Statistical Computing with R
Masters in Data Science 503 (S7&8)
Second Batch, SMS, TU, 2023

Shital Bhandary

Associate Professor

Statistics/Bio-statistics, Demography and Public Health Informatics

Patan Academy of Health Sciences, Lalitpur, Nepal

Faculty, Data Analysis and Decision Modeling, MBA, Pokhara University, Nepal

Faculty, FAIMER Fellowship in Health Professions Education, India/USA.

Review Preview (Unit 2, Part 2 & 3)

- Data wrangling
- Data munching
- Tidy data
- dplyr package and its use for data manipulation
- Database and R
- Reading database in R
- dbplyr package and its use for relational database manipulation with dplyr
- Text Mining

Data wrangling (Course book Chapter 9-16)

- Data wrangling is the art of getting your data into R in a useful form for visualization and modelling.

- Data wrangling is very important: without it you can’t work with your own data! There are three main parts to data wrangling:

    - Import
    - Tidy
    - Transform

Import data in R:

- We have already covered this in the previous classes
- More here: https://r4ds.had.co.nz/data-import.html
- Reach this chapter well as there are some important import functions that are part of this course and may not have discussed so far
- We will discuss about reading “database” in the second part of this class

Tidy data in R

- Tidy data is a consistent way to organize your data in R.
- Getting your/our data into this format requires some upfront work, but that work pays off in the long term.
- Once you/we have tidy data and the tidy tools provided by packages in the **tidyverse**, you will spend much less time **munging/cleaning data** from one representation to another, allowing you to spend more time on the analytic questions at hand.
- Tidy data in tidyverse packages are stored as “tibble”

Let us see what is “tibble” first: https://r4ds.had.co.nz/tibbles.html

- The variant of the data frame used by “tidiverse” is called: **tibble**.
- Tibbles are data frames, but they tweak some older behaviours to make life a little easier.
- R is an old language, and some things that were useful 10 or 20 years ago now get in your way.
- It’s difficult to change base R without breaking existing code, so **most innovation occurs in packages**.
- The **tibble** package provides opinionated data frames that make working in the **tidyverse** a little easier.
- It’s particularly **useful for large datasets** because it only prints the first few rows.

Note:

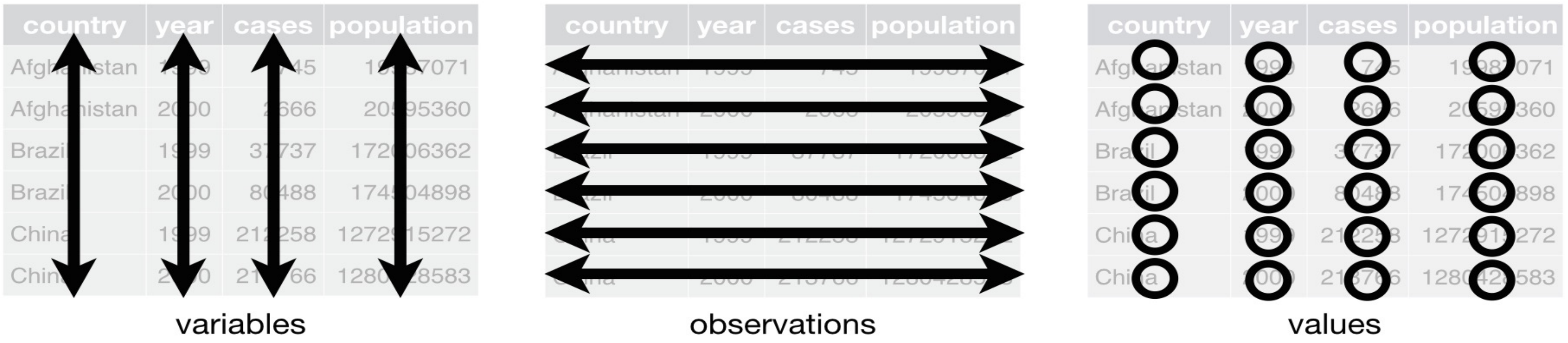
- All the functions of “tidyverse” package works fast with tibble so it will be wise to say that data frame/s should be converted to tibble before using functions of “tidyverse” package

- However, most of the packages of the “tidyverse” super package works well with the data frame too!

- There are two main differences in the usage of a data frame vs a tibble: printing, and subsetting. https://posit.co/blog/tibble-1-0-0/

There are three interrelated rules which make a dataset tidy:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.



Example: Which one is "tidy"? Why?

