CSSS508, Week 3 Manipulating and Summarizing Data Chuck Lanfear Apr 14, 2021 Updated: Apr 18, 2021



Death to Spreadsheets

Today we'll talk more about dplyr: a package that does in R just about any calculation you've tried to do in Excel, but more *transparently*, *reproducibly*, and *safely*.

Don't be the next sad research assistant who makes headlines with an Excel error (<u>Reinhart & Rogoff, 2010</u>)

Modifying Data Frames with dplyr



But First, Pipes (%>%)

dplyr uses the magrittr forward pipe operator, usually called simply a
pipe. We write pipes like %>% (Ctrl+Shift+M).

Pipes take the object on the *left* and apply the function on the *right*: x %>% f(y) = f(x, y). Read out loud: "and then..."

library(dplyr)
library(gapminder)
gapminder %>% filter(country == "Canada") %>% head(2)

A tibble: 2 x 6
country continent year lifeExp pop gdpPercap
<fct> <fct> <int> <dbl> <int> <dbl>
1 Canada Americas 1952 68.8 14785584 11367.
2 Canada Americas 1957 70.0 17010154 12490.

W CS&SS

Pipes save us typing, make code readable, and allow chaining like above, so we use them *all the time* when manipulating data frames.

Using Pipes

Pipes are clearer to read when you have each function on a separate line (inconsistent in these slides because of space constraints).

```
take_these_data %>%
    do_first_thing(with = this_value) %>%
    do_next_thing(using = that_value) %>% ...
```

Stuff to the left of the pipe is passed to the *first argument* of the function on the right. Other arguments go on the right in the function.

If you ever find yourself piping a function where data are not the first argument, use . in the data argument instead.

```
yugoslavia %>% lm(pop ~ year, data = .)
```

Pipe Assignment

When creating a new object from the output of piped functions, place the assignment operator at the beginning.

```
lm_pop_year <- gapminder %>%
filter(continent == "Americas") %>%
lm(pop ~ year, data = .)
```

W CS&SS

No matter how long the chain of functions is, assignment is always done at the top. $^{\rm 1}$

[1] Note this is just a stylistic convention: If you prefer, you *can* do assignment at the end of the chain.

Filtering Rows (subsetting)

Recall last week we used the filter() command to subset data like so:

```
Canada <- gapminder %>%
    filter(country == "Canada")
head(Canada)
```

##	#	A tibble	e: 6 x 6				
##		country	continent	year	lifeExp	рор	gdpPercap
##		<fct></fct>	<fct></fct>	<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>
##	1	Canada	Americas	1952	68.8	14785584	11367.
##	2	Canada	Americas	1957	70.0	17010154	12490.
##	3	Canada	Americas	1962	71.3	18985849	13462.
##	4	Canada	Americas	1967	72.1	20819767	16077.
##	5	Canada	Americas	1972	72.9	22284500	18971.
##	6	Canada	Americas	1977	74.2	23796400	22091.

Excel analogue: Filter!

Another Operator: %in%

Common use case: Filter rows to things in some set.

We can use %in% like == but for matching *any element* in the vector on its right¹.

##	#	A tibble:	2 x 6				
##		country	continent	year	lifeExp	рор	gdpPercap
##		<fct></fct>	<fct></fct>	<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>
##	1	Slovenia	Europe	2002	76.7	2011497	20660.
##	2	Slovenia	Europe	2007	77.9	2009245	25768.

W CS&SS

[1] The c() function is how we make **vectors** in R, which are an important data type.

distinct()

You can see all the *unique values* in your data for combinations of columns using distinct():

gapminder %>% distinct(continent, year)

##	# /	۹ ti	ibble:	60) х	2	
##		cor	ntiner	nt	yea	ar	
##		<fo< td=""><td>ct></td><td><</td><td><in<sup>-</in<sup></td><td>t></td><td></td></fo<>	ct>	<	<in<sup>-</in<sup>	t>	
##	1	Asi	ia		19	52	
##	2	Asi	ia		19	57	
##	3	Asi	ia		19	62	
##	4	Asi	ia		19	67	
##	5	Asi	ia		19	72	
##	6	Asi	ia		19	77	
##	7	Asi	ia		198	82	
##	8	Asi	ia		198	87	
##	9	Asi	ia		199	92	
##	10	Asi	ia		199	97	
##	#		with	50	mo	re	rows

W CS&SS

distinct() drops unused variables!

