

Data Transformation I

PSY 410: Data Science for Psychology

Dr. Sara Weston

2026-04-06

Data transformation

A question you can't answer yet

You collected survey data from 500 participants. You only want women over 25 who passed the attention check.

...

How do you get to *just those rows*?

...

That's what `filter()` does. And it's just the beginning — today we learn four verbs that turn raw data into exactly what you need.

Pause after revealing the question. Let students sit with the discomfort of not knowing how to do this yet — that tension is the hook. When you reveal `filter()`, frame it as “this is the tool that solves that exact problem.” Spend ~2 minutes here.

The dplyr verbs

Verb	What it does
<code>filter()</code>	Pick rows by their values
<code>arrange()</code>	Reorder rows
<code>select()</code>	Pick columns by name
<code>mutate()</code>	Create new columns
<code>summarize()</code>	Collapse to a summary

Today: `filter()`, `arrange()`, `select()`, `mutate()`

Emphasize the word “verbs” — dplyr is designed so each function is an action word that describes what it does. Mention that `summarize()` will come next session. Students don't need to memorize this table; it's a roadmap so they know where the hour is going.

Our dataset: flights

```
# All 336,776 flights departing NYC in 2013
glimpse(flights)
```

Rows: 336,776

Columns: 19

```
$ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
$ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
$ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -
1~
$ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
$ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -
1~
$ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
$ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
$ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
$ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
$ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
$ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
$ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
$ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
$ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
$ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-
01 0~
```

Understanding the data

- `year`, `month`, `day` — departure date
- `dep_time`, `arr_time` — actual times (HHMM format)

- `dep_delay`, `arr_delay` — delays in minutes (negative = early)
- `carrier` — airline code
- `origin`, `dest` — airport codes
- `air_time`, `distance` — in minutes and miles

Spend ~3 minutes on these two slides. Have students look at the `glimpse()` output and call out what they notice — number of rows, column types, etc. Point out that negative delay means the flight left early. Students often ask about the HHMM time format (e.g., 517 means 5:17 AM) — acknowledge it’s awkward and we’ll mostly use the delay columns instead.

filter()

filter() picks rows

```
# All flights on January 1st
filter(flights, month == 1, day == 1)
```

```
# A tibble: 842 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517           515           2     830           819
2  2013     1     1     533           529           4     850           830
3  2013     1     1     542           540           2     923           850
4  2013     1     1     544           545          -1    1004          1022
5  2013     1     1     554           600          -6     812           837
6  2013     1     1     554           558          -4     740           728
7  2013     1     1     555           600          -5     913           854
8  2013     1     1     557           600          -3     709           723
9  2013     1     1     557           600          -3     838           846
10 2013     1     1     558           600          -2     753           745
# i 832 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

Type this live in the console. Emphasize the structure: `filter(data, condition)`. Point out that the output is a tibble with fewer rows — `filter()` didn’t change the original data, it returned a new subset. Students often worry about “losing” data; reassure them that the original `flights` object is untouched.

Comparison operators

Operator	Meaning
==	equal to
!=	not equal to
<, >	less than, greater than
<=, >=	less/greater than or equal

Warning

Use == for comparison, not =!

The == vs = mistake is the single most common error in this session. Demonstrate what happens when you use = instead — R will throw an error or silently do the wrong thing. Have students say it aloud: “double equals for comparison.” You’ll repeat this warning several times today and it’s worth it.

Filter examples

```
# Flights to Los Angeles
filter(flights, dest == "LAX")

# Flights with arrival delay over 2 hours
filter(flights, arr_delay > 120)

# Flights by United Airlines
filter(flights, carrier == "UA")
```

Multiple conditions

Conditions separated by , are combined with AND:

```
# January 1st flights (both must be true)
filter(flights, month == 1, day == 1)
```

Equivalent to:

```
filter(flights, month == 1 & day == 1)
```

Logical operators

Operator	Meaning
&	AND (both true)
	OR (either true)
!	NOT (negation)

Using OR

```
# Flights in November OR December
filter(flights, month == 11 | month == 12)
```