#Creating tibble:

table1 <- tibble(

country = c("Afghanistan", "Afghanistan", "Brazil", "Brazil", "China", "China"),

year = c(1999, 2000, 1999, 2000, 1999, 2000),

cases = c(745,2666,37737,80488,212258,213766),

population = c(19987071,20595360,172006362, 174504898, 1272915272,1280428583)

)

Data frame to tibble: as_tibble(data_frame)

Tibble to data frame: as.data.frame(tibble_data)

- table1
- A tibble: 6 x 4
- year country cases population
- <dbl> <chr> <dbl> <dbl>
- 1 1999 Afghanistan 745 19987071
- 2 2000 Afghanistan" 2666 20595360
- 3 1999 Brazil 37737 172006362
- 4 2000 Brazil 80488 174504898
- 5 1999 China 212258 1272915272
- 6 2000 China 213766 1280428583

- **dbl = duble instead of number in "tibble"!**

Example: Which one is “tidy”? Why?

- table2
- #> # A tibble: 12 × 4
- #> country year type count
- #> <chr> <int> <chr> <int>
- #> 1 Afghanistan 1999 cases 745
- #> 2 Afghanistan 1999 population 19987071
- #> 3 Afghanistan 2000 cases 2666
- #> 4 Afghanistan 2000 population 20595360
- #> 5 Brazil 1999 cases 37737
- #> 6 Brazil 1999 population 172006362
- #> # … with 6 more rows

- table3
- #> # A tibble: 6 × 3
- #> country year rate
- #> * <chr> <int> <chr>
- #> 1 Afghanistan 1999 745/19987071
- #> 2 Afghanistan 2000 2666/20595360
- #> 3 Brazil 1999 37737/172006362
- #> 4 Brazil 2000 80488/174504898
- #> 5 China 1999 212258/1272915272
- #> 6 China 2000 213766/1280428583

Example: Which one is “tidy”? Why?

Spread across two tibbles

table4a # cases

- #> # A tibble: 3 × 3
- #> country `1999` `2000`
- #> * <chr> <int> <int>
- #> 1 Afghanistan 745 2666
- #> 2 Brazil 37737 80488
- #> 3 China 212258 213766

table4b # population

- #> # A tibble: 3 × 3
- #> country `1999` `2000`
- #> * <chr> <int> <int>
- #> 1 Afghanistan 19987071 20595360
- #> 2 Brazil 172006362 174504898
- #> 3 China 1272915272 1280428583

Why ensure that your data is tidy? There are two main advantages:

- There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

- There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine.

- dplyr, ggplot2, and all the other packages in the **tidyverse** are designed to work with tidy data.

Tidy data: Pivoting – Longer to wider

- table2
- #> # A tibble: 12 × 4
- #>   country   year type   count
- #>   <chr>   <int> <chr>   <int>
- #> 1 Afghanistan 1999 cases   745
- #> 2 Afghanistan 1999 population 19987071
- #> 3 Afghanistan 2000 cases   2666
- #> 4 Afghanistan 2000 population 20595360
- #> 5 Brazil   1999 cases   37737
- #> 6 Brazil   1999 population 172006362
- #> # … with 6 more rows

```
table2 %>%
   pivot_wider(names_from = type, values_from = count)
```

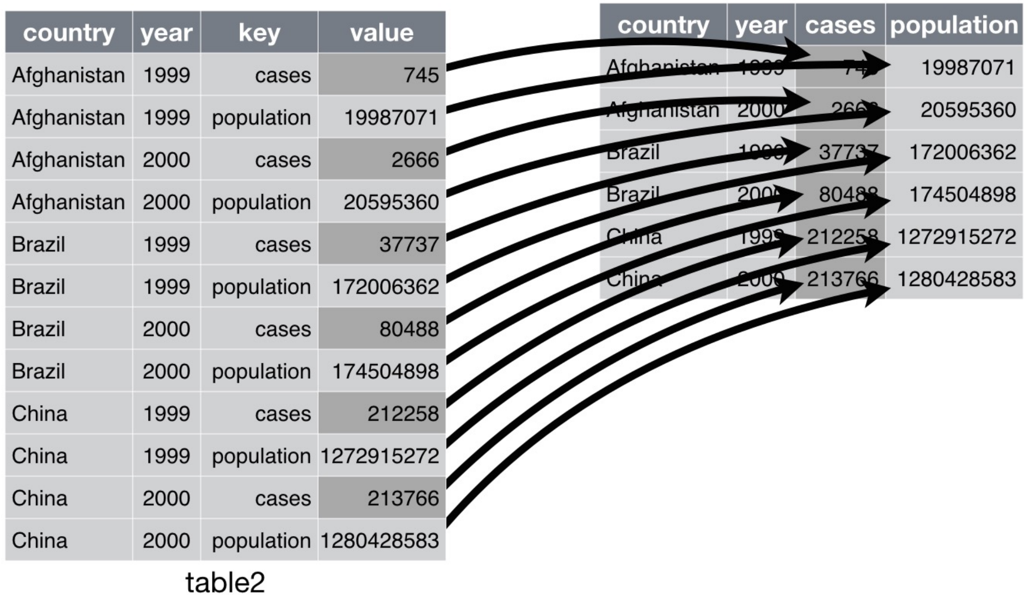


Figure 12.3: Pivoting table2 into a wider, tidy form.

