#: Data wrangling ## Cleaning our data in R ### R > excel {.unnumbered} Before we begin wrangling our data set, let's quickly discuss why cleaning data in R is important. First and foremost, my Excel file never needs to change. Along with this, I only need to save it once. No saving a new copy every time I make a new column or want to write a formula.

In this section, we will go over some important packages and key functions that will aid in the transition from Excel-based data wrangling to exclusively R-based data wangling. SO, without further ado, lettuce begin.

What we use to wrangle

When we wrangle our data sets, there is one 'umbrella' package I find to me the most versatile. This is **tidyverse**. **Tidyverse** houses many useful packages for data manipulation, but in this section I will only be referring to one. This package is the **dplyr** package. While I rely on these packages a lot, I also use functions pre-installed in Base R while wrangling.

Please see the cheat sheets (here) for both of these packages.

dplyr Examples

Let's start by installing the needed packages. Remember to remove the # symbol to install these packages. We will also call in our Palmer Penguins data set that you downloaded earlier.

```
#install.packages("dplyr")
library(dplyr)

#install.packages("tidyverse")
library(tidyverse)

library(palmerpenguins)
data(package = 'palmerpenguins')
```

NAs in R

Before we begin, we must first check for missing values. R *does not* love when NAs get thrown into the mix, especially when running numerical commands, like the **mean** function.
To resolve any missing values, we must first determine if there are any NAs, and where they may be.

First, we will look to see IF and WHERE potential NAs are in our data set. We will do this by using the **which** function, followed by the **is.na** function within out penguins data set. What these two functions do together is which locates all of the columns where NAs are present, based on the **is.na** function. If we wanted to find all of the columns where there were no NAs, we could simply change **is.na** to **!is.na**.

```
which(is.na(penguins), arr.ind = TRUE)
```

```
      row col
 [1,]   4   3
 [2,] 272   3
 [3,]   4   4
 [4,] 272   4
 [5,]   4   5
 [6,] 272   5
 [7,]   4   6
 [8,] 272   6
 [9,]   4   7
[10,]   9   7
[11,]  10   7
[12,]  11   7
[13,]  12   7
[14,]  48   7
[15,] 179   7
[16,] 219   7
[17,] 257   7
[18,] 269   7
[19,] 272   7
```

Now that we see there are NAs riddled throughout, we will name a new object (using the same name), but omitting all NAs.

In this example, I am using the **na.omit** function to remove all NAs from our penguin data set. I am also naming this new object (penguins again for ease).

```
penguins <-
  na.omit(penguins) #I must add this because there are NAs within this data set and these
```

select()

Let's start by selecting for only the columns we are interested in. This can be useful when removing variables we are not currently interested in. **Remember**, at any manipulation, you

can save the changes as a new object which will maintain the integrity of the original if you must back track for whatever reason.

In this example, we use the **select** function to choose which columns we want to look at.

```
# select()
penguins %>% #From the penguins data set
  select(species, bill_length_mm, year) # selecting columns species, bill_length, and
```

```
# A tibble: 333 x 3
   species bill_length_mm  year
   <fct>            <dbl> <int>
 1 Adelie            39.1  2007
 2 Adelie            39.5  2007
 3 Adelie            40.3  2007
 4 Adelie            36.7  2007
 5 Adelie            39.3  2007
 6 Adelie            38.9  2007
 7 Adelie            39.2  2007
 8 Adelie            41.1  2007
 9 Adelie            38.6  2007
10 Adelie            34.6  2007
# ... with 323 more rows
```

Now that we have selected for certain columns, let's say we want to view everything except for one or several columns. Instead of typing out every column we want, we can simple type out the one(s) we don't.