```
# A tibble: 55,403 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013    11     1     5           2359             6     352           345
2  2013    11     1    35           2250            105     123           2356
3  2013    11     1   455            500             -5     641           651
4  2013    11     1   539            545             -6     856           827
5  2013    11     1   542            545             -3     831           855
6  2013    11     1   549            600            -11     912           923
7  2013    11     1   550            600            -10     705           659
8  2013    11     1   554            600             -6     659           701
9  2013    11     1   554            600             -6     826           827
10 2013    11     1   554            600             -6     749           751
# i 55,393 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

Common mistake: students will write `filter(flights, month == 11 | 12)`. This doesn't throw an error but returns the wrong result because R evaluates 12 as TRUE. Show this mistake live so they can see the problem. This sets up `%in%` nicely as a safer alternative.

A useful shortcut: %in%

```
# Same as above, but cleaner
filter(flights, month %in% c(11, 12))
```

```
# A tibble: 55,403 x 19
```

```
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013    11     1     5           2359           6     352           345
2  2013    11     1    35           2250          105     123           2356
3  2013    11     1   455           500           -5     641           651
4  2013    11     1   539           545           -6     856           827
5  2013    11     1   542           545           -3     831           855
6  2013    11     1   549           600          -11     912           923
7  2013    11     1   550           600          -10     705           659
8  2013    11     1   554           600           -6     659           701
9  2013    11     1   554           600           -6     826           827
10 2013    11     1   554           600           -6     749           751
```

```
# i 55,393 more rows
```

```
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

%in% checks if the value is in the vector.

Emphasize that %in% is the safer, cleaner way to check “is this value one of these options?” Students will use this constantly for filtering by condition, demographic groups, etc. It also scales — imagine checking for 5 or 6 values with |.

Missing values: NA

NA means “not available” — a missing value.

```
x <- c(1, 2, NA, 4)
x > 2
```

```
[1] FALSE FALSE  NA  TRUE
```

Any operation with NA returns NA (it's unknown!).

This is a conceptual hurdle. Explain the logic: "Is NA greater than 2? We don't know, because we don't know what NA is. So the answer is also NA." The analogy of a blank cell in a spreadsheet works well. Students coming from SPSS may be used to missing values being coded as -99 or similar — emphasize that R has a built-in concept of missingness. Spend ~3 minutes on NA and `is.na()` together.

Checking for NA

Use `is.na()` to check for missing values:

```
x <- c(1, 2, NA, 4)
is.na(x)
```

```
[1] FALSE FALSE TRUE FALSE
```

```
# Keep rows where dep_delay is NOT missing
filter(flights, !is.na(dep_delay))
```

Pair coding break

Your turn: 10 minutes

With a partner, using the `flights` dataset:

1. Find all **United Airlines** ("UA") flights
2. that were **more than 2 hours late** arriving
3. and were flying **to Los Angeles** ("LAX")

How many flights match? Which origin airport had the most?

Tip

You'll need `filter()` with multiple conditions. Think about which operators you need.

Set a visible 10-minute timer. Circulate and look for common issues: forgetting quotes around "UA" and "LAX", using = instead of ==, using > instead of > 120 for minutes vs. hours. The answer is 26 flights; EWR has the most. The `bonus_count()` question previews next session's content — it's fine if they don't get to it. When time is up, reveal the solution and walk through it line by line.

Before we move on

[Submit your code on Canvas](#) for participation credit. Paste what you have — it doesn't need to work perfectly.

arrange()

arrange() reorders rows

```
# Sort by departure delay (smallest first)
arrange(flights, dep_delay)
```

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013    12     7     2040           2123         -43     40           2352
2  2013     2     3     2022           2055         -33    2240           2338
3  2013    11    10     1408           1440         -32    1549           1559
4  2013     1    11     1900           1930         -30    2233           2243
5  2013     1    29     1703           1730         -27    1947           1957
6  2013     8     9      729            755         -26    1002            955
7  2013    10    23     1907           1932         -25    2143           2143
8  2013     3    30     2030           2055         -25    2213           2250
9  2013     3     2     1431           1455         -24    1601           1631
10 2013     5     5      934            958         -24    1225           1309
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

After the pair coding break, this is a reset point. Arrange is simpler than filter, so it's a good place to regain momentum. Point out the negative values at the top — those are flights that left early. Ask: “What if we want the most delayed flights first?” to set up desc()).

Descending order

