Lesson 2 - Basic Life Skills: How to Read, Write, and Manipulate (Your Data)

Contents

A quick intro to the Introduction to R for Data Science 1

Reading in data & writing data **3**

Dataset: Sequencing the V3-V5 hypervariable regions of the 16S rRNA gene 3

tsv/csv files (utils and readr) 3

Excel spreadsheets 6

A quick intro to the dplyr package **8**

A reminder/warning about vectors 11

Resources **19**

Post-Lesson Assessment **19**

Appendix **20**

Googlesheets 20

A quick intro to the Introduction to R for Data Science

This ‘Introduction to R for Data Science’ is brought to you by the Centre for the Analysis of Genome Evolution & Function’s (CAGEF) bioinformatics training initiative. This CSB1020 was developed based on feedback on the needs and interests of the Department of Cell & Systems Biology and the Department of Ecology and Evolutionary Biology.

This lesson is the second in a 6-part series. The idea is that at the end of the series, you will be able to import and manipulate your data, make exploratory plots, perform some basic statistical tests, test a regression model, and make some even prettier plots and documents to share your results.

How do we get there? Today we are going to be learning about how to subset and filter our data and perform all of the manipulations one has to do in daily coding life. We will learn about tidy data and how it makes data analysis less of a pain. We will perform some basic statistics on our newly transformed dataset. Next week we will learn how to tidy our data, subset and merge data and generate descriptive statistics. The next lesson will be data cleaning and string manipulation; this is really the battleground of coding - getting your data into the format where you can analyse it. After that, we will make all sorts of plots - from simple data exploration to interactive plots - this is always a fun lesson.

The structure of the class is a code-along style. It is hands on. The lecture AND code we are going through are available on GitHub for download at https://github.com/eacton/CAGEF, so you can spend the time coding and not taking notes. As we go along, there will be some challenge questions and multiple choice questions on Socrative. At the end of the class if you could please fill out a post-lesson survey (https://www.surveymonkey.com/r/SMGKMCS), it will help me further develop this course and would be greatly appreciated.

Data Wrangling Is The
HARDEST
Part
Photo by Pascal under Creative Commons (CC0 1.0) - Adaptation by LutzFinger.com

Figure 1:

Highlighting

`grey background` - a package, function, code, command or directory
italics - an important term or concept or an individual file or folder
bold - heading or a term that is being defined
blue text - named or unnamed hyperlink

Packages Used in This Lesson

The following packages are used in this lesson:

`dplyr`
`readr`
`readxl`
`tibble`
(`googlesheets`- optional)

Please install and load these packages for the lesson. In this document I will load each package separately, but I will not be reminding you to install the package. Remember: these packages may be from CRAN OR Bioconductor.

Data Files Used in This Lesson

-ENV_pitlatrine.csv
-SPE_pitlatrine.csv
-books.alpha.xlsx
-adult_income.tsv

These files can be downloaded at https://github.com/eacton/CAGEF/tree/master/Lesson_2/data. Right-click on the filename and select 'Save Link As...' to save the file locally. The files should be saved in the same folder you plan on using for your R script for this lesson.

Or click on the blue hyperlink at the start of the README.md at https://github.com/eacton/CAGEF/tree/master/Lesson_2 to download the entire folder at DownGit.

Objective: At the end of this session you will be able to use the dplyr package to subset and manipulate your data and to perform simple calculations. You will be able to import data into R (tsv, csv, xls(x), googlesheets) and export your data again.

Reading in data & writing data

Dataset: Sequencing the V3-V5 hypervariable regions of the 16S rRNA gene

16S rRNA sequencing of 30 latrines from Tanzania and Vietnam at different depths (multiples of 20cm). Microbial abundance is represented in Operational Taxonomic Units (OTUs). Operational Taxonomic Units (OTUs) are groups of organisms defined by a specified level of DNA sequence similarity at a marker gene (e.g. 97% similarity at the V4 hypervariable region of the 16S rRNA gene). Intrinsic environmental factors such as pH, temperature, organic matter composition were also recorded.

We have 2 csv files:

1. A metadata file (Naming conventions: [Country_LatrineNo_Depth]) with sample names and environmental variables.

2. OTU abundance table.

B Torondel, JHJ Ensink, O Gunvirusdu, UZ Ijaz, J Parkhill, F Abdelahi, V-A Nguyen, S Sudgen, W Gibson, AW Walker, and C Quince. Assessment of the influence of intrinsic environmental and geographical factors on the bacterial ecology of pit latrines Microbial Biotechnology, 9(2):209-223, 2016. DOI:10.1111/1751-7915.12334

---

tsv/csv files (utils and readr)

Let's read our metadata file into R. While we do these exercises, we are going to become friends with the Help menu. Let's start by using the `read.table()` function which takes in the path to our file.

```r
meta <- read.table(file = "data/ENV_pitlatrine.csv")
```

To see the result, we can look at 'meta' in the Environment pane and see that there are 82 observations of one variable. If we click on the arrow next to 'meta' we can now see that we have a column 'V1' and the variable type is a factor. We can also hover the mouse over the column name for this information.

We can click on 'meta' in the Environment pane or type `View(meta)` in the Console to open a spreadsheet-like view of 'meta' in a new tab. This is pretty ugly looking. Why?

In the help file the default `sep` or what is separating columns is expected to be a space. We need to use specify a comma instead.

```r
meta <- read.table(file = "data/ENV_pitlatrine.csv", sep = ",")
```

We can either inspect meta again in the spreadsheet-like view or `head()` will show us the first 6 rows of our data frame. `tail()` would show the last 6 rows.

```r
head(meta)
```

```
##         V1   V2   V3    V4    V5   V6   V7   V8         V9 V10  V11   V12
## 1 Samples   pH Temp    TS    VS  VFA CODt CODs perCODsbyt NH4 Prot Carbo
## 2   T_2_1 7.82 25.1 14.53 71.33   71  874  311         36 3.3 35.4    22
## 3  T_2_10 9.08 24.2 37.76 31.52    2  102    9          9 1.2 18.4    43
## 4  T_2_12 8.84 25.1 71.11  5.94    1   35    4         10 0.5    0    17
## 5   T_2_2 6.49 29.6 13.91 64.93  3.7  389  180         46 6.2 29.3    25
## 6   T_2_3 6.46 27.9 29.45 26.85 27.5  161   35         22 2.4 19.4    31
```

Better. Looking in the Environment pane 'meta' now has 82 rows and 12 columns, meaning our columns are separated appropriately, but what about our column titles? We want the names in the 1st row to be our column headings.

```
meta <- read.table(file = "data/ENV_pitlatrine.csv", sep = ",", header = TRUE)
head(meta)
```

```
##   Samples   pH Temp    TS    VS  VFA CODt CODs perCODsbyt NH4 Prot Carbo
## 1   T_2_1 7.82 25.1 14.53 71.33 71.0  874  311         36 3.3 35.4    22
## 2  T_2_10 9.08 24.2 37.76 31.52  2.0  102    9          9 1.2 18.4    43
## 3  T_2_12 8.84 25.1 71.11  5.94  1.0   35    4         10 0.5  0.0    17
## 4   T_2_2 6.49 29.6 13.91 64.93  3.7  389  180         46 6.2 29.3    25
## 5   T_2_3 6.46 27.9 29.45 26.85 27.5  161   35         22 2.4 19.4    31
## 6   T_2_6 7.69 28.7 65.52  7.03  1.5   57    3          6 0.8  0.0    14
```

We use `header=TRUE` to specify that the first row we read in will be the column headers. We now have 81 rows and 12 columns.