Tidy data: Pivoting – Wider to Longer

- table4a
- #> # A tibble: 3 × 3
- #>   country   `1999` `2000`
- #> * <chr>   <int> <int>
- #> 1 Afghanistan   745   2666
- #> 2 Brazil   37737 80488
- #> 3 China   212258 213766

```
table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to =
"year", values_to = "cases")
```

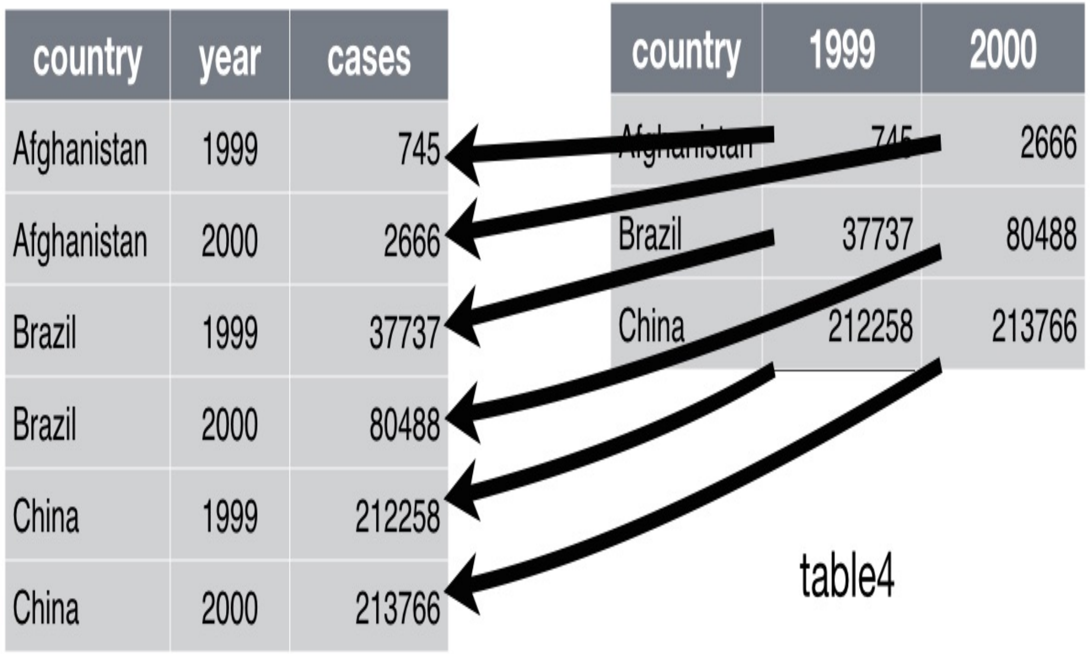


Figure 12.2: Pivoting table4 into a longer, tidy form.

Tidy data: Separate

- table3
- #> # A tibble: 6 × 3
- #>   country   year rate
- #> * <chr>   <int> <chr>
- #> 1 Afghanistan 1999 745/19987071
- #> 2 Afghanistan 2000 2666/20595360
- #> 3 Brazil   1999 37737/172006362
- #> 4 Brazil   2000 80488/174504898
- #> 5 China   1999 212258/1272915272
- #> 6 China   2000 213766/1280428583

```
table3 %>%
  separate(rate, into = c("cases", "population"))
OR
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 12.4: Separating table3 makes it tidy

Tidy data: Unite

- **unite()** is the inverse of **separate()**: it combines multiple columns into a single column.

- You'll need it much less frequently than **separate()**, but it's still a useful tool to have in your back pocket.

```
table5 %>%
  unite(new, century, year)
```

OR

```
table5 %>%
  unite(new, century, year, sep = "")
```

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

Figure 12.5: Uniting table5 makes it tidy

Missing values

- Changing the representation of a dataset brings up an important subtlety of missing values.

- Surprisingly, a value can be missing in one of two possible ways:

- **Explicitly**, i.e. flagged with NA.
- **Implicitly**, i.e. simply not present in the data.

Missing values: Example

#Create a tibble with missing values:

```
stocks <- tibble(
        year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
        qtr    = c(   1,    2,    3,    4,    2,    3,    4),
        return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

Missing values: Example

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is **explicitly missing**, because the cell where its value should be instead contains NA.
- The return for the first quarter of 2016 is **implicitly missing**, because it simply does not appear in the dataset.

Missing values: Example

- stocks %>%
- pivot_wider(names_from = year, values_from = return)
- #> # A tibble: 4 × 3
- #> qtr `2015` `2016`
- #> <dbl> <dbl> <dbl>
- #> 1 1 1.88 NA
- #> 2 2 0.59 0.92
- #> 3 3 0.35 0.17
- #> 4 4 NA 2.66

Missing values: What will happen now?

