

# Data in/and R

EC 425/525, Lab 2

Edward Rubin

12 April 2019

# Prologue

# Schedule

## Last time

Getting to know R—objects, functions, *etc.*

## Today

Working with data in R.

- The `data.frame` class
- The `dplyr` package

## Upcoming

**Due Monday** Step 1 of our research-project proposal.

# Matrices

## Quick review

1. `mat ← matrix(data = 1:10, ncol = 2)` creates a  $5 \times 2$  `matrix` object containing the numbers 1 through 10 (filled by column).
2. `mat[1, ]` grabs the first row of our matrix `mat`.
3. `mat[3,2] ← NA` assigns `NA` to row-3 column-2 element of `mat`.
4. `head(mat, 3)` returns up to the first three rows of `mat`.
5. `matrix(data = rnorm(100), ncol = 10)` creates a  $10 \times 10$  matrix filled with random draws from  $N(\mu = 0, \sigma^2 = 1)$ .
6. `mat[3,2] ← "Carrots"` assigns the `character` object `"Carrots"` to the `[3,2]` element of `mat`, forcing all elements of `mat` to `character`.

# Matrices

## Next steps

Matrices are convenient two-dimensional arrays on which math "works."<sup>†</sup>

But matrices also require all elements to be of the same class.

Q What if we have datasets whose variables (columns) have different classes?

<sup>†</sup> At least for `numeric` and `logical` matrices.

# Matrices

## Next steps

Matrices are convenient two-dimensional arrays on which math "works."<sup>†</sup>

But matrices also require all elements to be of the same class.

**Q** What if we have datasets whose variables (columns) have different classes?

**A** We need a more flexible table-like object for our data.

<sup>†</sup> At least for `numeric` and `logical` matrices.

# Matrices

## Next steps

Matrices are convenient two-dimensional arrays on which math "works."<sup>†</sup>

But matrices also require all elements to be of the same class.

**Q** What if we have datasets whose variables (columns) have different classes?

**A** We need a more flexible table-like object for our data.

Maybe a `data.table`?

<sup>†</sup> At least for `numeric` and `logical` matrices.

# Matrices

## Next steps

Matrices are convenient two-dimensional arrays on which math "works."<sup>†</sup>

But matrices also require all elements to be of the same class.

**Q** What if we have datasets whose variables (columns) have different classes?

**A** We need a more flexible table-like object for our data.

Maybe a `data.table`? Or a `data.frame`?

<sup>†</sup> At least for `numeric` and `logical` matrices.

# Matrices

## Next steps

Matrices are convenient two-dimensional arrays on which math "works."<sup>†</sup>

But matrices also require all elements to be of the same class.

**Q** What if we have datasets whose variables (columns) have different classes?

**A** We need a more flexible table-like object for our data.

Maybe a `data.table`? Or a `data.frame`?

We'll start with `data.frame`.

<sup>†</sup> At least for `numeric` and `logical` matrices.

# Matrices

## Next steps

Matrices are convenient two-dimensional arrays on which math "works."<sup>†</sup>

But matrices also require all elements to be of the same class.

**Q** What if we have datasets whose variables (columns) have different classes?

**A** We need a more flexible table-like object for our data.

Maybe a `data.table`? Or a `data.frame`?

We'll start with `data.frame`.

We will spend a good amount of time on data frames, as they make up a huge part of your workflow.

<sup>†</sup> At least for `numeric` and `logical` matrices.

# Data frames

A `data.frame` is R's base, spreadsheet-like object that holds variables.

# Data frames

A `data.frame` is R's base, spreadsheet-like object that holds variables.

Example

# Data frames

A `data.frame` is R's base, spreadsheet-like object that holds variables.

## Example

```
#>      id first_name fave_num is_tired loves_econ
#> 1      1     Karmin      68    TRUE    FALSE
#> 2      2   Raychelle      57    TRUE     TRUE
#> 3      3    Jemelle      10    TRUE     TRUE
#> 4      4     Yusif       90    TRUE     TRUE
#> 5      5  Catherine      24    TRUE     TRUE
#> 6      6      Glory        4    TRUE     TRUE
#> 7      7     Kaelah      33   FALSE     TRUE
#> 8      8    Lysette      96    TRUE     TRUE
#> 9      9      Cisco      89    TRUE     TRUE
#> 10   10     Harman      69    TRUE     TRUE
#> 11   11   Jennelle      64    TRUE     TRUE
#> 12   12   Crayton     100    TRUE     TRUE
```

# Data frames

A `data.frame` is R's base, spreadsheet-like object that holds variables.

