

Dplyr

# What is the tidyverse?



"The **tidyverse** is an **opinionated** collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures."

# Plan for today

- Learn basic syntax for nearly all `tidyverse` packages
- Introduce functions that come from the `dplyr` package
  - `filter()`
  - `select()`
  - `mutate()`
  - `summarize()`
  - `group_by()`

# About the MIDUS dataset

Variables available in this data file:

- **Demographic variables:** age, sex
- **Physical health variables:** self-rated physical health, heart problems, father had heart attack, BMI
- **Mental health variables:** self-rated mental health, self-esteem, life satisfaction (*life overall, work, health, relationship with spouse/partner, relationship with children*), hostility (*stress reactivity & aggression*)

Please load in `midus`, make sure:

- Make sure the variables `sex`, `heart_self`, and `heart_father` are `factor()` variables (rather than characters)
- Use the same `na.omit()` function to remove all `NA` values

# Syntax & Piping

- All of the **tidyverse** packages use **piping** as a way to make code easier to read.
- Think of it kind of like making a cohesive paragraph of code, rather than scribbling down a bunch of random lines.
- The format looks like this:

```
originalData %>%  
  function1(someVariable) %>%  
  function2(someVariable) %>%  
  function3(someVariable)
```

# Syntax & Piping

```
originalData %>%  
  function1(someVariable) %>%  
  function2(someVariable) %>%  
  function3(someVariable)
```

First thing that enters is your original data.frame. The end of the line has this `%>%` symbol. This is called a **pipe**.

# Syntax & Piping

```
originalData %>%  
  function1(someVariable) %>%  
  function2(someVariable) %>%  
  function3(someVariable)
```

Next up is some function that is performed on a variable. This variable COMES FROM the `originalData` data.frame. Another way to think about it is that the function *inherits* the data.frame from above. That means you don't need to keep re-typing `originalData`.

Again, the end of the line is followed by the `%>%` pipe operator.

# Syntax & Piping

```
originalData %>%  
  function1(someVariable) %>%  
  function2(someVariable) %>%  
  function3(someVariable)
```

Same thing for the next function. However, instead of inheriting from `originalData`, function 2 will inherit *the output* of function 1!

Again, the end of the line is followed by the `%>%` pipe operator.



# Syntax & Piping

```
originalData %>%  
  function1(someVariable) %>%  
  function2(someVariable) %>%  
  function3(someVariable)
```

Finally, we get to function 3. It will inherit *the output* of function 2.

Notice that there is no `%>%` pipe operator at the end of this line. That's because this "paragraph" of code is now over.

# Syntax & Piping

- These `%>%` pipes are used to perform **SEQUENTIAL** tasks!
- You can read the `%>%` as *and then...*

**We are R-Ladies** @WeAreRLadies · Sep 13, 2019

This is how I explain the 'pipe' to [#rstats](#) newbies...

```
##### %>%: Used to perform sequential tasks
```

```
I woke up %>% showered %>% dressed %>% glammed up %>% took breakfast  
%>% showed up to work
```

- Don't use `<-` *inside* the piped function. Only at the very beginning if you want to store the output.
- Keep `%>%` and the *end* of each line! Not at the beginning.
- Shortcut for inserting pipe:
  - `command + shift + m` for Mac users
  - `control + shift + m` for Windows users

# filter() Function

To illustrate how this works, let's start with the `filter()` function. `filter()` is another way to subset your data.frame based on some condition. It is the `tidyverse` equivalent of `subset()`.

Let's say we want to make a new data.frame that included only female participants...

```
femaleMidus <- midus %>%  
  filter(sex == "Female")
```

ID	sex	age	BMI	physical_health_self	mental_health_self	self_esteem	life_satisfaction
10011	Female	52	25.991	5	4	41	
10015	Female	53	32.121	3	3	31	
10023	Female	78	24.752	2	4	34	
10028	Female	63	24.049	5	5	42	
10030	Female	56	27.342	4	5	37	

# Spelling/capitalization etc. always count

Let's say we want to make a new data.frame that included only female participants...

```
femaleMidus <- midus %>%  
  filter(sex == "female")
```

ID	sex	age	BMI	physical_health_self	mental_health_self	self_esteem	life_satisfaction
----	-----	-----	-----	----------------------	--------------------	-------------	-------------------

# Now with multiple logical operators

Let's say we want to make a new data.frame that included male participants who have reported having some form of heart problem and are over the age of 50.

```
oldMenHeart <- midus %>%  
  filter(sex == "Male" & heart_self == "Yes" & age > 50)
```

ID	sex	age	BMI	physical_health_self	mental_health_self	self_esteem	life_s
10039	Male	53	31.872	1	4	35	
10067	Male	62	29.254	3	3	36	
10088	Male	79	29.289	4	4	34	
10131	Male	71	24.826	4	4	43	
10143	Male	57	25.105	3	5	35	
10173	Male	58	28.481	4	5	49	

# Is tidyverse totally different from base R?

**No!** You still have:

- objects
- assignment of objects
- functions
- functions that take in arguments
- logical operators like `==` and `>`
- multiple logical operators like `&` and `|`

The only thing that's different is the inclusion of `%>%` and the way you build your "code paragraphs". But all of the principles that we've learned thus far, still apply to everything in the tidyverse.

