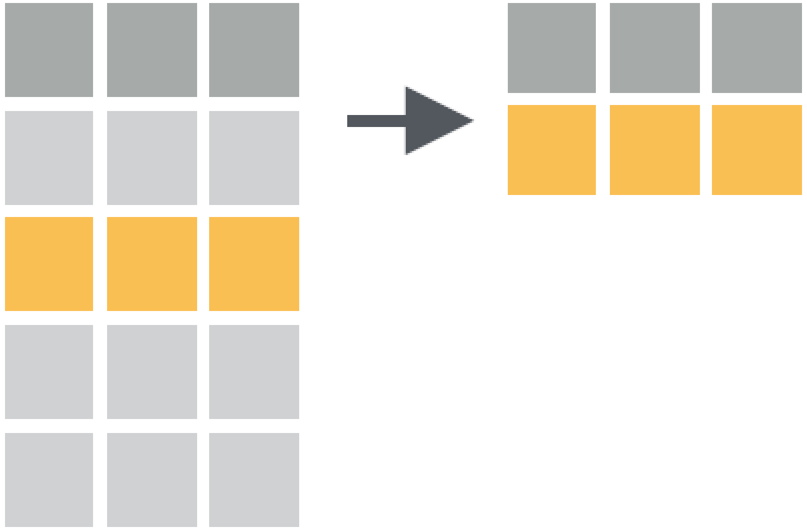
Wrangle Data

Sometimes the data we are working with is not quite ready for whatever it is we want to do.

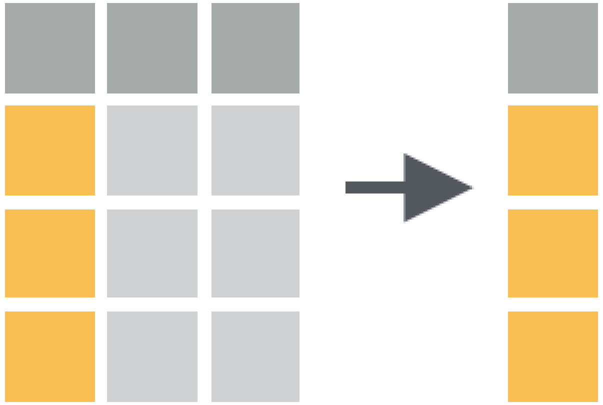
`dplyr`: your new best friend.