These samples are not replicates. Each represents a combination of a different country, latrine, and depth. In this case, we might prefer to have Samples as character data, not a factor. (Note: TRUE and FALSE can be abbreviated as T and F)

```
meta <- read.table(file = "data/ENV_pitlatrine.csv", sep = ",", header = T, stringsAsFactors = F)
str(meta)
```

```
## 'data.frame':    81 obs. of  12 variables:
##  $ Samples   : chr  "T_2_1" "T_2_10" "T_2_12" "T_2_2" ...
##  $ pH        : num  7.82 9.08 8.84 6.49 6.46 7.69 7.48 7.6 7.55 7.68 ...
##  $ Temp      : num  25.1 24.2 25.1 29.6 27.9 28.7 29.8 25 28.8 28.9 ...
##  $ TS        : num  14.5 37.8 71.1 13.9 29.4 ...
##  $ VS        : num  71.33 31.52 5.94 64.93 26.85 ...
##  $ VFA       : num  71 2 1 3.7 27.5 1.5 1.1 1.1 30.9 24.2 ...
##  $ CODt      : int  874 102 35 389 161 57 107 62 384 372 ...
##  $ CODs      : int  311 9 4 180 35 3 9 8 57 57 ...
##  $ perCODsbyt: int  36 9 10 46 22 6 8 13 15 15 ...
##  $ NH4       : num  3.3 1.2 0.5 6.2 2.4 0.8 0.7 0.9 21.6 20.4 ...
##  $ Prot      : num  35.4 18.4 0 29.3 19.4 0 14.1 7.6 33.1 44.5 ...
##  $ Carbo     : num  22 43 17 25 31 14 28 28 47 48 ...
```

Don't forget! `str()` is still the best way to look at your data structure without taking your hands off the keyboard.

This is as good a time as any to mention that you do not have to specify the name for every argument in a function. If unnamed, arguments are assumed to be input in the order they appear in the function documentation. You can place arguments out of order if they are named (`header = TRUE` is the second argument to `read.table()`, but can be placed in the 3rd position above due to being named).

Only leave out argument names that are fairly obvious. In the example below, we can easily assume what character string is supposed to be the file, but there are many TRUE/FALSE argument options and it is not obvious what `TRUE` is referring to without looking up the documentation. I would also retain `sep=` as there are other quoted argument options, and it is better to be explicitly obvious than implicitly so (ie. the file is a .csv so it is comma separated).

```
meta <- read.table("data/ENV_pitlatrine.csv", TRUE, ",", stringsAsFactors = FALSE)
```

Challenge

Read our metadata table into R using any previously unused function under Usage in the `read.table()` Help menu. Save your result in a variable called 'dat'.

Hopefully this exercise forced you to look at the differences in the default values of the parameters in these different functions. I suggest that you keep these in mind as we look at some functions that were aimed to make a few typing shortcuts. Let's load the `readr` library.

```
library(readr)
```

Use `read_csv()` to read in your metadata file. What is different from `read.csv()`?

```
meta <- read_csv("data/ENV_pitlatrine.csv")
```

```
## Parsed with column specification:
## cols(
##   Samples = col_character(),
##   pH = col_double(),
##   Temp = col_double(),
##   TS = col_double(),
##   VS = col_double(),
##   VFA = col_double(),
##   CODt = col_integer(),
##   CODs = col_integer(),
##   perCODsbyt = col_integer(),
##   NH4 = col_double(),
##   Prot = col_double(),
##   Carbo = col_double()
## )
```

```
head(meta)
```

```
## # A tibble: 6 x 12
##   Samples    pH  Temp    TS    VS   VFA  CODt  CODs perCODsbyt   NH4  Prot
##   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <int> <int>      <int> <dbl> <dbl>
## 1 T_2_1    7.82  25.1  14.5 71.3   71     874   311         36   3.3  35.4
## 2 T_2_10   9.08  24.2  37.8 31.5    2     102     9          9   1.2  18.4
## 3 T_2_12   8.84  25.1  71.1  5.94   1      35     4         10   0.5   0
## 4 T_2_2    6.49  29.6  13.9 64.9    3.7   389   180         46   6.2  29.3
## 5 T_2_3    6.46  27.9  29.4 26.8   27.5   161    35         22   2.4  19.4
## 6 T_2_6    7.69  28.7  65.5  7.03   1.5    57     3          6   0.8   0
## # ... with 1 more variable: Carbo <dbl>
```

Note that `readr` tells you exactly how it parsed your file, and how each column is encoded. It also saves us from typing `stringsAsFactors = FALSE`. Every. Time. The syntax is a bit different - for example, `col_names = TRUE` instead of `header = TRUE`, `col_types = 'c'` instead of `colClasses = 'character'`. It is a matter of personal preference what you want to use. If you have a csv file you are really struggling with, I would try `read_csv` as it has some pretty sensible defaults.