## Example

```
#>           name height mass gender homeworld species
#> 1      Luke Skywalker    172   77 male Tatooine Human
#> 2          C-3PO       167   75 <NA> Tatooine Droid
#> 3          R2-D2       96   32 <NA> Naboo Droid
#> 4      Darth Vader     202  136 male Tatooine Human
#> 5      Leia Organa     150   49 female Alderaan Human
#> 6      Owen Lars      178  120 male Tatooine Human
#> 7 Beru Whitesun lars   165   75 female Tatooine Human
#> 8          R5-D4       97   32 <NA> Tatooine Droid
#> 9      Biggs Darklighter 183   84 male Tatooine Human
#> 10     Obi-Wan Kenobi   182   77 male Stewjon Human
#> 11 Anakin Skywalker    188   84 male Tatooine Human
#> 12     Wilhuff Tarkin   180    NA male Eriadu Human
```

# Data frames

## Creation

The `data.frame()` function creates...

# Data frames

## Creation

The `data.frame()` function creates... `data.frame` objects.

# Data frames

## Creation

The `data.frame()` function creates... `data.frame` objects.

You'll generally define data frames by passing the function  
**(1)** column names and **(2)** values for the columns.

```
data.frame(var1 = 1:5, var2 = "apple", var3 = rnorm(5))
```

# Data frames

## Creation

The `data.frame()` function creates... `data.frame` objects.

You'll generally define data frames by passing the function  
**(1)** column names and **(2)** values for the columns.

```
data.frame(var1 = 1:5, var2 = "apple", var3 = rnorm(5))
```

You can also assign the values using already-existing objects, e.g.,

```
# An object with value
tmp ← rnorm(5)
# Creating the data frame
data.frame(var1 = 1:5, var2 = "apple", var3 = tmp)
```

# Data frames

## Creation

```
# Creating the data frame  
data.frame(var1 = 1:5, var2 = "apple", var3 = rnorm(5))
```

```
#>   var1   var2       var3  
#> 1     1 apple -0.6250393  
#> 2     2 apple -1.6866933  
#> 3     3 apple  0.8377870  
#> 4     4 apple  0.1533731  
#> 5     5 apple -1.1381369
```

(What a beauty.)

# Data frames

## Creation

```
# Creating the data frame  
data.frame(var1 = 1:5, var2 = "apple", var3 = rnorm(5))
```

```
#>   var1   var2       var3  
#> 1     1 apple -0.6250393  
#> 2     2 apple -1.6866933  
#> 3     3 apple  0.8377870  
#> 4     4 apple  0.1533731  
#> 5     5 apple -1.1381369
```

(What a beauty.)

Notice that R assumes we want to repeat "apple" for the entire column.

# Data frames

## Creation

You can also create data frames from other objects (e.g., matrices) using the function `as.data.frame()`<sup>†</sup>.

However, your data frame's columns will only have names if your matrix's columns had names.

<sup>†</sup> Or just plain, old `data.frame()`.

# Data frames

## Indexing

Consider a data frame `our_df <- data.frame(x = 1:3, y = 4:6, z = 7:9)`.

**Option 1** Index data frames just as you index matrices in R.

- `our_df[1,1]` grabs the value in the first row of the first variable.
- `our_df[2, ]` returns the second row of `our_df` (as a data frame).
- `our_df[,3]` returns the third column (variable) of `our_df` (as a vector).

# Data frames

## Indexing

Consider a data frame `our_df ← data.frame(x = 1:3, y = 4:6, z = 7:9)`.

**Option 1** Index data frames just as you index matrices in R.

- `our_df[1,1]` grabs the value in the first row of the first variable.
- `our_df[2, ]` returns the second row of `our_df` (as a data frame).
- `our_df[,3]` returns the third column (variable) of `our_df` (as a vector).