- stocks %>%

```
pivot_wider(names_from = year, values_from = return) %>%
pivot_longer(
cols = c(`2015`, `2016`),
names_to = "year",
values_to = "return",
values_drop_na = TRUE
)
```

Missing values:
We can use “complete” command!

- stocks %>%
-   complete(year, qtr)
- #> # A tibble: 8 × 3
- #>   year   qtr return
- #>   <dbl> <dbl> <dbl>
- #> 1 2015   1   1.88
- #> 2 2015   2   0.59
- #> 3 2015   3   0.35
- #> 4 2015   4 NA
- #> 5 2016   1 NA
- #> 6 2016   2   0.92
- #> # … with 2 more rows

Missing values: Another example (tibble by row or tribble!)

- treatment <- tribble(

```
        ~ person,           ~ treatment, ~response,
        "Derrick Whitmore", 1,           7,
        NA,                 2,           10,
        NA,                 3,           9,
        "Katherine Burke",  1,           4
)
```

- treatment

Missing values: fill() for another example

- treatment %>%
- fill(person) **# "tidyr" package is required here!**
- #> # A tibble: 4 × 3

#> person	treatment	response
#> <chr>	<dbl>	<dbl>
#> 1 Derrick Whitmore	1	7
#> 2 Derrick Whitmore	2	10
#> 3 Derrick Whitmore	3	9
#> 4 Katherine Burke	1	4

Question/Queries?

Transform/manipulate data with “dplyr”

- To learn five key dplyr functions that allow you to solve the vast majority of your data manipulation challenges:

  - Pick observations by their values **(filter())**.
  - Reorder the rows **(arrange())**.
  - Pick variables by their names **(select())**.
  - Create new variables with functions of existing variables **(mutate())**.
  - Collapse many values down to a single summary **(summarise())**.

- These can all be used in conjunction with **group_by()** which changes the scope of each function from operating on the entire dataset to operating on it group-by-group.

Data manipulation with “dplyr”

- These six functions provide the verbs for a language of data manipulation.

- All verbs work similarly:

  - The first argument is a data frame.
  - The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
  - The result is a new data frame.

- Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

Let’s use them with nycflighst13 data

- library(dplyr)
- library(nycflights13)
- flights
- #> # A tibble: 336,776 × 19
- #> year month day dep_time sched_dep…[1] dep_d…[2] arr_t…[3] sched…[4] arr_d…[5] carrier
- #> <int> <int> <int> <int> <int> <dbl> <int> <int> <dbl> <chr>
- #> 1 2013 1 1 517 515 2 830 819 11 UA
- #> 2 2013 1 1 533 529 4 850 830 20 UA
- #> 3 2013 1 1 542 540 2 923 850 33 AA
- #> 4 2013 1 1 544 545 -1 1004 1022 -18 B6
- #> 5 2013 1 1 554 600 -6 812 837 -25 DL
- #> 6 2013 1 1 554 558 -4 740 728 12 UA
- #> # … with 336,770 more rows, 9 more variables

Filter: What will happen?

- **filter(flights, month == 1, day == 1)**
- #> # A tibble: 842 × 19
- #>   year month   day dep_time sched_dep…[1] dep_d…[2] arr_t…[3] sched…[4] arr_d…[5] carrier
- #>   <int> <int> <int>   <int>   <int>   <dbl>   <int>   <int>   <dbl> <chr>
- #> 1 2013   1   1   517   515   2   830   819   11 UA
- #> 2 2013   1   1   533   529   4   850   830   20 UA
- #> 3 2013   1   1   542   540   2   923   850   33 AA
- #> 4 2013   1   1   544   545   -1   1004   1022   -18 B6
- #> 5 2013   1   1   554   600   -6   812   837   -25 DL
- #> 6 2013   1   1   554   558   -4   740   728   12 UA
- #> # … with 836 more rows, 9 more variables

Are these better?

- jan1 <- filter(flights, month == 1, day == 1)
- (jan1 <- filter(flights, month == 1, day == 1))

- dec25 <- filter(flights, month == 12, day == 25)
- (dec25 <- filter(flights, month == 12, day == 25))

- filter(flights, month = 1)   #Why error?
- filter(flights, month == 1)   #Works now? Why?

More with filter:

- filter(flights, month == 11 | month == 12)   #What?
- filter(flights, month == 11 | 12)   #Works?
- nov_dec <- filter(flights, month %in% c(11, 12))   #Works?

- De Morgan's Law:
- filter(flights, !(arr_delay > 120 | dep_delay > 120))   #Works?
- filter(flights, arr_delay <= 120, dep_delay <= 120)   #Works?