Note that the default behavior of distinct() is to drop all unspecified columns. If you want to get distinct rows by certain variables without dropping the others, use distinct(.keep_all=TRUE):

gapminder %>% distinct(continent, year, .keep_all=TRUE)

##	# A tibble: 60	x 6				
##	country	continent	year	lifeExp	рор	gdpPercap
##	<fct></fct>	<fct></fct>	<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>
##	1 Afghanistan	Asia	1952	28.8	8425333	779.
##	2 Afghanistan	Asia	1957	30.3	9240934	821.
##	3 Afghanistan	Asia	1962	32.0	10267083	853.
##	4 Afghanistan	Asia	1967	34.0	11537966	836.
##	5 Afghanistan	Asia	1972	36.1	13079460	740.
##	6 Afghanistan	Asia	1977	38.4	14880372	786.
##	7 Afghanistan	Asia	1982	39.9	12881816	978.
##	8 Afghanistan	Asia	1987	40.8	13867957	852.
##	9 Afghanistan	Asia	1992	41.7	16317921	649.
##	10 Afghanistan	Asia	1997	41.8	22227415	635.
##	# with 50 m	nore rows				

Sampling Rows: sample_n()

We can also filter *at random* to work with a smaller dataset using sample_n() or sample_frac().

set.seed(413) # makes random numbers repeatable
yugoslavia %>% sample_n(size = 6, replace = FALSE)

W CS&SS

##	#	A tibble: 6 x 6					
##		country	continent	year	lifeExp	рор	gdpPercap
##		<fct></fct>	<fct></fct>	<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>
##	1	Bosnia and Herzegovina	Europe	1987	71.1	4338977	4314.
##	2	Bosnia and Herzegovina	Europe	1967	64.8	3585000	2172.
##	3	Montenegro	Europe	2002	74.0	720230	6557.
##	4	Montenegro	Europe	1987	74.9	569473	11733.
##	5	Slovenia	Europe	1952	65.6	1489518	4215.
##	6	Serbia	Europe	1982	70.2	9032824	15181.

Use set.seed() to make all random numbers in a file come up exactly the same each time it is run. Read Details in ?set.seed if you like your brain to hurt.

Sorting: arrange()

Along with filtering the data to see certain rows, we might want to sort it:

yugoslavia %>% arrange(year, desc(pop))

##	# /	A tibble: 60	Эхб					
##		country		continent	year	lifeExp	рор	gdpPercap
##		<fct></fct>		<fct></fct>	<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>
##	1	Serbia		Europe	1952	58.0	6860147	3581.
##	2	Croatia		Europe	1952	61.2	3882229	3119.
##	3	Bosnia and	Herzegovina	Europe	1952	53.8	2791000	974.
##	4	Slovenia		Europe	1952	65.6	1489518	4215.
##	5	Montenegro		Europe	1952	59.2	413834	2648.
##	6	Serbia		Europe	1957	61.7	7271135	4981.
##	7	Croatia		Europe	1957	64.8	3991242	4338.
##	8	Bosnia and	Herzegovina	Europe	1957	58.4	3076000	1354.
##	9	Slovenia		Europe	1957	67.8	1533070	5862.
##	10	Montenegro		Europe	1957	61.4	442829	3682.
##	#	with 50	more rows					

The data are sorted by ascending year and descending pop.

UW CS&SS

Keeping Columns: select()

Not only can we limit rows, but we can include specific columns (and put them in the order listed) using select().

yugoslavia %>% select(country, year, pop) %>% head(4)

##	#	A tibbl	tibble: 4 x 3					
##		country	/		year	рор		
##		<fct></fct>			<int></int>	<int></int>		
##	1	Bosnia	and	Herzegovina	1952	2791000		
##	2	Bosnia	and	Herzegovina	1957	3076000		
##	3	Bosnia	and	Herzegovina	1962	3349000		
##	4	Bosnia	and	Herzegovina	1967	3585000		

Dropping Columns: select()

We can instead drop only specific columns with select() using - signs:

yugoslavia %>% select(-continent, -pop, -lifeExp) %>% head(4)

##	#	A tibbl	le: 4	́+х3		
##		country	/		year	gdpPercap
##		<fct></fct>			<int></int>	<dbl></dbl>
##	1	Bosnia	and	Herzegovina	1952	974.
##	2	Bosnia	and	Herzegovina	1957	1354.
##	3	Bosnia	and	Herzegovina	1962	1710.
##	4	Bosnia	and	Herzegovina	1967	2172.