Excel spreadsheets

But what happens if we have a good, old-fashioned excel file? The `readxl` package will recognize both xls and xlsx files. It expects tabular data.

```
library(readxl)

head(read_excel("data/books_alpha.xlsx"))
```

```
## # A tibble: 6 x 16
##   `UK's most borr~ `Desert Island ~ `Pulitzer Prize~ `Askmetafilter.~
##   <chr>            <chr>            <chr>            <chr>
## 1 July 2009-June ~ (Feb 2008-Feb 2~ (1918-2010)      http://ask.meta~
## 2 206 bones        Anna Karenina    A Bell for Adano 1984.0
## 3 7th heaven       Blake            A Confederacy o~ Aesop's Fables
## 4 7th heaven       Breakfast of Ch~ A Death in the ~ Against the Gra~
## 5 8th confession   Decline and Fall A Fable          Alice's Adventu~
## 6 A darker domain  His Dark Materi~ A Good Scent fr~ An Explanation ~
## # ... with 12 more variables: `LibraryThing.com (top 50)` <chr>, `World
## #   Book Day Poll (top 100)` <chr>, `Telegraph 100 Novels Everyone Should
## #   Read` <chr>, `Goodreads.com Books That Everyone Should Read At Least
## #   Once (top 100)` <chr>, `Bspcn.com 30 Books Everyone Should Read Before
## #   They're 30` <chr>, `Guardian 1000 Novels Everyone Must Read` <chr>,
## #   `Bighow.com 100 Greatest Books of All Time Everyone Must Read` <chr>,
## #   `The Best 100 Lists Top 100 Novels of All Time` <chr>, `Man Booker
## #   Prize winners` <chr>, `Oprah's Book Club List` <chr>, `1001 Books You
## #   Should Read Before You Die (Cassell, 2005)` <chr>, `Author's own top
## #   five...` <lgl>
```

This doesn't look like a workbook. Why not? The `read_excel()` function defaults to reading in the first worksheet. You can specify which sheet you want to read in by position or name. Let's see what the name of our sheets are.

```
excel_sheets("data/books_alpha.xlsx")
```

```
## [1] "Lists"             "All alphabetised" "Top titles"
## [4] "dropoff"
```

Note that the argument to both of these functions was the path to our data sheet. We can save this path into a variable to save ourselves some typing.

```
path <- "data/books_alpha.xlsx"
```

It is possible to subset from a sheet by specifying cell numbers or ranges. Here we are grabbing sheet 1, and subsetting cells over 2 columns - C1:D9.

```
read_excel(path, sheet = 1, range = "C1:D9")
```

```
## # A tibble: 8 x 2
##   `Pulitzer Prize winners (Fiction~ `Askmetafilter.com Books Everyone Sho~
##   <chr>                             <chr>
## 1 (1918-2010)                       http://ask.metafilter.com/42616/A-boo~
## 2 A Bell for Adano                  1984.0
## 3 A Confederacy of Dunces           Aesop's Fables
## 4 A Death in the Family             Against the Grain
## 5 A Fable                           Alice's Adventures in Wonderland
## 6 A Good Scent from a Strange Moun~ An Explanation of the Birds
## 7 A Summons to Memphis              Animal Farm
## 8 A Thousand Acres                  Atlas Shrugged
```

We could alternatively specify the sheet by name. This is how you would simply grab rows.

```r
read_excel(path, sheet = "Top titles", range = cell_rows(1:9))
```

```
## # A tibble: 8 x 2
##   Title                                `No of mentions`
##   <chr>                                           <dbl>
## 1 To Kill a Mockingbird                              11
## 2 1984.0                                              9
## 3 Catch-22                                            9
## 4 Crime and Punishment                                9
## 5 One Hundred Years of Solitude                       9
## 6 The Catcher in the Rye                              9
## 7 The Great Gatsby                                    9
## 8 The Hitchhiker's Guide to the Galaxy                9
```

And likewise, how you would subset just columns from the same sheet.

```r
read_excel(path, sheet = 1, range = cell_cols("B:D"))
```

```
## # A tibble: 113 x 3
##    `Desert Island Disc~ `Pulitzer Prize winner~ `Askmetafilter.com Books ~
##    <chr>                <chr>                   <chr>
##  1 (Feb 2008-Feb 2011)  (1918-2010)             http://ask.metafilter.com~
##  2 Anna Karenina        A Bell for Adano        1984.0
##  3 Blake                A Confederacy of Dunces Aesop's Fables
##  4 Breakfast of Champi~ A Death in the Family   Against the Grain
##  5 Decline and Fall     A Fable                 Alice's Adventures in Won~
##  6 His Dark Materials   A Good Scent from a St~ An Explanation of the Bir~
##  7 Jonathan Strange an~ A Summons to Memphis    Animal Farm
##  8 Legacy               A Thousand Acres        Atlas Shrugged
##  9 Les Misérables       Advise and Consent      Aunt Julia and the Script~
## 10 Letters to a Young ~ Alice Adams             Blindness
## # ... with 103 more rows
```

You can also use a list version of the apply function `lapply()` to read in all sheets at once. Each sheet will be stored as a data frame inside of a list object. If you remember, `apply()` took in a matrix, a row/column specification (MARGIN), and a function. `lapply()` takes in a list (which does not have rows and columns) and a function.

While so far we are used to functions finding our variables globally (in the global environment), `lapply()` is looking locally (within the function) and so we need to explicitly provide our path. We will get more into local vs global variables in our functions lesson. For now, I just want you to be able to read in all worksheets from an excel workbook.

```r
ex <- lapply(excel_sheets(path), read_excel, path = path)

str(ex)
```

```
## List of 4
##  $ :Classes 'tbl_df', 'tbl' and 'data.frame':    999 obs. of  16 variables:
##   ..$ UK's most borrowed library books                                  : chr [1:999] "July 2009-
##   ..$ Desert Island Discs book choices                                  : chr [1:999] "(Feb 2008-
##   ..$ Pulitzer Prize winners (Fiction 1948-, Novel pre-1948)            : chr [1:999] "(1918-2010
##   ..$ Askmetafilter.com Books Everyone Should Read                      : chr [1:999] "http://ask
##   ..$ LibraryThing.com (top 50)                                         : chr [1:999] "http://www
##   ..$ World Book Day Poll (top 100)                                  : chr [1:999] "Latest rel
##   ..$ Telegraph 100 Novels Everyone Should Read                      : chr [1:999] "(2009) htt
##   ..$ Goodreads.com Books That Everyone Should Read At Least Once (top 100): chr [1:999] "(Created J
##   ..$ Bspcn.com 30 Books Everyone Should Read Before They're 30      : chr [1:999] "(2010) htt
##   ..$ Guardian 1000 Novels Everyone Must Read                        : chr [1:999] "(2009) htt
##   ..$ Bighow.com 100 Greatest Books of All Time Everyone Must Read   : chr [1:999] "(2010) htt
##   ..$ The Best 100 Lists Top 100 Novels of All Time                  : chr [1:999] "http://www
##   ..$ Man Booker Prize winners                                       : chr [1:999] "(1969-2010
##   ..$ Oprah's Book Club List                                         : chr [1:999] "http://www
##   ..$ 1001 Books You Should Read Before You Die (Cassell, 2005)      : chr [1:999] "we didn't
##   ..$ Author's own top five...                                       : logi [1:999] NA NA NA N
##  $ :Classes 'tbl_df', 'tbl' and 'data.frame':    2003 obs. of  2 variables:
##   ..$ Title         : chr [1:2003] "1974.0" "1977.0" "1984.0" "1984.0" ...
##   ..$ No of mentions: num [1:2003] NA NA NA NA NA NA NA NA NA NA ...
##  $ :Classes 'tbl_df', 'tbl' and 'data.frame':    246 obs. of  2 variables:
##   ..$ Title         : chr [1:246] "To Kill a Mockingbird" "1984.0" "Catch-22" "Crime and Punishment"
##   ..$ No of mentions: num [1:246] 11 9 9 9 9 9 9 9 8 8 ...
##  $ :Classes 'tbl_df', 'tbl' and 'data.frame':    0 obs. of  1 variable:
##   ..$ add to Books Everyone Should Read http://www.brainpickings.org/index.php/2012/01/30/writers-to
```

We now have a list object with each worksheet being one item in the list.