```
# Most delayed flights first
arrange(flights, desc(dep_delay))
```

```
# A tibble: 336,776 x 19
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>
1	2013	1	9	641	900	1301	1242	1530
2	2013	6	15	1432	1935	1137	1607	2120
3	2013	1	10	1121	1635	1126	1239	1810
4	2013	9	20	1139	1845	1014	1457	2210
5	2013	7	22	845	1600	1005	1044	1815
6	2013	4	10	1100	1900	960	1342	2211
7	2013	3	17	2321	810	911	135	1020
8	2013	6	27	959	1900	899	1236	2226
9	2013	7	22	2257	759	898	121	1026
10	2013	12	5	756	1700	896	1058	2020

```
# i 336,766 more rows
```

```
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
# tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
# hour <dbl>, minute <dbl>, time_hour <dtm>
```

Multiple sort columns

```
# Sort by month, then by day, then by departure time
arrange(flights, month, day, dep_time)
```

```
# A tibble: 336,776 x 19
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>
1	2013	1	1	517	515	2	830	819
2	2013	1	1	533	529	4	850	830
3	2013	1	1	542	540	2	923	850
4	2013	1	1	544	545	-1	1004	1022
5	2013	1	1	554	600	-6	812	837
6	2013	1	1	554	558	-4	740	728
7	2013	1	1	555	600	-5	913	854
8	2013	1	1	557	600	-3	709	723

```

 9 2013     1     1     557           600           -3           838           846
10 2013     1     1     558           600           -2           753           745
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>

```

Quick mental model

This is a breathing slide — use it to consolidate before moving on. Pause and ask students to point to which verb they’d use for a given task: “I want only participants in the treatment group” (filter). “I want to see who scored highest” (arrange with desc). Spend ~1 minute.

You’ve now seen three verbs. Each answers a different question:

Verb	Question it answers
<code>filter()</code>	Which rows do I keep?
<code>arrange()</code>	What order should they be in?
<code>select()</code>	Which columns do I need?
<code>mutate()</code>	What new columns should I create?

We’ve done the first two. Now let’s pick our columns.

`select()`

`select()` picks columns

```
# Just these three columns
select(flights, year, month, day)
```

```
# A tibble: 336,776 x 3
  year month  day
  <int> <int> <int>
1  2013     1     1
2  2013     1     1
3  2013     1     1
4  2013     1     1
5  2013     1     1

```

```

6 2013 1 1
7 2013 1 1
8 2013 1 1
9 2013 1 1
10 2013 1 1
# i 336,766 more rows

```

Select a range

```

# All columns from year to day
select(flights, year:day)

```

```

# A tibble: 336,776 x 3
  year month  day
  <int> <int> <int>
1 2013     1     1
2 2013     1     1
3 2013     1     1
4 2013     1     1
5 2013     1     1
6 2013     1     1
7 2013     1     1
8 2013     1     1
9 2013     1     1
10 2013     1     1
# i 336,766 more rows

```

Select helpers

Helper	What it does
<code>starts_with("x")</code>	Columns starting with “x”
<code>ends_with("x")</code>	Columns ending with “x”
<code>contains("x")</code>	Columns containing “x”
<code>everything()</code>	All remaining columns

This is where `select` becomes genuinely useful for psychology data. Connect it to Qualtrics exports: “Imagine you have 200 columns and all your BDI items start with ‘bdi_’. You can grab them all with one line.” Students often ask if these are case-sensitive — yes, by default. You can use `contains("delay", ignore.case = TRUE)` if needed but don’t belabor it.

Using select helpers

