Data Manipulation with dplyr (basics)

Outline

The dplyr package

Tidy data

The basic functions

- Manipulating rows: filter(), arrange(), slice()
- Manipulating columns: mutate(), select(), summarize()
- Others: group_by()

dplyr

The *dplyr* package is a part of the *tidyverse* and is the main package for data manipulation (grammar for data manipulation)

dplyr expects that the data will be the tidy format!

If you know the querying language SQL, the verbs (functions) are going to sound very similar

dplyr functions all have similar form:

- First argument is the data (data frame/tibble)
- Subsequent arguments describe your proposed actions
- Result is a data frame / tibble

Tidy Data

Tidy data is the mantra of the *tidyverse*!

Basic principles of tidy data:

Every row is a case (person/place/thing being observed)

Every column is a variable

Hadley Wickham (creator of *tidyverse*) wrote a paper building up the theory of tidy data: You can find it here

We will spend a large amount of time on tidying data (*dplyr* and *tidyr*)!

Data Transformation with dplyr : : **CHEAT SHEET**

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column**

&

Each **observation**, or **case**, is in its own **row**

pipes

x %>% f(y) becomes **f(x, y)**

Summarise Cases

These apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summarise(.data, …)
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
summarise_if() - Apply funs to all cols of one type.

Group Cases

Use **group_by()** to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

group_by(.data, ..., add = FALSE**)**
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

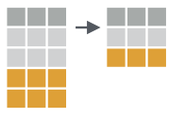
ungroup(x, …**)**
Returns ungrouped copy of table.
ungroup(g_iris)

Manipulate Cases

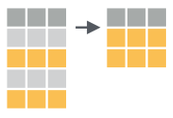
EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, …**)** Extract rows that meet logical criteria. *filter(iris, Sepal.Length > 7)*

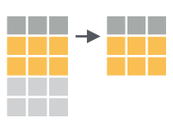


distinct(.data, ..., .keep_all = FALSE**)** Remove rows with duplicate values.
distinct(iris, Species)



sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()**)** Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()**)** Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*



slice(.data, …**)** Select rows by position.
slice(iris, 10:15)

top_n(x, n, wt**)** Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

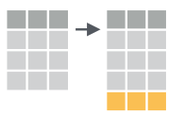
<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::Logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, …**)** Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES



add_row(.data, ..., .before = NULL, .after = NULL**)**
Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1**)** Extract column values as a vector. Choose by name or index.
pull(iris, Sepal.Length)

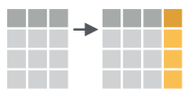
select(.data, …**)**
Extract columns as a table. Also **select_if()**.
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match**)**	**num_range(**prefix, range**)**	**:**, e.g. mpg:cyl
ends_with(match**)**	**one_of(**…**)**	**-**, e.g, -Species
matches(match**)**	**starts_with(**match**)**	

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).

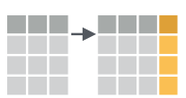


mutate(.data, …**)**
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, …**)**
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

mutate_all(.tbl, .funs, …**)** Apply funs to every column. Use with **funs()**. Also **mutate_if()**.
mutate_all(faithful, funs(log(.), log2(.)))
mutate_if(iris, is.numeric, funs(log(.)))

mutate_at(.tbl, .cols, .funs, …**)** Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().
mutate_at(iris, vars(-Species), funs(log(.)))



add_column(.data, ..., .before = NULL, .after = NULL**)** Add new column(s). Also **add_count()**, **add_tally()**. *add_column(mtcars, new = 1:32)*

rename(.data, …**)** Rename columns.
rename(iris, Length = Sepal.Length)

filter()

filter() manipulates rows by keeping qualifying rows

```
filter(data, ...)
```

`data:` The data to be manipulated

`...` : One or more logical tests to match

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
    1     1     a
    2     2     b
    3     2     a
    4     3     b
    5     1     a
```

```
tb %>% filter(x == 1)

# A tibble: 1 x 3
      x     y     c
  <int> <int> <chr>
    1     1     a
```

filter()

What logical test can we use inside of filter?

Logical and boolean operators to use with filter()

<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::Logic** and **?Comparison** for help.

x < y	Less than
x > y	Greater than
x == y	Equal to
x <= y	Less than or equal to
x >= y	Greater than or equal to
x != y	Not equal to
x %in% y	Group membership
is.na(x)	Is NA
!is.na(x)	Is not NA

filter()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     2     2     a
4     3     3     b
5     1     1     a
```

Common mistakes

```
tb %>% filter(x = 1)
```

✘

```
tb %>% filter(c == a)
```

What's wrong with these function calls?

```
tb %>% filter(x == 1)
```

✓

```
tb %>% filter(c == "a")
```

filter()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

Filter can take in multiple logical tests!