You can subset the sheet you would like to work with using the syntax 'list[[x]]' and store it as a variable using `data.frame()` (or work with all sheets at once - see the `purrr` package for working with list objects).

```r
str(ex[[3]])
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    246 obs. of  2 variables:
##  $ Title         : chr  "To Kill a Mockingbird" "1984.0" "Catch-22" "Crime and Punishment" ...
##  $ No of mentions: num  11 9 9 9 9 9 9 9 8 8 ...
```

```r
excel3 <- data.frame(ex[[3]])
```

At this point, you will be able to use your excel worksheet as a normal data frame in R.

If you are a googlesheets person, there is a package (surprisingly called 'googlesheets') that will allow you to get your worksheets in and out of R. Checkout the appendix at the end of this lesson for a brief tutorial.

Challenge

Read in the 'adult_income.tsv' dataset. How many rows and columns in the dataset? Can you delete the last column? What flavour of variables are there? Change a column name. Write the data to a csv file.

Bonus:

Try reading in one of your own datasets. Write it to a different file format.

A quick intro to the dplyr package

To be able to answer any questions with our data, we need the ability to select and filter parts of our data.

The `dplyr` package was made by Hadley Wickham to help make data frame manipulation easier. It has 5 major functions:

1. `filter()` - subsets your data frame by row
2. `select()` - subsets your data frame by columns

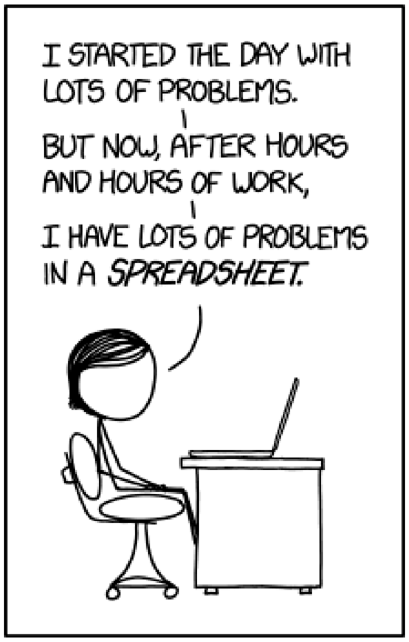

I STARTED THE DAY WITH LOTS OF PROBLEMS.
BUT NOW, AFTER HOURS AND HOURS OF WORK,
I HAVE LOTS OF PROBLEMS IN A *SPREADSHEET*.

Figure 2:

3. `arrange()` - orders your data frame alphabetically or numerically by ascending or descending variables
4. `mutate()`, `transmute()` - create a new column of data
5. `summarize()` - reduces data to summary values (for example using `mean()`, `sd()`, `min()`, `quantile()`, etc)

Let's load the library.

```
library(dplyr)
library(tibble)
```

Let's read in our dataset, store it in a variable, and check out the structure.

```
dat <- read_csv("data/SPE_pitlatrine.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   Taxa = col_character()
## )

## See spec(...) for full column specifications.
```

```
View(dat)
```

You can take a look at the first few rows of your data frame using the `head()` function.

```
head(dat)
```

```
## # A tibble: 6 x 82
##   Taxa T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3 T_3_5
##   <ch> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 2 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 3 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 4 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 5 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 6 Aci~     0      0      0     2     2     5     0    15     2     1     0
## # ... with 70 more variables: T_4_3 <int>, T_4_4 <int>, T_4_5 <int>,
## #   T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>, T_5_4 <int>,
## #   T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>, T_6_8 <int>,
## #   T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>, T_9_5 <int>,
## #   V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>, V_11_3 <int>,
## #   V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>, V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>
```

Likewise, to inspect the last rows, you can use the `tail()` function. You can specify the number of rows to `head()` or `tail()`.

```r
tail(dat, 10)
```

```
## # A tibble: 10 x 82
##    Taxa  T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3
##    <chr> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int>
##  1 Opit~     0      0      0     0     0     0     0     0     0     0
##  2 Plan~     0      0      0     0     0     2     0     8     0     0
##  3 Sphi~     2      0      0     4     4     1     0     1     1     0
##  4 Spir~    62      0      0    72   266    43     1   175    19     7
##  5 Subd~     0      0      0     0     0     0     0     0     0     0
##  6 Syne~    21      4      0    84   110   333     0   658   125    70
##  7 Ther~     0      0      0     3     3     3     0     2     1     0
##  8 Ther~     0      0      0     0     0     0     0     0     0     0
##  9 Ther~     0      0      0     0     0     1     0     1     0     0
## 10 Unkn~   759     24      6  1342  1555  1832     7  3240   676   306
## # ... with 71 more variables: T_3_5 <int>, T_4_3 <int>, T_4_4 <int>,
## #   T_4_5 <int>, T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>,
## #   T_5_4 <int>, T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>,
## #   T_6_8 <int>, T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>,
## #   T_9_5 <int>, V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>,
## #   V_11_3 <int>, V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>,
## #   V_14_1 <int>, V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>,
## #   V_15_3 <int>, V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>,
## #   V_18_1 <int>, V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>,
## #   V_19_2 <int>, V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>,
## #   V_20_1 <int>, V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>,
## #   V_22_4 <int>, V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>,
## #   V_5_1 <int>, V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>,
## #   V_7_1 <int>, V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>,
## #   V_9_2 <int>, V_9_3 <int>, V_9_4 <int>
```

