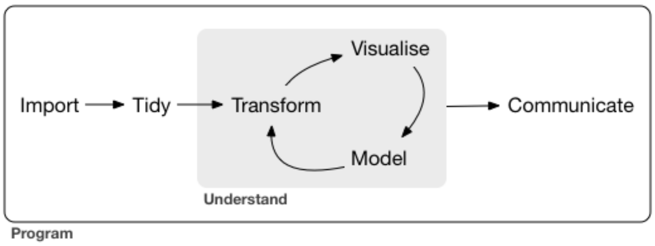
R4DS Chapter 1 & 2 Introduction

1. What you will learn



2. Tidyverse

install.packages("tidyverse")

3. R Markdown

a. Headers

Big
Small
H3
H4
H5
Smallest

b. List

LIST	R Markdown
1. Item-1	1. Item-1
2. Item-2	2. Item-2
- sub-Item-1	• sub-Item-1
- sub-Item-2	• sub-Item-2
4. Item-3	3. Item-3
* sub-Item-1	• sub-Item-1
+ sub-Item-2	• sub-Item-2

c. Text Format

i. Bold: ** Bold Text ** / - Bold Text -

ii. Italic: * Italic Text * / _Italic Text _

d. Chunks

// Check real R Markdown file

e. Display variable

`var`

f. Knitting

// Check R Studio

R4DS Chapter 4 Workflow: Basics

1. Coding Basics

a. Calculator

    1 / 200 * 30 // #> [1] 0.15
    (59 + 73 + 2) / 3 // #> [1] 44.66667
    sin(pi / 2) // #> [1] 1

b. Create new objects/Assignment Statement

    x <- 3 * 4
    object_name <- value

2. Object Names

a. recommend snake_case

b. <u>start with letter</u>

c. can include: number, _ , . , letter

d. be descriptive

3. Calling functions

    function_name(arg1 = val1, arg2 = val2, ...)

Example:

    seq(1, 10) // #> [1] 1 2 3 4 5 6 7 8 9 10

R4DS Chapter 3 Data Visualization

1. Prerequisites

```
library(tidyverse)
```

2. Creating a Plot

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, …))
```

3. Aesthetic Mapping

a. Color

i. color by group

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, color = group_name)
```

ii. set color

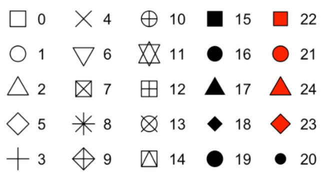
```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, color = "color_name")
```

b. Shape

i. shape by group

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, shape = group_name)
```

ii. set shape



c. Size

i. size by group

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, size = group_name)
```

ii. set size

* The size of a point in mm

d. Transparency

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, alpha = group_name or value)
```

3. Facets

a. To facet your plot by a single variable, use facet_wrap()

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, …)) +
    facet_wrap(~ var) // can set row numbers with facet_wrap(~ var , nrow = num)
```

b. To facet your plot on the combination of two variables, use facet_grid()

```
ggplot(data = <DATA>) +
    geom_function(mapping = aes(x = var1, y = var2, …)) +
```

facet_grid(varA ~ varB)

* If you prefer to not facet in the rows or columns dimension, use a . instead of a variable name like facet_grid(. ~ cyl)

4. Geometric Objects

a. geom_point() *Point*

i. normal

```
ggplot(data = mpg) +
    geom_point(mapping = aes(x = displ, y = hwy))
```

ii. position adjustment (position = "jitter")

Adds a small amount of random noise to each point. This spreads the points out to avoid overlapping in the graph.

```
ggplot(data = mpg) +
    geom_point(mapping = aes(x = displ, y = hwy), position = "jitter")
```

* usually used with geom_smooth to show trend.

b. geom_smooth() *Line*

i. normal

```
ggplot(data = mpg) +
    geom_smooth(mapping = aes(x = displ, y = hwy))
```

ii. set line type

```
ggplot(data = mpg) +
    geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```

iii. linear model

```
ggplot(data = mpg) +
    geom_smooth(mapping = aes(x = displ, y = hwy), method = “lm”, se = FALSE)
```

iv. geom_smooth(se=FALSE)

adds a smooth trend line on the graph which generally adapt to the shape of the data and helps reflect the trend of points in the dataset.

c. geom_bar() *Bar*

Bar charts, calculate new values to plot: bin your data and then plot bin counts, the number of points that fall in each bin.

i. normal (automatically count)

```
ggplot(data = diamonds) +
    geom_bar(mapping = aes(x = cut))
```

ii. map the height of the bars to the raw values of a y variable

```
ggplot(data = demo) +
    geom_bar(mapping = aes(x = cut, y = freq), stat = "identity")
```

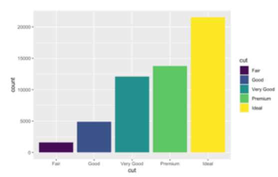
iii. display a bar chart of *proportion*

```
ggplot(data = diamonds) +
    geom_bar(mapping = aes(x = cut, y = stat(prop), group = 1))
```

iv. color the graph

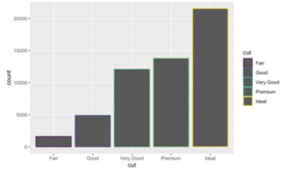
1) fill (fill by x variable)

```
ggplot(data = diamonds) +
    geom_bar(mapping = aes(x = cut, fill = cut))
```



2) color

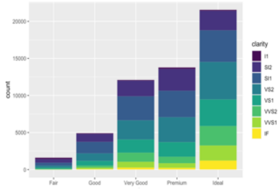
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, colour = cut))



v. position adjustment (fill by non-x variable)

1) raw (stacking)

ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity))



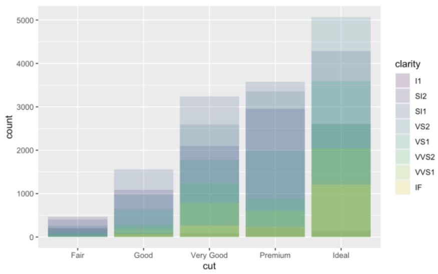
2) position adjustment

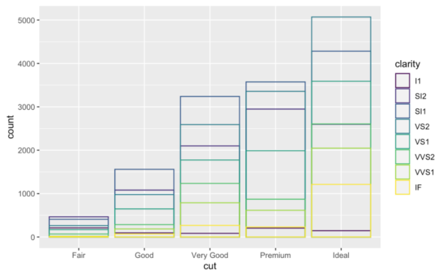
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity), **position = "XXX"**)

a) position = "identity"

place each object exactly where it falls in the context of the graph, which is not very useful for bars, because it overlaps them.

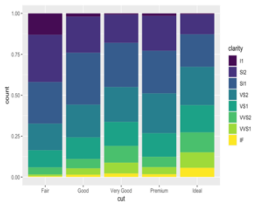
```
ggplot(data = diamonds, mapping = aes(x = cut, fill = clarity)) +
  geom_bar(alpha = 1/5, position = "identity")
ggplot(data = diamonds, mapping = aes(x = cut, colour = clarity)) +
  geom_bar(fill = NA, position = "identity")
```





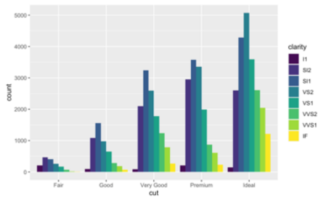
b) position = "fill"

works like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.



c) position = "dodge" (position = "dodge2")

places overlapping objects directly beside one another. This makes it easier. to compare individual values.



d. geom_boxplot() *Boxplot*

ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()

*define only y -> get single boxplot of distribution of certain variable;
define both xy -> get mutiple boxplots.

e. Coordinate system

i. coord_flip() switches the x and y axes.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```

ii. coord_quickmap() sets the aspect ratio correctly for maps.
iii. coord_polar() uses polar coordinates.

f. The Layered Grammar of Graphics

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

*When you use several geom_functions() together, please make sure you get the *layer* like colors' group, aesthetic settings right for each function.

Lecture

1. residuals: check Week2/Lecture1/last 5 minutes