```
# All delay-related columns
select(flights, contains("delay"))
```

```
# A tibble: 336,776 x 2
```

```
  dep_delay arr_delay
    <dbl>     <dbl>
1         2         11
2         4         20
3         2         33
4        -1        -18
5        -6        -25
6        -4         12
7        -5         19
8        -3        -14
9        -3         -8
10       -2          8
```

```
# i 336,766 more rows
```

Using select helpers

```
# All time columns
select(flights, ends_with("time"))
```

```
# A tibble: 336,776 x 5
```

```
  dep_time sched_dep_time arr_time sched_arr_time air_time
    <int>         <int>     <int>         <int>         <dbl>
1     517           515         830             819         227
2     533           529         850             830         227
3     542           540         923             850         160
4     544           545        1004            1022         183
5     554           600         812             837         116
6     554           558         740             728         150
7     555           600         913             854         158
8     557           600         709             723          53
9     557           600         838             846         140
10    558           600         753             745         138
```

```
# i 336,766 more rows
```

Reorder columns

```
# Move air_time and distance to the front
select(flights, air_time, distance, everything())
```

```
# A tibble: 336,776 x 19
```

	air_time	distance	year	month	day	dep_time	sched_dep_time	dep_delay
	<dbl>	<dbl>	<int>	<int>	<int>	<int>	<int>	<dbl>
1	227	1400	2013	1	1	517	515	2
2	227	1416	2013	1	1	533	529	4
3	160	1089	2013	1	1	542	540	2
4	183	1576	2013	1	1	544	545	-1
5	116	762	2013	1	1	554	600	-6
6	150	719	2013	1	1	554	558	-4
7	158	1065	2013	1	1	555	600	-5
8	53	229	2013	1	1	557	600	-3
9	140	944	2013	1	1	557	600	-3
10	138	733	2013	1	1	558	600	-2

```
# i 336,766 more rows
```

```
# i 11 more variables: arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

Exclude columns

Use - to remove columns:

```
# Remove year column
select(flights, -year)
```

```
# A tibble: 336,776 x 18
```

	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time
	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>
1	1	1	517	515	2	830	819
2	1	1	533	529	4	850	830
3	1	1	542	540	2	923	850
4	1	1	544	545	-1	1004	1022
5	1	1	554	600	-6	812	837
6	1	1	554	558	-4	740	728
7	1	1	555	600	-5	913	854

```

8      1      1      557          600          -3          709          723
9      1      1      557          600          -3          838          846
10     1      1      558          600          -2          753          745
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>

```

Exclude columns

```

# Remove multiple columns
select(flights, -year, -month, -day)
select(flights, -(year:day)) # same thing

```

The pattern so far

Notice: every verb has the same shape.

```

# verb(data, what_you_want)
filter(flights, month == 1)
arrange(flights, dep_delay)
select(flights, year, month, day)

```

Data goes first. Then you describe what you want. This consistency is by design.

Hammer this point home — the consistent structure (data first, then what you want) is what makes dplyr learnable. Once you know one verb, you already know the shape of the next. This also sets up the pipe: “If data always goes first, what if we could just pass it automatically?”

mutate()

mutate() creates new columns

```

# Calculate total delay
mutate(flights, total_delay = dep_delay + arr_delay)

```

```

# A tibble: 336,776 x 20
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517           515           2     830           819
2  2013     1     1     533           529           4     850           830
3  2013     1     1     542           540           2     923           850
4  2013     1     1     544           545          -1    1004          1022
5  2013     1     1     554           600          -6     812           837
6  2013     1     1     554           558          -4     740           728
7  2013     1     1     555           600          -5     913           854
8  2013     1     1     557           600          -3     709           723
9  2013     1     1     557           600          -3     838           846
10 2013     1     1     558           600          -2     753           745
# i 336,766 more rows
# i 12 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>, total_delay <dbl>

```

Scroll the output to the right to show that the new column appears at the end. Students sometimes think it didn't work because they can't see it. Emphasize the `name = expression` syntax — the left side is the new column name you're choosing, the right side is the calculation. This is different from `==` (comparison) and `=` in filter (a mistake). Spend ~2 minutes.