# select() function

This is another way to select variables. It can replace indexing, which is helpful when you are in these **tidyverse** code chunks (or paragraphs).

This function can take in column indexes, variable names, or both!

```
# first 3 columns only!  
firstThree <- midus %>%  
  select(1:3)
```

	ID	sex	age
1	10001	Male	61
2	10002	Male	69
6	10011	Female	52
8	10015	Female	53
10	10018	Male	49
11	10019	Male	51

# select() function

```
# BMI, both heart_self and heart_father  
otherThree <- midus %>%  
  select(BMI, 10:11)
```

	BMI	heart_self	heart_father
1	26.263	No	No
2	24.077	No	Yes
6	25.991	No	No
8	32.121	No	Yes
10	22.499	No	No
11	29.987	No	No



# select() function

To remove a variable, put a – (minus) sign in front of the variable you want to get rid of

```
# Keep all variables EXCEPT sex & physical_health_self  
removal <- midus %>%  
  select(-sex, -5)
```

	ID	age	BMI	mental_health_self	self_esteem	life_satisfaction	hostility	h
1	10001	61	26.263	4	42	7.750	5.5	N
2	10002	69	24.077	5	34	8.250	6.0	N
6	10011	52	25.991	4	41	7.000	5.5	N
8	10015	53	32.121	3	31	7.375	6.0	N
10	10018	49	22.499	4	41	8.500	6.0	N
11	10019	51	29.987	5	38	7.625	4.5	N

# mutate() function

`mutate()` is kind of tricky. On it's own, will simply add a new variable to the end of your data.frame based on something.

For example, if we wanted to get the square root of BMI...

```
sqrMidus <- midus %>%  
  mutate(BMI_sqrt = sqrt(BMI))  
  
head(sqrMidus)
```

```
##      ID    sex age   BMI physical_health_self mental_health_self self_este  
## 1 10001   Male  61 26.263                2                4  
## 2 10002   Male  69 24.077                5                5  
## 3 10011 Female  52 25.991                5                4  
## 4 10015 Female  53 32.121                3                3  
## 5 10018   Male  49 22.499                4                4  
## 6 10019   Male  51 29.987                4                5  
##   life_satisfaction hostility heart_self heart_father BMI_sqrt  
## 1                7.750         5.5         No        No 5.124744  
## 2                8.250         6.0         No        Yes 4.906832  
## 3                7.000         5.5         No        No 5.098137  
## 4                7.375         6.0         No        Yes 5.667539
```

# mutate() function

BUT, you can add different endings (suffixes) to it

- `mutate_at()`
- `mutate_all()`
- `mutate_if()`

I find `mutate_at()` to be the most useful, personally. It is especially nice for making sure the variables you need to be factors are actually factors!

Note: you can add suffixes `_at`, `_all`, and `_if` to many `tidyverse` functions! `mutate()` happens to be the one where I find this most useful, so I'm using it as an example.

# mutate() function

For example, to set up the `midus` data.frame, you were asked to make sure that `sex`, `heart_self`, and `heart_father` were all considered factors. Your code probably looked something like:

```
midus$sex <- factor(midus$sex)
midus$heart_self <- factor(midus$heart_self)
midus$heart_father <- factor(midus$heart_father)
```

When instead, it could look something like this:

```
midus <- midus %>%
  mutate_at(vars(2, 10, 11), list(factor))
```

- `vars(2, 10, 11)` says "OK, I'm going to mutate some variables. Which ones?"
- `list(factor)` says, "give me a list of functions you want me to apply to each of the variables you fed me"

Note: I have found that the help documentation for some of these functions has not updated accordingly. Search the internet and pay attention to your package

# THERE IS NO RIGHT WAY TO CODE!

Whether you used this...

```
midus$sex <- factor(midus$sex)
midus$heart_self <- factor(midus$heart_self)
midus$heart_father <- factor(midus$heart_father)
```

...or this...

```
midus <- midus %>%
  mutate_at(vars(2, 10, 11), list(factor))
```

....**doesn't matter at all!** The only things that count are:

- Were you able to do what you wanted to?
- Can YOU read the code and know what it's doing?
- Can SOMEONE ELSE read the code and know what it's doing?