It is often extremely useful to subset your data by some logical condition (`==` (equal to), `!=` (not equal to), `>` (greater than), `>=` (greater than or equal to), `<` (less than), `<=` (less than or equal to), `&` (and), `|` (or)). For example, we may want to keep all rows that have either Fusobacteria or Methanobacteria. Using the `filter()` function our first argument will be our data frame, followed by the information for the rows we want to subset by.

```r
filter(dat, Taxa == "Fusobacteria" | Taxa == "Methanobacteria")
```

```
## # A tibble: 2 x 82
##   Taxa T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3 T_3_5
##   <ch> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 Fus~     0      0      0     0     0     1     0     2     0     0     0
## 2 Met~     0      0      0     1     0     0     0     0     0     0     0
## # ... with 70 more variables: T_4_3 <int>, T_4_4 <int>, T_4_5 <int>,
## #   T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>, T_5_4 <int>,
## #   T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>, T_6_8 <int>,
## #   T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>, T_9_5 <int>,
## #   V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>, V_11_3 <int>,
## #   V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>, V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>
```

Will this work?

```r
filter(dat, Taxa == c("Fusobacteria", "Methanobacteria"))
```

```
## # A tibble: 0 x 82
## # ... with 82 variables: Taxa <chr>, T_2_1 <int>, T_2_10 <int>,
## #   T_2_12 <int>, T_2_2 <int>, T_2_3 <int>, T_2_6 <int>, T_2_7 <int>,
## #   T_2_9 <int>, T_3_2 <int>, T_3_3 <int>, T_3_5 <int>, T_4_3 <int>,
## #   T_4_4 <int>, T_4_5 <int>, T_4_6 <int>, T_4_7 <int>, T_5_2 <int>,
## #   T_5_3 <int>, T_5_4 <int>, T_5_5 <int>, T_6_2 <int>, T_6_5 <int>,
## #   T_6_7 <int>, T_6_8 <int>, T_9_1 <int>, T_9_2 <int>, T_9_3 <int>,
## #   T_9_4 <int>, T_9_5 <int>, V_1_2 <int>, V_10_1 <int>, V_11_1 <int>,
## #   V_11_2 <int>, V_11_3 <int>, V_12_1 <int>, V_12_2 <int>, V_13_1 <int>,
## #   V_13_2 <int>, V_14_1 <int>, V_14_2 <int>, V_14_3 <int>, V_15_1 <int>,
## #   V_15_2 <int>, V_15_3 <int>, V_16_1 <int>, V_16_2 <int>, V_17_1 <int>,
## #   V_17_2 <int>, V_18_1 <int>, V_18_2 <int>, V_18_3 <int>, V_18_4 <int>,
## #   V_19_1 <int>, V_19_2 <int>, V_19_3 <int>, V_2_1 <int>, V_2_2 <int>,
## #   V_2_3 <int>, V_20_1 <int>, V_21_1 <int>, V_21_4 <int>, V_22_1 <int>,
## #   V_22_3 <int>, V_22_4 <int>, V_3_1 <int>, V_3_2 <int>, V_4_1 <int>,
## #   V_4_2 <int>, V_5_1 <int>, V_5_3 <int>, V_6_1 <int>, V_6_2 <int>,
## #   V_6_3 <int>, V_7_1 <int>, V_7_2 <int>, V_7_3 <int>, V_8_2 <int>,
## #   V_9_1 <int>, V_9_2 <int>, V_9_3 <int>, V_9_4 <int>
```

Wait a second - why is this answer different? Why do we have 0 rows instead of 2?

A reminder/warning about vectors

```r
c(1,2,3) + c(10,11)
```

```
## Warning in c(1, 2, 3) + c(10, 11): longer object length is not a multiple
## of shorter object length

## [1] 11 13 13
```

Vectors recycle. In this case, R gave us a warning that our vectors don't match. It returned to us a vector of length 3 (our longest vector), and it recycled the 10 from the shorter vector to add to the 3.

However, R will assume that you know what you are doing as long as your one of your vector lengths is a multiple of your other vector length. Here the shorter vector is recycled twice. No warning is given.

```
c(1,2,3,4) + c(10,11)
```

```
## [1] 11 13 13 15
```

Let's go back to our example. In this code, I am looking through the Taxa column for when Taxa is equal to Fusobacteria OR Taxa is equal to Methanobacteria.

```
filter(dat, Taxa == "Fusobacteria" | Taxa == "Methanobacteria")
```

However, with a vector, I am alternately going through all values of Taxa and asking: does the first value match Fusobacteria? does the second value match Methanobacteria? Then the vector recycles and asks: does the third value match Fusobacteria? does the 4th value match Methanobacteria? We end up with a data frame of zero observations when we are expecting a data frame of 2 observations.

```
filter(dat, Taxa == c("Fusobacteria", "Methanobacteria"))
```

Be careful when filtering. You have been warned.

We just filtered for multiple Taxa (multiple rows based on the identity of values values in 1 column), however you can also filter for rows based on values in multiple columns. For example, to get only Taxa that had OTUs at sites T_2_1 and T_2_10 using `dplyr`, you can use the following filter:

```
filter(dat, T_2_10 != 0 & T_2_1 != 0)
```

```
## # A tibble: 7 x 82
##   Taxa T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3 T_3_5
##   <ch> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 Act~   110      3      5   240   414   227     1   811    89    36     7
## 2 Bac~    19     60     36    32    33    11    17    41    23    11    43
## 3 Bac~  1547      8      0   718   679   143     1   924   314   138    22
## 4 Clo~  6213     71      0  8999 10944 12169    28 17471  3431  1277   277
## 5 Gam~    23     30     27    17    25     4     0    37    20     5     4
## 6 Syn~    21      4      0    84   110   333     0   658   125    70    13
## 7 Unk~   759     24      6  1342  1555  1832     7  3240   676   306    69
## # ... with 70 more variables: T_4_3 <int>, T_4_4 <int>, T_4_5 <int>,
## #   T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>, T_5_4 <int>,
## #   T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>, T_6_8 <int>,
## #   T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>, T_9_5 <int>,
## #   V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>, V_11_3 <int>,
## #   V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>, V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>
```

You can subset columns by using the `select()` function. You can also reorder columns using this function. I want to to compare the depth of latrine 2 at 9cm compared to 10cm, but I want Taxa in the last column.

```
select(dat, T_2_9, T_2_10, Taxa)
```

```
## # A tibble: 52 x 3
##    T_2_9 T_2_10 Taxa
##    <int>  <int> <chr>
##  1     0      0 Acidobacteria_Gp1
##  2     0      0 Acidobacteria_Gp10
##  3     0      0 Acidobacteria_Gp14
##  4     0      0 Acidobacteria_Gp16
##  5     0      0 Acidobacteria_Gp17
##  6    15      0 Acidobacteria_Gp18
##  7     0      0 Acidobacteria_Gp21
##  8     0      0 Acidobacteria_Gp22
##  9     0      0 Acidobacteria_Gp3
## 10     0      0 Acidobacteria_Gp4
## # ... with 42 more rows
```