- Three key functions:
  - `filter()` to subset data based on rows



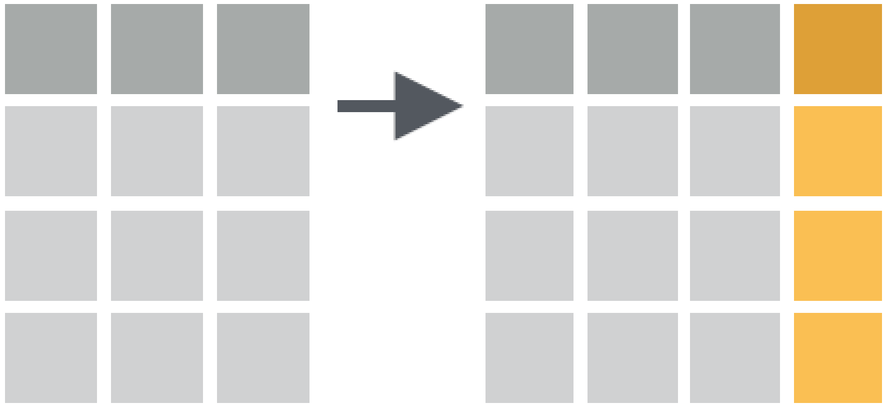
`dplyr`

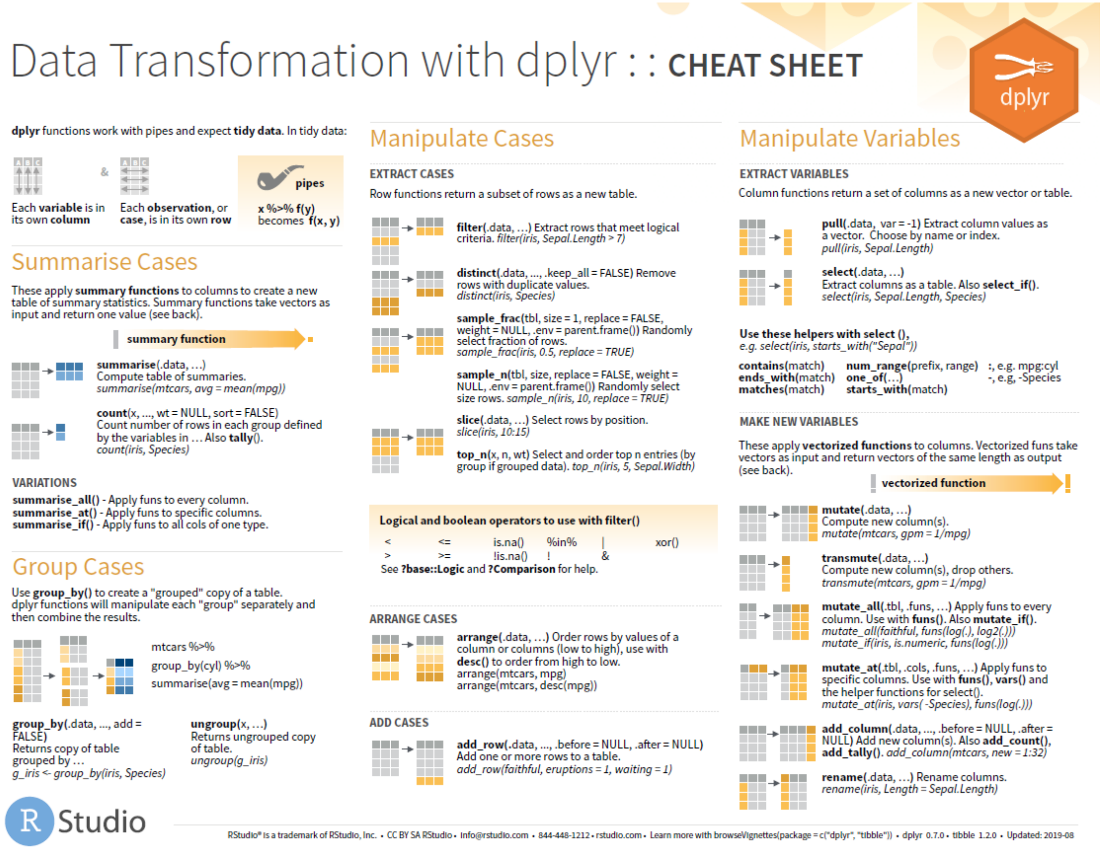
- Three key functions:
  - `filter()` to subset data based on rows
  - `select()` to subset data based on columns



`dplyr`

- Three key functions:
  - `filter()` to subset data based on rows
  - `select()` to subset data based on columns
  - `mutate()` to add or modify values in columns





Data Transformation with dplyr : : CHEAT SHEET

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes

x %>% f(y) becomes **f(x, y)**

Summarise Cases

These apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function

summarise(.data, …)
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
summarise_if() - Apply funs to all cols of one type.

Group Cases

Use **group_by()** to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))

group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

ungroup(x, …**)**
Returns ungrouped copy of table.
ungroup(g_iris)

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, …) Extract rows that meet logical criteria. *filter(iris, Sepal.Length > 7)*

distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
distinct(iris, Species)

sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*

slice(.data, …) Select rows by position.
slice(iris, 10:15)

top_n(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::Logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, …) Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES

add_row(.data, ..., .before = NULL, .after = NULL)
Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1) Extract column values as a vector. Choose by name or index.
pull(iris, Sepal.Length)

select(.data, …)
Extract columns as a table. Also **select_if**().
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match)	**num_range**(prefix, range)	:, e.g. mpg:cyl
ends_with(match)	**one_of**(…)	-, e.g, -Species
matches(match)	**starts_with**(match)	

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).

vectorized function

mutate(.data, …)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, …)
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

mutate_all(.tbl, .funs, …) Apply funs to every column. Use with **funs()**. Also **mutate_if()**.
mutate_all(faithful, funs(log(.), log2(.)))
mutate_if(iris, is.numeric, funs(log(.)))