Arrange: Example

- arrange(flights, year, month, day)
- #> # A tibble: 336,776 × 19
- #>   year month   day dep_time sched_dep…[1] dep_d…[2] arr_t…[3] sched…[4] arr_d…[5] carrier
- #>   <int> <int> <int>   <int>   <int>   <dbl>   <int>   <int>   <dbl> <chr>
- #> 1 2013   1   1   517   515   2   830   819   11 UA
- #> 2 2013   1   1   533   529   4   850   830   20 UA
- #> 3 2013   1   1   542   540   2   923   850   33 AA
- #> 4 2013   1   1   544   545   -1   1004   1022   -18 B6
- #> 5 2013   1   1   554   600   -6   812   837   -25 DL
- #> 6 2013   1   1   554   558   -4   740   728   12 UA
- #> # … with 336,770 more rows, 9 more variables

What will happen now?

- Arrange will sort the data in ascending order
- arrange(flights, desc(dep_delay))
- Use desc() to re-order by a column in descending order
- Missing values are always sorted at the end

Select: Example

- # Select columns by name
- select(flights, year, month, day)
- #> # A tibble: 336,776 × 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
- #> # … with 336,770 more rows

- # Select all columns between year and day (inclusive)
- select(flights, year:day)
- #> # A tibble: 336,776 × 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
- #> # … with 336,770 more rows

Select: “except” example

- # Select all columns except those from year to day (inclusive)
- select(flights, -(year:day))
- #> # A tibble: 336,776 × 16
- #>   dep_time sched…[1] dep_d…[2] arr_t…[3] sched…[4] arr_d…[5] carrier flight tailnum origin
- #>   <int>   <int>   <dbl>   <int>   <int>   <dbl> <chr>   <int> <chr>   <chr>
- #> 1   517   515   2   830   819   11 UA   1545 N14228 EWR
- #> 2   533   529   4   850   830   20 UA   1714 N24211 LGA
- #> 3   542   540   2   923   850   33 AA   1141 N619AA JFK
- #> 4   544   545   -1   1004   1022   -18 B6   725 N804JB JFK
- #> 5   554   600   -6   812   837   -25 DL   461 N668DN LGA
- #> 6   554   558   -4   740   728   12 UA   1696 N39463 EWR
- #> # … with 336,770 more rows, 6 more variables

Select: More

- There are a number of helper functions you can use within select():

- starts_with("abc"): matches names that begin with “abc”.

- ends_with("xyz"): matches names that end with “xyz”.

- contains("ijk"): matches names that contain “ijk”.

- matches("(.)\\1"): selects variables that match a **regular expression**.
- This one matches any variables that contain repeated characters.

- num_range("x", 1:3): matches x1, x2 and x3.

- See ?select for more details.

More on regular expression are available here: https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html

Note:

- select() can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned.
- Instead, use rename(), which is a variant of select() that keeps all the variables that aren't explicitly mentioned
- rename(flights, tail_num = tailnum)
- Another option is to use select() in conjunction with the everything() helper.
- This is useful if you have a handful of variables you'd like to move to the start of the data frame.
- select(flights, time_hour, air_time, everything())

Mutate: Example

- Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns.
- That's the job of mutate().
- mutate() always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables.

```
#Addiing variables in flights_sml:
flights_sml <- select(flights,
            year:day,
            ends_with("delay"),
            distance,
            air_time
)
mutate(flights_sml,
            gain = dep_delay - arr_delay,
      speed = distance / air_time * 60
)
```

Mutate: Example

- Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns.
- That's the job of mutate().
- mutate() always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables.

```
#Adding one more variable:

mutate(flights_sml,
  gain = dep_delay - arr_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)

#Note that you/we can refer to columns that you've just created
```

Transmute and other useful creation functions More@ https://r4ds.had.co.nz/transform.html

- If you only want to keep the new variables, use transmute()

```
transmute(flights,
gain = dep_delay - arr_delay,
hours = air_time / 60,
gain_per_hour = gain / hours
)
```

- Arithmetic operators: +, -, *, /, ^
- Modular arithmetic: %/% (integer division) and %% (remainder)
- Use: Compute hour and minute from dep_time with:
- transmute(flights,

```
        dep_time,
        hour = dep_time %/% 100,
        minute = dep_time %% 100)
```

Summarise: Works best for group summaries

- summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
- #> # A tibble: 1 × 1
- #>   delay
- #>   <dbl>
- #> 1 12.6

- by_day <- group_by(flights, year, month, day)
- summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
- #> # A tibble: 365 × 4
- #> # Groups: year, month [12]
- #>   year month   day delay
- #>   <int> <int> <int> <dbl>
- #> 1 2013   1   1 11.5
- #> 2 2013   1   2 13.9
- #> 3 2013   1   3 11.0
- #> 4 2013   1   4 8.95
- #> 5 2013   1   5 5.73
- #> 6 2013   1   6 7.15
- #> # … with 359 more rows

Multiple operations: pipes

```
#What will happen?
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

```
#What will happen?
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
#And now?
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))
```

How to remove cancelled flights? And, get summaries by groups!