**Option 2** Reference values/variables using columns' names.

- `our_df$x` returns the column named `x` (as a vector). **New:** \$
- `our_df[, "x"]` returns the column named `x` (as a vector).
- `our_df["x"]` returns the column named `x` (as a data frame).
- `our_df[, c("x", "y")]` returns a data frame with variables "x" and "y".

# Data frames

## Names (of columns)

The columns (variables) in your data frame have names.<sup>†</sup>

Q What if you want to see/know those names?

<sup>†</sup> If you don't name the columns, then R will.

# Data frames

## Names (of columns)

The columns (variables) in your data frame have names.<sup>†</sup>

**Q** What if you want to see/know those names?

**A** You've got a few options.

<sup>†</sup> If you don't name the columns, then R will.

# Data frames

## Names (of columns)

The columns (variables) in your data frame have names.<sup>†</sup>

**Q** What if you want to see/know those names?

**A** You've got a few options.

1. The `names()` function returns the *names* of an object.

<sup>†</sup> If you don't name the columns, then R will.

# Data frames

## Names (of columns)

The columns (variables) in your data frame have names.<sup>†</sup>

**Q** What if you want to see/know those names?

**A** You've got a few options.

1. The `names()` function returns the *names* of an object.
2. `head(your_df)` will show you the first 6 rows of `your_df`.

*Note:* May provide too much output if you have a lot of columns.

<sup>†</sup> If you don't name the columns, then R will.

# Data frames

## Names (of columns)

The columns (variables) in your data frame have names.<sup>†</sup>

**Q** What if you want to see/know those names?

**A** You've got a few options.

1. The `names()` function returns the *names* of an object.
2. `head(your_df)` will show you the first 6 rows of `your_df`.  
*Note:* May provide too much output if you have a lot of columns.
3. In RStudio: `View(your_df)` or look in your Environment tab.

<sup>†</sup> If you don't name the columns, then R will.

# Data frames

## Naming

The `names()` function will also help you rename any/all variables.

# Data frames

## Naming

The `names()` function will also help you rename any/all variables.

Change the names of **all variables** (include a name for each variable):

```
# Set new names  
names(our_df) ← c("name1", "name2", "name3")
```

# Data frames

## Naming

The `names()` function will also help you rename any/all variables.

Change the names of **all variables** (include a name for each variable):

```
# Set new names
names(our_df) ← c("name1", "name2", "name3")
```

Change the name of **the second variable** (only):

```
# Set new names
names(our_df)[2] ← "name2"
```

# Data frames

## Adding variables

Just as we referenced **existing** variables using `$var_name`,  
we can create **new** variables using `$new_var`, e.g.,

```
# Add a variable to our_df  
our_df$new_var ← 1:100
```

# Data frames

## Adding variables

Just as we referenced **existing** variables using `$var_name`,  
we can create **new** variables using `$new_var`, e.g.,

```
# Add a variable to our_df  
our_df$new_var ← 1:100
```

If you want to use existing columns to create a new variable

```
# Create interaction: xy = x * y  
our_df$xy ← our_df$x * our_df$y
```

# Data frames

## Adding variables

Just as we referenced **existing** variables using `$var_name`,  
we can create **new** variables using `$new_var`, e.g.,

```
# Add a variable to our_df  
our_df$new_var ← 1:100
```

If you want to use existing columns to create a new variable

```
# Create interaction: xy = x * y  
our_df$xy ← our_df$x * our_df$y
```

**Q** Isn't there a better/faster/less-typing way?

# Data frames

## Adding variables

Just as we referenced **existing** variables using `$var_name`,  
we can create **new** variables using `$new_var`, e.g.,

```
# Add a variable to our_df  
our_df$new_var ← 1:100
```

If you want to use existing columns to create a new variable

```
# Create interaction: xy = x * y  
our_df$xy ← our_df$x * our_df$y
```

**Q** Isn't there a better/faster/less-typing way?

**A** Yes. Enter `dplyr`

# Data frames

## Adding variables

Just as we referenced **existing** variables using `$var_name`, we can create **new** variables using `$new_var`, e.g.,

```
# Add a variable to our_df  
our_df$new_var ← 1:100
```

If you want to use existing columns to create a new variable

```
# Create interaction: xy = x * y  
our_df$xy ← our_df$x * our_df$y
```

**Q** Isn't there a better/faster/less-typing way?

**A** Yes. Enter `dplyr` (also: `data.table`, which we'll leave for the future).

# dplyr

## Intro

It's a package.