`dplyr` also includes some helper functions that allow you to select variables based on their names. For example, if we only wanted the samples from Vietnam, we could use `starts_with()`.

```
select(dat, Taxa, starts_with("V"))
```

```
## # A tibble: 52 x 53
##    Taxa  V_1_2 V_10_1 V_11_1 V_11_2 V_11_3 V_12_1 V_12_2 V_13_1 V_13_2
##    <chr> <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
##  1 Acid~     0      0      0      0      0      0      0      0      0
##  2 Acid~     0      0      0      0      0      0      0      0      0
##  3 Acid~     0      0      0      0      0      0      0      0      0
##  4 Acid~     0      0      1      0      0      0      0      0      0
##  5 Acid~     0      0      1      0      0      0      0      0      0
##  6 Acid~     0      0      0      0      0      0      0      0      0
##  7 Acid~     0      0      0      0      0      0      0      0      0
##  8 Acid~     0      0      0      0      0      0      0      0      0
##  9 Acid~     0      0      1      0      1      0      0      0      0
## 10 Acid~     0      0      7      0      3      0      0      0      0
## # ... with 42 more rows, and 43 more variables: V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>
```

Or all latrines with depths of 4 cm using `ends_with()`.

```
select(dat, Taxa, ends_with("4"))
```

```
## # A tibble: 52 x 8
##    Taxa               T_4_4 T_5_4 T_9_4 V_18_4 V_21_4 V_22_4 V_9_4
##    <chr>               <int> <int> <int>  <int>  <int>  <int> <int>
##  1 Acidobacteria_Gp1       0     0     0      0      0      0    38
##  2 Acidobacteria_Gp10      0     0     0      0      0      0     0
##  3 Acidobacteria_Gp14      0     0     0      0      0      0    17
##  4 Acidobacteria_Gp16      0     0     0      0      0      0     0
##  5 Acidobacteria_Gp17      0     0     0      0      0      0     0
##  6 Acidobacteria_Gp18      1     0     0      0      0      0     0
##  7 Acidobacteria_Gp21      0     0     0      0      0      0     0
##  8 Acidobacteria_Gp22      0     0     0      0      0      0     0
##  9 Acidobacteria_Gp3       0     0     0     10      0      0    19
## 10 Acidobacteria_Gp4       0     0     0      0      0      0     0
## # ... with 42 more rows
```

You can look up other 'select_helpers' in the help menu.

Challenge

Check out 'select_helpers' in the help menu. Grab all of the columns that contain depths for Well 2, whether it is from Vietnam or Tanzania. Retain Taxa either as rownames or in a column.

The `arrange()` function helps you to sort your data. The default is ordered from smallest to largest (or a-z for character data). You can switch the order by specifying `desc` (descending) as shown below.

Let's say we want to look at Tanzania, Well 2, at 1cm depth. We want to arrange our data frame in descending order of OTUs, and we want to find the unique Taxa that have > 0 OTUs and are not 'Unknown'. Our goal is to have the Taxa present in that particular well in order of decreasing abundance. How would you go about solving this problem?

```r
t21 <- select(dat, Taxa, T_2_1)

desc_t21 <- arrange(t21, desc(T_2_1))

filt_t21 <- filter(desc_t21, T_2_1 !=0 & Taxa != "Unknown")

select_taxa <- select(filt_t21, Taxa)

uni <- unique(select_taxa)
```

`unique()` is a function that removes duplicate rows. How many rows did it remove in this case? How do you know?

While this code answered the question, it also created a bunch of new variables that we aren't interested in. These 'intermediate variables' were used to store data that got passed as input to the next function. This would quickly clutter our global environment if this was our strategy for data analysis.

The `dplyr` package, and some other common packages for data frame manipulation allow the use of the pipe function, `%>%`. This is equivalent to `|` for any unix peeps. **Piping** allows the output of a function to be passed to the next function without making intermediate variables. Piping can save typing, make your code more readable, and reduce clutter in your global environment from variables you don't need. The keyboard shortcut for `%>%` is `CTRL+SHIFT+M`.

We are going to see how pipes work in conjunction with our first function, `filter()` and then see the benefits for our more complex example.

```r
filter(dat, Taxa == "Fusobacteria" | Taxa == "Methanobacteria")
#equivalent to
```

```r
dat %>% filter(Taxa == "Fusobacteria" | Taxa == "Methanobacteria")
#equivalent to
```

```r
dat %>% filter(., Taxa == "Fusobacteria" | Taxa == "Methanobacteria")
```

```
## # A tibble: 2 x 82
##   Taxa T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3 T_3_5
##   <ch> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 Fus~     0      0      0     0     0     1     0     2     0     0     0
## 2 Met~     0      0      0     1     0     0     0     0     0     0     0
## # ... with 70 more variables: T_4_3 <int>, T_4_4 <int>, T_4_5 <int>,
## #   T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>, T_5_4 <int>,
## #   T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>, T_6_8 <int>,
## #   T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>, T_9_5 <int>,
## #   V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>, V_11_3 <int>,
## #   V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>, V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>
```

You'll notice that when piping, we are not explicitly writing the first argument (our data frame) to `filter()`, but rather passing the first argument to filter using `%>%`. The dot `.` is sometimes used to fill in the first argument as a placeholder (this is useful for nested functions (functions inside of functions), which we will come across a bit later).

What would working with pipes look like for our more complex example?

```r
dat %>% select(Taxa, T_2_1) %>% arrange(desc(T_2_1)) %>% filter(T_2_1 !=0 & Taxa != "Unknown") %>%
  select(Taxa) %>% unique()
```

When using more than 2 pipes `%>%` it gets hard to follow for a reader (or yourself). Starting a new line after each pipe, allows a reader to easily see which function is operating and makes it easier to follow your logic. Using pipes also has the benefit that extra intermediate variables do not need to be created, freeing up your global environment for objects you are interested in keeping.

```r
dat %>%
  select(Taxa, T_2_1) %>%
  arrange(desc(T_2_1)) %>%
  filter(T_2_1 !=0 & Taxa != "Unknown") %>%
  select(Taxa) %>%
  unique()
```

```
## # A tibble: 14 x 1
##    Taxa
##    <chr>
##  1 Clostridia
##  2 Bacteroidia
##  3 Erysipelotrichi
##  4 Actinobacteria
##  5 Spirochaetes
##   6 Gammaproteobacteria
##   7 Synergistia
##   8 Bacilli
##   9 Alphaproteobacteria
## 10 Mollicutes
## 11 Anaerolineae
## 12 Betaproteobacteria
## 13 Sphingobacteria
## 14 Flavobacteria
```

R will throw an error if you try to do a calculation on a dataframe which contains both numeric and character data.