```
lm1 <- lm(y~x, dataSet)
residuals1 <- residuals(lm1)
dataSet <- dset %>%
  mutate(residuals = residuals1)
ggplot(dset2, aes(x=x, y=residuals1)) +
  geom_point() +
  geom_hline(yintercept = 0, color="gray") +
  ylab("Residuals") +
  xlab("x") +
  ggtitle("Linear Model Residuals")
```

2. Check Dataset

```
spec(data)
str(data)
```

3. geom_histogram (also used in geom_bar)
a. binwidth = <u>num</u> : set bin's width of histogram.
b. boundary = <u>num</u> : set boundary of histogram.
c. center = <u>num</u> : set center of histogram.
d. bins = <u>num</u> : set number of bins.
*all defined in mapping = aes(…)
*histogram each bin is connected horizontally, bar has space between each bin.

4. geom_density() Density Plot
adjust = num : smooth of the graph (num greater, the graph get smoother)

Other about ggplot

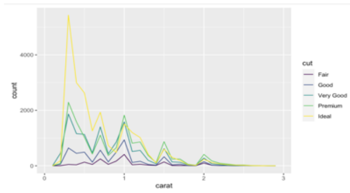
1. Adjust Texts on Labels

```
ggplot(data = planets_3) +
    geom_bar(mapping = aes(x = factor(method), y = stat(prop), group = 1)) +
    theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
    xlab("Method") +
    ylab("Proportions") +
    ggtitle("Observations of Exoplanet for Each Method")
```

2. points are jittered horizontally, but not vertically

```
geom_point(position = position_jitter(height = 0, width = 0.03), size = 0.01)
```

3. geom_freqpoly() performs the same calculation as geom_histogram(), but instead of displaying the counts with bars, uses lines instead.



1. Prerequisites

   library(tidyverse)

2. Data Types & Basic Information

a. Data Types

   i. int stands for integers.

   ii. dbl stands for doubles, or real numbers.

   iii. chr stands for character vectors, or strings.

   iv. dttm stands for date-times (a date + a time).

   v. lgl stands for logical, vectors that contain only TRUE or FALSE.

   vi. fctr stands for factors, which R uses to represent categorical variables with fixed. possible values.

   vii. date stands for dates.

b. dplyr Basics

   i. filter() Pick observations by their <u>values</u>.

   ii. arrange() Reorder the rows.

   iii. select() Pick variables by their <u>names</u>.

   iv. mutate() Create new variables with functions of existing variables.

   v. summarise() Collapse many values down to a single summary.

3. filter()

a. Comparsion & Logical Operators

   i. comparison operators

      1) normal: >, >=, <, <=, != (not equal), and == (equal)

      2) floating points: near(valueA, valueB)

   ii. logical operators

      & is "and"

      | is "or"

      ! is "not"

   iii. Examples

      filter(flights, month == 11 | month == 12) // finds all flights departed in Nov or Dec

b. x %in% y

   select every row where x is one of the values in y

   filter(flights, month %in% c(11, 12)) // finds all flights departed in Nov or Dec

   filter(year %in% c(1998,2002,2006,2010,2014,2018)

c. Missing value (NA)

   i. Basic information

      NA represents an unknown value so missing values, <u>almost any operation involving an unknown value will also be unknown</u>.

   ii. determine if a value is NA

      is.na(<u>value</u>) return true if value is NA

   iii. Example

      df <- tibble(x = c(1, NA, 3)) // df is dataSet, x is variable filter(df, is.na(x) | x > 1) // select value that is NA or greater than 1

4. arrange()

Takes a data frame and a set of column names to order by, and Missing values are. always sorted at the end.

a. increasing order

arrange(data, colName)

b. decending order

arrange(data, **desc**(colName))

5. select()

a. Basic

i. select columns by name

select(dataSet, year, month, day)

ii. select colums between columns

select(flights, year:day) // Select all columns between year and day (inclusive)

iii. select columns except certain columns

select(flights, -(year:day)) // Select all columns except those from year to day

b. Helper functions for selecting

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

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

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

iv. matches("(.)\\1"): selects variables that match a regular expression.

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

c. Special use other than selecting

i. Rename

???

*A better solution: rename(flights, tail_num = tailnum)

ii. Move variable to the start of Data Frame / Reorder varaible

// Move time_hour, air_time to the start of data frame flights

select(flights, time_hour, air_time, everything())

#vars <- c("year", "month", "day", "dep_delay", "arr_delay")

6. mutate()

a. Basic

adds new columns that are functions of existing columns at the end of the dataset

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

* only want to keep the new variables, use transmute() or use mutate() with select()

b. Arithmetic operator for mutate

+, -, *, /, ^, %/% (integer division), %% (remainder), logical comparison log(), log2(), log10()

c. Useful functions for mutate

i. Ranking (mutate a "rank" column)

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)
#> [1]  1  2  2 NA  4  5
min_rank(desc(y))
#> [1]  5  3  3 NA  2  1
row_number(y)
#> [1]  1  2  3 NA  4  5
dense_rank(y)
#> [1]  1  2  2 NA  3  4
percent_rank(y)
#> [1] 0.00 0.25 0.25 NA 0.75 1.00
cume_dist(y)
#> [1] 0.2 0.6 0.6 NA 0.8 1.0
```

ii. Cumulative calculation

```
cumsum(), cumprod(), cummin(), cummax(), cummean()
x // #>   [1]   1   2   3   4   5   6   7   8   9 10
cumsum(x) // #>   [1]   1 3 6 10 15 21 28 36 45 55
cummean(x) // #>   [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

7. summarise()

summarise() is not terribly useful unless we pair it with group_by(). This changes the unit of analysis from the complete dataset to individual groups.

a. Missing value

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

b. Counts

i. count

```
group_by(tailnum) %>%
summarise(n = count(var)))
```

ii. count non-missing value

```
group_by(tailnum) %>%
summarise(n = sum(!is.na( )))
```

iii. count the number of distinct (unique) values n_distinct(x)

```
group_by(tailnum) %>%
  summarise(carriers = n_distinct(carrier))
```

iv. n = n() return the size of current group (gives the count of category grouping by)

c. Useful Functions

i. measure of center

mean(x), median(x)

ii. measure of spread

sd(x): standard deviation
IQR(x): interquartile range
mad(x): median absolute deviation

iii. measure of rank

min(x), max(x)
quantile(x, 0.25): find a value of x that is greater than 25% of the values in x, and less than the remaining 75%

iv. measure of position

first(x), nth(x, positionNum), last(x)

v. sum()

Counts and proportions of logical values: sum(x > 10), sum(is.na(x)), mean(y == 0). When used with numeric functions, TRUE is converted to 1 and FALSE to 0. This makes sum() and mean() very useful: **sum(x) gives the number of TRUEs in x**, and **mean(x) gives the proportion**.