In this example, I tell R to remove the sex column using the **select** function again.

```
# select()
penguins %>% #From the penguins data set
  select(-sex)# selecting all columns except for sex
```

```
# A tibble: 333 x 7
   species island    bill_length_mm bill_depth_mm flipper_length~1 body_~2  year
   <fct>   <fct>              <dbl>         <dbl>            <int>   <int> <int>
 1 Adelie  Torgersen           39.1          18.7              181    3750  2007
 2 Adelie  Torgersen           39.5          17.4              186    3800  2007
 3 Adelie  Torgersen           40.3          18                195    3250  2007
 4 Adelie  Torgersen           36.7          19.3              193    3450  2007
 5 Adelie  Torgersen           39.3          20.6              190    3650  2007
```

```
 6 Adelie  Torgersen           38.9          17.8               181    3625  2007
 7 Adelie  Torgersen           39.2          19.6               195    4675  2007
 8 Adelie  Torgersen           41.1          17.6               182    3200  2007
 9 Adelie  Torgersen           38.6          21.2               191    3800  2007
10 Adelie  Torgersen           34.6          21.1               198    4400  2007
# ... with 323 more rows, and abbreviated variable names 1: flipper_length_mm,
#   2: body_mass_g
```

Within the **select** function, you can also look for items based on their spelling. This can be especially helpful if you suspect there to be a spelling error somewhere if your data set. In this example, we will search our data set for any variable name that starts with the letter 'b'.

```
# select()
penguins %>% #From the penguins data set
  select(starts_with('b')) # selecting columns that start with 'b' and using starts_with
```

```
# A tibble: 333 x 3
   bill_length_mm bill_depth_mm body_mass_g
            <dbl>         <dbl>       <int>
 1           39.1          18.7        3750
 2           39.5          17.4        3800
 3           40.3          18          3250
 4           36.7          19.3        3450
 5           39.3          20.6        3650
 6           38.9          17.8        3625
 7           39.2          19.6        4675
 8           41.1          17.6        3200
 9           38.6          21.2        3800
10           34.6          21.1        4400
# ... with 323 more rows
```

rename()

Now that we have viewed and selected for different columns and such, we manipulate our data set further. We will start by renaming some columns. Notice with rename, there are two methods you can use. One *without* quotation marks, and one *with.*

In this example, using the **rename** function, I am changing 'species' to 'Species and 'year' to 'Year'.

```r
# rename()
penguins %>%
  select(species, bill_length_mm, year) %>% #selecting the columns I want to look at
  rename( #rename function. notice here the two methods of changing names
    Species = species, #changing species to Species without quotes
    "Year" = year #changing year to Year with quotes
  )
```

```
# A tibble: 333 x 3
   Species bill_length_mm  Year
   <fct>            <dbl> <int>
 1 Adelie            39.1  2007
 2 Adelie            39.5  2007
 3 Adelie            40.3  2007
 4 Adelie            36.7  2007
 5 Adelie            39.3  2007
 6 Adelie            38.9  2007
 7 Adelie            39.2  2007
 8 Adelie            41.1  2007
 9 Adelie            38.6  2007
10 Adelie            34.6  2007
# ... with 323 more rows
```

arrange()

let's arrange some stuff this is equivalent to sort!

One of the first steps we take in *Excel* is the **sorting** of our data sets. Whether that be the sorting of plots, or dates, or anything; we start by sorting. The same is possible in R. We do this using the **arrange** function.

In this example, we will be sorting by bill length in an increasing order (smallest to largest). Notice here that R will *default* to the order of small-large with the **arrange** function.

```r
# select() and arrange()
penguins %>%
  select(species, bill_length_mm, year) %>%
  arrange(bill_length_mm) #I want to look at bill length in an increasing order from
```

```
# A tibble: 333 x 3
   species bill_length_mm  year
```

```
   <fct>            <dbl> <int>
 1 Adelie            32.1  2009
 2 Adelie            33.1  2008
 3 Adelie            33.5  2008
 4 Adelie            34    2008
 5 Adelie            34.4  2007
 6 Adelie            34.5  2008
 7 Adelie            34.6  2007
 8 Adelie            34.6  2008
 9 Adelie            35    2008
10 Adelie            35    2009
# ... with 323 more rows
```

In this example, we will be sorting by bill length in a decreasing order (largest to smallest). Notice here, we need the **arrange** function to tell R we will be changing the order. Once that command is established, we can further command the order.