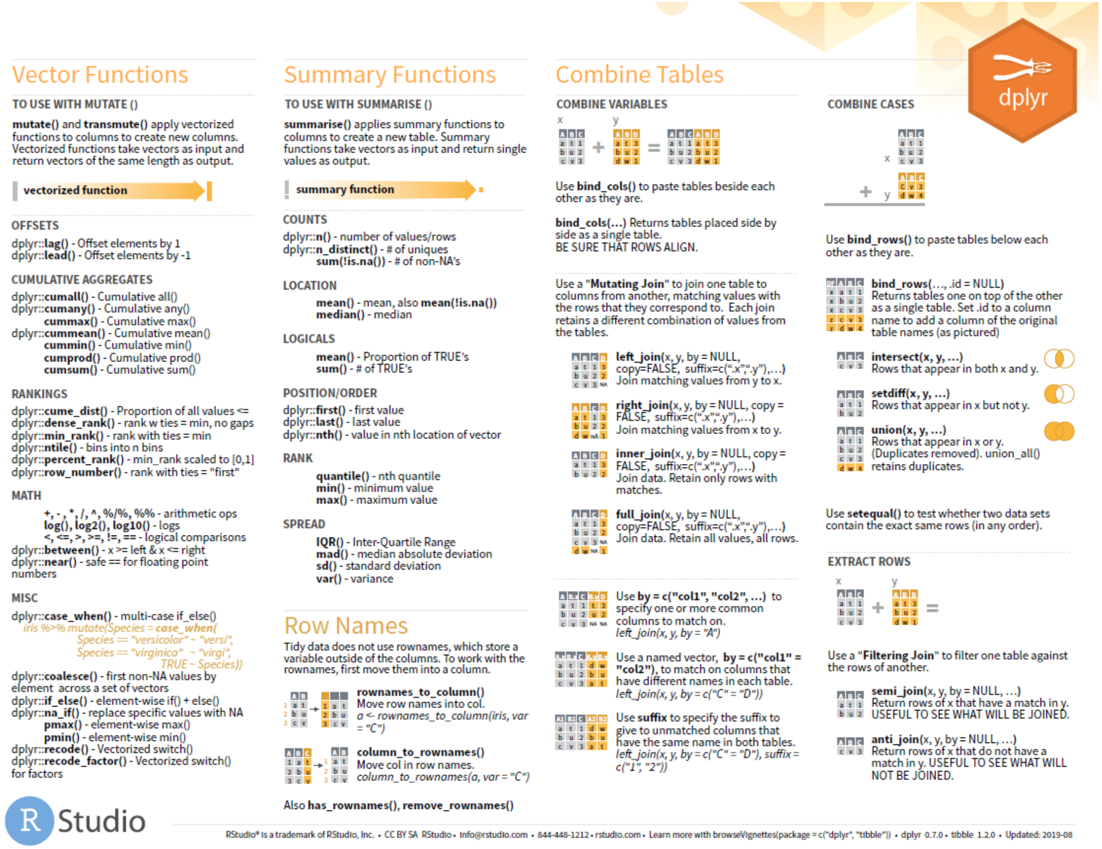
mutate_at(.tbl, .cols, .funs, …) Apply funs to specific columns. Use with **funs(), vars()** and the helper functions for select().
mutate_at(iris, vars(-Species), funs(log(.)))

add_column(.data, ..., .before = NULL, .after = NULL) Add new column(s). Also **add_count(), add_tally()**. *add_column(mtcars, new = 1:32)*

rename(.data, …) Rename columns.
rename(iris, Length = Sepal.Length)

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more with browseVignettes(package = c("dplyr", "tibble")) • dplyr 0.7.0 • tibble 1.2.0 • Updated: 2019-08



Vector Functions

TO USE WITH MUTATE ()

mutate() and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

vectorized function

OFFSETS

dplyr::**lag()** - Offset elements by 1
dplyr::**lead()** - Offset elements by -1

CUMULATIVE AGGREGATES

dplyr::**cumall()** - Cumulative all()
dplyr::**cumany()** - Cumulative any()
cummax() - Cumulative max()
dplyr::**cummean()** - Cumulative mean()
cummin() - Cumulative min()
cumprod() - Cumulative prod()
cumsum() - Cumulative sum()

RANKINGS

dplyr::**cume_dist()** - Proportion of all values <=
dplyr::**dense_rank()** - rank w ties = min, no gaps
dplyr::**min_rank()** - rank with ties = min
dplyr::**ntile()** - bins into n bins
dplyr::**percent_rank()** - min_rank scaled to [0,1]
dplyr::**row_number()** - rank with ties = "first"

MATH

+, - , *, /, ^, %/%, %% - arithmetic ops
log(), log2(), log10() - logs
<, <=, >, >=, !=, == - logical comparisons
dplyr::**between()** - x >= left & x <= right
dplyr::**near()** - safe == for floating point numbers

MISC

dplyr::**case_when()** - multi-case if_else()

```
iris %>% mutate(Species = case_when(
          Species == "versicolor" ~ "versi",
          Species == "virginica"  ~ "virgi",
                        TRUE ~ Species))
```

dplyr::**coalesce()** - first non-NA values by element across a set of vectors
dplyr::**if_else()** - element-wise if() + else()
dplyr::**na_if()** - replace specific values with NA
pmax() - element-wise max()
pmin() - element-wise min()
dplyr::**recode()** - Vectorized switch()
dplyr::**recode_factor()** - Vectorized switch() for factors

Summary Functions

TO USE WITH SUMMARISE ()

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function

COUNTS

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

LOCATION

mean() - mean, also **mean(!is.na())**
median() - median

LOGICALS

mean() - Proportion of TRUE's
sum() - # of TRUE's

POSITION/ORDER

dplyr::**first()** - first value
dplyr::**last()** - last value
dplyr::**nth()** - value in nth location of vector

RANK

quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

IQR() - Inter-Quartile Range
mad() - median absolute deviation
sd() - standard deviation
var() - variance

Row Names

Tidy data does not use rownames, which store a variable outside of the columns. To work with the rownames, first move them into a column.

rownames_to_column()
Move row names into col.
a <- rownames_to_column(iris, var = "C")

column_to_rownames()
Move col in row names.
column_to_rownames(a, var = "C")

Also **has_rownames(), remove_rownames()**

Combine Tables

COMBINE VARIABLES

Use **bind_cols()** to paste tables beside each other as they are.

bind_cols(…) Returns tables placed side by side as a single table.
BE SURE THAT ROWS ALIGN.

Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…)
Join matching values from y to x.

right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join matching values from x to y.

inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join data. Retain only rows with matches.

full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…)
Join data. Retain all values, all rows.

Use **by = c("col1", "col2", …)** to specify one or more common columns to match on.
left_join(x, y, by = "A")

Use a named vector, **by = c("col1" = "col2")**, to match on columns that have different names in each table.
left_join(x, y, by = c("C" = "D"))

Use **suffix** to specify the suffix to give to unmatched columns that have the same name in both tables.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

COMBINE CASES

Use **bind_rows()** to paste tables below each other as they are.

bind_rows(…, .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

intersect(x, y, …)
Rows that appear in both x and y.

setdiff(x, y, …)
Rows that appear in x but not y.

union(x, y, …)
Rows that appear in x or y. (Duplicates removed). union_all() retains duplicates.

Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

EXTRACT ROWS

Use a "**Filtering Join**" to filter one table against the rows of another.

semi_join(x, y, by = NULL, …)
Return rows of x that have a match in y. USEFUL TO SEE WHAT WILL BE JOINED.

anti_join(x, y, by = NULL, …)
Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more with browseVignettes(package = c("dplyr", "tibble")) • dplyr 0.7.0 • tibble 1.2.0 • Updated: 2019-08

About the data

- ebird
- wq

- read in using:
  - notes_01 script: bit.ly/gb2020-notes01
  - or 02_wrangle.R script (in zipped folder)

`filter()`

- The eBird data set has a lot of rows. Let’s examine entries from a particular state.

```
filter(ebird, state == ‘AK’)

ebird %>%
      filter(state == ‘AK’)
```

We tend to use pipes!
x %>% f(y) = f(x,y)
“and then…”

`filter()`

- We use `==` to specify an exact condition
  - < less than
  - <= less than or equal to
  - == equals
  - != *not* equal to
  - >= greater than or equal to
  - > greater than
- The condition must return either true or false, for each row.

See ?base::Logic and ?Comparison for help

`filter()` with multiple conditions

You can filter on more than one criterion

- Example: we want all the birds from Alaska, but only in the year 2008

```
ebird %>%
        filter(state == ‘AK’,
               year == 2008)
```

It does not matter if you put `state` or `year` first; you can do either.

`filter()` with multiple conditions

You can filter on more than one criterion

- What if we want to look at birds from more than one state?

```
ebird %>%
      filter(year == 2008,
       state %in% c(‘AK’, ‘AL’, ‘MS’)
       )
```

can use
single quotes (‘ ’)
or double (“ “)

YOUR TURN 1

1. How could we pull out birds in Alaska (AK), before 2010? (Hint: *you can use the same symbols on year that you would use with any other numbers*)
2. Filter the data to contain only the species "American Coot" from MS and FL (your instructors' states), in all years *except* 2010. Assign this object to a data frame and verify (using `unique()`) that you did it right.

`select()`

- Simplify your data by keeping only the columns you want to work with. You can do this two ways:

```
ebird %>%
        select(species, state, year)

ebird %>%
        select(-samplesize, -presence)
```

`select()` order DOES matter

You can rearrange your columns of data this way

```
ebird %>%
        select(species, state, year)
```

species	state	year
American Coot	Florida	2008

`select()` order DOES matter

You can rearrange your columns of data this way

```
ebird %>%
        select(year, species, state)
```

year	species	state
2008	American Coot	Florida

`select()` order DOES matter

You can also just move one or a few column and keep the original order of all the other columns

```
ebird %>%
        select(year, everything())
```

`select()` has helper functions

Use these functions with select, e.g.

```
select(ebird, starts_with(“e”))
```

contains()

ends_with()

matches()

num_range()

one_of(…)

starts_with()

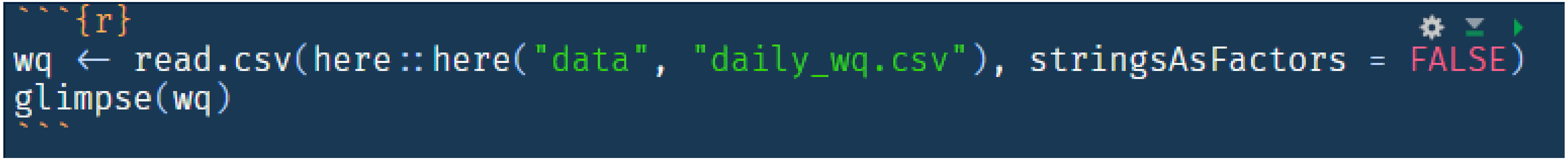
YOUR TURN

From the ebird data, subset to only include the species American Coot, from the states FL, AL, and MS. Keep only the state, year, and presence columns.

What is the proper order of operations in this case?