W CS&SS

Helper Functions for select()

select() has a variety of helper functions like starts_with(), ends_with(), and contains(), or can be given a range of contiguous columns startvar:endvar. See ?select for details.

These are very useful if you have a "wide" data frame with column names following a pattern or ordering.

#	A tibble:	6 × 292									
	married10	married11	married12	married13	married14	married15	married16	married17	married18	married19	married20
	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>	<db1></db1>	<db1></db1>	<db1></db1>	<db1></db1>	<db1></db1>	<db1></db1>	<db1></db1>
1	NA	NA	0	0	0	0	0	0	0	0	0
2	NA	NA	NA	NA	0	0	0	0	1	1	NA
3	NA	NA	0	NA	0	0	0	0	0	0	NA
4	NA	NA	NA	NA	0	0	0	0	0	0	0
5	NA	NA	0	0	0	0	0	0	0	0	0
6	NA	NA	0	0	0	0	0	0	0	0	0
#	with 2	281 more va	ariables: n	arried21 <	<dbl>, marı</dbl>	ried22 <db< td=""><td>l>, married</td><td>d23 <db1>,</db1></td><td>married24</td><td><db1>,</db1></td><td></td></db<>	l>, married	d23 <db1>,</db1>	married24	<db1>,</db1>	
#	married2	25 <dbl>, r</dbl>	married26 <	dbl>, in_s	schooll0 <	dbl>, in_so	choolll <db< td=""><td>ol>, in_sch</td><td>nooll2 <db< td=""><td>l>, in_sch</td><td><pre>>> coll3 <dbl>,</dbl></pre></td></db<></td></db<>	ol>, in_sch	nooll2 <db< td=""><td>l>, in_sch</td><td><pre>>> coll3 <dbl>,</dbl></pre></td></db<>	l>, in_sch	<pre>>> coll3 <dbl>,</dbl></pre>
#	in_schoo	oll4 <dbl></dbl>	, in_school	15 <dbl>,</dbl>	in_school:	16 <dbl>,</dbl>	in_school17	⁷ <dbl>, in</dbl>	n_school18	<db1>,</db1>	
#	in_schoo	oll9 <dbl></dbl>	, in_school	20 <dbl>,</dbl>	in_school2	21 <dbl>, [·]</dbl>	in_school22	2 <dbl>, in</dbl>	n_school23	<db1>,</db1>	

DYS %>% select(starts_with("married"))
DYS %>% select(ends_with("18"))

CS&SS

select(where())

An especially useful helper for select is where() which can be used for selecting columns based on functions that check column types.

gapminder %>% select(where(is.numeric)) %>% head(3)

##	#	A tib	ole: 3 x	4	
##		year	lifeExp	рор	gdpPercap
##		<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>
##	1	1952	28.8	8425333	779.
##	2	1957	30.3	9240934	821.
##	3	1962	32.0	10267083	853.

gapminder %>% select(where(is.factor)) %>% head(3)

- ## # A tibble: 3 x 2
- ## country continent
- ## <fct> <fct>
- ## 1 Afghanistan Asia
- ## 2 Afghanistan Asia
- ## 3 Afghanistan Asia

W CS&SS

int (integer) and dbl (double) are both
types of numeric data.

Renaming Columns with select()

We can rename columns using select(), but that drops everything that isn't mentioned:

yugoslavia %>%	
<pre>select(Life_Expectancy = lifeExp) %></pre>	%
head(4)	

##	#	A tibble: 4 x 1
##		Life_Expectancy
##		<dbl></dbl>
##	1	53.8
##	2	58.4
##	3	61.9
##	4	64.8

W CS&SS



Safer: Rename Columns with rename()

rename() renames variables using the same syntax as select() without dropping unmentioned variables.

```
yugoslavia %>%
select(country, year, lifeExp) %>%
rename(Life_Expectancy = lifeExp) %>%
head(4)
```

```
## # A tibble: 4 x 3
```

W CS&SS

##		country	/		year	Life_Expectancy
##		<fct></fct>			<int></int>	<dbl></dbl>
##	1	Bosnia	and	Herzegovina	1952	53.8
##	2	Bosnia	and	Herzegovina	1957	58.4
##	3	Bosnia	and	Herzegovina	1962	61.9
##	4	Bosnia	and	Herzegovina	1967	64.8