In this example, I use the **arrange** function, followed by the **desc** function (descending), commanding the order of bill length to go from big to small values.

```
# select(), arrange(), and desc()
penguins %>%
  select(species, bill_length_mm, year) %>%
  arrange(desc(bill_length_mm)) #using the desc() function to command the order from
```

```
# A tibble: 333 x 3
   species   bill_length_mm  year
   <fct>              <dbl> <int>
 1 Gentoo              59.6  2007
 2 Chinstrap           58    2007
 3 Gentoo              55.9  2009
 4 Chinstrap           55.8  2009
 5 Gentoo              55.1  2009
 6 Gentoo              54.3  2008
 7 Chinstrap           54.2  2008
 8 Chinstrap           53.5  2008
 9 Gentoo              53.4  2009
10 Chinstrap           52.8  2008
# ... with 323 more rows
```

Now, let's say we want to see bill length in the same descending order, but we want order this by year. This is done with a very simple addition to our *arrange()* section. To accomplish

this, we add the year variable first (remembering the the default for **arrange** is small-large) followed by the bill length command (which is the same as the previous example.)

```
# select() and arrange()
penguins %>%
  select(species, bill_length_mm, year) %>%
  arrange(year, desc(bill_length_mm)) #year and bill separated by a comma
```

```
# A tibble: 333 x 3
   species   bill_length_mm  year
   <fct>              <dbl> <int>
 1 Gentoo              59.6  2007
 2 Chinstrap           58    2007
 3 Chinstrap           52.7  2007
 4 Chinstrap           52    2007
 5 Chinstrap           52    2007
 6 Chinstrap           51.7  2007
 7 Chinstrap           51.3  2007
 8 Chinstrap           51.3  2007
 9 Chinstrap           51.3  2007
10 Chinstrap           50.6  2007
# ... with 323 more rows
```

filter()

Within R, we also have the ability to subset out data sets and pull out rows with specific values. Let's say I *only* want to look at data from *2007*. To accomplish this, we will use the **filter** function.

In this example, we will be adding the **filter** function as well as recall our knowledge of (operators) within R.

```
# select() and filter()
penguins %>%
  select(species, bill_length_mm, bill_depth_mm, year) %>%
  filter(year == 2007) #using the '==' operator to show everything with the year 2007
```

```
# A tibble: 103 x 4
   species bill_length_mm bill_depth_mm  year
   <fct>            <dbl>         <dbl> <int>
 1 Adelie            39.1          18.7  2007
```

```
 2 Adelie            39.5          17.4  2007
 3 Adelie            40.3          18    2007
 4 Adelie            36.7          19.3  2007
 5 Adelie            39.3          20.6  2007
 6 Adelie            38.9          17.8  2007
 7 Adelie            39.2          19.6  2007
 8 Adelie            41.1          17.6  2007
 9 Adelie            38.6          21.2  2007
10 Adelie            34.6          21.1  2007
# ... with 93 more rows
```

What if I want to see within the year 2007, which penguins have bill lengths higher than the mean of them *all*? This can be accomplished by, again, adding an operator, but also calling another function. We will command R further with **mean** function from base R. Notice I am separating each line in the **filter** function with a comma. This allows me to add multiple commands within the same function.

```
# select(), filter(), and mean()

penguins %>%
  select(species, bill_length_mm, year) %>%
  filter(
    year == 2007,#using the '==' operator to show everything with the year 2007
    bill_length_mm < mean(bill_length_mm), # using the '<' to to view every row where the
  )
```

```
# A tibble: 49 x 3
   species bill_length_mm  year
   <fct>            <dbl> <int>
 1 Adelie            39.1  2007
 2 Adelie            39.5  2007
 3 Adelie            40.3  2007
 4 Adelie            36.7  2007
 5 Adelie            39.3  2007
 6 Adelie            38.9  2007
 7 Adelie            39.2  2007
 8 Adelie            41.1  2007
 9 Adelie            38.6  2007
10 Adelie            34.6  2007
# ... with 39 more rows
```

Let's say we are interested in manipulating our data set by species.