```
n = sum(logic_condtion) // check num of values in group match certain condition
n = sum(is.na(x)) // check num of missing value in group
```

8. group_by()

a. group_by() multiple variables (progressive)

makes it easy to progressively roll up a dataset

```
daily <- group_by(flights, year, month, day)
per_day <- summarise(daily, flights = n())
per_month <- summarise(per_day, flights = sum(flights))
per_year    <- summarise(per_month, flights = sum(flights))
```

b. group_by single variable (*use single variable to define a row*)

```
flights %>%
  group_by(year) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))
```

c. group_by mutliple variable (*use several variables to define a row*)

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

d. ungrouping

```
daily %>%
  ungroup() %>%
```

[END]

R4DS Chapter 7 Exploratory Data Analysis

1. Prerequisites
   library(tidyverse)

2. Basic Terms

a. ***variable*** is a quantity, quality, or property that you can measure.

b. ***value*** is the state of a variable when you measure it. The value of a variable may change from measurement to measurement.

c. ***observation*** is a set of measurements made under similar conditions (you usually make all of the measurements in an observation at the same time and on the same object). An observation will contain several values, each associated with a different variable. I'll sometimes refer to an observation as a data point.

d. ***Tabular data*** is a set of values, each associated with a variable and an observation Tabular data is tidy if each value is placed in its own "cell", each variable in its own column, and each observation in its own row.

e. ***Variation*** is the tendency of values of a variable to change from measurement to measurement.

f. A variable is ***categorical*** if it can only take one of a small set of values, usually saved as factors or character vectors

g. A variable is **continuous** if it can take any of an infinite set of ordered values numbers and date-times are two examples of continuous variables.

h. ***Covariation*** is the tendency for the values of two or more variables to vary together in a related way.

3. Visualization

a. Variation

i. categorical

geom_bar() // displays how many observations occurred with each x value

ii. continuous

1) geom_histogram() // To examine the distribution of a continuous variable

2) geom_freqpoly() // overlay multiple histograms in the same plot

b. Covariation

i. categorical & continuous

1) geom_freqpoly()

2) geom_bar()

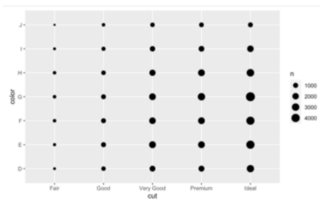
3) geom_histogram()

4) geom_boxplot()

ii. categorical & categorical

1) geom_count()

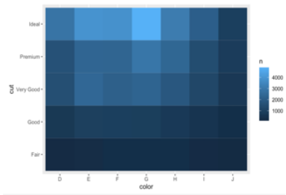
```
ggplot(data = diamonds) +
  geom_count(mapping = aes(x = cut, y = color))
```



2) compute the count then visualise with geom_tile() and the fill aesthetic

```
diamonds %>%
```

```
    count(color, cut)
  diamonds %>%
    count(color, cut) %>%
    ggplot(mapping = aes(x = color, y = cut)) +
    geom_tile(mapping = aes(fill = n))
```

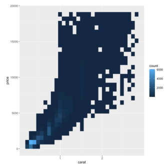


iii. continuous & continuous

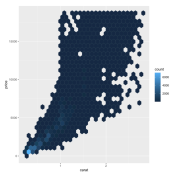
1) geom_point() then using the alpha aesthetic to add transparency to avoid ***overplot***

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price), alpha = 1 / 100)
```

2) geom_bin2d()



3) geom_hex()



4) geom_boxplot() with some adjustments

4. Remove Missing Value

1. not modify origin data frame

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

2. modify origin data frame

i. flights %>%
```
    group_by(year, month, day) %>%
    summarise(mean = mean(dep_delay, na.rm = TRUE))
```
ii. ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +
```
     geom_point(na.rm = TRUE)
```

5. Remove Unusual Values

replacing the unusual values with missing values, then drop_na()

```
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y)) %>%
  drop_na()
```

* ifelse(logicalVector, arg1, arg2) has three arguments. The first argument test should be a ***logical vector*** The result will contain the value of the second argument when test is TRUE, and the value of the third argument when it is false.

6. case_when()

case_when() usually used with mutate(), general format:

```
mutate(colName = case_when(
            logical_condition ~ content_for_col,
            logical_condition ~ content_for_col,
            logical_condition ~ content_for_col))
```

Example:

```
mutate(season = case_when(
            month %in% c("Sep","Oct","Nov") ~ "Fall",
            month %in% c("Dec","Jan","Feb") ~ "Winter",
            month %in% c("Mar","Apr","May") ~ "Spring",
            month %in% c("Jun","Jul","Aug") ~ "Summer"))
```

7. cut year

```
breaks <- seq(1869, 2019, 30)
labels <- str_c((breaks + 1)[-6], breaks[-1], sep = "-")
dataSetNew <- dataSet %>%
   mutate(period = cut(year, breaks =breaks, labels = labels)
```

8. as.XXX

a. as.numeric(col)

Converts a column into a numeric value column.

b. as.factor(col)

Specify a column type to be factor (also called categorical or enumerative), rather. than numeric.

c. as.character(col)

The function returns a string of 1's and 0's or a character vector of features depending on the nature of the fingerprint supplied.

9. separate()

separate() turns a single character column into multiple columns.

// Check Chapter 12 Tidy Data

10. rename()

```
rename(dataSet, c("originCol" = "newCol"))
```

11. Check Missing Data

```
count_na <- function(x) {
  return(sum(is.na(x)))}
dataSet %>%
   summarize_all(count_na)
```

R4DS Chapter 10 Tibbles

1. Introduction

Tibbles are data frames, but they tweak some older behaviours to make life a little easier.

```
library(tidyverse)
```

2. Creating Tibbles

a. Coerce a Data Frame to a Tibble

```
as_tibble(variable)
```

b. Create a New Tibble from Individual Vectors

```
tibble(
x = 1:5,
y = 1,
z = x ^ 2 + y
)
```

```
#> # A tibble: 5 x 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

c. Create Customised Tibble

i. column headings start with ~.

ii. entries are separated by commas.

ii. #line is optional, but adding that can make it clear where the header is.

```
tribble(
  ~x, ~y, ~z,
  #--|--|----
  "a", 2, 3.6,
  "b", 1, 8.5
)
```

*Comparing with data.frame(), tibble() does much less that it never changes the type of the inputs, never changes the names of variables, and never creates row names.

* It's possible for a tibble to have column names that are not valid R variable names, aka ***non-syntactic*** names (e.g. \`:)\`, \` \`, \`2000\`).

3. tibbles() vs. data.frame()

a. Printing

i. Tibbles have a refined print method that shows only the first 10 rows, and all the. columns that fit on screen, which makes it much easier to work with large data.

(1) control the number of rows (n) and the width of the display

```
dataSet %>%
print(n = 10, width = Inf)
```

(2) control the default print behaviour

```
options(tibble.print_max = n, tibble.print_min = m) //more than n rows print m rows
```

options tibble.print_min = Inf) // print all rows
options(tibble.width = Inf) // print all colums regardless the width of screen
(3) use RStudio's data viewer to get a scrollable view of the complete dataset
nycflights13::flights %>%
View()

b. Subsetting ($ and [[]])

i. Extract by name

dataFrame$name
dataFrame[["*name*"]]
dataFrame %>% .*$name* // use these in a pipe need the special placeholder "."

ii. Extract by position

dataFrame[[positionNum]]

* Compared to a data.frame, tibbles are more strict: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

4. Interacting with Older Code

Some older functions don't work with tibbles. If you encounter one of these functions, use as.data.frame() to turn a tibble back to a data.frame.

class(as.data.frame(tibbles))

R4DS Chapter 11 Data Import

1. Prerequisites

   library(tidyverse)

2. Functions of readr's

a. read_csv() reads comma delimited files.
b. read_csv2() reads semicolon separated files.
c. read_tsv() reads tab delimited files.
d. read_delim() reads in files with any delimiter.
e. read_fwf() reads fixed width files.
f. read_table() reads common variation of fixed width files where columns are separated by white space.
g. read_log() reads Apache style log files.

3. Use of read_csv()

a. read the file through path

   variable <- read_csv("*pathway*")

b. supply an inline csv file

   read_csv("*a,b,c*
   1,2,3
   4,5,6")