Multiple new columns

```

mutate(flights,
  # Create new variables
  total_delay = dep_delay + arr_delay,
  speed = distance / air_time * 60, # mph
  # Can reference columns you just created!
  delay_per_mile = total_delay / distance
)

```

```

# A tibble: 336,776 x 22
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517           515           2     830           819
2  2013     1     1     533           529           4     850           830
3  2013     1     1     542           540           2     923           850
4  2013     1     1     544           545          -1    1004          1022

```

```

5 2013 1 1 554 600 -6 812 837
6 2013 1 1 554 558 -4 740 728
7 2013 1 1 555 600 -5 913 854
8 2013 1 1 557 600 -3 709 723
9 2013 1 1 557 600 -3 838 846
10 2013 1 1 558 600 -2 753 745
# i 336,766 more rows
# i 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>, total_delay <dbl>, speed <dbl>,
#   delay_per_mile <dbl>

```

Useful functions in mutate()

Arithmetic: +, -, *, /, ^, %% (modulo), %/% (integer division)

Logs: log(), log2(), log10()

Offsets: lead(), lag() (for time series)

Cumulative: cumsum(), cummean(), cummax()

Ranking: min_rank(), dense_rank(), row_number()

Common transformations

```

# Z-score (standardize)
mutate(data, score_z = (score - mean(score)) / sd(score))

# Log transform (for skewed data)
mutate(data, rt_log = log(reaction_time))

# Create categories from continuous
mutate(data, age_group = case_when(
  age < 30 ~ "young",
  age < 60 ~ "middle",
  TRUE ~ "older"
))

```

Don't dwell on `case_when()` — show it exists but tell students it's a preview. The z-score and log transform examples connect to stats concepts they already know. If anyone asks about `case_when()` syntax, the tilde (~) means “then” — read it as “if age < 30, then ‘young’.” The TRUE line is the “everything else” catch-all.

mutate() for psychology

```
# Calculate scale scores (mean of items)
mutate(survey,
  bdi_total = (bdi_1 + bdi_2 + bdi_3 + bdi_4) / 4,
  # Or rowwise if you have many items:
  anxiety = rowMeans(select(., anx_1:anx_20), na.rm = TRUE)
)

# Create dummy codes
mutate(survey,
  female = if_else(gender == "female", 1, 0),
  treatment = if_else(condition == "treatment", 1, 0)
)

# Reverse code items
mutate(survey,
  item_5r = 8 - item_5 # For 1-7 scale
)
```

This slide is the payoff for psych students — these are tasks they’ll actually do. Reverse coding especially resonates because many have encountered it in methods classes. Walk through the reverse coding formula: for a 1-7 scale, subtracting from 8 flips 1 to 7, 7 to 1, etc. The rowMeans example is advanced — mention it exists but don’t expect them to write it yet.

The pipe makes your code read like a sentence

The problem with nested functions

What if we want to:

1. Filter to January flights
2. Select departure time and delay
3. Arrange by delay

Nested approach:

```
arrange(
  select(
    filter(flights, month == 1),
```

```
    dep_time, dep_delay
  ),
  dep_delay
)
```

This is hard to read!

Let the nested code sit on screen for a moment. Ask students to figure out what it does — they’ll struggle. That’s the point. You read nested code inside-out, which is unnatural. The next two slides show two solutions, then the pipe as the winner. This is a “problem, bad solution, good solution” arc — don’t skip the setup.

Intermediate objects approach

```
# Save each step
jan_flights <- filter(flights, month == 1)
jan_selected <- select(jan_flights, dep_time, dep_delay)
jan_arranged <- arrange(jan_selected, dep_delay)
```

Works, but clutters your environment with objects.