# A `filter()` & `mutate_at()` example

Let's say we `filter()` so that we only have females in our data.set.

```
femalesOnly <- midus %>%  
  filter(sex == "Female")
```

In our new data.frame, the variable `sex` should only have 1 level for "Female". That is, all the "Male" participants have been removed. So as a factor, there should only be 1 category or 1 level. Let's check:

```
levels(femalesOnly$sex)
```

```
## [1] "Female" "Male"
```

Uh oh! That's not quite right.

# A `filter()` & `mutate_at()` example

Let's tell R to make `sex` into a factor again (kind of like re-populate the variable).

```
femalesOnly <- midus %>%  
  filter(sex == "Female") %>%  
  mutate_at(vars(sex), list(factor))  
  
# check the levels again  
levels(femalesOnly$sex)
```

```
## [1] "Female"
```

Now we got it! You could have first done the `filter()` function, ended the code chunk/paragraph, and then typed: `femalesOnly$sex <- factor(femalesOnly$sex)`. The downside to this is that it's nice to keep all your functions (verbs/actions) in one place, if you can.

# summarize() function

This is great for summarizing your data (*shocking, I know* 😬)

Remember that awfulness for making bar plots? This is how we can do it easily!

```
midus %>%  
  summarize(meanAge = mean(age))
```

```
##      meanAge  
## 1 56.09118
```



# summarize() function

You can go crazy with this!

```
midus %>%  
  summarize(meanAge = mean(age), # mean  
            sdAge = sd(age), # standard deviation  
            varAge = var(age), # variance  
            medianAge = median(age)) # median
```

```
##      meanAge      sdAge      varAge medianAge  
## 1 56.09118 12.30031 151.2976          55
```

**Fun fact: the person that wrote much of the tidyverse packages is from New Zealand, where they use British spellings. Therefore, summarise() is the exact same thing as summarize(). Your tab-complete might fill in the British versions!**

# group\_by() function

We can make `summarize()` even more powerful by adding the `group_by()` function.

You will NOT see anything directly change to your data.frame if you were to just run this factor. However, on the back end (behind the scenes), it tells R to do something *for each level of a categorical variable*.

If we want the mean age of those with and without heart problems:

```
midus %>%  
  group_by(heart_self) %>%  
  summarize(meanAge = mean(age))
```

```
## # A tibble: 2 x 2  
##   heart_self meanAge  
##   <fct>      <dbl>  
## 1 No        54.6  
## 2 Yes       63.0
```

# group\_by() function

We can go crazy with this too!

```
midus %>%  
  group_by(heart_self, sex) %>%  
  summarize(meanAge = mean(age),  
            sdAge = sd(age),  
            meanBMI = mean(BMI),  
            sdBMI = sd(BMI))
```

```
## # A tibble: 4 x 6  
## # Groups:   heart_self [2]  
##   heart_self sex    meanAge sdAge meanBMI sdBMI  
##   <fct>      <fct>    <dbl> <dbl>   <dbl> <dbl>  
## 1 No       Female    54.9  12.3    27.5  6.42  
## 2 No       Male     54.3  11.5    28.2  4.74  
## 3 Yes     Female    61.2  11.8    28.0  6.60  
## 4 Yes     Male     64.6  11.1    28.9  4.92
```

# Pro Tips

As you can see, the suite of `tidyverse` packages can be really, really helpful! Some things to keep in mind:

- You can put a non-tidyverse function into one of these code chunks (paragraphs)
  - If you do this, you sometimes need to give the function an input argument. Use the `.` for this.
  - Ex: `midus %>% na.omit(.)`
- You can have as many functions in each paragraph as you want. Just remember that everything is *sequential*!
  - If the output of your paragraph isn't what you think it should be, go line by line until you find the problem. Do NOT include the `%>%` when you run the line of code, though! R will wait for you to finish your sentence...

# Other useful `dp`lyr functions

- `recode()` is great for recoding variables. I especially like this for when you have something like `1` and `2` reflecting categorical variables. Recode them into something more meaningful! This is often nested within a `mutate()` or `mutate_at()` function.
- `rename()` for renaming columns
- `arrange()` will order the rows of a data.frame by some column.
- `n_distinct()` finds the number of unique entries. For example, if you have "male" and "female", the result of `n_distinct()` should be 2, even if there are thousands of rows. Now let's say there's a spelling error in one of these rows (e.g., "feemale"), now the result of `n_distinct()` will be 3...that should let you know there's a problem.
- lots & lots of others...