```
#> # A tibble: 2 x 3
#>       a     b     c
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

c. skip the first *n* lines of import files

   read_csv("*The first line of metadata* //this line will be ignore
   The second line of metadata //this line will be ignore
   x,y,z
   1,2,3", skip = *2*)

d. drop all lines start with #

   read_csv("*# A comment I want to skip* //this line will be ignore
   x,y,z
   1,2,3", comment = "#")

e. not to treat the first row as headings

   (treat the first row as heading check (b))

   read_csv("*1,2,3\n4,5,6*", col_names = *FALSE*)

f. set col_names

   read_csv("*1,2,3\n4,5,6*", col_names = c(*"x", "y", "z"*))

```
#> # A tibble: 2 x 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

g. specify value to represent Missing Values

   read_csv("*a,b,c\n1,2,.*", na = ".")

4. **Parsing a Vector** (parse_*() functions)
These functions take a character vector and return a more specialised vector like a. logical, integer, or date.

a. genral format
   i. *variable* <- *parse_**(c(*"1", "231", ".", "456"*), na = ".")
   ii. *parse_**(c(*"1", "231", ".", "456"*), na = ".")

b. get problems from parsing (failure parsing will be missing in the output)
   problems(*variable*)

c. parsers
   i. parse_logical() parses logicals like TRUE, FALSE, NA.
   ii. parse_integer() parses integers.
   iii. parse_double() a strict numeric parser.
   iv. parse_number() a flexible numeric parser by ignoring non-numerical characters.
   v. parse_character() parses characters, important in "character encodings."
   vi. parse_factor() creates factors, the data structure that R uses to represent. categorical. variables with fixed and known values.
   vii. parse_datetime() parse various date & time specifications.
   viii. parse_date() parse various date & time specifications.
   ix. parse_time() parse various date & time specifications.

d. Numbers
   i. why parsing numbers is tricky
      (1) People write numbers differently in different parts of the world.
      (2) Numbers are often surrounded by other characters like "10%".
      (3) Numbers often have grouping marks to make them easier to read like "10,000".
   ii. Solution
      (1) different decimal marks
         parse_double("*1,23*", locale = locale(decimal_mark = ","))
      (2) ignore non-numerical characters
         parse_number("*$100*")
      (3) ignore the "grouping mark"
         parse_number("*123.456.789*", locale = locale(grouping_mark = "."))

e. Strings
   i. specify encoding while parsing
      parse_character(x1, locale = locale(encoding = "*Latin1*"))
   ii. find the correct encoding
      guess_encoding(charToRaw(*strings*)) // guess_encoding() can also be path to files

f. Factors
   R uses factors to represent categorical variables with a known set of possible. values.

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
#> Warning: 1 parsing failure.
#> row col           expected   actual
#>   3  -- value in level set bananana
#> [1] apple  banana <NA>
#> attr(,"problems")
#> # A tibble: 1 x 4
#>     row   col expected           actual
#>   <int> <int> <chr>              <chr>
#> 1     3    NA value in level set bananana
#> Levels: apple banana
```

g. Dates, date-times, and times

i. parse_datetime()

expects an ISO8601 date-time, date are organised from biggest to smallest: year, month, day, hour, minute, second.

parse_datetime("*2010-10-01T1915*") // "2010-10-01 19:15:00 UTC"

parse_datetime("*20101010*") // "2010-10-10 UTC"

ii. parse_date()

expects a four digit year, a - or /, the month, a - or /, then the day.

parse_date("*2010-10-01*") // "2010-10-01"

iii. parse_time()

expects the hour, : , minutes, (optionally : and seconds), and (optionally am/pm specifier).

parse_time("*01:10 am*") // 01:10:00

parse_time("*20:10:01*") // 20:10:01

iv. personalize format

(1) Year

%Y (4 digits)

%y (2 digits) //00-69 -> 2000-2069, 70-99 -> 1970-1999

(2) Month

%m (2 digits)

%b (abbreviated name, like "Jan")

%B (full name, "January")

(3) Day

%d (2 digits)

%e (optional leading space)

(4) Time

%H 0-23 hour

%I 0-12, must be used with %p

%p AM/PM indicator

%M minutes

%S integer seconds

%OS real seconds

%Z Time zone (as name, e.g. America/Chicago)

%z (as offset from UTC, e.g. +0800).

(5) Non-digits

%. skips one non-digit character

%* skips any number of non-digits

(6) General Form

```
parse_date("01/02/15", "%m/%d/%y") // "2015-01-02"
parse_date("01/02/15", "%d/%m/%y") // "2015-02-01"
parse_date("01/02/15", "%y/%m/%d") // "2001-02-15"
```

5. Parsing a File (readr)

a. strategy to get types of each column

i. readr reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column: using guess_parser() returns readr's best guess, then parse_guess() uses that guess. to parse columns.

ii. details

The heuristic tries each of the following types, stopping when it finds a match:

(1) logical: contains only "F", "T", "FALSE", or "TRUE".

(2) integer: contains only numeric characters (and -).

(3) double: contains only valid doubles (including numbers like 4.5e-5).

(4) number: contains valid doubles with the grouping mark inside.

(5) time: matches the default time_format.

(6) date: matches the default date_format.

(7) date-time: any ISO8601 date.

If none of these rules apply, then the column will stay as a vector of strings.

b. problems

i. Don't always work for larger files: The first thousand rows might be a special case, and readr guesses a type that is not sufficiently general; The column might contain a lot of missing values. If the first 1000 rows contain only NAs, readr will guess that it's a logical vector.

```
problems(variable)
```

iv. solutions (set types of column by yourself)

Every parse_xyz() function has a corresponding col_xyz() function. You use parse_xyz() when the data is in a character vector in R already; you use col_xyz() when you want to tell readr how to load the data.

```
variable <- read_csv("challenge.csv"),
   col_types = cols(
      x = col_double(),
      y = col_date()
   )
)
```

c. Other Strategies

i. look at just one more row than the default

```
variable <- read_csv("challenge.csv"), guess_max = 1001)
```

ii. read in all the columns as character vectors

```
variable <- read_csv("challenge.csv"),
   col_types = cols(.default = col_character())
)
// This is particularly useful in conjunction with type_convert(), which applies the.
```

parsing heuristics to the character columns in a data frame.

iii. If you're reading a very large file, you might want to set n_max to a smallish. number like 10,000 or 100,000. That will accelerate your iterations while you eliminate common problems.

*iv. If you're having major parsing problems, sometimes it's easier to just read into a character vector of lines with read_lines(), or even a character vector of length 1 with read_file(). Then you can use the string parsing skills you'll learn later to parse more exotic formats.