About the data

Daily averages of water quality data (e.g., temperature, salinity, dissolved oxygen, etc.) from 6 National Estuarine Research Reserves



~~~
```{r}
wq <- read.csv(here::here("data", "daily_wq.csv"), stringsAsFactors = FALSE)
glimpse(wq)
```
~~~

YOUR TURN

We have read in the daily water quality data for a few stations. Create a new data frame called `wq_trimmed` where, from `wq`, you:

1. **Select** the following columns: station_code, month, day, temp, sal, do_pct, and depth.
2. **Filter** for rows where `depth` is *_not_* missing. (Hint: `is.na` is the function that checks to see if a value *_is_* missing. How would you look for "_not_" `is.na`?   It's similar to "not equal to" …)

YOUR TURN: how did you do?

```
wq_trimmed <- wq %>%
        select(station_code, month,
                        day, temp, sal,
                        do_pct, depth) %>%
   filter(!is.na(depth))
```

Modifying with `mutate()`

dplyr::mutate
add column(s), keep existing.
A
B
C
1
5
10
0
2
-3
4
9
6
3
2
14
9
8
THE PLAN:
A B C
+ =
Horst '18

Modifying with `mutate()`

- mutate() makes creating new variables out of other variables in your dataset easy
- Depth is recorded in meters; perhaps feet would be better for the public in the US:

```
wq_trimmed <- wq_trimmed %>%
        mutate(depth_ft = depth * 3.28)
```

Why 3.28?!

Modifying with `mutate()`

- You can also use other columns, AND use a column after creating it

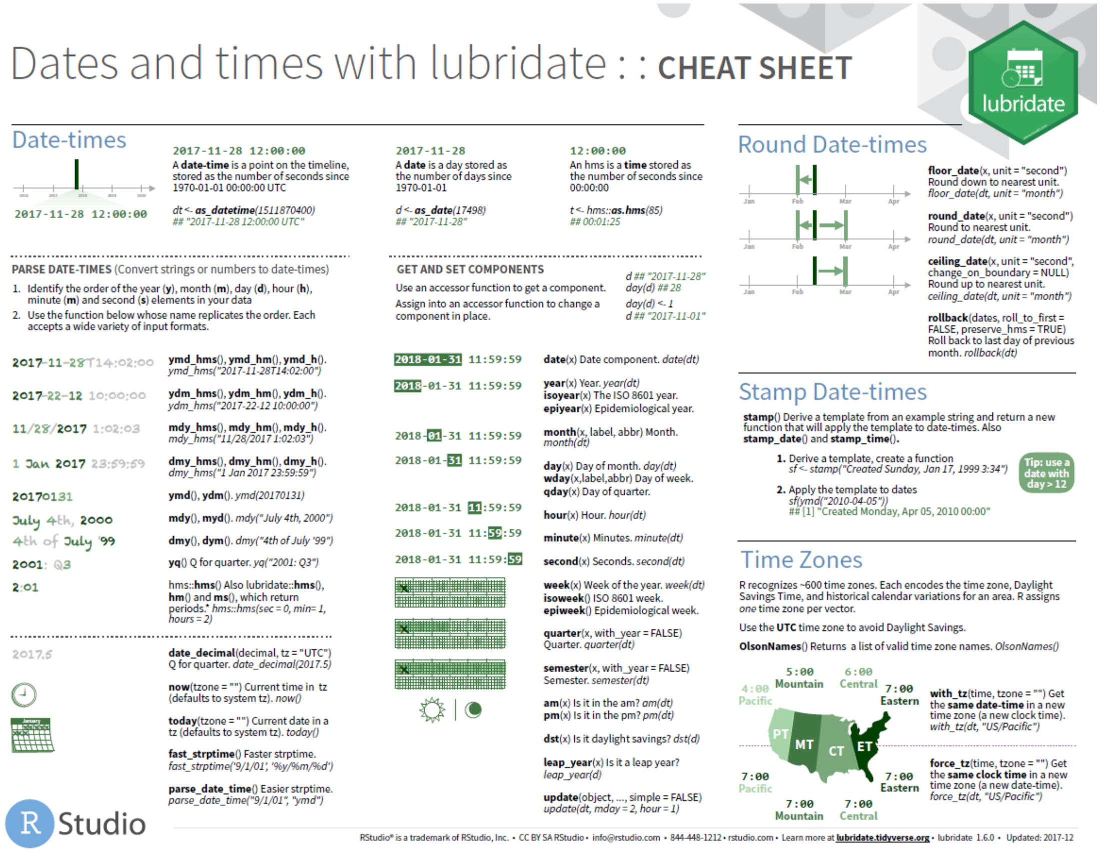
```
wq_trimmed <- wq_trimmed %>%
        mutate(monthday = paste(month, day, sep = “-”),
                   meaningless = sal + temp,
                   even_more_meaningless = meaningless + 5)
```

YOUR TURN

There are two parts to this. You can approach them separately or within the same series of pipes. Remember to save the result as the new, better, `wq_trimmed` data frame!