# dplyr

## Intro

It's a package. `dplyr` is not installed by default, so you'll need to install it.<sup>†</sup>

<sup>†</sup> or just `p_load(dplyr)` after loading `pacman`.

# dplyr

## Intro

It's a package. `dplyr` is not installed by default, so you'll need to install it.<sup>†</sup>

`dplyr` is part of the `tidyverse` (Hadleyverse), and it follows a grammar-based approach to programming/data work.

<sup>†</sup> or just `p_load(dplyr)` after loading `pacman`.

# dplyr

## Intro

It's a package. `dplyr` is not installed by default, so you'll need to install it.<sup>†</sup>

`dplyr` is part of the `tidyverse` (Hadleyverse), and it follows a grammar-based approach to programming/data work.

- `data` compose the subjects of your stories
- `dplyr` provides the *verbs* (action words) :  
`filter()`, `mutate()`, `select()`, `group_by()`, `summarize()`, `arrange()`

<sup>†</sup> or just `p_load(dplyr)` after loading `pacman`.

# dplyr

## Intro

It's a package. `dplyr` is not installed by default, so you'll need to install it.<sup>†</sup>

`dplyr` is part of the `tidyverse` (Hadleyverse), and it follows a grammar-based approach to programming/data work.

- `data` compose the subjects of your stories
- `dplyr` provides the *verbs* (action words) :  
`filter()`, `mutate()`, `select()`, `group_by()`, `summarize()`, `arrange()`

**Bonus** `dplyr` is pretty fast and able to interact with SQL databases.

<sup>†</sup> or just `p_load(dplyr)` after loading `pacman`.

# dplyr

## Manipulating variables: `mutate()`

`dplyr` streamlines adding/manipulating variables in your data frame.

### Function `mutate(.data, ...)`

- Required argument `.data`, an existing data frame
- Additional arguments Names and values of the new variables
- Output An updated data frame

# dplyr

## Manipulating variables: `mutate()`

`dplyr` streamlines adding/manipulating variables in your data frame.

### Function `mutate(.data, ...)`

- **Required argument** `.data`, an existing data frame
- **Additional arguments** Names and values of the new variables
- **Output** An updated data frame

### Example

```
mutate(.data = our_df, new1 = 7, new2 = x * y)
```

# dplyr

## mutate()

Example Take the data frame

```
my_df <- data.frame(x = 1:4, y = 5:8)
```

# dplyr

## mutate()

Example Take the data frame

```
my_df <- data.frame(x = 1:4, y = 5:8)
```

mutate() allows us to create many new variables with one call.

```
mutate(.data = my_df,  
       xy = x * y,  
       x2 = x^2,  
       y2 = y^2,  
       xy2 = xy^2,  
       is_x_max = x == max(x)  
)
```

# dplyr

## mutate()

Example Take the data frame

```
my_df <- data.frame(x = 1:4, y = 5:8)
```

mutate() allows us to create many new variables with one call.

```
mutate(.data = my_df,  
       xy = x * y,  
       x2 = x^2,  
       y2 = y^2,  
       xy2 = xy^2,  
       is_x_max = x == max(x))
```

```
#>   x y xy x2 y2 xy2 is_x_max  
#> 1 1 5 5  1 25  25 FALSE  
#> 2 2 6 12 4 36 144 FALSE  
#> 3 3 7 21 9 49 441 FALSE  
#> 4 4 8 32 16 64 1024 TRUE
```

Notice mutate() returns the original *and* new columns.

# dplyr

## mutate() vs. transmute()

As their names imply, `mutate()` and `transmute()` are very similar functions.

- `mutate()` returns the *original and new* columns (variables).
- `transmute()` returns only the *new* columns (variables).

# dplyr

## mutate() vs. transmute()

As their names imply, `mutate()` and `transmute()` are very similar functions.

- `mutate()` returns the *original* and *new* columns (variables).
- `transmute()` returns only the *new* columns (variables).