6. Writing to a File

a. write_csv(), write_tsv(), write_excel_csv()

```
write_csv(variableStoreData, "fileName.csv")
```

*Note that the type information is lost when you save to csv.

b. write_rds(), read_rds()

uniform wrappers store data in R's custom binary format called RDS. (Type information will not get lost)

```
write_rds(variableStoreData, "fileName.csv")
```

c. write_feather()

The feather package implements a fast binary file format that can be shared across programming languages.

```
library(feather)
write_feather(variableStoreData, "fileName.csv")
```

7. Other Types of Data

To get other types of data into R, starting with the tidyverse packages listed below:

a. haven reads SPSS, Stata, and SAS files.

b. readxl reads excel files (both .xls and .xlsx).

c. DBI along with a database specific backend (e.g. RMySQL, RSQLite, RPostgreSQL etc) allows to run SQL queries against a database and return a data frame.

d. jsonlite for json

e. xml2 for XML

Lecture

1. replace column names of import files

```
colnames(dataframe) <- c("col1","col2",…)
```

1. Prerequisites

library(tidyverse)
library(lubridate)

2. Creating Date/Times

a. Introduction

i. ***date***: Tibbles print this as <date>.
ii. ***time***: Tibbles print this as <time>.
iii. ***date-time***: date plus a time, uniquely identifies an instant in time (typically to the. nearest second). Tibbles print this as <dttm>.

b. Get the current date or date-time

i. current date
today()
ii. current date-time
now()

c. Create a date/time

i. from String

(1) parse Stings into date-times //Check Chapter11/4/g

(2) lubridate functions

a) identify the order in which year, month, and day appear in your dates

ymd("*2017-01-31*") // "2017-01-31"
mdy("*January 31st, 2017*") // "2017-01-31"
dmy("*31-Jan-2017*") // "2017-01-31"
* These functions also take unquoted numbers.
ymd(*20170131*) // "2017-01-31"

b) To create a date-time, add an underscore and one or more of “h”, “m”, and. “s” to the name of the parsing function.

ymd_hms("*2017-01-31 20:11:59*") // "2017-01-31 20:11:59 UTC"
mdy_hm("*01/31/2017 08:01*") // "2017-01-31 08:01:00 UTC"

c) force the creation of a date-time from a date by supplying a timezone

ymd(*20170131*, tz = "*UTC*") // "2017-01-31 UTC"

ii. from individual date-time components

(1) Condition

have individual components of date-time spread across multiple cols

```
#> # A tibble: 336,776 x 5
#>    year month   day  hour minute
#>   <int> <int> <int> <dbl>  <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # … with 336,770 more rows
```

(2) To create a date/time from this sort of input, use make_date() for dates, or. make_datetime() for date-times.

a) five-time-cols (year, month, day, hour, minute)

make_datetime(year, month, day, hour, minute)

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
#> # A tibble: 336,776 x 6
#>    year month   day  hour minute departure
#>   <int> <int> <int> <dbl>  <dbl> <dttm>
#> 1  2013     1     1     5     15 2013-01-01 05:15:00
#> 2  2013     1     1     5     29 2013-01-01 05:29:00
#> 3  2013     1     1     5     40 2013-01-01 05:40:00
#> 4  2013     1     1     5     45 2013-01-01 05:45:00
#> 5  2013     1     1     6      0 2013-01-01 06:00:00
#> 6  2013     1     1     5     58 2013-01-01 05:58:00
#> # … with 336,770 more rows
```

b) four-time-cols (year, month, day, time)

```
var1 <- function(year, month, day, time) {
make_datetime(year, month, day, time %/% 100, time %% 100)
} // use modulus arithmetic to pull out the hour and minute components
dataSet2 <- originDataSet %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    … …
  ) // other steps
```

iii. **from an existing date/time object**

(1) offsets from "***Unix Epoch***" (Unix Epoch: 1970-01-01)

a) in seconds

as_datetime(*60 * 60 * 10*) // "1970-01-01 10:00:00 UTC"

b) in dats

as_date(*365 * 10 + 2*) // "1980-01-01"

(2) offsets from costomized date

as_date("*2021-02-25*") + ddays(*-2*) // "2021-02-23"

3. Date-time Components

a. Getting components

```
datetime <- ymd_hms("2016-07-08 12:34:56")
year(datetime) // 2016
month(datetime) // 7
mday(datetime) // 8 (day of the month)
yday(datetime) //190 (day of the year)
wday(datetime) // 6 (day of the week)
```

*For month() and wday() you can set label = TRUE to return the abbreviated name of the month or day of the week. Set abbr = FALSE to return the full name.

```
month(datetime, label = TRUE) // Jul
wday(datetime, label = TRUE, abbr = FALSE) // Friday
```

b. Rounding

An alternative approach to plotting individual components is to round the date to a nearby unit of time, with floor_date(), round_date(), and ceiling_date().

```
foor_date() // round down
ceiling_date() // round up
```

round_date() // round to

c. Setting components

i. Basics

datetime <- ymd_hms("*2016-07-08 12:34:56*")
year(*datetime*) <- *2020* // datetime get "2020-07-08 12:34:56 UTC"
month(*datetime*) <- *01* // datetime get "2020-01-08 12:34:56 UTC"
hour(*datetime*) <- hour(*datetime*) + *1* // datetime get "2020-01-08 13:34:56 UTC"

ii. create a new date-time

(1) update(datetime, year = 2020, month = 2, mday = 2, hour = 2)
// "2020-02-02 02:34:56 UTC"

(2) ymd("2015-02-01") %>%
update(mday = 30) // "2015-03-02"

* If values are too big, they will roll-over.

4. Time Spans

a. Three important classes represent time spans, use seconds.

i. ***durations*** represents an exact number of seconds.

ii. ***periods*** represents human units like weeks and months, don't have a fixed length in seconds.

iii. ***intervals*** represents a starting and ending point.

b. Duration

subtract two dates, you get a difftime object;
a difftime class object records a time span of seconds, minutes, hours, days, or weeks;
this ambiguity can make difftimes a little painful to work with;
duration always uses seconds can be a good alternative.

i. General

h_age <- *today()* - ymd(*19791014*) // get difftime object
as.duration(*h_age*) // get duration "1293494400s (~40.99 years)"

ii. Constructors of duration

dseconds(*n*) return n seconds
dminutes(*n*) return n*60 seconds
dhours(*n*) return n*3600 seconds
ddays(*n*) return n*86400 seconds
dweeks(*n*) return n*86400*7 seconds
dyears(*n*) return n*31557600 seconds
dxxxx(*n:m*) return (m-n) output for n*a seconds to m*b seconds
dxxxx(c(*n, m*)) return n*a seconds and m*b seconds

*duration can be added and multiplied
2 * dyears(*1*) // "63115200s (~2 years)"
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)

iii. Because durations represent an exact number of seconds, sometimes may get an unexpected result. (be careful with time zone like DST)

c. Periods

i. Constructors of periods

seconds(*15*) // "15S"

minutes(*10*) // "10M 0S"

hours(c(*12*, *24*)) // "12H 0M 0S" "24H 0M 0S"

days(*7*) // "7d 0H 0M 0S"

months(*1:6*) // "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d. 0H 0M 0S" "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"

weeks(*3*) // "21d 0H 0M 0S"

years(*1*) // "1y 0m 0d 0H 0M 0S"

* periods can be added and multiplied

ii. Compariosn between Periods and Duration

```
# A leap year
ymd("2016-01-01") + dyears(1)
#> [1] "2016-12-31 06:00:00 UTC"
ymd("2016-01-01") + years(1)
#> [1] "2017-01-01"
```

```
# Daylight Savings Time
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

d. Intervals

To make a more accurate measurement, you'll have to use an interval, which is a duration with a starting point.

next_year <- today() + years(1)

today() %--% next_year / ddays(1) // 365

e. Summary

	date	date time	duration	period	interval	number
date	-		- +	- +		- +
date time		-	- +	- +		- +
duration	- +	- +	- + /			- + × /
period	- +	- +		- +		- + × /
interval			/	/		
number	- +	- +	- + ×	- + ×	- + ×	- + × /

The allowed arithmetic operations between pairs of date/time classes.

