fuji     2    TRUE           22
3   Vesta  Yashica    NA   FALSE           11
4  Ashley    Panon    NA    TRUE           99
```

This result has fewer columns than the previous, because there were more pairings included. Also, **R.dat** is clearly visible in the first three columns.

(Antomym for Unique) Key Values

The values in a key variable (column) do not have to be unique. The following dataframes has multiple listings under the **NAMES** variable for “Odin”.

```
dat.C

    NAMES C.Camera C.Logic C.Double.Int
1 Minerva    Canon    TRUE           33
2    Odin     Fuji    TRUE           22
3    Odin  Yashica   FALSE           11
4   Vesta   Pentax   FALSE           99

dat.D

    NAMES D.Camera D.Int
1 Minerva    Canon     1
2    Odin     Fuji     2
3    Odin  Graflex     3
4    Odin  Yashica     4
5    Erin    Panon     5
```

Since the $right\_join(\quad)$ function keeps only rows from the left dataframe, each row from $dat.D$ with “Odin” will be matched to each row in $dat.C$ that contains “Odin”. This will result in six “Odin” rows.

```
right_join(x = dat.C, y = dat.D, by = "NAMES")

    NAMES C.Camera C.Logic C.Double.Int D.Camera D.Int
1 Minerva    Canon    TRUE           33    Canon     1
2    Odin     Fuji    TRUE           22     Fuji     2
3    Odin     Fuji    TRUE           22  Graflex     3
4    Odin     Fuji    TRUE           22  Yashica     4
5    Odin  Yashica   FALSE           11     Fuji     2
6    Odin  Yashica   FALSE           11  Graflex     3
7    Odin  Yashica   FALSE           11  Yashica     4
8    Erin     <NA>      NA           NA    Panon     5
```

[73] The non-key columns from **L.dat** are added onto the copy of **R.dat**.
[74] This is not how the code for the function works, but it will give you an intuition for what is happening.
[75] The default is to add “.x” to the left dataframe and “.y” to the right dataframe.

Exercises

Lab

Chapter 48: Bibliography (Not Finished Yet)

Sanchez, G. 2013. *Handling and Processing Strings in r*. Trowchez Editions.

Wickham, Hadley. 2016. *Ggplot2*. Springer.

Wickham, Hadley, Romain François, Lionel Henry, and Kirill Müller. 2021. *Dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr.