dplyr & lubridate

Christian Odenwald, Julia Illmer, Fabian Sommerauer

Setup

Install dplyr & lubridate:

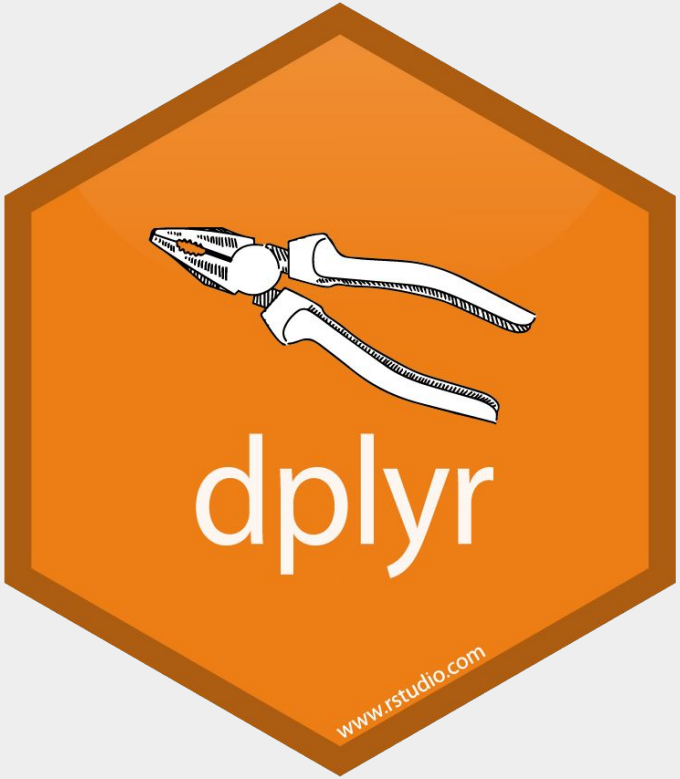
```
install.packages("dplyr")
install.packages("lubridate")
```

Download datasets:

```
install.packages("nycflights13")
```

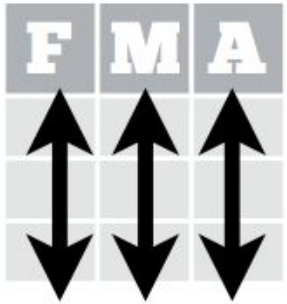
dplyr

Data Frame Transformation and Manipulation



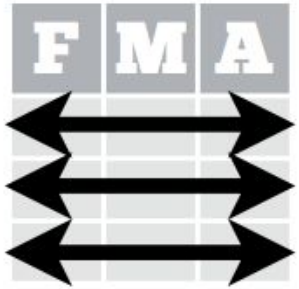
dplyr Basics: Tidy Data

In a tidy data set:



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

&



Each **observation** is saved in its own **row**

dplyr Basics: Uses

- Select subsets of data
- Group and aggregate data
- Reorder dataframes
- Create new rows / columns

Why use dplyr?

1. Simplicity and Understandability
2. Integration of External Databases
3. Well integrated in the R ecosystem (tidyverse)

Piping

```
f(x,y) → x %>% f(y)
f(x,y) → y %>% f(x, .)
```

Avoids storing every intermediate result

```
flights %>%
  filter(tailnum=="N14228") %>%
  select(tailnum, air_time, origin, dest) %>%
  slice_head(n=4)
```

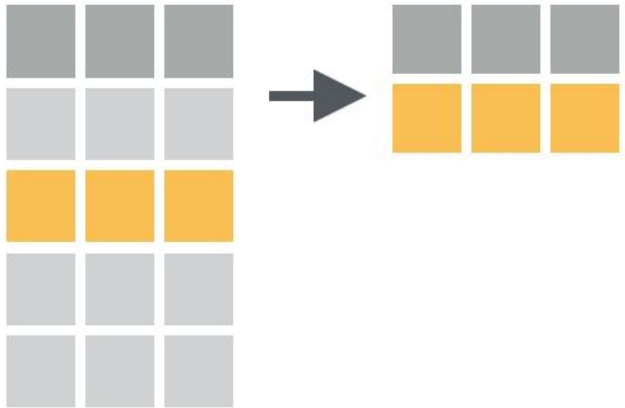
Observation Extraction

Filter()

```
filter(.data, ..., .preserve = FALSE)
```

Takes (multiple) boolean expressions as argument

Outputs rows which fulfill all conditions



Filter()

```
# select planes with less than 10 seats
filter(planes, seats < 10)
planes %>% filter(seats < 10)


# planes with 9-11 seats
planes %>% filter(between(seats, 9, 11))
planes %>% filter(near(seats, 10, 2))
```

Filter()

```
# planes from the year 1998 or 2004
planes %>% filter(year == 1998 | year == 2004)



# find planes from EMBRAER where the year is missing
planes %>% filter(is.na(year) &
                    manufacturer == "EMBRAER")
planes %>% filter(is.na(year),
                    manufacturer == "EMBRAER")
```

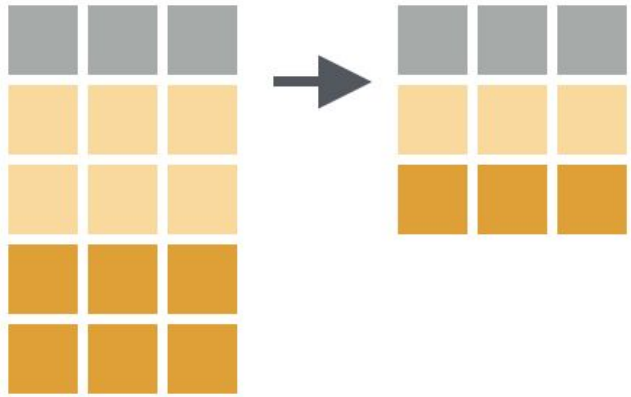
Distinct()

```
distinct(.data, ..., .keep_all = FALSE)
```