Column Naming Practices

- Good column names will be self-describing. Don't use inscrutable abbreviations to save typing. RStudio's autocompleting functions take away the pain of long variable names: Hit TAB while writing code to autocomplete.
- *Valid* "naked" column names can contain upper or lowercase letters, numbers, periods, and underscores. They must start with a letter or period and not be a special reserved word (e.g. TRUE, if).
- Names are case-sensitive: Year and year are not the same thing!
- You can include spaces or use reserved words if you put backticks around the name. Spaces can be worth including when preparing data for ggplot2 or pander since you don't have to rename axes or table headings.

Column Name with Space Example

```
library(pander)
yugoslavia %>% filter(country == "Serbia") %>%
select(year, lifeExp) %>%
rename(Year = year, `Life Expectancy` = lifeExp) %>%
head(5) %>%
pander(style = "rmarkdown", caption = "Serbian life expectancy")
```

Year	Life Expectancy
1952	58
1957	61.69
1962	64.53
1967	66.91
1972	68.7

Table: Serbian life expectancy

W CS&SS

Create New Columns: mutate()

In dplyr, you can add new columns to a data frame using mutate().

##	#	A tib	ole: 5 x	5		
##		year	рор	lifeExp	pop_million	life_exp_past_40
##		<int></int>	<int></int>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>
##	1	1952	6860147	58.0	6.86	18.0
##	2	1957	7271135	61.7	7.27	21.7
##	3	1962	7616060	64.5	7.62	24.5
##	4	1967	7971222	66.9	7.97	26.9
##	5	1972	8313288	68.7	8.31	28.7

W CS&SS

Note you can create multiple variables in a single mutate() call by separating the expressions with commas.

ifelse()

A common function used in mutate() (and in general in R programming) is ifelse(). It returns a vector of values depending on a logical test.

ifelse(test = x==y, yes = first_value , no = second_value)

```
Output from ifelse() if x==y is...
```

- TRUE: first_value the value for yes =
- FALSE: second_value the value for no =
- NA: NA because you can't test for NA with an equality!

For example:

W CS&SS

example <- c(1, 0, NA, -2)
ifelse(example > 0, "Positive", "Not Positive")

[1] "Positive" "Not Positive" NA

"Not Positive"

22 / 50

ifelse() Example

```
## # A tibble: 3 x 4
```

##		country	<pre>short_country</pre>	year	рор
##		<fct></fct>	<chr></chr>	<int></int>	<int></int>
##	1	Bosnia and Herzegovina	B and H	1952	2791000
##	2	Croatia	Croatia	1952	3882229
##	3	Montenegro	Montenegro	1952	413834

Read this as "For each row, if country equals 'Bosnia and Herzegovina, make short_country equal to 'B and H', otherwise make it equal to that row's value of country."

This is a simple way to change some values but not others!

Note: country is a factor--use as.character() to convert to character.



recode()

recode() is another useful function to use inside mutate(). Use recode()
to change specific values to other values, particularly with factors. You can
change multiple values at the same time. Note if a value has spaces in it, you'll
need to put it in backticks!

yugoslav mutate	via %>% e(country = recode(country,
	`Bosnia and Herzegovina`="B and H",
	Montenegro="M")) %>%
distir	<pre>nct(country)</pre>
## # A t: ## cou	ibble: 5 x 1 ntry

- ## 1 B and H
- ## 2 Croatia
- ## 3 M
- ## 4 Serbia
- ## 5 Slovenia

case_when()

case_when() performs multiple ifelse() operations at the same time. case_when() allows you to create a new variable with values based on multiple logical statements. This is useful for making categorical variables or variables from combinations of other variables.

```
gapminder %>%
mutate(gdpPercap_ordinal =
    case_when(
        gdpPercap < 700 ~ "low",
        gdpPercap >= 700 & gdpPercap < 800 ~ "moderate",
        TRUE ~ "high" )) %>% # Value when all other statements are FALSE
    slice(6:9) # get rows 6 through 9
```