I want to know how many species I have to further filter this set. To accomplish this, I will use the the **count** function to view how many species I have and their associated values within the data set.

```
# count()
penguins %>%
  count(species)
```

```
# A tibble: 3 x 2
  species       n
  <fct>     <int>
1 Adelie      146
2 Chinstrap    68
3 Gentoo      119
```

It appears there are three species within my data set. For one reason or another, I want to filter out *Adelie* from further interpretations. To do this, I will add another line below the bill length filter.

This new line says *when species equals Chinstrap OR Gentoo*, keep them in the data set.

```
# filter()
penguins %>%
  select(species, bill_length_mm, year) %>%
  filter(
    year == 2007,#using the '==' operator to show everything with the year 2007
    bill_length_mm < mean(bill_length_mm),# using the '<' to to view every row where the b
    species == "Chinstrap" | species == "Gentoo" # look in species and pull out chinstrap
  )
```

```
# A tibble: 7 x 3
  species   bill_length_mm  year
  <fct>              <dbl> <int>
1 Gentoo              43.3  2007
2 Gentoo              40.9  2007
3 Gentoo              42    2007
4 Gentoo              42.9  2007
5 Gentoo              42.8  2007
6 Chinstrap           42.4  2007
7 Chinstrap           43.2  2007
```

Another way to accomplish the same task is to tell R which values to *exclude*, rather than *include*. This is done by using the 'does not equal' operator to command R to return every species value that is not Adelie.

```
# filter()
penguins %>%
  select(species, bill_length_mm, year) %>%
  filter(
    year == 2007,#using the '==' operator to show everything with the year 2007
    bill_length_mm < mean(bill_length_mm),# using the '<' to to view every row where the b
    species != "Adelie" #does not equal operator
  )
```

```
# A tibble: 7 x 3
  species   bill_length_mm  year
  <fct>              <dbl> <int>
1 Gentoo              43.3  2007
2 Gentoo              40.9  2007
3 Gentoo              42    2007
4 Gentoo              42.9  2007
5 Gentoo              42.8  2007
6 Chinstrap           42.4  2007
7 Chinstrap           43.2  2007
```

```
#the output is the same!
```

Now that we only the data we want to see, let's create some new columns and row values. Let's say we want to add a new column combining *species and year* and a new column with the *rounded values of bill length*. We will be using the **mutate** function here. Along with this, we then want to *rearrange* our data set for viewing purposes of our new variables. This will be done with the **select** function.

In this example, I have created the column 'sp_year' which will contain both species and year, but keep their respective values separated by a dash. I then created a new column of the rounded bill length values using the **round** function. *Notice* with these new columns, the first step is to name the new column and then command R what to put in. Lastly, using the **select** function, I command R to order this data set as follows.

```
penguins %>%
  select(species, bill_length_mm, year) %>%
  mutate( #mutate()
    sp_year = paste(species, "-", year), #adding a new column named 'sp_year' and pasting
```

```
    rn_bill_length_mm = round(bill_length_mm) #creating a column of rounded bill lengths
  ) %>%
  select(species, year, sp_year, bill_length_mm, rn_bill_length_mm) #placing these new col
```

```
# A tibble: 333 x 5
   species  year sp_year       bill_length_mm rn_bill_length_mm
   <fct>   <int> <chr>                  <dbl>             <dbl>
 1 Adelie   2007 Adelie - 2007           39.1                39
 2 Adelie   2007 Adelie - 2007           39.5                40
 3 Adelie   2007 Adelie - 2007           40.3                40
 4 Adelie   2007 Adelie - 2007           36.7                37
 5 Adelie   2007 Adelie - 2007           39.3                39
 6 Adelie   2007 Adelie - 2007           38.9                39
 7 Adelie   2007 Adelie - 2007           39.2                39
 8 Adelie   2007 Adelie - 2007           41.1                41
 9 Adelie   2007 Adelie - 2007           38.6                39
10 Adelie   2007 Adelie - 2007           34.6                35
# ... with 323 more rows
```

summarize()

Now that we are confident in our wrangling, we can investigate some summary statistics.