Removes duplicate rows from .data

Variables used for Uniqueness-Check can be specified

Per default compares rows by all variables



Distinct()

```
# get distinct plane manufacturers
planes['manufacturer'] %>% distinct()
planes %>% distinct(manufacturer)


# get number of distinct manufacturer-model pairs
planes %>% distinct(manufacturer, model) %>% nrow()
```

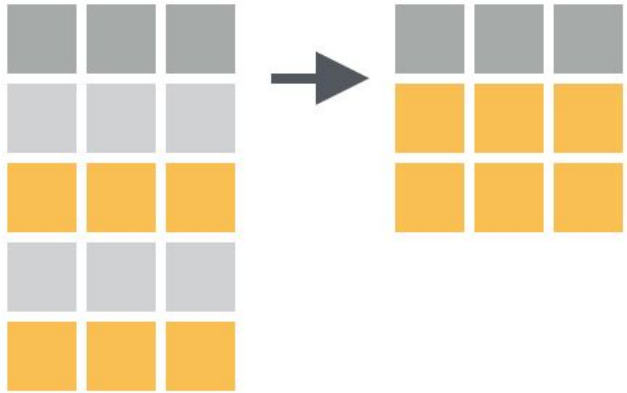
Slice()

```
slice(.data, ..., .preserve = FALSE)
```

Select rows by integer indices

Exclusion of rows using negative indices

```
slice_head() / slice_tail()
slice_min() / slice_max()
slice_sample()
```



Slice()

```
# select first, third and fifth airline
airlines %>% slice(c(1,3,5))



# select everything except the first,
# third and fifth airline
airlines %>% slice(-c(1,3,5))
```

Slice()

```
# select rows 4 to 10
airlines %>% slice(4:10)


# select 3rd airline from the back
airlines %>% slice(n()-2)
```

Slice_head() / Slice_tail()

```
# select first/last 5 rows
airlines %>% slice_head(n=5)
airlines %>% slice_tail(n=5)


# select first/last 20% of rows
airlines %>% slice_head(prop=0.2)
airlines %>% slice_tail(prop=0.2)
```

Slice_min() / Slice_max()

```
# select at least 5 planes with min/max number of seats
planes %>% slice_min(seats, n=5)
planes %>% slice_max(seats, n=5)


# select exactly 5 planes with min/max number of seats
# (output first in case of ties)
planes %>% slice_min(seats, n=5, with_ties = F)
planes %>% slice_max(seats, n=5, with_ties = F)
```

Slice_sample()

```
# uniformly sample 5 airlines
airlines %>% slice_sample(n=5)


# uniformly sample 5 airlines (allow duplicates)
airlines %>% slice_sample(n=5, replace = T)
```

Slice_sample()

```
# sample weather entries with
# more weight given to entries
# with high temperature
weather %>% filter(!is.na(temp))
       %>% slice_sample(n = 5, weight_by = temp)
```

Exercises [filter, distinct, slice]

1. Which planes have exactly 4 engines?
2. How many flights go from LaGuardia (LGA) to Baltimore (BWI)?
3. Get a list of distinct plane types
4. Get the last 3 entries of ‘planes’
5. Get the 10th, 11th and 12th entry of ‘planes’

Variable Extraction

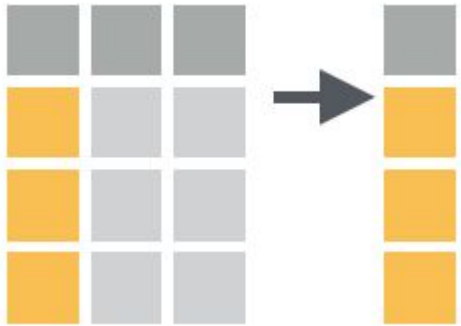
Select()

```
select(.data, ...)
```

Selects columns from .data

Also allows reordering and renaming

Supports extensive operators & Helpers



Select()

```
# select name, latitude and longitude from all airports
airports %>% select(name, lat, lon)


# select columns year - model from planes
planes %>% select(year:model)
```

Select()

```
# select wind information from weather entries
# [wind_dir, wind_speed, wind_gust]
weather %>% select(starts_with("wind_"))
weather %>% select(contains("wind_"))


# select time information which is not departure related
# [arr_time, air_time, ...]
flights %>% select(ends_with("_time") &
                             !contains("dep_"))
```

Select()

```
# select columns from weather which contain
# 'wind' or 'pre' using regular expressions
weather %>% select(matches("wind|pre"))


# select numeric columns from planes
flights %>% select(where(is.numeric))
```

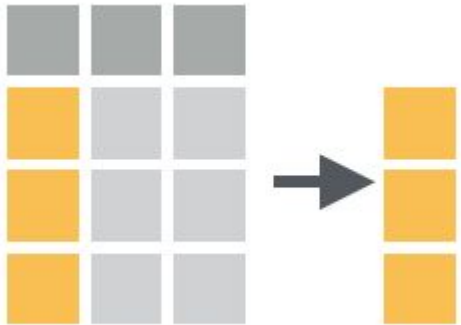
Pull()

```
pull(.data, var = -1, name = NULL, ...)
```