##	#	A tibble: 4	x 7					
##		country	continent	year	lifeExp	рор	gdpPercap	gdpPercap_ordinal
##		<fct></fct>	<fct></fct>	<int></int>	<dbl></dbl>	<int></int>	<dbl></dbl>	<chr></chr>
##	1	Afghanistan	Asia	1977	38.4	14880372	786.	moderate
##	2	Afghanistan	Asia	1982	39.9	12881816	978.	high
##	3	Afghanistan	Asia	1987	40.8	13867957	852.	high
##	4	Afghanistan	Asia	1992	41.7	16317921	649.	low

pull()

Sometimes you want to extract a single column from a data frame as a *vector* (or single value).

pull() *pulls* a column of a data frame out as a vector.

gapminder %>% pull(lifeExp) %>% head(4)

[1] 28.801 30.332 31.997 34.020

4 x 1

gapminder %>% select(lifeExp) %>% head(4)

##	#	A tibble:
##		lifeExp
##		<dbl></dbl>
##	1	28.8
##	2	30.3
##	3	32.0
##	4	34.0

S&SS

Note the difference between these two operations: The second yields only one column but is still a data frame.

In-Line pull()

pull() is particularly useful when you want to use a vector-only command in a dplyr chain of functions (say, in an in-line expression).

This in-line code...

```
The average life expectancy in Afghanistan from 1952 to 2007
was `r gapminder %>% filter(country=="Afghanistan") %>%
pull(lifeExp) %>% mean() %>% round(1)` years.
```

... will produce this output:

The average life expectancy in Afghanistan from 1952 to 2007 was 37.5 years.

mean() can only take a vector input, not a dataframe, so this won't work with
select(lifeExp) instead of pull(lifeExp).



Summarizing with dplyr

–UW CS&SS——

General Aggregation: summarize()

summarize()
takes your column(s) of data and computes something using
every row:

- Count how many rows there are
- Calculate the mean
- Compute the sum
- Obtain a minimum or maximum value

You can use any function in summarize() that aggregates *multiple values* into a *single value* (like sd(), mean(), or max()).

summarize() Example

For the year 1982, let's get the number of observations, total population, mean life expectancy, and range of life expectancy for former Yugoslavian countries.

yugoslavia %>%		
filter(year == 1982) %>%		
summarize(n_obs	=	n(),
total_pop	=	sum(pop),
<pre>mean_life_exp</pre>	=	mean(lifeExp),
range_life_exp	=	<pre>max(lifeExp) - min(lifeExp))</pre>

A tibble: 1 x 4
n_obs total_pop mean_life_exp range_life_exp
<int> <int> <dbl> <dbl>
1 5 20042685 71.3 3.94

These new variables are calculated using *all of the rows* in yugoslavia

Avoiding Repetition

summarize(across())

Maybe you need to calculate the mean and standard deviation of a bunch of columns. With **across()**, put the variables to compute over first (using c() or select() syntax) and put the functions to use in a list() after.

```
yugoslavia %>%
filter(year == 1982) %>%
summarize(across(c(lifeExp, pop), list(avg = ~mean(.), sd = ~sd(.))))
## # A tibble: 1 x 4
```

Note it automatically names the summarized variables based on the names given in list().



Whoa, too many (and)

It can get hard to read code with lots of **nested** functions--functions inside others.

Break things up when it gets confusing!

```
yugoslavia %>%
filter(year == 1982) %>%
summarize(
    across(
        c(lifeExp, pop),
        list(
            avg = ~mean(.),
            sd = ~sd(.)
        )
        )
     )
```

RStudio also helps you by tracking parentheses: Put your cursor after a) and see!



Avoiding Repetition

There are additional ways to use across() for repetitive operations:

 across(everything()) will summarize / mutate *all* variables sent to it in the same way. For instance, getting the mean and standard deviation of an entire dataframe:

dataframe %>%
 summarize(across(everything(), list(mean = ~mean(.), sd = ~sd(.))))

 across(where()) will summarize / mutate all variables that satisfy some logical condition. For instance, summarizing every numeric column in a dataframe at once:

```
dataframe %>%
  summarize(across(where(is.numeric), list(mean = ~mean(.), sd = ~sd(.))))
```

You can use all of these to avoid typing out the same code repeatedly!



group_by()

The special function group_by() changes how functions operate on the data, most importantly summarize().

Functions after group_by() are computed *within each group* as defined by variables given, rather than over all rows at once. Typically the variables you group by will be integers, factors, or characters, and not continuous real values.