```
not_cancelled <- flights %>%
          filter(!is.na(dep_delay),
          !is.na(arr_delay))


not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean =
mean(dep_delay))
```

- #> # A tibble: 365 × 4
- #> # Groups: year, month [12]
- #>   year month   day mean
- #>   <int> <int> <int> <dbl>
- #> 1 2013   1   1 11.4
- #> 2 2013   1   2 13.7
- #> 3 2013   1   3 10.9
- #> 4 2013   1   4 8.97
- #> 5 2013   1   5 5.73
- #> 6 2013   1   6 7.15
- #> # … with 359 more rows

Counts: Example

- Whenever you do any aggregation, it’s always a good idea to include either a count (n()), or a count of non-missing values (sum(!is.na(x))).

- That way you can check that you’re not drawing conclusions based on very small amounts of data.

```
# What happens now?

delays <- not_cancelled %>%
        group_by(tailnum) %>%
        summarise(
        delay = mean(arr_delay)
)

hist(delays$delay)
```

What happens now?

```
delays <- not_cancelled %>%
        group_by(tailnum) %>%
        summarise(
        delay = mean(arr_delay,
na.rm = TRUE),
        n = n()
)
```

```
# Plots

hist(delays$n)

hist(delays$delay)

plot(delays$n, delays$delay)
```

Useful summary functions: https://r4ds.had.co.nz/transform.html

```
# When do the first and last flights leave each day?

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = min(dep_time),
    last = max(dep_time)
  )
```

- # Why is distance to some destinations more variable than to others?

```
not_cancelled %>%
      group_by(dest) %>%
      summarise(distance_sd =
       sd(distance)) %>%
      arrange(desc(distance_sd))
```

Useful summary functions: https://r4ds.had.co.nz/transform.html

```
# Which destinations have the most carriers?
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
```

- # How many flights left before 5am? (these usually indicate delayed flights from the previous day)

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))
```

Useful summary functions: https://r4ds.had.co.nz/transform.html

\# What proportion of flights are delayed by more than an hour?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(hour_prop = mean(arr_delay > 60))
```

\#Find all groups bigger than a threshold:

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
```

Popular destination: head and tail
(Are these results VALID?)

- head(popular_dests$dest)
- [1] "IAH" "IAH" "MIA" "BQN" "ATL" "ORD“

- IAH = Texas
- MIA = Miami
- BQN = Puerto Rico
- ATL = Atalanta
- ORD = Chichago

- tail(popular_dests$dest)
- [1] "BNA" "DCA" "SYR" "BNA" "CLE" "RDU“

- BNA = Nashville
- DCA = Washigton (Reagan Nat.)
- SYR = New York (Syracuse)
- CLE = Cleveland
- RDU = North Carolina

Bonus: dplyr “slice” function with examples https://dplyr.tidyverse.org/reference/slice.html

#What will happen?

flights %>% slice(1L)

flights %>% slice(n())

flights %>% slice(5:n())

slice(flights,-(1:4))

- flights %>% slice_sample(n=5)
- flights %>% slice_sample(n=5, replace = TRUE)
- **set seed(123)**
- train_data <- flights %>% slice_sample(prop=0.8)
- train_data
- test_data <- flights %>% slice_sample(prop=0.2)
- test_data

Question/Queries?

Database and R

- R Data Import/Export (rio):
  - https://cran.r-project.org/doc/manuals/r-release/R-data.html

- R Data Import/Export (rio): Relational database
  - https://cran.r-project.org/doc/manuals/r-release/R-data.html#Relational-databases

- Why use a database?
  - There are limitations on the types of data that R handles well.

Why use a database?

- Since all data being manipulated by R are resident in memory, and several copies of the data can be created during execution of a function, **R is not well suited to extremely large data sets**.
- Data objects that are more than a (few) hundred megabytes in size can cause R to run out of memory, particularly on a 32-bit operating system.
- **R does not easily support concurrent access to data**. That is, if more than one user is accessing, and perhaps updating, the same data, the changes made by one user will not be visible to the others.

Why use a database?

- **R does support persistence of data**, in that you can save a data object or an entire worksheet from one session and restore it at the subsequent session, but the format of the stored data is specific to R and not easily manipulated by other systems.

- Database management systems (DBMSs) and, in particular, relational DBMSs (RDBMSs) *are* designed to do all of these things well.

- The sort of statistical applications for which DBMS might be used are to extract a 10% sample of the data, to cross-tabulate data to produce a multi-dimensional contingency table, and to extract data group by group from a database for separate analysis.

R interface package for Database:

- There are several packages available on CRAN to help R communicate with DBMSs. They provide different levels of abstraction.
- Some provide means to copy whole data frames to and from databases.
- All have functions to select data within the database via SQL queries, and to retrieve the result as a whole as a data frame or in pieces (usually as groups of rows).
- All **except RODBC** are tied to one DBMS, but there has been a proposal for a unified 'front-end' package **DBI** (https://developer.r-project.org/db/) in conjunction with a 'back-end', the most developed of which is **RMySQL**.