Extract single column as a vector

Variable name or integer for indexing

Optional name parameter for names of named vector



Pull()

```
# extract altitudes of airports as a vector
airports %>% pull(alt)


# we can also use numbers for indexing
airports %>% pull(1) # first col
airports %>% pull(-1) # last col


# pull named vector of years with model column as names
planes %>% pull(model, year)
```

Exercises [select, pull]

1. Select the year and model column from planes.

2. Select the flight, tailnum, origin, destination and delay information from flights.

3. Select flight column and all character columns from flights

Exercises [select, pull]

4. Extract a vector of the 4th column of flights. Get the value at position 400.

5. Extract a named vector of airport time zones (tz) together with the airport name. What is the 500th entry?

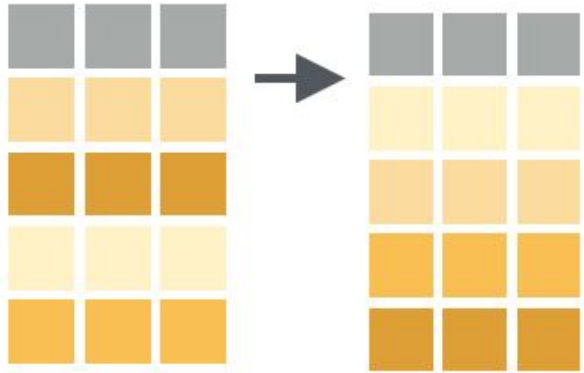
Table Manipulation

Arrange()

```
arrange(.data, ..., .by_group = FALSE)
```

Orders rows by values of selected columns

Using desc(col) for descending order



Arrange()

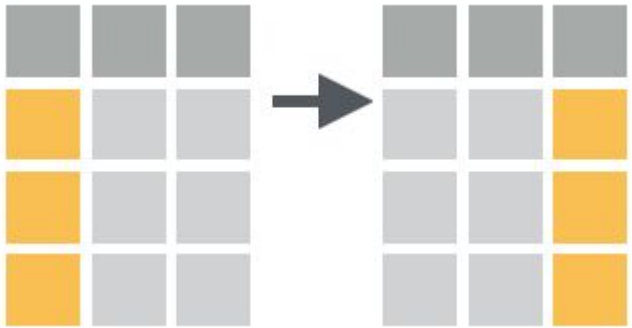
```
# sort planes first by year and model
# (higher priority for year)
planes %>% arrange(year, model)


# sort planes descending by Nr. of seats
planes %>% arrange(desc(seats))
```

Relocate()

```
relocate(.data, ..., .before = NULL, .after = NULL)
```

Move columns to locations defined by .before and .after

Per default moves columns to start of data frame



Relocate()

```
# move model & manufacturer to the beginning of planes
planes %>% relocate(model, manufacturer)


# move year to the end
planes %>% relocate(year, .after=last_col())


# renames are also possible
planes %>% relocate(velocity=speed, .before=engines)
```

Relocate()

```
# using where for custom selection
# move numeric variables to the beginning
planes %>% relocate(where(is.numeric),
                    .before=where(is.character))
```

Rename()

`rename(.data, ...)`

Renames columns using `(new_name = old_name)` as arguments

`rename_with(.data, .fn, .cols = everything(), ...)`

renames columns using a function

.fn should return character vector of new column names

Rename()

```
# rename abbreviations lat, lon and alt to full form
airports %>% rename(latitude=lat, longitude=lon,
                    altitude=alt)
```

Rename_with()

```
# rename_with to use arbitrary functions
# convert all columns to uppercase
flights %>% rename_with(toupper)


# column conditions
# convert columns ending with '_time' to uppercase
flights %>% rename_with(toupper, ends_with("_time"))
```

Rename_with()

```
# formulas for functions with multiple variables
# replace '_' with ':' in variable names
flights %>% rename_with(
                ~gsub("_", ":", ., fixed = TRUE)
                )
```

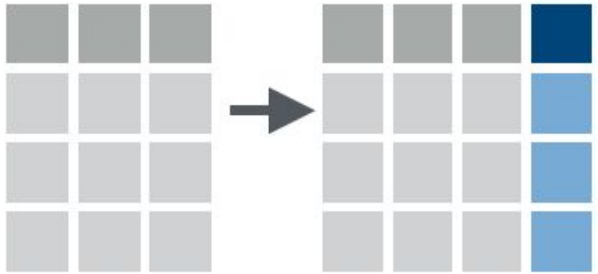
Mutate()

`mutate(.data, ...)`

Adds new variables to existing ones

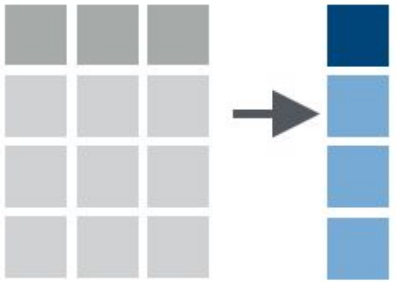
Name value pairs define new columns

Can utilize **vectorized functions**



`transmute(.data, ...)`

Drops existing variables



Mutate()

```
# create column speed per engine
planes %>% filter(!is.na(speed))
       %>% mutate(speed_per_engine=speed/engines)

# remove type and manufacturer column
planes %>% mutate(type=NULL, manufacturer=NULL)

# convert carrier and origin columns to factor
flights %>% mutate(carrier=as.factor(carrier),
                   origin=as.factor(origin))
```

Transmute()

```
# collect origin, destination, air time and distance and
# calculate avg speed of each flight
flights %>% transmute(origin, dest, air_time, distance,
                      avg_speed=distance/air_time*60)
```

Exercises [arrange, relocate, rename, mutate]

1. Sort weather data by temperature and humidity.
2. Sort airports descendingly by name. What is the 4th entry?
3. Move tailnum and flight to the beginning of the flights table.
4. From airports rename column ‘tzone’ to ‘timezone’ and move it after the ‘name’ column

Exercises [arrange, relocate, rename, mutate]

5. Change all column names of weather to uppercase.

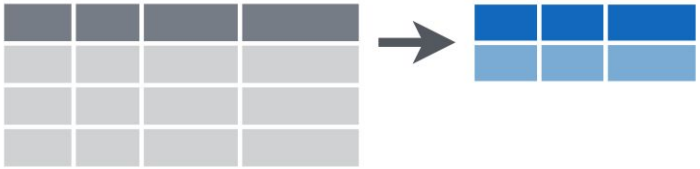
6. Create new ‘alt_centered’ column for airports which is the altitude minus the mean altitude.

7. Normalize temperature and humidity from weather by their respective maximum.

Summarise Data

Summarise data: summarise()

```
summarise(.data, …, .groups = NULL)
```



summarises data into single row of values

creates a new data frame

Summarise data: useful summary functions

functions that take a vector of values and return a single value

- `sum()`
- `mean()`
- `median()`
- `min()`
- `max()`
- `n()/ n_distinct()`
- `first()`
- `last()`
- `nth()`
- `IQR()`
- `sd()`
- `var()`

Summarise data: useful summary functions

functions that take a vector of values and return a single value

```
# Ex.1: summarise the total airtime

flights %>%
    summarise(total_airtime = sum(air_time, na.rm=TRUE))


# Ex.2: summarise the total airtime and the average distance
flights %>%
    summarise(total_airtime = sum(air_time, na.rm=TRUE),
        average_distance = mean(distance))
```

Group data: group_by()

```
group_by(.data, ..., .add = FALSE,
.drop = group_by_drop_default(.data))
```

group_by() takes an existing tbl and converts it into a grouped tbl where operations are performed "by group".

```
ungroup(x, …)
```

ungroup() removes grouping.

Group data: group_by, ungroup

```
# Ex.3: find the total flown distance within each month

flights %>%
  group_by(month) %>%
  summarize(total_distance = sum(distance))


# Ex.4: group by multiple columns: get the average delay grouped by the
year, month and day
daily_delay <- flights %>%
  group_by(year, month, day) %>%
  summarise(daily_mean_delay = mean(dep_delay, na.rm =TRUE))
# Ex.5: remove grouping, and return to operations on ungrouped data
daily_delay %>%
  ungroup() %>%
  summarise(yearly_mean_delay = mean(daily_mean_delay,na.rm=TRUE))
```

Summarise data: count()

```
count(x, ..., wt = NULL, sort = FALSE, name = NULL)
```

count() lets you quickly count the unique values of one or more variables

Supply wt to perform weighted counts, switching the summary from n = n() to n = sum(wt)

```
df %>% count(a, b)
```

is roughly equivalent to

```
df %>% group_by(a, b) %>% summarise(n = n())
```

Summarise data: count()

Count number of rows with each unique value of variable (with or without weights).

```
# Ex.6: count total number of flights

flights %>%
    count()

# Ex.7: count number of flights in each month
flights %>%
    count(month)

# Ex.8: + sort from most common to least
flights %>%
  count(month, sort = TRUE)

# Ex.9: count the total flown distance in each month
flights %>%
  count(month, wt = distance, sort = TRUE)
```

Summarise data: summarise(), group_by()

```
# Task 1:
# Summarise the weather dataset to find the following
columns: min_temp (with the lowest temperature),
max_windspeed (with the maximum wind speed), and
average_pressure (with the mean of the sea level
pressure)

# Task 2:
# Summarise the weather data frame to get the maximal
temperature of each month
```

Summarise data: summarise(), group_by()

```
# Task 3:
# Summarise the weather data frame to get the average
temperature for each day of the year.


# Task 4:
# Group the plane dataset by manufacturer, and summarize
to create the columns min_seats (with the feweset
passenger seats) and max_seats (with most passenger
seats).Then arrange for max_seats in descending order.

# Task 5:
# Use the data frame flights: Which carrier has the worst
delays? (hint: additional step of arrange() = useful)
```

Summarise data: count()

```
# Task 6a (use the airport data frame):
# Use count() to find the number of airports in each
timezone, using a second argument to sort in descending
order. Which timezone has the most airports?

# Task 6b (use the flights data frame):
# Count the total distance in miles a specific plane (by
plane tail number) flew, and sort in descending order.
```

Multi-Variable Manipulation: across(), c_across() & rowwise()

```
across(.cols = everything(), .fns = NULL, ..., .names = NULL)
```

allows to apply a function (or functions) across multiple columns at once

.cols → columns to transform

.fns → functions to apply (mean, median, min/max,...)