First, let's look at the means and standard deviations of both bill length and depth. This will be done by name new columns and then using either the **mean** function or **sd** function to produce a desired output.

```
penguins %>%
  summarize( #summarize to run summary stats
  bill_length_mean = mean(bill_length_mm), #new column with mean value of bill length
  bill_length_sd = sd(bill_length_mm), #new column with standard deviation value of bill l
  bill_depth_mean = mean(bill_depth_mm), #new column with mean value of bill depth
  bill_depth_sd = sd(bill_depth_mm) #new column with standard deviation value of bill dept
  )
```

```
# A tibble: 1 x 4
  bill_length_mean bill_length_sd bill_depth_mean bill_depth_sd
             <dbl>          <dbl>           <dbl>         <dbl>
1             44.0           5.47            17.2          1.97
```

The last output was informative, but lets look a little deeper. I now want to group these new values by species. Using the **group_by** function, we can tell R to group our data set by one, or more variables.

In this example, I am telling R to **group_by** species, and then provide me with the means and standard deviations of bill length and depth.

```
penguins %>%
  group_by(species) %>% #grouping by one column, species
  summarize( #summarize to run summary stats
    bill_length_mean = mean(bill_length_mm), #new column with mean value of bill length
    bill_length_sd = sd(bill_length_mm), #new column with standard deviation value of bill
    bill_depth_mean = mean(bill_depth_mm), #new column with mean value of bill depth
    bill_depth_sd = sd(bill_depth_mm) #new column with standard deviation value of bill de
  )
```

```
# A tibble: 3 x 5
  species   bill_length_mean bill_length_sd bill_depth_mean bill_depth_sd
  <fct>                <dbl>          <dbl>           <dbl>         <dbl>
1 Adelie                38.8           2.66            18.3         1.22
2 Chinstrap             48.8           3.34            18.4         1.14
3 Gentoo                47.6           3.11            15.0         0.986
```

Following the trend of the last example, let's further group our data set. I want to now see these same values but by species AND year. Using the **group_by** function again, we can accomplish this.

In this example, the only change is I added ', year' into my **group_by** function.

```
penguins %>%
  group_by(species, year) %>% #grouping by two columns, species and year
  summarize( #summarize to run summary stats
    bill_length_mean = mean(bill_length_mm), #new column with mean value of bill length
    bill_length_sd = sd(bill_length_mm), #new column with standard deviation value of bill
    bill_depth_mean = mean(bill_depth_mm), #new column with mean value of bill depth
    bill_depth_sd = sd(bill_depth_mm) #new column with standard deviation value of bill de
  )
```

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

```
# A tibble: 9 x 6
# Groups:   species [3]
  species    year bill_length_mean bill_length_sd bill_depth_mean bill_depth_sd
  <fct>     <int>            <dbl>          <dbl>           <dbl>         <dbl>
1 Adelie     2007             38.9           2.44            18.8         1.23
2 Adelie     2008             38.6           2.98            18.2         1.09
3 Adelie     2009             39.0           2.56            18.1         1.24
4 Chinstrap  2007             48.7           3.47            18.5         1.00
5 Chinstrap  2008             48.7           3.62            18.4         1.40
6 Chinstrap  2009             49.1           3.10            18.3         1.10
7 Gentoo     2007             47.1           3.29            14.7         0.919
8 Gentoo     2008             47.0           2.66            14.9         0.993
9 Gentoo     2009             48.6           3.19            15.3         0.967
```

Practice on your own

Now that we have worked through some examples with Palmer Penguins, let's try and work through a data set of our own.

Attached here is a (Slug data set.)

Remember, you will need to import this file into R in the correct format!

Your task is to [1] input it into R, [2] investigate the variables and classes of these variables, [3] produce an output using *each* of the functions we just covered, [4] and at least *one* example where you use **select, rename, arrange, filter, mutate, and group_by** in the same command line. In part 3, for each change to the data set, save the changed data set as a new object. For part 4, save this object as, 'Final_Changes'. If you conduct more than one iteration of part 4, add the associated number at the end of each name. For example, Final_Changes_1, Final_Changes_2, etc.