The pipe: |>

The pipe takes the result of one function and passes it as the first argument to the next:

```
# Same result, much cleaner!
flights |>
  filter(month == 1) |>
  select(dep_time, dep_delay) |>
  arrange(dep_delay)
```

```
# A tibble: 27,004 x 2
  dep_time dep_delay
  <int>     <dbl>
1     1900         -30
2     1703         -27
3     1354         -22
4     2137         -22
5       704         -21
```

```
6      2050      -20
7      2134      -20
8      2050      -20
9      2140      -19
10     1947      -18
# i 26,994 more rows
```

Reading piped code

Read `|>` as “**then**”:

```
flights |> # Start with flights, THEN
  filter(month == 1) |> # filter to January, THEN
  select(dep_time, dep_delay) |> # select these columns, THEN
  arrange(dep_delay) # arrange by delay
```

Read this code aloud using “then” for every pipe — have students read it with you. This is the key insight of the pipe: code that reads like English. From this point forward, nearly all code in the course will use the pipe. Emphasize that each line does one thing, and notice how the data argument disappeared from each function because the pipe passes it automatically.

Keyboard shortcut

The pipe is so common, there’s a shortcut:

- **Windows/Linux:** Ctrl + Shift + M
- **Mac:** Cmd + Shift + M

...

Tip

In RStudio settings, make sure “Use native pipe operator” is enabled (Tools → Global Options → Code)

Have everyone try the shortcut right now. If their RStudio inserts `%>%` instead of `|>`, walk them through the settings change. This is worth the 60 seconds it takes — they’ll type the pipe hundreds of times this quarter.

Note: |> vs %>%

You'll see both:

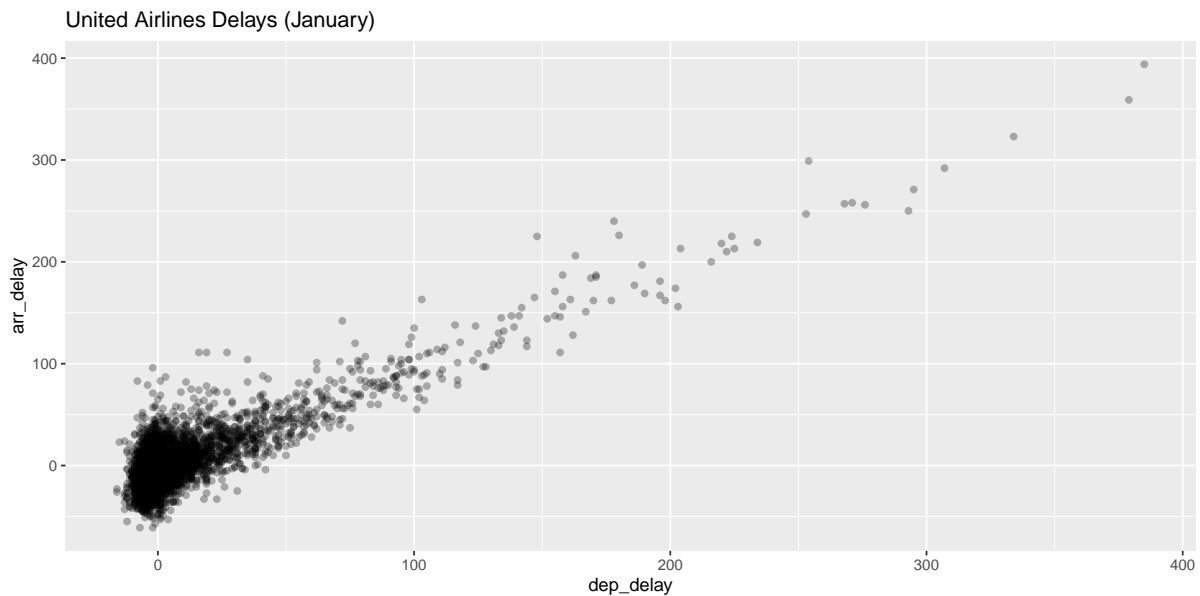
- |> — the **native** pipe (built into R 4.1+)
- %>% — the **magrittr** pipe (older, from tidyverse)

They work almost identically. We'll use |> since it's now standard.