.names → how to name the output columns {.col}

```
rowwise()
```

performing operations over rows -> groups into individual rows → Each row is its own group

doesn't really do anything itself; it just changes how the other verbs work

```
c_across(cols = everything())
```

works with rowwise() to make it easy to perform row-wise aggregations

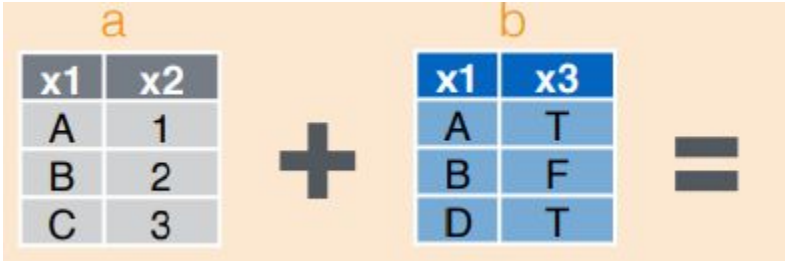
Multi-Variable Manipulation: across() & c_across()

```
# Ex.10: combine summarise() and across() to find the mean times
for all the flight data

flights %>%
  summarise(across(.cols = ends_with("time"),
                   .fns = mean,
                   .names = "mean_{.col}", na.rm = TRUE))


# Ex.11: use rowwise() & c_across() to calculate the actual
dep_time for the first 10 rows

flights %>%
  head(10) %>%
  rowwise() %>%
  mutate(calc_deptime = sum(c_across(sched_dep_time:dep_delay)))
```

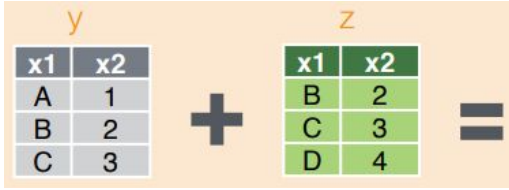
Multi-Variable Manipulation: across() & c_across()

```
# Which statement is true?
1.  across() and c_across() can be used interchangeably with
    rowwise().
2.  c_across() is used with rowwise() to choose columns to apply a
    function over for the row.
3.  across() is used only with rowwise() to choose columns to apply
    a function over for the row.
4.  across() and c_across() can be used interchangeably without
    rowwise().
```

Combine Datasets



Combine tables: bind_rows() & bind_cols()



for binding many data frames into one

they return the same type as the first input

assume correct order & length

```
bind_rows(..., .id = NULL)
```

columns are matched by name, and any missing columns will be filled with NA

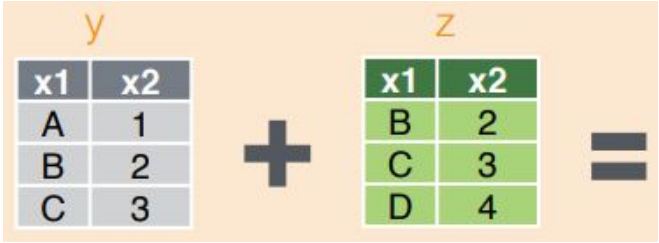
x1	x2
A	1
B	2
C	3
B	2
C	3
D	4

```
bind_cols(...,.name_repair = c("unique", "universal",
"check_unique", "minimal"))
```

rows are matched by position, so all data frames must have the same number of rows

x1	x2	x1	x2
A	1	B	2
B	2	C	3
C	3	D	4

Set operations



be aware that they all remove duplicates

x,y → objects to perform set function on (ignoring order)

`intersect(x, y, …)`

Rows that appear in both x and y

x1	x2
B	2
C	3

`union(x, y, …)`

Rows that appear in either or both x and y

x1	x2
A	1
B	2
C	3
D	4

`setdiff(x, y, …)`

Rows that appear in x but not y

x1	x2
A	1

Combine tables: bind_rows() & bind_cols()

```
# Ex.12: combine the flights & weather dataset by using
bind_rows

bind_rows(flights, weather)


# Ex.13: combine the flights & weather dataset by using
bind_cols. What happens?

bind_cols(flights, weather)
```

Set operations: intersect(), setdiff() & union()

```
# Ex.14a: select just the weather data of origin & temp and
store it in temp_data

temp_data <- weather %>%
  select(origin, temp)

# Ex.14b: then create two different tables named temp_data_1
and temp_data_2 & try the different set operations

temp_data_1 <- temp_data[1:10, ]
temp_data_2 <- temp_data[101:110, ]

intersect(temp_data_1, temp_data_2)
setdiff(temp_data_1,temp_data_2)
union(temp_data_1,temp_data_2)
```

Combine tables: bind_rows() & set operations

```
# Task 7

# Make two new tables routes_jan & routes_dec including
the origin and dest of flights from january, and
december.[hint: use filter() & select()]

# a) how many routes are flown in both january and december?
# b) how many routes are flown exclusively in january?
# c) how many different routes are generally flown in both
or either january and december?

# Task 7d
# Then bind routes_jan & routes_dec again together using
bind_rows to create a table of routes from the beginning
and ending of the year.
```

Joins: Mutating joins

The mutating joins add columns from y to x, matching rows based on the keys:

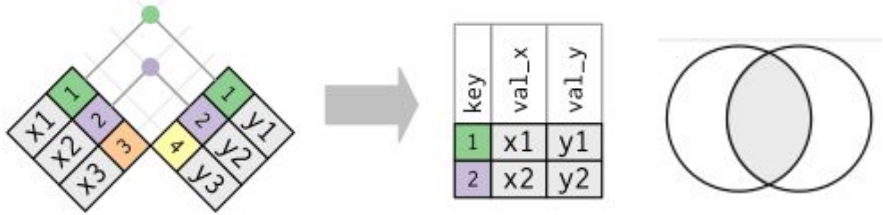
- `inner_join()`: includes all rows in x and y.

Outer joins:

- `left_join()`: includes all rows in x.
- `right_join()`: includes all rows in y.
- `full_join()`: includes all rows in x or y.

If a row in x matches multiple rows in y, all the rows in y will be returned once for each matching row in x.