If you supply them with the comma, they will be combined with “and”

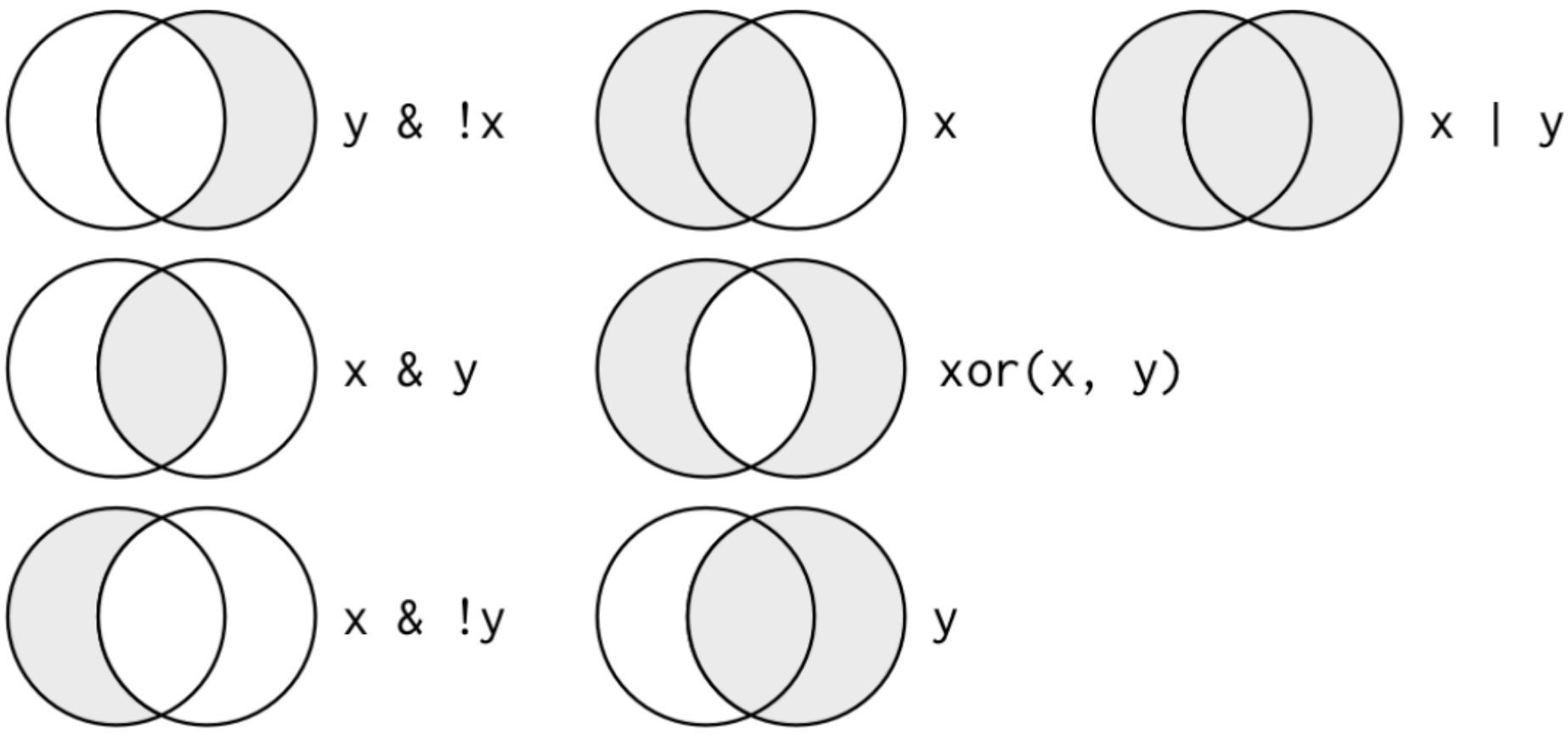
The same operation can be done with boolean operators!

```
tb %>% filter(y == 2, c == "a")    =    tb %>% filter(y == 2 & c == "a")
```

?base::Logic

a & b	and
a | b	or
xor(a, b)	exactly or
!a	not

filter()



Credit: R4DS

Your Turn!