*Note* Both functions return a new object as *output*—they do not update the object in R's memory. (This is the case for all functions in `dplyr`.)

# dplyr

## Pipes

We can't go much deeper into the land of `dplyr` without mentioning pipes.

# dplyr

## Pipes

We can't go much deeper into the land of `dplyr` without mentioning pipes.

A *pipe* in programming allows you to take the output of one function and plug it into another function as an argument/input.

# dplyr

## Pipes

We can't go much deeper into the land of `dplyr` without mentioning pipes.

A *pipe* in programming allows you to take the output of one function and plug it into another function as an argument/input.

In `dplyr`, the expression for a pipe is `%>%`.

# dplyr

## Pipes

We can't go much deeper into the land of `dplyr` without mentioning pipes.

A *pipe* in programming allows you to take the output of one function and plug it into another function as an argument/input.

In `dplyr`, the expression for a pipe is `%>%`.

R's pipe specifically plugs the returned object to the *left* of the pipe into the first argument of the function on the *right* of the pipe, e.g.,

# dplyr

## Pipes

We can't go much deeper into the land of `dplyr` without mentioning pipes.

A *pipe* in programming allows you to take the output of one function and plug it into another function as an argument/input.

In `dplyr`, the expression for a pipe is `%>%`.

R's pipe specifically plugs the returned object to the *left* of the pipe into the first argument of the function on the *right* of the pipe, e.g.,

```
rnorm(10) %>% mean()
```

```
#> [1] 0.4854731
```

# dplyr

## Pipes

Pipes help avoid lots of nested functions, prevent excessive writing to your disc, and increase the readability of our R scripts.

# dplyr

## Pipes

Pipes help avoid lots of nested functions, prevent excessive writing to your disc, and increase the readability of our R scripts.

**Example** Three ways to draw 100  $N(0,1)$  observations and calculate the interquartile range (IQR: difference between the 75<sup>th</sup> and 25<sup>th</sup> percentiles).

```
# Save each intermediate step
draw ← rnorm(100)
end_points ← quantile(draw, probs = c(0.25, 0.75))
diff(end_points)

# Lots of nesting
diff(quantile(rnorm(100), probs = c(0.25, 0.75)))

# Piping 🍽
rnorm(100) %>% quantile(probs = c(0.25, 0.75)) %>% diff()
```

# dplyr

## Pipes

By default, R pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

# dplyr

## Pipes

By default, R pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

*E.g.,* `a %>% fun(3)` is equivalent to `fun(arg1 = a, arg2 = 3)`.

# dplyr

## Pipes

By default, R pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

E.g., `a %>% fun(3)` is equivalent to `fun(arg1 = a, arg2 = 3)`.

If you want to pipe output into a different argument, you use a period (`.`).

# dplyr

## Pipes

By default, R pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

E.g., `a %>% fun(3)` is equivalent to `fun(arg1 = a, arg2 = 3)`.

If you want to pipe output into a different argument, you use a period (`.`).

- `b %>% fun(arg1 = 3, .)` is equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(3, .)` is also equivalent to `fun(arg1 = 3, arg2 = b)`.

# dplyr

## Pipes

By default, R pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

E.g., `a %>% fun(3)` is equivalent to `fun(arg1 = a, arg2 = 3)`.

If you want to pipe output into a different argument, you use a period (`.`).

- `b %>% fun(arg1 = 3, .)` is equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(3, .)` is also equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(., .)` is equivalent to `fun(arg1 = b, arg2 = b)`.

# dplyr

## Pipes

By default, R pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

E.g., `a %>% fun(3)` is equivalent to `fun(arg1 = a, arg2 = 3)`.

If you want to pipe output into a different argument, you use a period (`.`).

- `b %>% fun(arg1 = 3, .)` is equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(3, .)` is also equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(., .)` is equivalent to `fun(arg1 = b, arg2 = b)`.

The `magrittr` package contains even more piping power.<sup>†</sup>

<sup>†</sup> `magrittr` = Magritte (of *this is not a pipe* fame) plus R.

# dplyr

## %>% and dplyr

Each `dplyr` function begins with a `.data` argument so that you can easily pipe in data frames (recall: `mutate(.data, ...)`).