Mutating joins: inner_join()




```
inner_join(x, y, by = “key”,suffix = c(".x", ".y"))
```

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

a subset of x rows

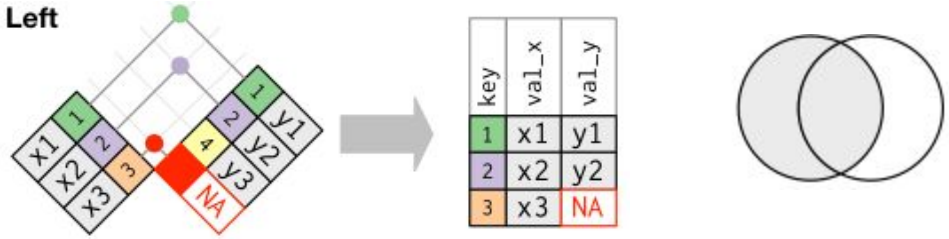
use by to tell dplyr which variable is the key

If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them

unmatched rows are not included in the result -> easy to lose observations

keeps only observations that appear in both tables

Mutating joins: left_join()



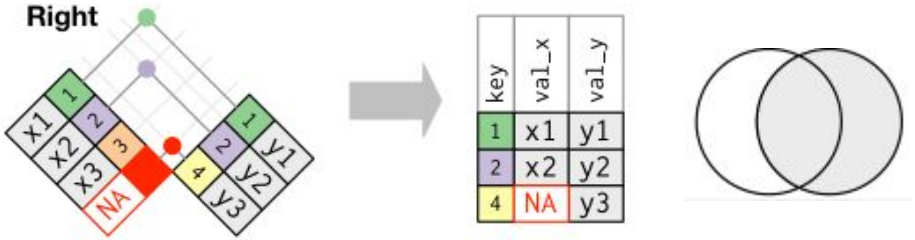

```
left_join(x, y, by = “key”,suffix = c(".x", ".y"))
```

keeps all observations in x → missing values will be represented by NA

all x rows

use this whenever you look up additional data from another table, because it preserves the original observations even when there isn’t a match

Mutating joins: right_join()

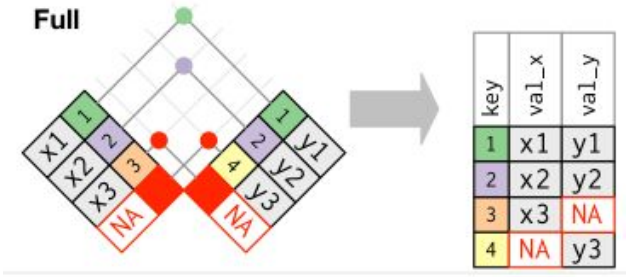


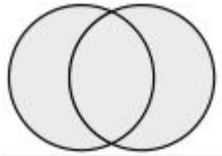

```
right_join(x, y, by = “key”,suffix = c(".x", ".y"))
```

keeps all observations in y

a subset of x rows, followed by unmatched y rows

Mutating joins: full_join()





```
full_join(x, y, by = “key”,suffix = c(".x", ".y"))
```

keeps all observations in x and y

all x rows, followed by unmatched y rows

Mutating joins: inner_join() & left_join

```
# Ex.15: combine the flights & weather dataset by using the key “time_hour” &
“origin”

flights %>%
  inner_join(weather, by = c("time_hour",”origin”))

flights %>%
  inner_join(weather, by = c("year","month","day","hour","time_hour",”origin”))

# Ex.16: combine the flights & weather dataset by using the key “time_hour” &
“origin” + rename duplicate variables in “_flights” & “_weather”
flights %>%
    inner_join(weather, by = c("time_hour",”origin), suffix = c("_flights",
"_weather"))


# Ex.17: use left_join to add the airport data to the flights data (be aware that
the 3 letter destination code is named differently in both tables)
flights %>%
  left_join(airports, by = c("dest" = "faa"))
```

Joins: Filtering joins

keep or remove observations from the first table

do not add new variables

filter rows from x based on the presence or absence of matches in y

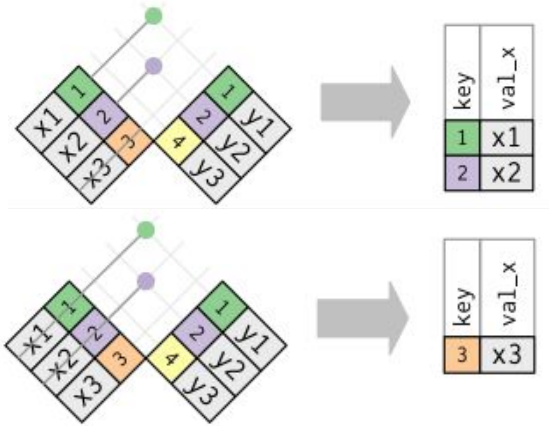
filtering joins never duplicate rows like mutating joins do

- `semi_join(x, y, by = NULL)`

  what observations in x are **also** in y?

- `anti_join(x, y, by = NULL)`

  what observation in x are **not** in y?