The dplyr package has a dataset called *starwars*

I want you to filter out all of the humans in the dataset!

arrange()

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
2     2     2 b
3     3     2 a
4     4     3 b
5     5     1 a
```

arrange() arranges rows! By default, in ascending order

```
tb %>% arrange(y)
```

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
2     5     1 a
3     2     2 b
4     3     2 a
5     4     3 b
```

```
tb %>% arrange(desc(y))
```

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
1     4     3 b
2     2     2 b
3     3     2 a
4     1     1 a
5     5     1 a
```

desc() will sort by descending order

You can also arrange based on multiple variables

arrange()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

You can also arrange based on multiple variables!

```
tb %>% arrange(y,c)
```

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
2     5     1 a
3     3     2 a
4     2     2 b
5     4     3 b
```

```
tb %>% arrange(y,desc(c))
```

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
2     5     1 a
3     2     2 b
4     3     2 a
5     4     3 b
```

Ties are broken from left to right with multiple variables

slice()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

slice() allows you to select certain rows from the data

```
tb %>% slice(1)
```

```
# A tibble: 1 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
```

```
tb %>% slice(1:3)
```

```
# A tibble: 3 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
2     2     2 b
3     3     2 a
```

```
tb %>% slice(-c(2,4))
```

```
# A tibble: 3 x 3
      x     y c
  <int> <int> <chr>
1     1     1 a
2     3     2 a
3     5     1 a
```

Your Turn!

I want you to arrange the data by height so that the tallest characters are at the top! Now show me the top 5 tallest characters

mutate()

mutate() allows you to create and/or modify columns

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

```
tb %>% mutate(z = x+y)
```

```
# A tibble: 5 x 4
      x     y c         z
  <int> <int> <chr> <int>
1     1     1 a         2
2     2     2 b         4
3     3     2 a         5
4     4     3 b         7
5     5     1 a         6
```

```
tb %>% mutate(z = x+y, w = x*y)
```

```
# A tibble: 5 x 5
      x     y c         z     w
  <int> <int> <chr> <int> <int>
1     1     1 a         2     1
2     2     2 b         4     4
3     3     2 a         5     6
4     4     3 b         7    12
5     5     1 a         6     5
```

mutate()

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
      1     1 a
      2     2 b
      3     2 a
      4     3 b
      5     1 a
```

mutate() allows you to create and/or modify columns

```
tb %>% mutate(z = x+y, z2 = z^2)
```

```
# A tibble: 5 x 5
      x     y c         z    z2
  <int> <int> <chr> <int> <dbl>
      1     1 a         2     4
      2     2 b         4    16
      3     2 a         5    25
      4     3 b         7    49
      5     1 a         6    36
```

Rolls the computations over

```
tb %>% mutate(y = x+y)
```

```
# A tibble: 5 x 3
      x     y c
  <int> <int> <chr>
      1     2 a
      2     4 b
      3     5 a
      4     7 b
      5     6 a
```

Replaces variables as well

select()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

select() allows you select/deselect certain columns

```
tb %>% select(x,y)
```

```
# A tibble: 5 x 2
      x     y
  <int> <int>
1     1     1
2     2     2
3     3     2
4     4     3
5     5     1
```

```
tb %>% select(1,2)
```

```
# A tibble: 5 x 2
      x     y
  <int> <int>
1     1     1
2     2     2
3     3     2
4     4     3
5     5     1
```

```
tb %>% select(1:2)
```

```
# A tibble: 5 x 2
      x     y
  <int> <int>
1     1     1
2     2     2
3     3     2
4     4     3
5     5     1
```

Names are preferred if possible for more readable code

select()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

select() allows you select/deselect certain columns

```
tb %>% select(-3)
```

```
# A tibble: 5 x 2
      x     y
  <int> <int>
1     1     1
2     2     2
3     3     2
4     4     3
5     5     1
```

```
tb %>% select(-c)
```

```
# A tibble: 5 x 2
      x     y
  <int> <int>
1     1     1
2     2     2
3     3     2
4     4     3
5     5     1
```

select()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     2     2     a
4     3     3     b
5     1     1     a
```

There are different helper functions that can make selecting variables easier!