1. Remove `monthday` and the `meaningless_thing`s from the `wq_trimmed` data frame.
2. The same person that wants to see `depth` in feet rather than meters *also* wants you to turn `temp` into Fahrenheit, from Celsius. You've looked up the conversion. Now create a new column, `temp_f`, with the new variable.
   F = (9/5)(temp in C) + 32

Dates and Times



Dates and times with lubridate : : CHEAT SHEET

Date-times

2017-11-28 12:00:00

2017-11-28 12:00:00
A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

*dt <- **as_datetime**(1511870400)*
## "2017-11-28 12:00:00 UTC"

2017-11-28
A **date** is a day stored as the number of days since 1970-01-01

*d <- **as_date**(17498)*
## "2017-11-28"

12:00:00
An hms is a **time** stored as the number of seconds since 00:00:00

*t <- hms::**as.hms**(85)*
## 00:01:25

PARSE DATE-TIMES (Convert strings or numbers to date-times)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data
2. Use the function below whose name replicates the order. Each accepts a wide variety of input formats.

2017-11-28T14:02:00	**ymd_hms**(), **ymd_hm**(), **ymd_h**(). *ymd_hms("2017-11-28T14:02:00")*
2017-22-12 10:00:00	**ydm_hms**(), **ydm_hm**(), **ydm_h**(). *ydm_hms("2017-22-12 10:00:00")*
11/28/2017 1:02:03	**mdy_hms**(), **mdy_hm**(), **mdy_h**(). *mdy_hms("11/28/2017 1:02:03")*
1 Jan 2017 23:59:59	**dmy_hms**(), **dmy_hm**(), **dmy_h**(). *dmy_hms("1 Jan 2017 23:59:59")*
20170131	**ymd**(), **ydm**(). *ymd(20170131)*
July 4th, 2000	**mdy**(), **myd**(). *mdy("July 4th, 2000")*
4th of July '99	**dmy**(), **dym**(). *dmy("4th of July '99")*
2001: Q3	**yq**() Q for quarter. *yq("2001: Q3")*
2:01	hms::**hms**() Also lubridate::**hms**(), **hm**() and **ms**(), which return periods.* *hms::hms(sec = 0, min= 1, hours = 2)*
2017.5	**date_decimal**(decimal, tz = "UTC") Q for quarter. *date_decimal(2017.5)*
	now(tzone = "") Current time in tz (defaults to system tz). *now()*
	today(tzone = "") Current date in a tz (defaults to system tz). *today()*
	fast_strptime() Faster strptime. *fast_strptime('9/1/01', '%y/%m/%d')*
	parse_date_time() Easier strptime. *parse_date_time("9/1/01", "ymd")*

GET AND SET COMPONENTS

Use an accessor function to get a component.

Assign into an accessor function to change a component in place.

d ## "2017-11-28"
day(d) ## 28
day(d) <- 1
d ## "2017-11-01"

2018-01-31 11:59:59	**date**(x) Date component. *date(dt)*
2018-01-31 11:59:59	**year**(x) Year. *year(dt)* **isoyear**(x) The ISO 8601 year. **epiyear**(x) Epidemiological year.
2018-01-31 11:59:59	**month**(x, label, abbr) Month. *month(dt)*
2018-01-31 11:59:59	**day**(x) Day of month. *day(dt)* **wday**(x,label,abbr) Day of week. **qday**(x) Day of quarter.
2018-01-31 11:59:59	**hour**(x) Hour. *hour(dt)*
2018-01-31 11:59:59	**minute**(x) Minutes. *minute(dt)*
2018-01-31 11:59:59	**second**(x) Seconds. *second(dt)*
	week(x) Week of the year. *week(dt)* **isoweek**() ISO 8601 week. **epiweek**() Epidemiological week.
	quarter(x, with_year = FALSE) Quarter. *quarter(dt)*
	semester(x, with_year = FALSE) Semester. *semester(dt)*
	am(x) Is it in the am? *am(dt)* **pm**(x) Is it in the pm? *pm(dt)*
	dst(x) Is it daylight savings? *dst(d)*
	leap_year(x) Is it a leap year? *leap_year(d)*
	update(object, ..., simple = FALSE) *update(dt, mday = 2, hour = 1)*

Round Date-times

floor_date(x, unit = "second") Round down to nearest unit. *floor_date(dt, unit = "month")*

round_date(x, unit = "second") Round to nearest unit. *round_date(dt, unit = "month")*

ceiling_date(x, unit = "second", change_on_boundary = NULL) Round up to nearest unit. *ceiling_date(dt, unit = "month")*

rollback(dates, roll_to_first = FALSE, preserve_hms = TRUE) Roll back to last day of previous month. *rollback(dt)*

Stamp Date-times

stamp() Derive a template from an example string and return a new function that will apply the template to date-times. Also **stamp_date**() and **stamp_time**().