5. Time Zones

a. names of time zones tend to be ambiguous (e.g. Eastern Standard Time in US, but. both Australia and Canada also have EST)

<continent>/<city> e.g. America/New_York

b. get current time zone

Sys.timezone()

c. In R, the time zone is an attribute of the date-time that only controls printing.

x1 <- ymd_hms("*2015-06-01 12:00:00*", tz = "*America/New_York*")

// "2015-06-01 12:00:00 EDT"

d. c() will often drop the time zone

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"
```

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2015-06-01 12:00:00 EDT" "2015-06-01 12:00:00 EDT"
#> [3] "2015-06-01 12:00:00 EDT"
```

*modify timezone for gourps

(1) Keep the instant in time the same, and change how it's displayed. (Use this. when the instant is correct, but you want a more natural display)

x4a <- with_tz(*x4*, tzone = "*Australia/Lord_Howe*")

(2) Change the underlying instant in time. (Use this when you have an instant that. has been labelled with the incorrect time zone)

x4b <- force_tz(*x4*, tzone = "*Australia/Lord_Howe*")

e. Unless otherwise specified, lubridate always uses UTC (Coordinated Universal Time) is the standard time zone used by the scientific community.

[END]

R4DS Chapter 12 Tidy Data

1. Prerequisites

library(tidyverse)

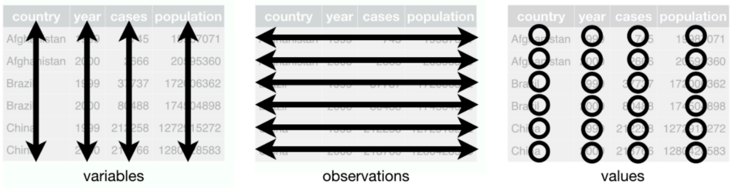
2. Tidy Data

a. Tidy Dataset

i. Each variable must have its own column.

ii. Each observation must have its own row.

iii. Each value must have its own cell.



* Three rules are interrelated.

b. Advantages

i. Having a consistent data structure, it's easier to learn the tools that work with it.

ii. Placing variables in columns allows R's vectorised nature to shine.

3. Pivoting

a. Longer

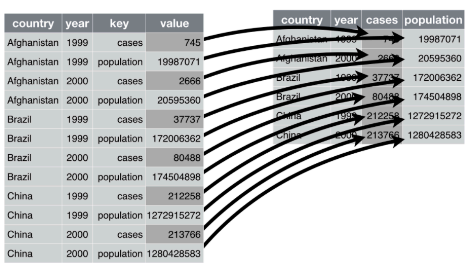
pivot_longer() makes datasets longer by increasing the number of rows and decreasing the number of columns. Used in case that variables and values are not in their positions.

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

b. Wider

pivot_wider() makes datasets longer by decreasing the number of rows and. increasing the number of columns. Used when an observation is scattered across multiple rows.

```
table2 %>%
  pivot_wider(names_from = key, values_from = value)
```



4. Separating and Uniting

a. Separate

i. separate() pulls apart one column into multiple columns, by splitting wherever a separator character appears.

ii. By default, separate() will split values wherever it sees a non-alphanumeric character like "/".

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

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

iii. To use a specific character to separate a column, you can pass the character to the sep argument of separate().

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

iv. Default behaviour in separate() leaves the type of the column as is, you can ask. separate() to try and convert to better types using convert = TRUE.

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

v. You can also pass a vector of integers to sep in separate() will interpret the integers as positions to split at.

i. Positive values start at 1 on the far-left of the strings; Negative value start at -1. on the far-right of the strings.

ii. The length of sep should be one less than the number of names in into.

b. Unite

i. unite() combines multiple columns into a single column.

ii. The default will place an underscore "_" between the values from different cols.

```
table5 %>%
  unite(newColName, col_1, col_2, sep = "")
```

5. Missing Values

a. Forms of missing value in dataset

i. Explicitly // i.e flagged with NA.

ii. Implicitly // i.e simply not present in the data.

* An explicit missing value is the presence of an absence; an implicit missing value. is the absence of a presence.

b. Turn explicit missing values implicit: values_drop_na = TRUE

```
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*
)

c. Make missing values explicit: complete() / fill()

i. complete() takes a set of columns, and finds all unique combinations. It then ensures the original dataset contains all those values, filling in explicit NAs where necessary.

stocks %>%
complete(*year*, *qtr*)

ii. fill() takes a set of columns where you want missing values to be replaced by the most recent non-missing value (sometimes called last observation carried forward).

treatment %>%
fill(*colName*)

6. Case Studies (Steps for Tidying Dataset)

a) Find/Specify variables (variables can be everywhere in untidy dataset).

b) Flip/Convert variables into correct positions.

c) Find meanings of variables(columns), what data they represent.

i) count() variables to get some clues

ii) obeserve and compare

d) Separate columns and drop redundant columns.

e) Handle missing values.

```
who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_to = "key",
    values_to = "cases",
    values_drop_na = TRUE
  ) %>%
  mutate(
    key = stringr::str_replace(key, "newrel", "new_rel")
  ) %>%
  separate(key, c("new", "var", "sexage")) %>%
  select(-new, -iso2, -iso3) %>%
  separate(sexage, c("sex", "age"), sep = 1)
```

7. Non-Tidy Data

If your data does fit naturally into a rectangular structure composed of observations and variables, I think tidy data should be your default choice. But there are good reasons to use other structures; tidy data is not the only way.

[END]

R4DS Chapter 13 Relation Data

1. Introduction

a. Prerequisites

library(tidyverse)

library(nycflights13) // we use the nycflights13 to learn about relational data

b. Relational Data

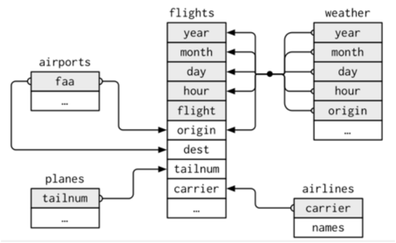
i. Multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

ii. Relations are always defined between a pair of tables.

c. Prepare

Find what variables connect one dataset to another.

e.g



1) flights connect to planes via tailnum.
2) flights connect to airlines via carrier.
3) flights connect to airports via the origin and dest.
4) flights connect to weather via origin, year, month, day, and hour.

… …

2. Keys

a. General

i. The variables used to connect each pair of tables are called ***keys***.

ii. A key is a variable (or set of variables) that uniquely identifies an observation.

e.g Each plane is uniquely identified by its tailnum.

b. Types

i. Primary Key

uniquely identifies an observation in its own table.

e.g planes$tailnum is a primary key because it uniquely identifies each plane in. the planes table.

ii. Foreign Key

uniquely identifies an observation in another table.

e.g flights$tailnum is a foreign key because it appears in the flights table where it matches each flight to a unique plane.

* A variable can be both a ***primary key*** and a ***foreign key***: be primary key in one dataset, and be foreign key in another dataset at the same time.

iii. Surrogate Key

If a table lacks a primary key, it's sometimes useful to add one with mutate(), which is called ***surrogate key***.

c. Find Primary Key

i. Decide possible primary key

e.g date & flight / tail number are primary key in the flights table

ii. Check

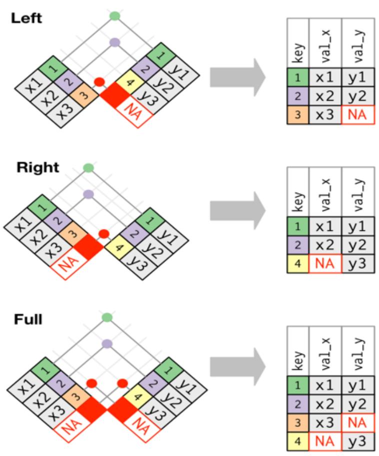
```
flights %>%
  count(year, month, day, flight) %>%
```

filter(n > 1)

// A tibble: 29,768 x 5 which suggests your possible primary key is not TRUE

A tibble: 0 x n suggests your primary key make some sense.

iii. Sometimes a table doesn't have an explicit primary key: each row is an observation, but no combination of variables reliably identifies it. If a table lacks a primary key, it's sometimes useful to add one with mutate().



3. Mutating Joins

a. General

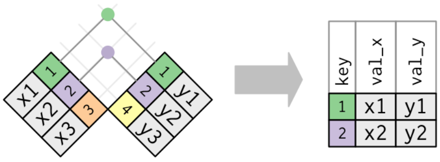
A mutating join allows you to combine variables from two tables. It first matches. observations by their keys, then copies across variables from one table to the other.

e.g
```
flights2 %>%
    select(-origin, -dest) %>%
    left_join(airlines, by = "carrier")
```
//This can also use mutate() to achieve the same effect (but more complicated)
```
flights2 %>%
    select(-origin, -dest) %>%
    mutate(name = airlines$name[match(carrier, airlines$carrier)])
```

b. Inner Joins

An inner join matches pairs of observations whenever their ***keys*** are equal.

The output of an inner join is a new data frame that contains the key, the x values, and the y values.