Filtering joins: semi_join() & anti_join

```
# Ex.18: use semi_join to filter for flights whose destination
is one of the airports listed in the airports dataset

flights %>%
    semi_join(airports, by = c("dest" = "faa"))

# Ex.19: use anti_join to filter for flights whose destination
is not one of the airports listed in the airports dataset

flights %>%
  anti_join(airports, by = c("dest" = "faa"))
```

Mutating joins & filtering joins

```
# Task 9
# combine the flights & planes dataset by using inner_join & the
key “tailnum” & rename duplicate variables in “_flights” &
“_planes” -> which variables are duplicates?


# Task 10
# repeat Task 1 but this time with the verb left_join. Compare the
resulting number of rows: Which one has more rows? Why?

# Task 11
# Create a dataset with only the flights that do not match a plane
from the planes dataset, based on the tailnum. (use a join verb!)
What’s the resulting number of rows?

# Task 12
# What do you get by this command?
airports %>%
    anti_join(flights, by = c("faa" = "dest"))
```

All joins: which definition belongs to which join?

```
# Definition 1:
Keep all informations from both tables.
 → full-, semi-, or inner-join?

# Definition 2:
You want to keep only observations that match perfectly between tables.
 → full-, anti-, or inner-join?

# Definition 3:
Filter the first table for observations that match the second.
 → right-, semi-, or anti-join?

# Definition 4:
You want to keep all observations in the first table, including matching observations in the
second table.
 → left-, inner-, or right-join?

# Definition 5:
Filter the first table for observations that don't match the second.
 → inner-, right-, or anti-join?

# Definition 6:
You want to keep all observations in the second table, including matching observations in
the first table.
 → left-, full-, or right-join?
```

lubridate

Working with dates and times

Why use lubridate?

Example case 1:

- Medical study: determine time-in-study.
- e.g. Oct 8, 2019 - May 22, 2020

Example case 2:

- Appointment in different time zones

Why use lubridate?

lubridate helps with dealing with problematic time features such as:

- gap years and seconds
- daylight saving time
- time zones

And makes it easier to parse and manipulate dates.

lubridate - overview

1. Parsing date times
2. Getting and setting components
3. Dealing with time intervals
4. Dealing with time zones
5. Arithmetic with date times

Parsing date-times

- dates can come in different formats
  - 20220425
  - 04-25-2022
  - 25/04/2022
- goal: convert strings or numbers to date-times
- year (y), month (m), day (d), hour (h), minute (m), second (s), quarter (q)

- date-time is point in timeline, stored in numbers of days/seconds since 1970-01-01 00:00:00 UTC

Parsing date-times - syntax

```
ymd(x)
myd(x)
dmy(x)
ymd_hms(x, tz="...")
...

today()
now()

parse_date_time(x, orders)
make_datetime(year, month, day, hour, ...)
```

Parsing date-times - example

```
> arrive <- ymd_hms("2020-02-06 12:00:00", tz =
"Australia/Melbourne")

> arrive

[1] "2020-02-06 12:00:00 AEDT"

> leave <- ymd_hms("2020-03-06 14:00:00", tz =
"Australia/Melbourne")

> leave

[1] "2020-03-06 14:00:00 AEDT"
```

Parsing date-times - example

```
> as.numeric(ymd("19700103"))
[1] 2
> as.numeric(ymd_hms("19700101 00:01:30"))
[1] 90
```

Getting and setting components

- given date/date-time
- extract or set given information
  - second, minute, hour
  - wday, yday
  - week
  - day, month, year
  - tz

Getting and setting components - example

```
> arrive
[1] "2020-02-06 12:00:00 AEDT"
> hour(arrive) <- 16
> arrive
[1] "2020-02-06 16:00:00 AEDT"
```

Exercise: parsing

1. Store this year's New Year's Day's noon in a variable as a date-time.

   Find two different ways of doing this.

   (Hint: from strings and from numbers)

2. Change this date-time to half past twelve.

Exercise: parsing - solution

```
> a <- ymd_hms("2022-01-01 12:00:00")
> a
[1] "2022-01-01 12:00:00 UTC"
> b <- make_datetime(2022, 01, 01, 12)
> b
[1] "2022-01-01 12:00:00 UTC"
```

Exercise: parsing - Solution

```
> minute(a) <- 30

> a

[1] "2022-01-01 12:30:00 UTC"
```

Dealing with time zones

Use case 1:

- same moment in different time zones
- with_tz()

Use case 2:

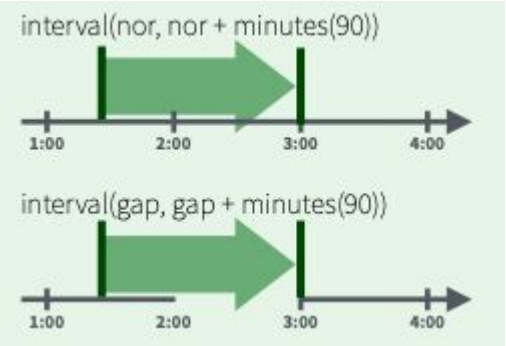
- combine a clock with a time zone
- create a new moment
- force_tz()

Dealing with time zones - example

```
> grep("Vienna", OlsonNames(), value=TRUE)
[1] "Europe/Vienna"
> call <- ymd_hms("20200227 18:00:00", tz="Europe/Vienna")
> with_tz(call, tzone = "Australia/Melbourne")
[1] "2020-02-28 04:00:00 AEDT"
> wrong <- force_tz(call, tzone="Australia/Melbourne")
> with_tz(wrong, tzone = "Europe/Vienna")
[1] "2020-02-27 08:00:00 CET"
```

Dealing with time intervals

- interval class object
- defined by start and end point



Dealing with time intervals - syntax

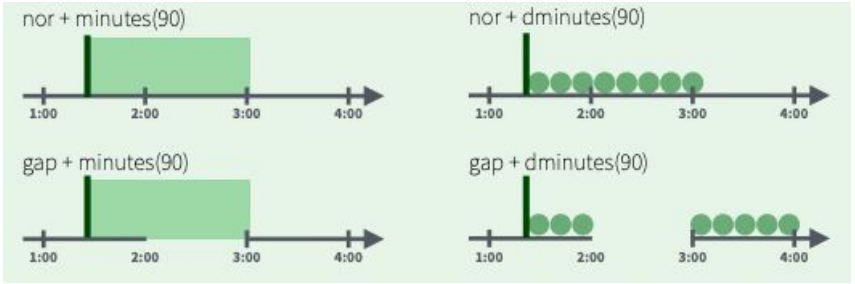
```
int_start()

int_end()

int_flip()

int_shift()

int_overlaps()

%within%
```