Packages using DBI

- Package RMySQL on CRAN provides an interface to the MySQL database system (see https://www.mysql.com and Dubois, 2000) or its fork MariaDB (see https://mariadb.org/).

- The description here applies to versions 0.5-0 and later: earlier versions had a substantially different interface.

- The current version requires the DBI package, and this description will apply with minor changes to all the other back-ends to DBI.

Packages using DBI: MySQL

- MySQL exists on Unix/Linux/macOS and Windows: there is a ‘Community Edition’ released under GPL but commercial licenses are also available.
- MySQL was originally a ‘light and lean’ database. (It preserves the case of names where the operating file system is case-sensitive, so not on Windows.)
- The call **dbDriver("MySQL")** returns a database connection manager object, and then a call to **dbConnect** opens a database connection which can subsequently be closed by a call to the generic function **dbDisconnect**.
- Use dbDriver("Oracle"), dbDriver("PostgreSQL") or dbDriver("SQLite") with those DBMSs and packages ROracle, RPostgreSQL or RSQLite respectively.

Packages using DBI: MySQL

- SQL queries can be sent by either dbSendQuery or dbGetQuery.
- **dbGetquery** sends the query and retrieves the results as a data frame.
- **dbSendQuery** sends the query and returns an object of class inheriting from "**DBIResult**" which can be used to retrieve the results, and subsequently used in a call to **dbClearResult** to remove the result.
- Function **fetch** is used to retrieve some or all of the rows in the query result, as a list.
- The function **dbHasCompleted** indicates if all the rows have been fetched, and **dbGetRowCount** returns the number of rows in the result.

Packages using DBI: MySQL

##install “RMySQL” package, if needed!

##load “RMySQL” package

- library(RMySQL) # will load DBI as well;

open a connection to a MySQL database

- con <- dbConnect(dbDriver("MySQL"), dbname = "test")

list the tables in the database

- dbListTables(con)

load a data frame into the database, deleting any existing copy

- data(USArrests)
- dbWriteTable(con, "arrests", USArrests, overwrite = TRUE)
- dbListTables(con)

Packages using DBI: MySQL

- ## get the whole table
- > dbReadTable(con, "arrests")
- Murder Assault UrbanPop Rape
- Alabama 13.2 236 58 21.2
- Alaska 10.0 263 48 44.5
- Arizona 8.1 294 80 31.0
- Arkansas 8.8 190 50 19.5
- ...

Packages using DBI: MySQL

```
## Select from the loaded table
dbGetQuery(con, paste("select row_names, Murder from arrests",
                      "where Rape > 30 order by Murder"))

#$Remove table
dbRemoveTable(con, "arrests")

## disconnect from database
dbDisconnect(con)
```

Package RODBC

- Package RODBC on CRAN provides an interface to database sources supporting an Microsoft Open Database Connectivity (ODBC) interface.

- This is very widely available, and allows the same R code to access different database systems.

- RODBC runs on Unix/Linux, Windows and macOS, and almost all database systems provide support for ODBC.

- It has been tested Microsoft SQL Server, Access, MySQL, PostgreSQL, Oracle and IBM DB2 on Windows and MySQL, MariaDB, Oracle, PostgreSQL and SQLite on Linux.

Package RODBC

- ODBC is a client-server system, and it has been successfully connected to a DBMS running on a Unix server from a Windows client, and vice versa.
- On Windows ODBC support is part of the OS.
- On Unix/Linux you will need an ODBC Driver Manager such as unixODBC (http://www.unixODBC.org) or iOBDC (http://www.iODBC.org: this is pre-installed in macOS) and an installed driver for your database system.
- Windows provides drivers not just for DBMSs but also for Excel (.xls) spreadsheets, DBase (.dbf) files and even text files.

Package RODBC

- Many simultaneous connections are possible.
- A connection is opened by a call to **odbcConnect** or **odbcDriverConnect**, which returns a handle used for subsequent access to the database.
- Printing a connection will provide some details of the ODBC connection, and calling **odbcGetInfo** will give details on the client and server.
- A connection is closed by a call to close or **odbcClose**, and also (with a warning) when not R object refers to it and at the end of an R session.

Package RODBC

- Details of the tables on a connection can be found using **sqlTables**.
- Function **sqlSave** copies an R data frame to a table in the database, and **sqlFetch** copies a table in the database to an R data frame.
- An SQL query can be sent to the database by a call to **sqlQuery**. This returns the result in an R data frame. (sqlCopy sends a query to the database and saves the result as a table in the database.)
- A finer level of control is attained by first calling **odbcQuery** and then **sqlGetResults** to fetch the results. The latter can be used within a loop to retrieve a limited number of rows at a time, as can function **sqlFetchMore**.