```
x %>%
  inner_join(y, by = "key")
```

* Unmatched rows are not included in the result, so that inner joins are usually not. appropriate for use in analysis because it's too easy to lose observations.

c. Outer Joins

i. An outer join keeps observations that appear in at least one of the tables.

ii. Types

1) Left Joins

keeps all observations in x

2) Right Joins

keeps all observations in y

3) Full Joins

keeps all observations in x and y

*The most commonly used join is the left join: you use this whenever you look up additional data from another table, because it preserves the original observations even when there isn't a match. Left join should be your default join.

*xxx*_join (*dataset_1*, *dataset_2*, by = "*key*")

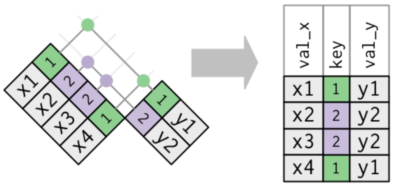
d. Duplicate Keys

i. General

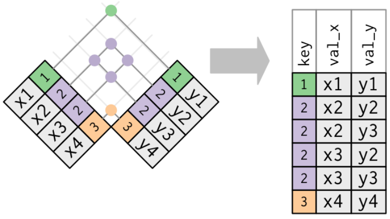
All the diagrams have assumed that the keys are unique, but that's not the case.

ii. Types

1) One table has duplicate keys (one-to-many relationship)



2) Both tables have duplicate keys (Very Possible Error Keys)



iii. Defining Key Columns

1) The default, by = NULL

uses all variables that appear in both tables called ***natural join***.

flights2 %>%

left_join(*weather*)

2) A character vector, by = "com_var"

uses only some of the common variables.

flights2 %>%

left_join(*planes*, by = "*tailnum*")

3) A named character vector, by = c("var_x" = "var_y")

matches variable var_x in table x to variable var_b in table y.

flights2 %>%

left_join(*airports*, c("*dest*" = "*faa*"))

// For example, we want to combine airport info with flights, then we use destination as a key to left-join corresponding data in airport.

e. Other Implementations

dplyr	merge
`inner_join(x, y)`	`merge(x, y)`
`left_join(x, y)`	`merge(x, y, all.x = TRUE)`
`right_join(x, y)`	`merge(x, y, all.y = TRUE)`,
`full_join(x, y)`	`merge(x, y, all.x = TRUE, all.y = TRUE)`

4. Filtering Joins

a. Filtering joins match observations in the same way as mutating joins, but affect the.

observations, not the variables.

b. Types

i. semi_join(x, y) keeps all observations in x that have a match in y.

ii. anti_join(x, y) drops all observations in x that have a match in y.

*Only the existence of a match is important other than which observation is. matched.

c. Semi-joins are useful for matching filtered summary tables back to the original rows.

e.g you've found the top ten most popular destinations, and now you want to find each flight that went to one of those destinations.

```
flights %>%
  semi_join(top_dest)
```

5. Join Problems/Solutions

a. Getting (un)lucky and find a combination that's unique in your current data but the. relationship might not be true in general.

b. Check that none of the variables in the primary key are missing. If a value is missing. then it can't identify an observation.

c. Check that your foreign keys match primary keys in another table. The best way to. do this is with an anti_join().

6. Set Operations

All these operations work with a complete row, comparing the values of every variable.

a. intersect(x, y): return only observations in both x and y.

b. union(x, y): return unique observations in x and y.

c. setdiff(x, y): return observations in x, but not in y.

[END]

1. Prerequisites
library(tidyverse)

2. String Basics
a. Create Strings
  i. General
    e.g *string1* <- "*This is a string*"
  ii. Single/Double quotation marks
    1) double: "\""
    2) single: '\' '
  iii. Backslash
    "\\"
  iv. Special characters
    1) newline
      "\n"
    2) tab
      "\t"
    3) non-English characters
      e.g "\u00b5"
  * see the complete list by requesting help on ": ?''", or ?'"'
  v. Multiple strings
    Multiple strings are often stored in a character vector.
    c("one", "two", "three") // #> [1] "one"   "two"   "three"
b. String Lengths
  str_length() tells you the number of characters in a string
  str_length("*fuck*") // #> [1] 4
  str_length(c("*a*", "*R for data science*", *NA*)) // #> [1] 1 18 NA
c. Combining Strings
  i. To combine two or more strings, use str_c()
    str_c("x", "y") // #> [1] "xy"
  ii. Use the sep argument to control how combines strings separated
    str_c("x", "y", sep = ", ") // #> [1] "x, y"
  iii. Print missing value NA as "NA", use str_replace_na()
    e.g x <- c("*abc*", *NA*)
      str_c("|-", *x*, "-|") // #> [1] "|-abc-|" NA
      str_c("|-", str_replace_na(*x*), "-|") // #> [1] "|-abc-|" "|-NA-|"
  iv. A vector of strings (multiple strings)
    1) Combine for different options with mult-str
      str_c("pre-", c("a", "b", "c"), "-fix") // #> [1] "pre-a-fix" "pre-b-fix" "pre-c-fix"
    2) collapse a vector of strings into a single string
      str_c(c("x", "y", "z"), collapse = ", ") // #> [1] "x, y, z"
  v. Objects of length 0 are silently dropped by str_c() during combining.

d. Subsetting Strings

extract parts of a string using str_sub(): str_sub() takes *start* and *end* arguments which give the (inclusive) position of the substring.

i. Positive start&end arguments

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3) // #> [1] "App" "Ban" "Pea"
```

ii. Negative start&end arguments

negative numbers count backwards from end

```
str_sub(x, -3, -1) // #> [1] "ple" "ana" "ear"
```

*str_sub() won't fail if the string is too short: it will just return as much as possible

e.g str_sub("a", 1, 5) // #> [1] "a"

*the assignment form of str_sub() to modify strings

str_sub(*x*, *1*, *1*) <- *stringx*

e. Locales

The locale is specified as a ISO 639 language code, which is a two or three letter abbreviation. different languages have different rules for changing case. You can pick which set of rules to use by specifying a locale. (R's default is English)

i. Change the text to lower case with locale

str_to_lower(*strings*, locale = "*localeAbrv*")

ii. Change the text to upper case with locale

```
str_to_upper(c("i", "ı"), locale = "tr") // Turkish I's uppercase #> [1] "İ" "I"
```

iii. Sorting

order() and sort() functions sort strings using the current locale.

```
x <- c("apple", "eggplant", "banana")
// Case A English
str_sort(x, locale = "en") // #> [1] "apple"      "banana"     "eggplant"
// Case B Hawaiian
str_sort(x, locale = "haw") // #> [1] "apple"      "eggplant" "banana"
```

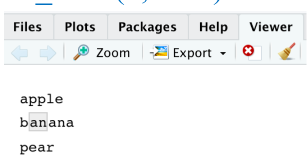
3. Matching Patterns with Regular Expressions

To learn regular expressions, we'll use str_view() and str_view_all().

a. Basic Matches

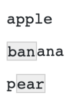
i. Simple

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```



ii. Match any character around assigned one with .

```
str_view(x, ".a.")
```



iii. Match the character "."