Dealing with time intervals - example

```
> melbourne <- interval(arrive, leave)

> melbourne

[1] 2020-02-06 16:00:00 AEDT--2020-03-06 14:00:00 AEDT

> dad <- interval(ymd("20200303", tz="Australia/Melbourne"),
ymd("20200311", tz="Australia/Melbourne"))

> dad

[1] 2020-03-03 AEDT--2020-03-11 AEDT

> int_overlaps(melbourne, dad)

[1] TRUE
```

Arithmetic with date-times

- duration vs. period
  - duration: mathematically precise results
    - 1 year = 365 days
    - time span in seconds
  - period: the “intuitive” results
    - “calendar” calculations



- helper functions for durations: begin with “d” or “e”
  - years(), months(), …
  - dseconds(), dminutes(), …

Arithmetic with date-times - example

```
> as.period(dad)
[1] "8d 0H 0M 0S"
> as.duration(dad)
[1] "691200s (~1.14 weeks)"
```

Arithmetic with date-times - example

```
> melbourne / ddays(1)
[1] 28.91667
> melbourne %/% days(1)
[1] 28
> arrive + months(1)
[1] "2020-03-06 16:00:00 AEDT"
> arrive + dmonths(1)
[1] "2020-03-08 02:30:00 AEDT"
```

Arithmetic with date-times - example

```
> arrive2 <- update(arrive, year=2021)
> arrive2 + months(1)
[1] "2021-03-06 16:00:00 AEDT"
> arrive2 + dmonths(1)
[1] "2021-03-09 02:30:00 AEDT"
```

Exercise: arithmetic

How old is a person born on January 1, 2000 today?

What time is 24 hours after 1 pm on March 26, 2022?

Print the start times of the next 3 meetings of this course.

Exercise: arithmetic - solution

```
> age <- today() - ymd('2000-01-01')
> age
Time difference of 8149 days
> as.duration(age)
[1] "704073600s (~22.31 years)"
```

Exercise: arithmetic - solution

```
> one_pm <- ymd_hms("2016-03-26 13:00:00", tz =
"Europe/Vienna")
> one_pm
[1] "2016-03-26 13:00:00 CET"
> one_pm + ddays(1)
[1] "2016-03-27 14:00:00 CEST"
```

Exercise: arithmetic - solution

```
> meetings <- ymd_hms('2022-04-25 16:00:00', tz =
"Europe/Vienna") + weeks(0:2)

> meetings

[1] "2022-04-25 16:00:00 CEST" "2022-05-02 16:00:00 CEST"

"2022-05-09 16:00:00 CEST"
```

Exercise: combination of dplyr + lubridate (1)

Create a new column that stores the departure time of each flight as a date-time.

Exercise: combination of dplyr + lubridate (1) - solution

```
> flights %>%
+   select(year, month, day, hour, minute)
# A tibble: 336,776 × 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
...
```

Exercise: combination of dplyr + lubridate (1) - solution

```
> flights %>%
+   select(year, month, day, hour, minute) %>%
+   mutate(departure = make_datetime(year, month, day, hour,
minute))
# A tibble: 336,776 × 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
...
```

Exercise: combination of dplyr + lubridate (2)

Create new where you store:

- departure time
- arrival time
- scheduled departure time
- scheduled arrival time

as date-times.

(Hint: Filter out na-entries)

(Hint: You need to extract hours and minutes from the time columns.)

Exercise: combination of dplyr + lubridate (2) - solution

```
> make_datetime_100 <- function(year, month, day, time) {
+   make_datetime(year, month, day, time %/% 100, time %%
100)
+ }
```

Exercise: combination of dplyr + lubridate (2) - solution

```
> flights_dt <- flights %>%
+   filter(!is.na(dep_time), !is.na(arr_time)) %>%
+   mutate(
+     dep_time = make_datetime_100(year, month, day, dep_time),
+     arr_time = make_datetime_100(year, month, day, arr_time),
+     sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
+     sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
+   ) %>%
+   select(origin, dest, ends_with("delay"), ends_with("time"))
> flights_dt
```

Exercise: combination of dplyr + lubridate (3)

Create new columns that store the scheduled and actual air time as a time period.

Exercise: combination of dplyr + lubridate (3) - solution

```
> flights_d <- flights_dt %>%
+   mutate(sched_air_time = as.period(sched_arr_time -
sched_dep_time),
+          air_time = as.period(arr_time - dep_time)) %>%
+   select(origin, dest, ends_with("delay"),
ends_with("time"))
> flights_d
```

Sources

https://www.rstudio.com/resources/cheatsheets/

https://dplyr.tidyverse.org/reference/

https://www.gastonsanchez.com/intro2cwd/eda-dplyr.html

http://zevross.com/blog/2014/03/26/four-reasons-why-you-should-check-out-the-r-package-dplyr-3/

Sources

https://r4ds.had.co.nz/dates-and-times.html

https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html

https://data.library.virginia.edu/working-with-dates-and-time-in-r-using-the-lubridate-package/

https://rawgit.com/rstudio/cheatsheets/main/lubridate.pdf

https://lubridate.tidyverse.org/reference/lubridate-package.html