1. Derive a template, create a function
   sf <- stamp("Created Sunday, Jan 17, 1999 3:34")
2. Apply the template to dates
   sf(ymd("2010-04-05"))
   ## [1] "Created Monday, Apr 05, 2010 00:00"

Tip: use a date with day > 12

Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns *one* time zone per vector.

Use the **UTC** time zone to avoid Daylight Savings.

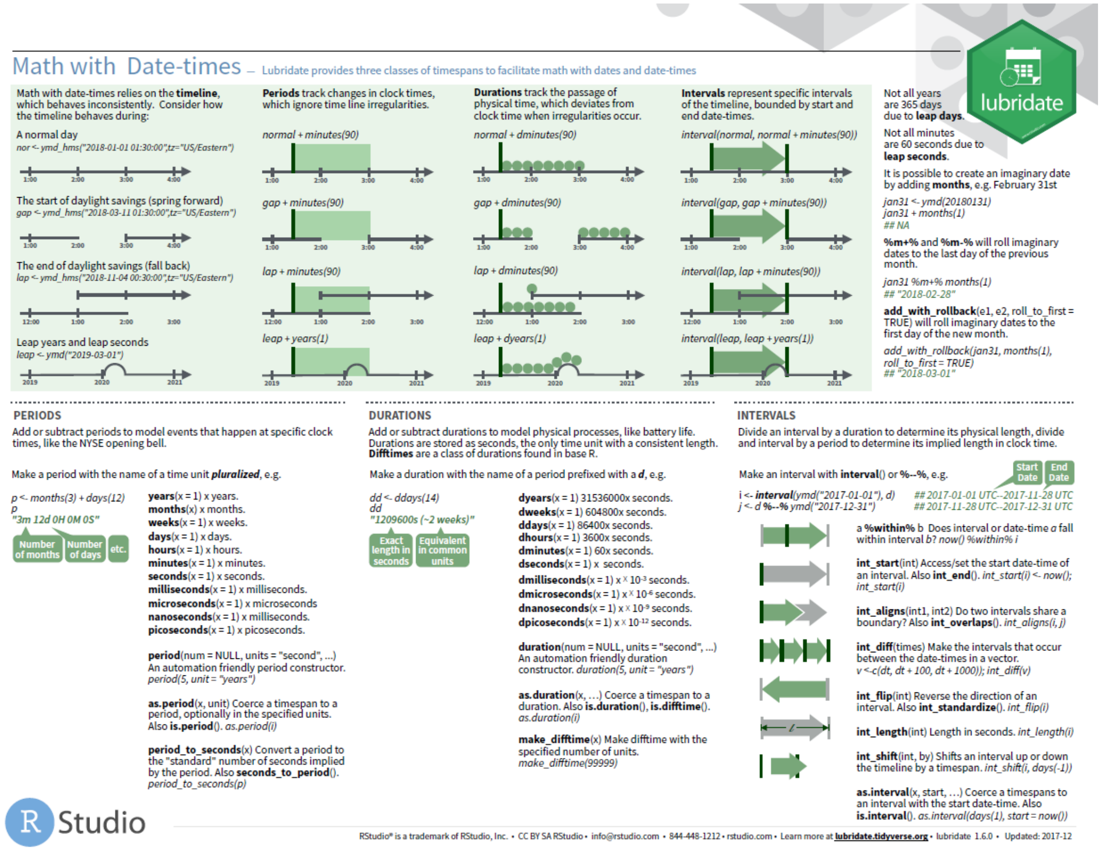
OlsonNames() Returns a list of valid time zone names. *OlsonNames()*

with_tz(time, tzone = "") Get the **same date-time** in a new time zone (a new clock time). *with_tz(dt, "US/Pacific")*

force_tz(time, tzone = "") Get the **same clock time** in a new time zone (a new date-time). *force_tz(dt, "US/Pacific")*

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **lubridate.tidyverse.org** • lubridate 1.6.0 • Updated: 2017-12



Math with Date-times – Lubridate provides three classes of timespans to facilitate math with dates and date-times

Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

A normal day
nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")

The start of daylight savings (spring forward)
gap <- ymd_hms("2018-03-11 01:30:00",tz="US/Eastern")

The end of daylight savings (fall back)
lap <- ymd_hms("2018-11-04 00:30:00",tz="US/Eastern")

Leap years and leap seconds
leap <- ymd("2019-03-01")

Periods track changes in clock times, which ignore time line irregularities.

normal + minutes(90)
gap + minutes(90)
lap + minutes(90)
leap + years(1)

Durations track the passage of physical time, which deviates from clock time when irregularities occur.

normal + dminutes(90)
gap + dminutes(90)
lap + dminutes(90)
leap + dyears(1)

Intervals represent specific intervals of the timeline, bounded by start and end date-times.

interval(normal, normal + minutes(90))
interval(gap, gap + minutes(90))
interval(lap, lap + minutes(90))
interval(leap, leap + years(1))

Not all years are 365 days due to **leap days**.