```
tb %>% select(one_of(c("x", "y")))
```

```
# A tibble: 5 x 2
      x     y
  <int> <int>
1     1     1
2     2     2
3     2     2
4     3     3
5     1     1
```

Helper functions
starts_with("x")
ends_with("x")
contains("x")
one_of(c("x", "y", "z"))

These functions have to do with regular expressions.
We will talk about these later!

Your Turn!

Create another column called *ratio* that contains the height to mass ratio (height the numerator and mass the denominator), then select the name and ratio, and arrange so we can see the lowest ratio people.

summarize()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

summarize() applies a summary function that creates a real-valued statistic

```
tb %>% summarize(xbar = mean(x))
```

```
# A tibble: 1 x 1
   xbar
  <dbl>
1     3
```

There are lots of variants of summarize as well!

We will usually see summarize() used in conjunction with group_by()

group_by()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     1     1     a
```

group_by() groups certain variables together to enable conditional computation. We will usually use group_by() in conjunction with summarize() to create statistics for each group

group_by() enables the popular split-apply-combine strategy seamlessly.

```
diamonds %>%
 group_by(cut) %>%
 summarize(mean = mean(price))
```

```
# A tibble: 5 x 2
  cut         mean
  <ord>      <dbl>
1 Fair       4359.
2 Good       3929.
3 Very Good  3982.
4 Premium    4584.
5 Ideal      3458.
```

Your Turn!

Count how many characters are from each *homeworld* using the *count* function

Find the average height of characters from each *homeworld* using the *group_by / summarize* combo

across()

```
# A tibble: 5 x 3
      x     y     c
  <int> <int> <chr>
1     1     1     a
2     2     2     b
3     3     2     a
4     4     3     b
5     5     1     a
```

across() allows you to apply the same transformation to multiple columns. It is successor to mutate and summarize variants, and thus usually used inside mutate and summarize

```
tb %>%
mutate(across(x:y, mean))
```

```
# A tibble: 5 × 3
      x     y   z
  <dbl> <dbl> <chr>
1     3   1.8   a
2     3   1.8   b
3     3   1.8   a
4     3   1.8   b
5     3   1.8   a
```

```
iris %>%
    group_by(Species) %>%
    summarize(across(starts_with("Sepal"),
list(mean = mean, sd = sd)))
```

```
# A tibble: 3 × 5
  Species    Sepal.Length_mean Sepal.Length_sd
  <fct>                  <dbl>           <dbl>
1 setosa                  5.01           0.352
2 versicolor              5.94           0.516
3 virginica               6.59           0.636
# … with 2 more variables: Sepal.Width_mean <dbl>,
#   Sepal.Width_sd <dbl>
```

Other functions!

There are many other functions in *dplyr,* and we will discuss some later in detail (combining datasets), but here are some others quickly!

count() - count the number of rows in each group (used with group_by)

ungroup() - removes the grouping (combine part of split-apply-combine)

distinct() - Remove rows with duplicate values

slice_sample() - Samples a specified fraction or number of the data (important in the modeling part of this course sequence)

slice_min()/slice_max - Selects top n or bottom n rows of data

add_row() - Add another row of data

pull() - Select a certain column (Basically select() but returns a vector not a data frame)

transmute() - Like mutate() but drops all other columns

rowwise() - Helps to do rowwise summarizations. Row version of group_by

Other functions!

add_column()/add_tally()/add_count() - All three add a column to the dataset, just in different ways

rename() - rename column names

rownames_to_column() - moves row names to a column of data. Tidy data does not have row names.

column_to_rownames() - Opposite of above function

Lots of joining functions that will be described in a later slide-deck: bind_cols(), full_join(), semi_join(), left_join(), right_join(), inner_join(), anti_join(), bind_rows(), intersect(), setdiff(), union(), etc.

A full list of functions from dplyr can be found here!

dplyr extensions

dplyr is extremely popular for data manipulation, but it does have some downfalls, primarily with large datasets and computations!

There are dplyr “extension” packages that make working with other computational backends accessible and easy!

dtplyr: translates your dplyr code to code that will run in the highly optimized data.table R package.

dbplyr: translates your dplyr code to SQL code and will query from databases.

sparklyr: translates code and interfaces with Apache Spark