# dplyr

## %>% and dplyr

Each `dplyr` function begins with a `.data` argument so that you can easily pipe in data frames (recall: `mutate(.data, ...)`).

The common workflow in `dplyr` will look something like

```
new_df ← old_df %>% mutate(cool stuff here)
```

which takes `old_df`, does some cool stuff with `mutate()`, and then saves the output of `mutate()` as `new_df`.

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Only keep rows where x is 3
some_df %>% filter(x = 3)
#>     x   y
#> 1  3 13
```

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Only keep rows where x > 7
some_df %>% filter(x > 7)

#>      x   y
#> 1   8 18
#> 2   9 19
#> 3 10 20
```

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Keep rows where y/x > 3
some_df %>% filter(y/x > 3)

#>     x     y
#> 1  1 11
#> 2  2 12
#> 3  3 13
#> 4  4 14
```

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Keep rows where x>7 OR y<12
some_df %>%
  filter(x > 7 | y < 12)
```

```
#>      x   y
#> 1   1 11
#> 2   8 18
#> 3   9 19
#> 4 10 20
```

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Keep rows where 15 ≤ y ≤ 18
some_df %>%
  filter(between(y, 15, 18))
```

```
#>     x   y
#> 1  5 15
#> 2  6 16
#> 3  7 17
#> 4  8 18
```

# dplyr

## filter()

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

### Example

```
# Create a dataset
some_df ← data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Keep rows where y > 20
some_df %>% filter(y > 20)

#> [1] x y
#> <0 rows> (or 0-length row.names)
```

If you filter your data frame down to nothing, R returns a 0-row data frame with the names/number of columns from the original data frame.

# dplyr

## select()

Just as `filter()` grabs row-based subsets of your data frame,  
`select()` grabs column-based subsets.

# dplyr

## select()

Just as `filter()` grabs row-based subsets of your data frame,  
`select()` grabs column-based subsets.

You can select columns using their **names**

```
our_df %>% select(var10, var100)
```

# dplyr

## select()

Just as `filter()` grabs row-based subsets of your data frame,  
`select()` grabs column-based subsets.

You can select columns using their **names**

```
our_df %>% select(var10, var100)
```

you can select columns using their **numbers**

```
our_df %>% select(10, 100)
```

# dplyr

## select()

Just as `filter()` grabs row-based subsets of your data frame, `select()` grabs column-based subsets.

You can select columns using their **names**

```
our_df %>% select(var10, var100)
```

you can select columns using their **numbers**

```
our_df %>% select(10, 100)
```

or you can select columns using **helper functions**

```
our_df %>% select(starts_with("var10"))
```

# dplyr

## select()

Just as `filter()` grabs row-based subsets of your data frame, `select()` grabs column-based subsets.

You can select columns using their **names**

```
our_df %>% select(var10, var100)
```

you can select columns using their **numbers**

```
our_df %>% select(10, 100)
```

or you can select columns using **helper functions**

```
our_df %>% select(starts_with("var10"))
```

`select()` helps you narrow down a dataset to its necessary features.

# dplyr

## summarize()

Hopefully you're starting to see that functions' names in `dplyr` tell you what the function does.

`summarize()`<sup>†</sup> summarizes variables—you choose the variables and the summaries (e.g., `mean()` or `min()`).

<sup>†</sup> or `summarise()` if you ❤️ 🇬🇧

# dplyr

## summarize()

Hopefully you're starting to see that functions' names in `dplyr` tell you what the function does.

`summarize()`<sup>†</sup> summarizes variables—you choose the variables and the summaries (e.g., `mean()` or `min()`).