Excel analogue: pivot tables

	А	В	С	D	E	F	G	н	- I	J
1	Category	(AII) 🔻								
2										
3	Sum of Amount	Column 💌								
4	Row Labels 🛛 💌	Apple	Banana	Beans	Broccoli	Carrots	Mango	Orange	Grand Total	
5	Australia	20634	52721	14433	17953	8106	9186	8680	131713	
6	Canada	24867	33775		12407		3767	19929	94745	
7	France	80193	36094	680	5341	9104	7388	2256	141056	
8	Germany	9082	39686	29905	37197	21636	8775	8887	155168	
9	New Zealand	10332	40050		4390			12010	66782	
10	United Kingdom	17534	42908	5100	38436	41815	5600	21744	173137	
11	United States	28615	95061	7163	26715	56284	22363	30932	267133	
12	Grand Total	191257	340295	57281	142439	136945	57079	104438	1029734	
13										

group_by() example

yugoslavia %>%	
group_by(year) %>%	
summarize(num_countries total_pop total_gdp_per_cap	<pre>= n_distinct(country), = sum(pop), = sum(pop*gdpPercap)/total_pop) %>%</pre>
head(5)	

##	#	A tib	ole: 5 x 4		
##		year	num_countries	total_pop	<pre>total_gdp_per_cap</pre>
##		<int></int>	<int></int>	<int></int>	<dbl></dbl>
##	1	1952	5	15436728	3030.
##	2	1957	5	16314276	4187.
##	3	1962	5	17099107	5257.
##	4	1967	5	17878535	6656.
##	5	1972	5	18579786	8730.

Because we did group_by() with year then used summarize(), we get one row per value of year!

Each value of year is its own group!

UW CS&SS

Window Functions

IS&SS

Grouping can also be used with mutate() or filter() to give rank orders within a group, lagged values, and cumulative sums. You can read more about window functions in this <u>vignette</u>.

##	#	A tibble: 4 x 5				
##	#	Groups: country [2]				
##		country	year	рор	lag_pop	pop_chg
##		<fct></fct>	<int></int>	<int></int>	<int></int>	<int></int>
##	1	Bosnia and Herzegovina	2002	4165416	NA	NA
##	2	Bosnia and Herzegovina	2007	4552198	4165416	386782
##	3	Croatia	2002	4481020	NA	NA
##	4	Croatia	2007	4493312	4481020	12292

Joining (Merging) Data Frames

-UW CS&SS

When Do We Need to Join Tables?

- Want to make columns using criteria too complicated for ifelse() or case_when()
 - We can work with small sets of variables then combine them back together.
- Combine data stored in separate data sets: e.g. UW registrar data with police stop records.

W CS&SS

 Often large surveys are broken into different data sets for each level (e.g. household, individual, neighborhood)

Joining in Concept

W CS&SS

We need to think about the following when we want to merge data frames A and B:

- Which rows are we keeping from each data frame?
- Which *columns* are we keeping from each data frame?
- Which variables determine whether rows *match*?

Join Types: Rows and columns kept

There are many types of joins¹...

W CS&SS

- A %>% left_join(B): keep all rows from A, matched with B wherever possible (NA when not), keep columns from both A and B
- A %>% right_join(B): keep all rows from B, matched with A wherever possible (NA when not), keep columns from both A and B
- A %>% inner_join(B): keep only rows from A and B that match, keep columns from both A and B
- A %>% full_join(B): keep all rows from both A and B, matched wherever possible (NA when not), keep columns from both A and B
- A %>% semi_join(B): keep rows from A that match rows in B, keep columns from only A
- A %>% anti_join(B): keep rows from A that don't match a row in B, keep columns from only A

[1] Usually left_join() does the job.

Matching Criteria

We say rows should *match* because they have some columns containing the same value. We list these in a by = argument to the join.

Matching Behavior:

- No by: Match using all variables in A and B that have identical names
- by = c("var1", "var2", "var3"): Match on identical values of var1, var2, and var3 in both A and B
- by = c("Avar1" = "Bvar1", "Avar2" = "Bvar2"): Match identical values of Avar1 variable in A to Bvar1 variable in B, and Avar2 variable in A to Bvar2 variable in B

Note: If there are multiple matches, you'll get *one row for each possible combination* (except with semi_join() and anti_join()).