The `column_to_rownames()` function is used here to move our character data, Taxa, to rownames. We will then have a numeric dataframe on which to do any calculations while preserving all information. This function requires the column variable to be quoted. Click on 'dat' in the Global Environment to see the difference.

```
taxa_rownames_dat <- dat %>% column_to_rownames("Taxa")
```

We can use the reverse function `rownames_to_column` to recreate the Taxa column from rownames.

```
taxa_colnames_dat <- taxa_rownames_dat %>% rownames_to_column("Taxa")
```

An example of where this might be useful would be calculating the total number of OTUs for each Taxa using `rowSums()`. Without the above functions, you would have to subset out the character column, Taxa, losing important information.

```
dat[, -1] %>% rowSums()
```

```
##  [1]     66     15     19      2      1     31      9      5    162     34
## [11]     11    163      1      1  40978  33238   1013  26799  55450  27171
## [21]    600      3      1     41 277296     60      7  12970   2442      5
## [31]  10441     74  40319    871  63588     89      3     77      1      2
## [41]   5242     46    764    533  22049   2069      1   3150   2347      1
## [51]      2  57237
```

If instead, the dataframe where the Taxa column was converted to rownames, the output is a named vector; the rownames were preserved for the names of the vector elements.

```
taxa_rownames_dat %>% rowSums()
```

```
##     Acidobacteria_Gp1    Acidobacteria_Gp10    Acidobacteria_Gp14
##                    66                    15                    19
##    Acidobacteria_Gp16    Acidobacteria_Gp17    Acidobacteria_Gp18
##                     2                     1                    31
##    Acidobacteria_Gp21    Acidobacteria_Gp22     Acidobacteria_Gp3
##                     9                     5                   162
##     Acidobacteria_Gp4     Acidobacteria_Gp5     Acidobacteria_Gp6
##                    34                    11                   163
##     Acidobacteria_Gp7     Acidobacteria_Gp9        Actinobacteria
##                     1                     1                 40978
##   Alphaproteobacteria          Anaerolineae               Bacilli
##                 33238                  1013                 26799
##           Bacteroidia    Betaproteobacteria           Caldilineae
##                 55450                 27171                   600
##            Chlamydiae           Chloroflexi        Chrysiogenetes
##                     3                     1                    41
##            Clostridia         Cyanobacteria     Dehalococcoidetes
##                277296                    60                     7
##            Deinococci   Deltaproteobacteria Epsilonproteobacteria
##                 12970                  2442                     5
##       Erysipelotrichi          Fibrobacteria         Flavobacteria
##                 10441                    74                 40319
##          Fusobacteria   Gammaproteobacteria      Gemmatimonadetes
##                   871                 63588                    89
##            Holophagae          Lentisphaeria      Methanobacteria
##                     3                    77                     1
##       Methanomicrobia             Mollicutes            Nitrospira
##                     2                  5242                    46
##             Opitutae       Planctomycetacia      Sphingobacteria
##                   764                   533                 22049
##          Spirochaetes          Subdivision3           Synergistia
##                  2069                     1                  3150
##        Thermomicrobia        Thermoplasmata          Thermotogae
##                  2347                     1                     2
##              Unknown
##                 57237
```

`mutate()` is a function to create new column, most often the product of a calculation. For example, let's calculate the total number of OTUs for each Taxa using `rowSums()` by adding an additional column to our table. You must specify a column name for the column you are creating.

An annoying part of `dplyr` is that it will drop rownames with most functions (ie. `mutate()`, `filter()` and `arrange()`, but not `select()`). It doesn't make much sense to take row sums if we can't tell what the rows are. However, to do the `rowSums()` calculation we cannot have character data in our rows. Therefore `.[ , -1]` is everything but our character data (which is our first column).

You may have noticed that I included the dot (`.`) in the bracket for `rowSums()`. This is because it is inside a nested function (`mutate()` is a function and `rowSums()` is inside it). Without an argument to `rowSums()` an error would be generated that "argument 'x' is missing with no default".

```
dat %>% mutate(total_OTUs = rowSums(.[,-1])) %>% head()
```

```
## # A tibble: 6 x 83
##   Taxa T_2_1 T_2_10 T_2_12 T_2_2 T_2_3 T_2_6 T_2_7 T_2_9 T_3_2 T_3_3 T_3_5
##   <ch> <int>  <int>  <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 2 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 3 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 4 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 5 Aci~     0      0      0     0     0     0     0     0     0     0     0
## 6 Aci~     0      0      0     2     2     5     0    15     2     1     0
## # ... with 71 more variables: T_4_3 <int>, T_4_4 <int>, T_4_5 <int>,
## #   T_4_6 <int>, T_4_7 <int>, T_5_2 <int>, T_5_3 <int>, T_5_4 <int>,
## #   T_5_5 <int>, T_6_2 <int>, T_6_5 <int>, T_6_7 <int>, T_6_8 <int>,
## #   T_9_1 <int>, T_9_2 <int>, T_9_3 <int>, T_9_4 <int>, T_9_5 <int>,
## #   V_1_2 <int>, V_10_1 <int>, V_11_1 <int>, V_11_2 <int>, V_11_3 <int>,
## #   V_12_1 <int>, V_12_2 <int>, V_13_1 <int>, V_13_2 <int>, V_14_1 <int>,
## #   V_14_2 <int>, V_14_3 <int>, V_15_1 <int>, V_15_2 <int>, V_15_3 <int>,
## #   V_16_1 <int>, V_16_2 <int>, V_17_1 <int>, V_17_2 <int>, V_18_1 <int>,
## #   V_18_2 <int>, V_18_3 <int>, V_18_4 <int>, V_19_1 <int>, V_19_2 <int>,
## #   V_19_3 <int>, V_2_1 <int>, V_2_2 <int>, V_2_3 <int>, V_20_1 <int>,
## #   V_21_1 <int>, V_21_4 <int>, V_22_1 <int>, V_22_3 <int>, V_22_4 <int>,
## #   V_3_1 <int>, V_3_2 <int>, V_4_1 <int>, V_4_2 <int>, V_5_1 <int>,
## #   V_5_3 <int>, V_6_1 <int>, V_6_2 <int>, V_6_3 <int>, V_7_1 <int>,
## #   V_7_2 <int>, V_7_3 <int>, V_8_2 <int>, V_9_1 <int>, V_9_2 <int>,
## #   V_9_3 <int>, V_9_4 <int>, total_OTUs <dbl>
```

Note that if I use `rowSums()` outside of another function (ie. not nested), I do not need to specify the data frame. While mutate creates a new column for the result of the `rowSums()` function, using `rowSums()` alone produces an output in vector format of the length of the number of rows in the data frame.