To match an ".", you need the regexp \.. Unfortunately, this creates a problem: We use strings to represent regular expressions, and \ is also used as an escape symbol in strings. So, to create the regular expression \. we need the string "\\."

str_view(c("abc", "a.c", "bef"), "a\\.c")

```
abc
a.c
bef
```

iv. Match the character "."

To match a literal \ you need to write "\\\\".

x <- "a\\b"

str_view(x, "\\\\")

```
a\b
```

b. Anchors

i. ^ to match the start of the string.

x <- c("apple", "banana", "pear")

str_view(x, "^a")

```
apple
banana
pear
```

ii. $ to match the end of the string.

str_view(x, "a$")

```
apple
banana
pear
```

**if you begin with power (^), you end up with money ($)*

iii. Match an <u>exact</u> string

x <- c("apple pie", "apple", "apple cake")

str_view(x, "apple") // This will match the strings contain the word "apple"

str_view(x, "^apple$") // This will match an EXACT string!

c. Character Classes and Alternatives

There are a number of special patterns that match more than one character.

i. \d: matches any digit.

ii. \s: matches any whitespace (e.g. space, tab, newline).

iii. [abc]: matches a, b, or c.

iv. [^abc]: matches anything except a, b, or c.

* to create a regular expression containing \d or \s, you'll need to escape the \ for the. string, so you'll type "\\d" or "\\s".

v. A character class containing a single character is a nice alternative to backslash. escapes when you want to include a single metacharacter in a regex. Many people find this more readable.

e.g str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c") // a.c

str_view(c("abc", "a.c", "a*c", "a c"), ".[*]c") // a*c
str_view(c("abc", "a.c", "a*c", "a c"), "a[ ]") // a c

* This works for most (but not all) regex metacharacters: $. | ? * + ( ) [{. Unfortunately, a few characters have special meaning even inside a character class and must be handled with backslash escapes:] \ ^ -.

vi. Pick between one or more alternative patterns with abc|xyz
str_view(c("grey", "gray"), "gr(e|a)y")

```
grey
gray
```

d. Repetition

Controlling how many times a pattern matches:

i. Simple control

1) ?: 0 or 1 (select zero or one matched item/ matching the shortest string possible)
2) +: 1 or more
3) *: 0 or more

e.g x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

ii. Precise control

1) {n}: exactly n
2) {n,}: n or more
3) {,m}: at most m
4) {n,m}: between n and m

e.g str_view(x, "C{2}")

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

str_view(x, "C{2,}")

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

str_view(x, "C{2,3}")

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

e. Grouping and Backreferences

A capturing group stores the part of the string matched by the part of the regular. expression inside the parentheses. You can refer to the same text as previously matched by a capturing group with backreferences, like \1, \2 etc.

e.g str_view(fruit, "(..)\\1", match = TRUE)

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

1. Prerequisites
   N/A

2. Create a New Function
a. pick a name for the function
b. list the inputs, or arguments, to the function inside function.
c. place the code you have developed in body of the function, a { block that.
   immediately follows function(...)

```
name_of_function <- function(parameters) {
    code for function
}
```

d. Examples:

```
rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10)) // #> [1] 0.0 0.5 1.0
```

e. if requirement (like parameters or code) changes, function has to be changed

```
x <- c(1:10, Inf)
rescale01(x) // #> [1]   0   0   0   0   0   0   0   0   0   0 NaN
rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE, finite = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x) // #> [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444.
                   0.5555556 0.6666667 [8] 0.7777778 0.8888889 1.0000000 Inf
```

3. Naming and Commenting
a. Naming
   i. the name of function will be short, but clearly evoke what the function does.
   ii. function names should be verbs, and arguments should be nouns.
   iii. If your function name is composed of multiple words using "snake_case".
   iv. If have a family of functions that do similar things, make sure they have.
      consistent names and arguments.
   v. avoid overriding existing functions and variables.
b. Commenting
   i. use comments, lines starting with #, to explain the "**why**" of your code.
   ii. break up your file into easily readable chunks. Use long lines of - and = to make.
      it easy to spot the breaks.
      Example:

```
# Load data -------------------------------------
# Plot data --------------------------------------
```

4. Conditional Execution

a. General Form

i. Format (ususally use inside the function())

1) if-else

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

2) if

```
if (condition) {
  # code executed when condition is TRUE
}
```

3) multiple conditions

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}
```

ii. Details

1) The condition must evaluate to either TRUE or FALSE; If it's a vector, you'll. get a warning message; if it's an NA, you'll get an error.

2) use || (or) and && (and) to combine multiple logical expressions; never use | or & in an if statement: these are vectorised operations that apply to multiple values (that's why you use them in filter()).

3) be wary of floating point numbers when using ==; x == NA is useless.

b. switch()

if you end up with a very long series of chained if statements, you should consider. rewriting. One useful technique is the switch() function.

```
function(x, y, op) {
  switch(op,
    plus = x + y,
    minus = x - y,
    times = x * y,
    divide = x / y,
    stop("Unknown op!")
  )
}
```

5. Function Arguments

a. General

i. The arguments to a function typically fall into two broad sets: one set supplies the data to compute on, and the other supplies arguments that control the details of the computation.

ii. Generally, data arguments should come first. Detail arguments should go on the end.

iii. specify a ***default value*** in the same way you call a function with a named argument. function(x, conf = 0.95) function(x, *default_value_name* = *value*)

iv. When you call a function, if you override the default value of a detail argument, you should use the full name.

b. Names of arguments

i. **x, y, z**: vectors.

ii. **w**: a vector of weights.

iii. **df**: a data frame.

iv. **i, j**: numeric indices (typically rows and columns).

v. **n**: length, or number of rows.

vi. **p**: number of columns.

c. Check arguments

It's good practice to check important preconditions, and throw an error (with. stop()), if they are not true.

```
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(w)
}
```

d. Dot-dot-dot

This special argument captures any number of arguments that aren't otherwise. matched.

```
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
```

* If you just want to capture the values of the ..., use list(...).

6. Return Values

a. Explicit return statements

i. The value returned by the function is usually the last statement it evaluates, but you. can choose to return early by using return().

```
complicated_function <- function(x, y, z) {
if (length(x) == 0 || length(y) == 0) {
  return(0) }
```

ii. Conditions

1) A common reason to do this is because the inputs are empty.
2) Another reason is because you have a if statement with one complex block and one simple block. (the first block is very long, by the time you get to the else, you've forgotten the condition. One way to rewrite it is to use an early return for the simple case)

b. Pipeable functions

i. ***transformations***

With transformations, an object is passed to the function's first argument and a. modified object is returned.

ii. ***side-effects***

With side-effects, the passed object is not transformed. Instead, the function performs an action on the object, like drawing a plot or saving a file.

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}
```

invisible() means that the input df doesn't get printed out

```
mtcars %>%
  show_missings() %>% // #> Missing values: 0
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings() // #> Missing values: 18
```

7. Environment

```
f <- function(x) {
  x + y
}
```

In many programming languages, this would be an error, because y is not defined. inside the function. In R, this is valid code because R uses rules called ***lexical scoping*** to find the value associated with a name. Since y is not defined inside the function, R will look in the ***environment*** where the function was defined.

NEW PAGE
START HERE
DO NOT REMOVE