Need to get more complicated? Break it into multiple operations.



nycflights13 Data

We'll use data in the <u>nycflights13 package</u>. Install and load it:

install.packages("nycflights13") # Uncomment to run library(nycflights13)

It includes five dataframes, some of which contain missing data (NA):

- flights: flights leaving JFK, LGA, or EWR in 2013
- airlines: airline abbreviations
- airports: airport metadata
- planes: airplane metadata
- weather: hourly weather data for JFK, LGA, and EWR

Note these are *separate data frames*, each needing to be *loaded separately*:

data(flights)
data(airlines)
data(airports)
and so on...



Join Example #1

Who manufactures the planes that flew to Seattle?

flights %>% filter(dest == "SEA") %>% select(tailnum) %>%
left_join(planes %>% select(tailnum, manufacturer),
by = "tailnum") %>%
<pre>count(manufacturer) %>% # Count observations by manufacturer</pre>
<pre>arrange(desc(n)) # Arrange data descending by count</pre>

##	#	A tibble: 6 x 2	
##		manufacturer	n
##		<chr></chr>	<int></int>
##	1	BOEING	2659
##	2	AIRBUS	475
##	3	AIRBUS INDUSTRIE	394
##	4	<na></na>	391
##	5	BARKER JACK L	2
##	6	CIRRUS DESIGN CORP	2

UW CS&SS

Note you can perform operations on the data inside functions such as left_join() and the *output* will be used by the function.

Join Example #2

Which airlines had the most flights to Seattle from NYC?

```
flights %>% filter(dest == "SEA") %>%
    select(carrier) %>%
    left_join(airlines, by = "carrier") %>%
    group_by(name) %>%
    tally() %>%
    arrange(desc(n))
```

W CS&SS

tally() is a shortcut for summarize(n(.)): It creates a variable n equal to the number of rows in each group.

Join Example #3

Is there a relationship between departure delays and wind gusts?

```
library(ggplot2)
flights %>%
  select(origin, year, month, day, hour, dep_delay) %>%
  inner_join(weather,
            by = c("origin", "year", "month", "day", "hour")) %>%
  select(dep_delay, wind_gust) %>%
  # removing rows with missing values
  filter(!is.na(dep_delay) & !is.na(wind_gust)) %>%
  ggplot(aes(x = wind_gust, y = dep_delay)) +
     geom_point() +
     geom_smooth()
```

Because the data are the first argument for ggplot(), we can pipe them straight into a plot.



Wind Gusts and Delays



Check out those 1200 mph winds!¹

·UW CS&SS

[1] These observations appear to have been fixed in the current data.

Redo After Removing Extreme Outliers, Just Trend

flights %>%
 select(origin, year, month, day, hour, dep_delay) %>%
 inner_join(weather, by = c("origin", "year", "month", "day", "hour")) %>%
 select(dep_delay, wind_gust) %>%
 filter(!is.na(dep_delay) & !is.na(wind_gust) & wind_gust < 250) %>%
 ggplot(aes(x = wind_gust, y = dep_delay)) +
 geom_smooth() +
 theme_bw(base_size = 16) +
 xlab("Wind gusts in departure hour (mph)") +
 ylab("Average departure delay (minutes)")

I removed geom_point() to focus on the mean trend produced by
geom_smooth().

Wind Gusts and Delays: Mean Trend



-UW CS&SS

Tinkering Suggestions

Some possible questions to investigate:

- What are the names of the most common destination airports?
- Which airlines fly from NYC to your home city?
- Is there a relationship between departure delays and precipitation?
- What is the distribution of departure times for flights leaving NYC over a 24 hour period?
 - Are especially late or early arrivals departures to some regions or for some airlines?

Warning: flights has 336776 rows, so if you do a sloppy join, you can end up with many matches per observation and have the data *explode* in size.

Homework 3

Pick something to look at in the nycflights13 data and write up a .Rmd file showing your investigation. Upload both the .Rmd file and the .html file to Canvas. You must use at least once: mutate(), summarize(), group_by(), and any join. *Include at least one nicely formatted plot (ggplot2) and one table (pander)*. In plots and tables, use "nice" variable names (try out spaces!) and rounded values (<= 3 digits).

This time, *include all your code in your output document* (echo=TRUE), using comments and line breaks separating commands so that it is clear to a peer what you are doing (or trying to do!). You must write up your observations briefly in words as well.

Note: If you want to see the nycflights13 dataframes in the environment, you will need to load *each one*: airlines, airports, flights, planes, and weather (e.g. data(flights)).