`transmute()` will also create a new variable, but it will drop the existing variables (it will give you a single column of your new variable). The output for `transmute()` is a data frame of one column.

```
dat %>% transmute(total_OTUs = rowSums(.[ , -1]))
```

```
## # A tibble: 52 x 1
##    total_OTUs
##         <dbl>
##  1         66
##  2         15
##  3         19
##  4          2
##  5          1
##  6         31
##  7          9
##  8          5
##  9        162
## 10         34
## # ... with 42 more rows
```

It is up to you whether you want to keep your data in a data frame or switch to a vector if you are dealing with a single variable. Using a `dplyr` function will maintain your data in a data frame. Using non-dplyr functions will switch your data to a vector if you have a 1-dimensional output.

`dplyr` has one more super-useful function, `summarize()` which allows us to get summary statistics on data. I'll give one example and then we'll come back to this function once our data is in 'tidy' format.

`n()` is a `dplyr` function to count the number of observations in a group - we are simply going to count the instances of a Taxa. In order for R to know we want to count the instances of each Taxa (as opposed to say, each row), there is a function `group_by()` that we can use to group variables or sets of variables together (treating them like a factor). That way if we had more than one case of "Clostridia", they would be grouped together and when we counted the number of cases, the result would be greater than 1. `group_by()` is useful for calculations and plotting on subsets of your data without having to explicitly change your variables into factors.

We can arrange the results in descending order to see if there is more than one row per Taxa.

```
dat %>%
  group_by(Taxa) %>%
  summarize(n = n()) %>%
  arrange(desc(n))
```

```
## # A tibble: 52 x 2
##    Taxa                   n
##    <chr>              <int>
##  1 Acidobacteria_Gp1      1
##  2 Acidobacteria_Gp10     1
##  3 Acidobacteria_Gp14     1
##  4 Acidobacteria_Gp16     1
##  5 Acidobacteria_Gp17     1
##  6 Acidobacteria_Gp18     1
##   7 Acidobacteria_Gp21     1
##   8 Acidobacteria_Gp22     1
##   9 Acidobacteria_Gp3      1
##  10 Acidobacteria_Gp4      1
## # ... with 42 more rows
```

What is the result if you don't use `group_by()`? Can anyone think of another way to see if there is more than one row per Taxa?

Challenge

Let's try to answer a question with our data:

- Is there more Clostridia in Tanzania or Vietnam?

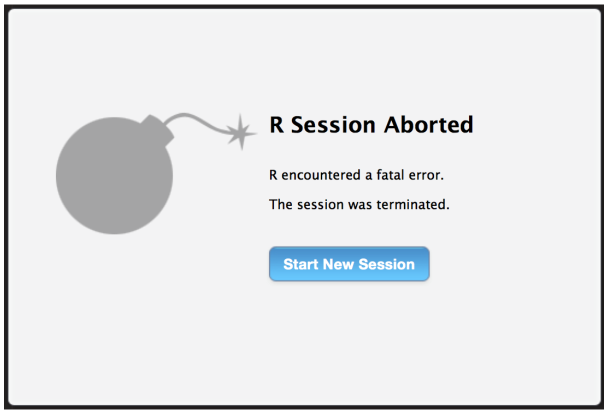
Resources

https://github.com/jennybc/googlesheets
http://stat545.com/block009_dplyr-intro.html
http://stat545.com/block010_dplyr-end-single-table.html
http://stat545.com/bit001_dplyr-cheatsheet.html
http://dplyr.tidyverse.org/articles/two-table.html

Post-Lesson Assessment

Your feedback is essential to help the next cohort of trainees. Please take a minute to complete the following short survey: https://www.surveymonkey.com/r/SMGKMCS

Thanks for coming!!!



R Session Aborted

R encountered a fatal error.

The session was terminated.

Appendix

Googlesheets

Googlesheets have a similar structure to excel workbooks, the only tricky thing is getting the name of your googlesheet to input.

I have a googlesheet to share from my Google Drive:
https://docs.google.com/spreadsheets/d/1JTy5sCtQz8PmlpDrgvOnwZ1tB2pxOndxoNMTHQR_PrQ/edit?usp=sharing.

Add the sheet to your own Google Drive. Then load googlesheets.

```r
library(googlesheets)
```

If you load googlesheets and ask it to list the googlesheets you have, googlesheets will open a new window and ask if it can have access to your googlesheets. If you say yes, you can return to R and breathe a sigh of relief. Your import of your googlesheets worked.

```r
gs_ls()

#if you have many googlesheets like I do you may need to search for your sheet
#tail(gs_ls())
```

Register the sheet you are going to use (gather information on the sheet from the API) with the sheet title using `gs_title()`.

```r
books <- gs_title('Books Everyone Should Read')
```

If this did not work for some reason, download books_alpha.xlsx and save it to your current working directory. Upload the file to Google Drive using `gs_upload()`.

```r
books <- gs_upload(file = "data/books_alpha.xlsx")
```

The structure of the googlesheet is a list.

```r
str(books)
```

How many worksheets are in this spreadsheet and what are their names? To find the number of worksheets, I subset from the list `n_ws`. To get the names of the worksheets, I grab the worksheets `ws` (itself a data frame) and subset the column `ws_title` for their titles.

```r
books$n_ws

books$ws$ws_title
```

Read in the worksheet you want to access using `gs_read()`. You can do this by specifying the worksheet number or title.

```r
book_sheet1 <- gs_read(books, ws = 1)
```

The last 2 columns don't hold much information. You can specify a subset of this sheet similarly to the excel sheets using `range`. You can do this by selecting the cell columns with `cell_cols`, or by using the cell range.

```r
booksfilt <- gs_read(books, ws = 1, range = cell_cols(1:14))
#or
booksfilt <- gs_read(books, ws = 1, range = "A1:N1000")
```

Here we specify the worksheet by name, and sort by rows using `cell_rows`. Here are the top recommendations for what to read, when you are no longer in academia.

```
books_alpha <- gs_read(books, ws = "Top titles", range = cell_rows(1:10))
```

One nice thing about the googlesheets package is that all of the functions begin with gs_ which is great for finding functions and tab completion.

You can use gs_download to save this file to your computer as an excel workbook by giving the title of the googlesheet and the name for the output file. You can overwrite an existing file by selecting overwrite = TRUE.

```
gs_download(gs_title("Books Everyone Should Read"), to = "books_alpha.xlsx", overwrite = TRUE)
```

You can upload sheets you've made on your computer to use in googlesheets. This file is now in your google account online.

```
books <- gs_upload("books_alpha.xlsx")
```