Not all minutes are 60 seconds due to **leap seconds**.

It is possible to create an imaginary date by adding **months**, e.g. February 31st

jan31 <- ymd(20180131)
jan31 + months(1)
## NA

%m+% and **%m-%** will roll imaginary dates to the last day of the previous month.

jan31 %m+% months(1)
## "2018-02-28"

add_with_rollback(e1, e2, roll_to_first = TRUE) will roll imaginary dates to the first day of the new month.

add_with_rollback(jan31, months(1), roll_to_first = TRUE)
## "2018-03-01"

PERIODS

Add or subtract periods to model events that happen at specific clock times, like the NYSE opening bell.

Make a period with the name of a time unit ***pluralized***, e.g.

p <- months(3) + days(12)
p
"3m 12d 0H 0M 0S"

years(x = 1) x years.
months(x) x months.
weeks(x = 1) x weeks.
days(x = 1) x days.
hours(x = 1) x hours.
minutes(x = 1) x minutes.
seconds(x = 1) x seconds.
milliseconds(x = 1) x milliseconds.
microseconds(x = 1) x microseconds
nanoseconds(x = 1) x milliseconds.
picoseconds(x = 1) x picoseconds.

period(num = NULL, units = "second", ...) An automation friendly period constructor. *period(5, unit = "years")*

as.period(x, unit) Coerce a timespan to a period, optionally in the specified units. Also **is.period**(). *as.period(i)*

period_to_seconds(x) Convert a period to the "standard" number of seconds implied by the period. Also **seconds_to_period**(). *period_to_seconds(p)*

DURATIONS

Add or subtract durations to model physical processes, like battery life. Durations are stored as seconds, the only time unit with a consistent length. **Difftimes** are a class of durations found in base R.

Make a duration with the name of a period prefixed with a ***d***, e.g.

dd <- ddays(14)
dd
"1209600s (~2 weeks)"

dyears(x = 1) 31536000x seconds.
dweeks(x = 1) 604800x seconds.
ddays(x = 1) 86400x seconds.
dhours(x = 1) 3600x seconds.
dminutes(x = 1) 60x seconds.
dseconds(x = 1) x seconds.
dmilliseconds(x = 1) x × 10^{-3} seconds.
dmicroseconds(x = 1) x × 10^{-6} seconds.
dnanoseconds(x = 1) x × 10^{-9} seconds.
dpicoseconds(x = 1) x × 10^{-12} seconds.

duration(num = NULL, units = "second", ...) An automation friendly duration constructor. *duration(5, unit = "years")*

as.duration(x, …) Coerce a timespan to a duration. Also **is.duration**(), **is.difftime**(). *as.duration(i)*

make_difftime(x) Make difftime with the specified number of units. *make_difftime(99999)*

INTERVALS

Divide an interval by a duration to determine its physical length, divide and interval by a period to determine its implied length in clock time.

Make an interval with **interval**() or **%--%**, e.g.

*i <- **interval**(ymd("2017-01-01"), d)* *## 2017-01-01 UTC--2017-11-28 UTC*
j <- d %--% ymd("2017-12-31") *## 2017-11-28 UTC--2017-12-31 UTC*

a **%within%** b Does interval or date-time *a* fall within interval *b*? *now() %within% i*

int_start(int) Access/set the start date-time of an interval. Also **int_end**(). *int_start(i) <- now(); int_start(i)*

int_aligns(int1, int2) Do two intervals share a boundary? Also **int_overlaps**(). *int_aligns(i, j)*

int_diff(times) Make the intervals that occur between the date-times in a vector. *v <-c(dt, dt + 100, dt + 1000)); int_diff(v)*

int_flip(int) Reverse the direction of an interval. Also **int_standardize**(). *int_flip(i)*

int_length(int) Length in seconds. *int_length(i)*

int_shift(int, by) Shifts an interval up or down the timeline by a timespan. *int_shift(i, days(-1))*

as.interval(x, start, …) Coerce a timespans to an interval with the start date-time. Also **is.interval**(). *as.interval(days(1), start = now())*

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **lubridate.tidyverse.org** • lubridate 1.6.0 • Updated: 2017-12

YOUR TURN

`mutate` can be used to add a complete vector of the same value.

Fill in the skeleton code to create a column for a full yyyy-mm-dd style date, then make a **line** graph of temperature throughout the year at the water quality stations, and color them by station_code.

Which of these stations do you think is at the NERR in Alaska?

Group-wise operations with group_by() and summarize()

- group_by() changes each function from operating on the full dataset to specified groups. *This can be done in conjunction with other* dplyr *functions!*
- summarize() reduces multiple values down to a single summary

```
ebird %>%
        group_by(state, species) %>%
        summarize(mean_presence = mean(presence, na.rm = TRUE),
                  max_presence = max(presence, na.rm = TRUE)
                  )
```

YOUR TURN

Group the `wq_trimmed` dataset to calculate monthly average temperature and salinity, and their standard deviations, at each station.