Piping into ggplot

The pipe works beautifully with ggplot:

```
flights |>
  filter(month == 1, carrier == "UA") |>
  ggplot(aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.3) +
  labs(title = "United Airlines Delays (January)")
```



Point out the switch from |> to + once you enter ggplot territory. Students will mix these up — it's the most common pipe-related error. The rule: |> between dplyr steps, + between ggplot layers. This example also previews the power of combining transformation and visualization in one pipeline, which is the core of exploratory data analysis.

Putting it together

A complete workflow

```
# Which airlines have the longest delays in summer?
flights |>
  filter(month %in% c(6, 7, 8)) |>           # Summer months
  filter(!is.na(arr_delay)) |>           # Remove NAs
  mutate(delay_hours = arr_delay / 60) |>  # Convert to hours
  select(carrier, delay_hours) |>        # Keep relevant columns
  arrange(desc(delay_hours))             # Longest delays first
```

```
# A tibble: 84,124 x 2
  carrier delay_hours
  <chr>      <dbl>
1 MQ          18.8
2 MQ          16.5
3 DL          14.9
4 DL          14.2
5 AA          13.4
6 DL           13
7 DL          12.8
8 VX          11.3
9 AA          10.8
10 VX         10.5
# i 84,114 more rows
```

Walk through this pipeline step by step — read each line aloud with “then.” This is the payoff moment where all four verbs and the pipe come together. Point out that each line has a comment explaining what it does — encourage students to comment their pipelines the same way. Ask: “Could you have written this at the start of class?” The answer should feel like genuine progress.

Get a head start

Your turn!

Using the `flights` dataset:

1. Filter to American Airlines (“AA”) flights to Los Angeles (“LAX”)
2. Create a new variable `speed` (`distance / air_time * 60`)
3. Select carrier, origin, dest, and speed
4. Arrange by speed (highest first)
5. What’s the fastest AA flight to LAX?

This is the end-of-deck individual exercise and a head start on Assignment 2. Give students ~10 minutes of work time. This exercise requires combining all four verbs in a single pipeline, which is harder than it looks. Circulate and help with pipe syntax — the most common issue is forgetting `|>` at the end of a line or putting it at the start of the next line. Remind them about the keyboard shortcut.

Bonus: keeping only new columns

`mutate()` adds new columns while keeping existing ones.

Use `.keep = "none"` to *only* keep the new columns:

```
flights |>
  mutate(
    total_delay = dep_delay + arr_delay,
    speed = distance / air_time * 60,
    .keep = "none"
  )
```

```
# A tibble: 336,776 x 2
  total_delay speed
      <dbl> <dbl>
1         13  370.
2         24  374.
3         35  408.
4        -19  517.
5        -31  394.
6          8  288.
7         14  404.
8        -17  259.
9        -11  405.
10         6  319.
# i 336,766 more rows
```

This is a bonus slide — skip it if you’re short on time. It’s useful for students who want to reduce clutter in their output, but it’s not essential for the assignments.

Wrapping up

The dplyr workflow

```
data |>
  filter(<conditions>) |>    # Pick rows
  select(<columns>) |>      # Pick columns
  mutate(<new vars>) |>     # Create columns
  arrange(<order>)          # Sort rows
```

The pipe (`|>`) connects these verbs into a readable workflow.

Before next class

Read:

- [R4DS Ch 3: Data transformation](#) (section 3.5)
- [R4DS Ch 4: Workflow: code style](#)

Practice:

- Transform `flights` in different ways
- Create new variables with `mutate()`
- Build multi-step pipelines

Remind students that Assignment 1 was due today. Preview next session: “Wednesday we’ll learn `group_by()` and `summarize()`, which let you calculate statistics for groups — like mean reaction time by condition. That’s the last piece of the dplyr puzzle.” Section 3.5 in R4DS covers exactly what we’ll do next.

Key takeaways

1. `filter()` picks rows by condition
2. `arrange()` reorders rows (use `desc()` for descending)
3. `select()` picks columns (helpers like `contains()` are useful)
4. `mutate()` creates new columns
5. **The pipe** `|>` makes code readable and elegant

The one thing to remember

The pipe turns a wall of nested code into a sentence you can read aloud.

Next time: `group_by()` and `summarize()`