Package RODBC: Example using PostgreSQL

install RODBC package, if needed

#Load ROBDC package

- library(RODBC)

tell it to map names to l/case

- channel <- odbcConnect("testdb", uid="ripley", case="tolower")

load a data frame into the database

- data(USArrests)
- sqlSave(channel, USArrests, rownames = "state", addPK = TRUE)
- rm(USArrests)

list the tables in the database

- sqlTables(channel)

Package RODBC

```
## list it
sqlFetch(channel, "USArrests", rownames = "state")

## an SQL query, originally on one line
sqlQuery(channel, "select state, murder from USArrests
        where rape > 30 order by murder")

## remove the table
sqlDrop(channel, "USArrests")
## close the connection
odbcClose(channel)
```

Getting Microsoft Access data in R: (Self-Learning)

- https://www.r-bloggers.com/2013/01/getting-access-data-into-r/
- https://leowong.ca/blog/connect-to-microsoft-access-database-via-r/
- https://cran.r-project.org/web/packages/RODBC/RODBC.pdf
- Same arguments holds true for Excel and any text files as they can also be imported with RODBC package in R!

dplyr and dbplyr packages for Database: https://r4ds.hadley.nz/databases.html

- Setting up a client-server or cloud DBMS is out of the scope of this course, so we’ll instead use an in-process DBMS that lives entirely in an R package: **duckdb**.
- install.packages(“duckdb”)

- Connecting to **duckdb** is particularly simple because the defaults create a temporary database that is deleted when you quit R.
- library(duckdb)
- library(dbplyr)   #Also required for “db” related functions

dplyr and dbplyr packages for Database: https://r4ds.hadley.nz/databases.html

#Making a connection:

- con <- DBI::dbConnect(duckdb::duckdb())

#Con is a new database so we need to add some tables there

- DBI::dbWriteTable(con, "mpg", ggplot2::mpg)
- DBI::dbWriteTable(con, "diamonds", ggplot2::diamonds)

#Check the list of tables in con with:

- DBI::dbListTables(con)

dplyr and dbplyr packages for Database: https://r4ds.hadley.nz/databases.html

- #What will happen with this?
- con %>% DBI::dbReadTable("diamonds") %>% as_tibble()

#You/we can use the “SQL” to get the same tibble as follows:

```
sql <- "
        SELECT carat, cut, clarity, color, price
        FROM diamonds
        WHERE price > 15000
"
```

#Print the sql object as tibble as follows:

```
as_tibble(DBI::dbGetQuery(con, sql))
```

dbplyr basics:

#To use dbplyr, you must first use **tbl()** to create an object that represents a database table:

- diamonds_db <- tbl(con, "diamonds")   #load ggplot2 for this data
- diamonds_db

#This object is lazy; when you use dplyr verbs on it, dplyr doesn’t do any work: it just records the sequence of operations that you want to perform and only performs them when needed.

- big_diamonds_db <-diamonds_db %>% filter(price > 1500) %>% select(carat:clarity, price)
- big_diamonds_db

dbplyr basics:

- You can tell this object represents a database query because it prints the DBMS name at the top, and while it tells you the number of columns, it typically doesn’t know the number of rows.
- This is because finding the total number of rows usually requires executing the complete query!
- big_diamonds_db %>% show_query() **#See SQL code generated by dbplyr**
- <SQL>
- SELECT "carat", "cut", "color", "clarity", "price"
- FROM "diamonds"
- WHERE ("price" > 1500.0)

dbplyr basics:

- To get all the data back into R, you call collect().
- Behind the scenes, this generates the SQL, calls dbGetQuery() to get the data, then turns the result into a tibble:

- big_diamonds <- big_diamonds_db %>% collect()
- big_diamonds

dbplyr basics:

- Typically, you’ll use dbplyr to select the data you want from the database, performing basic filtering and aggregation using the translations described earlier.
- Then, once you’re ready to analyse the data with functions that are unique to R, you’ll collect() the data to get an in-memory tibble, and continue your work with pure R code.

What will happen?

- dbplyr::copy_nycflights13(con)

- flights <- tbl(con, "flights")
- flights %>% show_query()

- planes <- tbl(con, "planes")
- planes %>% show_query()

What will happen?

- flights %>% filter(dest == "IAH") %>% arrange(dep_delay) %>% show_query()
- **WHERE and ORDER BY control which rows are included and how they are ordered**

- flights %>% group_by(dest) %>% summarize(dep_delay = mean(dep_delay, na.rm = TRUE)) %>% show_query()
- **GROUP BY converts the query to a summary, causing aggregation to happen**

More here:

- Chapter 22: Databases
- R for Data Science, 2nd Edition
- https://r4ds.hadley.nz/

Question/Queries?

Thank you!

@shitalbhandary