```
the_df %>% summarize(  
  mean(x), mean(y), mean(z),  
  min(x), max(x),  
)
```

would return a  $1 \times 5$  data frame with the means of `x`, `y`, and `z`; the minimum of `x`; and the maximum of `x`.

<sup>†</sup> or `summarise()` if you ❤️ 🇬🇧

# dplyr

## summarize() and group\_by()

While sample-wide summarizes are certainly interesting, `dplyr` has one last gem for us: `group_by()`.

`group_by()` groups your observations by the variable(s) that you name.

# dplyr

## summarize() and group\_by()

While sample-wide summarizes are certainly interesting, `dplyr` has one last gem for us: `group_by()`.

`group_by()` groups your observations by the variable(s) that you name.

Specifically, `group_by()` returns a *grouped data frame* that you can then feed to `summarize()`, `mutate()`, or `transmute` to perform grouped calculations, e.g., each group's mean.

# dplyr

## Example: Grouped summaries

```
# Create a new data frame
our_df <- data.frame(
  x = 1:6,
  y = c(0, 1),
  grp = rep(c("A", "B"), each = 3)
)
```

```
#>   x y grp
#> 1 1 0   A
#> 2 2 1   A
#> 3 3 0   A
#> 4 4 1   B
#> 5 5 0   B
#> 6 6 1   B
```

# dplyr

## Example: Grouped summaries

```
# Create a new data frame
our_df ← data.frame(
  x = 1:6,
  y = c(0, 1),
  grp = rep(c("A", "B"), each = 3)
)
```

```
#>   x y grp
#> 1 1 0 A
#> 2 2 1 A
#> 3 3 0 A
#> 4 4 1 B
#> 5 5 0 B
#> 6 6 1 B
```

```
# For dataset 'our_df' ...
our_df %>%
  # Group by 'grp'
  group_by(grp) %>%
  # Take means of 'x' and 'y'
  summarize(mean(x), mean(y))
```

```
#> # A tibble: 2 x 3
#>   grp     `mean(x)` `mean(y)`
#>   <fct>      <dbl>      <dbl>
#> 1 A          2        0.333
#> 2 B          5        0.667
```

# dplyr

## Example: Grouped mutation

```
# Create a new data frame
our_df <- data.frame(
  x = 1:6,
  y = c(0, 1),
  grp = rep(c("A", "B"), each = 3)
)
```

```
#>   x y grp
#> 1 1 0   A
#> 2 2 1   A
#> 3 3 0   A
#> 4 4 1   B
#> 5 5 0   B
#> 6 6 1   B
```

# dplyr

## Example: Grouped mutation

```
# Create a new data frame
our_df ← data.frame(
  x = 1:6,
  y = c(0, 1),
  grp = rep(c("A", "B"), each = 3)
)
```

```
#>   x y grp
#> 1 1 0 A
#> 2 2 1 A
#> 3 3 0 A
#> 4 4 1 B
#> 5 5 0 B
#> 6 6 1 B
```

```
# Add grp means for x and y
our_df %>%
  group_by(grp) %>%
  mutate(
    x_m = mean(x), y_m = mean(y)
  )
```

```
#> # A tibble: 6 x 5
#> # Groups:   grp [2]
#>       x     y   grp     x_m     y_m
#>   <int> <dbl> <fct> <dbl> <dbl>
#> 1     1     0   A     0.333  0.333
#> 2     2     1   A     0.333  0.333
#> 3     3     0   A     0.333  0.333
#> 4     4     1   B     0.667  0.667
#> 5     5     0   B     0.667  0.667
#> 6     6     1   B     0.667  0.667
```

# dplyr

## arrange()

arrange() will sorts the rows of a data frame using the inputted columns.

R defaults to starting with the "lowest" (smallest) at the top of the data frame. Use a - in front of the variable's name to reverse sort.

```
# As is  
our_df
```

```
#>     x y grp  
#> 1 1 0 A  
#> 2 2 1 A  
#> 3 3 0 A  
#> 4 4 1 B  
#> 5 5 0 B  
#> 6 6 1 B
```

```
# As is  
our_df %>% arrange(y, grp, -x)
```

```
#>     x y grp  
#> 1 3 0 A  
#> 2 1 0 A  
#> 3 5 0 B  
#> 4 2 1 A  
#> 5 6 1 B  
#> 6 4 1 B
```

# Table of contents

## Data and R

1. Schedule
2. Matrix review
3. The `data.frame`
  - o Basic examples
  - o Creating
  - o Indexing
  - o Names
  - o Adding variables

## dplyr

1. Intro
2. `mutate()`
3. `transmute()`
4. Pipes (`%>%`)
5. `filter()`
6. `select()`
7. `summarize`
8. `summarize()` and `group_by()`
9. `arrange()`